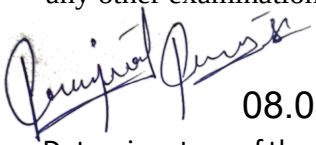
Master Thesis

Comparison of Machine Learning and Classical Methods for Real-Time Robust Ultrasonic Distance Measurement

Submitted by:
Athkar Praveen Prajwal
Matriculation no: 1394663
First examiner: Dr. Andreas Pech
Second examiner: Dr. Peter Nauth
Date of start: 11 March 2024
Date of submission: 08 August 2024

Statement

I confirm that I have written this thesis on my own. No other sources were used except those referenced. Content which is taken literally or analogously from published or unpublished sources is identified as such. The drawings or figures of this work have been created by me or are provided with an appropriate reference. This work has not been submitted in the same or similar form or to any other examination board.



08.08.2024

Date, signature of the student

Content

1. Introduction

The global sweep of Machine Learning (ML) during the past few years has been nothing short of astounding. From the largest of enterprises in banking, robotics, automotive, or home automation to small ones have been riding on advancements into this space. Today, ML is applied to the real world in a dazzling array of products: from search engines and credit card fraud prevention tools to mobile phones with face recognition capabilities and deep learning systems that have one hand (or wing) on algorithms for bending data into knowledge about cellular cancer treatments. In terms of results, healthcare is one field where machine learning has demonstrated to have a broader positive impact for the applications such as personalized medicine based on individual DNA and genomic structure or in predictive diagnostics through checking patterns from large medical image recognition using deep neural networks, automatic managing old aged handwritten cards into digitalized format (Tessaro et al., 2020)(Bode et al., 2020).

The real-time ultrasonic distance measurement field has evolved by conjugation of either traditional and machine learning-based technique. These include established classical methods providing original and straightforward solutions to both filtering noise (e.g. Kalman filter) and accurate distance estimation problems using thresholding. While these methods still work, they can have a hard time adjusting and keeping pace with (dynamically) changing environments. On the other hand, machine learning techniques like Recurrent Neural Networks (RNN), Long Short-Term Memory (LSTM) networks, Gated Recurrent Units (GRU), and Bi-directional LSTM (BiLSTM), may be more successful in capturing complex temporal dependencies and non-linear interactions among data (Aishin, 2024). This research strives to explain a detailed comparative study of these conventional approach in the scope to enable state-of-the-art real-time robust ultrasonic distance measurement and the machine learning approach for occupancy detection in smart office environments. This comparison is an attempt to identify the strategies that are most effective in each scenario, where high accuracy, precision, robustness, and reliability are considered and incorporated in both the approaches.

We must also consider that thresholding is an important approach in ADC signal first peak detection due to its capability to separate between noise and meaningful signal variations. In an ADC signal, the signals are frequently accompanied by a variety of noise sources that can suppress true signal peaks. Through the use of thresholds, one may efficiently filter out small changes and arbitrary noise that do not reflect actual signal peaks by setting a precise signal amplitude position that must be surpassed in order for a peak to be identified. This guarantees that only significant changes in the signal, corresponding to real peaks, are detected, making peak detection more accurate and reliable by selecting an appropriate threshold point. As a result, it is a reliable system for applications such as instrumentation systems, communication systems, and signal processing in the automotive and healthcare industries that require precise signal analysis. Furthermore, by lowering the amount of false positives and the computational cost, thresholding reduces the processing burden involved with analyzing every little change in the signal. This technique allows for faster and more exact detection of the initial peak, which is typically required for performing subsequent signal processing operations and correctly interpreting data.

In chapter 2, the theoretical background is explained along with the components used in the project. This section also has the literature research of the project. Chapter 3 deals with the requirements analysis along with clarification of what all modifications are to be done in the current script in order to make the project a working model. Initial state, target prototype and use case prototype is also explained in this section. Chapter 4 deals with realisation of the project. In this section evaluation results of both the models are discussed. Finally, all the previous chapters are concluded in the final chapter 5 namely of summary and perspectives.

1.1 Purpose of Research

Machine learning applied to real-time ultrasonic distance measurement attempts to optimally improve the efficiency, robustness, and accuracy of distance measurement systems that are used for a broad data acquisition and data analysis applications in industrial and scientific interests. At present, the conventional ultrasound distance pricing methods, like time-of-flight and phase-shift technologies, still have big limitation in noise, signal distortion and environmental problems. Built and measured using classical methods consisting of thresholding algorithms and noise filtering algorithms, this technique allows for faster and more exact detection of the initial peak, which is typically required for performing subsequent signal processing operations and correctly interpreting data. The classical methods that are subject to the practical realities of fallible measurements and must be considered when creating a classical model. And, the machine learning method also provide a powerful way to correct for

heterogeneous, sparse and noisy data being spread from high-throughput sensors, which significantly generate valuable information in signal processing and measurements.

In this research, the purpose is to investigate the advantages of using machine learning techniques like Recurrent Neural Networks (RNN), Gated Recurrent Units (GRU), Long Short-Term Memory (LSTM) and Bidirectional Long Short-Term Memory (BiLSTM) networks to surpass the barrier of classical methods or not. The findings in this study will then be exploited to identify the most accurate and precise methods for distance measurement in a broad range of conditions, and can be further used to enhance the design of the ultrasonic measurement systems for real-time applications (Hoang et al., 2019), (Gao et al., 2020), (Noh, 2021).

1.2 Scope of the Proposed Classical Method Solution

The solution is to improve the Tkinter application reader to process and display the ADC signal, that is read from an ADC, acquired with a Red Pitaya sensor. This application also incorporates a powerful signal processing pipeline, which implements Kalman filtering for smoothing signals and provides sophisticated peak detection, including Yanowitz-Bruckstein and Niblack's local thresholding. It calculates the distance using detected peak with respect to the environmental temperature and the sampling frequency. The improvements include an interface for dynamic and interactive plotting using matplotlib, a toolbar to zoom and save figures, and a simple function to manage the graphical display. The application enables users to modify signal processing parameters, record data, and view processed signals with detected peaks and corresponding distance range parameters. It is a great asset to learn ultrasonic signal analysis and distance estimation in real-time, useful for academic study, research or a type of sensing solution for Red Pitaya Sensors as it can work seamlessly with the hardware.

2. Theoretical Background

The theoretical background and the required technologies are content of this chapter.

2.1 Red Pitaya based Ultrasonic Sensor

Red Pitaya is a flexible and affordable microchip STEM Lab board that generates and measures analog signals. It is sometimes referred to as the Swiss Army Knife for engineers. With two high-precision 14-bit analog-to-digital converters and two 125 MS/s analog-to-digital converters, it combines analog and digital input/output capabilities. With its powerful dual-core CPU and many connectivity choices (Wi-Fi, USB, and Ethernet), the platform can handle a wide range of signal processing and measurement tasks. Red Pitaya's dual fast ADCs and DACs allow two sine waves with any phase difference to be produced and acquired simultaneously. Running Linux, the device may be accessed from PCs, tablets, and other devices via a web server, USB-serial terminal, or SSH protocol. Its open-source hardware and software, coupled with an easy-to-use web interface, enhance data visualization and remote control. Additionally, the board integrates an ARM processor, an FPGA, ADCs, DACs, and an ultrasonic sensor, enabling it to serve as a comprehensive lab equipment replacement for sensor signal processing and classification using trained network algorithms (Arnaldi, 2017).

