Investigation of a Low-Cost Heterodyne Interferometry System for Diagnostics for MAST-U

Summer Placement Project 2017

Christopher J. Hickling

August 25, 2017

Contents

1 Abstract **1**

2 Introduction **2**
2.1 Interferometry 2

3 Experimental Set-up **4**
3.1 Red Pitaya 4
3.2 Koheron Laser Board 5
3.3 Koheron Client 7
3.4 Double Mach-Zehnder Interferometer Setup 8

4 Experimental **9**
4.1 Collecting data 9
4.2 Manipulating Data 10
4.3 Inducing a phase change by the use of a speaker 16

5 Bibliography **18**

Appendix A Code **ii**
A.1 Saving Data ii
A.2 Hilbert Transform on Saved Data v
A.3 Hilbert Transform in Real-time x

Abstract

The aim of this project is to investigate and test the effectiveness of a new interferometer system for diagnostic purposes. This system is based on the use of a 1550nm laser diode interferometer system designed to measure the electron density of plasma in MAST-U (Mega Amp Spherical Tokamak - Upgrade). The benefit of using this new laser system over the old is that because of the low cost, and lack of health risks, many interferometer channels can be made to look at multiple interesting points of the Tokamak, e.g. X-Point in divertor region.

Introduction

2.1 Interferometry

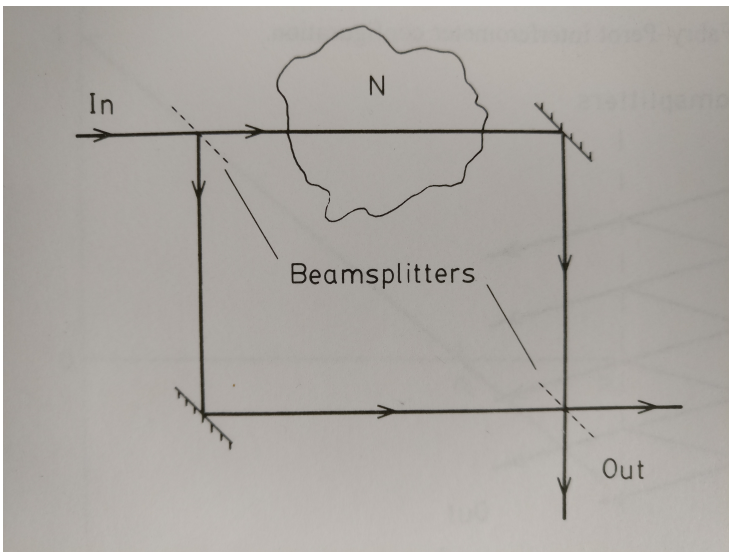

Figure 2.1. A schematic for a Mach-Zehnder interferometer where the Region N corresponds to a region of plasma. Image taken from Principles of Plasma Diagnostics [1, p. 97]

An interferometer, by design, is an incredibly sensitive piece of distance measuring apparatus. They work on the principle of interference between two laser beam paths. If a laser is split into two beam paths of unequal lengths, an interference pattern will form if they are recombined. The rate at which this interference pattern chirps is proportional to the difference in path length of the arms. If, for example, the path lengths were identical, no interference would take place upon recombination. By inserting a plasma into one of these arms a path difference will become apparent as the density will change the refractive index of the plasma. A higher density results in a higher refractive index, and therefore a larger apparent path length.Usually a standard Mach-Zehnder interferometer set-up is used [1], seen in figure 2.1, as this uses two arms that do not pass each other.

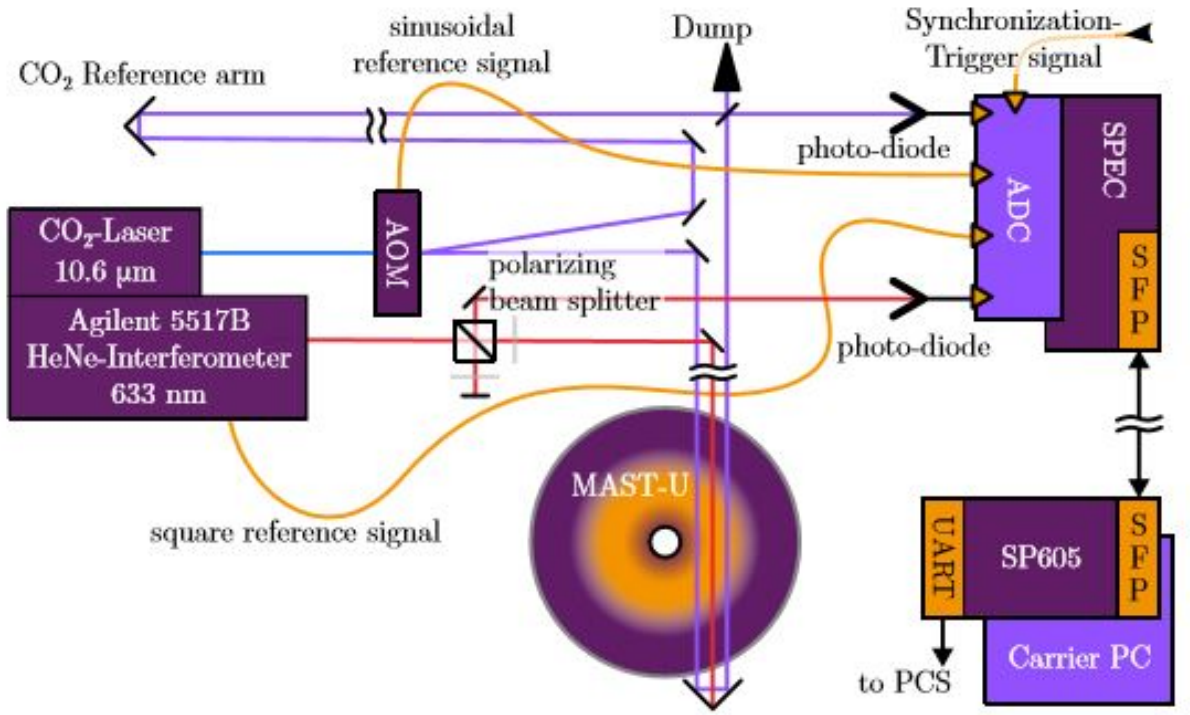

Figure 2.2. A schematic of the interferometer currently in place on MAST-U. This shows the complexity of the system and the difficulty of expanding it to include more interferometers. [2]

Currently in MAST (Mega Amp Spherical Tokamak) a heterodyne system is in place. This consists of a CO_2 laser in combination with a HeNe laser integrated with a FPGA system designed by J.K.Brunner [2]. The choice behind these lasers is that 10.6μm wavelength has strong interaction with the plasma. The HeNe does not have a high level of interaction with the plasma. This means the two lasers can be sent along the same beam path and interfere. So long as the modulation frequency of each of the lasers is known to a high precision, the plasma density can then be derived. One of the largest limitations to this system is the temperature dependence of the CO2 laser. The laser needs to be water-cooled in order to have stability in temperature. If it does not have a stable temperature the laser output will be effected, and since this system by design is sensitive, this would heavily effect results.
Another limitation is the cost of the CO_2 laser, as it is very expensive and requires a large amount of maintenance. The wavelength of the laser is chosen to have strong interaction with the plasma, but it also has strong interaction with human skin and eyes. As a result this laser has very strict safety precautions and requirements that come with its installation. These precautions are made more apparent since the laser has to enter the vacuum vessel via a glass viewing port which leaves the beam path exposed for a short distance.

Experimental Set-up

3.1 Red Pitaya

The Red Pitaya (RP) [3] is a open source micro controller with a built in Field Programmable Gate Array (FPGA) with a very user friendly user interface. It operates two inputs and two outputs at 125MS/s. FPGA's are notorious for being difficult to program as they are intrinsically complicated. The stand out function of a FPGA is the way it processes information. It can perform many tasks in parallel pipelined processing i.e. performing the same operation on many data sets. The number of parallel pipelines that can be implemented depends entirely on the hardware limitations of the FPGA. This often means that the limiting factor of the speed at which a RP can process data is the speed at which you can transmit data to it. This gives FPGA's a large advantage over conventional computers when it comes to real time data applications.

Figure 3.1. The Red Pitaya showing all the connected ports. Each of the analogue ports can sample at 125MHz.

