TWO SENIOR PROJECTS:

2.4 GHz, 40% EFFICIENCY RADIO FREQUENCY AMPLIFIER, IEEE DESIGN CONTEST

&

DESIGN AND IMPLEMENTATION OF A SOFTWARE COSTAS LOOP FOR AUDIO FREQUENCIES

BY

ROBERT J. TONG

SUNDAY, AUGUST 27, 2017

HOW TO READ THIS DOCUMENT

This document combines two senior project reports. The first senior project documents designing a class AB RF amplifier. The second, discusses the design and implementation of a software Costas loop for audio frequencies. The first report begins on the next page, while the Costas loop report starts on page 24. The two reports are orthogonal from one another. It is not a prerequisite to read the RF amplifier report before reading the Costas loop report. This document is merely two reports combined into one document. The second report, about the Costas loop, was written as a replacement to the first. However, it was decided, to include the first senior project, in order to make available information and insights that were encountered during the RF amplifier design and construction process, that someone else might find helpful.

2.4 GHz, 40% EFFICIENCY RADIO FREQUENCY AMPLIFIER, IEEE DESIGN CONTEST

BY

ROBERT TONG

A SENIOR PROJECT

ELECTRICAL ENGINEERING DEPARTMENT

CALIFORNIA POLYTECHNIC STATE UNIVERSITY

SAN LUIS OBISPO

2015

Table of Contents

Acknowledgement

Thank you Dr. Arakaki, for not only allowing me to use your CNC mill, but also for instructing the RF courses that provided me with foundational knowledge in matching networks and ADS that enabled this senior project.

And thanks to Dr. Derickson for letting me borrow his PHA -1+ pre-amplifiers that allowed me to generate input powers not capable with the microwave lab's existing equipment.

Abstract
In this senior project, a 4W 2.4 GHz power amplifier (PA) is designed as an entry for IEEE's high frequency amplifier design contest. The design uses Cree's CGH40010F Gallium Nitride (GaN) transistor, because it is capable of outputting 10W up to 6 GHz. Cree also provides simulation models for the Agilent Design System (ADS) – the RF design software used to design this PA. In order to maximize both efficiency and linearity, the PA is class AB biased. Matching networks are implemented on FR4 substrate. Tuning the gate voltage on the final design minimizes intermodulation distortion (IMD) and improves linearity. The final design exhibits 40% power added efficiency (PAE) and greater than 20 dBc IMD within a 5 MHz bandwidth.
See page 4 – background' for information on PAE and IMD.

I. Introduction

Every year, the International Microwave Symposium holds a high frequency amplifier design contest. This year, the competition is focusing on efficient RF amplifiers, because demand for cellular traffic has highly increased wireless network energy consumption. In the past decade, the telecommunication industry has been responsible for about 2% of global CO_2 emissions, which is predicted to double by 2020 because of the exponential growth of mobile traffic [1]. Base stations consume 85% of the total mobile communications energy budget [1]. In cellular broadcasting, the RF amplifier consumes the most power. Reducing the mobile communication's carbon footprint can be achieved with more efficient RF amplifiers.

II. IEEE Design Contest Specifications (Requirements)

The IEEE student design contest specifies that a competing amplifier must:

- Operate between 1 and 10 GHz.
- When excited by a single carrier frequency of up to ¼ W, output at least 4 W.
- Amplifiers are scored by their PAE (see equation 1, page 4).
    - Two-tone input stimulus may not exceed 22 dBm per tone.
    - Two-tone frequencies must be spaced by 5 MHz.
    - Carrier to intermodulation ratio (C/I) must be greater than 30 dBc.

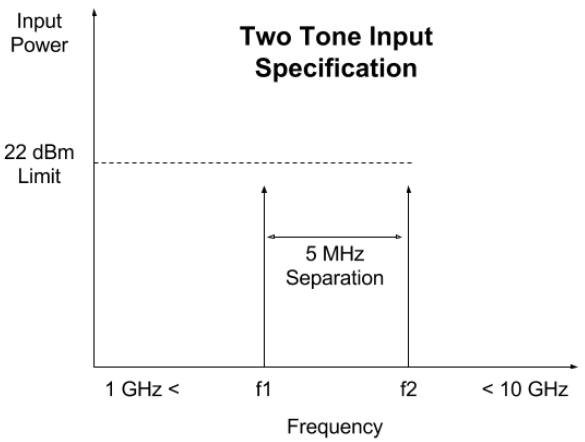


Figure 1. Input Specification

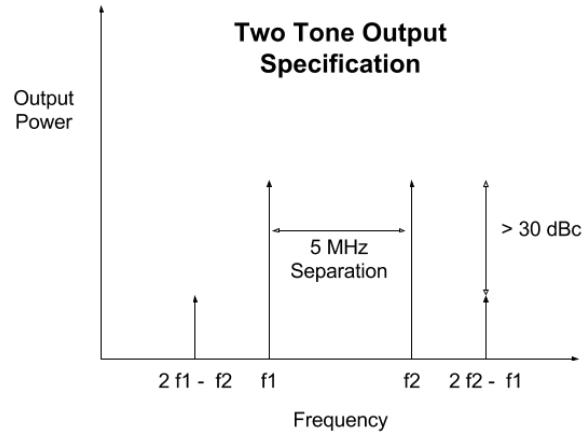


Figure 2. Output Specification

III. Background

- Importance of Linearity

Modern communication systems encode digital information by varying the amplitude and phase of transmitted signals. Therefore, in order to preserve this information, amplifiers must behave linearly.

- Frequency Selection

The project PA was designed for 2.442 GHz, which is WiFi's middle channel.

- Power Added Efficiency (PAE)

Equation 1:

$$PAE = \frac{P_{out} - P_{in}}{P_{DC}} \times 100\%$$

PAE takes into account the power added by the amplifier, therefore is preferred over drain efficiency:

Equation 2:

$$\eta_{Drain\ Efficiency} = \frac{P_{out}}{P_{DC}} \times 100\%$$

- Inter Modulation Distortion (IMD)

IMD is used to gauge an amplifier's non-linearity. If a two-tone input were injected into a perfectly linear amplifier, it will produce an amplified version of those two-tones at the same exact frequencies. A non-linear amplifier would produce additional tones other than the two original signal frequencies called IMD.

- Two-Tone Test

The PA's linearity was tested by inputting two 20 dBm amplitude tones. This is equivalent to testing the PA's response to a suppressed carrier AM signal, which tests the PA's response to small and large input amplitudes. The PA's distortion, or non-linearity, can be observed by looking at its output spectra's IMD. The spurious free dynamic range was not measured in this experiment, because the amplifier is guaranteed to generate spurs when it is driven by two 20 dBm sinusoids.

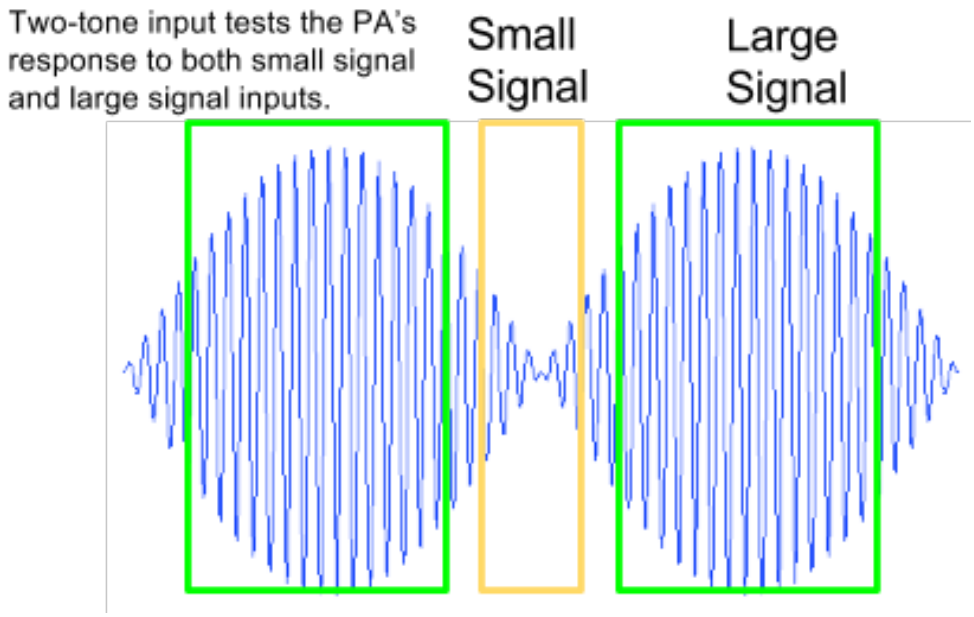


Figure 3. How Suppressed AM Carrier Tests Large and Small Signal Responses

IV. Design Overview of 4W Power Amplifier in ADS

ADS's *Two-Tone PAE* design template aided setting up an ADS schematic to optimize the transistor's IMD and PAE.

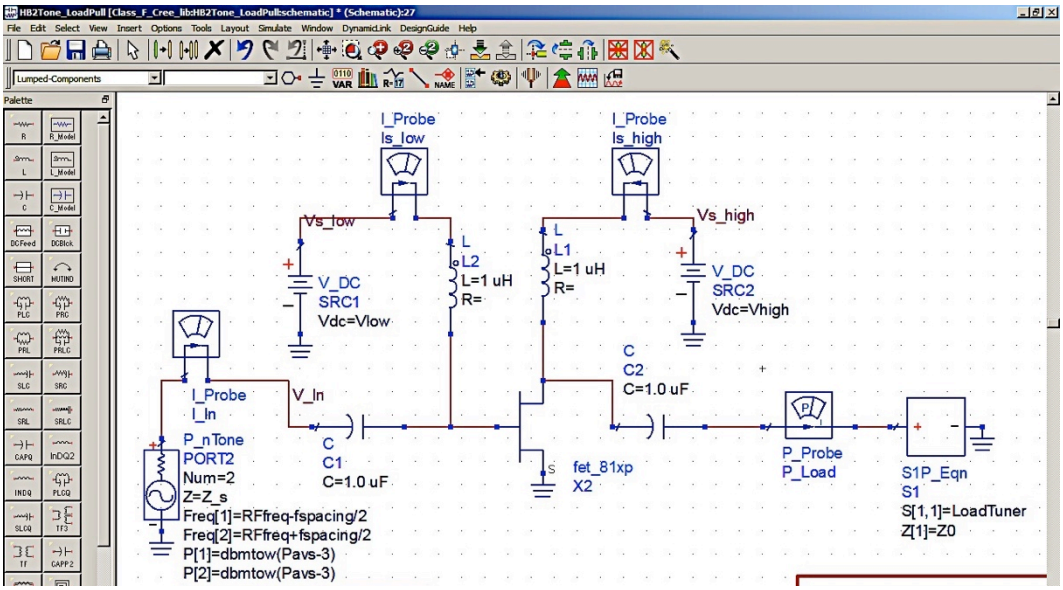


Figure 4. Two-Tone PAE Template Screen Shot

Gate and drain voltages, input power, and matching network dimensions were optimized in order to maximize carrier to IMD ratio and output power. Intermodulation spurs – four in total: $2f_1$-f_2, $2f_2$–f_1, $3f_1$–$2f_2$, and $3f_2$–$2f_1$ were also optimized to be 30 dB below the lesser of the two output powers.

- **Viewing the Simulation Results.**

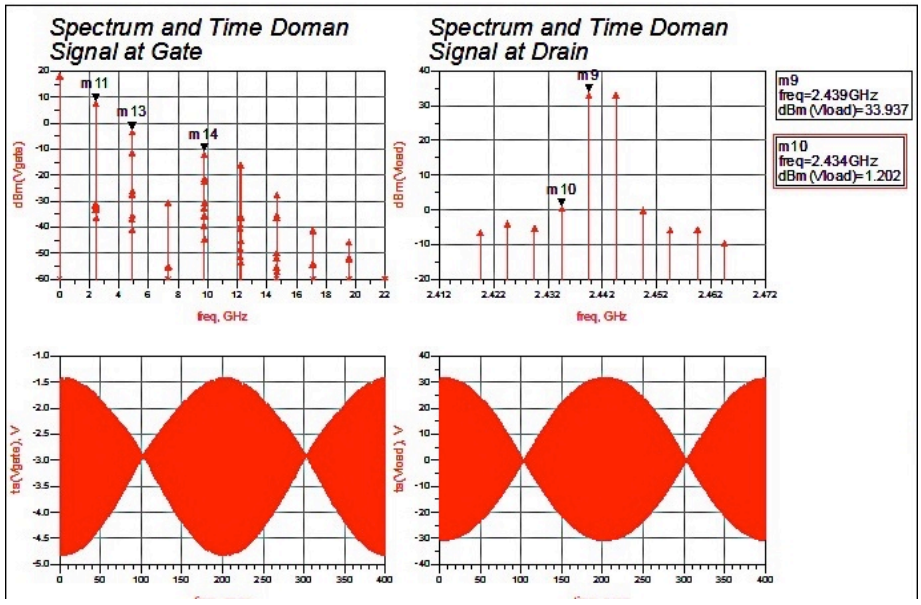


Figure 5. Data Display Output for Two-Tone Simulation

The upper-left plot depicts the transistor's gate voltage spectra from DC to

22 GHz. The upper right graph depicts the load's voltage spectra from 2.412 to 2.472 GHz. Its time domain waveform is the bottom right graph. The lower left graph displays the gate time domain waveform – it exhibits high distortion caused by the gate diode (the transistor is a JFET).

- **Optimization Strategy.** The microstrip geometries were optimized using the Random and Quasi-Newton optimizers. The Random optimizer is able to quickly trial and error a wide range of solutions, while the Quasi-Newton optimizer tunes an existing design in order to enhance its performance. In this way, the Quasi-Newton algorithm was used to refine the best solution produced by the Random optimizer.

  Bias networks. The transistor is biased using quarter wave transmission lines. The quarter wave transmission lines transform the AC short-circuits created by voltage supply bypass capacitors to an open circuit seen by the matching networks.

  Output Matching Network. A double stub network was chosen as the output network, because it has lower Q (Quality) factor than a two-element match, this improves operational bandwidth by making the matching network impedance less sensitive to changes in frequency.

  Input Match. The input matching network consists of an open circuited stub at the transistor's gate and a quarter wave transformer. Using the tuning feature in ADS, the input was tuned to 50 ohms (figure 6). Manual tuning provided better results than the optimizer.

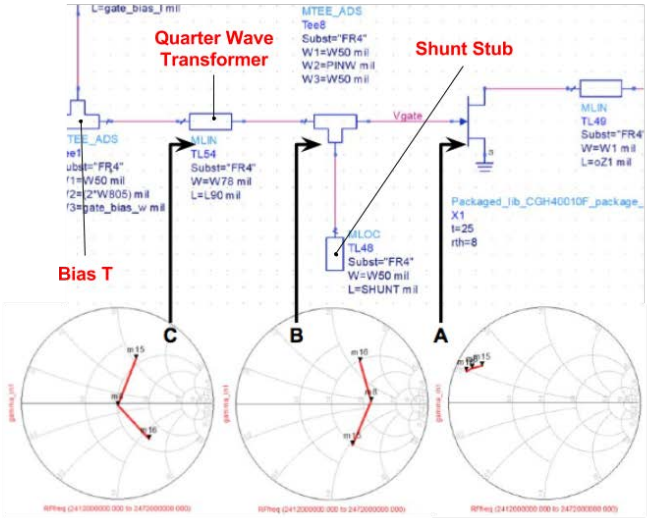


Figure 6. Input Matching

- **Final Design**
  The microstrip networks were fabricated using an LPKF automated CNC (computer numerical control) mill. The design measured 4 ½ by 2 ½ inches. A hole for the transistor was cut into the PCB with the CNC mill, so that the transistor could be mounted through the PCB and onto a metal heat sink.

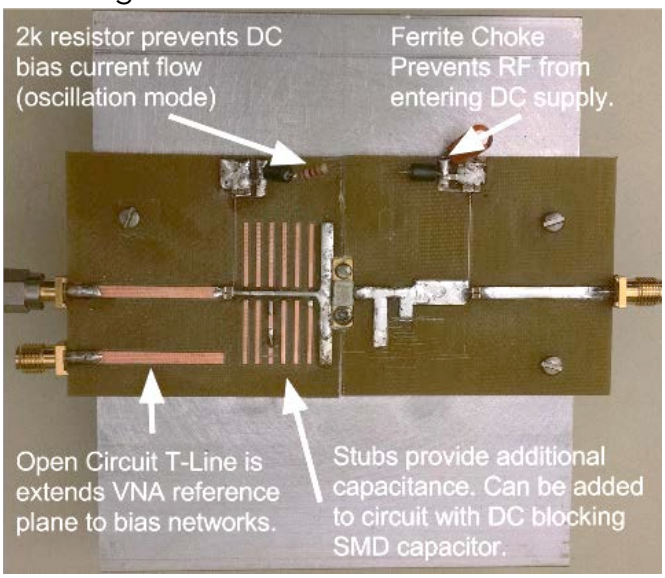


Figure 7. Final Amplifier

- **Output Match S11 Verification** (Seen by transistor)
  Figure 8 plots the measured and simulated output matching network impedances.

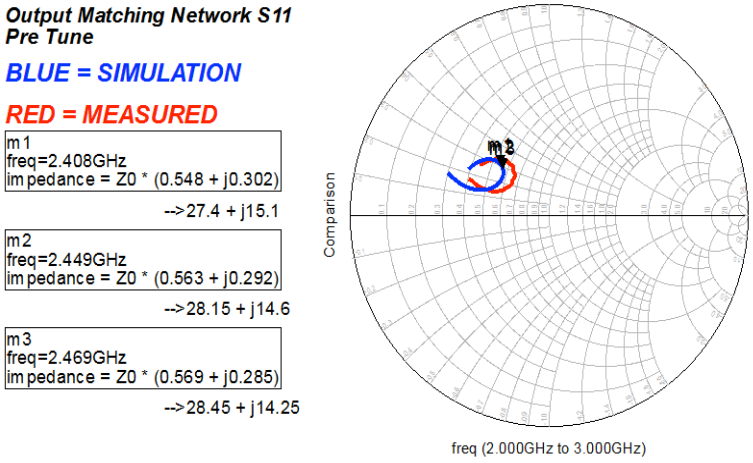


Figure 8. Output Matching Network, Simulation vs. Measured

The FR4 substrate's loss shifted the ideal S11 towards the center of the Smith chart.

- **Output Tuning**
  Tuning the output network by extending stub length reduced resistance, which is illustrated in Figure 9.

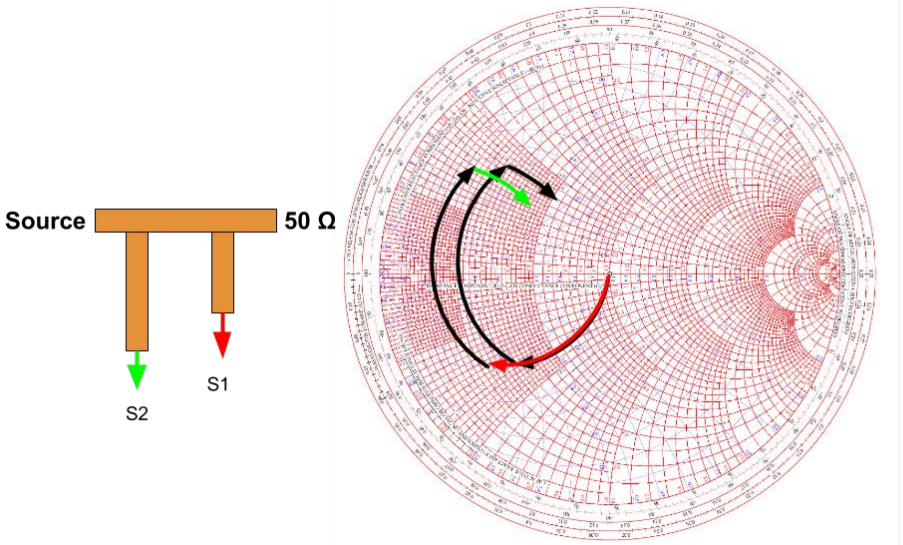


Figure 9. Output Tuning Depiction

Figure 10 reports the impedances after tuning.

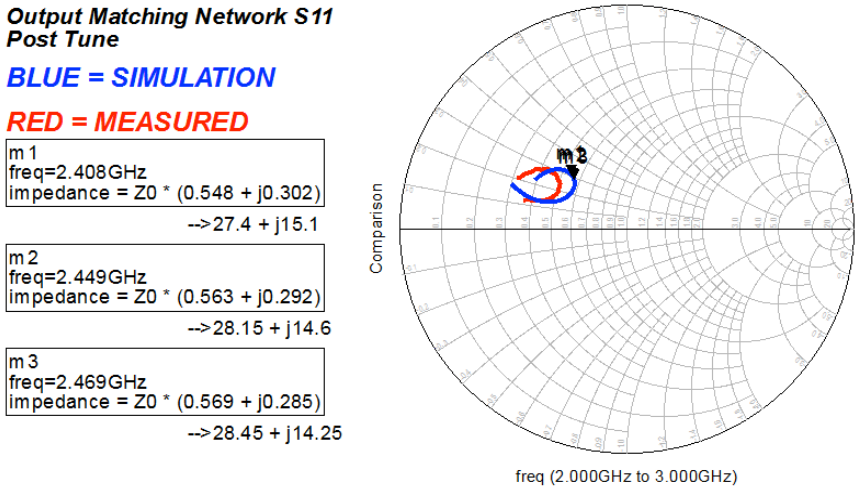


Figure 10. Output Matching Network Post Tune

- **Input Tuning**
  Tuning the input was difficult because of the transistor's non-linearity. The input impedance exhibited by the transistor is dependent on driving power. This is problematic, because the VNA outputs a maximum of 7 dBm. In order to make the tuning environment similar to the PA's final operating conditions, the gate was

biased to sink 300 mA drain current, which is the same amount it sinks when driven by two 20 dBm tones. Altering the bias voltage also changes the transistor's input impedance.

A small flat head screwdriver was used as a capacitive probe and placed at various points along the input quarter wave transmission line in figure 6.

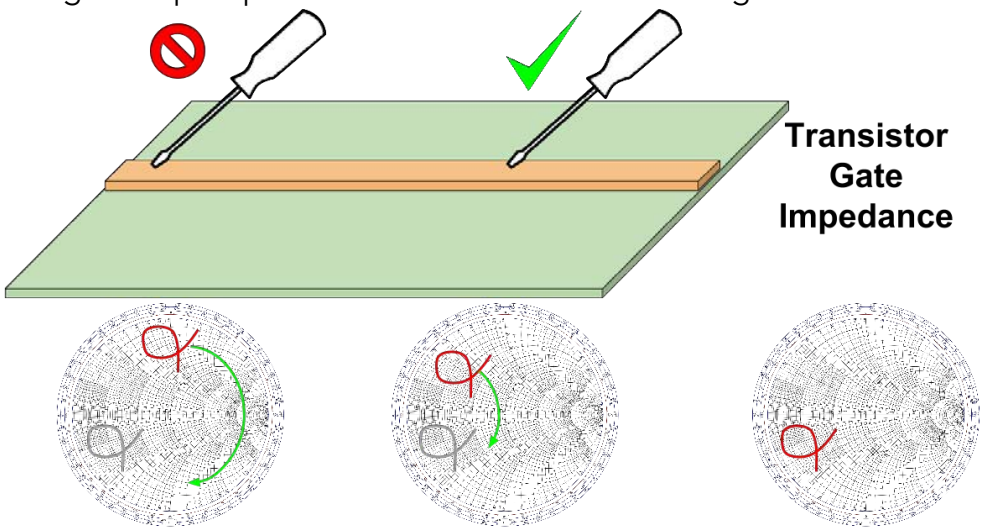


Figure 11. Input Tuning with Screwdriver

A tuning stub was bridged to the quarter wave transmission line where the screwdriver moved the input impedance through the center of the Smith chart. The stub was then trimmed until the input impedance was approximately matched. Tuning the input helped decrease the PA's IMD by approximately 3 dB.

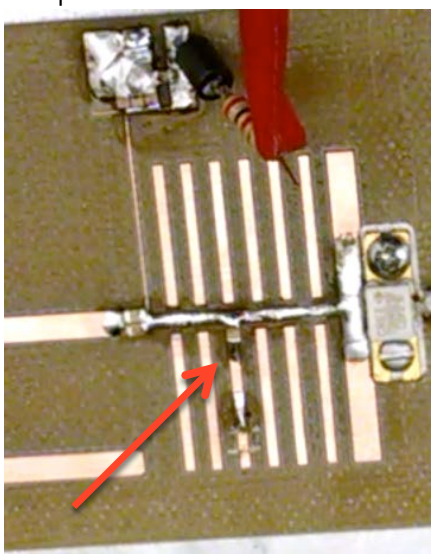


Figure 12. Connecting the Tuning Stub

V. Test Setup

A 3-port VNA, configured to measure mixer conversion loss, acted as the signal generator. The VNA was put into CW mode with the RF and LO frequencies separated by 5 MHz centered around 2.442 GHz. Since the VNA can only output a maximum of 7 dBm, pre-amplifiers were built to amplify the RF and LO ports to a combined power of 23 dBm. Figure 13 depicts the experiment setup.

