Lunar Ranging Measurements using 23-cm Radar

Sunny He

January 17, 2017

Advisors: Daniel Marlow and Norman Jarosik

Submitted in partial fulfillment
of the requirements for the degree of
Bachelor of Science in Engineering
Department of Electrical Engineering
Princeton University

I hereby declare that this Independent Work report represents my own work in accordance with University regulations.

/s/ Sunny He
Sunny He

Lunar Ranging Measurements using 23-cm Radar

Sunny He

Abstract

Earth-based astronomical radar has historically used very high transmitted power for high-resolution imaging. However, DSP techniques show opportunities to conduct lower-power operations over the Earth-Moon-Earth path, as shown by the increasing popularity of amateur Earth-Moon-Earth communications. This project presents the design and implementation of RF hardware and digital signal processing software to enable 23-cm wavelength monostatic radar measurements on the TLM-18 dish. Using a Costas-10 code transmitted with 250W of output power, the distance to the moon was determined to be 350000 ± 10000km. Distance measurements agree with accepted values to within 4%, and possible sources of error are discussed. The work presented here demonstrates the viability of using commercially available RF hardware to perform astronomical measurements and bringing radio astronomy within the grasp of smaller institutions and interested individuals.

Acknowledgements

Thank you to the Department of Electrical Engineering and the School of Engineering and Applied Science for their gracious funding in support of this project.

I would like to give a generous thank you to my wonderful advisors, Professor Dan Marlow and Dr. Norm Jarosik, for your invaluable advice and tireless support throughout the course of this project. The TLM-18 dish would never have been brought back to life without your dedication and ingenuity, and am truly honored to have an opportunity to work with this magnificent instrument.

Many thanks to Professor David Wentzlaff for taking the time to be my second reader.

Special thanks to Professor Joe Taylor, for introducing me to the opportunity to work with the TLM-18 dish and for offering advice on the signal-processing necessary to put this project in the right direction.

In addition, I would like to thank Vincent Po, for your kind assistance with all sorts of small things that add up to make a huge difference, and David "Radd" Radcliff and Jean Bausmith, for performing the behind-the-scenes wizardry necessary for making ELE independent work a streamlined and enjoyable experience.

Finally, a warm thank you to my parents, for your never-ending love and support in all of my endeavors.

Contents

1 Introduction

This project involves the design and implementation of various software and hardware systems to enable 23-cm wavelength lunar ranging experiments on the TLM-18 dish. This chapter aims to introduce the motivation behind the development of this low-power astronomical radar system and give some background on the TLM-18 dish facility.

1.1 Motivation

The history of active radar measurements of the solar system objects is almost as old as radar itself. In fact, the site at which the TLM-18 dish is located was host to the 1946 research effort Project Diana, which used high-power modified military radars to measure the distance to the Moon[1]. Since then, the Moon has been the subject of much study with active radar, with increasingly larger telescopes using higher power levels to perform higher resolution mapping and study of finer surface details of the Moon. For instance, during the 1960's and 1970's the Arecibo and Haystack Observatories completed detailed radar maps of the Moon using wavelengths between 7.5m and 3.8cm with transmitted powers ranging from 0.2 to 1 MW[2]. While these large installations are capable of incredible sensitivity and resolving power, they bring with them complexity and cost.

While high-power planetary radar capable of kilometer-scale imaging may be out of the question for smaller institutions or individuals, developments in digital signal processing and software defined radio have made exploration of the Earth-Moon-Earth (EME) path far more accessible. In particular, the amateur radio community has make significant progress in the development of EME as a medium for communications. With the release of advanced digital protocols such as JT65, the lower signal strengths of small-scale hardware can be compensated for by more robust protocols and software processing [3].

This project aims to apply a similar approach to the question of astronomical radar, by using DSP techniques to increase the overall performance of hardware systems. Coupled with the recent availability of lower-cost software defined radios such as the Ettus USRP line, it is hoped that astronomical radar techniques can be implemented on more modest hardware setups. Demonstrating the viability of constructing smaller-scale radio telescopes would help reduce the traditionally large capital investments associated with radio astronomy and bring this field within the grasp of smaller institutions and interested individuals.

1.2 TLM-18 Dish

The dish used for this project is a 18m diameter parabolic dish located at the Camp Evans Historic District in Wall Township, NJ about 60km East of Princeton University. This dish was constructed in the 1960's as a tracking antenna used by the United States Air Force for missile tracking and satellite telemetry reception [4]. The dish is designed to be capable of tracking in altitude and azimuth at a rate of up to $10°$/sec and had an original half-power beamwidth of $5.2°$ [4].

Figure 1: Exterior view of TLM-18 dish

This dish was transferred to the InfoAge Science Center in 2012, and Professor Dan Marlow and Dr. Norm Jarosik have been heavily involved in the refurbishing of mechanical components and installation of new receiving hardware. A new feedhorn tuned to the 21cm wavelength band was installed at this time, as well as hardware for amateur EME communications.

2 System Design

A number of different hardware components were added to implement the necessary transmit capability. Systems were constructed and tested on campus before being installed at the TLM-18 dish site. This section focuses on the technical details of the receive, transmit, and control circuitry.

2.1 Receive Hardware

The majority of the receive RF hardware was repurposed from previous projects using the dish. The feedhorn provides two RF connections, one for vertical polarization and one for horizontal polarization. For this project, only the vertical polarization was used for transmitting and taking measurements. While it would be possible to use two receivers in parallel to capture both polarizations, the added complexity was judged to be out of scope for this project and proved to be unnecessary.

A series of Low-Noise Amplifiers and filters condition the incoming signal before being fed into the receivers. The YIG filter is a tunable band-pass filter and was set to pass the 1.296GHz band of interest. A block diagram of the receive hardware is shown in Figure 2.

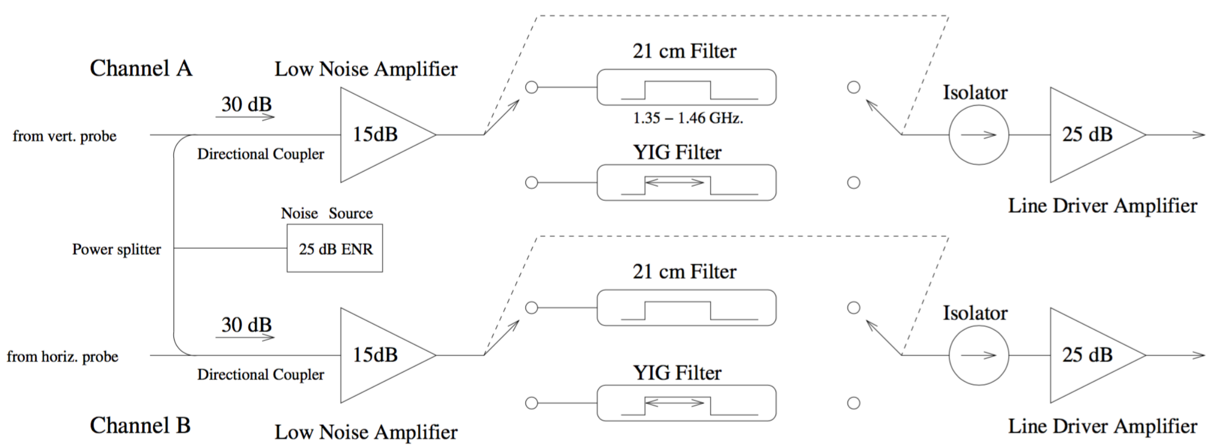

Figure 2: RF receive path block diagram (figure reproduced from [5]).

A Ettus USRP B210 Software Defined Radio was used as the transmitter and receiver. The USRP is controlled by a computer over USB3 in full duplex mode, concurrently sending received samples and sample values to transmit.