Name	Type	Connector	Description
IN1/IN2	Input	SMA-F	RF input(High-Z, 1MΩ // 10pF
OUT1 / OUT2	Output	SMA-F	RF output (50Ω)
Ethernet	Full Duplex	RJ45	1000 Base-T Ethernet Connection
USB	Full Duplex	A USB	Used for standard USB Devices
Micro USB (Console)	Full Duplex	Micro B USB	Used for console connection
Micro USB (Power)	Input	Micro B USB	5V / 2A power supply
Micro SD	Full Duplex	Micro SD slot	Micro SD memory card

Table 3.1. Full Specifications of the Red Pitaya's input/output ports. Both fast analogue inputs and fast analogue outputs operate at 125MHz per channel.

3.2 Koheron Laser Board

The Koheron Laser board [4] that docks onto the RP is equipped with a 1550nm laser and a 60MHz photo-detector, this can be seen in figure 3.2. The laser has a maximum output of $3mW$ which marks the laser as a Class 1, meaning it is safe for any activity with no need for personal protective equipment. In addition to this to enable a second channel an addition of a Koheron PD100 photodetector board [5] (figure 3.3) is also needed. Using the RP and the Koheron Laser Board a simple interferometer can be built [6].

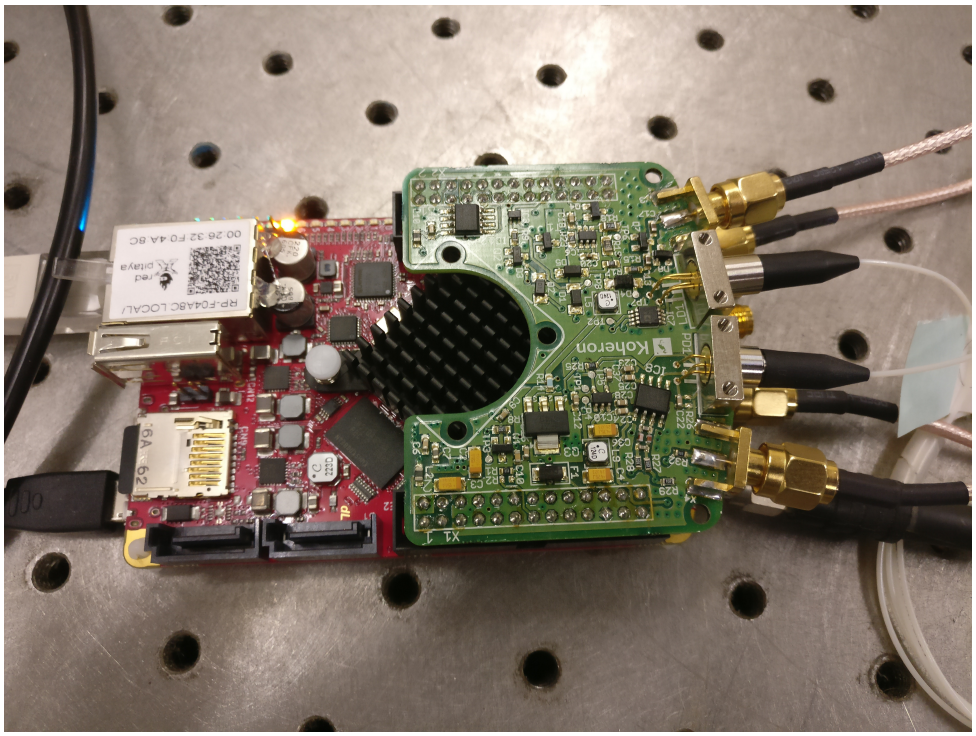

Figure 3.2. An image of the Red Pitaya with the laser board attached. From the top of the image to the bottom, the laser board has the connections; Input (setting laser current and modulation from the RP), laser output, laser input (photodetector) and output (connects to the RP).

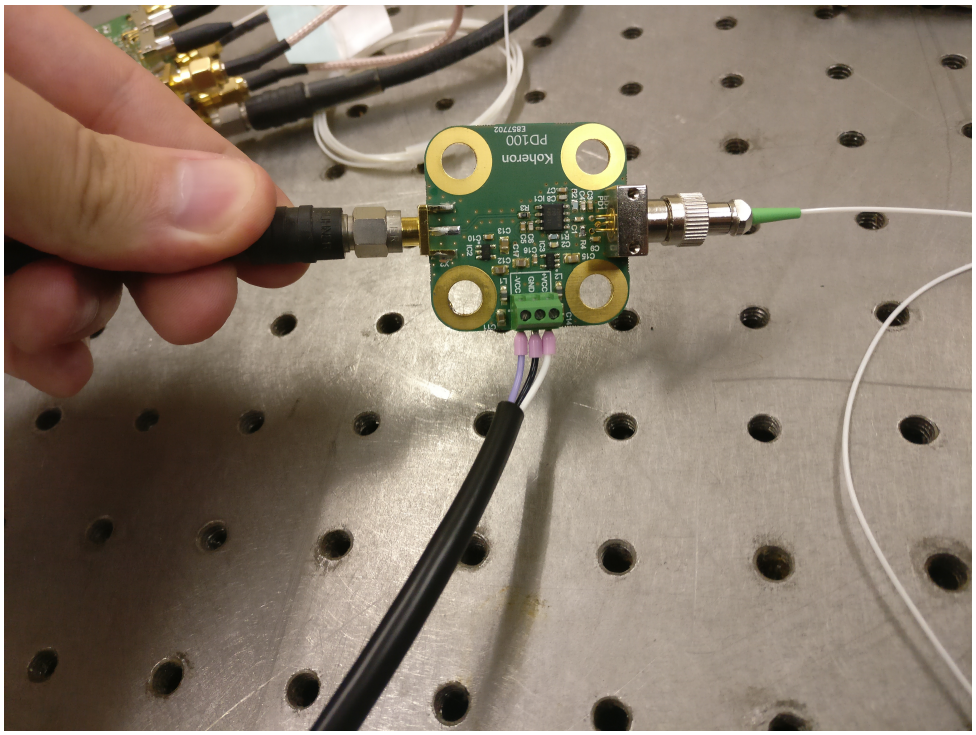

Figure 3.3. The PD100 laser board takes a fibre optic input and converts it into an electrical signal using an Indium Gallium Arsenide photo diode. The PD100 requires a DC current rated between -15 to +15V. Best optimization was found at -6.8V and +6.8V. Increasing this voltage causes the signal to saturate.

3.3 Koheron Client

As part of the received package from Koheron, the RP comes preloaded with the software to run the Koheron client [7]. This gives full control over the laser system including a visual output of the data collected by the two inputs on the board. The main benefit of this is that it gives an immediate representation of what applying different modulation frequencies to the laser results in. The number of chirps that can be seen in a discreet period of $65\mu s$ is dependant on the modulation frequency applied. The chirps always occur at the maxima or minima of the modulation. In an entire wavelength there should be one maxima and one minima. Modulation can also only be applied in integer divisions between 4096 (equation 3.2) and 8192 (equation 3.1). This is a limitation designated by the hardware. Hence, modulations can be achieved between the following values.

$$125 * 10^6 Hz/8192 = 15300Hz = 15.3KHz \tag{3.1}$$

$$125 * 10^6 Hz/4096 = 62500000Hz = 62.5MHz \tag{3.2}$$

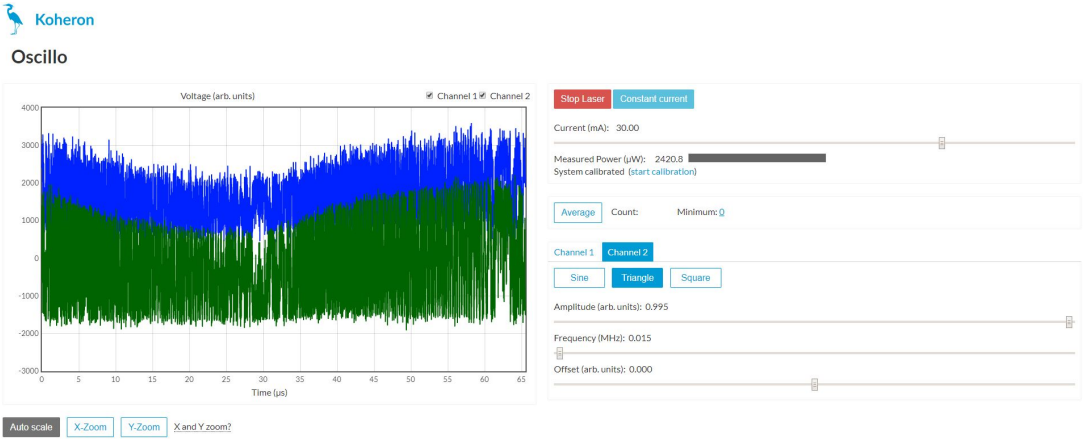

Figure 3.4. Raw data within the oscillo client. The way in which the current and modulation are set can be seen. The blue dataset shows the scene interferometer, the green shows the reference.

