



DELIVERABLE D4.2

MMWAVE 3D POSITIONING PROTOTYPE

DECEMBER 15, 2023

MARIOS RASPOPOULOS, STELIOS IOANNOU, ANDREY SESYUK
INTERDISCIPLINARY SCIENCE PROMOTION & INNOVATIVE RESEARCH EXPLORATION (INSPIRE)



Co-funded by
the European Union



RESEARCH
& INNOVATION
FOUNDATION

"This work was co-funded by the European Union under the programme of social cohesion "THALIA 2021-2027", through Research and Innovation Foundation (Project: CONCEPT/0722/0031)".

Abstract

This report describes the technical characteristics, the setup, the implementation and validation of the algorithms and hardware developed in T4.2. The aim of this task was developing the positioning algorithms based on geometric approaches reported in literature in order to use in conjunction with mmWave hardware to setup a 3D mmWave positioning prototype that will be tested and evaluated in T3.2. Following the evaluation of the various commercial mmWave sensors and the precision analysis conducted and reported in Deliverable D4.1 a positioning prototype utilizing the most promising sensors was implemented in a laboratory environment to test various geometric positioning approaches. The performance analysis of these algorithms is reported in Deliverable D3.2.

The system is a combination of mmWave Sensors from Texas Instruments (IWR1642 and IWR1843) each of which is connected to a Raspberry PI4 and all of these are wirelessly connected through a python client-server application to a Windows-based machine that runs an MS SQL server. Additionally, an IMU is deployed on a drone (DJI 2S) and connected to a Raspberry PI3 which also reports its data to the central server again through a client server application. The central Server runs a python script that reports the real-time data to the SQL Server. The SQL-stored location-specific data is then accessed through another Microsoft-based machine which runs the various positioning algorithms on MATLAB.

Table of Contents

Abstract	1
Table of Contents.....	2
1 Positioning System Architecture	3
2 System Components	4
2.1 mmWave Sensor	4
2.2 The IMU sensor	8
2.3 Central Server	10
2.4 SQL Database	12
2.5 MATLAB Implementation.....	14
3 Pictures of the Prototype.....	14
4 Appendix	22
4.1 Code 1 - Run Code for mmWave Sensors on Raspberry Pi	22
4.2 Code 2 - Parsing Code for mmWave Sensors on Raspberry Pi	23
4.3 Code 3 - Example of mmWave Configuration File	26
4.4 Code 4 - Central PC Python Script.....	27
4.5 Code 5 - SQL Server Script	30
4.6 MATLAB Code for accessing the database	35
5 References	37

1 Positioning System Architecture

The system begins with the IWR1642BOOST mmWave sensors, each configured by a Python script running on a Raspberry Pi 4. These sensors actively detect targets in their respective fields of view, capturing essential data related to distance, angle, and velocity. The Raspberry Pi 4 acts as a local processing hub, collecting this information from the sensors and sending it through a serial port to another Raspberry Pi 4, serving as a personal data collector. On the personal Raspberry Pi 4, a Python script parses the incoming data, extracting relevant information about the detected targets. Once processed, this data is transmitted to a central PC over a TCP connection, facilitating real-time communication between the distributed components of the system. Simultaneously, an IMU on a drone, connected to a Raspberry Pi 3, gathers orientation data about the drone's movement and transmits this data to the central PC, contributing to the comprehensive dataset. The central PC serves as a focal point for data collection. It receives and consolidates the information from multiple Raspberry Pi 4s and the Raspberry Pi 3 connected to the IMU. A Python script on the central PC then orchestrates the transfer of this amalgamated data to an MS SQL server, which acts as a robust and scalable storage solution for the extensive dataset generated by the sensors and IMU.

The stored data in the MS SQL server becomes a valuable resource for subsequent analysis and application. A MATLAB script is employed to retrieve data from the SQL server, enabling the process and analysis of the information for various systems and algorithms aimed at achieving 3D positioning. This modular and distributed architecture provides a flexible and scalable framework, capable of accommodating advancements or modifications in hardware and algorithms to further enhance the system's capabilities for accurate 3D positioning.

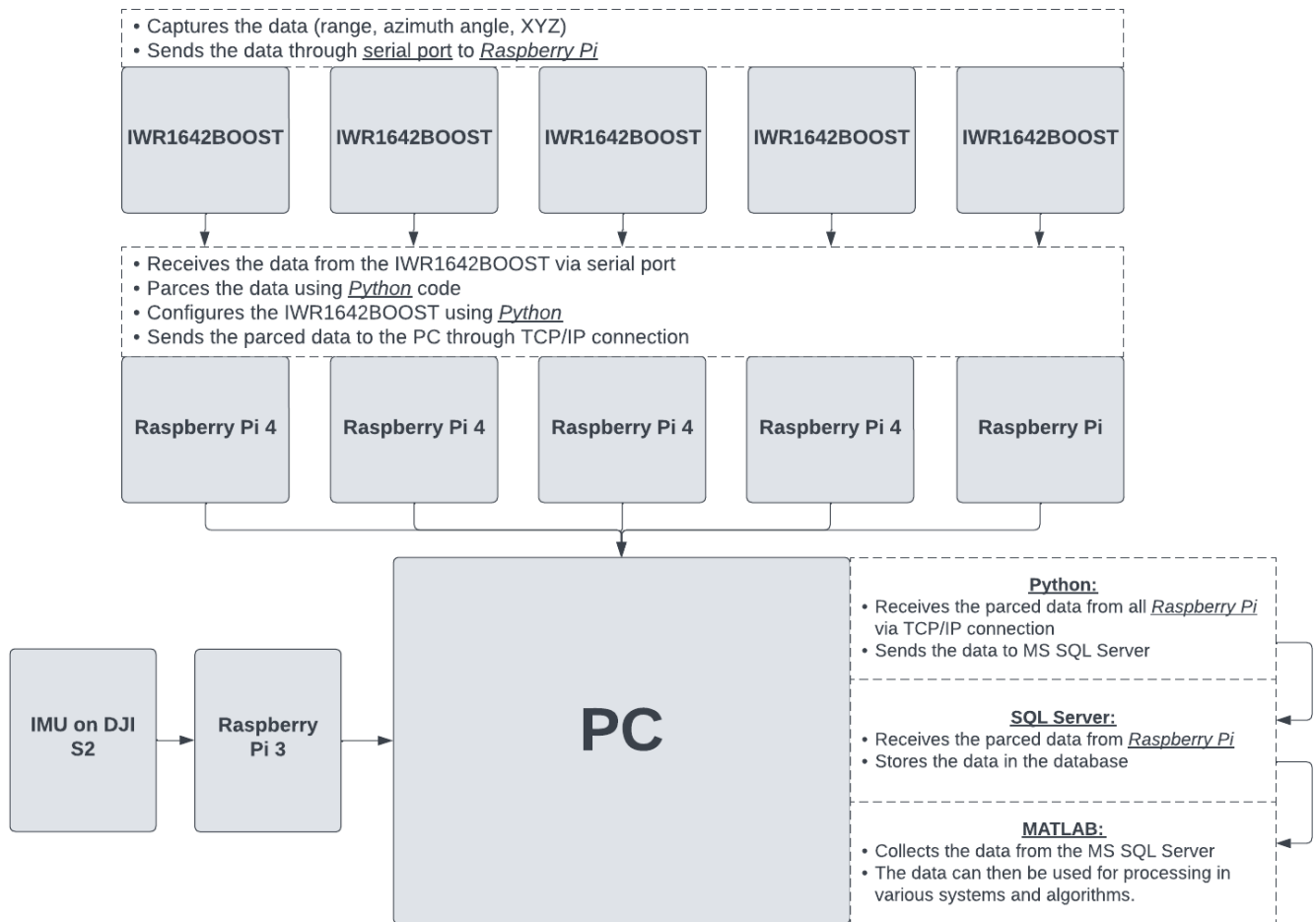


Figure 1: Thesis System Setup

2 System Components

2.1 mmWave Sensor

The IWR1642BOOST mmWave sensor is an advanced radar sensor developed by Texas Instruments, designed for applications in industrial settings, robotics, automotive, and more. The term "mmWave" refers to millimeter-wave technology, which operates in the frequency range of 30 to 300 GHz, enabling high-resolution and accurate sensing capabilities. The IWR1642BOOST is built on Texas Instruments' highly integrated radar technology platform, making it a powerful tool for various sensing applications.

One notable feature of the IWR1642BOOST is its integration of digital signal processing (DSP) cores on the chip, allowing for real-time signal processing directly on the sensor. This feature can reduce the data bandwidth requirements and simplify the system design. Additionally, the module supports various interfaces for easy connectivity with microcontrollers, FPGAs, and other processing units. Developers and engineers can leverage the accompanying software development kit (SDK) provided by Texas Instruments to customize and optimize the sensor's performance for

specific applications. The IWR1642BOOST thus stands as an integral component in the rapidly evolving landscape of sensor technologies, playing a crucial role in enabling the next generation of intelligent and autonomous systems. The IWR1642BOOST sensor is equipped with 4 receiving (Rx) and 2 transmitting (Tx) antennas operating at frequencies between 76-81 GHz allowing for a 120-degree field of view and ranging capabilities of up to 72 meters. Similarly, the IWR1843BOOST possesses a Frequency Modulated Continuous Wave (FMCW) transceiver which allows it to provide ranging, angling, and velocity information about the target. However, due to an additional TX antenna, in addition to the azimuth angling information (X-Y), it is also able to provide the elevation data of the target (Z). In contrast, the Infineon Distance2Go mmWave sensor is equipped with 1 Rx and 1 Tx antenna and operates between 24-26 GHz with a field of view of 20 degrees and a maximum detection range of around 20 meters. While the TI sensor performs range and angle measurements, the Infineon one can only measure range.

Both sensors collect the data about the detected target and then send it to a personal Raspberry Pi 4 through a serial connection. This data is transferred in a form of a binary string which is then read and parsed using a Python script. When parsed, the string is presented as seen below and contains the number of the Raspberry which serves as the identification of the source Raspberry, discerning its unique number within the network of multiple sensors. This serves as a crucial reference point for coordinating and attributing the specific sensor that the data was received from. Following, the timestamp offers a precise record of the moment the data was transmitted. This temporal information is paramount for synchronization across various sensors, ensuring a unified timeline for the operation of the system and comprehensive analysis.

The rest of the string consists of all the data captured by the IWR mmWave sensor within a certain frame. This includes the number of objects detected within the sensor's frame, providing an initial glimpse into the density of the observed environment. Each detected object is labelled and includes information such as the XYZ coordinates (in meters), the object's captured velocity (in meters per second), the range to the object (in meters), azimuth and elevation angles (in degrees), as well as signal metrics that contribute to the comprehensiveness of the data, such as the Signal-to-Noise Ratio (SNR) measured in 0.1dB increments, offering insights into the quality and reliability of the acquired signals and the noise level, also in 0.1dB increments, which provides information about the ambient interference affecting the sensor readings.

Data from IWR1642/1843												
Raspberry Number	Time Stamp	Number of Objects	Object Number	X (m)	Y (m)	Z (m)	V (m/s)	Range (m)	Azim (deg)	Evel (deg)	SNR (0.1dB)	Noise (0.1dB)

All sensors are also launched and configured using the same Python script. This configuration essentially determines the combination of TX/RX antennas to ensure the most optimal performance of the sensor depending on its application,

as well as the operational frequency, the duration of the frame, the maximum unambiguous range and the Angle of Arrival Field of View (AoA FoV). These desirable configurations include combinations of sensors which can be optimised for best range resolution, best velocity resolution and best range. Since the positioning system requires detection of the target over an entire room, the sensor was set up to enable all the TX/RX antennas for a full field of view of angle of arrival, range and doppler. Specifically (4RX, 2TX) for the IWR1642BOOST and (4RX,3TX) for the IWR1843BOOST.

