KTH ROYAL INSTITUTE OF TECHNOLOGY

Build Your Own Radar System Report

Group2

Giandomenico Panettieri
Christian Giuseppe Terranova
Marko Savić
Job Mathew George

October 27, 2023

CONTENTS

1. MATLAB CODES FOR RANGE AND VELOCITY MEASUREMENTS

For details on the code implementation, please refer to subsections A.1.1 and A.1.2. A brief overview of the continuous wave (CW) and frequency modulated continuous wave (FMCW) radar working principles is provided in subsections 1.1 and 1.2. This background is necessary to establish the objectives that guided the development of the signal processing codes presented in later sections.

1.1. CW RADAR FOR VELOCITY MEASUREMENT WORKING PRINCIPLE

The Continuous Wave Radar (CW Radar) operates by detecting the relative velocity of objects moving towards the radar through the observation of the Doppler shift phenomenon. In this radar system, a continuous electromagnetic wave is transmitted at a fixed and known carrier frequency. At the radar receiver, the frequency of the back-scattered wave is detected. When an object approaches the radar with a relative velocity denoted as Δv, a frequency shift, Δf, is observed in the received wave compared to the transmitted wave[1].
It is possible to measure Δf using an **RF mixing stage** that mixes the received signal with the signal of a local oscillator (L.O.) tuned to the carrier frequency f_c. After the mixer, the received signal is downconverted to **baseband**, enabling further processing using inexpensive and discrete filters and amplifiers.
For these reasons the core of the code is a fast Fourier transform (FFT).

1.2. FMCW RADAR WORKING PRINCIPLE

The Frequency Modulated Continuous Wave Radar (FMCW Radar) operates by detecting the range of objects through the observation of the frequency delay between transmitted and received signal. In this radar system, a continuous wave signal with a frequency that varies over time is transmitted. The reflected signal is mixed with the transmitted signal, this mixing process generates a beat frequency, which is the difference in frequency between the transmitted and received signals. The beat frequency is proportional to the round-trip time to the target and back and is proportional to the range of the target[2].
The beat frequency can be measured by using an **RF mixing stage** that mixes the received signal with the transmitted signal. After the mixer and discrete filters and amplifiers, the IFFT algorithm is used to convert the beat frequency information into the range domain.

1.3. CW RADAR VELOCITY MEASUREMENT CODE TEST

In order for the code to be considered valid, it must produce coherent spectrograms when it operates to the provided audio file "VelocityTestFile.m4a". The code A.1.1 produces correct results.

[1] This frequency shift can be described by the equation $\Delta f = f_c \cdot \Delta v / c$ where Δf represents the observed frequency shift, f_c is the carrier frequency, and c is the speed of light.
[2] It is possible to find the range using the equation $R = c\Delta t/2$

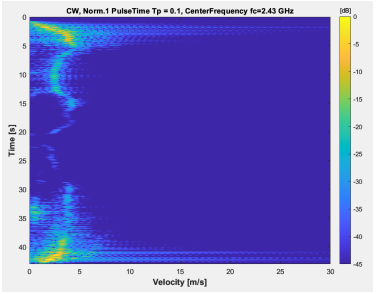


Figure 1.1:
Normalization 1: Divides the entire data within the spectrogram by the maximum value.

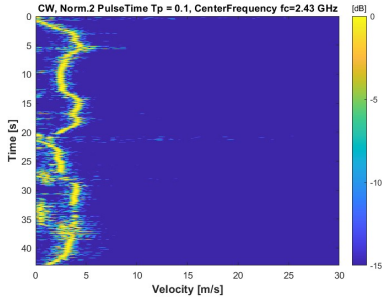


Figure 1.2:
Normalization 2: Divides each row of the spectrogram by its respective maximum value.

It is of strong interest to study how the change of the main parameters affect the measurement. The main parameters for this code are:

1. **T_p pulse width**: Observation time of the FFT algorithm. The audio file is quantized in windows of duration T_p, the longer is T_p the narrower is the window bandwidth. This allows us to have better frequency resolution and precision as the velocity is a shift in frequency of the carrier frequency.

2. **f_c carrier frequency:** it determines the maximum range of the radar and it is fundamental in the conversion of Doppler frequency to the relative velocity between the target and the radar[3].

3. **Sampling Frequency:** This parameter defines the bandwidth of the Fast Fourier Transform (FFT).

4. **Total Observation Time:** This parameter is pivotal as it determines the achievable frequency resolution[4].

The effect on the frequency domain when a large signal is observed within a window of duration T_p is to convolve the desired signal with a sinc function that is approximately of width T_p as shown in figure 1.3 and in figure 1.4. This is the usual trade off between time domain resolution and frequency domain resolution.

As the carrier frequency is changed in the code (but not in the actual transmitting wave) the effect is to change the scale factor between the relative velocity of the target and the frequency axis (Figure 1.5).

The zero padding is a very important trick as it makes the plot more readable . However, it is not able to actually increase the frequency resolution of our signal as that depends mostly on the window size T_p. Figure 1.6 and figure 1.7 show the visual improvement.

[3]The velocity measured with the Doppler is $\Delta v = \lambda \Delta f_D/2$ where Δf_D is the Doppler shift, $\lambda = c/f_c$ and f_c is the carrier frequency

[4]According to the FFT working principle, more samples in the time domain result in more samples in the frequency domain for a fixed bandwidth, thus enanching frequency resolution.

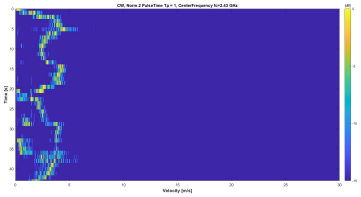


Figure 1.3:
Long window duration T_p resulting in a worse time resolution, but less frequency uncertainty.

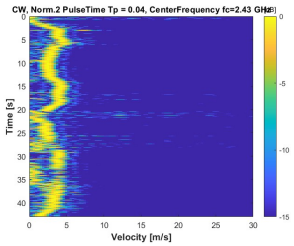


Figure 1.4:
Short window duration T_p resulting in high frequency uncertainty, but very good time resolution.

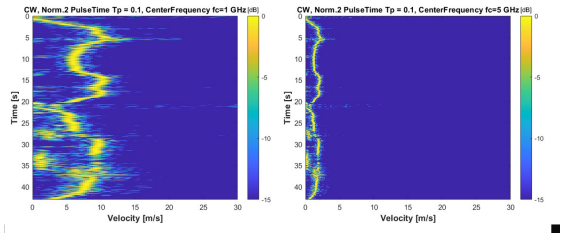


Figure 1.5:
On the right a large value carrier frequency, while on the left a low value carrier frequency. The carrier frequency has been modified only in the signal processing part.

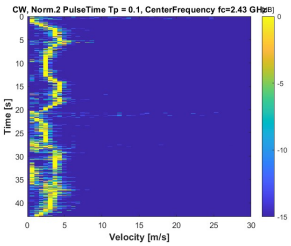


Figure 1.6:
No zero padding.

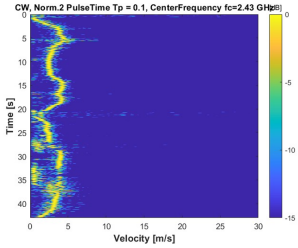


Figure 1.7:
High order zero padding.

1.4. FMCW Radar Range Measurement Code Test

In order for the code to be considered valid, it must produce coherent spectrograms when it operates to the provided audio file "RangeTestFile.m4a" (for details on the code, refer to A.1.2). In this context, to extract data regarding the range, only the up-chirp signal needs to be processed. For this reason, a sync signal that points at when the up-chirp signal is sent by the transmitter is required to process data.

A wider bandwidth of the FMCW results in a better range resolution[5]. This improved range resolution enables the radar to distinguish smaller differences in ranges. As the bandwidth is changed in the code (but not in the actual transmitting wave) the effect is to change the scale factor of the range axis (compare figure 1.8 with figure 1.9).

The **MS clutter rejection algorithm** effectively eliminates background clutter, leading to a significant improvement, as demonstrated in Figure 1.11. In contrast, Figure 1.10 shows the result when no MS clutter rejection is applied.

The impact of **zero padding** is evident when comparing Figure 1.12 and Figure 1.13. The effects of zero padding have already been discussed in subsection 1.3.

The **MTI clutter rejection algorithm**, also known as the Moving Target Indicator, is utilized to filter out signals from stationary or slow-moving objects, enhancing the capability to differentiate moving targets. This is accomplished by subtracting the current data from the previous one, as demonstrated in Figure 1.14, or by incorporating more previous pulses for higher-order corrections if needed. In this context, no significant improvement were obtained with higher-order corrections.

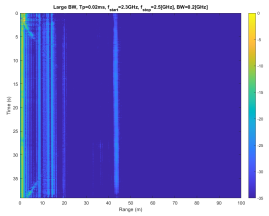


Figure 1.8:
Large BW

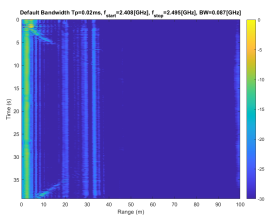


Figure 1.9:
Default BW.

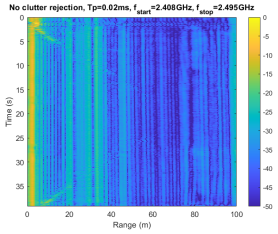


Figure 1.10:
Without MS clutter rejection

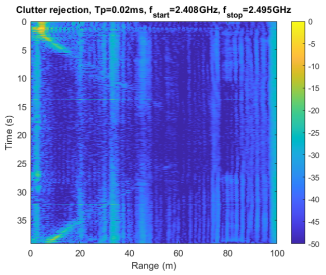


Figure 1.11:
With MS clutter rejection

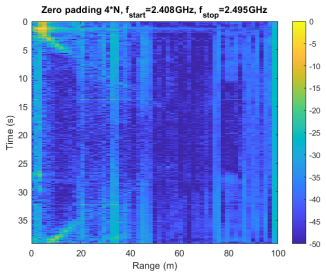


Figure 1.12:
Without Zero Padding

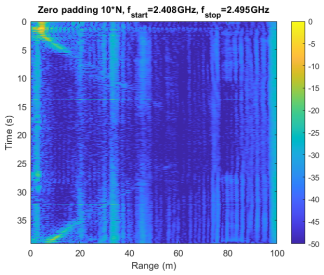


Figure 1.13:
With Zero Padding

[5]The range resolution is $dR = c/2\Delta f$, where c is the speed of light and Δf is the bandwidth of the FMCW.

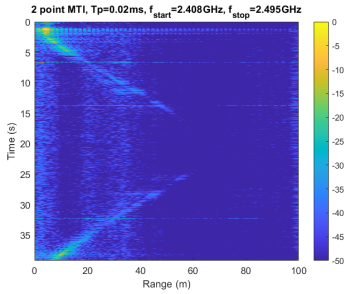


Figure 1.14:
Resulting range spectrogram after 2-pulses MTI.

2. Circuit implementation for short Range Radar at 2.4 GHz

For the circuit implementation, the group has incorporated the voltage regulator, the modulator, and the video amplifier. Our primary focus was on constructing the hardware and characterizing the key parameters of the entire circuit. Specifically, measurements for the gain of the amplification stage, the -3dB point of the filter, the triangular shape of the voltage driving the VCO, and the synchronization between the sync signal and the up-chirp signal were conducted by the group.

2.1. DC voltage regulator

The system was supplied partly with 12V and 5V. The 5V DC signal was provided by means of the IC LM2940CT, which is a low-dropout voltage regulator stable under a variable current. 12V was provided using a USB device connected to the USB port of a laptop. To indicate that the voltage regulator was being supplied with 12V, a simple LED was connected to the 12V input terminal with a current limiting resistor. Both the output and the input had a filtering capacitor to supply a more stable voltage. The actual circuit implementation is reported in figure 2.1.

2.2. Baseband video amplifier circuit

The overall baseband video amplifier schematic is reported in the figure 2.1.
The circuit comprises an initial **gain amplifier** strategically placed to minimize the influence of noise in subsequent stages. The input is AC-coupled to the circuit via the input capacitance. Following the gain amplifier, there is a **two-stage Sallen-Key low pass filter**.
The input signal of the video amplifier is also the down-converted version of the received signal. To mitigate the quantization noise of the subsequent ADCs, it becomes essential to adjust the gain while attempting to empirically determine an optimal gain setting.

