LOCK-IN DETECTION USING EXPEYES-17:
Square Wave Reference and Phase Detection

A project submitted for the evaluation of

INTEGRATED PHYSICS LABORATORY in SEM-VIII

by

ADHILSHA A (2011006)

to the

School of Physical Sciences
National Institute of Science Education and Research
Bhubaneswar

April 20, 2024

ACKNOWLEDGEMENTS

I express my sincere gratitude to the individuals whose support and guidance were integral to the completion of this report. I extend my deepest thanks to Dr. Ashok Mohapatra, our dedicated course instructor, whose insightful teachings greatly enriched my academic experience.

I also acknowledge and appreciate the invaluable guidance provided by Dr. G. Santhosh Babu. His expertise, suggestions, and corrections were pivotal in steering this project towards successful completion. Furthermore, I extend our heartfelt thanks to Mr. Sakthivel V. A and Mr. Rudranarayan Mohanty, our diligent lab assistants, as well as the teaching assistant, Ms. Nandini Mondal, whose tireless efforts in assisting with experimental details were invaluable. Their contributions played a crucial role in the smooth execution of the experiments.

ABSTRACT

Lock-in detection, the popular signal processing technique, is renowned for its ability to extract weak signals from noise. Lock-in amplifiers are widely used in numerous optical equipment and experimental configurations to detect weak signals superimposed on a noisy background. This motivates us to implement a Lock-in amplifier using the advantages of the ExpEYES-17 hardware and the growing computational power of Python programming software.

The implementation includes the measurement of the amplitude and phase of the expected component of the signal specified by the reference signal. We also implement two variations, which use a sin reference signal and a square reference signal. The implementation is then tested using two common experiments: measurement of low resistance and measurement of mutual inductance. A comparison with the theoretical values is also made to validate the implementation. The results show that the implementation is able to measure the amplitude with very high accuracy and phase with reasonable accuracy.

Contents

List of Figures

Chapter 1

Introduction

One of the significant hurdle in experiments involving noisy signals is to get the best possible signal recovery. Because conventional amplifiers are unable to efficiently isolate the appropriate frequency components, they can only be used in in settings with severely deteriorated noise. Due to this shortcoming, the intended signal is overshadowed by increased noise levels, which results in insufficient signal-to-noise ratios. This imbalance highlights the urgent need for novel strategies to close this gap and improve signal recovery in noisy settings.

In various scientific and technological domains, Lock-in Detection is a powerful signal processing method that is used to separate weak signals from noisy backgrounds. It operates by rejecting all other noise frequencies and using phase-sensitive detection to isolate a specific frequency component, given the reference signal [5]. Applications ranging from spectroscopy and communications to biological imaging and materials characterization benefit greatly from its ability to precisely measure and analyse signals that are masked by interference or lost in noise.

With its hardware and software components, ExpEYES (Experiments for Young Engineers and Scientists) offers a flexible platform for conducting physics experiments. This platform enables the implementation of sophisticated signal processing techniques including lock-in detection. Researchers and educators can get practical insights into lock-in detection's operation and applications by using ExpEYES to conduct hands-on exploration of its concepts.

This experiment explores the basics of lock-in detection and its implementation

using ExpEYES, demonstrating its importance in signal recovery and measurement in noisy situations.

We conduct our experiment with the following **objectives**:

1. To implement the Lock-in detection using a sin-wave reference.
2. To implement the Lock-in detection using a square-wave reference.
3. To test the square-wave reference implementation using validation measurement of low resistance and measurement of mutual inductance.

Chapter 2

Theory

2.1 Lock-in detection

Lock-in-detection involves the use of a reference signal with single frequency component to analyse a noisy signal from which the same component needs to identified. For this, we multiply the signal with the reference signals of same phase and of quadrature reference to get an in_phase and quadrature_phase components. The components can be then used to find the amplitude and phase of the buried signal from noisy background. The reference signal can be either a sine wave or a square wave.

2.1.1 Sin Wave reference

Let the input signal be of the form:

$$V_s(t) = V_s \sin(\omega_s t + \phi) \tag{2.1}$$

where, ω_s is the frequency, V_s is the amplitude and ϕ is the phase of the input signal. We take two forms of the reference signal, one with same phase as input signal and the other with a 90° phase difference. They are of the form,

$$\begin{aligned} V_{r0}(t) &= V_r \sin(\omega_r t) \\ V_{r90}(t) &= V_r \cos(\omega_r t) \end{aligned} \tag{2.2}$$

where ω_r is the frequency and V_r is the amplitude of the reference signal.

Now, when we multiply the input signal with each of the reference signals we get,

$$\begin{aligned} V_s(t) * V_{r0}(t) &= V_s * V_r \sin(\omega_s t + \phi) * \sin(\omega_r t) \\ V_s(t) * V_{r90}(t) &= V_s * V_r \sin(\omega_s t + \phi) * \cos(\omega_r t) \end{aligned}$$

Using trigonometric identity for $sin(x)sin(y)$ and $sin(x)cos(y)$ along with the assumptions $\omega_s = \omega_r$ and $V_r = 2V$, we can rewrite the above equations as:

$$\begin{aligned} V_{r0} &= V_s \left[\cos\phi - \cos\left(2\omega_r t + \phi\right)\right] \\ V_{r90} &= V_s \left[\sin\left(2\omega_r t + \phi\right) - \sin\phi\right] \end{aligned} \tag{2.3}$$

With a low pass filter, FFT in our case, the $2\omega_r$ high frequency term is isolated. The zero-frequency peaks (V') obtained from the FFT outputs corresponds to the DC parts of the signal given in the above equation.

$$\begin{aligned} V'_{r0} &= FFT\left(V_{r0}\right)[0]/N = V_s \cos\phi \\ V'_{r90} &= FFT\left(V_{r90}\right)[0]/N = V_s \sin\phi \end{aligned} \tag{2.4}$$

The peaks of FFT obtained through can be changed into amplitudes by dividing with the total number of samples, N. The final amplitude of the signal is obtained as:

$$V_s = \sqrt{\left(V'_{r0}\right)^2 + \left(V'_{r90}\right)^2} \tag{2.5}$$

The phase is obtained as:

$$\phi = \arctan\left(\frac{V'_{r90}}{V'_{r0}}\right) \tag{2.6}$$

2.1.2 Square Wave Reference

Let the signal under consideration be,

$$V\left(t\right) = V_{DC} + V_s \sin\left(\omega_s t + \phi\right) + noise\left(t\right) \tag{2.7}$$

As discussed before, we need two orthogonal reference signals using which we carry out quadrature multiplication as seen in above section. In case of square wave, we need at least $4N$ points per cycle to make this multiplication possible (further in

discussions). This implies a constraint on the sampling frequency $f(s)$ based on the signal frequency f.

$$f_s \;=\; 4Nf \tag{2.8}$$

where $N = 1, 2, 3...$ This means that that $4N$ points are sampled in a single period of the signal[1]. The in-phase and quadrature-phase reference signal of the same frequency as the signal in a single period are:

$$S(n) = \begin{cases} 1 & \text{if } 0 \leq n \leq 2N-1 \\ -1 & \text{if } 2N \leq n \leq 4N-1 \end{cases} \tag{2.9}$$

$$C(n) = \begin{cases} 1 & \text{if } 0 \leq n \leq N-1, 3N \leq n \leq 4N-1 \\ -1 & \text{if } N \leq n \leq 3N-1 \end{cases} \tag{2.10}$$

On multiplying the input signal and the references for 1 cycle, we get,

$$I \;=\; \frac{1}{4N} \sum_{n=0}^{4N-1} V\,(n)\, S\,(n) \tag{2.11}$$

$$Q \;=\; \frac{1}{4N} \sum_{n=0}^{4N-1} V\,(n)\, C\,(n)$$

Since the $S(n)$ and $C(n)$ only contains 1 and -1, the multiplication and accumulation progress will become additions and subtractions [3]. The multiplications generate a huge number of harmonics, and the averaging procedures can filter out the higher frequency components. Then the results of in-phase and quadrature-phase output are given as (with $V(n)$ being the n^{th} sample):

$$I\,(n) \;=\; \frac{2V_s}{\pi} \cos\left(\phi \;-\; \frac{\pi}{4N}\right) \tag{2.12}$$

$$Q\,(n) \;=\; \frac{2V_s}{\pi} \sin\left(\phi \;-\; \frac{\pi}{4N}\right)$$

The amplitude and the initial phase can be obtained by:

$$V_s \;=\; \frac{\pi}{2}\sqrt{I^2 \;+\; Q^2} \tag{2.13}$$

$$\phi \;=\; \arctan\left(\frac{Q}{I}\right) + \frac{\pi}{4N} \tag{2.14}$$

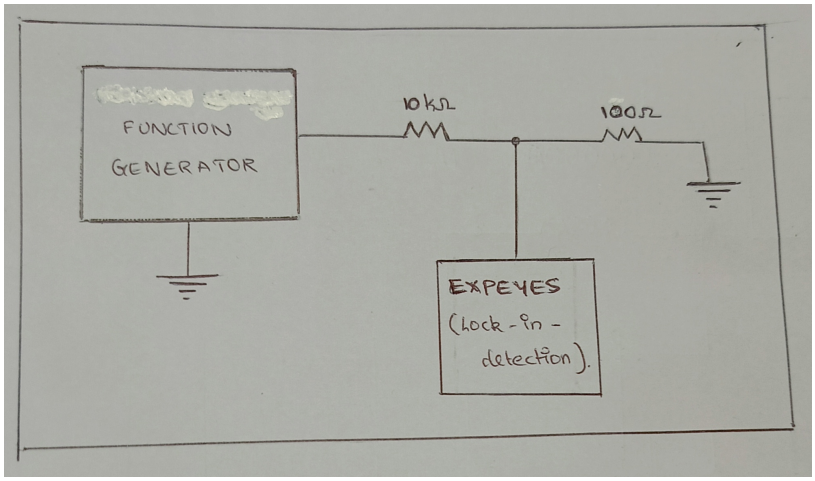

Figure 2.1: Simple circuit diagram for low resistance measurement

2.2 Measurement of Low Resistance

Because of the intrinsic noise and offsets that are usually present in low resistance measurements, the signal obtained need to analysed with methods robust to noise. For the same, we use lock-in detection to analyse the signal from low resistance with respect to a signal from a larger resistance or the input signal. See a simple circuit for such a measurement in Figure 2.1.