The Raspberry Pi 4 presents a compelling choice over its predecessor, the Raspberry Pi 3, particularly when employed in conjunction with mmWave sensors. Several factors contribute to the preference for the Raspberry Pi 4 in such applications. First and foremost, the Raspberry Pi 4 boasts significant improvements in processing power. With a quad-core ARM Cortex-A72 CPU clocked at higher frequencies, the Pi 4 offers enhanced computational capabilities compared to the Pi 3. This increased processing power proves beneficial when handling the data-intensive tasks often associated with mmWave sensors, such as processing high-resolution radar data or implementing complex algorithms for signal processing and object detection. The additional RAM allows for smoother multitasking and improved overall system performance, contributing to a more responsive and capable computing environment. The connectivity options on the Raspberry Pi 4 are also superior, featuring USB 3.0 ports that facilitate faster data transfer rates compared to the USB 2.0 ports on the Raspberry Pi 3. This is particularly relevant when interfacing with mmWave sensors that may produce substantial amounts of data. The higher data transfer speeds can prevent bottlenecks in the data pipeline and ensure efficient communication between the Raspberry Pi and the mmWave sensor.

Having available, the range (r), the azimuth (θ) and the elevation (φ) measurements from the anchor to the target one can estimate the coordinates of the target with respect to the body-frame coordinate system of the anchor using standard spherical to Cartesian coordinate conversion according to Figure 2 and Equation below:

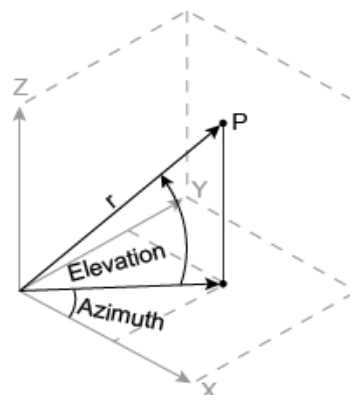
$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = r \begin{bmatrix} \cos(\vartheta)\cos(\varphi) \\ \cos(\vartheta)\sin(\varphi) \\ \sin(\vartheta) \end{bmatrix}$$


Figure 2 - Spherical to Cartesian Conversion

To properly determine the coordinates of the target within the room's coordinate plane, it was imperative to align the coordinate system of the sensor (body frame coordinate system) to that of the room (Local Coordinate System). Achieving this alignment involves a series of calculations that account for the sensor's yaw, pitch, and roll. These adjustments were critical in ensuring that the sensor's data correspond accurately to the room's coordinate plane, allowing for reliable 3D positioning. Assuming that the anchor is first rotated by an angle ψ around the z-axis (yaw), then by an angle θ around y-axis (pitch) and finally by an angle ϕ around the x-axis (roll) the 3x3 rotation matrix is given by:

$$R = R_z \cdot R_y \cdot R_x$$

where,

$$R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \sin(\phi) & \cos(\phi) \end{bmatrix}$$

$$R_y = \begin{bmatrix} \cos(\phi) & 0 & \sin(\phi) \\ 0 & 1 & 0 \\ -\sin(\phi) & 0 & \cos(\phi) \end{bmatrix}$$

$$R_z = \begin{bmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

With reference to Figure and considering that body-frame measurement from a sensor positioned at $A = [x_a y_a z_a]$ is $P' = [x' y' z']$ then the local coordinates $P = [xyz]$ of the target can be calculated using:

$$P = [R \cdot P'^T]^T + A$$

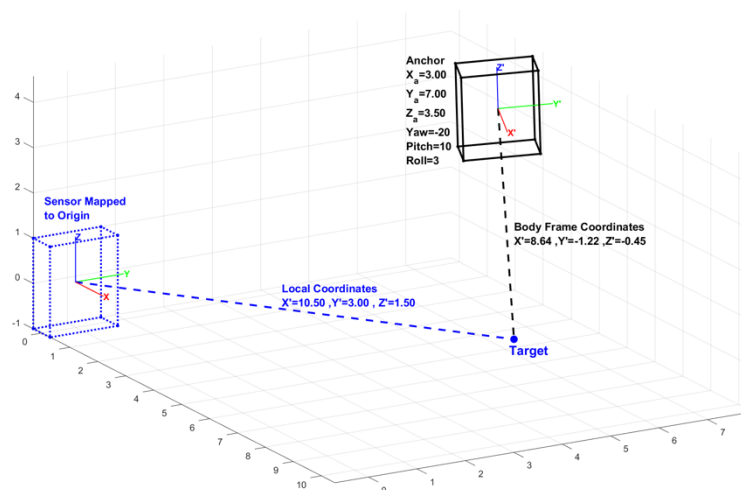


Figure 3 - Body Frame to Local Coordinates Conversion

2.2 The IMU sensor

An Inertial Measurement Unit (IMU) is a sensor that measures the orientation and motion of an object in three-dimensional space. To understand the orientation of an IMU, it's essential to be familiar with coordinate systems and concepts like the Body-frame, NED (North-East-Down), and ENU (East-North-Up) reference frames. The Body-frame, also known as the Body-relative frame, is a coordinate system that is attached to and moves with the object to which the IMU is attached. In this frame, the object's orientation and motion are typically measured. The x, y, and z axes of the Body-frame are usually aligned with the object's forward, right, and upward directions, respectively. The Body-frame is essential for understanding how the object is oriented in space, as it provides information about roll, pitch, and yaw angles. The NED coordinate system is an Earth-relative frame of reference used in navigation and geodesy. In this system, the origin (0, 0, 0) is typically located at a fixed point on the Earth's surface. The positive x-axis points to the geographic North, the positive y-axis points to the East, and the positive z-axis points downward, toward the centre of the Earth. NED coordinates are used for navigation, especially in aviation and marine applications. The ENU coordinate system is another Earth-relative frame of reference that is similar to NED but with a different axis orientation. In ENU, the positive x-axis points to the geographic East, the positive y-axis points to the North, and the positive z-axis points upward, away from the Earth's surface. ENU coordinates are often used in geodetic applications and for localization in robotics. The orientation of an IMU is typically expressed in terms of the rotation between the Body-frame and a global Earth-relative frame (NED or ENU). Commonly, orientation is represented using Euler angles or rotation matrices, which describe the object's roll, pitch, and yaw angles in relation to the global frame. These angles provide a way to understand how the IMU is oriented in space and how it is moving.

The Raspberry Pi 3, despite being an older model, remains a reliable and cost-effective choice for applications involving Inertial Measurement Units (IMUs). The Raspberry Pi 3's capabilities are generally sufficient for processing data from IMUs, making it a practical option for many projects. While not as powerful as the Raspberry Pi 4's processor, it is still more than capable of handling the data generated by IMUs. The absence of USB 3.0 on the Raspberry Pi 3 may be a limitation in terms of data transfer speed compared to the Raspberry Pi 4, but for many IMU applications, the lower data rates are acceptable. Another consideration is power consumption. The Raspberry Pi 3 generally consumes less power than the Raspberry Pi 4, making it suitable for projects where energy efficiency is a priority. This can be especially important in applications where the device is battery-powered or needs to operate in resource-constrained environments.

Integrating Inertial Measurement Units (IMUs) with Raspberry Pi or Arduino platforms can present challenges, primarily associated with the availability and compatibility of drivers. Drivers are essential pieces of software that enable the communication between the hardware (IMUs) and the operating system or microcontroller. The limitations often stem from differences in the types of IMUs and the support provided by the platforms. One major limitation is the diversity of IMUs available in the market, each with its unique specifications and communication protocols. Not all IMUs have standardized drivers readily available for popular development boards like Raspberry Pi or Arduino. This variability can result in the need for custom driver development, which requires a good understanding of the IMU's specifications and the intricacies of interfacing with the specific platform. Moreover, the availability of drivers for different operating systems or microcontroller platforms can vary significantly. While some IMUs come with pre-existing drivers that are compatible with popular systems, others may lack comprehensive support. This can lead to situations where users need to rely on community-developed drivers or invest time in creating their own, introducing complexity and potential reliability issues.

The MPU6050, being a popular 6-axis IMU, is well-supported within the Arduino ecosystem. There are well-documented libraries and drivers available that facilitate easy integration with Arduino boards. The community-driven nature of the Arduino platform has contributed to the creation of comprehensive resources, making it relatively straightforward for users to access and implement drivers for the MPU6050. The MPU9250, which includes a 9-axis sensor with additional magnetometer functionality, also benefits from good driver support within the Arduino environment. Libraries such as the "MPU9250" or "I2Cdevlib" provide ready-to-use drivers, simplifying the process of interfacing with Arduino boards. The ICM20948, another 9-axis IMU known for its high-level integration and advanced features, might have varying levels of support depending on the specific platform. In the case of Arduino, there may be libraries available that support the ICM20948, although the extent of functionality and community support might not be as extensive as with more widely used IMUs like the MPU series. For Raspberry Pi, the driver support landscape may differ. Raspberry Pi has a broader range of applications, and the availability of drivers might not be as standardized as in the Arduino ecosystem. However,

efforts from the community and third-party developers often result in the creation of Python libraries or wrappers that enable the use of various IMUs, including the MPU6050, MPU9250, and ICM20948, with Raspberry Pi.

2.3 Central Server

A client-server TCP (Transmission Control Protocol) connection between two computers in Python involves one computer (the client) making a connection to another computer (the server) to exchange data. Python provides several libraries for implementing this kind of communication, but the 'socket' library is one of the most common choices. One of the computers runs a Python program that acts as the server. The server listens on a specific port for incoming connections. It binds to an IP address and port number, making it reachable for clients. The other computer runs a Python program that acts as the client. The client program connects to the server's IP address and port number using a separate socket. Once the client successfully connects to the server, a bidirectional data channel is established. Both the client and server can send and receive data to and from each other using their respective socket objects. Data sent over the network needs to be encoded before sending and decoded upon reception. This is done to ensure that the data is in a suitable format for transmission.

The TCP client-server architecture serves as a pivotal framework for orchestrating the simultaneous activation of programs on multiple Raspberry Pi devices, specifically aimed at initiating the operation of mmWave sensors. This intricate system not only streamlines the deployment process but also enables the seamless adjustment of each mmWave sensor to a new coordinate system, all orchestrated remotely from a central PC. At the core of this operation is the client-server model, which facilitates communication between the central PC and the network of Raspberry Pi devices which are connected to an individual mmWave sensor. The client, residing on the central PC, initiates the communication, while the server component is deployed on each Raspberry Pi device, awaiting instructions. This architecture enables the efficient and synchronized activation of programs on all connected devices, ensuring a cohesive and unified start-up process.

One of the primary advantages of employing TCP in this context is to transmit information regarding the position and orientation of each mmWave anchor. This data, sent from the central PC to each Raspberry Pi device, allows the mmWave sensors to dynamically adjust to a new coordinate system. This remote coordination is not only efficient but also mitigates the need for manual adjustments on each individual device. The potential for human error is minimized, and the risk of overlooking a device during setup is significantly reduced. The significance of this TCP-based approach becomes particularly evident when considering the potential scale of the sensor network. Rather than starting each Raspberry Pi device individually and manually configuring its parameters, this centralized system ensures a rapid and simultaneous deployment. Moreover, in scenarios where the setup undergoes changes, such as the addition or

relocation of mmWave anchors, the ability to remotely update the configuration avoids the laborious task of adjusting each device manually. This not only saves time and effort but also enhances the scalability and maintainability of the overall system.

To ensure the stability and reliability of the TCP client-server connections with the central PC, each Raspberry Pi was assigned a static IP. This approach simplifies the network management process and mitigates potential connection issues that may arise in dynamic IP environments. Assigning static IP addresses to the Raspberry Pi devices provides a fixed and predictable point of contact for the central PC, allowing for consistent and reliable communication between the devices. Assigning a static IP address to a Raspberry Pi is a relatively straightforward task, often accomplished by configuring the device's network settings. This involves accessing the Raspberry Pi's configuration files and manually specifying the desired IP address. The decision to opt for static IP addresses, as opposed to dynamic ones assigned by a DHCP server, is driven by the need for consistency in the network topology.

In the context of the TCP client-server connection, each Raspberry Pi is intentionally assigned a static IP address above .150. This deliberate choice serves a dual purpose. Firstly, it helps to avoid potential conflicts with other devices on the network that might be assigned dynamic IP addresses within the common range (usually below .110). This deliberate offset ensures that the Raspberry Pi devices have a dedicated and reserved portion of the IP address space, reducing the likelihood of address collisions. Secondly, by assigning addresses above .150, it establishes a clear separation between the static IP addresses allocated to the Raspberry Pi devices and any potential dynamic IP addresses assigned by a DHCP server. This demarcation minimizes the risk of overlap and interference, contributing to the overall network stability.

