Intro to Modeling CW Radar

Michael Lazar

2014-03-24

Presentation Outline

- Introduction to Radar
  - History
  - Applications
- Doppler Effect
- System Implementation
- Complex Mixer
  - Determine approaching vs. receding
- Tuning Fork
  - Used to calibrate radar units
- Frequency Modulated CW

Radar Background

- The transmission and reflection of radio waves was first observed by Heinrich Hertz in 1887.
- The first primary interest came from the military in the 1920s and 1930s.
- There has been a resurgence in the past decade.
  - Auto safety systems, tank level monitoring, motion sensors, speed guns
- There are two popular signaling techniques: pulse-based and continuous-wave.

Doppler Effect

- The Doppler Effect is the basis for CW signaling techniques.
- Assume a stationary observer transmits a signal, the signal hits a target, and the signal is reflected back.
- Why the factor of 2?
- E.g.
  - 24 GHz carrier frequency
  - 10 m/s target velocity (22 MPH)
  - 1.6 kHz frequency shift

$$v_{target}\begin{cases} approaching, & v > 0 \\ receeding, & v < 0 \end{cases}$$

$$Doppler\ Frequency:\ \Delta f = \frac{2 * v_{target}}{c} f_c$$

$$f_{recieve} = \left(1 + \frac{2 * v_{target}}{c}\right) f_c$$

$$\theta_{recieve} = 2\pi \frac{d}{c}(f_c + f_{recieve})$$

Python Introduction

- Python 2.7.5
- NumPy – vector operations
- SciPy – signal processing functions
- Use named tuples as containers for object parameters.

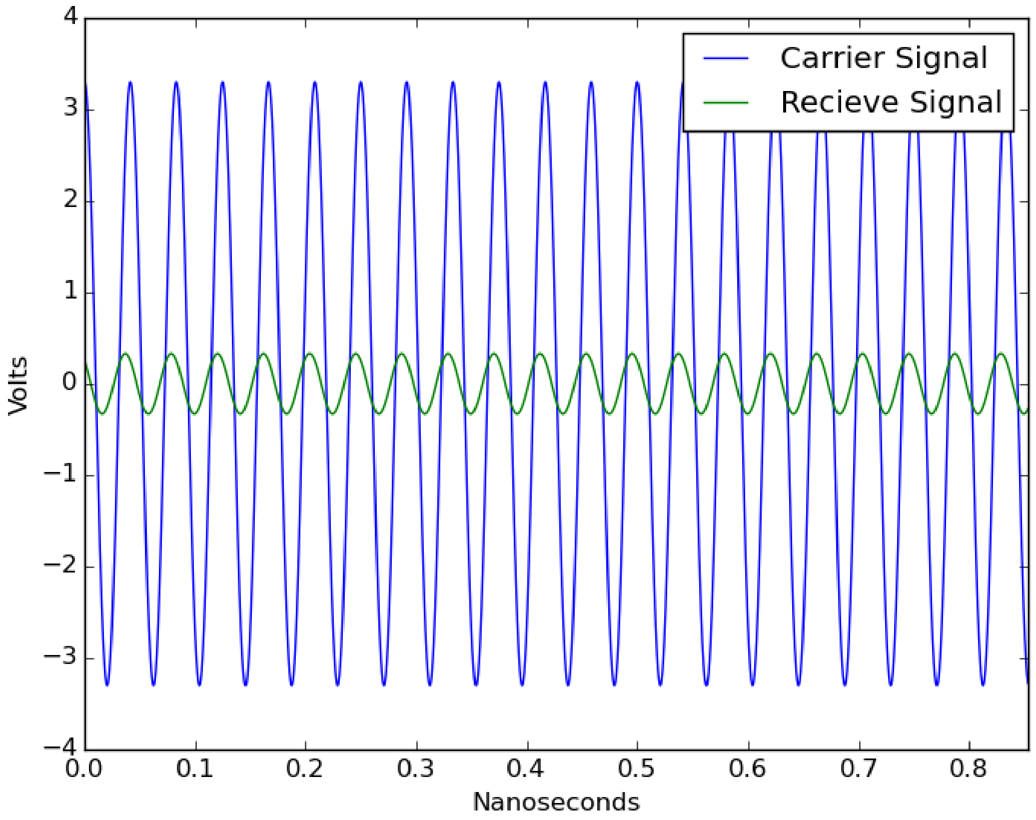
```
from collections import namedtuple

import numpy as np
import scipy as sp
import pylab as plt

from scipy import signal
from scipy.constants import c, pi

#from pylab import style
#style.use('ggplot')

Signal = namedtuple('Signal', ['amplitude', 'frequency', 'phase'])
Target = namedtuple('Target', ['distance', 'velocity', 'r_pct'])
FMTarget = namedtuple('FMTarget', ['frequency', 'mod_index', 'r_pct'])

def build_real_signal(t, params):
    "Construct a sinusoidal waveform containing a single frequency."

    return params.amplitude * np.cos(2*pi*t*params.frequency + params.phase)


def build_complex_signal(t, params):
    "Construct a complex sinusoidal waveform containing a single frequency."

    signal = params.amplitude*np.exp(1j*(2*pi*params.frequency*t+params.phase))
    return np.real(signal), np.imag(signal)


def fft(*args, **kwargs):
    "Alias for running an FFT and shifting it into the [-fs/2, fs/2] window."

    result = np.fft.fft(*args, **kwargs)
    result = np.fft.fftshift(result)
    return result
```