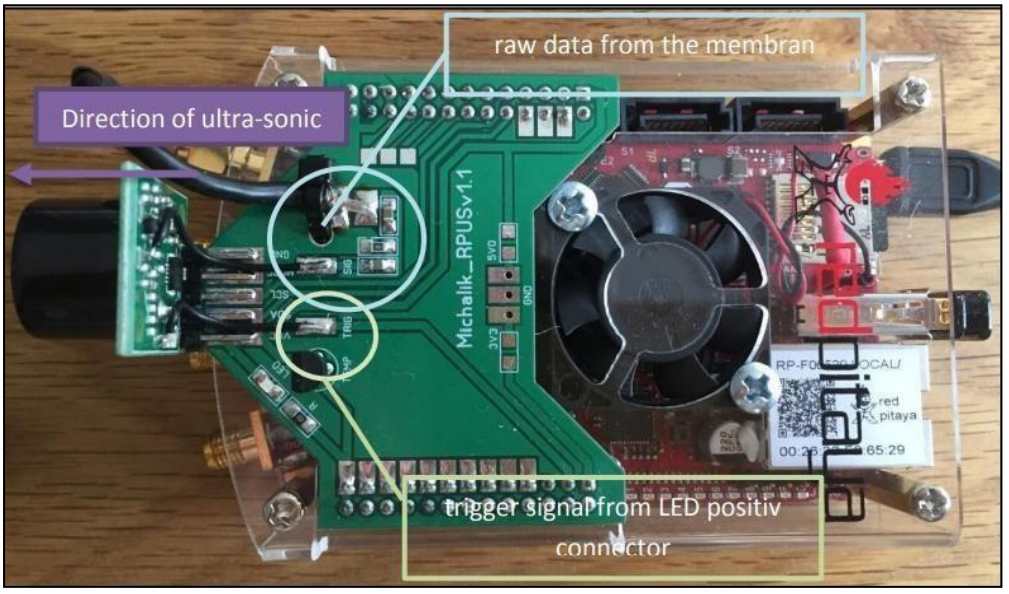


Figure 2.1: Red Pitaya based ultrasonic sensor (Tiniya, 2023).

2.1.1 Ultrasonic sensor

An ultrasonic sensor is a technical instrument that measures the distance to an object using ultrasonic waves. The sensors work by emitting sound waves at high frequencies by a transducer, generally around 20 kHz or higher, in the direction of a particular object; upon being reflected, the waves return to the sensor. A sensor counts the time that the sound signal needs to reach the object and returns the beams, using these values to calculate the distance based on the speed of sound. Usually, the ultrasonic sensors are used for non-contact distance measurement-for example, in object detection. They are also used for level measurement in tanks and proximity sensing in industrial automation. They are perfect for this latter application because they can work in several environmental conditions in which optical sensors just will not work: dust, light, and high temperature (Pech et al., 2019).

An SRF02 Ultrasonic Sensor is a versatile, compact device generally applied in robotics and automation for distance measurement and object detection. The SRF02, producing an ultrasonic wave and receiving its echo, can detect a distance ranging from 15 cm to 2.5 m. Special applications for this sensor include proximity detection for mobile robots, sensing levels in tanks, and security systems. It has a simple interface, can work in dirty conditions, and is compatible with most microcontrollers, making the SRF02 a very reliable and cheap solution in most industrial and hobby applications. This, coupled with low power consumption and high sensitivity, assures efficient and reliable operation, which makes it further preferable for professional and educational settings.

Equipped with these, the SRF02 is the most cost-effective available in the range finder market with ultrasonic. An exciting feature that the SRF02 has is the ability to independently send ultrasonic bursts, hence saving the need for a dedicated reception cycle. A reception cycle can also be well conducted independent of a preceding burst. These sensors calculate the distance to a target by timing the intervals between the emission and reception of ultrasonic pulses. Ultrasonic sensors are also suitable for detecting transparent objects. They may detect objects of various color, surface texture, and material composition except those of very soft substances like wool, which absorb sound. Ultrasonic sensors are, therefore, always considered the ideal option when optical technologies may be having difficulty detecting transparent or other challenging materials or objects (Hosur et al., 2016).

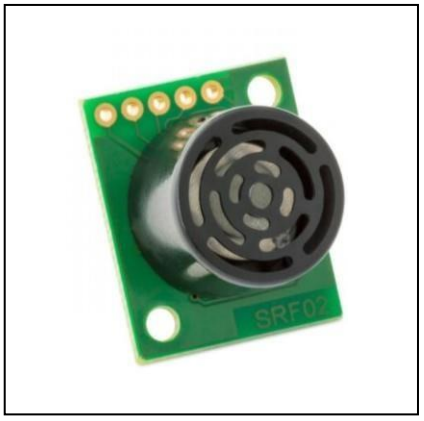


Figure 2.2: Ultrasonic sensor SRF02.

2.1.2 Ultrasonic Data Collection Software

The Frankfurt University of Applied Science Master of Information Technology Engineering students developed an Ultrasonic Data Collection Software that measured both raw and analogue data from the ultrasonic sensor (Schäfer, 2020). The collected data is arranged into rows with amplitude values ranging from 0 to 1000 and columns with frequency values between 34.9 kHz and 44.9 kHz. Figure 2.6 displays the software's user interface. The 85 columns in the created dataset are thoughtfully spaced at intervals of 1.08 to 1.09 kHz for the frequency values. Beyond only collecting data, the application has the ability to use the Fast Fourier Transform (FFT) to evaluate the data that has been gathered. The exported data also includes data headers that include crucial details including data length, an embedded model's classification results, and sampling frequency (Pongsomboon, 2022). When a classification model is present, it means that the program can categorize or label collected data based on predefined standards. Class 1 refers to objects, whereas class 2 refers to people. As can be seen by the volume of information in the output data headers, this program offers a good solution for ultrasonic data analysis by fusing technical features like FFT and categorization with straightforward user interfaces via the GUI making it user-friendly.

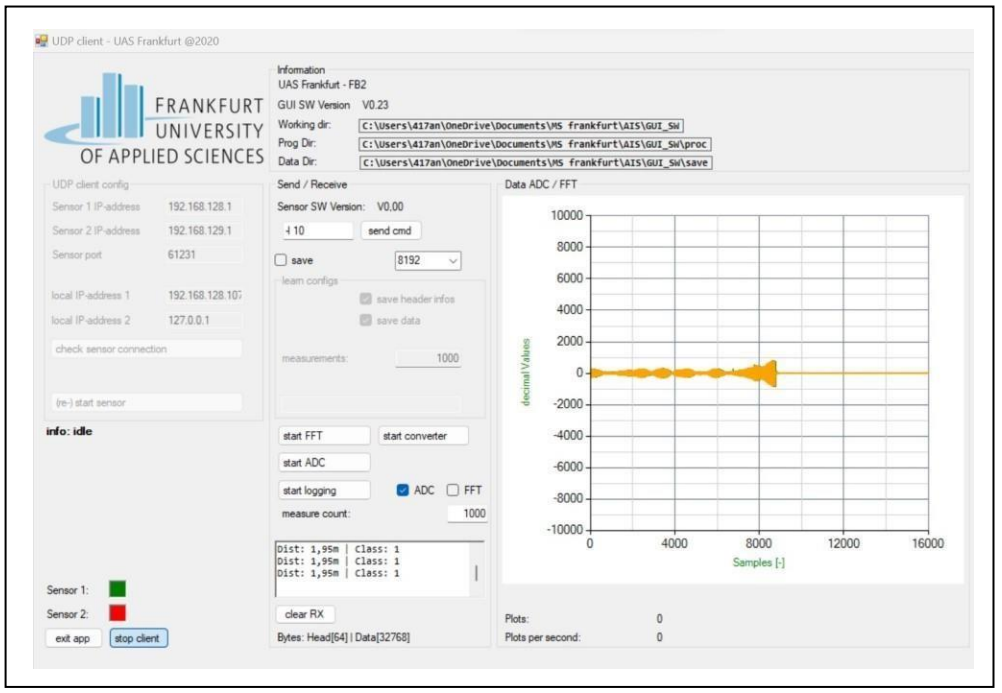


Figure 2.3: GUI of Ultrasonic Data Collection Software.

2.1.3 Working Methodology of Ultrasonic Sensors

Sound waves have different velocities depending upon the medium through which they are traveling. In general words, the density and stiffness of the medium cause a difference in the speed of sound. The speed of the sound varies depending on the physical characteristics of the environment (Hosur et al., 2016).

The sensor ultrasonic pulse echo method then shows in Figure 2.4. The sensor's transducer emits an ultrasonic pulse which propagates from the transducer and bounces off the object it hits. The transducer will receive the reflected signal and the received signal can be processed, for instance, measuring the distance of the object from the transducer (in the case of ultrasonic sensors, derived from the flight of time - time that ultrasonic pulse takes to travel from the transducer back to it after reflection) (Pech et al., 2019).

