Control Systems

Quick Start with Python

Dan Boschen September 2024

License

Copyright © 2024 C. Daniel Boschen

This document “Control Systems - Quick Start with Python” by Dan Boschen is licensed under CC BY-NC-ND 4.0.

To view a copy of this license, visit https://creativecommons.org/licenses/by-nc-nd/4.0/ .

While every precaution has been taken in the preparation of this notebook, the author, publisher, and distribution partners assume no responsibility for any errors or omissions, or any damages resulting from the use of any information contained within it.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Link to Python Control Library Docs: https://python-control.readthedocs.io/en/0.9.4/

In [1]:
```
import numpy as np
import scipy.signal as sig
import matplotlib.pyplot as plt
import control as con
import scipy.fft as fft
import numpy.random as rand
import math
# sound processing
from IPython.display import Audio
import wave
```

In [2]:
```
%matplotlib ipympl
```

In [3]:
```
# configurations

# disable max open figure warning
plt.rcParams.update({'figure.max_open_warning': 0})
```

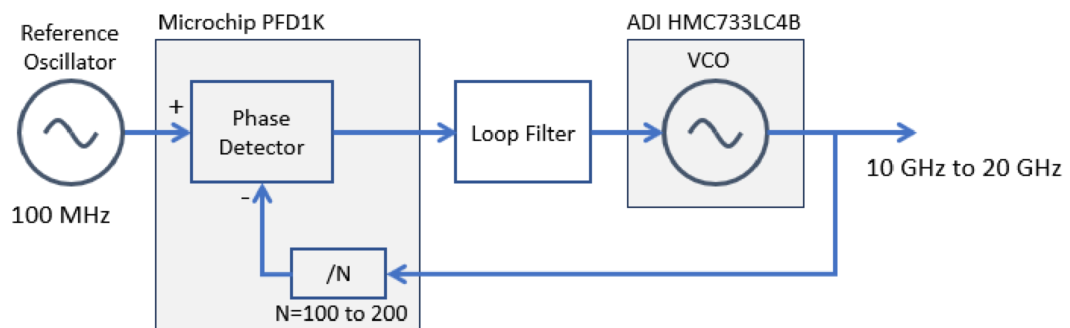
Analog Phase Lock Loop Implementation

For this simulation we will model a PLL using the Microchip PFD1K 8 GHz Phase/Frequency Detector

to lock an HMC733LC4B 10 to 20 GHz VCO to a 100 MHz reference for outputs from 10 to 20 GHz in 100 MHz steps

This will require a prescaler of 10e9/100e6 = 100 up to 20e9/100e6 = 200

We'll design for a loop BW of 1 MHz.



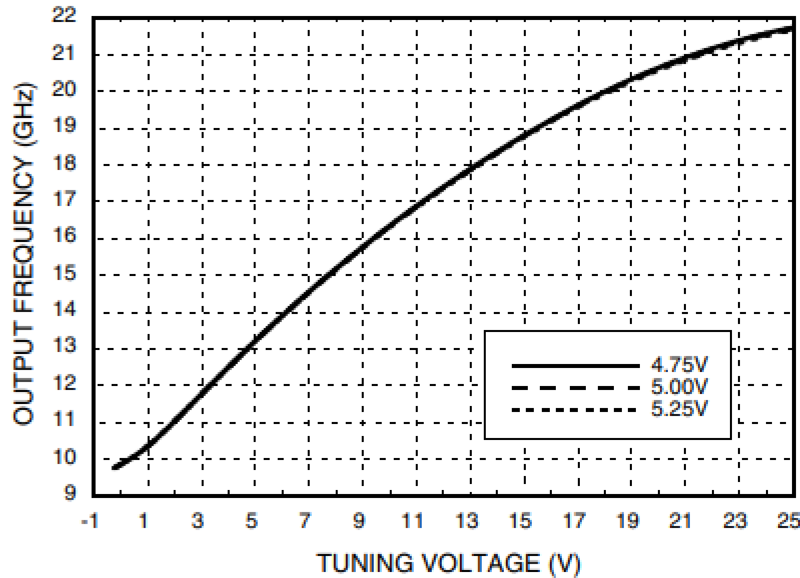
Analog PLL Loop Model

VCO

HMC733LC4B 10-20 GHz VCO

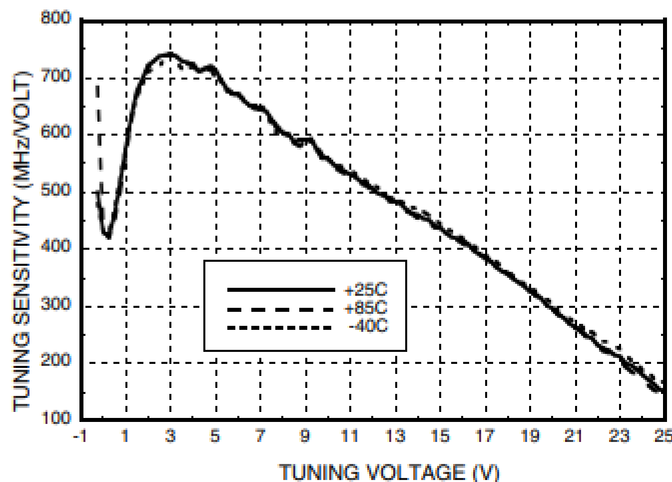
Establish VCO gain (tuning slope) from ADI datasheet:

https://www.analog.com/media/en/technical-documentation/data-sheets/hmc733.pdf



At 10 GHz the control voltage = 0V, at 20 GHz the control voltage = 18V

Howeer the curve suggests the tuning slope is not quite linear. Conveniently the slope (derivative) is provided to us in the datahsheet:



We see from this that we have a frequency dependent gain constant. (Best approach in my opinion is to provide a linearization translation so that we have a constant slope independent of frequency- in this quick example we will determine the gain coefficients for operation at 15 GHz and then from that determined the variability as the frequency is increased and decreased. The different N for each setting also effects the loop parameters).

From the datasheet plots we see that for operation at a 15 GHz output, the tuning voltage is nearly 7.5V, and at 7.5V the slope is approximately 600 MHz/Volt.

In radian frequency this is $K_V = 2\pi 600e6 = 3.77e9$ (rad/sec)/V

For use in a phase lock loop, the phase vs time of the VCO output is the integral of it's frequency vs time. (Since frequency is a change in phase versus a change in time or $d\phi/dt$). The output frequency is directly proportional to the input control voltage, thus in the time domain, the VCO is an integrator as well as unit translator from volts to phase and we have the complete operation of the VCO in the Laplace domain as:

$$\frac{K_V}{s} = 3.77e9 \text{ rad/V}$$

The "s" that appears in the formula above is complex frequency, not to be confused with seconds. $s = \sigma + j\omega$ and has units of 1/seconds (hence frequency).

PFD

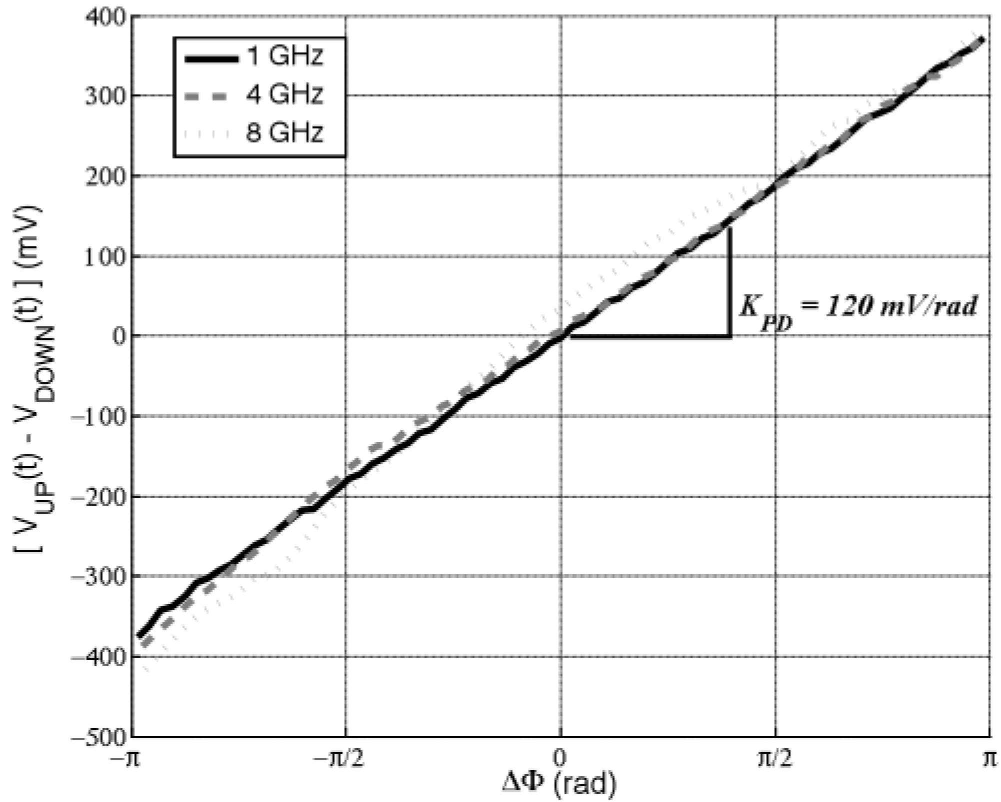
PFD1K

Establish Phase Detector Gain K_{PD} from Microchip Datasheet:

https://ww1.microchip.com/downloads/aemDocuments/documents/RFDS/ProductDocuments/D

Output voltage vs phase with differential output properly terminated to convert currents to voltage:

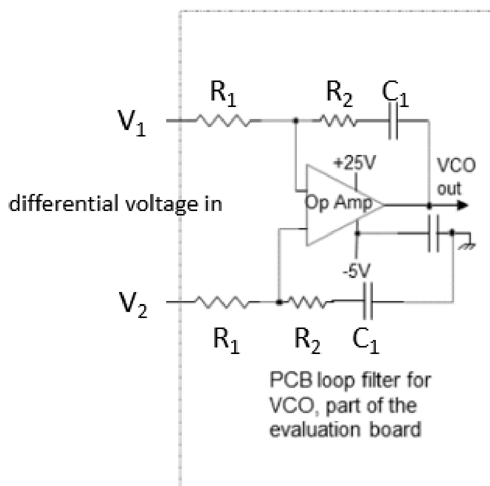
Figure 1-7. Diff. Output Voltage vs. Frequency (0 dBm Pin)



Loop Filter

The evaluation board for the PFD1K includes a simple Proportional-Integral (PI) Loop Filter (see page 17 of the datasheet), which integrates the differential voltage out of the PFD and adds a proportional gain to produce the single control voltage the the VCO:

Differential PI Loop Filter



$$V_{out} = \frac{R_2 + \frac{1}{sC_1}}{R_1}(V_2 - V_1)$$

$$\frac{Vout}{V_{in}} = H(s) = \frac{1 + sR_2C_1}{sR_1C_1}$$

We will abbreviate R_2C_1 as τ_2 and R_1C_1 at τ_1

Thus

$$H(s) = \frac{1 + s\tau_2}{s\tau_1}$$

And we see that the loop filter has a *zero* at:

$$1 + s\tau_2 = 0$$

$$s = \frac{-1}{\tau_2}$$

and as an integrator, has a pole at $s = 0$, and a gain of $\frac{1}{\tau_1}$.

Rewriting into it's proportional and integral components, we get:

$$H(s) = \frac{1}{s\tau_1} + \frac{\tau_2}{\tau_1}$$

Open Loop Gain

We will create a Bode plot (by plotting the open loop gain) to see how τ_2 adjusts the gain, and the effect of the zero as adjusted with τ_2

The Open Loop Gain is the result of cascading the following components, resulting in a product of their gains:

VCO: K_V/s
Phase Detector: K_{PD}
Loop Filter: $\frac{1+s\tau_2}{s\tau_1}$
Frequency Divider: $1/N$

The product of the above is the "open loop gain" as:

$$G_{OL}(s) = \frac{k_V\, k_{PD}}{N s}\frac{1 + s\tau_2}{s\tau_1} = \frac{k_V\, k_{PD}}{N\tau_1}\frac{1 + s\tau_2}{s^2}$$

In [4]:
```
# Loop Equations

a_kv= 2*np.pi*600e6       # VCO gain in rad/v  (from HMC733 datasheet)
a_kpd = 0.120             # Phase detector gain v/rad (from PFD1K datasheet)
a_lbw = 2*np.pi* 1e6      # target loop bw in rad/sec (cuz Dan said)
a_N = 150                 # mid value for N (divider setting to get 15 GHz output)

# since we'll iterate on loop filter gain constants, make the open loop gain a func
# Note: numerator and denominator polynomials are entered in positive powers of s i
def gol_analog(tau1, tau2, N):
    return a_kv * a_kpd/(N*tau1)*con.tf([tau2, 1], [1, 0, 0])
```

Starting Loop Values

We can get an initial value for τ_1 by first neglecting the effects of τ_2 by setting $\tau_2 = 0$ and choosing a zero dB crossing on the Bode gain plot to be the loop BW.

The zero dB gain crossing is when:

$$|G_{OL}(s)| = 1$$

With $s = j\omega_c$, the loop bandwidth.

With $\tau_2 = 0$ and $|G_{OL}(j\omega_c)| = 1$ the solution for τ_1 becomes:

$$\tau_1 = \frac{k_V k_{PD}}{N\omega_c^2}$$

We'll then add the zero at (45° phase margin) or slightly below (higher phase margin) the loop bandwidth for stability.

This will increase the bandwidth slightly, so then iterate on both from these starting values to decrease the loop gain using τ_1, and increase or decrease τ_2 while observing response on Bode plot for desired gain and phase margin.