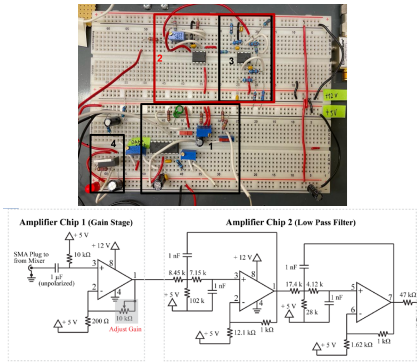


Figure 2.1: Top figure 1.Modulator, 2.Video amplifier, 3.Filter, 4.Voltage regulator. Bottom figure is the video amplifier schematic.

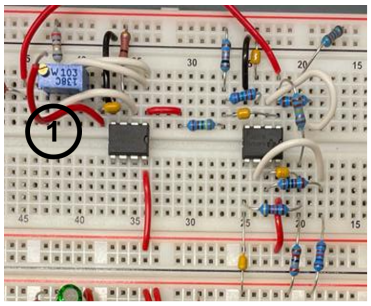


Figure 2.2: Tuning resistor for the gain.

2.2.1. Gain setup and measurement

To determine the appropriate gain setting, a $2V_{pp}$ signal at $1.74kHz$ is applied to the input of the video amplifier. The choice of a $1.74kHz$ frequency is based on the empirical observation that it corresponds to the peak of the frequency response, although it may vary between projects. The tuning resistor for the gain amplifier must be adjusted to identify the point at which clipping occurs.
As clipping originates from the gain amplifier[6], the measurement is conducted at its output [7,8].

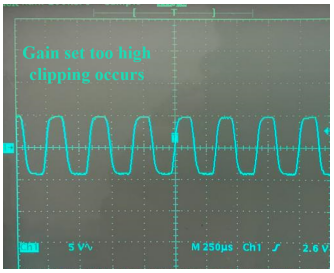


Figure 2.3:
Gain exceed a factor 4 causing clipping.

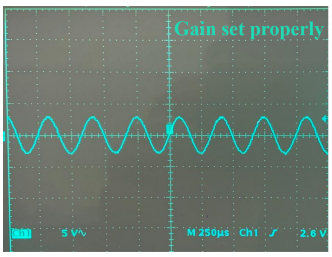
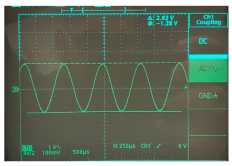


Figure 2.4:
Gain properly adjusted.

[6] The two Sallen-Key cells have both a unitary gain.
[7] Formally, one could use any frequency up to approximately $\sim GBWP/gain$ for this measurement, where $GBWP$ is the gain-bandwidth product of the employed operational amplifier. However, there is no need for us to investigate such high frequencies.
[8] It is better to measure clipping directly from the gain amplifier output since the filter stage can modify the harmonic content of the distorted signal.

2.2.2. VIDEO AMPLIFIER -3DB POINT MEASUREMENT

The -3dB point typically defines the bandwidth of a (Butterworth) filter, serving as the upper limit for the signal bandwidth. Using the same sinusoidal input as described in 2.2.1 ($2V_{pp}$ at $1.74kHz$) it is possible to adjust the gain of the gain amplifier to obtain a sinusoidal voltage of $2.62V_{pp}$ at $1.74kHz$ at the output.
The input frequency is then increased while maintaining a constant voltage amplitude. The -3dB point corresponds to the input frequency at which the output voltage amplitude, measured at the filter output, is $1.57V_{pp}$, i.e., $2.62V_{pp}$ divided by $\sqrt{2}$[9].
When the measurements are performed with the oscilloscope, after the $47k\Omega$ resistor (see the schematic in figure 2.1), the -3dB point is located near $11.4kHz$ as reported in figure 2.2.2.
The same measurement can be performed by means of a multimeter, as illustrated in figure 2.6. What can be observed is that the -3dB point is now located at $14.2kHz$.
Justification for this observation is needed. The most plausible explanation is that the $47k\Omega$ resistor and the oscilloscope's input capacitor introduce an additional pole alongside those already created by the Sallen-Kay cells. If true, probing the filter OpAmp output directly with the oscilloscope would short the reading capacitor to ground, resulting in no additional pole being introduced. In figure 2.7 the measurement is reported and it sustain the argument since now the $-3dB$ point occurs near $14.6kHz$ that is very close to the frequency measured with the multimeter.



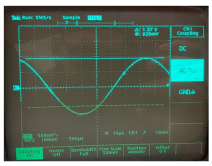


Figure 2.5:
-3dB point measurement with oscilloscope after the video amplifier $47k\Omega$ output resistor

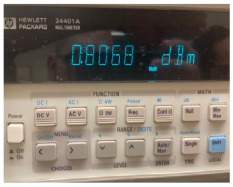
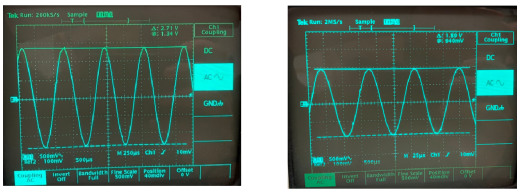


Figure 2.6:
-3dB point measurement with a multimeter after the video amplifier $47k\Omega$ output resistor

2.3. MODULATOR CIRCUIT

The XR-2206 modulator circuit generates the desired triangular voltage signal that drives the VCO. It also produces the sync square wave signal synchronized with the chirps. A toggle switch allows switching between CW and FMCW radar modes. The up-chirp time for FMCW is set to 20ms, resulting in a 40ms triangle wave period. The ramp amplitude is set to 1.2 Vpp, achieving a frequency swing from 2.4GHz to 2.5GHz on the VCO[10].

[9] The gain set in this experiment is a random gain. What is important is to know what is the amplitude value at least one decade before the -3dB point frequency.
[10] The voltage values have been set according to the data-sheet of the modulator



Oscilloscope measurement
Input 1.74kHz 2V
Output 2.71V

Oscilloscope measurement(-3dB)
Input 14.7kHz 2V
Output (before47kΩ) 1.89V

Figure 2.7:
-3dB point measurement with oscilloscope at the last OpAmp output of the video amplifier.

2.3.1. Upchirp signal and Sync signal measurement

The FMCW radar depends on precise synchronization between the upchirp/downchirp signal and the sync signal. Failure to meet this requirement can severely impact signal processing, leading to the elaboration of incorrect data. The modulator XR-2206 is employed to generate the upchirp and the downchirp signal that drives the VCO (Voltage Controlled Oscillator). By means of a tuning resistor (figure 2.9 resistor labeled with number 2), the modulator circuit is able to vary the up-chirp duration. A proper calibration of the apparatus leads to the result obtained in figure 2.8.

Figure 2.8:
Upchirp signal and Sync signal. The figure shows that the two signal are synchronous as the square-wave points both the sent upchirps and downchirps.

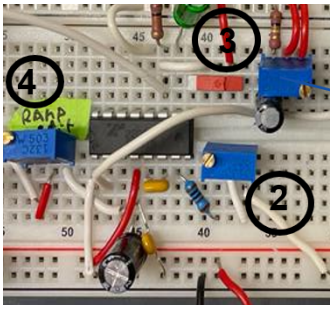


Figure 2.9:
Tuning resistors of the modulator circuit. (2) sets Tp duration – chirp rate. (3) sets DC offset of the tuning voltage. (4) sets ramp magnitude.

3. Measurements With The Built Radar

With the radar built in section 2 it is now possible to perform velocity (3.1) and range (3.2) measurements of a target. Moreover, one could also measure both range and velocity in (pseudo)real time with the provided code in A.3.1.

3.1. Velocity Measurements in CW mode

Refer to appendix 1.3 for more details in the code implementation.
Figure 3.1 plots the velocity measured for a single target, while figure 3.2 shows an example of multiple targets velocity acquisition. An example of captured data is plotted in figure 3.3. To obtain a pseudo real time velocity measurement, one can apply the FFT algorithm each $T_p * SamplingFrequency$ samples received. In figure 3.4 it is shown an example of this procedure for a given time window [11].

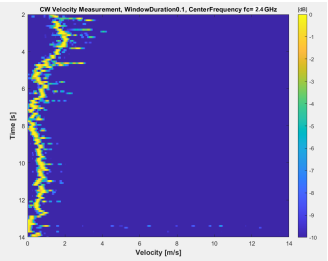


Figure 3.1:
Velocity measurement of a single target.

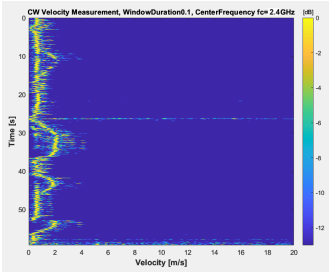


Figure 3.2:
Velocity measurement of two targets.

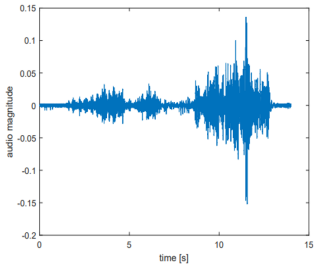


Figure 3.3:
Example of captured data.

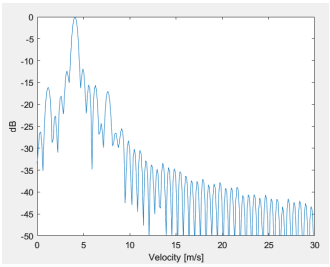


Figure 3.4:
Real time velocity measurement plot example at a specific time.

3.2. Range Measurements in FMCW mode

To measure range, the radar is set up in FMCW mode. As depicted in figure 3.5, it is possible to detect the position of objects located in front of the radar antennas. However, if measurements are performed indoor, it is possible to detect walls or other static objects. In the context of the measurement reported in figure 3.5, a wall at 14 meters from the radar is also detected.
If the goal is to measure the range of moving targets exclusively, the Moving Target Indicator (MTI) algorithm, as shown in Figure 3.6, can be employed. This approach is also used for

[11] One should imagine the plot in figure 3.4 to change every T_p seconds. The highest peak represents the velocity of the strongest scatterer. Multiple targets can be measured as well.

measuring the range of multiple targets, as illustrated in Figure 3.7.

Figure 3.8 shows an example of processed data where it is possible to appreciate the real sync signal[12] (blue line) coming from the circuit (Figure 2.8) which points at the transmission of the up-chirp signal. The real sync signal is parsed during signal processing to obtain a clear square wave. The yellow signal in figure 3.8 is the received back-scatered signal. When the parsed sync signal is high, the beat frequency between the transmitted signal and the received signal is measured to recover range [13].

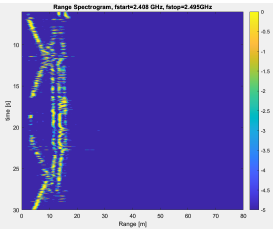


Figure 3.5: Single target range captured.

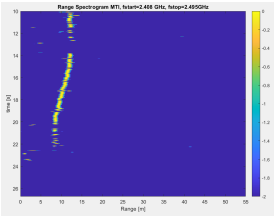


Figure 3.6: MTI Moving Target Indicator applied with a single target.

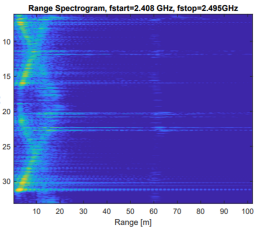


Figure 3.7: MTI applied with multiple targets.

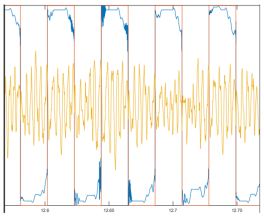


Figure 3.8: Captured datas.