As given in the Figure 2.1, We can find the current through the circuit, which is:

$$I = \frac{V_{in}}{r + R} \tag{2.15}$$

where V_{in} is the total applied voltage, r is the low resistance and R is the high resistance.

Since the same current passes through both resistors, we can write:

$$\frac{V_R}{R} = \frac{V_r}{r} = \frac{V_{in}}{r + R} \approx \frac{V_{in}}{R} \tag{2.16}$$

where V_R denotes the voltage across R and V_r denotes the voltage across r. The last approximation above is valid when $R >> r$, which is the choice in our experiment.

Rearranging the terms, we get:

$$r = \frac{RV_r}{V_{in}} \tag{2.17}$$

Since, the voltage signal from low resistance could be extremely small, an amplification can be done before lock-in detection. Thus, the output from Lock-in detection, V_{out} will be:

$$V_{out} = \alpha V_r \tag{2.18}$$

where α is the amplification factor.

Combining these equations, we get:

$$r = \frac{RV_{out}}{\alpha V_{in}} \tag{2.19}$$

2.3 Measurement of Mutual inductance

When two coils are arranged side by side and an AC current is sent through the *primary coil*, an AC voltage of the same frequency is induced in the *secondary coil* due to the phenomenon of **Mutual Induction**. To formulate this, let the primary current vary as:

$$I \;=\; I_0 \sin(2\pi f t) \tag{2.20}$$

where f is the frequency in Hertz, then emf induced in the secondary coil is:

$$\begin{aligned} V \;&=\; -M\frac{dI}{dt} \\ V \;&=\; -2\pi M f I_0 \sin\left(2\pi f t \;+\; \frac{\pi}{2}\right) \\ V \;&=\; -2\pi M f \frac{V_0}{R} \sin\left(2\pi f t \;+\; \frac{\pi}{2}\right) \end{aligned} \tag{2.21}$$

where M is the mutual inductance and R is the total resistance. From the above equations, we can conclude the following:

1. The phase difference between the primary current and the induced emf is $\pi/2$.
2. The emf is proportional to the amplitude V_0 of the input since the resistance is a constant ($I_0 = V_0/R$).
3. The emf is proportional to the frequency f.

We use the lock-in detector to measure the voltage across the secondary coil (V_{sec}). If α is the gain of the lock-in amplifier and V_{out} is the voltage obtained after lock-in detection, then the voltage across the secondary coil will be:

$$V_{sec} = \frac{V_{out}}{\alpha} \tag{2.22}$$

$\therefore$ the mutual inductance can be calculated as

$$M = \frac{V_{out} R}{2\pi f \alpha V_0} \tag{2.23}$$

To show the collection of the required inputs, a sample circuit is shown in Figure 2.2.

In Figure 2.2, we use an Op-Amp (non-inverting mode). If R_{in} is the input resistor and R_f is the feedback resistor (potentiometer in use). Then, the value of α becomes:

$$\alpha = 1 + \frac{R_f}{R_{in}} \tag{2.24}$$

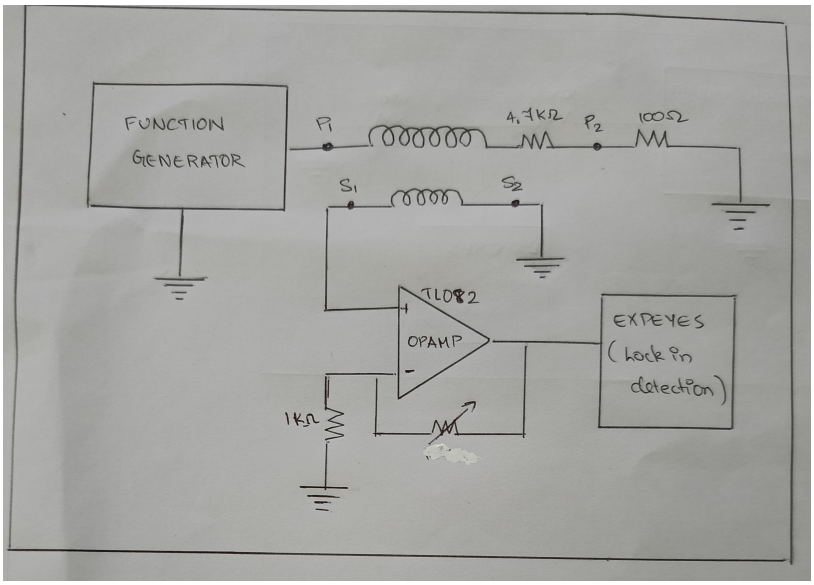

Figure 2.2: Simple circuit diagram for Mutual inductance measurement

Chapter 3

Experimental Setup and Methods

3.1 Data collection using ExpEYES-17

ExpEYES-17 is interfaced and powered by the computer's USB port and may be programmed in Python. It can be used as a low-frequency oscilloscope, function generator, programmable voltage source, frequency counter, and data logger. Connectors on the top panel allow you to attach external signals, as seen in Figure 3.1. The program can monitor and regulate the voltages at these terminals. To use the EYES17 hardware, the Python modules for `eyes17` must be installed [2].

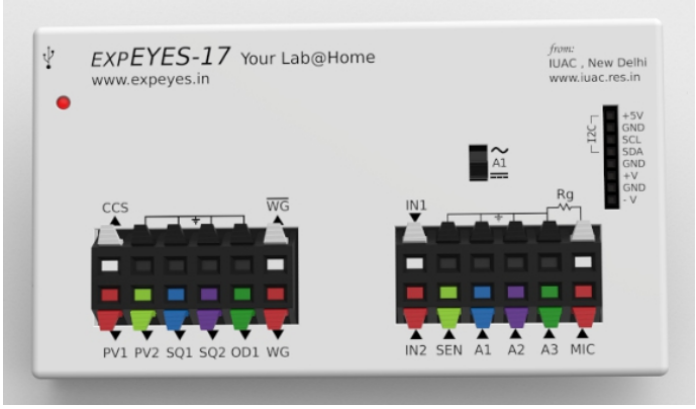

Figure 3.1: ExpEYES-17

To collect the signal and reference using ExpEYES-17, we use the `capture2()` from the Python library `eyes17`, which connects and operates the related hardware.

As per the User manual [2], The number of samples can be up to 10000, and the time gap between two consecutive samples is in the range $[2\mu s, 1000\mu s]$.

After setting a reasonable sampling frequency using the frequency knowledge and

doing a warmup-test run (further details in discussion), we can start taking the data continuously for the current ExpEYES17 settings. A sample code is given below.

```
import eyes17
p=eyes17.eyes.open()

N_sample = 5000
f = 500 # Hz - prior frequency knowledge
N_div=64 # samples per cycle
t_gap = (1/(f*N_div))*10**6 #micro sec

warmup_values = [100]
for val in range(500,10001,500):
    warmup_values.append(val)

def warmup_expeyes(t_gap, warmup_values):
    for val in (warmup_values):
       t,v, tt,vv = p.capture2(val, t_gap)

warmup_expeyes(N_sample, t_gap) # user-defined function for warmup-test run

t, sig, t_, ref = p.capture2(N_sample, t_gap) # signal connected to 'A1' and
    reference to 'A2'
```

3.2 Sin wave reference & Lock-in detection

The following steps are the steps implemented:

1. First, both input and reference is captured as discussed in the above section.

2. Then, the signal can be triggered using the reference signal at hand and also from the right side of the signal (more in discussion).

3. Using the frequency information from the reference signal, we generate *sin* (in-phase) and *cos* (quadrature) signals of amplitude $2V$.

4. Following this, the signal is then multiplied with the reference signals and then passed through `scipy.fft` to obtain the FFT plots.