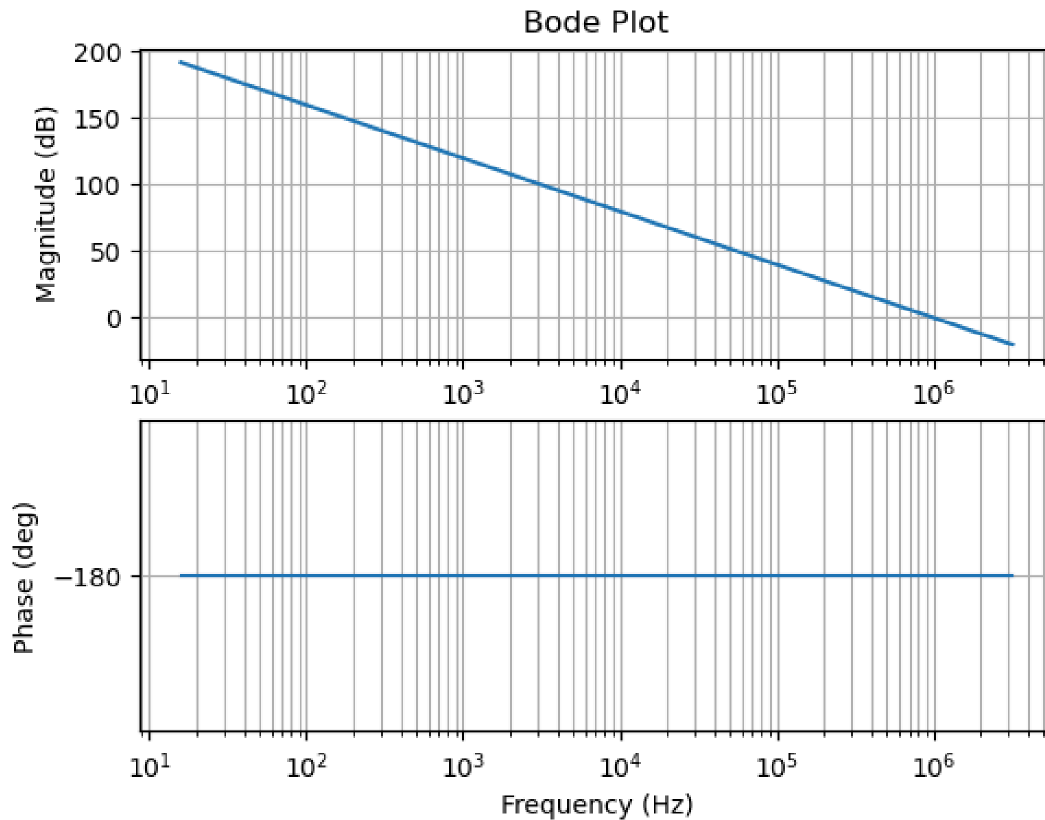
In [5]:
```
print(f"Target loop bw = {a_lbw:0.2f} rad/sec")

a_tau1_init = (a_kv * a_kpd)/(a_N * a_lbw**2)
print(f"Initial value for tau1 = {a_tau1_init:0.2e}")
```

```
Target loop bw = 6283185.31 rad/sec
Initial value for tau1 = 7.64e-08
```

In [6]:
```
# To demonstrate show Bode Plot with tau2=0 resulting in the cascade of two integra
a_tau2=0
a_gol = gol_analog(a_tau1_init, a_tau2, a_N)

plt.figure()

__ = con.bode(a_gol, dB=True, Hz=True, omega_limits=[100, 20e6])
plt.subplot(2,1,1)
plt.title("Bode Plot")
```

Out[6]:
```
Text(0.5, 1.0, 'Bode Plot')
```

Figure



If the above Bode plot has a 0dB crossing on the magnitude plot when the frequency is 1 MHz: **Success!** We have properly set τ_1 (a gain constant). The closed loop bandwidth will be where the open loop frequency magnitude response crosses 0 dB. Adjusting τ_1 will simply move the gain curve up and down, and thus adjust the loop bandwidth.

As implemented thus far, with $\tau_2 = 0$, the loop will not be stable, given the phase of the open loop gain is at 180 degrees when the gain passes through 0. (The critera for stability using the open loop Bode plot is for the phase to be < 180 degrees when the gain passes through 0 dB).

This is where adding the zero with τ_2 comes in.

The two poles at $s = 0$ (DC) cause the Bode magnitude to drop -40 dB/decade, and the phase be at -180° (-90° for each pole). The zero will add an increase to the magnitude +20 dB/decade, and a +90° increase to the phase at an intercept frequency given by:

$$f_c = \frac{1}{2\pi\tau_2}$$

If we place the zero right at the loop bw, this will provide 45° of phase margin.

If we rearrange/simplify the formula for open loop gain to show overall gain, poles and zeros, we can get more insight into how we may adjust these parameters:

$$G_{OL}(s) = \frac{k_V k_{PD}}{N\tau_1}\frac{1+s\tau_2}{s^2} = \frac{K}{\tau_1}\frac{1+s\tau_2}{s^2}$$

$$= K\frac{\tau_2}{\tau_1}\frac{1/\tau_2 + s}{s^2}$$

From this we see that we can set the zero as $s = -1/\tau_2$, and move the gain up and down as the ratio τ_2/τ_1. And therefore we have independent adjustment of our loop bandwidth and phase margin (which controls the damping factor).

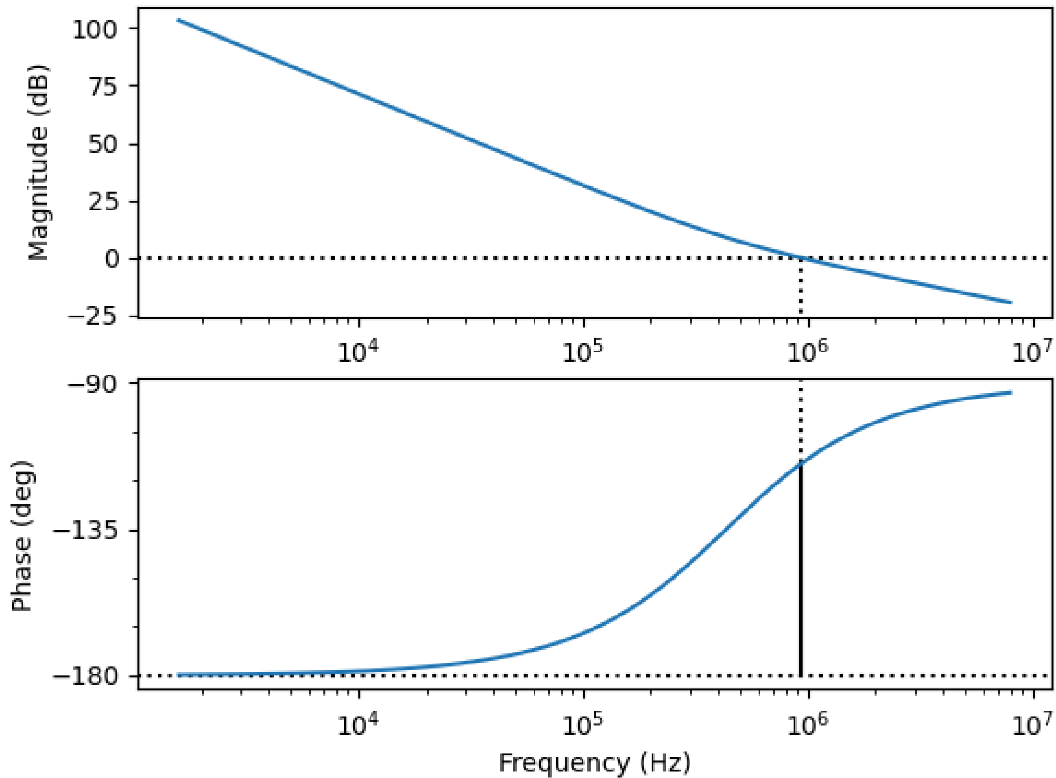
In [14]:

```
# inial values were tau2 = 1/lbw and tau1 = 1.4 x tau2 computed above for a 45 degr
# then iterate to increase phase margin to increase the damping factor and keep the
# end result after interating: tau2 = 2.3/lbw, tau1 = 2.7 x tau1 computed above

# 1/tau1 is the integral gain, and tau2/tau1 is the proportional gain

a_tau2 = 2.3/a_lbw       # adjusts phase as 1/tau2, this will change the zeo crossi
a_tau1 = 2.7*a_tau1_init
a_gol = gol_analog(a_tau1, a_tau2, a_N)

plt.figure()

__ = con.bode(a_gol, dB=True, Hz=True, margins=True, omega_limits=[10000, 50e6])
plt.subplot(2,1,1)
plt.title("Bode Plot")
plt.show()
```

Figure

Gm = inf dB (at nan Hz), Pm = 65.15 deg (at 938776.44 Hz)

Bode Plot



Adjusting τ_2 has a dominant effect on the phase, but because it reduced the slope of the gain, it has a secondary effect on the bandwidth. We notice above that the zero crossing is now higher than 1 MHz, which is fixed by reducing the gain (increasing τ_1).

From this we can then tweak τ_1 (gain as the ratio of τ_2/τ_1, and therefore bandwidth) and τ_2 (phase margin). If we want to reduce the ringing (increase the dampling factor ζ), then we need to increase the phase margin.

We could determine exact solution for the specific 2nd order PI Loop (such as has been done by Floyd Gardner, "Phase Lock Techniques"), but this exercise in iterative tuning gives insight into what to do for a broader range of applications.

Closed Loop

Example closed loop gains of interest are:

- The gain from the reference input to VCO output, to determine tracking of the reference
- The gain from the vco output to the vco output (to determine attenuation of VCO phase noise)

$$G_{CL}(s) = \frac{G_F(s)}{1 + G_{OL}(s)}$$

Ref to VCO out

$$G_{F1}(s) = N G_{OL}(s)$$

$$G_{CL1}(s) = \frac{G_{F1}(s)}{1 + G_{OL}(s)} = \frac{N G_{OL}(s)}{1 + G_{OL}(s)}$$

VCO out to VCO out

$$G_{F2}(s) = 1$$

$$G_{CL2}(s) = \frac{1}{1 + G_{OL}(s)}$$

In [15]:
```
# Closed Loop from Ref Input to VCO Output

a_gcl1 = a_N * a_gol/(1+a_gol)


# good practice to always use the minreal to reduce the transfer function!
print(f"Transfer function before using minreal: {a_gcl1}")


a_gcl1 = con.minreal(a_N * a_gol/(1+a_gol))

print(f"Transfer function after using minreal:{a_gcl1}")
```

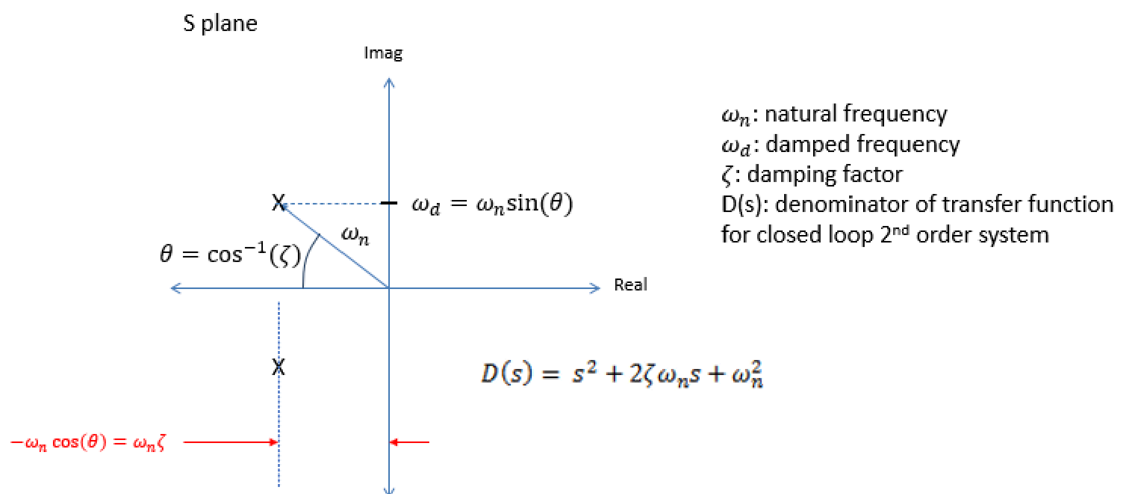
```
Transfer function before using minreal:
   8.029e+08 s^3 + 2.193e+15 s^2
-----------------------------------
s^4 + 5.352e+06 s^3 + 1.462e+13 s^2

2 states have been removed from the model
Transfer function after using minreal:
   8.029e+08 s + 2.193e+15
-----------------------------
s^2 + 5.352e+06 s + 1.462e+13
```

In [16]:
```
# Closed Loop from VCO in to VCO out

a_gcl2 = con.minreal(1/(1+a_gol))
print(a_gcl2)
```

```
0 states have been removed from the model

             s^2
-----------------------------
s^2 + 5.352e+06 s + 1.462e+13
```

Damping Factor

For a second order system the damping factor ζ is the cosine of the angle to the pole from the negative real axis



As damping factor approaches 0, rise time will get faster at the expense of more ringing and overshoot.

As damping factor approaches 1, rise time and overshoot will decrease.

Once the damping factor is at 1, the poles are on the real axis, and the system is "underdamped".

A damping factor close to 0.7 is typically desirable as it offers a good compromise for balanceing rise time and overshoot/ringing considerations.

In [17]:
```
# For a 2nd order system, the damping factor is the cosine of the angle to the pole
# negative real axis.
# A Dampling
print(f"Damping factor is {np.cos(np.pi-np.angle(con.poles(a_gcl1)[0])):0.2f}")
```

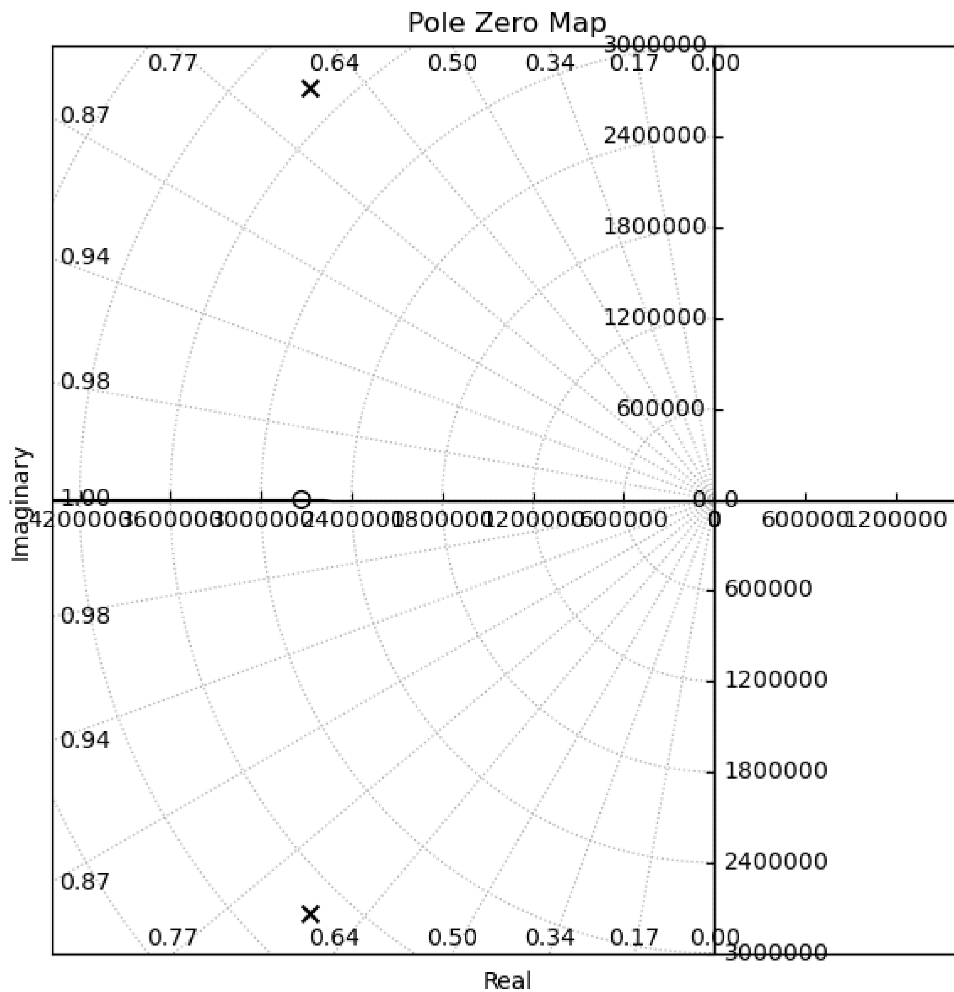
```
Damping factor is 0.70
```

Pole Zero Map

Becuase 2nd order systems are so common the pole zero mapping utilities in Python (and Matlab/Octave) will include the option to superimpose lines of constant natural frequency and damping factor. Higher order systems will also approximate a 2nd order system when there are two poles closest to the jω axis (dominant poles) with other poles significantly further to the left (~ >10x) into the left half plane.