Doppler Effect

- Declare parameters for the carrier signal and the radar target.
- Select a sampling frequency greater than 2x the signal's frequency.
- Generate the reflected signal's parameters using the carrier signal and the formulas for Doppler Effect.

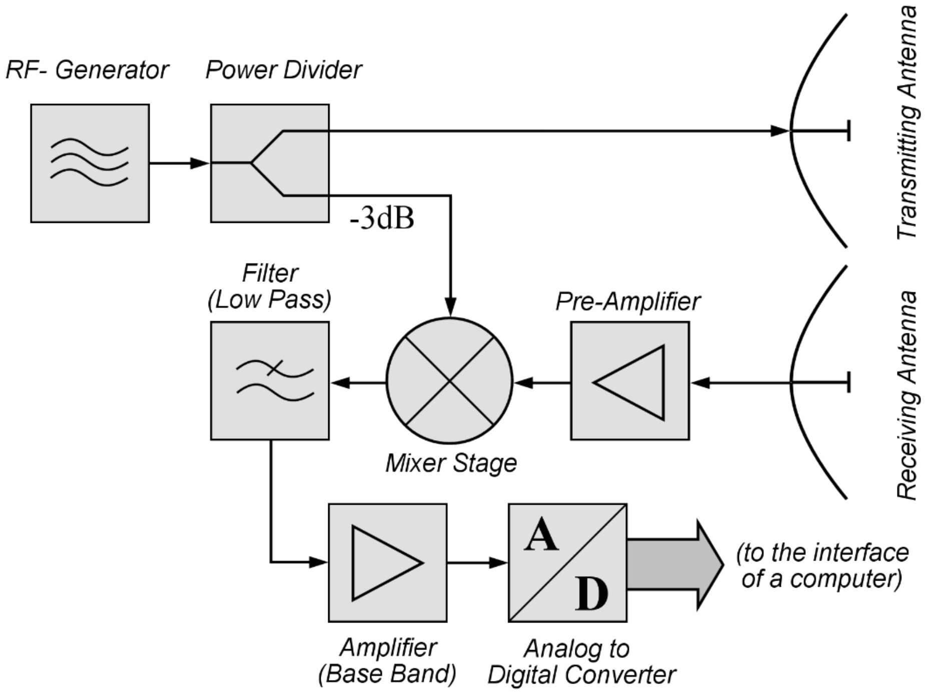
```
###############################################
# Variable Declarations
###############################################
fs = 24e11                    # Sample Rate [Hz]
n_samples = 2048              # Number of samples

carrier = Signal(
    amplitude = 3.3,          # [V]
    frequency = 24e9,         # [Hz]
    phase = 0,                # [rad]
    )

target = Target(
    distance = 1.0,           # [m]
    velocity = 10.0,          # [m/s]
    r_pct = 0.1,              # Ratio of signal reflected
    )
###############################################

# Generate Carrier Signal
t = np.linspace(0, n_samples/fs, n_samples)
tx = build_real_signal(t, carrier)

# Calculate Return Signal
doppler_shift = carrier.frequency * (2*target.velocity/c)
reflected = Signal(
    amplitude = carrier.amplitude * target.r_pct,
    frequency = carrier.frequency + doppler_shift,
    phase = 2*pi * (target.distance/c) * (2*carrier.frequency + doppler_shift),
    )

# Generate Return Signal
rx = build_real_signal(t, reflected)
```

Doppler Effect



CW Hardware Abstraction

- Follow the previous example and set the RF generator to 24 GHz.
- The Mixer creates sum and difference terms, with the difference term equal to the Doppler Shift.
- The low pass filter removes the sum term and shifts the signal into base band.
- All post-processing is handled in the digital domain.