2.2 Transmit Hardware

While the USRP B210 is capable of transmitting at the required frequency, its output power is only rated to a maximum of 10dBm. The existing Kuhne MKU PA 23CM-250W

CU Power Amplifier had a required drive level of 4 to 6W with a maximum of 10W, or 16.0 to 37.8dBm with a maximum of 40dBm. Thus, a pre-amplifier would be necessary to increase the USRP's transmitted signal to a level sufficient to drive the power amplifier without overloading the amplifier. To determine the needed gain, the USRP's transmit power at 1.296GHz was measured with a laboratory RF power meter. The software used to control the USRP used arbitrary value from 0 to 90 to represent the power level, so this was also a useful opportunity to relate the software drive values to actual power measurements.

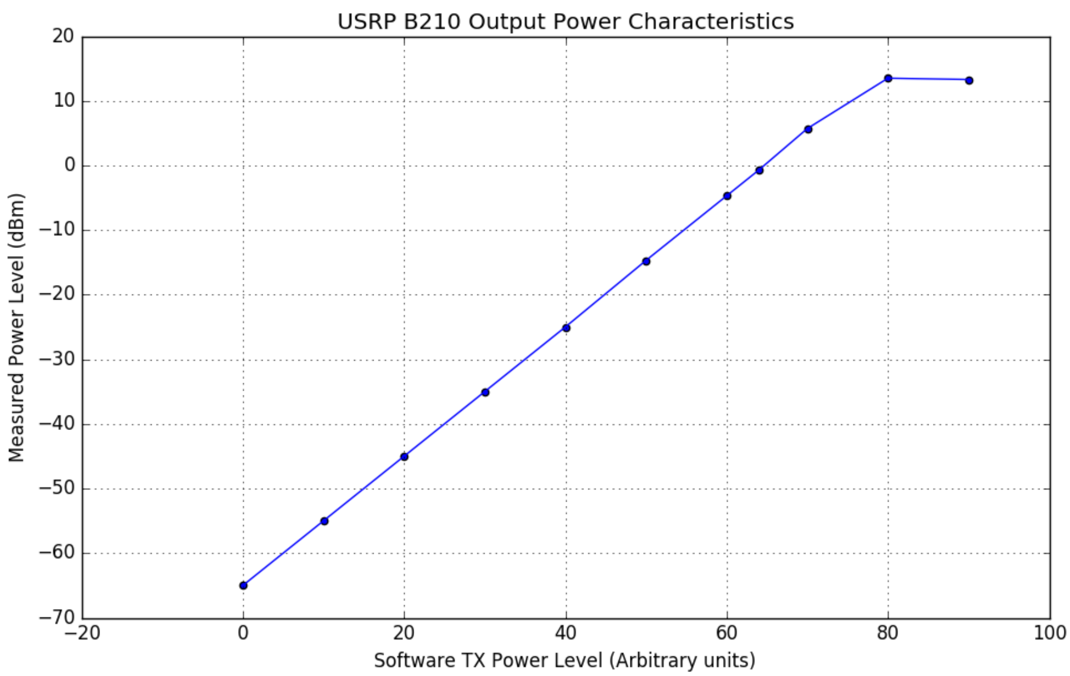

Figure 3: USRP transmit power characteristics.

The USRP's maximum output power peaked at 13.5dBm, slightly above its rated value. With this knowledge, a Kuhne MKU PA 23CM-30W HY amplifier was selected to serve as a pre-amplifier. With a rated gain of at least 24dB, this amplifier was more than sufficient to drive the output amplifier. The unit received was measured to have a gain of about 35.3dB, so a series of attenuators were added between the USRP output and amplifier input to avoid overloading the power amplifier input. In addition, a simple interface circuit was built to allow the USRP to toggle the 12V amplifier enable lines with its on board GPIO pins.

A block diagram of the transmit hardware is shown in Figure 4.

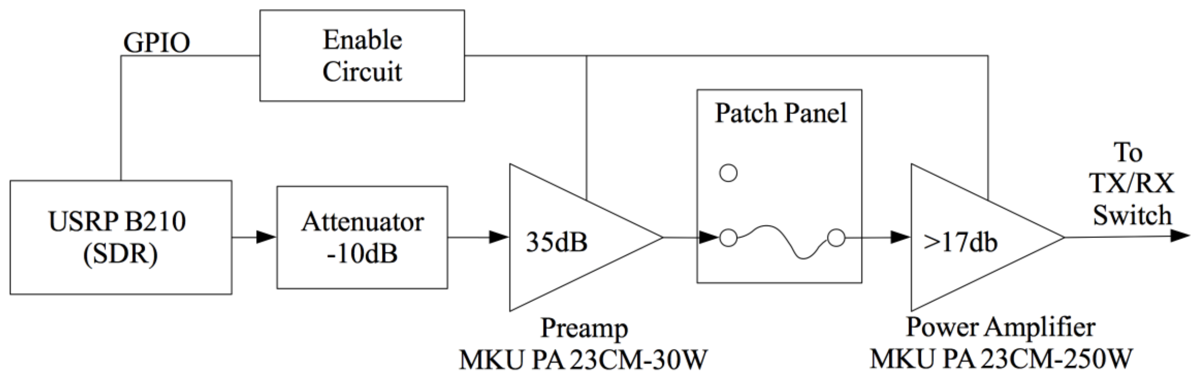

Figure 4: RF transmit path block diagram.

This configuration gives a final transmit power of 250W at 1.296GHz at the amplifier output.

2.3 Installation

All hardware components were mounted on metal panels for installation at the TLM-18 site. RF signals were broken out to connectors so that the system could be easily reconfigured with patch cables with minimal disruption to the existing setup.

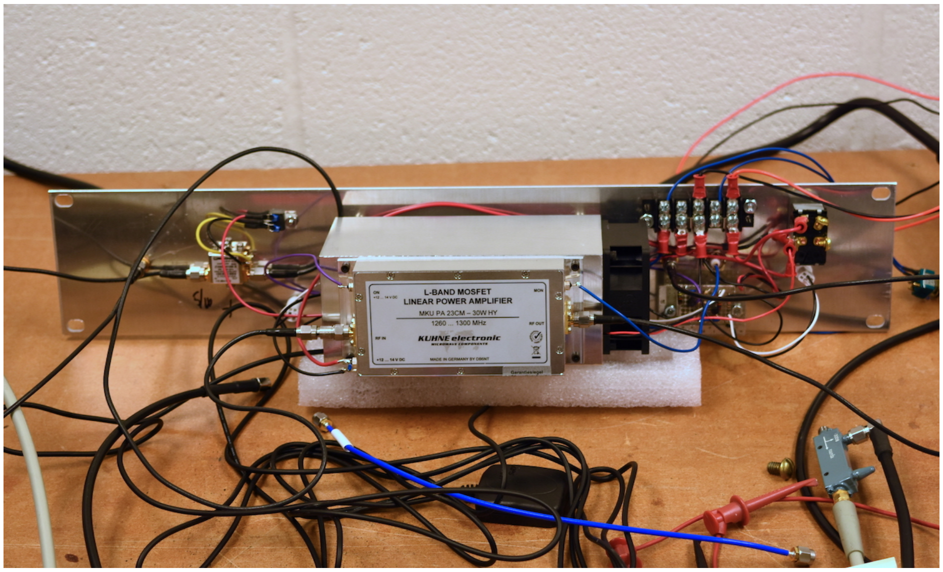

Figure 5: Transmit hardware mounted on panel.