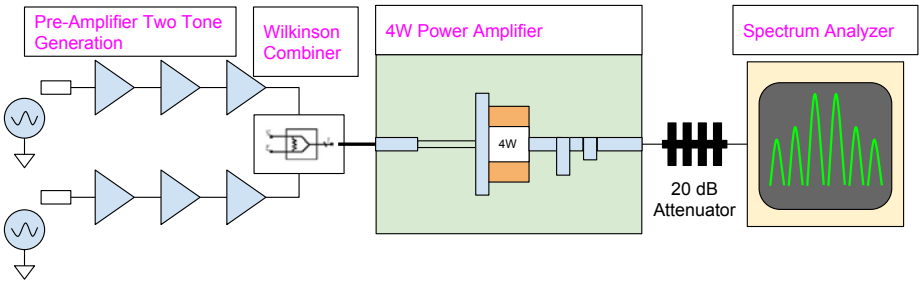


Figure 13. Test Setup

V.A. Pre – Amplifiers

Three Mini Circuits PHA-1+ amplifiers were cascaded to achieve 16 dB of amplification. The amplifiers can be seen in figures 14 and 15.

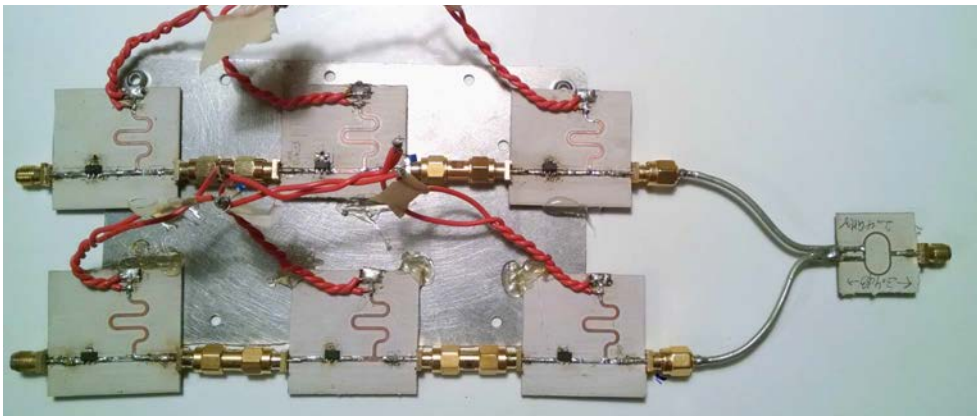


Figure 14. Cascaded Amplifiers

The PHA-1+ requires 5 V_{DC} and operates from 50 MHz to 6 GHz. The amplifiers achieved 10 dB of gain when driven by matched source and load and greater than 15 dB input and output return loss.

The curved quarter wave transmission lines (figure 15) provide DC bias. Curved transmission lines save PCB space.

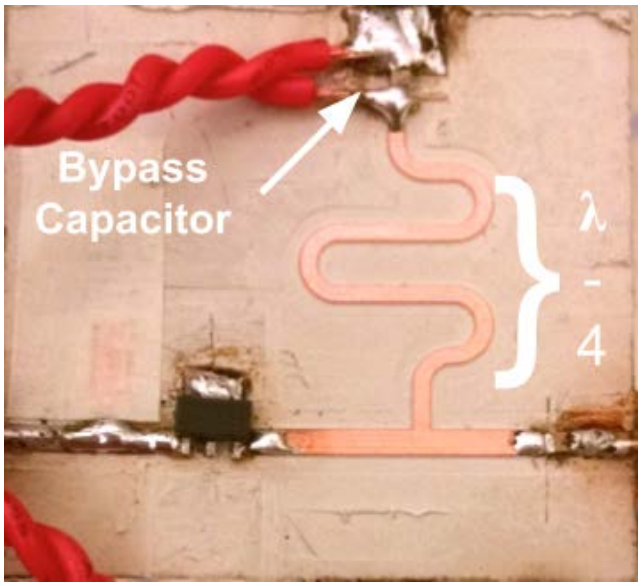


Figure 15. Individual Pre-Amplifier

The quarter wave transmission line transforms the AC short-circuit, created by the SMD supply bypass capacitor, to open circuit at 2.442 GHz. The capacitor is also intended to short-circuit parasitics, introduced by the power supply wires.

The S-parameters of the amplifier shown in figure 15 are plotted in figures 16-18.

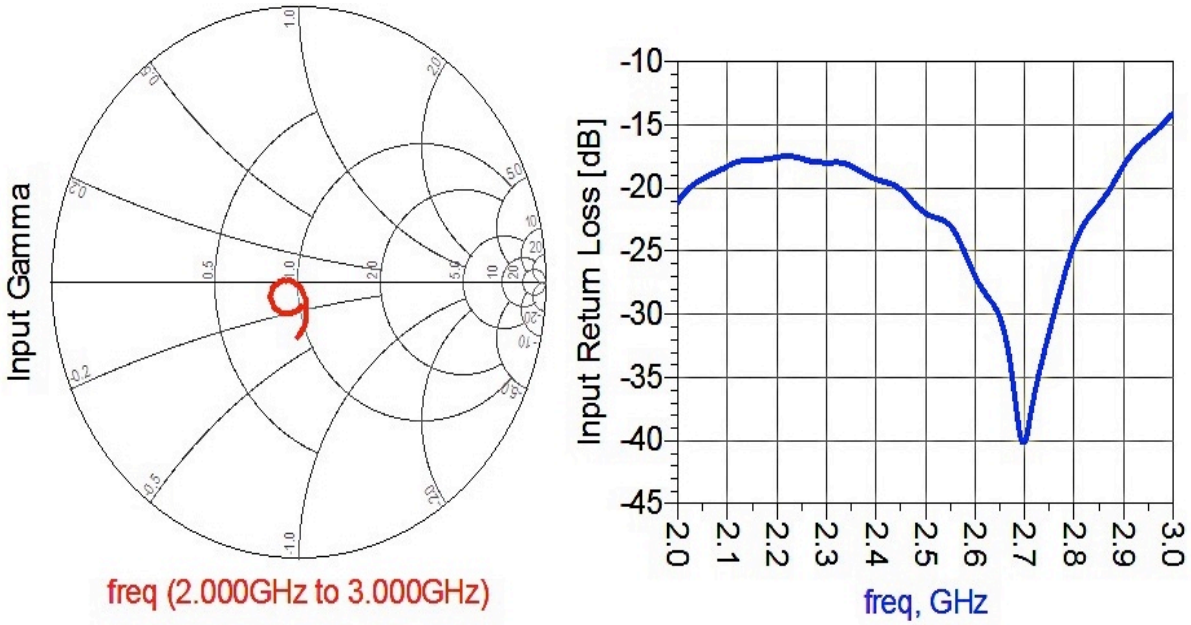


Figure 16. PHA-1+ S_{11}, 5VDC

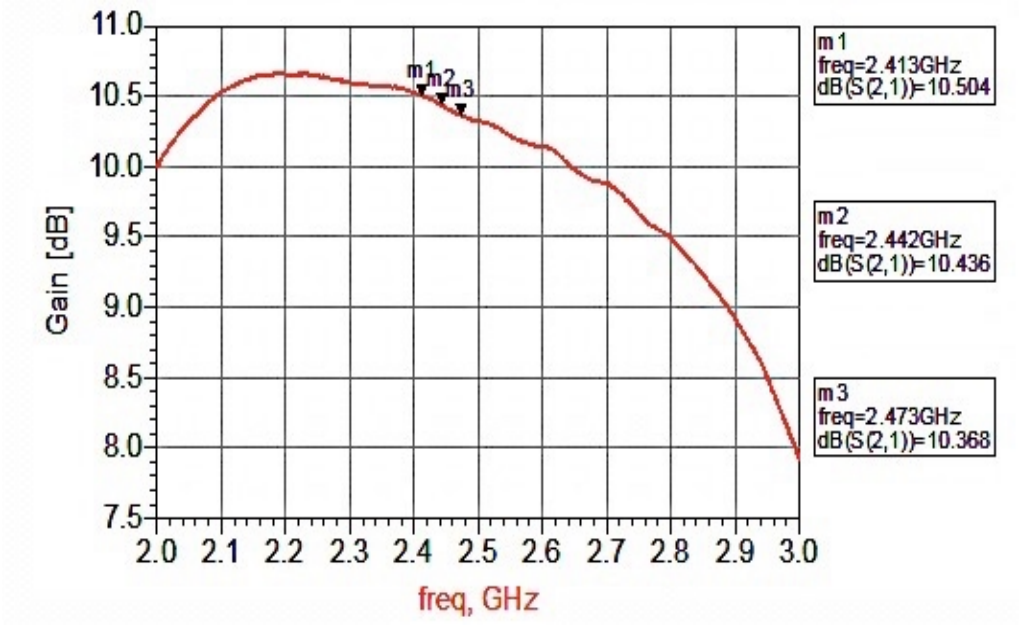


Figure 17. PHA-1+ S_{21} Gain, 5VDC

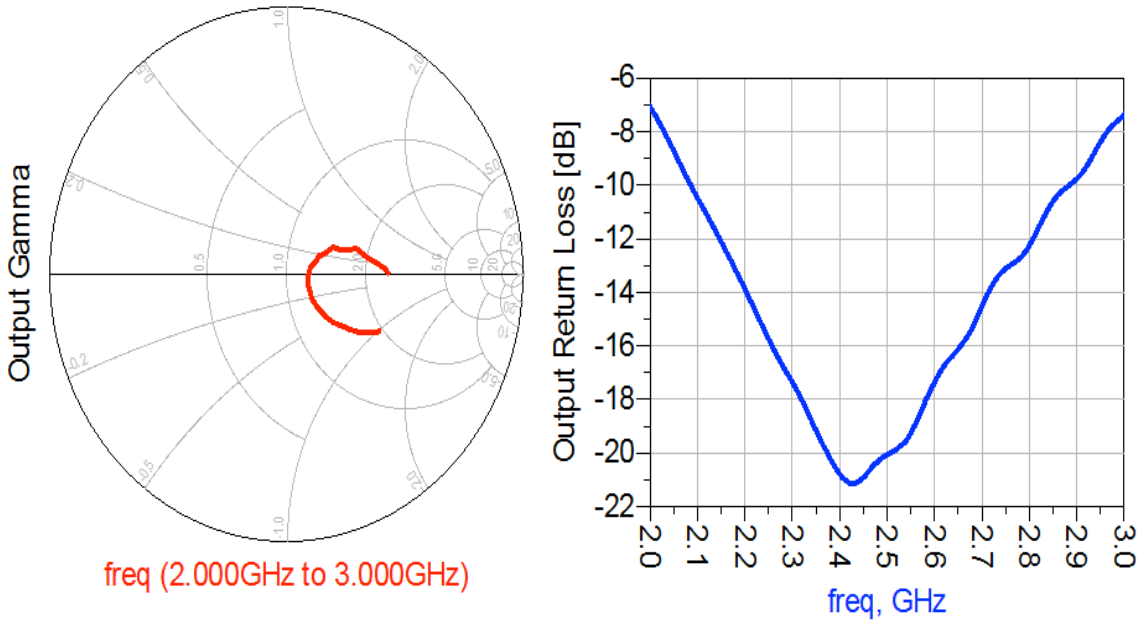


Figure 18. PHA-1+ S_{22}, 5VDC

Section V.B. Wilkinson Combiner

The assembled preamplifier yielded 23 dBm of available power.

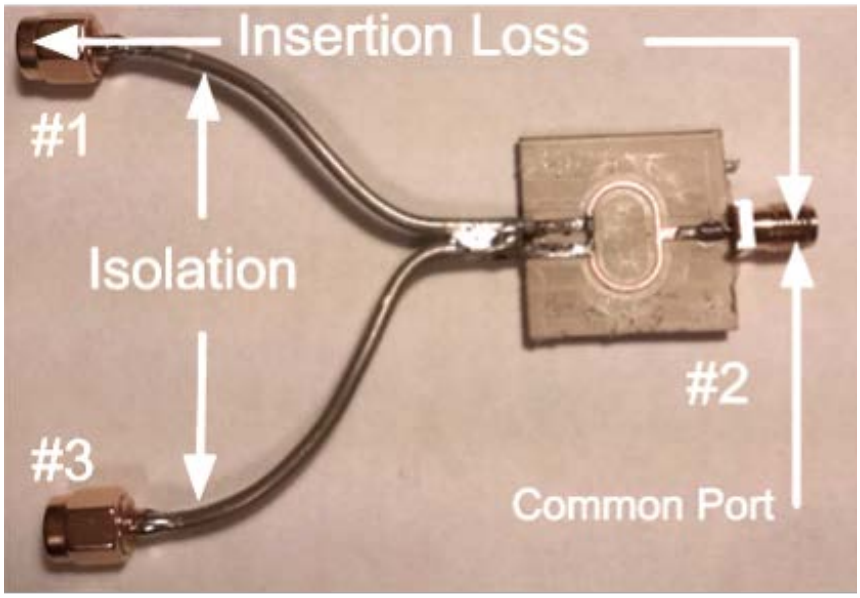


Figure 19. Microstrip Wilkinson Combiner

The combiner's input return loss, insertion loss, and isolation are displayed below. The plots reference the ports numbers depicted in figure 19.

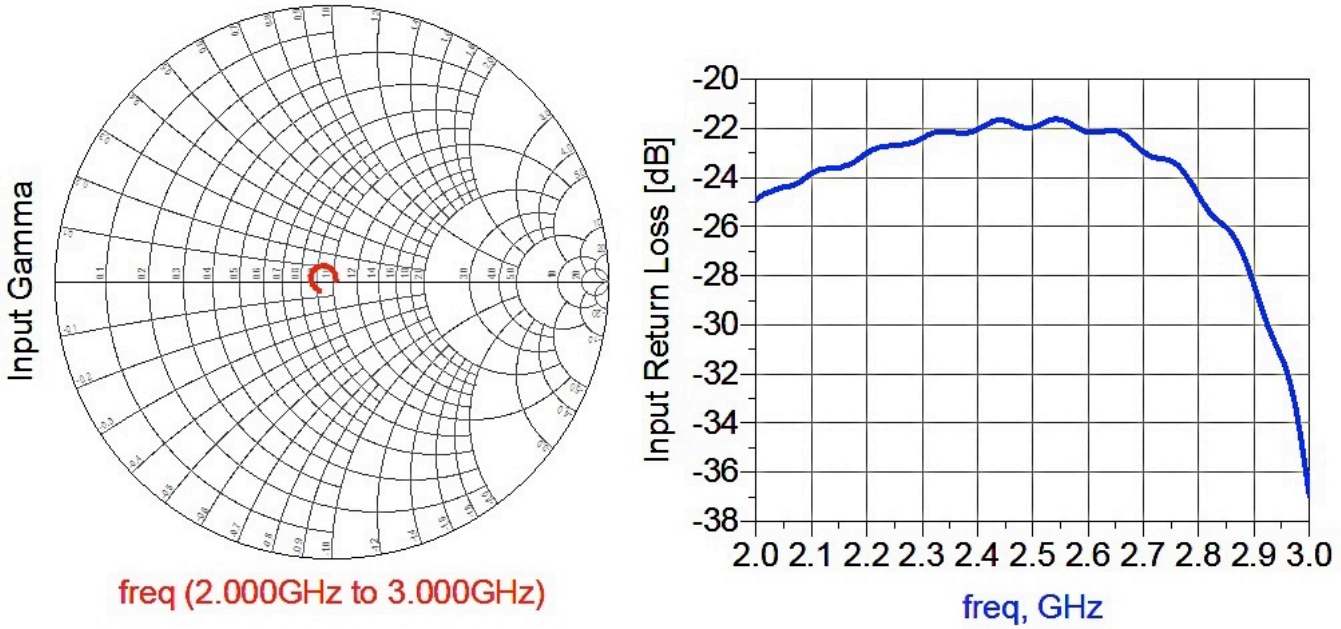


Figure 20. Wilkinson Combiner S_{11}

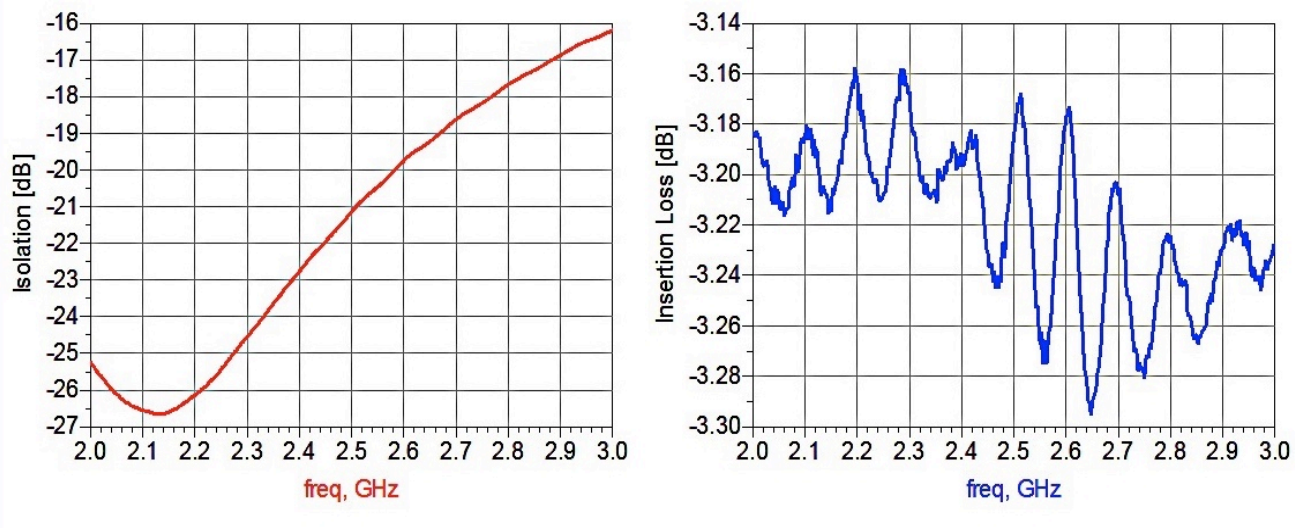


Figure 21. Wilkinson Combiner Isolation Red, Insertion Loss Blue

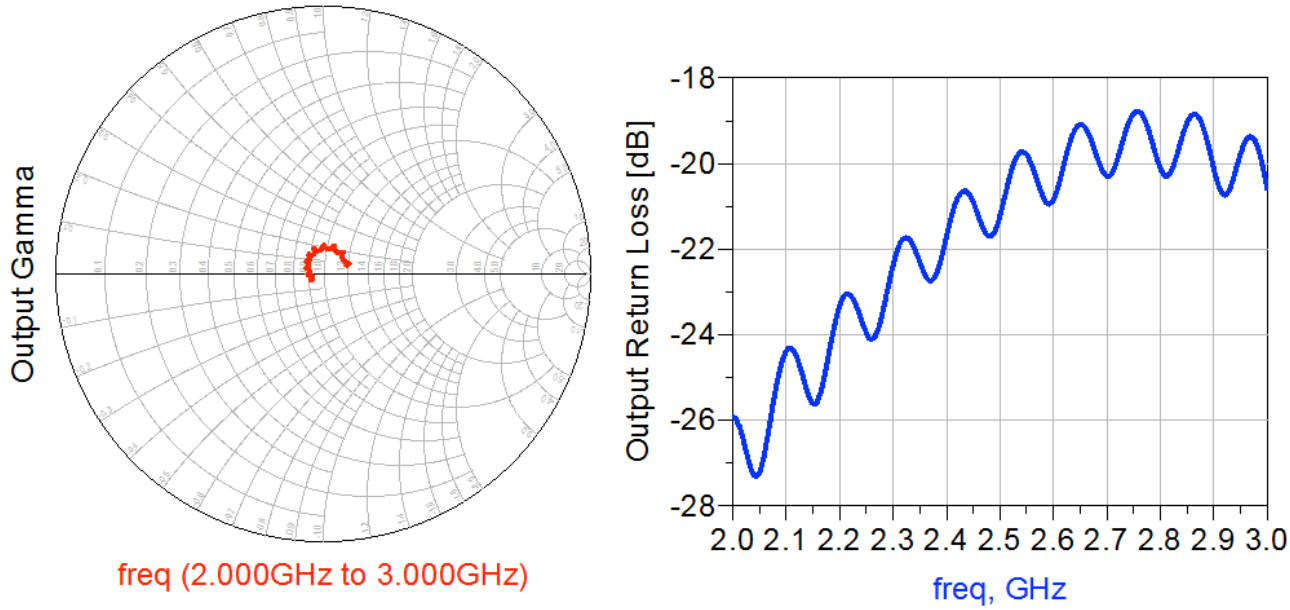


Figure 22. Wilkinson Combiner S_{22}

Section V.C. Available Two-Tone Power

The final two-tone power created by the cascaded amplifiers and Wilkinson combiner yielded approximately 23 dBm of available power (20 dBm per tone).

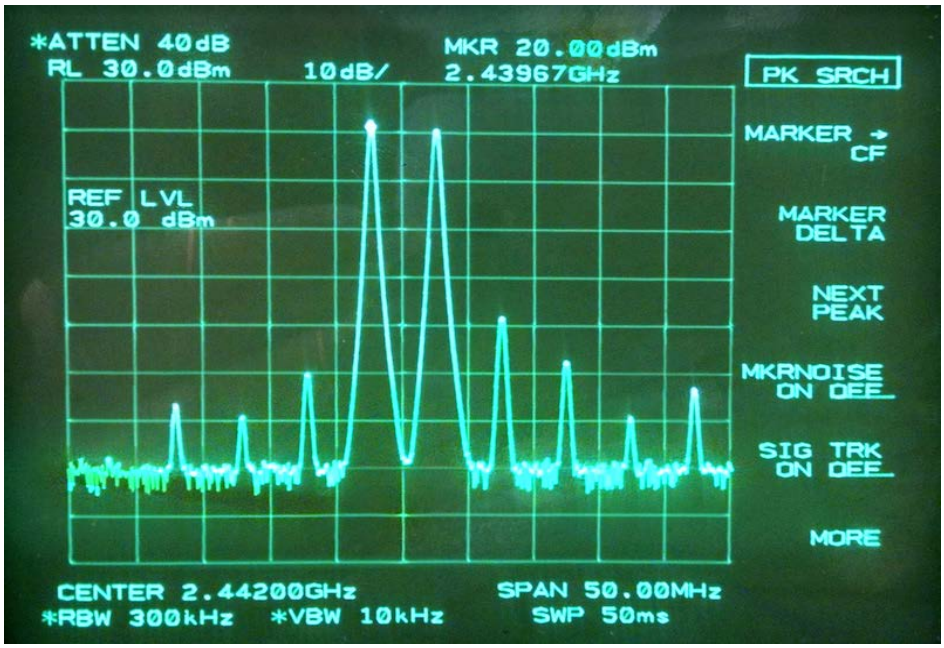


Figure 23. Available Source Power (dBm) vs. Frequency (Hz)

IMD was caused by Signal leakage from port 1 to 3.

Section VI. Measuring Output Power

- **Spectrum Analyzer Setup**

A spectrum analyzer was used to measure the PA's spectral output power. Since its absolute maximum input power is 30 dBm, a 20 dB attenuator was used to attenuate the output of the final PA, which was approximately 36 dBm.

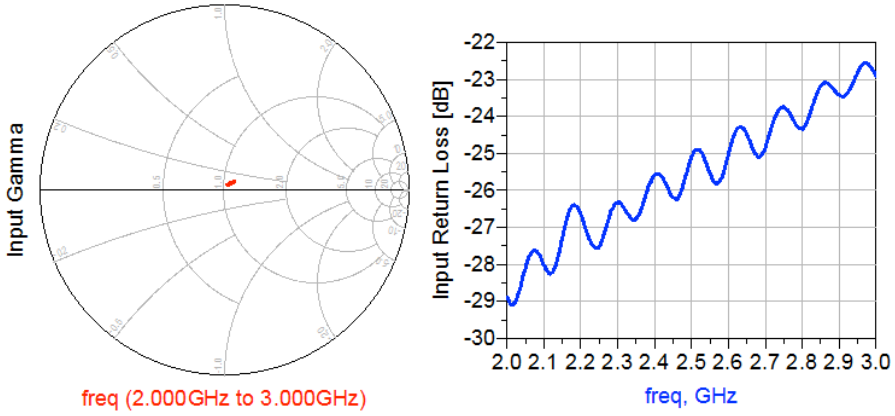


Figure 24. Attenuator Input Return Loss

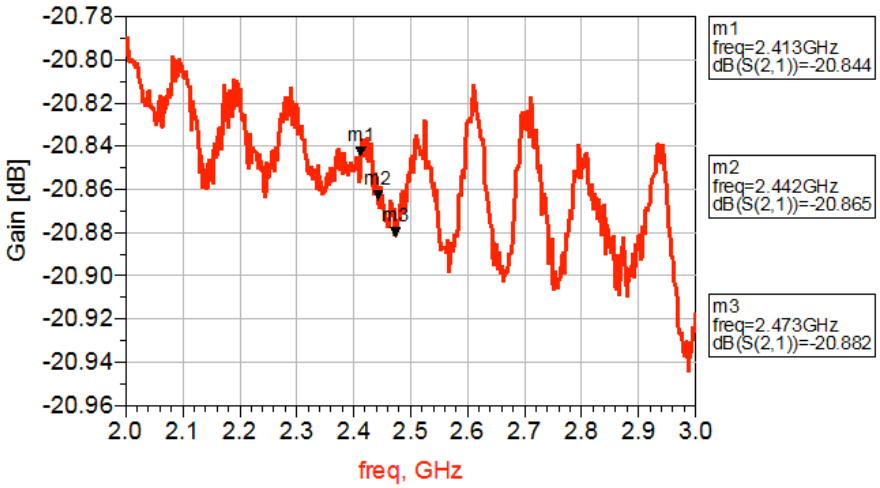


Figure 25. Attenuator Insertion Loss

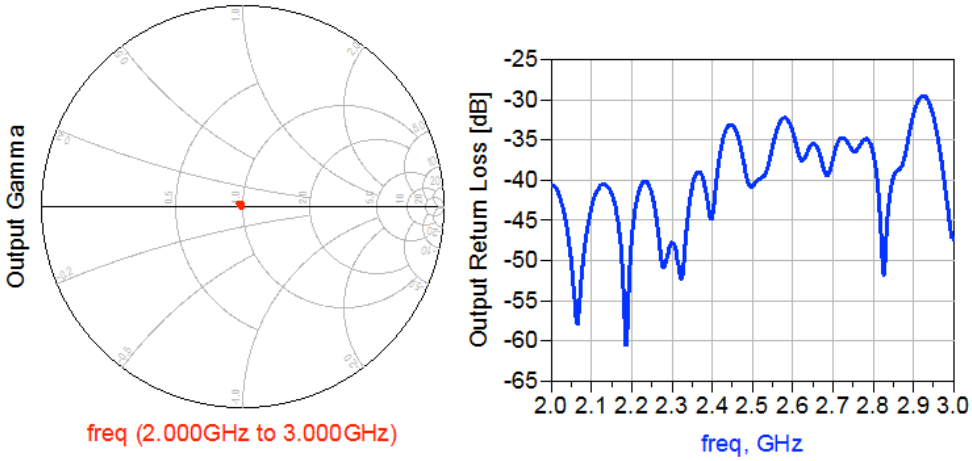


Figure 26. Attenuator Output Return Loss

- **PAE Measurements**

The amplifier achieved 40% PAE with an output power of 4.16W, which is what the simulation predicted. However, its IMD did not meet IMS requirements. Figure 27 shows the output spectra.