5. After identifying the zero peaks and converting them to amplitude, we can use equations (2.5) and (2.6) to find the amplitude and phase of the signal in question.

The code for the process is given below.

```
 # function generator settings
f = 500 #Hz

# expeyes17 settings
N_sample = 8192
N_div=64
t_gap = (1/(f*N_div))*10**6 #us

# experimental settings
warmup = True # if warmup is needed
warmup_repititive = False # if warmup need to be reapeated every run
triggering = True
right_clip = True

auto_freq = True # if True, the frequency is calculated from the reference
freq_method = 'fft' # 'fft' or 'zero_crossing'

# internal settings dont touch!
warmup_flag = 0

print("settings ready!")

if warmup and warmup_flag == 0:
    if warmup_repititive: print("Performing repititive warmup...")
    else: print("Performing a warmup run.. coz the Expeyes misbehaving otherwise!")

    warmup_expeyes(t_gap)
    warmup_flag = 1
elif not warmup:
    print("NO WARMuP.. POSSIBLE FAILURE OF CAPTURE2...")
t,v, tt,vv = p.capture2(N_sample, t_gap)

v= np.array(v)
vv = np.array(vv)

# trigger the reference signal vv by rising edge
def trigger(vv):
    for i in range(len(vv)-1):
        if vv[i] < 0 and vv[i+1] > 0:
            return i + 1
    return i

if triggering:
    trigger_index = trigger(vv)
    # print(f"Trigger index = {trigger_index}")

    triggered_t = t[trigger_index:]
    triggered_v = v[trigger_index:]
    triggered_vv = vv[trigger_index:]
else:
    triggered_t = t
    triggered_v = v
    triggered_vv = vv

def right_clip_trigger(triggered_vv):

    right_clip_index = len(triggered_vv) - 1
    for i in range(right_clip_index, 0, -1):
        if triggered_vv[i] > 0 and triggered_vv[i-1] < 0:
            return i - 1
    return i

```

```
63  if right_clip:
64      right_clip_index = right_clip_trigger(triggered_vv)
65      # print(f"Right clip index = {right_clip_index}")
66
67      triggered_t = triggered_t[:right_clip_index]
68      triggered_v = triggered_v[:right_clip_index]
69      triggered_vv = triggered_vv[:right_clip_index]
70
71  V_sin_ref=[]
72  V_cos_ref=[]
73  t_ref = []
74
75  if auto_freq:
76      if freq_method == 'fft':
77          freq=fftfreq(len(vv),(t[1]-t[0])*0.001)
78          sine_fft=fft(vv)
79          xf = fftshift(freq)
80          yplot = (1/len(v)) * fftshift(sine_fft)
81
82          # only positive frequencies
83          xf2 = xf[xf>0]
84          yplot2 = np.abs(yplot[xf>0])
85
86          # finding the max peak value
87          peak_value = np.max(yplot2)
88
89          # finding the first peak that is greater than 0.5 times the peak value
90
91          peak_index = np.where(yplot2 > 0.8*peak_value)[0][0]
92
93          freq = xf2[peak_index]
94
95
96      elif freq_method == 'zero_crossing':
97          zero_crossings = np.where(np.diff(np.sign(vv)))[0]
98          freq = 1/(2*(t[zero_crossings[1]]-t[zero_crossings[0]]))
99
100     print(f"Auto detected frequency = {freq} Hz")
101 else:
102     freq = f
103
104
105 curr_N_sample = len(triggered_v)
106 for i in range(curr_N_sample):
107     V_sin_ref.append(2*m.sin(2*m.pi*freq*i*t_gap*10**-6))
108     V_cos_ref.append(2*m.cos(2*m.pi*freq*i*t_gap*10**-6))
109     t_ref.append(i*t_gap)
110
111 print(len(V_sin_ref))
112
113 v_pm_sin = []
114 v_pm_cos = []
115 for i in range(curr_N_sample):
116     v_pm_sin.append(v[i]*V_sin_ref[i])
117     v_pm_cos.append(v[i]*V_cos_ref[i])
118
119 freq=fftfreq(curr_N_sample,(t[1]-t[0])*0.001)
120 sine_fft=fft(v_pm_sin)
121 cos_fft=fft(v_pm_cos)
122
123 xf = fftshift(freq)
124 yplot = (1/curr_N_sample) * fftshift(sine_fft)
125
126 yplot = (1/curr_N_sample) * fftshift(cos_fft)
```

```
127
128 V = np.sqrt((sine_fft[0].real/len(sine_fft))**2+(cos_fft[0].real/len(cos_fft))**2)
129 phase = np.arctan2((cos_fft[0].real/len(cos_fft)),(sine_fft[0].real/len(sine_fft)))
130 print("Voltage is",V," V")
131 print("Phase of the output Voltage",np.rad2deg(phase),' deg')
```

This implementation is a stepping zone to square wave Lock-in and has been tested before. Our experiments will be mainly focused on testing the capabilities of square wave Lock-in with square wave back-end. The reasons will be explained in the next sections.

3.3 Square wave reference & Lock-in detection

Here, the goal is to implement Lock-in detection, given a signal and a square wave reference. Since, the sin-wave implementation works by extracting information from the reference signal and create the *sin* and *cos* references required, we can use that as the back-end to achieve the expected results given in Section 2.1.1.

Another implementation would be to explicitly use the square wave reference characteristics to perform quadrature multiplications and use it to calculate the results as discussed before in 2.1.2. Our experiments will focus mainly here, since the others have been tested before by other students.

3.3.1 Sin wave back-end

The following steps are the steps implemented:

1. First, both input and reference is captured as discussed in the above section with `N_div` (the number of samples per cycle) a multiple of 4.

2. Then, the steps from the sin wave implementation are repeated since the back-end calculations are *sin-cos* reference signals.

This will yield the same results from that of sin wave reference implementation and therefore not pursued further.

3.3.2 Square wave back-end

Here, the in-phase and quadrature signals used in reference will be square waves. The following steps are the steps implemented:

1. First, both input and reference is captured as discussed above.

2. Then, the signal can be triggered using the reference signal at hand and also from the right side of the signal (more in discussion).

3. Using the frequency information from the reference signal (obtained through frequency or detecting cycle change in reference signal), we generate in-phase and quadrature square wave signals of amplitude $1V$ (range is -1 to 1).

4. We also implemented an alternative, where we take the acquired reference signal and get the quadrature form by removing the first `N_div/4` samples (a quarter of a cycle defined in data collection). Then, the signals and sliced from the right to match the quadrature signal length.

5. Following this, the signal is then multiplied with the reference signals and summed over as discussed in equation (2.11).

