Lock-In Amplifier Detection Lab 1

April 2, 2024

Remember to restart the kernal in your previous Jupyter Lab document to break the connection with any other devices before beginning this one.

[ ]:
```python
%matplotlib inline
from pathlib import Path
from time import monotonic, sleep

import numpy as np
import matplotlib.pyplot as plt
import math

import qcodes as qc
from qcodes.dataset import (
    Measurement,
    initialise_or_create_database_at,
    load_by_guid,
    load_by_run_spec,
    load_or_create_experiment,
    plot_dataset,
)
from qcodes.dataset.descriptions.detect_shapes import␣
 ↪detect_shape_of_measurement
from qcodes.logger import start_all_logging
start_all_logging()

from scipy.optimize import curve_fit
import numpy as np

from ultolib import (anritsu, korad, spincore)
from ultolib.spincore import pulse
import qcodes.instrument_drivers.stanford_research as stanford_research
```

[ ]:
```python
#Initialize Instruments:
# Note : this will generate two deprecation warnings when creating the␣
 ↪pulse_blaster
pulse_blaster = spincore.PulseBlasterESRPRO(name='pulse_blaster',␣
 ↪board_number=0)
```

```
pulse_blaster.core_clock(500)                          #Sets the clock speed to 500␣
 ↪MHz
                                                       #must be called immediately␣
 ↪after connecting to the PulseBlaster
lock_in_amp = stanford_research.SR830(name='lock_in_amp', address='ASRL5::
 ↪INSTR', terminator='\r')
microwave_src=anritsu.MG3681A(name='microwave_src', address='ASRL4::INSTR',␣
 ↪terminator='\r\n')
microwave_src.output_level_unit('dBm')
microwave_src.IQ_modulation('EXT')
microwave_src.output('OFF')
pulse_blaster.stop()
```

0.1 Lock-In Amplifier Based Detection

On the lock-in amplifier front panel - 'Channel 1' - displays the DC value corresponding to the amplitude of the input signal at the reference frequency of the lock-in. In the below segment, write a pulse sequence that will produce a 200Hz signal that is on for the first half of the time and off for the 2nd half. NOTE: You can reference the "instrument introduction" document for information on how to program pulse sequences in the pulse blaster.

[ ]:
```
ref_f = 200                                #The lock-in amplifier reference␣
 ↪frequence.
ref_D =                                #The lock-in amplifier reference duty cycle.
T_ref_on =
T_ref_off =

pulse_blaster.reset_channel_buffer()   #Clear the previous pulse sequence.
pulse_blaster.ch0.pulse_sequence_buffer.set(
    #Your pulse program here
)                                      #Define the new pulse sequence
pulse_blaster.plot_channel_buffer()    #This function plots the newly defined␣
 ↪pulse sequence.
pulse_blaster.flush_channel_buffer()   #Initiates the pulse sequence
```

Notice that since bit 0 of the PulseBlaster is connected to the REF IN port of the Lock-In Amplifier (LIA), the reference frequency of the LIA should have changed to 200 Hz. If this did not happen, please make sure the LIA is set to use an external reference. We now wish to measure the NV photoluminescence (PL) signal over time. To do so, we must run a measurement. This is initialized by first defining the parameters, as shown below:

[ ]:
```
T = qc.ManualParameter('time', unit='s')
LI_R = qc.ManualParameter('signal', unit='V')
```

We now either initialize or create the experiment path for saving data gathered through this experiment with the above defined parameters.

```
[ ]: initialise_or_create_database_at(Path.cwd() / "Photoluminescence Measurement.
     ↪db")
     experiment = load_or_create_experiment(
         experiment_name='Photoluminescence Measurement',
         sample_name=""
     )

     meas = Measurement(exp=experiment, name='Photoluminescence Measurement')
     meas.register_parameter(T)
     meas.register_parameter(LI_R)
```