Another addition to enhance the reliability of our system, was ensuring that confirmation messages are systematically sent from both the mmWave devices and IMUs, which are intricately connected to the Raspberry Pis. These confirmation messages serve as a feedback mechanism, allowing us to verify the operational status of each Raspberry Pi and the associated peripherals, making sure that the mmWave sensors and IMU are running and collecting measurements. By receiving confirmation messages at the connection stage, we can promptly identify any anomalies or issues with a specific Raspberry Pi, mmWave device, or IMU. The incorporation of confirmation messages not only acts as an early warning system but also streamlines the troubleshooting process. In the event of a discrepancy or malfunction, the confirmation messages pinpoint the source of the problem, facilitating an efficient and targeted resolution.

2.4 SQL Database

All the real time data collected from the mmWave sensors and the IMU is stored in a custom-developed SQL Database to facilitate easy integration with any programming language and accessibility from anywhere in the network. More specifically, in our case the real-time context needs to be made available to MATLAB where all the experimentation and testing takes place. The physical and logical connections to the SQL database are shown in Figure 4. All the mmWave sensors as well as the drone on-board IMU are connected to a Raspberry which establishes a wireless connection to the computer through a wireless router that creates a local area network. A separate laptop that runs MATLAB establishes another wireless connection to retrieve the real time data reported by all the sensors.

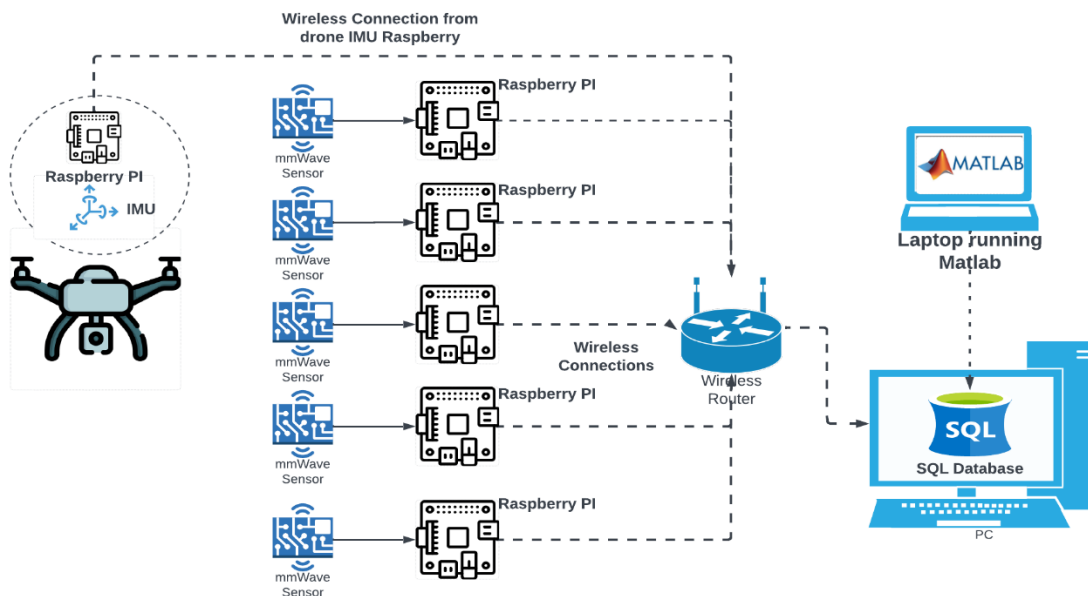


Figure 4: Connections to the SQL Database

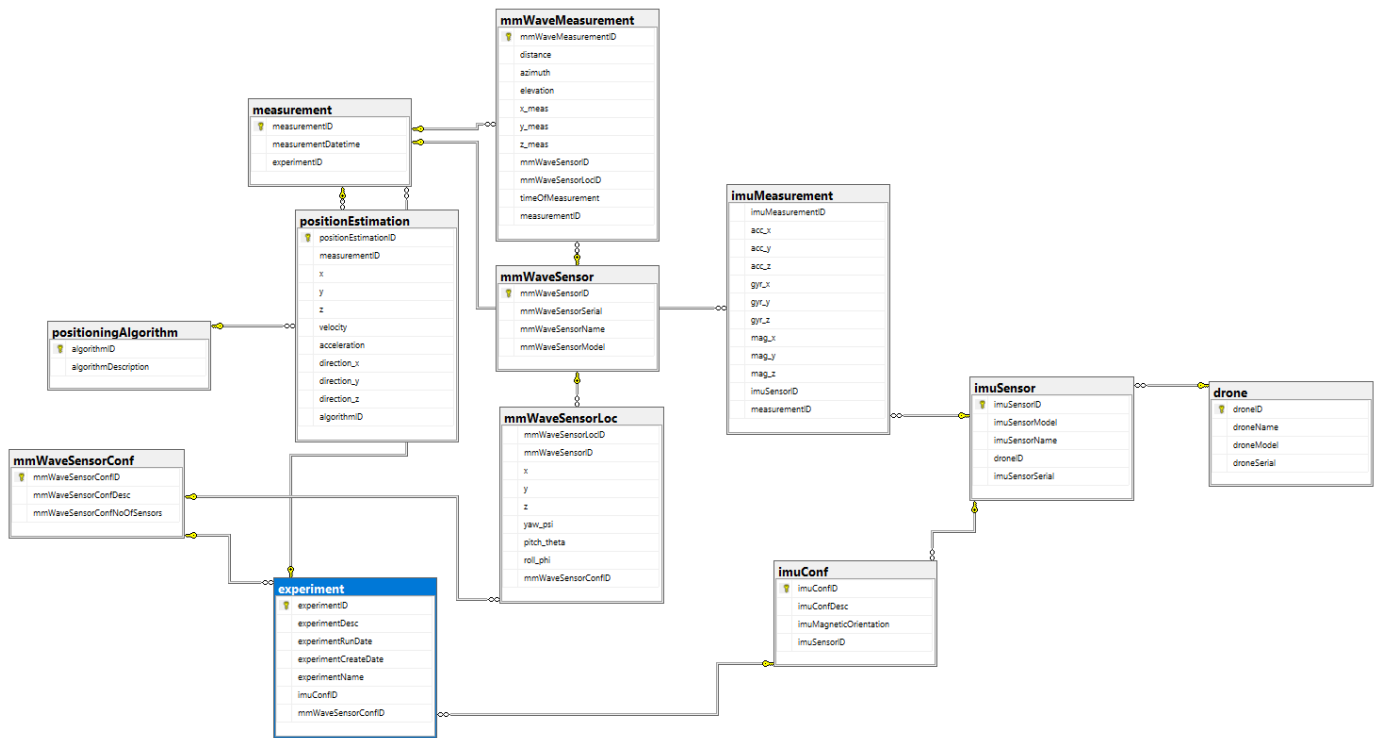


Figure 5: SQL Database for storing real time data.

The schema of the SQL Database is shown in Figure 5. The **experiment** table is considered as the root of this schema, it holds the information about the experiment including the date of creation, the run day, the name of the experiment, a small description and has a foreign the configuration of the mmWave sensors and the imu. The **mmWaveSensor** Configuration provides information about how the mmWave sensor network is laid out physically in terms of the location (x,y,z) and orientation (yaw, pitch, roll) of the each sensor (through the **mmWaveSensorLoc** table) as well as a link to the actual **mmWaveSensor** table that contains information about the hardware type of the sensor (model, serial, name). The **imuConf** table contains a description about the IMU setup including the initial magnetic orientation, its hardware specification (model, name, serial through the **imuSensor** table) and the drone that it is attached to (through the **drone** table). All this description constitutes the setup of the experiment.

During measurement collection, and at every time instance a timestamped entry is generated in the **measurement** table which provides link to the **mmWaveMeasurement** and the **imuMeasurement** table that contain the actual data collected by each sensor during that time instance.

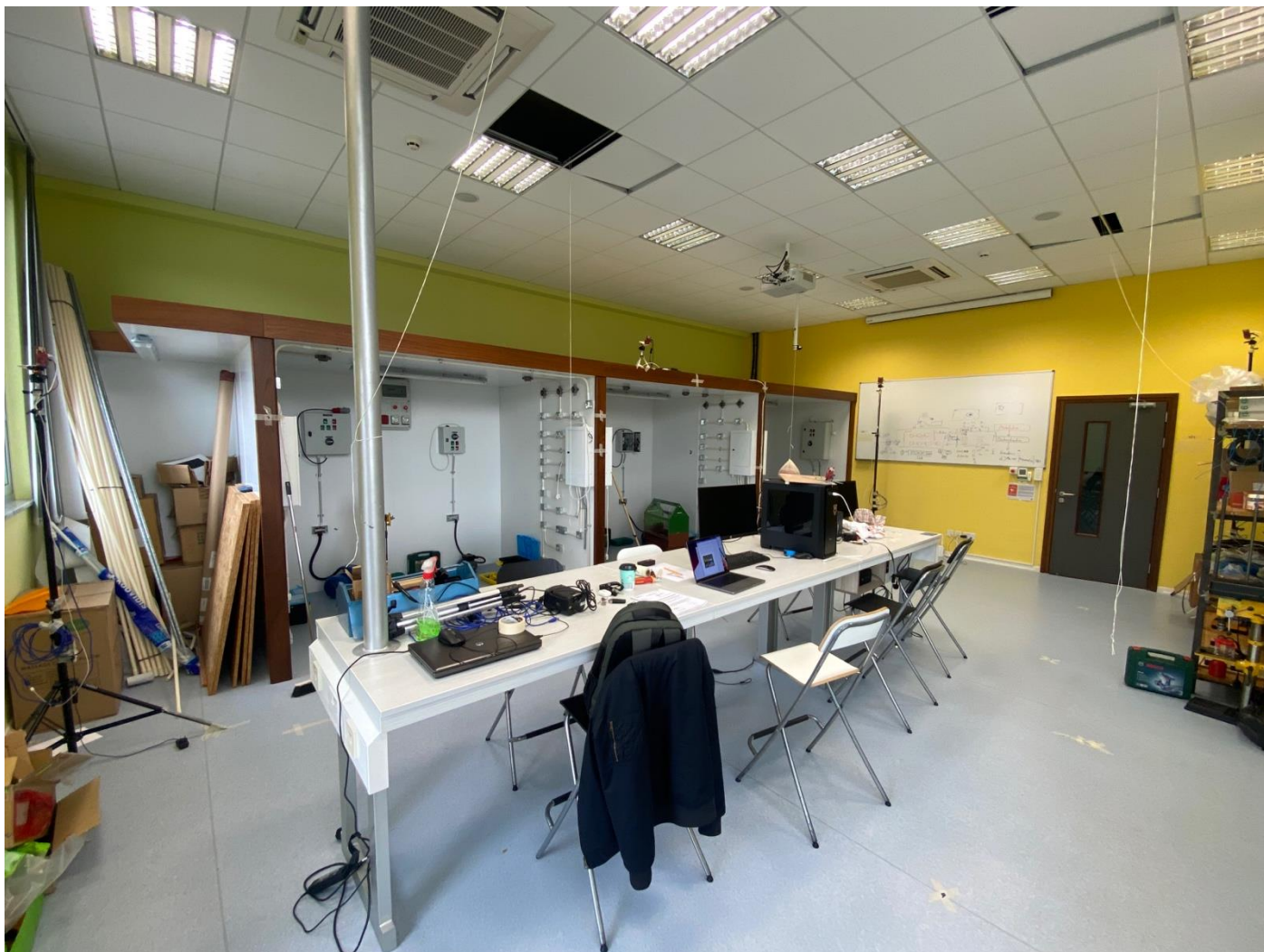
All the measurement data is accessible through a laptop computer that runs matlab on which all the algorithmic implementation and testing takes place. Position estimation is done using different algorithms, the information of which is also stored in the database in the **positionAlgorithm** table. The results, upon calculation are returned **positionEstimation** table which are linked in time with the real time measurements.

2.5 MATLAB Implementation

The MATLAB code that establishes the connection to the SQL database to retrieve the real-time data as well as carry out the positioning experimentation using various positioning algorithms is shown in Appendix 4.6.