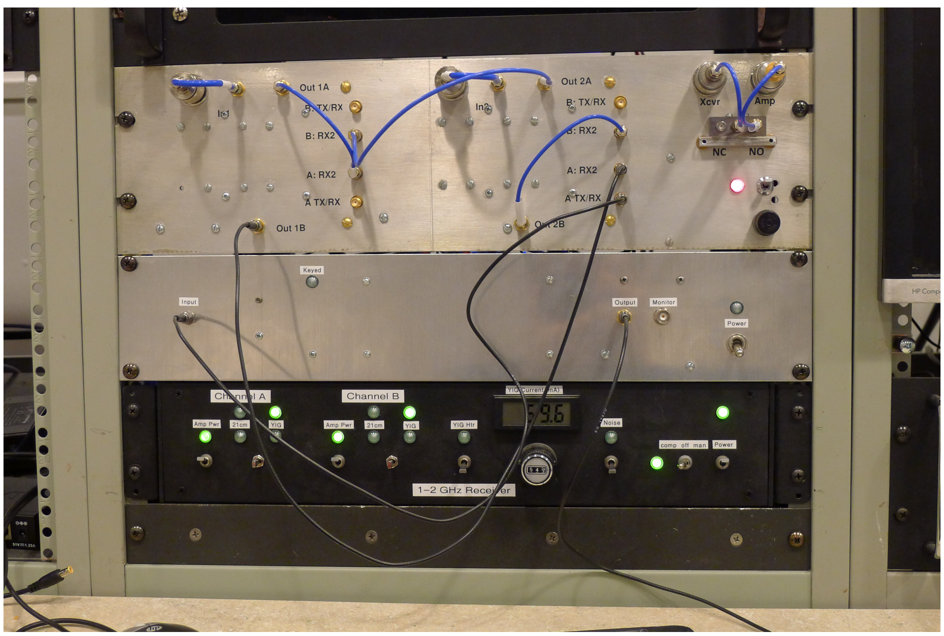

Figure 6: Panels installed in TLM-18 operator console.

3 Costas Codes and Waveform Generation

Due to the large distance and high relative velocity between the receiver on the Earth and the Moon, the received echo would have large time shift due to travel time and frequency Doppler shift. A class of multiple-frequency-shift-keyed codes called Costas codes were chosen for their low correlation ambiguity in both the time and frequency axes.

3.1 Costas Code Properties

Costas codes are composed of a sequence of frequency-hopping pulses of equal length and spaced equally in the frequency domain[6]. Thus they can also be visualized as a sequence of symbols sent via multiple-frequency-shift-keying. For certain sequences of frequency changes, Costas demonstrated that the waveform will exhibit very low ambiguity, representing a low chance of false positives when the received echo is correlated with the transmitted waveform. Furthermore, these Costas codes are very sensitive to shifts in frequency, providing an excellent way to detect the Doppler shift of a received echo.

For this project, a Costas code of length 10 was used, with the 10 pulses sent in 0.1 sec. The specific frequency pattern was 2,4,8,5,10,9,7,3,6,1 [6]. Thus, the first pulse of length 0.01 sec has frequency $2 \times \frac{1}{0.01sec} = 200$ Hz, the second pulse has frequency $4 \times \frac{1}{0.01sec} = 400$ Hz, and so on. The complex waveform was generated using Numpy. The spectrogram of the full code waveform is shown in Figure 7.

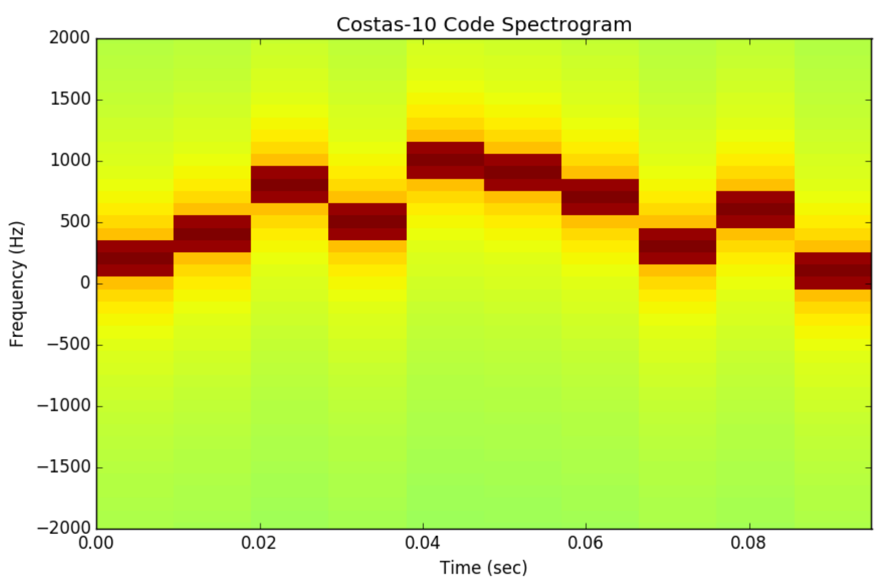

Figure 7: Costas-10 code waveform spectrogram.

3.2 Simulation

The range-doppler correlation properties of the Costas-10 waveform were verified using Numpy simulations. Cross-correlation $f \star g$ of time domain signals $f(t)$ and $g(t)$ is implemented as convolution with a conjugated, time-reversed copy of the Costas-10 waveform. In mathematical notation,

$$f(t) \star g(t) = f^*(-t) * g(t) \tag{1}$$

A computationally efficient method of calculating the cross-correlation can be derived by taking advantage of the convolution theorem. Taking the Fourier transform of Equation 1,

$$F\{f \star g\} = F\{f^*(-t) * g(t)\} \tag{2}$$

$$F\{f \star g\} = F\{f(t)\}^* \cdot F\{g(t)\}\} \tag{3}$$

$$f \star g = F^{-1}\{F\{f(t)\}^* \cdot F\{g(t)\}\} \tag{4}$$

The time domain cross-correlation of the Costas-10 waveform with itself, reveals a sharp peak at a shift of 0 with quickly receding sidelobes at larger shift amounts. The autocorrelation shows a respectable 7dB amplitude separation from the main peak to the largest sidelobe, setting an upper estimate for the SNR increase this code can provide.

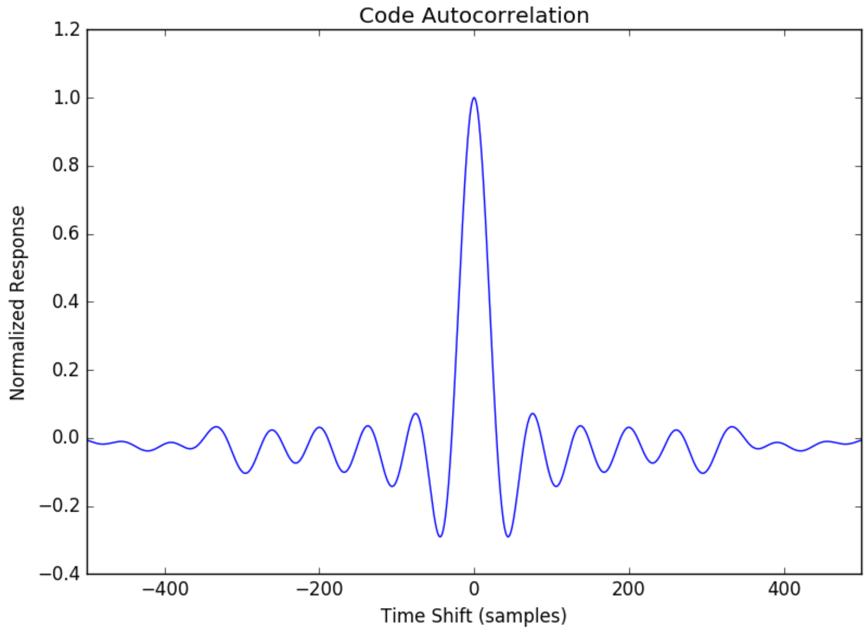

Figure 8: Costas-10 code time-domain autocorrelation.