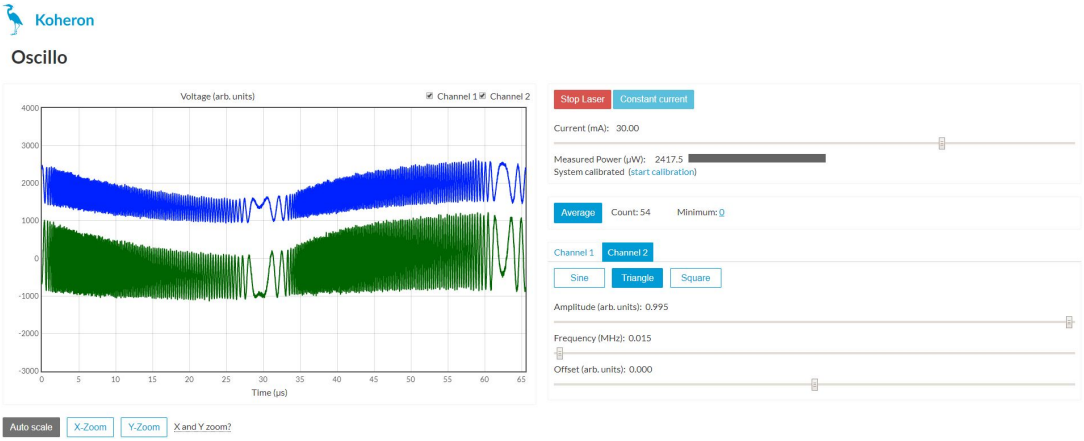

Figure 3.5. Averaged data within the oscillo client. The way in which the current and modulation are set can be seen. The blue dataset shows the scene interferometer, the green shows the reference.

3.4 Double Mach-Zehnder Interferometer Setup

By setting up two Mach-Zehnder interferometers using the same laser input but having isolated photodetector outputs a more realistic temperature correction can be made. The system is made out of single mode fibres, using couplers as beam splitters. This is because if you actually have two interferometers, a reference and a scene, each consisting of two arms. The reference interferometer has a beam splitter to split the beam, a 5 meter extension and then a beam combiner. This causes a consistent interference pattern where the rate of chirp is proportional to the temperature difference within the laser diode.

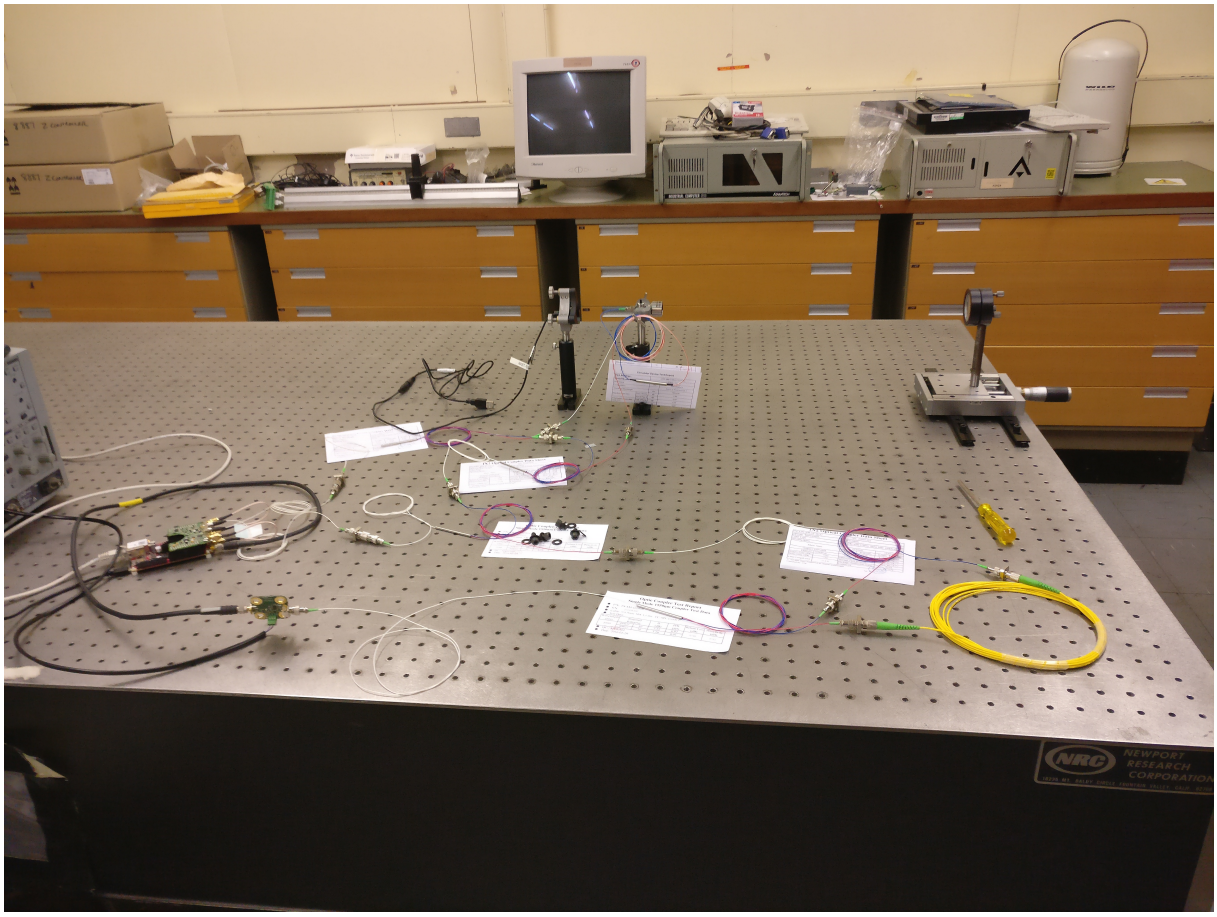

Figure 3.6. The full setup showing the two interferometers fully connected. The scene extension of the scene interferometer can be seen at the back, as the airgap is between the collimator and the translation stage. The reference interferometer extension is the yellow winding of single mode fibre cable in the bottom right. There is a laser diode behind the collimator only for the purpose of aligning the mirror.

This is in contrast to the scene interferometer, where the extension is provided by 4m of circulator fibres, into a collimator and a half a meter air gap to reflect off of a mirror and back into the collimator (1m air gap in total). This results in the same 5m extension but this interferometer has phase shifts proportional to the temperature fluctuations and phase differences caused by vibrations in the mirror.

Experimental

4.1 Collecting data

In order to collect data from the RP code must be written to interface with the RP and the Koheron Client. There are already many inbuilt drivers and scripts in place but non of them fulfil all the needs of the project. Firstly in order to allow a computer to control the RP the Koheron package has to be installed. This was done via the software Anaconda [8] which contains python 2.7 [9] as well as many inbuilt python packages such as numpy [10] and scipy [11]. Using the Anaconda Prompt and typing "pip install koheron", the required packages will be downloaded to the computer. The closest program to the needs was "Oscillo.py". This had means to gather data from the DAC and also control the current supplied to the laser. By adding in the code and drivers required to define a modulation to the laser, the board could then be controlled entirely by a Python API instead of the client. Data can be collected from the DAC directly once the client has been interfaced with, this can be seen in appendix A.1.

Data collection is handled by decimated data, a driver written by Koheron (contained within the software development kit [7]). The way this driver works is it collects 16384 data points directly from the DAC and transferrers them to the PC. In order to collect the data as quickly as possible, this data collection is just performed in a single loop with no processing done on it. The collection and post processing could be done simultaneously at the cost of resolution of data. Currently 500 data sets (8192*2) can be collected in 5 seconds. This can be converted into a frequency by the following:

$$8192 * 2 = 16384$$

$$(500 * 16384)/5 = 1638400 = 1.64MHz$$

$$1.64/125 = 0.0131072 = 1.31\% \tag{4.1}$$

This means that in its current form, as shown by equation 4.1, 98.7% of data is not transferred to the PC. This is mostly due to the fact that the RP can handle the data much faster than the PC can accept it. This results in the PC in the system being a huge bottleneck. A potential fix for this could be to set-up a second RP as a server that sits in between the PC and the first RP. This would save data directly into its RAM and pass the data to the PC at a rate the PC can handle. This would not be useful in a real-time application scenario, but in terms of data analysis this could massively increase resolution.

4.2 Manipulating Data

This section is based mainly around the code listed under "Hilbert Transform on Saved Data" in appendix A.2.