In [29]:
```
plt.figure(figsize=(7,7))
__ = con.pzmap(a_gcl1, grid=True);
```

Figure



Closed Loop Time Domain Response (Step)

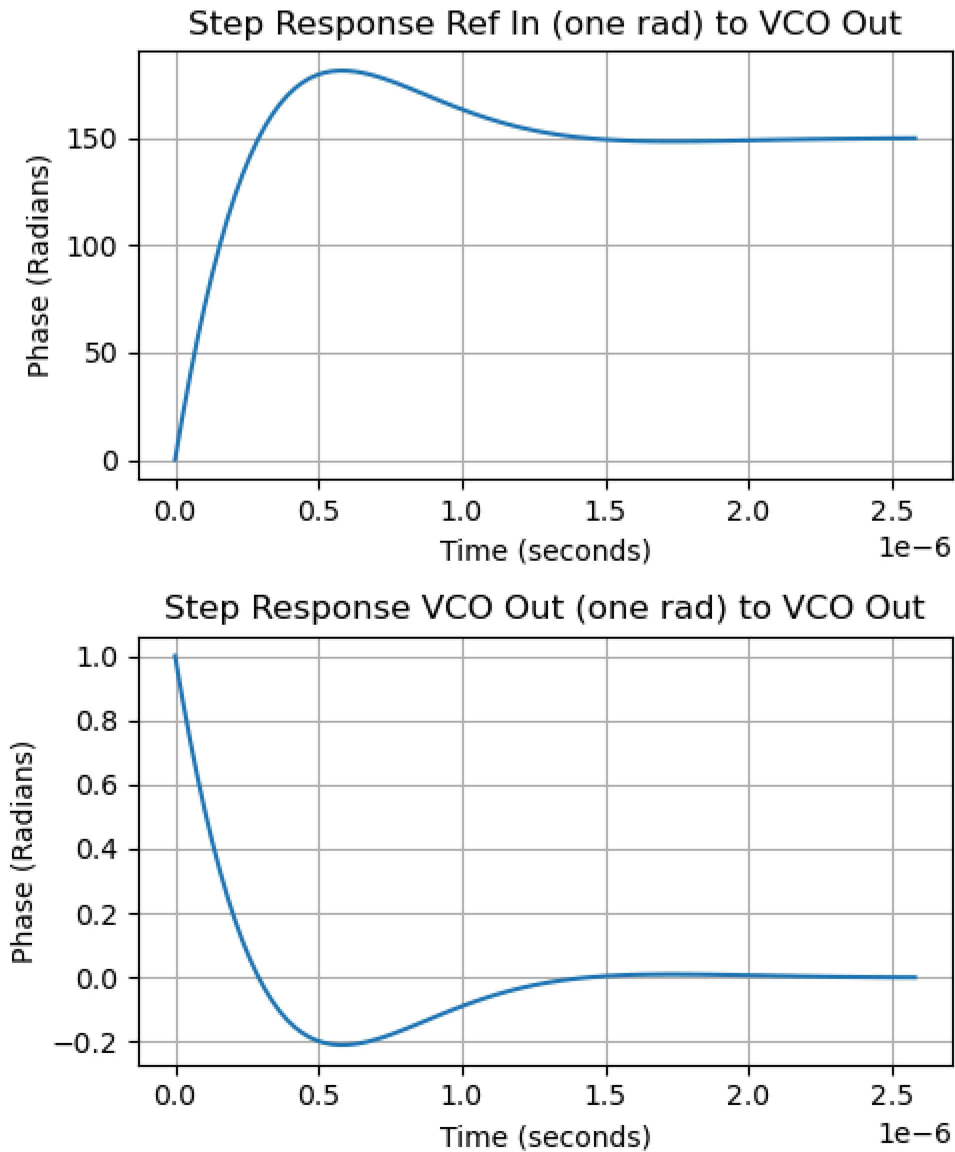
Interpreting the step response results:

The step responses shown are for a normalized step at the input for an actual small signal step (within the loops linear operating range). The response is for a step from 0 to 1, so in this case 1 radian, and given the frequency multiplication of this loop, we get a 150 radian phase step at the output (which then gets divided in the divider by 150, producing the equal 1 radian step at the other input to the phase detector).

In [33]:

```
plt.figure(figsize=(5,6))
plt.subplot(2,1,1)
plt.plot(*con.step_response(a_gcl1))
plt.xlabel("Time (seconds)")
plt.ylabel("Phase (Radians)")
plt.title("Step Response Ref In (one rad) to VCO Out")
```

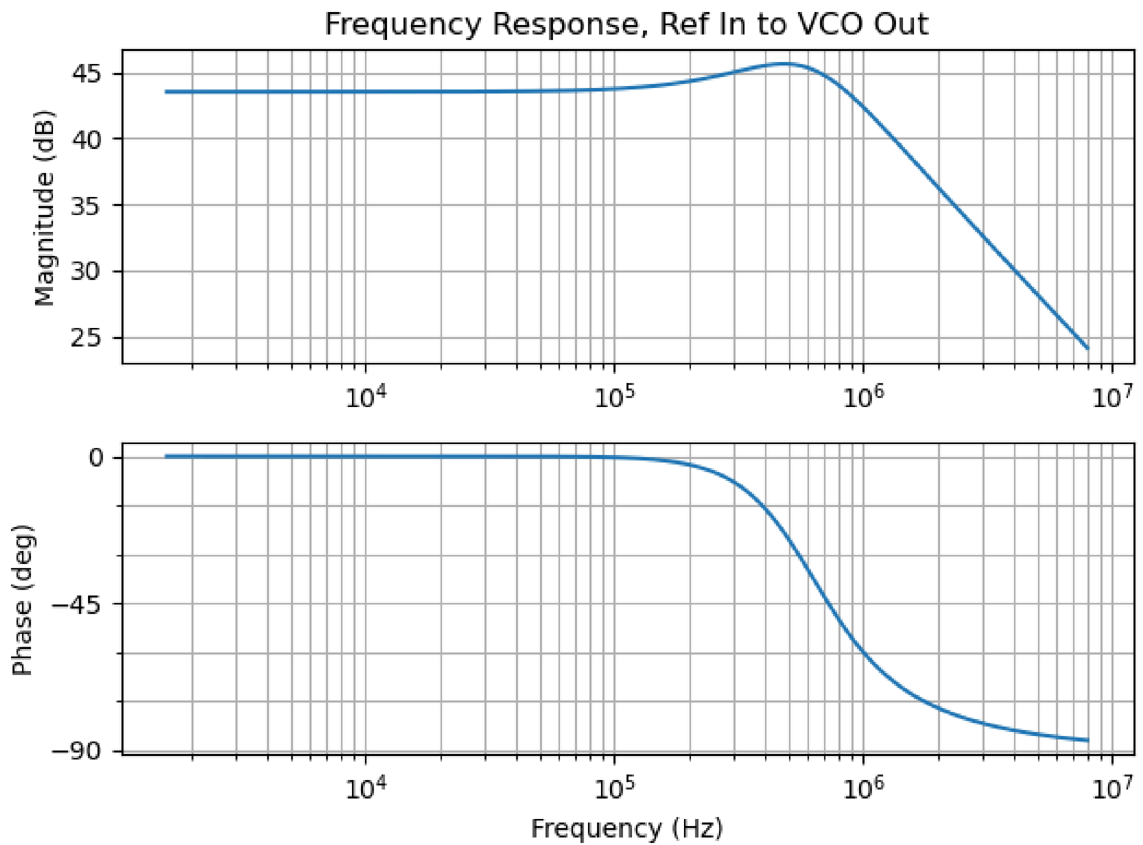
```
plt.grid()
plt.subplot(2,1,2)
plt.plot(*con.step_response(a_gcl2))
plt.xlabel("Time (seconds)")
plt.ylabel("Phase (Radians)")
plt.title("Step Response VCO Out (one rad) to VCO Out")
plt.grid()
plt.tight_layout()
```



Closed Loop Frequency Domain Response

In [34]:
```
plt.figure()
__ = con.bode(a_gcl1, dB=True, Hz=True, omega_limits=[10000, 50e6])
plt.subplot(2,1,1)
plt.title("Frequency Response, Ref In to VCO Out")
plt.tight_layout()
```

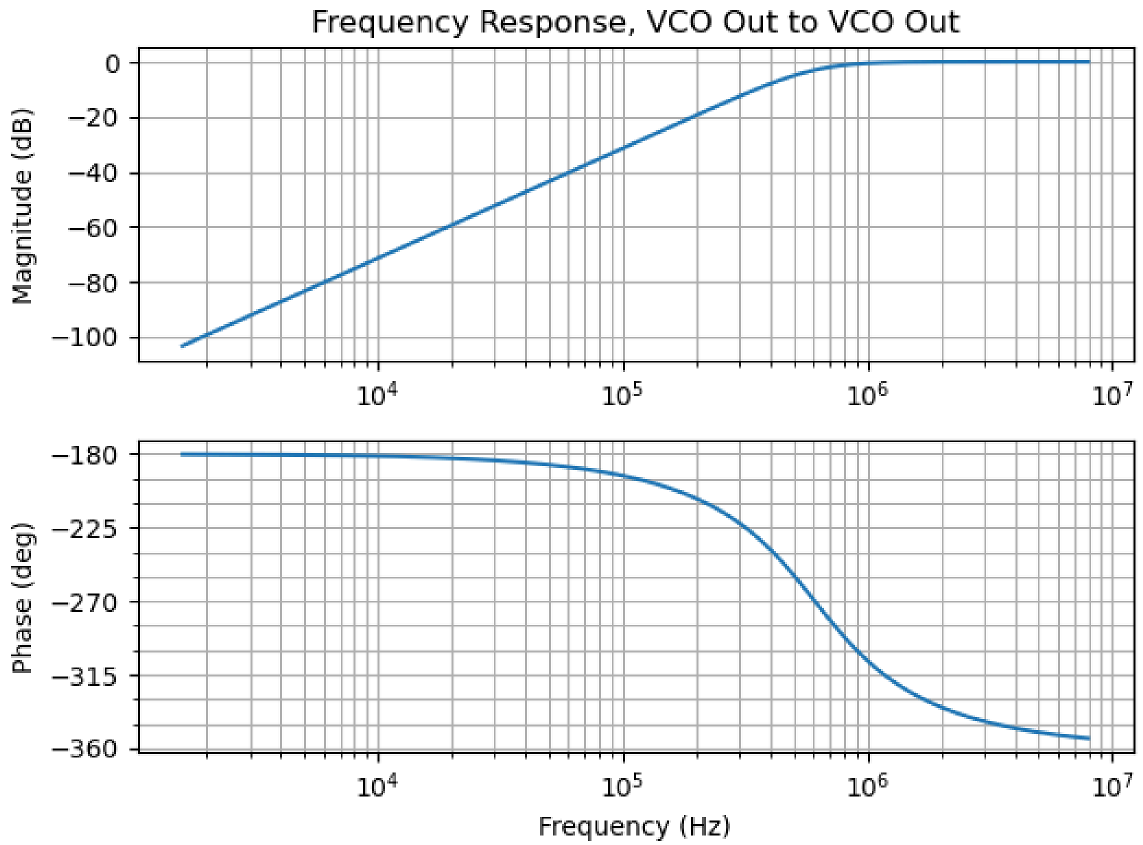
Figure



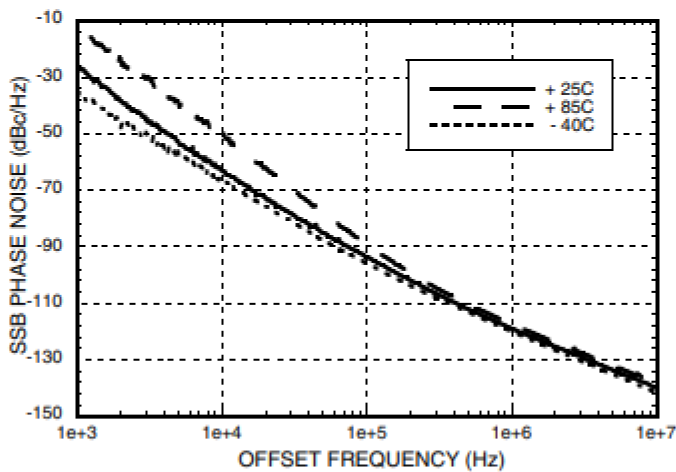
Note that the gain for the lowwer frequency is consistent with $20\text{Log}_{10}\text{N} = 20\text{Log}_{10}(150) = 43.5$ dB consistent with a frequency multiplication from the reference to the output (multiplying frequency multiplies phase).

In [35]:
```
plt.figure()
__ = con.bode(a_gcl2, dB=True, Hz=True, omega_limits=[10000, 50e6])
plt.subplot(2,1,1)
plt.title("Frequency Response, VCO Out to VCO Out")
plt.tight_layout()
```

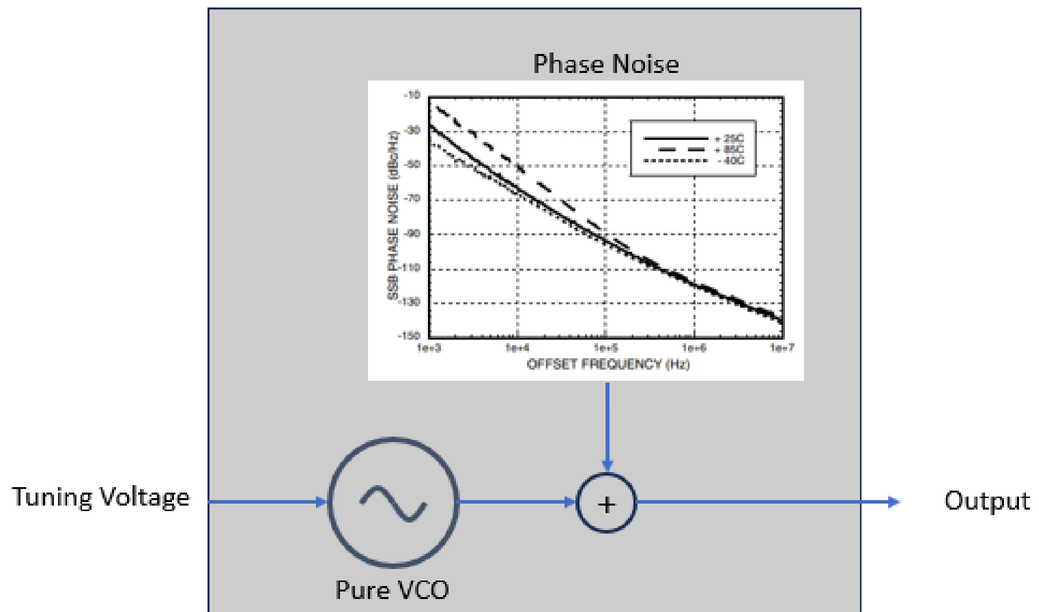
Figure



Note the following phase noise plot from the HMC733 datasheet:



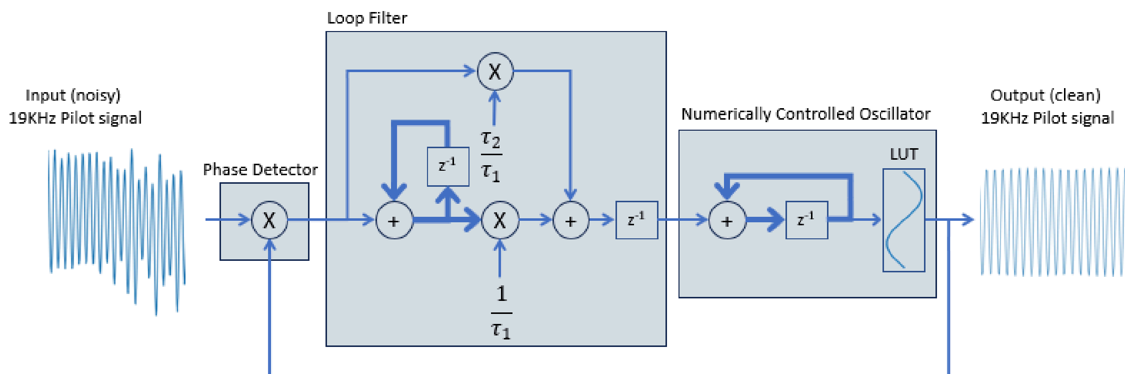
In the loop, the VCO is modelled as a pure VCO (with gain K_V/s) followed by a summation with the phase noise at the output:



Thus the frequency response for "VCO Out to VCO Out" refers to the input at the noise input to this summer, and the output at the output of the summer. For low frequency offsets in phase noise fluctuations, the loop will track the phase noise and thus attenuate it according to the frequency response given (and for the low frequency offsets, it will pass the reference oscillator phase noise with gain according to $20\log_{10}(N)$.