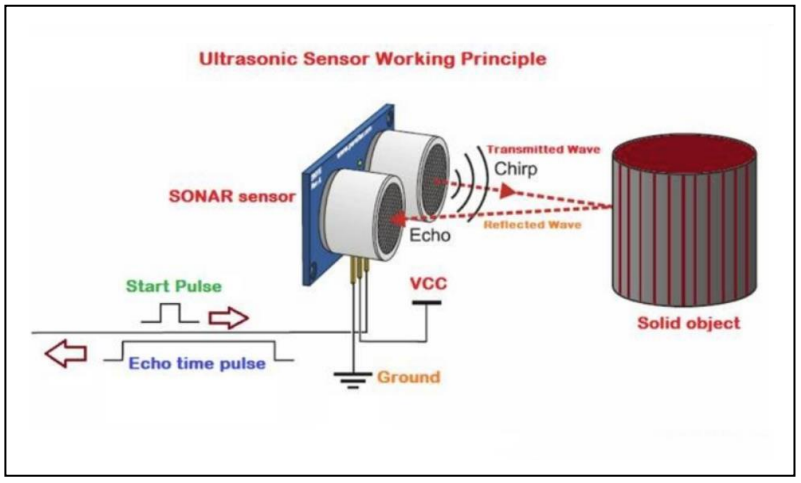


Figure 2.4: Working principle of ultrasonic sensor (DFRobot, 2023).

2.1.4 Advantages of Ultrasonic Sensor

Infrared sensors can be interfered with by all kinds of dust, smoke, fog, and other particles that obstruct vision expecting knowledge of the surface of reflection. But, none of these things affect ultrasonic sensors. Optical-based sensors find it difficult to detect water and glass and hence they are not ideal to be used in environments having smoky, foggy, and bright ambient lighting environments. Although they reflect off water and glass, ultrasonic waves are detectable. RADAR and LIDAR address these concerns by offering high precision but are extremely costly. Ultrasonic sensors are already used in parking and driving assistance systems as they are much cheaper than camera or radar sensors. Therefore, adding yet another sensor to this array is much more feasible and practical (Pech et al., 2019).

2.2 Kalman Filter

The Kalman filter was an invention of mathematics for state estimation of linear dynamic systems under uncertain or polluted measurements. It is widely used in many fields, such as engineering, robotics, economics, and others, that require some kind of prediction or estimation based on noisy data. It is a potent estimation technique, developed by Rudolf Kalman in the 1960s, in that it yields the optimum estimates by combining past measurements with a mathematical system model (Mehra, 1971). Below is the detailed explanation of the Kalman filter model:

- **State Space Representation:** The state of the system is represented by the Kalman Filter, which is a collection of parameters adequate to describe the system in its whole at any given moment. Values such as position, velocity, orientation, and so forth may be contained in the state. The state thus varies with time by several linear equations (Simon, 2010).

- **System Dynamics:** The system dynamics describe how the system's current state will change in the future, given its present state. Typically, this is represented by a linear dynamic system in the form of state transition equations (Rigatos, 2012).

- **Observation Model:** The system's state is never observed; instead, noisy measurements are taken from the sensors. The observation model describes how the measurements depend on the system's state. Once again, this is typically expressed as a linear equation (Revach et al., 2021).

- **Noise Models:** Assume that both the system dynamics and the observation process are corrupted by noise. For a concrete approximation, the noise is often modeled with Gaussian properties with known statistics (Mariani & Ghisi, 2007).

2.2.1 Kalman Filter Steps

- **Initialization:** The state and covariance (uncertainty) of the state are the initial estimates the Kalman Filter requires at the start. These might originate from prior knowledge of, or assumptions about, the system being considered.

- **Prediction:** Since we are considering our belief about the natural state and how it changes, the Kalman Filter predicts the next state and the error covariance. It simply predicts the state from the previous one together with the covariance of this prediction. This operation is a predictable operation; in that way, it allows factorization and approximation and is thus available in linear projection form concerning the state applied in the system dynamics.

$$x\hat{}_k = F_k x\hat{}_{k-1} + B_k u_k$$

Equation 2.1

$$P_k = F_k P_{k-1} F_k^T + Q_k$$

Equation 2.2

Where: $x\hat{}_k$ is the predicted state at time k, F_k is the state transition matrix, B_k is the control input matrix, u_k is the control input (if applicable), P_k is the predicted covariance matrix, Q_k is the covariance of the process noise.

- **Correction (Update):** Here, the Kalman Filter combines the predicted state with the actual measurement, applying their relative uncertainties, and computes a Kalman Gain for determining how much the prediction shall be corrected based on exact measurement. Then, it updates the state estimate accordingly.

$$K_k = P_k H_k^T (H_k P_k H_k^T + R_k)^{-1}$$

Equation 2.3

$$\hat{x}_k = \hat{x}_k + K_k(z_k - H_k \hat{x}_k)$$

Equation 2.4

$$P_k = (I - K_k H_k) P_k$$

Equation 2.5

Where: K_k is the Kalman Gain, H_k is the observation matrix, R_k is the covariance of the measurement noise, z_k is the actual measurement, $\hat{x}_k$ is the updated state, and P_k is the updated covariance.

2.2.2 Advantages

The Kalman Filter accomplishes a linear unbiased estimate of the state by considering that the measurements are affected by noise. It is computationally efficient, especially for systems with large state spaces. It is strong against uncertainties and noise, hence robust in real-life applications. A Kalman Filter can adapt itself to both changes in the system dynamics or changes like the noise (Bai et al., 2013).

2.2.3 Limitations and Considerations

The Kalman Filter assumes the system dynamics and observation model to be linear. For nonlinear systems, extensions like the Extended Kalman Filter (EKF) or Unscented Kalman Filter (UKF) are used. The accuracy of the Kalman Filter relies on the accuracy of the system model and the noise assumptions, with deviations from such assumptions potentially impacting performance. While the Kalman Filter is efficient, it can become computationally demanding for extensive state spaces or high update rates (Kuti et al., 2013).

2.2.4 Applications

Such applications include estimation for navigation systems, which estimate the positions and velocities of vehicles using GPS and inertial sensors; control systems for evaluating the state of a control system so that one can make real-time adjustments; tracking and localization for tracking moving objects using sensor measurements, and economic forecasting of economic variables based on the noisy data (Auger et al., 2013).

2.3 Niblack's Local Thresholding Algorithm

Niblack's method is a well-known image processing local thresholding algorithm for image binarization processes. It is designed to be capable of adjusting local contrast and illumination changes in an image by setting the threshold of the pixel depending on its neighborhood. This provides local adaptability for binarization but differs from global threshold methods, where only one threshold is imposed on the entire image. This is thus more robust for documents and images where lighting is not even, as it can adapt to local changes in illumination and contrast. The level of detail and noise handling can also be controlled because k, which is the sensitivity parameter, is used. This allows the algorithm corresponding to various image natures to be adjusted instantly (Niblack, 1986).

2.3.1 Methodology

- **Local neighborhood with Mean and Standard Deviation calculation:** At each pixel in an image, the algorithm looks in a local neighborhood of size $w \times w$ centered around the pixel. The size of this window may drastically change the output: smaller windows would be noisier, and large windows could blur critical structural details. And, within each window, the local mean (m) and standard deviation (s) of the pixel intensity values are computed.

- **Threshold Calculation:** The threshold T for each pixel is calculated using the below formula, where k is a user-defined parameter that influences the sensitivity of the thresholding:

$$T = m + k \cdot s$$

Equation 2.6

- **Binarization:** The pixel is then compared to the threshold T. If the pixel intensity is greater than T, it is set to white (or 1 in a binary image); otherwise, it is set to black (or 0).

2.3.2 Applications

- **Document Image Binarization:** Niblack's method is widely used in preprocessing steps for OCR (Optical Character Recognition), where maintaining the integrity of text is crucial. Improvements often involve combining it with other techniques to handle varying conditions better (Nasri et al., 2018).

- **Image Segmentation:** The method is also applied in various image segmentation tasks, especially where the object boundaries need to be accurately detected under varying lighting conditions (Wang et al., 2008).

- **Noise Reduction:** Combining Niblack's method with noise reduction techniques, such as Adaptive Wavelet Thresholding, can improve results in noisy images (Kurniadi et al., 2020).

- **Enhanced Variants:** Several studies propose modifications to Niblack's algorithm to improve its performance. For instance, adaptive methods to determine the parameter k based on local image characteristics have been suggested (Saddami et al., 2017).