4 Data Acquisition and Post-Processing

4.1 GNURadio Flowgraph

Control of the USRP software defined radios was accomplished using the GNURadio open-source software package. GNURadio provides C++ and Python libraries for constructing DSP algorithms by chaining together various modules. A Python script was written to configure the USRP SDR's, read the Costas-10 waveform from a file and pass the samples to the USRP transmitter, and record received complex I/Q samples to a separate file. The first version of the data recording script was written with a graphical interface allowing the user to manually activate and deactivate the transmitter and GPIO pins controlling the transmit/receive enable circuitry. An updated version uses threading to automatically activate and deactivate the transmitter after set amounts of time. The GNURadio flowgraph was configured to receive and transmit samples at a 1MHz sampling rate, referenced from the GPS-locked clock on the USRP software defined radios. The recorded complex samples were then analyzed using SciPy.

In total, ten usable trials were recorded on the TLM-18 dish on December 10, 2016 between 21:39 and 22:20 UTC. In each trial, the transmitter was enabled for a period of about one second, and the waveform was repeated multiple times within that transmission period. A half second delay was inserted before the start of transmission to give time for the amplifiers to activate. A sample spectrogram for one recording run is shown in Figure 9. The crosstalk from the transmission and echo from the moon are both clearly visible.

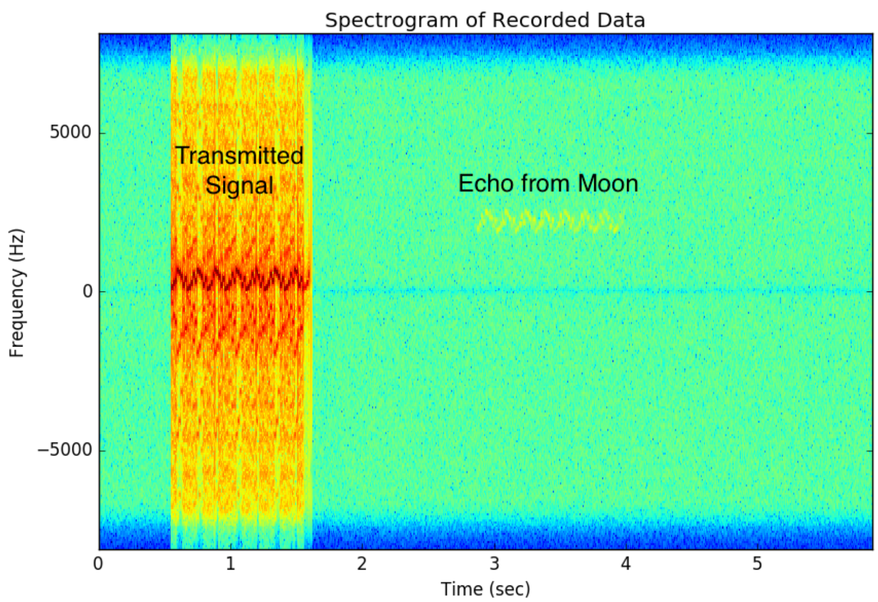

Figure 9: Example spectrogram of scaled recorded data.

4.2 Cross-correlation

A Python script was written to perform cross-correlation on the received data using SciPy. The process of time-domain correlation was similar to that used in the simulation of the Costas-10 waveform. However, to account for Doppler shift, the cross-correlation was repeated with frequency-shifted copies of the original waveform. This produced a 2D correlation matrix, with time on one axis and frequency on the other. By locating maxima in this Range-Doppler matrix, the time and frequency of instances of the Costas-10 code could be easily calculated.

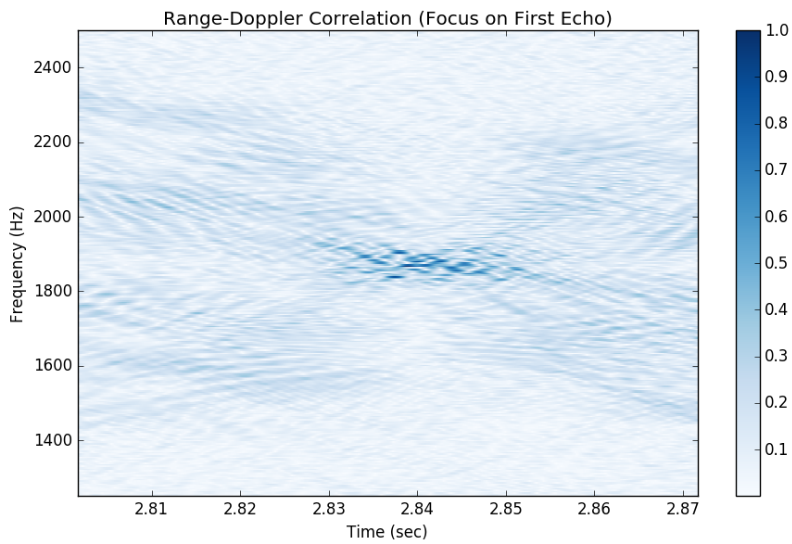

Figure 10: Example range-doppler plot near beginning of echo.

The crosstalk from the outgoing transmission provides a useful timestamp for measuring the time the signal took to reach the moon and return. The difference in time between the start of transmission of the Costas-10 waveform and the start of the echo ΔT represents the round trip time, which can then be converted to a distance using the speed of light C.

$$d = \frac{\Delta T}{2C} \tag{5}$$

4.3 Sample Rate Recovery

One issue discovered during the data analysis was that there appeared to be serious discrepancies in the time scaling of the recorded data. Even visual inspection of the recorded data revealed that the round-trip time measured from the start of transmission to the arrival of the echo was about half the expected value. In addition, while the software was configured to start transmitting 0.5 sec after the start of the script, the recorded data would show the crosstalk start much earlier, around 0.3 sec.

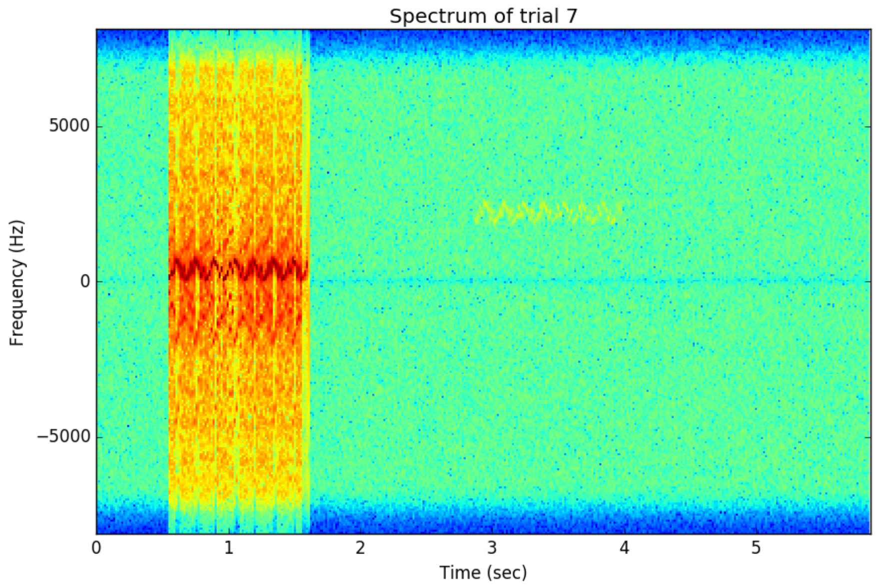

Figure 11: Example spectrogram of raw data, without sample rate correction.

Upon further examination it was found that running the graphical interface while performing recordings was causing excessive load on the computer which was interfacing with the USRP. This in turn was causing indications of dropped samples in the recorded data, as the computer failed to keep up with the steady stream of samples the USRP receiver was pushing out at 1MHz. As a result, the effective recorded rate was less than the defined 1MHz, but by an unknown amount.