Figure 27. Amplifier Output after 20 dB Attenuator

Table 1. PAE Calculation from

Symbol	Sample Calculation	Result
V_{DD}	Measured	28.1V
I_D	Measured	350 mA
Attenuator	Fig. 25	20.85 dB
P_{IN} Tone	Fig. 23	20 dBm
P_{OUT} Tone	Fig. 27	12.33 dBm
P_{DC}	$28.1\ \text{V} \times 350\ \text{mA}$	9.8 W
P_{IN}	$2 \times 20\ \text{dBm}$	23 dBm
P_{IN}	$10^{\frac{23-30}{10}}$	200 mW
P_{OUT}	$12.33\ \text{dBm} + 3\ \text{dB} + 20.85\ \text{dB}$	36.18 dBm
P_{OUT}	$10^{\frac{36.18-30}{10}}$	4.15 W
PAE	$\frac{4.15\ W - 0.2\ W}{9.8\ W} \times 100\%$	40%
Drain Efficiency	$\frac{4.15\ W}{9.8\ W} \times 100\%$	42%

- **IMD Dependence on Gate Bias Voltage**

IMD products were recorded as gate voltage was swept from -3.2 to -2.5V in 0.1 V increments. Amplifier linearity depended on gate voltage bias, which is depicted in Figure 28.

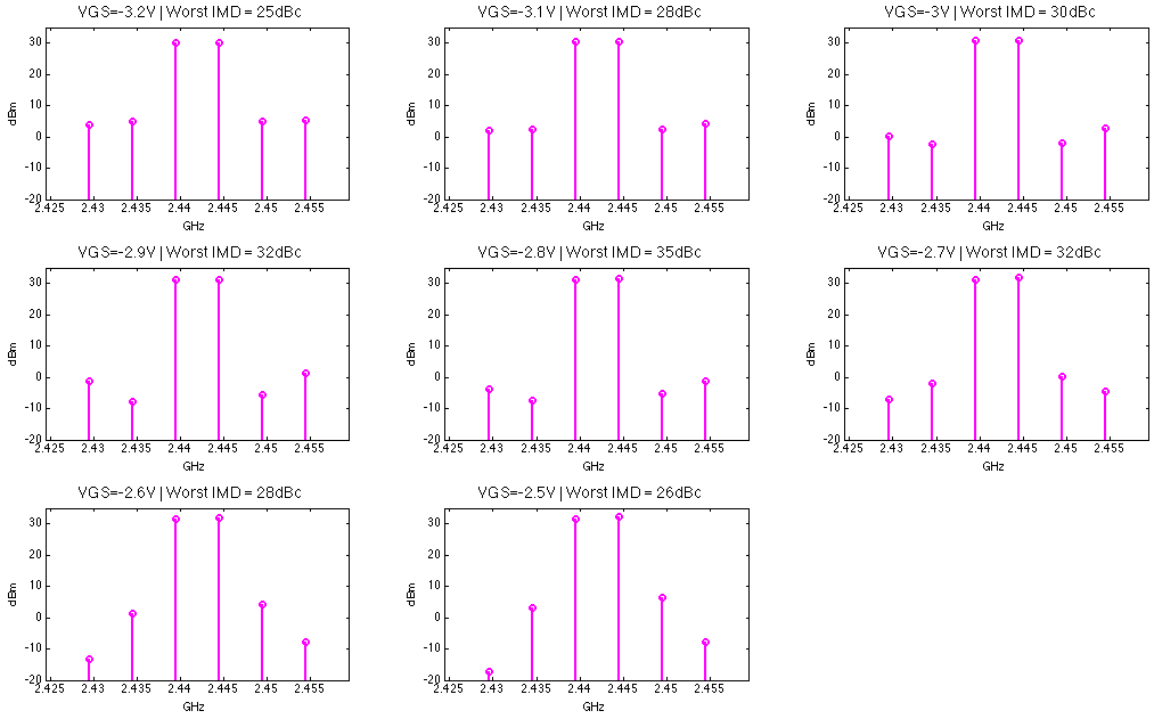


Figure 28. IMD Dependence on Gate Voltage

Figure 27 shows tuning gate bias voltage minimizes IMD products. Table 2 also shows that PAE is optimal when IMD is minimized.

Table 2. PAE vs. Gate Voltage

VGS [-V]	DC PWR [W]	PAE [%]
3.2	5.5	33.2
3.1	6.0	33.5
3.0	6.4	34.0
2.9	6.7	35.0
2.8	7.1	35.1
2.7	7.6	35.2
2.6	8.1	35.0
2.5	8.6	33.6

In order to understand the transistor's IMD dependence on gate voltage, ADS was used to measure the transistor's transconductance. In ADS, the transistor's gate voltage was swept from -3V to -2V while I_D was recorded with V_{DS} equal to 28V.

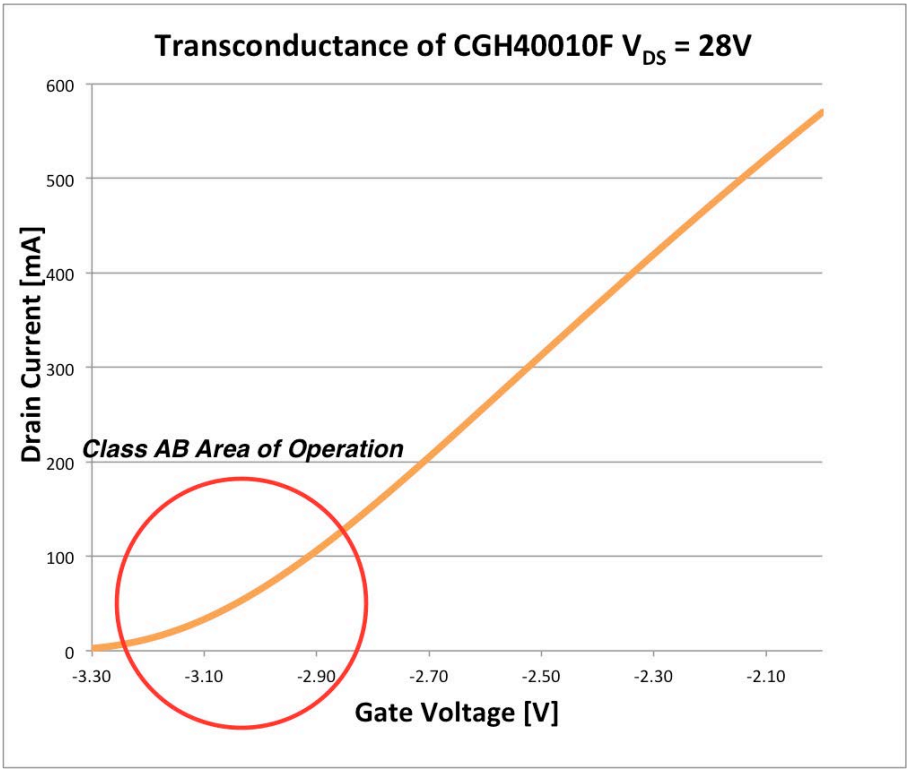


Figure 29. Transistor Transconductance

Class AB amplifiers are biased in the square law region of the device (red circle in figure 29). Sinusoidal variation of v_{gs} in the square law region does not yield a linear current waveform. In order to understand the relationship between IMD and gate voltage bias, Taylor series coefficients were found for a range of gate voltages [2]. Equation 3 represents a transistor's transconductance behavior biased at V_{GS}.

Equation 3:

$$i_{ds}(V_{GS} + v_{gs}) = I_{ds}(V_{GS}) + g_{m1}v_{gs} + g_{m2}v_{gs}^2 + g_{m3}v_{gs}^3 + g_{m4}v_{gs}^4 + g_{m5}v_{gs}^5 + \cdots$$

Substituting $v_{gs} = V_s\cos(\omega_1 t) + V_s\cos(\omega_2 t)$, an expression for third order intermodulation distortion (IMD3) was derived in [2].

Equation 4:

$$IMD3 = 20\log\left|\frac{\frac{3}{4}g_{m3}V_s^3 + \frac{25}{8}g_{m5}V_s^5}{g_{m1}V_s + \frac{9}{4}g_{m3}V_s^3 + \frac{25}{4}g_{m5}V_s^5}\right|$$

From equation 4, minimizing the g_{m3} and g_{m5} coefficients can minimize a transistor's IMD3. Minimizing g_{m3} and g_{m5} can be accomplished by adjusting the gate bias voltage.

The CGH40010F's DC transconductance was simulated in ADS. I_D vs. V_{GS} data was imported into in Excel and Taylor series coefficients were calculated. Derivatives were calculated by taking the slope of adjacent points.

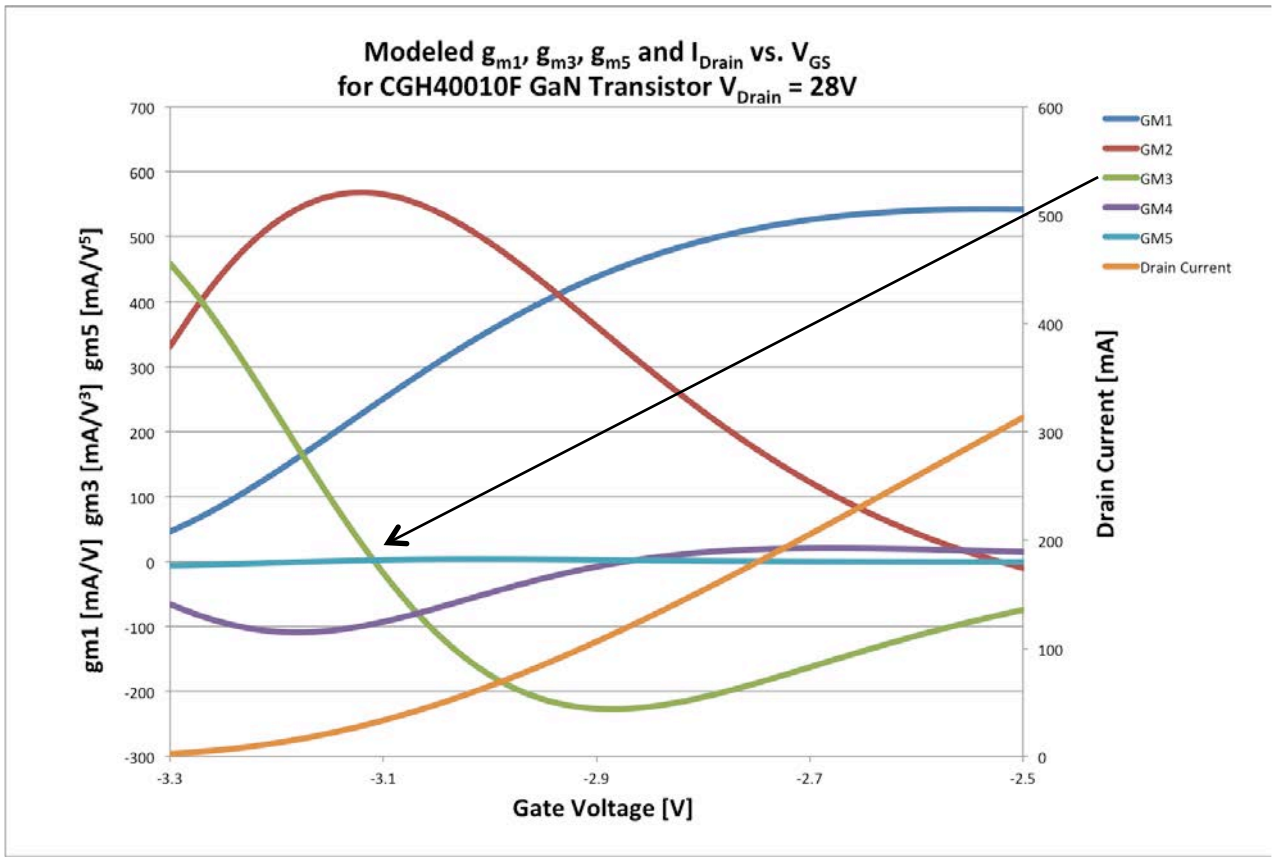


Figure 30. CGH40010F Taylor Series Coefficients

The above analysis suggests V_{GS} = -3.1 V minimizes IMD, which is incorrect. This is because it is not an RF model. It does not take into account V_{GS} dependent input impedance, and other frequency dependent factors. However, the analysis in [2] explains why tuning the gate voltage lowers IMD.

- **Continuous Wave Test**

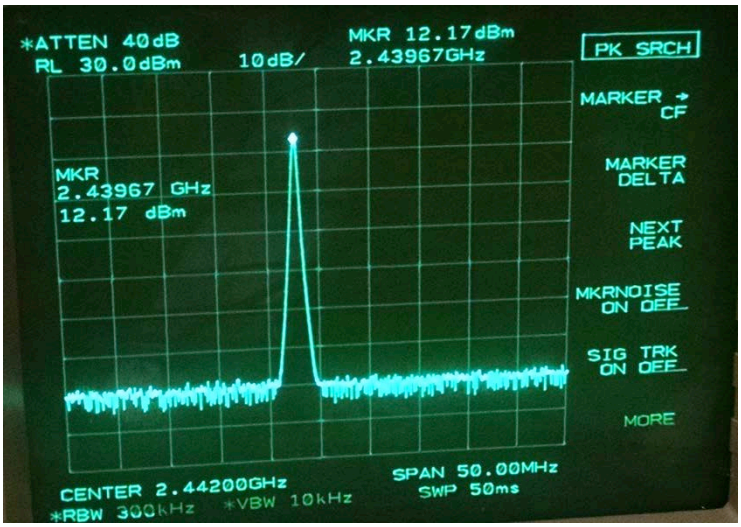


Figure 31. Output Power from CW Test

The PA did not meet the IMS CW requirement. When driven with a 23 dBm tone, the PA outputted 33 dBm; 3 dB less than the requirement.

Table 3. CW Output Power Calculation

Symbol	Sample Calculation	Result
Attenuator	Fig. 25	20.85 dB
P_{OUT} Tone	Fig. 30	12.17 dBm
Total P_{OUT} [dBm]	$20.85\ \text{dB} + 12.17\ \text{dBm}$	33.02 dBm
Total P_{OUT} [W]	$10^{\frac{33.02-30}{10}}$	2.00 W

VII. Conclusion

Even though the amplifier did not meet contest requirements, the project was an opportunity to learn more about class AB RF amplifiers - especially seeing how gate bias affects IMD. The project also provided an opportunity to become familiar with ADS's harmonic balance simulation capabilities, model libraries, and design guide. Keysight's knowledge center was an invaluable resource in learning how to use ADS. Bias networks were a major concern during this project. Often, designers used inductive RF chokes, which behave erratically at RF frequencies [3] [4]. In this design, quarter wave transmission lines were reliable in transforming the DC bias feeds, which were AC short-circuited to ground, to an open circuit. Lowering IMD was accomplished by ensuring good return loss at the output of the last pre-amplifier stages. This required every pre-amplifier to have input and output return loss of

about 20 dB, which required every pre-amplifier to be measured separately with a VNA – this took time. The mismatch between stages was caused by broken SMD coupling capacitors. The flexibility of the Rogers substrate allowed bending of the ceramic capacitors capacitor, which caused them to crack. If time were allowed for another revision of this project, the pre-amplifiers would be redesigned using another CGH40010F GaN transistor in order to increase available power. Power can then be controlled with a variable attenuator. The increased attenuation would not only improve the return loss seen by the pre-amplifier, but also the return loss seen by the main PA itself. Attenuation improves return loss by attenuating reflected signals seen by the driving amplifier – incident signals are much higher than reflections, which increases the load's match.

Contact Information:

rjtong@calpoly.edu
robbytong@gmail.com

Works Cited

[1] Nahas, M, S Abdul-Nabi, L Bouchnak, and F Sabeh. "Reducing Energy Consumption in Cellular Networks by Adjusting Transmitted Power of Base Stations." *2012 Symposium on Broadband Networks and Fast Internet (RELABIRA)*, (2012): 39-44.

[2] Van Der Heijden, M.P, H.C de Graaff, L.C.N de Vreede, J.R Gajadharsing, and J.N Burghartz. "Ultra-linear Distributed Class-AB LDMOS RF Power Amplifier for Base Stations." *2001 IEEE MTT-S International Microwave Sympsoium Digest (Cat. No.01CH37157)*, 2 (2001): 1363-1366 vol.2.

[3] Tsang, Kai Shing. *Class-F Power Amplifier with Maximized PAE*. n.p.: DigitalCommons@CalPoly, .

[4] Kim, Bumjin, D Derickson, and C Sun. "A High Power, High Efficiency Amplifier Using GaN HEMT." *2007 Asia-Pacific Microwave Conference*, (2007): 1-4.

Design and Implementation of a Software Costas Loop for Audio Frequencies

By

Robert J. Tong

Electrical Engineering Department

California Polytechnic State University

San Luis Obispo

2017

TABLE OF CONTENTS

1. Abstract

A Costas loop is implemented in software with Python. A small-signal linear model is derived in order to determine loop stability. An interactive program was developed to tune loop parameters in order to obtain critically damped step response. The effect of lowpass filter phase delay on stability is discussed.

2. List of Acronyms

PLL – Phase Locked Loop
FFT – Fast Fourier Transform
VCO – Voltage Controlled Oscillator
BPSK – Binary Phase Shift Keying/Keyed
SOS – Second Order System

3. Table of Programs and their Descriptions

Table 1. Table of Python Programs

`error_detector.py`	Plots the error detector response of the PLL and Costas loop. Given an initial error, the loop action to minimize phase error is iteratively plotted. Figures Produced: 1. 2 subplots consisting of the PLL and Costas loop error detector. Both are phase sweep from -2π to $+2\pi$.
`linear_error_detector.py`	The Costas loop error detector is linearized in order to model it in the frequency domain. The linear model is verified by showing the linearized error detector closely resembles the real error detector for small phase deviations. Figures Produced: 1. Error detector response to phase step.
`LowPass.py`	Biquad lowpass filter. This file contains the lowpass filter class that is used in other programs. Figures Produced: None

`lpf_frequency.py`	Plot the Z-domain frequency response of the biquad lowpass filter. Figures Produced: 1. Biquad lowpass filter magnitude response in analog and digital domains.
`lpf_impulse.py`	Takes the 2048-point FFT of the biquad filter's impulse response. This is to verify that the digital filter has the desired frequency response. Figures Produced: 1. FFT magnitude of biquad impulse response.
`Integrator.py`	Trapezoidal integrator. This file contains the integrator class used in other programs. Figures Produced: None
`costa_designer.py`	Interactively plots the linearized closed loop frequency response of the Costas loop. Also plots the transient phase response. Has GUI sliders that allow the user to change loop parameters. Figures Produced: 1. Costa loop design window with 5 subplots depicting the magnitude and phase of the closed and open loop responses, in addition to the transient step response.
`step.py`	Used by costa_designer.py to generate transient response. Figures Produced: None

`costa.py`	Software Costas loop. It was designed to be translated into faster languages such as C. Figures Produced: 1. Figure with subplots illustrating Costas loop error signal, and VCO in-phase output overlapped on top of input reference signal to show accuracy of phase tracking. 2. FFT of error detector signal to illustrate the double frequency component produced by Costas loop mixers.
`simplified_costa.py`	Simplified software Costas loop. Minimizes calls to plotting library to avoid code bloating and obfuscation. Figures Produced: 1. Figure with subplots illustrating Costas loop error signal, and VCO in-phase output overlapped on top of input reference signal.

4. A Note about Python and Scientific Computing

Python can be used for scientific computing using open source libraries that enable it to have some of MATLAB's basic features, such as figures, plots, subplots, arrays, multiplying arrays, etc. The two open source libraries that enable this are NumPy and Matplotlib. NumPy is used to for numerical computing, offering routines such as sin, cos, FFT, etc. Matplotlib is used to plot data, using an interface that is very similar to MATLAB.

The following table shows some syntax comparisons between Python with NumPy and Matplotlib, and MATLAB.

You must first import the libraries:

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

Python	MATLAB

`# Create an array of zeros`
`output = np.zeros(123)`
 	
`# Generate 3 kHz sinewave`
`fs = 44.1E3`
`fc = 3E3`
 	
`times = np.arange(0, 123)/fs`
`signal = np.cos(2.0 * np.pi * fc * times)`
	`# Create an array of zeros`
	`output = zeros(1, 123)`
	
	`# Generate 3 kHz sinewave`
	`fs = 44.1E3`
	`fc = 3E3`
	
	`times = (0:123)/fs`
	`signal = cos(2.0 * pi * fc * times)`

```
# Plot 3 kHz sinewave
plt.subplot(2,1,1)
plt.plot(times, signal)
plt.show()
```

```
# Plot 3 kHz sinewave
subplot(2,1,1)
plot(times, signal)
```

See http://www.numpy.org/, and https://matplotlib.org/ for more information.

5. Introduction

Before showing the Costas loop block diagram, it is helpful to know that the Costas loop is a modification of the PLL. The only difference between the Costas loop and the PLL is its error detector. The Costas loop error detector is ideally (1), while the PLL error detector is ideally (2).

$$e(t) = \sin\left(2 \cdot \left(\phi(t) - \theta(t)\right)\right) \quad (1)$$

$$e(t) \quad \sin\left(\quad (t) - \theta(t)\right) \quad (2)$$

Where the difference $\phi(t) - \theta(t)$ represents the phase delta between the input and VCO signals. Although the Costas loop is implemented in software, such that the oscillators are driven with digital floating point signals, the analog nomenclature, such as VCO, is maintained.

Most of the hardware in the Costas loop is used to implement (1). Figure 1 depicts the Costas loop in its entirety.

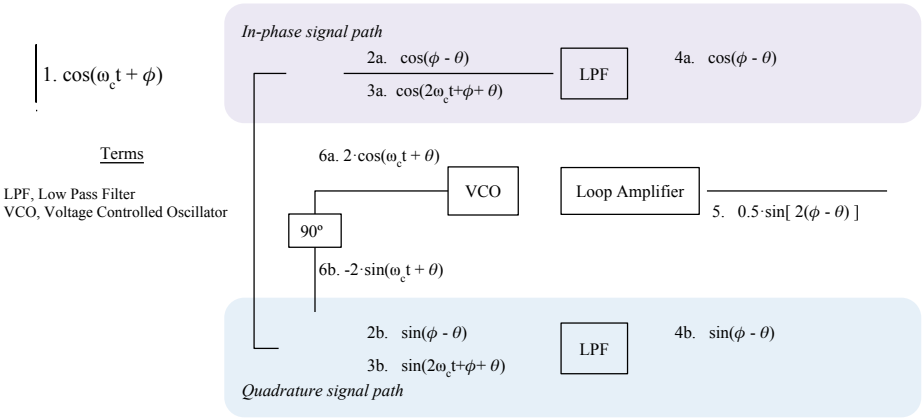


Figure 1. Costas Loop Block Diagram

Various signals in Figure 1 are assigned reference numbers since they will be referred to throughout this paper.

Table 2. Signal Descriptions

Reference	Description
1	Input signal
2a	Unfiltered in-phase DC component
2b	Unfiltered quadrature DC component

3a	Unfiltered in-phase double frequency component
3b	Unfiltered quadrature double frequency component
4a	Filtered in-phase DC component
4b	Filtered quadrature DC component
5	Error detector output
6a	VCO in-phase output
6b	VCO quadrature output

Note:
References to signals in Table 2 are indicated by surrounding the reference signal number with curly brackets.