2.4 Yanowitz-Bruckstein Adaptive Thresholding

Adaptive thresholding is an essential tool in the field of image processing. Several authors, including Yanowitz and Bruckstein, considered a more sophisticated way for adaptive thresholding, considering both spatial and intensity information from the image. In contrast to global thresholding methods using one threshold value for the whole image, the Yanowitz-Bruckstein method adapts the threshold value locally based on both the pixel intensity distribution and the local structure of the image. The method first identifies initial seed points in the image, which are used to construct an initial threshold surface. Seed points are manually selected around regions of high contrast with known background and foreground regions. The algorithm then refines that surface iteratively by threshold propagation through the image. Local intensity gradients perform the refinement. As the process deals with images having non-uniform illumination and variation in the background texture, the Yanowitz-Bruckstein algorithm is

considered a powerful tool for a variety of tasks, including document image analysis, medical image segmentation, and object detection in complex scenes (Yanowitz & Bruckstein, 1995).

2.4.1 Methodology

This algorithm becomes extremely useful with images where the background and the foreground cannot be thresholded with a single global threshold due to the natural variations in the lighting condition or non-uniform background. In this way, with local adaptation of threshold, more accurate separation of foreground from background can be done (e.g., objects of interest) with the Yanowitz-Bruckstein method.

- **Initialization:** A first guess of the threshold is made. It can be global, which means a single value is used for the whole image from the start, or it can be local, based on statistics in the image for initial guess values.

- **Iteration:** For each pixel or local region within the image, the method adjusts the threshold based on some local criterion. This way, the adjustment is kind of iterative; the algorithm goes several times through the image and refines the thresholding for this number.

- **Local Criterion:** The threshold's adaptation may be based on local image features such as the intensity values of the neighboring pixels. The adapted threshold, therefore, is non-invariant but ensures that the threshold gets more in line with the adaptiveness to local image content.

- **Convergence:** This process keeps repeating itself until the threshold values do not change significantly from one iteration to another or when a maximum number of iterations is reached.

2.4.2 Applications

The Yanowitz-Bruckstein algorithm has been applied and improved in numerous articles. For instance, Blayvas, Bruckstein, and Kimmel developed another method of creating the adaptive threshold surface based on a multiple-resolution approximation concept. The new process reduces the computational expense but maintains its smoothness and improves the performance in a wide range of illumination conditions (Blayvas et al., 2001). Another improvement by Yazid and Arof showed an easier and more accurate threshold surface, and the results being compared to other images, such as medical images and documents' images, demonstrated superiority in the performance on thresholding over the different algorithms on adaptive thresholding (Yazid & Arof, 2013).

2.5 Sonar Distance Calculation

The addition of the presented distance, calculated by sonar, depends heavily on environmental conditions and ultrasonic sensor operational parameters: especially ambient temperature and the sampling rate. As the sound speed is one of the underlying parameters for any sonar measurement, the speed in air differs with temperature and needs compensation to stay exact. As given by Jiao (2015), the sound speed v at temperature T (°C) is provided as $v \approx 331.45 + 0.606 \times T$ m/s. This necessitates the addition of the sonar system temperature sensors and compensation to adjust the calculated distance from the target at any given time depending on the temperature. The sample rate of the ultrasonic sensors is also significant in the accurate resolution of distances, more so when the environmental conditions are dynamic (Jiao, 2015).

Higher sampling rates improve the resolution and accuracy but at the price of a higher computational load. In such regard, Sahoo and Udgata (2020) proposed the idea that neural networks must be embedded to cater to dynamic adaptation in sampling rate and processing algorithms, taking into account changes in the environment to enhance measurement precision (Sahoo & Udgata, 2020). In practice, FPGA-based implementation systems can adaptively modify the speed of sound in measurements, thereby minimizing the measurement errors primarily at fluctuating temperatures. This has been shown by Gultekin et al. in their work (Gultekin et al., 2017). Such implementations allow for accurate distance measurements, essential for applications ranging from robotics to environmental monitoring. In other words, precise distance calculators by sonar must consider temperature-induced variability of the speed of sound and use appropriate high sampling rates, which can be handled with advanced sensor systems and adaptive algorithms.

2.6 Tkinter library

Tkinter is a powerful GUI library included with the standard Python distribution. It gives a platform-independent, object-oriented encapsulation of widgets to help developers create windows where they can put buttons, labels, and other controls using geometry managers (pack(), grid(), and place()).Simplicity and ease of use have made Tkinter one of the most popular choices among novice and experienced programmers working on cross-platform applications.

Creators based Tkinter on the widely used Tk GUI toolkit under UNIX-like systems, Windows, and macOS so that the development results can efficiently work on any of these platforms without modification. The fact that Tkinter is based on the Tk GUI toolkit, which is widely used under UNIX-like systems, Windows, and macOS, brings out clear evidence that applications designed with Tkinter can run on various platforms without any modifications (Kravets et al., 2020). Ease of use and integration with other useful Python libraries make it the most suitable choice for teaching GUI programming in Python within the educational context (Lee & Hubbard, 2015). Practical applications in Tkinter include the development of various domains' desktop applications. One may quote a Typing Speed Tester game, for instance, where this tk library provides users with a user-friendly interface to test and increase their typing speed (Gupta, 2021).

In the scientific community, Tkinter has been used to build a flexible and functional scientific application that interfaces the user with complex applications in atomic physics (Tahat & Tahat, 2011). It is also evident that Tkinter has been helpful for managing the operations of hotels, with it being employed at the front end in managing the hotel and connecting seamlessly above databases behind the scenes (Kurzadkar et al., 2022). More so, it permits the development of even complex data processing tools: processing and manipulation applications for PDF files, which are thus quite convenient and practical. Tkinter is generally distinctive with its simplicity, flexibility, and comprehensive functionality, meaning that Tkinter is a super solution for GUI application development in Python.

2.7 Performance Evaluation of the GUI Software

A GUI's performance is assessed by examining a number of factors to make sure it effectively satisfies user demands. The primary factors taken into account in this study were:

- In the context of GUI software, accuracy refers to how successfully the program performs its intended functions. This may be assessed by subjecting the program to rigorous testing and contrasting expected and actual outcomes under various circumstances.

- Precision in GUI software means the consistency and exactness of the software’s responses and actions and can be measured by running repeated tests and seeing if the software gives the same and detailed results every time.

- Robustness means the software can withstand unexpected conditions without crashing or giving incorrect results and can be tested by stress testing, introducing edge cases and seeing how the software handles them.

- Repetition rate of GUI software means how often the software can do repetitive tasks. This is important for tasks that require frequent user interaction or batch processing and can be measured by scripting repeated tasks and monitoring the software’s performance over time to ensure no latency or loss of responsiveness.

- Memory consumption means the amount of RAM used by the software during its operation and can be monitored using profiling tools to measure and analyze memory usage trends, to ensure the software stays within limits and doesn’t consume too many system resources.

- Real-time performance means the software can do tasks and respond to inputs in a reasonable time, which is important for a smooth user experience.

2.8 Deep learning in Occupancy detection

2.8.1 Overview and Approach

Their ability to learn high-level knowledge and generalize from data allows neural networks to form the main technology for occupancy detection and estimation in a smart office setting, where long-term temporal occupant presence patterns have been predicted with ample data. Therefore, traditional machine learning methods like random forests and support vector machines (SVM) are commonly used for this purpose when working with non-text data (Dong et al., 2018). Given new data, the trained models predict occupancy levels and trends using time series or point features from multiple sensors and informative sources of data provided as labeled datasets. Nonetheless, information is lost due to human-designed feature extraction in traditional machine learning techniques, which takes unnecessary time and energy (Schmidhuber, 2024).

Several challenges arise in the smart office building context with respect to occupancy detection and analysis. Deep learning, which is a subset of machine learning, provides an effective way for handling these. For example, traditional methods (e.g., column-based one-hot encoding for each category variable attribute and vectorization in NLP tasks by sequence operation) may require a lot of manual feature engineering work such as identity transformation that will be applied to numerical features, whereas deep learning strategies, including DNNs, can provide the capability to perform useful modeling based on raw data from various domains without those handcrafting (Zou et al., 2017). This makes deep learning's ability to mimic the patterns and underlying relationships in data more effective, especially where the sensor type is time-series like ultrasonic.

The focus is particularly on deep learning methods, specifically recurrent neural networks (RNNs), for processing sequential data derived from integrated ultrasonic sensors with the Red Pitaya controller. LSTMs and GRUs can model long-term dependencies within a sequence, which is well-suited for the temporal aspect of occupancy/non-occupancy analysis (Hochreiter & Schmidhuber, 1997).