6. Then, we can apply equations (2.13) and (2.14) to calculate the amplitude and phase of the expected component.

The code used for the same is given below:

```
# function generator settings
f = 500 #Hz

# expeyes17 settings
N_sample = 10000
```

```
N_div=64
t_gap = (1/(f*N_div))*10**6 #us
voltage_range_A1 = 0.25
voltage_range_A2 = 1.5
p.select_range('A1',voltage_range_A1)
p.select_range('A2',voltage_range_A2)

# experimental settings
warmup = True # if True, the warmup is done ince at beginning
warmup_repititive = False #if True, the warmup is done for each run
triggering = False
right_clip = False
t_gap_reset = True # if True, the t_gap will be reset based on the detected frequency

auto_freq = True #if True, the frequency is automatically calculated
                    # use fft or check the indices of zero crossing pf reference to
                        find frequency
sq_wave_quadrature_method = 'phase_lag' # can be either 'sin_cos' or 'phase_lag'
                                        # generating square wave or using phase_lag

# internal settings dont touch!
warmup_flag = 0

print("settings ready!")

if t_gap_reset:
    if warmup and warmup_flag == 0:
        if warmup_repititive: print("Performing repititive warmup...")
        else: print("Performing a warmup run.. coz the Expeyes misbehaving
            otherwise!")

        warmup_expeyes(t_gap)
    elif not warmup:
        print("NO WARMuP.. POSSIBLE FAILURE OF CAPTURE2...")
    t,v, tt,vv = p.capture2(N_sample, t_gap)

    if freq_method == 'fft':
        freq=fftfreq(len(vv),(t[1]-t[0])*0.001)
        sine_fft=fft(vv)
        xf = fftshift(freq)
        yplot = (1/len(v)) * fftshift(sine_fft)

        # only positive frequencies
        xf2 = xf[xf>0]
        yplot2 = np.abs(yplot[xf>0])

        # finding the max peak value
        peak_value = np.max(yplot2)

        # finding the first peak that is greater than 0.5 times the peak value

        peak_index = np.where(yplot2 > 0.8*peak_value)[0][0]

        freq = xf2[peak_index]


    elif freq_method == 'zero_crossing':
        zero_crossings = np.where(np.diff(np.sign(vv)))[0]
        freq = 1/(2*(t[zero_crossings[1]]-t[zero_crossings[0]]))

    print(f"Auto detected frequency = {freq} Hz")

    f = freq
    t_gap = freq_detection_reset_tgap(f,N_div)
```

```
68
69      print(f"t_gap = {t_gap} us")
70      print(f"N_sample = {N_sample}")
71
72
73  def Lockin_detector_single_run(warmup_flag, f, t_gap):
74      if warmup and warmup_flag == 0:
75          if warmup_repititive: print("Performing repititive warmup...")
76          else: print("Performing a warmup run.. coz the Expeyes misbehaving
                otherwise!")
77
78          warmup_expeyes(t_gap)
79          warmup_flag = 1
80      elif not warmup:
81          print("NO WARMuP.. POSSIBLE FAILURE OF CAPTURE2...")
82      t,v, tt,vv = p.capture2(N_sample, t_gap)
83
84      v= np.array(v)
85      vv = np.array(vv)
86
87      # trigger the reference signal vv by rising edge
88
89      def trigger(vv):
90          for i in range(len(vv)-1):
91              if vv[i] < 0 and vv[i+1] > 0:
92                  return i + 1
93          return i
94
95      def right_clip_trigger(triggered_vv):
96
97          right_clip_index = len(triggered_vv) - 1
98          for i in range(right_clip_index, 0, -1):
99              if triggered_vv[i] > 0 and triggered_vv[i-1] < 0:
100                 return i - 1
101         return i
102
103
104     if triggering:
105         trigger_index = trigger(vv)
106         # print(f"Trigger index = {trigger_index}")
107
108         triggered_t = t[trigger_index:]
109         triggered_v = v[trigger_index:]
110         triggered_vv = vv[trigger_index:]
111     else:
112         triggered_t = t
113         triggered_v = v
114         triggered_vv = vv
115
116     if auto_freq:
117         if freq_method == 'fft':
118             freq=fftfreq(len(vv),(t[1]-t[0])*0.001)
119             sine_fft=fft(vv)
120             xf = fftshift(freq)
121             yplot = (1/len(v)) * fftshift(sine_fft)
122
123             # only positive frequencies
124             xf2 = xf[xf>0]
125             yplot2 = np.abs(yplot[xf>0])
126
127             # finding the max peak value
128             peak_value = np.max(yplot2)
129
130             # finding the first peak that is greater than 0.5 times the peak value
```

```
            peak_index = np.where(yplot2 > 0.8*peak_value)[0][0]

            freq = xf2[peak_index]

        elif freq_method == 'zero_crossing':
            zero_crossings = np.where(np.diff(np.sign(vv)))[0]
            freq = 1/(2*(t[zero_crossings[1]]-t[zero_crossings[0]]))

        print(f"Auto detected frequency = {freq} Hz")
    else:
        freq = f

    V_sin_ref=[]
    V_cos_ref=[]
    t_ref = []

    curr_N_sample = len(triggered_v)
    for i in range(curr_N_sample):
        V_sin_ref.append(amp*m.sin(2*m.pi*freq*i*t_gap*10**-6))
        V_cos_ref.append(amp*m.cos(2*m.pi*freq*i*t_gap*10**-6))
        t_ref.append(i*t_gap)

    if sq_wave_quadrature_method == 'sin_cos':
        v_sq_inphase = np.sign(V_sin_ref)
        v_sq_quadrature = np.sign(V_cos_ref)
    elif sq_wave_quadrature_method == 'phase_lag':
        triggered_vv = np.sign(triggered_vv)

        v_sq_inphase = triggered_vv

        # finding the next zero crossing of the reference signal
        zero_crossings = np.where(np.diff(np.sign(vv)))[0]
        index_difference = (zero_crossings[1]-zero_crossings[0]) // 2

        v_sq_quadrature = np.sign(triggered_vv[index_difference:])

        # making the length of all signals equal
        v_sq_inphase = v_sq_inphase[:len(v_sq_quadrature)]
        triggered_t = triggered_t[:len(v_sq_quadrature)]
        triggered_v = triggered_v[:len(v_sq_quadrature)]
        triggered_vv = triggered_vv[:len(v_sq_quadrature)]
        t_ref = t_ref[:len(v_sq_quadrature)]
        V_sin_ref = V_sin_ref[:len(v_sq_quadrature)]
        V_cos_ref = V_cos_ref[:len(v_sq_quadrature)]

    if right_clip:
        right_clip_index = right_clip_trigger(triggered_vv)
        # print(f"Right clip index = {right_clip_index}")

        triggered_t = triggered_t[:right_clip_index]
        triggered_v = triggered_v[:right_clip_index]
        triggered_vv = triggered_vv[:right_clip_index]
        v_sq_inphase = v_sq_inphase[:right_clip_index]
        v_sq_quadrature = v_sq_quadrature[:right_clip_index]
        V_sin_ref = V_sin_ref[:right_clip_index]
        V_cos_ref = V_cos_ref[:right_clip_index]
        t_ref = t_ref[:right_clip_index]

    curr_N_sample = len(triggered_v)

    y_inphase = triggered_v * v_sq_inphase
    y_quadrature = triggered_v * v_sq_quadrature

    # performing the operations
```

```
    sum_y_inphase = np.sum(y_inphase) / len(y_inphase)
    sum_y_quadrature = np.sum(y_quadrature) / len(y_quadrature)

    N_paper = N_div // 4 # N from the equations

    # Observed amplitude and phase
    A = np.sqrt(sum_y_inphase ** 2 + sum_y_quadrature ** 2) *( np.pi / 2)

    phase = np.arctan2(sum_y_quadrature, sum_y_inphase) + (np.pi / (4 * N_paper))

    print('Observed Amplitude:', A)
    print(f'Observed Phase: {np.rad2deg(phase)} deg')

    return A, np.rad2deg(phase), warmup_flag
```

3.4 Measurement of low resistance

For low resistance measure, we implement the followings steps:

1. Connect two resistors of resistance R and r in series such that $R >> r$, where r is the resistance of interest.

2. The input voltage is applied across $R + r$ and the signal is taken across just r.

3. This signal in our case was in a range compatible with the lowest resolution of the ExpEYES-17 and hence we did not implement any amplification. Hence, $\alpha = 1$.

4. This signal and the a square wave reference in phase with the input is passed into ExpEYES-17 and Lock-in detection is implemented.

   The circuit diagram is given in Figure 3.2.

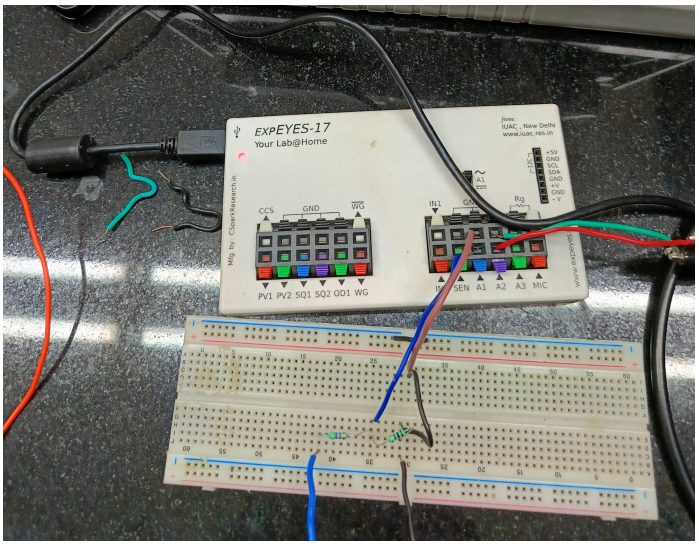

Figure 3.2: Low resistance measurement setup

AC signals with V_{peak} varying from 1 V to 3.5 V in steps of 0.5 V. For each signal, the V_{out} was calculated and recorded in an array. This was repeated for different frequencies ranging from 300 Hz to 700 Hz in steps of 200 Hz. For each (frequency, V_{in}), we ran 20 trials and take average. This was done to check if the result was invariant of frequencies, hence lesser data in frequencies.

The code used for the data collection and analysis is shown below:

```
frequencies = [] # Hz

Vin_allfreq = [] # Vp (V)

# Vout observed through Lockin detector (V)
Vout_obs_allfreq = []

# Phase observed through Lockin detector (degrees)
Phase_obs_allfreq = []



trial_name = 'trial5_r'

# ====== below code is repeated for each (Vin,freq) pair====#
curr_freq = 700
Curr_Vin = 3.5

if len(frequencies) == 0 or frequencies[-1] != curr_freq:
    frequencies.append(curr_freq)

    Vin = [] # Vpp (V)