For example, the VCO in-phase output is {6a}.

Furthermore, the Costas loop in Figure 1, can be reorganized into two high level blocks: Error detector and loop amplifier.

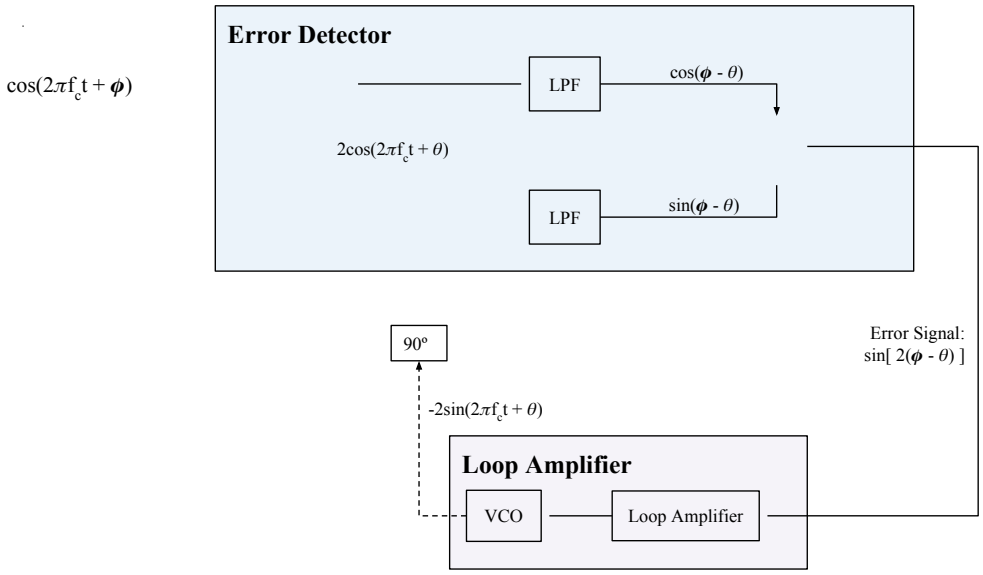


Figure 2. Re-drawn Costas Loop

Figure 2 shows that all of the multipliers, lowpass filters, and VCO signals are used to generate the error signal.

What is so Special About the Costas Loop?

The Costas loop error detector (2) enables it to decode BPSK data. In BPSK modulation, data is conveyed by phase shifting a constant carrier by 0º or 180º.

The Costas loop error detector enables it to decode BPSK data, because it drives the VCO to track to either 0º or 180º phase difference between it and the input signal. This behavior is illustrated below in Figures 3, 4, and 5. The ability to track either 0º or 180º is illustrated first in Figure 3, by plotting the PLL and Costas loop error detector output as a function of $\phi - \theta$.

Figure 4, shows the PLL and Costas loop both converging to zero with an initial -0.25π phase difference. Then, in Figure 5, the same test is repeated, but with an initial phase difference of -0.95π. The Costas loop converges to $-\pi$, while the PLL converges to zero. This unique behavior, allowing either zero or $\pm\pi$ phase difference, allows the Costas loop to demodulate BPSK data.

For example, in a differentially encoded system, a phase shift of π indicates '1' (one), and no phase shift indicates '0' (zero), the Costas loop would be unaffected by input phase shifts of 180º. If a '1' were sent, {4a}, would toggle from either +1 to -1 or -1 to +1.

However, a PLL cannot decode BPSK data, because its error detector forces the VCO to have zero input signal phase difference. When the input signal shifts by 180º, the PLL control loop adjusts the VCO to achieve zero phase difference.

Figure 3 (below) plots (1) and (2) as a function of phase difference $\phi - \theta$.

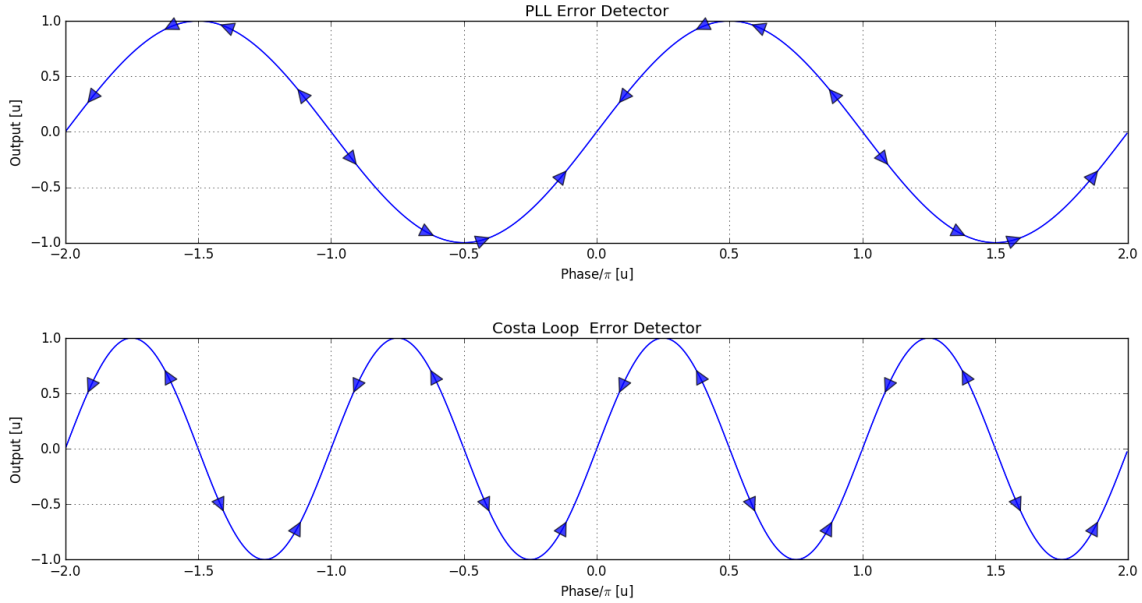


Figure 3. `error_detector.py`: PLL and Costas Loop Error Detector

The blue arrows in Figure 3 indicate the direction the error detector drives the loop amplifier and VCO. The PLL error detector shows the loop converging to either 0, or $\pm 2\pi$. The Costas loop converges to either 0, $\pm\pi$, or $\pm 2\pi$. An example of this is shown in Figures 4 and 5, where the initial phase difference is set to be -0.25π.

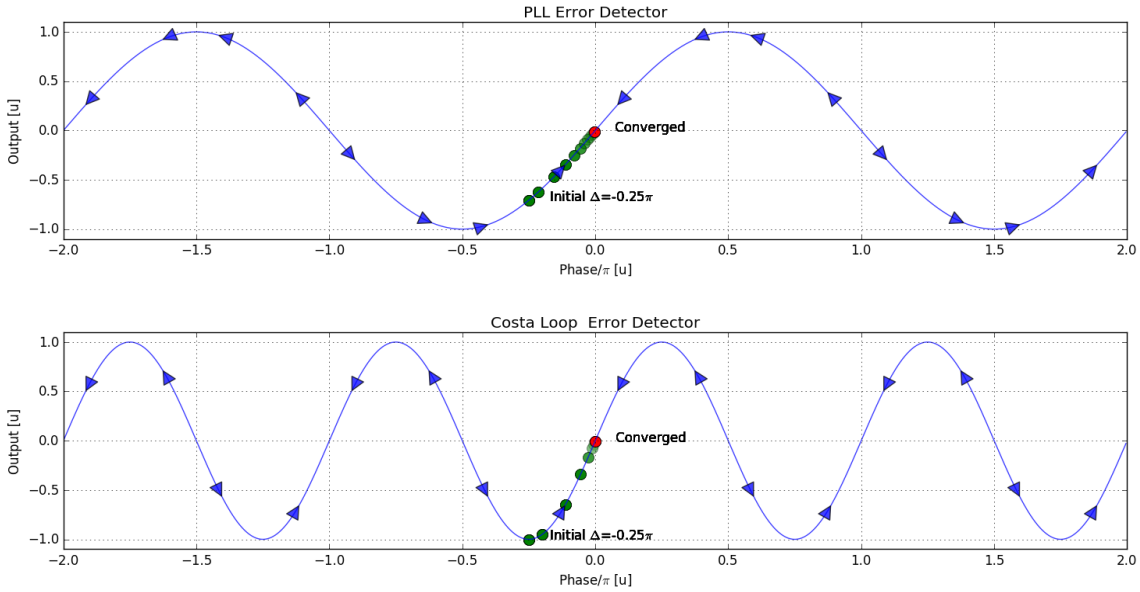


Figure 3. `error_detector.py`: PLL and Costas Loop Convergence with Initial -0.25π Phase Difference

Both the PLL and Costas loop converge to 0º phase difference when initial phase difference is -0.25π.

Instead of $-\pi$ to model BPSK modulation, -0.95π is used in Figure 4. This is because the PLL has zero error at π. It will already be converged. However, it is unstable at this point, since any perturbation will cause the PLL to converge to either -2π, or 0.

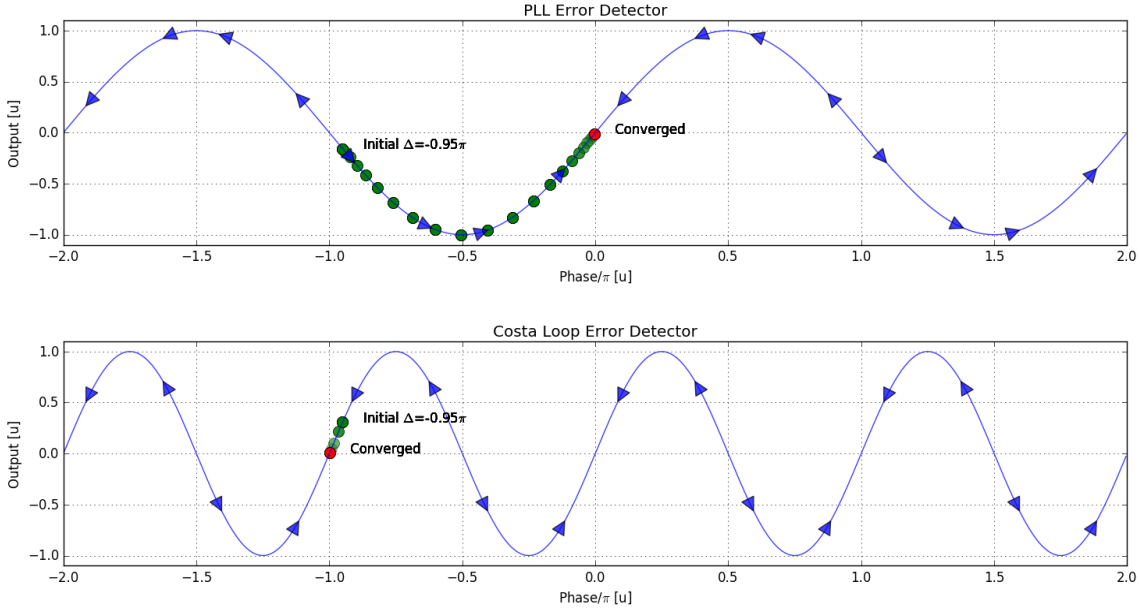


Figure 4. `error_detector.py`: PLL and Costas Loop Convergence with Initial -0.95π Phase Difference

At –0.95π, the PLL converges to 0, while the Costas loop converges to –π.

The Costas loop error detector enables it to demodulate BPSK modulated data.

---

6. Modelling the Costas Loop in the Frequency Domain

The Costas loop error detector is linearized to derive its small-signal frequency domain model. Then, the loop amplifier from [1] is introduced. The loop amplifier enables the Costas loop to track input signals with a frequency offset.

Error Detector Modeling

Figure 5 depicts a simplified version of the error detector. It ignores double frequencies {3a} {3b}, because it assumes the lowpass filters will sufficiently suppress them.

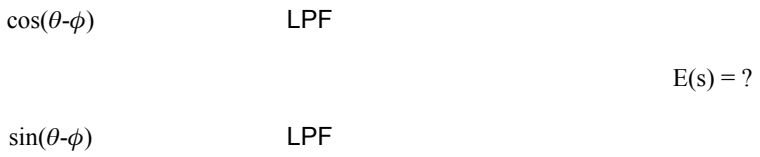


Figure 5. Error Detector

If the phase difference is small, then the following approximations can be made:

$$\cos(\quad -\theta) \approx 1 \qquad (3)$$

$$\sin(\quad -\theta) \approx \quad -\theta \qquad (4)$$

{4a} becomes unity, making {5} equal to {4b}.
Then, the error signal is approximated to be:

$$e(t) \quad h_{lpf}(t) \quad \big(\quad (t) - \theta(t) \big) \qquad (5)$$

$$E(s) \quad H_{lpf}(s) \quad \big(\Phi(s) - \Theta(s)\big) \qquad (6)$$

Error Detector Model Verification

`linear_vs_real.py` compares the phase step error detector response of the simplified linear model (5) to the complete Costas loop error detector implementation (Figure 5).

Figure 6 depicts the two error detector's responses to a small phase step of 0.1 radians.

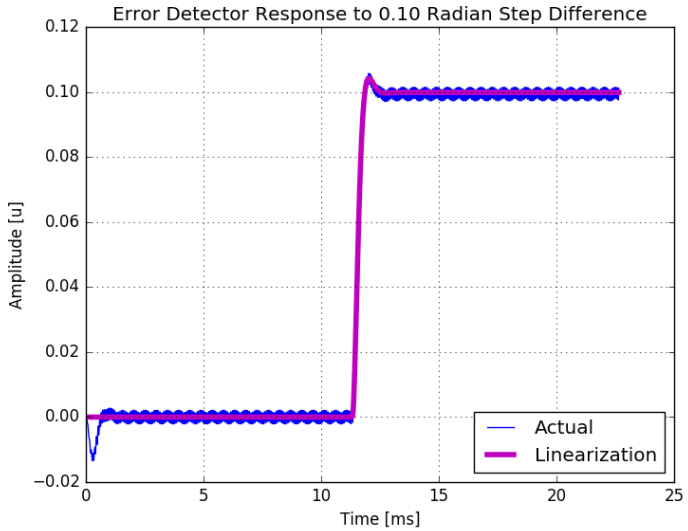


Figure 6. `linear_error_detector.py`: Small Signal Error Detector Responses

The linear approximation holds well for small phase differences.

Figure 7 is the same test but with a 0.5 radian step difference.

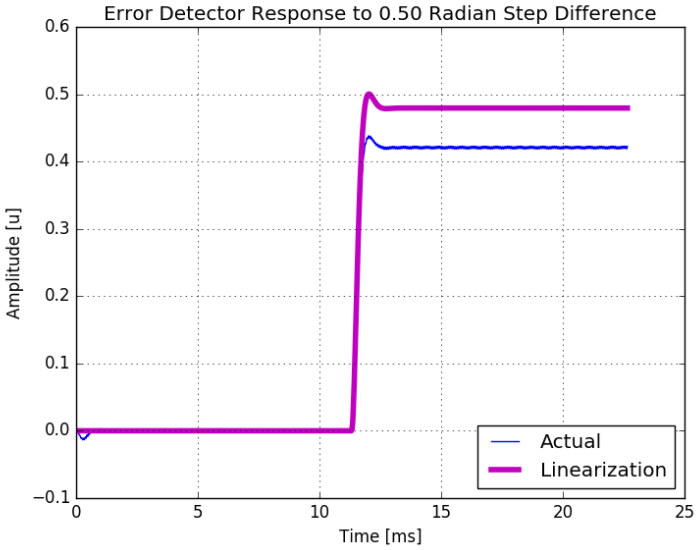


Figure 7. `linear_error_detector.py`: Large Signal Error Detector Responses

The approximation made in (5) breaks down for larger phase differences.

Loop Amplifier

The equation for the loop amplifier is given by (7) from [1].

$$F(s) = G\frac{s+a}{s} \quad (7)$$

(7), is to use d to achieve a closed loop response that has zero steady state error to ramping phase inputs, i.e. the Costas loop will be able to lock onto input signals with higher or lower frequencies from its VCO's center frequency within a certain bandwidth.

Figure 8 (below) illustrates the linear PLL block diagram that (7) was referenced from.

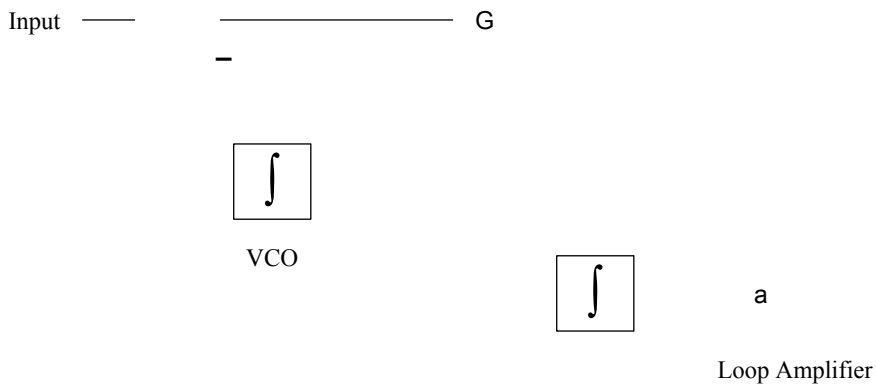


Figure 8. Second Order PLL Block Diagram

The closed loop response for the second order PLL is given by:

$$H_{PLL}(s) = \frac{H_{open}(s)}{1 + H_{open}(s)} \quad (8)$$

$$H_{open}(s) = \frac{F(s)}{s} = \frac{G(s+a)}{s^2} \quad (9)$$

$$H_{PLL}(s) = \frac{G(s+a)}{s^2 + Gs + Ga} \quad (10)$$

Equating (9) with the standard form of a second order system (SOS):

$$H_{SOS}(s) = \frac{\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2} \quad (11)$$

$$G = 4\pi\zeta f_n \quad (12)$$

$$a = \frac{\pi f_n}{\zeta} \quad (13)$$

Using the above loop amplifier makes the loop a type II system, because the loop possesses two perfect integrators.

Putting Everything Together

The analytical Costas loop model combines the error detector given by (6), and the loop amplifier in (7). $G_{open}(s)$, the open loop gain of the Costas loop is:

$$G_{open}(s) = H_{lpf}(s) \cdot G \cdot \frac{s + a}{s^2} \quad (14)$$

The closed loop response then becomes:

$$H_{Costa}(s) = \frac{G_{open}(s)}{1 + G_{open}(s)} \quad (15)$$

---

7. Using the Bilinear Transform to Convert from S-Domain to Z-Domain

The integrator is implemented using trapezoidal integration [2].

Trapezoidal integration is given by the following difference equation [2]:

$$y[n] - y[n-1] = \frac{T}{2}(x[n-1] + x[n]) \quad (16)$$

Where T is the sampling period.
Converting to the Z-domain, the transfer function for an integrator is given by:

$$H_{Integrator}(z) = \frac{T}{2}\frac{(1 + \quad ^{-} \quad)}{1 - z^{-1}} \quad (17)$$

$H_{Integrator}(z)$ is equated with 1/s since they both represent integration.

$$\frac{1}{s} \approx \frac{T}{2}\frac{1+\ ^{-}}{1-z^{-1}} \qquad (18)$$

$$s \approx \frac{2}{T}\frac{1-\ ^{-}}{1+z^{-1}} \qquad (19)$$

(19) is the bilinear transform.

The following discusses how to implement (18) in software.

Figure 9 (below) shows the progression in which (18) was reduced to its canonical (minimal z^{-1} delay blocks) form [3].

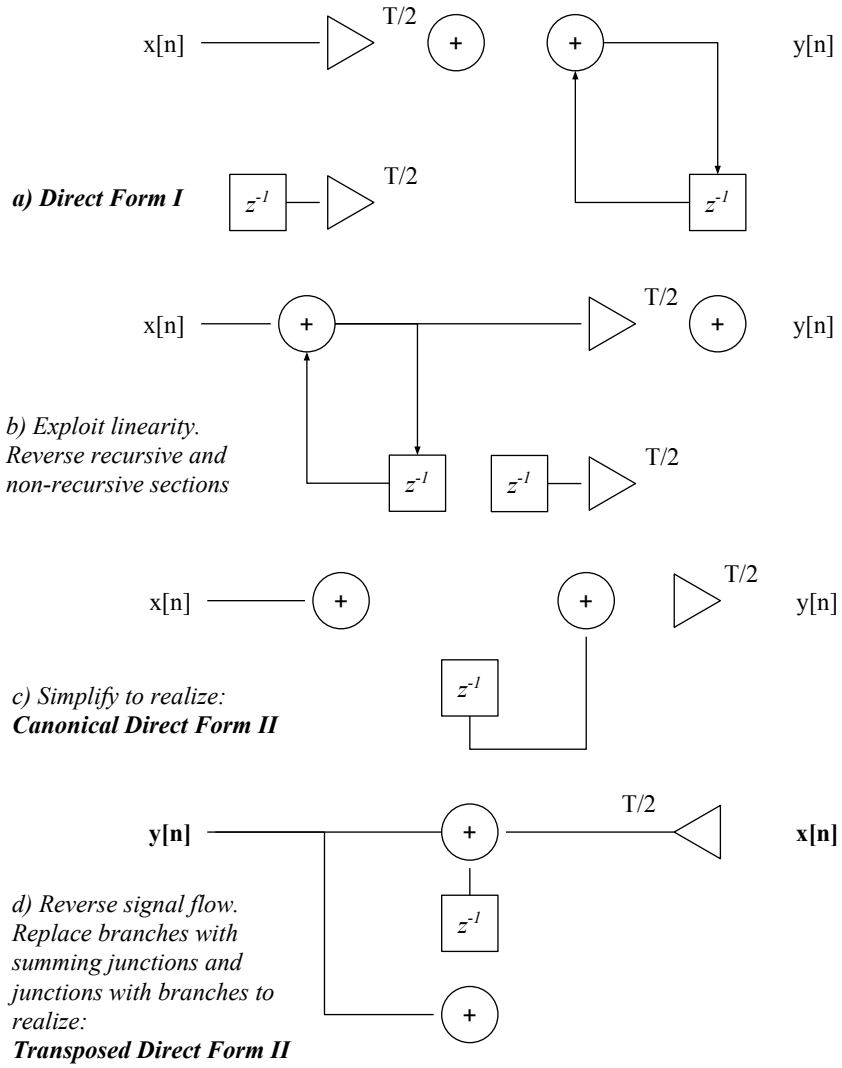


Figure 9. Deriving the Transposed Direct Form II of (13)

Figure 9 represents the steps taken in order to realize the transposed direct II form of (18).

The process first begins with Figure 9(a), in which (18) is directly translated into gain, delay, and summing blocks. The non-recursive and recursive sections are then swapped in Figure 9(b), which are then simplified to realize its canonical (minimal delay blocks) form. The canonical form, which is also called direct II form, is depicted in Figure 9(c). Finally, Figure 9(c) is transposed to realize Figure 9(d). The transpose operation reverses signal flow directions, replacing summing junctions with branches, and branches with summing junctions [3].

An example of implementing Figure 9(d) in Python is given below.

The following Python code implements Figure 9(d).

```
import numpy as np

...

save = 0.0
temp = 0.0
out  = np.zeros(len(sig))
fs   = 44.1E3
T    = 1.0 / fs

for k in range(len(sig)):
     temp   = sig[k] * T / 2.0
     out[k] = temp + save
     save   = temp + out[k]
```

---

8. Biquad Low Pass Filter

The biquad lowpass filter used in the Costas loop error detector is discussed in this section.

The biquad LPF is the bilinear transformation of a second-order lowpass filter. The second-order lowpass filter is described by (20) [4]

$$H(s) = \frac{\omega_p^2}{s^2 + \frac{\omega_p}{Q}s + \omega_p^2} \qquad (20)$$

Where ω_p represents the passband frequency.

The bilinear transform (19), is repeated below:

$$s = 2f_s\frac{z-1}{z+1}$$

For convenience,

$$k = 2f_s$$

Substitute (19) into (20),

$$H(z) = \frac{\omega_p^2}{k^2\left(\frac{z-1}{z+1}\right)^2 + \frac{\omega_p k}{Q}\left(\frac{z-1}{z+1}\right) + \omega_p^2} \quad (21)$$

Simplify,

$$H(z) = \frac{\omega_p^2(z^2+2z+1)}{\left(k^2 + \frac{\omega_p k}{Q} + \omega_p^2\right)z^2 + \left(2\omega_p^2 - 2k^2\right)z + \left(\omega_p^2 - \frac{\omega_p k}{Q} + k^2\right)} \quad (22)$$

$$A = k^2 + \frac{\omega_p k}{Q} + \omega_p^2 \quad (23)$$

$$B \quad 2\omega_p^2 - 2k^2 \quad (24)$$

$$C = \omega_p^2 - \frac{\omega_p k}{Q} + k^2 \quad (25)$$

$$H(z) = \frac{\omega_p^2}{A}\frac{1 + 2^{-} + ^{-2}}{1 + \frac{B}{A}z^{-1} + \frac{C}{A}z^{-2}} \quad (26)$$

The biquad filter derives its name from the fact that both the numerator and the denominator are quadratic functions of z.

ω_p must be pre-warped to the z-domain [5].

$$\omega_p = 2f_s \tan\left(\frac{f_p}{2f_s}\right) \quad (27)$$

Where f_s is the sampling rate, and $f_p{}'$ is the *desired* cutoff frequency for the biquad filter.