An LSTM predictive model was developed that can be included with the software application of Red Pitaya for dynamic smart office regular detection. Long Short-Term Memory networks (LSTM; Greff et al., 2015) are a type of recurrent neural network commonly used in time-series analysis to store information over long sequence intervals, making them effective for modeling the complex temporal dependencies inherent with ultrasonic sensor data. Using labeled data from an auxiliary system, the LSTM model is trained to discover patterns in both occupancy behavior and occupants' absence or presence. After training, the LSTM model can be easily integrated into Red Pitaya's software to provide live occupancy detection and some level of prediction based on data from ultrasonic sensor monitoring managed through the Red Pitaya system (Aishin, 2024).

2.8.2 Recurrent Neural Network (RNN)

Recurrent neural networks (RNNs) are a kind of deep learning technology tailored for sequential data e.g., text, speech, time series, and sensor readings. RNNs, having a feedback connection, allows them to have an internal state/memory, giving them the ability to remember things and understand one input in context with this memory, unlike feed-forward neural networks (Mikolov et al., 2010). Recurrent Neural Networks (RNNs) add loops to these networks, allowing information to persist, which creates a form of memory in the network. This looping mechanism allows RNNs to capture similar dependencies and context in sequential data, thereby making them suitable for applications like occupancy detection, where prediction needs the identification of patterns and sequences in sensor readings. But traditional RNNs suffer from vanishing gradients, where the network is unable to learn temporal dependencies of data over long time scales. The former issue arises due to the repeated multiplication of weight matrices during backpropagation, which can lead gradients either toward zero (vanishing), and then nothing is learned from long sequences, or towards infinity/NaN values, hence making the RNN to explode.

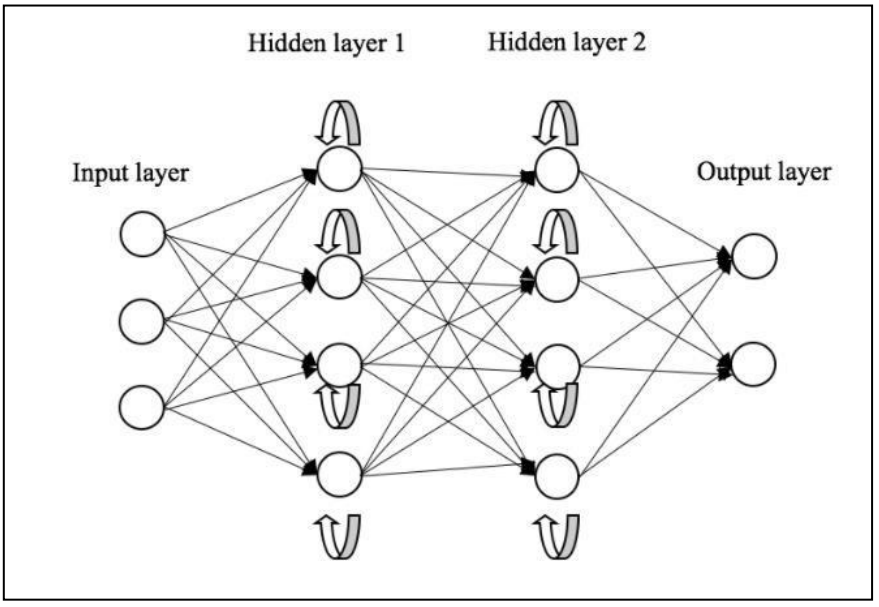


Figure 2.5: RNN architecture (Open Data Science, 2020).

2.8.3 Gated Recurrent Unit (GRU) Neural Network

RNNs handle well with the sequential data, and GRUs are a type of enhanced RNN structure. GRUs were introduced as a simplification of LSTM networks, designed to reduce complexity (Cho et al., 2014). They give the GRUs the ability to solve vanishing gradient and exploding gradient problems with long sequence dependencies needed, for example, in natural language processing models which involve audio analysis and time series evaluations. At the heart of GRUs is a gating mechanism on data flow in network connections. This mechanism has two different gatekeepers: a reset door and an update passage. The reset gate controls how much of the previous hidden state is forgotten, while the update gate dictates a function to enhance input and change the overall hidden state (Chung et al., 2014). Gate control in GRUs helps solve vanishing gradients encountered by ordinary RNNs so that long-term dependencies can be learned more effectively through sequences (Cho et al., 2014). Since GRUs are really good at exploiting the contextual information across sequential data (which we have said is a characteristic of natural language) they work well in problems that involve hyper-text data, like language modeling and machine translation or forecasting some time series. In simple terms, GRUs are a beefed-up version of RNN with gate components to determine the extent of relevance that needs to be passed. This makes them powerful in real-world domains for which sequential modeling and prediction of sequences are required, as they can learn temporal dependencies between sequences effectively, thus able to remember long-dependent memories.

- **Update Gate:** The update gate controls how much of the past information needs to be passed along to the future. It decides the extent to which the previous hidden state needs to be updated with the new hidden state (Guo et al., 2022).

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t])$$

Equation 2.7

- **Reset Gate:** The reset gate determines how much of the past information to forget. It controls the contribution of the previous hidden state to the candidate hidden state (Kuan et al., 2017).

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t])$$

Equation 2.8

- **Candidate Hidden State:** The candidate hidden state is a new hidden state that is computed using the reset gate. It represents the possible new state that combines the current input and the past state influenced by the reset gate (Lynn et al., 2019).

$$\sim h_t = tanh(W \cdot [r_t * h_{t-1}, x_t])$$

Equation 2.9

- **Final Hidden State:** The final hidden state at the current time step is computed by combining the previous hidden state and the candidate hidden state, weighted by the update gate (Kuan et al., 2017).

$$h_t = (1 - z_t) * h_{t-1} + z_t * \sim h_t$$

Equation 2.10

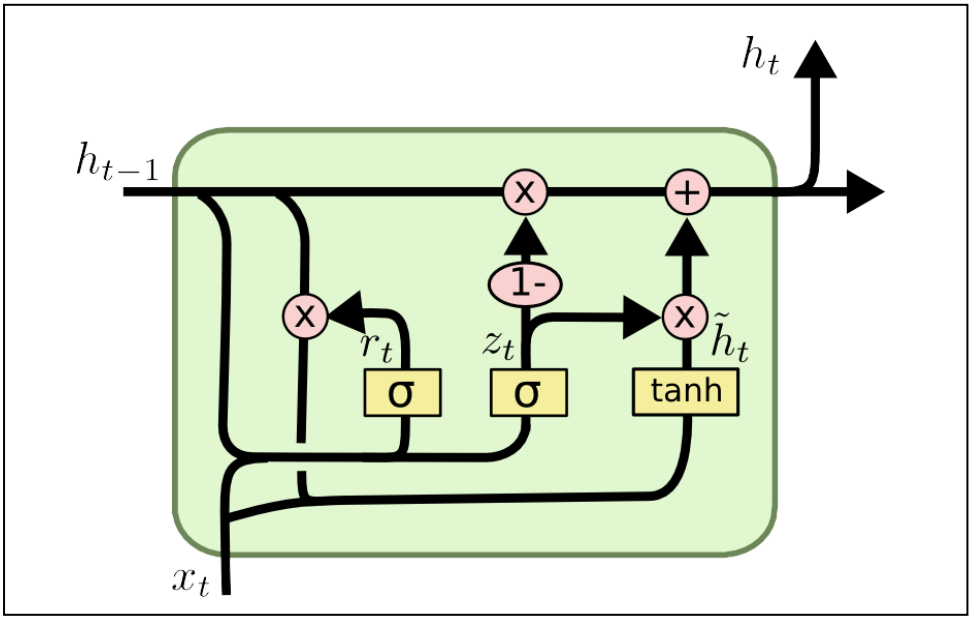


Figure 2.6: GRU architecture (Bibi et al., 2020).

2.8.4 Long Short-Term Memory (LSTM) Neural Network

Long Short-Term Memory is one other sort of RNN architecture that has been designed to tackle the problem in processing sequence data efficiently. It was devised to tackle the vanishing gradient problem that traditional RNNs encountered when trying to learn long-range dependencies (Hochreiter & Schmidhuber, 1997). LSTMs are great at handling sequential data, which makes them perfect for applications such as language processing, audio recognition, time-sequence analysis, and occupancy prediction (Khan et al., 2022).