4. SAR Image Signal Processing and Measurement

Synthetic-aperture radar (SAR) is a type of radar system that uses signal processing techniques to synthesize an antenna aperture that is much larger than the physical antenna, thus achieving enhanced angular resolution. This is accomplished by having an antenna mounted on a platform that moves along a trajectory. This require the target to not move. It means that **SAR is suited for landscape imaging**.
For the purposes of this project, the SAR will be implemented by mounting the radar built in 2 atop a movable platform. The platform will be translated in steps of few centimeters twenty-four times across a straight line. Each individual data collection will employ a frequency modulated continuous wave (FMCW) measurement. The acquired data is then processed using the Range Migration Algorithm.
The range migration algorithm (RMA) used for SAR signal processing is described in Section 4.1. Additionally, Section 4.2 reports on a significant experiment conducted by the group to demonstrate the capabilities of the SAR system.

[12] The real sync signal is the sync signal produced by the modulator discussed in 2.3
[13] Range can be recovered through equation $f_b = (4\Delta f f_m R)/c$, where f_b is the beat frequency, f_m is the parsed sync frequency, c is the speed of light and R is the desired range.

4.1. Range Migration Algorithm (RMA) Code Implementation and Testing

The radar measurement will produce an audio file that contains the information of the measurement. For this reason, the implemented algorithm should produce known results when applied to the provided audio file "SARTestFile.m4a".
The algorithm begins by creating the DataMatrix, where each row, indexed by 'i', contains a number of samples equal to $T_{rp} * SamplingFrequency$ derived from the 'i'-th measurement. It also generates the SyncMatrix, with each row indexed by 'i' representing the sync relative to the data in the 'i'-th row of the DataMatrix.
With DataMatrix and SyncMatrix being created, the RMA then constists of four steps:

1. **Cross Range Fourier Transform**: calculate the FFT of the spatial components. Figure 4.1 plots the magnitude of the cross range FFT and figure 4.2 plots the relative angle

2. **Mathced Filter**: Matched filters correlate a known signal (template) with an unknown signal to detect the template presence.

3. **Stolt Interpolation**: Transforms the 2-D SAR data from the range spatial frequency and azimuth (wave-number) domain (k_x, k_r) to the wave-number domain (k_x, k_r). The result of the Stolt interpolation is reported in figure 4.3

4. **Inverse Fourier Transform(IFFT)**: To convert the Stolt matrix into a image, a rectangle subsection of the data is used. The 2-D IDFT is applied to the rows and then columns seperately and the resulting SAR image is computed as shown in figure 4.4[14]

At the end of the RMA it is possible to distinguish the distance of static objects on the chosen landscape.

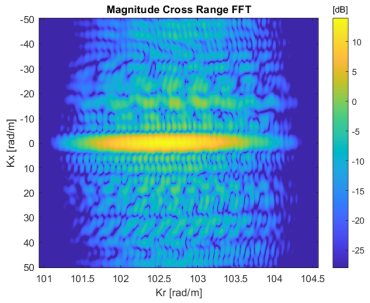


Figure 4.1:
Cross range magnitude.

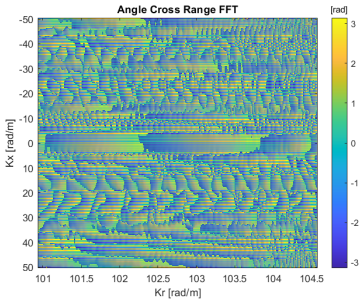


Figure 4.2:
Cross range phase.

[14] In figure 4.4 the cross range distance is plotted considering the absolute value of the range.

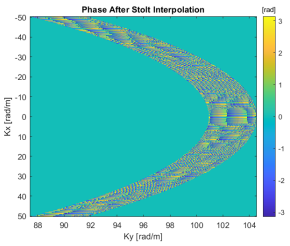


Figure 4.3:
Phase after Stolt interpolation.

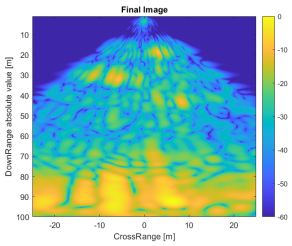


Figure 4.4:
SAR image.

4.2. SAR Test on a Landscape

In this subsection a test performed in a landscape is reported along with a satellite image for comparison. The location is a square with a lawn and some obstacles (figure 4.5). Figure 4.6 shows the satellite image of the same location. In figure 4.7 the resulting SAR image after RMA is depicted. The data has been normalized by the highest value that is the back-scattered signal coming from the closest central obstacle on the scene (a trunk placed on the ground at circa 5 m from the radar antenna).

Figure 4.5:
Landscape from the team point of view.

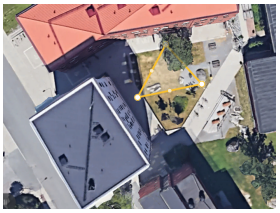


Figure 4.6:
Landscape from the satellite.

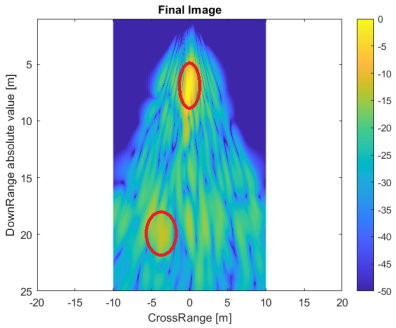


Figure 4.7:
Resulting SAR image.

5. Software Defined Radio at 5.8 GHz

The second phase of the project involves the creation of a radar system utilizing Software Defined Radio (SDR) technology operating at a frequency of 5.8 GHz. The software platform employed for this purpose is GNU Radio, a versatile toolkit designed for the implementation of software-defined radios and signal processing systems. The block diagram will be implemented using USRP B205mini-i. Within this context, GNU Radio was harnessed to develop the following radar functionalities:

1. A Continuous Wave CW mode radar, employing IQ demodulation without a Power Amplifier, enabling the measurement of both range and velocity for single or multiple targets.

2. Low-Intermediate Frequency LowIF CW mode radar, serving the same measurement purposes as described in item 1.

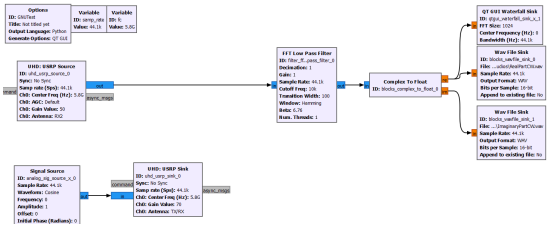


Figure 5.1:
Block diagram for CW mode radar.

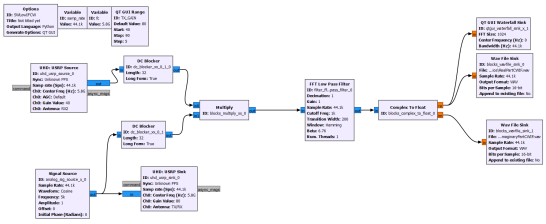


Figure 5.2:
Block diagram for CW Low IF mode radar.

In figure 5.1, the CW mode radar is realized by transmitting a 5.8 GHz signal using the *USRP-Sink* block. The receiver chain employs direct conversion[15] to down-convert the signal to baseband, achieved through the *USRP-Source* block[16]. Subsequently, a low-pass filter is used into the receiver chain to restrict the signal bandwidth, mitigating noise and interference. In this context, the *complex-to-float converter*[17] is employed to extract both the in-phase and quadrature signals.
Figure 5.2 illustrates a more intricate schematic used to implement the Low IF CW mode for the radar. In this configuration, the *Signal-Source* block generates a 4 kHz cosine waveform, which is then modulated by a 5.8 GHz carrier frequency and transmitted through the *USRP-Sink* block. The receiver chain initiates by down-converting the transmitted signal to the baseband. This down-conversion is achieved by multiplying the signal with a 5.85 GHz local oscillator. However, in the Low IF CW mode, the objective is not to recover the signal near DC but rather in the vicinity of the 4 kHz region within the spectrum. To accomplish this, a *lock-in amplifier* is utilized as a band-pass filter. The lock-in amplifier consists of a multiplier

[15] Direct conversion receivers multiply the RF signal with a Local Oscillator (L.O.) signal tuned at the same carrier frequency of the RF signal.
[16] USRP-Source block: A block in the USRP system used for signal reception
[17] Complex-to-float converter: A component used to recover both the in-phase (I) and quadrature (Q) signals.

driven by a phase-coherent 4 kHz cosine waveform (essentially the same 4 kHz cosine used for modulation) and a subsequent low-pass filter with a cut-off frequency located at 1kHz and a stop-band frequency located at 1.2kHz [18].

5.1. Velocity direction measurements

The purpose of this subsection is to describe the tests that were performed using the software-defined radio (SDR) and the application of IQ demodulation to determine the target velocity direction. In this section the power amplifier is not used. Figure 5.3 shows the velocity of two moving targets as captured by the SDR system in CW mode, while Figure 5.4 represents the same scenario as captured by the SDR system in CW Low IF mode. A similar test is also showed in figure 5.5 and 5.6 where this time only a single target velocity has been measured.
Using the available IQ demodulation system it is also possible to measure the velocity direction of the target, i.e., if the target is moving toward the radar or away from the radar.
Figure 5.7 plots the velocity magnitude of a target moving back and forth[19] when the SDR operates in CW mode[20] . Nevertheless, exploiting QM modulation, it is possible to process the in-phase (denoted as I) and the quadrature-phase signal (denoted as Q). In particular, processing the complex signal $I + jQ$ provides the velocity of a target that is moving away from the radar, while processing $I - jQ$ provides the velocity of a target that is moving toward the radar. The code developed for this purpose is reported in A.1.4, while figure 5.8 and figure 5.9 show the capability of the system to distinguish the directions.

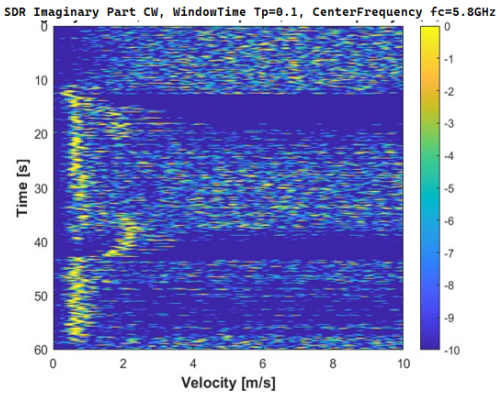


Figure 5.3:
Multiple targets detection in CW mode.

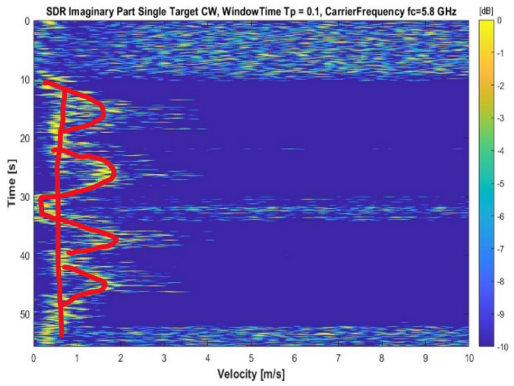


Figure 5.4:
Multiple targets detection in Low IF CW mode.

[18] It is of utmost significance that the low-pass filter effectively attenuates the 4 kHz frequency, as this frequency encapsulates information that was originally present at the 0 Hz frequency prior to the mixing process. If this requirement is not meet, there will be no performance difference between CW mode and Low IF CW mode.

[19] The velocity measurement algorithm developed in section 2 cannot distinguish velocity direction, for this reason the code needs to be adjusted accordingly. The code has been revisited in both A.1.4 and A.1.5

[20] Similar results can be obtained also in the Low IF CW mode. The purpose of the figures is to show tat the velocity direction is distinguishable

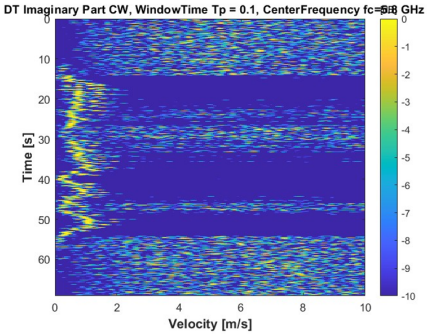


Figure 5.5:
Single target detection in CW mode.