Raw data has been collected from the RP as quickly as possible. This has now left many datasets saved in a CSV format. By selecting the folder location of the datasets, the user will be asked how many csv's they want to analyse. This is just a feature to limit compiling time to look for small details. The code works by first band passing the dataset by applying a Butterworth filter between specified frequencies. Next the filtered signal is passed through a Hilbert Transform function. The Hilbert transform is a linear operator that takes a real input and extends it into the complex plane. It is useful for defining instantaneous properties of functions e.g. Amplitude and frequency.

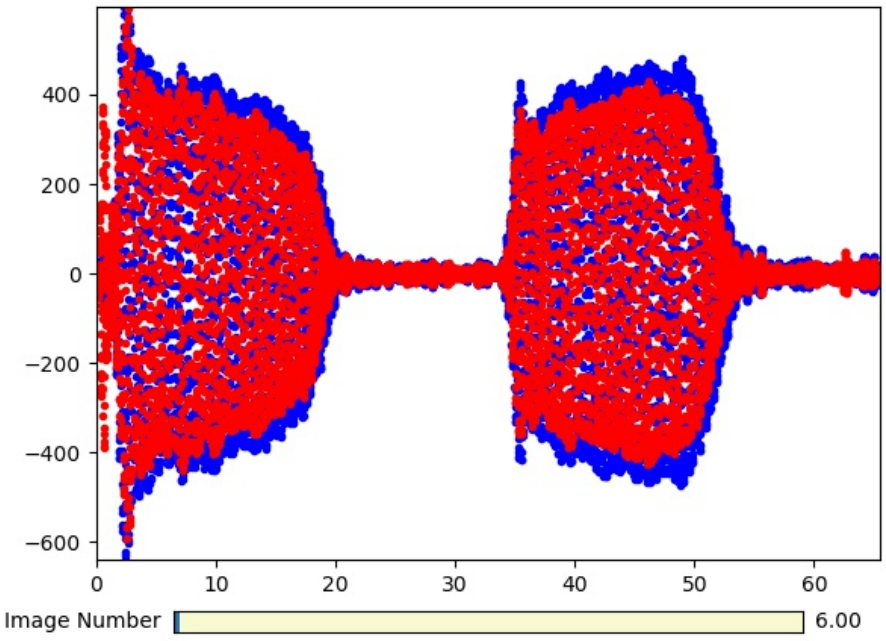

Figure 4.1. The instantaneous amplitude of a function, as generated from the Hilbert transform. This is simply the modulus of the transformed signal. It shows the envelope of the signal.

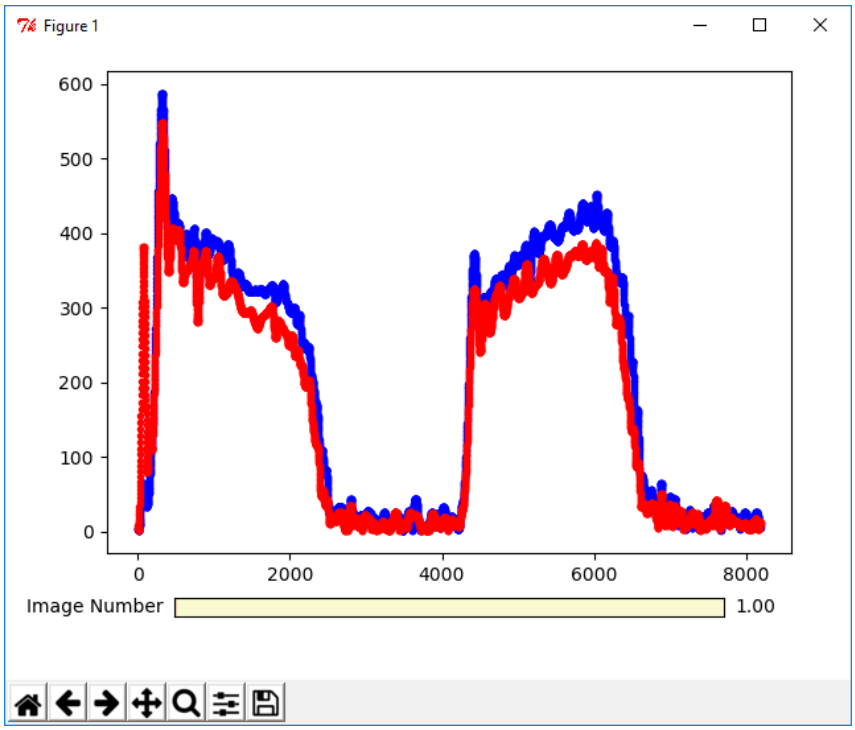

Figure 4.2. The instantaneous amplitude of a function, as generated from the Hilbert transform. This is simply the modulus of the transformed signal. It shows the envelope of the signal.

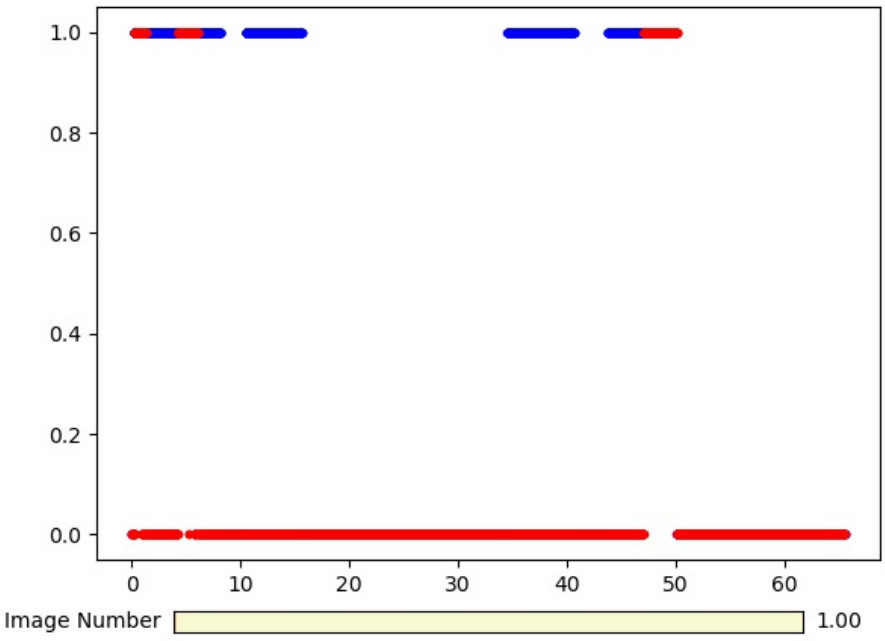

Figure 4.3. The threshold instantaneous amplitude around the mean of the instantaneous amplitude. This can now be used as an operator to isolate only the regions of clear information.

As show in figure 4.1, the data has been filtered around the fringe frequency. This has

eliminated the approximate region of the chirp. As then shown in figure 4.2, the envelope of this filtered signal is then outlined. This is useful, as we can now threshold this to ensure that only specified portions of the signal are analysed. This is done by finding the mean of the instantaneous amplitude and setting any values less than the mean to 0 and greater than the mean to 1. This can then act like an operator to isolate useful portions of the signal. This can be seen in figure 4.3.