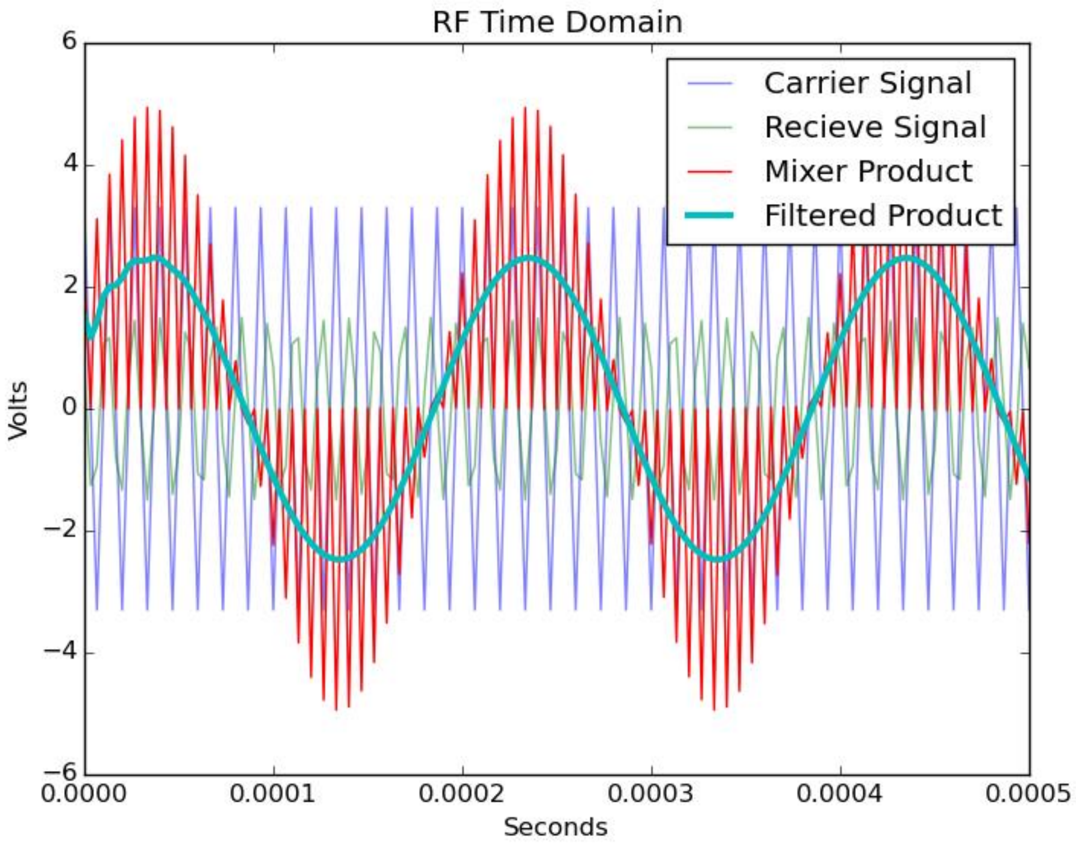
CW Simulation

- In order to simulate the analog stage without loss of information, we need to set our initial sample frequency in the RF range.
- Memory becomes an issue

  7ms * 100 GHz * 8 bytes = 5.6 GB
- Pick a reflected frequency for demonstration purposes.
- Actual LPF would be implemented in hardware (R's and C's)

```
#############################################
# Variable Declarations
#############################################

# Sample frequency should be at least 2x the sum term of the carrier and
# reflected frequencies.
fs = 300000.0                 # Sample Rate [Hz]

# The ADC sample rate only needs to be greater than 2x the difference term.
adc_fs = 25000.0              # Sample Rate [Hz]
adc_samples = 2048            # Number of samples

carrier = Signal(
    amplitude = 3.3,          # [V]
    frequency = 75000,        # [Hz]
    phase = 0,                # [rad]
    )

# For demonstration purposes, assume that the following signal is the result
# of the doppler effect.
reflected = Signal(
    amplitude = 1.5,          # [V]
    frequency = 70000,        # [Hz]
    phase = 1.1,              # [rad]
    )
#############################################

# Generate Signals
t = np.linspace(0, adc_samples/adc_fs, fs * (adc_samples/adc_fs))
tx = build_real_signal(t, carrier)
rx = build_real_signal(t, reflected)

# Mixer Stage
mixer_product = tx * rx

# Low-pass Filter
# Note: This is a simplified filter and not what is actually done in hardware.
h = signal.firwin(50, cutoff=0.5)
filtered_product = signal.filtfilt(h, [1.0], mixer_product)
```