Figure 10 is the direct form I realization of (26).

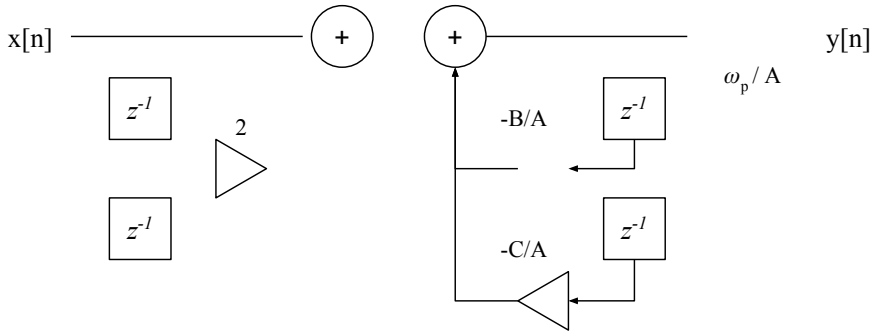


Figure 10. Direct Form I Realization of (26)

Figure 11 is the canonical direct form II realization of (26).

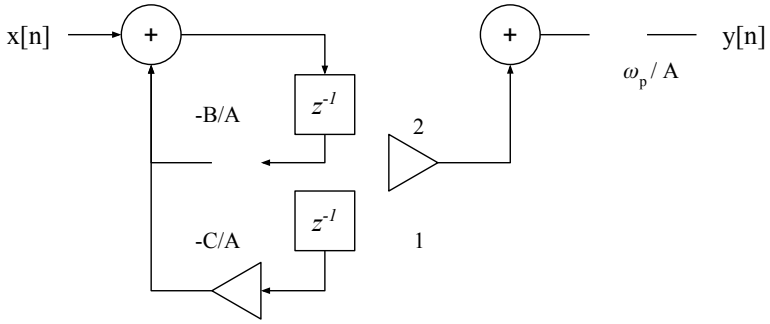


Figure 11. Canonical Direct Form I Realization of (26)

Figure 11 uses the half the number of delay blocks that figure 10 uses.

Figures 12 and 13, validate (26) and the canonical realization depicted by Figure 11. Figure 12 compares the frequency response of (26) to that of the second-order lowpass filter it was based upon. Figure 13 plots the FFT of the impulse response produced by the filter depicted in Figure 11, in order to ensure its behavior is what was intended.

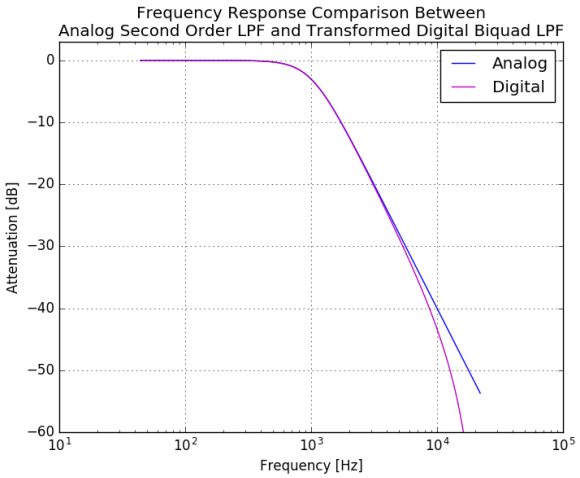


Figure 12. `biquad.py`: Frequency Response of (26) with f_p' set to 1 kHz.

The responses of both filters are approximately aligned until 10 kHz, after which, the attenuation of the digital filter increases at a faster rate than its analog counterpart. This is due to the mapping of the analog frequency domain's $f = \infty$ Hz, to the Nyquist frequency $f = f_s/2$.

The fast Fourier transform (FFT) of the impulse response produced by a program implementing Figure 11 is shown below in Figure 13. Figure 13 plots the frequency response up to the Nyquist frequency.

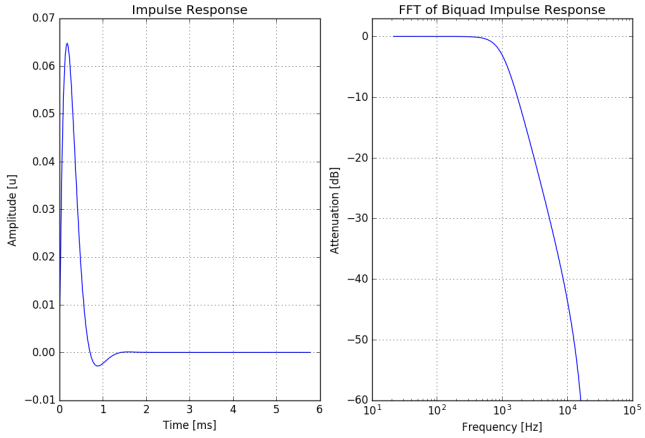


Figure 13. `biquad_fft.py`: FFT of Impulse Response Produced by Figure 11

Since the FFT of the 2048-point impulse response produced by the implementation of Figure 11 closely resembles the expected frequency response depicted in figure 12, the minimal z-delay block implementation depicted by Figure 11 is correct.

---

9. Software Costas Loop Implementation Details

Implementation of the software Costas loop is discussed by walking through `simple_costa.py`. `simple_costa.py` is a stripped-down version of `costa.py`. It focuses only on implementing the software Costas loop algorithm and plotting its output signals.

Figure 14 (below) describes the implementation of each block in the Costas loop and their corresponding Python file.

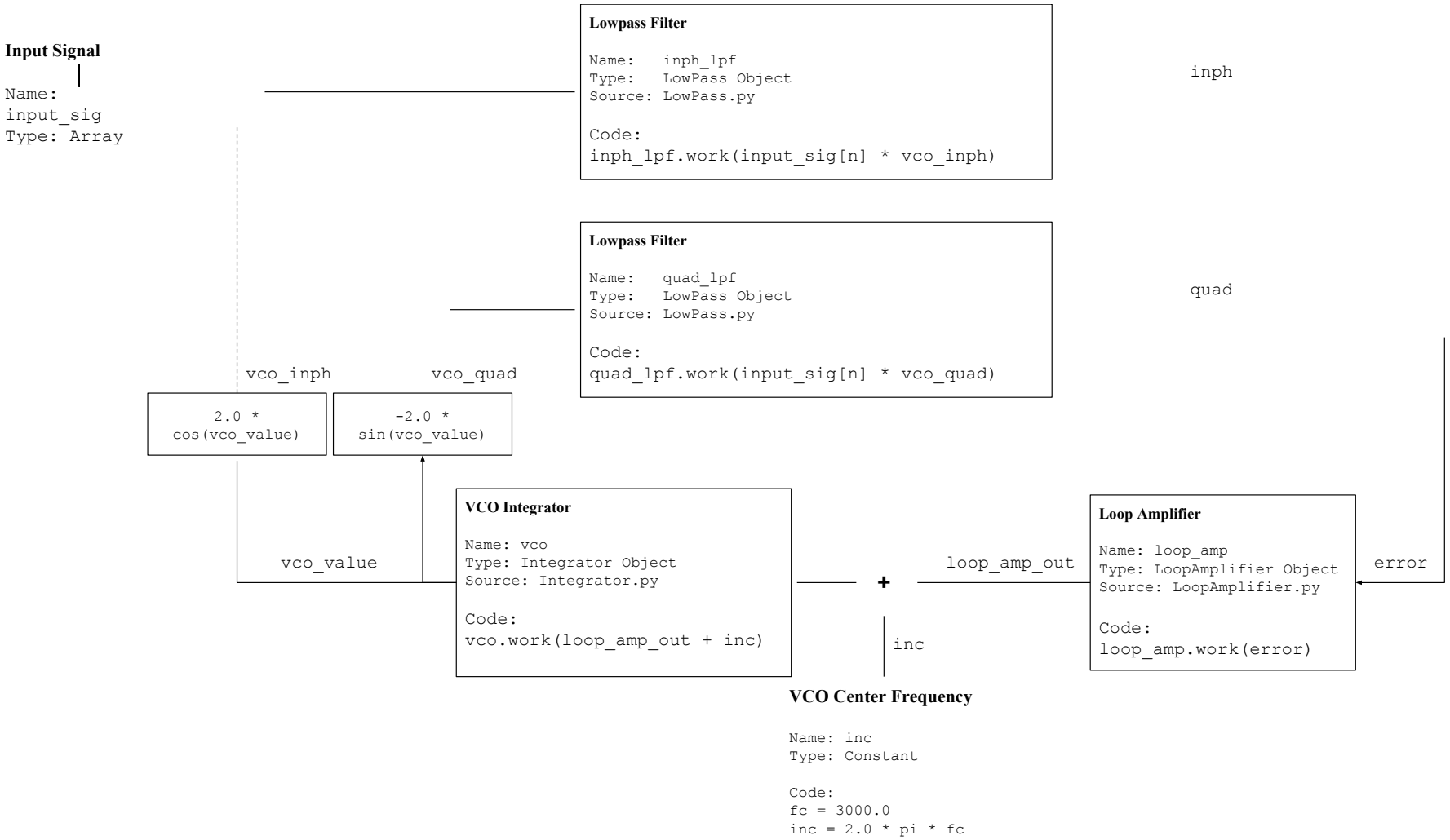


Figure 14. Block Diagram of `costa.py`

A new addition to the Costas loop block diagram is shown in Figure 14 – the summing of the loop amplifier output with a constant DC term representing the VCO center frequency. This is an important detail that [1] does not discuss.

Outcomes of `simplified_costa.py`

The Costas loop in `simplified_costa.py` was used to verify one, whether or not the software algorithm is able to track the phase of an input signal, and two, whether or not a set of loop parameters leads to an unstable response. Stability is tested by instantaneously increasing the phase of the input signal by 45º (phase step) after the loop has converged.

Program Flow of `simplified_costa.py`

`simplified_costa.py` is divided into the following sections:

i. User customization
ii. Calculating constants
iii. Initializing signal processing blocks
iv. Initializing arrays for the input and output data
v. The main loop
vi. Plotting the error signal and VCO output against the input signal.

i. User Customization

The following example configures how many input samples, and correspondingly output samples, should be generated for the simulation. It also configures the sampling rate, VCO center frequency, and loop parameters that were discussed in (22), (12), and (13).

```
# ==================
# User Customization
# ==================

# Points
pts = 8000

# Sampling Frequency
fs = 44.1E3

# VCO Center Frequency
fc = 3E3

# Biquad Quality Factor
biq_qual = 1.0/np.sqrt(2.0)

# Biquad Cutoff Frequency
biq_fcut = 1000.0
```

```
# Damping Ratio Zeta
pll_zeta = 1.0/np.sqrt(2.0)

# Natural Loop Oscillating Frequency
pll_fnat = 170.0

# How much to phase step by
phase_step = np.pi/4.0
```

ii. Calculate Constants

This section calculates constants such as the DC value used to drive the VCO integrator in order to set the center frequency. The variable `nstep`, is the index that the 45º phase step occurs at.

```
# ====================
# Calculate Constants
# ====================

# VCO Center frequency
inc = 2.0*np.pi*fc

# Index when phase step will occur
nstep = int(pts/8)
```

iii. Initializing Signal Processing Blocks

This section instantiates the lowpass filters for error signal generation, the VCO phase integrator, and loop amplifier objects.

```
# ====================================
# Instantiate Signal Processing Blocks
# ====================================

# In-phase and quadrature lowpass filters
inph_lpf = LowPass(biq_fcut, fs, biq_qual)
quad_lpf = LowPass(biq_fcut, fs, biq_qual)

# Loop Amplifier object
amp = LoopAmplifier(pll_zeta, pll_fnat, fs)

# VCO Integrator object
vco = Integrator(fs)
```

iv. Initializing Arrays for Input and Output Data

In this section generates the 3 kHz input signal with 45º phase step, and creates output arrays to hold the VCO in-phase signal and error signal.

```
# ================================================
# Create Arrays for Holding Input and Output Data
# ================================================

# Create a times array
times = np.arange(pts)/fs

# Time in milliseconds, used for plotting
times_ms = times * 1000.0

# Input phase ramp signal
phase = 2.0*np.pi*fc*times

# Create a step signal with a max of phase_step (see user customization above)
step  = np.append(np.zeros(nstep), phase_step * np.ones(pts-nstep))

# Add it to the phase ramp
phase += step

# Create the input signal array
input_sig = np.cos(phase)

# Create an array to hold the output signal
outpu_sig = np.zeros(pts)

# Create an array to hold the error signal
error_sig = np.zeros(pts)
```

v. The Main Loop

The main loop implements Figure 14.

```
# =========
# Main Loop
# =========

for n in xrange(pts):
       # Generate VCO inphase and quadrature signals
       vco_inph = 2.0 * np.cos( vco.value() )
       vco_quad =-2.0 * np.sin( vco.value() )

       # Downconvert the input signal
       inph = inph_lpf.work( input_sig[n] * vco_inph )
       quad = quad_lpf.work( input_sig[n] * vco_quad )

       # Generate the error signal
       error = inph * quad

       loop_amp_out = amp.work(error)

       vco.work(inc+loop_amp_out)
       outpu_sig[n] = vco_inph
       error_sig[n] = error
```

vi. Plotting the error signal and VCO output against the input signal.

This section simply plots the output arrays. It is not necessary to understand this section to understand how to implement the Costas loop in software.

```
plt.figure(1)

# =====================
# Plot the error signal
# =====================

err_ax = plt.subplot(2,1,1)
err_ax.plot(times_ms, error_sig)
err_ax.set_xlim((times_ms[0], times_ms[-1]))
err_ax.set_ylim((-1, 1))
err_ax.set_xlabel('Time [ms]')
err_ax.set_ylabel('Amplitude [u]')
err_ax.set_title('Costas Loop Error Signal')
err_ax.grid(True)

# ======================================================
# Plot the VCO In Phase Signal Against the Input Signal
# ======================================================

# Since there is a lot of data and cycles, we are only interested in
# behavior around the phase step

# The first index we wish to start from is one natural closed loop oscillation
# cycle before the phase step
nstart = nstep - int(fs / pll_fnat)

# The last index is two natural closed loop oscillation cycles after the phase
# step
nstop  = nstep + int(fs / pll_fnat * 2)

# ztimes i.e. zoomed times, with 0 occuring at the phase step
ztimes = (times[nstart:nstop] - times[nstep]) * 1000.0

zom_inp = input_sig[nstart:nstop]
zom_out = outpu_sig[nstart:nstop]

zom_ax = plt.subplot(2,1,2)

zom_ax.plot(ztimes, zom_inp)
zom_ax.plot(ztimes, zom_out/2.0)
zom_ax.legend(['%.1f kHz Input'%(fc/1000.0), 'VCO In-Phase'], loc='lower left')

zom_ax.set_xlabel('Time After Phase Step [ms]')
zom_ax.set_ylabel('Amplitude [u]')
zom_ax.set_title('Input & VCO Quadrature Signals')
zom_ax.set_xlim((ztimes[0], ztimes[-1]))
zom_ax.set_ylim((-1.5, 1.5))

plt.grid(True)
plt.tight_layout()
plt.show()
```

The following is the output of `simplified_costa.py`:

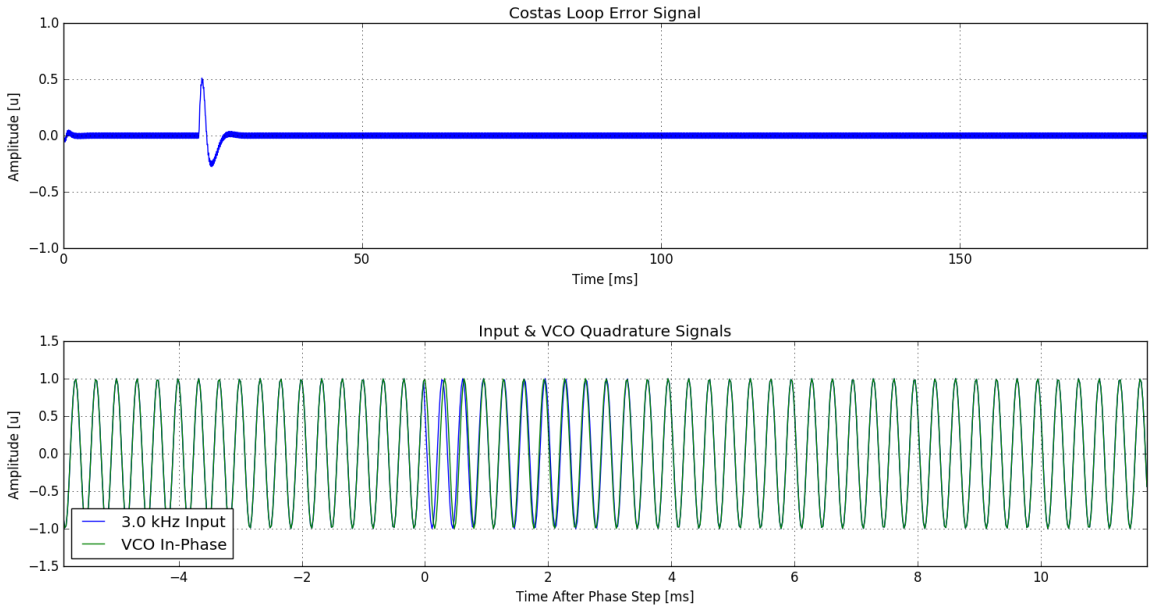


Figure 15. Expected Output of `simplified_costa.py`

---

10. Interactive Program to Help Determine Stable Loop Parameters

`costa_designer.py` interactively plots the closed, open, and transient behavior of (14) and (15) as a function of damping ratio ζ, natural loop frequency ω_n, lowpass filter passband frequency $f_p{}'$, and lowpass quality factor Q. In addition to plotting the linear model's frequency response, it plots the transient response of the Costas loop, which are shown in Figure 16.

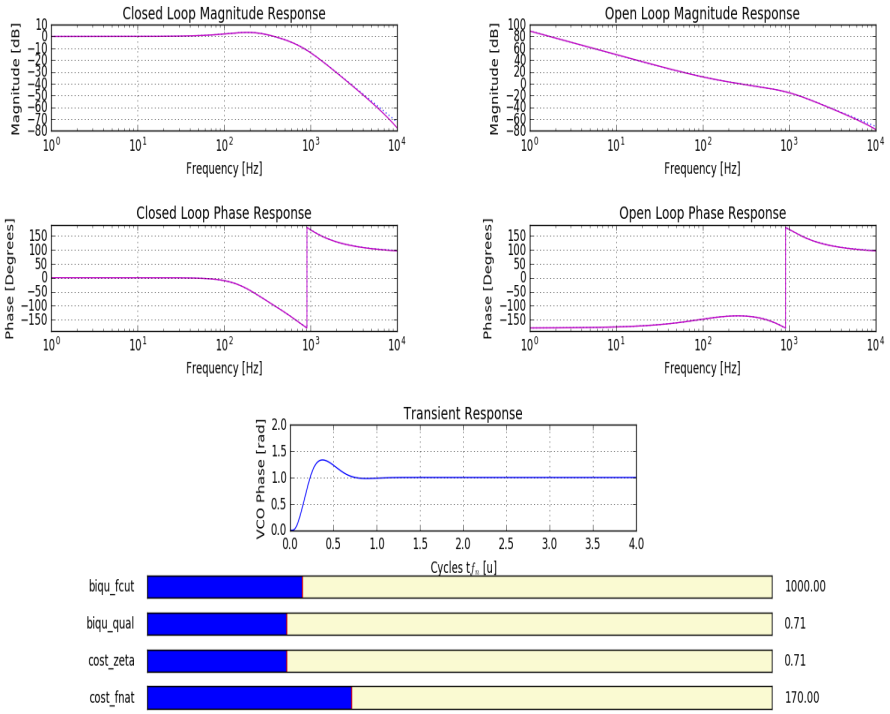


Figure 16. `costa_designer.py`: Critically Damped Tuning

Figure 17 (below), depicts the Costas loop model used to generate the transient response in `costa_designer.py`.

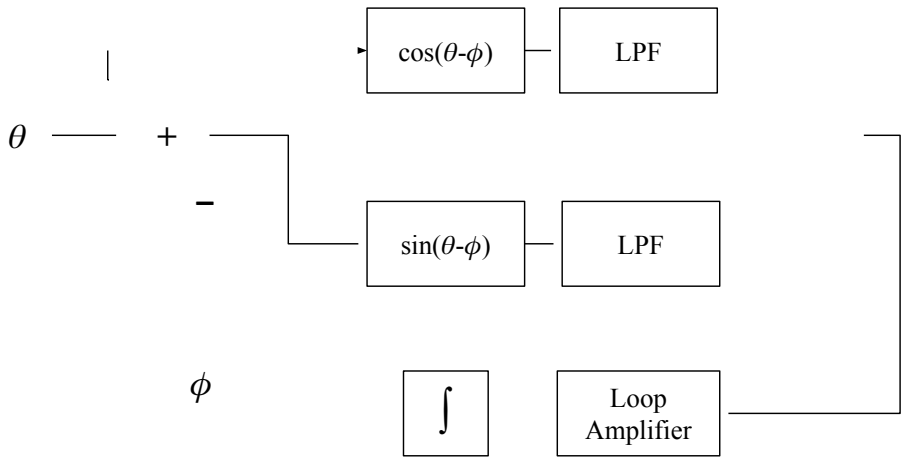


Figure 17. Baseband Only Costas Loop

Since only the transient phase behavior of the Costas loop is of interest, the model used to determine the step response omits {1}, {2a}, {2b}, {3a}, {3b}, {6a}, and {6b}.

After manual tuning, the following parameters were chosen to achieve the step response shown in Figure 16.

Table 3. Loop Configuration

Biquad LPF Cutoff Frequency (f_c)	1 kHz
Biquad LPF Quality Factor (Q)	0.71 (Rounded up from $1/\sqrt{2}$)
Loop Amplifier Zeta (ζ)	0.71 (Rounded up from $1/\sqrt{2}$)
Loop Amplifier Natural Frequency (ω_n)	170 Hz

Verification of `costa_designer.py`

The purpose of this section, is to validate the accuracy of the Costas loop frequency domain, and transient step response model. In order to do so, `costa_designer.py` was used to determine loop parameters ζ, ω_n, f_p', and Q that yield overdamped, underdamped, and unstable step responses. The parameters were then run in costa.py, to see if its response to a phase stepped input signal, followed the predictions made by `costa_designer.py`.