Digital PLL Implementation Model

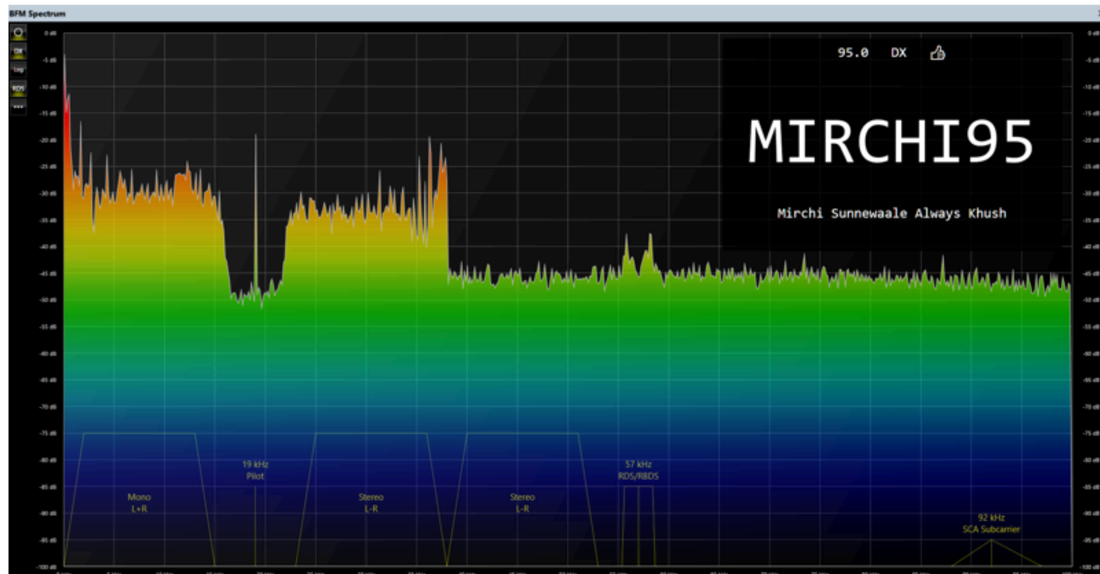
For a demonstration of a Digital PLL, we'll use the implementation below to capture and track the 19 KHz pilot in an FM broadcast signal.



All Digital Phase Lock Loop
Update Rate 192 KHz

```
In [42]: fs = 192e3          # sampling rate

         acc_size = 48       # accumulator size in NCO
         lut_addr=14         # LUT address size in NCO
         lut_out=16          # LUT output size in NCO
```

Test Signal : FM Broadcast Pilot Tone



By SDRConsole by sdr-radio.com - File is a screenshot taken from SDRConsole application showing the Pilot Signal of a FM Broadcast Radio channel., CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=129702149

```
In [37]: # Open FM demodulated multiplexed FM Radio Broadcast signal downloaded from
         # https://www.sigidwiki.com/wiki/FM_Broadcast_Radio

         with wave.open("./data/SDRSharp_20150804_205139Z_0Hz_IQ.wav", 'r') as f:
             # extract and plot waveform
             srate = f.getframerate()
             print(f"Sample rate is {srate/1000} KHz")
             signal = f.readframes(-1)
             # from bytes to int16
             signal = np.frombuffer(signal, dtype = "int16")
             params = f.getparams()

         # seperate I and Q channels to be 2 x array
         fmIQ = signal.reshape(-1,2).T

         fm_wfm = fmIQ[1] / np.std(fmIQ[1])   # signal is almost entirely on fmIQ[1] as a re
```

```
Sample rate is 192.0 KHz
```

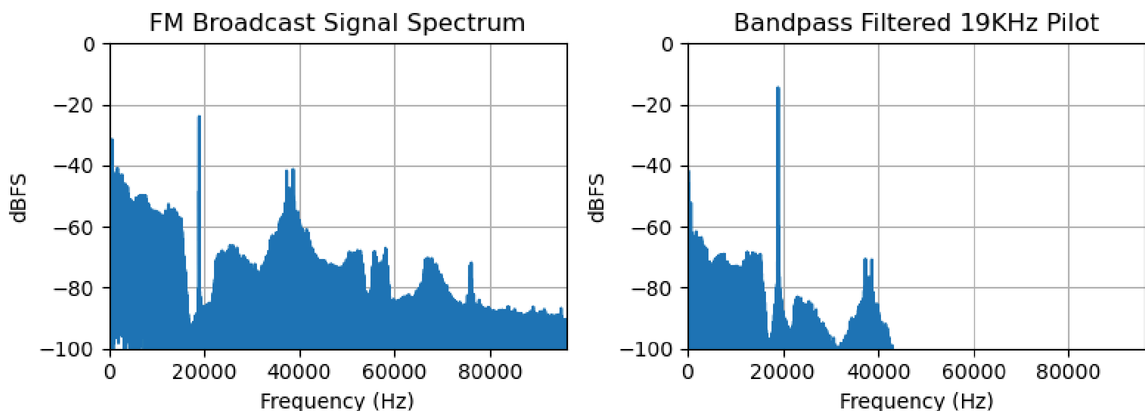
```
In [38]: def plot_spectrum(wfm):
             nsamps = len(wfm)
             win = sig.windows.kaiser(nsamps, 12)

             # the following scales by the coherent gain of the window to provide an accurat
             # of tones dB relative to full scale. It will overestimate the spectrum for noi
             # that is spread over multiple bins. To scale noise accurately, we would instea
```

```
    # by the non-coherent gain of the window (covered more in my DSP for Wireless C
    freq_out = fft.fft(wfm * win) / np.sum(win)
    freq_axis = fft.fftfreq(nsamps)
    freq_out = freq_out[freq_axis>=0]
    freq_axis = freq_axis[freq_axis>=0]

    plt.plot(freq_axis * srate, 20*np.log10(np.abs(freq_out)))
    plt.grid()
```

In [39]:
```
# bandpass filter 19 KHz

r=.99
ftone = 19e3
wn = 2 * np.pi * ftone / srate
pilot = sig.lfilter([1-r], [1., -2*r*np.cos(wn), r**2], fm_wfm)

scale = 1.8*2**(lut_out) /2**(np.std(pilot))  # for scaling pilot to digital precis

pilot =  (pilot * scale).astype('int')
```

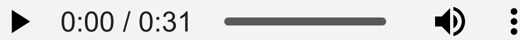
In [40]:
```
plt.figure(figsize=(8,3))
plt.subplot(1,2,1)
plot_spectrum(fm_wfm)
plt.axis([0, srate/2, -100, 0])
plt.title("FM Broadcast Signal Spectrum")
plt.xlabel("Frequency (Hz)")
plt.ylabel("dBFS")

plt.subplot(1,2,2)
plot_spectrum(pilot/2**(lut_out-1))
plt.axis([0, srate/2, -100, 0])
plt.title("Bandpass Filtered 19KHz Pilot")
plt.xlabel("Frequency (Hz)")
plt.ylabel("dBFS")
plt.tight_layout()
plt.show()
```

Figure



In [20]:
```
Audio(fm_wfm, rate = srate)
```

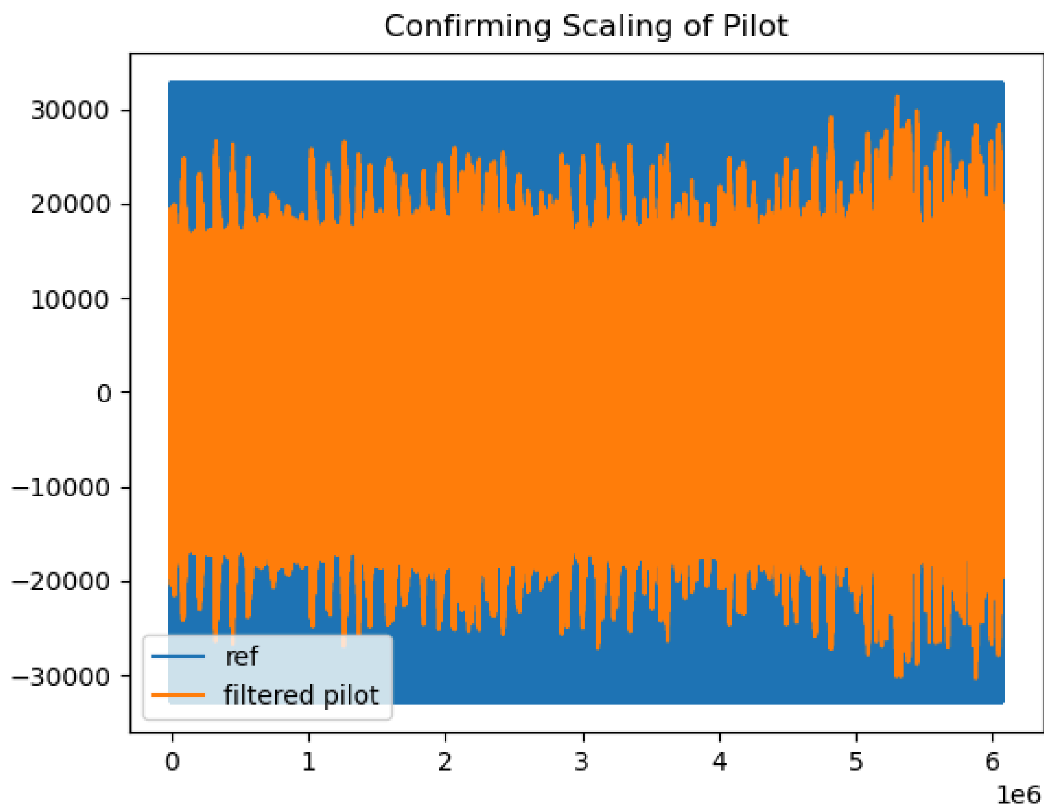
Out[20]:



In [41]:

```
# compare scale of filtered pilot to "clean" reference signal
n = np.arange(len(pilot))
clean = 2**(lut_out-1)*np.cos(2*np.pi*19e3/srate *n)
plt.figure()


plt.plot(clean, label="ref")
plt.plot(pilot, label="filtered pilot")
plt.title("Confirming Scaling of Pilot")
plt.legend(loc="lower left");
```

Figure



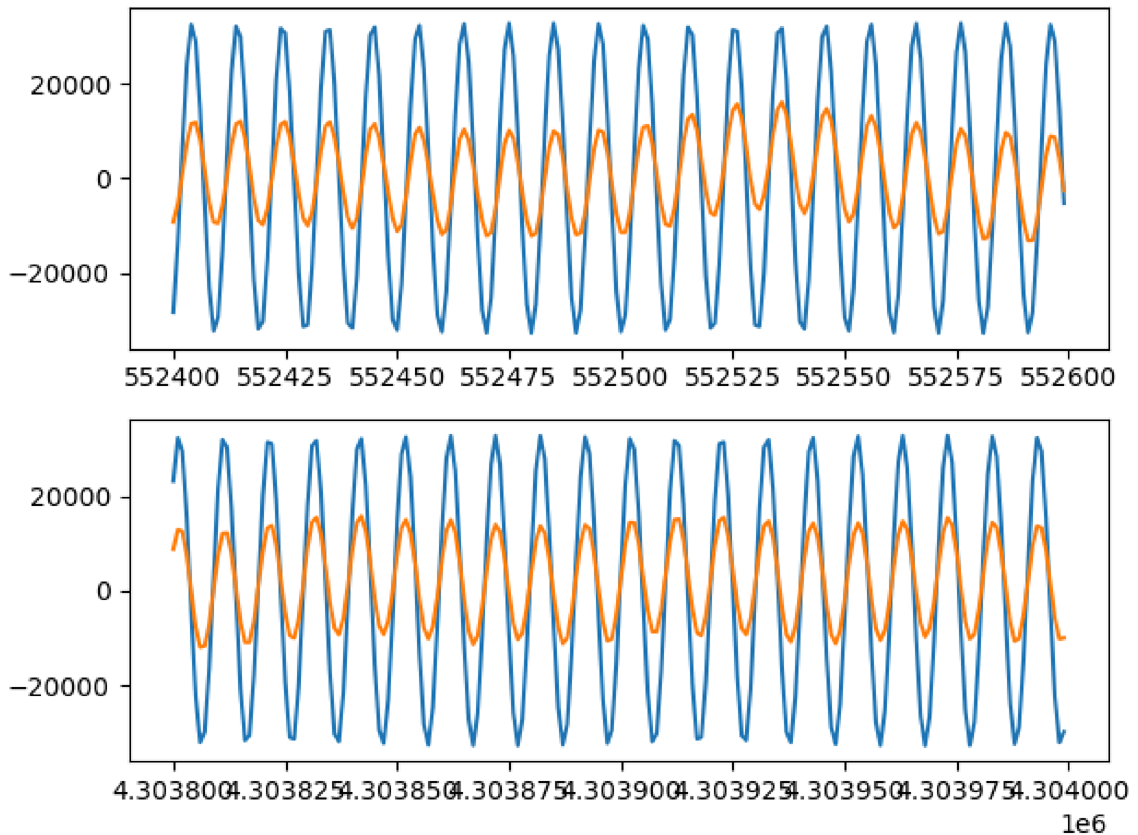
The Pilot and Reference are not locked above as we see below with a zoom in at two arbitrary locations near the start and end of the sequence:

In [43]:

```
range1 = np.arange(552400,552600)
range2 = np.arange(4303800,4304000)
plt.figure()
plt.subplot(2,1,1)
plt.plot(range1, clean[range1], label="ref")
plt.plot(range1, pilot[range1], label="filtered pilot")
plt.subplot(2,1,2)
plt.plot(range2, clean[range2], label="ref")
```

```
plt.plot(range2, pilot[range2], label="filtered pilot")
plt.tight_layout()
```

Figure



If we multiply and filter the above signals, we can see what the phase is of the reference relative to the 19 KHz pilot prior to locking to it:

In [104…

```
def phase_det(tone, ref, f, fs, ntaps=91, fpass=None, fstop=None):
    '''
    running phase detector of x relative to y
    x: tone (1darray)
    y: reference (1darray)
    f: (approximate) frequency of tone (float)
    fs: sampling rate (float) same units as f
    ntaps: number of taps in filter
    fpass: filter passband corner
    fstop: filter stopband corner
    '''
    if fpass is None:
        fpass = 0.8 * f
    if fstop is None:
        fstop = f

    phase = (np.sign(tone) * np.sign(ref))* np.pi

    # filter to pass difference signal as phase and reject sum signal as 2f:
    coeff = sig.firls(ntaps, [0, fpass, fstop, fs/2], [1, 1, 0, 0], fs=fs)
```

```
    # zero phase filter
    result = sig.filtfilt(coeff, 1, phase)
    return result


filtered_phase = phase_det(pilot, clean, ftone, fs, ntaps = 501, fpass = 500, fstop
```

Time Sequenced Component Model

This is not the Loop Model but a model of the actual implementation.