CW Simulation

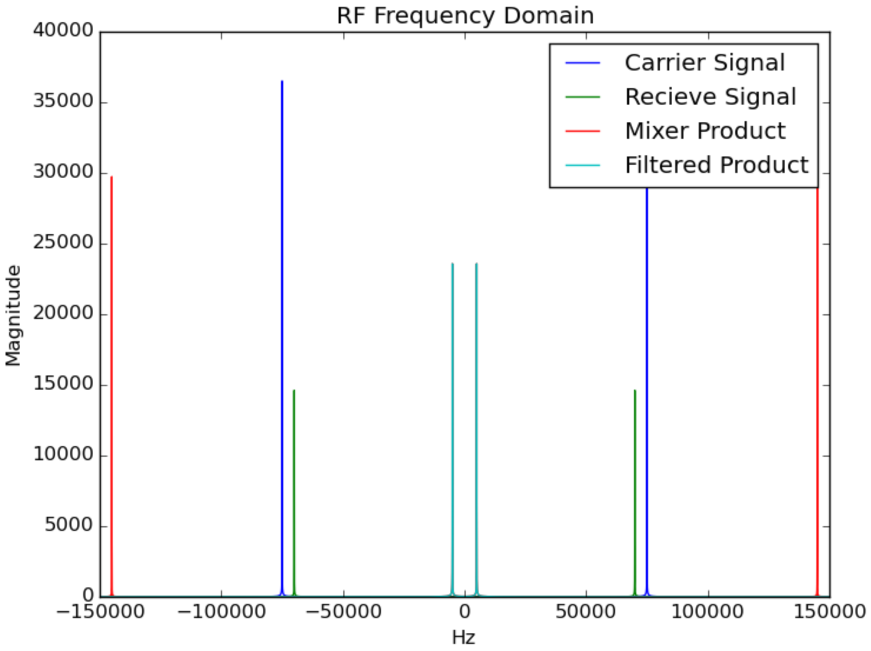


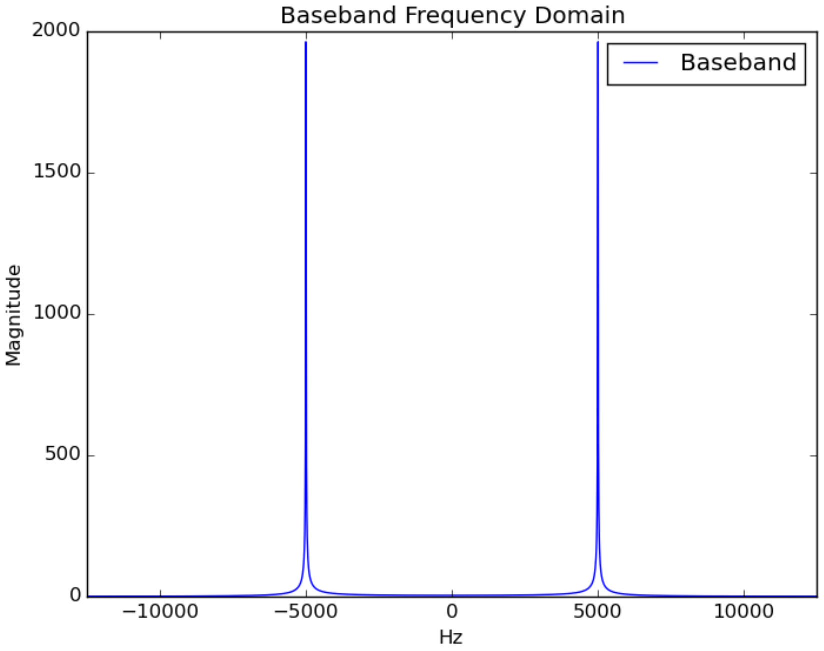
CW Simulation

- Run DFTs for visualization purposes.
- Simulate the ADC by decimating the signal.
    - adc_fs must be a factor of fs.
- Run a DFT on the baseband signal and look for targets.

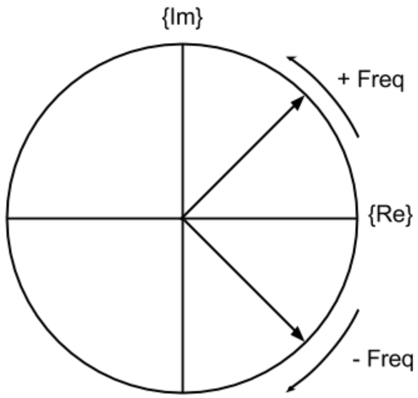
```
# Generate Signals
t = np.linspace(0, adc_samples/adc_fs, fs * (adc_samples/adc_fs))
tx = build_real_signal(t, carrier)
rx = build_real_signal(t, reflected)

# Mixer Stage
mixer_product = tx * rx

# Low-pass Filter
# Note: This is a simplified filter and not what is actually done in hardware.
h = signal.firwin(50, cutoff=0.5)
filtered_product = signal.filtfilt(h, [1.0], mixer_product)

# Fourier Transform (for plotting)
f = np.linspace(-fs/2, fs/2, fs * (adc_samples/adc_fs))
tx_fft = fft(tx)
rx_fft = fft(rx)
mixer_product_fft = fft(mixer_product)
filtered_product_fft = fft(filtered_product)

# ADC Stage (Downsample)
adc_t = t[::fs/adc_fs]
baseband = filtered_product[::fs/adc_fs]

# Fourier Transform
adc_f = np.linspace(-adc_fs/2, adc_fs/2, adc_samples)
baseband_fft = fft(baseband)
```

