Lightlab Documentation

Release 1.1.0

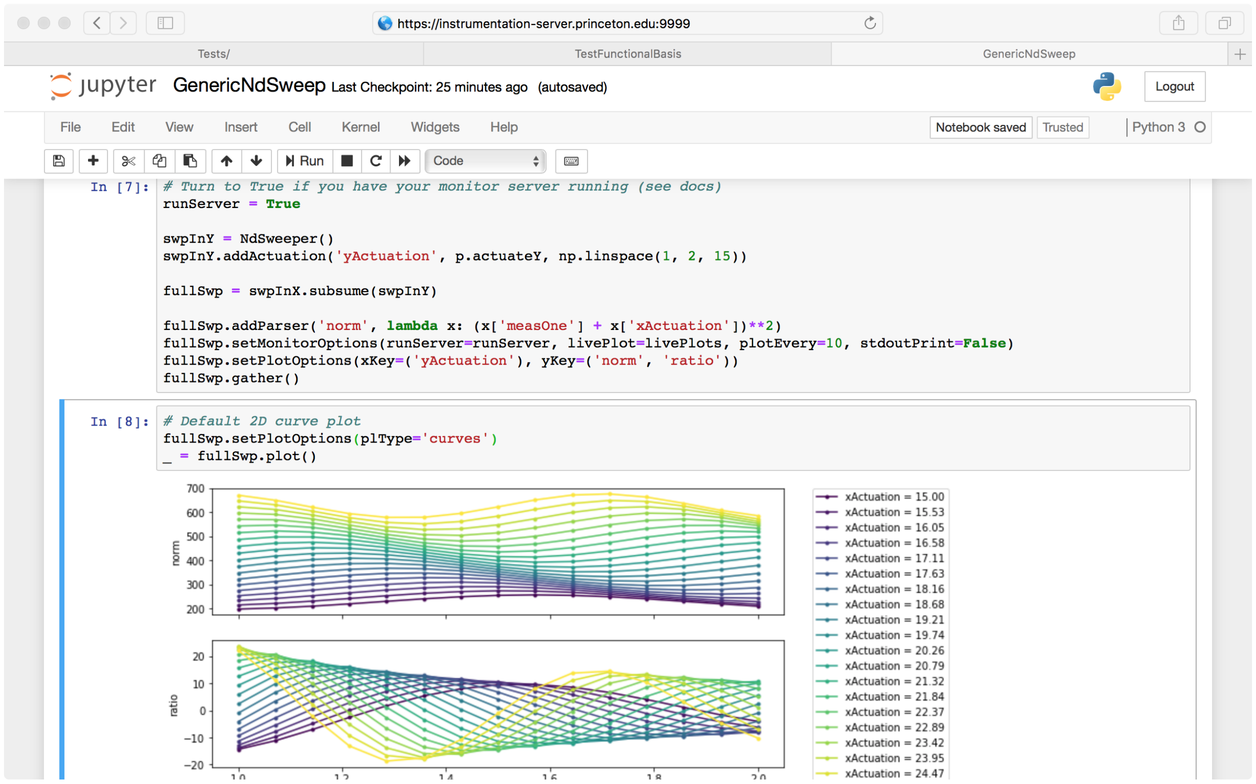
Alex Tait, Thomas Ferreira de Lima

Jul 14, 2020

Contents:

This package offers the ability to control multi-instrument experiments, and to collect and store data and methods very efficiently. It was developed by researchers in an integrated photonics lab (hence lightlab) with equipment mostly controlled by the GPIB protocol. It can be used as a combination of these three tasks:

1. Consolidated multi-instrument remote control
2. Virtual laboratory environments: repeatable, shareable
3. Utilities for experimental research: from serial comm. to testing, analysis, gathering, post-processing – to paper-ready plotting
4. All structured in python



```
In [7]: # Turn to True if you have your monitor server running (see docs)
        runServer = True

        swpInY = NdSweeper()
        swpInY.addActuation('yActuation', p.actuateY, np.linspace(1, 2, 15))

        fullSwp = swpInX.subsume(swpInY)

        fullSwp.addParser('norm', lambda x: (x['measOne'] + x['xActuation'])**2)
        fullSwp.setMonitorOptions(runServer=runServer, livePlot=livePlots, plotEvery=10, stdoutPrint=False)
        fullSwp.setPlotOptions(xKey=('yActuation'), yKey=('norm', 'ratio'))
        fullSwp.gather()

In [8]: # Default 2D curve plot
        fullSwp.setPlotOptions(plType='curves')
        _ = fullSwp.plot()
```

Fig. 1: `lightlab` in a Jupyter notebook

We wrote this documentation with love to all young experimental researchers that are not necessarily familiar with all the software tools introduced here. We attempted to include how-tos at every step to make sure everyone can get through the initial steps.

Warning: This is not a pure software package. Lightlab needs to be run in a particular configuration. Before you continue, carefully read the *Pre-requisites* and the *Getting Started to Python, Jupyter, git* sections. It contains necessary information about setup steps you need to take care before starting.

CHAPTER 1

Pre-requisites

If you intend to perform any kind of experiment automation, please read this section carefully. However, to load and visualize data, or to run a virtual experiment, the following is not needed.

1.1 Hardware

In order to enjoy lightlab's experiment control capabilities, we assume that you have VISA compatible hardware: at least one computer with a GPIB card or USB-GPIB converter; one instrument; and your favorite VISA driver installed. Just kidding, there is a one-company monopoly on that (see *pyvisa*).

There are other devices or GPIB controllers that are not VISA-compliant and do not need any driver installation, and can be used directly via a TCP socket. Prologix, for example, offers a GPIB-ethernet controller with a built-in TCP socket server. We have included a driver for that in lightlab (see *Using and creating drivers for instruments*).

1.2 pyvisa

We rely heavily on pyvisa for instrument control. It provides a wrapper layer for a VISA backend that you have to install in your computer prior to using lightlab. This is typically going to be a *National Instruments* backend, but the pyvisa team is working on a new pure-python backend (pyvisa-py). Refer to pyvisa_installation for installation instructions.

Currently we are also working with *python3*. This might present some minor inconvenience in installation, but it allows us to write code that will be supported in the long term. All dependencies are easily available in python3.

Proceed with enjoying lightlab once you have the following output:

```
>>> import visa
>>> rm = visa.ResourceManager()
```

(continues on next page)

(continued from previous page)

```
>>> print(rm.list_resources())
('GPIB0::20::INSTR', 'GPIB1::24::INSTR', 'ASRL1::INSTR', 'ASRL2::INSTR',
↪'ASRL3::INSTR', 'ASRL4::INSTR')
```

CHAPTER 2

Installation

You can install the lightlab package like any other python package:

```
pip install lightlab
```

If you are new to python programming, jupyter notebooks, you might want to sit down and patiently read the *Getting Started to Python, Jupyter, git* Pages. If you want to develop and write drivers, good for you. It's on github along with guides on contributing and can be cloned by:

```
git clone https://github.com/lightwave-lab/lightlab.git
```

Your environment will be slightly different if you're developing, described *here*.

If you need more detailed installation instructions, they are available in *Installation Instructions*.

2.1 Installation Instructions

2.1.1 Pre-requisites

If you intend to perform any kind of experiment automation, please read this section carefully. However, to load and visualize data, or to run a virtual experiment, the following is not needed.

Hardware

In order to enjoy lightlab's experiment control capabilities, we assume that you have VISA compatible hardware: at least one computer with a GPIB card or USB-GPIB converter; one instrument; and your favorite VISA driver installed. Just kidding, there is a one-company monopoly on that (see *below*).

pyvisa

We rely heavily on pyvisa for instrument control. It provides a wrapper layer for a VISA backend that you have to install in your computer prior to using lightlab. This is typically going to be a *National Instruments* backend, but the pyvisa team is working on a new pure-python backend (pyvisa-py). Refer to pyvisa_installation for installation instructions. If you need to install in ubuntu, see ubuntu_installation.

Warning: Currently we are also working with *python3*. This might present some minor inconvenience in installation, but it allows us to write code that will be supported in the long term. All dependencies are easily available in python3 and are automatically installed with pip.

Proceed with installing lightlab once you have something that looks like the following output:

```
>>> import visa
>>> rm = visa.ResourceManager()
>>> print(rm.list_resources())
('GPIB0::20::INSTR', 'GPIB1::24::INSTR', 'ASRL1::INSTR', 'ASRL2::INSTR', 'ASRL3::INSTR
↪', 'ASRL4::INSTR')
```

2.1.2 Installation in personal computer

Regular users can install lightlab with pip:

```
$ pip install lightlab
```

For more experienced users: install the lightlab package like any other python package, after having downloaded the project from github.:

```
$ python3 install setup.py
```

If you are new to python programming, jupyter notebooks, you might want to sit down and patiently read the *Getting Started to Python, Jupyter, git* Pages.

More detailed installation instructions

2.1.3 Server Installation Instructions (Advanced)

The `state` module saves information about instruments, benches, and remote hosts in a file called `~/.lightlab/labstate.json`. Normally you wouldn't have to change the location of this file. But if you so desired to, it suffices to use the shell utility `lightlab`:

```
$ lightlab config set labstate.filepath '~/.lightlab/newlocation.json'
$ lightlab config get labstate.filepath
labstate.filepath: ~/.lightlab/newlocation.json
```

It is also possible to set a system default for all users with the `--system` flag:

```
$ sudo lightlab config --system set labstate.filepath /usr/local/etc/lightlab/
↪labstate.json
Password:
----saving /usr/local/etc/lightlab.conf----
[labstate]
filepath = /usr/local/etc/lightlab/labstate.json
```

But all users must have write access to that file in order to make their own alterations. A backup is generated every time a new version of labstate is saved in the following format `labstate_{timestamp}.json`.

2.1.4 Centrallized server (Tutorial)

The instructions below allow you to control multiple instruments connected to a network of hosts from a single location.

The basic setup is that there is one central lab computer that is the "instrumentation server." Other computers connect to the instruments through GPIB/USB/etc. These are "hosts." All of the hosts need National Instruments (NI) Measurement and Automation eXplorer (MAX). Start a *NI Visa Server* in each host, and naturally connect from the server via pyvisa.

Host machines

You first need to install NI-VISA in all machines, including the server, which can also play the dual role of a host, since it can also be connected to instruments. Download NI-VISA here. Installing for MacOS, Windows, Linux (Fedora-like) was a matter of following NI's instructions. Installing in ubuntu machines was a little trickier, but here is what worked for us.

Installing NI-visa on Windows

Todo: Include instructions.

Installing NI-visa on Windows

Warning: Currently not supported.

Installing NI-visa (32-bit) on Ubuntu (64-bit)

Followed instructions found here, but in computers with EFI secure boot, like all modern ones, we need to sign the kernel modules for and add the certificate to the EFI. For this, follow these instructions.

Sign all modules in `/lib/modules/newest_kernel/kernel/natinst/*/*/.ko`

Run the following after sudo updateNIdrivers (reboot required!):

```
kofiles=$(find /lib/modules/$(uname -r)/kernel/natinst | grep .ko)
for kofile in $kofiles; do
    sudo /usr/src/linux-headers-$(uname -r)/scripts/sign-file sha256 /home/tlima/MOK.
↪priv /home/tlima/MOK.der $kofile
done
```

Then start nipalk:

```
sudo modprobe nipalk
sudo /etc/init.d/nipal start
```

Test with:

```
visaconf  # for configuring, for example, GPIB interfaces
NIvisaic  # for testing instrument control
```

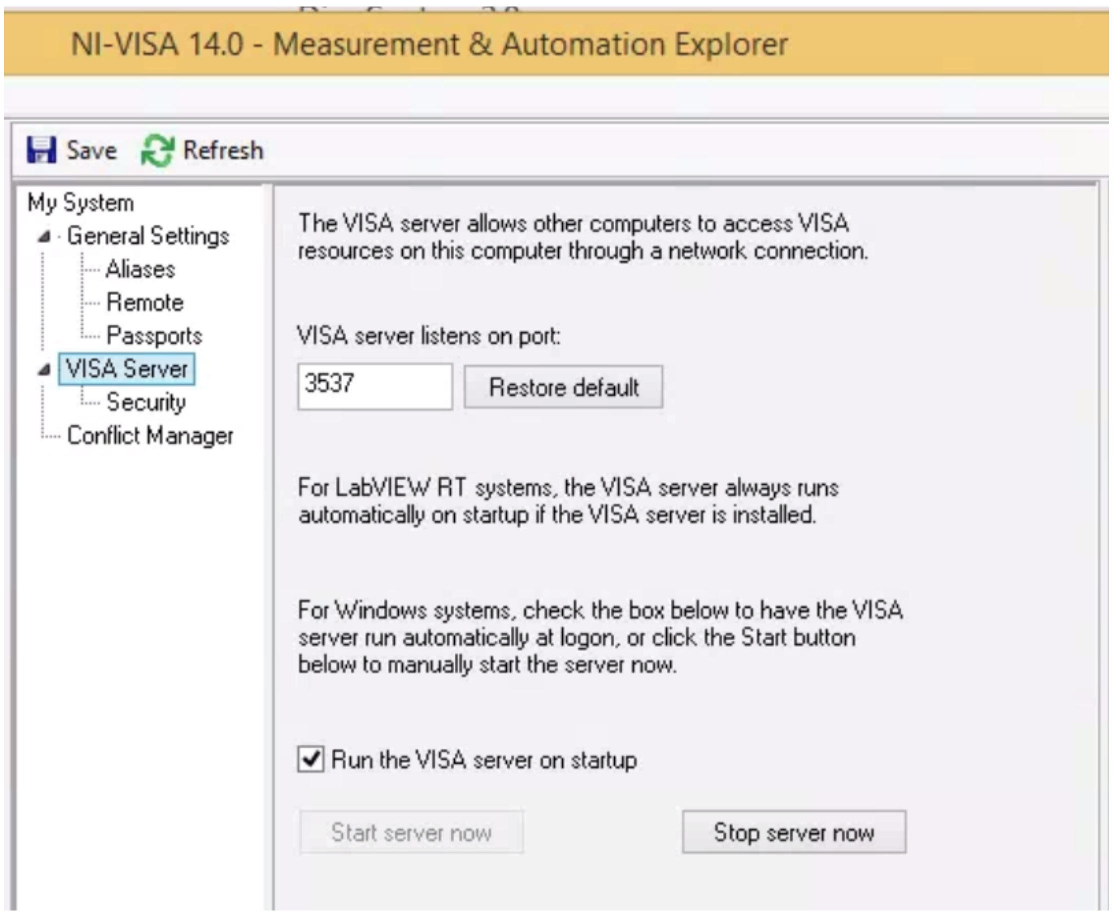
Opening NI-visa servers on all hosts

Open NI-MAX. In the main menu bar: Tools > NI-VISA > VISA options. This will open a panel.

In My System > VISA Server, check "Run the VISA server on startup." Click "Run Server Now."

In My System > VISA Server > Security, click the Add button, and put in a "*" under Remote Addresses. This white flags all other computers.

Click Save at the top left.


NI-VISA 14.0 - Measurement & Automation Explorer

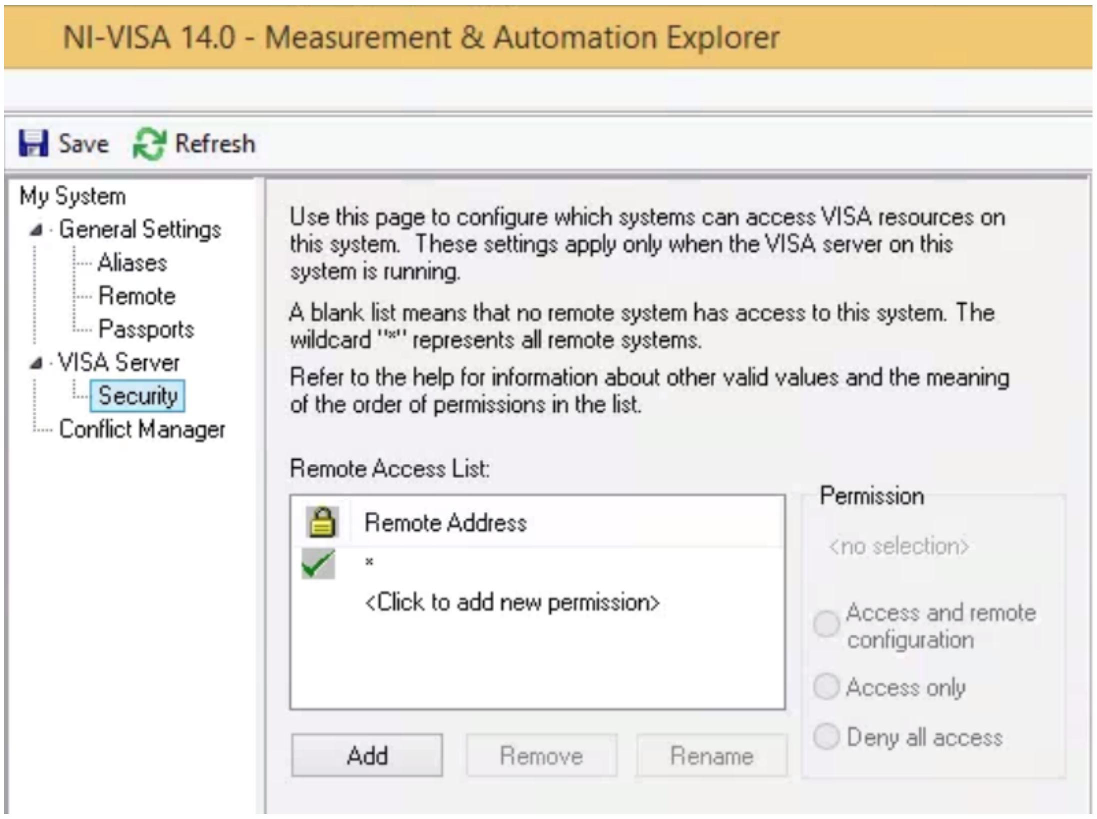
The VISA server allows other computers to access VISA resources on this computer through a network connection.

VISA server listens on port: 3537

For LabVIEW RT systems, the VISA server always runs automatically on startup if the VISA server is installed.

For Windows systems, check the box below to have the VISA server run automatically at logon, or click the Start button below to manually start the server now.

☑ Run the VISA server on startup


NI-VISA 14.0 - Measurement & Automation Explorer

Use this page to configure which systems can access VISA resources on this system. These settings apply only when the VISA server on this system is running.

A blank list means that no remote system has access to this system. The wildcard "*" represents all remote systems.

Refer to the help for information about other valid values and the meaning of the order of permissions in the list.

Remote Access List:

Remote Address
*
<Click to add new permission>

Permission: <no selection>

( ) Access and remote configuration
( ) Access only
( ) Deny all access

Troubleshooting

If you have been using Tektronix drivers, there might be a conflict with which VISA implementation will get used. These can be managed in the Conflict Manager tab.

General settings > Passports: Tulip sometimes gives trouble. The box should be checked, at least on 32-bit systems. Bugs were un-reproducible for us.

Instrumentation server machine

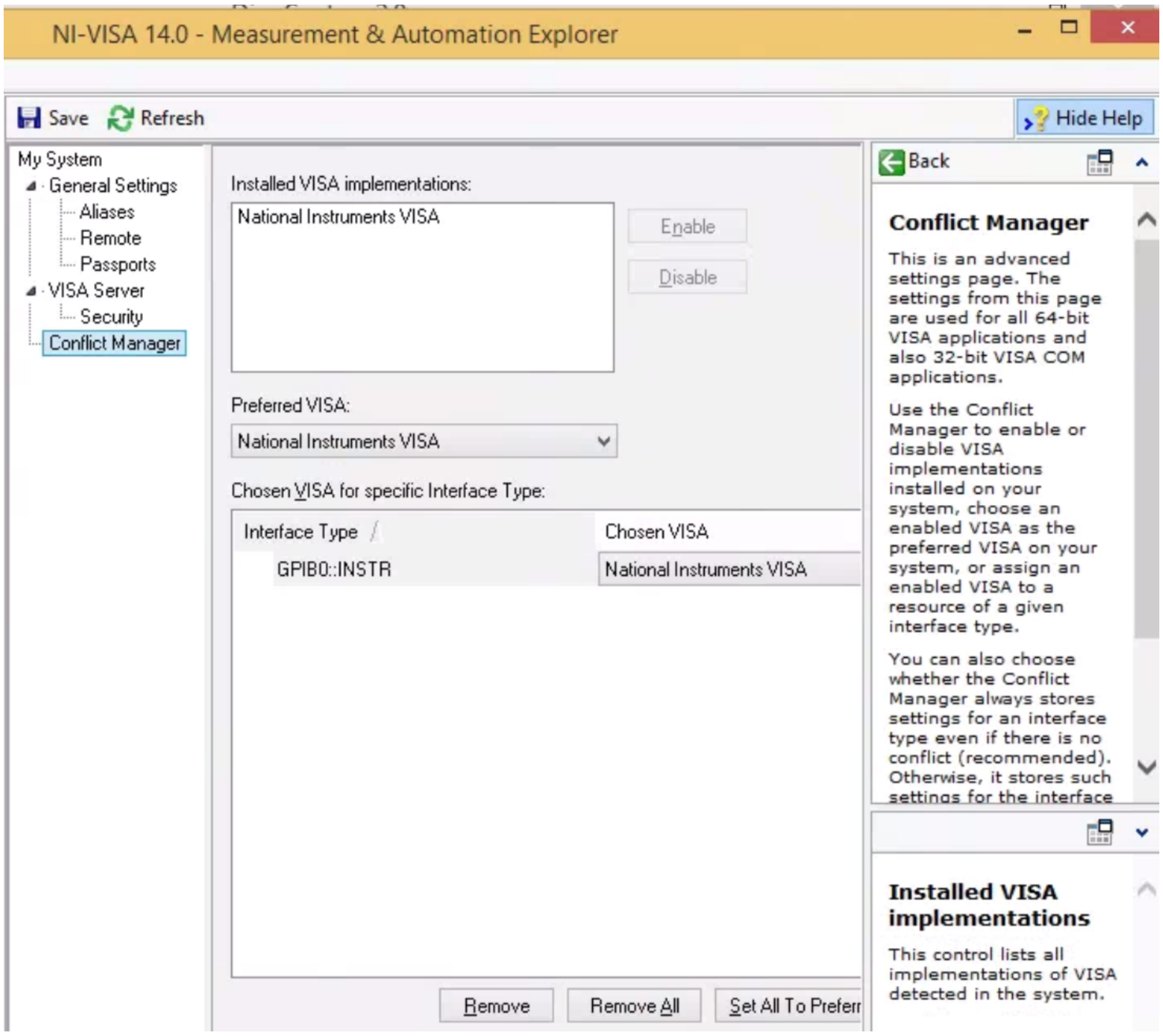
The below assumes that this system is Linux.

User configuration

There are several types of users.

- sysadmin (you)
- super-users a.k.a. root (you, possibly other lab members who know UNIX)
- `lightlab` developers
- `lightlab` users
- those with lab access, meaning they are allowed to configure and access hardware (you, most grad students)
- those without lab access, meaning they can still see data and write data analysis code (most undergrads)

In the below examples, we will use the following usernames



Installed VISA implementations:

National Instruments VISA

Preferred VISA:

National Instruments VISA

Chosen VISA for specific Interface Type:

Interface Type	Chosen VISA
GPIB0::INSTR	National Instruments VISA

Conflict Manager

This is an advanced settings page. The settings from this page are used for all 64-bit VISA applications and also 32-bit VISA COM applications.

Use the Conflict Manager to enable or disable VISA implementations installed on your system, choose an enabled VISA as the preferred VISA on your system, or assign an enabled VISA to a resource of a given interface type.

You can also choose whether the Conflict Manager always stores settings for an interface type even if there is no conflict (recommended). Otherwise, it stores such settings for the interface

Installed VISA implementations

This control lists all implementations of VISA detected in the system.

- arthur: you, sysadmin
- lancelot: a grad student and `lightlab` developer
- bedivere: a grad student user
- galahad: an undergrad who is anayzing bedivere's data

Set up a user on this computer corresponding to every user who will be using the lab. Make sure port 22 is open for ssh access. Give them all a tutorial on ssh, python, and ipython. Give yourself and lancelot a tutorial on git, SSHFS, pip, and jupyter.

Install basic tools globally

`lightlab` requires python 3.6. You also will need to use virtual environments to execute compiled code, install and freeze dependencies, and launch IPython servers. The first time, install these on your system environment:

```
$ sudo apt-get update
$ sudo apt-get install python3.6

$ sudo apt-get install python-pip python-dev build-essential
$ sudo pip install --upgrade pip
$ sudo pip install --upgrade virtualenv
```

For different versions of Ubuntu/Linux, you are on your own. See here and there.

Initializing labstate, setting lab accessors

Make a *jupyter* "user":

```
sudo useradd -m jupyter
sudo passwd jupyter
<enter a new password twice>
```

Make a *jupyter* group specifying who is allowed to run jupyter servers and change the labstate:

```
sudo groupadd jupyter
sudo usermod -a -G jupyter arthur
sudo usermod -a -G jupyter lancelot
sudo usermod -a -G jupyter bedivere
# <do not add galahad>
```

The *jupyter* user home directory can be accessed by any user and written only by the *jupyter* users:

```
cd /home
sudo chown root jupyter
sudo chgrp jupyter jupyter
sudo chmod a+r jupyter
sudo chmod a+x jupyter
sudo chmod g+w jupyter
```

We want to place `labstate.json` in `/home/jupyter/labstate.json`. As documented *above*, this can be done system-wide with:

```
# Running from an environment in which lightlab is installed
sudo lightlab config --system set labstate.filepath /home/jupyter/labstate.json
```

If anybody outside of group *jupyter* tries to change the labstate, it will not work.

The first time labstate is initialized, you'll want to add the hosts and benches in the lab. This is documented in *Making and changing the lab state*.

Handling virtual environments that install lightlab

Install virtualenvwrapper with pip.

To make all users see the same virtualenvwrapper, create a file in `/etc/profile.d/virtualenvwrapper.sh` and place the following bash script:

```
# Working with multiple virtualenv's
export WORKON_HOME=/home/jupyter/Envs
source /usr/local/bin/virtualenvwrapper.sh
```

Note: Make sure that `/home/jupyter/Envs` belongs to the group *jupyter* and that permissions are set so that users necessary can have access to it.

Then, follow the instructions (adapted from `virtualenvwrapper.sh`'s source code):

```
#  1. Create a directory to hold the virtual environments.
#     (mkdir /home/jupyter/Envs).
#  5. Run: workon
#  6. A list of environments, empty, is printed.
#  7. Run: mkvirtualenv lightlab
#  8. Run: workon
#  9. This time, the "lightlab" environment is included.
# 10. Run: workon lightlab
# 11. The virtual environment lightlab is activated.
```

Then, every user in the machine can call `workon lightlab` to activate lightlab's virtualenvironment.

Running a jupyter server for the regular users

Important: Securing a jupyter notebook server.

Please follow instructions in Securing a notebook server if you and more user plan to connect to the server remotely.

Jupyter notebooks can run arbitrary system commands. Since jupyter does not yet support key authentication, the only protection is strong passwords. There should *never* be a jupyter server launched by root.

Developers can run their own virtual environments, but there are two reasons to have a centralized one like this. 1) keeps data and notebooks centralized so they can be shared to outsiders and git-tracked easily, 2) serves users who are not developers and who therefore do not need an environment that links dynamically to lightlab.

Create a directory for your lab's data gathering notebooks and data. Ours is called lightdata:

```
cd /home/jupyter
mkdir lightdata
chgrp lightdata jupyter
chmod a+r lightdata
chmod a+x lightdata
```

(continues on next page)

(continued from previous page)

```
chmod g+w lightdata
chmod +t lightdata
```

The last line sets the sticky bit. That means when a file is created within that directory, it can only be modified or deleted by its owner (i.e. the person that created it).

Finally, after having adapted security instructions above, you should have an SSL certificate and port configuration setup in `/home/username/.jupyter/jupyter_notebook_config.py`, start your jupyter server from within the virtual environment by doing the following:

```
# logged in as any user in jupyter group
cd /home/jupyter/lightdata
workon lightlab

# in case you have just created this virtual environment
pip install lightlab

# and other packages you find useful. See our full list
# in dev-requirements.txt in our github page.
pip install jupyter pyusb pyserial

# set a password for your notebook. This will be stored
# in /home/username/.jupyter/jupyter_notebook_config.json
jupyter notebook password

# starts the jupyter notebook process and stays alive
# until stopped with Ctrl-C
jupyter notebook
```

If you have developers, set up CI for your own fork (optional)

If you are constantly helping with the development of lightlab, it is possible to utilize CI (continuous integration) to automate reinstallation of the package. In our case, we use Gitlab CI/CD in a different machine to trigger the deploy in the instrumentation server.

User: getting started

These are instructions that you may give to potential users in this setup. We recommend you placing the source code of lightlab inside `/home/jupyter/lightdata/lightlab` for their convenience. The source code has tutorial notebooks in `lightlab/notebooks`. We also recommend placing this documentation in `docs`, which can be modified by you, to make it easier. Jupyter servers can render `.md` files and can also serve `html` pages such as this one.

Connecting to the instrumentation server

First, make sure that your have a user account set up on the your server. Let's say your domain is "school.edu" First, do a manual log on to change your password to a good password. From your local machine:

```
$ ssh -p 22 <remote username>@<server hostname>.school.edu
<Enter old password>
$ passwd
<Enter old, default password, then the new one>
```

Make an RSA key

On your local machine:

```
ssh-keygen -t rsa -C "your.email@school.edu" -b 4096
```

You do not have to make a password on your ssh key twice, so press enter twice. Then copy that key to the server with:

```
$ ssh-copy-id <remote username>@<server hostname>.school.edu
<Enter new password>
```

Faster logging on

In your local machine, add the following lines to the file `~/.ssh/config`:

```
Host <short name>
     HostName <server name>.school.edu
     User <remote username>
     Port 22
     IdentityFile ~/.ssh/id_rsa
```

You can now `ssh <short name>`, but it is recommended that you use MOSH to connect to the server:

```
$ mosh <short name>
```

MOSH is great for spotty connections, or if you want to close your computer and reopen the ssh session automatically.

Using jupyter notebooks

Jupyter notebooks are interactive python sessions that run in a web browser. If you are just a user, your sysadmin will set up a notebook server and give you a URL and password. Some examples can be found in the `lightlab/notebooks/Tests` directory.

2.2 Getting Started to Python, Jupyter, git

Todo: Include more tutorial pages and useful links for intriductory python.

2.2.1 An engineer's guide to modern lab control

Author: *Thomas Ferreira de Lima* (tlima@princeton.edu)

Introduction

Over the years, software engineering has evolved into a very prominent field that penetrates all industrial sectors. Its core principles and philosophy was to make life easier for consumers to achieve their goals. That was when Apple and Microsoft were created. Then, as the field evolved, it has become important to make software engineering as inclusive as possible to new "developers", and to make collaboration as seamless as possible. This is the age of the apps. Now, software programming is becoming considered as fundamental as math and science, and are starting to enter school curricula.

Meanwhile, in academic circles and other engineering industries have lagged in software sophistication. Here, I propose a few techniques that we can borrow from software engineering to make our collaborative work in the lab more productive. My inspiration draws from the fact that in software engineering teams, the source code describes the entirety of a product. And if it is well documented, a new member of the team can learn and understand how it works in high or low level without the need for person-to-person training. In other words, all knowledge is documented in source code, instead of a mind hive. This is not the case in research groups. When a PhD student leaves, all his or her know-how suddenly exits the lab.

The concepts

Software programming

The first tool that is instrumental to this method is software programming. Computers were created with the intention to automate or facilitate menial tasks. The tendency is to delegate more and more of our labor to the machine, so we can move on to the bigger picture.

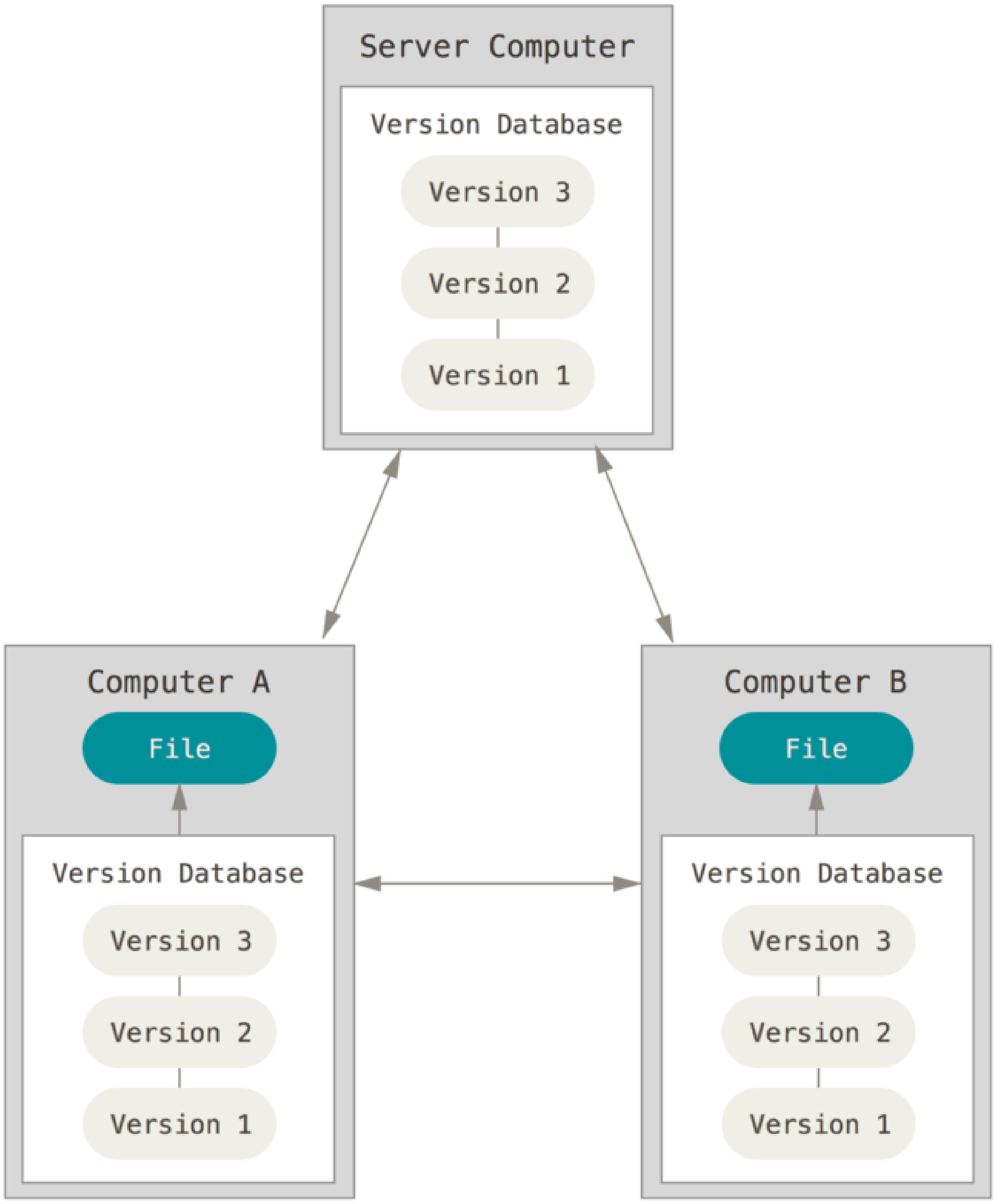
In our lab, a scientific experiment depends on controlling many instruments at the same time. The more complex the experiment, as they ten to be with integrated circuits, the more instruments are needed and the more complicated the calibration procedures and execution algorithms are. Most of these instruments are designed to have an electronic interface compatible with computers. These can be chiefly USB, which stands for Universal Serial Bus, or GPIB, for General Purpose Interface Bus, or Ethernet. Through these ports, computers can launch commands and probe results at the speed that the interface supports. As a result, one can control instrumentation of an experiment via the computer, i.e. via software. This is called a cyber-physical system. But this is not all. Software can also be used to perform any kind of algorithm. Which means that in a cyber-physical host, an experiment can be *defined* entirely by a computer program.

Computers programs can be written in a programming language, such as Python, MATLAB, C, Fortran, Java etc. There are many, but they all have the same purpose: to translate english words into machine code. Over a century of math, engineering, logic, and language science has passed since Ada Lovelace wrote the first algorithm intended for a machine. Python is a very modern language, still in active development, that became ubiquitous in the software engineering world due to its flexibility. It is considered a high-level language, meaning that its representation is very close to plain English, while its machine inner-workings are very hidden insides. Normally, these programming languages result in programs that are slower than the ones written in a more low-level language. Python's popularity stems from the fact that it can directly interface with a lot of these other faster languages, and it is fast enough for most people with modern computers. It also offers myriad open-source libraries that offer everything from web apps to numerical simulation to deep learning. So Python nowadays is the favorite first language of scientists, engineers, developers, students etc.

Version control

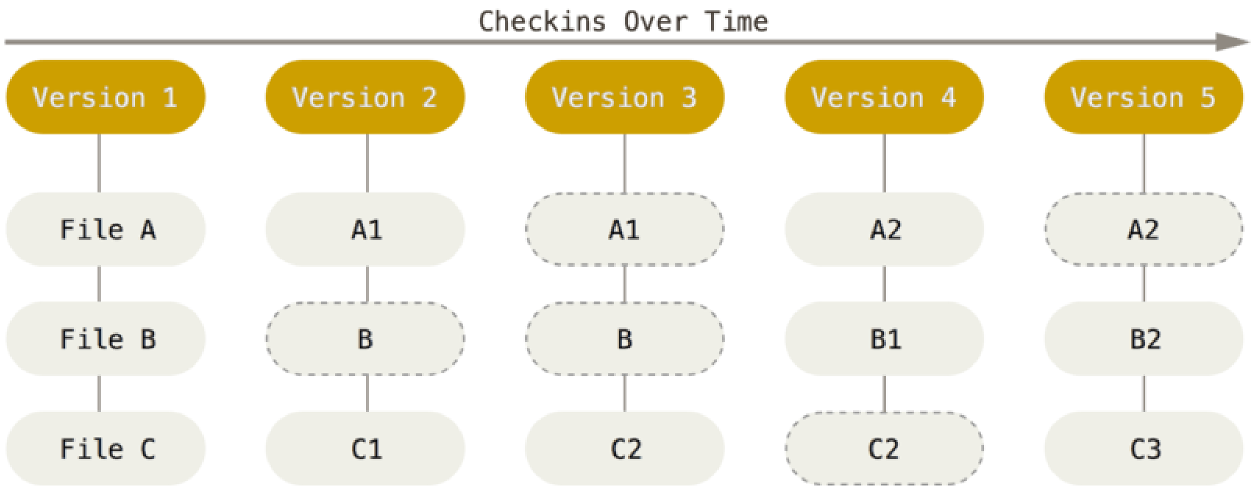
It is possible to use a particular programming language to write routines and small scripts that can have inputs, crunch some numbers, and produce outputs. However, actual programming languages were created to support Turing complete applications, which support an infinite complexity of internal states and behaviors. When source code became too complicated, i.e. around the time of the Apollo missions, computer scientists invented object-oriented programming, which made possible the modularization of source codes. This meant that programmers could change a piece of the code that interacted with the entire application without necessarily having to fully understand the entire source code. As a result, programmers needed a central location to store the code so they could edit it at the same time. This was called version control. Version control has become standard in all industries that deal with software. It is so efficient that it allows thousands of programmers to collaborate on an opensource project, each one submitting small changes, without risking introducing new bugs.

There are many technical ways to achieve version control, and many different software written to accommodate these techniques. The most popular are Git, Subversion, Mercurial and Microsoft's Team Foundation Server. Like it or not, today, Git dominates the version control software arena, and is rendering the others rather obsolete. So let's talk about version control as designed by Git's developers.



Version control with Git

The most basic concept of version control is revision tracking. Every revision to the source code is recorded by a “commit”. The commit records the changes made by the user respective to the previous revision. You can think of it as a linear graph, where the nodes represent the different revisions of the entire source code and the arrows the history connecting them. This is useful because the history of any project is automatically recorded and documented. Teams also use this feature to track how active their developers are.

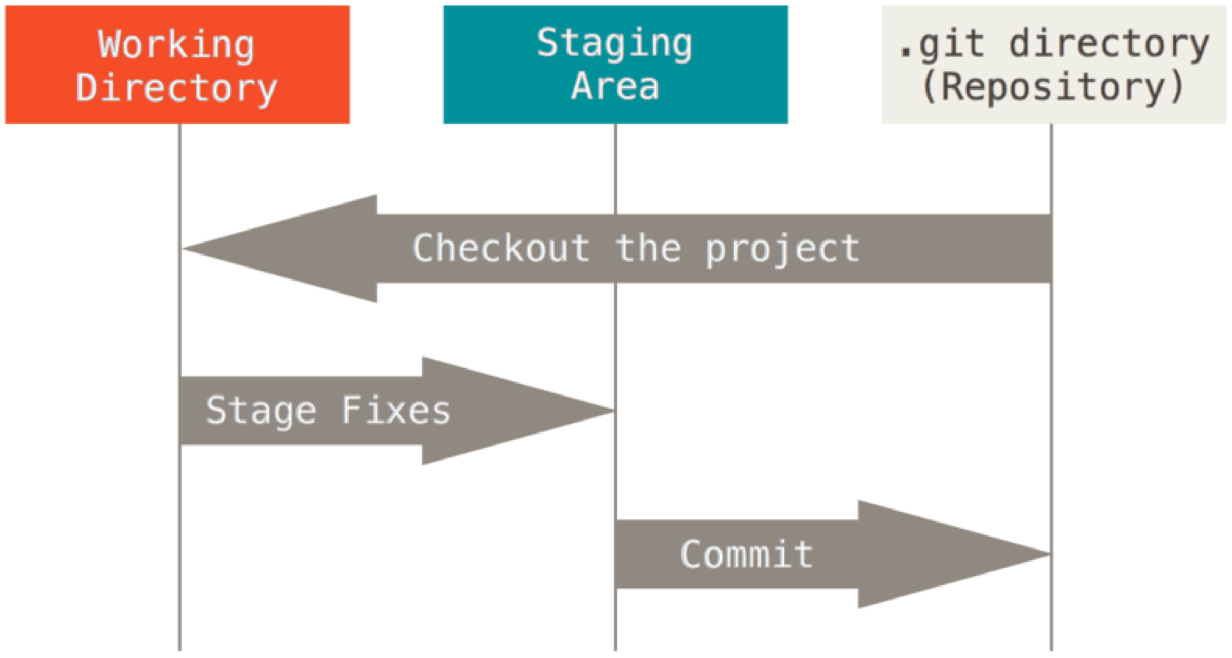


Commits are created and stored in what is called a repository, which is a data structure that keeps track of all commits made in history. In Git, this repository is stored in your computer, so that you can interact with it offline. The process typically works as follows. You work on the documents and code normally with your favorite editor, changing them on disk. When you have finished a desired set of changes, you create a commit and document what you have included in that particular commit, so that future you or collaborators have a sense of what changes were made before looking into the code. When a commit is triggered, the software automatically detects the changes that were made to every file, including whether it was deleted, renamed, or whether its metadata was changed. It then creates a manifest of all these changes, compresses it, and generate what is then called a "commit". After that, the commit is automatically stored in your "local repository", which is hidden inside a folder named ".git".

```
git commit -m "message"
```

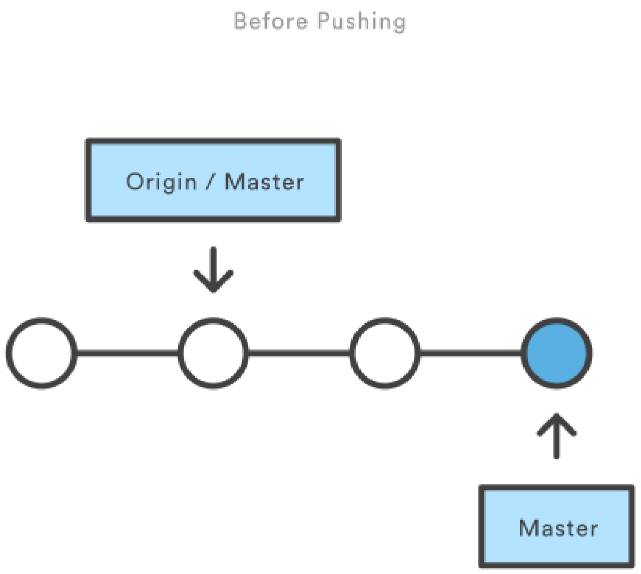
There are two concepts which, at this point, confuses most people unfamiliar with version control: staging and remote vs. local. But they are not complicated at all. The concept of staging can be understood by the following example. Say that there is a project/repository with two main parts: a numerical simulation part, and an experimental data processing part. Their code is contained in different files. You have made changes to both of these files because you are working on them at the same time, but you have finished implementing a desired change in the simulation file, but the one on experimental data is still in progress. Therefore, if you want to commit the changes you have made on the simulations while ignoring the rest, a staging step is necessary prior to commit. You add the simulation file to what is called a stage, leaving the experimental processing out of the stage. This allows you to commit only what is on the stage.

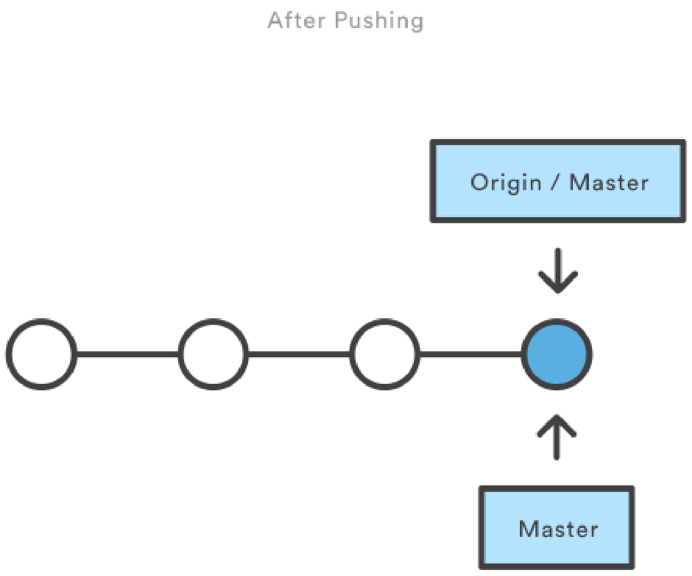
```
# Edit simulation file
git add simulation.py
git commit -m "finished simulation"
```



Another interesting property of Git is its ability to separate remote and local copies of the repository. In order to make the source code available to others, it needs to be uploaded somewhere remote. That is the raison d'être of a remote repository. There are web services that can host remote repositories, most famously GitHub, where virtually all the opensource projects are stored nowadays. The local copy of the repository is an exact and entire copy of the remote one, that is why one must "clone" it to the local computer. Clone, in this case, means download the current version plus all other versions in history. Therefore, after a commit is created in the local repository, it must be "pushed" to the remote copy so others can see it and "pull" to their local copy.