Below is a bit and cycle accurate Component Object model. A Component Object takes inputs and provides outputs on each sample of a "master clock" for discrete time time stepped simulations. Modelling with Component Objects is detailed in my course "Python Applications for Digital Design and Signal Processing". This is a simulation of the actual implementation which would capture non-linear effects, and after we'll develop the much simpler Loop Model for comparison.

In [55]:

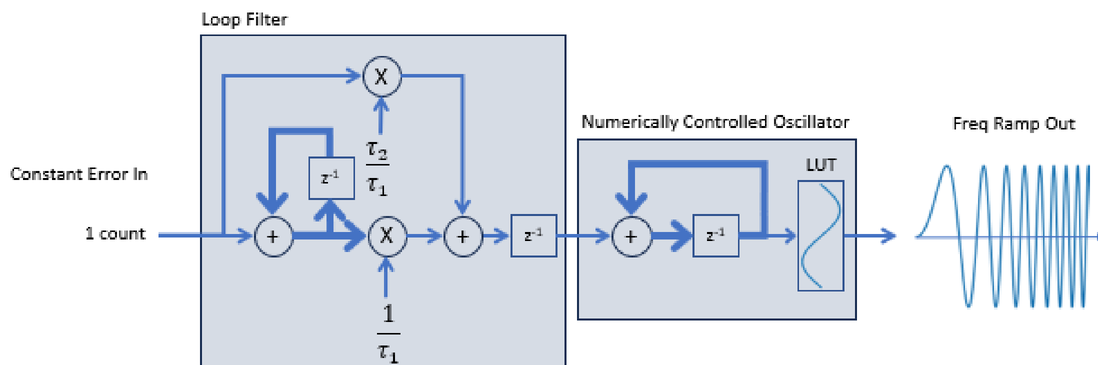
```
# NCO Component:

def Nco(sum1=0, acc_size=28, lut_addr=14, lut_out=16):
    '''
    NCO as a Component Object
    Parameters are object initialation:
    sum1: initial state (count) for accumulator
    acc_size: accumulator precision in bits (wrap on overflow)
    lut_addr: look-up table address precision in bits
    lut_out: look-up table data precision in bits

    (fcw input size is one less than acc_size)
    Dan Boschen 9/25/2023

    To use:
    instantiate:  my_nco= NCO(....)
    prime:        my_nco.send(None)
    pass in fcw and pcw samples and get sample out for each clock cycle:
                  output = my_nco.send((fcw, pcw))

    '''
    data = None
    while True:
        fcw, pcw = yield data

        sum2 = (sum1 + pcw) % 2**acc_size         # max bit width acc_size
        sum2 = sum2 // 2**(acc_size - lut_addr)   # phase truncation
        sum1 = (sum1 + fcw) % 2**acc_size     # modulo acccumulator

        sine = math.sin((2 * math.pi * sum2) / 2**lut_addr)
        # maps -1/+1 sine to the signed digital range -2**(lut_out-1) to 2**(lut_ou
        data = round(((sine+1)/2  * (2**(lut_out)-1)- (2**(lut_out)-1)/2)-.5)
```

In [56]:
```python
# Loop Filter Component

def PropIntFilter(accum, integral, proportional):

    sum_out = accum

    while True:
        # allows for updating integral and proportional gain on each input sample
        error_sig = yield sum_out

        accum += error_sig
        sum_out = integral * accum + proportional * error_sig
```

In [57]:
```python
# Top Level DPLL Component Model

def Dpll(nco, loopfilter, integral, bitw):
    '''
    nco: instantiated and primed NCO Component: requires fcw, pcw inputs and provid
    fcw: initial state for nco input
    loopfilter: instantiated and primed Loop Filter Component: requires err input a
    bitw: bit width of input and output (currently limited to be the same)
    '''
    while True:
        signal_in = yield signal_out

        # phase detector
        phase_err = int((signal_in * signal_out)/2**(bitw-1))      # scales back to

        # loop filter
        fcw = loopfilter.send(phase_err)

        # NCO
        signal_out = nco.send((fcw, 0))    # no phase change input (pcw) used
```

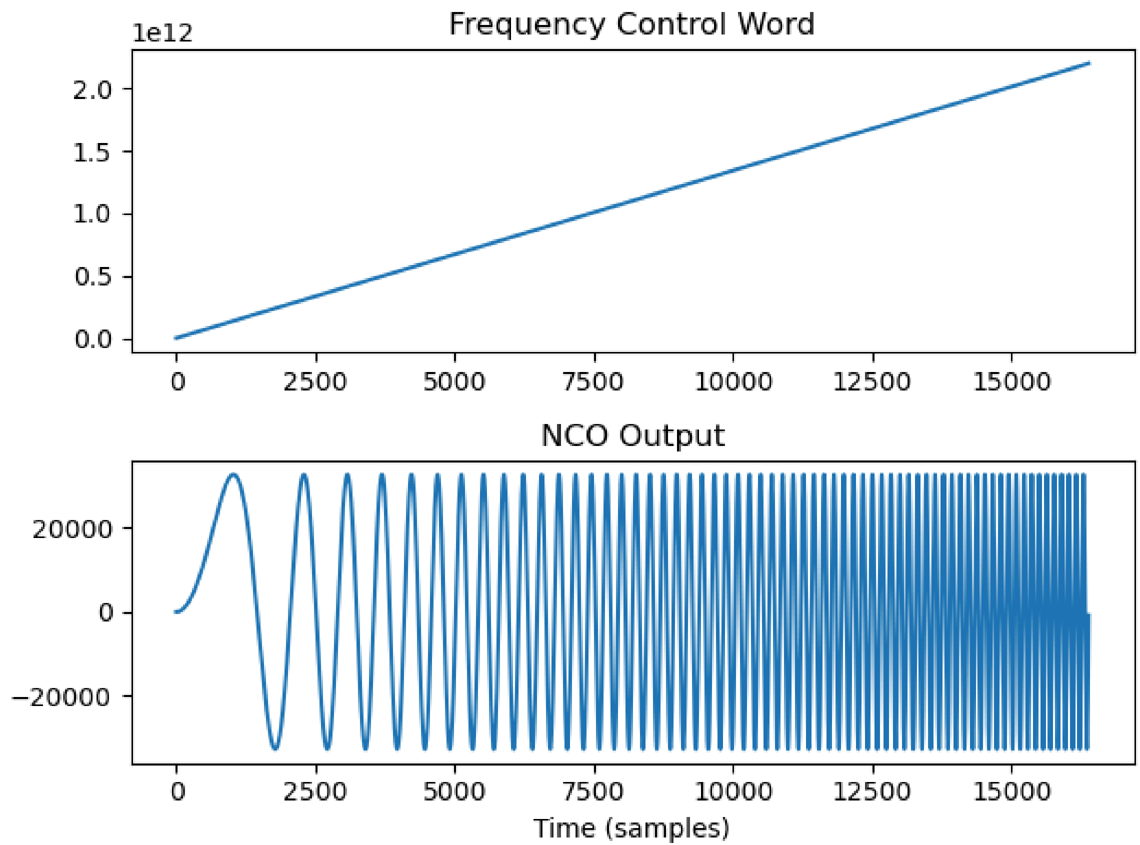
Functional Tests

Functional Test of Loop Filter and NCO

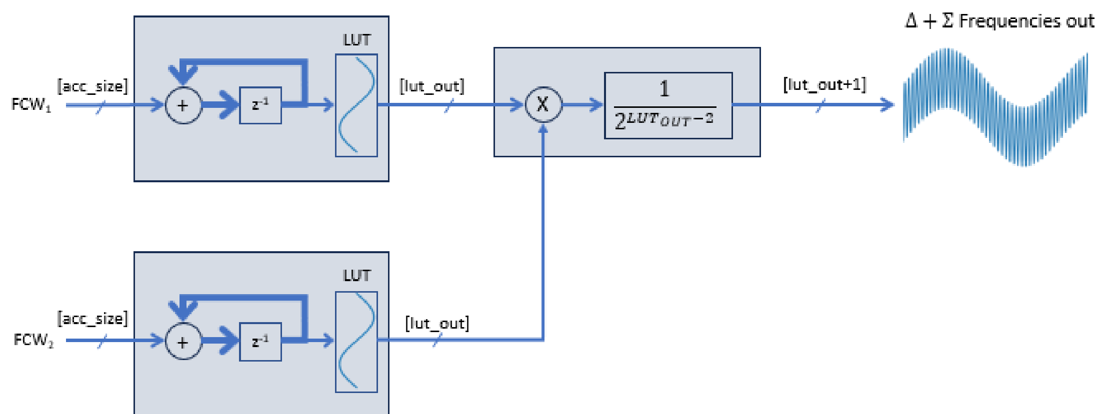


In [58]:

```
# Functional Test of Loop Filter and NCO
# Simple Open Loop Test with P=0 (integrate only) resulting in ramping FCW


nsamps = 2**14     # number of samples ot simulate

# instantiate and prime components
nco = Nco(sum1=0, acc_size=acc_size, lut_addr=lut_addr, lut_out=lut_out)
nco.send(None)

loop_filter = PropIntFilter(accum=0, integral= 2**(acc_size-lut_out-5), proportiona
loop_filter.send(None)

# run sim
result = []
fcw_result = []

error = 1
for n in range(nsamps):
    fcw = loop_filter.send(error)
    fcw_result.append(fcw)
    result.append(nco.send((fcw,0)))

# plot results
plt.figure()
plt.subplot(2,1,1)
plt.plot(fcw_result)
plt.title("Frequency Control Word")
plt.subplot(2,1,2)
plt.plot(result)
plt.title("NCO Output")
plt.xlabel("Time (samples)")
plt.tight_layout()
```



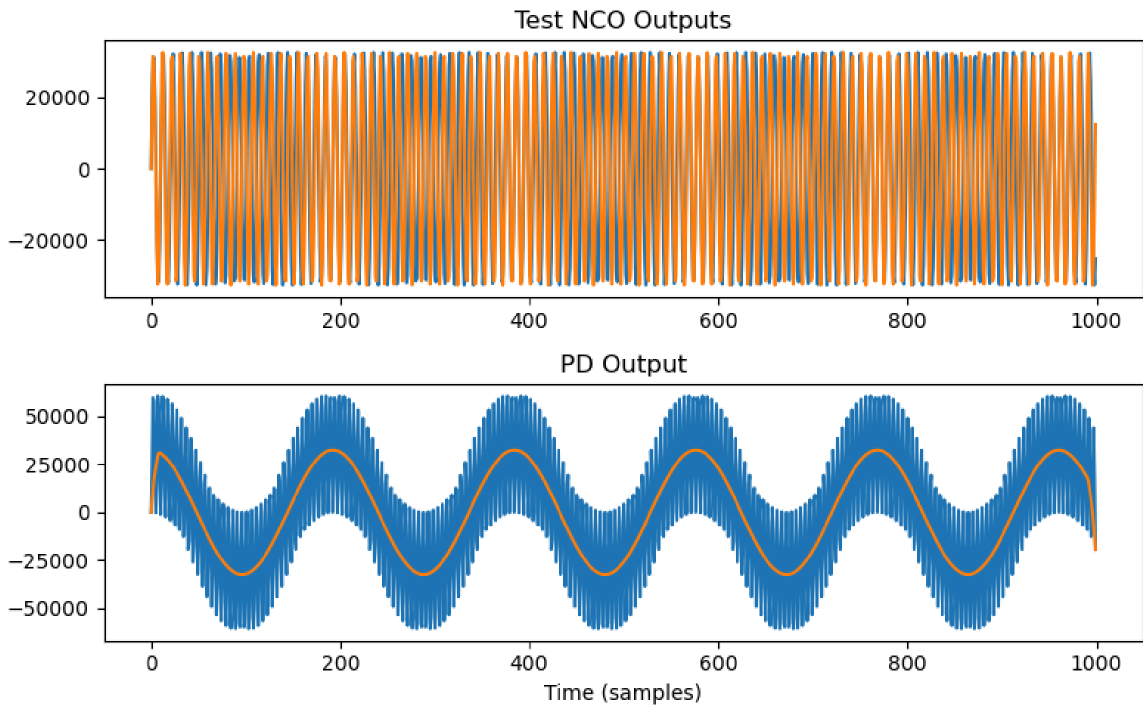
Functional Test of NCO and Phase Detector



In [59]:
```
# Phase Detector

nsamps = 1000

# create two NCO's offset in frequency
# set initial fcw for 19KHz
fcw1 = int(19e3 * (2**acc_size)/fs)       # frequency for nco1
fcw2 = int(20e3 * (2**acc_size)/fs)       # frequency for nco2
print(f"{fcw1=}")
print(f"{fcw2=}")
```

```
# instantiate and prime components
nco1 = Nco(sum1=0, acc_size=acc_size, lut_addr=lut_addr, lut_out=lut_out)
nco1.send(None)

nco2 = Nco(sum1=0, acc_size=acc_size, lut_addr=lut_addr, lut_out=lut_out)
nco2.send(None)

loop_filter = PropIntFilter(accum=0, integral= 2**(acc_size-lut_out-5), proportiona
loop_filter.send(None)

# run sim:
result_nco1 = []
result_nco2 = []
result_pd = []

for n in range(nsamps):
    fcw = loop_filter.send(error)
    fcw_result.append(fcw)
    nco1_out = nco1.send((fcw1,0))
    nco2_out = nco2.send((fcw2,0))
    phase_err = int((nco1_out * nco2_out)/2**(lut_out-2))
    result_nco1.append(nco1_out)
    result_nco2.append(nco2_out)
    result_pd.append(phase_err)
    fcw = loop_filter.send(phase_err)

# plot results
plt.figure(figsize=(8,5))
plt.subplot(2,1,1)
plt.plot(result_nco1)
plt.plot(result_nco2)
plt.title("Test NCO Outputs")
plt.subplot(2,1,2)
plt.plot(result_pd)
plt.plot(sig.filtfilt(np.ones(10), 10, result_pd))
plt.title("PD Output")
plt.xlabel("Time (samples)")
plt.tight_layout()
```

```
fcw1=27854294570325
fcw2=29320310074026
```