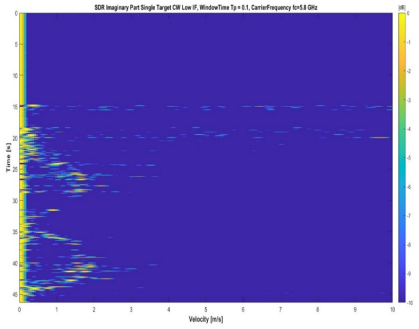


Figure 5.6:
Single target detection in Low IF CW mode.

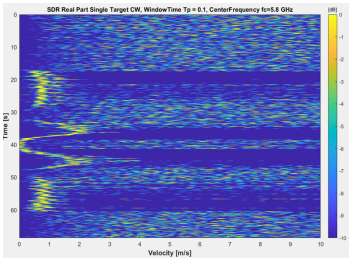


Figure 5.7:
Velocity absolute value of a single target.

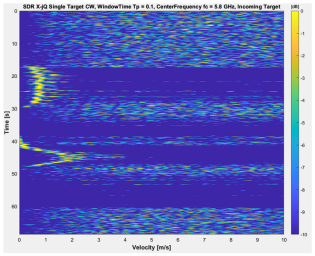


Figure 5.8:
Target velocity when it is moving toward the radar.

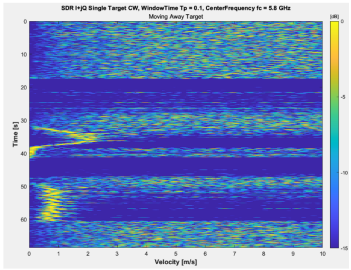


Figure 5.9:
Target velocity when it is moving away from the radar.

5.2. Gain sweep with external power amplifier and range dependence

Additional gain is provided by the external power amplifier that is located before the transmission antenna and after the transmitting port of USRP B205mini-i.
In the CW mode, in principle, increasing the gain should allow for a higher read back-scattered signal. However, for a high gain value, saturation of subsequent stage amplifiers is observed. This phenomenon is clearly evident through the evenly spaced red dots in the 85dB region of the waterfall plot in figure 5.10, which represents harmonic distortion resulting from saturation. From the same figure, it is also evident that the distortion is produced **after the down-conversion mixing stage**[21].
At lower power levels, the target (10 meters) is not distinctly visible. Increasing the gain improves target visibility, but if the gain is pushed to excessive levels, it can lead to a degradation in data quality due to noticeable harmonic distortion, as evidenced in the plot in Figure 5.11.
In a Low-IF architecture, the IF frequency is chosen to be close to DC (4 KHz in this context) or

[21] In a down-conversion chain, after the down-converting mixer, a VGA (Variable Gain Amplifier) is usually placed to provide from 20dB to 100dB gain for the base-band signal. This huge amplification can cause severe harmonic distortions when a powerful unwanted signal is coupled with a less powerful wanted signal.

at a lower frequency compared to the LO frequency. Figures 5.12 and 5.13 show that the CW Low IF mode is not badly affected by the increasing transmitted signal. The reason for this is that now there is no self mixing L.O. occurring.

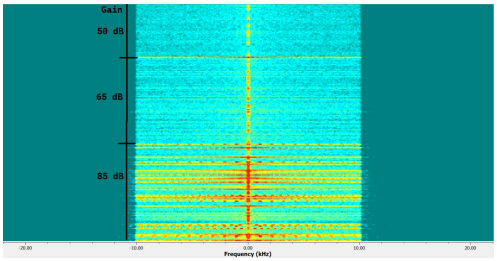


Figure 5.10:
Waterfall spectrogram as the gain increases.

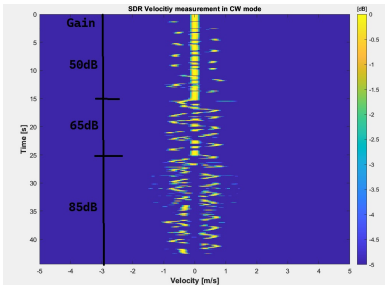


Figure 5.11:
CW, resulting spectrogram.

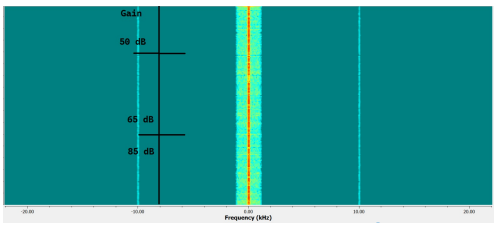


Figure 5.12:
CW IF Waterfall with variable gain.

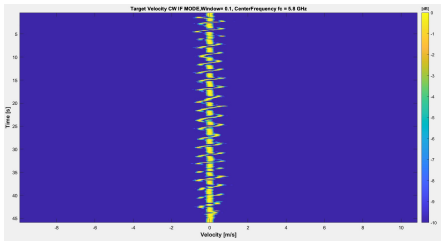


Figure 5.13:
CW IF, resulting spectrogram.

5.2.1. Is it the LO leakage or absolute output power which is the limiting factor?

Increasing the output power increases the range performances of the radar. However, a more powerful signal leaks from the transmitter chain to the receiver chain. If the radar is operating in a CW mode, the leaking signal is the down-converting L.O. itself. This is known as **self-mixing L.O.**[22] and creates a strong unwanted baseband signal. The Variable Gain Amplifier (VGA) then amplifies this signal, leading to harmonic distortion. The desired signal, i.e., the Doppler frequency shift, needs to be retrieved in proximity to the DC signal, which becomes increasingly contaminated by unwanted signals as the gain of the power amplifier is raised.
In Low IF CW radar mode, this issue is alleviated as the desired signal is not generated in close proximity to the Local Oscillator (L.O.). Instead, it is produced with a frequency offset from the L.O., thereby preventing it from being affected by the unwanted signal generated due to L.O. leakage.

[22] In direct conversion receivers, the self-mixing L.O. occurs at very high frequencies when the parasitism make the L.O. to leak toward previous RF stages. In this context however, the self-mixing L.O. occurs with a different leakage mechanism.

5.3. RANGE PERFORMANCES COMPARISON WITH AND WITHOUT POWER AMPLIFIER

In this subsection the results of measurements with and without the power amplifier are compared. In both measurements a person waves his hand like a pendulum at the distance of 3.3m from the radar antennas[23]. A human hand is expected to have a low radar cross-section. The spectrograms of the measurements are reported in the set of figures 5.14, 5.15, 5.16 and 5.17.

In the CW mode, for the range measurement, and with the power amplifier, the gain is varied from 40dB to 75dB. In figure 5.14 the spectrogram shows that the hand becomes registered after the gain is set over 50dB and that it is distinguishable all the way to 75dB, which is considered a success. The gain therefore should be set in the range of 60db to 75dB to have a clearer image.

Nevertheless, when the power amplifier is unplugged, the overall phase noise obtained by both the Local Oscillator and the V.C.O. (Voltage controlled oscillator) dominates the plot as shown in figure 5.15[24]. The phase noise has a $1/f^2$ (random walk) trend, meaning that a low frequency signal (low velocity in our case) tends to be less visible unless one increases the signal to noise ratio to overcome the phase noise power spectral density even at lower frequencies. The reason for this is that without the power amplifier the SNR at the receiver is drastically reduced, therefore a power amplifier is desirable to detect low Doppler frequency shifts in the CW setup.

Transitioning to the Low IF CW mode, one would anticipate similar range performance as depicted in figure 5.14. Indeed, figure 5.16 shows that the waving hand is clearly detected only if the gain of the power amplifier is beyond 50dB. The best visibility is around 55dB to 60dB. Beyond this gain the target visibility is reduced as it could be that the target is reducing its own radar cross-section while swinging the hand vigorously. Without amplification the received signal is again dominated by the phase noise of the Local Oscillator[25], as depicted in figure 5.17 and therefore almost indistinguishable.

[23] Since the measurements involve human subjects, it is important to acknowledge that minor imperfections may arise, with the most significant being the variability in radar cross-section.

[24] Phase noise is not an issue in 5.2 since the target is a human body having a larger radar cross section. In this context the hand was chosen on purpose.

[25] The phase noise of the local oscillator couples with the up-converted 4kHz cosine. In the receiver chain, when the signal gets down-converted, a proper SNR still has to be guaranteed since the phase noise has not been removed.

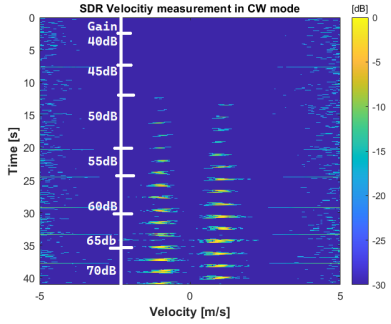


Figure 5.14:
SDR Velocity measurement in CW mode with power amplifier, target at 3.3m.

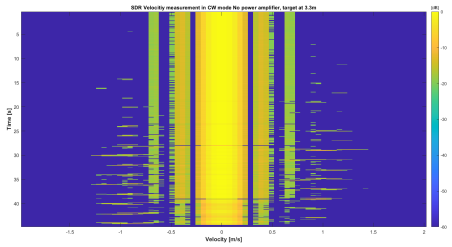


Figure 5.15:
SDR velocity measurement in CW mode, No power amplifier, target at 3.3m.

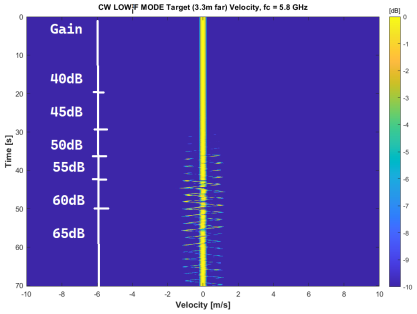


Figure 5.16:
SDR velocity measurement in CW Low IF mode, with power amplifier, target at 3.3m

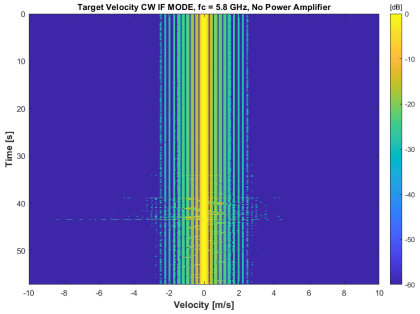


Figure 5.17:
SDR velocity measurement in CW Low IF mode, no power amplifier, target at 3.3m

5.4. Monitor breathing pattern and heart rate of an individual

This section of measurements was focused on the low IF CW. Low IF CW radar is more sensitive to minute movements, such as chest wall motion caused by the heartbeat. This heightened sensitivity allows for the detection of subtle physiological signals related to the heart rate, making it an effective method for monitoring heartbeats. This measurement displays such an attempt to detect the breathing and heart-rate of an individual.

Figure 5.18 shows the movement of the chest while measuring the breathing pattern of a specific target. This is achieved using the static phase equation for the range of the moving target, i.e., the chest of a person. The phase difference between the I and the Q signals contains information about the range variation of the target[26]. In this context, the target distance from the radar is 55cm. Figure 5.19 is the FFT of the breathing pattern. It gives an idea of the spectral components of the respiration signal.

[26] Utilizing the formula $R = c/(4\pi f_c)\phi$, a clear breathing pattern of a person standing in front of the radar can be obtained.

Figure 5.20 is the breathing pattern, similar to the figure 5.18, except here, instead of using CW low IF, CW is used. The breathing pattern is not visible clearly. The reason might be that the phase noise is too high in this mode that the low frequency signal isn't visible as phase noise dominates the low frequencies with a relation $1/f^2$. As a result, the phase from which the range is recovered appears to be noisy. Figure 5.21 shows a plot of the relative phase difference between I and Q. From the graph, it can be deduced that the phase is noisy and thus the resulting range in CW mode is unreliable.

Furthermore, the figures 5.22 and 5.20 refer to the heart rate measurements. Figure 5.22 is the measurement of a person's heart beat. The target was asked to hold their breath during the measurements. Also, the target slight movements during the measurement have to be taken into account as the figure shows that the target is getting farther from the radar (order of millimeters!). Figure 5.23 is the resulting FFT of the heart rate measurement. It is observed that the peak is around 1Hz and it points to a reasonable heart rate for a person.