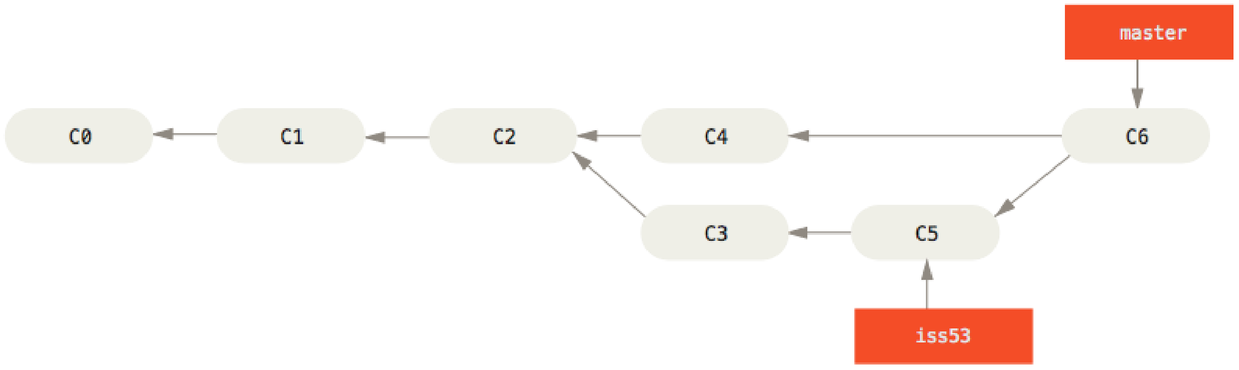
```
# Edit simulation file
git add simulation.py
git commit -m “finished simulation”
git push
```





The other main property of Git is that it can automatically "merge" a number of edits together in one step. Its algorithm is very powerful, works flawlessly when it can, and falls back to human intervention in case of "conflicts". When two collaborators create local commits, their history tree forks into two parallel versions that need to be conciliated. If one pushes first, the other's push will fail and abort, because its local repository does not agree with the most recent state of the remote repository. So the proper procedure is to sync the local with the remote by "pulling" changes from remote:

```
# Edit simulation file
git add simulation.py
git commit -m "finished simulation"
git pull  # this is where the merge happens
git push
```



The merge algorithm works in the following way. It attempts to add all modifications from both revisions to a stage. First, if the modified files are different, then both files are simply added to the stage. If the same file is modified, then Git will start a "diff" operation. It will go through line by line on each revision of the file until it detects a discrepancy. The revisions considered are the baseline (the revision agreed upon prior to the commit), the remote, and the local. Each discrepancy is judged as addition, deletion or simply edits. If Git detects a discrepancy both in the remote and the local commits, then a conflict is triggered, and the user must resolve it themselves by choosing to maintain changes from one revision or the other, or altering the line altogether. After the merge operation is finished, all files are added to the stage and a *new* commit is created. This commit is special because it has two "parents", so the history graph will look like three branches which merged together. Note that this process is designed such that no changes are lost during merge. It is an automatic way of doing a very tedious task that humans used to do in the past.

Here is a tutorial on Git.

Servers, hosts and clients

In order to make this all work, we need *servers*, *hosts*, and *clients*. A computer server can refer to the software or the device used in the "client—server" model. So you can have many *software* servers running on different *server* machines. As you can see, it can get complicated really fast. So unless otherwise specified, let us understand the word server as powerful computers that are expected to be turned on and connected to the network at all times.

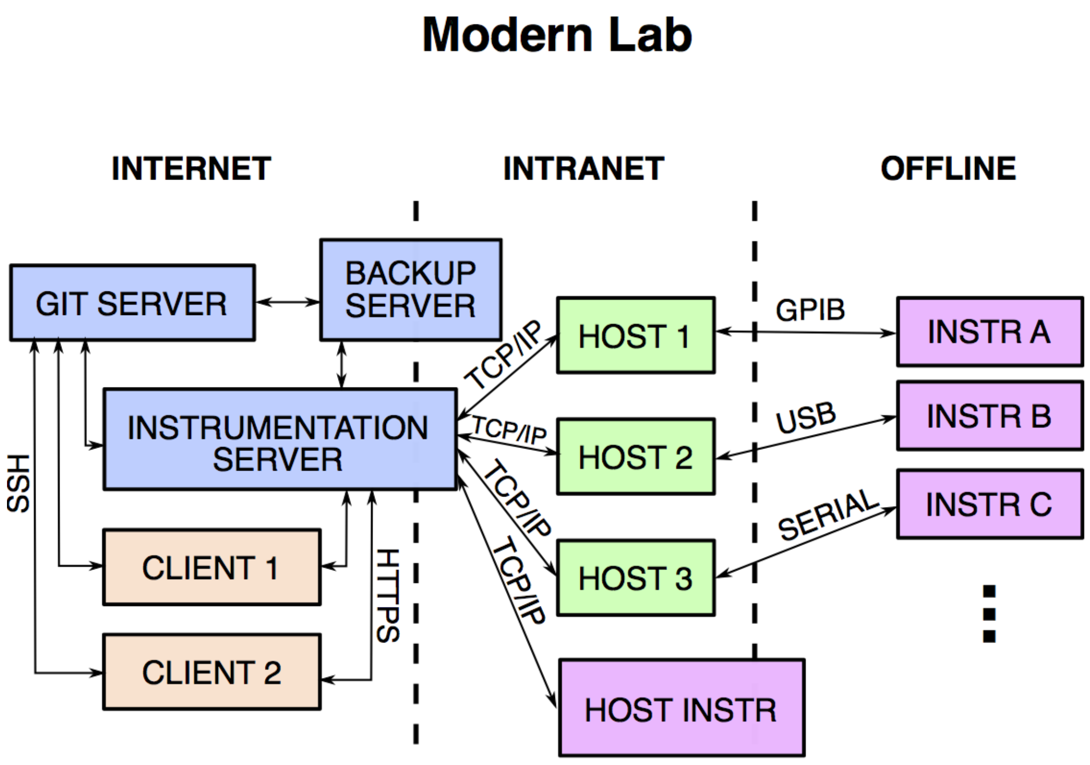
A *host* is any computer (or device) connected to the network. So all servers are hosts, but not all hosts are servers. If one wants to be able to control a certain instrument via the network, this instrument should either be a host itself or be connected to one via some interface bus. There are so many ways to do this that it would be counterproductive to introduce them all. But it is important to understand why these hosts cannot be servers. Simply, when you connect a new instrument to the host, sometimes one must install new software, update software drivers, or even reboot the machine. Stuff that cannot be allowed on a server that serves multiple clients at the same time.

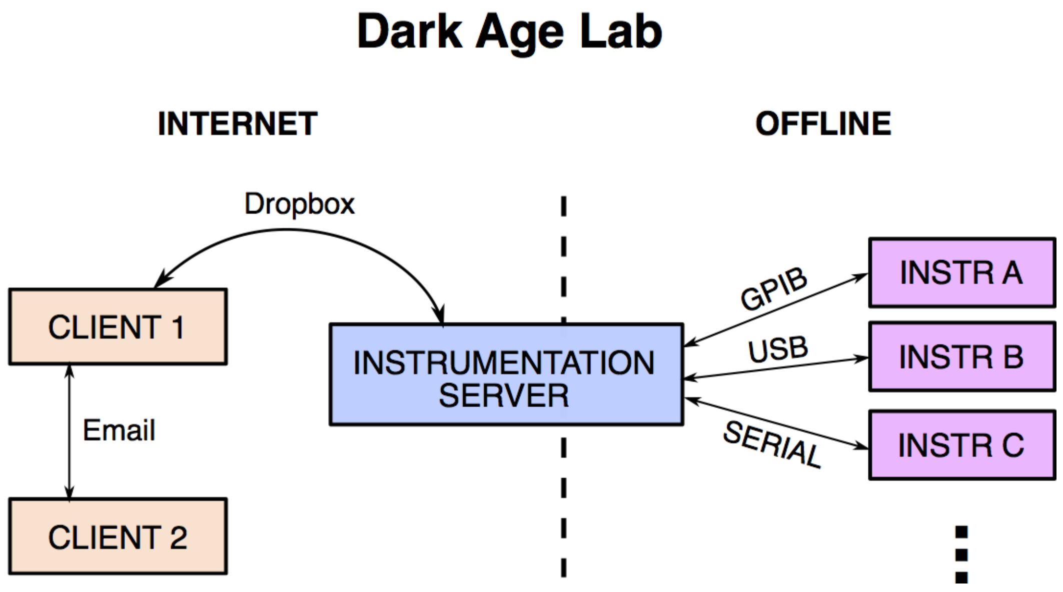
Finally, a *client* is a workstation that depends on resources offered by the server. It can be our personal computers.

In most research laboratories that require some sort of automation, researchers typically use one single computer to directly connect to the instruments that execute the experiment. A scientist can do this, download the data to her personal computer, go home, and crunch the numbers. This has been a good enough practice for simple experiments where there was a single person dealing with the instrumentation and the data analysis. However, when multiple persons need to have access to the most recent data, or even access to the experiment, it makes more sense to have a client—server—host implementation. In software engineering, the source code of some large projects such as Facebook grew to hundreds of gigabytes, with compilation times up to days. For them, having the source code stored and compiled on a supercomputer server is crucial.

The tools

In the following sections, I describe the tools that we need to be used to accommodate a team of two or more researchers operating various experiments in lab with multiple instruments connected to different hosts. Based on the the concepts described above, we need a central Git repository server, a server that connects to all hosts and a program that controls instruments and collects data from the hosts.





The Git server

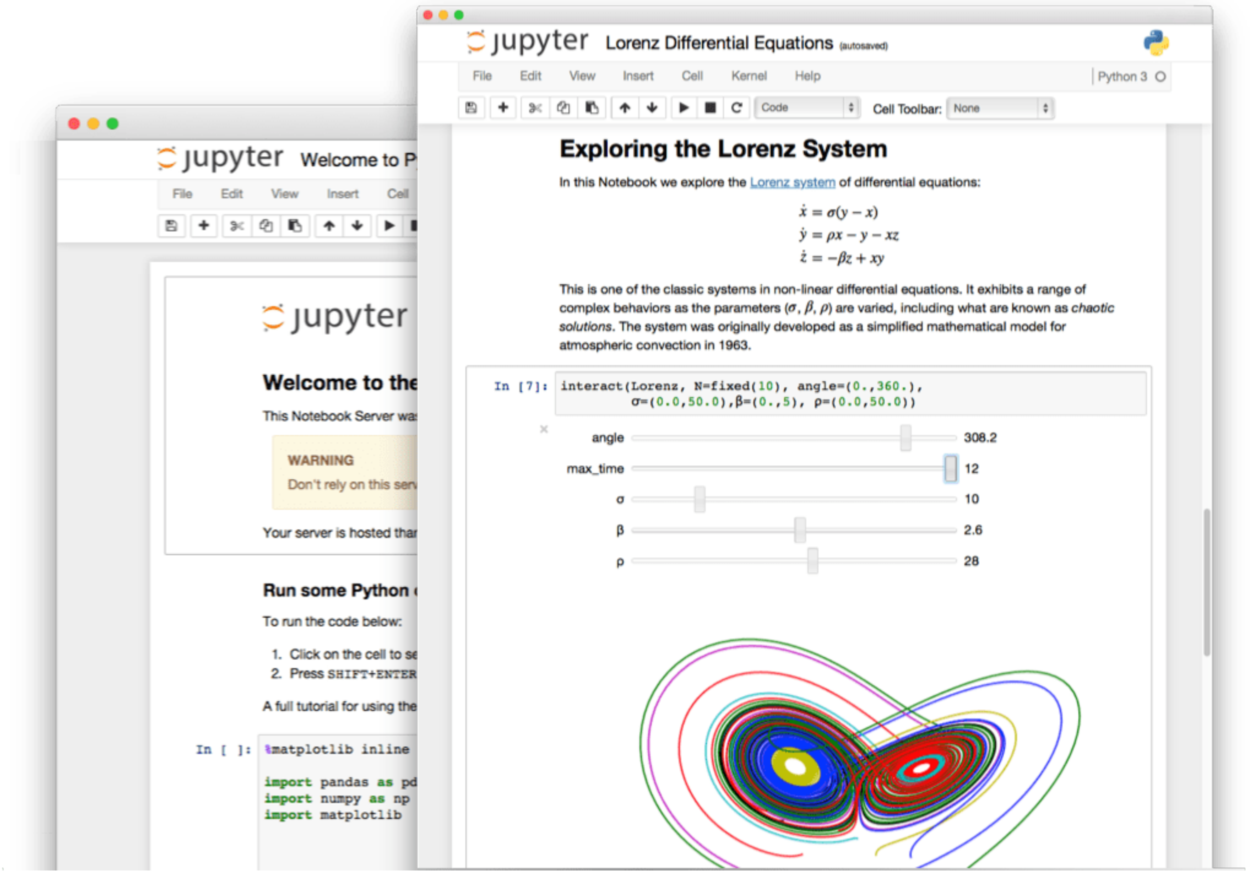
As previously mentioned, the Git repository is a set of files that can be stored anywhere. There are services online that offer free storage for opensource projects or paid storage for closed source projects. The most famous one is github.com. It is also possible to install a Git (software) server on a local server for free, so long as you possess the hardware. Gitlab, for example, has the same functionality of Github and it is also easy to use and install. It allows the admins to control which users have access to which repositories, which can be useful to protect confidential data. And since Git repositories are the same everywhere, they can be exported to other services very easily.

The instrumentation server

Another server has to be created and loaded with drivers from the instrument vendors, and also loaded with software modules that will support connecting to the hosts. This server can be created in the same machine as the Git one, but it is a good idea to separate them, because Git has to be extremely available at all times to everyone so that collaboration does not stop. It is quite a disturbance when Git goes offline, even if once a month, whereas the instrumentation server could go offline routinely for maintenance.

Software programming with Python notebooks

Python is a dynamic programming language, which in computer science means that it can be executed line by line instead of compiled into machine code. Because of this, Python can be used as a scripting language, like MATLAB, as well as a full-fledged object-oriented programming language, like C++. This flexibility means that one can build computer programs that are installed into the operational system of the computer, which can be accessed by Python scripts in the same computer or in another computer in the network. These programs, in Python language, are called *packages*. Other languages call them *libraries*, but essentially it means the same thing.



Exploring the Lorenz System

In this Notebook we explore the Lorenz system of differential equations:

$$\dot{x} = \sigma(y - x)$$
$$\dot{y} = \rho x - y - xz$$
$$\dot{z} = -\beta z + xy$$

This is one of the classic systems in non-linear differential equations. It exhibits a range of complex behaviors as the parameters (σ, β, ρ) are varied, including what are known as *chaotic solutions*. The system was originally developed as a simplified mathematical model for atmospheric convection in 1963.

```
In [7]: interact(Lorenz, N=fixed(10), angle=(0.,360.),
                 σ=(0.0,50.0),β=(0.,5), ρ=(0.0,50.0))
```

angle	308.2
max_time	12
σ	10
β	2.6
ρ	28

Welcome to the

This Notebook Server wa

WARNING

Don't rely on this serv

Your server is hosted thar

Run some Python

To run the code below:

1. Click on the cell to se
2. Press SHIFT+ENTER

A full tutorial for using the

```
In [ ]: %matplotlib inline

        import pandas as pd
        import numpy as np
        import matplotlib
```

A Jupyter notebook is an "opensource web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text." It is a kind of document that exists "live" in a server, like Google Doc. It is interactive and can be shared with other users. Here is a list of interesting jupyter notebooks. It can be used to plot data beautifully, write LaTeX annotations, and store logic and results in the same file! If this notebook application is installed in the instrumentation server, one gains the ability to interactively control experiments, collect data, analyze it, and plot publication-quality figures on the same notebook. This workflow, combined with the possibility of "versioning" the notebooks in a Git repository, is a superior way of making sure the experiments are reproducible, well documented, and self-explanatory to anyone in the lab who wants to start afresh.

The lightlab package

The *lightlab* Python package is being developed in the Lightwave Lab to be essentially our own version of LabVIEW + MATLAB. The opensource community built enough libraries for Python that would render these two software obsolete. While many companies still release drivers and plugins for LabVIEW and MATLAB, they are also easy to interface with opensource libraries. As of 2017, we can essentially control every remote-controlled instrument with the *lightlab* package.

The lightlab package contains three things: instrument drivers, laboratory virtualization, and calibration models for photonic devices. (It has been decided to remove the calibration models from the project, and give it its own package, so I will not explore it here).

Instrument drivers

Instrument drivers are pieces of code responsible to command and control instruments. For example, a Keithley 2400 source meter can be controlled via GPIB protocol. National Instruments offers a set of low-level drivers (NI-VISA) that can be installed in Linux or Windows hosts, which allows us to establish connection, send and receive GPIB (or, more modernly, VISA) commands easily. These are files that have to be installed directly into the operational system. Then, we can install an opensource package called PyVISA, written in Python, which interfaces with the low-level NI-VISA drivers. The lightlab package contains an object built onto PyVISA, representing the Keithley 2400 instrument. This object contains functions that can translate commands such as turn on, turn off, ramp up current or voltage, read resistance, voltage or current values; into VISA commands that can be sent through the NI-VISA drivers. This object can be accessed directly from the Jupyter notebook.

Laboratory virtualization (under development)

Another thing present in the lightlab package is the virtualization of instruments. The idea is very well suited for automated testing and data collection of devices. In this module, every object that we interact with in lab will have a corresponding Python object. An instrument is an object that understand where the instrument is located in lab, where it is connected to, and via what host it can be accessed to. Similarly, a device object contains a map of different ports it can be connected to. This way, users can design the experiment entirely on the computer with a Python notebook, simulate the expected behavior, and **using the same code**, perform the experiment in real life. This creates the idea of a "source code " of the experiment, which can be executed by future users or users in different labs with different instruments.

Appendix

Bash

Digital security

Private keys

Two-factor authentication

2.3 Making your changes to lightlab

The following texts should help you in the process or making changes to lightlab itself. If you are looking for a way to include your own instrument driver, you will find instructions in *Using and creating drivers for instruments*. After you are done, please consider submitting a pull request via github! :) We the community will be glad to provide feedback.

2.3.1 Developer Guide

This section covers topics that have great documentation online. The main differences in this workflow stem from the hardware aspect of lightlab. That means almost all development should occur on the machine in lab that is going to be accessing the instruments. First, follow the instructions for connecting to the instrumentation server for users.

In this section

Setting up Git

Your sysadmin should go on github and fork the repo into their or the lab's github account. Alternatively (not recommended), you can download the project and make a new repo on an internal Git server, like GitLab.

On the instrumentation server, go in and clone that repo. When you develop, it should be on a branch based off of the development branch. There is no need to clone the repo to your local computer.

File system sync

It is recommended that you use SSHFS to mirror your work to your local computer, so you can use your favorite text editor. While you are editing on your local machine, you should have a ssh session on the remote machine in order to run the commands.

1. Install SSHFS on your local system.
   - Linux: `sudo apt-get install sshfs`
   - **OSX: Download binaries and then**
     - Install FUSE for macOS
     - Install SSHFS for macOS
2. Make shortcuts in your `.bashrc` or `.bash_profile`

Linux:

```
alias mntlight='sshfs <server>:/home/lancelot/Documents /path/to/local/dir -C -o␣
↪allow_other'
alias umntlight='fusermount -u /path/to/local/dir'
```

MacOS:

```
alias mntlight='sshfs <server>:/home/lancelot/Documents /path/to/local/dir -C -o␣
↪allow_other,auto_cache,reconnect,defer_permissions,noappledouble'
alias umntlight='umount /path/to/local/dir'
```

4. Now you can mount and unmount your remote calibration-instrumentation folder with:

   ```
   $ mntlight
   $ unmtlight
   ```

Example directory structure and environment: Non-developers

Lightlab is meant to be used by other code, usually via jupyter notebooks. We suggest that this user code be in a virtual environment with the following structure

```
> bedivere/Documents
| > myWork
| | > .git (optional)
| | requirements.txt
| | Makefile
| | > notebooks
| | | labSetup.ipynb
| | | gatherData.ipynb
| | -
| | > data
| | | someData.pkl
| | -
| -
-
```

Where the contents of `requirements.txt` will include "lightlab" and other packages you need for your work.

The Makefile has targets for making a virtual environment and launching jupyter

```
venv: venv/bin/activate
venv/bin/activate: requirements.txt
    test -d venv || virtualenv -p python3 --prompt "(myWork-venv) " --distribute venv
    venv/bin/pip install -Ur requirements.txt
    touch venv/bin/activate

jupyter:
    source venv/bin/activate; jupyter notebook; \

getjpass: venv
    venv/bin/python -c 'from notebook.auth import passwd; print(passwd())'
```

With these things in place, you can run `make jupyter` have a fully fledged, clean environment with lightlab installed.

Notebooks contain the procedures used to configure labstate, gather data, save data, analyze data, and make nice plots you can use in papers. The `labSetup.ipynb` file will look like *this one*, but populated with your lab's sensitive addresses, ports, namespaces, etc.

Example directory structure and environment: Developers

If you are developing lightlab, you will likely have some other notebooks to test. Those should go in a different directory with a different virtual environment. It can be git tracked in a different repo. Here is an example directory structure:

```
> lancelot/Documents
| > lightlab
| | > .git
| | Makefile
| | setup.py
| | etc...
| -
| > myWork
| | > .git (optional)
| | requirements.txt
| | Makefile
| | .pathtolightlab
| | > notebooks
| | | labSetup.ipynb
| | | gatherData.ipynb
| | -
| | > data
| | | someData.pkl
| | -
| -
-
```

Where the Makefile has a modification for dynamic installation of lightlab.

```
# myStuff/Makefile
PATH2LIGHTLABFILE=.pathtolightlab

venv: venv/bin/activate
venv/bin/activate: requirements.txt
    test -d venv || virtualenv -p python3 --prompt "(myWork-venv) " --distribute venv
    venv/bin/pip install -Ur requirements.txt
    touch venv/bin/activate
    source venv/bin/activate; venv/bin/pip install -e $(shell cat
↪$(PATH2LIGHTLABFILE))
```

The highlighted line will dynamically link the environment to your version of lightlab under development. If you have autoreload on in ipython, then text changes in lightlab will take effect immediately (excluding adding new methods). It is important that “lightlab” is **not** in your `requirements.txt` file.

The contents of `.pathtolightlab` are:

```
/home/lancelot/Documents/lightlab
```

If myWork is a git repo, your `.gitignore` should include:

```
.pathtolightlab
```

Running jupyter from your `myWork` environment

Password protect

Jupyter lets you run commands on your machine from a web browser. That is dangerous because anybody with an iphone can obliviate your computer with `rm -rf /`, and they can obliviate your research with `currentSource(applyAmps=1e6)`. Be safe on this one.

On the lab computer, copy and modify the provided template:

```
$ mkdir ~/.jupyter
$ cp /home/jupyter/.jupyter/jupyter_notebook_config.py ~/.jupyter
```

then generate a password with:

```
$ make getjpass
Enter password: <Enters password>
Verify password: <Enters password>
```

This will produce one line containing a hash of that password of the form:

```
sha1:b61b...frq
```

Choose an unused port. Port allocations on your lab computer should be discussed with your group. Let's say you got :8885.

When you have a port and a password hash, update the config file:

```
$ nano ~/.jupyter/jupyter_notebook_config.py
```

```
...
## Hashed password to use for web authentication.
c.NotebookApp.password = 'sha1:b61b...frq' # hash from above
...
## The port the notebook server will listen on.
c.NotebookApp.port = 8885 # port from above
```

Launch the server

To launch the server from `myWork`, just run:

```
$ make jupyter
```

(see Makefile target above). Except that will lock up your shell session. Instead, you can spin off a process to serve jupyter in a tmux:

```
$ tmux new -s myNotebookServer
$ make jupyter
<Ctrl-b, d>  # to detach
```

You can now acces your notebooks anywhere with your password at: `https://<server name>.school.edu:<port>`.

If for some reason you want to reconnect to this process, you can use `tmux attach-process -t myNotebookServer` or `tmux ls` followed by picking the right name.

Git and jupyter

They do not play nice. Here are some *strategies* for not losing too much work.

Running monitor server from your `myWork` environment

`lightlab` offers tools for monitoring progress of long sweeps. See *`ProgressWriter`*. These servers are launched from your own environment, not lightlab's. So far, this is just for long sweeps that simply tell you how far along they are, and when they will complete.

First, you must get another port allocated to you, different from the one you used for Jupyter. Put that in a file called `.monitorhostport` in `myWork` (where the Makefile is). Let's say that port is 8000:

```
$ echo 8000 > .monitorhostport
$ mkdir progress-monitor
```

Add the following target to your `Makefile`:

```
monitorhost:
    ( \
        source venv/bin/activate; \
        cd progress-monitor; \
        python3 -m http.server $(shell cat .monitorhostport); \
    )
```

If this is a repo, add the following to `.gitignore`:

```
.monitorhostport
progress-monitor/*
```

To then launch the server from a tmux:

```
$ tmux new -s myMonitorServer
$ make monitorhost
<Ctrl-b, d>  # to detach
```

Note: I have tried making a daemon launch automatically from the lightlab.util.io library. I have not yet verified that it is safe, so it is currently disabled.

Todo: How will this work for non-developers?

Testing `myWork`

It's not really necessary in this example where there is just a notebook. If you are developing your own library-like functions, it is generally good practice, but

Never put hardware accessing methods in a unittest

Unittests are designed to be run in an automated way in a repeatable setting. Firstly, the real world is not repeatable. Secondly, an automated run could do something unintended and damaging to the currently connected devices.

- genindex

- modindex
- search

2.3.2 Contributing to `lightlab`

We follow this Git branching workflow. Feature branches should base off of development; when they are done, they must pass tests and test-nb's; finally they are merged to development.

Testing `lightlab`

First off, your change should not break existing code. You can run automated tests like this:

```
make test-unit
make test-nb
```

The test-nb target runs the **notebooks** in notebooks/Tests. This is a cool feature because it allows you to go in with jupyter and see what's happening if it fails. We recommend using the nbval approach. It checks for no-exceptions, not accuracy of results. If you want to check for accuracy of results, do something like:

```
x = 1 + 1
assert x == 2
```

in the cell.

Make tests for your features! It helps a lot. Again, **Never put hardware accessing methods in a unittest**.

To run just one test, use a command like:

```
$ source venv/bin/activate
$ py.test --nbval-lax notebooks/Tests/TestBook.ipynb
```

Documenting

Documenting as you go is helpful for other developers and code reviewers. So useful that we made a whole *tutorial* on it. We use auto-API so that docstrings in code make it into the official documentation.

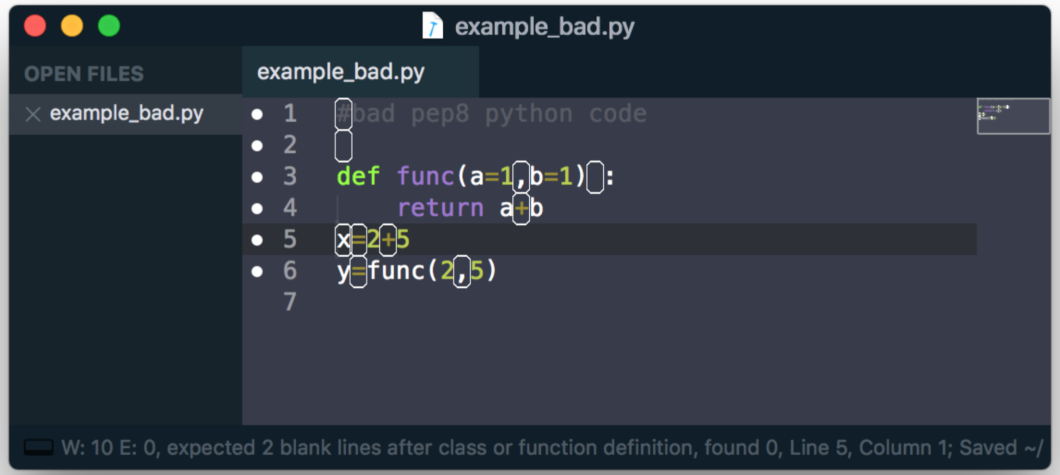
For non-hardware features, a good strategy is to use tests that are both functional and documentation by example. In cases where visualization is helpful, use notebook-based, which can be linked from this documentation or in-library docstrings like this. Otherwise, you can make pytest unittests in the tests directory, which can be linked like this: `test_virtualization`.

For new hardware drivers, as a general rule, document its basic behavior in `lightlab/notebooks/BasicHardwareTests`. Make sure to save with outputs. Finally, link it in the docstring like this:

```
class Tektronix_DPO4034_Oscope(VISAInstrumentDriver, TekScopeAbstract):
''' Slow DPO scope. See abstract driver for description

    `Manual <http://websrv.mece.ualberta.ca/electrowiki/images/8/8b/MSO4054_
↪Programmer_Manual.pdf>`__

    Usage: :any:`/ipynbs/Hardware/Oscilloscope.ipynb`

'''
instrument_category = Oscilloscope
...
```

Linting

As of now, we don't require strict PEP-8 compliance, but we might in the future. However, we try to follow as many of their guidelines as possible:



```
#bad pep8 python code

def func(a=1,b=1) :
    return a+b
x=2+5
y=func(2,5)
```

W: 10 E: 0, expected 2 blank lines after class or function definition, found 0, Line 5, Column 1; Saved ~/

Fig. 1: Example of valid python code that violates some of the PEP8 guidelines.

Sometimes the linter is wrong. You can tell it to ignore lines by adding comment flags like the following example:

```
x = [x for x in sketchy_iterable]  # pylint: disable=not-an-iterable
from badPractice import *  # noqa
```

`# noqa` is going to ignore pyflakes linting, whereas `# pylint` configures *pylint* behavior.

If you use Sublime editor

Everyone has their favorite editor. We like Sublime Text. If you use Sublime, here is a good linter. It visually shows what is going on while you code, saving lots of headaches

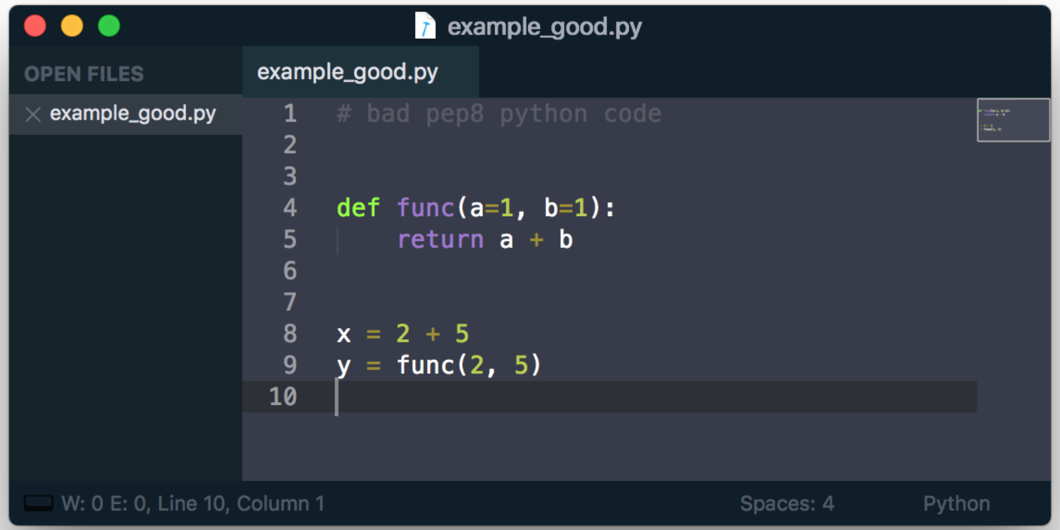
Sublime also helps you organize your files, autocomplete, and manage whitespace. This is sublime-lightlab. Put it in the `lightlab/` directory and call it something like `sublime-lightlab.sublime-project`.

By the way, you can make a command-line Sublime by doing this in Terminal (for MacOS):

```
ln -s "/Applications/Sublime Text.app/Contents/SharedSupport/bin/subl" /usr/local/bin/
↪subl
```

Adding a new package

Two ways to do this. The preferred method is to add it to the package requirements in `setup.py`. The other way is in the venv. In that case, make sure you freeze the new package to the requirements file:



```python
# bad pep8 python code


def func(a=1, b=1):
    return a + b


x = 2 + 5
y = func(2, 5)

```

Fig. 2: Fixing the PEP8 violations of the previous figure.

```
$ source venv/bin/activate
$ pip install <package>
$ make pip-freeze
$ git commit -m "added package <package> to venv"
```

Warning: If your code imports an external package, the sphinx documentation will try to load it and fail. The solution is to mock it. Lets say your source file wants to import:

```python
import scipy.optimize as opt
```

For this to pass and build the docs, you have to go into the `docs/sphinx/conf.py` file. Then add that package to the list of mocks like so:

```python
MOCK_MODULES = [<other stuff>, 'scipy.optimize']
```

2.3.3 How to document your code

In this section

This documentation is created with Sphinx. It has automatic API build, so *write good docstrings*! Documentation for the latest stable lightlab is on readthedocs (link).

Todo: link to the yet-to-exist readthedocs page

If you set up CI for your lab's fork, you might choose to host the documentation based on your lab's development branch. See the Travis for an example of that.

When you are developing, you can build what is in your HEAD directory with `make docs`. Presumably, you have SSHFS setup to your instrumentation server. On that mount, navigate to `lightlab/docs/_build/html/` and open `index.html`. This will use your default browser.

1. Manually

You can write documentation pages manually in the `docs/_static/` directory using ReSt

- ReST primer

In this documentation at the upper right corner, there is a "View page source" link that is very useful.

Bibliographic references

Use inline references with the ``:cite:`auth:99` `` directive.

At the end of the page, put this command to display the reference:

```
.. bibliography:: /light-bibliography.bib
```

The bibtex source is located at *docs/light-bibliography.bib*

2. Via the docstrings

Documentation of API is autogenerated. That means whatever you put in the code docstrings will end up formatted nicely on this site. It also means you have to follow some rules about it.

You should do functions like this:

```
def foo(a, b, *args):
    ''' My cool function
        << Blank line causes a rendered line break >>
        This function does some stuff with ``a`` and ``b``:
            * one thing
            * another thing
        << Blank line after indented thing, otherwise you get Warnings >>
        Pretty neat eh?
        << Blank line before argument list, otherwise you get Warnings >>
        Args:
            a (int): an input
            b (int): another input
            \*args: more inputs
```

(continues on next page)

(continued from previous page)

```
        Returns:
            (int): an output
    '''
```

This is called Google docstring format. It will render as follows.

foo (*a*, *b*, **args*)

My cool function

This function does some stuff with `a` and `b`:

- one thing
- another thing

Pretty neat eh?

Parameters

- **a** (*int*) – an input
- **b** (*int*) – another input
- ***args** – more inputs

Returns an output

Note, if you look at the source of this .rst file, the rendered documentation is in python format using lists of `:param:`. You should use python docstring format if manually documenting in the doc source. In the *code*, use Google format.

Real examples can be found by browsing the API section of this documentation. If you see something you like, click on the link to view the source. Then you can see how the docstring did that.

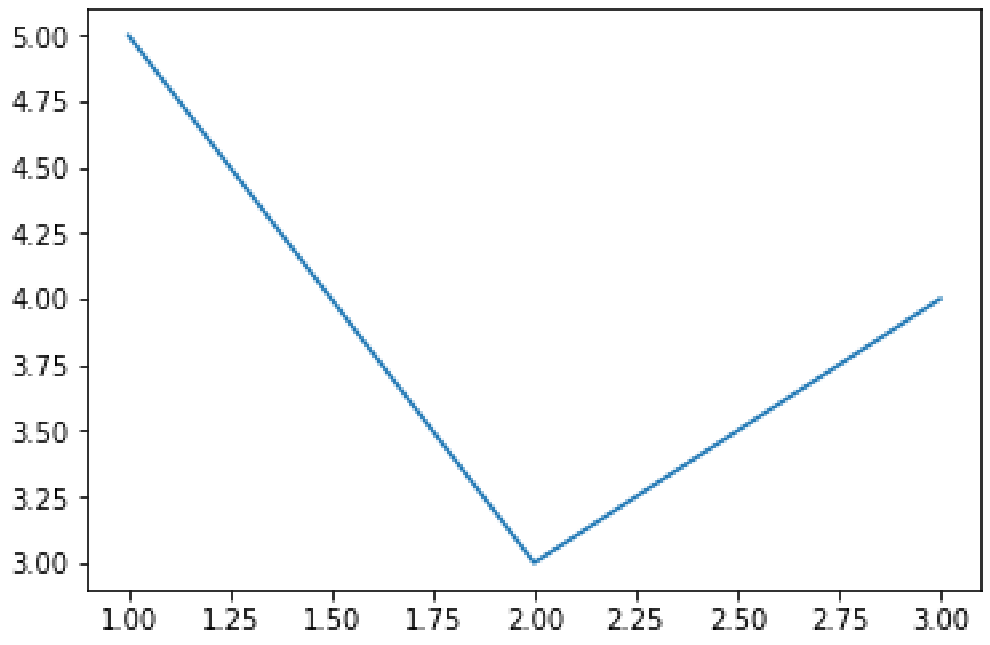
3. Via IPython Notebooks

The nbsphinx package by Matthias Geier can convert .ipynb files with outputs into html. The idea here is that it is sometimes instructive for the reader to play with some knobs to see how something works. Real code examples are also useful. It also supports running on build, but that is not recommended. These notebooks should be saved with outputs.

As of now, documentation notebooks are in lightlab/docs/ipynbs/. "Tests/" notebooks should correspond exactly to what is in lightlab/notebooks/Tests, and basic "Hardware/" notebooks should correspond to lightlab/notebooks/BasicHardwareBehavior. After running and saving, *copy* that notebook over (do not try to symlink). Other notebooks can be placed in Others/. You can reference them in the documentation like so

Example Notebook

```
In [3]: import matplotlib.pyplot as plt
        plt.plot([1,2,3], [5, 3, 4])
        plt.show()
```



Generic N-D sweeps

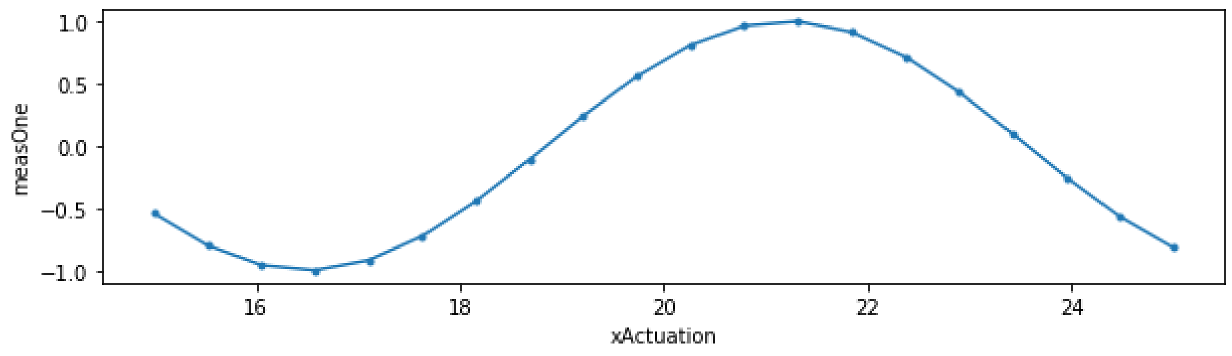
```
In [1]: %load_ext autoreload
        %autoreload 2

        import matplotlib.pyplot as plt
        import numpy as np
        from lightlab.util.sweep import NdSweeper
```

```
In [2]: # Turn this on when executing interactively
        livePlots = False
```

```
In [3]: class Plant():
            def __init__(self):
                self.x = 2
                self.y = 2

            def actuateX(self, newX, rounded=False):
                self.x = round(newX) if rounded else newX

            def actuateY(self, newY):
                self.y = newY

            def measure(self):
                return (np.sin(self.x * self.y/3), self.y / self.x)
```

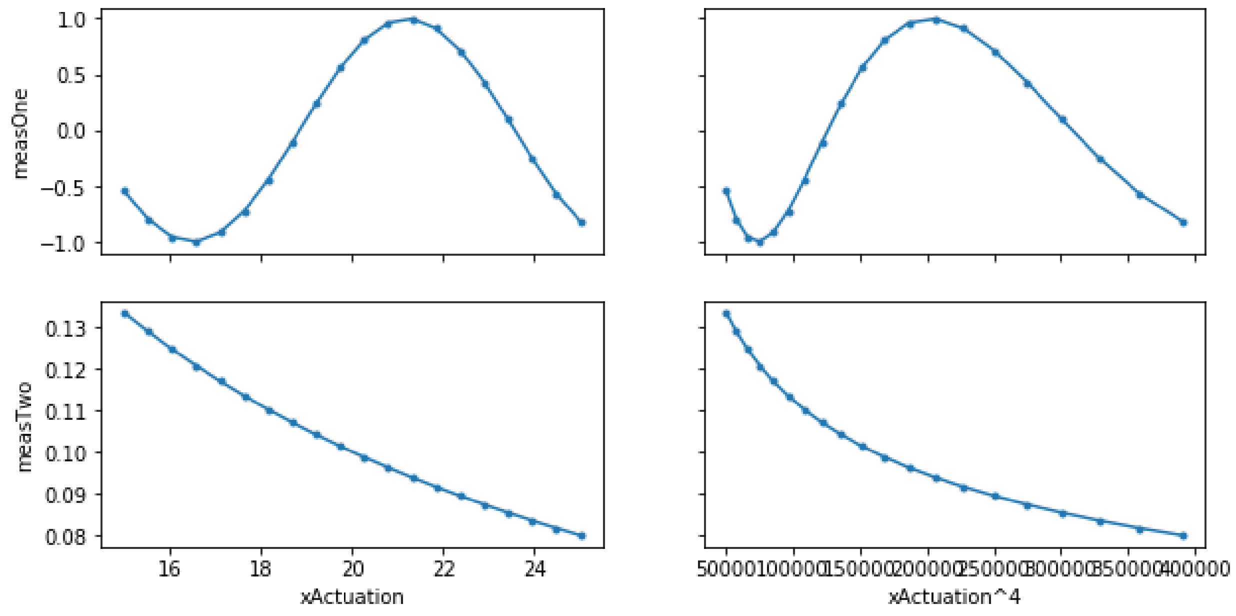
Simplest case

```
In [4]: p = Plant()

        swpInX = NdSweeper()
        swpInX.addActuation('xActuation', lambda x: p.actuateX(x), np.linspace(15, 25, 20))
        swpInX.addMeasurement('measOne', lambda: p.measure()[0])
        swpInX.setMonitorOptions(stdoutPrint=False, livePlot=livePlots)

        swpInX.gather()
        if not livePlots:
            swpInX.plot()
```



Multiple measurements and a domain parser

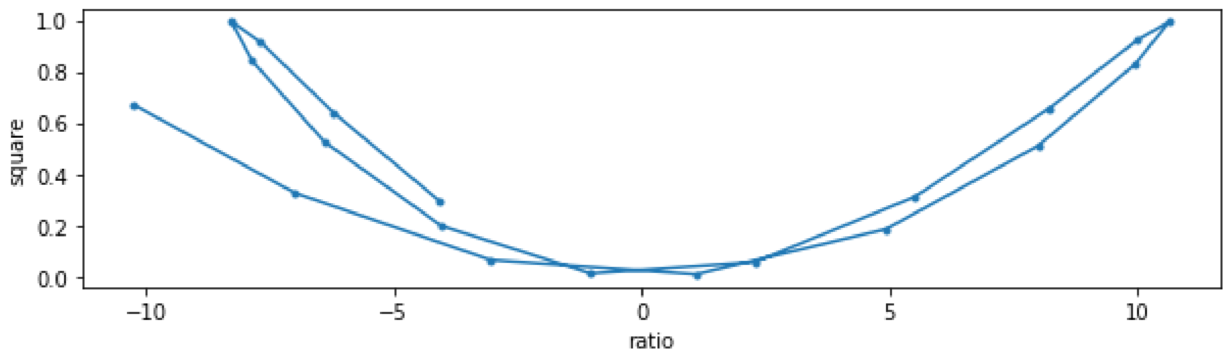
```
In [5]: swpInX.addMeasurement('measTwo', lambda: p.measure()[1])
        swpInX.addParser('xActuation^4', lambda d: d['xActuation']**4)
        swpInX.setPlotOptions(xKey=('xActuation', 'xActuation^4'))
        swpInX.gather()
        if not livePlots:
            swpInX.plot()
```



Adding new parsers after the data has been gathered

```
In [6]: swpInX.addParser('ratio', lambda d: d['measOne'] / d['measTwo'])
        swpInX.addParser('square', lambda d: d['measOne'] * d['measOne'])
        swpInX.setPlotOptions(xKey='ratio', yKey='square')
        swpInX.plot()
```

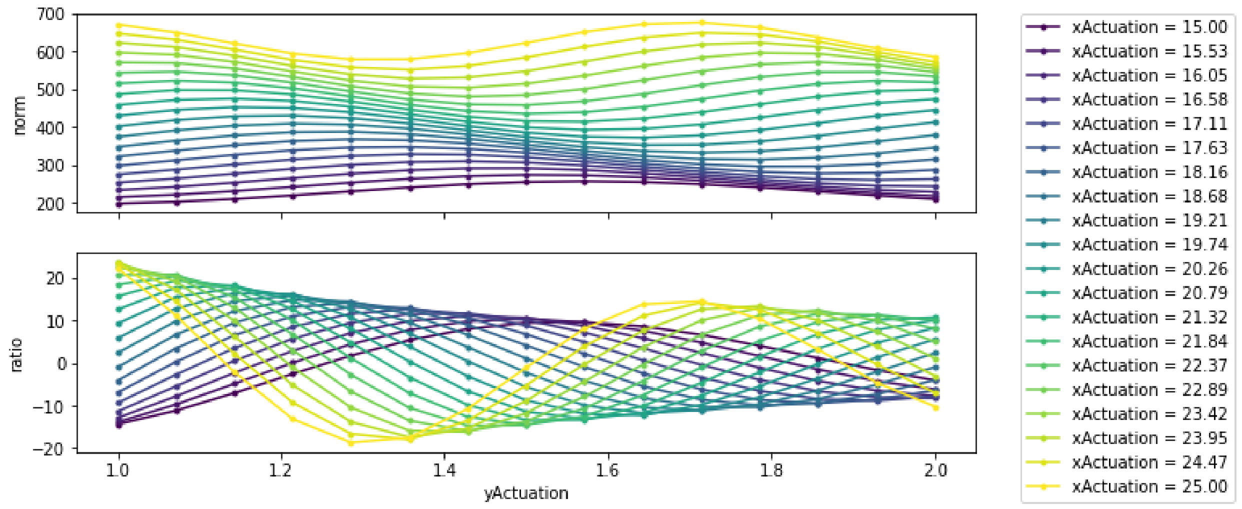
```
Out[6]: array([[<matplotlib.axes._subplots.AxesSubplot object at 0x1082c6278>]],
              dtype=object)
```