To recover the timing of the data, the assumption was made that samples were being dropped at a fairly consistent rate, such that the sample loss could be modeled as a linear scaling of the sample rate. 5 test trials taken without the graphical interface and showing no packet loss events demonstrated that the start of the crosstalk arrived consistently 0.523 ± 0.0001sec after the start of recording. By counting the number of samples of delay in the data sets with dropped samples, this metric could be used to calculate an "actual" sample rate for each data set. The results of this process are presented in Table 1.

Trial Number	"Actual" Sample Rate (Hz)	Round Trip Time (sec)	Earth-Moon Distance (km)
1	656719	2.307	345839
2	656643	2.308	345921
3	662131	2.307	345814
4	642116	2.388	357998
5	643052	2.364	354283
6	650329	2.345	351518
7	649577	2.343	351194
8	660327	2.307	345768
9	641099	2.380	356727
10	661004	2.253	337681
Average	652300	2.330	349274
2σ	16346	0.083	12422

Table 1: Round Trip Time and Distance Results

5 Conclusion

The TLM-18 dish was successfully upgraded with the hardware systems necessary to perform ranging experiments at 23-cm wavelength. The round trip time for a signal to traverse the Earth-Moon-Earth path was measured to be 2.33 ± 0.08sec, corresponding to a Earth-Moon distance of 350000 ± 10000km.

However, there remains a systematic error in these measurements. Calculations based on accepted lunar orbit data place the expected distance to the moon between 364043 and 364006 km during the observation period [7]. As noted earlier, this is likely a symptom of dropped samples during recording, which would artificially decrease the number of samples recorded between the transmission and the echo, thereby causing the observed round-trip-time to be shorter than expected. While a second data collection session could not be conducted during the course of this project due to unexpected mechanical issues in the TLM-18's tracking systems, the systems and software remains in place for when these issues may be resolved.

Other possible directions for further development include further enhancement of the TLM-18 systems or adaptation of the software to more generalized setups. The current

TLM-18 setup requires physical operator presence to monitor the operation and switching of the various mechanical and electrical systems. While data collection can be performed remotely by logging into the control computer over the network, it would be helpful to refine and test the integration and actuation of other components such that physical presence is no longer mandatory, thereby increasing the availability of the TLM-18 dish. Furthermore, the software developed for this project could potentially be used for ranging experiments with other radios. Adaptation of the Numpy analysis scripts to generate and accept real-valued audio samples would allow ranging experiments on traditional voice transceivers, which are far more common than the USRP software-defined-radios used in this project.

Overall, this project demonstrates the viability of performing Earth-based active radar astronomy with relatively low-cost experimental setups. It is hoped that the methods used and problems encountered will be instructive for others seeking to begin their own foray into the field of radar astronomy.

Appendices

A Python Source Code

A.1 correlate.py

Used for performing autocorrelation simulations on Costas codes and tests of correlation routines.

```
from scipy import signal
import numpy as np
import matplotlib.pyplot as plt

costas = np.fromfile(open("costas10_64k.dat"), dtype=np.complex64)
```

```
#data = np.fromfile(open("out.dat"), dtype=np.complex64)
data = np.pad(costas, (500,500), 'constant', constant_values = 0)



# Autocorrelation
plt.figure(1);
plt.title("Code Autocorrelation");

timedomain = np.abs(np.correlate(costas,np.pad(costas, (500,500), 'constant',
    constant_values = 0)))
timedomain /= timedomain.max();

plt.plot(np.arange(-500,501,1), timedomain);
plt.xlabel('Time Shift (samples)');
plt.ylabel('Normalized Response');
plt.xlim([-500,500]);
#plt.plot(np.abs(signal.fftconvolve(costas,costas[::-1],mode="full")));

print 10 * np.log10(timedomain[signal.argrelmax(timedomain)])

# FFT correlation
fft_costas = np.fft.fft(costas, n = 2048);
fft_data = np.fft.fft(data, n = 2048);
fft_corr = np.fft.ifft(fft_costas.conjugate() * fft_data);
#fft_corr = signal.fftconvolve(costas, data[::-1],mode="full");

fft_corr = np.abs(signal.argrelmax(fft_corr)) / fft_corr.max();
print fft_corr[signal.argrelmax(fft_corr)]

```

```
# Doppler correlation
plt.figure(3)
plt.title("Doppler correlation");
doppler_corr = np.correlate(np.abs(fft_data),np.abs(np.pad(fft_costas,(128,128),'constant', constant_values = 0)))
plt.plot(np.arange(-128,129,1), doppler_corr);
plt.axvline(x=np.argmax(np.abs(doppler_corr)) - 128, ymin=0, ymax=1.0, color="black", linestyle="dashed")
print "Doppler: ", np.argmax(doppler_corr) - 128;

print doppler_corr
doppler_corr = np.abs(signal.argrelmax(doppler_corr));
doppler_corr /= doppler_corr.max();
print doppler_corr[signal.argrelmax(doppler_corr)]

#plt.plot(np.abs(fft_costas));

# Cross Correlation
corr = np.correlate(data, costas);
print "Direct Peak: ", np.argmax(np.abs(corr));
print "FFT Peak: ", np.argmax(np.abs(fft_corr));

plt.figure(2);
plt.subplot(211)
plt.title("Direct cross correlation");
plt.plot(np.abs(corr));
plt.axvline(x=np.argmax(np.abs(corr)), ymin=0, ymax=1.0, color="black", linestyle="dashed")
```

```
    
plt.subplot(212)
plt.title("FFT cross correlation");
plt.plot(np.abs(fft_corr));
plt.axvline(x=np.argmax(np.abs(fft_corr)), ymin=0, ymax=1.0, color="black",
    linestyle="dashed")

plt.show();
```

A.2 generate_costas.py

Generates a Costas-10 waveform at the specified sample rate and saves the complex I/Q samples to a file.

```
import numpy as np
import matplotlib.pyplot as plt
import sys

costas_pattern = np.matrix([2,4,8,5,10,9,7,3,6,1]);

# Set sample rate
if len(sys.argv) > 1:
    sample_rate = int(sys.argv[1]);
else:
    sample_rate = 32000;

# Set output filename (ex. costas10_32k.dat)
if sample_rate > 1000:
    filename = 'costas' + str(costas_pattern.size) + '_' +
```

```
        str(sample_rate/1000)+'k.dat'


pulse_per_sec = 100.0;
delta_f = pulse_per_sec;

if delta_f * np.max(costas_pattern) >= sample_rate / 2:
    print 'WARNING: max freq', delta_f * np.max(costas_pattern), 'violates
        Nyquist rate'

costas = costas_pattern.transpose() * np.ones((1, (sample_rate /
    pulse_per_sec)));
costas = costas.reshape(costas.size);

t = np.arange(0.0, costas.size) / sample_rate;
t = t.reshape((1,t.size));

phase = 2 * np.pi * np.cumsum(costas * delta_f) / sample_rate
FSK = np.exp(1j * phase);

print FSK.size, 'samples'
print 'Saving to:', filename;
FSK.astype(np.complex64).tofile(filename);

PLOT = 0;

if PLOT:
    plt.figure(1);
    plt.plot(t.transpose(), np.real(FSK.transpose()))
```

```
    plt.xlabel('time [s]');

    plt.figure(2);
    FSK_fft = np.fft.fft(FSK, n = 2048).transpose();
    plt.plot(np.fft.fftshift(FSK_fft));

    plt.show();
```