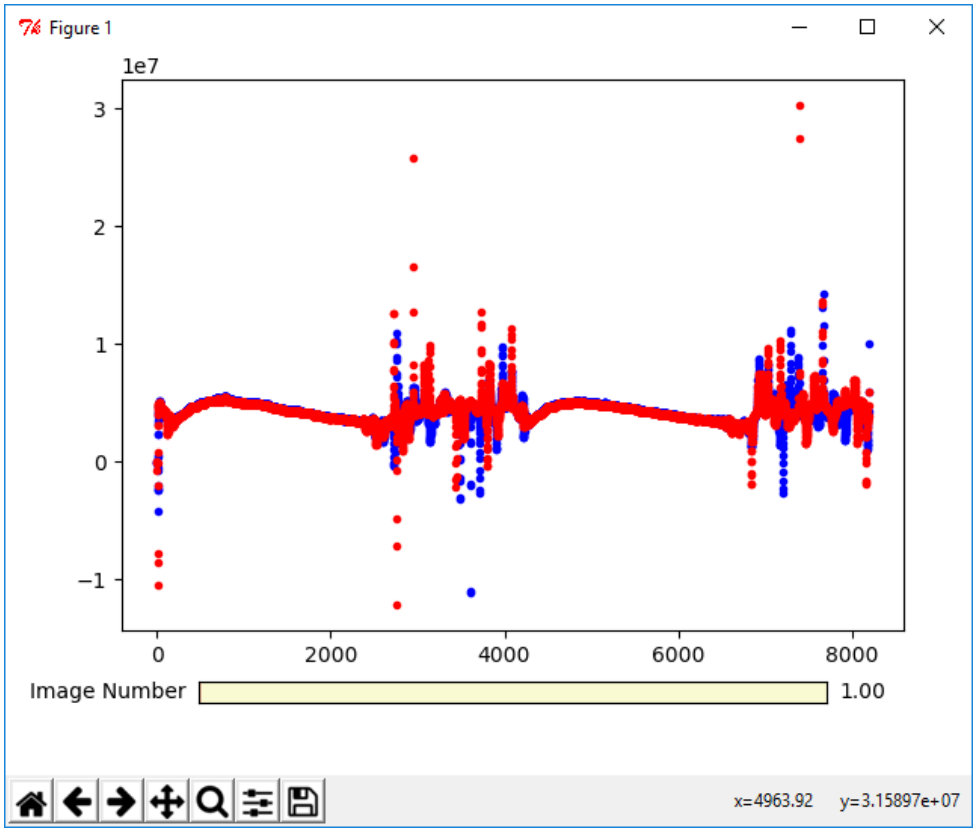

Figure 4.4. The instantaneous frequency as obtained by the Hilbert transform. X-Axis is data points against y-axis which is frequency. It is clear to see the bandpass is working, and the region that has been selected is a clear signal.

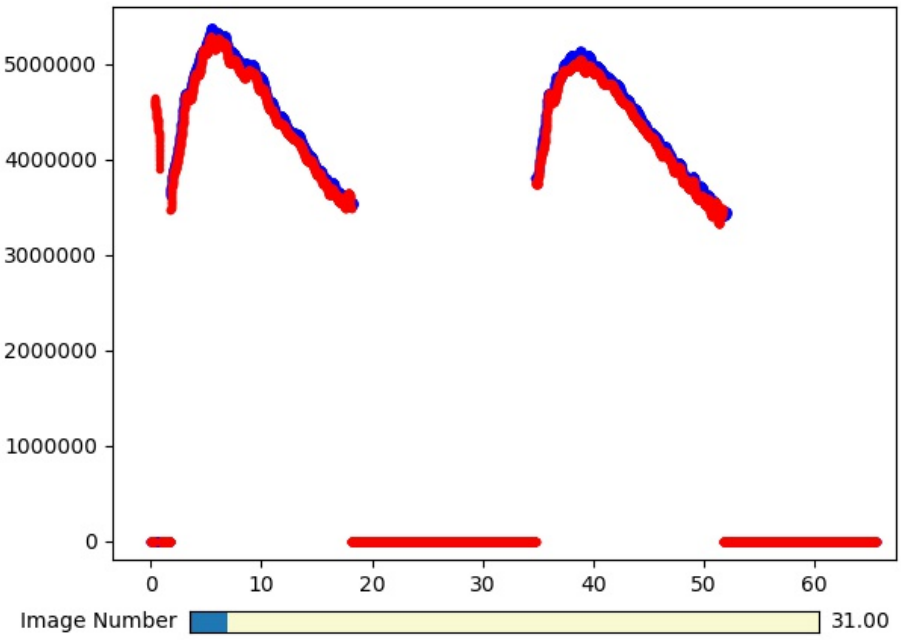

Figure 4.5. The instantaneous frequency has now been acted on by the threshold operator. This results in only the regions of clear, slightly varying frequency are shown. All other values appear to be zero.

The Hilbert transform can be used to obtain the instantaneous frequency also by performing a numpy-python function called "diff", which calculated the discreet difference across the signal. The resulting frequency can be seen in figure 4.4. This can be further developed by applying the thresholding operator to completely isolate the clear parts of the signal, as can be seen in figure 4.5. It can also be noted at this point that the fact that these two frequencies are not identical means that there exists a path difference between the two interferometers. For future application this will have to be corrected, but due to the fluctuations in the system a reliable path difference cannot be calculated.

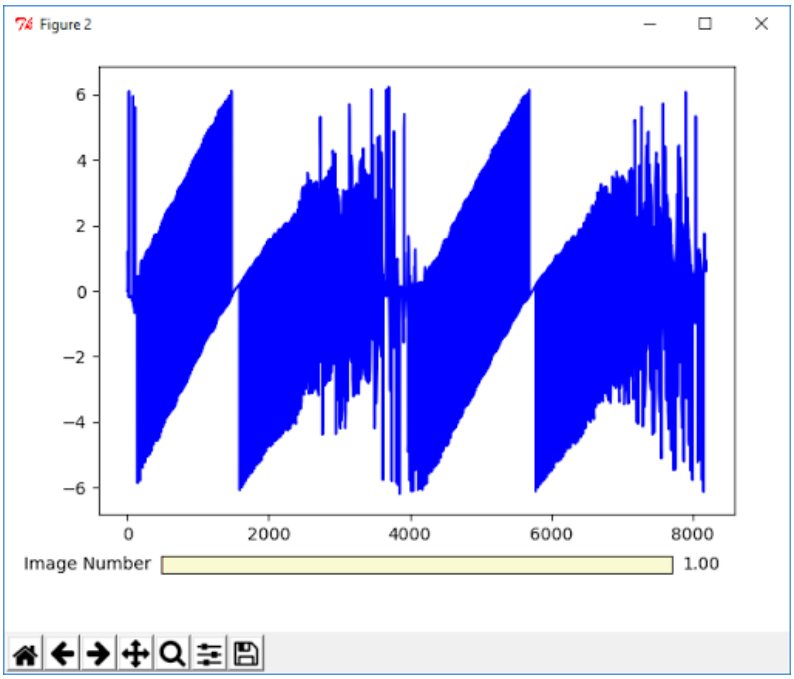

Figure 4.6

From the information in the instantaneous frequency the phase can also be obtained from a numpy-python [10] function called "angle". This works by using a arctan function on the real and imaginary parts of the signal. The result will look like a sawtooth wave as it is repeating between $\pm 2\pi$ degrees as shown in figure 4.6.

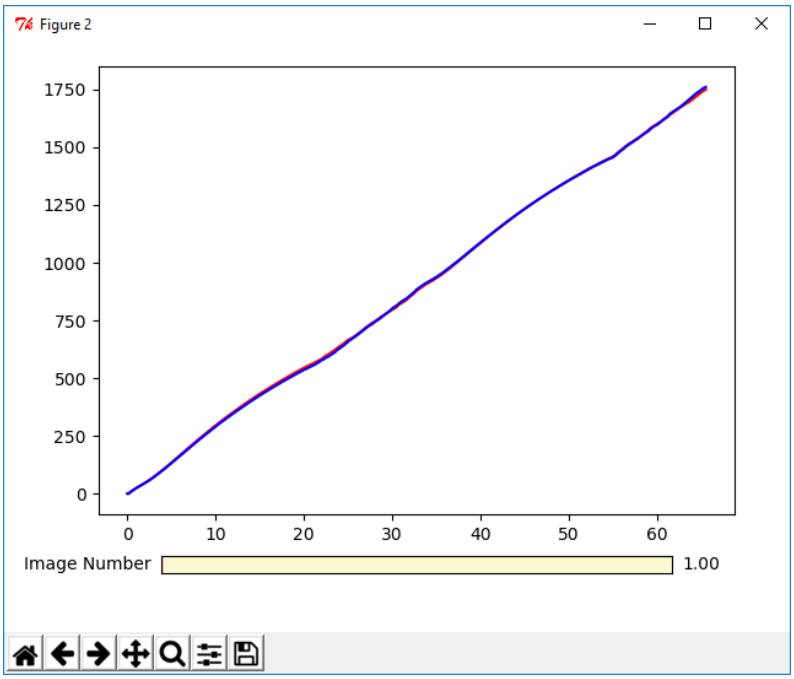

Figure 4.7. The unwrapped instantaneous phase of the two channels. A small deviation between the two can be seen. This largely represents the path difference between the two channels. However, this has large fluctuations as a function of time, hence it may also have a large temperature dependence.

By performing another Numpy-python function on the phase, "unwrap", this stops

repeating between -2π and +2π. As a result an accumulative phase is obtained, seen in figure 4.7.