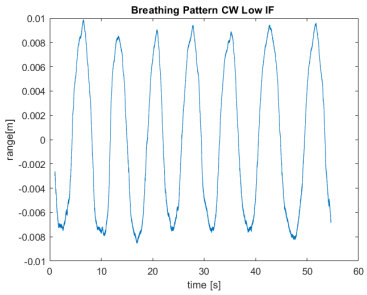


Figure 5.18:
Breathing pattern CW Low IF mode

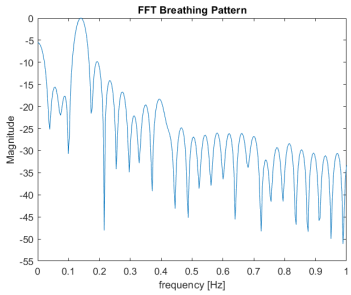


Figure 5.19:
FFT of breathing pattern

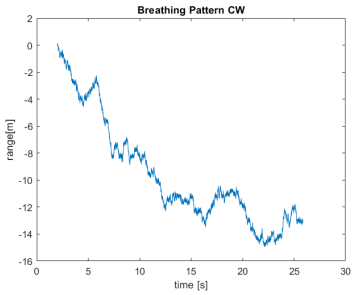


Figure 5.20:
Breathing pattern CW mode

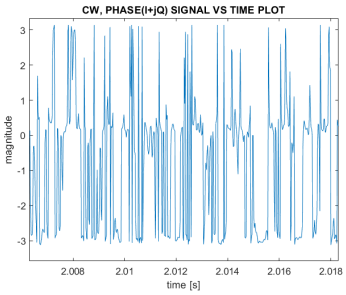


Figure 5.21:
CW mode, Phase(I+JQ) vs Time plot.

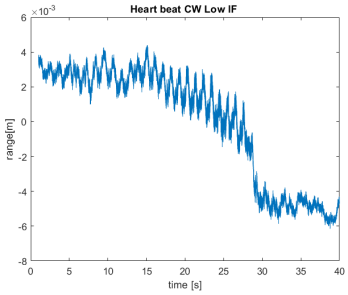


Figure 5.22:
Heart rate of a person.

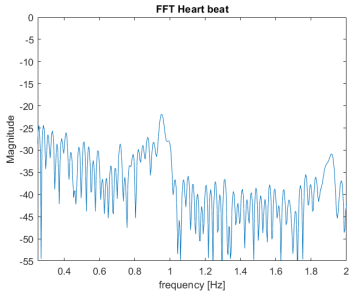


Figure 5.23:
Resulting heart beat FFT.

6. Extra tasks

The group was able to achieve the following extra tasks succesfully:

1. Movie of real-time/ pseudo real-time velocity measurements,
2. movie of real-time/ pseudo real-time range measurements.

Other extra tasks were not done due to lack of time and academic commitments. The group members have not a solid background in signal processing. However, what it was accomplished satisfies group members as they are proud of the work done, while trying something new. The missing extra tasks are:

1. Extract velocity from range using two point finite difference method and compare it to the slope of the range cure,
2. simultaneous range and velocity estimation by processing the up and down chirp only for a single target (for extra mark)
3. Range measurement for a single target, at close range (using target motion within the ambiguous range) with two-step frequency (square wave) CW mode in SDR. Use the step height of 100 MHz which means that the unambiguous range is 1.5 meters.

7. Final Considerations on the project

The group found the practical hardware implementation and experimentation in the electronics lab to be a very engaging and rewarding aspect of the project. Developing the short-range radar circuitry and characterizing its performance provided valuable hands-on experience. However, some expectations around the depth of learning in the digital signal processing domain were not entirely met. While motivated to explore challenging new topics like SAR processing, the limitations of the group's existing signal processing foundations meant that time had to be prioritized for simply implementing algorithms rather than developing a comprehensive theoretical understanding. In retrospect, committing additional early efforts to

consolidate baseline signal processing concepts may have allowed for more effective experimentation and true mastery of the material as the project progressed. Overall, the scale and open-ended nature of the radar systems design presented an excellent learning experience.

Appendices

A. MATLAB CODES

These codes were implemented as part of the project. The structure was designed with simplicity in mind to optimize memory usage and enhance code readability. Comprehensive details are documented within the code comments.

A.1. MAIN CODES

A.1.1. CONTINOUS WAVE VELOCITY SIGNAL PROCESSING

```
clear all;
close all;
clc;
% Constants
fc = 2.43e9;
Tp = 0.1;
N = 4;
c  = 299792458;          % speed of light
SamplingFrequency=44100; % Referred to the audio signal

% Data from Audio
% An audio has Stereo/Mono options.
% In this case we have a stereo file,
% thus the audioread() function provides 2 column vectors.
% Only one of these two vectors contains our data.
AudioVector = audioread("Task3_1_b.m4a");
AudioVector = -AudioVector';
AudioVector = AudioVector(1,:); % select datas from the first row
AudioVector = AudioVector-mean(AudioVector); % Mean subtraction
% time duration of input data
AudioDuration = length(AudioVector)/SamplingFrequency;
% time axis
dt = 1/SamplingFrequency;
t = 0 : dt : AudioDuration-dt;
%frequency and velocity axis
BandWidth = SamplingFrequency;
bin = SamplingFrequency/(length(AudioVector)*(N+1));
f = 0 : bin : BandWidth-bin;
v = f * (c/(2*fc));   % Velocity axis

% Ensemble Matrix initialisation
% Number of time windows that we can construct from our signal
NumberOfTimeWindows = floor(AudioDuration/(Tp));
SamplesInWindow = Tp*SamplingFrequency;
WindowCollectionMatrix = zeros(NumberOfTimeWindows,...
SamplesInWindow*(1+N));

% Filling the Ensemble Matrix With also Zero Padding
% This is the way we perform Zero Padding in this script
% If A = [1 2 3] and we want to add a certain chunck of zeros
% ZeroVector = zeros(1, N*length(A));
% AZeroPadding = [A ZeroVector];
% We have to do this procedure for all rows of our EnsembleMatrix
    for kk = 1 : NumberOfTimeWindows

    % TemporaryVector is a local utilised vector, it contains
    % the AudioVector datas associated to the kk-th Window.
    TemporaryVector = AudioVector( ((kk-1)*SamplesInWindow +1) :
          ((kk)*SamplesInWindow) );
    TemporaryZeroVector = zeros(1, N*length(TemporaryVector));
    TemporaryVectorZeroPadding = [TemporaryVector
          TemporaryZeroVector];

    WindowCollectionMatrix(kk,:) = TemporaryVectorZeroPadding;
end

% FFT
Spectrogram = zeros(size(WindowCollectionMatrix));

for kk = 1 : NumberOfTimeWindows

    Spectrogram(kk,:) = fft(WindowCollectionMatrix(kk,:));

end

% Spectrogram dB conversion

 Spectrogram = 20*log10( abs(Spectrogram) );
 % Normalization 2 as default
%
% This time we normalise row by row dividing all the row by the
% maximum number in that row
Spectrogram = Spectrogram-max(Spectrogram')';
figure(2)
imagesc(v,t,Spectrogram); axis([0 30 0 AudioDuration]); clim([-15
      0])
text = "CW, Norm.2 PulseTime Tp = " +Tp+", CenterFrequency
      fc="+fc/1e9+" GHz";
title(text)


% This is just plot fashion.
xlabel("Velocity [m/s]",'FontSize',12,'FontWeight','bold');
ylabel("Time [s]",'FontSize',12,'FontWeight','bold');
hcb=colorbar;
hcb.Title.String = "[dB]";
```