```

```
    # Vout observed through Lockin detector (V)
    Vout_obs = []

    # Phase observed through Lockin detector (degrees)
    Phase_obs = []

    f = curr_freq

    t_gap = (1/(f*N_div))*10**6 #us
    warmup_flag = 0
    print(f"Running for f = {f} Hz")
    print(f"t_gap = {t_gap} us")


Vin.append(Curr_Vin)
for i in range(20):
    A , phase, warmup_flag = Lockin_detector_single_run(warmup_flag, f, t_gap)

    if i == 0:
        Vout_obs.append([A])
        Phase_obs.append([phase])
    else:
        Vout_obs[-1].append(A)
        Phase_obs[-1].append(phase)

if Curr_Vin == 3.5: # final voltage in each series.
    Vin = np.array(Vin)
    Vout_obs = np.array(Vout_obs)
    Phase_obs = np.array(Phase_obs)

    Vin_allfreq.append(Vin)
    Vout_obs_allfreq.append(Vout_obs)
    Phase_obs_allfreq.append(Phase_obs)

#------------------------------------------------------------#

# saving the data after all runs
np.save(f'Vin_allfreq{trial_name}.npy',Vin_allfreq)
np.save(f'Vout_obs_allfreq{trial_name}.npy',Vout_obs_allfreq)
np.save(f'Phase_obs_allfreq{trial_name}.npy',Phase_obs_allfreq)
np.save(f'frequencies{trial_name}.npy',frequencies)

#------------------------------------------------------------#

# load data and process
trial_name_ = 'trial5_r'

Vin_allfreq = np.load(f'Vin_allfreq{trial_name_}.npy')
Vout_obs_allfreq = np.load(f'Vout_obs_allfreq{trial_name_}.npy')
Phase_obs_allfreq = np.load(f'Phase_obs_allfreq{trial_name_}.npy')
frequencies = np.load(f'frequencies{trial_name_}.npy')


Vin_rms = Vin_allfreq / (np.sqrt(2)) # Vp to Vrms
Vout_obs_avg = np.mean(Vout_obs_allfreq/ (np.sqrt(2)), axis=2) #average over trials
Vout_obs_std = np.std(Vout_obs_allfreq/ (np.sqrt(2)), axis=2) #std over trials

plt.figure(figsize=(10,5))


slopes_all_freq = []
error_slopes_all_freq = []
points_to_avoid = []

```

```
for i in range(len(frequencies)):

    slope, intercept = np.polyfit(Vin_rms[i],Vout_obs_avg[i],1)
    error_slopes_all_freq.append(np.std(Vout_obs_avg[i] - (slope * Vin_rms[i] +
        intercept)))

    slope = Vout_obs_avg[i] / Vin_rms[i]
    avg_slope = np.mean(slope)
    std_slope = np.std(slope)
    slopes_all_freq.append(avg_slope)
    error_slopes_all_freq.append(std_slope)

marker_sizes = [15,9,3]
for i in range(len(frequencies)):
    plt.plot(Vin_rms[i],Vout_obs_avg[i],'o-',label=f'f = {frequencies[i]} Hz; slope
        = {slopes_all_freq[i]:.5f} +/-
        {error_slopes_all_freq[i]:.5f}',markersize=marker_sizes[i])



plt.xlabel('Vin (Vrms) [V]')
plt.ylabel('Vout (Vrms) [V]')
plt.title('Vout vs Vin')
plt.legend()
plt.grid()
plt.savefig(f'Vout_vs_Vin{trial_name_}.png')
plt.show()

avg_slope = np.mean(slopes_all_freq)
std_slope = np.std(slopes_all_freq)

plt.plot(frequencies,slopes_all_freq,'o-',label= f'slope = {avg_slope:.6f} +/-
    {std_slope:.6f}')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Slope (V{out-rms}/V{in-rms})')
plt.title('Slope vs Frequency')
plt.ylim(0,slopes_all_freq[0]*1.1)
plt.grid()
plt.legend()
plt.savefig(f'Slope_vs_Frequency{trial_name_}.png')
plt.show()

r_theoretical = 100 # ohm
R = 4.7e3
alpha = 1

index = 1 # doesnt make much difference since the slope is constant
slope = slopes_all_freq[index]
slope_std = error_slopes_all_freq[index]


r_observed = R * avg_slope / alpha
R_obs_error = r_observed * (slope_std / slope)
relative_error_R = (abs(r_observed - r_theoretical) / r_theoretical) * 100 # in
    percentage

print(f"R_observed = {r_observed:.3f}(+/-){R_obs_error:.3f} ohm")
print(f"R_theoretical = {r_theoretical} ohm")
print(f"Relative error = {relative_error_R:.3f}%")
```

3.5 Measurement of Mutual inductance

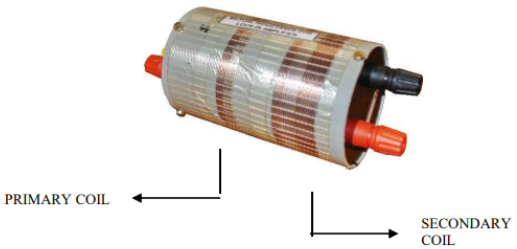

Figure 3.3: Mutual inductance coil

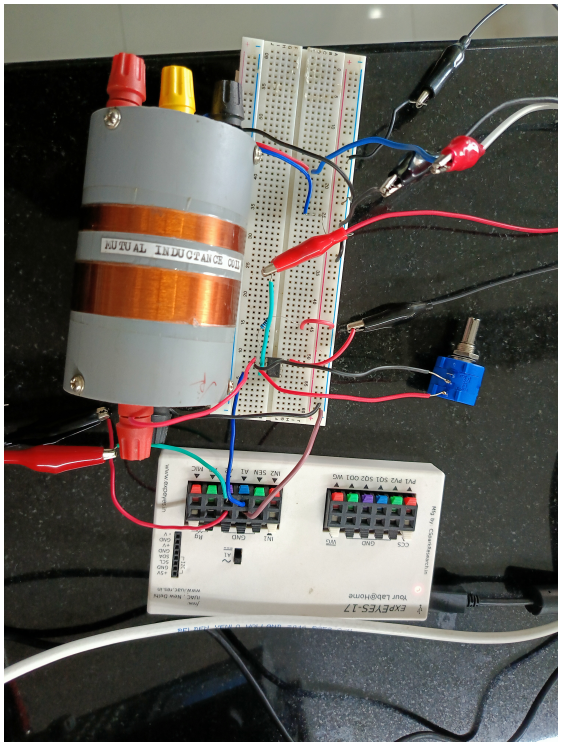

Figure 3.4: Mutual inductance experiment setup

Specifications and setup of mutual inductance coil used[4] given in Figure 3.3 and the experimental setup in Figure 3.4:

- A primary coil of about 20 turns is wound using insulated copper wire, which can carry a current of about a few mA.
- There are three banana terminals: Red, yellow, and black at one end of the coil.
- A 4.7 kΩ resistor and the primary coil are connected between red and yellow banana terminals.
- Between the yellow and black, a 100 Ω resistor is connected.

- The signal generator ground must be connected to the black terminal on the primary side of the coil box, and the other terminal of the signal generator must be connected to the red banana terminal on the primary side of the coil box.
- On the same, a secondary coil of about 100 turns is wound at a distance of a few centimeters from the end of the primary coil.
- The terminals of the secondary coil are brought to two banana terminals on the other end of the insulating former.

AC signals with V_{peak} varying from 2 V to 4 V in steps of 0.5 V. For each signal, the V_{out} was calculated and recorded in an array. This was repeated for different frequencies 500 Hz, 700 Hz, 900 Hz, 1200 Hz and 1500 Hz. For each (frequency, V_{in}), we ran 20 trials and take average. This will validate both proportionality related to input voltage and frequency.

The code for the data and analysis is given below:

```
trial_name_ = 'trial1'

Vin_allfreq = np.load(f'Vin_allfreq{trial_name_}.npy')
Vout_obs_allfreq = np.load(f'Vout_obs_allfreq{trial_name_}.npy')
Phase_obs_allfreq = np.load(f'Phase_obs_allfreq{trial_name_}.npy')
frequencies = np.load(f'frequencies{trial_name_}.npy')

# Vrms from Vp
Vin_rms = Vin_allfreq / (np.sqrt(2))
Vout_obs_avg = np.mean(Vout_obs_allfreq, axis=2) / (np.sqrt(2)) # average of
    Vout_obs across all trials
Vout_obs_std = np.std(Vout_obs_allfreq, axis=2) / (np.sqrt(2)) # std of Vout_obs
    across all trials

plt.figure(figsize=(10,5))

slopes_all_freq = []
error_slopes_all_freq = []

for i in range(len(frequencies)):

    slope = Vout_obs_avg[i] / Vin_rms[i]
    avg_slope = np.mean(slope)
    std_slope = np.std(slope)
    slopes_all_freq.append(avg_slope)
    error_slopes_all_freq.append(std_slope)
    plt.plot(Vin_rms[i],Vout_obs_avg[i],'o-',label=f'f = {frequencies[i]} Hz; slope
        = {slopes_all_freq[i]:.3e} +/- {error_slopes_all_freq[i]:.3e}')