3 Pictures of the Prototype





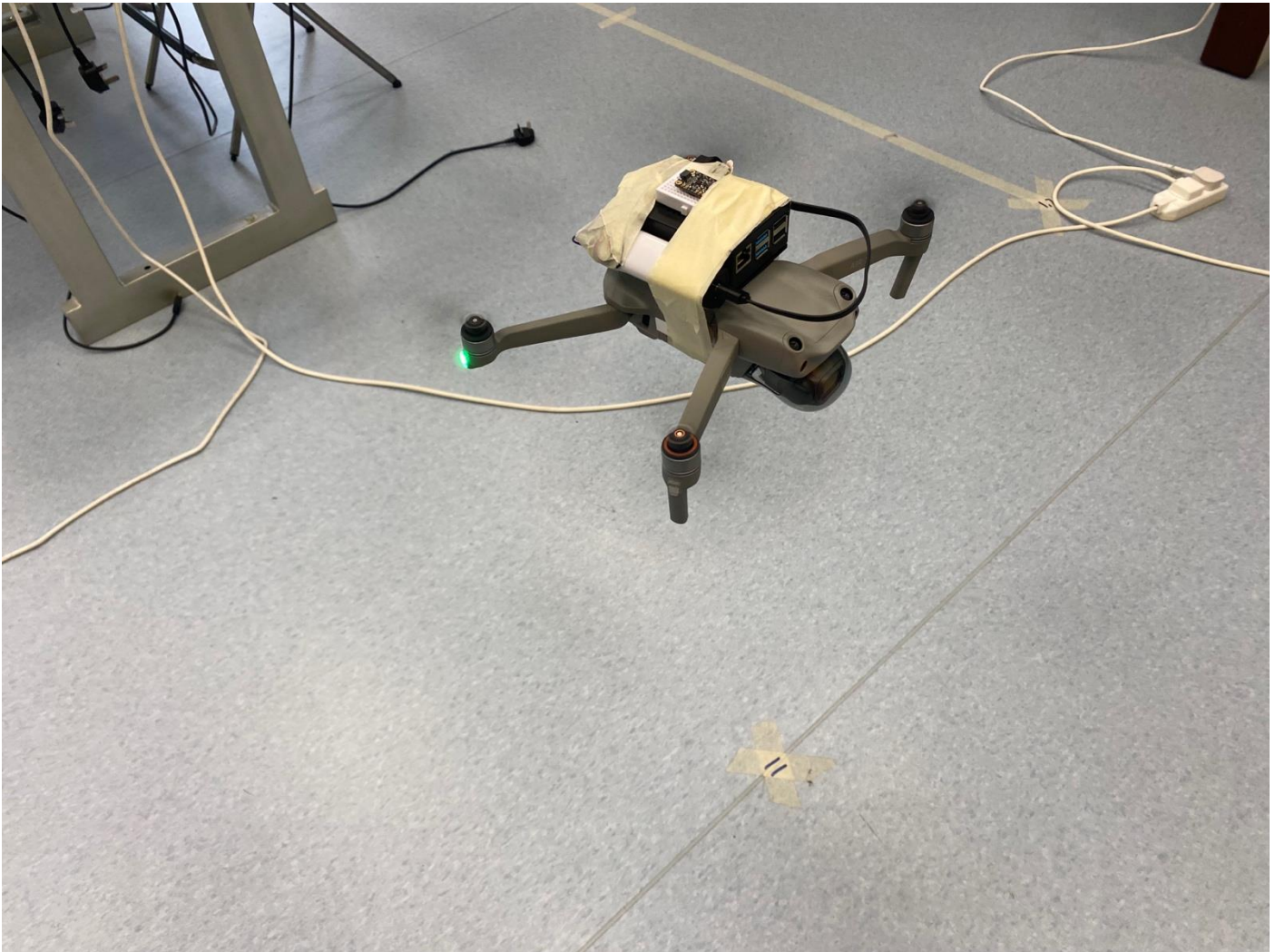












Include pictures of the various components and the complete setup

4 Appendix

4.1 Code 1 - Run Code for mmWave Sensors on Raspberry Pi

```
import os
import sys
import serial
import time
import numpy as np
import socket
from datetime import datetime
from parser_XY_test import parser_one_mmw_demo_output_packet

import socket
import numpy as np

HOST = "192.168.30.152" # Standard loopback interface address (localhost)
PORT = 65432 # Port to listen on (non-privileged ports are > 1023)

data = []

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    s.bind((HOST, PORT))
    s.listen()
    conn, addr = s.accept()
    with conn:
        print(f"Connected by {addr}")
        while True:
            data = conn.recv(1024)
            data = data.decode('utf-8')
            split_data = data.split(',')

            message=HOST+" received the data"
            tt = message.encode('utf-8')
            conn.sendall(tt)
            conn.close()
            break

IDtemp=split_data[0]
ID=int(IDtemp[1:])
X=float(split_data[1])
Y=float(split_data[2])
Z=float(split_data[3])
sensor_delay=float(split_data[4])
yaw_psi=float(split_data[5])
pitch_theta=float(split_data[6])
roll_phi_temp=split_data[7]
roll_phi=float(roll_phi_temp[:-1])
print(ID, X, Y, Z, sensor_delay, yaw_psi, pitch_theta, roll_phi)

# -----

serverAddress = ('192.168.30.150',2222)
bufferSize = 102400
UDPClient = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# Change the configuration file name
configFileName = 'xwr18xx_profile_2023_07_26T08_46_17_507.cfg'
CLIport = {}
Dataport = {}
byteBuffer = np.zeros(2**15,dtype = 'uint8')
byteBufferLength = 0;

# -----

CLIport = serial.Serial('/dev/ttyACM0', 115200)
Dataport = serial.Serial('/dev/ttyACM1', 921600)

# Read the configuration file and send it to the board
config = [line.rstrip('\r\n') for line in open(configFileName)]
for i in config:
    CLIport.write((i+'\n').encode())
    print(i)
    time.sleep(0.01)

# -----

# Function to parse the data insi de the configuration file
configParameters = {} # Initialize an empty dictionary to store the configuration parameters

# Read the configuration file and send it to the board
config = [line.rstrip('\r\n') for line in open(configFileName)]
for i in config:
    # Split the line
    splitWords = i.split(" ")

    # Hard code the number of antennas, change if other configuration is used
    numRxAnt = 4
    numTxAnt = 2

    # Get the information about the profile configuration
    if "profileCfg" in splitWords[0]:
        startFreq = int(float(splitWords[2]))
        idleTime = int(splitWords[3])
        rampEndTime = float(splitWords[5])
        freqSlopeConst = float(splitWords[8])
        numAdcSamples = int(splitWords[10])
        numAdcSamplesRoundTo2 = 1;
```

```

while numAdcSamples > numAdcSamplesRoundTo2:
    numAdcSamplesRoundTo2 = numAdcSamplesRoundTo2 * 2;

digOutSampleRate = int(splitWords[11]);

# Get the information about the frame configuration
elif "frameCfg" in splitWords[0]:

    chirpStartIdx = int(splitWords[1]);
    chirpEndIdx = int(splitWords[2]);
    numLoops = int(splitWords[3]);
    numFrames = int(splitWords[4]);
    framePeriodicity = int(splitWords[5]);

while True:

    byteCount = Dataport.inWaiting()
    s = Dataport.read(byteCount)
    readNumBytes = byteCount
    allBinData = s

# init local variables
totalBytesParsed = 0;
numFramesParsed = 0;

# parser one mmw_demo output_packet extracts only one complete frame at a time
# so call this in a loop till end of file
while (totalBytesParsed < readNumBytes):

    # parser one mmw_demo output_packet function already prints the
    # parsed data to stdout. So showcasing only saving the data to arrays
    # here for further custom processing
    parser result, \
    headerStartIndex, \
    totalPacketNumBytes, \
    numDetObj, \
    numTlv, \
    subFrameNumber, \
    detectedX array, \
    detectedY array, \
    detectedZ array, \
    detectedV array, \
    detectedRange array, \
    detectedAzimuth array, \
    detectedElevation array, \
    detectedSNR array, \
    detectedNoise array = parser one mmw_demo output_packet(allBinData[totalBytesParsed::1], readNumBytes-totalBytesParsed, ID, X, Y, Z,
yaw psi, pitch theta, roll phi)

    if (parser result == 0):
        totalBytesParsed += (headerStartIndex+totalPacketNumBytes)
        numFramesParsed+=1

    else:

        break

time.sleep(sensor delay)

UDPstring = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
print(UDPstring)
UDPstring = str(UDPstring).encode('utf-16')
UDPCliet.sendto(UDPstring, serverAddress)

```

4.2 Code 2 - Parcing Code for mmWave Sensors on Raspberry Pi

```

import struct
import math
import binascii
import codecs
import socket
import time
import numpy as np
from datetime import datetime

# definations for parser pass/fail
TC_PASS = 0
TC_FAIL = 1

serverAddress = ('192.168.30.150',2222)
UDPCliet = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

def getUInt32(data):
    """
    This function coverts 4 bytes to a 32-bit unsigned integer.
    @param data : 1-dimension byte array
    @return : 32-bit unsigned integer
    """
    return (data[0] +
            data[1]*256 +
            data[2]*65536 +
            data[3]*16777216)

def getUInt16(data):
    """
    This function coverts 2 bytes to a 16-bit unsigned integer.
    @param data : 1-dimension byte array
    @return : 16-bit unsigned integer
    """
    return (data[0] +
            data[1]*256)

def getHex(data):
    """

```



```

This function converts 4 bytes to a 32-bit unsigned integer in hex.

@param data : 1-dimension byte array
@return    : 32-bit unsigned integer in hex
"""
return (binascii.hexlify(data[:-1]))

def checkMagicPattern(data):
    """
    This function check if data array contains the magic pattern which is the start of one mmw demo output packet.

    @param data : 1-dimension byte array
    @return    : 1 if magic pattern is found
                0 if magic pattern is not found
    """
    found = 0
    if (data[0] == 2 and data[1] == 1 and data[2] == 4 and data[3] == 3 and data[4] == 6 and data[5] == 5 and data[6] == 8 and data[7] == 7):
        found = 1
    return (found)

def parser_helper(data, readNumBytes):
    """
    This function is called by parser_one_mmw_demo_output_packet() function or application to read the input buffer, find the magic number,
    header location, the length of frame, the number of detected object and the number of TLV contained in this mmw demo output packet.

    @param data : 1-dimension byte array holds the the data read from mmw demo output. It ignorant of the fact that data
    is coming from UART directly or file read.
    @param readNumBytes : the number of bytes contained in this input byte array

    @return headerStartIndex : the mmw demo output packet header start location
    @return totalPacketNumBytes : the mmw demo output packet length
    @return numDetObj : the number of detected objects contained in this mmw demo output packet
    @return numTlv : the number of TLV contained in this mmw demo output packet
    @return subFrameNumber : the sbuframe index (0,1,2 or 3) of the frame contained in this mmw demo output packet
    """

    headerStartIndex = -1

    for index in range(readNumBytes):
        if checkMagicPattern(data[index:index+8:1]) == 1:
            headerStartIndex = index
            break

    if headerStartIndex == -1: # does not find the magic number i.e output packet header
        totalPacketNumBytes = -1
        numDetObj = -1
        numTlv = -1
        subFrameNumber = -1
        platform = -1
        frameNumber = -1
        timeCpuCycles = -1
    else: # find the magic number i.e output packet header
        totalPacketNumBytes = getUInt32(data[headerStartIndex+12:headerStartIndex+16:1])
        platform = getHex(data[headerStartIndex+16:headerStartIndex+20:1])
        frameNumber = getUInt32(data[headerStartIndex+20:headerStartIndex+24:1])
        timeCpuCycles = getUInt32(data[headerStartIndex+24:headerStartIndex+28:1])
        numDetObj = getUInt32(data[headerStartIndex+28:headerStartIndex+32:1])
        numTlv = getUInt32(data[headerStartIndex+32:headerStartIndex+36:1])
        subFrameNumber = getUInt32(data[headerStartIndex+36:headerStartIndex+40:1])

    return (headerStartIndex, totalPacketNumBytes, numDetObj, numTlv, subFrameNumber)

def parser_one_mmw_demo_output_packet(data, readNumBytes, ID, X translation, Y translation, Z translation, yaw psi, pitch theta, roll phi):
    """
    This function is called by application. Firstly it calls parser helper() function to find the start location of the mmw demo output packet,
    then extract the contents from the output packet.
    Each invocation of this function handles only one frame at a time and user needs to manage looping around to parse data for multiple
    frames.

    @param data : 1-dimension byte array holds the the data read from mmw demo output. It ignorant of the fact that data
    is coming from UART directly or file read.
    @param readNumBytes : the number of bytes contained in this input byte array

    @return result : parser result. 0 pass otherwise fail
    @return headerStartIndex : the mmw demo output packet header start location
    @return totalPacketNumBytes : the mmw demo output packet length
    @return numDetObj : the number of detected objects contained in this mmw demo output packet
    @return numTlv : the number of TLV contained in this mmw demo output packet
    @return subFrameNumber : the sbuframe index (0,1,2 or 3) of the frame contained in this mmw demo output packet
    @return detectedX_array : 1-dimension array holds each detected target's x of the mmw demo output packet
    @return detectedY_array : 1-dimension array holds each detected target's y of the mmw demo output packet
    @return detectedZ_array : 1-dimension array holds each detected target's z of the mmw demo output packet
    @return detectedV_array : 1-dimension array holds each detected target's v of the mmw demo output packet
    @return detectedRange_array : 1-dimension array holds each detected target's range profile of the mmw demo output packet
    @return detectedAzimuth_array : 1-dimension array holds each detected target's azimuth of the mmw demo output packet
    @return detectedElevAngle_array : 1-dimension array holds each detected target's elevAngle of the mmw demo output packet
    @return detectedSNR_array : 1-dimension array holds each detected target's snr of the mmw demo output packet
    @return detectedNoise_array : 1-dimension array holds each detected target's noise of the mmw demo output packet
    """

    headerNumBytes = 40

    PI = 3.14159265

    detectedX_array = []
    detectedY_array = []
    detectedZ_array = []
    detectedV_array = []
    detectedRange_array = []
    detectedAzimuth_array = []
    detectedElevAngle_array = []
    detectedSNR_array = []
    detectedNoise_array = []

    result = TC_PASS

    # call parser_helper() function to find the output packet header start location and packet size
    (headerStartIndex, totalPacketNumBytes, numDetObj, numTlv, subFrameNumber) = parser_helper(data, readNumBytes)

    if headerStartIndex == -1:

```

```

result = TC_FAIL
print("***** Frame Fail, cannot find the magic words *****")
else:
    nextHeaderStartIndex = headerStartIndex + totalPacketNumBytes
    if int(headerStartIndex) + int(totalPacketNumBytes) > int(readNumBytes):
        result = TC_FAIL
        print("***** Frame Fail, readNumBytes may not long enough *****")
    elif int(nextHeaderStartIndex) + 8 < int(readNumBytes) and
checkMagicPattern(data[int(nextHeaderStartIndex):int(nextHeaderStartIndex)+8:1]) == 0:
        result = TC_FAIL
        print("***** Frame Fail, incomplete packet *****")
    elif int(numDetObj) <= 0:
        result = TC_FAIL
        print("***** Frame Fail, numDetObj = %d *****" % (numDetObj))
    elif int(subFrameNumber) > 3:
        result = TC_FAIL
        print("***** Frame Fail, subFrameNumber = %d *****" % (subFrameNumber))
    else:
        # process the 1st TLV
        tlvStart = int(headerStartIndex) + int(headerNumBytes)

        tlvType = getUInt32(data[tlvStart+0:tlvStart+4:1])
        tlvLen = getUInt32(data[tlvStart+4:tlvStart+8:1])
        offset = 8

        # the 1st TLV must be type 1
        if tlvType == 1 and int(tlvLen) < int(totalPacketNumBytes):#MMWDEMO UART MSG DETECTED POINTS

            # TLV type 1 contains x, y, z, v values of all detect objects.
            # each x, y, z, v are 32-bit float in IEEE 754 single-precision binary floating-point format, so every 16 bytes represent x, y,
            # z, v values of one detect objects.

            # for each detect objects, extract/convert float x, y, z, v values and calculate range profile and azimuth
            for obj in range(int(numDetObj)):
                # convert byte0 to byte3 to float x value
                x = struct.unpack('<f', codecs.decode(binascii.hexlify(data[tlvStart + offset:tlvStart + offset+4:1]),'hex'))[0]

                # convert byte4 to byte7 to float y value
                y = struct.unpack('<f', codecs.decode(binascii.hexlify(data[tlvStart + offset+4:tlvStart + offset+8:1]),'hex'))[0]

                # convert byte8 to byte11 to float z value
                z = struct.unpack('<f', codecs.decode(binascii.hexlify(data[tlvStart + offset+8:tlvStart + offset+12:1]),'hex'))[0]

                # convert byte12 to byte15 to float v value
                v = struct.unpack('<f', codecs.decode(binascii.hexlify(data[tlvStart + offset+12:tlvStart + offset+16:1]),'hex'))[0]

                # calculate range profile from x, y, z
                compDetectedRange = math.sqrt((x * x)+(y * y)+(z * z))

                # calculate azimuth from x, y
                if y == 0:
                    if x >= 0:
                        detectedAzimuth = 90
                    else:
                        detectedAzimuth = -90
                else:
                    detectedAzimuth = math.atan(x/y) * 180 / PI

                # calculate elevation angle from x, y, z
                if x == 0 and y == 0:
                    if z >= 0:
                        detectedElevAngle = 90
                    else:
                        detectedElevAngle = -90
                else:
                    detectedElevAngle = math.atan(z/math.sqrt((x * x)+(y * y))) * 180 / PI

                detectedX_array.append(x)
                detectedY_array.append(y)
                detectedZ_array.append(z)
                detectedV_array.append(v)
                detectedRange_array.append(compDetectedRange)
                detectedAzimuth_array.append(detectedAzimuth)
                detectedElevAngle_array.append(detectedElevAngle)

                offset = offset + 16
            # end of for obj in range(numDetObj) for 1st TLV

        # Process the 2nd TLV
        tlvStart = tlvStart + 8 + tlvLen

        tlvType = getUInt32(data[tlvStart+0:tlvStart+4:1])
        tlvLen = getUInt32(data[tlvStart+4:tlvStart+8:1])
        offset = 8

        if tlvType == 7:

            # TLV type 7 contains snr and noise of all detect objects.
            # each snr and noise are 16-bit integer represented by 2 bytes, so every 4 bytes represent snr and noise of one detect objects.

            # for each detect objects, extract snr and noise
            for obj in range(int(numDetObj)):
                # byte0 and byte1 represent snr. convert 2 bytes to 16-bit integer
                snr = getUInt16(data[tlvStart + offset + 0:tlvStart + offset + 2:1])
                # byte2 and byte3 represent noise. convert 2 bytes to 16-bit integer
                noise = getUInt16(data[tlvStart + offset + 2:tlvStart + offset + 4:1])

                detectedSNR_array.append(snr)
                detectedNoise_array.append(noise)

                offset = offset + 4
            else:
                for obj in range(numDetObj):
                    detectedSNR_array.append(0)
                    detectedNoise_array.append(0)
        # end of if tlvType == 7

```

```

for obj in range(numDetObj):
    detectedRange = detectedRange_array[obj]
    detectedRange = str(detectedRange).encode('utf-16')

    dt = datetime.now()
    ts = datetime.timestamp(dt)

    X_Anchor = X_translation
    Y_Anchor = Y_translation
    Z_Anchor = Z_translation

    X = detectedX_array[obj]
    Y = detectedY_array[obj]
    Z = detectedZ_array[obj]

    P_ = np.array([Y, -X, Z]).reshape(-1,1)

    phi = roll_phi/180*math.pi
    theta = pitch_theta/180*math.pi
    psi = yaw_psi/180*math.pi

    Rx = np.array([[1, 0, 0],
                   [0, np.cos(phi), -np.sin(phi)],
                   [0, np.sin(phi), np.cos(phi)]])

    Ry = np.array([[np.cos(theta), 0, np.sin(theta)],
                   [0, 1, 0],
                   [-np.sin(theta), 0, np.cos(theta)]])

    Rz = np.array([[np.cos(psi), -np.sin(psi), 0],
                   [np.sin(psi), np.cos(psi), 0],
                   [0, 0, 1]])

    RotationMatrix = np.dot(np.matmul(Rz, Ry), Rx)

    P = np.matmul(RotationMatrix, P_)

    P = P.flatten().tolist()

    X_New = P[0]+X_Anchor
    Y_New = P[1]+Y_Anchor
    Z_New = P[2]+Z_Anchor

    string = [ID, ts, numDetObj, obj, X_New, Y_New, Z_New, detectedV_array[obj], detectedRange_array[obj], detectedAzimuth_array[obj],
    detectedElevAngle_array[obj], detectedSNR_array[obj], detectedNoise_array[obj]]
    print(string)
    string = str(string).encode('utf-16')
    UDPCClient.sendto(string, serverAddress)

    return (result, headerStartIndex, totalPacketNumBytes, numDetObj, numTlv, subFrameNumber, detectedX_array, detectedY_array, detectedZ_array,
    detectedV_array, detectedRange_array, detectedAzimuth_array, detectedElevAngle_array, detectedSNR_array, detectedNoise_array)

```

4.3 Code 3 - Example of mmWave Configuration File

```

% *****
% Created for SDK ver:03.06
% Created using Visualizer ver:3.6.0.0
% Frequency:77
% Platform:xWR18xx
% Scene Classifier:best_range_res
% Azimuth Resolution(deg):15 + Elevation
% Range Resolution(m):0.044
% Maximum unambiguous Range(m):9.02
% Maximum Radial Velocity(m/s):1
% Radial velocity resolution(m/s):0.13
% Frame Duration(msec):100
% RF calibration data:None
% Range Detection Threshold (dB):15
% Doppler Detection Threshold (dB):15
% Range Peak Grouping:enabled
% Doppler Peak Grouping:enabled
% Static clutter removal:disabled
% Angle of Arrival FoV: Full FoV
% Range FoV: Full FoV
% Doppler FoV: Full FoV
% *****
sensorStop
flushCfg
dfeDataOutputMode 1
channelCfg 15 7 0
adcCfg 2 1
adcbufCfg -1 0 1 1 1
profileCfg 0 77 267 7 57.14 0 0 70 1 256 5209 0 0 30
chirpCfg 0 0 0 0 0 0 1
chirpCfg 1 1 0 0 0 0 4
chirpCfg 2 2 0 0 0 0 2
frameCfg 0 2 16 0 100 1 0
lowPower 0 0
guiMonitor -1 1 1 0 0 0 1
cFarCfg -1 0 2 8 4 3 0 15 1
cFarCfg -1 1 0 4 2 3 1 15 1
multiObjBeamForming -1 1 0.5
clutterRemoval -1 1
calibDcRangeSig -1 0 -5 8 256
extendedMaxVelocity -1 0
lvdsStreamCfg -1 0 0 0

```

```
compRangeBiasAndRxChanPhase 0.0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
measureRangeBiasAndRxChanPhase 0 1.5 0.2
CQRxSatMonitor 0 3 5 121 0
CQSigImgMonitor 0 127 4
analogMonitor 0 0
aoaFovCfg -1 -90 90 -90 90
cfarFovCfg -1 0 0 8.92
cfarFovCfg -1 1 -1 1.00
calibData 0 0 0
sensorStart
```