Subsuming to 2D and progress server

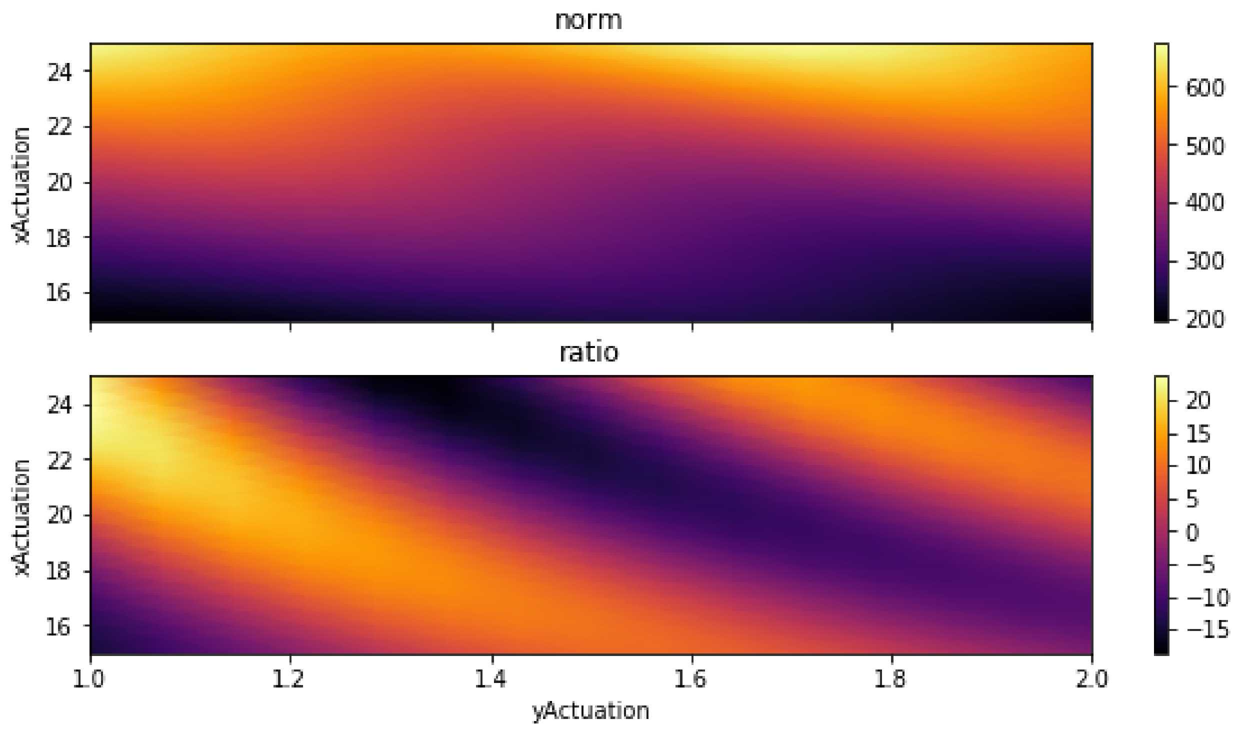
```
In [7]: # Turn to True if you have your monitor server running (see docs)
        runServer = False

        swpInY = NdSweeper()
        swpInY.addActuation('yActuation', p.actuateY, np.linspace(1, 2, 15))

        fullSwp = swpInX.subsume(swpInY)

        fullSwp.addParser('norm', lambda x: (x['measOne'] + x['xActuation'])**2)
        fullSwp.setMonitorOptions(runServer=runServer, livePlot=livePlots, plotEvery=10, stdoutPrint=
        fullSwp.setPlotOptions(xKey=('yActuation'), yKey=('norm', 'ratio'))
        fullSwp.gather()
```

```
In [8]: # Default 2D curve plot
        fullSwp.setPlotOptions(plType='curves')
        _ = fullSwp.plot()
```

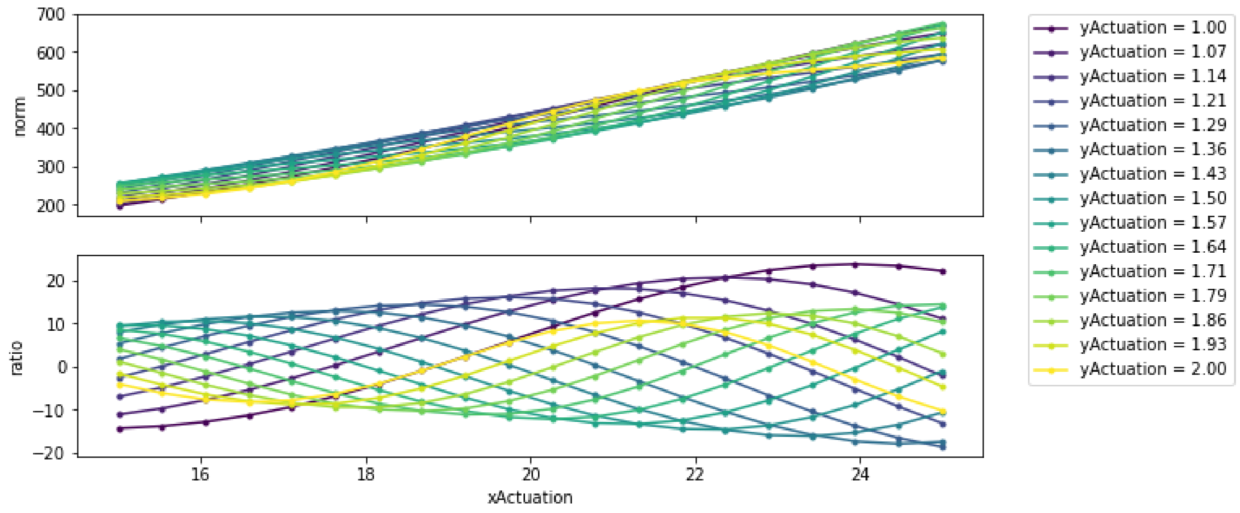


```
In [9]: # Surface plot
        fullSwp.setPlotOptions(plType='surf')
```

```
_ = fullSwp.plot()
```

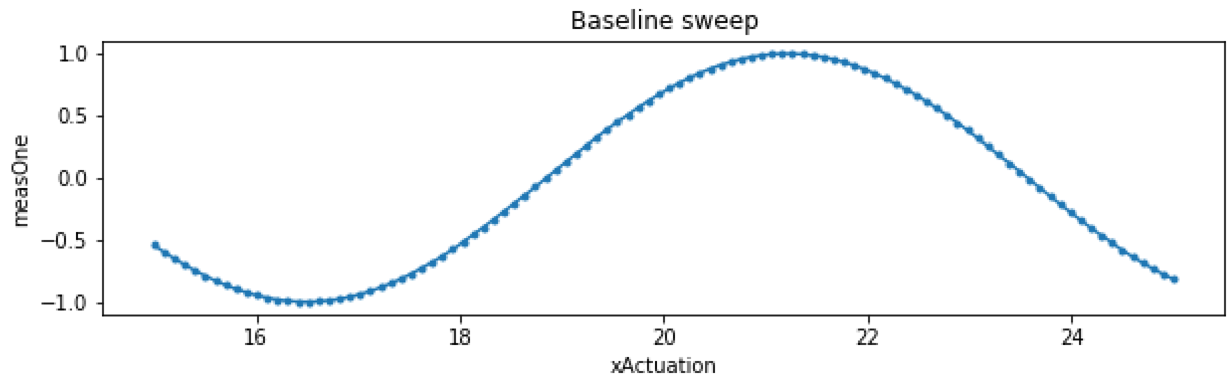


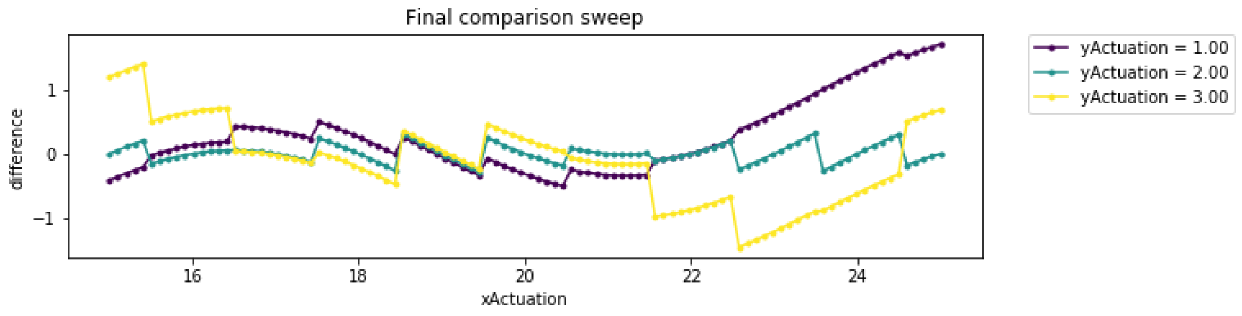
```
In [10]: # curves, but with domains reversed. This can reveal other things
         fullSwp.setPlotOptions(plType='curves', xKey='xActuation')
         _ = fullSwp.plot()
```



Using static data to compare subsequent sweeps

```
In [11]: p = Plant()

         # First do a 1d sweep
         swpA = NdSweeper()
         swpA.addMeasurement('measOne', lambda: p.measure()[0])
         swpA.addActuation('xActuation', lambda x: p.actuateX(x, rounded=False), np.linspace(15, 25,
         swpA.setMonitorOptions(stdoutPrint=False)
         swpA.gather()
         swpA.plot()
         plt.title('Baseline sweep')

         # Get its data
         baseline = swpA.data['measOne']

         # do a 2d sweep where each line is compared to the 1d line
         swpB = NdSweeper()
         swpB.addMeasurement('measOne', lambda: p.measure()[0])
         # The order of these calls matters
         swpB.addActuation('xActuation', lambda x: p.actuateX(x, rounded=True), np.linspace(15, 25, 1
         swpB.addStaticData('baseline', baseline)
         swpB.addActuation('yActuation', lambda y: p.actuateY(y), np.linspace(1, 3, 3))
         swpB.addParser('difference', lambda d: d['measOne'] - d['baseline'])
         swpB.setMonitorOptions(stdoutPrint=False)
         swpB.setPlotOptions(xKey='xActuation', yKey='difference')
         swpB.gather()
         swpB.plot()
         _ = plt.title('Final comparison sweep')
```



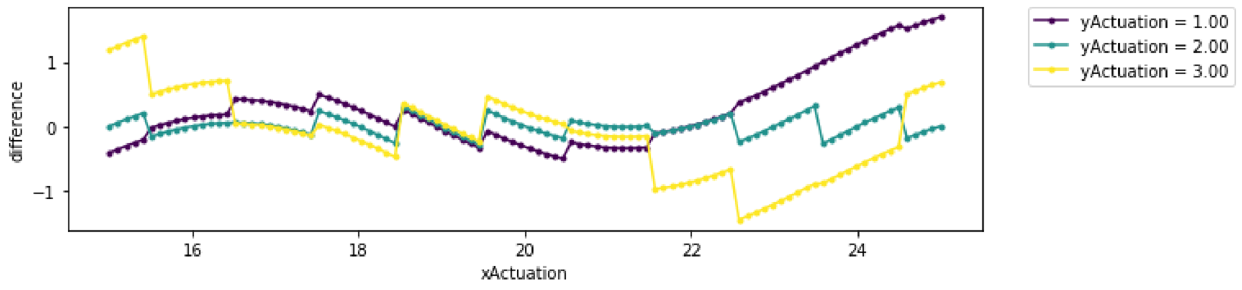


Saving and loading

```
In [12]: import lightlab.util.io as io
         io.fileDir = '.'
         fname = 'temp-ndsweep'
         swpB.saveObj(fname)

         swpC = NdSweeper.loadObj(fname)
```

```
        swpC.setPlotOptions(xKey='xActuation', yKey='difference')
        swpC.plot()

        import os
        os.remove(fname + '.pkl')
```

```
Saving to file: /Users/atait/Dropbox/Documents/gitProjects/experiment-code/lightlab/notebooks/Tests/t
```



```
In [ ]:
```

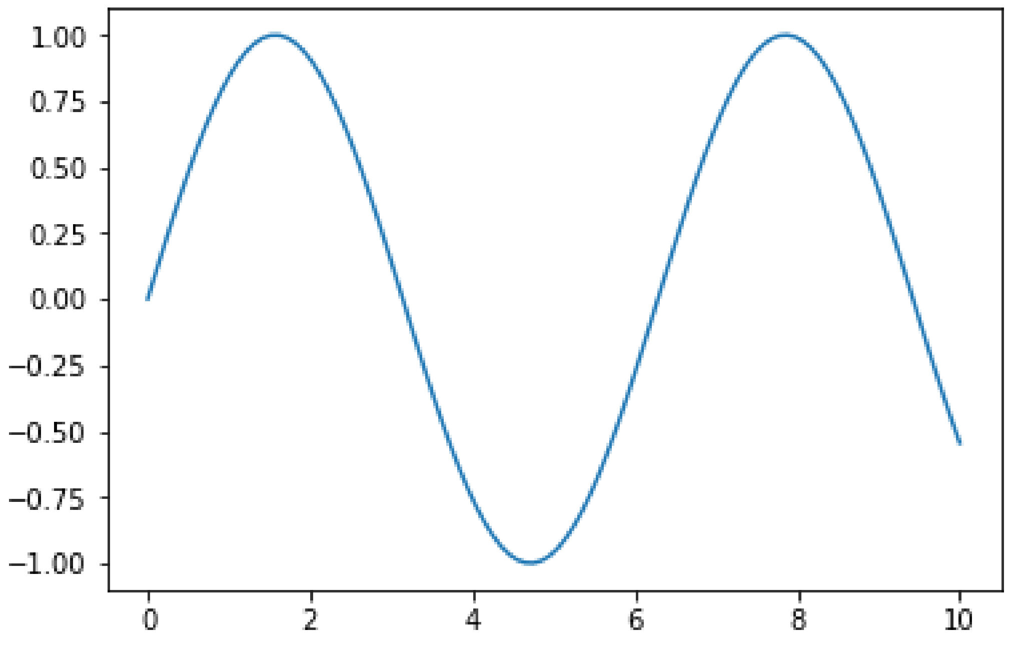
Simple sweep

```
In [1]: import matplotlib.pyplot as plt
        import numpy as np

        import lightlab.util.sweep as sUtil
```

```
In [2]: # Define the system used for this notebook
        class Plant():
            def __init__(self):
                self.x = 2

            def actuateX(self, newX, rounded=False):
                self.x = round(newX) if rounded else newX

            def measure(self):
                return np.sin(self.x)
```

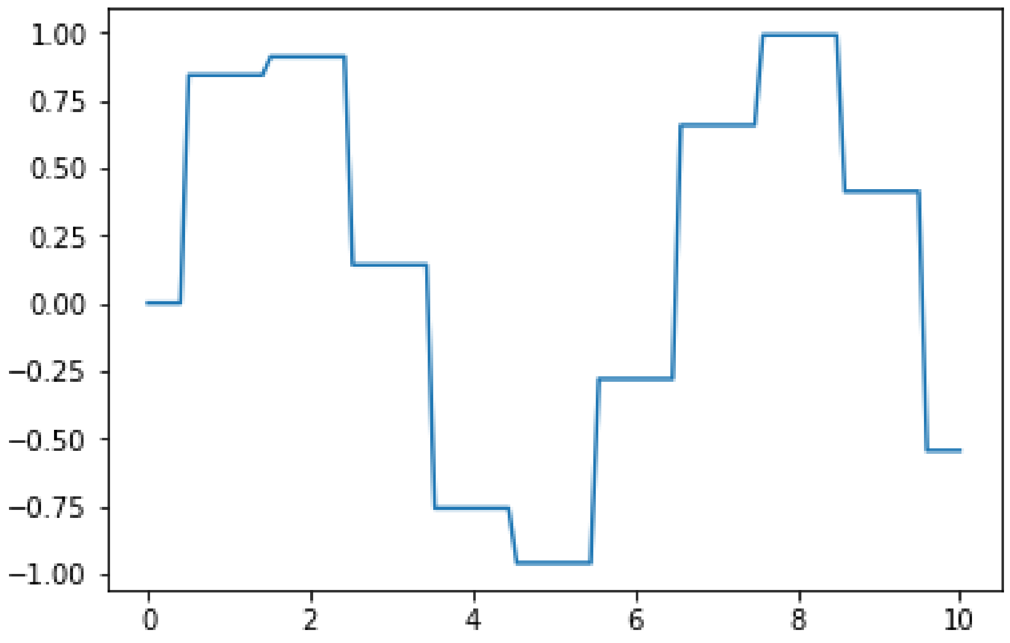
```
In [3]: p = Plant()
        x = np.linspace(0,10,100)
        y = sUtil.simpleSweep(p.actuateX, x, p.measure)
        plt.plot(x,y)
```

```
Out[3]: [<matplotlib.lines.Line2D at 0x10b2220b8>]
```



```
In [4]: # Now with a lambda function
        y = sUtil.simpleSweep(lambda v: p.actuateX(v, rounded=True), x, p.measure)
        plt.plot(x,y)
```

```
Out[4]: [<matplotlib.lines.Line2D at 0x10b2b9ac8>]
```



```
In [ ]:
```

Instrument: `Clock`

```
In [1]: %load_ext autoreload
        %autoreload 2
        from start import start

        clk = start('Agilent 83712B clock')
```

```
It is alive
HEWLETT-PACKARD,83712B,US37101574,REV  10.04
Here is what to test:
startup
enable
frequency
```

```
In [2]: origFreq = clk.frequency
        clk.frequency = 500e6
        print(clk.frequency / 1e6, 'MHz')
        clk.frequency = origFreq

        origEnable = clk.enable
        clk.enable = not origEnable
        print(clk.enable)
        clk.enable = origEnable

        # Context management. Downside right now is you need to go through to the config string
        with clk.driver.tempConfig('FREQ', 600e6):
            print(clk.frequency / 1e6, 'MHz')
        assert clk.frequency == origFreq
```

```
500.0 MHz
False
600.0 MHz
```

```
In [ ]:
```

Instrument: `CurrentSource`

```
In [1]: %load_ext autoreload
        %autoreload 2
        from start import start

        cs = start('Current Source (andromeda)')
```

```
It is alive
x = 2
[x+1, x+1.5] = [3.01, 3.51]
Current Source
Here is what to test:
startup
setChannelTuning
getChannelTuning
off
```

```
In [2]: cs.setChannelTuning({3: 1.1}, mode='milliamp')
        tDict = cs.getChannelTuning(mode='milliamp')
        assert tDict[3] == 1.1
        print(tDict)
        cs.off()
        tDict = cs.getChannelTuning(mode='milliamp')
        assert all(t == 0.0 for t in tDict.values())
```

```
{0: 0.0, 1: 0.0, 2: 0.0, 3: 1.1, 4: 0.0, 5: 0.0, 6: 0.0, 7: 0.0, 8: 0.0, 9: 0.0, 10: 0.0, 11: 0.0, 12
```

```
In [ ]:
```

Instrument: `FunctionGenerator`

```
In [1]: %load_ext autoreload
        %autoreload 2
        from start import start

        synth = start('Function Generator')
```

```
2018-04-17 03:12:10,885 - WARNING - lightlab.visa:
        Function generator GPIB is broken, so cannot ensure if live
```

```
It is alive
Function generator, HP 8116A
Here is what to test:
startup
frequency
waveform
amplAndOffs
amplitudeRange
duty
```

Problems here

1. Our particular synth GPIB is broken, so it cannot query
2. The below are currently not working with pretty obscure VISA errors

TODO

```
In [ ]: # synth.frequency(100)
        # synth.waveform('sine')
        # maxVolt = synth.amplitudeRange[1]
        # synth.amplAndOffs((maxVolt / 10, 0))
```

```
In [ ]:
```

Instrument: `Keithley` and `SourceMeter`

```
In [1]: %load_ext autoreload
        %autoreload 2
        from start import start

        keithley = start('Keithley 25')
```

```
It is alive
KEITHLEY INSTRUMENTS INC.,MODEL 2400,4087737,C32   Oct  4 2010 14:20:11/A02  /U/K
Here is what to test:
startup
setCurrent
getCurrent
measVoltage
setProtectionVoltage
```

```
protectionVoltage
setProtectionCurrent
protectionCurrent
enable
setPort
setCurrentMode
setVoltageMode
setVoltage
getVoltage
measCurrent
```

```
In [2]: keithley.setCurrentMode()
        with keithley.warmedUp():
            keithley.setCurrent(.1e-4)
            print(keithley.measVoltage())

        keithley.setVoltageMode()
        with keithley.warmedUp():
            keithley.setVoltage(1e-3)
            print(keithley.measCurrent())
```

```
2018-04-17 01:30:26,674 - WARNING - lightlab:
        Keithley compliance voltage of 1 reached
2018-04-17 01:30:26,676 - WARNING - lightlab:
        You are sourcing 1.001966e-08mW into the load.
```

```
1.001966
-2.573609e-11
```

```
In [ ]:
```

Instrument: `LaserSource`

```
In [1]: %load_ext autoreload
        %autoreload 2
        from start import start

        dfbs = start('Laser Array 01')
        # dfbs = start('Laser Array 11')
        # dfbs = start('Laser Array 12')
```

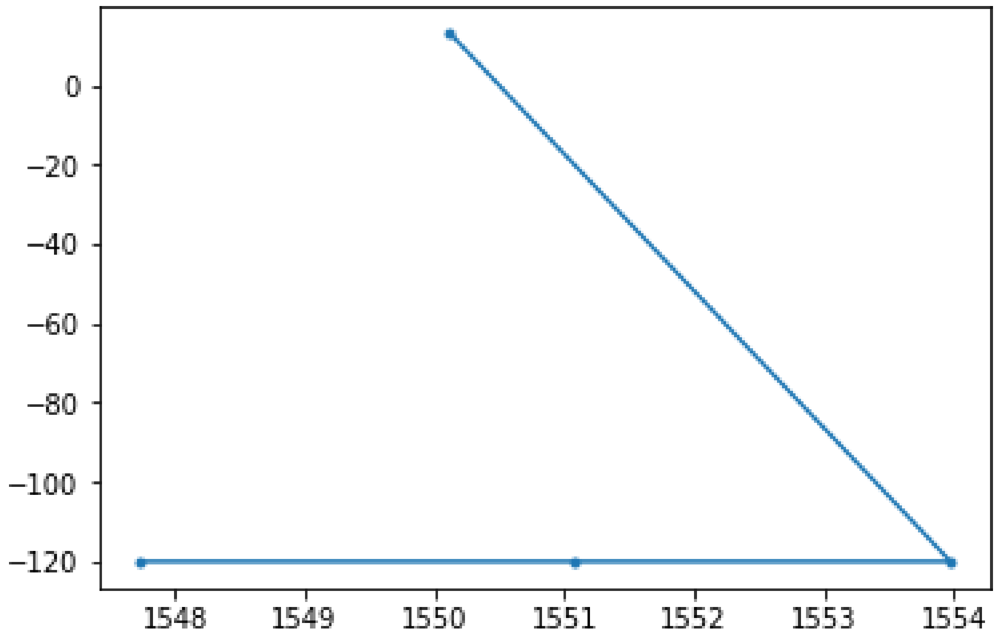
```
It is alive
ILX Lightwave,7900 System,79006021,3.42
Here is what to test:
startup
setChannelEnable
getChannelEnable
setChannelWls
getChannelWls
setChannelPowers
getChannelPowers
getAsSpectrum
off
allOn
enableState
wls
powers
wlRanges
allOff
```

```
In [2]: print('Blocked out channels are', dfbs.driver.useChans)
        ena = dfbs.getChannelEnable()
        dfbs.off()
        dfbs.setChannelEnable({0: 1})
        print(dfbs.wlRanges)
```

```
Blocked out channels are range(0, 4)
DFB settling for 3 seconds.
done.
DFB settling for 3 seconds.
done.
((1549.27, 1550.97), (1552.48, 1554.18), (1550.07, 1551.77), (1546.87, 1548.57))
```

```
In [3]: dfbs.getAsSpectrum().simplePlot('.-')
```

```
Out[3]: [<matplotlib.lines.Line2D at 0x7feba2fe00b8>]
```



```
In [4]: dfbs.off()
```

```
DFB settling for 3 seconds.
done.
```

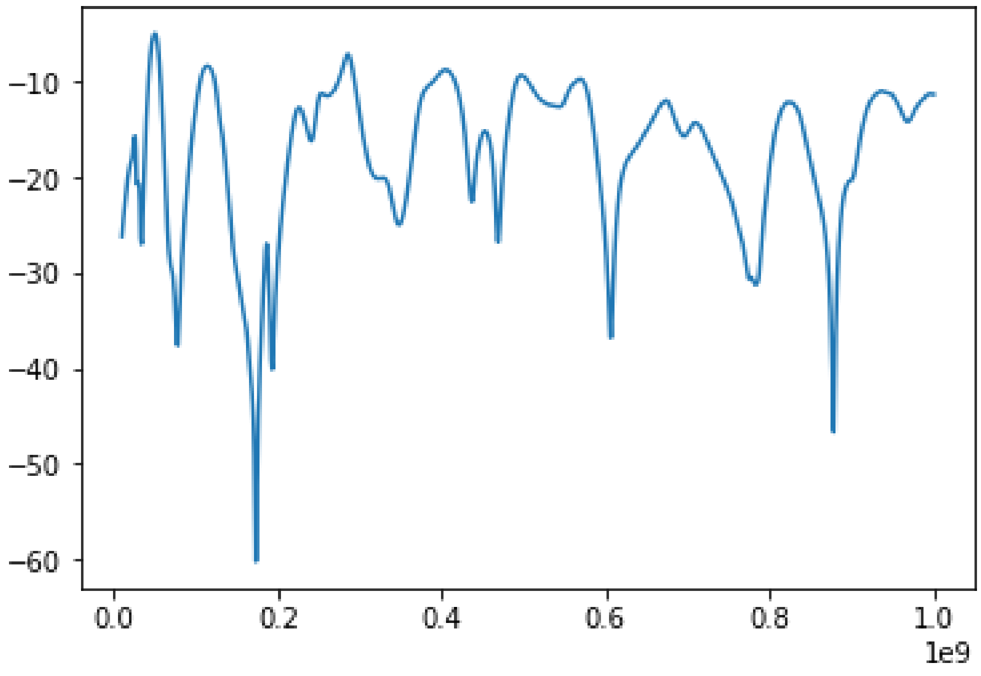
```
In [1]: %load_ext autoreload
        %autoreload 2
        from start import start

        cs = start('Current Source (andromeda)')
```

```
It is alive
x = 2
[x+1, x+1.5] = [3.01, 3.51]
Current Source
Here is what to test:
startup
setChannelTuning
getChannelTuning
off
```

```
In [2]: cs.setChannelTuning({3: 1.1}, mode='milliamp')
        tDict = cs.getChannelTuning(mode='milliamp')
        assert tDict[3] == 1.1
        print(tDict)
        cs.off()
        tDict = cs.getChannelTuning(mode='milliamp')
        assert all(t == 0.0 for t in tDict.values())

{0: 0.0, 1: 0.0, 2: 0.0, 3: 1.1, 4: 0.0, 5: 0.0, 6: 0.0, 7: 0.0, 8: 0.0, 9: 0.0, 10: 0.0, 11: 0.0, 12

In [ ]:
```

Instrument: `NetworkAnalyzer`

```
In [ ]: %load_ext autoreload
        %autoreload 2
        from start import start

        pna = start('PNA5222A')

In [3]: # Setup a S21 measurement
        pna.measurementSetup(measType='S21')
        pna.sweepSetup(startFreq=10e6, stopFreq=1e9, nPts=1000)
        pna.sweepEnable(True)
        spct = pna.spectrum()
        spct.simplePlot()
        pna.run() # put it back to live
```



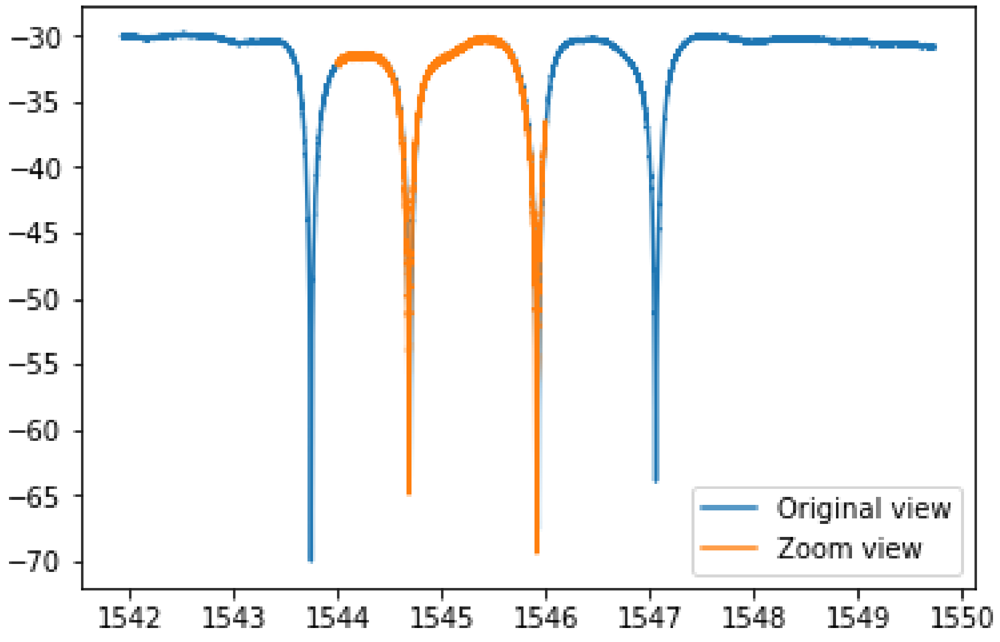
```
In [ ]:
```

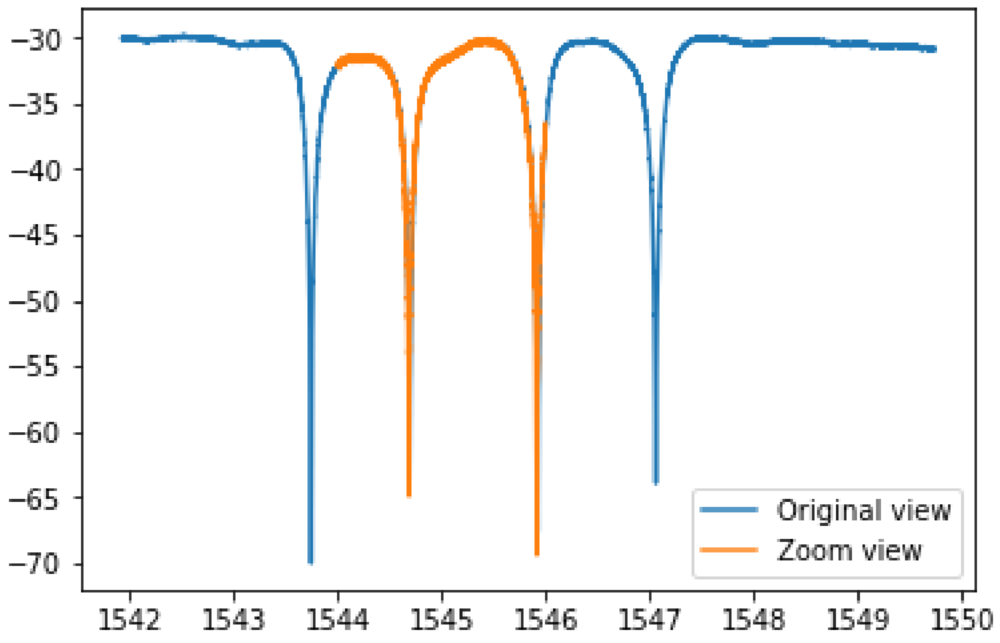
Instrument: `OpticalSpectrumAnalyzer`

```
In [1]: %load_ext autoreload
        %autoreload 2
        from start import start

        osa = start('Apex Optical Spectrum Analyzer')
```

```
It is alive
Apex AP2440A
Here is what to test:
startup
spectrum
wlRange
```

```
In [4]: osa.spectrum(avgCnt=3).simplePlot(livePlot=True, label='Original view')
        oldWlRange = osa.wlRange
        osa.wlRange = [1544, 1546]
        osa.spectrum().simplePlot(livePlot=True, label='Zoom view')
        osa.wlRange = oldWlRange
```





```
In [ ]:
```

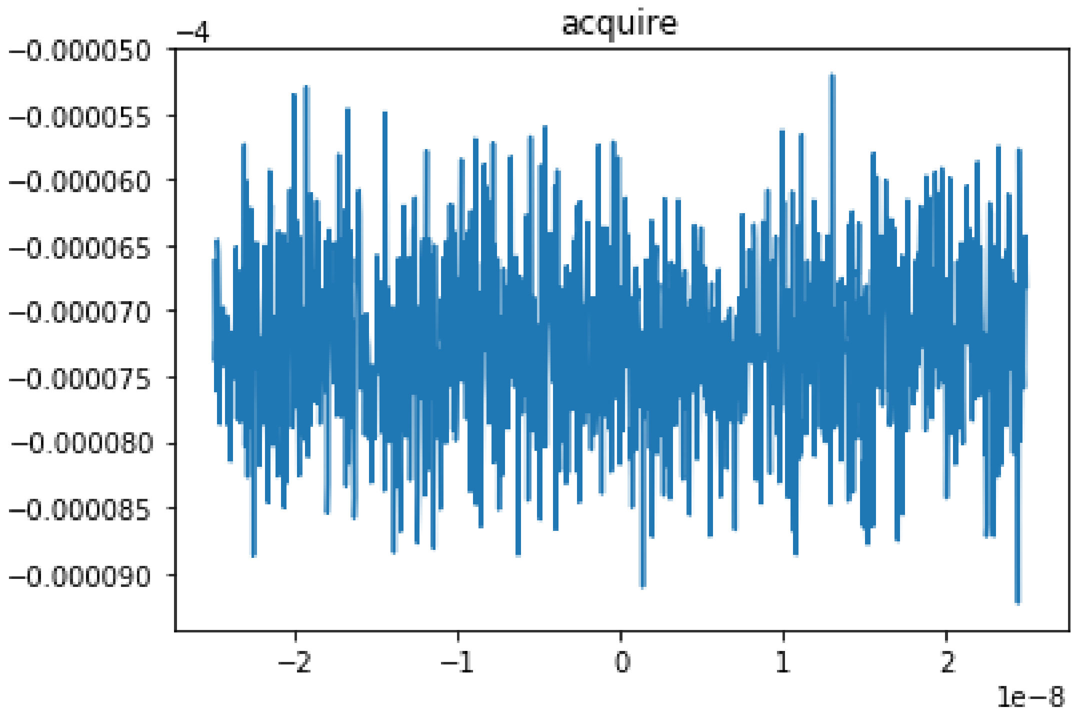
Instrument: `Oscilloscope`

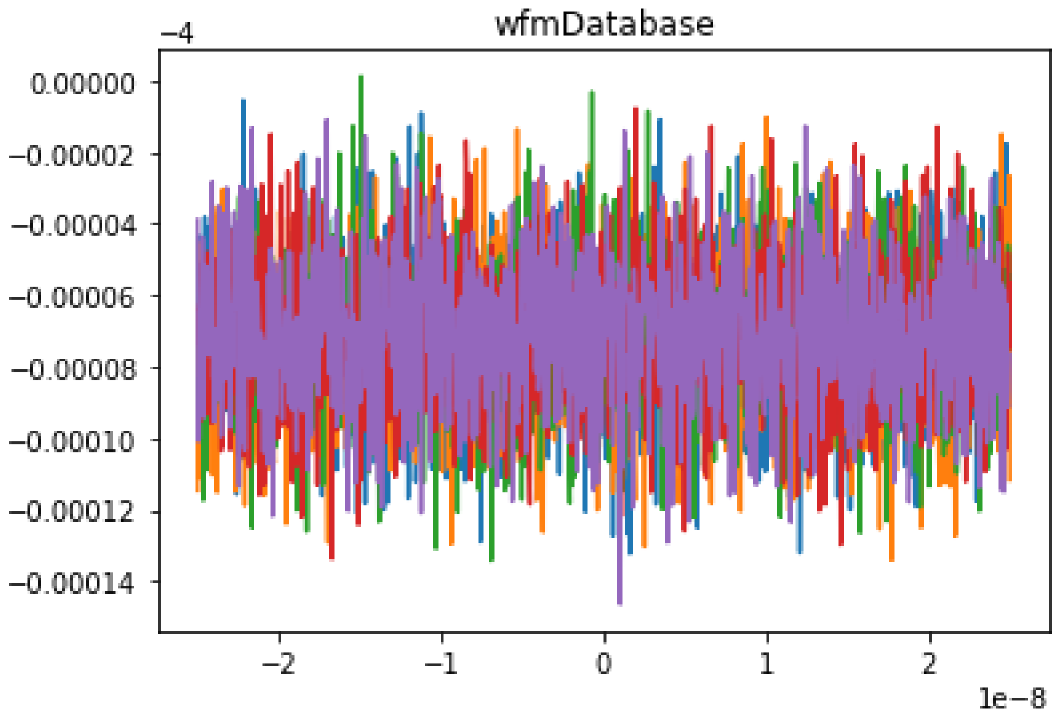
```
In [1]: %load_ext autoreload
        %autoreload 2
        import matplotlib.pyplot as plt
        from start import start

        scope = start('Sampling Scope DSA8300')
        # scope = start('Slow Scope DPO4032')
        # scope = start('Real Time Scope TDS6154C')
```

```
It is alive
TEKTRONIX,DSA8300,C040232,CF:91.1CT FV:6.3.1.3
Here is what to test:
startup
acquire
wfmDb
run
histogramStats
```

```
In [2]: chan = 1
        scope.acquire([chan], avgCnt=10, duration=None, position=None, nPts=None)[0].simplePlot()
        plt.title('acquire')

        if 'DSA' in scope.name:
            stddev, pdf = scope.histogramStats(chan, nWfms=3, untriggered=False)
            print(stddev, 'and', pdf)

        if 'DPO' not in scope.name:
            plt.figure()
            bund = scope.wfmDb(chan, nWfms=5)
            bund.simplePlot()
            plt.title('wfmDatabase')
            scope.run()
```

```
22.11896509818E-6 and [72.4     97.1125 99.9375]
```





Some error printing tests

```
In [3]: # The error should suggest going down to the driver
        try:
            scope.getConfigParam
        except AttributeError as err:
            print(err.args[0])
```

```
Sampling Scope DSA8300 has no attribute getConfigParam
It looks like you are trying to access a low-level attribute
Use ".driver.getConfigParam" to get it
```

```
In [4]: try:
            scope.histogramStats
        except AttributeError as err:
            print(err.args[0])
        else:
            print('This scope implements histogramStats')
```

```
This scope implements histogramStats
```

```
In [ ]:
```

Instrument: `PowerMeter`

```
In [1]: %load_ext autoreload
        %autoreload 2
        from start import start

        # pm = start('Advantest Q8221')
        pm = start('Power Meter HP')
```

```
It is alive
HEWLETT PACKARD,8152A,0,REV 2.0
Here is what to test:
startup
powerDbm
powerLin
```

```
In [2]: print('In dBm ', pm.powerDbm())
        print('In lin ', pm.powerLin())
```

```
In dBm  -17.05
In lin  0.019678862897068445
```

```
In [ ]:
```

Instrument: `PulsePatternGenerator`

```
In [1]: %load_ext autoreload
        %autoreload 2
        import time
        from start import start

        ppg = start('Anritsu MP1763B Pulse Pattern Generator')
```

```
It is alive
ANRITSU,MP1761A,0,0001
Here is what to test:
```

```
startup
setPrbs
setPattern
getPattern
on
syncSource
amplAndOffs
```

```
In [2]: ppg.on(True)

        origSrc = ppg.syncSource()
        print('Sync source is', origSrc)
        ppg.syncSource('clock64')
        ppg.syncSource(origSrc)

        ppg.amplAndOffs()
```

```
Sync source is fixed
```

```
Out[2]: (0.4, 0.0)
```

```
In [3]: # Mess with the pattern here, watch the lights in lab alternate
        ppg.setPrbs(16)
        print('Pattern was', ppg.getPattern())
        for _ in range(4):
            print('flipping')
            ppg.setPattern(1 - ppg.getPattern())
            time.sleep(1)
```

```
Pattern was [0 0 1 0 0 1 0 1 0 1 0 1 0 1 1 1]
flipping
flipping
flipping
flipping
```

```
In [ ]:
```

Instrument: `VariableOpticalAttenuator`

```
In [1]: %load_ext autoreload
        %autoreload 2
        from start import start

        voa = start('HP 8156A Optical Attenuator (corinna)')
```

```
It is alive
HEWLETT-PACKARD,HP8156A,3328G01226,1.02
Here is what to test:
startup
on
off
attenDB
attenLin
```

```
In [2]: voa.attenDB = 0
        voa.attenDB = 3
        voa.attenLin = 0

        print(voa.attenDB)  # this should give you the maximum 60
        voa.off()
```

```
60.0
```

Instrument configuration

```
In [1]: from lightlab.laboratory.state import lab
        from lightlab.equipment.lab_instruments import *

        host = lab.hosts['gunther']
        bench = lab.benches['bert']

        # Do not edit this
        print('Host available addresses:')
        for resource in host.list_gpib_resources_info():
            print(resource)
```

```
Host available addresses:
visa://labdns-gunther.school.edu/GPIB0::16::INSTR
visa://labdns-gunther.school.edu/GPIB0::18::INSTR
visa://labdns-gunther.school.edu/GPIB0::21::INSTR
```

```
In [2]: # Uncomment only one at a time
        info = Keithley_2400_SM, 'Keithley 21', host.gpib_port_to_address(21), {}
        # info = ILX_7900B_LS, 'Laser Array 12', host.gpib_port_to_address(12), dict(useChans=range(8

        # Do not edit this
        theDriver, theName, theAddress, extraKwargs = info
        newInst = theDriver(name=theName,
                            address=theAddress,
                            bench=bench,
                            host=host,
                            **extraKwargs)

        try:
            oldInst = lab.instruments_dict[newInst.name]
        except KeyError:
            print('This is a new instrument')
        else:
            print('You are overwriting! Make sure everything is specified (i.e. ports, useChans, etc
            oldInst.display()
            print('\n*** TO ***\n')
            newInst.display()
```

```
You are overwriting! Make sure everything is specified (i.e. ports, useChans, etc.)

Keithley 21
Bench: Bench bert
Host: Host brian
address: GPIB0::21::INSTR
driver_class: Keithley_2400_SM
=====
Ports
=====
   No ports.
***

*** TO ***
```

```
Keithley 21
Bench: Bench bert
Host: Host gunther
address: visa://labdns-gunther.school.edu/GPIB0::21::INSTR
driver_class: Keithley_2400_SM
=====
Ports
=====
   No ports.
***
```

```
In [3]: # Make the change and save. Be careful!
        lab.deleteInstrumentFromName(newInst.name)  # deleting previous instance, if it is there
        lab.insertInstrument(newInst)  # inserting new instance
        lab.saveState()
```

```
In [4]: # Test it
        gotten = lab.instruments_dict[newInst.name]
        gotten.isLive()
```

```
Out[4]: True
```

- genindex
- modindex
- search

2.3.4 Making and changing the lab state

One time: Hosts and benches

First you need to add some hosts and benches to the lab. This usually happens only once. Suppose we have a computer called “brian” that is the localhost actually running the notebooks (note: it can be also viewed as a server). It is physically located on Bert’s bench:

```
from lightlab.laboratory.state import lab
from lightlab.laboratory.instruments import LocalHost, Host, Bench

# Start by making a host. This is a real computer.
brianHost = LocalHost(name='brian')  # name is optional
assert brianHost.isLive()  # Sends a ping request
lab.updateHost(brianHost)
lab.saveState()
```

Next, let’s add a remote host called “gunther”. It connects to some instruments and is running VISA server that will be contacted by the central server (brian):

```
guntherHost = Host(name='gunther',into labstate
                   hostname='labdns-gunther.school.edu',
                   mac_address='00:00:00:00:00:01',  # optional
                   os='windows')  # optional
assert guntherHost.isLive()  # will send a ping
lab.updateHost(guntherHost)
lab.saveState()
```

Next, a bench. Benches are not strictly necessary but useful by convention:

```
bertBench = Bench(name='bert')
lab.updateBench(bertBench)
lab.saveState()
```

Note: For `isLive` to work, the host must be configured to respond to pings.

Instruments

Instruments can be configured many times, for example, if they move. An example of setting one of them is below. You should copy this ipynb into your operating (`myWork`) directory as a template to run with jupyter.

Now you get that instrument from any other notebook with the command:

```
from lightlab.laboratory.state import lab
keithley = lab.instruments_dict['Keithley 21']
```

- genindex
- modindex
- search

2.4 Tutorials

2.4.1 Measured functions

In this section

`MeasuredFunction` is the datatype workhorse. Most data can be formulated as one variable vs. another, the ordinate and abscissa. What we measure is discrete, but we can assume it represents something continuous. That means interpolation and math are supported with appropriate processing of abscissa basis.

Basic manipulation is supported, such as splicing, deleting segments, adding points, etc. Math is also supported with a scalar and a measured function and two measured functions (with appropriate abscissa basis handling.)

Child classes include `Spectrum`, meant for cases where the abscissa is frequency or wavelength and the ordinate is power or transmission. It has extra methods for conversion from linear to decibel power units. Also `Waveform` is meant for cases where abscissa is time.

Peak finding

The data module is particularly good with peaks. A very basic classless peak finder comes with `findPeaks()`. The arguments are arrays and indeces. It is more useful to do peakfinding in an object-oriented way with `findResonanceFeatures()`. The `ResonanceFeature` class stores information on the position, width, and height of peaks, in addition to more powerful aspects like refining position based on convolution with a known peak shape.

Much of this functionality is handled within the `SpectrumMeasurementAssistant`, good for when you are looking at real spectra of a single device over and over again. Makes assumptions such as background not changing and filter shape not changing. The notebook doesn't really show the full potential of SpectrumMeasurementAssistant.

Descent-based function inversion

Inverting a measured function is desirable for evoking a particular response that was measured. For example, finding the proper wavelength shift needed to set a given transmission value, based on a known MeasuredFunction of transmission vs. wavelength. Descent functions use linear interpolation. Descent only works on monotonically increasing (decreasing) sections. When the entire object is monotonic, use the `MeasuredFunction.invert` method. When the function is peak-like, it is possible to specify a direction to start the descent until either the target value is reached, or the function changes slope.

FunctionBundle and FunctionalBasis

Often there are two abscissas. The "third dimension" could be a continuous variable (as in *MeasuredSurface*) or a discrete variable (as in *FunctionBundle*). They each have different implications and operations and subclasses. *Spectrogram* inherits *MeasuredSurface* with continuous time as the second abscissa. *FunctionalBasis* is basically a bundle with increased attention paid to linear algebra and function order for the sake of decomposing, synthesizing, and projecting weighted additions of other functions.