```

```
plt.xlabel('Vin (Vrms) [V]')
plt.ylabel('Vout (Vrms) [V]')
plt.title('Vout vs Vin')
plt.legend()
plt.grid()
plt.savefig(f'Vout_vs_Vin{trial_name_}.png')
plt.show()

# slope2 from frequency vs slope

slopes_all_freq = np.array(slopes_all_freq)
error_slopes_all_freq = np.array(error_slopes_all_freq)
frequencies = np.array(frequencies)

slope2, _ = np.polyfit(frequencies,slopes_all_freq,1)

# calculate the standard deviation of the slope2
slope2_std = np.sqrt(np.sum(error_slopes_all_freq**2) / (len(frequencies) - 2)) /
    np.sqrt(np.sum((frequencies - np.mean(frequencies))**2))

plt.figure(figsize=(10,5))
plt.errorbar(frequencies,slopes_all_freq, fmt='o-', label=f'Slope2 = {slope2:.3e}
    +/- {slope2_std:.3e}')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Slope (Vout/Vin)')
plt.title('Slope vs Frequency')
plt.legend()
plt.grid()
plt.savefig(f'Slope_vs_Frequency{trial_name_}.png')
plt.show()

R = 1e3
R_f = 10e3

R = 0.998e3
R_f = 53.8e3

R_coil = 4.7e3

G = 1 + (R_f / R) # Gain

M = (R_coil * slope2) / (2 * np.pi * G)
M_error = M * (slope2_std / slope2)
relative_error = (M_error / M) * 100

print(f"Mutual Inductance = {M:.3e} +/- {M_error:.3e} H")
print(f"Relative error = {relative_error:.3f} % (with respect to the observed)")

# calculating the avg phase and std phase across all frequencies and trials
avg_phase = np.mean(Phase_obs_allfreq) # degrees
std_phase = np.std(Phase_obs_allfreq) # degrees

print(f"Phase = {avg_phase:.3f} +/- {std_phase:.3f} degrees")
```

Chapter 4

Observations, Calculations and Error Analysis

Find the implementations and files in the Github repository.

4.1 Sin wave reference - implementation

For the sin wave implementation, we tested with a normal signal as well as a noisy signal with a phase.

For the former, the input and reference signals are shown in Figure 4.1, and the signal recreated using the lock-in detection is depicted in Figure 4.2.

For the latter, The input and reference signals are shown in Figure 4.3, whereas the recreated signal is shown in Figure 4.4. The calculated outputs and errors (See section 4.5) are given below.

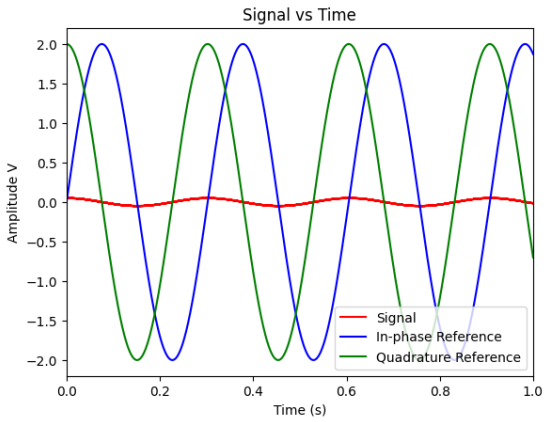

Figure 4.1: Input and reference signals - normal (sin implementation)

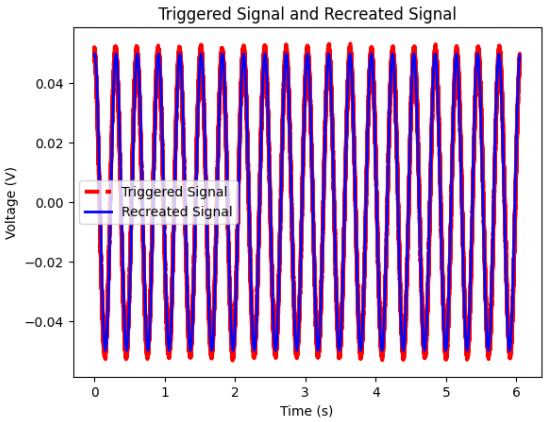

Figure 4.2: Input and recreated signal - normal (sin implementation)

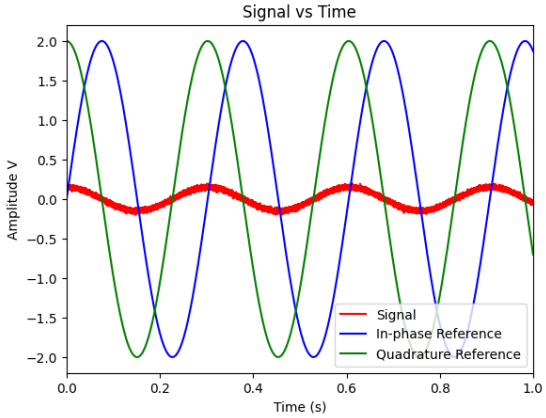

Figure 4.3: Input and reference signals - noisy (sin implementation)

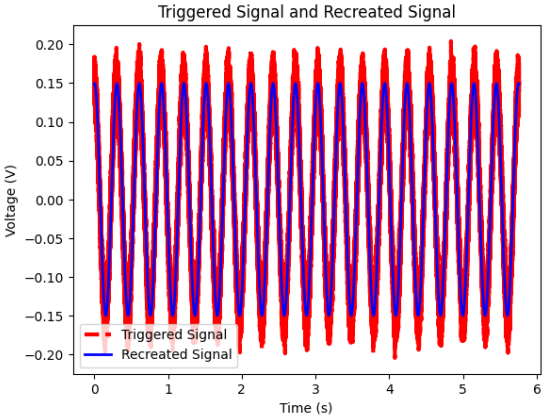

Figure 4.4: Input and recreated signals - noisy (sin implementation)

```
'''
OUTPUT
------
Calculated Amplitude: 0.1497 V
Calculated Phase: 82.9352 degrees
Calculated Frequency: 3.3069 Hz

Original Amplitude: 0.15
Original Phase: 90.0
Original Frequency: 3.3

Relative Error for Amplitude: 0.19828005884412842 %
Relative Error for Phase: 7.849793339801733 %
Relative Error for Frequency: 0.20926491417123058 %
'''
```

4.2 Square wave reference - implementation

For the square wave back-end, we did the same testing, and for the normal signal, the input and reference signals are shown in Figure 4.5, and the signal recreated using the lock-in detection is depicted in Figure 4.6.

For the latter, The input and reference signals are shown in Figure 4.7, whereas the recreated signal is shown in Figure 4.8. The calculated outputs and errors for this (See section 4.5) are given below.

```
'''
```

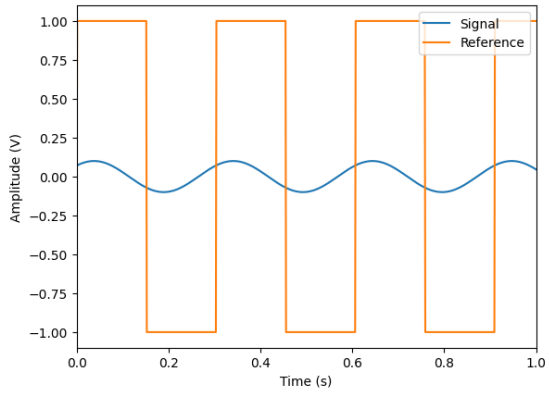

Figure 4.5: Input and reference signals - normal (square implementation)

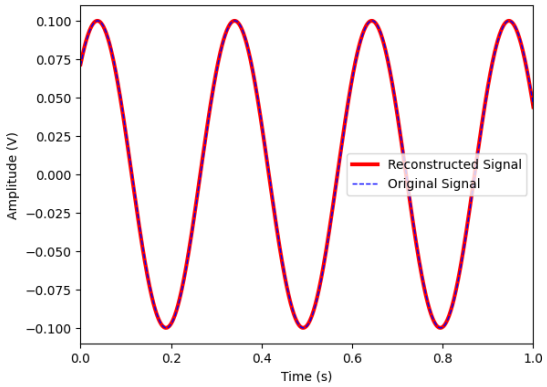

Figure 4.6: Input and recreated signal - normal (square implementation)

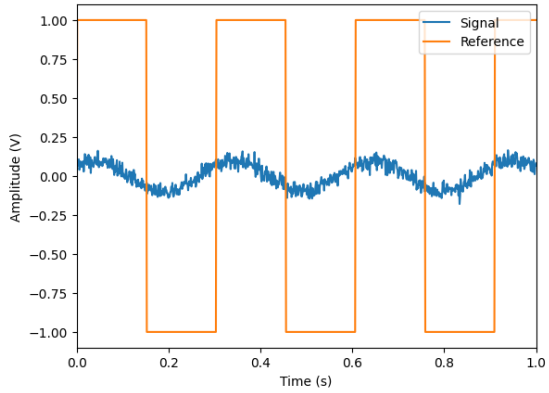

Figure 4.7: Input and reference signals - noisy (square implementation)

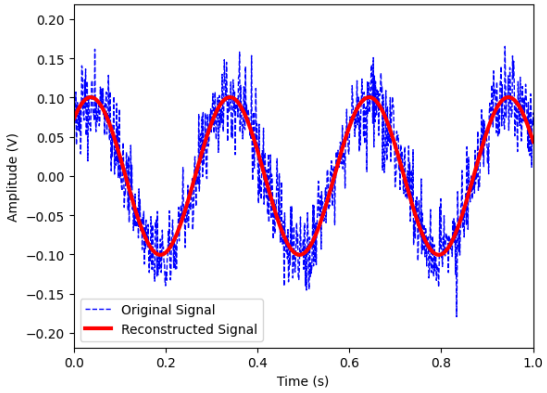

Figure 4.8: Input and recreated signals - noisy (square implementation)

```
OUTPUT
------
Observed Amplitude: 0.10046719832951305
Observed Phase: 45.93217260846667 deg

Original Amplitude: 0.1 V
Original Phase: 45.93217260846667 deg

Relative Amplitude Error: 0.47 %
Relative Phase Error: 2.07 %
```

4.3 Low Resistance

With the circuit given in Figure 2.1, we placed a 100 Ω small resistor, 4.7 $k\Omega$ as the large resistor, and acquired data. With the Square wave implementation of Lock-in detection implemented in Section 3.3.2 and steps for measurement of low resistance in Section 3.4, we got the following results.