A.3 generate_callsign.py

Generates a CW waveform encoding callsign in morse code for identification purposes.

```
import numpy as np
import scipy.io.wavfile as sciowav
import matplotlib.pyplot as plt
import scipy.signal

# dot = 30ms / WPM

WPM = 30;
sample_rate = 64000;
frequency = 450;
filter_cutoff = 600;

callsign = np.array([1,0,2,0,0,0,
                   2,0,2,0,1,0,0,0,
                   2,0,1,0,1,0,1,0,1,0,0,0,
                   2,0,2,0,1,0,0,0,
                   1,0,2,0,1,0,0,0,0,0,0]);
```

```

# Generate waveforms for space, dot, and dash
sec_per_dot = 1.2 / WPM;
print "Sec/dot: ", sec_per_dot;
print "Samples/dot: ", sec_per_dot * sample_rate;

space = np.zeros(int(sec_per_dot * sample_rate));
print "Space len: ", space.size;
dot = np.exp(frequency * 2j * np.pi * np.arange(0, sec_per_dot, 1.0 /
    sample_rate));
print "Dot len: ", dot.size;
dash = np.exp(frequency * 2j * np.pi * np.arange(0, 3 * sec_per_dot, 1.0 /
    sample_rate));
print "Dash len: ", dot.size;

# Concat symbols to make up complete callsign
out = np.array([]);
for i in range(0, callsign.size):
    if callsign[i] == 0:
        out = np.concatenate((out, space));
    elif callsign[i] == 1:
        out = np.concatenate((out, dot));
    elif callsign[i] == 2:
        out = np.concatenate((out, dash));
    else:
        print "Invalid value: ", callsign[i];

# LPF to avoid discontinuities
b, a = scipy.signal.butter(6, filter_cutoff * 2.0 / sample_rate, btype='low',
```

```
    analog=False)
out_filtered = scipy.signal.lfilter(b, a, out);

#plt.plot(np.abs(out_filtered));
#plt.show()

# Write wav file
sciowav.write("callsign.wav", sample_rate,
    (np.imag(out_filtered)*32767).astype(np.int16));
# Write complex file
out_filtered.astype(np.complex64).tofile("callsign.dat");
```

A.4 USRP_Costas_NoGUI.py

Python script using GNURadio to perform an automated transmission and receive cycle.

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
##################################################
# GNU Radio Python Flow Graph
# Title: USRP_Costas_NoGui
# Generated: Wed Dec 21 16:16:38 2016
#
# Usage: python USRP_Costas_NoGUI.py [tx_file] [rx_file] [samp_rate]
##################################################

from gnuradio import analog
from gnuradio import blocks
from gnuradio import eng_notation
```

```
from gnuradio import gr
from gnuradio import gr, blocks
from gnuradio import uhd
from gnuradio.eng_option import eng_option
from gnuradio.filter import firdes
from optparse import OptionParser
import sys
import threading
import numpy
import time


class top_block(gr.top_block):

    def __init__(self):
        gr.top_block.__init__(self, "Top Block")

        ##################################################
        # Variables
        ##################################################
        self.tx_gain = tx_gain = 40
        self.rx_gain = rx_gain = 40
        self.lo_frequency = lo_frequency = 0
        self.lo_amplitude = lo_amplitude = 1
        self.COSTAS_LEN = COSTAS_LEN = 10

        self.samp_rate = samp_rate = 1000000;
        if len(sys.argv) > 3:
            self.samp_rate = samp_rate = int(sys.argv[3]);
```

```
        print('Sample Rate: ' + str(self.samp_rate));
        print(samp_rate);

        self.TX_FILENAME = TX_FILENAME = 'costas' + str(COSTAS_LEN) + '_' +
            str(samp_rate/1000)+'k.dat'
        if len(sys.argv) > 1:
            self.TX_FILENAME = TX_FILENAME = sys.argv[1];
        print('Transmitting from file: ' + self.TX_FILENAME);

        self.RX_FILENAME = RX_FILENAME = 'rx.dat'
        if len(sys.argv) > 2:
            self.RX_FILENAME = RX_FILENAME = sys.argv[2];
        print('Writing data to file: ' + self.RX_FILENAME);



        ##################################################
        # Blocks
        ##################################################
        print('===== Initializing USRP =====');
        self.uhd_usrp_source_0 = uhd.usrp_source(
         ",".join(("serial=30AD2A4", "")),
         uhd.stream_args(
            cpu_format="fc32",
            channels=range(1),
         ),
        )
        self.uhd_usrp_source_0.set_samp_rate(samp_rate)
        self.uhd_usrp_source_0.set_center_freq(1296e6, 0)
```

```
        self.uhd_usrp_source_0.set_gain(rx_gain, 0)
        self.uhd_usrp_source_0.set_antenna('RX2', 0)
        self.uhd_usrp_sink_0 = uhd.usrp_sink(
         ",".join(("", "")),
         uhd.stream_args(
            cpu_format="fc32",
            channels=range(1),
         ),
        )
        self.uhd_usrp_sink_0.set_samp_rate(samp_rate)
        self.uhd_usrp_sink_0.set_center_freq(1296e6, 0)
        self.uhd_usrp_sink_0.set_gain(tx_gain, 0)
        self.uhd_usrp_sink_0.set_antenna('TX/RX', 0)
        self.blocks_multiply_xx_0 = blocks.multiply_vcc(1)
        self.blocks_file_source_0 = blocks.file_source(gr.sizeof_gr_complex*1,
            self.TX_FILENAME, False)
        self.blocks_file_meta_sink_0 =
            blocks.file_meta_sink(gr.sizeof_gr_complex*1, RX_FILENAME, samp_rate,
            1, blocks.GR_FILE_FLOAT, True, 10000000, "", False)
        self.blocks_file_meta_sink_0.set_unbuffered(False)
        self.blocks_delay_0 = blocks.delay(gr.sizeof_gr_complex*1, samp_rate/2)
        self.LO_sig_source = analog.sig_source_c(samp_rate, analog.GR_COS_WAVE,
            lo_frequency, lo_amplitude, 0)

        print self.uhd_usrp_sink_0.get_samp_rate()
        print "RX: ", self.uhd_usrp_source_0.get_samp_rate()

        # GPIO sequencer, run in separate thread
        def _function_probe_gpio():
```

```
        print('===== GPIO Thread begin =====');
        # ----- Initialize GPIO ----- #
        print('Initalizing GPIO');
        # Set all pins (mask 0xff) to GPIO mode (value 0) on mboard 0
        self.uhd_usrp_sink_0.set_gpio_attr('FP0','CTRL', 0, 0xff, 0);
        # Set all pins to Output direction (value 1)
        self.uhd_usrp_sink_0.set_gpio_attr('FP0','DDR', 0xff, 0xff, 0);
        # Set all pins to logic high
        self.uhd_usrp_sink_0.set_gpio_attr('FP0','OUT', 0xff, 0xff, 0);
        print('Readback: ' + hex(self.uhd_usrp_sink_0.get_gpio_attr('FP0',
            'READBACK', 0)));

        print 'Keying: Set pin 0 to low'
        self.uhd_usrp_sink_0.set_gpio_attr('FP0','OUT', 0xfe, 0xff, 0);
        print('Readback: ' + hex(self.uhd_usrp_sink_0.get_gpio_attr('FP0',
            'READBACK', 0)));

        # Transmit for 2.5 sec
        time.sleep(2.5);

        # Set all pins to logic high
        print 'Unkeying'
        self.uhd_usrp_sink_0.set_gpio_attr('FP0','OUT', 0xff, 0xff, 0);
        print('Readback: ' + hex(self.uhd_usrp_sink_0.get_gpio_attr('FP0',
            'READBACK', 0)));

        # Restore pins to high impedance (value 0)
        self.uhd_usrp_sink_0.set_gpio_attr('FP0','DDR', 0, 0xff, 0);

```

```
            # Receive for 3 sec
            time.sleep(3);
            print('Stopping...');
            self.stop();