4.4 Code 4 - Central PC Python Script

```
import time
import statistics
import math
import numpy as np
import socket
import json
from mySQLfunctions import *

DRIVER_NAME = 'SQL SERVER'
SERVER_NAME = '192.168.30.150,49170'
DATABASE_NAME = 'thesisDB'

connection_string = f"""
    DRIVER={{DRIVER_NAME}};
    SERVER={{SERVER_NAME}};
    DATABASE={{DATABASE_NAME}};
    Trust_Connection=yes;
    uid = 'thesisProject'
    pwd = 'thesisProject'
"""

connection_string = "DSN=thesisDSN;UID=sa;PWD=thesisProject"
connection_string = "DSN=thesisDSN;Trust_Connection=yes;"
connection_string = "DSN=thesisDSN;"

try:
    conn=odbc.connect(connection_string)
    print(conn)
except Exception as e:
    print(e)
    print('task is terminated')
    sys.exit()
else:
    cursor = conn.cursor()

sensor_delay = 0.25

mmWaveSensorsConf='testConfiguration2'

mmWaveSensors=[ # IWR1843 Sensors
    ('192.168.30.151', 'IWR1843BOOST', '7101300600', 0.46, 0.33, 2.81, 45, 15, 0),
    ('192.168.30.152', 'IWR1843BOOST', '7101300531', 0.34, 4.32, 1.23, 0, -35, 0),
    ('192.168.30.153', 'IWR1843BOOST', '7101300620', 0.53, 8.50, 2.39, -45, 15, 0),
    ('192.168.30.154', 'IWR1843BOOST', '7101300571', 4.06, 8.50, 1.995, -90, 0, 0),
    ('192.168.30.155', 'IWR1843BOOST', '7101300634', 3.96, 0.33, 2.84, 90, 15, 0)
]

totalNumberOfSensors=len(mmWaveSensors)

mmWaveSensorsConfiguration=configureSensors(mmWaveSensorsConf,mmWaveSensors, conn, cursor)
mmWaveSensorConfiguration_ID=mmWaveSensorsConfiguration[0][2]

mmWaveSensorsIndices=1,2,3,4,5

counter=-1;
for sensor in mmWaveSensors:
    counter=counter+1;
    HOST = sensor[0] # The server's hostname or IP address
    PORT = 65432 # The port used by the server

    try:
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
            s.connect((HOST, PORT))
            test = [mmWaveSensorsIndices[counter], sensor[3], sensor[4], sensor[5], sensor_delay, sensor[6], sensor[7], sensor[8]]
            test = str(test).encode('utf-8')
            s.sendall(test)
            data = s.recv(1024)
            print(f"Received {data.decode('utf-8')!r}")
            s.close
    except:
        print(sensor[0] + ' not reachable')

drone_record = ('UCLan DJI', 'DJI Air 2S', '3YTSJ58')
drone_ID=insert_drone(drone_record,conn,cursor)

imuSensor_record = ('192.168.30.200', 'MPU 9250 - Grove IMU 9DOF v2.0', 'v2206282016', drone_ID)
imuSensor_ID = insert_imuSensor(imuSensor_record,conn,cursor)
```

```
imuConf_record = ['Mounted on DJI Air 2S', 0, imuSensor_ID]
imuConf_ID = insert_imuConf(imuConf_record,conn,cursor)

experiment_record = ('Experiment 4', 'Test Algorithms', datetime.datetime.now(),imuConf_ID, mmWaveSensorConfiguration_ID)
experiment_ID = insert_experiment(experiment_record,conn,cursor)

imuHOST=imuSensor_record[0]
try:
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.connect((imuHOST, PORT))
        test = [imuConf_ID,experiment_ID]
        test = str(test).encode('utf-16')
        s.sendall(test)
        data = s.recv(1024)
        print(f"Received {data.decode('utf-16')!r}")
        s.close
except:
    print('IMU with IP: '+imuSensor_record[0] + ' not reachable')

timeWindow=1.2*sensor_delay
bufferSize = 10240
msgFromServer = "Hello Client"
ServerPort = 2222
imuServerPort =2223
ServerIP = '192.168.30.150'
bytesToSend = msgFromServer.encode('utf-16')
RPIsocket_mmWave = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
RPIsocket_mmWave.bind((ServerIP, ServerPort))
print('mmWave Server is Up and Listening...')
RPIsocket_imu = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
RPIsocket_imu.bind((ServerIP, imuServerPort))
print('imu Server is Up and Listening...')
cnt = 0
message1 = "Hello"

def tcp_sim(num_seconds,RPIsocket, bufferSize,timeStampIndex):

    data_list = []

    start_time = time.time()

    # Receive the data from the TCP and put all the data into a list every second
    while time.time() - start_time < num_seconds:
        try:
            message, address = RPIsocket.recvfrom(bufferSize)
            message = message.decode('utf-16')
            message_split = message.split(' ')

            if message_split[timeStampIndex] >= str(start_time):
                data_list.append(message)

        except:
            pass

    return data_list

def remove_outliers(data, z_threshold=0.7):
    # Convert each data point from JSON to dictionary
    data = [json.loads(data_str) for data_str in data]
    data = np.array(data)
    data = data.astype('<U12')

    x = data[:, 4].astype(float)
    y = data[:, 5].astype(float)
    z = data[:, 6].astype(float)

    # Compute the z-scores for x and y

    z_x = np.abs((x - np.mean(x)) / np.std(x))
    z_y = np.abs((y - np.mean(y)) / np.std(y))
    z_z = np.abs((z - np.mean(z)) / np.std(z))

    # Keep only the data points with z-scores below the threshold
    filtered_data = data[(z_x < z_threshold) & (z_y < z_threshold) & (z_z < z_threshold)]

    return filtered_data

def compute_average_center(filteredArray):

    SizeOfResultArray = len(filteredArray)

    x_total = 0
    y_total = 0
    z_total = 0

    for data_entry in filteredArray:
        x_total = x_total + float(data_entry[4])
        y_total = y_total + float(data_entry[5])
        z_total = z_total + float(data_entry[6])

    x_average = x_total/SizeOfResultArray
    y_average = y_total/SizeOfResultArray
```

```

z_average = z_total/SizeOfResultArray

CenterOfGravityXYZ = []
CenterOfGravityXYZ.append(x_average)
CenterOfGravityXYZ.append(y_average)
CenterOfGravityXYZ.append(z_average)

CenterOfGravityXYZ = tuple(CenterOfGravityXYZ)

return CenterOfGravityXYZ

def sensor_measurements(filteredArray, CenterOfGravityXYZ, totalNumberOfSensors):
    distanceToCentreOfGravity=[float('inf')] * totalNumberOfSensors
    distanceToSensor = [0] * totalNumberOfSensors
    azimuthFromSensor = [0] * totalNumberOfSensors
    elevationFromSensor = [0] * totalNumberOfSensors
    velocity = [0] * totalNumberOfSensors
    Xvalue = [0] * totalNumberOfSensors
    Yvalue = [0] * totalNumberOfSensors
    Zvalue = [0] * totalNumberOfSensors

    for data_entry in filteredArray:
        X = float(data_entry[4])
        Y = float(data_entry[5])
        Z = float(data_entry[6])

        distance = math.sqrt((CenterOfGravityXYZ[0] - X) ** 2 + (CenterOfGravityXYZ[1] - Y) ** 2 + (CenterOfGravityXYZ[2] - Z) ** 2)
        firstValue=data_entry[0]
        current_sensor=int(float(firstValue))

        if distance<=distanceToCentreOfGravity[current_sensor-1]:
            distanceToCentreOfGravity[current_sensor - 1]=distance
            distanceToSensor[current_sensor-1]=float(data_entry[8])
            azimuthFromSensor[current_sensor - 1] = float(data_entry[9])
            elevationFromSensor[current_sensor - 1] = float(data_entry[10])
            velocity[current_sensor - 1] = float(data_entry[7])
            Xvalue[current_sensor - 1] = float(data_entry[4])
            Yvalue[current_sensor - 1] = float(data_entry[5])
            Zvalue[current_sensor - 1] = float(data_entry[6])

        else:
            continue

    return distanceToSensor, azimuthFromSensor, elevationFromSensor, velocity, Xvalue, Yvalue, Zvalue

while 1:
    process_start_time = time.time()

    imu_array = tcp_sim(timeWindow,RPIsocket_imu, bufferSize,0)
    imu_string_temp=imu_array[-1]
    imu_string=imu_string_temp.split(', ')

    acc_x = float(imu_string[1])
    acc_y = float(imu_string[2])
    acc_z = float(imu_string[3])
    gyr_x = float(imu_string[4])
    gyr_y = float(imu_string[5])
    gyr_z = float(imu_string[6])
    mag_x = float(imu_string[7])
    mag_y = float(imu_string[8])
    mag_z_temp = imu_string[9]
    mag_z = float(mag_z_temp[:-1])
    print(acc_x,acc_y,acc_z,gyr_x,gyr_y,gyr_z,mag_x,mag_y,mag_z)

    measurement_record = (datetime.datetime.now(), experiment_ID)
    measurement_ID = insert_measurement(measurement_record, conn, cursor)
    imuMeasurement_record = (acc_x, acc_y, acc_z, gyr_x, gyr_y, gyr_z, mag_x, mag_y, mag_z, imuSensor_ID, measurement_ID)
    imuMeasurementID = insert_imuMeasurement(imuMeasurement_record, conn, cursor)

    time1 = time.time()
    result_array = tcp_sim(timeWindow,RPIsocket_mmwWave, bufferSize,1)
    time2 = time.time()
    print(time2-time1)

    if (len(result_array)==0):

        distanceList=[0]*totalNumberOfSensors
        azimuthList = [0] * totalNumberOfSensors
        elevationList = [0] * totalNumberOfSensors
        velocityList = [0] * totalNumberOfSensors
        XList = [0] * totalNumberOfSensors
        YList = [0] * totalNumberOfSensors
        ZList = [0] * totalNumberOfSensors

    else:
        filtered_array = remove_outliers(result_array)
        if (len(filtered_array)==0):
            distanceList = [0] * totalNumberOfSensors
            azimuthList = [0] * totalNumberOfSensors
            elevationList = [0] * totalNumberOfSensors
            velocityList = [0] * totalNumberOfSensors
            XList = [0] * totalNumberOfSensors
            YList = [0] * totalNumberOfSensors
            ZList = [0] * totalNumberOfSensors

```

```

else:
    CenterOfGravityXYZ = compute_average_center(filtered_array)
    distanceList,azimuthList,elevationList,velocityList,XList,YList,ZList
sensor_measurements(filtered_array,CenterOfGravityXYZ,totalNumberOfSensors)

print(distanceList)

for i in range(len(distanceList)):
    distance = distanceList[i]
    azimuth = azimuthList[i]
    elevation = elevationList[i]
    xMeas = XList[i]
    yMeas = YList[i]
    zMeas = ZList[i]
    sensorConf=mmWaveSensorsConfiguration[i]
    mmWaveMeasurement_record = (distance, azimuth, elevation, xMeas, yMeas, zMeas, sensorConf[0], measurement_ID)
    insert_mmWaveMeasurement(mmWaveMeasurement_record, conn, cursor)

```

4.5 Code 5 - SQL Server Script

```

import sys
import datetime
import pyodbc as odbc

def insert_drone(drone_record,conn,cursor):
    insert_statement = """
        INSERT INTO drone (droneName, droneModel, droneSerial)
        VALUES (?,?,?)
        SELECT @@IDENTITY AS [Last-Inserted Identity Value]
    """

    #check if the drone exists in the database based on its serialNumber
    #select command returns the last entry in the table that matches the serial
    select_statement = """
        SELECT TOP 1 droneID from drone
        where drone.droneSerial='%s'
        ORDER BY droneID DESC
    """ % drone_record[-1]

    try:
        cursor.execute(select_statement)
        res = cursor.fetchone()
        # if drone exists return the primary key
        if (bool(res)):
            print("Drone with serial '%s' exists in the Database with ID '%s' updated" % (drone_record[-1], res[0]))
            update_statement = """
                UPDATE drone
                SET drone.droneName='%s', drone.droneModel='%s'
                where drone.droneSerial='%s'
            """ % (drone_record[0], drone_record[1], drone_record[2])
            cursor.execute(update_statement)
            cursor.commit()
            return res[0]
        #else add the drone and return its primary key
        else:
            cursor.execute(insert_statement, drone_record)
            cursor.execute("SELECT @@identity")
            res = cursor.fetchone()
            print("Drone with serial '%s' added in the Database with ID '%s'" % (drone_record[-1], res[0]))

    except Exception as e:
        cursor.rollback()
        print(e.value)
        print('drone transaction rolled back')

    else:
        print('Drone records inserted successfully')
        cursor.commit()

    return int(res[0])

def insert_mmWaveSensor(mmWaveSensor_record,conn,cursor):
    insert_statement = """
        INSERT INTO mmWaveSensor (mmWaveSensorName, mmWaveSensorModel, mmWaveSensorSerial)
        VALUES (?,?,?)
        SELECT @@IDENTITY AS [Last-Inserted Identity Value]
    """

    #check if the mmWave exists in the database based on its serialNumber
    #select command returns the last entry in the table that matches the serial
    select_statement = """
        SELECT TOP 1 mmWaveSensorID from mmWaveSensor
        where mmWaveSensor.mmWaveSensorSerial='%s'
        ORDER BY mmWaveSensorID DESC
    """ % mmWaveSensor_record[-1]

    try:
        cursor.execute(select_statement)