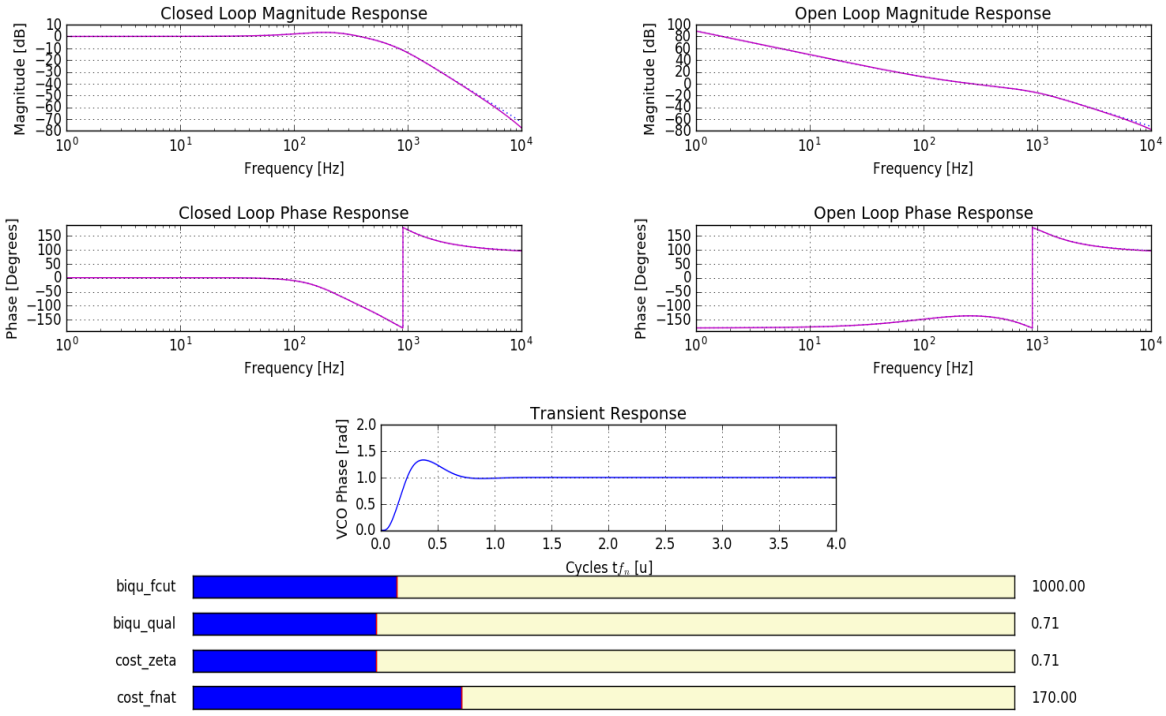


Figure 18. `costa_designer.py`: Critically Damped

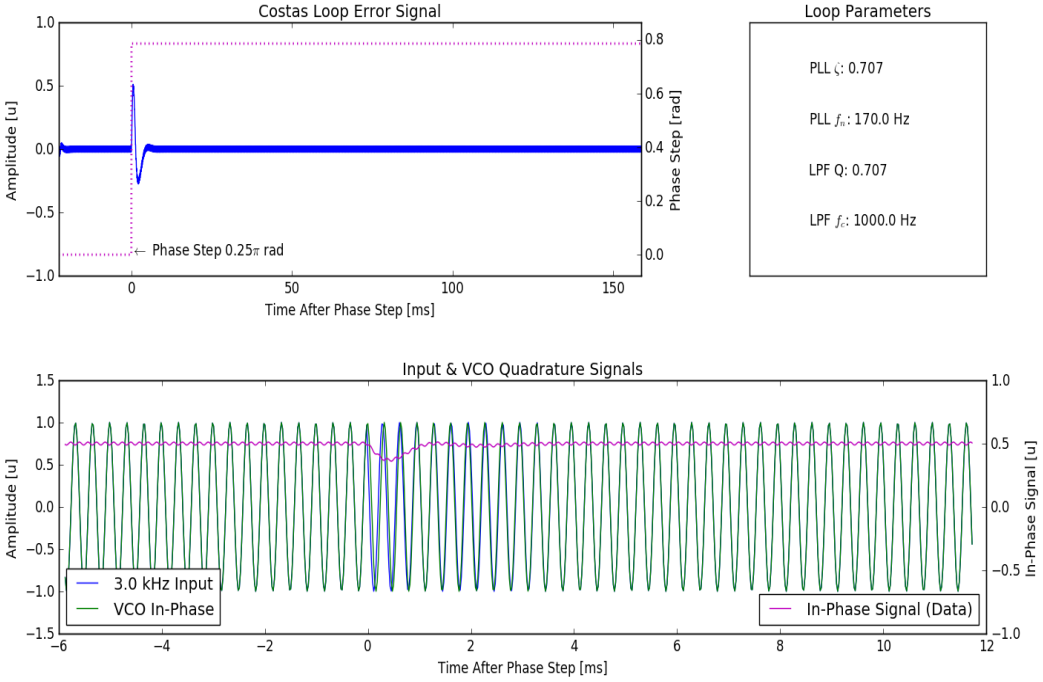


Figure 19. `costa.py`: Critically Damped

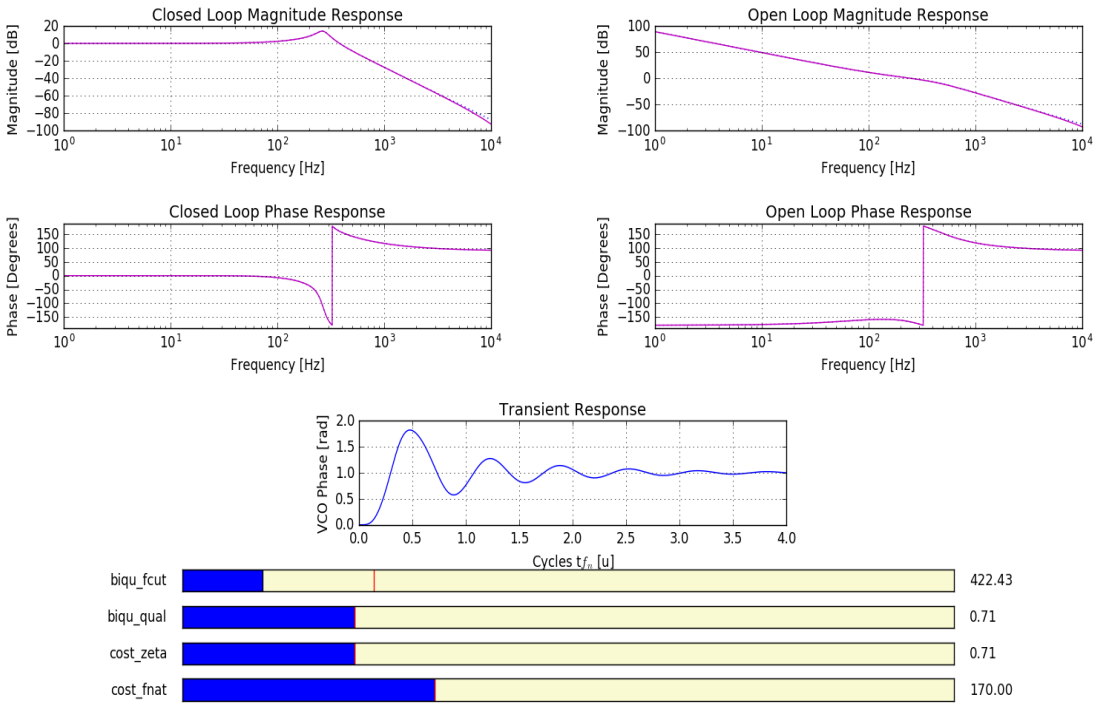


Figure 20. `costa_designer.py`: Underdamped

The closed loop magnitude response has gain peaking near the natural loop frequency.

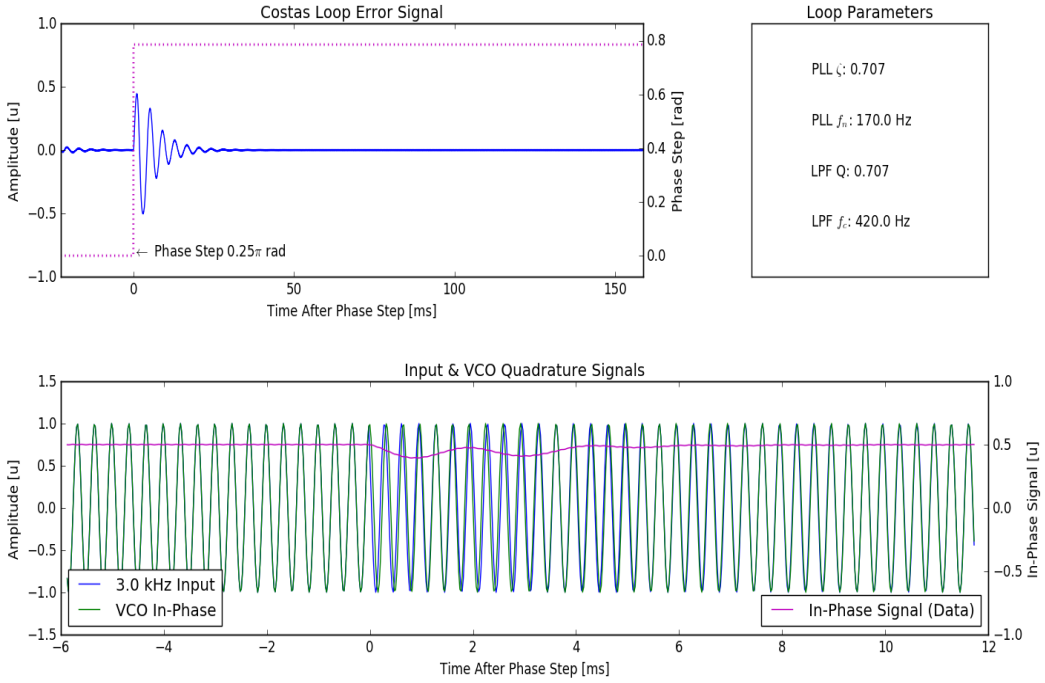


Figure 21. `costa.py`: Underdamped

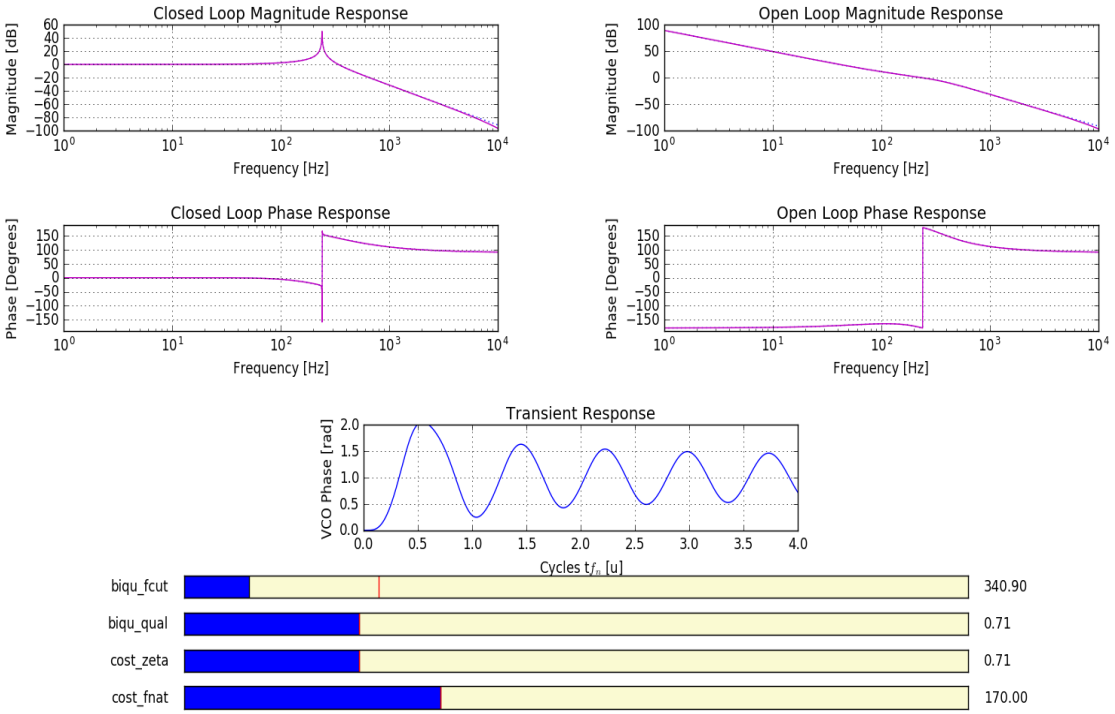


Figure 22. `costa_designer.py`: Unstable

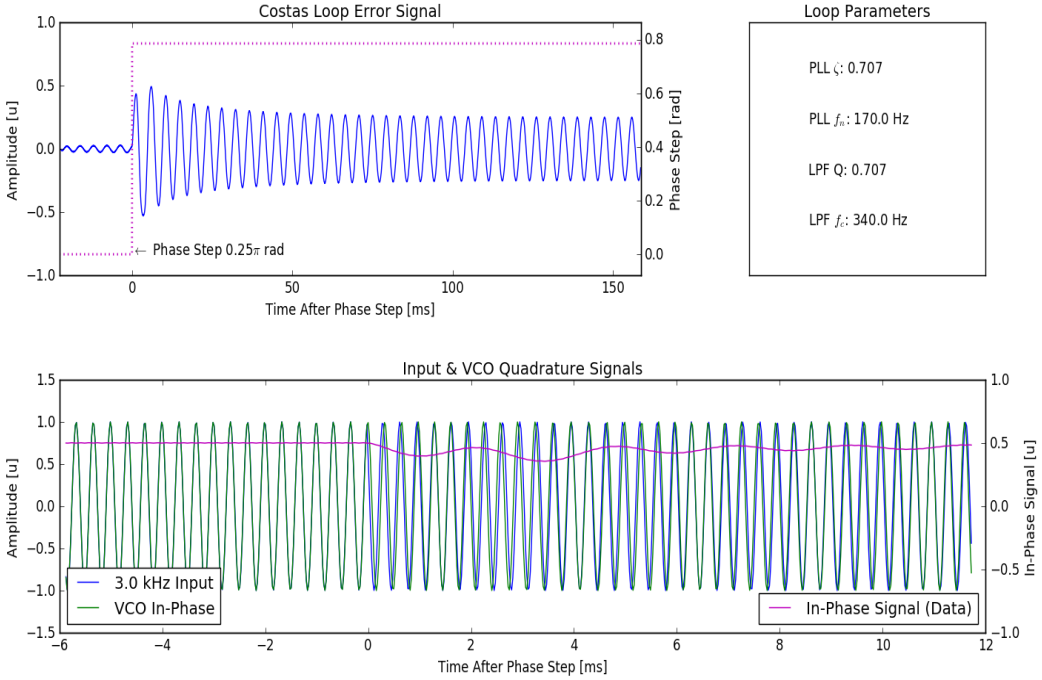


Figure 23. `costa.py`: Unstable

The VCO in-phase signal {6a} warbles around the input signal {1} before settling.

11. Other Test Inputs

The critically damped Costas loop (configured with the Table 3 parameters), was subjected to a signal with a 180º phase step – to simulate a BPSK signal. In addition, to a phase step, it was also tested with an input sinusoid 250 Hz higher than its VCO's center frequency.

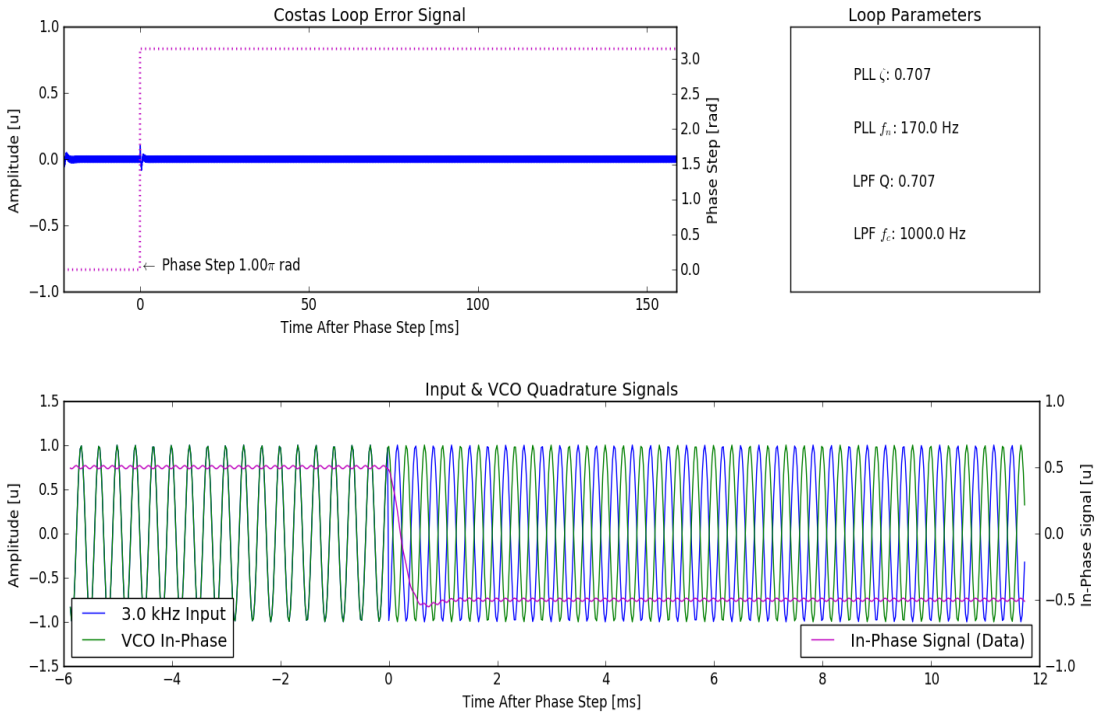


Figure 24. `costa.py`: Response to BPSK Signal

Virtually no error signal was produced by the error detector for a phase shift of 180º. As a result, no phase adjustment is made to the VCO, and the in-phase signal {4a} can be used for further communications processing.

The response to a 3.25 kHz input signal – 250 Hz higher than its VCO's center frequency is shown in Figure 25 (below).

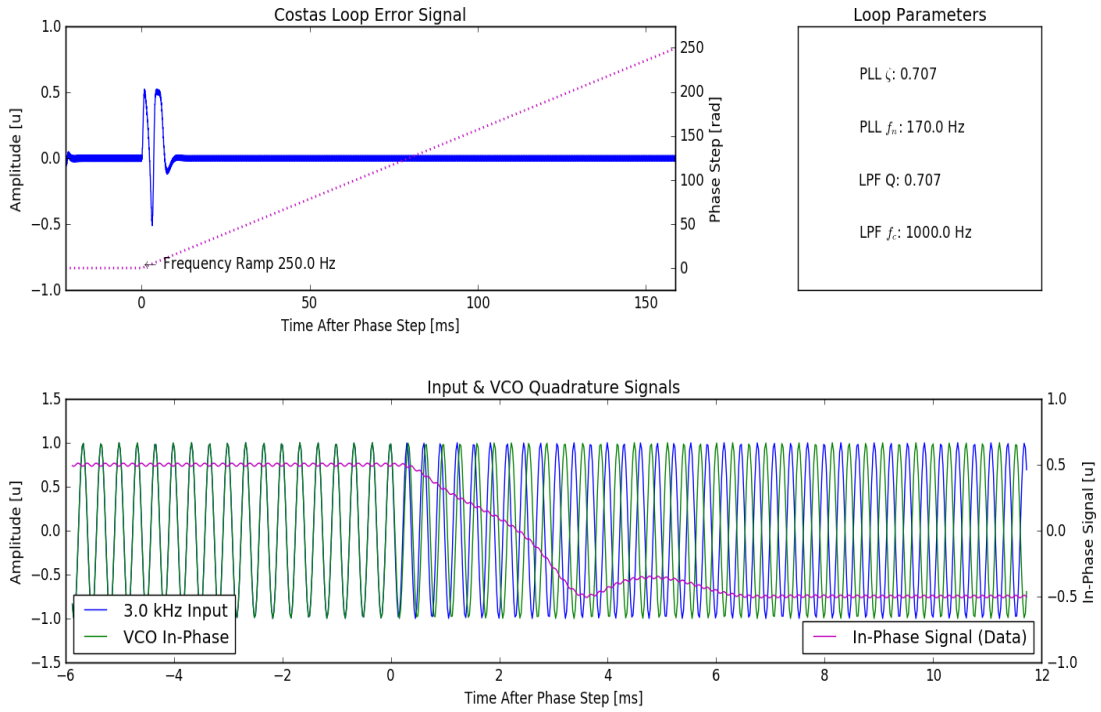


Figure 25. `costa.py`: Response to 250 Hz Phase Ramp

The response shown in Figure 25 is a good example of the Costas loop indiscriminately locking onto either 0º or 180º relative to the input signal. In Figure 25, the 250 Hz frequency step begins at 0 ms, causing the input signal to compress, which causes the VCO to start lagging the input. The error detector signal becomes positive, driving the loop amplifier to accelerate the VCO phase. However, the initial correction is not enough to close the widening phase lag between the input signal and the VCO. The VCO lags even further, slipping to more than 90º lagging. The VCO becomes closer to achieving 180º phase lock instead. From the error detector's perspective, the Costas loop now leads the input signal, which is why the error signal becomes negative in Figure 25. The phase lead is short lived because the input signal's phase is still increasing at a greater rate than the Costas loop's. In addition, the error detector, because it is being driven by negatively by the loop amplifier, hinders VCO phase acceleration even more. With everything going against it, the VCO quickly starts lagging 180º, at which point, it hits the gas and this time, is able to keep up with the input signal.

Underdamped Costas Loop Response to 250 Hz Frequency Ramp

The Costas loop, when configured as an underdamped system (Figure 21), cannot track a 250 Hz phase ramp.

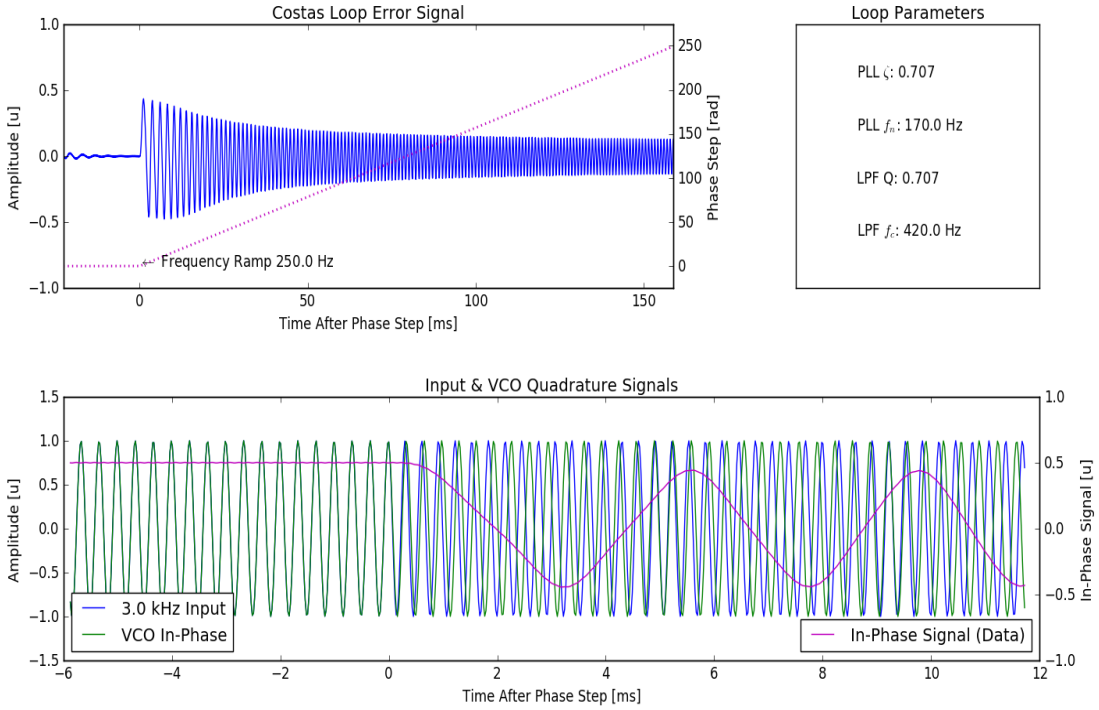


Figure 26. `costa.py`: Underdamped Response to 250 Hz Phase Ramp

The Costas loop, when configured to have an underdamped response, could not track the same 250 Hz phase response that the critically damped configuration could. Figure 26 demonstrates the importance of loop tuning in achieving optimal acquisition bandwidth.

---

12. Design Tradeoffs

The dominant variable in determining the closed loop step response is lowpass filter passband frequency $f_p{}'$. Decreasing $f_p{}'$ leads to a decrease in loop phase margin. However, decreasing $f_p{}'$ is advantageous in minimizing error detector ripple caused by double frequency components {3a} and {3b}. Therefore, an increase in error detector ripple is exchanged for enhanced loop stability. In order to illustrate error detector ripple, the FFT of the error signals of the critically damped ($f_p{}'$=1kHz) and underdamped ($f_p{}'$= 420 Hz) Costas loop are depicted in Figure 27 and 28.

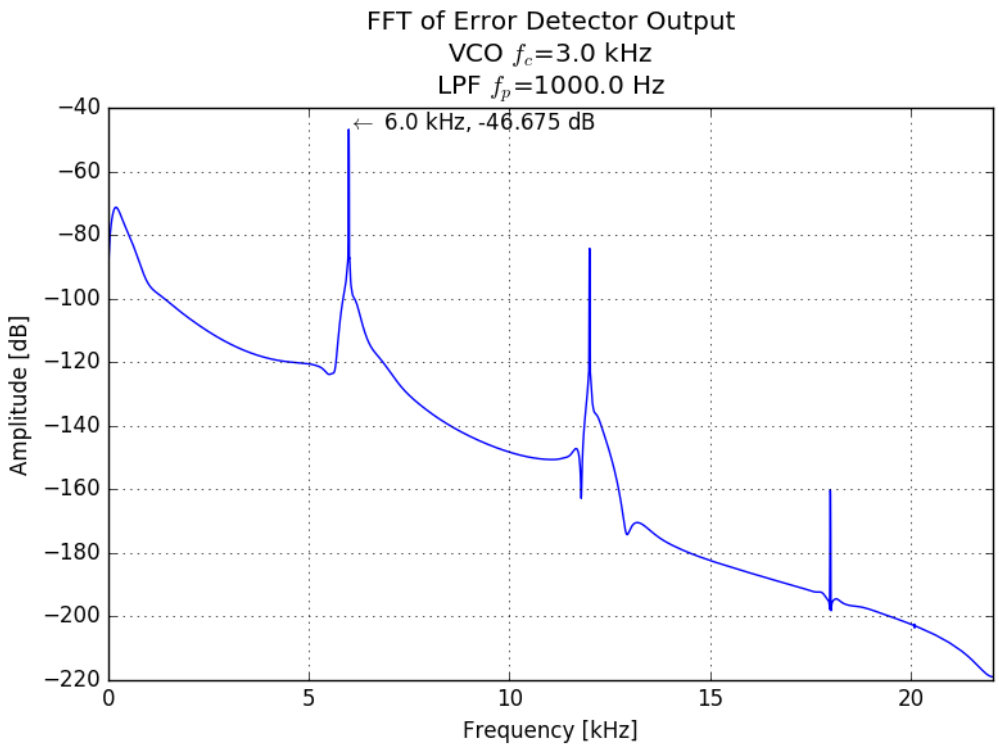


Figure 27. `costa.py`: Critically Damped Error Detector Spectrum

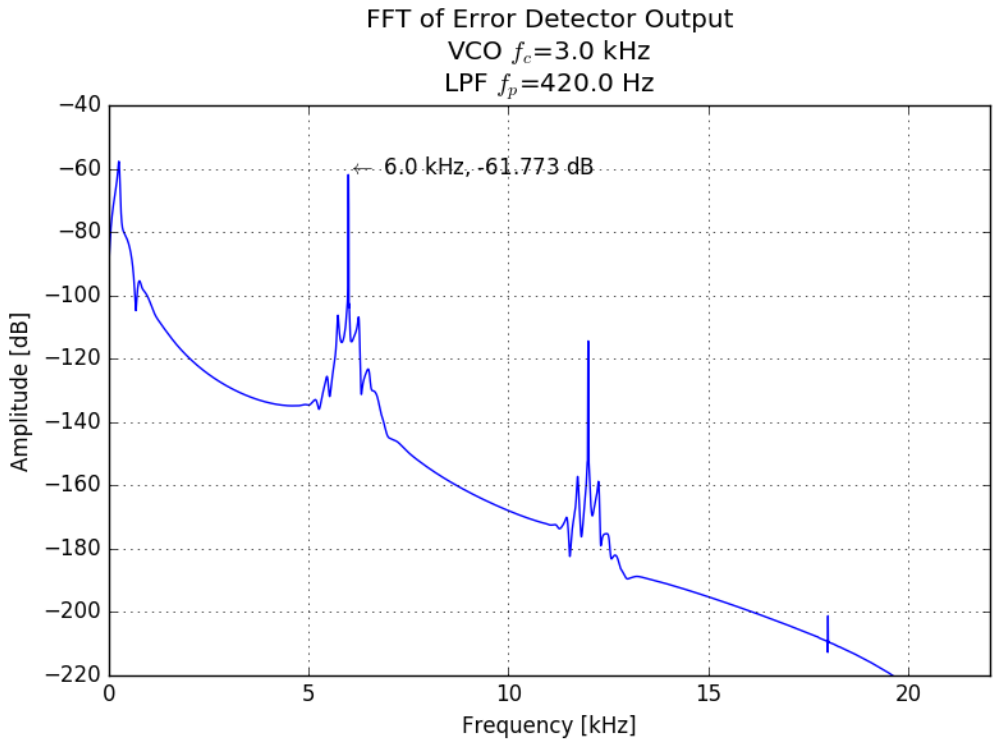


Figure 28. `costa.py`: Underdamped Error Detector Spectrum

Lowering lowpass filter passband frequency decreased the double frequency component by 15 dB, while making the loop more unstable, and decreasing acquisition bandwidth.