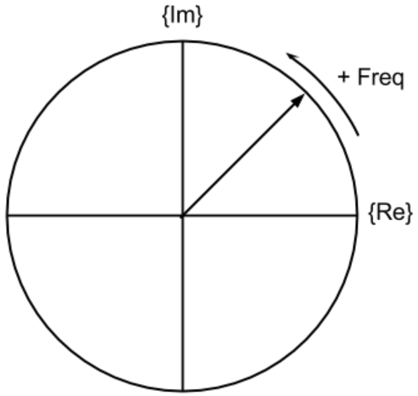
CW Simulation





Complex Mixer

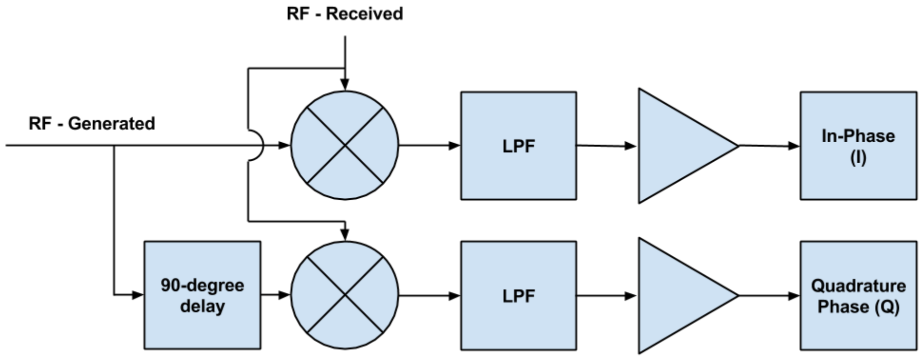




- The target's direction determines the sign of the difference term.
- If we can find the imaginary component, we can determine the direction of the phasor.
- The DFT, by design, allows for complex signals.

$$X_k \stackrel{\text{def}}{=} \sum_{n=0}^{N-1} x_n \cdot e^{-i2\pi kn/N}, \quad k \in \mathbb{Z}$$

Complex Mixer



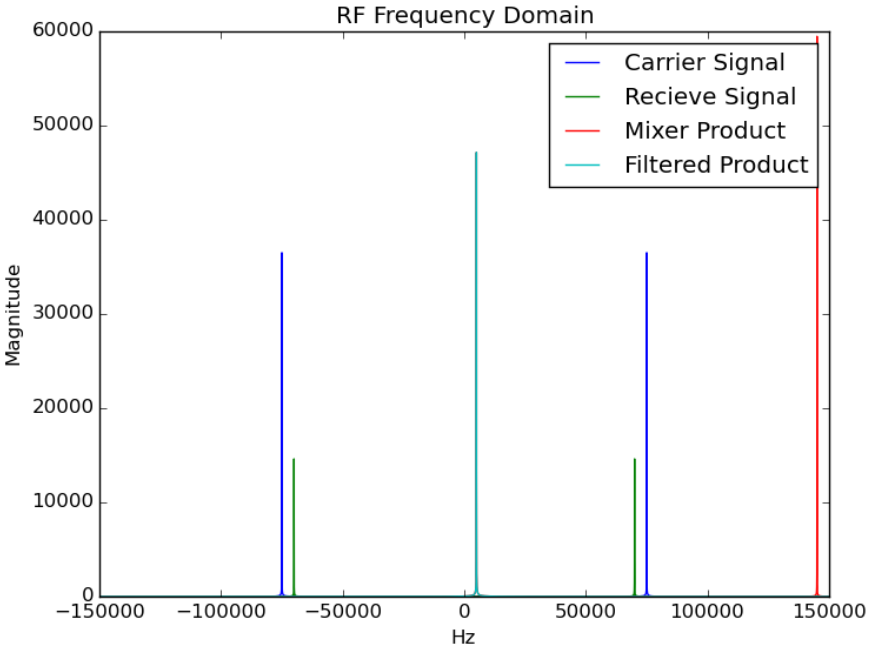
- The generated signal is delayed by 90 degrees in hardware using a Polyphase Filter.
- The mixing product of the delayed signal and the received signal produces the imaginary component of the Base Band.
- Often referred to as In-Phase (real) and Quadrature-Phase (imaginary), or I/Q for short.