`FunctionBundle`(*measFunList=None*)

A bundle of *`MeasuredFunction`*'s: "z" vs. "x", "i"

The key is that they have the same abscissa base. This class will take care of resampling in a common abscissa base.

The bundle can be:

- iterated to get the individual :class'~lightlab.util.data.one_dim.MeasuredFunction''s
- operated on with other `FunctionBundles`
- plotted with :meth'simplePlot' and `multiAxisPlot()`

Feeds through **callable** signal processing methods to its members (type MeasuredFunction), If the method is not found in the FunctionBundle, and it is in it's member, it will be mapped to every function in the bundle, returning a new bundle.

Distinct from a `MeasuredSurface` because the additional axis does not represent a continuous thing. It is discrete and sometimes unordered.

Distinct from a `FunctionalBasis` because it does not support most linear algebra-like stuff (e.g. decomposision, matrix multiplication, etc.). This is not a strict rule.

- genindex
- modindex
- search

2.4.2 Using and creating drivers for instruments

In this section

Drivers are the original impetus for sharing this project. Writing drivers can be fun (the first few times). It exercises the full range of electrical engineering knowledge. It can be a snap, or it can take multiple PhD students several days to realize which cable needed a jiggle. The reward is automated, remote lab control!

The module page `lab_instruments` contains all the instruments necessary available in lightlab. If your equipment is available (e.g. a very common `Keithley_2400_SM`), then you can use it directly with:

```
from lightlab.equipment.lab_instruments import Keithley_2400_SM
k = Keithley_2400_SM(name="My Keithley", address="GPIB0::23::INSTR")
if k.isLive():
    print("Connection is good")

help(k)  # should display all commands available to be used.
```

The address format for the Instrument is either a VISA-compatible resource name (parsed by *pyvisa*). In this example, the Keithley instrument is configured to have the address 23, and it is plugged directly to the host. Alternatively, it can be connected to a computer with an instance of the NI Visa Server, in which case the address would be `visa://alice.school.edu/GPIB0::23::INSTR`, where `alice.school.edu` is the hostname of the computer hosting the Visa Server.

Alternatively, it can be written as `prologix://prologix_ip_address/gpib_primary_address[:gpib_secondary_address]`, e.g. `prologix://alice.school.edu/23`, for use with the Prologix GPIB-Ethernet controller.

The instrument abstraction

In `lightlab`, there are two layers of abstraction for instrumentation

1. **`Instrument`, such as**
   - `Oscilloscope`
   - `Keithley`
2. **`VISAInstrumentDriver`, such as**
   - *`Tektronix_DPO4032_Oscope`*
   - *`Tektronix_DPO4034_Oscope`*
   - *`Keithley_2400_SM`*

An `Instrument` refers to a category of instruments that do certain things. A `VISAInstrumentDriver` describes how a particular piece of equipment does it. As a rule of thumb, there is a different driver for each model of instrument.

All oscilloscopes have some form of acquiring a waveform, and user code makes use of that abstraction. If you have a scope other than a TEKTRONIX DPO4032, you are on your own with the driver. BUT, if you can make your low-level driver for that scope to meet the abstraction of `Oscilloscope`, then your scope will be equivalent to my scope, in some sense. That means all of the rest of the package becomes usable with that scope.

The critical part of an Instrument child class are its `essentialMethods` and `essentialProperties`. Initialization and book keeping are all done by the super class, and implementation is done by the driver. The driver must implement all of the essential methods and properties, and then the `Instrument` will take on these data members as its own.

As in the case of Tektronix_DPO4032_Oscope and Tektronix_DPO4034_Oscope, there is substantial overlap in implementation. We can save a lot of work by abstracting some of the common behavior, which leads to the third major concept of abstract drivers, found in the module:

3. ***`abstract_drivers`*, which includes**
    - `DPO_Oscope`
    - `MultiModalSource`

Before writing a fresh driver, check out the abstract ones to see if you can partially use existing functionality (e.g. if you are making one for a DPO4038).

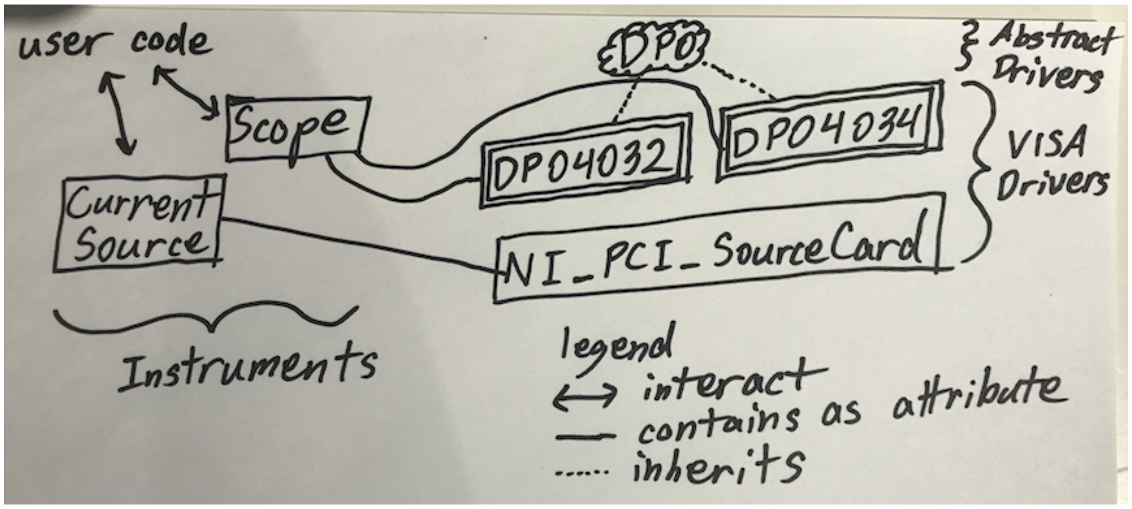


Fig. 3: Three concepts for `lightlab` instrumentation. 1) Instruments, 2) VISAInstrumentDrivers, 3) Abstract drivers.

Writing a `VISAInstrumentDriver`

For new developers, you will likely have instruments not yet contained in `lightlab`. We encourage you to write them, test them, and then create a pull request so that others won't have to re-invent the wheel.

Basics

A communication session with a message-based resource has the following commands

- open
- close
- write

- read
- query (a combination of write, then read)

The PyVISA package provides the low level communication. Drivers can be GPIB, USB, serial, or TCP/IP – the main difference is in the address. PyVISA also has a resource manager for initially finding the instrument. `lightlab` has a wrapper for this that works with multiple remote Hosts. See *Making and changing the lab state* for putting a Host in the labstate.

Plug your new instrument (let's say GPIB, address 23) into host "alice", then, in an ipython session

```
> from lightlab.laboratory.state import lab
> for resource in lab.hosts['alice'].list_resources_info():
...   print(resource)
visa://alice.school.edu/USB0::0x0699::0x0401::B010238::INSTR
visa://alice.school.edu/TCPIP0::128.112.48.124::inst0::INSTR
visa://alice.school.edu/ASRL1::INSTR
visa://alice.school.edu/ASRL3::INSTR
visa://alice.school.edu/ASRL10::INSTR
visa://alice.school.edu/GPIB0::18::INSTR
visa://alice.school.edu/GPIB0::23::INSTR
```

That means the instrument is visible, and we know the full address:

```
> from lightlab.equipment.lab_instruments.visa_connection import VISAObject
> newInst = VISAObject('visa://alice.school.edu/GPIB0::23::INSTR')
> print(newInst.instrID())
KEITHLEY INSTRUMENTS INC.,MODEL 2400, ...
```

That means the instrument is responsive, and basic communication settings are correct already. Time to start writing.

Troubleshooting 1: Write termination

Try this:

```
> newInst.open()
> newInst.mbSession.write_termination = ''
> newInst.mbSession.clear()
> print(newInst.instrID())
```

and play around with different line terminations. There are also different options for handshaking to be aware of, as well as baud rate attributes. For debugging at this level, we recommend the NI visaic.

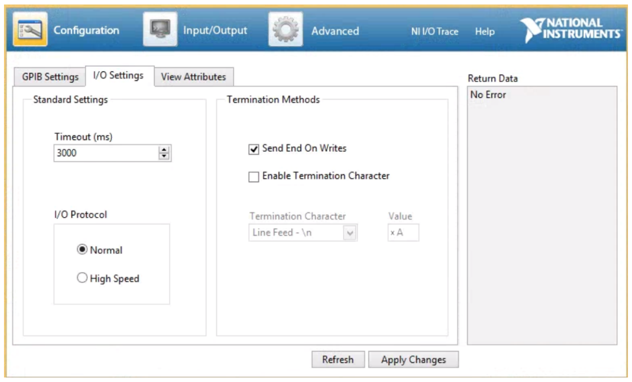
When you find something that works, overload the `open` method. Do not try to set these things in the `__init__` method.

Troubleshooting 2: No "*IDN?" behavior

Some instruments don't even though it is a nearly universal requirement. In that case, find some simple command in the manual to serve as your "is this instrument alive?" command. Later, overload the `instrID` method.

Configurable

Many instruments have complex settings and configurations. These are usually accessed in a message-based way with `write(':A:PARAM 10')` and `query(':A:PARAM?')`. We want to create a consistency between driver and hardware, but



Standard Settings

Timeout (ms): 3000

I/O Protocol: ● Normal ○ High Speed

Termination Methods

☑ Send End On Writes

☐ Enable Termination Character

Termination Character: Line Feed - \n Value: xA

Return Data: No Error

Fig. 4: NI Visa Interactive Control window. Change around line settings, then write "*IDN?" in the Input/Output. See attributes for more advanced settings.

1. we don't care about the entire configuration all the time, and
2. it doesn't make sense to send configuration commands all the time.

`Configurable` builds up a minimal notion of consistent state and updates hardware only when it might have become inconsistent. The above is done with `setConfigParam('A:PARAM', 10)` and `getConfigParam('A:PARAM')`. If you set the parameter and then get it, the driver will not communicate with the instrument – it will look up the value you just set. Similarly, it will avoid setting the same value twice. For example,:

```
# Very slow
def acquire(self, chan):
    self.write(':CH ' + str(chan))
    return self.query(':GIVE:DATA?')

# Error-prone
def changeChannel(self, chan):
    self.write(':CH ' + str(chan))

def acquire(self):
    return self.query(':GIVE:DATA?')

# Good (using Configurable)
def acquire(self, chan):
    self.setConfigParam('CH', chan)
    return self.query(':GIVE:DATA?')
```

Both support a `forceHardware` kwarg and have various options for message formatting.

`Configurable` also has support for saving, loading, and replaying configurations, so you can put the instrument in the exact same state as it was for a given experiment. Save files are human-readable in JSON.

Difference between `__init__`, `startup`, and `open`

`__init__` should set object attributes based on the arguments. The `super().__init__` will take care of lab book keeping. It should not call `open`.

`open` initiates a message based session. It gets called automatically when `write` or `query` are called.

`startup` (optional) is called immediately after the first time the instrument is opened.

How to read a programmer manual

You need the manual to find the right commands. You are looking for a command reference, or sometimes coding examples. They are often very long and describe everything from scratch. They sometimes refer to programming with vendor-supplied GUI software – don't want that. Here is a very old school manual for a power meter. It is 113 pages, and you need to find three commands. Go to the contents and look for something like "command summary."

which turns into the following driver (complete, simplified). If possible, link the manual in the docstring.

```
class HP8152(VISAInstrumentDriver):
    ''' The HP 8152 power meter

        `Manual <http://www.lightwavestore.com/product_datasheet/OTI-OPM-L-030C_pdf4.
↪pdf>`_
    '''
    def startup(self):
        self.write('T1')

    def powerDbm(self, channel=1):
        '''
            Args:
                channel (int): 1 (A), 2 (B), or 3 (A/B)
        '''
        self.write('CH' + str(channel))
        returnString = self.query('TRG')
        return float(returnString)
```

Newer equipment usually has thousand-page manuals, but they're hyperlinked.

- genindex
- modindex
- search

2.4.3 Doing sweeps

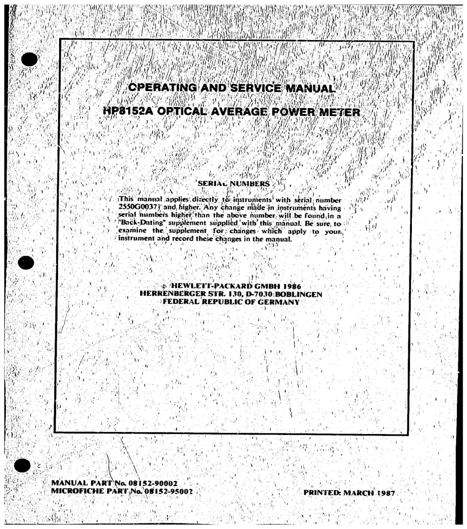
In this section

Note: this section is probably more appropriately named actuate/measure setups. This includes sweeps but it also includes command-control (more than just sweeps), as well as peak search and binary search.

Todo: relabel accordingly

Sweeps are incredibly common in experiments because they are about repeated measurements of one thing as it changes in relation to other things.

Sweeps are like loops, but with some special properties. That's why the package provides a generalized sweeper class for taking care of a lot of the common issues.



OPERATING AND SERVICE MANUAL

HP8152A OPTICAL AVERAGE POWER METER

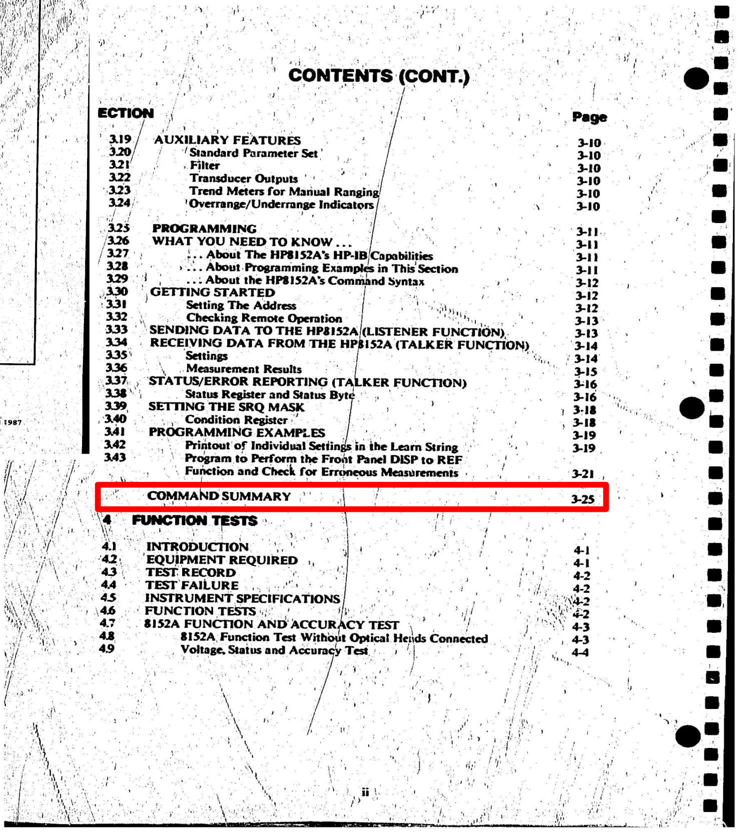
SERIAL NUMBERS

This manual applies directly to instruments with serial number 2550G00371 and higher. Any change made in instruments having serial numbers higher than the above number will be found in a "Back-Dating" supplement supplied with this manual. Be sure to examine the supplement for changes which apply to your instrument and record these changes in the manual.

© HEWLETT-PACKARD GMBH 1986
HERRENBERGER STR. 130, D-7030 BOBLINGEN
FEDERAL REPUBLIC OF GERMANY

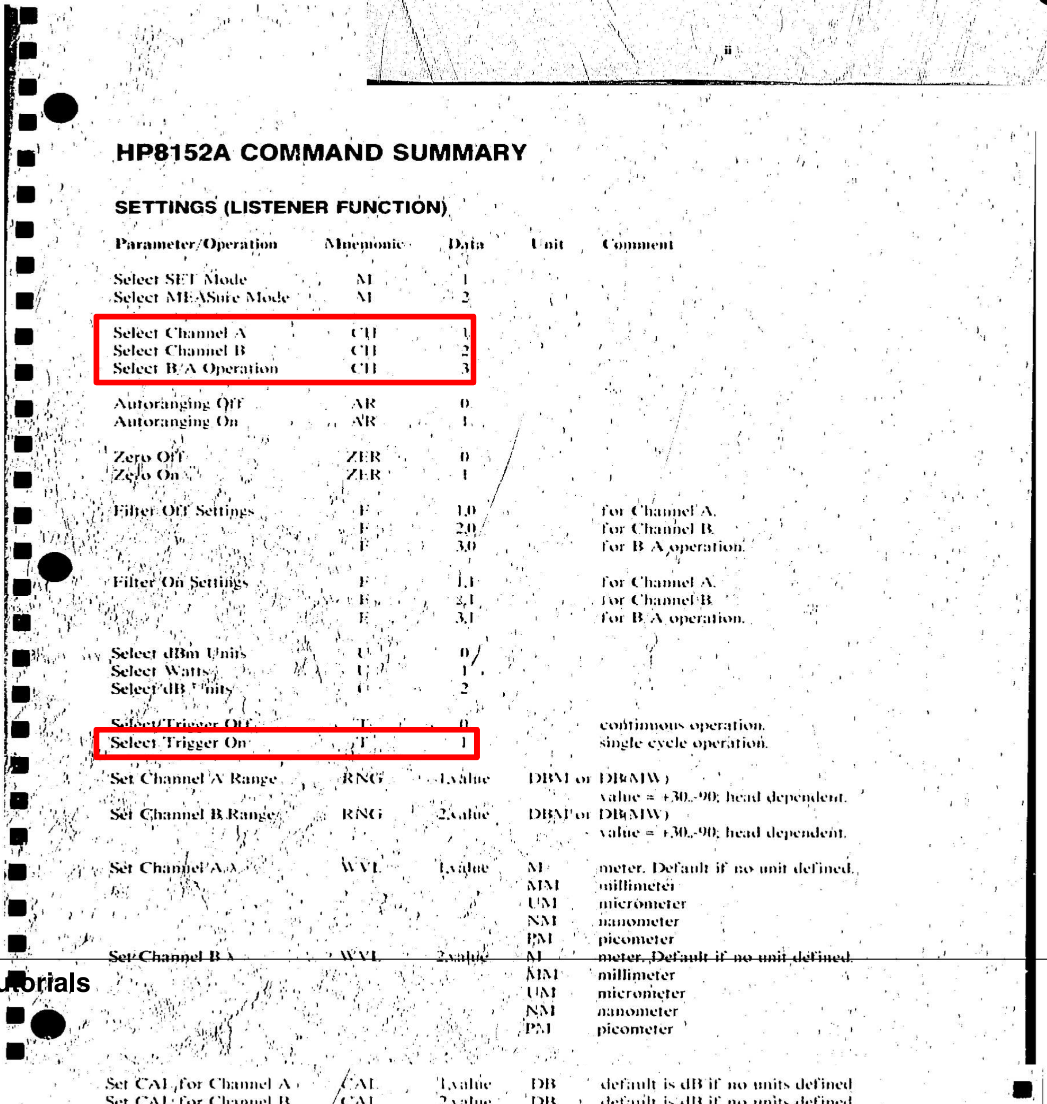
MANUAL PART No. 08152-90002
MICROFICHE PART No. 08152-95002

PRINTED: MARCH 1987



CONTENTS (CONT.)

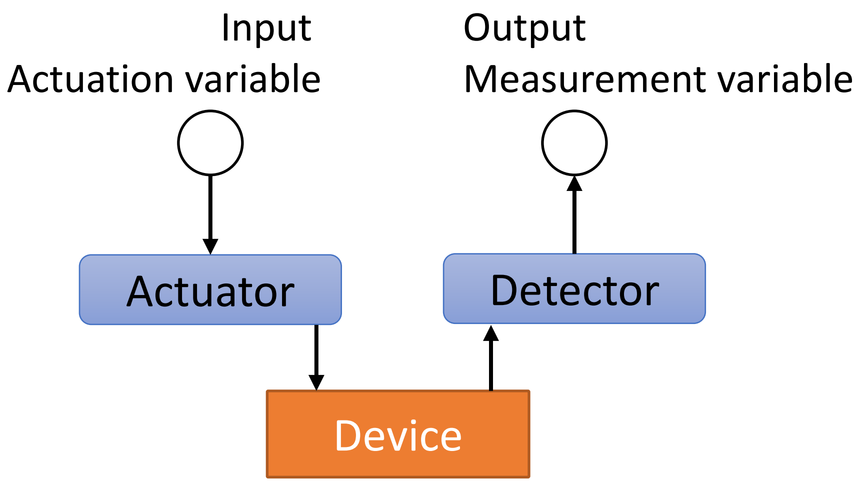
ii



HP8152A COMMAND SUMMARY

SETTINGS (LISTENER FUNCTION)

Parameter/Operation	Mnemonic	Data	Unit	Comment
Select SET Mode	M	1		
Select MEASure Mode	M	2		
Select Channel A	CH	1		
Select Channel B	CH	2		
Select B/A Operation	CH	3		
Autoranging Off	AR	0		
Autoranging On	AR	1		
Zero Off	ZER	0		
Zero On	ZER	1		
Filter Off Settings	F	1,0		for Channel A.
	F	2,0		for Channel B.
	F	3,0		for B/A operation.
Filter On Settings	F	1,1		for Channel A.
	F	2,1		for Channel B.
	F	3,1		for B/A operation.
Select dBm Units	U	0		
Select Watts	U	1		
Select dB Units	U	2		
Select Trigger Off	T	0		continuous operation.
Select Trigger On	T	1		single cycle operation.
Set Channel A Range	RNG	1,value	DBM or DB(MW)	value = +30..-90; head dependent.
Set Channel B Range	RNG	2,value	DBM or DB(MW)	value = +30..-90; head dependent.
Set Channel A λ	WVL	1,value	M	meter. Default if no unit defined.
			MM	millimeter
			UM	micrometer
			NM	nanometer
			PM	picometer
Set Channel B λ	WVL	2,value	M	meter. Default if no unit defined.
			MM	millimeter
			UM	micrometer
			NM	nanometer
			PM	picometer
Set CAL for Channel A	CAL	1,value	DB	default is dB if no units defined
Set CAL for Channel B	CAL	2,value	DB	default is dB if no units defined



Basic concepts

A typical sweep might look like this:

```
actVals = np.linspace(0, 1, 10)
measVals = np.zeros(len(actVals))
for i, vA in enumerate(actVals):
    actuate(vA)
    measVals[i] = measure()
plt.plot(actVals, measVals)
```

There are a few things going on here. Every time a measurement is taken, it is paired with an actuation. In other words, something in the lab changes that you control, and then you look at what happened.

1. An actuation *procedure*: `actuate`
2. A measurement *function*: `measure`
3. A series of actuation arguments: `actVals`
4. Corresponding measurement results: `measVals` (pre-allocated)
5. Post processing, in this case, plotting

The role of the `for` loop is to get one argument and pass it to the actuation procedure, then take one measurement and store it in the pre-allocated array.

A major problem here is that the important information is distributed all throughout the for loop structure. We would like to specify those things upfront. The *simpleSweep()* function does this in a bare bones version.

Challenges of more advanced sweeps

- The code gets difficult to read
- Often they are repeated with only small changes *somewhere* in the loop
- They can take a long time
- Processing and analysis occur only after they complete

The information can be distributed all throughout the code. This is especially the case when there are multiple dimensions, intermediate monitoring (e.g. plotting) and analysis (e.g. peak picking), and various data formats. What if we want to make a small change? The location in code is not obvious.

Since they take a long time, we want to get intermediate information out to the user via progress printing and reporting, maybe even visualization. Progress reporting can tell you when the sweep is likely to finish, so you can decide whether there's enough time to get a coffee or to get some sleep.

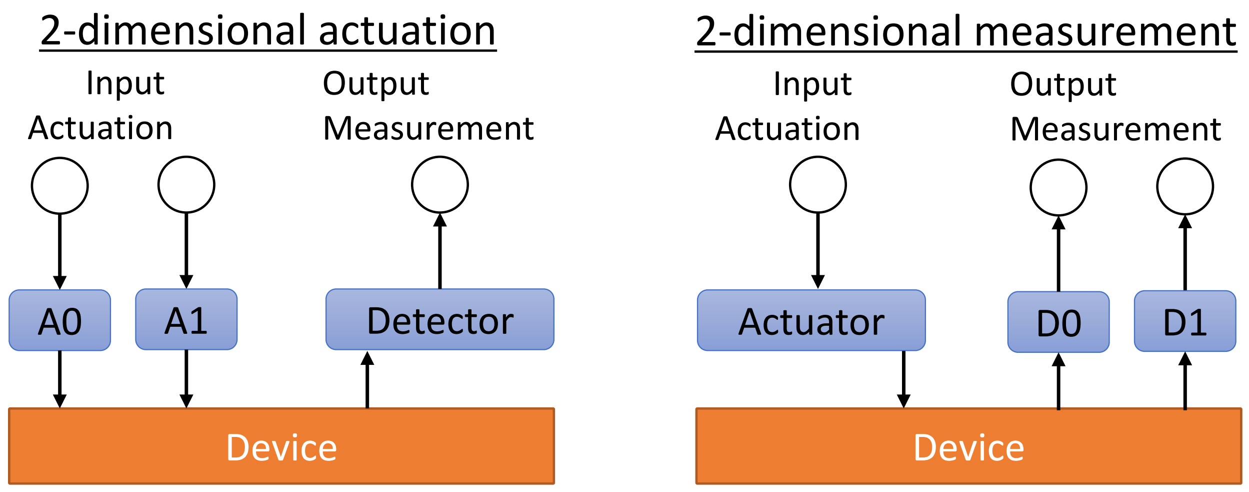
Intermediate analysis can also show you how it's going to decide whether to continue or stop. The relevant information could require lots of processing, such as if you want to know how a peak is moving. We want to put arbitrarily advanced analysis within the loop, and connect it to intermediate plotting.

The worst is when you finish a sweep and the bulk processing at the end throws an exception. You have to repeat the sweep. Or if you are returning to an old notebook to fix up a figure for a paper. You have to repeat the sweep. We want convenient ways to save the data an reload it as if the sweep had just occured fresh.

`Sweeper` is a way to re-organize the for-actuate-measure setup. All of the important information can be specified at the beginning. All of the bells and whistles like monitoring and plotting happen under the hood. It has two important subclasses, `NdSweeper` and `CommandControlSweeper`.

N-dimensional sweeps with `NdSweeper`

Concept



Sweeps can occur in several dimensions of actuation and/or measurement. Suppose we want to see how some measured (dependent) variables depends on two actuated (independent) variables

```
1 aAct = np.linspace(0, 1, 10)
2 bAct = np.linspace(10, 20, 3)
3 measMat = np.zeros((len(aAct), len(bAct)))
4 for ia, a in enumerate(aAct):
5     act_1(a)
6     for ib, b in enumerate(bAct):
7         act_2(b)
8         measMat[ia, ib] = measure()
9 plt.pcolormesh(aAct, bAct, measMat)
```

The for loops get nested with each sub-row calling its own actuate. Measurement always happens in the inner-loop. Alternatively, all actuation can happen on the inner loop by flipping lines 4 and 5. The order and precedence of actuation calls is critical.

In the package, all of this functionality and more is implemented in the `NdSweeper`. One specifies the domain (`aAct`, `bAct`) and the functions to call in each dimension (`act_1` and `act_2`). One also specifies the measurements that should be taken (`meas_1`, `meas_2`). The sweep is executed with the `gather()` method.

Usage

`NdSweeper` also supports a *`subsume()`* method which combines a N-dimensional sweep with a M-dimensional sweep into a (N+M)-dimensional sweep.

Basic data structure concept

NdSweeper has attributes containing function pointers. These tell it what to do when actuating, measuring, or parsing. The actuation values are specified at the time of the actuation function. *All of these things must have name/key* that is unique within the sweep. All of their value data is stored in a common data structure that has N array-like sweep dimension(s) and one dictionary-like dimension for different data memebers. When a sweep completes, the entire grid of values for a given data member can be accessed with `swp.data[key]`, returning an ndarray. On the other hand, *all* of the data for a given sweep point can be accessed with `swp.data[ndindex]`, returning a dict. (Don't worry about the implementation of that structure)

Specifying actuation

Actuation values are determined when specified. Their dimensions determine the sweep and data dimension. The order that they are added affects the sweep priority, such that the first sweep addded will be swept at each point of the second added, etc. An actuation function has one argument which is provided by the actuation value at that index. If there is a return, that is treated as a separate measurement. Doing on every point is specifiable.

`NdSweeper.`**`addActuation`**(*name*, *function*, *domain*, *doOnEveryPoint=False*)

Specify an actuation dimension: what is called, the domain values to use as arguments.

Parameters

- **`name`** (*`str`*) – key for accessing this actuator's value data
- **`function`** (*`func`*) – actuation function, usually linked to hardware. One argument.
- **`domain`** (*`ndarray, None`*) – 1D array of arguments that will be passed to the function. If None, the function is called with a None argument every point (if doOnEveryPoint is True).
- **`doOnEveryPoint`** (*`bool`*) – call this function in the inner loop (True) or once before the corresponding rows(False)

Specifying measurement

Measurement values are filled in point-by-point for every sweep index. They depend only on external function results, not on stored data. Measurement functions are called with no arguments. Returning is mandatory. The order does not matter theoretically, but it is preserved (first added, first called).

Special case: if the actuation method has a return type that is *not* `NoneType`, a measurement will automatically be created to capture these values. This measurement key will be the actuation key, plus `'-return'`.

`NdSweeper.`**`addMeasurement`**(*name*, *function*)

Specify a measurement to be taken at every sweep point.

Parameters

- **`name`** (*`str`*) – key for accessing this measurement's value data

- **function** (*func*) – measurement function, usually linked to hardware. No arguments.

Parsers: what and how

Parsers are functions of the sweep data (which may include the results of other parsers). They have one argument, a dictionary of data members *at a given sweep point*. The order they are added is important if the execution of one parser depends on the result of another. Parsers added after the sweep is gathered will be fully calculated automatically. During the sweep, parsers are calculated at every point. They typically do not interact with hardware nor do they depend on sweep index; however, they are allowed to interact with persistent external objects, such as a plotting axis.

`NdSweeper.`**`addParser`**(*name*, *function*)
Adds additional parsing formulas to data, and reparses, if data has been taken

Parameters

- **name** (*str*) – key for accessing this parser's value data
- **function** (*func*) – parsing function, not linked to hardware. One dictionary argument.

Static data

Parsing functions can depend on values that are not measured during the sweep. Give it a name key and it can be accessed by parsers just like a measurement. When adding static data, it will expand to fit the shape of the sweep, to some extent (see the docstring). That means you can add static data that is constant using a scalar and variable using an ndarray.

`NdSweeper.`**`addStaticData`**(*name*, *contents*)
Add a ndarray or scalar that can be referenced by parsers

The array's shape must match that of the currently loaded actuation grid.

If you then `addActuation()`, the static data automatically expands in dimension to accomodate. Values are filled by tiling in the new dimension.

Add static data after the actuations that have different static data, but before the actuations for which you want that data to be constant.

Parameters

- **name** (*str*) – key for accessing this data
- **contents** (*scalar, ndarray*) – data contents

Tricks with array actuation

Some actuation procedure can not be separated into different functions, each with one argument. Some need multiple arguments, and you may be interested in sweeping both. The memory allocation is the same:

```
aAct = np.linspace(0, 1, 10)
bAct = np.linspace(10, 20, 3)
measMat = np.zeros((len(aAct), len(bAct)))
```

But the `for` loop is fundamentally different

```
for ia, a in enumerate(aAct):
    for ib, b in enumerate(bAct):
        act(a, b)
        measMat[ia, ib] = measure()
```

What this means is that we need to restructure how the sweep is specified, and the functions the user gives it.

Todo: Array actuation is not currently supported by NdSweeper, but should be. Fundamentally, CommandControlSweeper is of the array actuation type, and that is implemented. Perhaps this calls for a new subclass of `Sweeper`

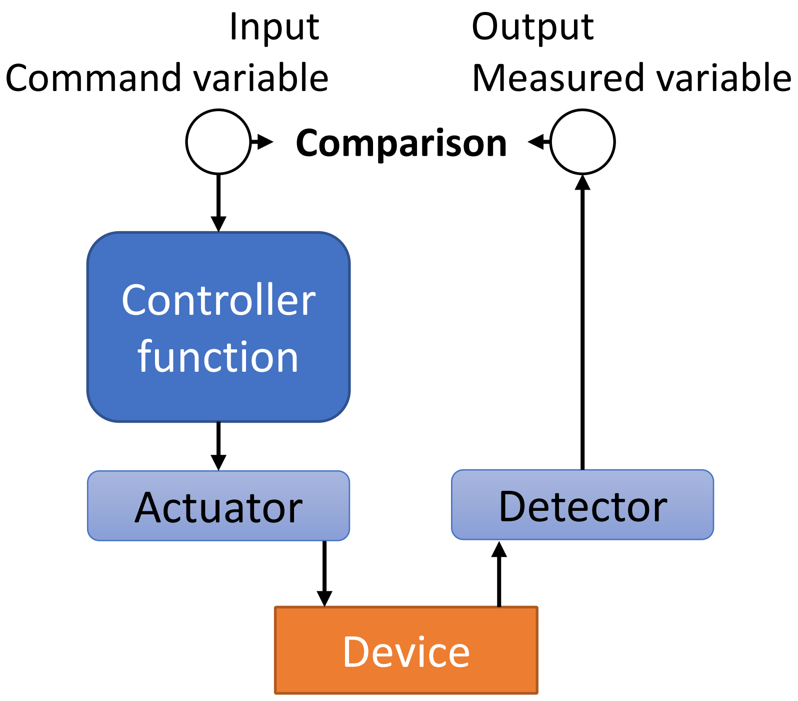
- genindex
- modindex
- search

Command-control sweeps

Note for documenter The basics of this section should go on a different page about command-control without sweeping. Then on this page, it can focus just on the challenge of sweeping them

Concept

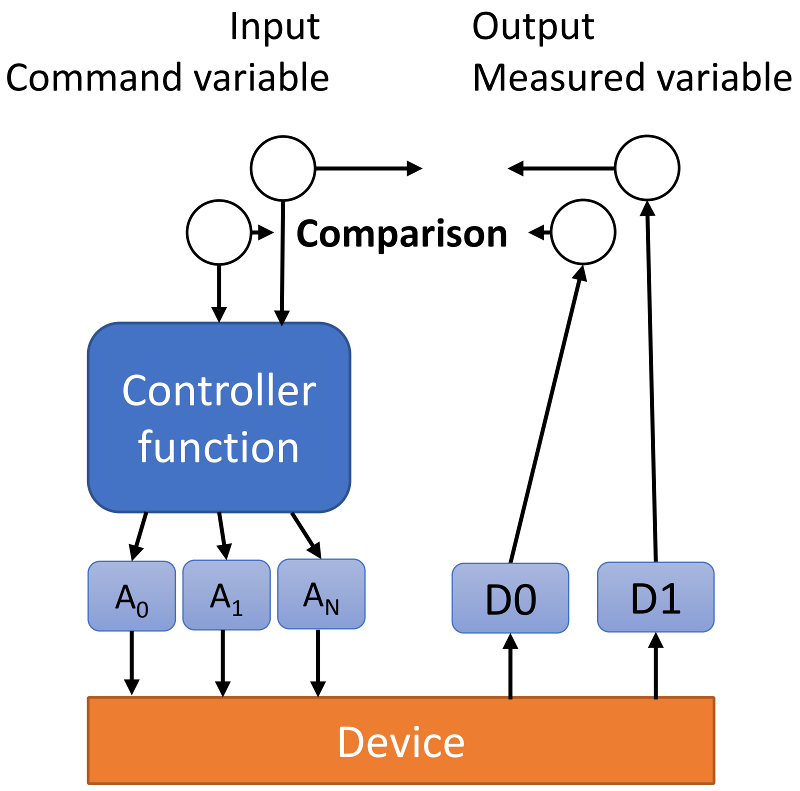
These are special in that the actuation function attempts to invert the behavior of the physical system, such that the input is nominally seen as the measured output.



Since they are trying to reproduce a response equal to the input, the number of actuation and measurement dimensions are equal. So in 1D:

```
ctrlVals = np.linspace(0, 1, 10)
measVals = np.zeros(len(ctrlVals))
for i, cVal in enumerate(ctrlVals):
    actVal = control(cVal)
    actuate(actVal)
    measVals[i] = measure()
```

Note that the `actuate` function is still there, but its argument comes from the `control` function. Ideally, `ctrlVals` will equal `measVals`. Their difference gives us an idea of control error.



In 2D, the control function is rarely seperable, which means these sweeps fall into the array actuation type.

```
aCtrl = np.linspace(0, 1, 10)
bCtrl = np.linspace(10, 20, 3)
ctrlMat = np.zeros((len(aAct), len(bAct), 2))
measMat = np.zeros((len(aAct), len(bAct), 2))
for ia, a in enumerate(aAct):
    for ib, b in enumerate(bAct):
        ctrlMat[ia, ib, :] = [a, b]
        actArr = control([a, b])
        actuate(actArr)
        measMat[ia, ib, :] = measure()
```

Notice that `measMat` is now 3 dimensional, with the third dimension corresponding do which variable. Highlighted lines show how to construct the expected `ctrlMat`. It makes more sense to fill that control matrix before doing the actual sweep. This can instead be done with meshgrid commands:

```
aGrid, bGrid = np.meshgrid(aCtrl, bCtrl)
ctrlMat = np.array((aGrid, bGrid)).T # ctrlMat.shape == (10, 3, 2)
```

There is an advantage to doing this at first in that the sweep loop is simplified and more flexible.

```
for swpIndex in np.ndindex(ctrlMat.shape[:-1]):
    actArr = control(ctrlMat[swpIndex])
    actuate(actArr)
    measMat[swpIndex] = measure()
```

Voila! This structure is the same as the 1-dimensional command-control sweep: one line each for control, actuate, and measure. It takes advantage of NumPy's n-dimensional for loop iterator.

Usage

- genindex
- modindex
- search

The `Sweeper` class: features and options

Progress monitoring

Use *`setMonitorOptions()`* to set and get. To see how the sweep is coming along, you can choose to print to stdout or to serve a page available anywhere online. If plotting is also set up, you can live plot every point in your notebook as it is being taken. Here are the options

stdoutPrint Print the sweep index to stdout to see progress

cmdCtrlPrint (only with CommandControlSweeper) Print the sweep index, command value, and measured value to see the errors

livePlot Refresh plots every data point when in an IPython notebook. Options specified in `setPlotOptions` will be used.

plotEvery Number of points to wait before refreshing live plot

runServer Print the sweep index to a file that is served online

Note: If your actuate-measure routine is fast, then live plotting can slow down the sweep with the need to refresh graphics. Set `plotEvery` to an integer more than 1 to do less plotting.

Warning: Live plotting is not yet supported for surf plots, and there are a few bugs with 1D command-control plots.

If `runServer==True`, to serve the page, you must first start the server (see *here*), making sure to set up the right domain, domainHostName, monitorServerDir, and monitorServerPort. If you are using `Sweeper`, it configures your sweep to write to the server.

Note: To instead do it manually, you would make a `ProgressWriter`:

```
prog = io.ProgressWriter(swpName, swpShape, runServer=True)
```

and then call `prog.update()` every inner-loop iteration.

Plotting

Todo: Another bug when using xKey equal to the major sweep axis. It sometimes only displays movement along x=constant lines.

Use *`setPlotOptions()`* to set and get. Different plots are available for different kinds of sweeps. Some of the options are only valid with a given type. For most purposes, the best options are detected automatically, so you don't have to set them. Here are the options.

NdSweeper

plType (str)

- **`'curves'` (1D or 2D)** Standard line plots. If 2D, a set of lines with a legend will be produced.
- **`'surf'` (2D only)** Standard surface color plot

xKey (str, tuple) Abscissa variable(s)

yKey (str, tuple) Ordinate variable(s)

cmap-surf colormap

cmap-curves colormap

A grid of axes will be produced that depends on the length of the tuples xKey and yKey. If both xKey and yKey are strings, only one plot axis is made. By default, the x (y) keys are filled with the actuation (measurement) variables that are detected to be scalar.

CommandControlSweeper

plType (str)

- **`'curves'` (1D only)** A line plot *[TFerreiradLimaN+16a]* showing mean and variances of measured vs. command
- **`'cmdErr'` (1D or 2D)** A special grid plot *[TFerreiradLimaN+16b]* showing mean quivers and variance ellipses

Saving and loading

`Sweeper` provides two sets of save/load. The file is determined by the `io.fileDir` variable and the object's `savefile` attribute. These can be combined with a gathering boolean to determine whether you want to retake the sweep or load it from a saved version.

`save` and `load` do just the `data` attribute.

```
swp = NdSweeper(...)
...
swp.savefile = 'dummy'
if isGathering:
    swp.gather()
    swp.save()
else:
    swp.load()
```

Saving the entire object is good if the domains change, which is particularly important for command-control types. The problem is that references to bound functions cannot be pickled. The `saveObj` and `cls.loadObj` methods try to do the entire object, while leaving out the actuation and measurement function references.

```
myfile = 'dummy'
if isGathering:
    swp = CommandControlSweeper(...)
    ...
    swp.gather()
    swp.saveObj(myfile)
else:
    swp = sUtil.CommandControlSweeper.loadObj(myfile)
```

Todo: NdSweeper has no loadObj yet. This seems reasonable to do by stripping the bound references. Consider deprecating saving/loading just data and the savefile attribute.

- genindex
- modindex
- search

Other actuate-measure situations

Peak search and binary search can be done interactively with a peaked or monotonic (respectively) system. Those examples are found in here

Todo: Currently peak search is like a n-point 1-D Nelder Meade search. That could be extended to multiple dimensional optimization.

- genindex
- modindex
- search

2.4.4 Characterization in time

Presumably, you want to get some advanced knowledge about how your devices behave in time. That could be either on short timescales, much faster than measurements can complete, or on long timescales, much slower than you're willing to sit there.

Monitoring a value over time is pretty self explanatory. See *`monitorVariable()`*.

Strobe tests are much more interesting. Check out *`sweptStrobe()`* in here

- genindex
- modindex
- search

2.4.5 Virtualization

In this section

Virtual experiments are meant to behave exactly like a real lab would, except by using code calls to simulators rather than real instruments. This is useful for several reasons

1. Developing/debugging procedures quickly and safely
2. Validating that procedures will work and not go out of range before running on a real device
3. Unit testing code that refers to instruments in a repeatable virtual environment

This section refers to the example in

Procedures, virtualization, abstract procedures

Demonstration of using an actuate/measure procedure to get data and analyze something about the data.

The procedure itself needs to be developed and debugged. This notebook shows how to do that virtually. When ready it goes to experiment by flipping a switch.

Experimental setup:

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        from IPython import display

        from lightlab.laboratory.virtualization import VirtualInstrument, DualInstrument
        import lightlab.laboratory.virtualization as virtUtil
```

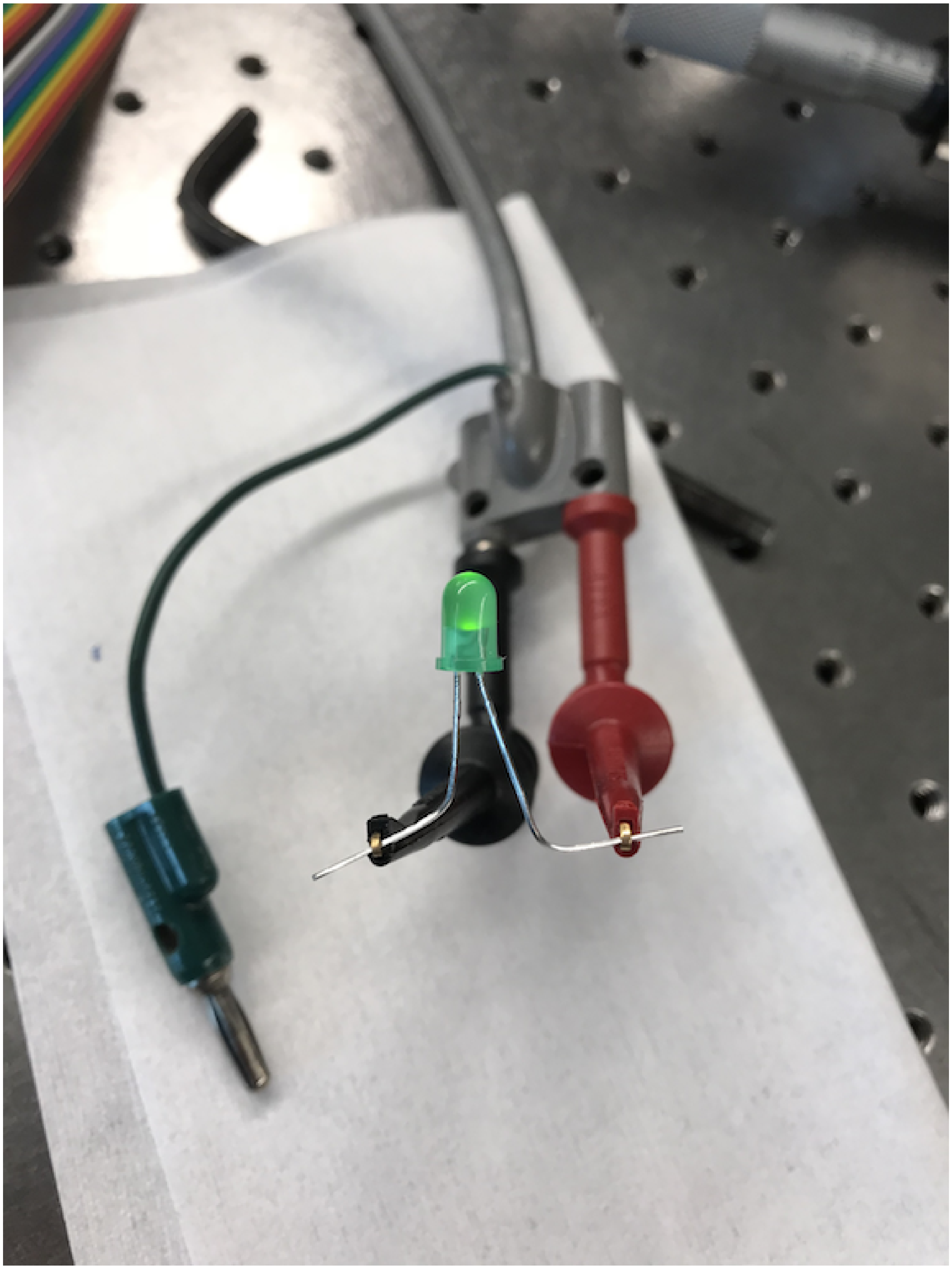
Semi-libraries

These are python files in the same directory as this notebook that have some functions and classes. Import them to the notebook. You should be writing/developing them simultaneously with the notebook. This practice is recommended because .py files work well with `git diff` but .ipynb files do not. As a rule of thumb, don't put instrument access within the semi-libraries. You can use them for commonly used and/or long procedures, functions, sweep declarations, etc.