---

13. Conclusion

It is possible to convert the analog Costas loop into a digital software algorithm by using the bilinear transform to convert transfer functions expressed in the Laplace domain to the Z-domain. Lowpass filter cutoff frequency $f_p{}'$ is the most dominant variable in determining the stability of the Costas loop. As such, it cannot be ignored when choosing loop parameters. Tuning the Costas loop to be critically damped enabled it to have the widest acquisition bandwidth. Phase margin can be increased by increasing $f_p{}'$, at the expense of increased double frequency ripple produced by the Costas loop mixers. `costa.py` is suitable for BPSK demodulation within audio frequencies. Other ideas for its application are demodulation of BPSK data received by computer microphone. Because the algorithm is implemented in Python, `costa.py` can be used as an Undergraduate lab project to demonstrate digital signal processing and communications principles.

14. References

[1] R. Thamvichai, T. Bose and W. Tranter, *Basic Simulation Models of Phase Tracking Devices Using MATLAB (Synthesis Lectures on Communications)*, 1st ed. Morgan & Claypool Publishers, pp. 21-23.

[2] R. Thamvichai, T. Bose and W. Tranter, *Basic Simulation Models of Phase Tracking Devices Using MATLAB (Synthesis Lectures on Communications)*, 1st ed. Morgan & Claypool Publishers, pp. 97-101.

[3] A. Ambardar, *Analog and digital signal processing*, 2nd ed. Pacific Grove, Calif.: Brooks/Cole, 1999, pp. 637-639.

[4] A. Ambardar, *Analog and digital signal processing*, 2nd ed. Pacific Grove, Calif.: Brooks/Cole, 1999, p. 379.

[5] A. Ambardar, *Analog and digital signal processing*, 2nd ed. Pacific Grove, Calif.: Brooks/Cole, 1999, pp. 691-693.

Program Listings

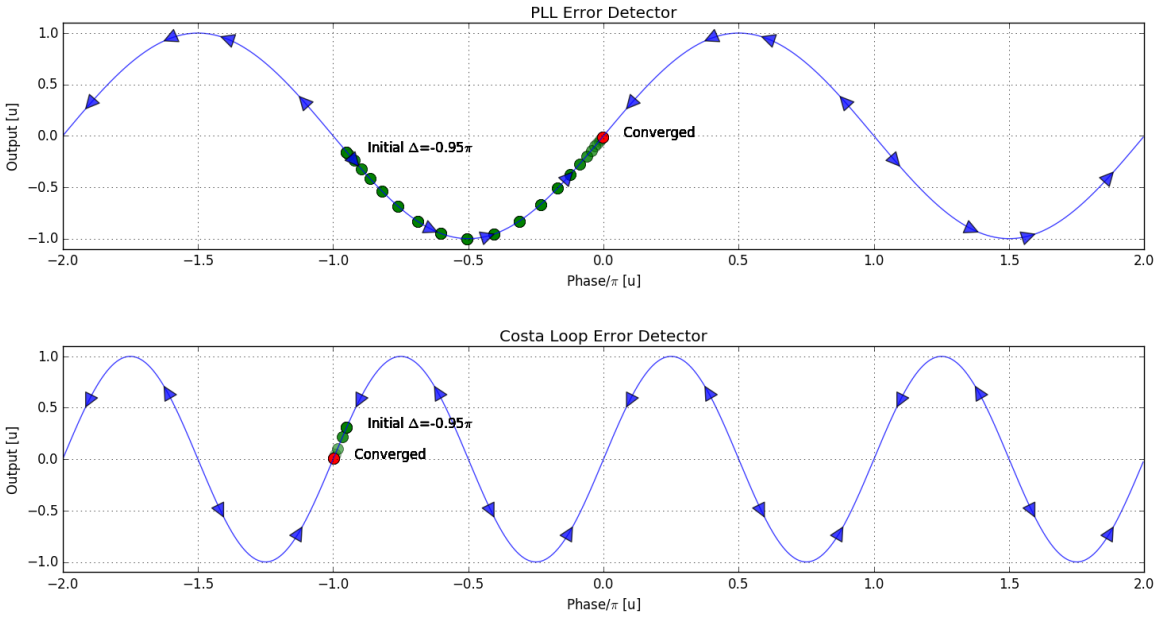
---

error_detector.py

```
# FILENAME: error_detector.py
#
# DESCRIPTION: PLL and Costas loop error detector analysis
#
# AUTHOR: Robby J. Tong          DATE: 5/14/2017
# EMAIL: robbytong@gmail.com
#
# CHANGES:
#

import matplotlib
matplotlib.use('TkAgg')
import matplotlib.pyplot as plt
import numpy as np

def costa_error(x):
	return np.sin(2.0 * pi * x)

def pll_error(x):
	return np.sin(pi * x)

def plot_phase_plane(ax, error, initial_phase=0.25):
	global pi
	pts = 100
	pi = np.pi
	segments = [-2.0, -1.5, -1.0, -0.5, +0.0, +0.5, +1.0, +1.5, +2.0]

	for k in range(1, len(segments)):
		print k

		def add_arrow(pnt):
			def phase(x):
				diff = (segments[k] - segments[k-1])
				return float(x)/pts * diff + segments[k-1]

			def contour(x):
				return error(phase(x))

			# default is to the left
			add = -1

			# if the error is positive, then the tail should be to our right
			if contour(pnt) >= 0:
				add = 1

			#plot arrow at the middle of the segment
			ax.annotate('', xy=(phase(pnt-add), contour(pnt-add)),
xytext=(phase(pnt+add), contour(pnt+add)), arrowprops=dict(alpha=0.75, facecolor='blue'))

		add_arrow(pts / 32 * 6)
		add_arrow(-pts / 32 * 6)

		def find_valley(initial_phase):
			vco_phase = 0.0
			gain = 0.05
```

```
                        diff = vco_phase - initial_phase
                        err = error(diff)
                        ax.plot(diff, err, 'og', markersize=10)
                        ax.text(diff, err, '     Initial $\Delta$=%.2f$\pi$'%diff)

                        cycles = 2
                        while np.power(err, 2.0) > 1E-4:
                                vco_phase -= err * gain
                                diff = vco_phase - initial_phase
                                err = error(diff)
                                cycles += 1
                                if cycles >= 2:
                                        ax.plot(diff, err, 'og', alpha=np.abs(err),
markersize=10)
                                        cycles = 0

                        ax.plot(diff, err, 'or', markersize=10)
                        ax.text(diff, err, '     Converged')

                find_valley(initial_phase)

                ax.set_ylim((-1.1, 1.1))

        tot_pts = pts * len(segments)
        phases = (np.arange(tot_pts) / float(tot_pts) - 0.5)  * (segments[-1] - segments[0])
        contour = error(phases)
        ax.plot(phases, contour, alpha=0.75, color='b')


ax1 = plt.subplot(211)
ax2 = plt.subplot(212)

initial_phase = 0.95

# PLL Error
plot_phase_plane(ax1, pll_error, initial_phase)
ax1.set_title('PLL Error Detector')
ax1.set_xlabel('Phase/$\pi$ [u]')
ax1.set_ylabel('Output [u]')
ax1.grid(True)

# Costa Error
plot_phase_plane(ax2, costa_error, initial_phase)
ax2.set_title('Costa Loop Error Detector')
ax2.set_xlabel('Phase/$\pi$ [u]')
ax2.set_ylabel('Output [u]')
ax2.grid(True)

plt.tight_layout()
plt.show()
```

Sample Output

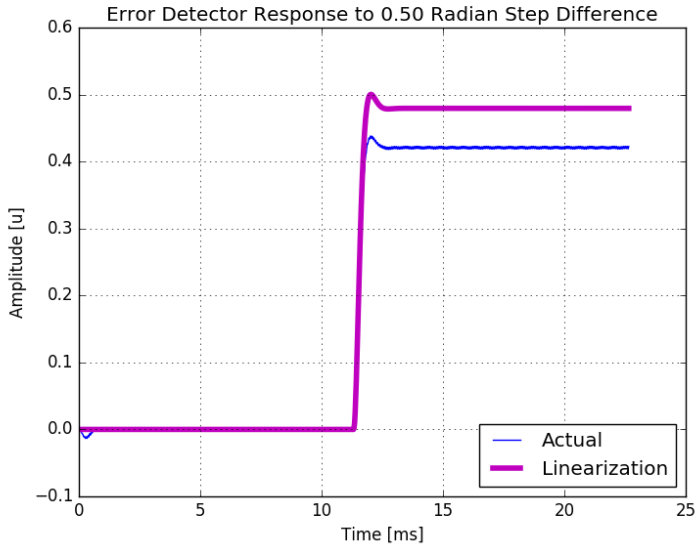


linear_error_detector.py

```
# FILENAME: linear_error_detector.py
#
# DESCRIPTION: Compare the error detector output of the Costas loop to its
# linear approximation.
#
# AUTHOR: Robby J. Tong          DATE: 5/7/2017
# EMAIL: robbytong@gmail.com
#
# CHANGES:
#

import matplotlib
matplotlib.use('TkAgg')
import matplotlib.pyplot as plt
from Integrator import Integrator
from LowPass import LowPass
import numpy as np
import sys

class LinearLoop(object):
	def __init__(self, fpass, fnatural, fsample):
		pll_zeta = 1.0/np.sqrt(2.0)
		self.G = 4.0*np.pi*pll_zeta*fnatural
		self.a = fnatural*np.pi/pll_zeta
		self.vco1 = Integrator(fs)
		self.amp = Integrator(fs)
		self.lpf1 = LowPass(fpass, fsample)
		self.lpf2 = LowPass(fpass, fsample)

	def work(self, x):
		s1 = x-self.vco1.value()
		s2 = self.lpf1.work(s1)
		return s2

class CostasLoop(object):
	def __init__(self, fpass, fnatural, fcenter, fsample):
		pll_zeta = 1.0/np.sqrt(2.0)
		self.G = 4.0*np.pi*pll_zeta*fnatural
		self.a = fnatural*np.pi/pll_zeta
		self.inc = 2.0*np.pi*fc
		self.vco1 = Integrator(fs)
		self.vco2 = Integrator(fs)
		self.amp = Integrator(fs)
		self.lpf1 = LowPass(fpass, fsample)
		self.lpf2 = LowPass(fpass, fsample)

	def work(self, x):
		phase = self.vco1.value() + self.vco2.value()
		vco_inp = np.cos(phase)
		vco_qup =-np.sin(phase)
		inp = self.lpf1.work(2.0 * x * vco_inp)
		qup = self.lpf2.work(2.0 * x * vco_qup)
		s2 = inp * qup
		self.vco1.work(self.inc)
		return (s2, vco_qup)

step = 0.5

args = sys.argv[1:]
```

```
if len(args) == 1:
       step = float(args[0])

pts = 1000
fs = 44.1E3
fc = 7E3
nstep = int(pts/2)
times = np.arange(pts)/fs
steps = step * np.append(np.zeros(nstep), np.ones(pts-nstep))
phase = 2.0*np.pi*fc*times + steps

sig = np.cos(phase)
out = np.zeros(pts)
wav = np.zeros(pts)
lin = np.zeros(pts)

real_costa = CostasLoop(1000.0, 170.0, fc, fs)
line_costa = LinearLoop(1000.0, 170.0, fs)

for n in xrange(pts):
       error1, quadrature = real_costa.work(sig[n])
       error2 = line_costa.work(steps[n])
       lin[n] = error2
       wav[n] = quadrature
       out[n] = error1

plt.plot(times*1000.0, out)
plt.plot(times*1000.0, lin, linewidth=4, color='m')
plt.gca().set_title('Error Detector Response to %.2f Radian Step Difference'%step)
plt.gca().set_ylabel('Amplitude [u]')
plt.gca().set_xlabel('Time [ms]')
plt.grid(True)
plt.legend(['Actual', 'Linearization'], loc='lower right')
plt.show()
```

Sample Output



LowPass.py

```
# FILENAME: LowPass.py
#
# DESCRIPTION: Biquadratic lowpass filter implemented using direct form II.
#
# AUTHOR: Robby J. Tong          DATE: 5/7/2017
# EMAIL: robbytong@gmail.com
#
# CHANGES:
#

import matplotlib
matplotlib.use('TkAgg')
import matplotlib.pyplot as plt
import numpy as np
from numpy.fft import fft
pi = np.pi

## Digital Filter LowPass Filter Class
#
# @param fp -3 dB pass band frequency
# @Q quality factor
# @fs sampling rate
class LowPass(object):
       def __init__(self, fp, fs=44.1E3, Q=0.7071):
              wp = 2.0 * pi * fp
              k = 2.0 * fs
              k_2  = k * k
              wp_2 = wp * wp
              A = (k_2 + wp * k / Q + wp_2)
              B = (  2.0 * (wp_2 - k_2)   )
              C = (wp_2 - wp * k / Q + k_2)
              self.a0 = 1.0
              self.a1 = -B/A
              self.a2 = -C/A
              self.z1 = 0
              self.z2 = 0
              self.gain = wp_2 / A

    ## Read in a new sample
    # @param self - see Python documentation for class methods
    # @param x input sample
       def work(self, x):
              w1 = x + self.a1 * self.z1 + self.a2 * self.z2

              y = w1 + 2.0 * self.z1 + self.z2

              self.z2 = self.z1
              self.z1 = w1

              return y * self.gain
```

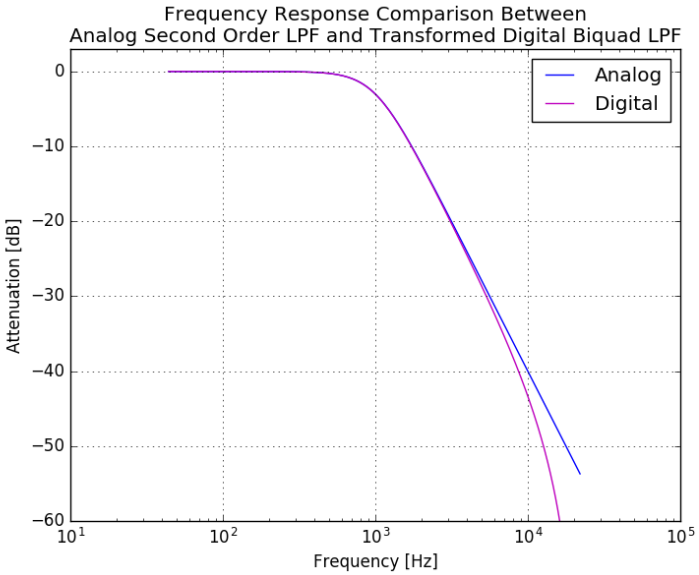
No Sample Output

lpf_frequency.py

```
# FILENAME: lpf_frequency.py
#
# DESCRIPTION: Plot the frequency response of Biquadratic filter against
# its analog second order form to verify bilinear transform conversion was
# successful.
#
# AUTHOR: Robby J. Tong          DATE: 5/7/2017
# EMAIL: robbytong@gmail.com
#
# CHANGES:
#

import matplotlib
matplotlib.use('TkAgg')
import matplotlib.pyplot as plt
import numpy as np
pi = np.pi

fs = 44.1E3
fp = 1E3
pts = 500
fny = fs / 2.0
wp = 2.0 * pi * fp

freqs = np.arange(pts) / float(pts) * fny

def H_lp(s, wp, Q):
	num = wp * wp
	den = s * s + wp / Q * s + wp * wp
	return num / den

def mag(x):
	return 20.0 * np.log10( np.abs(x) )

def G_lp(z, wp, Q, fs):
	z_2 = np.power(z, -2.0)
	z_1 = np.power(z, -1.0)
	k = 2.0 * fs
	k_2 = k * k
	wp_2 = wp * wp

	A = k_2 + wp * k / Q + wp_2
	B = 2.0 * wp_2 - 2.0 * k_2
	C = wp_2 - wp * k / Q + k_2
	print 'A/A: %f'%(A/A)
	print 'B/A: %f'%(B/A)
	print 'C/A: %f'%(C/A)
	print 'wp^2/A: %f'%(wp_2/A)

	num = wp_2 * (1.0 + 2.0 * z_1 + z_2)
	den = A + B * z_1 + C * z_2

	return num / den

def warp(fp, fs):
	return 2.0 * fs * np.tan(0.5 * fp / fs)

digital = G_lp(np.exp(2.0j * pi * freqs / fs), 2.0 * pi * warp(fp, fs), 0.7071, fs)
```

```
analog = H_lp(2j * pi * freqs, wp, 0.7071)

plt.semilogx(freqs, mag(analog))
plt.semilogx(freqs, mag(digital), color='m')
plt.legend(['Analog', 'Digital'])
plt.gca().set_ylim((-60, 3))
plt.gca().set_ylabel('Attenuation [dB]')
plt.gca().set_xlabel('Frequency [Hz]')
plt.gca().set_title('Frequency Response Comparison Between\nAnalog Second Order LPF and
Transformed Digital Biquad LPF')
plt.grid(True)
plt.show()
```

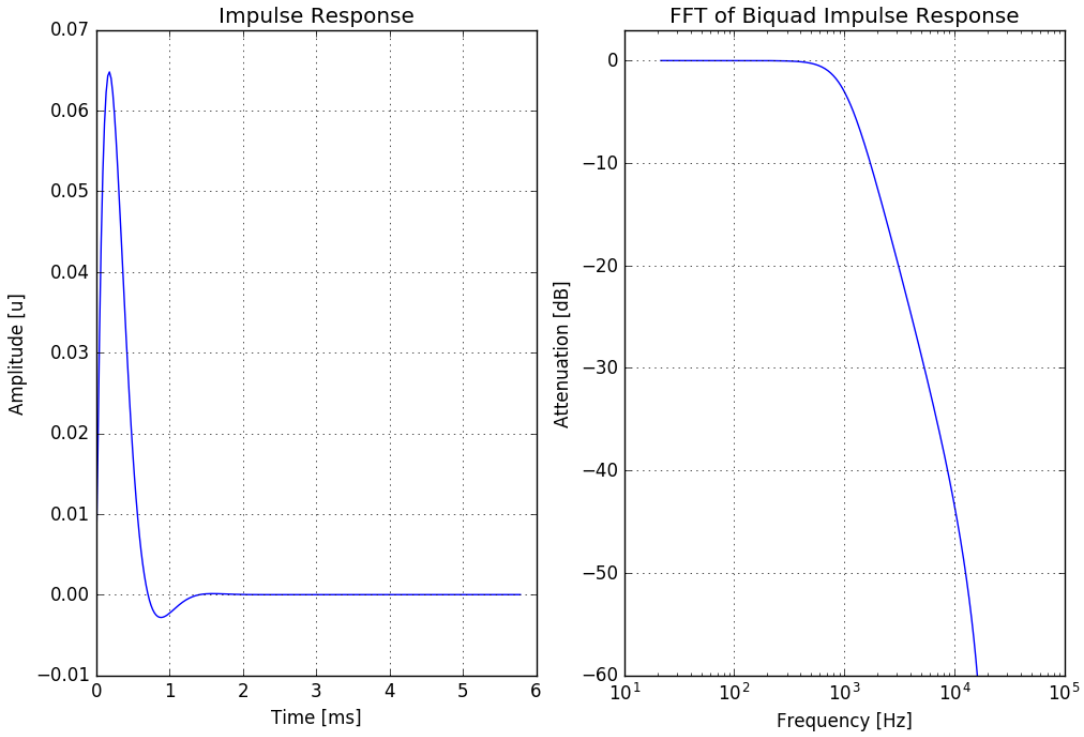
Sample Output



lpf_impulse.py

```
# FILENAME: lpf_impulse.py
#
# DESCRIPTION: Take the FFT of the impulse response produced
# biquad lowpass filter (Lowpass.py) to verify its frequency
# response.
#
# AUTHOR: Robby J. Tong          DATE: 5/7/2017
# EMAIL: robbytong@gmail.com
#
# CHANGES:
#

import matplotlib
matplotlib.use('TkAgg')
import matplotlib.pyplot as plt
import numpy as np
from numpy.fft import fft
from LowPass import LowPass
pi = np.pi

fs = 44.1E3
fp = 1E3
pts = 2048

lpf = LowPass(1000.0)

out = np.zeros(pts)
inp = np.zeros(pts)
inp[0] = 1.0

for k in xrange(pts):
       out[k] = lpf.work(inp[k])

resp = 20.0 * np.log10( np.abs(fft(out)) )
freq = np.arange(pts) / float(pts) * fs

plt.subplot(121)
plt.plot(np.arange(pts/8)/fs*1000.0, out[:pts/8])
plt.gca().set_title('Impulse Response')
plt.gca().set_xlabel('Time [ms]')
plt.gca().set_ylabel('Amplitude [u]')
plt.grid(True)

plt.subplot(122)
plt.semilogx(freq[:pts/2], resp[:pts/2])
plt.gca().set_ylim((-60, 3))
plt.gca().set_title('FFT of Biquad Impulse Response')
plt.gca().set_xlabel('Frequency [Hz]')
plt.gca().set_ylabel('Attenuation [dB]')
plt.grid(True)
plt.show()
```

Sample Output



Integrator.py

```
# FILENAME: Integrator.py
#
# DESCRIPTION: Integrator derived using bilinear transform.
#
# AUTHOR: Robby J. Tong          DATE: 5/7/2017
# EMAIL: robbytong@gmail.com
#
# CHANGES:
#

import numpy as np

class Integrator(object):
	def __init__(self, fs):
		self.lasty = 0
		self.twofs = 2.0*fs

	def work(self, value):
		y = value+self.lasty
		self.lasty = value+y
		return y/self.twofs

	def value(self):
		return self.lasty/self.twofs
```