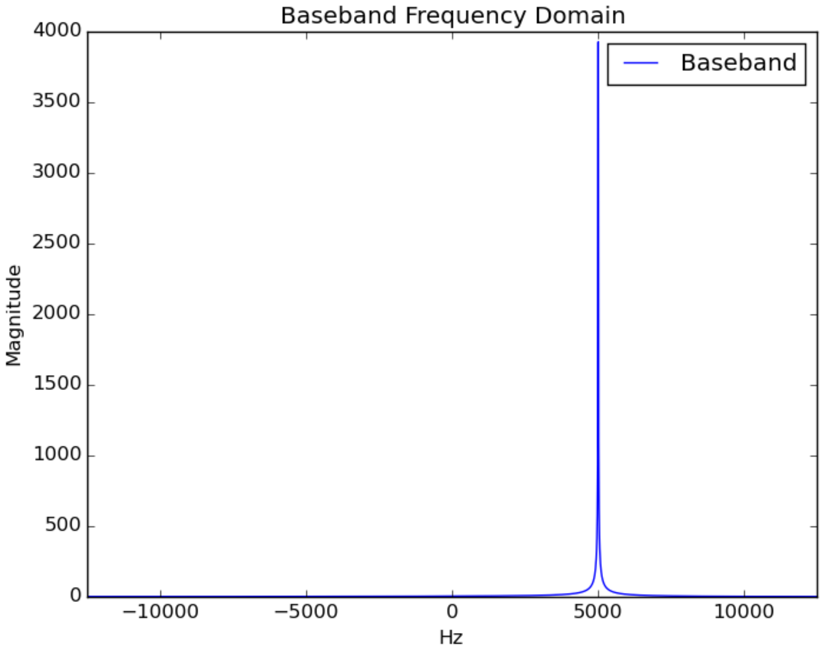
Complex Mixer Simulation

- The complex carrier waveform is split into two “real” signals.
- I and Q follow separate paths until they are joined at the baseband FFT.

```
# Generate Signals
t = np.linspace(0, adc_samples/adc_fs, fs * (adc_samples/adc_fs))
tx, tx_90 = build_complex_signal(t, carrier)
rx = build_real_signal(t, reflected)

# Mixer Stage
mixer_i = tx * rx
mixer_q = tx_90 * rx

# Low-pass Filter
# Note: This is a simplified example and not what is actually done in hardware.
h = signal.firwin(50, cutoff=0.5)
filtered_i = signal.filtfilt(h, [1.0], mixer_i)
filtered_q = signal.filtfilt(h, [1.0], mixer_q)

# Fourier Transform (For plotting)
f = np.linspace(-fs/2, fs/2, fs * (adc_samples/adc_fs))
tx_fft = fft(tx)
rx_fft = fft(rx)
mixer_product_fft = fft(mixer_i + 1j*mixer_q)
filtered_product_fft = fft(filtered_i + 1j*filtered_q)

# ADC Stage (Downsample)
adc_t = t[::fs/adc_fs]
baseband_i = filtered_i[::fs/adc_fs]
baseband_q = filtered_q[::fs/adc_fs]

# Fourier Transform
adc_f = np.linspace(-adc_fs/2, adc_fs/2, adc_samples)
baseband_fft = fft(baseband_i + 1j*baseband_q)
```

Complex Mixer Simulation





Tuning Forks

- Police officers are required to use tuning forks to verify the accuracy of their CW speed radars.
- For example, a particular tuning fork may be rated at 35 MPH for K band (24.150 GHz) radars. Striking the fork in front of a radar will always produce a target at 35 MPH.
- This phenomena is independent of the vibrating fork's speed and amplitude, only frequency matters.

Frequency modulation

- Using a tuning fork will induce an FM component into the signal.
- This component spreads the signal's energy into sidebands at
  $$f_c \pm f_m, 2f_m, 3f_m, \ldots$$
- If $\beta < 1$, the signal is "narrow band" and the majority of energy is located in the 1st sideband.
- This simulates a beat frequency at the same frequency as the tuning fork.

$$t_x = A_c \cos(2\pi f_c t)$$

$$r_x = A_c \cos(2\pi f_c t + \beta \sin(2\pi f_m t))$$

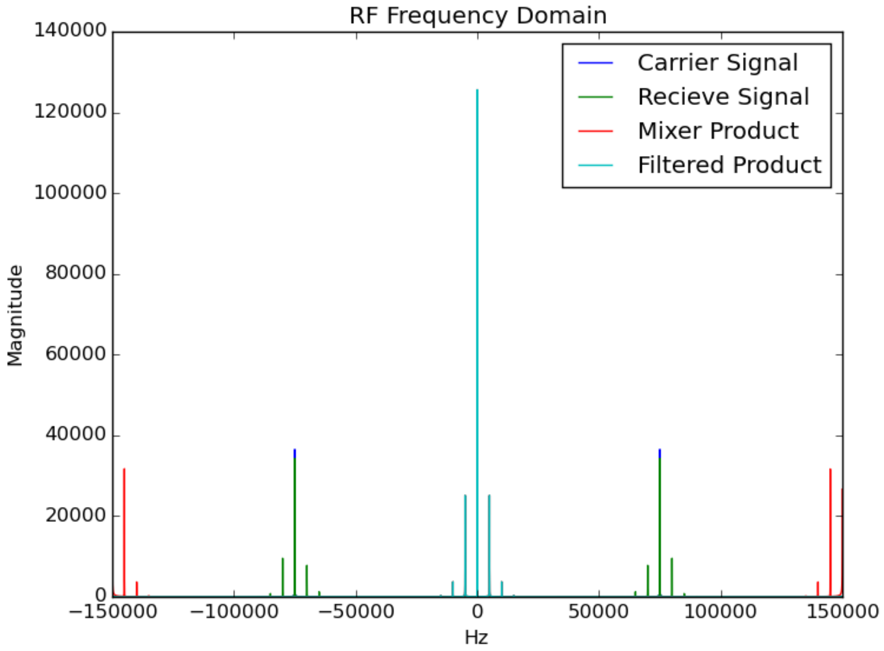
$$Modulation\ Index: \beta = \frac{\Delta f}{f_m}$$