```

```

res = cursor.fetchone()
# if drone exists return the primary key
if (bool(res)):
    print("mmWave Sensor with serial '%s' exists in the Database with ID '%s' updated" % (mmWaveSensor_record[-1], res[0]))
    update_statement = """
        UPDATE mmWaveSensor
        SET mmWaveSensor.mmWaveSensorName='%s', mmWaveSensor.mmWaveSensorModel='%s'
        where mmWaveSensor.mmWaveSensorSerial='%s'
        """ % (mmWaveSensor_record[0], mmWaveSensor_record[1], mmWaveSensor_record[2])
    cursor.execute(update_statement)
    cursor.commit()
    return res[0]
#else add the drone and return its primary key
else:
    cursor.execute(insert_statement, mmWaveSensor_record)
    cursor.execute("SELECT @@identity")
    res = cursor.fetchone()

except Exception as e:
    cursor.rollback()
    print(e.value)

else:
    cursor.commit()

return int(res[0])

def insert_imuSensor(imuSensor_record,conn,cursor):
    insert_statement = """
        INSERT INTO imuSensor (imuSensorName, imuSensorModel, imuSensorSerial, droneID)
        VALUES (?,?,,?)
        SELECT @@IDENTITY AS [Last-Inserted Identity Value]
        """

    #check if the imu exists in the database based on its serialNumber and the particular drone
    #select command returns the last entry in the table that matches the serial
    select_statement = """
        SELECT TOP 1 imuSensorID from imuSensor
        where imuSensor.imuSensorSerial='%s' and imuSensor.droneID='%d'
        ORDER BY imuSensorID DESC
        """ % (imuSensor_record[-2], imuSensor_record[-1])

    try:
        cursor.execute(select_statement)
        res = cursor.fetchone()
        # if drone exists return the primary key
        if (bool(res)):
            print("imu Sensor with serial '%s' on drone with ID: '%d' exists in the Database with ID '%s' updated" % (imuSensor_record[-2],
imuSensor_record[-1], res[0]))

            update_statement = """
                UPDATE imuSensor
                SET imuSensor.imuSensorName='%s', imuSensor.imuSensorModel='%s'
                where imuSensor.imuSensorSerial='%s' and imuSensor.droneID='%d'
                """ % (imuSensor_record[0], imuSensor_record[1], imuSensor_record[2], imuSensor_record[3])
            cursor.execute(update_statement)
            cursor.commit()
            return int(res[0])
        #else add the drone and return its primary key
        else:
            cursor.execute(insert_statement, imuSensor_record)
            cursor.execute("SELECT @@identity")
            res = cursor.fetchone()
            print("imu Sensor with serial '%s' on drone with ID: '%d' added in the Database with ID '%s'" % (imuSensor_record[-2],
imuSensor_record[-1], res[0]))

    except Exception as e:
        cursor.rollback()
        print(e.value)
        print('imu Sensor transaction rolled back')

    else:
        print('imu Sensor record inserted successfully')
        cursor.commit()

    return int(res[0])

def insert_experiment(experiment_record,conn,cursor):
    insert_statement = """
        INSERT INTO experiment (experimentName, experimentDesc, experimentRunDate,experimentCreateDate,imuConfID, mmWaveSensorConfID)
        VALUES (?,?,,?,?)
        SELECT @@IDENTITY AS [Last-Inserted Identity Value]
        """

    #check if the experiment record exists in the database based on the experiment Name
    #select command returns the last entry in the table that matches the name
    select_statement = """
        SELECT TOP 1 experimentID from experiment
        where experiment.experimentName='%s' and experiment.imuConfID='%d' and experiment.mmWaveSensorConfID='%d'
        ORDER BY experimentID DESC
        """ % (experiment_record[0], experiment_record[-2], experiment_record[-1])

    try:
        cursor.execute(select_statement)
        res = cursor.fetchone()
        # if drone exists return the primary key

```



```

if (bool(res)):
    print("Experiment with name %s exists in the Database with ID '%s' updated" % (experiment_record[0], res[0]))

    update_statement = """
        UPDATE experiment
        SET experiment.experimentDesc='%s', experiment.experimentRunDate='%s'
        where experiment.experimentName='%s' and experiment.imuConfID='%d' and experiment.mmWaveSensorConfID='%d'
        """ % (experiment_record[1], experiment_record[2].strftime('%Y-%m-%d %H:%M:%S'), experiment_record[0], experiment_record[-2],
experiment_record[-1])
    cursor.execute(update_statement)
    cursor.commit()
    return res[0]

#else add the drone and return its primary key
else:
    record=(experiment_record[0], experiment_record[1], experiment_record[2].strftime('%Y-%m-%d %H:%M:%S'),
experiment_record[2].strftime('%Y-%m-%d %H:%M:%S'), experiment_record[3], experiment_record[4])
    cursor.execute(insert_statement, record)
    cursor.execute("SELECT @@identity")
    res = cursor.fetchone()
    print("Experiment with name %s with creation Date: '%s' added in the Database with ID '%s'" % (experiment_record[0],
experiment_record[2].strftime("%m/%d/%Y, %H:%M:%S"), res[0]))

except Exception as e:
    cursor.rollback()
    print(e.value)
    print('Experiment transaction rolled back')

else:
    print('experiment record inserted successfully')
    cursor.commit()

return res[0]

def insert_imuConf(imuConf_record,conn,cursor):
    insert_statement = """
        INSERT INTO imuConf (imuConfDesc, imuMagneticOrientation, imuSensorID)
        VALUES (?, ?, ?)
        SELECT @@IDENTITY AS [Last-Inserted Identity Value]
        """

    #check if the imuConf record exists in the database based on the imu Sensor ID
    #select command returns the last entry in the table that matches the ID
    select_statement = """
        SELECT TOP 1 imuConfID from imuConf
        where imuConf.imuSensorID='%d'
        ORDER BY imuConfID DESC
        """ % (imuConf_record[2])

    try:
        cursor.execute(select_statement)
        res = cursor.fetchone()
        # if drone exists return the primary key
        if (bool(res)):
            print("Configuration for imu sensors with id '%s' exists in the Database with ID '%s' updated" % (imuConf_record[0], res[0]))

            update_statement = """
                UPDATE imuConf
                SET imuConf.imuConfDesc='%s', imuConf.imuMagneticOrientation='%d'
                where imuConf.imuSensorID='%d'
                """ % (imuConf_record[0], imuConf_record[1], imuConf_record[2])
            cursor.execute(update_statement)
            cursor.commit()
            return res[0]

        #else add the drone and return its primary key
        else:
            cursor.execute(insert_statement, imuConf_record)
            cursor.execute("SELECT @@identity")
            res = cursor.fetchone()
            print("Configuration for imu sensors with id '%s' added in the Database with ID '%s' updated" % (imuConf_record[0], res[0]))

    except Exception as e:
        cursor.rollback()
        print(e.value)
        print('imu Configuration transaction rolled back')

    else:
        print('imu Configuration record inserted successfully')
        cursor.commit()

    return int(res[0])

def insert_mmWaveConf(mmWaveSensorsConf,mmWaveSensors,conn,cursor):
    insert_statement = """
        INSERT INTO mmWaveSensorConf (mmWaveSensorConfDesc, mmWaveSensorConfNoOfSensors)
        VALUES (?, ?)
        SELECT @@IDENTITY AS [Last-Inserted Identity Value]
        """

    try:
        mmWaveConf_record = (mmWaveSensorsConf, len(mmWaveSensors))
        cursor.execute(insert_statement, mmWaveConf_record)
        cursor.execute("SELECT @@identity")
        res = cursor.fetchone()
        print(
            "Configuration for mmWave sensors with description '%s' and '%d' sensors added in the Database with ID '%s'" % (
                mmWaveConf_record[0], mmWaveConf_record[1], res[0]))
    
```

```

except Exception as e:
    cursor.rollback()
    print(e.value)
    print('mmWave Configuration transaction rolled back')

else:
    print('mmWave Configuration record inserted successfully')
    cursor.commit()

# Proceed normally and add the Sensors and their location
mmWaveConfRes = []
for mmWaveSensor in mmWaveSensors:
    mmWaveSensor_record = (mmWaveSensor[0], mmWaveSensor[1], mmWaveSensor[2])
    mmWaveSensor_ID = insert_mmWaveSensor(mmWaveSensor_record, conn, cursor)
    mmWaveSensorLoc_record = (mmWaveSensor_ID, mmWaveSensor[3], mmWaveSensor[4], mmWaveSensor[5],
mmWaveSensor[6], mmWaveSensor[7], mmWaveSensor[8], res[0])
    mmWaveSensorLocID = insert_mmWaveSensorLoc(mmWaveSensorLoc_record, conn, cursor)
    sensorConf=(mmWaveSensor_ID,int(mmWaveSensorLocID), int(res[0]))
    print(type(sensorConf))
    mmWaveConfRes.append(sensorConf)

return mmWaveConfRes

def insert_mmWaveSensorLoc(mmWaveSensorLoc_record,conn,cursor):
    insert_statement = """
    INSERT INTO mmWaveSensorLoc (mmWaveSensorID, x, y, z, yaw_psi, pitch_theta, roll_phi, mmWaveSensorConfID)
    VALUES (?, ?, ?, ?, ?, ?, ?, ?)
    """
    SELECT @@IDENTITY AS [Last-Inserted Identity Value]

    #check if the mmWaveConf record exists in the database based on the configuration description
    #select command returns the last entry in the table that matches the description
    select_statement = """
    SELECT TOP 1 mmWaveSensorLocID from mmWaveSensorLoc
    where mmWaveSensorLoc.mmWaveSensorID='%d' and mmWaveSensorLoc.mmWaveSensorConfID='%d'
    ORDER BY mmWaveSensorConfID DESC
    """ % (mmWaveSensorLoc_record[0], mmWaveSensorLoc_record[-1])

    try:
        cursor.execute(select_statement)
        res = cursor.fetchone()
        # if location exists return the primary key
        if (bool(res)):
            print("Location for mmWave sensors with ID: '%s' for configration with ID: '%s' exists in the Database with ID '%s' updated" %
(mmWaveSensorLoc_record[0], mmWaveSensorLoc_record[-1], res[0]))

            update_statement = """
            UPDATE mmWaveSensorLoc
            SET mmWaveSensorLoc.x='%d', mmWaveSensorLoc.y='%d', mmWaveSensorLoc.z='%d', mmWaveSensorLoc.yaw_psi='%d',
mmWaveSensorLoc.pitch_theta='%d', mmWaveSensorLoc.roll_phi='%d'
            where mmWaveSensorLoc.mmWaveSensorID='%d' and mmWaveSensorLoc.mmWaveSensorConfID='%d'
            """ % (mmWaveSensorLoc_record[1], mmWaveSensorLoc_record[2], mmWaveSensorLoc_record[3], mmWaveSensorLoc_record[4],
mmWaveSensorLoc_record[5], mmWaveSensorLoc_record[6], mmWaveSensorLoc_record[0], mmWaveSensorLoc_record[-1])
            cursor.execute(update_statement)
            cursor.commit()
            return res[0]
        #else add the drone and return its primary key
        else:
            cursor.execute(insert_statement, mmWaveSensorLoc_record)
            cursor.execute("SELECT @@identity")
            res = cursor.fetchone()
            print(
                "Location for mmWave sensors with ID: '%s' for configuration with ID: '%s' added in the Database with ID '%s'" % (
                    mmWaveSensorLoc_record[0], mmWaveSensorLoc_record[-1], res[0]))

    except Exception as e:
        cursor.rollback()
        print(e.value)
        print('mmWave Sensor Location transaction rolled back')

    else:
        print('mmWave Sensor Location record inserted successfully')
        cursor.commit()

    return int(res[0])

def configureSensors(mmWaveSensorsConf,mmWaveSensors, conn, cursor):
    # check if a mmWaveSensors Configuration with the same Description and the same number of Sensors does not exist
    select_statement = """
    SELECT TOP 1 mmWaveSensorConfID from mmWaveSensorConf
    where mmWaveSensorConf.mmWaveSensorConfDesc='%s' and mmWaveSensorConf.mmWaveSensorConfNoOfSensors='%d'
    ORDER BY mmWaveSensorConfID DESC
    """ % (mmWaveSensorsConf,len(mmWaveSensors))

    cursor.execute(select_statement)
    res = cursor.fetchone()

    if not(bool(res)):
        #if the configuration does not exist create it and return the ID
        mmWaveConfRes = insert_mmWaveConf(mmWaveSensorsConf, mmWaveSensors, conn, cursor)

    else:
        #check if any of the sensor location has a different value
        tempConfiguration = []
        for mmWaveSensor in mmWaveSensors:

```

```

#check if the sensor exists
# check if the mmWave exists in the database based on its serialNumber
# select command returns the last entry in the table that matches the serial
select_statement = """
    SELECT TOP 1 mmWaveSensorID from mmWaveSensor
    where mmWaveSensor.mmWaveSensorSerial='%s'
    ORDER BY mmWaveSensorID DESC
    """ % mmWaveSensor[2]
cursor.execute(select_statement)
sensor = cursor.fetchone()

if not(bool(sensor)):
    mmWaveConfRes = insert_mmWaveConf(mmWaveSensorsConf, mmWaveSensors, conn, cursor)
    break
else:
    # check if the location record exists in the database based on the configuration description
    # select command returns the last entry in the table that matches the description
    select_statement = """
        SELECT TOP 1 * from mmWaveSensorLoc
        where mmWaveSensorLoc.mmWaveSensorID='%d' and mmWaveSensorLoc.mmWaveSensorConfID='%d'
        ORDER BY mmWaveSensorConfID DESC
        """ % (sensor[0], res[0])
    cursor.execute(select_statement)
    sensorLoc = cursor.fetchone()
    if not (bool(sensorLoc)):
        mmWaveConfRes = insert_mmWaveConf(mmWaveSensorsConf, mmWaveSensors, conn, cursor)
        break
    else:
        mmWaveSensorLoc_record = (mmWaveSensor[3], mmWaveSensor[4], mmWaveSensor[5], mmWaveSensor[6], mmWaveSensor[7],
mmWaveSensor[8], res[0])
        isSame = (sensorLoc[2]==mmWaveSensorLoc_record)
        if not(isSame):
            mmWaveConfRes = insert_mmWaveConf(mmWaveSensorsConf, mmWaveSensors, conn, cursor)
            break
        else:
            tempConfiguration.append((int(sensor[0]), int(sensorLoc[0]), int(res[0])))

    else:
        print ("No change in the Sensor configuration was done. Database not updated")
        mmWaveConfRes = tempConfiguration

return mmWaveConfRes

def insert_measurement(measurement_record,conn,cursor):
    insert_statement = """
        INSERT INTO measurement (measurementDatetime, experimentID)
        VALUES (?,?)
        SELECT @@IDENTITY AS [Last-Inserted Identity Value]
    """

    try:
        cursor.execute(insert_statement, measurement_record)
        cursor.execute("SELECT @@identity")
        res = cursor.fetchone()

    except Exception as e:
        cursor.rollback()
        print(e.value)

    else:
        cursor.commit()

    return int(res[0])

def insert_imuMeasurement(imuMeasurement_record,conn,cursor):
    insert_statement = """
        INSERT INTO imuMeasurement (acc_x, acc_y, acc_z, gyr_x, gyr_y, gyr_z, mag_x, mag_y, mag_z, imuSensorID, measurementID)
        VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
        SELECT @@IDENTITY AS [Last-Inserted Identity Value]
    """

    try:
        cursor.execute(insert_statement, imuMeasurement_record)
        cursor.execute("SELECT @@identity")
        res = cursor.fetchone()

    except Exception as e:
        cursor.rollback()
        print(e.value)

    else:
        cursor.commit()

    return int(res[0])

def insert_mmWaveMeasurement(mmWaveMeasurement_record,conn,cursor):
    insert_statement = """
        INSERT INTO mmWaveMeasurement (distance, azimuth, elevation, x_meas, y_meas, z_meas, mmWaveSensorID, measurementID)
        VALUES (?, ?, ?, ?, ?, ?, ?, ?)
        SELECT @@IDENTITY AS [Last-Inserted Identity Value]
    """

    try:
        cursor.execute(insert_statement, mmWaveMeasurement_record)

```

```

        cursor.execute("SELECT @@identity")
        res = cursor.fetchone()

    except Exception as e:
        cursor.rollback()
        print(e.value)

    else:
        cursor.commit()

    return int(res[0])

```

4.6 MATLAB Code for accessing the database

```

% Download the JDBC driver from here
% https://learn.microsoft.com/en-us/sql/connect/jdbc/download-microsoft-jdbc-driver-for-sql-server?view=sql-server-ver16
% Extract it a folder and find the location of the jar file
% Provide the location of the jar file in the opts below
% in SQL management studio go to Expand your database Expand the "Security" Folder Expand "Users" Right click the user (the one
that's trying to perform the query) and select Properties.
% Select page Membership.Make sure you check all except db_denydatareader and db_denydatawriter
clear all
addpath('ekfukf');
close all
vendor = "Microsoft SQL Server";
opts = databaseConnectionOptions("jdbc",vendor);
opts = setoptions(opts, ...
    'DataSourceName','thesisDB', ...
    'JDBCdriverLocation','C:\Users\mrpopoulos\Documents\sqljdbc_12.2.0_enu\sqljdbc_12.2\enu\mssql-jdbc-12.2.0.jre8.jar', ...
    'DatabaseName','thesisDB','Server','192.168.30.109\SQLEXPRESS', ...
    'PortNumber',49170,'AuthenticationType','Server','trustServerCertificate','true');
saveAsDataSource(opts)

username=".....";
password=".....";
datasource = "thesisDB";

%Accuracy of the ranging sensors
MeasureNoiseVariance = [2.98e-03, 2.9e-03,1.8e-03, 1.2e-03,2.4e-03];

%Accuracy of the imu sensors
Accelerate_Variance = [3.9e-04,4.5e-4,7.9e-4];
Accelerate_Bias_Variance= [1.9239e-7, 3.5379e-7, 2.4626e-7];
Gyroscope_Variance = [8.7e-04, 1.2e-03, 1.1e-03];
Gyroscope_Bias_Variance = [1.3111e-9, 2.5134e-9, 2.4871e-9];

StaticBiasAccelVariance = [6.7203e-5, 8.7258e-5, 4.2737e-5];
StaticBiasGyroVariance = [2.2178e-5, 5.9452e-5, 1.3473e-5];

% Initial guesses for the initial position
Position_init = [20;100;-1.9];

conn = database(datasource,username,password);
%choose here the experiment entry that corresponds to the loaded (to SQL)
%data.

%Run once to get the configuration
selectquery = 'SELECT TOP 1 * FROM measurement ORDER BY measurementID DESC';
lastMeasurement = select(conn,selectquery)

measurement_ID=lastMeasurement.measurementID(1)-1
experimentID=lastMeasurement.experimentID;

selectquery = 'SELECT * FROM experiment where experimentID=' + string(experimentID);
mmWaveSensorConfiguration = select(conn,selectquery);
imuConfID=table2array(mmWaveSensorConfiguration(:, 'imuConfID'));
mmWaveSensorConfID=table2array(mmWaveSensorConfiguration(:, 'mmWaveSensorConfID'));
selectquery = 'SELECT * FROM mmWaveSensorLoc WHERE mmWaveSensorConfID='+string(mmWaveSensorConfID);
locations = select(conn,selectquery);
anchorPositions=table2array(locations(:,['x' 'y' 'z']));
configuration =
initializeEKF(anchorPositions,Accelerate_Variance,Accelerate_Bias_Variance,Gyroscope_Variance,Gyroscope_Bias_Variance,MeasureNoiseVar
iance,StaticBiasAccelVariance,StaticBiasGyroVariance,Position_init);
X_ekf=configuration.X;
P_ekf=configuration.P;

```

```

X_ukf=configuration.X;
P_ukf=configuration.P;
Q=configuration.Q;
R=configuration.R;
dX=configuration.dX;

global UKF;
UKF=configuration;

previousTime=datetime(lastMeasurement.measurementDatetime);
previousTime.Format=previousTime.Format+ ".SSSSS";

while (1)
    pause (1)

    selectquery = 'SELECT TOP 1 * FROM measurement ORDER BY measurementID DESC';
    lastMeasurement = select(conn,selectquery);
    newTime=datetime(lastMeasurement.measurementDatetime);
    newTime.Format=newTime.Format+ ".SSSSS";
    dt=seconds(newTime-previousTime);

    previousTime=newTime;
    measurement_ID=lastMeasurement.measurementID(1)-1;
    selectquery = ['SELECT * FROM mmWaveMeasurement WHERE measurementID=' num2str(measurement_ID)];
    mmWaveMeasurements = select(conn,selectquery);
    mmWaveSensorIDs=table2array(mmWaveMeasurements(:, 'mmWaveSensorID'));

    measurementAndLocationsTable=join(locations,mmWaveMeasurements, 'LeftKeys',2, 'RightKeys',5);
    sensorPositions=table2array(measurementAndLocationsTable(:,["x" "y" "z"]));
    azimuth=table2array(measurementAndLocationsTable(:, "azimuth"));
    elevation=table2array(measurementAndLocationsTable(:, "elevation"));
    distance=table2array(measurementAndLocationsTable(:, "distance"));
    selectquery = ['SELECT * FROM imuMeasurement WHERE measurementID=' num2str(measurement_ID)];
    imuMeasurement = select(conn,selectquery);
    acc= [imuMeasurement.acc_x imuMeasurement.acc_y imuMeasurement.acc_z];
    gyr= [imuMeasurement.gyr_x imuMeasurement.gyr_y imuMeasurement.gyr_z];
    mag= [imuMeasurement.mag_x imuMeasurement.mag_y imuMeasurement.mag_z];

    nonZeroDistanceIndices=find(distance~=0);
    mmWaveSensorIDs=mmWaveSensorIDs(nonZeroDistanceIndices);

    rangingVector=distance(nonZeroDistanceIndices);
    if isempty(mmWaveSensorIDs)
        anchorsIndices=[];
    else
        anchorsIndices=find(ismember(table2array(locations(:, 'mmWaveSensorID')),mmWaveSensorIDs));
    end

    UKF=[];
    UKF.AnchorPcs=length(rangingVector);
    UKF.AnchorPosition=configuration.AnchorPosition(:,anchorsIndices);
    UKF.R=diag(configuration.MeasureNoiseVariance(anchorsIndices));
    UKF.Q=configuration.Q;

    [X_ekf,dX,P_ekf]=runEKF(X_ekf,P_ekf,dX,dt,UKF,acc,gyr,rangingVector);
    [X_ukf,P_ukf]=runUKF(X_ukf,P_ukf,dt,acc,gyr,rangingVector);
    position_ekf=X_ekf(1:3)
    position_ukf=X_ukf(1:3)
    trilaterationXYX = triangulate(rangingVector')
end

```

5 References

- A. Bourdoux; et al. (2020). *6g white paper on localization and sensing*. Retrieved May 2023, from <https://arxiv.org/abs/2006.01779>
- Han, Y., Shen, Y., Zhang, X., Win, M. Z., & Meng, H. (n.d.). "Performance Limits and Geometric Properties of Array Localization," in *IEEE Transactions on Information Theory*, vol. 62, no. 2, pp. 1054-1075, Feb. 2016, doi: 10.1109/TIT.2015.2511778.
- Hao, Z; et. al. (2022). Millimetre-wave radar localization using indoor multipath effect,. *Sensors*, 22, 5671.
- Laoudias, C., Moreira, A., Kim, S., Lee, S., Wirola, A., & Fischione, C. (2018). A Survey of Enabling Technologies for Network Localization, Tracking, and Navigation. *IEEE Communications Surveys & Tutorials*, 20(4), 3607-3644.
- Saily, M; et. al. (2021). Positioning technology trends and solutions toward 6g. *IEEE 32nd Annual International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*, (pp. 1-7).
- T, Wild; et. al. (2021). "Joint design of communication and sensing for beyond 5g and 6g systems. *IEEE Access*, 9(30), 845-857.
- Wang, D., Fattouche, M., & Zhan, X. (2019). Pursuance of mm-Level Accuracy: Ranging and Positioning in mmWave Systems. *IEEE Systems Journal*, 13(2), 1169-1180.
- Yu, S., & Al., e. (2018). A Low-Complexity Autonomous 3D Localization Method for Unmanned Aerial Vehicles by Binocular Stereovision Technology. *10th International Conference on Intelligent Human-Machine Systems and Cybernetics (IHMSC)*, , (pp. 344-347).
- Zafari, F., Gkelias, A., & Leung, K. (2019). A Survey of Indoor Localization Systems and Technologies. *IEEE Communications Surveys & Tutorials*, 21(3), 2568-2599.
- Zhao, Y., Liang, J. C., Sha, X., & Li, W. (2020). Adaptive 3D Position Estimation of Pedestrians by Wearing One Ankle Sensor. *IEEE Sensors Journal*, 20(19), 11642-11651.