A.1.2. FMCW RANGE MEASUREMENT ALGORITHM

```
clear all;
close all;
clc;
% constants
c  = 299792458;       % speed of light
fstart =2.408e9;      % starting up-chirp frequency
fstop =2.495e9;       % stopping up-chirp frequency
Deltaf = fstop-fstart;
Tp = 2e-2;
% Data recovering from audio file
RangeTestFile = audioread("Task3_2_b.m4a");
RangeTestFile = -(RangeTestFile)';
BackscateredData = RangeTestFile(1,:);
Sync = RangeTestFile(2,:);

% Parsing Sync
ParseSync = -ones(1, length(Sync(1,:)));
for i = 1 : length(Sync(1,:))
    if Sync(i) > 0
        ParseSync(i) = 1;
    elseif Sync(i) < 0
        ParseSync(i) = -1;
    end
end
% Counting the number of upchirps
%
% NumberUpchirps will be used to count the
% rows of spectrogram. The proposed algorithm
% runs through all the samples of Sync.
% If  Sync is >0 then NumberUpchirps++ and
% we stop adding the NumberUpchirps by setting
% high WaitFor0. When a 0 comes, we set low
% WaitFor0 and we wait the next upchirp

NumberUpchirps = 0;
WaitFor0 = 0;
for i = 1:length(Sync)

    if Sync(i) < 0
        WaitFor0 = 0;
    end
    if Sync(i) > 0 && WaitFor0 == 0
        NumberUpchirps = NumberUpchirps + 1;
        WaitFor0 = 1;
    end

end
% Time axis
SamplingFrequency = 44100;
dt = 1/SamplingFrequency;
AudioDuration = length(Sync)*dt;
t= (0:NumberUpchirps-1)*2*Tp;
% Ensemble Matrix
Tp = 0.02;          %Upchirp time
EnsembleMatrix = zeros(NumberUpchirps, Tp*SamplingFrequency);
SampleCounter = 1;
for k = 1:NumberUpchirps
    %We increase SampleCounter untill Sync goes up
    while SampleCounter<=length(Sync) && Sync(SampleCounter)<=0.5
        SampleCounter = SampleCounter + 1;
    end
    % Check if SampleCounter went beyond the length of Sync.
    % This is to avoid errors
    if SampleCounter > length(Sync)
        break;
    end
    % Fill EnsembleMatrix with BackscateredData when Sync is high
    for j = 1 : Tp*SamplingFrequency
        if(SampleCounter+j <= length(ParseSync))
            EnsembleMatrix(k, j) = BackscateredData(SampleCounter +
                j);
        end
    end
    while SampleCounter <= length(Sync) && Sync(SampleCounter) >
        -0.5
    SampleCounter = SampleCounter + 1;
    end
end
% Zero Padding
% In order to better see the IFFT
N = 20;
EnsembleMatrixZeroPadding = zeros( NumberUpchirps , ...
(N+1)*length( EnsembleMatrix(1,:) ) );
EnsembleMatrixAddingZero = zeros(1 ,
      N*length(EnsembleMatrix(1,:)));
for i = 1 :NumberUpchirps
    EnsembleMatrixZeroPadding(i,:) = [EnsembleMatrix(i,:)...
    EnsembleMatrixAddingZero];
end
% MS Mean Subtraction
% We subtract the mean of each row
for i = 1 : length(EnsembleMatrix(1,:))
    EnsembleMatrixZeroPadding(:,i)=EnsembleMatrixZeroPadding(:,i)-
    mean(EnsembleMatrixZeroPadding(:,i));
end
EnsembleMatrixZeroPadding = MTI_Radar(EnsembleMatrixZeroPadding);
% ifft
%
% Here we create the Spectrogram and we turn it into dB
DataSpectrogram = zeros(size(EnsembleMatrixZeroPadding));
for i = 1: NumberUpchirps
    DataSpectrogram(i,:) = ifft(EnsembleMatrixZeroPadding(i,:));
end

DataSpectrogram = 20*log10(abs(DataSpectrogram));
DataSpectrogram = DataSpectrogram -max(DataSpectrogram,[],"all");
%cleaning spectrogram
threshold =-25;
indices_below_threshold = find(DataSpectrogram < threshold);
% Replace those elements with the desired value (-60)
DataSpectrogram(indices_below_threshold) = -60;
%range axis
Rmax = c*Tp/2;
dR = c/(2*Deltaf*(N+1));
R = 0:dR:dR*length(DataSpectrogram(1,:))-dR;

figure(1)
imagesc(R,t,DataSpectrogram(:,1:length(R))); clim([-50 0]);
      axis([0 100 0 AudioDuration-dt]);
xlabel("Range [m]"); ylabel("time [s]");
title("Range Spectrogram, fstart="+fstart*1e-9+" GHz,
      fstop="+fstop*1e-9+"GHz");
```

```
clear all;
close all;
clc;
% constants
c  = 299792458;          % speed of light
SamplingFrequency = 44100;
Tp = 0.02;               % Upchirp time
fstart =2.408e9;         % starting up-chirp frequency
fstop =2.495e9;          % stopping up-chirp frequency
Deltaf = fstop-fstart;
% Observation time
Trp = 0.02;              % segment duration
SamplesInTrp = Trp*SamplingFrequency;
% Audio manipulation
AudioVector = audioread("SAR_Test_File.m4a");
AudioVector = -(AudioVector)';
BackscateredData = AudioVector(1,:);
Sync = AudioVector(2,:);

% Time axis, for plot purpose
dt = 1/SamplingFrequency;
t_audio = (0 : length(AudioVector(1,:))-1)*dt;
t = 0: dt : Trp-dt;                          % the whole Trp
t_upchirp = 0:dt:Tp-dt;                      % 1 upchirp only
% ------------------------- WORKING PRINCIPLE
      ------------------------- %
%  We run through all the acquired samples. Two flags are used in
      this
%  context, namely
%     *flag_wait:
%         =1 indicates that we have collected Trp*SampleFrequency
      samples
%            so we have to wait for the FMCW to be turned off.
%         =0 indicates that we should have the FMCW turned off
      since the
%            Sync signal should be flat.
%         /* comment */
%         / problems may occour due to false triggering of
      flag_wait=0 /
%         / this major issue is caused by spurious samples in the
      sync /
%         / To solve this problem, secure_counter is added.
      flag_wait /
%         / can be put back to 0 only if at least
      Tp*SamplingFrequency /
%         / non-high and non-low values are counted.            /
%
% this is done through this custom function
[DataMatrix,SyncMatrix,ParseSyncMatrix,NumberOfUpchirpMatrix] =...
   ExtractSyncedDataSegments(BackscateredData,Sync,Trp,Tp);

figure(1)
plot(t, SyncMatrix(12,:)); hold on; plot(t,
      ParseSyncMatrix(1,:)); hold on; plot(t,DataMatrix(1,:))
xlabel("time [s]"); ylabel(" normalized magnitude"); title(" 12th
      segment example");

% We have to integrate through all the upchirps in a specific
      position(i.e.
% a row) and put the result of integration into
      IntegratedDataMatrix (thus we are
% parsing Data matrix)
IntegratedDataMatrix =
      IntegrateSyncedDataSegments(DataMatrix,ParseSyncMatrix,...
NumberOfUpchirpMatrix,Tp,Trp,SamplingFrequency);
%- Hilbert transform with hann windowing
CrossRangeMatrix = SARCustomHilbert(IntegratedDataMatrix,'hann');
% kr axis
dKr = 8*pi*Deltaf/(c*length(DataMatrix(1,:)));
Kr = ((4*pi*fstart)/c) : dKr : ((4*pi*fstop)/c)-dKr;
% X axis
lambda = c/fstart;
L = lambda;
dL = L/length(DataMatrix(:,1));
Xa = -L/2 : dL : L/2-dL;

figure(2)
imagesc(Kr, Xa, angle(CrossRangeMatrix));
xlabel('Kr [rad/m]'); ylabel('Synthetic Aperture Position
      Xa[m]'); title('Phase before Column track FFT');
clim([-pi pi]); colorbar; title(colorbar,'[rad]')

% FFT zero padding of CrossRangeMatrix
DummyZero = zeros(1024,length(CrossRangeMatrix));
CrossRangeMatrix = [DummyZero; CrossRangeMatrix; DummyZero];
      %performs zero pdding
CrossRangeMatrixFFT = fftshift(fft(CrossRangeMatrix,[],1),1);
CrossRangeMatrixFFT = CutLowValue(CrossRangeMatrixFFT,-25);

% Kx axis
dKx = 4*pi/(lambda*length(CrossRangeMatrix(:,1)));
Kx = (-2*pi/lambda) : dKx : (2*pi/lambda)-dKx;
% New Kr axis. It has to be adapted with respect to the zero
      padding
dKr = 4*pi*Deltaf/(c*length(CrossRangeMatrixFFT(1,:)));
Kr = ((4*pi*fstart)/c) : dKr : ((4*pi*fstop)/c)-dKr;

figure(3)
subplot(2,1,1)
imagesc(Kr,Kx,20*log10(abs(CrossRangeMatrixFFT)));
xlabel('Kr [rad/m]'); ylabel('Kx [rad/m]'); title('Magnitude
      After Column Track FFT');
clim([-30 0]); colorbar; title(colorbar,'[dB]')

subplot(2,1,2)
imagesc(Kr,Kx,angle(CrossRangeMatrixFFT));
xlabel('Kr [rad/m]'); ylabel('Kx [rad/m]'); title('Angle After
      Column Track FFT');
clim([-pi pi]); colorbar; title(colorbar,'[rad]')

%% Interpolation
% it is done entirely by means of CustomInterpolation function

[intpolCrossRangeMatrixFFT,Kye] =
      CustomInterpolation(Kr,Kx,CrossRangeMatrixFFT);

figure(4)
imagesc(Kye,Kx,angle(intpolCrossRangeMatrixFFT));
xlabel('Ky [rad/m]'); ylabel('Kx [rad/m]'); title('Phase After
      Stolt Interpolation');
clim([-pi+pi/10 pi-pi/10]); colorbar; title(colorbar,'[rad]')

%% Final Image extraction

c_range_1 = -25;
c_range_2 = 25;
d_range_1 = 1;
d_range_2 = 100;

[FinalTruncatedImage, CrossRange, DownRange] =
      GetImageFromCustomInterpolation( ...
   intpolCrossRangeMatrixFFT, Kye, c_range_1, c_range_2,
        d_range_1, d_range_2);

figure(5)
imagesc(CrossRange,DownRange, FinalTruncatedImage); colorbar;
      clim([-60 0]);
axis([c_range_1 c_range_2 d_range_1 d_range_2]);
xlabel('CrossRange [m]'); ylabel('DownRange absolute value [m]');
      title('Final Image');
```

A.1.4. SDR VELOCITY DETECTION AND VELOCITY DIRECTION ALGORITHM

```
fc = 5.8e9; Tp = 0.1; N = 4;
c  = 299792458;
SamplingFrequency=44100;
I = audioread("RealPartCWIF.wav")';
I = I-mean(I);
AudioDuration = length(I)/SamplingFrequency;
dt = 1/SamplingFrequency;
t = 0 : dt : AudioDuration-dt;
BandWidth = SamplingFrequency;
bin = SamplingFrequency/(length(I)*(N+1));
f = 0 : bin : BandWidth-bin;
v = f * (c/(2*fc)); %velocity axis
NumberOfTimeWindows = floor(AudioDuration/(Tp));
SamplesInWindow = Tp*SamplingFrequency;
WindowCollectionMatrixI = zeros(NumberOfTimeWindows,
      SamplesInWindow*(1+N));
for kk = 1 : NumberOfTimeWindows
    TemporaryVector = I( ((kk-1)*SamplesInWindow +1) :
          ((kk)*SamplesInWindow) );
    TemporaryZeroVector = zeros(1, N*length(TemporaryVector));
    TemporaryVectorZeroPadding = [TemporaryVector
          TemporaryZeroVector];
    WindowCollectionMatrixI(kk,:) = TemporaryVectorZeroPadding;
end
SpectrogramI = zeros(size(WindowCollectionMatrixI));
for kk = 1 : NumberOfTimeWindows
    SpectrogramI(kk,:) = fft(WindowCollectionMatrixI(kk,:));
end
 SpectrogramI = 20*log10( abs(SpectrogramI) );
 SpectrogramI = CutLowValue(SpectrogramI,-60);
SpectrogramI = SpectrogramI-max(SpectrogramI')';
Q = audioread("ImaginaryPartCWIF.wav")';
Q = Q-mean(Q);
NumberOfTimeWindows = floor(AudioDuration/(Tp));
WindowCollectionMatrixQ = zeros(NumberOfTimeWindows,
      SamplesInWindow*(1+N));
for kk = 1 : NumberOfTimeWindows
    TemporaryVector = Q( ((kk-1)*SamplesInWindow +1) :
          ((kk)*SamplesInWindow) );
    TemporaryZeroVector = zeros(1, N*length(TemporaryVector));
    TemporaryVectorZeroPadding = [TemporaryVector
          TemporaryZeroVector];
    WindowCollectionMatrixQ(kk,:) = TemporaryVectorZeroPadding;
end
SpectrogramQ = zeros(size(WindowCollectionMatrixQ));
for kk = 1 : NumberOfTimeWindows
    SpectrogramQ(kk,:) = fft(WindowCollectionMatrixQ(kk,:));
end
SpectrogramQ = 20*log10( abs(SpectrogramQ) );
SpectrogramQ = SpectrogramQ-max(SpectrogramQ')';
SpectrogramQ = CutLowValue(SpectrogramQ,-10);
X = I + 1i * Q; % Create complex signal
WindowCollectionMatrixX = zeros(NumberOfTimeWindows,
      SamplesInWindow * (1 + N));
for kk = 1 : NumberOfTimeWindows
TemporaryVector = X(((kk - 1) * SamplesInWindow + 1) : (kk *
      SamplesInWindow));
    TemporaryZeroVector = zeros(1, SamplesInWindow * N);
    TemporaryVectorZeroPadding = [TemporaryVector,
          TemporaryZeroVector];
    WindowCollectionMatrixX(kk, :) = TemporaryVectorZeroPadding;
end
SpectrogramX1 = zeros(size(WindowCollectionMatrixX));
for kk = 1 : NumberOfTimeWindows
    SpectrogramX1(kk, :) = fft(WindowCollectionMatrixX(kk, :));
end
SpectrogramX1Copy = SpectrogramX1;
SpectrogramX1 = 20 * log10(abs(SpectrogramX1));
SpectrogramX1 = SpectrogramX1 - max(SpectrogramX1')';
X = I - 1i * Q;
WindowCollectionMatrixX = zeros(NumberOfTimeWindows,
      SamplesInWindow * (1 + N));
for kk = 1 : NumberOfTimeWindows
    TemporaryVector = X(((kk - 1) * SamplesInWindow + 1) : (kk *
          SamplesInWindow));
    TemporaryZeroVector = zeros(1, SamplesInWindow * N);
    TemporaryVectorZeroPadding = [TemporaryVector,
          TemporaryZeroVector];
    WindowCollectionMatrixX(kk, :) = TemporaryVectorZeroPadding;
end
SpectrogramX2 = zeros(size(WindowCollectionMatrixX));
for kk = 1 : NumberOfTimeWindows
    SpectrogramX2(kk, :) = fft(WindowCollectionMatrixX(kk, :));
end
SpectrogramX2Copy = SpectrogramX2;
SpectrogramX2 = 20 * log10(abs(SpectrogramX2));
SpectrogramX2 = SpectrogramX2 - max(SpectrogramX2')';
SpectrogramX2 = CutLowValue(SpectrogramX2,-30);
SpectrogramX1 = CutLowValue(SpectrogramX1,-30);

vdsb = [-fliplr(v) v];
DirectionSpectrogram = [fliplr(SpectrogramX2) SpectrogramX1];
figure(1)
imagesc(vdsb, t, fliplr(DirectionSpectrogram));
axis([-10 10 0 AudioDuration]);
clim([-60 0])
title("Target Velocity CW IF MODE, fc = " + fc / 1e9 + " GHz")
xlabel("Velocity [m/s]", 'FontSize', 12, 'FontWeight', 'bold');
ylabel("Time [s]", 'FontSize', 12, 'FontWeight', 'bold');
hcb = colorbar;
hcb.Title.String = "[dB]";
figure(2)
subplot(2,2,1)
plot(t,I); xlabel("time [s]"); ylabel("magnitude"); title("CW LOW
      IF, REAL PART SIGNAL VS TIME PLOT")
subplot(2,2,2)
plot(t,Q); xlabel("time [s]"); ylabel("magnitude"); title("CW LOW
      IF, IMAG. PART SIGNAL VS TIME PLOT")
subplot(2,2,3)
plot(t,abs(X)); xlabel("time [s]"); ylabel("magnitude");
      title("CW LOW IF, ABS(I+jQ) SIGNAL VS TIME PLOT")
subplot(2,2,4)
plot(t,angle(X)); xlabel("time [s]"); ylabel("magnitude");
      title("CW LOW IF, PHASE(I+jQ) SIGNAL VS TIME PLOT")
```