LSTM networks have specialized units that are called memory cells. The activation of these cells is regulated by an input gate, forget gate, and output button in order to hold each memory unit until the time or permissible for considerable effect. Gates hold information in the cell and allow only certain things to flow into, out of, or through it. The mechanism of LSTM neural networks with the gates and cell states are explained below (Greff et al., 2017):

- **Initialization:** The initial cell state C_0 and hidden state h_0 are typically initialized to zero.

- **Gate Computation:** At each time step, the forget gate f_t, input gate i_t, and output gate o_t are computed using the current input x_t and the previous hidden state h_{t-1}.

- **Input Gate (i_t):** Manages the limitation of newly entered information to the cell.

$$i_t = \sigma(W_i \cdot (h_{t-1}, x_t) + b_i)$$

Equation 2.11

- **Forget Gate (f_t):** Specifies the portion of past information that will be thrown away.

$$f_t = \sigma(W_f \cdot (h_{t-1}, x_t) \ + \ b_f)$$

Equation 2.12

- **Output Gate (o_t):** Decides which information from the cell states will be carried to predict the next hidden state (h_t).

$$o_t = \sigma(W_o \cdot (h_{t-1}, x_t) \ + \ b_o)$$

Equation 2.13

- **Cell State Update:** The candidate cell state $\sim C_t$ is computed, and the final cell state C_t is updated using the forget gate and input gate.

$$\sim C_t = tanh(W_C \cdot [h_{t-1}, x_t] + \ b_C)$$

Equation 2.14

$$C_t = f_t * \ C_{t-1} + \ i_t * \sim C_t)$$

Equation 2.15

- **Hidden State Update:** The hidden state is updated using the output gate and the current cell state.

$$h_t = o_t * tanh\ (C_t)$$

Equation 2.16

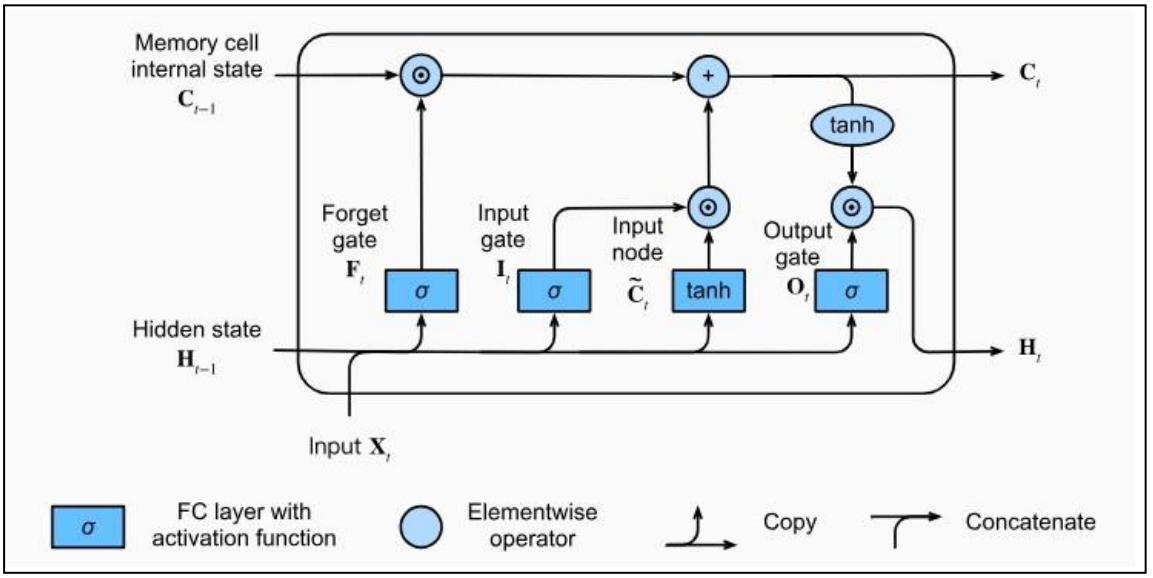


Figure 2.7: LSTM architecture (D2L.ai, 2023).

2.8.5 Bidirectional LSTM (BiLSTM) Neural Network

The BiLSTMs are basically an upgrade version of the standard LSTM architecture that can process data both in forward and backward directions (Graves & Schmidhuber, 2005). The regular deep LSTM's feed-forward from previous data, using prior events to help predict incoming inputs, but as an input, a part of the future can increase the accuracy in some scenarios. To alleviate this, BiLSTMs use two separate hidden LSTM layers; one processes the input sequence forwards, and the other handles it in reverse. At each time step, the outputs of these two layers are combined together, thus allowing us to extract future context as well (Schuster & Paliwal, 1997). It has the property to analyze sequences both forwards and backwards, which can help BiLSTMs in extracting relevant patterns/dependencies from the whole input sequence, making it quite promising for large contextual time-series forecasting.

Dealing with ultrasonic sensor data, BiLSTMs have shown to bring substantial improvements in occupancy detection. Due to the nature of occupancy, time patterns in occupancy data are often stateful (i.e., history and future states matter), as levels tend to be influenced by what happened before or desired immediately afterward. This allows the BiLSTM to capture those dependencies more effectively because it processes sensor data in forward and reverse directions, leading to improved accuracy for occupancy detection and prediction (Oshima et al., 2022). Moreover, BiLSTMs could benefit well where the traffic is cyclical or periodic behavior while the backward pass can also provide information on future states based on past trends.

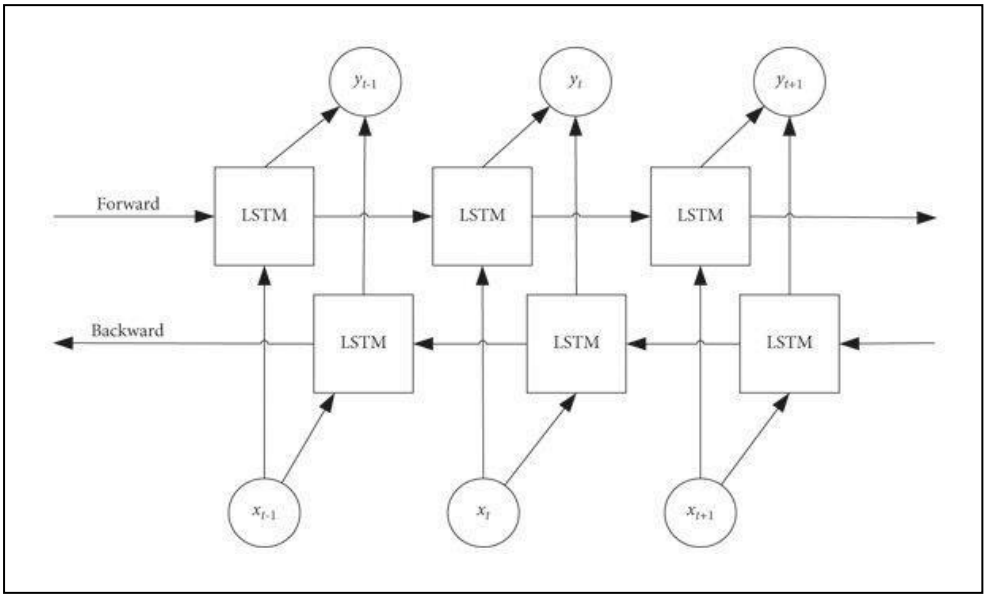


Figure 2.8: Schematic diagram of the BiLSTM network (Wang et al., 2021).

2.8.6 Comparative Analysis of GRU, LSTM & BiLSTM Neural Networks Based on Performance Metrics

Various performance metrics are in contrast and the LSTM, GRU, BiLSTM type algorithms which have widely been used for occupancy detection applications resulted. These are some results like accuracy, precision, recall, and overall prediction of the model.

- **Accuracy:** True positive predictions made out of all occupancy and non-occupancy cases. This is a standard model evaluation framework but it can be insufficient, especially for the imbalanced dataset or when there are different cost functions of FN and FP error (Sokolova & Lapalme 2009).

- **Precision:** Number of correct lodging events and negative predicates over the number of positive predictions by the model. Exclusively a method measuring the false positive reduction capability of the model (Goutte & Gaussier, 2005).

- **Recall:** Higher the recall, i.e., the true positive rate, the more actual positives our model can catch from the total Positives that are there. True Positive(Actual) - Recall = Row Count Actual where Row Count Actual is Total Positives in Dataset. As described in (Powers, 2020), this shows how the model individually predicted every actual occupancy.