4.3 Inducing a phase change by the use of a speaker

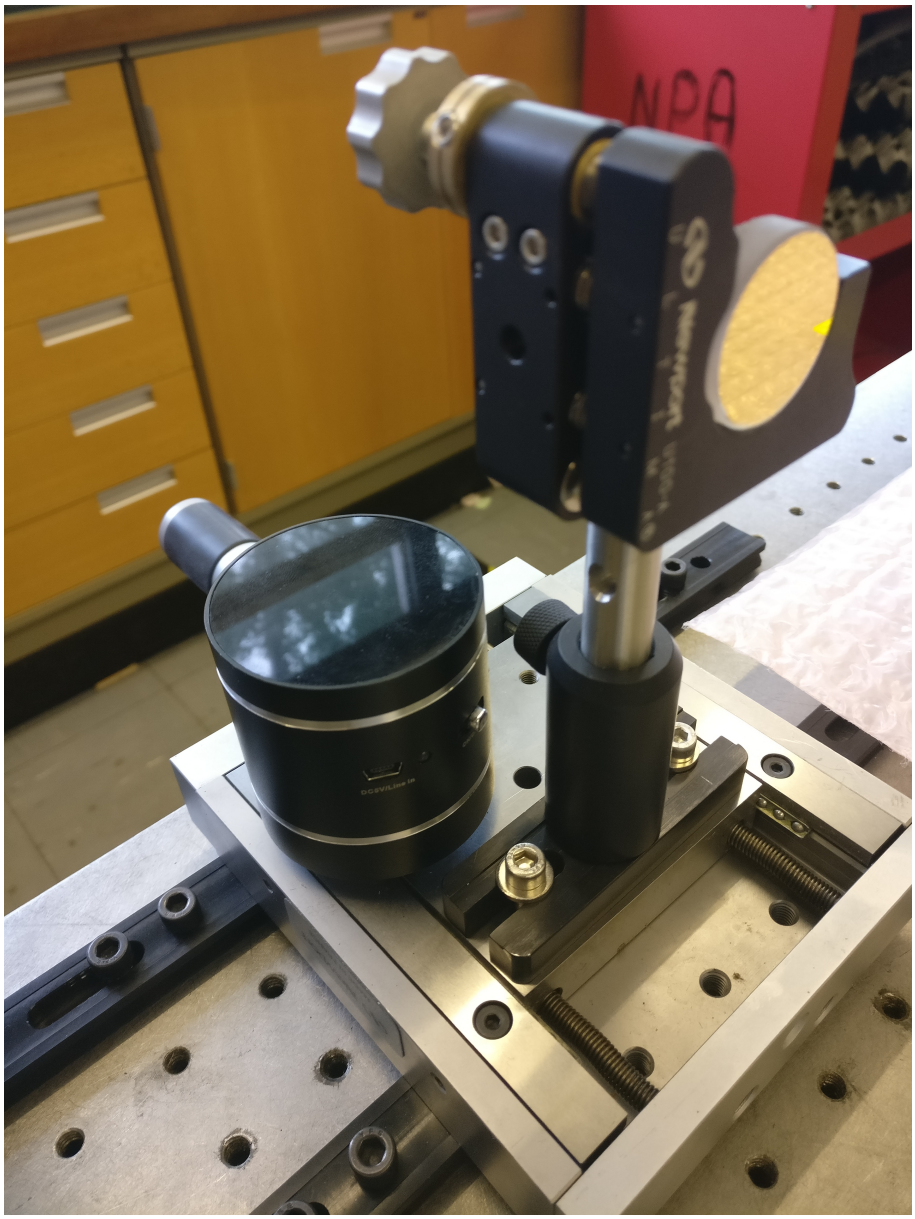

Figure 4.8. Vibrospeaker (a vibrating speaker) has been placed on the translation stage that is a mount for the mirror in the scene interferometer. By using bluetooth and a signal generator a single frequency can be sent to the speaker to play. This causes it to produce the sound, and hence vibrate at the requested frequency.

By using a vibrospeaker (a vibrating speaker) a frequency can be applied to the mirror in the hopes to add an additional modulation. In order to see this more clearly the Koheron program "Miniscope" was edited to output the unwrapped phase as a dynamic plot, appendix A.3. No visible changes could be seen with the use of the speaker, or the translation stage. It is believed this is because the sampling rate is too high, which is meaning that the frequency of the sound wave is not large enough to be seen on the time scale. To attempt to resolve this data was instead taken with an oscilloscope that had a variable sampling rate. This seemed to still not reveal this underlying vibration that

should be present from the vibrospeaker. Further investigation will be needed to explore the viability of this method as a diagnostic tool.

Bibliography

[1] I. H. Hutchinson. *Principles of Plasma Diagnostics*. Cambridge University Press, 2005, p. 460. ISBN: 9780521675741. DOI: 10.1017/CBO9780511613630.

[2] Kai Jakob Brunner. "FPGA-based High Performance Diagnostics For Fusion". PhD thesis. University of Durham, Mar. 2017. URL: http://etheses.dur.ac.uk/12026/%0A.

[3] Peter Leban. *Red Pitaya User Manual*. 2014. URL: http://docs-asia.electrocomponents.com/webdocs/144a/0900766b8144a693.pdf.

[4] Koheron. *Laser Board for the Red Pitaya v1*. URL: https://www.koheron.com/photonics/lbrp-laser-board-red-pitaya-v1.

[5] Koheron. *PD100 - Low Noise Photodetector*. URL: https://www.koheron.com/photonics/pd100-photodetection.

[6] Koheron. *A Simple Coherent Laser Sensor*. URL: https://www.koheron.com/blog/2015/09/10/laser-control.

[7] Koheron. *Software Development Kit*. 2016. URL: https://github.com/Koheron/koheron-sdk.

[8] Anaconda Continuum Analytics. *Anaconda Distribution*. 2016. URL: https://www.continuum.io/.

[9] Python Software Foundation. *Python Language Reference, version 2.7*. 2013. DOI: https://www.python.org/. URL: http://www.python.org.

[10] NumPy Community. *NumPy 1.7.0*. 2011. URL: http://www.numpy.org/.

[11] Pearu Peterson Eric Jones Travis Oliphant et al. {*SciPy*}*: Open source scientific tools for* {*Python*}. 0. URL: http://www.scipy.org/.

Code

A.1 Saving Data

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

#Prior to running Koheron must be installed to the python
    directory
#If using anaconda: anaconda prompt -> "pip install koheron"
from koheron import connect
from oscillo import Oscillo

import sys
import os
import numpy as np
import math
import csv

from Tkinter import Tk
import tkFileDialog

import matplotlib
matplotlib.use('TKAgg')
from matplotlib import pyplot as plt

#IP Address of the Red Pitaya (RP) can vary. It should show on
    the RP by yellow LED's
#each corresponding to a base-2 number. e.g. Two to the power of
     the number of the LED.
#If this is not functional, go to the commandprompt and type "
    nslookup Koheron"
host = os.getenv('HOST','10.209.1.125')
#opens a connection to the clientside of the application, in
    this case, oscillo.
client = connect(host, name='oscillo')

#Select a folder where you wish to save the data.
Tk().withdraw() # keep the root window from appearing
```

```
31 # show an "Open" dialog box and return the path to the selected
       file
32 filepath = tkFileDialog.askdirectory(title="Select A Folder",
       mustexist=1)
33 print(filepath)
34
35 #Choose how many data sets you wish to save. 1 data set ~= 0.01s
       in realtime.
36 loop = 5
37
38
39 driver = Oscillo(client)
40 driver.reset()
41 t = 1e6*driver.t
42
43 #initialise channel0 and 1 arrays for data collection.
44 #driver.wfm_size = 8192 data points = length of FIFO.
45 channel0 = np.zeros((loop, driver.wfm_size))
46 channel1 = np.zeros((loop, driver.wfm_size))
47
48 # Modulation on DAC
49 #Amplitude of the modulation
50 amp_mod = 0.995
51 #Frequency of the modulation. driver.sampling_rate = 125e6
52 freq_mod = driver.sampling_rate / driver.wfm_size * 10
53 #Input modulation for channel 0 and 1 accross all data points
       resepectively.
54 driver.dac[0, :] = amp_mod*np.cos(2 * np.pi * freq_mod * driver.
       t)
55 driver.dac[1, :] = amp_mod*np.sin(2 * np.pi * freq_mod * driver.
       t)
56 driver.set_dac()
57
58 time.sleep(0.001)
59
60 #Set averaging for the dataset. If false, no averaging is
       performed.
61 driver.set_num_average_min(2000)
62 driver.set_average(False)
63
64
65
66 #save data as array to gather asap
67 for i in range(0, loop):
68         driver.get_adc()
69         channel0[i] = driver.adc[0,:]
70         channel1[i] = driver.adc[1,:]
71
```