        # Setup GPIO sequencer thread
        self.function_probe_gpio_thread =
            threading.Thread(target=_function_probe_gpio)
        self.function_probe_gpio_thread.daemon = True



        self.uhd_usrp_source_0 = uhd.usrp_source(
         ",".join(("serial=30AD2A4", "")),
         uhd.stream_args(
            cpu_format="fc32",
            channels=range(1),
         ),
        )

        ##################################################
        # Connections
        ##################################################
        self.connect((self.LO_sig_source, 0), (self.blocks_multiply_xx_0, 0))
        self.connect((self.blocks_delay_0, 0), (self.blocks_multiply_xx_0, 1))
        self.connect((self.blocks_file_source_0, 0), (self.blocks_delay_0, 0))
        self.connect((self.blocks_multiply_xx_0, 0), (self.uhd_usrp_sink_0, 0))
        self.connect((self.uhd_usrp_source_0, 0), (self.blocks_file_meta_sink_0,
            0))

```

```
    def get_tx_gain(self):
        return self.tx_gain

    def set_tx_gain(self, tx_gain):
        self.tx_gain = tx_gain
        self.uhd_usrp_sink_0.set_gain(self.tx_gain, 0)


    def get_samp_rate(self):
        return self.samp_rate

    def set_samp_rate(self, samp_rate):
        self.samp_rate = samp_rate
        self.uhd_usrp_source_0.set_samp_rate(self.samp_rate)
        self.uhd_usrp_sink_0.set_samp_rate(self.samp_rate)
        self.blocks_file_source_0.open('costas' + str(self.COSTAS_LEN) + '_' +
            str(self.samp_rate/1000)+'k.dat', True)
        self.blocks_delay_0.set_dly(self.samp_rate/2)
        self.LO_sig_source.set_sampling_freq(self.samp_rate)

    def get_rx_gain(self):
        return self.rx_gain

    def set_rx_gain(self, rx_gain):
        self.rx_gain = rx_gain
        self.uhd_usrp_source_0.set_gain(self.rx_gain, 0)


    def get_lo_frequency(self):
```

```
        return self.lo_frequency

    def set_lo_frequency(self, lo_frequency):
        self.lo_frequency = lo_frequency
        self.LO_sig_source.set_frequency(self.lo_frequency)

    def get_lo_amplitude(self):
        return self.lo_amplitude

    def set_lo_amplitude(self, lo_amplitude):
        self.lo_amplitude = lo_amplitude
        self.LO_sig_source.set_amplitude(self.lo_amplitude)

    def get_COSTAS_LEN(self):
        return self.COSTAS_LEN

    def set_COSTAS_LEN(self, COSTAS_LEN):
        self.COSTAS_LEN = COSTAS_LEN
        self.blocks_file_source_0.open('costas' + str(self.COSTAS_LEN) + '_' +
            str(self.samp_rate/1000)+'k.dat', True)


def main(top_block_cls=top_block, options=None):

    tb = top_block_cls()
    print('Starting GNURadio blocks...');
    tb.start()
    # Start GPIO sequencer
    tb.function_probe_gpio_thread.start()
```

```
    #try:
    #    raw_input('Press Enter to quit: ')
    #except EOFError:
    #    pass
    #tb.stop()
    tb.wait()


if __name__ == '__main__':
    main()
```

A.5 range_doppler.py

Script for performing range-doppler correlation on input data and plotting the resulting matrix.

```
from scipy import signal
import scipy.misc
import numpy as np
import matplotlib.pyplot as plt

# python range_doppler.py filename [fs] [decimation] [start_sample]
    [num_samples] [fig_filename]

def range_doppler(filename, fs=64000, decimation=1, start_sample=0,
    num_samples=0, shiftrange=1024, fft_len=65536/2):

    data = np.fromfile(filename, dtype=np.complex64)
    if num_samples > 0:
```

```
        data = data[start_sample:start_sample + num_samples:];
    else:
        data = data[start_sample::];

    #costas = np.fromfile(open('costas10_' + str(fs/1000) + 'k.dat'),
        dtype=np.complex64);
    costas = np.fromfile(open('costas10_1000k.dat'), dtype=np.complex64);

    if decimation > 1:
        data = signal.decimate(data, decimation, zero_phase=True);
        costas = signal.decimate(costas, decimation, zero_phase=True);
        fs /= decimation;
        print ('New sample rate: ' + str(fs));

    # Generate fft of pattern
    fft_costas = np.fft.fft(costas, n = fft_len);

    # Plot spectrum
    #plt.plot(np.fft.fftshift(fft_costas));
    #plt.show();

    # FFT correlation for 0 frequency case
    fft_data = np.fft.fft(data, n = fft_len);
    fft_corr = np.fft.ifft(fft_costas.conjugate() * fft_data);

    # Range-doppler matrix
    range_doppler = np.zeros((2 * shiftrange + 1, fft_len), dtype=np.complex64);

    # Evaluate time-domain correlation for each fft bucket of width f_s/fft_len
```

```
        Hz.
    # Two sided
    for i in range(0, 2 * shiftrange + 1):
        range_doppler[i,:] = np.fft.ifft(np.roll(fft_costas, i -
            shiftrange).conjugate() * fft_data);
    # One sided
    #for i in range(-shiftrange - 1, 0):
    #   range_doppler[i + shiftrange + 1,:] = np.fft.ifft(np.roll(fft_costas,
        i).conjugate() * fft_data);
    return range_doppler, fs;

def plot_range_doppler(range_doppler, fs=64000, decimation=1, shiftrange=1024,
    fft_len=65536/2, filename=None):
    new_fs = fs/decimation;
    print 'New fs: ', new_fs
    if filename == None:
        plt.figure(1);
        #plt.axis([0, fft_len, 0, 2 * shiftrange + 1])

        range_doppler_abs = np.absolute(range_doppler);
        range_doppler_abs /= range_doppler_abs.max();

        plt.imshow(np.absolute(range_doppler_abs),
            cmap='Blues', #RdBu
            vmin= np.min(range_doppler_abs),
            vmax=np.max(range_doppler_abs),
            aspect='auto',
            origin='lower',
            interpolation='nearest',
```

```
        extent=(2.80164222,2.80164222 + fft_len/float(new_fs),-shiftrange*float(new_fs)/fft_len,shiftrange*float(new_fs)
        #extent=(0,fft_len/float(fs),0, 2 * shiftrange + 1))
    plt.xlim(2.80164222, 2.80164222 + 0.07)
    plt.ylim(1250,2500)
    plt.title('Range-Doppler Correlation (Focus on First Echo)')
    plt.xlabel('Time (sec)');
    plt.ylabel('Frequency (Hz)');
    plt.colorbar()




solution = np.unravel_index(range_doppler_abs.argmax(), range_doppler_abs.shape)

print 'Freq: bin', solution[0], (solution[0] - shiftrange) * float(new_fs)/fft_len, 'Hz';
print 'Delay: ', solution[1], 'samples', solution[1]/float(new_fs), 'sec';
print "Maxiumum", range_doppler_abs.max();
print "Average", range_doppler_abs.mean();
#print np.argwhere(np.absolute(range_doppler[solution[0]]) > 12 * np.absolute(range_doppler).mean())
#print np.argwhere(np.absolute(range_doppler)>np.absolute(range_doppler).max()*.99);

rd_thresh = np.copy(range_doppler_abs)
rd_thresh[rd_thresh < 15 * rd_thresh.mean()] = 0;

```