```
In [2]: from myProcedures import extractThreshold
```

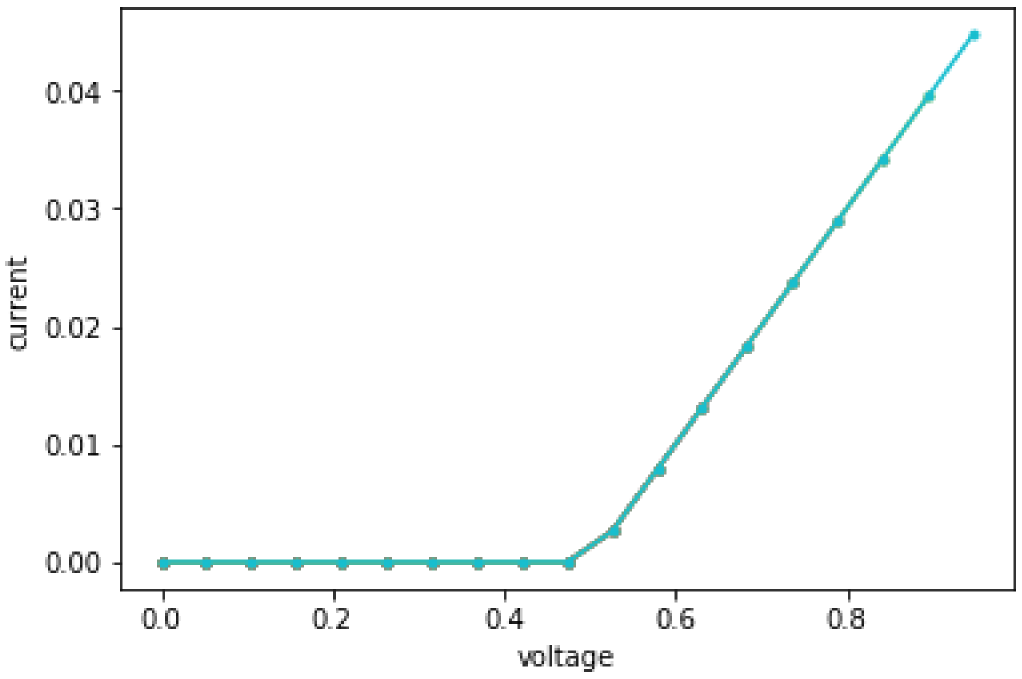
This is a model of a diode

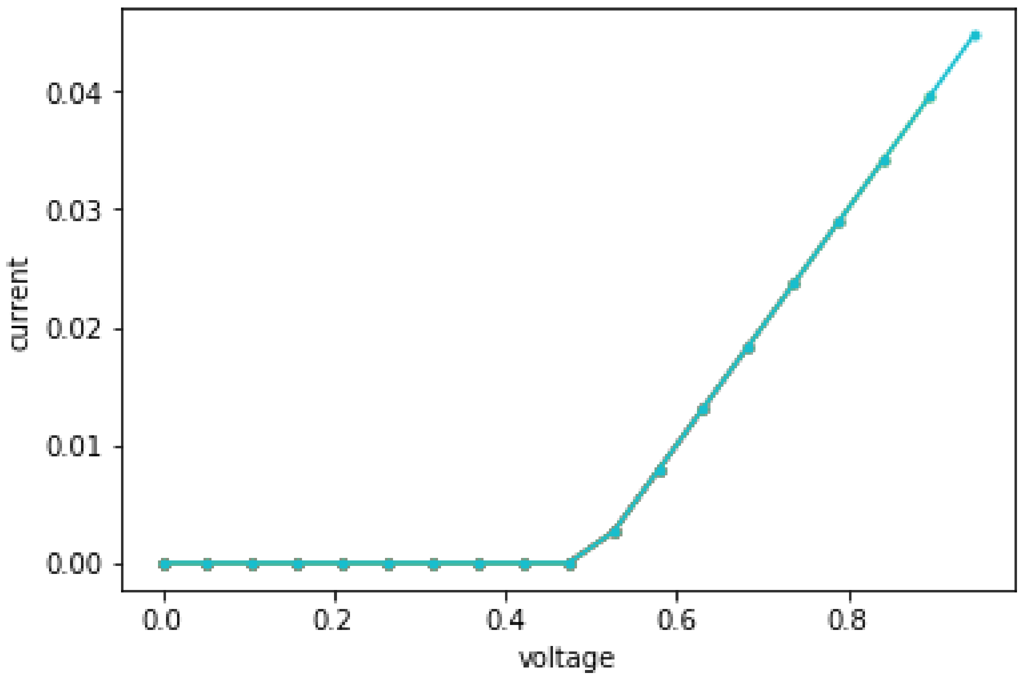
It has * parameters, like threshold voltage * methods for simulating: this applied voltage will give that observed current - apply (a.k.a. actuate) –> observe (a.k.a. measure)



It does not have state. - The observations are completely determined by the actuation *now* - This is not a requirement - The only reason for a simulation model to have state is if the device you are trying to model have hysteresis (or if you are caching)

```
In [3]: # To debug this procedure, you will simulate a diode similar to the real one
        class Diode():
            def __init__(self, threshV, Rinline=10):
                self.threshV = threshV
                self.Rinline = Rinline

            def ivResponse(self, atVoltage):
                return max(0, atVoltage - self.threshV) / self.Rinline

        myDiode = Diode(threshV=.5)

        # Stone age evaluation
        fi, ax = plt.subplots(figsize=(6,4))         # line 0
        plt.xlabel('voltage')
        plt.ylabel('current')
        vArr = np.linspace(0, 1, 20)
        iArr = np.zeros(len(vArr))
        for jV, V in enumerate(vArr):
            iArr[jV] = myDiode.ivResponse(V)
            plt.plot(vArr[:jV], iArr[:jV], '.-')
            display.clear_output(wait=True)
            display.display(plt.gcf())               # 10 line for loop
```





The parameter extraction procedure

A procedure consists of a sequence of actuation and measurement operations that interface with laboratory instruments. The actuation might be determined beforehand (i.e. sweep) or it could be changed depending on what is measured (i.e. search).

The **procedure** is often the most complicated part of your experimental code. The **procedure** is what you are developing and debugging here

Example, a parameter extraction type of procedure

In this example, we want to find the diode threshold voltage * Acquire: do a sweep in voltage, measuring current * Analyze: look for the maximum second-derivative

Notes

- NdSweeper class (overkill for now) and the concept of passing methods as arguments
- These methods are NOT called yet because
  - we don't yet know if this procedure is real or virtual (it could be both)

```
In [4]: extractThreshold?
```

The virtual instrument

This class basically holds the state that is normally found in real life. It interacts with the simulation model.

Why do you use the Virtual instrument instead of just using the simulator?

Because this is the API for the real life instruments. You should not have to make * prodecure 1: experimental using instruments, and * procedure 2: virtual using a simulation model, calling things like `ivResponse()`

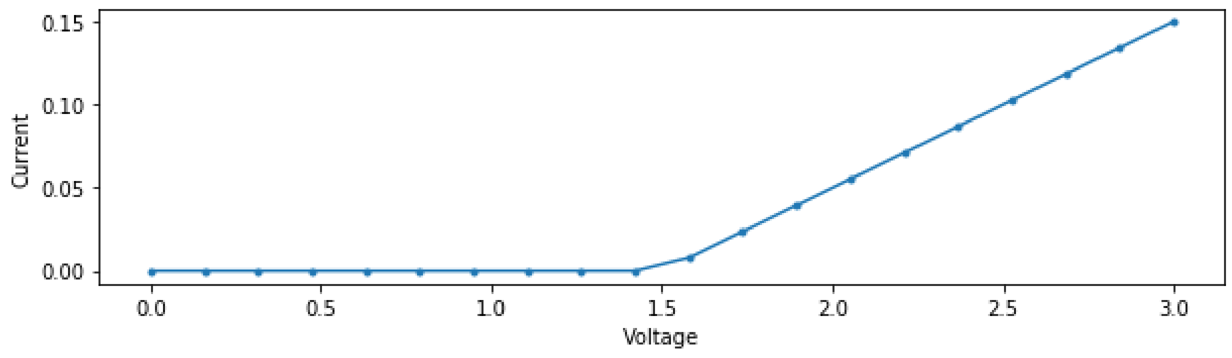
`VirtualKeithley` provides the same API as `Keithley` (the real instrument class)

```
In [5]: class VirtualKeithley(VirtualInstrument):
            def __init__(self, viResistiveRef):
                self.viResistiveRef = viResistiveRef
                self.__appliedVoltage = 0  # state

            def setVoltage(self, volts):
                self.__appliedVoltage = volts

            def measCurrent(self):
                return self.viResistiveRef.ivResponse(self.__appliedVoltage)
```

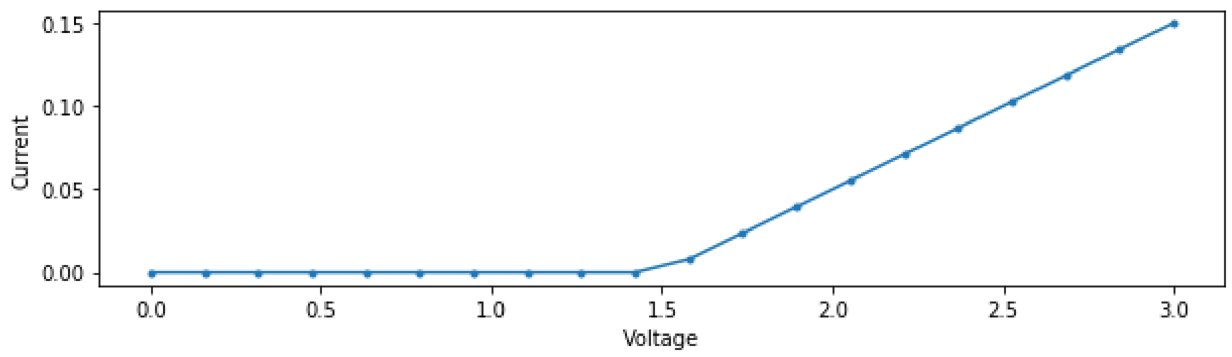
Running it

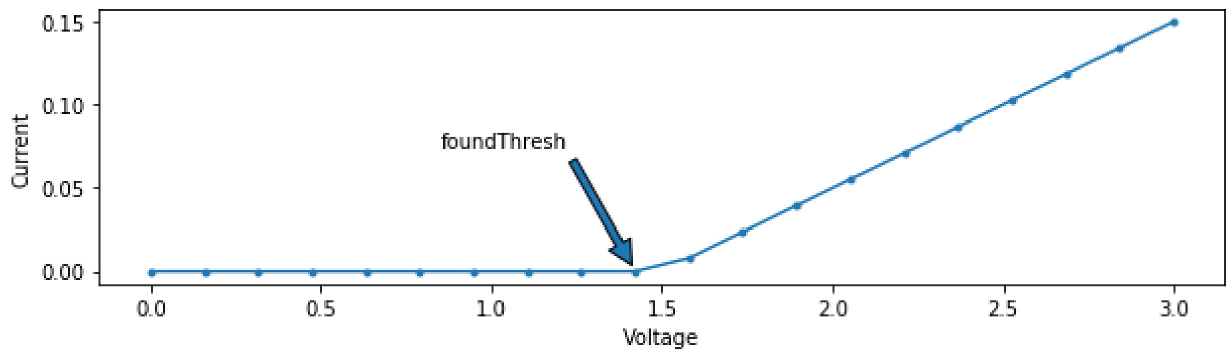
Make a diode model, connect it to the virtual keithley, execute the procedure. Then, get the extracted parameter from the procedure. Compare it to the hidden one. We are now *evaluating* a procedure.

```
In [6]: hiddenThresh = 1.5
        myDiode = Diode(hiddenThresh)
        keithley = VirtualKeithley(viResistiveRef=myDiode)

        foundThresh = extractThreshold(keithley, vMax=3)
        err = foundThresh - hiddenThresh
        print('Error =', abs(err) / hiddenThresh * 100, 'percent')
```



```
Error = 5.263157894736844 percent
```





Warning the following cells access real instruments

(This warning should be apparent in all your notebooks)

You can prevent any hardware access using the `virtualization.virtualOnly` variable.

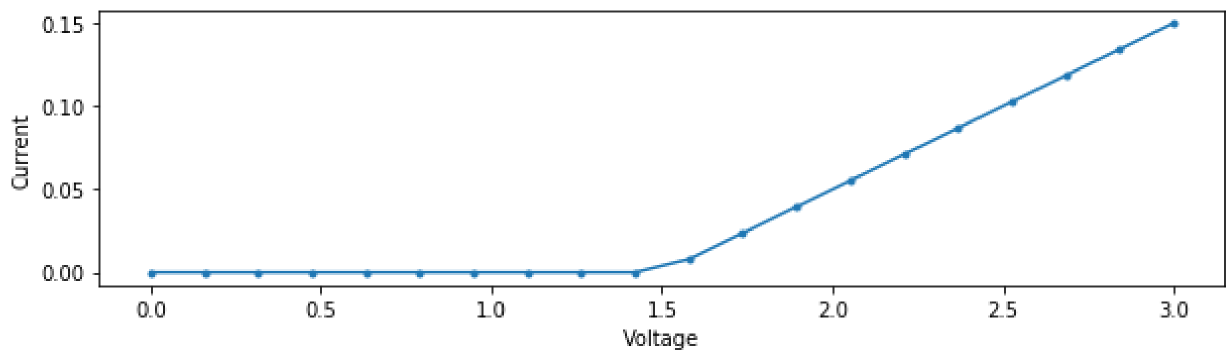
```
In [7]: virtUtil.virtualOnly = True
```

The hardware instrument

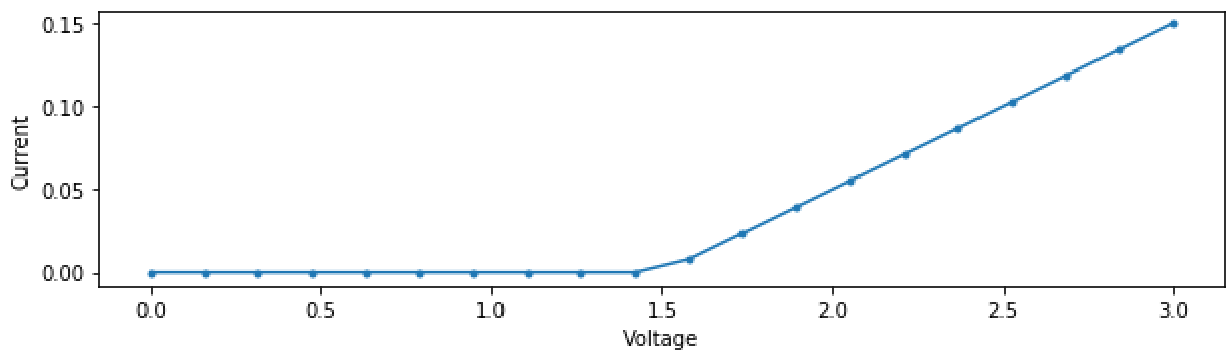
Is pulled from the instruments_dict. In this case, "Keithley 23." You need to build this dict elsewhere using the tools from `lightlab.laboratory.state`. We don't just want a `VirtualInstrument`, we want something that can switch between virtual and real on the fly. That is a `DualInstrument`.

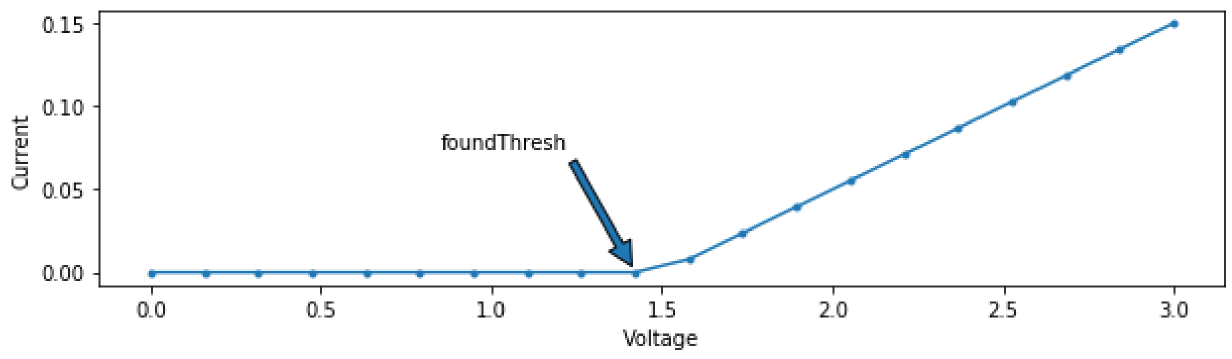
```
In [8]: if not virtUtil.virtualOnly:
            from lightlab.laboratory.state import lab
            dualKeithley = DualInstrument(real_obj=lab.instruments_dict['Keithley 23'],
                                          virt_obj=VirtualKeithley(myDiode))
            with dualKeithley.asReal():
                dualKeithley.setVoltage(0.)
                dualKeithley.setProtectionCurrent(50e-3)
        else:
            dualKeithley = DualInstrument(virt_obj=VirtualKeithley(myDiode))
```

```
In [10]: with dualKeithley.asVirtual():
             foundModel = extractThreshold(dualKeithley)
             print('The model threshold is', foundModel)
         with dualKeithley.asReal():
             foundDevice = extractThreshold(dualKeithley)
             print('The device threshold is', foundDevice)
```



```
The model threshold is 1.4210526315789473
```





```
In [ ]:
```

Procedural abstraction

A procedure is automated code that uses instruments. It could just be a simple sweep, or it could be a complex interactive search. The goal of a procedure could be extracting parameters from a device (see the demo in lightlab/notebooks/Examples), controlling something (such as a peak tracker), or calibrating something.

In a real setting, the procedure is given reference to a hardware `Instrument`. The instrument contains a driver that talks to the actual piece of equipment. This equipment is hooked up to a real-life device.

In a virtual setting, we can use a *`VirtualInstrument`* to provide a partial API that matches the real Instrument. In the example, the provided methods are `setVoltage` and `measCurrent`. The virtual setting needs a model to determine what will be measured given a particular actuation.

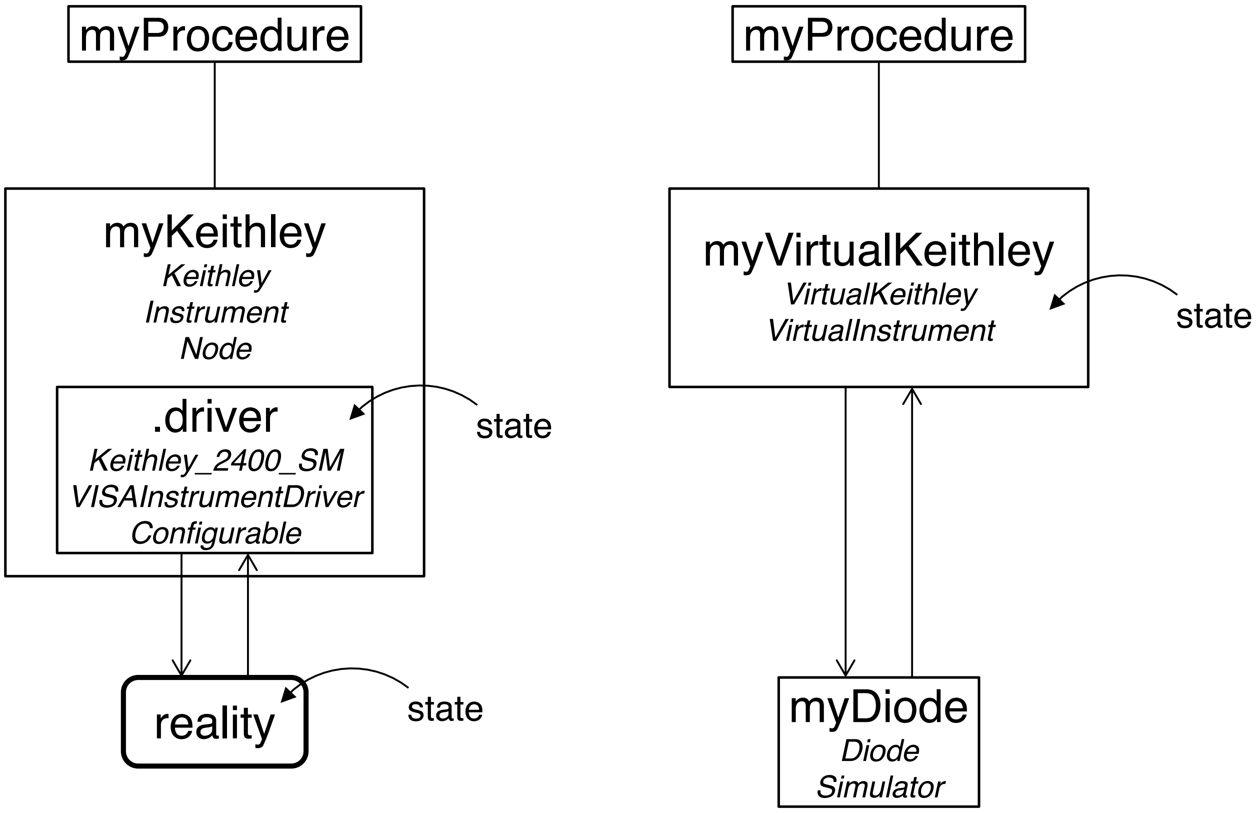


Fig. 6: Comparison of a real experiment and a virtual experiment. A key difference is where state is held.

Why separate VirtualInstrument and the simulation model?

Instead, we could make a class called `VirtualKeithleyWithDiodeAttached` that provides the same methods. It's `getVoltage` method would do the diode computation. There are a few reasons why we argue not to do that.

1. **Keeping state in one place** In the real experiment, the entire "state" of the lab can be described by what is in the drivers (which should match the configuration of the actual equipment). Similarly, for virtual, you should not have to go digging around the simulator to figure out the entire "state".
2. **Avoid creating a special purpose instrument for every experiment** You can re-use VirtualKeithley with a different model in its `viResistiveRef`.
3. **Enforces the proper namespace** Your procedure should not be able to directly see your model. It should only be interacting with Instrument-like things
4. **Functional simulators** This means, if the input is the same, the output is always the same. Also, the simulators cause no side-effects. Easy to test and debug. Easy to compose into larger simulators.

Note: This is not a hard and fast rule. Reasons to store state in the simulator is if there is hysteresis, or, for performance reasons, it might make sense to cache results within the simulator.

Clearly, `VirtualKeithleyWithDiodeAttached` is a bad instrument because it is not re-usable. It is a bad simulator because it cannot be composed with other simulators, and it is difficult to unit test because the return of `getVoltage` depends on history. These points come into play when simulation models get more complicated.

Dual Instruments

`DualInstrument` wraps two instruments: one real and one virtual. The procedure can be given a reference to the dual instrument, just as it was before. The dual construct makes sure that there is an exact correspondence between the two cases.

Dual instrument is *`Virtualizable`* which means it has an attribute `virtual` that controls the switch. More useful: it provides context managers called `asReal` and `asVirtual`. The benefit of context managers is they allow entry and exit operations, in this case, usually hardware warmup and cooldown methods. They can also be used to synchonize multiple Virtualizable things in more complex cases. See *`synchronize()`*.

- genindex
- modindex
- search
- genindex
- modindex
- search

2.5 Miscellaneous Documentation

2.5.1 Git with ipython notebooks

Interactive tutorials are in notebooks. A full "experiment" in the lab is contained in a notebook. Notebooks are supposed to change a lot and meant to be played with. They are graphical. They are also essential to track.

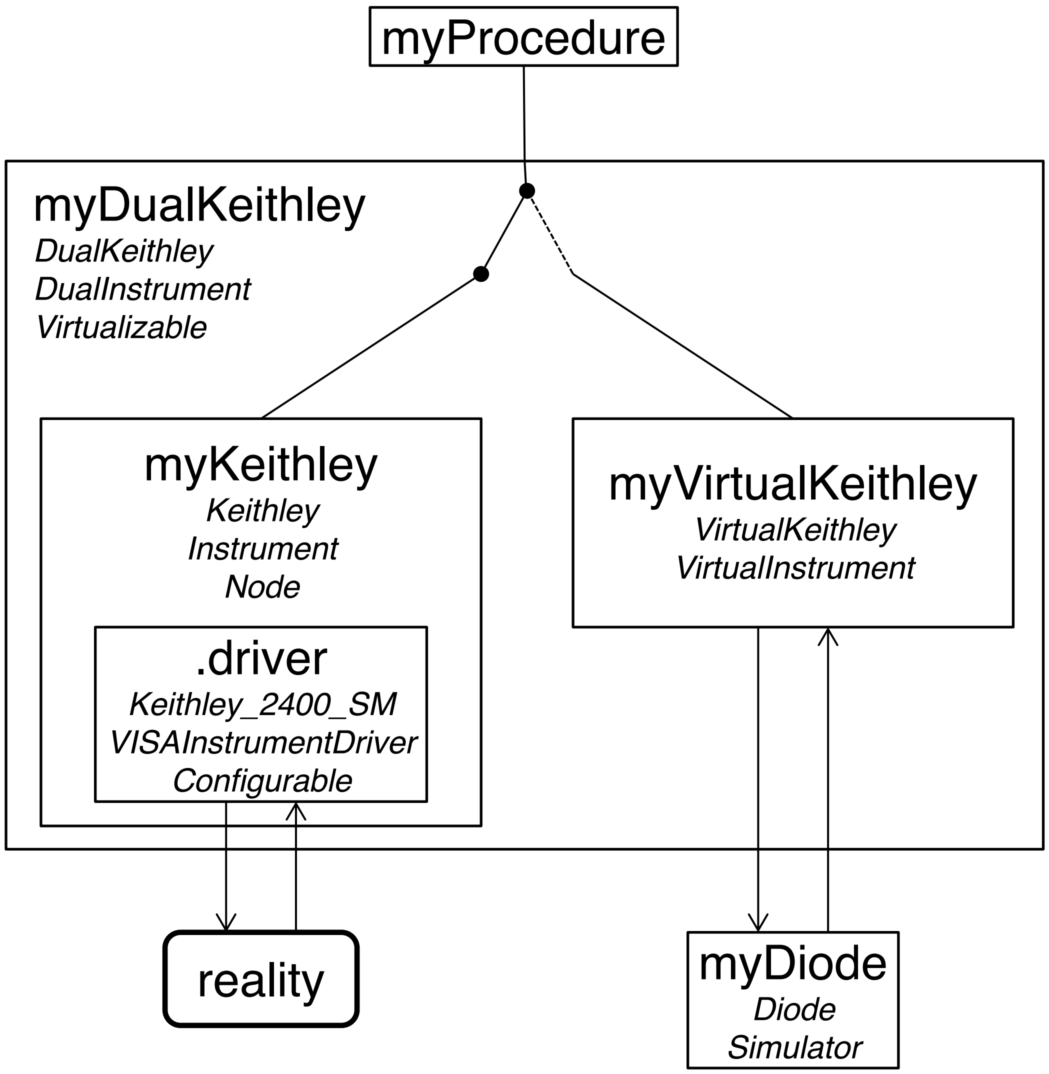


Fig. 7: A dual experiment for testing `myProcedure`. It can run either as virtual or as real by flipping a switch in `myDualKeithly`, without rewriting any code in `myProcedure`

Problem 1

- Diff-ing your work against someone else's is impossible
- Changes to binary outputs take up a huge amount of space, even if nothing significant actually changed

Jupyter notebooks have two sections: inputs (code, markdown) and outputs (stdout, plots, images). Interactive python notebook files embed compiled outputs. This is good if you want to restart a kernel but still see the output, or if you close the file, etc.

Solution 1 and Problem 2: The nbstripout filter

`nbstripout` is a Git filter and "hides" the output and some metadata in `.ipynb` files from Git such that it does not get committed. This allows only tracking the actual input code cells in Git. It is installed via the `requirements.txt`, but there is also some interesting discussion and documentation

There are three downsides:

1. What if you liked keeping those outputs without rerunning every commit?
2. It has to strip evvverything, including all those high-quality graphics, every single time you `git status`.
3. It crashes your essential commands. Very easy to get into a chicken-and-egg hole where you can't `diff` anything because __some__thing isn't JSON – causing a crash – but you can't figure out what isn't JSON because you can't see which files just changed.
4. It can corrupt files. That's why we made `cleannbline`.

Solution 2. Deactivate the nbstripout filter

```
source venv/bin/activate
nbstripout --uninstall
```

Never think about it again… until you have to merge.

Best practice

Ultimately, some of the work in notebooks will be lost. This is desireable in the case where two people made slightly different versions of the same figure. However, it is impossible to tell if something important changed in a source cell.

Use semi-libraries for long and complex code segments. These are regular python files in the same directory as the notebook. They can be diffed easily.

```
> notebooks/myFolder
| gatherData.ipynb
| libStuff.py
-
```

In "libStuff.py":

```
def squareIt(x):
    return x ** 2
```

In "gatherData.ipynb":

```
from libStuff import squareIt
y = squareIt(3)
```

The merge scenario

You have branches `development` and `cool-feature`, and you want to merge `cool-feature` into `development`. Both have lots of notebooks with outputs, possibly with corrupted first lines.

Preliminaries

`nbstripout` is in your venv, so activate the venv. Later, when we install the filter, it expects a clean attributes file.

```
source venv/bin/activate
rm .git/info/attributes <<don't have to do this every time>>
```

You should have a good file editor (Sublime) ready for lots of conflicts happening within unreadable (in multiple senses) `.ipynb` files. You will need some kind of "Find All Within Project." Have it going on your local machine with an SSHFS.

Be aware of the `cleannbline` script. Sometimes non-JSON and *non-unicode* characters get into the first line, making them unreadable for everything. This script cleans them.

Process

Create a test branch for merge

```
git checkout -b test_merge_cool-feature-into-development
```

Activate your filter

```
nbstripout --install
cat .git/info/attributes
```

should produce an output that looks like this

```
*.ipynb filter=nbstripout
*.ipynb diff=ipynb
```

Strip the notebooks on test branch

Run

```
git status
```

It takes some time. What is that error? It means that some of the notebooks are not valid JSON and cannot be parsed by the `nbstripout` filter.

In the crash log, it should point to a certain file, let's say `notebooks/Test.ipynb` First, clean it with

```
./cleannbline notebooks/Test.ipynb
```

Then, open that file in Sublime and search for <<<<. Sometimes conflicts in your stash can get hidden in a way that does not show up in Jupyter. `nbstripout` will crash. You can find it in Sublime.

Return to running `git status` until it completes without error. It should show a ton of modifications: those are the effects of stripping. Add those and commit

```
git add .
git commit -m "stripped notebooks for merge"
```

Strip the notebooks on cool-feature branch

Your filter is currently active, so when you try

```
git checkout cool-feature
```

it will automatically crash. As above though, it will point to a file. Keep going until `git status` completes. Add those and commit.

Side note: even though `git status` shows a ton of modifications, you should get a clean `git diff` (Although sometimes it will just crash, NBD). Both commands are applying the `.ipynb` filter... in some way.

Do the merge

```
git checkout test_merge_cool-feature-into-development
git merge cool-feature
```

You will get conflicts in two categories: notebooks and other. Since there are <<<< conflict markers everywhere, your `git diff` will crash while you're in the merge. It also doesn't point you to an offending file. Here is where you'll really appreciate Sublime.

Make sure Sublime opens the entire `notebooks` *directory*. That way Find All will search all the files.

1. Pick one file, let's say `notebooks/Test2.ipynb`
2. You might have to `./cleannbline notebooks/Test2.ipynb`
3. In sublime, fix all instances of <<<<, which are usually
    - Minor version changes or metadata stuff
    - Legitimate conflicts
4. When you are satisfied, go back and `git add notebooks/Test2.ipynb`

Repeat for all the notebooks. Then do the same for all the regular code files. When you run `git status` and everything is green, you are done. End the merge with

```
git commit
```

If for some reason, you want to abandon the merge while keeping the test_merge branch stripped, you can run `git reset --hard`

Finalize

Double check that everything went well (i.e. open some notebooks in Jupyter). If something screwed up in your merging *or* stripping, you can just delete the test_merge branch and start over.

Now we're going to make changes to the real `development` branch.

```
git checkout development
```

This will take a while. If it causes crashes, do the thing above to make sure all notebooks are valid JSON until you get a successful `git status`. Make a commit on the *real* branch

```
git add .
git commit -m "stripped notebooks from target branch"
git merge test_merge_cool-feature-into-development
```

This should succeed without conflict.

Cleanup

Remove the test branch

```
git branch -d test_merge_cool-feature-into-development
```

Then you **must** deactivate the filter

```
nbstripout --uninstall
```

Now you can move around the unclean branches without triggering crashes left and right.

While you're at it, leave the venv

```
deactivate
```

Some additional notes on the filter:

When you have the filter active and checkout a normal branch, it will checkout AND strip the outputs in git's mind (not the HEAD version though… confusing)

When you have the filter active and leave a branch that has outputs, it will generate changes, thereby not allowing you to checkout without committing changes

You can turn it on and off with the `nbstripout --install`, `nbstripout --uninstall` commands, as long as the attributes file has nothing else in it This is the easiest way to check: `cat .git/info/attributes`

2.5.2 Command-line tools

These are installed with lightlab.

lightlab config

The `lightlab config` tool manipulates an ini-style file that contains some configuration information for lightlab. This file can be stored in `/usr/local/etc/lightlab.conf` and/or `~/.lightlab/config.conf`. Values defined in the second overrides the first, which in turn overrides default values.

Here's how to use:

```
$ lightlab config

usage: lightlab config [-h] [--system] [command] ...

positional arguments:
  command     write-default: write default configuration
              get [a.b [a2.b2]]: get configuration values
              set a.b c: set configuration value
              reset a[.b]: unset configuration value
  params

optional arguments:
  -h, --help  show this help message and exit
  --system    manipulate lightlab configuration for all users. run as root.
$ lightlab config get # reads all variables
labstate.filepath: ~/.lightlab/labstate.json

$ lightlab config set labstate.filepath ~/.lightlab/newpath.json
----saving /Users/tlima/.lightlab/config.conf----
[labstate]
filepath = /Users/tlima/.lightlab/newpath.json

-------------------------------------------------
$ lightlab config set labstate.filepath '~/.lightlab/newpath.json'
----saving /Users/tlima/.lightlab/config.conf----
[labstate]
filepath = ~/.lightlab/newpath.json

-------------------------------------------------
$ lightlab config get
labstate.filepath: ~/.lightlab/newpath.json

$ lightlab config --system get
labstate.filepath: ~/.lightlab/labstate.json

$ lightlab config reset labstate # could be labstate.filepath
labstate.* reset.
----saving /Users/tlima/.lightlab/config.conf----
-------------------------------------------------

$ lightlab config get
labstate.filepath: ~/.lightlab/labstate.json

#### Interesting for server configurations

$ lightlab config --system set labstate.filepath '/usr/local/etc/lightlab/labstate-
↪system.json'
Write permission to /usr/local/etc/lightlab.conf denied. You cannot save. Try again␣
↪with sudo.

$ sudo lightlab config --system set labstate.filepath '/usr/local/etc/lightlab/
↪labstate-system.json'
Password:
----saving /usr/local/etc/lightlab.conf----
[labstate]
filepath = /usr/local/etc/lightlab/labstate-system.json
```

(continues on next page)

(continued from previous page)

```
------------------------------------------

$ lightlab config get
labstate.filepath: /usr/local/etc/lightlab/labstate-system.json
```

2.5.3 How to set up this sweet documentation workflow

Purely for informing other projects in the future. Users and developers on this project do not have to do any of this. It is setup for you.

1. Install what you need into your virtual environment:

```
$ pip install Sphinx
$ pip install sphinx_rtd_template
$ pip install sphinxcontrib-napoleon
$ pip freeze > requirements.txt
```

2. Set up the sphinx project:

```
$ sphinx-quickstart
```

3. Advanced configure within the `conf.py` file

   - Specify extensions. I use these:

```
extensions = ['sphinx.ext.autodoc',
    'sphinx.ext.napoleon',
    'sphinx.ext.todo',
    'sphinx.ext.mathjax',
    'sphinx.ext.ifconfig',
    'sphinx.ext.viewcode']
```

   - Configuration of Napoleon:

```
napoleon_google_docstring = True
napoleon_use_param = True
```

   - Configuration of Autodocumentation:

```
autodoc_member_order = 'bysource'
autoclass_content = 'both'
```

   - Template configuration for readthedocs style:

```
import sphinx_rtd_theme
html_theme = 'sphinx_rtd_theme'
html_theme_path = [sphinx_rtd_theme.get_html_theme_path()]
```

   - Mock up for external modules imported in your code:

```
import sys
from unittest.mock import MagicMock

class Mock(MagicMock):
```

(continues on next page)

(continued from previous page)

```
    @classmethod
    def __getattr__(cls, name):
            return MagicMock()

MOCK_MODULES = ['numpy',
    'matplotlib',
    'matplotlib.pyplot',
    'matplotlib.figure',
    'scipy',
    'scipy.optimize']
sys.modules.update((mod_name, Mock()) for mod_name in MOCK_MODULES)
```

4. Further documentation here
    - Sphinx overview
    - ReST primer
    - Napoleon

- genindex
- modindex
- search

- genindex
- modindex
- search

CHAPTER 3

API

3.1 lightlab package

Submodules:

3.1.1 lightlab.command_line module

Summary

Functions:

labstate_main	
main	

Data:

version	str(object='') -> str str(bytes_or_buffer[, encoding[, errors]]) -> str

Reference

main()

labstate_main(*args*)

Subpackages:

3.1.2 lightlab.equipment package

Subpackages:

lightlab.equipment.abstract_drivers package

Submodules:

lightlab.equipment.abstract_drivers.TekScopeAbstract module

Summary

Classes:

TekScopeAbstract	General class for several Tektronix scopes, including

Reference

class TekScopeAbstract(*headerIsOptional=True*, *verboseIsOptional=False*, *precedingColon=True*, *interveningSpace=True*, ***kwargs*)

Bases: *lightlab.equipment.abstract_drivers.configurable.Configurable*, *lightlab.equipment.abstract_drivers.AbstractDriver*

General class for several Tektronix scopes, including

- DPO 4034
- DPO 4032
- DSA 8300
- TDS 6154C

The main method is *acquire()*, which takes and returns a `Waveform`.

Todo: These behave differently. Be more explicit about sample mode:

```
timebaseConfig(avgCnt=1)
acquire([1])

acquire([1], avgCnt=1)
```

Does DPO support sample mode at all?

totalChans = None

startup()

timebaseConfig(*avgCnt=None*, *duration=None*, *position=None*, *nPts=None*)

Timebase and acquisition configure

Parameters

- **avgCnt** (*int*) – averaging done by the scope

- **duration** (*float*) – time, in seconds, for data to be acquired
- **position** (*float*) – trigger delay
- **nPts** (*int*) – number of samples taken

Returns (dict) The present values of all settings above

acquire(*chans=None*, *timeout=None*, ***kwargs*)
Get waveforms from the scope.

If chans is None, it won't actually trigger, but it will configure.

If unspecified, the kwargs will be derived from the previous state of the scope. This is useful if you want to play with it in lab while working with this code too.

Parameters

- **chans** (*list*) – which channels to record at the same time and return
- **avgCnt** (*int*) – number of averages. special behavior when it is 1
- **duration** (*float*) – window width in seconds
- **position** (*float*) – trigger delay
- **nPts** (*int*) – number of sample points
- **timeout** (*float*) – time to wait for averaging to complete in seconds If it is more than a minute, it will do a test first

Returns recorded signals

Return type list[*Waveform*]

wfmDb(*chan*, *nWfms*, *untriggered=False*)
Transfers a bundle of waveforms representing a signal database. Sample mode only.

Configuration such as position, duration are unchanged, so use an acquire(None, . . .) call to set them up

Parameters

- **chan** (*int*) – currently this only works with one channel at a time
- **nWfms** (*int*) – how many waveforms to acquire through sampling
- **untriggered** (*bool*) – if false, temporarily puts scope in free run mode

Returns all waveforms acquired

Return type (*FunctionBundle*(*Waveform*))

run(*continuousRun=True*)
Sets the scope to continuous run mode, so you can look at it in lab, or to single-shot mode, so that data can be acquired

Parameters **continuousRun** (*bool*) –

setMeasurement(*measIndex*, *chan*, *measType*)

Parameters

- **measIndex** (*int*) – used to refer to this measurement itself. 1-indexed
- **chan** (*int*) – the channel source of the measurement.
- **measType** (*str*) – can be 'PK2PK', 'MEAN', etc.

measure(*measIndex*)

Parameters **measIndex** (*int*) – used to refer to this measurement itself. 1-indexed

Returns (float)

autoAdjust (*chans*)
Adjusts offsets and scaling so that waveforms are not clipped

lightlab.equipment.abstract_drivers.configurable module

Summary

Exceptions:

AccessException

Classes:

Configurable	Instruments can be configurable to keep track of settings within the instrument
TekConfig	Wraps a dictionary attribute.

Reference

exception AccessException
Bases: Exception

class TekConfig (*initDict=None*)
Bases: object

Wraps a dictionary attribute. Uses dpath for operations.

Commands are defined as tuples (cStr, val). For example (':PATH:TO:CMD', 4). Use these by doing scope.write(' '.join(TekConfig.get('PATH:TO:CMD'))) The val is always a string.

Todo: :transferring subgroup from one instance to another. :returning a dictionary representing a subgroup (actually this might currently be happening in error) :transferring subgroup values to a different subgroup in the same instance (for example, CH1 to CH2)

separator = ':'

print (*subgroup=''*)

copy (*subgroup=''*)

get (*cStr, asCmd=True*)
Returns the value only, not a dictionary

Parameters **asCmd** (*bool*) – if true, returns a tuple representing a command. Otherwise returns just the value

set (*cStr, val*)
Takes the value only, not a dictionary

getList (*subgroup='', asCmd=True*)
Deep crawler that goes in and generates a command for every leaf.

Parameters

- **subgroup** (*str*) – subgroup must be a subdirectory. If '', it is root directory. It can also be a command string, in which case, the returned list has length 1
- **asCmd** (*bool*) – if false, returns a list of strings that can be sent to scopes

Returns list of valid commands (cstr, val) on the subgroup subdirectory

Return type list

setList (*cmdList*)
The inverse of getList

transfer (*source*, *subgroup=''*)
Pulls config from the source TekConfig object. This is useful for subgrouping.

For example, you might want to load from default only the trigger configuration.

Parameters

- **source** (TekConfig *or* dict) – the object from which config values are pulled into self
- **subgroup** (*str*) – subgroup must be a subdirectory. If '', it is root directory. It can also be a command string, in which case, only that parameter is affected

classmethod fromFile (*fname*, *subgroup=''*)

classmethod fromSETresponse (*setResponse*, *subgroup=''*)
setResponse (str): what is returned by the scope in response to query('SET?')

It will require some parsing for subgroup shorthand

save (*fname*, *subgroup=''*, *overwrite=False*)
Saves dictionary parameters in json format. Merges if there's something already there, unless overwrite is True.

Parameters

- **fname** (*str*) – file name
- **subgroup** (*str*) – groups of commands to write. If '', it is everything.
- **overwrite** (*bool*) – will make a new file exactly corresponding to this instance, otherwise merges with existing

class Configurable (*headerIsOptional=True*, *verboseIsOptional=False*, *precedingColon=True*, *interveningSpace=True*, ***kwargs*)
Bases: *lightlab.equipment.abstract_drivers.AbstractDriver*

Instruments can be configurable to keep track of settings within the instrument

This class is setup so that the hardware state is reflected exactly in the 'live' config **unless somebody changes something in lab**. Watch out for that and use `forceHardware` if that is a risk

This clas uses query/write methods that are not directly inherited, so the subclass or its parents must implement those functions

config = None
Dictionary of *TekConfig* objects.

initHardware ()
Runs upon first hardware access. Tells the instrument how to format its commands

`setConfigParam`(*cStr*, *val=None*, *forceHardware=False*)
Sets an individual configuration parameter. If the value has been read before, and there is no change, then it will **not** write to the hardware.

Parameters

- **`cStr`** (*`str`*) – name of the command
- **`val`** (*`any`*) – value to send. Detects type, so if it's an int, it will be stored as int
- **`forceHardware`** (*`bool`*) – will always send to hardware, in case it is critical or if it tends to be changed by pesky lab users

Returns Did it requre a write to hardware?

Return type (bool)

`getConfigParam`(*cStr*, *forceHardware=False*)
Gets a single parameter. If the value has been read before, and there is no change, then it will **not** query the hardware.

This is much faster than getting from hardware; however, it assumes that nobody in lab touched anything.

Parameters

- **`cStr`** (*`str`*) – name of the command
- **`forceHardware`** (*`bool`*) – will always query from hardware, in case it is critical or if it tends to be changed by pesky lab users

Returns command value. Detects type, so that `'2.5'` will return as `float`

Return type (any)

If the command is not recognized, attempts to get it from hardware

`tempConfig`(*cStr*, *tempVal*, *forceHardware=False*)
Changes a parameter within the context of a "with" block. Args are same as in *`getConfigParam()`*.

`getDefaultFilename`()
Combines the `lightlab.util.io.paths.defaultFileDir` with the *IDN? string of this instrument.

Returns the default filename

Return type (str)

`saveConfig`(*dest='+user'*, *subgroup="*, *overwrite=False*)
If you would like to setup a temporary state (i.e. taking some measurements and going back), use a file and *subgroup=*

Parameters **`subgroup`** (*`str`*) – a group of commands or a single command. If '', it means everything.

Side effects: if dest is object or dict, modifies it if dest is token, modifies the config library of self if dest is filename, writes that file

`loadConfig`(*source='+user'*, *subgroup="*)
Loads some configuration parameters from a source which is either:

- a file name string, or
- a special token ['+default' or '+init'], or
- some TekConfig object or dict you have out there

Parameters

- **source** (*str/TekConfig*) – load source
- **subgroup** (*str*) – a group of commands or a single command. If '', it means everything.

generateDefaults (*filename=None*, *overwrite=False*)
Attempts to read every configuration parameter. Handles several cases where certain parameters do not make sense and must be skipped

Generates a new default file which is saved in configurable.defaultFileDir

This takes a while.

Parameters

- **filename** (*str*) – simple name. You can't control the directory.
- **overwrite** (*bool*) – If False, stops if the file already exists.

lightlab.equipment.abstract_drivers.electrical_sources module

Summary

Classes:

MultiChannelSource	This thing basically holds a dict state and provides some critical methods
MultiModalSource	Checks modes for sources with multiple ways to specify.

Reference

class MultiModalSource
Bases: object

Checks modes for sources with multiple ways to specify.

Also checks ranges

Default class constants come from NI PCI source array

supportedModes = {'milliamp', 'baseunit', 'mwperohm', 'volt', 'amp', 'wattperohm'}

baseUnitBounds = [0, 1]

baseToVoltCoef = 10

v2maCoef = 4

exceptOnRangeError = False

classmethod enforceRange (*val*, *mode*)
Returns clipped value. Raises RangeError

classmethod val2baseUnit (*value*, *mode*)
Converts to the voltage value that will be applied at the PCI board Depends on the current mode state of the instance

Args: value (float or dict)

classmethod baseUnit2val(*baseVal*, *mode*)

Converts the voltage value that will be applied at the PCI board back into the units of th instance This is useful for bounds checking

Args: baseVal (float or dict)

class MultiChannelSource(*useChans=None*, ***kwargs*)

Bases: `object`

This thing basically holds a dict state and provides some critical methods

There is no mode

Checks for channel compliance. Handles range exceptions

maxChannel = None

elChans

Returns the blocked out channels as a list

setChannelTuning(*chanValDict*)

Sets a number of channel values and updates hardware

Parameters

- **chanValDict** (*dict*) – A dictionary specifying {channel: value}
- **waitTime** (*float*) – time in ms to wait after writing, default (None) is defined in the class

Returns was there a change in value

Return type (bool)

getChannelTuning()

The inverse of setChannelTuning

Parameters mode (*str*) – units of the value in ('mwperohm', 'milliamp', 'volt')

Returns the full state of blocked out channels in units determined by mode argument

Return type (dict)

off(**setArgs*)

Turn all voltages to zero, but maintain the session

lightlab.equipment.abstract_drivers.multimodule_configurable module

Summary

Classes:

ConfigModule	A module that has an associated channel and keeps track of parameters within that channel.
MultiModuleConfigurable	Keeps track of a list of `Configurable` objects, each associated with a channel number.

Reference

class ConfigModule(*channel*, *bank*, ***kwargs*)
Bases: `lightlab.equipment.abstract_drivers.configurable.Configurable`

A module that has an associated channel and keeps track of parameters within that channel. Updates only when changed or with `forceHardware`. It communicates with a bank instrument of which it is a part. When it writes to hardware, it selects itself by first sending `'CH 2'` (if it were initialized with channel 2)

Parameters

- **channel** (*int*) – its channel that will be written before writing/querying
- **bank** (`MultiModuleConfigurable`) – enclosing bank

selectPrefix = 'CH'

write(*writeStr*)
Regular write in the enclosing bank, except preceded by select self

query(*queryStr*)
Regular query in the enclosing bank, except preceded by select self

class MultiModuleConfigurable(*useChans=None*, *configModule_klass=<class 'lightlab.equipment.abstract_drivers.configurable.Configurable'>*, ***kwargs*)
Bases: `lightlab.equipment.abstract_drivers.AbstractDriver`

Keeps track of a list of `Configurable` objects, each associated with a channel number. Provides array and dict setting/getting.

Parameter values are cached just like in `Configurable`. That means hardware access is lazy: No write/queries are performed unless a parameter is not yet known, or if the value changes.

When the module classes are `ConfigModule`, then this supports multi-channel instruments where channels are selectable. This is used in cases where, for example, querying the wavelength of channel 2 would take these messages:

```
self.write('CH 2')
wl = self.query('WAVE')
```

Parameters

- **useChans** (*list (int)*) – integers that indicate channel number.
- **to key dictionaries and write select messages.** (*Used*) –
- **configModule_klass** (*type*) – class that members will be initialized as.
- **Configurable, this object is basically a container; however,** (*When*) –
- **ConfigModule, there is special behavior for multi-channel instruments.** (*when*) –

maxChannel = None

getConfigArray(*cStr*)
Iterate over modules getting the parameter at each

Parameters **cStr** (*str*) – parameter name

Returns values for all modules, ordered based on the ordering of `useChans`

Return type (np.ndarray)

setConfigArray(*cStr*, *newValArr*, *forceHardware=False*)
Iterate over modules setting the parameter to the corresponding array value.

Values for *ALL* channels must be specified. To only change some, use the dictionary-based setter: `setConfigDict`

Parameters

- **cStr** (*str*) – parameter name
- **newValArr** (*np.ndarray, list*) – values in same ordering as useChans
- **forceHardware** (*bool*) – guarantees sending to hardware

Returns did any require hardware write?

Return type (bool)

getConfigDict(*cStr*)

Parameters **cStr** (*str*) – parameter name

Returns parameter on all the channels, keyed by channel number

Return type (dict)

setConfigDict(*cStr*, *newValDict*, *forceHardware=False*)

Parameters

- **cStr** (*str*) – parameter name
- **newValDict** (*array*) – dict keyed by channel number
- **forceHardware** (*bool*) – guarantees sending to hardware

Returns did any require hardware write?

Return type (bool)

moduleIds
list of module ID strings

lightlab.equipment.abstract_drivers.power_meters module

Summary

Classes:

PowerMeterAbstract	For the HP_8152A and the Advantest_Q8221

Reference

class PowerMeterAbstract
Bases: *lightlab.equipment.abstract_drivers.AbstractDriver*

For the HP_8152A and the Advantest_Q8221

channelDescriptions = {1: 'A', 2: 'B', 3: 'A/B'}

validateChannel(*channel*)
Raises an error with info if not a valid channel

powerLin(*channel=1*)

Summary

Classes:

`AbstractDriver`	In case there is future functionality

Reference

class AbstractDriver
Bases: `object`

In case there is future functionality

lightlab.equipment.lab_instruments package

Submodules:

lightlab.equipment.lab_instruments.Advantest_Q8221_PM module

Summary

Classes:

`Advantest_Q8221_PM`	Q8221 Optical Multi-power Meter

Reference

class Advantest_Q8221_PM(*name='The Advantest power meter'*, *address=None*, ***kwargs*)
Bases: `lightlab.equipment.visa_bases.visa_driver.VISAInstrumentDriver`, `lightlab.equipment.abstract_drivers.power_meters.PowerMeterAbstract`

Q8221 Optical Multi-power Meter

Manual

Usage: *Instrument: PowerMeter*

instrument_category
alias of `lightlab.laboratory.instruments.interfaces.PowerMeter`

channelDescriptions = {1: 'A', 2: 'B', 3: 'A/B'}

startup()
Behaves the same as super.

Todo: Read manual and set the channels to DBM and default channel to A

- Default read: `"DBA-054.8686E+00\r\n"`
- query("CH1"): `"DBB-054.8686E+00\r\n"`

`open`()

`powerDbm`(*channel=1*)
The detected optical power in dB on the specified channel

Parameters **`channel`** (*`int`*) – Power Meter channel

Returns Power in dB or dBm

Return type (double)

lightlab.equipment.lab_instruments.Agilent_33220_FG module

Summary

Classes:

`Agilent_33220_FG`	Function Generator

Reference

`class Agilent_33220_FG`(*name='Agilent synth'*, *address=None*, ***kwargs*)
Bases: *`lightlab.equipment.visa_bases.visa_driver.VISAInstrumentDriver`*, *`lightlab.equipment.abstract_drivers.configurable.Configurable`*

Function Generator

Manual

Usage: *Instrument: FunctionGenerator*

`instrument_category`
alias of *`lightlab.laboratory.instruments.interfaces.FunctionGenerator`*

`amplitudeRange = (0.01, 10)`

`startup`()

`enable`(*enaState=None*)

`frequency`(*newFreq=None*)

`waveform`(*newWave=None*)
Available tokens are (with optional part in brackets): 'dc', 'sin[usoid]', 'squ[are]', 'ramp', 'puls[e]', 'nois[e]', 'user'

`setArbitraryWaveform`(*wfm*)
Arbitrary waveform

Todo: implement

`amplAndOffs`(*amplOffs=None*)
Amplitude and offset setting/getting

Only uses the data-bar because the other one is broken

Parameters

- **`amplOffs`** (*`tuple(float)`*) – new amplitude (p2p) and offset in volts
- **`either is None, returns but does not set`** (*`If`*) –

Returns amplitude and offset, read from hardware if specified as None

Return type (tuple(float))

Critical: Offset control is not working. Some sort of dictionary conflict in 'VOLT'

`duty`(*duty=None*)

duty is in percentage. For ramp waveforms, duty is the percent of time spent rising.

Critical: Again, this is having dpath troubles.

lightlab.equipment.lab_instruments.Agilent_83712B_clock module

Summary

Classes:

`Agilent_83712B_clock`	Where is manual?

Reference

`class Agilent_83712B_clock`(*name='The clock on PPG'*, *address=None*, ***kwargs*)

Bases: *`lightlab.equipment.visa_bases.visa_driver.VISAInstrumentDriver`*, *`lightlab.equipment.abstract_drivers.configurable.Configurable`*

Where is manual?

Usage: *Instrument: Clock*

`instrument_category`

alias of *`lightlab.laboratory.instruments.interfaces.Clock`*

`startup`()

`enable`(*enaState=None*)

`frequency`

lightlab.equipment.lab_instruments.Agilent_N5183A_VG module

Summary

Classes:

`Agilent_N5183A_VG`	Agilent N5183A Vector Generator

Reference

class Agilent_N5183A_VG(*name='The 40GHz clock'*, *address=None*, ***kwargs*)

Bases: `lightlab.equipment.visa_bases.visa_driver.VISAInstrumentDriver`, `lightlab.equipment.abstract_drivers.configurable.Configurable`

Agilent N5183A Vector Generator

Manual

Usage: *Instrument: Clock*

Todo: Clock interface does not see sweepSetup and sweepEnable

instrument_category

alias of `lightlab.laboratory.instruments.interfaces.Clock`

amplitude(*amp=None*)

Amplitude is in dBm

Parameters **amp** (*float*) – If None, only gets

Returns output power amplitude

Return type (float)

frequency(*freq=None*)

Frequency is in Hertz

Setting the frequency takes you out of sweep mode automatically

Parameters **freq** (*float*) – If None, only gets

Returns center frequency

Return type (float)

enable(*enaState=None*)

Enabler for the output

Parameters **enaState** (*bool*) – If None, only gets

Returns is RF output enabled

Return type (bool)

sweepSetup(*startFreq*, *stopFreq*, *nPts=100*, *dwell=0.1*)

Configure sweep. See instrument for constraints; they are not checked here.

Does not auto-enable. You must also call :meth:'sweepEnable'

Parameters

- **startFreq** (*float*) – lower frequency in Hz
- **stopFreq** (*float*) – upper frequency in Hz
- **nPts** (*int*) – number of points
- **dwell** (*float*) – time in seconds to wait at each sweep point

Returns None

sweepEnable(*swpState=None*)

Switches between sweeping (True) and CW (False) modes

Parameters **swpState** (*bool*) – If None, only gets, doesn't set.

Returns is the output sweeping

Return type (bool)

lightlab.equipment.lab_instruments.Agilent_N5222A_NA module

Summary

Classes:

Agilent_N5222A_NA	Agilent PNA N5222A , RF network analyzer

Reference

class Agilent_N5222A_NA(*name='The network analyzer'*, *address=None*, ***kwargs*)

Bases: *lightlab.equipment.visa_bases.visa_driver.VISAInstrumentDriver*, *lightlab.equipment.abstract_drivers.configurable.Configurable*

Agilent PNA N5222A , RF network analyzer

Manual

WARNING: The address is the same as the slow function generator, so don't use both on andromeda at the same time.

Steep learning curve.

Usage: *Instrument: NetworkAnalyzer*

Todo: All the RF equipment is reusing __enaBlock. Make this a method of Configurable.

When setting up general, you have to setup sweep before setting CW frequency, or else the CW freq becomes the start frequency. Why? See hack in sweepSetup.

instrument_category

alias of *lightlab.laboratory.instruments.interfaces.NetworkAnalyzer*

startup()

amplitude(*amp=None*)

Amplitude is in dBm

Parameters **amp** (*float*) – If None, only gets

Returns output power amplitude

Return type (float)

frequency(*freq=None*)

Frequency is in Hertz

Setting the frequency takes you out of sweep mode automatically

Parameters **freq** (*float*) – If None, only gets

Returns center frequency

Return type (float)

enable (*enaState=None*)
Enabler for the entire output

Parameters **enaState** (*bool*) – If None, only gets

Returns is RF output enabled

Return type (bool)

run ()

sweepSetup (*startFreq*, *stopFreq*, *nPts=None*, *dwell=None*, *ifBandwidth=None*)
Configure sweep. See instrument for constraints; they are not checked here.

Does not auto-enable. You must also call :meth:‘sweepEnable‘

Parameters

- **startFreq** (*float*) – lower frequency in Hz
- **stopFreq** (*float*) – upper frequency in Hz
- **nPts** (*int*) – number of points
- **dwell** (*float*) – time in seconds to wait at each sweep point. Default is minimum.

Returns None

sweepEnable (*swpState=None*)
Switches between sweeping (True) and CW (False) modes

Parameters **swpState** (*bool*) – If None, only gets, doesn’t set.

Returns is the output sweeping

Return type (bool)

normalize ()

triggerSetup (*useAux=None*, *handshake=None*, *isSlave=False*)

getSwpDuration (*forceHardware=False*)

measurementSetup (*measType=’S21’*, *chanNum=None*)

spectrum ()

multiSpectra (*nSpect=1*, *livePlot=False*)

lightlab.equipment.lab_instruments.Anritsu_MP1763B_PPG module

Summary

Classes:

Anritsu_MP1763B_PPG	ANRITSU MP1761A PulsePatternGenerator The PPG MP1763B at Alex’s bench, which also support MP1761A (by Hsuan-Tung 07/27/2017)

Reference

class Anritsu_MP1763B_PPG(*name='The PPG'*, *address=None*, ***kwargs*)

Bases: `lightlab.equipment.visa_bases.visa_driver.VISAInstrumentDriver`, `lightlab.equipment.abstract_drivers.configurable.Configurable`

ANRITSU MP1761A PulsePatternGenerator The PPG MP1763B at Alex's bench, which also support MP1761A (by Hsuan-Tung 07/27/2017)

Manual (MP1763C): http://www.otntech.com/modules/catalogue/download.php?id=52&mode=download&file_name=MP1763C.pdf

Usage: *Instrument: PulsePatternGenerator*

instrument_category

alias of `lightlab.laboratory.instruments.interfaces.PulsePatternGenerator`

storedPattern = None

startup()

setPrbs(*length*)

Generates a PRBS

setPattern(*bitArray*)

Data bitArray for the PPG to output.

Parameters **bitArray** (*ndarray*) – array that is boolean or binary 1/0

getPattern()

Inverts the setPattern method, so you can swap several patterns around on the fly. Does not communicate with the hardware as of now.

on(*turnOn=True*)

syncSource(*src=None*)

Output synchronizer is locked to pattern or not?

Parameters **src** (*str*) – either 'fixed', 'variable' or 'clock64'. If None, leaves it

Returns the set value as a string token

Return type (str)

amplAndOffs(*amplOffs=None*)

Amplitude and offset setting/getting

Parameters

- **amplOffs** (*tuple(float)*) – new amplitude and offset in volts
- **either is None, returns but does not set** (*If*) –

Returns amplitude and offset, read from hardware if specified as None

Return type (tuple(float))

bitseq(*chpulses*, *clockfreq*, *ext=0*, *addplot=False*, *mult=1*, *res=5*)

bitseq: Takes in dictionary 'chpulses', clock freq 'clockfreq', and opt. parameter 'ext.' Also includes plotting parameters (see below). chdelays: a dictionary in which keys are channel delays, and values contain a list of tuple pairs. Each pair contains pulse times (rising edges) and their duration (in ns). clockfreq: set the current clock frequency, in GHz ext: a continuous value from 0 to 1 which extends the pattern length, resulting in different synchronization between adjacent time windows. 0 – will result in maximum similarity between time windows, plus or minus variabilities resulting from delay lines. This is ideal when only approximate timings are required, since channels IDs can be shuffled by time scrolling through the same PPG pattern. 1 – will result in minimum similarity between adjacent time windows, at the cost of a larger total PPG pattern length. Anything beyond this value is not useful. Values between 0 and 1 will trade-off pattern length with window similarity. addplot: Adds a plot to visualize the output of the PPG along all channels. mult: graphing parameter - how many multiples of pattern length to display in time res: graphing parameter - how many sampling points per pattern bit Author: Mitchell A. Nahmias, Feb. 2018

classmethod PRBS_pattern(*order*, *mark_ratio=0.5*)

lightlab.equipment.lab_instruments.Apex_AP2440A_OSA module

Summary

Classes:

`Apex_AP2440A_OSA`	Class for the OSA

Functions:

`check_socket`

Data:

`WIDEST_WLRANGE`	list() -> new empty list list(iterable) -> new list initialized from iterable's items

Reference

check_socket(*host*, *port*)

class Apex_AP2440A_OSA(*name='The OSA'*, *address=None*, ***kwargs*)

Bases: `lightlab.equipment.visa_bases.visa_driver.VISAInstrumentDriver`

Class for the OSA

Basic functionality includes setting/getting wavelength range and sweeping Other functionality is for controlling TLS: on/off, wavelength (not implemented)

The primary function is spectrum, which returns a Spectrum object

Usage: *Instrument: OpticalSpectrumAnalyzer*

Initializes a fake VISA connection to the OSA.

instrument_category

alias of `lightlab.laboratory.instruments.interfaces.OpticalSpectrumAnalyzer`

MAGIC_TIMEOUT = 30

reinstantiate_session(*address*, *tempSess*)

startup()

Checks if it is alive, sets up standard OSA parameters

open()

close()

query(*queryStr*, *expected_talker=None*)

write(*writeStr*, *expected_talker=None*)
The APEX does not deal with write; you have to query to clear the buffer

instrID()
Overloads the super function because the OSA does not respond to *IDN? Instead sends a simple command and waits for a confirmed return

getWLrangeFromHardware()

wlRange

triggerAcquire()
Performs a sweep and reads the data Returns an array of dBm values as doubles :rtype: array

transferData()
Performs a sweep and reads the data

Gets the data of the sweep from the spectrum analyzer

Returns wavelength in nm, power in dBm

Return type (ndarray, ndarray)

spectrum(*average_count=1*)
Take a new sweep and return the new data. This is the primary user function of this class

tlsEnable

tlsWl

lightlab.equipment.lab_instruments.Arduino_Instrument module

Summary

Classes:

Arduino_Instrument	Read/write interface for an arduino.

Reference

class Arduino_Instrument(*name='Arduino'*, ***kwargs*)
Bases: *lightlab.equipment.visa_bases.visa_driver.VISAInstrumentDriver*

Read/write interface for an arduino. Could make use of TCPIP or maybe USB

Usage: TODO

Todo: To be implemented.

instrument_category
alias of *lightlab.laboratory.instruments.interfaces.ArduinoInstrument*

write(*writeStr*)

query (*queryStr*, *withTimeout=None*)

lightlab.equipment.lab_instruments.HP_8116A_FG module

Summary

Classes:

HP_8116A_FG	Function Generator

Reference

class HP_8116A_FG (*name='The slow synth (FUNCTION GENERATOR)'*, *address=None*, ***kwargs*)

Bases: *lightlab.equipment.visa_bases.visa_driver.VISAInstrumentDriver*, *lightlab.equipment.abstract_drivers.configurable.Configurable*

Function Generator

Manual

Usage: *Instrument: FunctionGenerator*

instrument_category
alias of *lightlab.laboratory.instruments.interfaces.FunctionGenerator*

amplitudeRange = (0.01, 10)

startup ()

instrID ()

enable (*enaState=None*)

frequency (*newFreq=None*)

waveform (*newWave=None*)
Available tokens are 'dc', 'sine', 'triangle', 'square', 'pulse'

amplAndOffs (*amplOffs=None*)
Amplitude and offset setting/getting

Only uses the data-bar because the other one is broken

Parameters

- **amplOffs** (*tuple (float)*) – new amplitude and offset in volts
- **either is None, returns but does not set** (*If*) –

Returns amplitude and offset, read from hardware if specified as None

Return type (tuple(float))

duty (*duty=None*)
duty is in percentage

lightlab.equipment.lab_instruments.HP_8152A_PM module

Summary

Classes:

`HP_8152A_PM`	HP8152A power meter

Reference

class HP_8152A_PM(*name='The HP power meter'*, *address=None*, ***kwargs*)

Bases: `lightlab.equipment.visa_bases.visa_driver.VISAInstrumentDriver`, `lightlab.equipment.abstract_drivers.power_meters.PowerMeterAbstract`

HP8152A power meter

Manual

Usage: *Instrument: PowerMeter*

Todo: Maybe allow a rapid continuous mode that just spits out numbers ('T0')

instrument_category

alias of `lightlab.laboratory.instruments.interfaces.PowerMeter`

channelDescriptions = {1: 'A', 2: 'B', 3: 'A/B'}

doReadDoubleCheck = False

startup()

open()

static proccessWeirdRead(*readString*)

The HP 8152 *sometimes* sends double characters. This tries to fix it based on reasonable value ranges.

We assume that the values encountered have a decimal point and have two digits before and after the decimal point

Arg: readString (str): what is read from query('TRG')

Returns checked string

Return type (str)

robust_query(**args*, ***kwargs*)

Conditionally check for read character doubling

powerDbm(*channel=1*)

The detected optical power in dB on the specified channel

Parameters **channel** (`int`) – Power Meter channel

Returns Power in dB or dBm

Return type (double)

lightlab.equipment.lab_instruments.HP_8156A_VA module

Summary

Classes:

HP_8156A_VA	HP8156A variable attenuator

Reference

class HP_8156A_VA(*name='The VOA on the GC bench'*, *address=None*, ***kwargs*)

Bases: *lightlab.equipment.visa_bases.visa_driver.VISAInstrumentDriver*

HP8156A variable attenuator

Manual

Usage: *Instrument: VariableOpticalAttenuator*

instrument_category

alias of *lightlab.laboratory.instruments.interfaces.VariableAttenuator*

safeSleepTime = 1

startup()

on()

off()

setAtten(*val*, *isLin=True*)

Simple method instead of property access

attenDB

attenLin

sendToHardware(*sleepTime=None*)

wavelength

calibration

lightlab.equipment.lab_instruments.HP_8157A_VA module

Summary

Classes:

HP_8157A_VA	HP8157A variable attenuator

Reference

class HP_8157A_VA(*name='The VOA on the Minerva bench'*, *address=None*, ***kwargs*)

Bases: *lightlab.equipment.visa_bases.visa_driver.VISAInstrumentDriver*

HP8157A variable attenuator

Manual

Usage: *Instrument: VariableOpticalAttenuator*

instrument_category
alias of `lightlab.laboratory.instruments.interfaces.VariableAttenuator`

safeSleepTime = 1

startup()

on()

off()

setAtten(*val*, *isLin=True*)
Simple method instead of property access

attenDB

attenLin

sendToHardware(*sleepTime=None*)

calibration
Calibration compensates for the insertion loss of the instruments.

wavelength

lightlab.equipment.lab_instruments.ILX_7900B_LS module

Summary

Classes:

`ILX_7900B_LS`	The laser banks (ILX 7900B laser source).
`ILX_Module`	Handles 0 to 1 indexing

Reference

class ILX_Module(*channel*, ***kwargs*)
Bases: `lightlab.equipment.abstract_drivers.multimodule_configurable.ConfigModule`

Handles 0 to 1 indexing

class ILX_7900B_LS(*name='The laser source'*, *address=None*, *useChans=None*, ***kwargs*)
Bases: `lightlab.equipment.visa_bases.visa_driver.VISAInstrumentDriver`, `lightlab.equipment.abstract_drivers.multimodule_configurable.MultiModuleConfigurable`

The laser banks (ILX 7900B laser source). Provides array-based and dict-based setters/getters for

- whether laser is on or off (`enableState`)
- tunable wavelength output (`wls`)
- output power in dBm (`powers`)

Setting/getting logic is implemented in `MultiModuleConfigurable`, which treats the channels as independent `ConfigModules`'s. This means that hardware communication is lazy – parameter values are cached, and messages are only sent when they are unknown or when they change.

Manual

Usage: *Instrument: LaserSource*

Todo: Multiple users at the same time is desirable. We are close. Non blocked-out channels are never touched, but there are still two issues

- Fundamental: VISA access with two python processes could collide
- **Inconvenience: Have to create two different labstate instruments** with different `useChans` for what is actually one instrument – maybe a slice method?

`instrument_category`
alias of *`lightlab.laboratory.instruments.interfaces.LaserSource`*

`maxChannel = 8`

`sleepOn = {'LEVEL': 5, 'OUT': 3, 'WAVE': 30}`

`powerRange = <MagicMock name='mock()' id='140061893981576'>`

`startup`()

`dfbChans`
Returns the blocked out channels as a list

Currently, this is not an essentialProperty, so you have to access like:

```
ch = LS.driver.dfbChans
```

Returns channel numbers, 0-indexed

Return type (list)

`setConfigArray`(*cStr*, *newValArr*, *forceHardware=False*)
When any configuration is set, there is an equilibration time.

This adds sleep functionality, only when there is a change, for an amount determined by the `sleepOn` class attribute.

`enableState`
**Returns* – (np.ndarray)* – enable states ordered like useChans

`setChannelEnable`(*chanEnableDict*)
Sets only some channel values with dict keyed by useChans, e.g. `chanEnableDict={0: 1, 2: 0}`

Parameters **`chanEnableDict`** (*`dict`*) – A dictionary keyed by channel with values 0 or 1

`getChannelEnable`()

Returns all channel enable states, keyed by useChans

Return type (dict)

`wls`
**Returns* – (np.ndarray)* – laser wavelengths in nanometers ordered like useChans

setChannelWls(*chanWavelengthDict*)

Sets only some channel values with dict keyed by useChans, e.g. `chanWavelengthDict={0: 1550.5, 2: 1551}`

Parameters **chanWavelengthDict** (*dict*) – A dictionary keyed by channel with nanometer values

getChannelWls()

Returns all channel wavelengths, keyed by useChans

Return type (dict)

powers

Laser powers

Returns laser output powers in dBm, ordered like useChans

Return type (np.ndarray)

setChannelPowers(*chanPowerDict*)

Sets only some channel values with dict keyed by useChans, e.g. `chanPowerDict={0: 13, 2: -10}`

Parameters **chanPowerDict** (*dict*) – A dictionary keyed by channel with dBm values

getChannelPowers()

Returns all channel powers, keyed by useChans

Return type (dict)

wlRanges

Min/max wavelengths than can be covered by each channl. Wavelengths in nm.

Returns maximum ranges starting from lower wavelength

Return type (list(tuple))

getAsSpectrum()

Gives a spectrum of power vs. wavelength, which has the wavelengths present as an abscissa, and their powers as ordinate (-120dBm if disabled)

It starts in dBm, but you can change to linear with the Spectrum.lin method

Returns The WDM spectrum of the present outputs

Return type (*Spectrum*)

allOff()

allOn()

off()

lightlab.equipment.lab_instruments.Keithley_2400_SM module

Summary

Classes:

Keithley_2400_SM	A Keithley 2400 driver.

Reference

class Keithley_2400_SM(*name=None*, *address=None*, ***kwargs*)

Bases: `lightlab.equipment.visa_bases.visa_driver.VISAInstrumentDriver`, `lightlab.equipment.abstract_drivers.configurable.Configurable`

A Keithley 2400 driver.

Manual:

Usage: *Instrument: Keithley and SourceMeter*

Capable of sourcing current and measuring voltage, such as a Keithley

Also provides interface methods for measuring resistance and measuring power

Parameters

- **currStep** (`float`) – amount to step if ramping in current mode. Default (None) is no ramp
- **voltStep** (`float`) – amount to step if ramping in voltage mode. Default (None) is no ramp
- **rampStepTime** (`float`) – time to wait on each ramp step point

instrument_category
alias of `lightlab.laboratory.instruments.interfaces.Keithley`

autoDisable = None

currStep = None

voltStep = None

rampStepTime = 0.01

startup()

setPort(*port*)

setVoltageMode(*protectionCurrent=0.05*)

setCurrentMode(*protectionVoltage=1*)

setCurrent(*currAmps*)
This leaves the output on indefinitely

setVoltage(*voltVolts*)

getCurrent()

getVoltage()

setProtectionVoltage(*protectionVoltage*)

setProtectionCurrent(*protectionCurrent*)

protectionVoltage

protectionCurrent

measVoltage()

measCurrent()

enable(*newState=None*)
get/set enable state

lightlab.equipment.lab_instruments.Keithley_2606B_SMU module

Driver class for Keithley 2606B.

The following programming example illustrates the setup and command sequence of a basic source-measure procedure with the following parameters: • Source function and range: voltage, autorange • Source output level: 5 V • Current compliance limit: 10 mA • Measure function and range: current, 10 mA

– Restore 2606B defaults. smua.reset() – Select voltage source function. smua.source.func = smua.OUTPUT_DCVOLTS – Set source range to auto. smua.source.autorangev = smua.AUTORANGE_ON – Set voltage source to 5 V. smua.source.levelv = 5 – Set current limit to 10 mA. smua.source.limiti = 10e-3 – Set current range to 10 mA. smua.measure.rangei = 10e-3 – Turn on output. smua.source.output = smua.OUTPUT_ON – Print and place the current reading in the reading buffer. print(smua.measure.i(smua.nvbuffer1)) – Turn off output. smua.source.output = smua.OUTPUT_OFF

Summary

Classes:

Keithley_2606B_SMU	Keithley 2606B 4x SMU instrument driver

Reference

class Keithley_2606B_SMU(*name=None*, *address=None*, *tsp_node: int = None*, *channel: str = None*, ***visa_kwargs*)

Bases: *lightlab.equipment.visa_bases.visa_driver.VISAInstrumentDriver*

Keithley 2606B 4x SMU instrument driver

Manual:

Usage: Unavailable

Capable of sourcing current and measuring voltage, as a Source Measurement Unit.

Parameters

- **tsp_node** – Number from 1 to 64 corresponding to the pre-configured TSP node number assigned to each module.
- **channel** – 'A' or 'B'

instrument_category

alias of *lightlab.laboratory.instruments.interfaces.Keithley*

MAGIC_TIMEOUT = 10

currStep = 0.0001

voltStep = 0.3

rampStepTime = 0.05

channel = None

tsp_node = None

reinstantiate_session(*address*, *tempSess*)

open()

close()

query(*queryStr*, *expected_talker=None*)

write(*writeStr*)

smu_string

smu_full_string

query_print(*query_string*, *expected_talker=None*)

smu_reset()

instrID()

is_master()
Returns true if this TSP node is the localnode.

The localnode is the one being interfaced with the Ethernet cable, whereas the other nodes are connected to it via the TSP-Link ports.

tsp_startup(*restart=False*)
Ensures that the TSP network is available.

- Checks if tsplink.state is online.
- If offline, send a reset().

smu_defaults()

startup()

set_sense_mode(*sense_mode='local'*)
Set sense mode. Defaults to local sensing.

setCurrent(*currAmps*)
This leaves the output on indefinitely

setVoltage(*voltVolts*)

getCurrent()

getVoltage()

setProtectionVoltage(*protectionVoltage*)

setProtectionCurrent(*protectionCurrent*)

compliance

measVoltage()

measCurrent()

protectionVoltage

protectionCurrent

enable(*newState=None*)
get/set enable state

setVoltageMode(*protectionCurrent=0.05*)

setCurrentMode(*protectionVoltage=1*)

lightlab.equipment.lab_instruments.NI_PCI_6723 module

Summary

Classes:

NI_PCI_6723	Primarily employs abstract classes.

Reference

class NI_PCI_6723(*name='The current source'*, *address=None*, *useChans=None*, ***kwargs*)

Bases: *lightlab.equipment.visa_bases.visa_driver.VISAInstrumentDriver*, *lightlab.equipment.abstract_drivers.electrical_sources.MultiModalSource*, *lightlab.equipment.abstract_drivers.electrical_sources.MultiChannelSource*

Primarily employs abstract classes. Follow the bases for more information

`VISAInstrumentDriver` provides communication to the board

`MultiModalSource` provides unit support and range checking

`MultiChannelSource` provides **notion of state** (stateDict) and channel support

Usage: *Instrument: CurrentSource*

instrument_category

alias of *lightlab.laboratory.instruments.interfaces.CurrentSource*

supportedModes = {'milliamp', 'mwperohm', 'volt', 'amp', 'wattperohm'}

baseUnitBounds = [0, 10]

baseToVoltCoef = 1

v2maCoef = 4

exceptOnRangeError = True

maxChannel = 32

targetPort = 16022

waitMsOnWrite = 500

MAGIC_TIMEOUT = 30

reinstantiate_session(*address*, *tempSess*)

startup()

open()

close()

query(*queryStr*, *expected_talker=None*)

write(*writeStr*, *expected_talker=None*)

The APEX does not deal with write; you have to query to clear the buffer

instrID()

There is no "*IDN?" command. Instead, test if it is alive, and then return a reasonable string

tcpTest(*num=2*)

setChannelTuning(*chanValDict*, *mode*, *waitTime=None*)
Sets a number of channel values and updates hardware

Parameters

- **chanValDict** (*dict*) – A dictionary specifying {channel: value}
- **waitTime** (*float*) – time in ms to wait after writing, default (None) is defined in the class

Returns was there a change in value

Return type (bool)

getChannelTuning(*mode*)
The inverse of setChannelTuning

Parameters **mode** (*str*) – units of the value in ('mwperohm', 'milliamp', 'volt')

Returns the full state of blocked out channels in units determined by mode argument

Return type (dict)

off()
Turn all voltages to zero, but maintain the session

wake()
Don't change the value but make sure it doesn't go to sleep after inactivity.

Good for long sweeps

sendToHardware(*waitTime=None*)
Updates current drivers with the present value of tuneState Converts it to a raw voltage, depending on the mode of the driver

Args:

lightlab.equipment.lab_instruments.RandS_SMBV100A_VG module

Summary

Classes:

RandS_SMBV100A_VG	Rohde and Schwartz SMBV100A

Reference

class RandS_SMBV100A_VG(*name='The Rohde and Schwartz'*, *address=None*, ***kwargs*)
Bases: *lightlab.equipment.visa_bases.visa_driver.VISAInstrumentDriver*, *lightlab.equipment.abstract_drivers.configurable.Configurable*

Rohde and Schwartz SMBV100A

Manual

Usage: TODO

This is a complicated class even though it is implementing about 1 percent of what the R&S can do. The philosophy is that there are several blocks that work independently.

1. Baseband digital modulation; accessed with `digiMod()`
2. Artificial Gaussian noise; accessed with `addNoise()`
3. RF carrier wave; accessed with `amplitude()`, `frequency()`, and `carrierMod()`

There are also global switches

1. All RF outputs; switched with `enable()`
2. All modulations; switched with `modulationEnable()`

instrument_category
alias of `lightlab.laboratory.instruments.interfaces.VectorGenerator`

amplitude(*amp=None*)
Amplitude is in dBm

Parameters **amp** (`float`) – If None, only gets

Returns output power amplitude

Return type (float)

frequency(*freq=None*)
Frequency is in Hertz. This does not take you out of list mode, if you are in it

Parameters **freq** (`float`) – If None, only gets

Returns center frequency

Return type (float)

enable(*enaState=None*)
Enabler for the entire output

Parameters **enaState** (`bool`) – If None, only gets

Returns is RF output enabled

Return type (bool)

modulationEnable(*enaState=None*)
Enabler for all modulation: data, noise, carrier

If this is False, yet device is enabled overall. Output will be a sinusoid

This is a global modulation switch, so:

```
modulationEnable(False)
```

is equivalent to:

```
carrierMod(False)
addNoise(False)
digiMod(False)
```

Parameters **enaState** (`bool`) – If None, only gets

Returns is global modulation enabled

Return type (bool)

addNoise(*enaState=True, bandwidth=None, cnRatio=None*)
Enabler for additive white gaussian noise modulations

Parameters

- **enaState** (*bool, None*) – If None, only sets parameters; does not change enable state
- **bandwidth** (*float*) – noise bandwidth in Hertz (typical = 1e6)
- **cnRatio** (*float*) – carrier-to-noise ratio in dB (typical = 10)

Returns is noise enabled

Return type (bool)

setPattern (*bitArray*)
Data pattern for digital modulation

Parameters **bitArray** (*ndarray*) – array that is boolean or binary 1/0

digiMod (*enaState=True, symbRate=None, amExtinct=None*)
Enabler for baseband data modulation

Data is derived from pattern.

Parameters

- **enaState** (*bool, None*) – if False, noise and RF modulations persist. If None, sets parameters but no state change
- **symbRate** (*float*) – bit rate in Symbols/s (typical = 3e6)
- **amExtinct** (*float*) – on/off ratio for AM, in percentage (0-100). 100 is full extinction

Returns is digital modulation enabled

Return type (bool)

Todo: From DM, only AM implemented right now. Further possibilities for formatting are endless

Possibility for arbitrary IQ waveform saving/loading in the :BB:ARB menu

carrierMod (*enaState=True, typMod=None, deviation=None, modFreq=None*)
Enabler for modulations of the RF carrier

Parameters

- **enaState** (*bool, None*) – if False, noise and data modulations persist. If None, sets parameters but no state change
- **typMod** (*str*) – what kind of modulation (of ['am', 'pm', 'fm']). Cannot be None when enaState is True
- **deviation** (*float, None*) – amplitude of the modulation, typMod dependent
- **modFreq** (*float, None*) – frequency of the modulation in Hertz (typical = 100e3)

Returns is carrier modulation of typMod enabled

Return type (bool)

There are three kinds of modulation, and they affect the interpretation of `deviation`.

- `typMod='am'`: depth (0–100) percent
- `typMod='pm'`: phase (0–50) radians
- `typMod='fm'`: frequency (0–16e6) Hertz

Only one type of modulation can be present at a time. `enaState` causes these effects:

- True: this type is enabled, other types are disabled
- False: all types are disabled
- None: sets parameters of this type, whether or not it is the one enabled

`listEnable`(*enaState=True*, *freqs=None*, *amps=None*, *isSlave=False*, *dwell=None*)

Sets up list mode.

If isSlave is True, dwell has no effect. Put the trigger signal into the **INST TRIG** port. If isSlave is False, it steps automatically every dwell time.

If both freqs and amps are None, do nothing to list data. If one is None, get a constant value from the frequency/amplitude methods. If either is a scalar, it will become a constant list, taking on the necessary length. If both are non-scalars, they must be the same length.

Parameters

- **`enaState`** (*bool*) – on or off
- **`freqs`** (*list*) – list data for frequency, in Hz
- **`amps`** (*list*) – list data for power, in dBm
- **`isSlave`** (*bool*) – Step through the list every time **INST TRIG** sees an edge (True), or every dwell time (False)
- **`dwell`** (*float*) – time to wait at each point, if untriggered

lightlab.equipment.lab_instruments.Tektronix_CSA8000_CAS module

Summary

Classes:

Tektronix_CSA8000_CAS	Communication analyzer scope

Reference

`class Tektronix_CSA8000_CAS`(*name='The DSA scope'*, *address=None*, ***kwargs*)

Bases: *lightlab.equipment.lab_instruments.Tektronix_DSA8300_Oscope.Tektronix_DSA8300_Oscope*

Communication analyzer scope

Note: @LightwaveLab: Is this different from the DSA? Maybe the DSA was the old one that got retired, but they are actually the same…

Not necessarily tested with the new abstract driver

Usage: *Instrument: Oscilloscope*

lightlab.equipment.lab_instruments.Tektronix_DPO4032_Oscope module

Summary

Classes:

Tektronix_DPO4032_Oscope	Manual: https://www.imperial.ac.uk/media/imperial-college/research-centres-and-groups/centre-for-bio-inspired-technology/7293027.PDF

Reference

class Tektronix_DPO4032_Oscope(*name='The DPO scope'*, *address=None*, ***kwargs*)

Bases: *lightlab.equipment.lab_instruments.Tektronix_DPO4034_Oscope.Tektronix_DPO4034_Oscope*

Manual: https://www.imperial.ac.uk/media/imperial-college/research-centres-and-groups/centre-for-bio-inspired-technology/7293027.PDF

totalChans = 2

timebaseConfig(*avgCnt=None*, *duration=None*)

Timebase and acquisition configure

Parameters

- **avgCnt** (*int*) – averaging done by the scope
- **duration** (*float*) – time, in seconds, for data to be acquired

Returns (dict) The present values of all settings above

acquire(*chans=None*, *timeout=None*, ***kwargs*)

Get waveforms from the scope.

If chans is None, it won't actually trigger, but it will configure.

If unspecified, the kwargs will be derived from the previous state of the scope. This is useful if you want to play with it in lab while working with this code too.

Parameters

- **chans** (*list*) – which channels to record at the same time and return
- **avgCnt** (*int*) – number of averages. special behavior when it is 1
- **duration** (*float*) – window width in seconds
- **position** (*float*) – trigger delay
- **nPts** (*int*) – number of sample points
- **timeout** (*float*) – time to wait for averaging to complete in seconds If it is more than a minute, it will do a test first

Returns recorded signals

Return type list[*Waveform*]

lightlab.equipment.lab_instruments.Tektronix_DPO4034_Oscope module

Summary

Classes:

Tektronix_DPO4034_Oscope	Slow DPO scope.

Reference

class Tektronix_DPO4034_Oscope(*name='The DPO scope'*, *address=None*, ***kwargs*)

Bases: *lightlab.equipment.visa_bases.visa_driver.VISAInstrumentDriver*, *lightlab.equipment.abstract_drivers.TekScopeAbstract.TekScopeAbstract*

Slow DPO scope. See abstract driver for description

Manual

Usage: *Instrument: Oscilloscope*

instrument_category

alias of *lightlab.laboratory.instruments.interfaces.Oscilloscope*

totalChans = 4

wfmDb()

Transfers a bundle of waveforms representing a signal database. Sample mode only.

Configuration such as position, duration are unchanged, so use an acquire(None, . . .) call to set them up

Parameters

- **chan** (*int*) – currently this only works with one channel at a time
- **nWfms** (*int*) – how many waveforms to acquire through sampling
- **untriggered** (*bool*) – if false, temporarily puts scope in free run mode

Returns all waveforms acquired

Return type (*FunctionBundle*(*Waveform*))

lightlab.equipment.lab_instruments.Tektronix_DSA8300_Oscope module

Summary

Classes:

Tektronix_DSA8300_Oscope	Sampling scope.

Reference

class Tektronix_DSA8300_Oscope(*name='The DSA scope'*, *address=None*, ***kwargs*)

Bases: *lightlab.equipment.visa_bases.visa_driver.VISAInstrumentDriver*, *lightlab.equipment.abstract_drivers.TekScopeAbstract.TekScopeAbstract*

Sampling scope. See abstract driver for description

Manual

Usage: *Instrument: Oscilloscope*

instrument_category
alias of `lightlab.laboratory.instruments.interfaces.Oscilloscope`

totalChans = 8

histogramStats (*chan*, *nWfms=3*, *untriggered=False*)
Samples for a bunch of waveforms. Instead of sending all of that data, It uses the scope histogram. It returns the percentage within a given sigma width

Returns standard deviation in volts (ndarray): proportion of points within [1, 2, 3] stddevs of mean

Return type (float)

lightlab.equipment.lab_instruments.Tektronix_PPG3202 module

Summary

Classes:

`Tektronix_PPG3202`	Python driver for Tektronix PPG 3202.

Reference

class Tektronix_PPG3202 (*name='Pattern Generator'*, *address=None*, ***kwargs*)
Bases: `lightlab.equipment.visa_bases.visa_driver.VISAInstrumentDriver`, `lightlab.equipment.abstract_drivers.configurable.Configurable`

Python driver for Tektronix PPG 3202.

Basic functionality includes setting all parameters on the main pannel and specifying data rate. Other functionality includes setting output data pattern on specifies channel.

Manual <https://www.tek.com/bit-error-rate-tester/patternpro-ppg-series-pattern-generator-manual/ppg1600-ppg3000-ppg3200-0>

instrument_category
alias of `lightlab.laboratory.instruments.interfaces.PatternGenerator`

setDataRate (*rate=None*)
Set the data rate of the PPG. Data rate can only be in the range of 1.5 Gb/s to 32 Gb/s

setMainParam (*chan=None*, *amp=None*, *offset=None*, *ptype=None*)
One function to set all parameters on the main window

setClockDivider (*div=None*)

setDataMemory (*chan=None*, *startAddr=None*, *bit=None*, *data=None*)

setHexDataMemory (*chan=None*, *startAddr=None*, *bit=None*, *Hdata=None*)

channelOn (*chan=None*)

channelOff (*chan=None*)

getAmplitude (*chan=None*)

getOffset (*chan=None*)

getDataRate()

getPatternType(*chan=None*)

getClockDivider()

lightlab.equipment.lab_instruments.Tektronix_RSA6120B_RFSA module

Summary

Classes:

Tektronix_RSA6120B_RFSA	TEKTRONIX RSA6120B, RF spectrum analyzer

Reference

class Tektronix_RSA6120B_RFSA(*name='The RF spectrum analyzer'*, *address=None*, ***kwargs*)

Bases: *lightlab.equipment.visa_bases.visa_driver.VISAInstrumentDriver*, *lightlab.equipment.abstract_drivers.configurable.Configurable*

TEKTRONIX RSA6120B, RF spectrum analyzer

Manual

Usage: TODO

Fairly simple class for getting RF spectra. The RSA6120 has a lot of advanced functionality, like spectrograms, which could be implemented later.

instrument_category
alias of *lightlab.laboratory.instruments.interfaces.RFSpectrumAnalyzer*

startup()

getMeasurements()

Returns tokens of currently active measurements

Return type (list[str])

setMeasurement(*measType='SPEC'*, *append=False*)
Turns on a measurement type

If append is False, turns off all measurements except for the one specified

See manual for other measurement types.

run(*doRun=True*)
Continuous run

After transferring spectra remotely, the acquisition stops going continuously. Call this when you want to run the display live. Useful for debugging when you are in lab.

sgramInit(*freqReso=None*, *freqRange=None*)

sgramTransfer(*duration=1.0*, *nLines=100*)
Transfers data that has already been taken. Typical usage:

```
self.sgramInit()
... << some activity >>
self.run(False)
self.spectrogram()
```

Currently only supports free running mode, so time is approximate. The accuracy of timing and consistency of timing between lines is not guaranteed.

spectrum (*freqReso=None*, *freqRange=None*, *typAvg='none'*, *nAvg=None*)
Acquires and transfers a spectrum.

Unspecified or None parameters will take on values used in previous calls, with the exception of typAvg – you must explicitly ask to average each time.

Parameters

- **freqReso** (`float, None`) – frequency resolution (typical = 1e3 to 10e6)
- **freqRange** (`array-like[float], None`) – 2-element frequency range
- **typAvg** (`str`) – type of averaging (of ['none', 'average', 'maxhold', 'minhold', 'avglog'])
- **nAvg** (`int, None`) – number of averages, if averaging

Returns power spectrum in dBm vs. Hz

Return type (lightlab.util.data.Spectrum)

lightlab.equipment.lab_instruments.Tektronix_TDS6154C_Oscope module

Summary

Classes:

`Tektronix_TDS6154C_Oscope`	Real time scope.

Reference

class Tektronix_TDS6154C_Oscope (*name='The TDS scope'*, *address=None*, ***kwargs*)
Bases: `lightlab.equipment.visa_bases.visa_driver.VISAInstrumentDriver`, `lightlab.equipment.abstract_drivers.TekScopeAbstract.TekScopeAbstract`

Real time scope. See abstract driver for description.

Manual

Usage: *Instrument: Oscilloscope*

instrument_category
alias of `lightlab.laboratory.instruments.interfaces.Oscilloscope`

totalChans = 4

Summary

Exceptions:

BuggyHardware	Not all instruments behave as they are supposed to.

Data:

`k`	str(object=’‘) -> str str(bytes_or_buffer[, encoding[, errors]]) -> str
`modname`	str(object=’‘) -> str str(bytes_or_buffer[, encoding[, errors]]) -> str
`mro`	list() -> new empty list list(iterable) -> new list initialized from iterable’s items

Reference

exception BuggyHardware

Bases: `Exception`

Not all instruments behave as they are supposed to. This might be lab specific. atait is not sure exactly how to deal with that.

lightlab.equipment.visa_bases package

Submodules:

lightlab.equipment.visa_bases.driver_base module

Summary

Classes:

InstrumentSessionBase	Base class for Instrument sessions, to be inherited and specialized by VISAObject and PrologixGPIBObject
TCPSocketConnection	Opens a TCP socket connection, much like netcat.

Data:

`CR`	str(object=’‘) -> str str(bytes_or_buffer[, encoding[, errors]]) -> str
`LF`	str(object=’‘) -> str str(bytes_or_buffer[, encoding[, errors]]) -> str

Reference

class InstrumentSessionBase

Bases: `abc.ABC`

Base class for Instrument sessions, to be inherited and specialized by VISAObject and PrologixGPIBObject

spoll()

LLO()

LOC()

open()

close()

write()

query()

wait()

clear()

query_raw_binary()

query_ascii_values(*message*, *converter='f'*, *separator=', '*, *container=<class 'list'>*)
Taken from pvisa.

instrID()
Returns the *IDN? string

timeout

class TCPSocketConnection(*ip_address*, *port*, *timeout=2*, *termination='n'*)
Bases: `object`

Opens a TCP socket connection, much like netcat.

Usage: s = TCPSocketConnection('socket-server.school.edu', 1111) s.connect() # connects to socket and leaves it open s.send('command') # sends the command through the socket r = s.recv(1000) # receives a message of up to 1000 bytes s.disconnect() # shuts down connection

Parameters

- **ip_address** (*str*) – hostname or ip address of the socket server
- **port** (*int*) – socket server's port number
- **timeout** (*float*) – timeout in seconds for establishing socket connection to socket server, default 2.

port = None
socket server's port number

connect()
Connects to the socket and leaves the connection open. If already connected, does nothing.

Returns socket object.

disconnect()
If connected, disconnects and kills the socket.

connected()
Context manager for ensuring that the socket is connected while sending and receiving commands to remote socket. This is safe to use everywhere, even if the socket is previously connected. It can also be nested. This is useful to bundle multiple commands that you desire to be executed together in a single socket connection, for example:

```
def query(self, query_msg, msg_length=2048):
    with self.connected():
        self._send(self._socket, query_msg)
```

(continues on next page)

(continued from previous page)

```
        recv = self._recv(self._socket, msg_length)
    return recv
```

startup()

send(*value*)
Sends an ASCII string to the socket server. Auto-connects if necessary.

Parameters **value** (*str*) – value to be sent

recv(*msg_length=2048*)
Receives an ASCII string from the socket server. Auto-connects if necessary.

Parameters **msg_length** (*int*) – maximum message length.

query(*query_msg*, *msg_length=2048*)

lightlab.equipment.visa_bases.prologix_gpib module

Summary

Classes:

PrologixGPIBObject	**param tempSess** If True, the session is opened and closed every time there is a command
PrologixResourceManager	Controls a Prologix GPIB-ETHERNET Controller v1.2 manual: http://prologix.biz/downloads/PrologixGpibEthernetManual.pdf

Reference

class PrologixResourceManager(*ip_address*, *timeout=2*)
Bases: *lightlab.equipment.visa_bases.driver_base.TCPSocketConnection*

Controls a Prologix GPIB-ETHERNET Controller v1.2 manual: http://prologix.biz/downloads/PrologixGpibEthernetManual.pdf

Basic usage:

```
p = PrologixResourceManager('prologix.school.edu')

p.connect()  # connects to socket and leaves it open
p.startup()  # configures prologix to communicate via gpib
p.send('++addr 23')  # talks to address 23
p.send('command value')  # sends the command and does not expect to read anything
p.query('command')  # sends a command but reads stuff back, this might hang if␣
↪buffer is empty
p.disconnect()
```

The problem with the above is that if there is any error with startup, send or query, the disconnect method will not be called. So we coded a decorator called `connected`, to be used as such:

```
p = PrologixResourceManager('prologix.school.edu')

with p.connected():
    p.startup()
    p.send('++addr 23')  # talks to address 23
    p.send('command value')  # sends the command and does not expect to read␣
↪anything
    p.query('command')  # sends a command but reads stuff back
```

If we try to send a message without the decorator, then we should connect and disconnect right before.