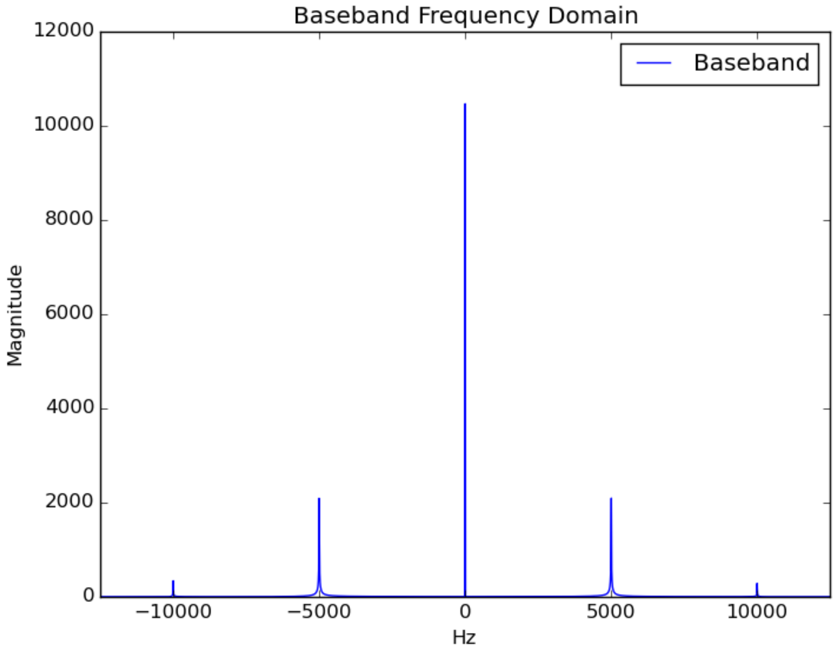
Frequency Modulation Simulation

- Set the modulation index to 0.5
- First, the FM component of the signal is generated.
- Then, the signal's phase is set as the FM component. Why is this ok?
    - Refer to the formula on the previous slide.
    - Send an array instead of a scalar.
- After rx is generated, follow the same steps as the Complex Mixer example.