- **F1-Score:** The F1-score is the harmonic mean of both precision and recall. This is established as the harmonic mean of these two scores to prevent data leakage (Powers, 2020).

- **ROC curve:** ROC (Receiver Operating Characteristics) is a widely used plot for binary classifiers that illustrates the number of true positive solutions you can offer when willing to accept any additional errors associated with including more new samples than current ones regarding classifying them as "True" or not. The AUC-ROC evaluates how well the model is predicting those true occupancy and non-occupancy events, with higher values representing better performance (Fawcett, 2006).

2.9 Confusion Matrix

The confusion matrix is a tool in form of a table where results from the classification model are compared with results obtained from the actual findings. This kind of matrix is usually applied in machine learning where the accuracy of a predictive model is tested. The confusion matrix that is depicted in figure 2 below. 9 has two axes: one for the labels that the model is supposed to predict the other for the actual observed label. The matrix usually consists of the following four values: TP means the number of cases the model accurately predicts the positive class, TN means that the model accurately predicts the negative class, FP means that the model incorrectly predicts the positive class and, last but not least, FN shows the number of times the model wrongly labels the negative class (Gad, 2021).

These test results can be used as the measures of evaluation of the classification models; for example, F1 score, recall, accuracy, and precision. The confusion matrix is very useful in problems concerning classification because the model works based on projecting a label or a class to each of the data points. From the confusion matrix, one can identify the model's weaknesses and fix them as needed (Gad, 2021).

		Predicted	
		Negative **(N)** -	Positive **(P)** +
Actual	Negative -	True Negative **(TN)**	**False Positive (FP) Type I Error**
	Positive +	**False Negative (FN) Type II Error**	True Positive **(TP)**

Figure 2.9: A confusion matrix.

- **Accuracy:** Accuracy is defined as the proportion of accurately detected cases among all instances.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

Equation 2.16

- **Recall:** Recall is the proportion of correctly predicted positive events to all actual positive occurrences.

$$Recall = \frac{TP}{TP + FN}$$

Equation 2.17

- **Precision:** Precision is defined as the proportion of correctly predicted positive cases among all positively predicted occurrences.

$$Precision = \frac{TP}{TP + FP}$$

Equation 2.18

- **F1-Score:** The F1-score is computed as the harmonic mean of recall and accuracy, providing a balanced assessment of both.

$$\text{F1} - \text{Score} = \frac{2 * Precision * Recall}{Precision + Recall}$$

Equation 2.19

Accuracy gives the overall performance of a model while precision and recall give information concerning how the model performs in predicting positive instances and identifying all the positive instances respectively. When the dataset is imbalanced, the F1-score is quite reasonable as it balances recall and accuracy to a certain extent.

2.10 PyCharm

Designed by Python experts for professional developers, PyCharm is a powerful IDE that has everything you need to write your code responsibly. Developed at JetBrains, it has a huge suite of features that provide pleasant coding for Python developer and also improve productivity.

- **Code Completion and Analysis:** PyCharm automatically generates good quality code and provides intelligent code completion, real-time error checking, and quick-fix suggestions as well. Predictive code completion and faster coding are possible as the IDE can predict your next development steps (context-aware suggestions) making it even easier and more efficient to work (JetBrains, n.d.).

- **Integrated Development Tools:** With a massive array of development tools that come preinstalled in PyCharm, such as an ultra-powerful debugger and test runner (that will become an inseparable part of your workflow), its Python profiler tool helps improve the performance time taken to build applications for production readiness while using Docker capabilities which simplify API requirements without configuring proxies manually (JetBrains, n.d.).

- **Project Navigation:** PyCharm includes powerful navigation features to let you jump between various parts of your codebase. Its features like "Navigate to Symbol", Recent Files, and a powerful search box make managing large projects a breeze (JetBrains, n.d.).

- **Refactoring & Code Generation:** The IDE provides a number of refactoring capabilities (rename variables, extract methods, change method signature...) It also offers great code generation functionality that helps in faster creation of common code constructs (JetBrains, n.d.).

- **Support for Web Development:** For example, it provides support for key web technologies like HTML, CSS, JavaScript, and SQL, which make PyCharm a good option for developers working with Python-based frameworks such as Django (JetBrains, n.d.).

- **Adaptability and Expandibility:** PyCharm has a ton of options for customization, allowing you to configure the IDE just how you like it. It is largely extensible using a multitude of plugins and extensions to tailor the IDE specifically for different types of hand out development workflows (TutorialsPoint, n.d.).

3. Requirement Analysis

3.1 General Objectives

The main objective of this research is to build a real-time robust distance measurement system based on the position of the first peak, ambient temperature, and the sampling frequency of the ultrasonic sensor, and to further compare it with the best configured LSTM neural network model for occupancy detection in smart office environments developed by (Aishin, 2024). The Tkinter GUI based Python program uses the ultrasonic sensor data to detect the first echo i.e. the first peak and it is further processed to measure the distance between the sensor and the person/object. Synchronization between these two systems should also be maintained to retain the accuracy of the distance measurement system. Through this integrated approach, the research endeavors to contribute to the advancement of robust distance measurement technology for prolonged real-world applications.

3.2 General structure of the system

The general structure of the real-time robust distance measurement system is shown in the Figure 3.1. The hardware used for this experiment consists of Red Pitaya based ultrasonic sensor and a laptop, which are connected via WLAN. The components of a distance measuring system with an ultrasonic sensor based around a Red Pitaya are shown in the image below. A machine with a software configuration, a distance calculation module, and a sensor hardware setup make up the major parts of the system. The sensor hardware design includes the Red Pitaya based ultrasonic sensor that transmits and receives the ultrasonic waves. A Python-based graphical user interface (GUI) that utilizes the sensor data runs on the machine with a software configuration. The distance calculation module is the last function in the GUI software, in which the peak index position, ambient temperature, and the Red Pitaya sampling rate are used to calculate the distance properly. These components are then interconnected, creating one system used to measure distance accurately and smoothly.

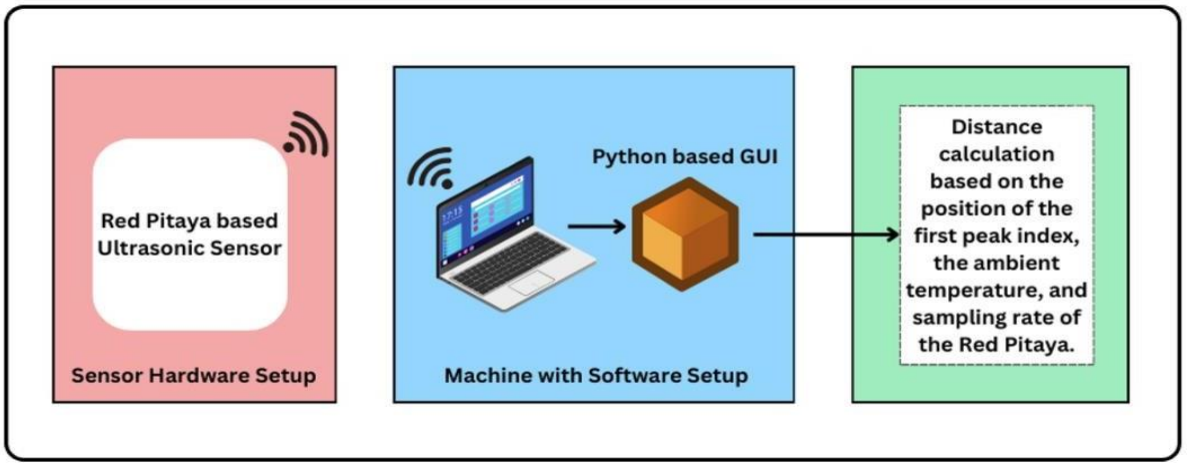


Figure 3.1:General structure of the system.

3.3 Research Objectives and Accomplishments

The focus of this research is how to optimize a classical model for real-time ultrasonic distance measurement with influences from ambient temperature and sampling frequency observed during operation that will be executed on Red Pitaya using an ultrasonic sensor. The model uses a Kalman filter to significantly reduce ADC signal fluctuation, thus improving measurement accuracy and reliability. This is with the one-step application of an adaptive or local thresholding method, configured by user input, which further improves the accuracy measured on the filtered signal for detection of the leading peak index position (refer Figure 3.21). When coupled with ambient temperature real-time data and sensor sampling frequency, it enables us to calculate the distance accurately. This research has achieved the integration of these high-performance signal processing techniques as a complete system and showed higher measurement uniformity along with robustness against environmental changes, bringing advanced developments in ultrasonic sensing technology.