```
72 #write data 1 row at a time into csv
73 for i in range(0, loop):
74         write_file = filepath+ '\\' + str(i) + '.csv'
75         with open(write_file, "wb") as csv_file:
76                 csv_writer = spamwriter = csv.writer(csv_file,
                       delimiter=',', quoting=csv.QUOTE_MINIMAL)
77                 csv_writer.writerow(('Time(us)', 'Channel 0', '
                       Channel 1'))
78                 for j in range(0, channel0[i].size):
79                         csv_writer.writerow((t[j], channel0[i][j
                               ], channel1[i][j]))
80         csv_file.close()
```

A.2 Hilbert Transform on Saved Data

```
import os
import numpy as np
import sys
import math

import matplotlib
from matplotlib import mlab
matplotlib.use('TKAgg')
from matplotlib import pyplot as plt
from matplotlib.widgets import Slider


from scipy import signal
from scipy.signal import hilbert, chirp
from scipy.signal import butter, lfilter, filtfilt

from Tkinter import Tk
import tkFileDialog
import tkSimpleDialog

#pop up window which allows easier locating of folders
Tk().withdraw() # keep the root window from appearing
## show an "Open" dialog box and return the path to the selected
    file
filepath = tkFileDialog.askdirectory(title="Select A Folder",
   mustexist=1)
print(filepath)

#Works out how many files are in a folder.
path, dirs, files = os.walk(filepath).next()
#take 2 as these sepresent subdirectories
n_files = len(files) - 2
print(n_files)

#allows a selection of number of files wanting to be analysed
n_files_sel = int(tkSimpleDialog.askstring("Data Popup","There
   are "+str(n_files)+" in this folder, how many would you like
   to use?"))

channel0 = np.zeros((n_files_sel, 8192))
channel1 = np.zeros((n_files_sel, 8192))

#only collects columns 2 and 3 so as to save time. Column 3
   should be indentical in all cases because it is the timebase.
for i in range(1, n_files_sel+1):
        csv_data = np.hsplit((np.genfromtxt(filepath+"//"+str(i
```

```
            -1)+".csv", delimiter=",", dtype=float, skip_header=
            True)),3)
42          channel0[i-1] = np.ravel(csv_data[1])
43          channel1[i-1] = np.ravel(csv_data[2])
44  t = np.ravel(csv_data[0])
45
46
47  class DiscreteSlider(Slider):
48      """A matplotlib slider widget with discrete steps."""
49      def __init__(self, *args, **kwargs):
50          """
51          Identical to Slider.__init__, except for the new keyword
                'allowed_vals'.
52          This keyword specifies the allowed positions of the
              slider
53          """
54          self.allowed_vals = kwargs.pop('allowed_vals',None)
55          self.previous_val = kwargs['valinit']
56          Slider.__init__(self, *args, **kwargs)
57          if self.allowed_vals==None:
58              self.allowed_vals = [self.valmin,self.valmax]
59
60      def set_val(self, val):
61          discrete_val = self.allowed_vals[abs(val-self.
              allowed_vals).argmin()]
62          xy = self.poly.xy
63          xy[2] = discrete_val, 1
64          xy[3] = discrete_val, 0
65          self.poly.xy = xy
66          self.valtext.set_text(self.valfmt % discrete_val)
67          if self.drawon:
68              self.ax.figure.canvas.draw()
69          self.val = val
70          if self.previous_val!=discrete_val:
71              self.previous_val = discrete_val
72              if not self.eventson:
73                  return
74              for cid, func in self.observers.iteritems():
75                  func(discrete_val)
76
77  #Slider plot used to compare two data sets with defined x axis
78  def slider_plot(x_data, c0_data, c1_data, c0_args,c1_args):
79          axcolor = 'lightgoldenrodyellow'
80          image_spacing = np.linspace(1,n_files_sel,n_files_sel)
81          t_sp = x_data
82          fig, ax = plt.subplots()
83          plt.subplots_adjust(left=0.12, bottom=0.2, right=0.93,
              top=0.96, wspace=0.2, hspace=0.2)
```

```
84          l, = plt.plot(t_sp, c0_data[0], c0_args)
85          g, = plt.plot(t_sp, c1_data[0], c1_args)
86          ax_img_number = plt.axes([0.2, 0.1, 0.65, 0.03],
                facecolor=axcolor)
87          s_img = DiscreteSlider(ax_img_number, 'Image Number', 1,
                n_files_sel, valinit=1, allowed_vals=image_spacing)
88
89          def updateplot(val):
90                  fig_number = int(s_img.val)
91                  l.set_ydata(c0_data[fig_number-1])
92                  g.set_ydata(c1_data[fig_number-1])
93                  ax.set_ylim(min(c0_data[fig_number-1]),max(
                        c1_data[fig_number-1]))
94                  ax.set_xlim(min(t_sp),max(t_sp))
95                  fig.canvas.draw_idle()
96
97          s_img.on_changed(updateplot)
98
99  #Slider plot used to compare two data sets with undefined x axis
100 def slider_plot_no_x(c0_data, c1_data, c0_args,c1_args):
101         axcolor = 'lightgoldenrodyellow'
102         image_spacing = np.linspace(1,n_files_sel,n_files_sel)
103         fig, ax = plt.subplots()
104         plt.subplots_adjust(left=0.12, bottom=0.2, right=0.93,
                top=0.96, wspace=0.2, hspace=0.2)
105         l, = plt.plot(t[0:c0_data[0].size],c0_data[0], c0_args)
106         g, = plt.plot(t[0:c0_data[0].size],c1_data[0], c1_args)
107         ax_img_number = plt.axes([0.2, 0.1, 0.65, 0.03],
                facecolor=axcolor)
108         s_img = DiscreteSlider(ax_img_number, 'Image Number', 1,
                n_files_sel, valinit=1, allowed_vals=image_spacing)
109
110         def updateplot(val):
111                 fig_number = int(s_img.val)
112                 l.set_ydata(c0_data[fig_number-1])
113                 g.set_ydata(c1_data[fig_number-1])
114                 fig.canvas.draw_idle()
115
116         s_img.on_changed(updateplot)
117
118 #slider plot used to display a single data set across multiple
        image numbers.
119 def slider_plot_1_data(c0_data, c0_args):
120         axcolor = 'lightgoldenrodyellow'
121         image_spacing = np.linspace(1,n_files_sel,n_files_sel)
122         fig, ax = plt.subplots()
123         plt.subplots_adjust(left=0.12, bottom=0.2, right=0.93,
                top=0.96, wspace=0.2, hspace=0.2)
```

```
124         l, = plt.plot(t[0:c0_data[0].size],c0_data[0], c0_args)
125         ax_img_number = plt.axes([0.2, 0.1, 0.65, 0.03],
                facecolor=axcolor)
126         s_img = DiscreteSlider(ax_img_number, 'Image Number', 1,
                n_files_sel, valinit=1, allowed_vals=image_spacing)
127         def updateplot(val):
128                 fig_number = int(s_img.val)
129                 l.set_ydata(c0_data[fig_number-1])
130                 fig.canvas.draw_idle()
131         s_img.on_changed(updateplot)
132
133
134
135 ###~~~~~~~~~~Try to calculate the phase between the two channels
        ~~~~~~~~~~###
136
137 #function to perform a bandpass between selected frequencies
138 def bandpass_data(t_in,data_in,fmin,fmax):
139         tt=t_in
140         del_t=tt[1]-tt[0]
141         fs=1/(del_t)
142         highf=fmax / (fs / 2)
143         lowf=fmin/(fs/2)
144         B, A = butter(5, [lowf,highf], btype='bandpass')
145         filtered_signal = lfilter(B, A,data_in, axis=0)
146         return filtered_signal
147
148 #performes a hilbert transform returning amplitude, frequency
        and phase
149 def hilbert_signal_phase(signal,fs):
150     #if signal.size < 2*22:
151     #    analytic_signal = hilbert(signal)
152     #else:
153     #    analytic_signal = hilbert(signal,N=2**22)
154     analytic_signal = hilbert(signal)
155     analytic_signal=analytic_signal[0:signal.size]
156     amplitude_envelope = np.abs(analytic_signal)
157     instantaneous_phase = np.unwrap(np.angle(analytic_signal))#,
            2*np.pi)
158     instantaneous_frequency = np.diff(instantaneous_phase) /
            (2.0*np.pi) * fs
159     #return amplitude_envelope,instantaneous_frequency,
            instantaneous_phase
160     return amplitude_envelope,instantaneous_frequency,
            instantaneous_phase
161
162 N = 8191
163 t_min = 0
```