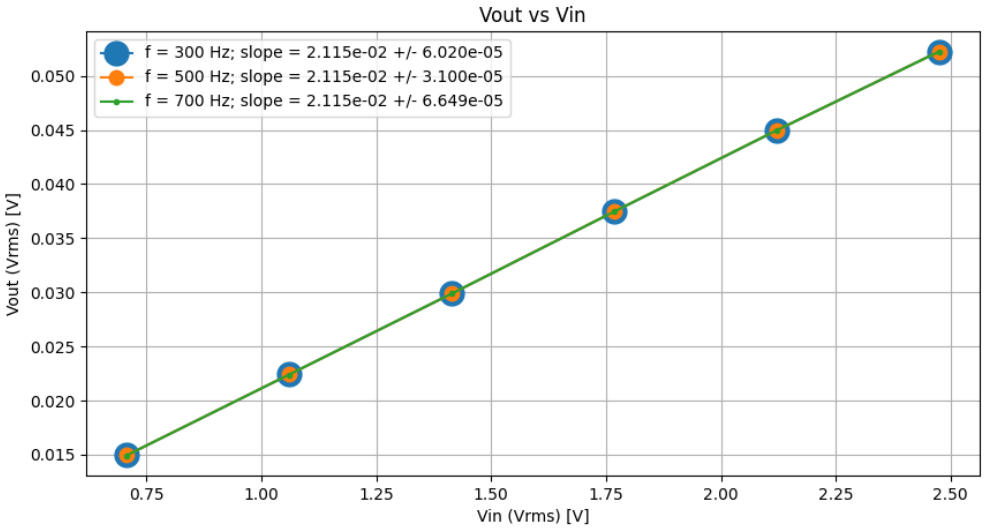

Figure 4.9: The calculation of slope for low resistance measurement for difference frequencies

As per equation (2.19), with $V_{\text{out}}/V_{\text{in}}$ as slope, we can write:

$$r = \frac{R.(\text{slope})}{\alpha} \tag{4.1}$$

where $\alpha = 1$ as discussed in 3.4. The acquired data and the slope calculated are shown in Figure 4.9 and the independence of this slope with respect to frequency is shown in Figure 4.10. The error calculation equation is discussed later. All these are implemented with the code, and the final output is:

```
R_observed = 99.401(+/-)0.146 ohm
R_theoretical = 100 ohm
Relative error = 0.599%
```

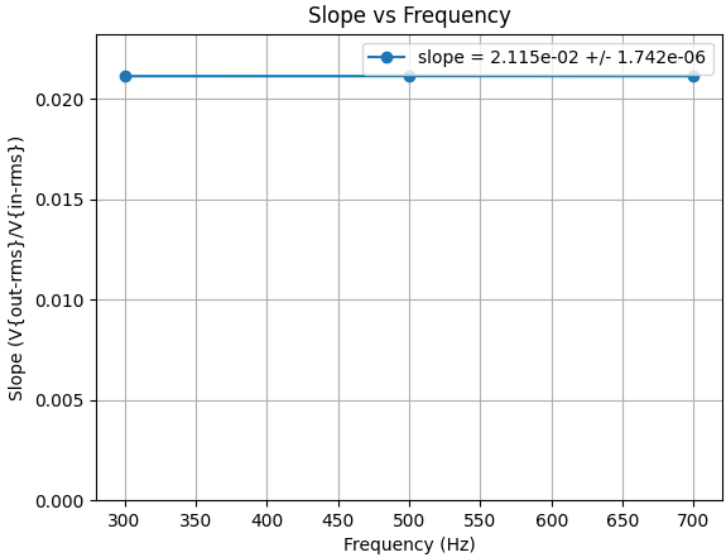

Figure 4.10: Independence of slopes in Figure 4.9 w.r.t. the frequency

4.4 Mutual Inductance

With the circuit given in Figure 2.2, we used a 0.998 kΩ (measured) input resistor and a potentiometer whose resistance was set to 53.8 kΩ as the feedback resistor for the Op-amp. With the Square wave implementation of Lock-in detection implemented in Section 3.3.2 and steps for measurement of mutual inductance in Section 3.5, we got the following results.

As per equation (2.23), with V_{out}/V_0 as slope1, we can write:

$$M = \frac{(\text{slope1})R}{2\pi f\alpha} \tag{4.2}$$

We calculated slope1 as the equation; the result is shown in Figure 4.11. By drawing slope1 vs f and taking the slope2 of this, we can rewrite the above equation as:

$$M = \frac{(\text{slope2})R}{2\pi\alpha} \tag{4.3}$$

where is calculated using equation (2.24) and results shown in Figure 4.12. The final values of the calculations (errors discussed later) are given below.

```
Mutual Inductance = 1.451e-04 +/- 1.052e-06 H
Relative error = 0.725 % (with respect to the observed)
Phase = -96.864 +/- 4.311 degrees
```

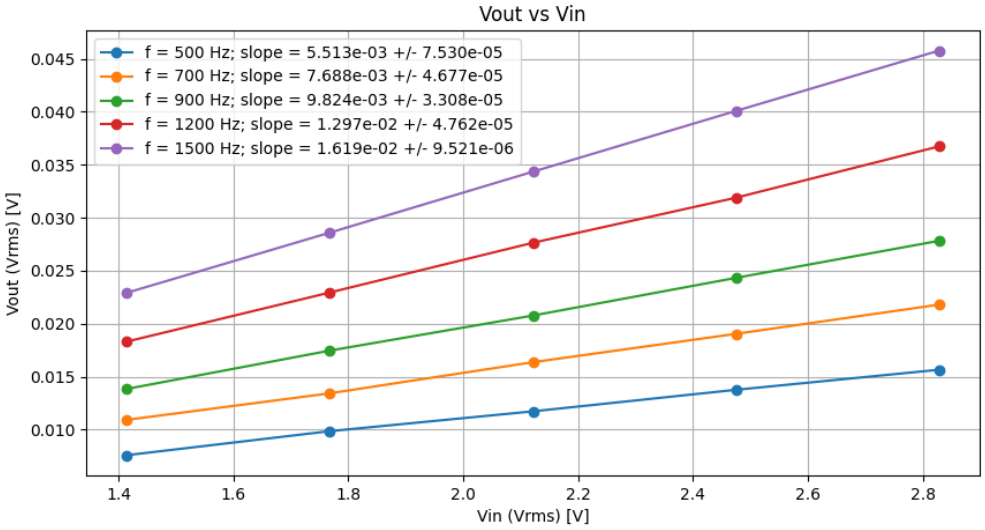

Figure 4.11: The calculation of slope1 for mutual inductance

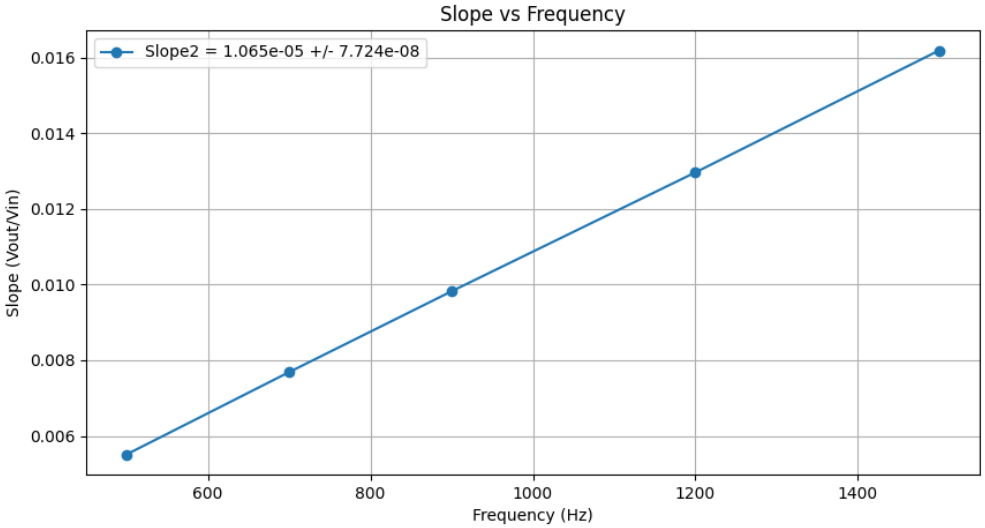

Figure 4.12: Calculation of slope2 from data in Figure 4.11

4.5 Error analysis

For the V_{out} vs V_{in} graphs, since the intercept is supposed to be zero, the slope (slope1 for mutual inductance) was calculated as:

$$\text{slope}_i = \frac{V_{\text{out}}}{V_{\text{in}}} \tag{4.4}$$

For each frequency (denoted by subscript i), the slope will be the average of this value over the data point, and the standard deviation ($\delta(\text{slope}_i)$) will give the corresponding error.