Closed Loop Simulation

In [129…

```
nsamps = int(8*fs)    # number of samples ot simulate; -1 = all samples

print(f"{nsamps=}")
print(f"{fs=}")
print(f"{acc_size=}")
print(f"{lut_addr=}")
print(f"{lut_out=}")



# instantiate and prime components
nco = Nco(sum1=0, acc_size=acc_size, lut_addr=lut_addr, lut_out=lut_out)
nco.send(None)


tau1 =  3.756604e-5      # from d_tau1 in Digital Phase Lock Loop Model
tau2 =  305.57749        # from d_tau2...
print(f"{tau1=:0.5f}")
print(f"{tau2=:0.5f}")

# other values determined:
#  LBW          tau1          tau2
#  20 Hz        0.0037566     3055.774907364391
#  100 Hz       0.00015       611.154981472878
#  200 Hz       3.756604e-5   305.57749

integral = int(1 / tau1)
print(f"{integral=:}")
```

```
proportional = int(tau2 / tau1)
print(f"{proportional=:}")

fcw_start = int(19e3 * (2**acc_size)/fs)
print(f"{fcw_start=}")
accum_state = fcw_start * tau1   # sets initial state at 19KHz
loop_filter = PropIntFilter(accum=accum_state, integral= integral, proportional=pro
loop_filter.send(None)

# run sim
result = []
test_point = []

nco_out =0
error = 1
for sample in pilot[:nsamps]:             #use pilot, clean,
    phase_err = (sample * nco_out) /2**(lut_out-1)
    # phase_err = np.sign(sample) * nco_out /2
    fcw = loop_filter.send(phase_err)
    nco_out = nco.send((fcw, 0))
    test_point.append(fcw)
    result.append(nco_out)
```

```
nsamps=1536000
fs=192000.0
acc_size=48
lut_addr=14
lut_out=16
tau1=0.00004
tau2=305.57749
integral=26619
proportional=8134407
fcw_start=27854294570325
```

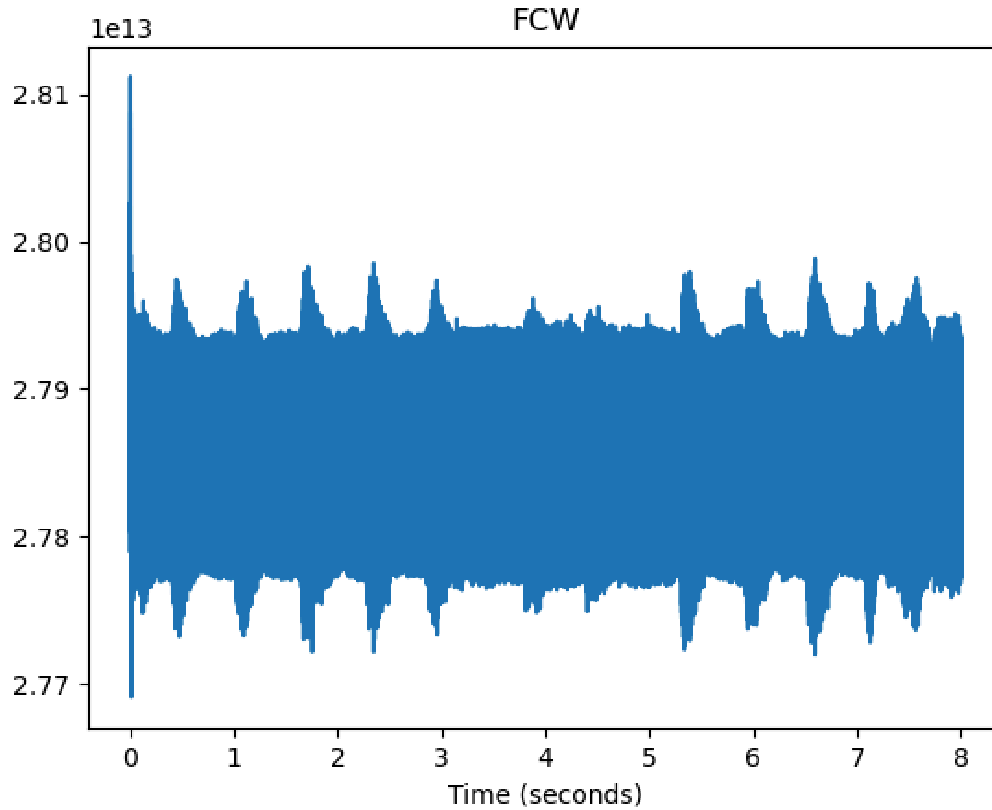
In [130…

```
plt.figure()
#plt.plot(result)
time = np.arange(len(test_point))/fs
plt.plot(time, np.array(test_point))
plt.xlabel("Time (seconds)")
plt.title("FCW")
```

Out[130…

```
Text(0.5, 1.0, 'FCW')
```

Figure



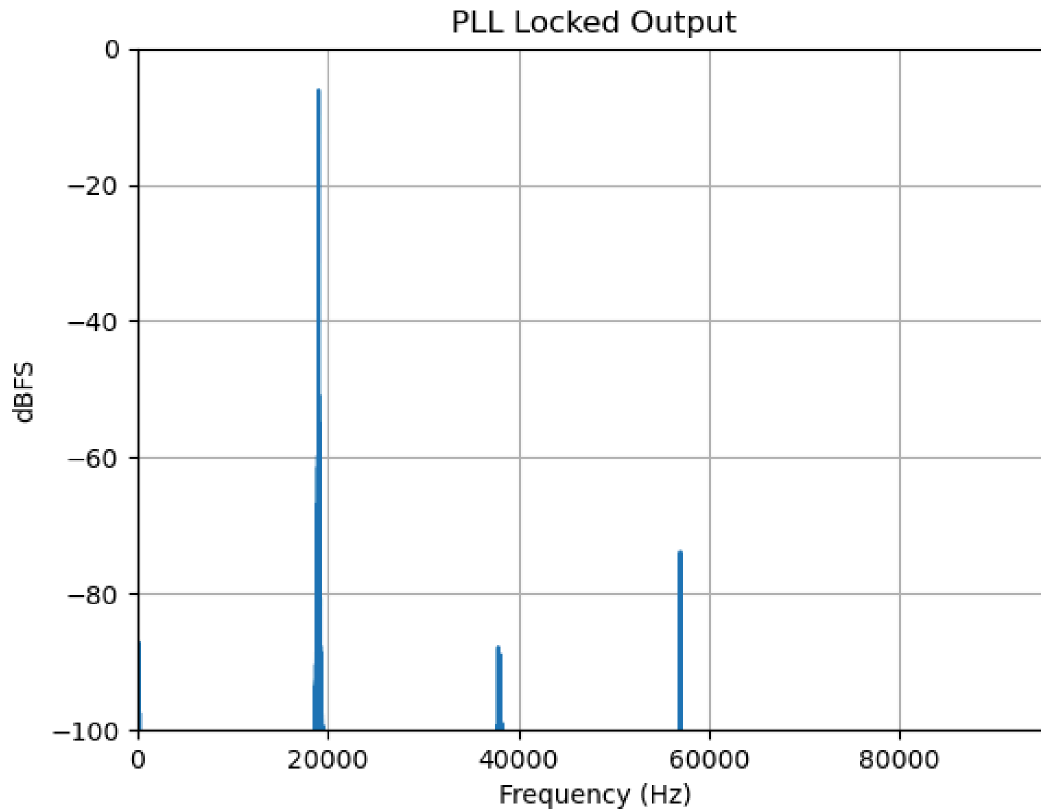
In [131…

```
plt.figure()
plot_start = int(.5*fs)
#plt.subplot(1,2,1)
plot_spectrum(np.array(result[plot_start:])/2**(lut_out-1))
plt.axis([0, srate/2, -100, 0])
plt.title("PLL Locked Output")
plt.xlabel("Frequency (Hz)")
plt.ylabel("dBFS")
```

Out[131…

```
Text(0, 0.5, 'dBFS')
```

Figure



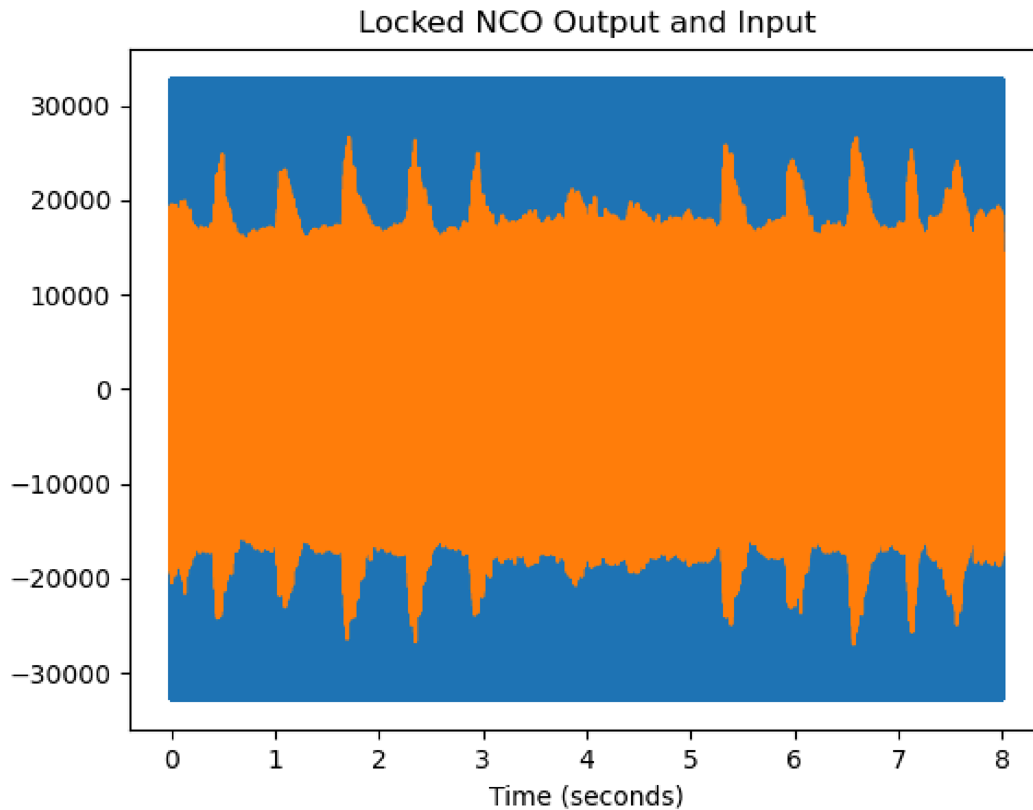
In [132…

```
plt.figure()

plt.plot(time, result)
plt.plot(time, pilot[:nsamps])
plt.title("Locked NCO Output and Input")
plt.xlabel("Time (seconds)")
```

Out[132…

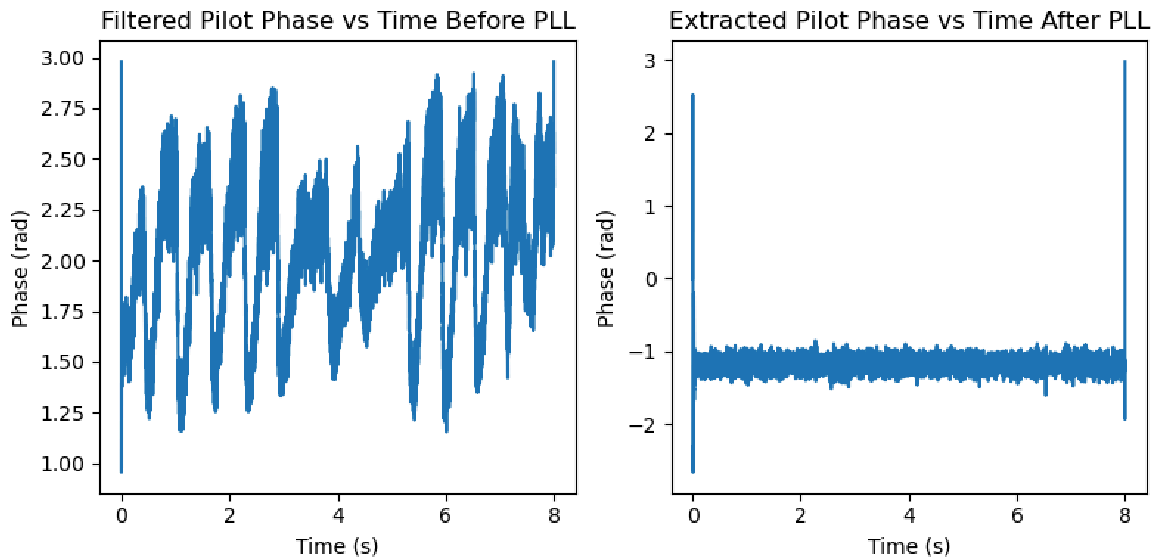
```
Text(0.5, 0, 'Time (seconds)')
```

Figure

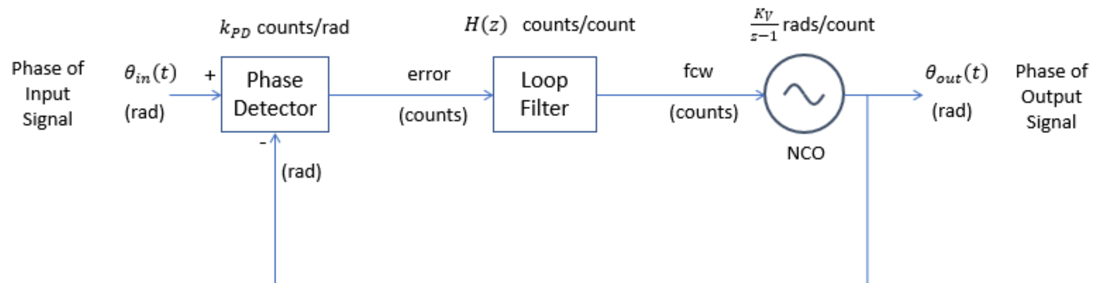


In [133…

```
filtered_phase_OL = phase_det(pilot[:nsamps], clean[:nsamps], ftone, fs, ntaps = 50
time_axis = np.arange(nsamps)/fs
filtered_phase_CL = phase_det(pilot[:nsamps], result, ftone, fs, ntaps = 501, fpass
plt.figure(figsize=(8,4))
plt.subplot(1,2,1)
plt.plot(time_axis , filtered_phase_OL)
plt.xlabel("Time (s)")
plt.ylabel("Phase (rad)")
plt.title("Filtered Pilot Phase vs Time Before PLL")
#plt.axis(
plt.subplot(1,2,2)
plt.plot(time_axis, filtered_phase_CL)
plt.xlabel("Time (s)")
plt.ylabel("Phase (rad)")
plt.title("Extracted Pilot Phase vs Time After PLL")
plt.tight_layout()
```



Digital Phase Lock Loop Model



Note the units used for error and FCW here are actual counts, so will match the digital values at those nodes.