```
p = PrologixResourceManager('prologix.school.edu')

p.send('++addr 23')  # opens and close socket automatically
```

Warning: If a second socket is opened from the same computer while the first was online, the first socket will stop responding and Prologix will send data to the just-opened socket.

Todo: Make this class a singleton to mitigate the issue above.

Parameters

- **ip_address** (*str*) – hostname or ip address of the Prologix controller
- **timeout** (*float*) – timeout in seconds for establishing socket connection to socket server, default 2.

port = 1234
port that the Prologix GPIB-Ethernet controller listens to.

startup()
Sends the startup configuration to the controller. Just in case it was misconfigured.

query(*query_msg, msg_length=2048*)
Sends a query and receives a string from the controller. Auto-connects if necessary.

Args: query_msg (str): query message. msg_length (int): maximum message length. If the received

message does not contain a '

', it triggers another socket recv command with the same message length.

class PrologixGPIBObject(*address=None, tempSess=False*)
Bases: `lightlab.equipment.visa_bases.driver_base.InstrumentSessionBase`

Parameters

- **tempSess** (*bool*) – If True, the session is opened and closed every time there is a command
- **address** (*str*) – The full visa address in the form: prologix://prologix_ip_address/gpib_primary_address:gpib_secondary_address

spoll()
Return status byte of the instrument.

LLO()
    This command disables front panel operation of the currently addressed instrument.

LOC()
    This command enables front panel operation of the currently addressed instrument.

termination
    Termination GPIB character. Valid options – '\r\n', '\r', '\n', ''.

open()
    Open connection with instrument. If `tempSess` is set to False, please remember to close after use.

close()
    Closes the connection with the instrument. Side effect: disconnects prologix socket controller

write(*writeStr*)

query(*queryStr*, *withTimeout=None*)
    Read the unmodified string sent from the instrument to the computer.

wait(*bigMsTimeout=10000*)

clear()
    This command sends the Selected Device Clear (SDC) message to the currently specified GPIB address.

query_raw_binary(*queryStr*, *withTimeout=None*)
    Read the unmodified string sent from the instrument to the computer. In contrast to query(), no termination characters are stripped. Also no decoding.

timeout
    This timeout is between the user and the instrument. For example, if we did a sweep that should take ~10 seconds but ends up taking longer, you can set the timeout to 20 seconds.

lightlab.equipment.visa_bases.visa_driver module

Summary

Exceptions:

IncompleteClass
InstrumentIOError

Classes:

DefaultDriver	alias of *lightlab.equipment.visa_bases.visa_driver.VISAInstrumentDriver*
DriverMeta	Driver initializer returns an instrument in `instrument_category`, not an instance of the Driver itself, unless * `instrument_category` is None * `directInit=True` is passed in
InstrumentSession	This class is the interface between the higher levels of lightlab instruments and the driver controlling the GPIB line.
VISAInstrumentDriver	Generic (but not abstract) class for an instrument.

Reference

exception InstrumentIOError
Bases: `RuntimeError`

class InstrumentSession(*address=None*, *tempSess=False*)
Bases: `lightlab.equipment.visa_bases.visa_driver._AttrGetter`

This class is the interface between the higher levels of lightlab instruments and the driver controlling the GPIB line. Its methods are specialized into either PrologixGPIBObject or VISAObject.

This was mainly done because the Prologix GPIB Ethernet controller is not VISA compatible and does not provide a VISA interface.

If the address starts with 'prologix://', it will use PrologixGPIBObject's methods, otherwise it will use VISAObject's methods (relying on pyvisa).

Warning: Since this is a wrapper class to either `PrologixGPIBObject`

or `VISAObject`, avoid using super() in overloaded methods. (see this)

reinstantiate_session(*address*, *tempSess*)

open()

close()

exception IncompleteClass
Bases: `Exception`

class DriverMeta(*name*, *bases*, *dct*)
Bases: `type`

Driver initializer returns an instrument in `instrument_category`, not an instance of the Driver itself, unless

- `instrument_category` is None
- `directInit=True` is passed in

Also checks that the API is satistied at compile time, providing some early protection against bad drivers, like this: `test_badDriver()`.

Checks that it satisfies the API of its Instrument.

This occurs at compile-time

class VISAInstrumentDriver(*name='Default Driver'*, *address=None*, ***kwargs*)
Bases: `lightlab.equipment.visa_bases.visa_driver.InstrumentSession`

Generic (but not abstract) class for an instrument. Initialize using the literal visa address

Contains a visa communication object.

instrument_category = None

startup()

open()

close()

DefaultDriver
alias of `lightlab.equipment.visa_bases.visa_driver.VISAInstrumentDriver`

lightlab.equipment.visa_bases.visa_object module

Summary

Classes:

`VISAObject`	Abstract class for something that communicates via messages (GPIB/USB/Serial/TCPIP/etc.).

Data:

`CR`	str(object='') -> str str(bytes_or_buffer[, encoding[, errors]]) -> str
`LF`	str(object='') -> str str(bytes_or_buffer[, encoding[, errors]]) -> str
`OPEN_RETRIES`	int(x=0) -> integer int(x, base=10) -> integer

Reference

class VISAObject(*address=None, tempSess=False*)

Bases: `lightlab.equipment.visa_bases.driver_base.InstrumentSessionBase`

Abstract class for something that communicates via messages (GPIB/USB/Serial/TCPIP/etc.). It handles message-based sessions in a way that provides a notion of object permanence to the connection with a particular address.

It acts like a `pyvisa` message-based session, but it is not a subclass; it is a wrapper. It only contains one (at at time). That means VISAObject can offer extra opening, closing, session management, and error reporting features.

This class relies on pyvisa to work

Parameters

- **tempSess** (`bool`) – If True, the session is opened and closed every time there is a command
- **address** (`str`) – The full visa address

resMan = None

mbSession = None

open()

Open connection with 5 retries.

close()

write(*writeStr*)

query(*queryStr, withTimeout=None*)

instrID()

Returns the *IDN? string

timeout

wait(*bigMsTimeout=10000*)

LLO()

LOC()

clear()

query_raw_binary()

spoll()

termination

3.1.3 lightlab.laboratory package

The laboratory module facilitates the organization and documentation of instruments, experiments and devices. The objects defined here are designed to be "hashable", i.e., easy to store and share.

Submodules:

lightlab.laboratory.devices module

This module contains virtual tokens for optical and electronic devices.

lightlab.laboratory.experiments module

This module contains tokens for experiments that use devices and instruments. This is useful to keep track of what is connected to what.

Summary

Classes:

Experiment	Experiment base class.
MasterExperiment	Does nothing except hold sub experiments to synchronize them.

Reference

class Experiment(*instruments=None*, *devices=None*, ***kwargs*)

Bases: *lightlab.laboratory.virtualization.Virtualizable*

Experiment base class.

This class is intended to be inherited by the user.

Usage:

```
experiment = Experiment()
with experiment.asVirtual():
    experiment.measure()  # measure is a DualFunction


# Quick tutorial on decorators:
with obj as foo:
```

(continues on next page)

(continued from previous page)

```
    foo.something()

# this is equivalent to
foo = obj.__enter__()
foo.something()
obj.__exit__()
```

lab

is_valid(*reset=True*)

valid

instruments = None

instruments_requirements = None

devices = None

validate_exprs = None

connections = None

name = None

startup()

global_hardware_warmup()

hardware_warmup()

hardware_cooldown()

asReal()
Wraps making self.virtual to False. Also does hardware warmup and cooldown

registerInstrument(*instrument, host=None, bench=None*)

registerInstruments(**instruments, host=None, bench=None*)

registerConnection(*connection*)

registerConnections(**connections*)

validate()

lock(*key*)

unlock()

display()

class MasterExperiment
Bases: `lightlab.laboratory.virtualization.Virtualizable`

Does nothing except hold sub experiments to synchronize them. This is purely a naming thing.

lightlab.laboratory.state module

This module contains classes responsible to maintain a record of the current state of the lab.

Users typically just have to import the variable `lab`.

Warning: Developers: do not import `lab` anywhere inside the *lightlab* package. This will cause the deserialization of the JSON file before the definition of the classes of the objects serialized. If you want to make use of the variable lab, import it like this:

```
import lightlab.laboratory.state as labstate

# developer code
device = function_that_returns_device()
bench = labstate.lab.findBenchFromInstrument(device)
```

Summary

Classes:

LabState	Represents the set of objects and connections present in lab, with the ability to safely save and load to and from a `.json` file.

Functions:

hash_sha256	Returns the hash of string encoded via the SHA-256 algorithm from hashlib
init_module	
patch_labstate	This takes the loaded JSON version of labstate (old_lab) and applies a patch to the current version of labstate.
timestamp_string	Returns timestamp in iso format (e.g.

Data:

`can_write`	bool(x) -> bool
`lab`	

Reference

`timestamp_string`()
Returns timestamp in iso format (e.g. 2018-03-25T18:30:55.328389)

`hash_sha256`(*string*)
Returns the hash of string encoded via the SHA-256 algorithm from hashlib

`class LabState`(*filename=None*)
Bases: *lightlab.laboratory.Hashable*

Represents the set of objects and connections present in lab, with the ability to safely save and load to and from a `.json` file.

`instruments_dict`
Dictionary of instruments, concatenated from `lab.instruments`.

`hosts = None`
list(*Host*) list of hosts

benches = None
list(*Bench*) list of benches

connections = None
list(dict(str -> str)) list of connections

devices = None
list(*Device*) list of devices

instruments = None
list(*Instrument*) list of instruments

updateHost(**hosts*)
Updates hosts in the hosts list.

Checks the number of instrumentation_servers. There should be exactly one.

Parameters ***** (Host) – hosts

Raises

- RuntimeError – Raised if duplicate names are found.
- TypeError – Raised if host is not of type *Host*

updateBench(**benches*)
Updates benches in the benches list.

Parameters ***** (Bench) – benches

Raises

- RuntimeError – Raised if duplicate names are found.
- TypeError – Raised if bench is not of type *Bench*

deleteInstrumentFromName(*name*)
Deletes an instrument by their name.

Example:

```
lab.deleteInstrumentFromName("Keithley2")
```

Parameters **name** (*str*) – Instrument name

insertInstrument(*instrument*)
Inserts instrument in labstate.

Parameters **instrument** (Instrument) – instrument to insert.

Raises

- RuntimeError – Raised if duplicate names are found.
- TypeError – Raised if instrument is not of type *Instrument*

insertDevice(*device*)
Inserts device in labstate.

Parameters **device** (Device) – device to insert.

Raises

- RuntimeError – Raised if duplicate names are found.
- TypeError – Raised if device is not of type *Device*

updateConnections(**connections*)

Updates connections between instruments and devices.

A connection is a tuple with a pair of one-entry dictionaries, as such:

```
conn = ({instr1: port1}, {instr2: port2})
```

The code assumes that there can only be one connection per port. This method performs the following action:

1. **verifies that *port* is one of *instr.ports*. Otherwise raises** a `RuntimeError`.
2. **deletes any connection in `lab.connections` that has** either `{instr1: port1}` or `{instr1: port1}`, and logs the deleted connection as a warning.
3. adds new connection

Parameters **`connections`** (*`tuple(dict)`*) – connection to update

devices_dict

Dictionary of devices, concatenated from `lab.devices`.

Access with `devices_dict[device.name]`

Todo: Logs a warning if duplicate is found.

findBenchFromInstrument(*instrument*)

Returns the bench that contains the instrument.

This obviously assumes that one instrument can only be present in one bench.

findBenchFromDevice(*device*)

Returns the bench that contains the device.

This obviously assumes that one device can only be present in one bench.

findHostFromInstrument(*instrument*)

Returns the host that contains the instrument.

This obviously assumes that one instrument can only be present in one host.

classmethod loadState(*filename=None*, *validateHash=True*)

Loads a *`LabState`* object from a file.

It loads and instantiates a copy of every object serialized with `lab.saveState(filename)`. The objects are saved with `jsonpickle`, and must be hashable and contain no C-object references. For convenience, lab objects are inherited from *:class:'lightlab.laboratory.Hashable*.

By default, the sha256 hash is verified at import time to prevent instantiating objects from a corrupted file.

A file version is also compared to the code version. If a new version of this class is present, but your `json` file is older, a `RuntimeWarning` is issued.

Todo: When importing older `json` files, know what to do to upgrade it without bugs.

Parameters

- **`filename`** (*`str or Path`*) – file to load from.
- **`validateHash`** (*`bool`*) – whether to check the hash, default True.

Raises

- `RuntimeWarning` – if file version is older than lightlab.
- `RuntimeError` – if file version is newer than lightlab.
- `JSONDecodeError` – if there is any problem decoding the .json file.
- `JSONDecodeError` – if the hash file inside the .json file does not match the computed hash during import.
- `OSError` – if there is any problem loading the file.

`filename`
Filename used to serialize labstate.

`saveState` (*fname=None*, *save_backup=True*)
Saves the current lab, together with all its dependencies, to a JSON file.

But first, it checks whether the file has the same hash as the previously loaded one. If file is not found, skip this check.

If the labstate was created from scratch, save with `_saveState()`.

Parameters

- **`fname`** (*`str or Path`*) – file path to save
- **`save_backup`** (*`bool`*) – saves a backup just in case, defaults to True.

Raises `OSError` – if there is any problem saving the file.

`init_module` (*module*)

`patch_labstate` (*from_version*, *old_lab*)
This takes the loaded JSON version of labstate (old_lab) and applies a patch to the current version of labstate.

lightlab.laboratory.virtualization module

Provides a framework for making virtual instruments that present the same interface and simulated behavior as the real ones. Allows a similar thing with functions, methods, and experiments.

Dualization is a way of tying together a real instrument with its virtual counterpart. This is a powerful way to test procedures in a virtual environment before flipping the switch to reality. This is documented in `tests.test_virtualization`.

`virtualOnly`
bool – If virtualOnly is True, any “`with`” statements using asReal will just skip the block. When not using a context manager (i.e. `exp.virtual = False`), it will eventually produce a `VirtualizationError`.

Summary

Exceptions:

`VirtualizationError`

Classes:

`DualFunction`	This class implements a descriptor for a function whose behavior depends on an instance's variable.
`DualInstrument`	Holds a real instrument and a virtual instrument.
`DualMethod`	This differs from DualFunction because it exists outside of the object instance.
`VirtualInstrument`	Just a placeholder for future functionality
`Virtualizable`	Virtualizable means that it can switch between two states, usually corresponding to a real-life situation and a virtual/simulated situation.

Data:

`virtualOnly`	bool(x) -> bool

Reference

class Virtualizable

Bases: `object`

Virtualizable means that it can switch between two states, usually corresponding to a real-life situation and a virtual/simulated situation.

The attribute synced refers to other Virtualizables whose states will be synchronized with this one

synced = None

synchronize(**newVirtualizables*)

Adds another object that this one will put in the same virtual state as itself.

Parameters **newVirtualizables** (**args*) – Other virtualizable things

virtual

Returns the virtual state of this object

asVirtual()

Temporarily puts this and synchronized in a virtual state. The state is reset at the end of the with block.

Example usage:

```python
exp = Virtualizable()
with exp.asVirtual():
    print(exp.virtual)  # prints True
print(exp.virtual)  # VirtualizationError
```

asReal()

Temporarily puts this and synchronized in a virtual state. The state is reset at the end of the with block.

If `virtualOnly` is True, it will skip the block without error

Example usage:

```python
exp = Virtualizable()
with exp.asVirtual():
    print(exp.virtual)  # prints False
print(exp.virtual)  # VirtualizationError
```

class VirtualInstrument

Bases: `object`

Just a placeholder for future functionality

asVirtual()

do nothing

class DualInstrument(*real_obj=None*, *virt_obj=None*)

Bases: *lightlab.laboratory.virtualization.Virtualizable*

Holds a real instrument and a virtual instrument. Feeds through `__getattribute__` and `__setattr__`: very powerful. It basically appears as one or the other instrument, as determined by whether it is in virtual or real mode.

This is especially useful if you have an instrument stored in the JSON labstate, and would then like to virtualize it in your notebook. In that case, it does not reinitialize the driver.

This is documented in `tests.test_virtualization`.

`isinstance()` and `.__class__` will tell you the underlying instrument type `type()` will give you the `DualInstrument` subclass:

```
dual = DualInstrument(realOne, virtOne)
with dual.asReal():
    isinstance(dual, type(realOne))   # True
    dual.meth is realOne.meth  # True
isinstance(dual, type(realOne))  # False
```

Parameters

- **real_obj** (`Instrument`) – the real reference
- **virt_obj** (`VirtualInstrument`) – the virtual reference

real_obj = None

virt_obj = None

virtual

Returns the virtual state of this object

class DualFunction(*virtual_function=None*, *hardware_function=None*, *doc=None*)

Bases: `object`

This class implements a descriptor for a function whose behavior depends on an instance's variable. This was inspired by core python's property descriptor.

Example usage:

```
@DualFunction
def measure(self, *args, **kwargs):
    # use a model to simulate outputs based on args and kwargs and self.
    return simulated_output

@measure.hardware
def measure(self, *args, **kwargs):
    # collect data from hardware using args and kwargs and self.
    return output
```

The "virtual" function will be called if `self.virtual` equals True, otherwise the hardware decorated function will be called instead.

hardware(*func*)

virtual(*func*)

class DualMethod(*dualInstrument=None, virtual_function=None, hardware_function=None, doc=None*)

Bases: `object`

This differs from DualFunction because it exists outside of the object instance. Instead it takes the object when initializing.

It uses __call__ instead of __get__ because it is its own object

Todo: The naming for DualFunction and DualMethod are backwards. Will break notebooks when changed.

exception VirtualizationError

Bases: `RuntimeError`

Subpackages:

lightlab.laboratory.instruments package

The Instruments module is divided into two: bases and interfaces.

All classes are imported into this namespace.

Submodules:

lightlab.laboratory.instruments.bases module

This module provides an interface for instruments, hosts and benches in the lab.

Summary

Exceptions:

NotFoundError	Error thrown when instrument is not found

Classes:

Bench	Represents an experiment bench for the purpose of facilitating its location in lab.
Device	Represents a device in lab.
Host	Computer host, from which GPIB/VISA commands are issued.
Instrument	Represents an instrument in lab.
LocalHost	
MockInstrument	

Reference

class Host (*name='Unnamed Host'*, *hostname=None*, ***kwargs*)
Bases: `lightlab.laboratory.Node`

Computer host, from which GPIB/VISA commands are issued.

mac_address = None

os = 'linux-ubuntu'

hostname = None

name

instruments

isLive()
Pings the system and returns if it is alive.

gpib_port_to_address(*port*, *board=0*)

Parameters

- **port** (*int*) – The port on the GPIB bus of this host
- **board** (*int*) – For hosts with multiple GPIB busses

Returns the address that can be used in an initializer

Return type (str)

list_resources_info(*use_cached=True*)
Executes a query to the NI Visa Resource manager and returns a list of instruments connected to it.

Parameters **use_cached** (*bool*) – query only if not cached, default True

Returns list of *pyvisa.highlevel.ResourceInfo* named tuples.

Return type list

list_gpib_resources_info(*use_cached=True*)
Like `list_resources_info()`, but only returns gpib resources.

Parameters **use_cached** (*bool*) – query only if not cached, default True.

Returns list of `pyvisa.highlevel.ResourceInfo` named tuples.

Return type (list)

get_all_gpib_id(*use_cached=True*)
Queries the host for all connected GPIB instruments, and queries their identities with `instrID()`.

Warning: This might cause your instrument to lock into remote mode.

Parameters **use_cached** (*bool*) – query only if not cached, default True

Returns dictionary with gpib addresses as keys and identity strings as values.

Return type dict

findGpibAddressById(*id_string_search*, *use_cached=True*)
Finds a gpib address using `get_all_gpib_id()`, given an identity string.

Parameters

- **id_string_search** (*str*) – identity string
- **use_cached** (*bool*) – query only if not cached, default True

Returns address if found.

Return type str

Raises *NotFoundError* – If the instrument is not found.

addInstrument(**instruments*)

Adds an instrument to lab.instruments if it is not already present.

Parameters ***instruments** (Instrument) – instruments

removeInstrument(**instruments*)

Disconnects the instrument from the host

Parameters ***instruments** (Instrument) – instruments

Todo: Remove all connections

checkInstrumentsLive()

Checks whether all instruments are "live".

Instrument status is checked with the *Instrument.isLive()* method

Returns True if all instruments are live, False otherwise

Return type (bool)

display()

Displays the host's instrument table in a nice format.

class LocalHost(*name=None*)

Bases: *lightlab.laboratory.instruments.bases.Host*

isLive()

Pings the system and returns if it is alive.

class Bench(*name*, **args*, ***kwargs*)

Bases: *lightlab.laboratory.Node*

Represents an experiment bench for the purpose of facilitating its location in lab.

name = None

instruments

devices

addInstrument(**instruments*)

Adds an instrument to lab.instruments if it is not already present and connects to the host.

Parameters ***instruments** (Instrument) – instruments

removeInstrument(**instruments*)

Detaches the instrument from the bench.

Parameters ***instruments** (Instrument) – instruments

Todo: Remove all connections

addDevice(**devices*)

Adds a device to lab.devices if it is not already present and places it in the bench.

Parameters ***devices** (Device) – devices

removeDevice(**devices*)
Detaches the device from the bench.

Parameters ***devices** (`Device`) – devices

Todo: Remove all connections

display()
Displays the bench's table in a nice format.

class Instrument(*name='Unnamed Instrument'*, *id_string=None*, *address=None*, ***kwargs*)
Bases: *`lightlab.laboratory.Node`*

Represents an instrument in lab.

This class stores information about instruments, for the purpose of facilitating verifying whether it is connected to the correct devices.

Driver feedthrough Methods, properties, and even regular attributes that are in `essential_attributes` of the class will get/set/call through to the driver object.

Do not instantiate directly Calling a **VISAInstrumentDriver** class will return an **Instrument** object

Short example:

```
osa = Apex_AP2440A_OSA(name='foo', address='NULL')
osa.spectrum()
```

Long example *Instrument configuration*

Detailed testing `test_driver_init()`

essentialMethods = ['startup']
list of methods to be fed through the driver

essentialProperties = []
list of properties to be fed through the driver

optionalAttributes = []
list of optional attributes to be fed through the driver

ports = None
list(str) Port names of instruments. To be used with labstate connections.

address = None
Complete Visa address of the instrument (e.g. `visa://hostname/GPIB0::1::INSTR`)

implementedOptionals

hardware_warmup()
Called before the beginning of an experiment.

Typical warmup procedures include RESET gpib commands.

hardware_cooldown()
Called after the end of an experiment.

Typical cooldown procedures include laser turn-off, or orderly wind-down of current etc.

warmedUp()
A context manager that warms up and cools down in a "with" block

Usage:

```
with instr.warmedUp() as instr:  # warms up instrument
    instr.doStuff()
    raise Exception("Interrupting experiment")
# cools down instrument, even in the event of exception
```

driver_class
Class of the actual equipment driver (from *lightlab.equipment.lab_instruments*)

This way the object knows how to instantiate a driver instance from the labstate.

driver_object
Instance of the equipment driver.

driver
Alias of *driver_object()*.

bench
Property that only accepts <class 'lightlab.laboratory.instruments.bases.Bench'> values

host
Property that only accepts <class 'lightlab.laboratory.instruments.bases.Host'> values

name
(property) Instrument name (can only set during initialization)

id_string
The id_string should match the value returned by `self.driver.instrID()`, and is checked by the command `self.isLive()` in order to authenticate that the intrument in that address is the intended one.

display()
Displays the instrument's info table in a nice format.

isLive()
Attempts VISA connection to instrument, and checks whether `instrID()` matches *id_string*.

Produces a warning if it is live but the id_string is wrong.

Returns True if "live", False otherwise.

Return type (bool)

connectHost(*new_host*)
Sets/changes instrument's host.

Equivalent to `self.host = new_host`

placeBench(*new_bench*)
Sets/changes instrument's bench.

Equivalent to `self.bench = new_bench`

class MockInstrument(*name='Unnamed Instrument'*, *id_string=None*, *address=None*, ***kwargs*)
Bases: *lightlab.laboratory.instruments.bases.Instrument*

exception NotFoundError
Bases: `RuntimeError`

Error thrown when instrument is not found

class Device(*name*, ***kwargs*)
Bases: *lightlab.laboratory.Node*

Represents a device in lab. Only useful for documenting the experiment.

Todo: Add equality function

name = None
device name

ports = None
list(str) port names

bench
Property that only accepts <class 'lightlab.laboratory.instruments.bases.Bench'> values

display()
Displays the device's info table in a nice format.

lightlab.laboratory.instruments.interfaces module

This module defines the essential interfaces for each kind of instrument.

Todo: Document every interface.

Summary

Classes:

ArduinoInstrument	Usage: TODO
Clock	Usage: *Instrument: Clock*
CurrentSource	Deprecated/Future
DSAOscilloscope	Usage: *Instrument: Oscilloscope*
FunctionGenerator	Usage: *Instrument: FunctionGenerator*
Keithley	Usage: *Instrument: Keithley and SourceMeter*
LaserSource	Usage: *Instrument: LaserSource*
NICurrentSource	Usage: *Instrument: CurrentSource*
NetworkAnalyzer	Usage: *Instrument: NetworkAnalyzer*
OpticalSpectrumAnalyzer	Usage: *Instrument: OpticalSpectrumAnalyzer*
Oscilloscope	Usage: *Instrument: Oscilloscope*
PatternGenerator	
PowerMeter	Usage: *Instrument: PowerMeter*
PulsePatternGenerator	Usage: *Instrument: PulsePatternGenerator*
RFSpectrumAnalyzer	Usage: TODO
SourceMeter	Usage: *Instrument: Keithley and SourceMeter*
VariableAttenuator	Usage: *Instrument: VariableOpticalAttenuator*
VectorGenerator	Todo: Usage example

Reference

class PowerMeter(*name='Unnamed Instrument'*, *id_string=None*, *address=None*, ***kwargs*)
Bases: *lightlab.laboratory.instruments.bases.Instrument*

Usage: *Instrument: PowerMeter*

essentialMethods = ['startup', 'powerDbm', 'powerLin']

class SourceMeter(*name='Unnamed Instrument'*, *id_string=None*, *address=None*, ***kwargs*)

Bases: `lightlab.laboratory.instruments.bases.Instrument`

Usage: *Instrument: Keithley and SourceMeter*

essentialMethods = ['startup', 'setCurrent', 'getCurrent', 'measVoltage', 'setProtecti

hardware_warmup()

Called before the beginning of an experiment.

Typical warmup procedures include RESET gpib commands.

hardware_cooldown()

Called after the end of an experiment.

Typical cooldown procedures include laser turn-off, or orderly wind-down of current etc.

class Keithley(*name='Unnamed Instrument'*, *id_string=None*, *address=None*, ***kwargs*)

Bases: `lightlab.laboratory.instruments.interfaces.SourceMeter`

Usage: *Instrument: Keithley and SourceMeter*

essentialMethods = ['startup', 'setCurrent', 'getCurrent', 'measVoltage', 'setProtecti

class VectorGenerator(*name='Unnamed Instrument'*, *id_string=None*, *address=None*, ***kwargs*)

Bases: `lightlab.laboratory.instruments.bases.Instrument`

Todo: Usage example

essentialMethods = ['startup', 'amplitude', 'frequency', 'enable', 'modulationEnable',

class Clock(*name='Unnamed Instrument'*, *id_string=None*, *address=None*, ***kwargs*)

Bases: `lightlab.laboratory.instruments.bases.Instrument`

Usage: *Instrument: Clock*

essentialMethods = ['startup', 'enable', 'frequency']

optionalAttributes = ['amplitude', 'sweepSetup', 'sweepEnable']

class NICurrentSource(*name='Unnamed Instrument'*, *id_string=None*, *address=None*, ***kwargs*)

Bases: `lightlab.laboratory.instruments.bases.Instrument`

Usage: *Instrument: CurrentSource*

essentialMethods = ['startup', 'setChannelTuning', 'getChannelTuning', 'off']

class CurrentSource(*name='Unnamed Instrument'*, *id_string=None*, *address=None*, ***kwargs*)

Bases: `lightlab.laboratory.instruments.bases.Instrument`

Deprecated/Future

essentialMethods = ['startup', 'setChannelTuning', 'getChannelTuning', 'off']

class FunctionGenerator(*name='Unnamed Instrument'*, *id_string=None*, *address=None*, ***kwargs*)

Bases: `lightlab.laboratory.instruments.bases.Instrument`

Usage: *Instrument: FunctionGenerator*

essentialMethods = ['startup', 'frequency', 'waveform', 'amplAndOffs', 'amplitudeRange

optionalAttributes = ['setArbitraryWaveform']

class LaserSource(*name='Unnamed Instrument'*, *id_string=None*, *address=None*, ***kwargs*)

Bases: `lightlab.laboratory.instruments.bases.Instrument`

Usage: *Instrument: LaserSource*

essentialMethods = ['startup', 'setChannelEnable', 'getChannelEnable', 'setChannelWls'

essentialProperties = ['enableState', 'wls', 'powers']

optionalAttributes = ['wlRanges', 'allOff']

class OpticalSpectrumAnalyzer(*name='Unnamed Instrument'*, *id_string=None*, *address=None*, ***kwargs*)

Bases: `lightlab.laboratory.instruments.bases.Instrument`

Usage: *Instrument: OpticalSpectrumAnalyzer*

essentialMethods = ['startup', 'spectrum']

essentialProperties = ['wlRange']

hardware_warmup()

Called before the beginning of an experiment.

Typical warmup procedures include RESET gpib commands.

class Oscilloscope(*name='Unnamed Instrument'*, *id_string=None*, *address=None*, ***kwargs*)

Bases: `lightlab.laboratory.instruments.bases.Instrument`

Usage: *Instrument: Oscilloscope*

essentialMethods = ['startup', 'acquire', 'wfmDb', 'run']

optionalAttributes = ['histogramStats']

hardware_cooldown()

Keep it running continuously in case you are in lab and want to watch

class DSAOscilloscope(*name='Unnamed Instrument'*, *id_string=None*, *address=None*, ***kwargs*)

Bases: `lightlab.laboratory.instruments.interfaces.Oscilloscope`

Usage: *Instrument: Oscilloscope*

essentialMethods = ['startup', 'acquire', 'wfmDb', 'run', 'histogramStats']

class PulsePatternGenerator(*name='Unnamed Instrument'*, *id_string=None*, *address=None*, ***kwargs*)

Bases: `lightlab.laboratory.instruments.bases.Instrument`

Usage: *Instrument: PulsePatternGenerator*

essentialMethods = ['startup', 'setPrbs', 'setPattern', 'getPattern', 'on', 'syncSourc

class RFSpectrumAnalyzer(*name='Unnamed Instrument'*, *id_string=None*, *address=None*, ***kwargs*)

Bases: `lightlab.laboratory.instruments.bases.Instrument`

Usage: TODO

essentialMethods = ['startup', 'getMeasurements', 'setMeasurement', 'run', 'sgramInit'

class VariableAttenuator(*name='Unnamed Instrument'*, *id_string=None*, *address=None*, ***kwargs*)

Bases: `lightlab.laboratory.instruments.bases.Instrument`

Usage: *Instrument: VariableOpticalAttenuator*

essentialMethods = ['startup', 'on', 'off']

essentialProperties = ['attenDB', 'attenLin', 'wavelength', 'calibration']

class NetworkAnalyzer(*name='Unnamed Instrument'*, *id_string=None*, *address=None*, ***kwargs*)
Bases: `lightlab.laboratory.instruments.bases.Instrument`

Usage: *Instrument: NetworkAnalyzer*

essentialMethods = ['startup', 'amplitude', 'frequency', 'enable', 'run', 'sweepSetup'

class ArduinoInstrument(*name='Unnamed Instrument'*, *id_string=None*, *address=None*, ***kwargs*)
Bases: `lightlab.laboratory.instruments.bases.Instrument`

Usage: TODO

essentialMethods = ['startup', 'write', 'query']

class PatternGenerator(*name='Unnamed Instrument'*, *id_string=None*, *address=None*, ***kwargs*)
Bases: `lightlab.laboratory.instruments.bases.Instrument`

essentialMethods = ['startup', 'setDataRate', 'setMainParam', 'setDataMemory', 'setHex

Summary

Classes:

FrozenDict	Don't forget the docstrings!!
Hashable	Hashable class to be used with jsonpickle's module.
NamedList	Object list that enforces that there are only one object.name in the list.
Node	Node is a token of an object that exists in a laboratory.
TypedList	Object list that enforces that there are only one object.name in the list and that they belong to a certain class (obj_type).

Functions:

typed_property	Property that only accepts instances of a class and stores the contents in self.name

Reference

class FrozenDict(*data*)
Bases: `collections.abc.Mapping`

Don't forget the docstrings!!

class Hashable(***kwargs*)
Bases: `object`

Hashable class to be used with jsonpickle's module. Rationale: This is a fancy way to do `self.__dict__ == other.__dict__`. That line fails when there are circular references within the __dict__. `Hashable` solves that.

By default, every key-value in the initializer will become instance variables. E.g. `Hashable(a=1).a == 1`

No instance variables starting with "__" will be serialized.

context = <MagicMock name='mock.Pickler()' id='140061894374568'>

class Node(***kwargs*)

Bases: `lightlab.laboratory.Hashable`

Node is a token of an object that exists in a laboratory. For example, subclasses are:

- a `Device`
- a `Host`
- a `Bench`
- an `Instrument`

bench = None

placeBench(*new_bench*)

typed_property(*type_obj, name*)

Property that only accepts instances of a class and stores the contents in self.name

class NamedList(**args, read_only=False*)

Bases: `collections.abc.MutableSequence`, `lightlab.laboratory.Hashable`

Object list that enforces that there are only one object.name in the list.

read_only = False

dict

values

keys

items()

check(*value*)

check_presence(*name*)

insert(*index, value*)

S.insert(index, value) – insert value before index

class TypedList(*obj_type, *args, read_only=False, **kwargs*)

Bases: `lightlab.laboratory.NamedList`

Object list that enforces that there are only one object.name in the list and that they belong to a certain class (obj_type).

check(*value*)

3.1.4 lightlab.util package

Submodules:

lightlab.util.characterize module

Timing is pretty important. These functions monitor behavior in various ways with timing considered. Included is strobeTest which sweeps the delay between actuate and sense, and monitorVariable for drift

Summary

Functions:

monitorVariable	Monitors some process over time.
strobeTest	Looks at a sense variable at different delays after calling an actuate function.
sweptStrobe	Takes in a NdSweeper and looks at the effect of delaying between actuation from measurement.

Reference

strobeTest (*fActuate*, *fSense*, *fReset=None*, *nPts=10*, *maxDelay=1*, *visualize=True*)

Looks at a sense variable at different delays after calling an actuate function. Good for determining the time needed to wait for settling. Calls each function once per delay point to construct a picture like the strobe experiment, or a sampling scope

Parameters

- **fActuate** (*function*) – no arguments, no return. Called first.
- **fSense** (*function*) – no arguments, returns a scalar or np.array. Called after a given delay
- **fReset** (*function*) – no arguments, no return. Called after the trial unless None. Usually of the same form as fActuate

Returns fSense values vs. delay

Return type (*FunctionBundle*)

sweptStrobe (*varSwp*, *resetArg*, *nPts=10*, *maxDelay=1*)

Takes in a NdSweeper and looks at the effect of delaying between actuation from measurement. Does the gathering.

Starts by taking start and end baselines, for ease of visualization.

Parameters

- **varSwp** (NdSweeper) – the original, with 1-d actuation, any measurements, any parsers
- **resetArg** (*scalar*) – argument passed to varSwp's actuate procedure to reset and equilibrate
- **nPts** (*int*) – number of strobe points
- **maxDelay** (*float*) – in seconds, delay of strobe. Also the time to soak on reset

Returns the strobe sweep, with accessible data. It can be regathered if needed.

Return type (*NdSweeper*)

Todo: It would be nice to provide timeconstant analysis, perhaps by looking at 50%, or by fitting an exponential

monitorVariable (*fValue*, *sleepSec=0*, *nReps=100*, *plotEvery=1*)

Monitors some process over time. Good for observing drift.

Parameters

- **valueFun** (*function*) – called at each timestep with no arguments. Must return a scalar or a 1-D np.array
- **sleepSec** (*scalar*) – time in seconds to sleep between calls

lightlab.util.config module

Summary

Exceptions:

InvalidOption	
InvalidSection	

Functions:

config_main	
config_save	Save config to a file.
get_config	
get_config_param	
parse_param	
print_config_param	
reset_config_param	
set_config_param	
validate_param	
write_default_config	

Data:

default_config	dict() -> new empty dictionary dict(mapping) -> new dictionary initialized from a mapping object's (key, value) pairs dict(iterable) -> new dictionary initialized as if via: d = {} for k, v in iterable: d[k] = v dict(**kwargs) -> new dictionary initialized with the name=value pairs in the keyword argument list.For example: dict(one=1, two=2).

Reference

write_default_config()

get_config()

parse_param(*param*)

exception InvalidSection
Bases: RuntimeError

exception InvalidOption
Bases: RuntimeError

validate_param(*section*, *option*)

`get_config_param`(*param*)

`print_config_param`(*param*)

`set_config_param`(*param*, *value*)

`reset_config_param`(*param*)

`config_save`(*config*, *omit_default=True*)
Save config to a file. Omits default values if omit_default is True.

`config_main`(*args*)

lightlab.util.gitpath module

All credit goes to https://github.com/MaxNoe/python-gitpath

Summary

Functions:

`abspath`	returns the absolute path for a path given relative to the root of the git repository

Reference

`root`
returns the absolute path of the repository root

`abspath`(*relpath*)
returns the absolute path for a path given relative to the root of the git repository

lightlab.util.measprocessing module

Useful stuff having to do with measurement processing. For example, if you want to set up a spectrum transmission baseline, or a weight functional basis Generally, these states are not device states, but could change from day to day

Summary

Classes:

`SpectrumMeasurementAssistant`	Class for preprocessing measured spectra Calculates background spectra by 1) smoothing, 2) tuning/splicing, and 3) peak nulling Also handles resonance finding (This could move to a separate manager or external function) Interfaces directly with OSA.

Reference

class SpectrumMeasurementAssistant(*nChan=1*, *arePeaks=False*, *osaRef=None*)

Bases: `object`

Class for preprocessing measured spectra Calculates background spectra by 1) smoothing, 2) tuning/splicing, and 3) peak nulling Also handles resonance finding (This could move to a separate manager or external function) Interfaces directly with OSA. It DOES NOT set tuning states.

useBgs = ['tuned', 'smoothed', 'const']

bgSmoothDefault = 2.0

rawSpect(*avgCnt=1*)

fgSpect(*avgCnt=1*, *raw=None*, *bgType=None*)

Returns the current spectrum with background removed.

Also plots so you can see what's going on, if visualize mode was specified

If raw is specified, does not sweep, just removes background

resonances(*spect=None*, *avgCnt=1*)

Returns the current wavelengths of detected peaks in order sorted by wavelength. Uses the simple find-Peaks function, but it could later use a convolutive peak finder for more accuracy. :param spect: if this is specified, then a new spectrum will not be taken

killResonances(*spect=None*, *avgCnt=1*, *fwhmsAround=3.0*)

fgResPlot(*spect=None*, *axis=None*, *avgCnt=1*)

Takes a foreground spectrum, plots it and its peaks. Currently the axis input is unused.

setBgConst(*raw=None*)

Makes a background the maximum transmission observed

setBgSmoothed(*raw=None*, *smoothNm=None*)

Attempts to find background using a low-pass filter. Does not return. Stores results in the assistant variables.

setBgTuned(*base*, *displaced*)

Insert the pieces of the displaced spectrum into where the peaks are It is assumed that these spectra were taken with this object's fgSpect method

setBgNulled(*filtShapes*, *avgCnt=3*)

Uses the peak shape information to null out resonances This gives the best estimate of background INDEPENDENT of the tuning state. It is assumed that the fine background taken by tuning is present, and the filter shapes were taken with that spect should be a foreground spect, but be careful when it is also derived from bgNulled

getBgSpect(*bgType=None*)

lightlab.util.plot module

Summary

Classes:

DynamicLine	A line that can refresh when called

Functions:

plotCovEllipse	Plots an ellipse enclosing *volume* based on the specified covariance matrix (*cov*) and location (*pos*).

Reference

class DynamicLine(*formatStr='b-', existing=None, geometry=[(0, 0), (4, 4)]*)

Bases: `object`

A line that can refresh when called

Parameters

- **formatStr** (*str*) – plotting line format
- **existing** (*Figure/DynamicLine*) – reference to an existing plot to which this DynamicLine instance will be added
- **geometry** (*list[Tuple, Tuple]*) – a 2-element list of 2-tuples of bottom-left (pixels) and width-height (inches)

refresh(*xdata, ydata*)

Refresh the data displayed in the plot

Parameters

- **xdata** (*array*) – X data
- **ydata** (*array*) – Y data

close()

Close the figure window.

Further calls to *refresh()* will cause an error

plotCovEllipse(*cov, pos, volume=0.5, ax=None, **kwargs*)

Plots an ellipse enclosing *volume* based on the specified covariance matrix (*cov*) and location (*pos*). Additional keyword arguments are passed on to the ellipse patch artist.

Parameters

- **cov** – The 2x2 covariance matrix to base the ellipse on
- **pos** – The location of the center of the ellipse. Expects a 2-element sequence of [x0, y0].
- **volume** – The volume inside the ellipse; defaults to 0.5
- **ax** – The axis that the ellipse will be plotted on. Defaults to the current axis.
- **kwargs** – passed to Ellipse plotter

lightlab.util.search module

Searching with actuate-measure functions, usually around peaks and monotonic functions

Summary

Exceptions:

`SearchRangeError`	The first argument is direction, the second is a best guess

Functions:

`binarySearch`	Gives the x where `evalPointFun(x) == targetY`, approximately.
`bracketSearch`	Searches outwards until it finds two X values whose Y values are above and below the targetY.
`doesMFbracket`	
`peakSearch`	Returns the optimal input that gives you the peak, and the peak value
`plotAfterPointMeasurement`	This mutates trackerMF

Reference

exception SearchRangeError

Bases: `lightlab.util.io.errors.RangeError`

The first argument is direction, the second is a best guess

plotAfterPointMeasurement(*trackerMF*, *yTarget=None*)

This mutates trackerMF

Parameters

- **trackerMF** (`MeasuredFunction`) – function that will be plotted
- **yTarget** (*float*) – plotted as dashed line if not None

peakSearch(*evalPointFun*, *startBounds*, *nSwarm=3*, *xTol=0.0*, *yTol=0.0*, *livePlot=False*)

Returns the optimal input that gives you the peak, and the peak value

You must set either xTol or yTol. Be careful with yTol! It is best used with a big swarm. It does not guarantee that you are that close to peak, just that the swarm is that flat

This algorithm is a modified swarm that is robust to outliers, sometimes. Each iteration, it takes <nSwarm> measurements and looks at the best (highest). The update is calculated by shrinking the swarm around the index of the best value. It does not compare between iterations: that makes it robust to one-time outliers. It attributes weight only by order of y values in an iteration, not the value between iterations or the magnitude of differences between y's within an iteration

Not designed to differentiate global vs. local maxima

Parameters

- **evalPointFun** (*function*) – y=f(x) one argument, one return. The function that we want to find the peak of
- **startBounds** (*list, ndarray*) – minimum and maximum x values that bracket the peak of interest
- **nSwarm** (*int*) – number of evaluations per iteration. Use more if it's a narrow peak in a big bounding area
- **xTol** (*float*) – if the swarm x's fall within this range, search returns successfully
- **yTol** (*float*) – if the swarm y's fall within this range, search returns successfully
- **livePlot** (*bool*) – for notebook plotting

Returns best (x,y) point of the peak

Return type (float, float)

doesMFbracket (*targetY*, *twoPointMF*)

bracketSearch (*evalPointFun*, *targetY*, *startBounds*, *xTol*, *hardConstrain=False*, *livePlot=False*)
Searches outwards until it finds two X values whose Y values are above and below the targetY.

Stop conditions

- brackets it: returns new bracketing x values
- step decreases until below xTol: raises RangeError
- 30 iterations: raises RangeError

Parameters

- **evalPointFun** (*function*) – y=f(x) one argument, one return. The function that we want to find the target Y value of
- **startBounds** (*list, ndarray*) – x values that usually do not bracket the value of interest
- **xTol** (*float*) – if *domain* shifts become less than this, raises RangeError
- **hardConstrain** (*bool, list*) – If list, will stay within those
- **livePlot** (*bool*) – for notebook plotting

Returns the bracketing range

Return type ([float, float])

binarySearch (*evalPointFun*, *targetY*, *startBounds*, *hardConstrain=False*, *xTol=0*, *yTol=0*, *livePlot=False*)
Gives the x where `evalPointFun(x) == targetY`, approximately. The final call to evalPointFun will be of this value, so no need to call it again, if your goal is to set to the target.

xTol and yTol are OR-ed conditions. If one is satisfied, it will terminate successfully. You must specify at least one.

Assumes that the function is monotonic in any direction It often works when there is a peak inside the `startBounds`, although not always.

Parameters

- **evalPointFun** (*function*) – y=f(x) one argument, one return. The function that we want to find the target Y value of
- **startBounds** (*list, ndarray*) – minimum and maximum x values that bracket the peak of interest
- **hardConstrain** (*bool, list*) – if not True, will do a bracketSearch. If list, will stay within those
- **xTol** (*float*) – if *domain* shifts become less than this, terminates successfully
- **yTol** (*float*) – if *range* shifts become less than this, terminates successfully
- **livePlot** (*bool*) – for notebook plotting

Returns the optimal X value

Return type (float)

lightlab.util.sweep module

Generalized sweep classes

Summary

Classes:

`Actuation`	
`CommandControlSweeper`	Generic command-control sweep for evaluating a controller.
`NdSweeper`	Generic sweeper.
`Sweeper`	

Functions:

`assertValidPlotType`	
`availablePlots`	Filter by dims and swpType
`loadPickle`	
`plotCmdCtrl`	sweepData should have ALL the command weights specified
`savePickle`	
`simpleSweep`	Basic sweep in one dimension, without function keys, parsing, or plotting.

Data:

`hArrow`	
`hCurves`	
`hEllipse`	
`interAx`	
`pTypes`	dict() -> new empty dictionary dict(mapping) -> new dictionary initialized from a mapping object's (key, value) pairs dict(iterable) -> new dictionary initialized as if via: d = {} for k, v in iterable: d[k] = v dict(**kwargs) -> new dictionary initialized with the name=value pairs in the keyword argument list.For example: dict(one=1, two=2).

Reference

savePickle(*savefile*, *data*, *compress=True*)

loadPickle(*savefile*)

class Sweeper

Bases: `object`

plotOptions = None

monitorOptions = None

gather()

save(*savefile=None*)

Save data only

Parameters **savefile** (*str/Path*) – file to save

load(*savefile=None*)

This is basically make it so that gather() and load() have the same effect.

It does not keep actuation or measurement members, only whatever was put in self.data

Parameters **savefile** (*str/Path*) – file to load

setPlotOptions(***kwargs*)

Valid options for NdSweeper

- plType
- xKey
- yKey
- axArr
- cmap-surf
- cmap-curves

Valid options for CommandControlSweeper

- plType

setMonitorOptions(***kwargs*)

Valid options for NdSweeper

- livePlot
- plotEvery
- stdoutPrint
- runServer

Valid options for CommandControlSweeper

- livePlot
- plotEvery
- stdoutPrint
- runServer
- cmdCtrlPrint

classmethod fromFile(*filename*)

class Actuation(*function=None*, *domain=None*, *doOnEveryPoint=False*)

Bases: object

function = None

domain = None

doOnEveryPoint = None

class NdSweeper

Bases: `lightlab.util.sweep.Sweeper`

Generic sweeper.

Here's the difference between measure and parse:

measure is a call to something, usually an instrument and some simpe post processing, like peak finding.

- It is stored in data
- When subsuming, only unique measurements are kept

parse gets this in a form to visualize interactively, perhaps save and/or score along the way

- When subsuming, all parse functions are maintained

Make sure that measure is *bound* if it is a method

Specify the hard domain and actuate dimensions

The sweep dimension order is major first, so put your slow actuations (e.g. tuning lasers) before the fast actuations (e.g. tuning current source)

Parameters

- **domain** (*tuple, iterable*) – the sweep values, or a tuple of sweep values for different dimensions
- **actuate** (*tuple, procedure-like*) – procedure, one argument per, that is called for each line of the sweep. Return is optional
- **actuNames** (*tuple, str, None*) – Names of actuator return values. These are stored as data if present, under the key ''actuName-return''
- **measure** (*dict*) – dict of functions, no arguments, called at every point. Use descriptive keys please.
- **parse** (*dict*) – dict of functions, operate on measurements, produce scalars Use descriptive keys please.

measure = None

actuate = None

parse = None

static = None

classmethod repeater(*nTrials*)

gather(*soakTime=None, autoSave=False, returnToStart=False*)

Perform the sweep

Parameters

- **soakTime** (*None, float*) – wait this many seconds at the first point to let things settle
- **autoSave** (*bool*) – save data on completion, if savefile is specified
- **returnToStart** (*bool*) – If True, actuates everything to the first point after the sweep completes

Returns None

addActuation(*name*, *function*, *domain*, *doOnEveryPoint=False*)
Specify an actuation dimension: what is called, the domain values to use as arguments.

Parameters

- **name** (*str*) – key for accessing this actuator's value data
- **function** (*func*) – actuation function, usually linked to hardware. One argument.
- **domain** (*ndarray, None*) – 1D array of arguments that will be passed to the function. If None, the function is called with a None argument every point (if doOnEveryPoint is True).
- **doOnEveryPoint** (*bool*) – call this function in the inner loop (True) or once before the corresponding rows(False)

addActuationObject(*name*, *actuationObj*)

reinitActuation()

addMeasurement(*name*, *function*)
Specify a measurement to be taken at every sweep point.

Parameters

- **name** (*str*) – key for accessing this measurement's value data
- **function** (*func*) – measurement function, usually linked to hardware. No arguments.

addParser(*name*, *function*)
Adds additional parsing formulas to data, and reparses, if data has been taken

Parameters

- **name** (*str*) – key for accessing this parser's value data
- **function** (*func*) – parsing function, not linked to hardware. One dictionary argument.

addStaticData(*name*, *contents*)
Add a ndarray or scalar that can be referenced by parsers

The array's shape must match that of the currently loaded actuation grid.

If you then `addActuation()`, the static data automatically expands in dimension to accomodate. Values are filled by tiling in the new dimension.

Add static data after the actuations that have different static data, but before the actuations for which you want that data to be constant.

Parameters

- **name** (*str*) – key for accessing this data
- **contents** (*scalar, ndarray*) – data contents

subsume(*other*, *useMinorOptions=False*)
Makes the argument sweep a minor sweep within this one

The new measurement dictionary will contain all measurements of both. If there is a duplicate key, the self measurement will take precedence

Existing data is discarded.

Parameters

- **other** (NdSweeper) – the minor sweep

- **useMinorOptions** (*bool*) – where do the options come from? If False, they come from the major (i.e. self)

copy (*includeData=True*)

Shallow copy, which means function pointers are maintained

If includeData, it does a deep copy of data

plot (*slicer=None*, *tempData=None*, *index=None*, *axArr=None*, *pltKwargs=None*)

Plots

Much of the behavior to figure out labels and numbers for axes comes from the plotOptions attribute.

The xKeys and yKeys are keys within this objects data dictionary (actuation, measurement, and parsers)
The total number of plots will be the product of len('xKey') and len('yKey'). xKeys can be anything, including parsed data members. By default it is the minor actuation variable yKeys can also be anything that has scalar elements. By default it is everything that is currently present, except xKeys and non-scalars

When doing line plots in 2D sweeps, the legend does automatic labelling.

Each line must correspond to an actuation dimension, otherwise it doesn't make sense. This is despite the fact that the xKeys can still be anything.

Usually, each line corresponds to a particular domain value of the major sweep axis; however, if that is specified as an xKey, the lines will correspond to the minor axis.

Surface plotting: Ignores whatever is in xKeys. The plotting domain is locked to the actuation domain in order to keep a rectangular grid. The values indicated in yKeys will become color data.

Parameters

- **slicer** (*tuple, slice*) – domain slices
- **axArr** (*ndarray), plt.axis*) – axes to plot on. Equivalent to what is returned by this method
- **pltKwargs** – passed through to plotting function

Todo:

- Graphics caching for 2D line plots

saveObj (*savefile=None*)

Also saves what are the actuation keys. This is important for plotting when you reload

classmethod loadObj (*savefile*, *functionSource=None*)

savefile must have been saved with saveObj. It restores actuation names and domains to help with plotting.

Functions referring to actuation and measurement cannot be saved.

functionSource: an instantiated object of class *cls* If you give it a functionSource, then those can be restored as well. This is very useful if you have a parser such as live plot spectra, or move stuff here or there. Also useful if you want to re-gather for some reason.

load (*savefile=None*)

This is basically make it so that gather() and load() have the same effect.

It does not keep actuation or measurement members, only whatever was put in self.data

Parameters **savefile** (*str/Path*) – file to load

simpleSweep(*actuate*, *domain*, *measure=None*)

Basic sweep in one dimension, without function keys, parsing, or plotting.

Parameters

- **actuate** (*function*) – a procedure or function of one argument called at every point
- **domain** (*ndarray*) – elements passed as an argument to actuate for each point
- **measure** (*function*, *None*) – a function of no arguments called at every point. None means the return of actuate will act as the measurement

Returns what is measured. Same length as domain

Return type (ndarray)

class CommandControlSweeper(*evaluate*, *defaultArg*, *swpInds*, *domain*, *nTrials=1*)

Bases: *lightlab.util.sweep.Sweeper*

Generic command-control sweep for evaluating a controller.

The command function called at each point takes one argument that is an array (length M) and returns an array **of equal length**.

The sweep is N (<= M) dimensional.

- The user specifies the mapping between the sweep domain and the argument/return array indeces
- The user specifies defaults for the other (M-N) arguments
- Some of the uncontrolled arguments can be monitored

Todo: How can we get this subsumed by a NdSweeper for trial repetition. CommandControlSweeper shouldn't be able to subsume as major

Parameters

- **evaluate** (*function*) – called at each point with array args/returns of equal length
- **defaultArg** (*ndarray*) – default value that will be sent to the evaluate function
- **swpIndeces** (*tuple*, *int*) – which channels to sweep
- **domain** (*tuple*, *iterable*) – the values over which the sweep channels will be swept

saveObj(*savefile=None*)

Instead of just saving data, save the whole damn thing.

Cannot save evaluate function because it is unbound.

classmethod loadObj(*savefile*)

This is basically make it so that gather() and load() have the same effect.

It does not keep actuation or measurement members, only whatever was put in self.data

gather(*autoSave=False*, *randomize=False*)

Executes the sweep

Todo: Store all outputs, but provide a way just to get the controlled ones

toSweepData()
Using the old school temporary definition from conductor

This will eventually be deprecated

plot(*index=None*, *axArr=None*)

score(*bits=False*, *worstCase=False*)
Takes full sweep data and returns the worst-case accuracy and precision

Parameters

- **bits** (*bool*) – if true, returns values as bits of dynamic range
- **worstCase** (*bool*) – if true, takes the performance at the worst weight, else averages via RMS

plotCmdCtrl(*sweepData*, *index=None*, *ax=None*, *interactive=False*)
sweepData should have ALL the command weights specified

Parameters

- **sweepData** (*tuple*) – cmdWeights, measWeights, monitWeights (optional) measWeights has shape (nTrials, len(swp1), len(sp2) or 1, len(sweepingChannels))
- **index** (*tuple*) – tells which parts of measured weights are valid. If None, assumes sweepData is complete
- **interactive** (*bool*) – show plot immediately after call, even with incomplete data (index must be specified)

Todo: Fix the global hack for persistent plots – actually, this is fine

availablePlots(*dims=None*, *swpType=None*)
Filter by dims and swpType

If the argument is none, do not filter by that

assertValidPlotType(*plType*, *dims=None*, *swpClass=None*)

Subpackages:

lightlab.util.data package

Useful stuff having to do with data handling and processing.

one_dim.MeasuredFunction is the workhorse.

The *Spectrum* class is nice for working with dbm and linear units

peaks.findPeaks() and *function_inversion.descend()* hold the low-level algorithms. Usually, users would interact with it via *MeasuredFunction*.

Submodules:

lightlab.util.data.basic module

Argument sanitizing and very basic array operations

Summary

Functions:

argFlatten	Takes a combination of multiple arguments and flattens the ones of type typs.
mangle	Sanitizes attribute names that might be “hidden,” denoted by leading ‘__’.
minmax	Returns a list of [min and max] of the array
rms	
verifyListOfType	Checks to see if the argument is a list or a single object of the checkType Returns a list, even if it is length one If arg is None, it returns None

Data:

MANGLE_LEN	int(x=0) -> integer int(x, base=10) -> integer

Reference

verifyListOfType(*arg*, *checkType*)

Checks to see if the argument is a list or a single object of the checkType Returns a list, even if it is length one If arg is None, it returns None

argFlatten(**argLists*, *typs=(<class 'list'>, <class 'tuple'>, <class 'set'>)*)

Takes a combination of multiple arguments and flattens the ones of type typs. None arguments are ignored, no error.

Parameters

- ***argLists** – multiple arguments that could be lists or tuples
- **typs** (*tuple*) – types of things to flatten

Returns (tuple)

It goes like this:

```
dUtil.argFlatten()                                          # == ()
dUtil.argFlatten(1)                                         # == (1,)
dUtil.argFlatten((3, 4))                                    # == (3, 4)
dUtil.argFlatten(1, (3, 4), np.zeros(2))                    # == (1, 3, 4,␣
↪ndarray([0,0]))
dUtil.argFlatten(1, [3, 4], np.zeros(2))                    # == (1, 3, 4,␣
↪ndarray([0,0]))
dUtil.argFlatten(1, [3, 4], np.zeros(2), typs=tuple)        # == (1, [3, 4],␣
↪ndarray([0,0]))
dUtil.argFlatten(1, [3, 4], np.zeros(2), typs=np.ndarray) # == (1, [3, 4], 0., 0.)
```

mangle(*name*, *klass*)

Sanitizes attribute names that might be “hidden,” denoted by leading ‘__’. In *Hashable* objects, attributes with this kind of name can only be class attributes.

See `test_instrument_overloading` for user-side implications.

Behavior:

```
mangle('a', 'B') == 'a'
mangle('_a', 'B') == '_a'
mangle('__a__', 'B') == '__a__'
mangle('__a', 'B') == '_B__a'
mangle('__a', '_B') == '_B__a'
```

rms (*diffArr*, *axis=0*)

minmax (*arr*)
Returns a list of [min and max] of the array

lightlab.util.data.function_inversion module

Finding the x-value that provides a targeted y-value for measured functions

Summary

Functions:

descend	From the start index, descend until reaching a threshold level and return that index If it runs into the invalidIndeces or an edge, returns i at the edge of validity and False for validPeak
interpInverse	Gives a float representing the interpolated x value that gives y=threshVal

Reference

descend (*yArr*, *invalidIndeces*, *startIndex*, *direction*, *threshVal*)
From the start index, descend until reaching a threshold level and return that index If it runs into the invalidIndeces or an edge, returns i at the edge of validity and False for validPeak

interpInverse (*xArrIn*, *yArrIn*, *startIndex*, *direction*, *threshVal*)
Gives a float representing the interpolated x value that gives y=threshVal

lightlab.util.data.one_dim module

One-dimensional data structures with substantial processing abilities

Summary

Classes:

MeasuredFunction	Array of x,y points.
Spectrum	Adds handling of linear/dbm units.
SpectrumGHz	Spectrum with GHz units in the abscissa
Waveform	Typically used for time, voltage functions.

Functions:

prbs_generator	Generator of PRBS bits.
prbs_pattern	Returns an array containing a sequence of a PRBS pattern.

Reference

prbs_generator(*characteristic*, *state*)

Generator of PRBS bits.

Example: polynomial = 0b1000010001 # 1 + X^5 + X^9 seed = 0b111100000

The above parameters will give you a PRBS9 bit sequence. Note: it might be inverted compared to the official definition, i.e., 1s are 0s and vice versa.

prbs_pattern(*polynomial*, *seed*, *length=None*)

Returns an array containing a sequence of a PRBS pattern.

If length is not set, the sequence will be 2^n-1 long, corresponding to the repeating pattern of the PRBS sequence.

class MeasuredFunction(*abscissaPoints*, *ordinatePoints*, *unsafe=False*)

Bases: `object`

Array of x,y points. This is the workhorse class of `lightlab` data structures. Examples can be found throughout Test notebooks.

Supports many kinds of operations:

1. **Data access (`mf(x)`, `len(mf)`, `mf[i]`, *`getData()`*)** Calling the object with x-values interpolates and returns y-values.
2. **Storage (*`copy()`*, *`save()`*, *`load()`*, `loadFromFile()`)** see method docstrings
3. **x-axis signal processing (*`getSpan()`*, *`crop()`*, *`shift()`*, *`flip()`*, *`resample()`*, *`uniformlySample()`*)** see method docstrings
4. **y-axis signal processing (*`getRange()`*, *`clip()`*, *`debias()`*, *`unitRms()`*, *`getMean()`*, *`moment()`*)** see method docstrings
5. **Advanced signal processing (*`invert()`*, *`lowPass()`*, *`centerOfMass()`*, *`findResonanceFeatures()`*)** see method docstrings
6. **Binary math (+, −, *, /, ==)**

   Operands must be either

   - the same subclass of MeasuredFunction, or
   - scalar numbers, or
   - functions/bound methods: these must be callable with one argument that is an ndarray

   If both are MeasuredFunction, the domain used will be the smaller of the two
7. **Plotting (*`simplePlot()`*)** Args and Kwargs are passed to pyplot's plot function. Supports live plotting for notebooks
8. **Others (*`deleteSegment()`*, *`splice()`*)** see method docstrings

Parameters

- **abscissaPoints** (*array*) – abscissa, a.k.a. independent variable, a.k.a. domain
- **ordinatePoints** (*array*) – ordinate, a.k.a. dependent variable, a.k.a. range

- **unsafe** (*bool*) – if True, faster, give it 1-D np.ndarrays of the same length, or you will get weird errors later on

absc = None
abscissa, a.k.a. the x-values or domain

ordi = None
ordinate, a.k.a. the y-values

getData()
Gives a tuple of the enclosed array data.

It is copied, so you can do what you want with it

Returns the enclosed data

Return type tuple(array,array)

copy()
Gives a copy, so that further operations can be performed without side effect.

Returns new object with same properties

Return type (MeasuredFunction/<childClass>)

save(*savefile*)

classmethod load(*savefile*)

simplePlot(**args*, *livePlot=False*, ***kwargs*)
Plots on the current axis

Parameters

- **livePlot** (*bool*) – if True, displays immediately in IPython notebook
- ***args** (*tuple*) – arguments passed through to `pyplot.plot`
- ****kwargs** (*dict*) – arguments passed through to `pyplot.plot`

Returns Whatever is returned by `pyplot.plot`

subsample(*newAbscissa*)
Returns a new MeasuredFunction sampled at given points.

getSpan()
The span of the domain

Returns the minimum and maximum abscissa points

Return type (list[float,float])

abs()
Computes the absolute value of the measured function.

mean()

max()
Returns the maximum value of the ordinate axis, ignoring NaNs.

argmax()
Returns the abscissa value at which the ordinate is maximum.

min()
Returns the minimum value of the ordinate axis, ignoring NaNs.

argmin()
Returns the abscissa value at which the ordinate is minimum.

getRange()
The span of the ordinate

Returns the minimum and maximum values

Return type (list[float,float])

crop(*segment*)
Crop abscissa to segment domain.

Parameters **segment** (*list[float, float]*) – the span of the new abscissa domain

Returns new object

Return type *MeasuredFunction*

clip(*amin*, *amax*)
Clip ordinate to min/max range

Parameters

- **amin** (*float*) – minimum value allowed in the new MeasuredFunction
- **amax** (*float*) – maximum value allowed in the new MeasuredFunction

Returns new object

Return type *MeasuredFunction*

shift(*shiftBy*)
Shift abscissa. Good for biasing wavelengths.

Parameters **shiftBy** (*float*) – the number that will be added to the abscissa

Returns new object

Return type *MeasuredFunction*

flip()
Flips the abscissa, BUT DOES NOTHING the ordinate.

Usually, this is meant for spectra centered at zero. I.e.: flipping would be the same as negating abscissa

Returns new object

Return type *MeasuredFunction*

reverse()
Flips the ordinate, keeping abscissa in order

Returns new object

Return type *MeasuredFunction*

debias()
Removes mean from the function

Returns new object

Return type *MeasuredFunction*

unitRms()
Returns function with unit RMS or power

getMean()

getMedian()

getVariance()

getStd()

resample(*nsamp=100*)
Resample over the same domain span, but with a different number of points.

Parameters **nsamp** (*int*) – number of samples in the new object

Returns new object

Return type *MeasuredFunction*

uniformlySample()
Makes sure samples are uniform

Returns new object

Return type *MeasuredFunction*

addPoint(*xyPoint*)
Adds the (x, y) point to the stored absc and ordi

Parameters **xyPoint** (*tuple*) – x and y values to be inserted

Returns it modifies this object

Return type None

correlate(*other*)
Correlate signals with scipy.signal.correlate.

Only full mode and direct method is supported for now.

lowPass(*windowWidth=None*, *mode=None*)

movingAverage(*windowWidth=None*, *mode='valid'*)
Low pass filter performed by convolving a moving average window.

The convolutional `mode` can be one of the following string tokens

- 'valid': the new span is reduced, but data is good looking
- 'same': new span is the same as before, but there are edge artifacts

Parameters

- **windowWidth** (*float*) – averaging window width in units of the abscissa
- **mode** (*str*) – convolutional mode

Returns new object

Return type *MeasuredFunction*

butterworthFilter(*fc*, *order*, *btype*)
Applies a Butterworth filter to the signal.

Side effects: the waveform will be resampled to have equally-sampled points.

Parameters **fc** (*float*) – cutoff frequency of the filter (cf. input to signal.butter)

Returns New object containing the filtered waveform

lowPassButterworth(*fc*, *order=1*)

Applies a low-pass Butterworth filter to the signal.

Side effects: the waveform will be resampled to have equally-sampled points.

Parameters **fc** (*float*) – cutoff frequency of the filter

Returns New object containing the filtered waveform

highPassButterworth(*fc*, *order=1*)

Applies a high-pass Butterworth filter to the signal.

Side effects: the waveform will be resampled to have equally-sampled points.

Parameters **fc** (*float*) – cutoff frequency of the filter

Returns New object containing the filtered waveform

bandPassButterworth(*fc*, *order=1*)

Applies a high-pass Butterworth filter to the signal.

Side effects: the waveform will be resampled to have equally-sampled points.

Parameters **fc** (*length-2 float sequence*) – cutoff frequency of the filter

Returns New object containing the filtered waveform

deleteSegment(*segment*)

Removes the specified segment from the abscissa.

This means calling within this segment will give the first-order interpolation of its edges.

Usually, deleting is followed by splicing in some new data in this span

Parameters **segment** (*list[float, float]*) – span over which to delete stored points

Returns new object

Return type *MeasuredFunction*

splice(*other*, *segment=None*)

Returns a Spectrum that is this one, except with the segment replaced with the other one's data

The abscissa of the other matters. There is nothing changing (abscissa, ordinate) point pairs, only moving them around from `other` to `self`.

If segment is not specified, uses the full domain of the other

Parameters

- **other** (MeasuredFunction) – the origin of new data
- **segment** (*list[float, float]*) – span over which to do splice stored points

Returns new object

Return type *MeasuredFunction*

invert(*yVals*, *directionToDescend=None*)

Descends down the function until yVal is reached in ordi. Returns the absc value

If the function is peaked, you should specify a direction to descend.

If the function is approximately monotonic, don't worry about it.

Parameters

- **yVals** (*scalar, ndarray*) – array of y values to descend to

- **directionToDescend** (*['left', 'right', None]*) – use if peaked function to tell which side. Not used if monotonic

Returns corresponding x values

Return type (scalar, ndarray)

centerOfMass()
Returns abscissa point where mass is centered

moment(*order=2*, *relativeGauss=False*)
The order'th moment of the function

Parameters **order** (*integer*) – the polynomial moment of inertia. Don't trust the normalization of > 2'th order. order = 1: mean order = 2: variance order = 3: skew order = 4: kurtosis

Returns the specified moment

Return type (float)

findResonanceFeatures(***kwargs*)
A convenient wrapper for *findPeaks()*

Parameters ****kwargs** – passed to *findPeaks()*

Returns the detected features as nice objects

Return type list[*ResonanceFeature*]

norm(*ord=None*)

class Spectrum(*nm*, *power*, *inDbm=True*, *unsafe=False*)
Bases: *lightlab.util.data.one_dim.MeasuredFunction*

Adds handling of linear/dbm units.

Use *lin()* and dbm() to make sure what you're getting what you expect for things like binary math and peakfinding, etc.

Parameters

- **nm** (*array*) – abscissa
- **power** (*array*) – ordinate
- **inDbm** (*bool*) – is the power in linear or dbm units?

inDbm
Is it in dbm units currently?

Returns

Return type bool

lin()
The spectrum in linear units

Returns new object

Return type *Spectrum*

db()
The spectrum in decibel units

Returns new object

Return type *Spectrum*

simplePlot(**args*, *livePlot=False*, ***kwargs*)
More often then not, this is db vs. wavelength, so label it

refineResonanceWavelengths(*filtShapes*, *seedRes=None*, *isPeak=None*)
Convolutional resonance correction to get very robust resonance wavelengths

Does the resonance finding itself, unless an initial approximation is provided.

Also, has some special options for `Spectrum` types to make sure db/lin is optimal

Parameters

- **filtShapes** (*list* [MeasuredFunction]) – shapes of each resonance. Must be in order of ascending abscissa/wavelength
- **seedRes** (*list* [ResonanceFeature]) – rough approximation of resonance properties. If None, this method will find them.
- **isPeak** (*bool*) – required to do peak finding, but not used if `seedRes` is specified

Returns the detected and refined features as nice objects

Return type list[*ResonanceFeature*]

Todo: take advantage of fft convolution for speed

findResonanceFeatures(***kwargs*)
Overloads *MeasuredFunction.findResonanceFeatures()* to make sure it's in db scale

Parameters ****kwargs** – kwargs passed to *findPeaks*

Returns the detected features as nice objects

Return type list[*ResonanceFeature*]

GHz()
Convert to SpectrumGHz

class SpectrumGHz(*GHz*, *power*, *inDbm=True*, *unsafe=False*)
Bases: *lightlab.util.data.one_dim.Spectrum*

Spectrum with GHz units in the abscissa

Use `lin()` and `dbm()` to make sure what you're getting what you expect for things like binary math and peakfinding, etc.

Parameters

- **GHz** (*array*) – abscissa
- **power** (*array*) – ordinate
- **inDbm** (*bool*) – is the `power` in linear or dbm units?

simplePlot(**args*, *livePlot=False*, ***kwargs*)
More often then not, this is db vs. wavelength, so label it

nm()
Convert to Spectrum

class Waveform(*t*, *v*, *unit='V'*, *unsafe=False*)
Bases: *lightlab.util.data.one_dim.MeasuredFunction*

Typically used for time, voltage functions. This is very similar to what is referred to as a "signal."

Use the unit attribute to set units different than Volts.

Has class methods for generating common time-domain signals

unit = None

classmethod pulse(*tArr*, *tOn*, *tOff*)

classmethod whiteNoise(*tArr*, *rmsPow*)

lightlab.util.data.peaks module

Implementation of core peak finding algorithm. It is wrapped to be more user-friendly by `findResonanceFeatures()`.

`ResonanceFeature` is a data storage class returned by `findResonanceFeatures()`

Summary

Exceptions:

`PeakFinderError`	

Classes:

`ResonanceFeature`	A data holder for resonance features (i.e.

Functions:

`findPeaks`	Takes an array and finds a specified number of peaks

Reference

class ResonanceFeature(*lam*, *fwhm*, *amp*, *isPeak=True*)

Bases: `object`

A data holder for resonance features (i.e. peaks or dips)

lam

float – center wavelength

fwhm

float – full width half maximum – can be less if the extinction depth is less than half

amp

float – peak amplitude

isPeak

float – is it a peak or a dip

copy()

Simple copy so you can modify without side effect

Returns new object

Return type *ResonanceFeature*

simplePlot(**args*, ***kwargs*)

Plots a box to visualize the resonance feature

The box is centered on the peak `lam` and `amp` with a width of `fwhm`.

Parameters

- ***args** – args passed to `pyplot.plot`
- ****kwargs** – kwargs passed to `pyplot.plot`

Returns whatever `pyplot.plot` returns

exception PeakFinderError

Bases: `RuntimeError`

findPeaks(*yArrIn*, *isPeak=True*, *isDb=False*, *expectedCnt=1*, *descendMin=1*, *descendMax=3*, *minSep=0*)

Takes an array and finds a specified number of peaks

Looks for maxima/minima that are separated from others, and stops after finding `expectedCnt`

Parameters

- **isDb** (*bool*) – treats dips like DB dips, so their width is relative to outside the peak, not inside
- **descendMin** (*float*) – minimum amount to descend to be classified as a peak
- **descendMax** (*float*) – amount to descend down from the peaks to get the width (i.e. FWHM is default)
- **minSep** (*int*) – the minimum spacing between two peaks, in array index units

Returns indeces of peaks, sorted from biggest peak to smallest peak array (float): width of peaks, in array index units

Return type array (float)

Raises `Exception` – if not enough peaks found. This plots on fail, so you can see what's going on

lightlab.util.data.two_dim module

Two dimensional measured objects where the second abscissa variable is either

- discrete (*FunctionBundle*), or
- continuous (*MeasuredSurface*)

Summary

Classes:

FunctionBundle	A bundle of *MeasuredFunction*'s: "z" vs.
FunctionalBasis	A FunctionBundle that supports additional linear algebra methods
MeasuredErrorField	A field that hold two abscissa arrays and two ordinate matrices
MeasuredSurface	Basically a two dimensional measured function: "z" vs.

Continued on next page

Table 78 – continued from previous page

Spectrogram	**param absc** same meaning as measured function

Reference

class FunctionBundle(*measFunList=None*)

Bases: *lightlab.laboratory.Hashable*

A bundle of *MeasuredFunction*'s: "z" vs. "x", "i"

The key is that they have the same abscissa base. This class will take care of resampling in a common abscissa base.

The bundle can be:

- iterated to get the individual :class'~lightlab.util.data.one_dim.MeasuredFunction''s
- operated on with other `FunctionBundles`
- plotted with :meth'simplePlot' and *multiAxisPlot()*

Feeds through **callable** signal processing methods to its members (type MeasuredFunction), If the method is not found in the FunctionBundle, and it is in it's member, it will be mapped to every function in the bundle, returning a new bundle.

Distinct from a *MeasuredSurface* because the additional axis does not represent a continuous thing. It is discrete and sometimes unordered.

Distinct from a *FunctionalBasis* because it does not support most linear algebra-like stuff (e.g. decomposision, matrix multiplication, etc.). This is not a strict rule.

Can be initialized fully, or initialized with None to be built interactively.

Parameters measFunList (*list[MeasuredFunction] or None*) – list of MeasuredFunctions that must have the same abscissa.

addDim(*newMeasFun*)

copy()

extend(*otherFunctionBund*)

max()

Returns a single MeasuredFunction(subclass) that is the maximum of all in this bundle

min()

Returns a single MeasuredFunction(subclass) that is the minimum of all in this bundle

mean()

Returns a single MeasuredFunction(subclass) that is the mean of all in this bundle

simplePlot(**args*, ***kwargs*)

multiAxisPlot(**args*, *axList=None*, *titleRoot=None*, ***kwargs*)

titleRoot must take one argument in its format method, which is given the index :returns: The axes that were plotted upon :rtype: (list(axis))

histogram()

Gives a MeasuredFunction of counts vs. ordinate values (typically voltage) Does not maintain any abscissa information

At this point, does not allow caller to set the arguments passed to np.histogram

This is mainly just for plotting

weightedAddition(*weiVec*)
Calculates the weighted addition of the basis signals

Parameters **weiVec** (*array*) – weights to be applied to the basis functions

Returns weighted addition of basis signals

Return type (*MeasuredFunction*)

moment(*order=2*, *allDims=True*, *relativeGauss=False*)
The order'th moment of all the points in the bundle.

Parameters

- **order** (*integer*) – the polynomial moment of inertia. Don't trust the normalization of > 2'th order. order = 1: mean order = 2: variance order = 3: skew order = 4: kurtosis
- **allDims** (*bool*) – if true, collapses all signals, returning a scalar

Returns the specified moment(s)

Return type (ndarray or float)

componentAnalysis(**args*, *pcaIca=True*, *lNorm=2*, *expectedComponents=None*, ***kwargs*)
Gives the waveform representing the principal component of the order

Parameters

- **pcaIca** (*bool*) – if True, does PCA; if False, does ICA
- **lNorm** (*int*) – how to normalize weight vectors. L1 norm uses the maximum abs weight, while L2 norm (default) is vector unit
- **expectedComponents** (*FunctionBundle or subclass*) – Used for flipping signs
- **kwargs** (*args,*) – Feed through to sklearn.decomposition.[PCA(), FastICA()]

Returns principal component waveforms

Return type (*FunctionBundle* or subclass)

correctSigns(*otherBundle*, *maintainOrder=True*)
Goes through each component and flips the sign if correlation is negative

ICA also has a permutation indeterminism.

class FunctionalBasis(*measFunList=None*)
Bases: *lightlab.util.data.two_dim.FunctionBundle*

A FunctionBundle that supports additional linear algebra methods

Created for weighted addition, decomposition, and component analysis

Can be initialized fully, or initialized with None to be built interactively.

Parameters **measFunList** (*list[MeasuredFunction] or None*) – list of Measured-Functions that must have the same abscissa.

classmethod independentDefault(*nDims*)
Gives a basis of non-overlapping pulses. Waveforms only

innerProds(*trial*)
takes the inner products of the trial function onto this basis.

magnitudes()
The inner product of the basis with itself

project(*trial*)
Projects onto normalized basis If the basis is orthogonal, this is equivalent to weight decomposition

decompose(*trial*, *moment=1*)
Uses the Moore-Penrose pseudoinverse to get weight decomposition without orthogonality

Parameters

- **trial** (`MeasuredFunction`) – signal to be decomposed
- **moment** (*`float`*) – polynomial moment of the basis to use when decomposing

matrixMultiply(*weiMat*)

getMoment(*weiVecs=None*, *order=2*, *relativeGauss=False*)
This is actually the projected moment. Named for compatibility with bss package

Make sure weiVecs is two dimensional

remainder(*trial*)
Gives the remaining parts of the signal that are not explained by the minimum-squared-error decomposition

covariance()
Returns covariance matrix of the basis, which is nDims x nDims

class MeasuredSurface(*absc*, *ordi*)
Bases: `object`

Basically a two dimensional measured function: “z” vs. “x”, “y”

Useful trick when gathering data: build incrementally using *`FunctionBundle.addDim()`*, then convert that to this class using *`MeasuredSurface.fromFunctionBundle()`*.

Parameters

- **absc** (*`ndarray`*) – same meaning as measured function
- **ordi** (*`ndarray`*) – two-dimensional array or matrix

classmethod fromFunctionBundle(*otherBund*, *addedAbsc=None*)
gives back a MeasuredSurface from a function Bundle

Parameters

- **otherBund** (`FunctionBundle`) – The source. The ordering of functions matters
- **addedAbsc** (*`np.ndarray`*) – the second dimension abscissa array (default, integers)

Returns (*`MeasuredSurface`*) new object

item(*index*, *dim=None*)

shape()

simplePlot(**args*, ***kwargs*)

class Spectrogram(*absc*, *ordi*)
Bases: *`lightlab.util.data.two_dim.MeasuredSurface`*

Parameters

- **absc** (*`ndarray`*) – same meaning as measured function
- **ordi** (*`ndarray`*) – two-dimensional array or matrix

class MeasuredErrorField(*nominalGrid*, *measuredGrid*)

Bases: `object`

A field that hold two abscissa arrays and two ordinate matrices

Error is the measuredGrid - nominalGrid, which is a vector field

errorAt(*testVec=None*)

invert(*desiredVec*)

zeroCenteredSquareSize()

Very stupid, just look at corner points

Returns square sides of nominal and measured grids

Return type (tuple(float))

lightlab.util.io package

Functions for filesystem handling

Submodules:

lightlab.util.io.errors module

Summary

Exceptions:

ChannelError	
DeprecatedError	Make sure to describe the new alternative
RangeError	It is useful to put the type of error ‘high’ or ‘low’ in the second argument of this class’ initializer

Reference

exception ChannelError

Bases: `Exception`

exception RangeError

Bases: `Exception`

It is useful to put the type of error ‘high’ or ‘low’ in the second argument of this class’ initializer

exception DeprecatedError

Bases: `Exception`

Make sure to describe the new alternative

lightlab.util.io.jsonpickleable module

Objects that can be serialized in a (sort of) human readable json format

Tested in `tests.test_JSONpickleable`.

Summary

Classes:

`HardwareReference`	Spoofs an instrument
`JSONpickleable`	Produces human readable json files.

Reference

class HardwareReference(*klassname*)

Bases: `object`

Spoofs an instrument

open()

class JSONpickleable(***kwargs*)

Bases: `lightlab.laboratory.Hashable`

Produces human readable json files. Inherits _toJSON from Hashable Automatically strips attributes beginning with __.

notPickled

set – names of attributes that will be guaranteed to exist in instances. They will not go into the pickled string. Good for references to things like hardware instruments that you should re-init when reloading.

See the test_JSONpickleable for much more detail

What is not pickled?

1. attributes with names in `notPickled`
2. attributes starting with __
3. VISAObjects: they are replaced with a placeholder HardwareReference
4. bound methods (not checked, will error if you try)

What functions can be pickled

1. module-level, such as np.linspace
2. lambdas

Todo: This should support unbound methods

Args: filepath (str/Path): path string to file to save to

notPickled = set()

copy()

This will throw out hardware references and anything starting with __

Good test for what will be saved

save(*filename*)

classmethod load(*filename*)

lightlab.util.io.paths module

Resolves several directories as follows. These can be overridden after import if desired.

1. **`projectDir`** The git repo of the file that first imported `io`
2. **`dataHome` = (default) `projectDir / "data"`** Where all your data is saved.
3. **`fileDir` = (default) `dataHome`** Where all the save/load functions will look. Usually this is set differently from notebook to notebook.
4. **`monitorDir` = (default) `projectDir / "progress-monitor"`** Where html for sweep progress monitoring will be written by `ProgressWriter`.
5. **`lightlabDevelopmentDir`** The path to a source directory of `lightlab` for development. It is found through the ".pathtolightlab" file. This is currently unused.

lightlab.util.io.progress module

Some utility functions for printing to stdout used in the project

Also contains web-based progress monitoring

Summary

Classes:

`ProgressWriter`	Writes progress to an html file for long sweeps.

Functions:

`printProgress`	Deletes current line and writes.
`printWait`	Prints your message followed by `...`
`ptag`	

Reference

`printWait`(**args*)

Prints your message followed by `...`

This displays immediately, but

- your next print will show up on the same line

Parameters **`*args`** (*`Tuple(str)`*) – Strings that will be written

`printProgress`(**args*)

Deletes current line and writes.

This is used for updating iterating values so to not produce a ton of output

Parameters **`*args`** (*`str, Tuple(str)`*) – Arguments that will be written

`class ProgressWriter`(*name*, *swpSize*, *runServer=True*, *stdoutPrint=False*, ***kwargs*)

Bases: `object`

Writes progress to an html file for long sweeps. Including timestamps. Has an init and an update method

You can then open this file to the internet by running a HTTP server.

To setup a continuously running server:

```
screen -S sweepProgressServer
(Enter)
cd /home/atait/Documents/calibration-instrumentation/sweepMonitorServer/
python3 -m http.server 8050
(Ctrl-a, d)
```

To then access from a web browser:: http://lightwave-lab-olympias.princeton.edu:8050

Todo: Have this class launch its own process server upon init Make it so you can specify actuator names

Parameters

- **name** (*str*) – name to be displayed
- **swpSize** (*tuple*) – size of each dimension of the sweep

progFileDefault = PosixPath('/home/docs/checkouts/readthedocs.org/user_builds/lightlab

tFmt = '%a, %d %b %Y %H:%M:%S'

static getUrl()
URL where the progress monitor will be hosted

update(*steps=1*)

classmethod tims(*epochTime*)

ptag(*s*)

lightlab.util.io.saveload module

Summary

Functions:

loadMat	returns a dictionary of data.
loadPickle	Uses pickle
loadPickleGzip	Uses pickle and then gzips the file.
pprintFileDir	Prints the contents of io.fileDir.
saveFigure	if None, uses the gcf()
saveMat	dataDict has keys as names you would like to appear in matlab, values are numpy arrays, N-D arrays, or matrices.
savePickle	Uses pickle
savePickleGzip	Uses pickle

Reference

pprintFileDir (*, *generate=False*)

Prints the contents of io.fileDir. If the file can be loaded by this module, it gives the command to do so.

Returns A sorted list of files

savePickle (*filename*, *dataTuple*)

Uses pickle

Parameters

- **filename** (`str, Path`) – file to write to
- **dataTuple** (`tuple`) – tuple containing almost anything

loadPickle (*filename*)

Uses pickle

savePickleGzip (*filename*, *dataTuple*)

Uses pickle

Parameters

- **filename** (`str, Path`) – file to write to
- **dataTuple** (`tuple`) – tuple containing almost anything

loadPickleGzip (*filename*)

Uses pickle and then gzips the file.

If it is named file.abc.gz, loads as file.abc.gz If it is named file.abc, loads as file.abc.pkl

saveMat (*filename*, *dataDict*)

dataDict has keys as names you would like to appear in matlab, values are numpy arrays, N-D arrays, or matrices.

loadMat (*filename*)

returns a dictionary of data. This should perfectly invert saveMat. Matlab files only store matrices. This auto-squeezes 1-dimensional matrices to arrays. Be careful if you are tyring to load a 1-d numpy matrix as an actual numpy matrix

saveFigure (*filename*, *figHandle=None*)

if None, uses the gcf()

3.1.5 Summary

Functions:

`log_to_screen`
`log_visa_to_screen`

Data:

CRITICAL	int(x=0) -> integer int(x, base=10) -> integer
DEBUG	int(x=0) -> integer int(x, base=10) -> integer
ERROR	int(x=0) -> integer int(x, base=10) -> integer
INFO	int(x=0) -> integer int(x, base=10) -> integer
NOTSET	int(x=0) -> integer int(x, base=10) -> integer

Continued on next page

Table 85 – continued from previous page

`WARNING`	int(x=0) -> integer int(x, base=10) -> integer

3.1.6 Reference

`log_to_screen`(*level=20*)

`log_visa_to_screen`(*level=30*)

3.2 tests package

Submodules:

3.2.1 tests.test_JSONpickleable module

3.2.2 tests.test_config module

3.2.3 tests.test_configurable module

3.2.4 tests.test_driverMetaclassing module

3.2.5 tests.test_electrical_sources module

3.2.6 tests.test_imports module

3.2.7 tests.test_instrument_overloading module

3.2.8 tests.test_labstate module

3.2.9 tests.test_multiChannelLaserSource module

3.2.10 tests.test_prologix module

3.2.11 tests.test_virtualization module

3.2.12 tests.test_visa_drivers module

Note: This documentation contains ipython notebooks. It is possible to open them with a jupyter kernel and run them interactively to play with knobs and see more plotting features.

- genindex
- modindex
- search

Bibliography

[TFerreiradLimaN+16a] A.N. Tait, T. Ferreira de Lima, M.A. Nahmias, B.J. Shastri, and P.R. Prucnal. Continuous calibration of microring weights for analog optical networks. *Photonics Technol. Lett.*, 28(8):887–890, April 2016. doi:10.1109/LPT.2016.2516440.

[TFerreiradLimaN+16b] Alexander N. Tait, Thomas Ferreira de Lima, Mitchell A. Nahmias, Bhavin J. Shastri, and Paul R. Prucnal. Multi-channel control for microring weight banks. *Opt. Express*, 24(8):8895–8906, Apr 2016. URL: http://www.opticsexpress.org/abstract.cfm?URI=oe-24-8-8895, doi:10.1364/OE.24.008895.

Python Module Index

Index