Now that the experiment is initialized, we wish to measure something meaningful. Write a pulse program that turns the green laser on and off at the same frequency as the lock-in amplifier reference. When run, the red photoluminescence collected from the NV sample will be modulated at the same frequency as the reference for the lock-in amplifier. The photodiode will collect the modulated photoluminescence from the NV sample, and feed its voltage output to the lock-in amplifier, giving us a reading of the signal with removal of the noise acheived by comparing the pulsed on voltage with the non-pulsed voltage in the lock-in amplifier.

```
[ ]: #Set your pulse sequence here:
     ref_f = 200                          #The lock-in amplifier reference␣
     ↪frequence.
     ref_D =                              #The lock-in amplifier reference duty␣
     ↪cycle.
     T_ref_on =
     T_ref_off =

     T_laser_on =

     pulse_blaster.reset_channel_buffer()  #Clear the previous pulse sequence.
     pulse_blaster.ch0.pulse_sequence_buffer.set(
         #Your pulse program here
     )                                     #Define the new pulse sequence for␣
     ↪channel 0.
     pulse_blaster.ch1.pulse_sequence_buffer.set(
         #Your pulse program here
     )                                     #Define the new pulse sequence for␣
     ↪channel 1.
     pulse_blaster.plot_channel_buffer()   #This function plots the newly defined␣
     ↪pulse sequence.
     pulse_blaster.flush_channel_buffer()
```

Here, we can set the desired components prior to running the experiment.

```
[ ]: # Make sure to only start the pulsing at the start of the experiment. If you␣
     ↪don't, you'll not observe how well calibrated the time constant is.
     pulse_blaster.stop()
```

```
#Set the lock-in amplifier time constant and sensivity using the code segment␣
↪below.
lock_in_amp.time_constant(#Your time constant here)
lock_in_amp.sensitivity(#Your sensitivity here)

t_const_wait = 3*lock_in_amp.time_constant()
t_start = 0 #start time (in seconds)
stepsize = 0.125 #step size (in seconds)
t_end = 10 #end time (in seconds)
```

Now we need to tell python to:

1. Loop through a set of times.
2. Wait for the lock-in amplfier to settle.
3. Measure the the lock-in amplifier voltage.

[ ]:
```
pulse_blaster.flush_channel_buffer() #Begins the pulse sequence
sleep(t_const_wait) #Lets the lock-in amplifier settle
with meas.run() as datasaver:
    datasaver.add_result((T, 0),
                         (LI_R, lock_in_amp.R())) #Adds the 1st data point␣
↪taken imediately after waiting for the lock-in amplifier to settle.
    for i in list(range(0, int((t_end - t_start)/stepsize))): #Determines the␣
↪number of steps based on the start and end time of data taking as well as␣
↪the step size.
        sleep(stepsize) #Freezes the pulse program for the wait time betweeen␣
↪points
        datasaver.add_result((T, i*stepsize + t_start),
                             (LI_R, lock_in_amp.R()))#Takes the desired data␣
↪and saves it for later use.
    LIA_data = datasaver.dataset  #convenient to have for data access and␣
↪plotting

LIA = LIA_data.to_pandas_dataframe()
plt.plot(LIA["time"], LIA["signal"])
plt.xlabel('Time(s)')
plt.ylabel('Signal(V)')
plt.title(f'LIA signal')
plt.show()
```

By reducing the pulse length from ~1ms down to ~1ns, you can show that :

1 - the luminescence signal is pulsed with sub us rise/fall times (despite what the scope shows!)

2 - the lockin can detect this robustly down to very short pulse lengths

Note: You will want to increase the reference frequency for this measurement to 20kHz.

```
[ ]: #Set your pulse sequence here:
ref_f = 20e3                            #The lock-in amplifier reference␣
 ↪frequence.
ref_D =                                 #The lock-in amplifier reference duty␣
 ↪cycle.
T_ref_on =
T_ref_off =

T_laser_on =

pulse_blaster.reset_channel_buffer()  #Clear the previous pulse sequence.
pulse_blaster.ch0.pulse_sequence_buffer.set(
    #Your pulse sequence here
)                                      #Define the new pulse sequence for␣
 ↪channel 0.
pulse_blaster.ch1.pulse_sequence_buffer.set(
    #Your pulse sequence here
)                                      #Define the new pulse sequence for␣
 ↪channel 1.
pulse_blaster.plot_channel_buffer()   #This function plots the newly defined␣
 ↪pulse sequence.
pulse_blaster.flush_channel_buffer()
```

Here, we can set the desired components prior to running the experiment.