For the calculation of slope 2, the calculation is done by fitting the data, and the following equation would give the error in slope 2.

$$\delta(\text{slope2}) = \frac{\sqrt{\sum_{i=1}^{n} \frac{\delta \text{slope}_i^2}{(n-2)}}}{\sqrt{\sum_{i=1}^{n} (f_i - \bar{\bar{f}})^2}} \tag{4.5}$$

For independent measurements x_i of the same quantity, the mean and error are given by:

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i \tag{4.6}$$

$$\delta x = \sqrt{\frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2} \tag{4.7}$$

For finding relative error of a quantity $x \pm \delta x$ with respect to the ground truth $\bar{x}$, w have:

$$\text{relative error}(\%) = \frac{x - \bar{x}}{\bar{x}} \times 100 \tag{4.8}$$

$$\text{relative error}(\%) = \frac{\delta x}{x} \times 100 \quad (\text{w.r.t. } x) \tag{4.9}$$

Chapter 5

Discussions

Why is the phase detection giving higher error relative to amplitude?

Any phase introduced by the electrical components in the experiment will alter the phase of the signal of interest and that will equally affect the in phase and quadrature products. Since according to eq.s (2.6) and (2.14) shows that the division of the DC components depends on the phase part ($sin(\phi)/cos(\phi)$). This errors can also couple with errors from FFT stemming from not having integer number of cycles. There are solutions to the errors from FFT such as triggering the signal from right side, or applying techniques like like Hann windowing which fades the signal from both ends. Since our sample size and sampling frequency are limited, the application of Hann windowing will limit the information whereas the right clipping did improve results without this loss.

Why is the sampling frequency varied for each frequency or signal?

Since we are using `capture2`, it was found that signal and reference can have only 5000 samples each. Since this is the case, we decided to sample the signal in such a way that the time gap between two consecutive samples is calculated so that at least 64 samples are taken per cycle. Thus we get a fair sampling of the signal and enough cycles in a reasonable scope of frequencies.

This required primary information on frequency, which can be obtained with some initial sampling rate and an FFT of the acquired trial reference.

Why are we performing a warmup-test run before data collection?

While testing the capabilities and limitations of the expEYES-17 hardware, we found that the `capture2` does not work without any explainable reasons. `capture1` work perfectly fine at the same time.

While troubleshooting with various parameters, we found that the `capture2` with a high number of samples doesn't work unless we start from a smaller number of samples and slowly build up to the required value. For example, to get 5000 samples, we need to run with 500, 1000, 1500,.. etc., and build up to 5000 samples. Oddly enough, this 'warmup' needs to be done only once for a certain sampling frequency and we can continue reading till we change the sampling frequency or number of samples. Thus, we came up with the warmup loop for this purpose. See code in 3.1.

Why do we need $4N$ samples from each cycle for square wave reference?

The intuition behind this was not explained in the paper. Nonetheless, by going through some references, we have reached a possible basis for the intuition. If the signal contains high frequency components, we will need to sample at a higher rate to avoid losing information that is in the signal. Generally speaking, sampling at twice the signal's maximum frequency is required to retain all of its information. This is called the *Nyquist rate*.

In case of quadrature signals of square waves, we have half the cycle of in_phase holding the 0 (or 1) value and the quadrature signal only lagging by a quarter of cycle. This indicates 50% overlap and to differentiate, between these signals, we need at least 4 points per cycle for each.

Thinking in terms of practical quadrature multiplication, quadrature signal is like lagging the original reference by $(1/4)^{\text{th}}$ of a cycle which is possible with at least 4 points per cycle. Thus, at least 4 points are needed to retain information from quadrature signals multiplication.

Is triggering necessary, and why?

Triggering is required depending on the type of back-end we are using. As discussed before in Section 3, if the frequency from the reference signal is identified through FFT and then the in-phase and quadrature signals (either sin or square) are produced, we need to trigger the signal to align it properly.

If the quadrature signal is produced by shifting the signal electronically, or by finding the frequency and shifting by $(1/4)^{\text{th}}$ of a cycle, the triggering itself need not be necessary. Nonetheless, triggering is a importance tool to be available since it helps with visualization the signal better.

Are there better methods to optimize these processes?

As mentioned in the previous question, we can use Hann windowing and other techniques to improve the FFT accuracy. Improving the data collecting will be a huge help to the implementation. The hurdle there is to improve the efficiency, sampling frequency, and Number of samples without weight on the resource cost. ExpEYES was resource efficient with its low cost but flexible sampling. The overall python codes can also be made more efficient in terms of memory and parallelized which is too advanced for the current experiment and implementation. This can possibly improve the implementation to real-time signal lock-in detection. The real-time signal lock-in is on standby, specifically due to the need of 'warmup' mentioned before and other extra options we added. Correcting this might need a deep dive in the packages of `eyes17` python package itself.

Can this implementation be called a Lock-in amplifier?

Since there are no components in the Lock-in implementation that amplifies the signal value, we are simply detecting the signal in its original form. But, as we did it for mutual inductance experiment where the signal is extremely weak (below the

threshold of resolution of data collection), we can add an amplified with certain gain G, and then pass the signal to our Lock-in implementation. Since the process is robust to noise, the Lock-in detector will still detect the amplified signal, even amidst amplified noise. Then we can use the gain G to figure out the original value.

Strictly speaking, the implementation does not amplify the signals since the resolution remains the same despite digital scaling, which does not improve the results, but simply scale it. Hence, it makes more sense to call it Lock-in detection.

5.1 Sources of error

The possible sources of error are:

- Due to the nature of electrical components, the circuits in use can cause a change in phase with respect to the expected one.
- The limited resolution and voltage limits of the ExpEYES-17 hardware.
- Errors within the ExpEYES-17 hardware in the collection of data (rare glitches).
- Use of incorrect or less precise circuit components and signal conditioning components, including filters and amplifiers, can cause distortion in the signal.
- The practical experimental setup (long wires, inefficient connections) causes power dissipation. Overheating of components causes additional losses in the circuit.
- The FFT works better with an integer number of cycles, especially for sin wave back-end and phase detection.
- Errors can be introduced by changes in external variables, such as ambient noise, temperature fluctuations, and electromagnetic interference.

- Irregularities in quadrature form of reference.

5.2 Precautions

Several precautions were taken during the experiment, including:

- Components with values closest to the specified ones are chosen with care.
- Components are individually checked, and the circuit is kept compact as much as possible using efficient connections. The circuit is turned off and kept at a moderate temperature and condition for observations.
- Securely fix all components to the breadboard, check for short circuits within the circuit, and test the signals before processing.
- Choose the Voltage limits and resolutions of A1 and A2 inputs of ExpEYES-17 according to the signal being measured.

Chapter 6

Results and Conclusions

The implementation of Lock-in detection using sin-wave and square-wave references has been illustrated in Sections 4.1 and 4.2 respectively.

As for the results of each experiment, through the measurement of low resistance using Lock-in detection, we measured the resistance to be $(99.4 \pm 0.1)\Omega$ whereas the ground truth was 100Ω.

For the measurement of mutual inductance, we achieved the following results:

- The emf is proportional to the input Voltage as shown in Figure 4.11.
- The emf is proportional to frequency as shown in Figure 4.12.
- There is a phase difference of $90°$ between the input and output as per the result of phase $= (-97 \pm 4)°$.
- The value of mutual inductance was calculated to be $(145 \pm 1)\mu H$, whereas the ground truth is said to be close to $150\mu H$ as per experiments from the previous semester.

To sum up, this project has illustrated how lock-in detection approaches may be used in practice with ExpEYES. It has shown how to use both a square wave reference signal and a sine wave reference for standard lock-in detection. A set of tests centered on low-resistance measurements and mutual inductance has confirmed that lock-in detection works well for separating weak signals from noisy backgrounds.

References

[1] G. Li, S. Zhang, M. Zhou, Y. Li, L. Lin. 2013. A method to remove odd harmonic interferences in square wave reference digital lock-in amplifier. *Review of Scientific Instruments*. 84 (2).

[2] Inter-University Accelerator Centre. ExpEYES-17 User Manual: Experiments for Young Engineers and Scientists. 97-99. [Link]

[3] M. Li, Y. Sun, Z. Liang, S, Zhang. 2023. Square wave reference digital lock-in detection using non-orthogonal demodulation. *Heliyon*. 9 (1).

[4] NISER. Teaching Laboratory Manual: Lock-in Amplifier. [link]

[5] S. Bhattacharyya, R. Ahmed, B. Purkayastha, and K. Bhattacharyya. 2016. Implementation of digital lock-in amplifier. *J. Phys.: Conf. Ser.* 759 012096.