A.1.5. BREATHING PATTERN AND HEART RATE CODE

```
fc = 5.8e9; Tp = 0.1; N = 5;
c  = 299792458;
SamplingFrequency=44100;
I = audioread("RealPartCWIF.wav")';
I = I-mean(I);
Q = audioread("ImaginaryPartCWIF.wav")';
Q = Q-mean(Q);
AudioDuration = length(I)/SamplingFrequency;
dt = 1/SamplingFrequency;
t = 0 : dt : AudioDuration-dt;
t_0=1;
t_1=40;
t = t(t_0*SamplingFrequency:t_1*SamplingFrequency);
I = I(t_0*SamplingFrequency:t_1*SamplingFrequency);
Q = Q(t_0*SamplingFrequency:t_1*SamplingFrequency);
X = I + 1i * Q; % Create complex signal
figure(2)
plot(t,angle(X)); xlabel("time [s]"); ylabel("magnitude");
title("CW LOW IF, PHASE(I+jQ) SIGNAL VS TIME PLOT")
figure(3)
Range = (c/(4*pi*fc))*unwrap(angle(X));
Range = Range-mean(Range);
plot(t,-Range); xlabel("time [s]"); ylabel("range[m]");
title("Heart beat CW Low IF ")
BW = SamplingFrequency;
bin = BW/(length(t)*(N+1));
f = 0:bin:BW-bin;
figure(4)
ZeroPaddingRange = zeros(1,N*length(t));
Range = [Range ZeroPaddingRange];
Range_FFT = fft([Range])/max(abs(fft(Range)));
plot(f,20*log10(abs(Range_FFT))); xlabel("frequency [Hz]");
ylabel("Magnitude"); title("FFT Heart beat");
axis([0.25 2 -55 0]);
```

A.2. CUSTOM FUNCTIONS

A.2.1. CUSTOMINTERPOLATION

```
function [intpolCrossRangeMatrixFFT,Kye] =
      CustomInterpolation(Kr,Kx,CrossRangeMatrixFFT)
%CUSTOMINTERPOLATION computes the interpolated Cross-Range Matrix
      FFT and returns the
% corresponding cross-range wave numbers.
% Syntax:
%   [intpolCrossRangeMatrixFFT, Kye] = KyMatrix(Kr, Kx,
      CrossRangeMatrixFFT)
% Input Arguments:
%   - Kr: Vector of range wave numbers.
%   - Kx: Vector of cross-range wave numbers.
%   - CrossRangeMatrixFFT: Original Cross-Range Matrix in the wave
      number domain.
% Output Arguments:
%   - intpolCrossRangeMatrixFFT: Interpolated Cross-Range Matrix
      in the wave number domain.
%   - Kye: Vector of interpolated cross-range wave numbers.
% Description:
%   This function takes the range wave numbers (Kr), cross-range
      wave numbers (Kx), and
%   the original Cross-Range Matrix in the wave number domain
      (CrossRangeMatrixFFT) as
%   input. It computes the cross-range wave numbers (Kyr) for each
      Kx value and then
%   interpolates the Cross-Range Matrix to obtain
      intpolCrossRangeMatrixFFT using the
%   interpolated Kyr values.
%   NaN values in the interpolated Cross-Range Matrix are replaced
      with a small
%   positive value (1e-30) to avoid numerical issues.
%   The function returns the interpolated Cross-Range Matrix
      (intpolCrossRangeMatrixFFT)
%   and the corresponding interpolated cross-range wave numbers
      (Kye).
    Ky = zeros( length(Kx), length(Kr) );
    for jj = 1 : length(Kx)
        Ky(jj,:) = sqrt(Kr.^2-Kx(jj).^2);
    end

    Kye_min =min(Ky,[],'all');
    Kye_max =max(Ky,[],'all');
    dKye = (Kye_max-Kye_min)/(length(Kx)/2);
    Kye = Kye_min : dKye : Kye_max-dKye;

    % interpolated CrossRangeMatrixFFT
    intpolCrossRangeMatrixFFT =
          zeros(length(CrossRangeMatrixFFT(:,1)),length(Kye));
    for jj = 1 : length(CrossRangeMatrixFFT(:,1))
        intpolCrossRangeMatrixFFT(jj,:) =
              interp1(Ky(jj,:),CrossRangeMatrixFFT(jj,:),Kye);
    end
    WhereNaN = isnan(intpolCrossRangeMatrixFFT);
    intpolCrossRangeMatrixFFT(WhereNaN)=1e-30;
end
```

A.2.2. CUTLOWVALUE

```
function B = CutLowValue(A,threshold)
%CUTLOWVALUE function replaces values in A that are less than or equal to a specified threshold with a very small value (1e-30).
% Inputs:
%   A - Input matrix.
%   threshold - Threshold for cutting low values.
% Output:
%   B - Output matrix with low values replaced.
%
    find_in_A = find(A<=threshold);
    A(find_in_A) = -120;
    B = A;
end
```

A.2.3. ENSEMBLEMATRIXZPFILL

```
function outmatrix
EnsembleMatrixZPfill(ParseSync,BackscateredData,NumberUpchirps,N)
% Creates the EnsembleMatrix from BackscateredData,
% with zero padding specified by N and using ParseSync.
% N=0 means No zero padding
% ParseSync must go from -1 to 1.
    Tp = 0.02;
    SamplingFrequency = 44100;
    LocalEnsembleMatrix = zeros(NumberUpchirps,
          Tp*SamplingFrequency);
    SampleCounter = 1; %This variable is used to count all
          BackscateredData samples
    for kk = 1:NumberUpchirps
        %We increase SampleCounter untill Sync goes up
        while SampleCounter <= length(ParseSync) &&
              ParseSync(SampleCounter) <= 0.5
            SampleCounter = SampleCounter + 1;
        end
        % Check if SampleCounter went beyond the length of Sync.
        if SampleCounter >= length(ParseSync)
            break;
        end
        % Fill EnsembleMatrix with BackscateredData if Sync high
        for j =1 : Tp*SamplingFrequency
            if(SampleCounter+j <= length(ParseSync))
                LocalEnsembleMatrix(kk, j) =
                      BackscateredData(SampleCounter + j);
            end
        end
    % Move SampleCounter to the start of the next low (<= -0.5)
          region in Sync
        while SampleCounter < length(ParseSync) &&
              ParseSync(SampleCounter) > -0.5
            SampleCounter = SampleCounter + 1;
        end
    end
    % Zero Padding
    % In order to better see the IFFT,

    LocalEnsembleMatrixZeroPadding = zeros( NumberUpchirps ,
          (N+1)*length( LocalEnsembleMatrix(1,:) ) );
    LocalEnsembleMatrixAddingZero = zeros(1 , N*length(
          LocalEnsembleMatrix(1,:) ) );
    for i = 1 :NumberUpchirps
        LocalEnsembleMatrixZeroPadding(i,:) =
              [LocalEnsembleMatrix(i,:)
              LocalEnsembleMatrixAddingZero];
    end
    outmatrix = LocalEnsembleMatrixZeroPadding ;
end
```

A.2.4. ExtractSyncedDataSegments

```
      function [DataMatrix,SyncMatrix,ParseSyncMatrix, ...
      NumberOfUpchirpMatrix] = ...
ExtractSyncedDataSegments(BackscateredData,Sync,Trp,Tp)
%EXTRACTSYNCEDDATASEGMENTS Extracts synchronized data segments
      from backscattered data.
%   [DataMatrix,SyncMatrix] =
      ExtractSyncedDataSegments(BackscateredData,Sync,Trp,Tp)
      extracts
%   segments of data from the BackscateredData array that are
      synchronized based on the Sync array.
SamplingFrequency = 44100;
SyncMatrix = zeros(1,Trp*SamplingFrequency);
DataMatrix = zeros(1,Trp*SamplingFrequency);
CopySyncVector = zeros(1,Trp*SamplingFrequency);
CopyDataVector = zeros(1,Trp*SamplingFrequency);
flag_wait = 1; flag_start = 0;
    secure_counter = 0;
    for i = 1:length(BackscateredData(1,:))
        if (Sync(i)<0.1) && (Sync(i)>-0.1) && flag_wait == 1
            secure_counter = secure_counter+1;
            if secure_counter == Trp*SamplingFrequency
                flag_wait = 0;
            end
        elseif (Sync(i)>0.8) && flag_wait == 0
            flag_start = 1;
        end
        if flag_start == 1
            flag_start = 0;
        for jj = 1 : Trp*SamplingFrequency % The next
              Tp*SamplingFrequency samples will be collected
            CopySyncVector(1,jj) = Sync(1,jj+i);
            CopyDataVector(1,jj) = BackscateredData(1,jj+i);
        end
        SyncMatrix = [SyncMatrix ; CopySyncVector];
        DataMatrix = [DataMatrix ; CopyDataVector];
        i = i+jj;  % brings i to the correct value
        secure_counter =0;
        flag_wait = 1;
    end
end
SyncMatrix = SyncMatrix(2:end,:); DataMatrix =
      DataMatrix(2:end,:);
ParseSyncMatrix = -ones(size(SyncMatrix));
NumberOfUpchirpMatrix = zeros(length(SyncMatrix(:,1)),1);
for jj = 1 : length(SyncMatrix(:,1))
    for i = 1 : Trp*SamplingFrequency
        if SyncMatrix(jj,i) > 0
            ParseSyncMatrix(jj,i) = 1;
        elseif SyncMatrix(i) < 0
            ParseSyncMatrix(jj,i) = -1;
        end
    end
    NumberOfUpchirpMatrix(jj) =
          UpchirpsCount(SyncMatrix(jj,:));
    end;
end;
```

A.2.5. IntegrateSyncedDataSegments

```
    function IntegratedDataMatrix =
 IntegrateSyncedDataSegments(DataMatrix,ParseSyncMatrix,...
 NumberOfUpchirpMatrix,Tp,Trp,SamplingFrequency)
%INTEGRATESYNCEDDATASEGMENTS Integrates, through all the
      upchirps, each row of a matrix DataMatrix that have synced
      data segments
%   organized through each row.
%   This function takes in a DataMatrix, a ParseSyncMatrix, a
      matrix containing the number of upchirps,
%   and parameters Tp, Trp, and SamplingFrequency. It then
      integrates through all the upchirps in a specific
%   position (i.e., a row) and puts the result of integration into
      IntegratedDataMatrix (thus parsing Data matrix).
DataMatrix =
      zeros(length(NumberOfUpchirpMatrix(:,1)),Tp*SamplingFrequency);
UpchirpDataMatrix = zeros(1,Trp*SamplingFrequency);
SingleUpchirpDataVector = zeros(1,Tp*SamplingFrequency);
AddingUpchirpDataVector = zeros(1,Tp*SamplingFrequency);
for jj = 1 : length(NumberOfUpchirpMatrix(:,1))
    LocalEnsembleMatrix =
          EnsembleMatrixZPfill(ParseSyncMatrix(jj,:), ...
        DataMatrixCopy(jj,:),NumberOfUpchirpMatrix(jj,:),0);
    for kk = 1 : NumberOfUpchirpMatrix(jj,:)
        DataMatrix(jj,:)
              =DataMatrix(jj,:)+LocalEnsembleMatrix(kk,:);
    end
    DataMatrix(jj,:)=DataMatrix(jj,:)/NumberOfUpchirpMatrix(jj);
end
IntegratedDataMatrix = DataMatrix;
end
```