Open Loop Gain:

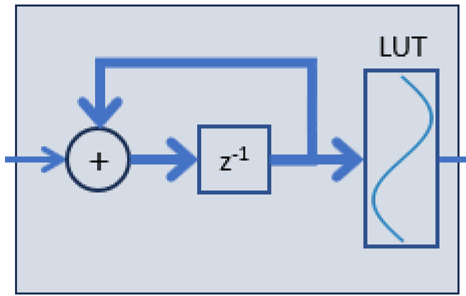
$$G_{OL}(z) = \frac{k_V k_{PD}}{z-1} H(z)$$

In [134…

```
fs = 192e3   # sampling rate in Hz. We'll use normalized radian frequency in the mo
d_lbw = 2*np.pi* 200/fs         # target loop bw in rad/sample
```

NCO

Numerically Controlled Oscillator



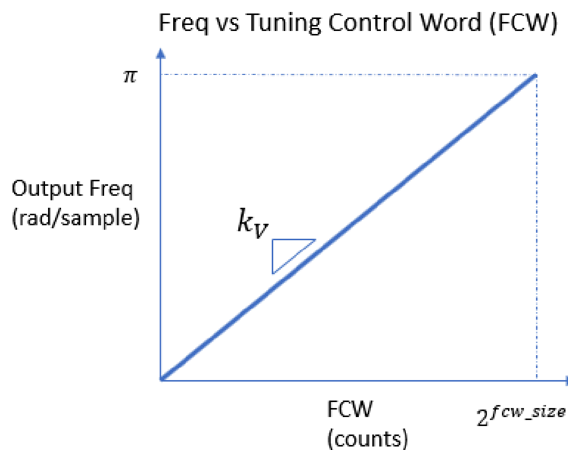
Input on left is Frequency Control Word (FCW) Output on right is the digitized sinusoid as the output of a Look-up Table (LUT) effectively containing one cycle of a sine wave.

For a small FCW, the accumulator will ramp up slowly. For a large FCW, the accumulator will ramp up more rapidly.

The Most Significant Bits of the accumulator are used as the address for the LUT. The accumulator wraps around on overflow, and thus produces a digitized sinusoidal output waveform with a frequency directly proportional to FCW, with a full range of DC to half the sampling rate.

Given a PLL implementation, we will work in units of phase, not frequency. In this context, the NCO, like the VCO, is an integrator, as a "phase accumulator". The NCO gain for the loop model is $k_V/(z-1)$, where k_V is the slope of the output frequency in radians/sample verus the frequency control word FCW. Note similarity of VCO gain for analog loop as $\frac{K_V}{s}$.

The frequency vs control word sensitivity is as shown in the plot below, resulting in



NCO Loop Model is

$$\frac{k_V}{z-1} \text{ rads/count}$$

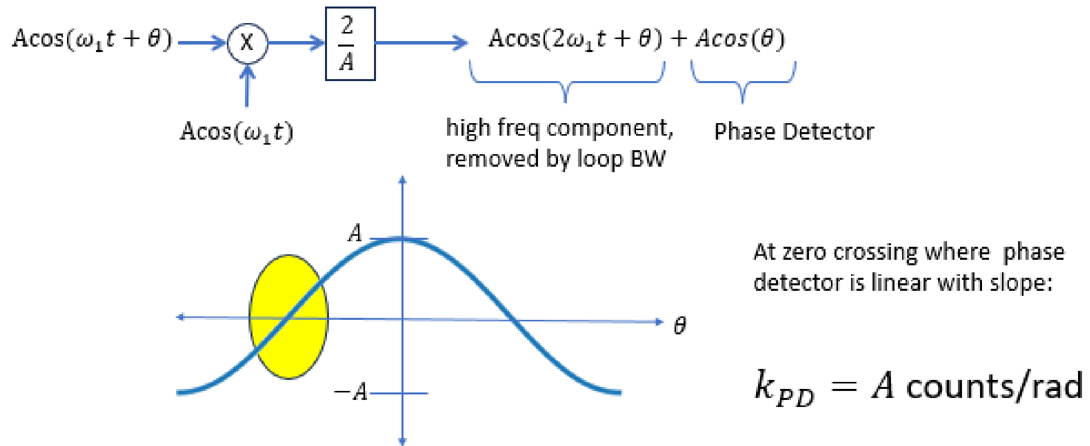
$$\text{Where } k_V = \frac{\pi}{2^{fcw_size}}$$

This is a good example of how the mapping from s to z for poles and zeros in vicinity of $z = 1$ is simply $s \leftrightarrow z-1$ when working in units of normalized frequency such that the time index is in samples ($T = 1$).

In [135...

```
# NCO
accum_size = 48
fcw_size = 47


d_kv= np.pi/2**fcw_size          # NCO gain in rad/count
```

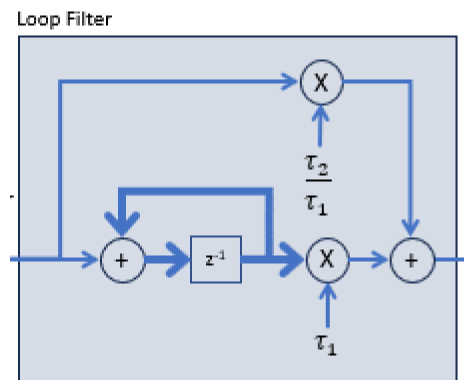
Phase Detector



In [136...

```
precision = 16                # precision of both phase detector inputs
d_kpd = 2**(precision-1)      # Phase detector gain counts/rad
```

PI Loop Filter



$$H(z) = P + I\frac{1}{z-1} = \frac{\tau_2}{\tau_1} + \frac{1}{\tau_1}\frac{1}{z-1}$$

$$= \frac{\tau_2(z-1)+1}{\tau_1(z-1)} = \frac{\tau_2 z + 1 - \tau_2}{\tau_1(z-1)}$$

Open Loop Gain

$$G_{OL}(z) = \frac{k_V k_{PD}}{z-1} H(z)$$

$$= (\frac{k_V k_{PD}}{z-1})(\frac{\tau_2 z + 1 - \tau_2}{\tau_1 (z-1)})$$

$$= (\frac{k_V k_{PD}}{\tau_1})(\frac{\tau_2 z + 1 - \tau_2}{(z-1)^2})$$

In [137…

```
# since we'll iterate on gain constants, make the open loop gain a function

def gol_digital(tau1, tau2):
    return (d_kv * d_kpd / tau1) * con.tf([tau2, (1-tau2)], [1, -2, 1], dt=1/fs)

# setting dt is what makes this a transfer function in z instead of s (digital inst
# setting dt will not affect the decision to use normalized frequency or not (gains
# but will effect the units on the horizontal axis for Bode plots
```

Starting Loop Values

Like we did for the analog 2nd order PLL, we'll first set $\tau_2 = 0$ and adjust τ_1 (primary gain control) such that the zero dB gain crossing is right at the loop bandwidth.

With $\tau_2 = 0$, the open loop gain simplifies to:

$$G_{OL}(z)|_{\tau_2=0} = (\frac{k_V k_{PD}}{\tau_1 (z-1)^2})$$

Similar to estimating τ_1 in the analog loop, but with added complexity of the unit circle on the z-plane being the frequency axis. Therefore we set $z = e^{j\omega_c}$ (like we set $s = \omega_c$ for the analog loop), and determine τ_1 such that $|G_{OL}(z)| = 1$

This becomes:

$$\tau_1 = \left| \frac{k_V k_{PD}}{(e^{j\omega_c} - 1)^2} \right|$$

Assuming a positive k_V and k_{PD} (when negative that is considered the negative feedback for the loop and only positive gain values are used), then

$$\tau_1 = \frac{k_V k_{PD}}{|(e^{j\omega_c} - 1)|^2}$$

Note for $\omega_c << 1$, $|(e^{j\omega_c} - 1)|^2 \approx \omega_c^2$ (looking at that graphically on the complex plane provides great intuition for this) and for these cases we end up with a similar equation to the analog loop:

$$\tau_1 \approx \frac{k_V k_{PD}}{\omega_c^2}, \text{ for } \omega_c << 1$$

This is intuitively pleasing as we would expect the loop models to match the analog models if we significantly oversample the loop. Since we are dealing with normalized frequencies in the digital case (divide by the sampling rate), as the sampling rate increases, ω_n will get increasingly smaller for the same loop bandwidth in Hz.

Since we are iterating after setting the initial values, this will be a sufficient estimate even for higher frequency cases.

We'll then add the zero at (45° phase margin) or slightly below (higher phase margin) the loop bandwidth for stability.

This will increase the bandwidth slightly, so then iterate on both from these starting values to decrease the loop gain using τ_1, and increase or decrease τ_2 while observing response on Bode plot for desired gain and phase margin.

In [139…

```
print(f"Target loop bw = {d_lbw:0.5f} rad/sample")
print(f" = {d_lbw/(2*np.pi):0.4f} cycles/sample")

d_tau1_init = (d_kv * d_kpd)/d_lbw**2
print(f"Initial value for tau1 = {d_tau1_init:0.2e}")
```

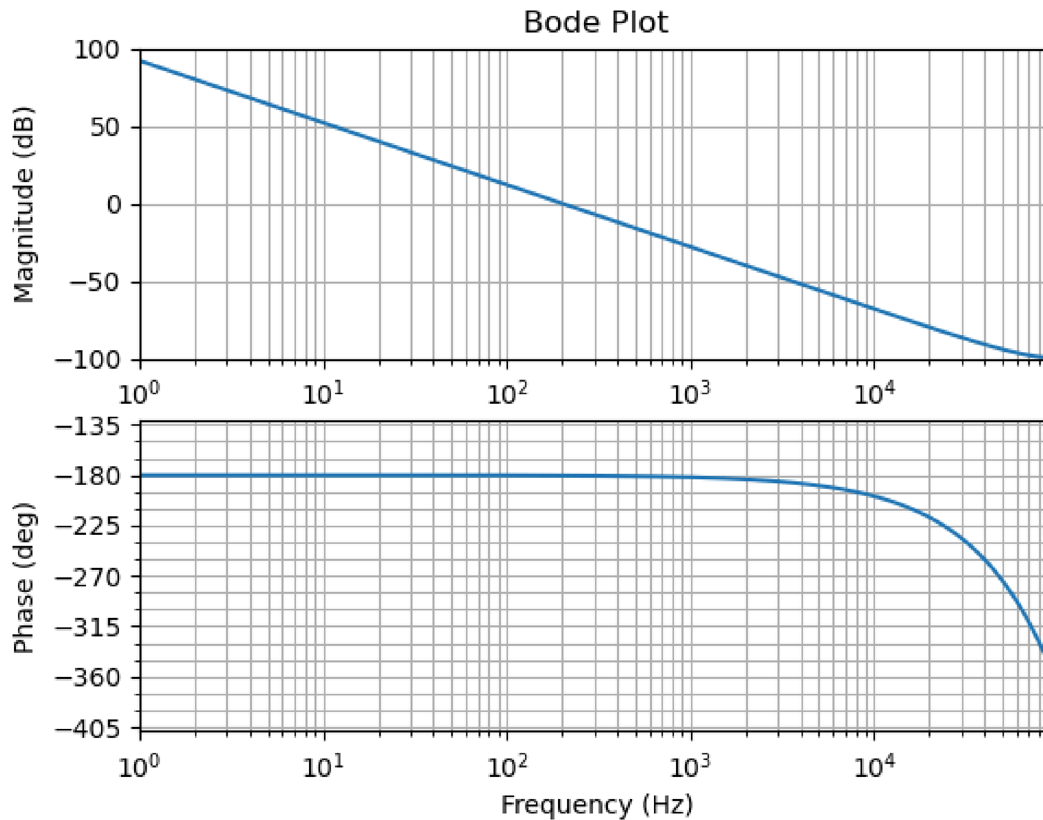
```
Target loop bw = 0.00654 rad/sample
 = 0.0010 cycles/sample
Initial value for tau1 = 1.71e-05
```

In [142…

```
# To demonstrate show Bode Plot with tau2=0 resulting in the cascade of two integra
d_tau2=0
d_gol = gol_digital(d_tau1_init, d_tau2)
print(d_gol)
plt.figure()

__ = con.bode(d_gol, Hz=True, dB=True)
plt.subplot(2,1,1)
plt.title("Bode Plot")
plt.axis([1, fs/2, -100, 100]);
plt.subplot(2,1,2);
#
```

```
  4.284e-05
-------------
z^2 - 2 z + 1

dt = 5.208333333333333e-06
```

Figure

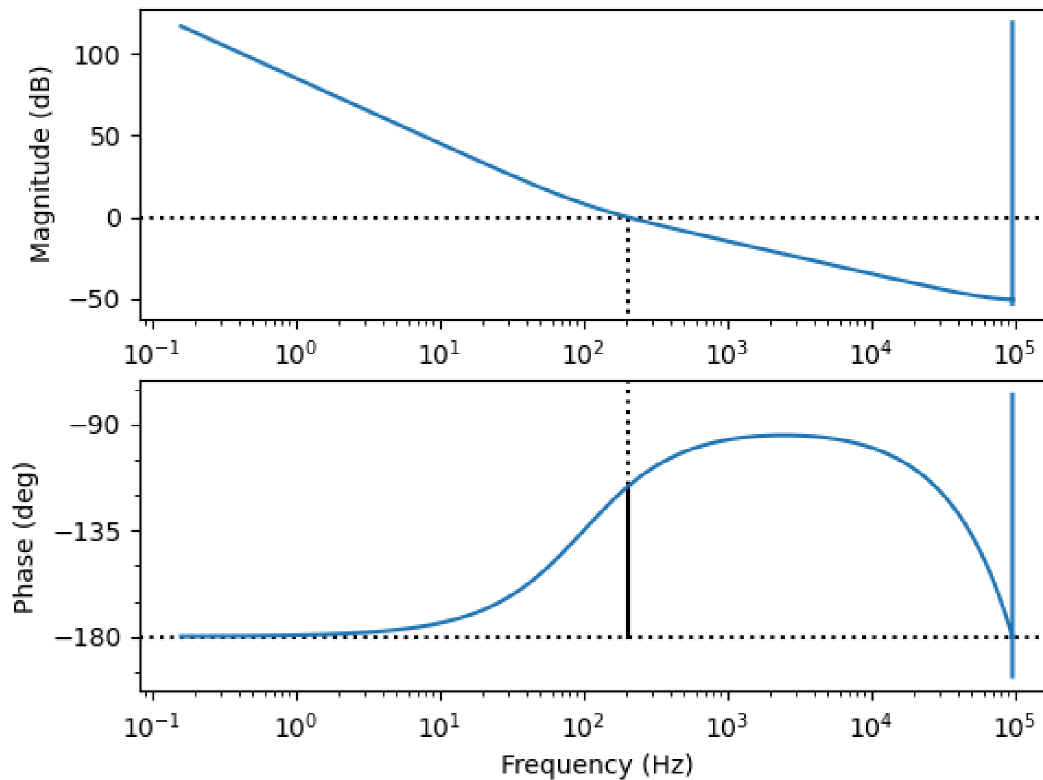


Note the additional lagging phase due to the parastic z^{-1} delays in the implementation. This will limit the minimum sampling rate to loop bandwidth ratio.

In [150…

```
# inial values were tau2 = 1/lbw and tau1 = 1.4 x tau1 computed above for a 45 degr
# then to increase phase margin to increase the damping factor and keep the same lo
# end result after interating: tau2 = 4fs/lbw, tau1 = 4 x tau1 computed above

d_tau2 =  2 / d_lbw             # adjusts phase as 1/tau2, this will change the zeo
d_tau1 =  2.2 * d_tau1_init       # adjusts gain as 1/tau1

print(f"{d_tau2=}")
print(f"{d_tau1=}")

d_gol = gol_digital(d_tau1, d_tau2)

plt.figure()

__ = con.bode(d_gol, dB=True, Hz=True, margins=True, method='frd', omega_limits=[1,
plt.subplot(2,1,1)
plt.title("Bode Plot")
plt.show()
```

```
d_tau2=305.577490736439
d_tau1=3.756604043507013e-05
```



The vertical line in plots above on right is to show Nyquist, and is not part of the response.