Keep in mind that you will need to figure out what time constant and sensativity will work for this new pulse program, as the signal will be weaker.

```
[ ]: # Make sure to only start the pulsing at the start of the experiment. If you␣
 ↪don't, you'll not observe how well calibrated the time constant is.
pulse_blaster.stop()
#Set the lock-in amplifier time constant and sensivity using the code segment␣
 ↪below.
lock_in_amp.time_constant(#Your time constant here)
lock_in_amp.sensitivity(#Your sensitivity here)

t_const_wait = 3*lock_in_amp.time_constant()
t_start = 0 #start time (in seconds)
stepsize = 0.125 #step size (in seconds)
t_end = 10 #end time (in seconds)
```

```
[ ]: pulse_blaster.flush_channel_buffer() #Begins the pulse sequence
sleep(t_const_wait) #Lets the lock-in amplifier settle
with meas.run() as datasaver:
    datasaver.add_result((T, 0),
                         (LI_R, lock_in_amp.R())) #Adds the 1st data point␣
 ↪taken imediately after waiting for the lock-in amplifier to settle.
```

```
    for i in list(range(0, int((t_end - t_start)/stepsize))): #Determines the␣
 ↪number of steps based on the start and end time of data taking as well as␣
 ↪the step size.
        sleep(stepsize) #Freezes the pulse program for the wait time betweeen␣
 ↪points
        datasaver.add_result((T, i*stepsize + t_start),
                             (LI_R, lock_in_amp.R()))#Takes the desired data␣
 ↪and saves it for later use.
    LIA_data = datasaver.dataset  #convenient to have for data access and␣
 ↪plotting

LIA = LIA_data.to_pandas_dataframe()
plt.plot(LIA["time"], LIA["signal"])
plt.xlabel('Time(s)')
plt.ylabel('Signal(V)')
plt.title(f'LIA signal')
plt.show()
```

0.2 Use as a tool

Notice that this is a great tool for figuring out how well chosen your sensativity and time constant values are for any given pulse sequence. For future experiments, you may find it helpful to come back to this experiment when calibrating these parameters for the given pulse sequence. Below is a framework for running this experiment with an arbitrary pulse sequence.

[ ]:
```
initialise_or_create_database_at(Path.cwd() / "Time_Sensitivity Calibration.db")
experiment = load_or_create_experiment(
    experiment_name='Time/Sensitivity Calibration',
    sample_name=""
)

meas = Measurement(exp=experiment, name='Time/Sensitivity Calibration')
meas.register_parameter(T)  # register the first independent parameter
meas.register_parameter(LI_R, setpoints=(T,))  # now register the dependent one
#meas.register_parameter(LI_R) if you want to fix the panda conversion issue.␣
 ↪better if you are hand analysing rather then relying on the provided␣
 ↪graphing methods. (Not sure how this works with monitoring).
```

[ ]:
```
#Place your pulse program here.
```

[ ]:
```
# Make sure to only start the pulsing at the start of the experiment. If you␣
 ↪don't, you'll not observe how well calibrated the time constant is.
pulse_blaster.stop()
#Set the lock-in amplifier time constant and sensivity using the code segment␣
 ↪below.
lock_in_amp.time_constant(#Your time constant here)
lock_in_amp.sensitivity(#Your sensitivity here)
```

```
#meas.write_period = 3 * lock_in_amp.time_constant() #Write period is the amount of time between start to finish of the experiment.
N = 3 #Change to change the number of time constants you want to wait before starting the measurement. This will help calibrate this component for getting a good signal in your experiment.
t_const_wait = N*lock_in_amp.time_constant()
t_start = 0 #start time (in seconds)
stepsize = 0.125 #step size (in seconds)
t_end = 10 #end time. The system will always round up to the nearest step size as to avoid missing a point if this is not divisible evenly. (in seconds)
```

[ ]:

```
pulse_blaster.flush_channel_buffer() #Begins the pulse sequence
sleep(t_const_wait) #Lets the lock-in amplifier settle
with meas.run() as datasaver:
    datasaver.add_result((T, 0),
                         (LI_R, lock_in_amp.R())) #Adds the 1st data point taken imediately after waiting for the lock-in amplifier to settle.
    for i in list(range(0, int((t_end - t_start)/stepsize))): #Determines the number of steps based on the start and end time of data taking as well as the step size.
        sleep(stepsize) #Freezes the pulse program for the wait time betweeen points
        datasaver.add_result((T, i*stepsize + t_start),
                             (LI_R, lock_in_amp.R()))#Takes the desired data and saves it for later use.
    LIA_data = datasaver.dataset  #convenient to have for data access and plotting

LIA = LIA_data.to_pandas_dataframe()
plt.plot(LIA["time"], LIA["signal"])
plt.xlabel('Time(s)')
plt.ylabel('Signal(V)')
plt.title(f'LIA signal')
plt.show()
```