```
    print signal.argrelmax(rd_thresh[solution[0]], order=new_fs/100)

    plt.figure();
    plt.title('Range Measurement for Highest Correlation Doppler Shift')
    plt.ylabel('Correlation value');
    plt.xlabel('Time (sec)');
    plt.plot(float(decimation) * np.arange(0, range_doppler_abs[solution[0]].size) / fs, range_doppler_abs[solution[0]]);
    if filename == None:
        plt.show();
    else:
        plt.savefig(filename);

if __name__ == '__main__':
    import sys
    if len(sys.argv) > 1:
        filename = sys.argv[1];
    else:
        print ('No filename given. Usage: python range_doppler.py filename [fs] [decimation] [start_sample] [num_samples]');

    if len(sys.argv) > 2:
        fs = int(sys.argv[2]);
    else:
        fs = 64000;

    if len(sys.argv) > 3:
        decimation = int(sys.argv[3]);
```

```
    else:
        decimation = 1;

    if len(sys.argv) > 4:
        start_sample = int(sys.argv[4]);
    else:
        start_sample = 0;

    if len(sys.argv) > 5:
        num_samples = int(sys.argv[5]);
    else:
        num_samples = 0;

    if len(sys.argv) > 6:
        fig_filename = sys.argv[6];
    else:
        fig_filename = None;

    if filename.endswith(".npy"):
        rd = np.load(filename);
        plot_range_doppler(rd, fs, decimation, filename=fig_filename);
    else:
        rd, new_fs = range_doppler(filename, fs, decimation, start_sample,
            num_samples);
        np.save('range_doppler.npy', rd);
```

A.6 plot_data.py

Utility script for plotting arbitrary complex data in time and frequency domain.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal

def plot(data, sample_rate=64000.0, decimation=1, filename=None):
    print "Size:", data.size,"=", data.size/sample_rate,"sec";

    if decimation > 1:
        print 'Decimating...'
        data = signal.decimate(data, decimation,zero_phase=True);
        sample_rate /= decimation;
        print ('New sample rate: ' + str(sample_rate))



    print "Maximum:", np.max(data), "@", np.argmax(data);
    print "Average:", np.mean(data);

    plt.figure(1)
    plt.plot(np.real(data));
    plt.xlabel("Time (samples)");
    plt.ylabel("Value");

    plt.figure(2)
    (spectrum, freqs, t, im) = plt.specgram(data, NFFT = 256, noverlap = 128, Fs
        = sample_rate,interpolation='nearest');
    if sample_rate * decimation == 1:
        plt.xlabel("Time (samples)");
```

```
    else:
        plt.xlabel("Time (sec)");
    #plt.title('Spectrum of ' + filename[:-4]);
    plt.title('Costas-10 Code Spectrogram');
    plt.ylabel("Frequency (Hz)");
    plt.xlim([0,data.size/sample_rate]);
    plt.ylim([-sample_rate/2, sample_rate/2]);

    print spectrum.shape;

    if filename != None:
        print('Saving to: ' + filename);
        plt.savefig(filename);
    else:
        plt.show();

if __name__ == '__main__':
    import sys
    print sys.argv;
    if len(sys.argv) > 2:
        fs = float(sys.argv[2]);
    else:
        fs = 64000.0

    if len(sys.argv) > 3:
        decimation = int(sys.argv[3]);
    else:
        decimation = 1
    plot(np.fromfile(open(sys.argv[1]), dtype=np.complex64), fs, decimation);
```

A.7 GPIOControl.py

Utility script for manually configuring and toggling the GPIO pins on the USRP B210.

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
##################################################
# GNU Radio Python Flow Graph
# Title: Top Block
# Generated: Wed Dec 21 00:48:25 2016
##################################################

from gnuradio import blocks
from gnuradio import gr
from gnuradio import uhd
import sys

USAGE_STRING = 'Usage: python GPIOControl.py get|set [pin_number] [0|1|x]'

class top_block(gr.top_block):

    def __init__(self):
        gr.top_block.__init__(self, "Top Block")

        # Create USRP Source
        self.uhd_usrp_source = uhd.usrp_source(
         ",".join(("serial=30AD2A4", "")),
         uhd.stream_args(
```

```
            cpu_format="fc32",
            channels=range(1),
        ),
    )

    if sys.argv[1] == 'get':
        print('Getting current pin values...');
        val = self.uhd_usrp_source.get_gpio_attr('FP0', 'READBACK', 0);
        print('Raw value: ' + hex(val));

    elif sys.argv[1] == 'set':
        pinmask = 1 << int(sys.argv[2]);
        print('Pinmask: ' + hex(pinmask));
        if sys.argv[3] == 'x':
            print('Setting pin ' + sys.argv[2] + ' to high impedance...');
            # Write a logic 1 first
            self.uhd_usrp_source.set_gpio_attr('FP0','OUT', 0xff, pinmask, 0);
            self.uhd_usrp_source.set_gpio_attr('FP0','DDR', 0, pinmask, 0);
            print('Readback: ' + hex(self.uhd_usrp_source.get_gpio_attr('FP0',
                'READBACK', 0)));
        elif sys.argv[3] == '1':
            # Enable GPIO Control
            self.uhd_usrp_source.set_gpio_attr('FP0','CTRL', 0, pinmask, 0);
            # Set pin to Output direction (value 1)
            self.uhd_usrp_source.set_gpio_attr('FP0','DDR', 0xff, pinmask, 0);
            print('Setting pin ' + sys.argv[2] + ' to logic high...');
            self.uhd_usrp_source.set_gpio_attr('FP0','OUT', 0xff, pinmask, 0);
            print('Readback: ' + hex(self.uhd_usrp_source.get_gpio_attr('FP0',
                'READBACK', 0)));
```

```
            elif sys.argv[3] == '0':
                # Enable GPIO control
                self.uhd_usrp_source.set_gpio_attr('FP0','CTRL', 0, pinmask, 0);
                # Set pin to Output direction (value 1)
                self.uhd_usrp_source.set_gpio_attr('FP0','DDR', 0xff, pinmask, 0);
                print('Setting pin ' + sys.argv[2] + ' to logic low...');
                self.uhd_usrp_source.set_gpio_attr('FP0','OUT', 0, pinmask, 0);
                print('Readback: ' + hex(self.uhd_usrp_source.get_gpio_attr('FP0',
                    'READBACK', 0)));
            else:
                print('Invalid pin value: ' + sys.argv[3]);
                print(USAGE_STRING);
        else:
            print('Invalid command: ' + sys.argv[1]);
            print(USAGE_STRING);

        self.stop();



def main(top_block_cls=top_block, options=None):
    tb = top_block_cls()
    tb.start()
    try:
        raw_input('Press Enter to quit: ')
    except EOFError:
        pass
    tb.stop()
    tb.wait()
```

```


if __name__ == '__main__':
    if len(sys.argv) < 2:
        print(USAGE_STRING)
    else:
        main()
```

References

[1] United Engineering Foundation. *Project Diana*. Sept. 2015. URL: http://ethw.org/Project_Diana.

[2] T. W. Thompson. "A Review of Earth-Based Radar Mapping of the Moon". In: *The Moon and the Planets* 20.2 (Sept. 1979), pp. 179–198. DOI: http://dx.doi.org/10.1007/BF00898069.

[3] Joe Taylor. "EME with JT65". In: *QST Magazine* (June 2005).

[4] F. S. Coxe. "Operational Characteristics of the TLM-18 Automatic Tracking Telemetry Antenna". In: *IRE Transactions on Space Electronics and Telemetry* SET-5.2 (June 1959), pp. 87–91. ISSN: 0096-252X. DOI: 10.1109/IRET-SET.1959.5008662.

[5] Norman Jarosik. "Physics 312 Pulsar Laboratory Handout". 2016.

[6] J. P. Costas. "A study of a class of detection waveforms having nearly ideal range-Doppler ambiguity properties". In: *Proceedings of the IEEE* 72.8 (Aug. 1984), pp. 996–1009. ISSN: 0018-9219. DOI: 10.1109/PROC.1984.12967.

[7] Torsten Hoffman. *MoonCalc*. Jan. 2017. URL: http://www.mooncalc.org/#/40.1849,-74.0565,17/2016.12.10/17:20/1.