Ignore the reported gain margin since the phase didn't cross 180 degrees it was unable to detect the margin (add an extra delay sample delay to the transfer function by changing denominator to [dpll.tau1, -dpll.tau1, 0] to see proper gain and phase margin computation for that case. What is significant in the above plot is the phase margin and showing us the zero crossing close to 400 Hz.

Closed Loop

In [146…

```
# Closed Loop from Ref Input to VCO Output

d_gcl1 = con.minreal(d_gol/(1+d_gol))

print(d_gcl1)
```

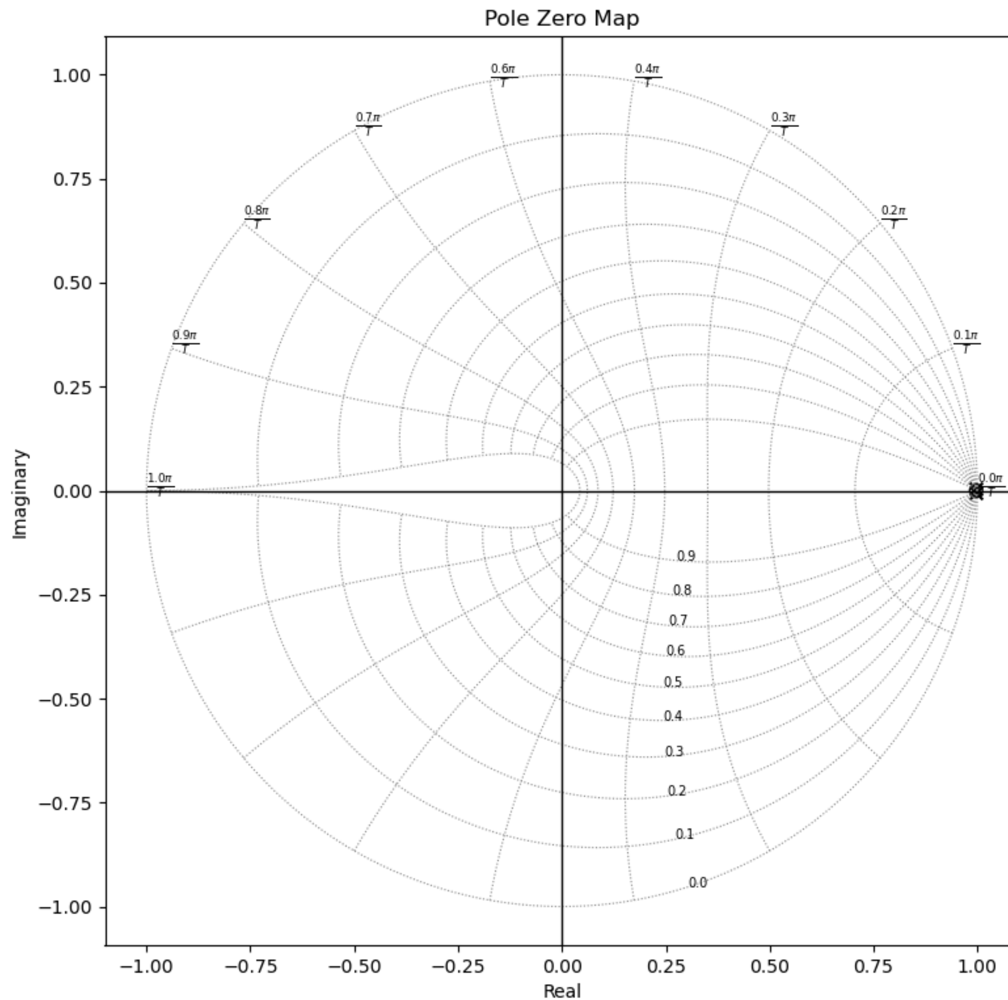
```
2 states have been removed from the model

 0.00595 z - 0.005931
----------------------
z^2 - 1.994 z + 0.9941

dt = 5.208333333333333e-06
```

Pole Zero Map

In [147…

```
plt.figure(figsize=(9,9))
__ = con.pzmap(d_gcl1, grid=True)
```

Figure



Closed Loop Time Domain Response (Step)

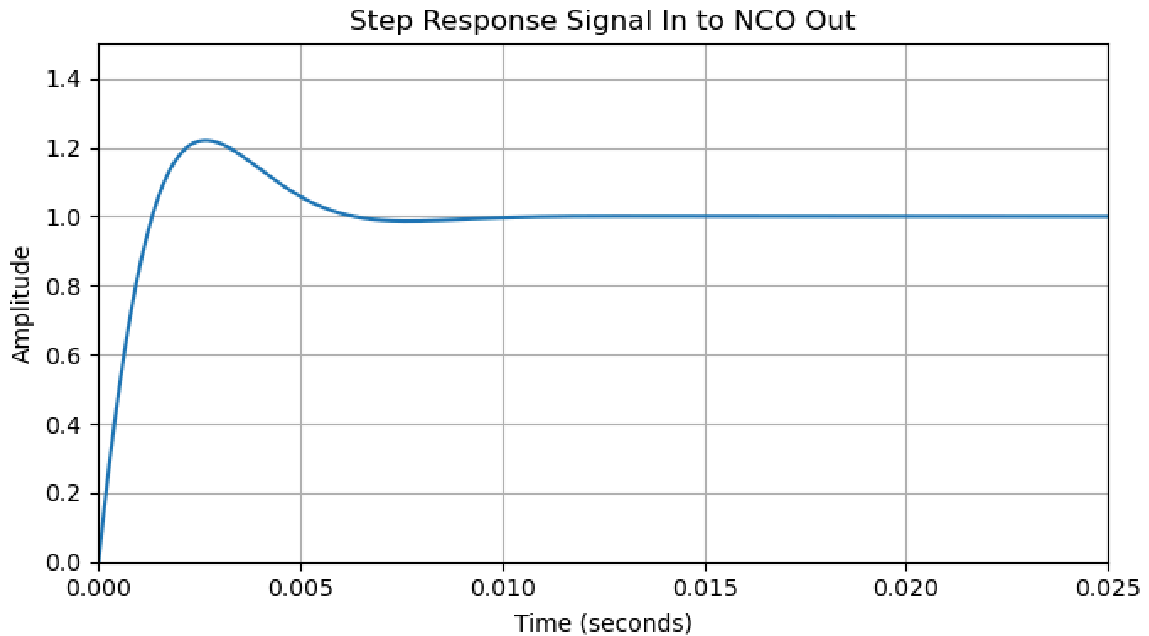
See notes above in the Closed Loop Time Domain Response for the analog loop about interpreting these plots. The plots show the response in the units of the output port to a normalized step in the units for that input port. (so in this case a response in phase to a step in phase).

In [148…

```
plt.figure(figsize=(7,4))
plt.plot(*con.step_response(d_gcl1))
plt.xlabel("Time (seconds)")
```

```
plt.ylabel("Amplitude")
plt.title("Step Response Signal In to NCO Out")
plt.grid()
plt.axis([0, .025, 0, 1.5])
plt.tight_layout()
```

Figure

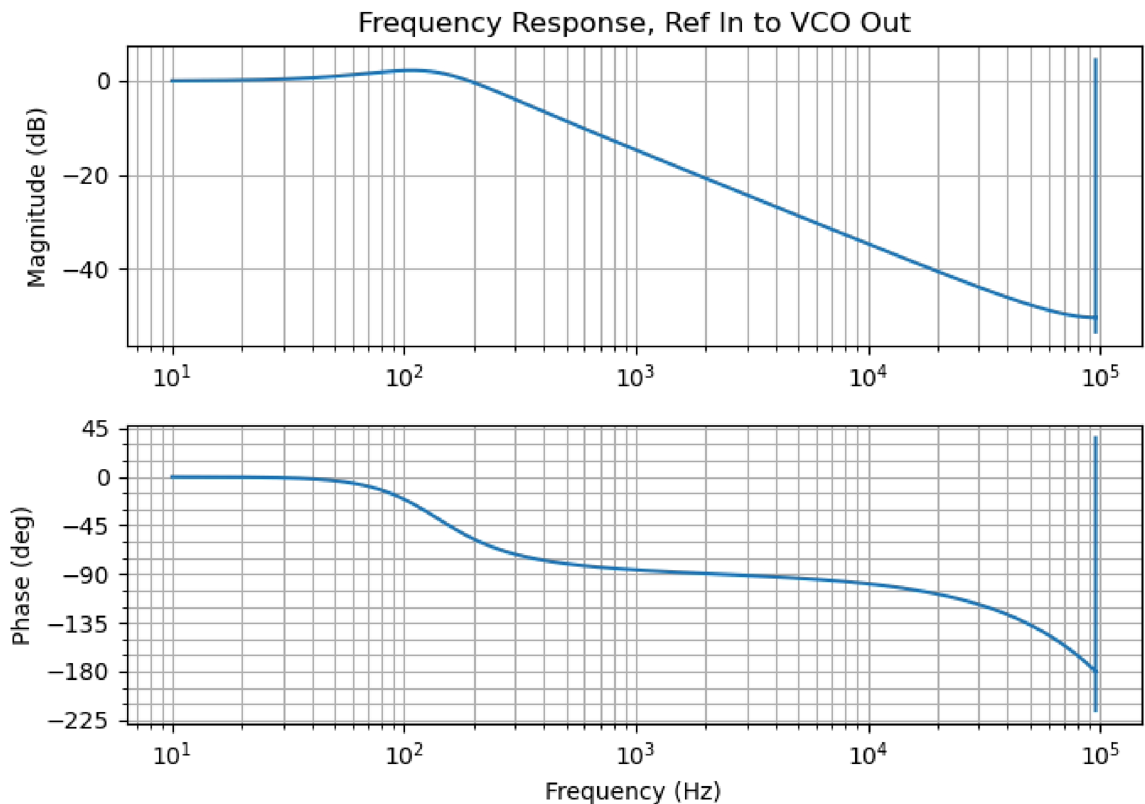


Closed Loop Frequency Domain Response

The vertical line in plot on right is to show Nyquist, and is not part of the response.

In [149…

```
plt.figure(figsize=(7,5))
__ = con.bode(d_gcl1, dB=True, Hz=True, omega_limits=[2*np.pi * 10, 2*np.pi*fs/2])
plt.subplot(2,1,1)
plt.title("Frequency Response, Ref In to VCO Out")
plt.tight_layout()
```



Mapping s to z

A deeper understanding of poles and zeros and there significance in placement on the s and z planes is very helpful in control system design.

The simple "matched-z" mapping was used to demonstrate mapping from the Laplace Transform to the z-Trasform using:

$$z = e^{sT}$$

The graphics below show how a grid in the s plane would transform to the z plane.

The matched z transform is instructional but has limited use as it doesn't in all cases maintain either the time or frequency domain response for the system. However when used to map systems with poles only, it will provide an impulse invariant result identical to the Method of Impulse Invariance (so retains the time domain response in the mapping for those cases).

In [151…

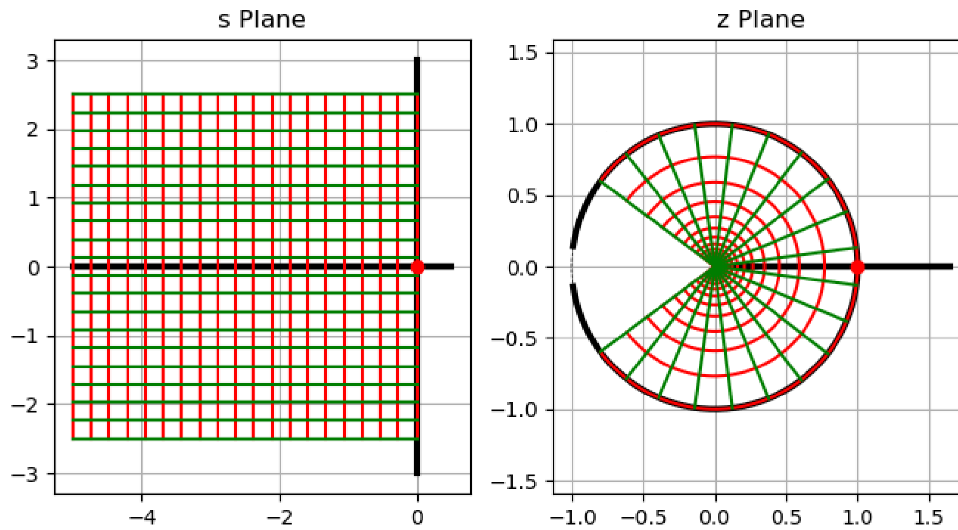
```
# graphically showing the mapping from s to z for z=e^s
# by drawing a

xbox = (-5, 0)
ybox = (-2.5, 2.5)
grids=20
```

```
vert = np.linspace(ybox[0], ybox[1],100)
omega = np.linspace(ybox[0], ybox[1], grids)

horiz = np.linspace(xbox[0], xbox[1],100)
sigma = np.linspace(xbox[0], xbox[1], grids)



yaxis = np.zeros(100)+1j*np.linspace(-3,3,100)
xaxis = np.linspace(-5,.5,100) + 1j*np.zeros(100)


plt.figure(figsize=(8,4))

# s plane
plt.subplot(1,2,1)


# plot vertical and horizontal axis
plt.plot(np.real(xaxis), np.imag(xaxis), 'k', linewidth=3)
plt.plot(np.real(yaxis), np.imag(yaxis), 'k', linewidth=3)

for s in sigma:
    plt.plot(np.real((s+1j*vert)), np.imag((s+1j*vert)), 'r')

for o in omega:
    plt.plot(np.real((horiz+1j*o)), np.imag((horiz+1j*o)), 'g')


# plot dot at orgin
plt.plot(0, 0, 'ro')

plt.axis('equal')
plt.title('s Plane')
plt.grid()
# zplane
plt.subplot(1,2,2)


#plot vertical and horizontal axis
plt.plot(np.real(np.exp(xaxis)), np.imag(np.exp(xaxis)), 'k', linewidth=3)
plt.plot(np.real(np.exp(yaxis)), np.imag(np.exp(yaxis)), 'k', linewidth=3)


# plot the unit circle
circle = np.linspace(0, 2*np.pi, 100)
plt.plot((np.cos(circle)), (np.sin(circle)), 'k--', linewidth=0.5)



for s in sigma:
    plt.plot(np.real(np.exp(s+1j*vert)), np.imag(np.exp(s+1j*vert)), 'r')

for o in omega:
    plt.plot(np.real(np.exp(horiz+1j*o)), np.imag(np.exp(horiz+1j*o)), 'g')
```

```
# plot dot at orgin
plt.plot(1, 0, 'ro')

plt.grid()
plt.axis('equal')
plt.title('z Plane')
```

Out[151… Text(0.5, 1.0, 'z Plane')

Figure



If the grid in the s plane extended to $-\pi$ and $+\pi$ vertically, then the circle would be completely filled. If the grid extends beyond $\pm\pi$ vertically, the mapping will repeat in side the unit circle (aliasing). Zoom in on the z-plane origin to see how the vertical grid lines to the far left map with increasingly (logarithmically) closer spacing.