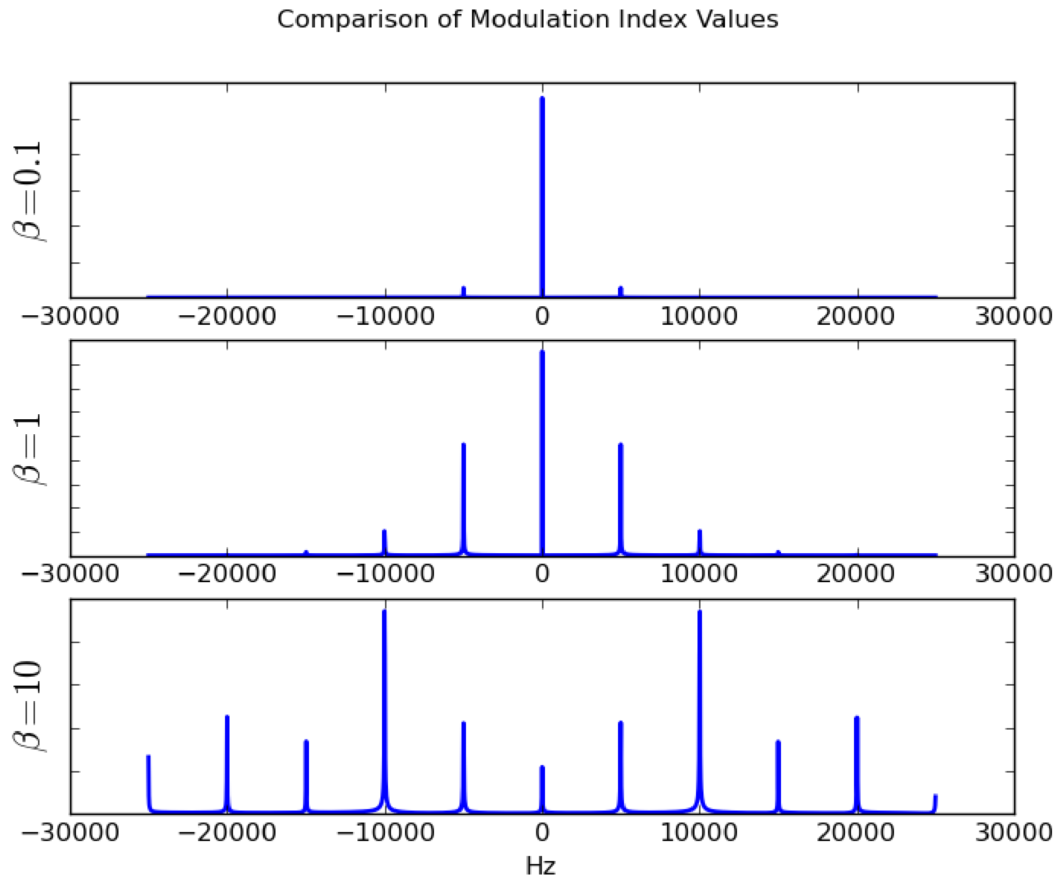
```
#############################################
# Variable Declarations
#############################################

# Sample frequency should be at least 2x the sum term of the carrier and
# reflected frequencies.
fs = 300000.0               # Sample Rate [Hz]

# The ADC sample rate only needs to be greater than 2x the difference term.
adc_fs = 25000.0            # Sample Rate [Hz]
adc_samples = 2048          # Number of samples

carrier = Signal(
    amplitude = 3.3,        # [V]
    frequency = 75000,      # [Hz]
    phase = 0,              # [rad]
    )

tuning_fork = FMTarget(
    frequency = 5000,       # [Hz]
    mod_index = 0.5,        # Modulation Index, should be < 1
    r_pct = 1.0,
    )
#############################################

# Generate Carrier Signal
t = np.linspace(0, adc_samples/adc_fs, fs * (adc_samples/adc_fs))
tx, tx_90 = build_complex_signal(t, carrier)

# Generate FM component
fm = Signal(
    amplitude = tuning_fork.mod_index,
    frequency = tuning_fork.frequency,
    phase = pi/2,
    )
fm_x = build_real_signal(t, fm)

# Generate Return Signal
reflected = Signal(
    amplitude = carrier.amplitude * tuning_fork.r_pct,
    frequency = carrier.frequency,
    phase = fm_x,
    )
rx = build_real_signal(t, reflected)
```

Frequency Modulation Simulation



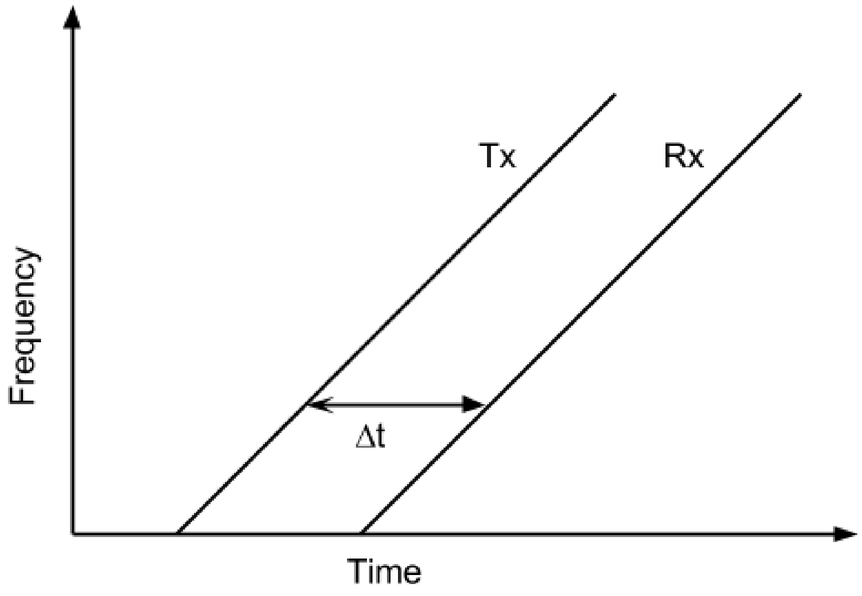


Frequency Modulation Simulation



FMCW

- Instead of sending out a constant frequency, transmit a linear "chirp".
- The delay between the transmit and receive chirps will create a constant frequency delta.
- Distance resolution is dependent on the chirp's bandwidth.
- Triangle waveforms are used to track a target's speed and distance.



Reference

Code Repository:

https://github.com/michael-lazar/radar_presentation

Images:

- Krewaldt, R. http://en.wikipedia.org/wiki/File:Heinrich_Rudolf_Hertz.jpg
- Whisky, C. (2012) http://en.wikipedia.org/wiki/File:Bsp2_CW-Radar.EN.png
- Helihark. http://commons.wikimedia.org/wiki/File:Tuning-fork.jpg