A.2.6. SARCustomHilbert

```
function A_Hilbert_out = SARCustomHilbert(A,hann_option)
%SARCustomHilbert performs the Hilbert transform
% on the input matrix A.
% The function can also applie a Hann window
% if the 'hann_option' is set to 'hann'.
%
% Inputs:
%   A - Input matrix.
%   hann_option - Optional argument. If set to 'hann',
%  Hann window is applied to the result of the Hilbert transform
%   row by row
% Outputs:
%   A_Hilbert_out - The output of the Hilbert transform.
% If 'hann_option' is set to 'hann', this will be the windowed
      result.

    if nargin < 1
        error ('A is a required input')
    end
    if nargin < 2
      hann_option = 'a';
    end
  % Hilbert transform: 1) FFT on each row (position)
    A_FFT = zeros(size(A));
    for jj = 1:length(A(:,1))
        A_FFT(jj,:) = fft(A(jj,:));
    end
    A_FFT_positive_frequency = ...
        A_FFT(:,1:length(A(1,:))/2);
    % 2 IFFT on positive frequency of each row
    A_Hilbert = zeros(size(A_FFT_positive_frequency));
    for jj = 1:length(A(:,1))
        A_Hilbert(jj,:) = ifft(A_FFT_positive_frequency(jj,:));
    end
    if hann_option == 'hann'
        % Hann windowing row by row
        window = hann(size(A_Hilbert,2));
        A_windowed_Hilbert = zeros(size(A_Hilbert));
        for i = 1:size(A, 1)
            A_windowed_Hilbert(i, :) = A_Hilbert(i, :) .* window';
        end
        A_Hilbert_out = A_windowed_Hilbert;
    else
        A_Hilbert_out = A_Hilbert;
    end
end
```

A.2.7. UPCHIRPSCOUNT

```
function NumberUpchirpsOut = UpchirpsCount(Sync)
% Calculates the Number of UpChirps of Sync.
% Sync is expected to be a row vector. The input vector mus swing from -1
% to 1.
NumberUpchirps = 0;
    WaitFor0 = 0;
    for i = 1:length(Sync)
        if Sync(i) < 0
            WaitFor0 = 0;
        end
        if Sync(i) > 0 && WaitFor0 == 0
            NumberUpchirps = NumberUpchirps + 1;
            WaitFor0 = 1;
        end
    end
    % Counts the number of upchirps
    NumberUpchirpsOut = NumberUpchirps;
end
```

A.2.8. GETIMAGEFROMCUSTOMINTERPOLATION

```
    function [FinalTruncatedImage, CrossRange, DownRange] =
          GetImageFromCustomInterpolation( ...
    intpolCrossRangeMatrixFFT, Kye, c_range_1, c_range_2,
          d_range_1, d_range_2)
%GETIMAGEFROMCUSTOMINTERPOLATION Process radar data and generate
      a truncated image
%   This function takes radar data in the form of a 2D FFT (Fast
      Fourier Transform)
%   matrix, performs various transformations and truncations, and
      generates a
%   final truncated radar image.
%
%   Input Arguments:
%   - intpolCrossRangeMatrixFFT: 2D FFT matrix representing the
      radar data.
%   - Kye: Cross-range wavenumber vector.
%   - c_range_1, c_range_2: Cross-range limits for truncation.
%   - d_range_1, d_range_2: Down-range limits for truncation.
%
%   Output Arguments:
%   - FinalTruncatedImage: The final truncated radar image.
%   - CrossRange: Vector representing cross-range positions in the
      image.
%   - DownRange: Vector representing down-range positions in the
      image.
%
%   Detailed Explanation:
%   - The function starts by performing a 2D IFFT (Inverse Fast
      Fourier Transform)
%     on the input radar data matrix, with zero-padding.
%   - Flips the resulting matrix to adjust for orientation.
%   - The truncated image region is defined based on specified
      cross-range and
%     down-range limits.
%   - Cross-range and down-range vectors are generated based on
      the specified limits.
%   - The final truncated radar image is computed by applying
      specific operations
%     on the truncated data, including multiplication by the
      squared absolute
%     values of the down-range positions and logarithmic scaling.
    % constants
    c  = 299792458;           % speed of light
    SamplingFrequency = 44100;
    Tp = 0.02;                % Upchirp time
    fstart =2.408e9;          % starting up-chirp frequency
    fstop =2.495e9;           % stopping up-chirp frequency
    Deltaf = fstop-fstart;
    lambda = c/fstart;
    % Range Data Matrix will be our final matrix
    % Perform ifft2D. Zero Padding is set to 4 by default.
    RangeDataMatrix = ifft2(intpolCrossRangeMatrixFFT, ...
        4*length(intpolCrossRangeMatrixFFT(:,1)), ...
        4*length(intpolCrossRangeMatrixFFT(1,:)) );
    % Rotate -90 RangeDataMatrix and flip left right
    RangeDataMatrix = fliplr(RangeDataMatrix);
    RangeDataMatrix = rot90(rot90(rot90(RangeDataMatrix, 1)));
    % definition of the truncated Image
    dx = lambda/2;                    % displacement of the radar
          on top of the rail (OK)
    dfy = (c/(2*pi))*(Kye(end)-Kye(1)+16);
    Rmax = (c/(2*dfy))*(length(RangeDataMatrix(:,1)));
    Rail_Rmax = length(intpolCrossRangeMatrixFFT(:,1))*dx;

    d_index_1 = round((size(RangeDataMatrix,1)/Rmax)*d_range_1);
    d_index_2 = round((size(RangeDataMatrix,1)/Rmax)*d_range_2);
    c_index_1 =
          round((size(RangeDataMatrix,2)/Rail_Rmax)*(c_range_1+(Rail_Rmax/2)));
    c_index_2 =
          round((size(RangeDataMatrix,2)/Rail_Rmax)*(c_range_2+(Rail_Rmax/2)));

    TruncatedRangeDataMatrix =
          RangeDataMatrix(d_index_1:d_index_2,c_index_1:c_index_2);
    TruncatedRangeDataMatrix = fliplr(TruncatedRangeDataMatrix);

    % cross range vector
    dcr = (c_range_2 - c_range_1)/(c_index_2-c_index_1);
    CrossRange = c_range_1 : dcr : c_range_2;

    % down range vector
    ddr = (d_range_2 - d_range_1)/(d_index_2-d_index_1);
    DownRange = d_range_1 : ddr : d_range_2;

    FinalTruncatedImage =
          zeros(length(DownRange),length(CrossRange));
    for ii = 1 : length(TruncatedRangeDataMatrix(1,:))
        FinalTruncatedImage(:,ii) = TruncatedRangeDataMatrix(:,ii)
              .* (abs(DownRange')).^2;
    end
    FinalTruncatedImage=
    20*log10(abs(FinalTruncatedImage)/max(abs(FinalTruncatedImage),[],'all'));
end
```

A.3. Extra Task Codes

A.3.1. Real Time Velocity Measurement

In this code we are simulating a real time USB serial communication with an audio file and the pause() function. To make it actually real time one should replace these with the function portread().

```
    clear all;
close all;
clc;
% Constants
fc = 2.43e9;
Tp = 0.1;                       % Window
N = 4;                          % Zero Padding
c  = 299792458;                 % speed of light
SamplingFrequency=44100;        % Referred to the audio signal

% time axis
dt = 1/SamplingFrequency;
SpectrogramTimeWindow= 5;
t = 0 : dt : SpectrogramTimeWindow-dt;
SamplesInWindow = Tp*SamplingFrequency;

%frequency and velocity
BandWidth = SamplingFrequency;
bin = SamplingFrequency/(SamplesInWindow*(1+N));
f = 0 : bin : BandWidth-bin;
v = f * (c/(2*fc));
NumberOfTimeWindows = floor(SpectrogramTimeWindow/(Tp));
Spectrogram_out = (-50)*ones(NumberOfTimeWindows,
      SamplesInWindow*(1+N));
TemporaryZeroVector = zeros(1, SamplesInWindow*N);
k=0;
NameOfAudio = 'Velocity_Test_File.m4a';
```

```
while 1
  % Signal extraction and zero padding
    VelocityTest = audioread(NameOfAudio, [(Tp*SamplingFrequency*k
          +1) ...
       (Tp*SamplingFrequency*(k+1))]);
    VelocityTest = -(VelocityTest)';
    VelocityData = VelocityTest(1,:);
    VelocityDataPadded = [ VelocityData TemporaryZeroVector ];

  % performs the FFT data by data and directly normalizes it
    NewSpectrogrmRow =
          20*log10(abs(fft(VelocityDataPadded))/max(abs(fft(VelocityDataPadded))));
  % puts on top of spectrogram the new data
    Spectrogram_out = PutOnTop(Spectrogram_out,NewSpectrogrmRow);
          %help PutOnTop
    Spectrogram_out = CutLowValue(Spectrogram_out,-20);
    figure(1)
    imagesc(v,t,Spectrogram_out); clim([-50 -10]);
    axis([ v(1) 30 0 SpectrogramTimeWindow])
    xlabel("Velocity [m/s]"); ylabel("time [s]");
    title("Velocity Spectrogram");
    figure(2)
    plot(v,NewSpectrogrmRow);axis([ 0 30 -50 0]);
    xlabel("Velocity [m/s]"); ylabel("dB");
    pause(Tp)
    k= k + 1;
end
```

A.3.2. Real Time Range Measurement

In this code we are simulating a real time USB serial communication with an audio file and the pause() function. To make it actually real time one should replace these with the function portread().

```
clear all;
close all;
clc;
% constants
c  = 299792458;       % speed of light
fstart =2.408e9;fstop =2.495e9;Deltaf = fstop-fstart;
% Time axis
SamplingFrequency = 44100;
Tp = 0.02;                              % Upchirp time
dt = 2*Tp;
N = 4;
SpectrogramTimeWindow= 10;              % spectrogram seconds
t = (0:SpectrogramTimeWindow/(2*Tp))*dt;
% Spectrogram initialization
TotalSpectrogram = zeros(SpectrogramTimeWindow/(2*Tp),
      (N+1)*Tp*SamplingFrequency);
% range axis
Rmax = c*Tp/2;
dR = c/(2*Deltaf*(N+1));
RefreshTime=0.36;                                % c RefreshTime of
      the Radar
TotalRows = SpectrogramTimeWindow/(2*Tp);      % total rows of the
      ensemble matrix
R = 0:dR:dR*length(TotalSpectrogram(1,:))-dR;
k=1;
while 1
    RangeTest = audioread('Task3_2_b.m4a',
          [(RefreshTime*SamplingFrequency*k +1) ...
       (RefreshTime*SamplingFrequency*(k+1))]);
    RangeTest = -(RangeTest)';
    BackscateredData = RangeTest(1,:);
    Sync = RangeTest(2,:);
     % Parsing Sync
```

```
    ParseSync = -ones(1, length(Sync(1,:)));
  for i = 1 : length(Sync(1,:))
    if Sync(i) > 0
        ParseSync(i) = 1;
    elseif Sync(i) < 0
        ParseSync(i) = -1;
    end
  end
  NumberUpchirps = UpchirpsCount(ParseSync);
  LocalEnsambleMatrix =
        EnsembleMatrixZPfill(ParseSync,BackscateredData,NumberUpchirps,N);
  for i = 1 : length(LocalEnsambleMatrix(1,:))
    LocalEnsambleMatrix(:,i)=LocalEnsambleMatrix(:,i)- ...
    mean(LocalEnsambleMatrix(:,i));
  end
  LocalSpectrogram = zeros(size(LocalEnsambleMatrix));
  for i = 1 : NumberUpchirps
    LocalSpectrogram(i,:) = abs( ifft(
          LocalEnsambleMatrix(i,:) ) )/max( abs( ifft(
          LocalEnsambleMatrix(i,:) ) ) );
  end
  LocalSpectrogram = 20*log10(LocalSpectrogram);
  LocalSpectrogram = LocalSpectrogram
        -max(LocalSpectrogram,[],"all");
  TotalSpectrogram = PutOnTop(TotalSpectrogram,LocalSpectrogram);
  TotalSpectrogram = CutLowValue(TotalSpectrogram,-20);
  imagesc(R,t,TotalSpectrogram(:,1:length(R))); clim([-50 0]);
        axis([0 100 0 t(length(t))]);
  xlabel("Range [m]"); ylabel("time [s]");
  title("Range Spectrogram, fstart="+fstart*1e-9+" GHz,
        fstop="+fstop*1e-9+"GHz");
  pause(RefreshTime/5);k=k+1;
end
```