No Sample Output

costa_designer.py

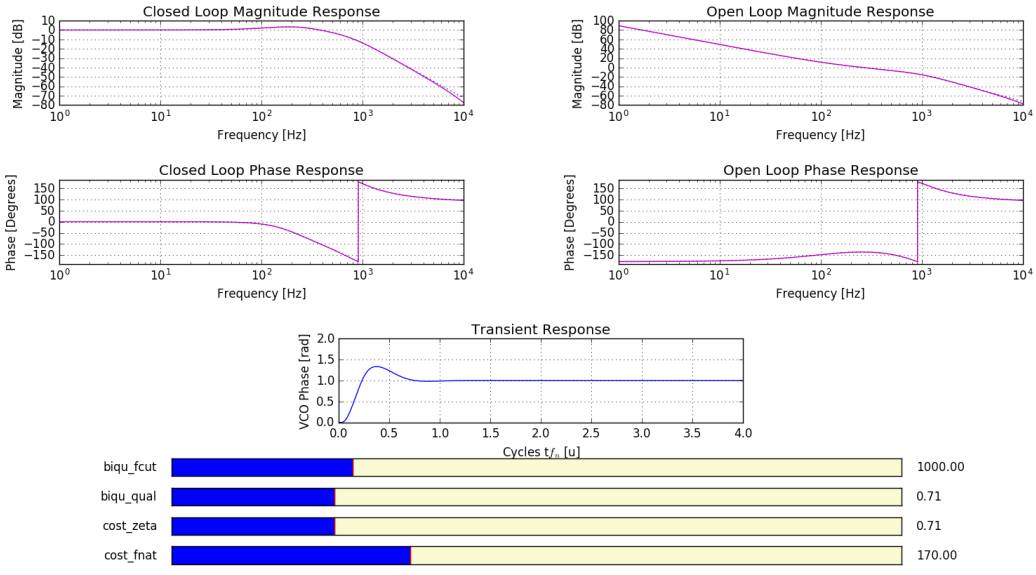
```
# FILENAME: costa_designer.py
#
# DESCRIPTION: Interactive plots the closed and open frequency responses of
# linearized Costas loop. Also plots the transient response, see step.py
# User can adjust:
# - 'pll_fnat' The natural frequency of the loop
# - 'pll_zeta' The damping ratio of the loop
# - 'biq_qual' The quality factor of the biquad lowpass filters
# - 'biq_fcut' The cutoff frequency of the biquad lowpass filters
#
# AUTHOR: Robby J. Tong          DATE: 5/7/2017
# EMAIL: robbytong@gmail.com
#
# CHANGES:
#

import matplotlib
matplotlib.use('TkAgg')
from matplotlib.widgets import Slider, Button, RadioButtons
from matplotlib.ticker import MultipleLocator, FormatStrFormatter
import matplotlib.pyplot as plt
import numpy as np
import step
import pdb

def biquad(s, q, fc):
       wc = 2.0*np.pi*fc
       return (wc**2.0)/(s**2.0+wc/q*s+wc**2.0)

# loop amp to achieve
# perfect second order loop
def loop_amp(s, fnat, zeta):
       gain = 4.0*np.pi*zeta*fnat
       a = np.pi*fnat/zeta
       return gain*(1.0+a/s)

def open_loop(s, loop_fnat, loop_zeta, biquad_qual, biquad_fcut):
       lpf = biquad(s, biquad_qual, biquad_fcut)
       amp = loop_amp(s, loop_fnat, loop_zeta)
       tot = lpf * amp / s
       return tot

def pll(s, loop_fnat, loop_zeta, biquad_qual, biquad_fcut):
       tot = open_loop(s, loop_fnat, loop_zeta, biquad_qual, biquad_fcut)
       return tot/(1.0+tot)

def bilinear_transform(fs, freqs):
       inv_z = np.exp(-2j*np.pi*freqs/fs)
       s = 2.0*fs*(1.0-inv_z)/(1.0+inv_z)
       return s

def plot_mag(info, laplace_resp, digital_resp, txt):
       continu = 20.0*np.log10(np.abs(laplace_resp))
       digital = 20.0*np.log10(np.abs(digital_resp))

       if info['handles'] is None:
              handles = {}
              ax = info['ax']
              handles['laplace'], = ax.semilogx(freqs, continu, ':', label='S-Domain')
```

```
                handles['digital'], = ax.semilogx(freqs, digital, color='m',
label='Bilinear')
                ax.set_title('%s Loop Magnitude Response'%txt)
                ax.set_xlabel('Frequency [Hz]')
                ax.set_ylabel('Magnitude [dB]')
                ax.set_xlim((0,freqs[-1]))
                ax.grid(True)
                #ax.set_ylim((-40, 40))
                info['handles'] = handles
        else:
                info['handles']['laplace'].set_ydata(continu)
                info['handles']['digital'].set_ydata(digital)
                ax = info['ax']
                ax.relim()
                ax.autoscale_view()
        #ax.legend(handles=[laplace, digital], loc='upper left', bbox_to_anchor=(1,1))

def plot_ang(info, laplace_resp, digital_resp, txt):
        continu = np.angle(laplace_resp)*180.0/np.pi
        digital = np.angle(digital_resp)*180.0/np.pi
        if info['handles'] is None:
                handles = {}
                ax = info['ax']
                handles['laplace'], = ax.semilogx(freqs, continu, ':', label = 'S-Domain')
                handles['digital'], = ax.semilogx(freqs, digital, color='m', label =
'Bilinear')
                ax.set_title('%s Loop Phase Response'%txt)
                ax.set_xlabel('Frequency [Hz]')
                ax.set_ylabel('Phase [Degrees]')
                ax.set_xlim((0,freqs[-1]))
                ax.set_ylim((-190, 190))
                ax.grid(True)
                info['handles'] = handles
        else:
                info['handles']['laplace'].set_ydata(continu)
                info['handles']['digital'].set_ydata(digital)
                ax = info['ax']
                ax.relim()
                ax.autoscale_view()

def plot_resp(pll_fnat, pll_zeta, biq_qual, biq_fcut):
        digital = {}
        laplace = {}

        digital['open'] = open_loop(z2s, pll_fnat, pll_zeta, biq_qual, biq_fcut)
        laplace['open'] = open_loop(  s, pll_fnat, pll_zeta, biq_qual, biq_fcut)

        digital['closed'] = pll(z2s, pll_fnat, pll_zeta, biq_qual, biq_fcut)
        laplace['closed'] = pll(  s, pll_fnat, pll_zeta, biq_qual, biq_fcut)

        plot_mag(axes['mag-closed'], laplace['closed'], digital['closed'], 'Closed')
        plot_ang(axes['ang-closed'], laplace['closed'], digital['closed'], 'Closed')

        plot_mag(axes['mag-open'], laplace['open'], digital['open'], 'Open')
        plot_ang(axes['ang-open'], laplace['open'], digital['open'], 'Open')
        plot_transient(pll_fnat, pll_zeta, biq_qual, biq_fcut)

def setup_slider(pll_fnat, pll_zeta, biq_qual, biq_fcut, pll_fnat_cb, pll_zeta_cb,
biq_qual_cb, biq_fcut_cb):
        plt.subplots_adjust(left=0.1, bottom=0.25)
        axcolor = 'lightgoldenrodyellow'
        ax1 = plt.axes([0.2, 0.01, 0.65, 0.03], axisbg=axcolor)
        ax2 = plt.axes([0.2, 0.06, 0.65, 0.03], axisbg=axcolor)
        ax3 = plt.axes([0.2, 0.11, 0.65, 0.03], axisbg=axcolor)
        ax4 = plt.axes([0.2, 0.16, 0.65, 0.03], axisbg=axcolor)
```

```
        sfnat = Slider(ax1, 'cost_fnat', 10, 500.0, valinit=pll_fnat)
        szeta = Slider(ax2, 'cost_zeta', 0.05, 3.0, valinit=pll_zeta)
        squal = Slider(ax3, 'biqu_qual', 0.05, 3.0, valinit=biq_qual)
        sfcut = Slider(ax4, 'biqu_fcut', 10, 4000.0, valinit=biq_fcut)

        sfnat.on_changed(pll_fnat_cb)
        szeta.on_changed(pll_zeta_cb)
        squal.on_changed(biq_qual_cb)
        sfcut.on_changed(biq_fcut_cb)
        return (sfnat, szeta, squal, sfcut)

def plot_transient(pll_fnat, pll_zeta, biq_qual, biq_fcut):
        pts = 5000
        phivco = step.pseudo_transient(pll_fnat, pll_zeta, biq_qual, biq_fcut, pts, fs, 1.0)
        phases = np.arange(pts)/fs*pll_fnat
        ax = axes['transient']['ax']
        handle = axes['transient']['handle']
        if handle is None:
                handle, = ax.plot(phases, phivco)
                ax.set_title('Transient Response')
                ax.set_ylabel('VCO Phase [rad]')
                ax.set_xlabel('Cycles t$f_n$ [u]')
                axes['transient']['handle'] = handle
                ax.set_ylim((-0.01, 2.0))
                ax.set_xlim((0, 4.0))
                ax.grid(True)
        else:
                handle.set_xdata(phases)
                handle.set_ydata(phivco)
                ax.relim()
                ax.autoscale_view()

plt.figure(0, facecolor='white')
pts = 10000
fs = 44.1E3
freqs = (1.0 + np.arange(pts))/float(pts) * 10E3
s = 2.0j * np.pi * freqs
z2s = bilinear_transform(fs, freqs)
pll_fnat = 170.0
pll_zeta = 1.0/np.sqrt(2.0)
biq_qual = 1.0/np.sqrt(2.0)
biq_fcut = 1000.0

axes = {}

axes['mag-closed'] = {'ax':plt.subplot2grid((6,4), (0, 0), colspan=2, rowspan=2),
'handles':None}
axes['ang-closed'] = {'ax':plt.subplot2grid((6,4), (2, 0), colspan=2, rowspan=2),
'handles':None}
axes['mag-open'] =   {'ax':plt.subplot2grid((6,4), (0, 2), colspan=2, rowspan=2),
'handles':None}
axes['ang-open'] =   {'ax':plt.subplot2grid((6,4), (2, 2), colspan=2, rowspan=2),
'handles':None}
axes['transient'] =  {'ax':plt.subplot2grid((6,4), (4, 1), colspan=2, rowspan=2),
'handle':None}
plot_resp(pll_fnat, pll_zeta, biq_qual, biq_fcut)
plt.tight_layout()

def pll_fnat_cb(val):
        global pll_fnat
        pll_fnat = val
        plot_resp(pll_fnat, pll_zeta, biq_qual, biq_fcut)

def pll_zeta_cb(val):
```

```
        global pll_zeta
        pll_zeta = val
        plot_resp(pll_fnat, pll_zeta, biq_qual, biq_fcut)

def biq_qual_cb(val):
        global biq_qual
        biq_qual = val
        plot_resp(pll_fnat, pll_zeta, biq_qual, biq_fcut)

def biq_fcut_cb(val):
        global biq_fcut
        biq_fcut = val
        plot_resp(pll_fnat, pll_zeta, biq_qual, biq_fcut)

sfnat, szeta, sfcut, squal = setup_slider(pll_fnat, pll_zeta, biq_qual, biq_fcut,
pll_fnat_cb, pll_zeta_cb, biq_qual_cb, biq_fcut_cb)
plt.show()
```

Sample Output



step.py

```
# FILENAME: step.py
#
# DESCRIPTION: Generate transient step response of Costas loop
#
# AUTHOR: Robby J. Tong          DATE: 5/7/2017
# EMAIL: robbytong@gmail.com
#
# CHANGES:
#
import matplotlib.pyplot as plt
import numpy as np
from Integrator import Integrator
from LowPass import LowPass
from LoopAmplifier import LoopAmplifier

# Simulation Variables
def pseudo_transient(pll_fnat, pll_zeta, biq_qual, biq_fcut, npts, fs, phase_step):
       twofs = 2.0*fs
       twopi = 2.0*np.pi

       # Loop Variables
       phivco = 0

       vco_int = Integrator(fs)
       amp = LoopAmplifier(pll_zeta, pll_fnat, fs)

       # Generate Arrays
       phases = phase_step*np.ones(npts)
       pvco = np.zeros(npts)
       filt1 = LowPass(biq_fcut, fs, biq_qual)
       filt2 = LowPass(biq_fcut, fs, biq_qual)

       for n in xrange(npts):
              pvco[n] = phivco
              s1 = phases[n] - phivco
              in_pha = filt1.work(np.cos(s1))
              qu_pha = filt2.work(np.sin(s1))
              err = in_pha * qu_pha

              loop_amp_out = amp.work(err)
              phivco = vco_int.work(loop_amp_out)

       return pvco
```

No Sample Output

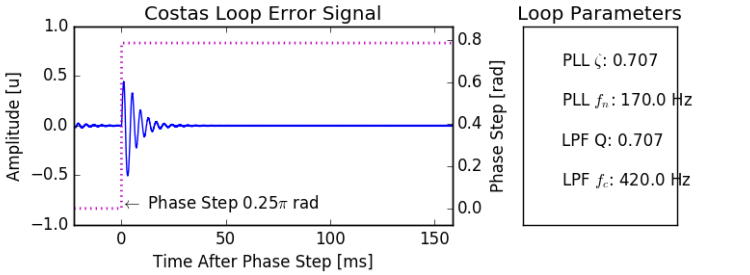
costa.py

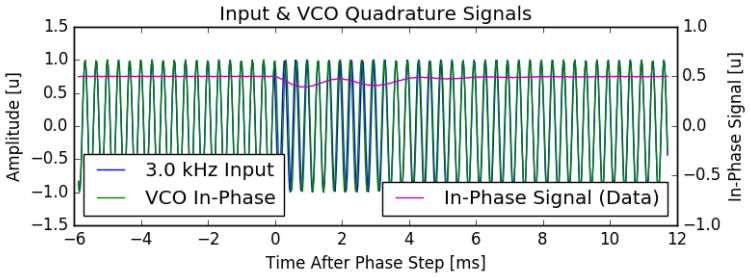
```
# FILENAME: costa.py
#
# DESCRIPTION: Software Costas Loop
#
# AUTHOR: Robby J. Tong          DATE: 5/7/2017
# EMAIL: robbytong@gmail.com
#
# CHANGES:
#
import matplotlib
matplotlib.use('TkAgg')
import matplotlib.pyplot as plt
import numpy as np
from numpy.fft import fft
from Integrator import Integrator
from LowPass import LowPass
from LoopAmplifier import LoopAmplifier

# ==================
# User Customization
# ==================

# Points
pts = 8000

# Sampling Frequency
fs = 44.1E3

# VCO Center Frequency
fc = 3E3

# Biquad Quality Factor
biq_qual = 1.0/np.sqrt(2.0)

# Biquad Cutoff Frequency
biq_fcut = 420.0

# Damping Ratio Zeta
pll_zeta = 1.0/np.sqrt(2.0)

# Natural Loop Oscillating Frequency
pll_fnat = 170.0

# How much to phase step by
phase_step = np.pi/4.0

# Input ramp signal instead?
ramp_test = False

# The slope of the ramp signal in Hz
freq = 250.0

# Show downconverted in phase signal?
show_in_phase = True

# ====================
# Calculate Constants
# ====================
```

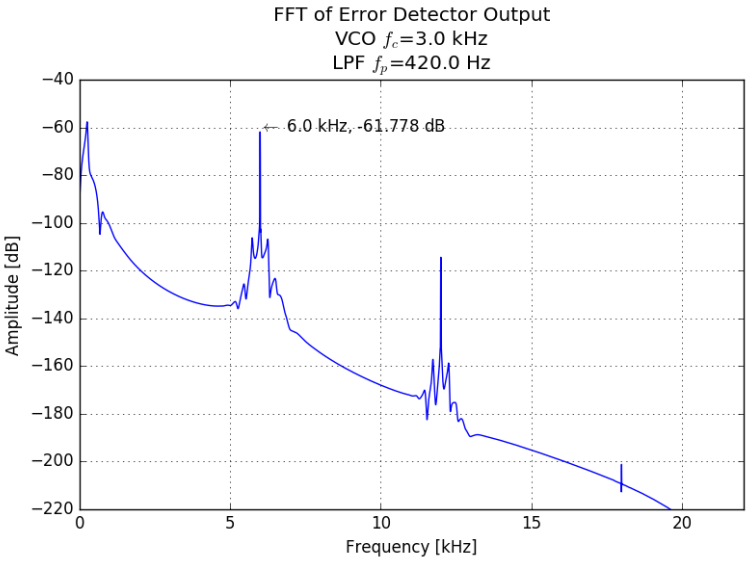
```
# Index out of points to place the phase step
nstep = int(pts/8)

# ====================================
# Instantiate Signal Processing Blocks
# ====================================

inp_lpf = LowPass(biq_fcut, fs, biq_qual)
qup_lpf = LowPass(biq_fcut, fs, biq_qual)

vco = Integrator(fs)
amp = LoopAmplifier(pll_zeta, pll_fnat, fs)

# ==============================
# Initialize Input Output Arrays
# ==============================

# Create a times array
times = np.arange(pts)/fs

# Create a phase array that has a center frequency of fc
# This is for the VCO and the input signal
phase = 2.0*np.pi*fc*times

if ramp_test:
       freq_ramp = np.arange(pts-nstep)/fs*freq*2.0*np.pi
       step  = np.append(np.zeros(nstep), freq_ramp)
else:
       step  = np.append(np.zeros(nstep), phase_step * np.ones(pts-nstep))

phase += step

sig = np.cos(phase)
out = np.zeros(pts)
raw = np.zeros(pts)
inc = 2.0*np.pi*fc
dat = np.zeros(pts)

for n in xrange(pts):
       vco_inp = np.cos(vco.value())
       vco_qup =-np.sin(vco.value())
       inp = inp_lpf.work(sig[n] * vco_inp)
       qup = qup_lpf.work(sig[n] * vco_qup)
       err = inp * qup * 4.0
       loop_amp_out = amp.work(err)
       vco.work(inc+loop_amp_out)
       out[n] = err
       raw[n] = vco_inp
       dat[n] = inp

times_ms = (times-times[nstep]) * 1000.0

plt.figure(1)

info_ax= plt.subplot2grid((2,3), (0,2), colspan=1, rowspan=1)
err_ax = plt.subplot2grid((2,3), (0,0), colspan=2, rowspan=1)
zom_ax = plt.subplot2grid((2,3), (1,0), colspan=3, rowspan=1)

info_ax.text(1,1, 'LPF $f_c$: %.1f Hz'%biq_fcut)
info_ax.text(1,2, 'LPF Q: %.3f'%biq_qual)
info_ax.text(1,3, 'PLL $f_n$: %.1f Hz'%pll_fnat)
info_ax.text(1,4, 'PLL $\zeta$: %.3f'%pll_zeta)
info_ax.set_title('Loop Parameters')
info_ax.set_xlim((0, 4))
info_ax.set_ylim((0, 5))
```

```
info_ax.yaxis.set_visible(False)
info_ax.xaxis.set_visible(False)

err_ax.plot(times_ms, out)
err_ax.set_xlim((times_ms[0], times_ms[-1]))
err_ax.set_ylim((-1, 1))
err_ax.set_xlabel('Time After Phase Step [ms]')
err_ax.set_ylabel('Amplitude [u]')
err_ax.set_title('Costas Loop Error Signal')

nstart = nstep - int(fs / pll_fnat)
nstop  = nstep + int(fs / pll_fnat * 2)
ztimes = (times[nstart:nstop] - times[nstep]) * 1000.0

pha_ax = err_ax.twinx()
pha_ax.plot(times_ms, step, ':', color='m', linewidth=2)
pha_ax.set_xlim((times_ms[0], times_ms[-1]))
pha_ax.set_ylabel('Phase Step [rad]')

if ramp_test:
       delta = step[-1] - step[0]
       pha_ax.text(0.0, 0.0, '$\leftarrow$ Frequency Ramp %.1f Hz'%freq)
       pha_ax.set_ylim(step[0]-delta*0.1, step[-1]+delta*0.1)
else:
       pha_ax.text(0.0, 0.0, '$\leftarrow$ Phase Step %.2f$\pi$ rad'%(phase_step/np.pi))
       delta = step[nstop] - step[nstart]
       pha_ax.set_ylim((step[nstart]-delta*0.1, step[nstop]+delta*0.1))

zom_ax.plot(ztimes, sig[nstart:nstop])
zom_ax.plot(ztimes, raw[nstart:nstop])
zom_ax.legend(['%.1f kHz Input'%(fc/1000.0), 'VCO In-Phase'], loc='lower left')
zom_ax.set_xlabel('Time After Phase Step [ms]')
zom_ax.set_ylabel('Amplitude [u]')
zom_ax.set_title('Input & VCO Quadrature Signals')
zom_ax.set_xlim((ztimes[0], ztimes[-1]))
zom_ax.set_ylim((-1.5, 1.5))

if show_in_phase:
       bpsk_ax = zom_ax.twinx()
       bpsk_ax.plot(ztimes, dat[nstart:nstop], 'm')
       bpsk_ax.set_ylabel('In-Phase Signal [u]')
       bpsk_ax.set_ylim((-1, 1))
       bpsk_ax.legend(['In-Phase Signal (Data)'], loc='lower right')


plt.tight_layout()
plt.figure(2)
err_fft = fft(out * np.blackman(pts))/float(pts)
freqs = np.arange(pts)/float(pts)*fs/1000.0
mags = 20.0 * np.log10(np.abs(err_fft))
plt.plot(freqs[:pts/2], mags[:pts/2])
plt.gca().set_xlim((0, fs/2000.0))
plt.gca().set_ylim((-220, -40))
plt.gca().set_xlabel('Frequency [kHz]')
plt.gca().set_ylabel('Amplitude [dB]')
k_double = int(2.0*fc/fs*pts)
plt.gca().text(freqs[k_double], mags[k_double], '$\leftarrow$ %.1f kHz, %.3f
dB'%(freqs[k_double], mags[k_double]))
plt.gca().set_title('FFT of Error Detector Output\nVCO $f_c$=%.1f kHz\nLPF $f_p$=%.1f
Hz'%(fc/1E3, biq_fcut))
plt.grid(True)
plt.tight_layout()

plt.show()
```

Sample Output







simple_costa.py

```
# FILENAME: simple_costa.py
#
# DESCRIPTION: Barebones Software Costas Loop
#
# AUTHOR: Robby J. Tong          DATE: 5/20/2017
# EMAIL: robbytong@gmail.com
#
# CHANGES:
#

# Import matplotlib graphing library
import matplotlib
# Tell matplotlib to use TkAgg 'backend'. TkAgg only works on Mac Osx
matplotlib.use('TkAgg')

# Import graphing object pyplot, rename it as plt
import matplotlib.pyplot as plt

# Import numerical mathematics library numpy, rename it as np
import numpy as np

# From ./Integrator.py import the Integrator class
from Integrator import Integrator

# From ./Lowpass.py import the LowPass class
from LowPass import LowPass

# From ./LoopAmplifier.py import the LoopAmplifier class
from LoopAmplifier import LoopAmplifier

# ==================
# User Customization
# ==================

# Points
pts = 8000

# Sampling Frequency
fs = 44.1E3

# VCO Center Frequency
fc = 3E3

# Biquad Quality Factor
biq_qual = 1.0/np.sqrt(2.0)

# Biquad Cutoff Frequency
biq_fcut = 1000.0

# Damping Ratio Zeta
pll_zeta = 1.0/np.sqrt(2.0)

# Natural Loop Oscillating Frequency
pll_fnat = 170.0

# How much to phase step by
phase_step = np.pi/4.0

# ==================
```

```
# Calculate Constants
# ====================

# VCO Center frequency
inc = 2.0*np.pi*fc

# Index when phase step will occur
nstep = int(pts/8)

# ====================================
# Instantiate Signal Processing Blocks
# ====================================

# In-phase and quadrature lowpass filters
inph_lpf = LowPass(biq_fcut, fs, biq_qual)
quad_lpf = LowPass(biq_fcut, fs, biq_qual)

# Loop Amplifier object
amp = LoopAmplifier(pll_zeta, pll_fnat, fs)

# VCO Integrator object
vco = Integrator(fs)

# ================================================
# Create Arrays for Holding Input and Output Data
# ================================================

# Create a times array
times = np.arange(pts)/fs

# Time in milliseconds, used for plotting
times_ms = times * 1000.0

# Input phase ramp signal
phase = 2.0*np.pi*fc*times

# Create a step signal with a max of phase_step (see user customization above)
step  = np.append(np.zeros(nstep), phase_step * np.ones(pts-nstep))

# Add it to the phase ramp
phase += step

# Create the input signal array
input_sig = np.cos(phase)

# Create an array to hold the output signal
outpu_sig = np.zeros(pts)

# Create an array to hold the error signal
error_sig = np.zeros(pts)

# =========
# Main Loop
# =========

for n in xrange(pts):
       # Generate VCO inphase and quadrature signals
       vco_inph = 2.0 * np.cos( vco.value() )
       vco_quad =-2.0 * np.sin( vco.value() )

       # Downconvert the input signal
       inph = inph_lpf.work( input_sig[n] * vco_inph )
       quad = quad_lpf.work( input_sig[n] * vco_quad )

       # Generate the error signal
```

```
        error = inph * quad

        loop_amp_out = amp.work(error)

        vco.work(inc+loop_amp_out)
        outpu_sig[n] = vco_inph
        error_sig[n] = error

plt.figure(1)

# =====================
# Plot the error signal
# =====================

err_ax = plt.subplot(2,1,1)
err_ax.plot(times_ms, error_sig)
err_ax.set_xlim((times_ms[0], times_ms[-1]))
err_ax.set_ylim((-1, 1))
err_ax.set_xlabel('Time [ms]')
err_ax.set_ylabel('Amplitude [u]')
err_ax.set_title('Costas Loop Error Signal')
err_ax.grid(True)

# ======================================================
# Plot the VCO In Phase Signal Against the Input Signal
# ======================================================

# Since there is a lot of data and cycles, we are only interested in
# behavior around the phase step

# The first index we wish to start from is one natural closed loop oscillation
# cycle before the phase step
nstart = nstep - int(fs / pll_fnat)

# The last index is two natural closed loop oscillation cycles after the phase
# step
nstop  = nstep + int(fs / pll_fnat * 2)

# ztimes i.e. zoomed times, with 0 occuring at the phase step
ztimes = (times[nstart:nstop] - times[nstep]) * 1000.0

zom_inp = input_sig[nstart:nstop]
zom_out = outpu_sig[nstart:nstop]

zom_ax = plt.subplot(2,1,2)

zom_ax.plot(ztimes, zom_inp)
zom_ax.plot(ztimes, zom_out/2.0)
zom_ax.legend(['%.1f kHz Input'%(fc/1000.0), 'VCO In-Phase'], loc='lower left')

zom_ax.set_xlabel('Time After Phase Step [ms]')
zom_ax.set_ylabel('Amplitude [u]')
zom_ax.set_title('Input & VCO Quadrature Signals')
zom_ax.set_xlim((ztimes[0], ztimes[-1]))
zom_ax.set_ylim((-1.5, 1.5))

plt.grid(True)
plt.tight_layout()
plt.show()
```

Sample Output