```
164 t_max = N / 125e6
165 t_range = np.linspace(t_min, t_max, N)
166
167
168 sampling = 125e6
169 N_data = 1024
170 n_angle = np.zeros((n_files_sel,513), dtype=float)
171 csv_filter_s1 = np.zeros((n_files_sel,8192))
172 csv_filter_s2 = np.zeros((n_files_sel,8192))
173 k=0
174
175 #perform a bandpass on the data between the desired intervals
176 for i in range(0, n_files_sel):
177         csv_filter_s1[i]=bandpass_data(t_range, channel0[i],3.5
              e6,5.5e6)
178         csv_filter_s2[i]=bandpass_data(t_range, channel1[i],3.5
              e6,5.5e6)
179
180 #perform hilbert transform on the now filtered data
181 AE1, IF1, Ph1 = hilbert_signal_phase(csv_filter_s1, 125e6)
182 AE2, IF2, Ph2 = hilbert_signal_phase(csv_filter_s2, 125e6)
183
184 #Threshold the amplitude against its mean. This value can now be
       used to remove areas of non-interest by multiplying any
      array by TAE1 or 2
185 phase = np.zeros((n_files_sel,8192))
186 TAE1 = np.zeros(AE1)
187 TAE2 = np.zeros(AE2)
188 for i in range(0,n_files_sel):
189         threshold_AE1 = np.mean(AE1[i])
190         threshold_AE2 = np.mean(AE2[i])
191         for j in range(0, 8192):
192                 if np.mean(AE1[i][j])<threshold_AE1:
193                         TAE1[i][j] = 0
194                 else:
195                         TAE1[i][j] = 1
196                 if np.mean(AE2[i][j])<threshold_AE2:
197                         TAE2[i][j] = 0
198                 else:
199                         TAE2[i][j] = 1
200 phase = (Ph1-Ph2)*TAE1
201
202 sys.exit
```

A.3 Hilbert Transform in Real-time

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

# Function generator with live acquisition plot for RedPitaya

import os
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal
from laser import Laser

from scipy import signal
from scipy.signal import hilbert
from scipy.signal import butter, lfilter

from koheron import connect
from oscillo import Oscillo

#Bandpasses the data between fmin and fmax
def bandpass_data(t_in,data_in,fmin,fmax):
        tt=t_in
        del_t=tt[1]-tt[0]
        fs=1/(del_t)
        highf=fmax / (fs / 2)
        lowf=fmin/(fs/2)
        B, A = butter(5, [lowf,highf], btype='bandpass')
        filtered_signal = lfilter(B, A,data_in, axis=0)
        return filtered_signal

#performes a hilbert transform returning amplitude, frequency
   and phase
def hilbert_signal_phase(signal,fs):
    #if signal.size < 2*22:
    #    analytic_signal = hilbert(signal)
    #else:
    #    analytic_signal = hilbert(signal,N=2**22)
    analytic_signal = hilbert(signal)
    analytic_signal=analytic_signal[0:signal.size]
    amplitude_envelope = np.abs(analytic_signal)
    instantaneous_phase = np.unwrap(np.angle(analytic_signal))#,
        2*np.pi)
    #instantaneous_phase = np.angle(analytic_signal)#, 2*np.pi)
    instantaneous_frequency = np.diff(instantaneous_phase) /
       (2.0*np.pi) * fs
    #return amplitude_envelope,instantaneous_frequency,
       instantaneous_phase
```

```
43      return amplitude_envelope,instantaneous_frequency,
            instantaneous_phase
44
45  host = os.getenv('HOST','10.209.1.125')
46  client = connect(host, name='oscillo')
47  driver = Oscillo(client)
48
49  #Starts the laser output at selected current
50  Laser(client).start()
51  Laser(client).set_current(30)
52
53  n = driver.wfm_size # Number of samples
54  fs = 125E6          # Sampling rate (Hz)
55  n_bits = 14         # ADC bits
56
57  #
        ------------------------------------------------------------------------

58  # Waveform generation
59  #
        ------------------------------------------------------------------------


60
61  # Modulation on DAC
62
63  #averaging settings
64  driver.set_average(False)
65  driver.set_num_average_min(0)
66
67  #(channel 1, channel 2) (0=sine, 1=triangle, 2=square)
68  driver.set_waveform_type(1, 1)
69
70  #modulation on each channel
71  driver.set_dac_amplitude(1, 0.8)
72  driver.set_dac_frequency(1, 125e6/8192)
73  driver.set_dac_offset(1, 0.0)
74
75  #Modulation only seems to apply after "pinging" the pitaya.
76  driver.get_modulation_status()
77
78  #
        ------------------------------------------------------------------------

79  # Data acquisition
80  #
        ------------------------------------------------------------------------


81
```

```
82 adc_channel = 0
83
84 #driver.set_average(True)
85
86 fig = plt.figure()
87 ax = fig.add_subplot(111)
88 t = np.arange(n) / fs # Time axis in us
89 y = np.zeros(n)
90 li1, = ax.plot(t[:8191], y[:8191])
91 li2, = ax.plot(t[:8191], y[:8191])
92 ax.set_ylim(-20,20)
93 ax.set_xlim(2.5e-6,15e-6)
94 ax.set_xlabel('Time (us)')
95 ax.set_ylabel('Signal (V)')
96 fig.canvas.draw()
97 plt.show(block=False)
98
99 csv_filter_s1 = np.zeros((8192))
100 csv_filter_s2 = np.zeros((8192))
101
102 while True:
103     try:
104                 #collects data from the Red Pitaya
105                 driver.get_adc()
106
107                 #Filters the data between the given frequencies
108                 csv_filter_s1=bandpass_data(t, driver.adc[0,:] /
                        2**n_bits,3.5e6,5.5e6)
109                 csv_filter_s2=bandpass_data(t, driver.adc[1,:] /
                        2**n_bits,3.5e6,5.5e6)
110
111                 #Performs a hilbert tranform on the filtered
                        data
112                 AE1, IF1, PH1 = hilbert_signal_phase(
                        csv_filter_s1, 125e6)
113                 AE2, IF2, PH2 = hilbert_signal_phase(
                        csv_filter_s2, 125e6)
114
115                 #Thresholds the data using the envolope given by
                        the hilbert
116                 TAE1 = AE1
117                 TAE2 = AE2
118
119                 threshold_AE1 = np.mean(AE1[i])
120                 threshold_AE2 = np.mean(AE2[i])
121                 for j in range(0, datapoints):
122                         if np.mean(AE1[j])<threshold_AE1:
123                                 TAE1[j] = 0
```

```
124                     else:
125                             TAE1[j] = 1
126                     if np.mean(AE2[j])<threshold_AE2:
127                             TAE2[j] = 0
128                     else:
129                             TAE2[j] = 1
130             #note frequency has only 8191 points, where as
                    all others have 8192.
131             #Uncomment the appropriate plot for the purpose
                    of the data.
132
133             #shows the phase of each channel
134             #li1.set_ydata(PH1[:8191])
135             #li2.set_ydata(PH2[:8191])
136
137             #shows the envelope of each channel
138             #li1.set_ydata(AE1[:8191])
139             #li2.set_ydata(AE2[:8191])
140
141             #shows the frequency of each channel
142             #li1.set_ydata(IF1[:8191])
143             #li2.set_ydata(IF2[:8191])
144
145             #shows the thresholded phase of each channel
146             #li1.set_ydata(PH1[:8191]*TAE1[:8191])
147             #li2.set_ydata(PH2[:8191]*TAE2[:8191])
148
149             #shows the thresholded envelope of each channel
150             #li1.set_ydata(AE1[:8191]*TAE1[:8191])
151             #li2.set_ydata(AE2[:8191]*TAE2[:8191])
152
153             #shows the thresholded frequency of each channel
154             #li1.set_ydata(IF1[:8191]*TAE1[:8191])
155             #li2.set_ydata(IF2[:8191]*TAE2[:8191])
156
157             #shows the thresholded difference in phase of
                    each channel
158             #line.remove(li2)
159             #li1.set_ydata(PH1[:8191]*TAE1[:8191] - PH
                    [:8191]*TAE2[:8191])
160
161             #shows the thresholded difference in frequency
                    of each channel
162             #line.remove(li2)
163             #li1.set_ydata(IF1[:8191]*TAE1[:8191] - IF2
                    [:8191]*TAE2[:8191])
164
165             #shows the bandpassed data of each channel
```

```
166                 #li1.set_ydata(csv_filter_s1[:8191])
167                 #li2.set_ydata(csv_filter_s2[:8191])
168 
169                 #shows the raw data of each channel
170                 #li1.set_ydata(driver.adc[0,:][:8191])
171                 #li2.set_ydata(driver.adc[1,:][:8191])
172 
173                 ax.relim()
174                 ax.autoscale_view(True, True, True)
175 
176                 plt.pause(0.001)
177                 fig.canvas.draw()
178 
179     except KeyboardInterrupt:
180         break
```