3.4 Data acquisition from the Red Pitaya ultrasonic sensor

The C application on Red Pitaya uses the instruction we sent as a switch case and sends bytes of ultrasonic data back. The address where the Red Pitaya server is listening is "<ip_address_of_redpitaya:port_number>". 192.168.128.1 IP address of configured Red Pitaya and 61231 is the port number on which the server is listening. The program for Red Pitaya sends data in response to the following commands: • -a 1: Channel will be used to send ADC data to the client. • -a 0: Transmission of data halts until receipt of another command. • -f 1: Enable the FFT data for the client connected. • -f 0: Stop data transmission until instructed again. Figure 3.2 displays the raw ADC data from the UDP client. The first 16 columns of data in each data file are headers, which are used to record the measurement settings. Table 3.1 provides a thorough summary of these headers.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1	64	32768	1	1	512	0	1953125	12	0	0	0.3	0	0	V0.2	0	0	-184
2	64	32768	1	1	512	0	1953125	12	0	0	0.3	0	0	V0.2	0	0	-170
3	64	32768	1	1	512	0	1953125	12	0	0	0.3	0	0	V0.2	0	0	-144
4	64	32768	1	1	512	0	1953125	12	0	0	0.3	0	0	V0.2	0	0	-203
5	64	32768	1	1	512	0	1953125	12	0	0	0.3	0	0	V0.2	0	0	-143
6	64	32768	1	1	512	0	1953125	12	0	0	0.3	0	0	V0.2	0	0	-166
7	64	32768	1	1	512	0	1953125	12	0	0	0.3	0	0	V0.2	0	0	-159
8	64	32768	1	1	512	0	1953125	12	0	0	0.3	0	0	V0.2	0	0	-168
9	64	32768	1	1	512	0	1953125	12	0	0	0.3	0	0	V0.2	0	0	-185
10	64	32768	1	1	512	0	1953125	12	0	0	0.3	0	0	V0.2	0	0	-144
11	64	32768	1	1	512	0	1953125	12	0	0	0.3	0	0	V0.2	0	0	-143
12	64	32768	1	1	512	0	1953125	12	0	0	0.3	0	0	V0.2	0	0	-133
13	64	32768	1	1	512	0	1953125	12	0	0	0.3	0	0	V0.2	0	0	-216
14	64	32768	1	1	512	0	1953125	12	0	0	0.3	0	0	V0.2	0	0	-157
15	64	32768	1	1	512	0	1953125	12	0	0	0.3	0	0	V0.2	0	0	-195

Figure 3.2: Raw ADC data received from the UDP Client.

Index	Header Description	Length (byte)	Value	Unit
1	Header length (count for fields)	4	64	-
2	Data length	4	32768	ADC_DATA: 32768
				FFT_DATA: 170
3	Class detected	4	1 or 2	1: Object
				2: Human
4	Measurement type	4	0 or 1	0: FFT
				1: ADC
5	Frequency resolution of a sampling time t depend on measurement type	4	512	Hz (1/s) or t (ns)
6	Normation	4	0,1,2	
7	Sampling frequency	4	1953125	Hz (1/s)
8	ADC Resolution	4	12	
9	Ambient temperature	4	-500…500	°C
10	Distance between sensor and first object (round-trip-time)	4	$0 - 2^{32}$	t (µs)
11	Time of flight of Ultrasound in air forward and backward → distance to first object	4	$0 - 2^{32}$	t (µs]
12	FFT Window length	4		
13	FFT Window Offset (index → freq_min_index)	4		

14	Software Version (RP) as a string	4	V0.2	
15	reserved1	4		
16	reserved2	4		
17	Data […]	64		2 byte each

Table 3.1: Header description of the raw ADC data received from the UDP Client.

3.4.1 Real-Time Ultrasonic ADC Data Extractor

The ***ultrasonic_data_extrator.py*** script defines a class called RedPitayaSensor, which is used to connect with a UDP server running on Red Pitaya ultrasonic sensor. This class establishes a connection, issues a command to the server, receives data, and processes that ultrasonic data. The received data is unpacked, converted to a Pandas dataframe, and printed. Detailed explanation of the python script is provided below:

```
import socket
import struct
import pandas as pd
import threading
```

Figure 3.3: The required libraries imported in ultrasonic_data_extrator.py.

```
class RedPitayaSensor:
    def __init__(self):
        self.buffer_size = 65536
        self.size_of_raw_adc = 16384
        self.msg_from_client = "-a 1"
        self.bytes_to_send = str.encode(self.msg_from_client)
        self.server_address_port = ("192.168.128.1", 61231)
        self.sensor_status_message = "Waiting to Connect with RedPitaya UDP Server!"
        print(self.sensor_status_message)
        self.udp_client_socket = socket.socket(family=socket.AF_INET, type=socket.SOCK_DGRAM)
        self.udp_client_socket.settimeout(10)  # Set timeout for socket operations
        self.send_msg_to_server()
        self.running = True
        self.thread = threading.Thread(target=self.receive_data_loop)
        self.thread.start()

    def get_sensor_status_message(self):
        return self.sensor_status_message
```

Figure 3.4: Creates a thread to process incoming data and initializes a number of settings, including the buffer size, message to transmit, server address, and port for setting up the UDP connection.

```python
def get_sensor_status_message(self):
    return self.sensor_status_message


1 usage
def send_msg_to_server(self):
    try:
        self.udp_client_socket.sendto(self.bytes_to_send, self.server_address_port)
    except socket.error as e:
        self.sensor_status_message = f"Failed to send message: {e}"
        print(self.sensor_status_message)
```

*Figure 3.5: The current sensor status message is returned by the **get_sensor_status_message** method, and any errors that may arise during the sending process are handled by the **send_msg_to_server** method, which transmits the encoded message to the UDP server.*

```python
def get_data_from_server(self):
    try:
        packet = self.udp_client_socket.recv(self.buffer_size)
    except socket.timeout:
        self.sensor_status_message = "Connection timed out."
        print(self.sensor_status_message)
        return None
    except socket.error as e:
        self.sensor_status_message = f"Error receiving data: {e}"
        print(self.sensor_status_message)
        return None

    try:
        header_length = int(struct.unpack( __format: '@f', packet[:4])[0])
        ultrasonic_data_length = int(struct.unpack( __format: '@f', packet[4:8])[0])
        header_data = [i[0] for i in struct.iter_unpack( __format: '@f', packet[:header_length])]
        ultrasonic_data = [i[0] for i in struct.iter_unpack( __format: '@h', packet[header_length:])]
    except struct.error as e:
        self.sensor_status_message = f"Data unpacking error: {e}"
        print(self.sensor_status_message)
        return None

    self.sensor_status_message = f"Sensor Connected Successfully at {self.server_address_port}!"
    print(self.sensor_status_message)
    print(f"Total Received: {len(packet)} Bytes.")
    print(f"Length of Header: {len(header_data)}")
    print(f"Length of Ultrasonic Data: {len(ultrasonic_data)}")

    df = pd.DataFrame(ultrasonic_data, columns=['raw_adc'])
    return df["raw_adc"]
```

Figure 3.6: The system receives and processes data from the UDP server. The data is unpacked using the struct module, where the header and ultrasonic data lengths are extracted and processed. The unpacked ultrasonic data is then stored in a Pandas DataFrame for further analysis.

```
def receive_data_loop(self):
    while self.running:
        data = self.get_data_from_server()
        if data is not None:
            print(f"Received Data: {data.head()}")

1 usage (1 dynamic)
def stop(self):
    self.running = False
    self.thread.join()
    self.udp_client_socket.close()
```

*Figure 3.7: The **stop** method ends the data reception thread and shuts the UDP connection, whereas the **receive_data_loop** method loops back and forth continually receiving data from the server until the running flag is set to False.*

3.5 Python based GUI Configuration for Real-Time Robust Ultrasonic Distance Measurement

A graphical user interface (GUI) for processing and analyzing ultrasonic data signals is created using the Python script ***Classical_GUI.py***, which makes use of the Tkinter package. Plotting capabilities, signal filtering, peak detection algorithms, and bespoke data extraction and signal processing techniques are all included. Detailed explanation of the python script is provided below:

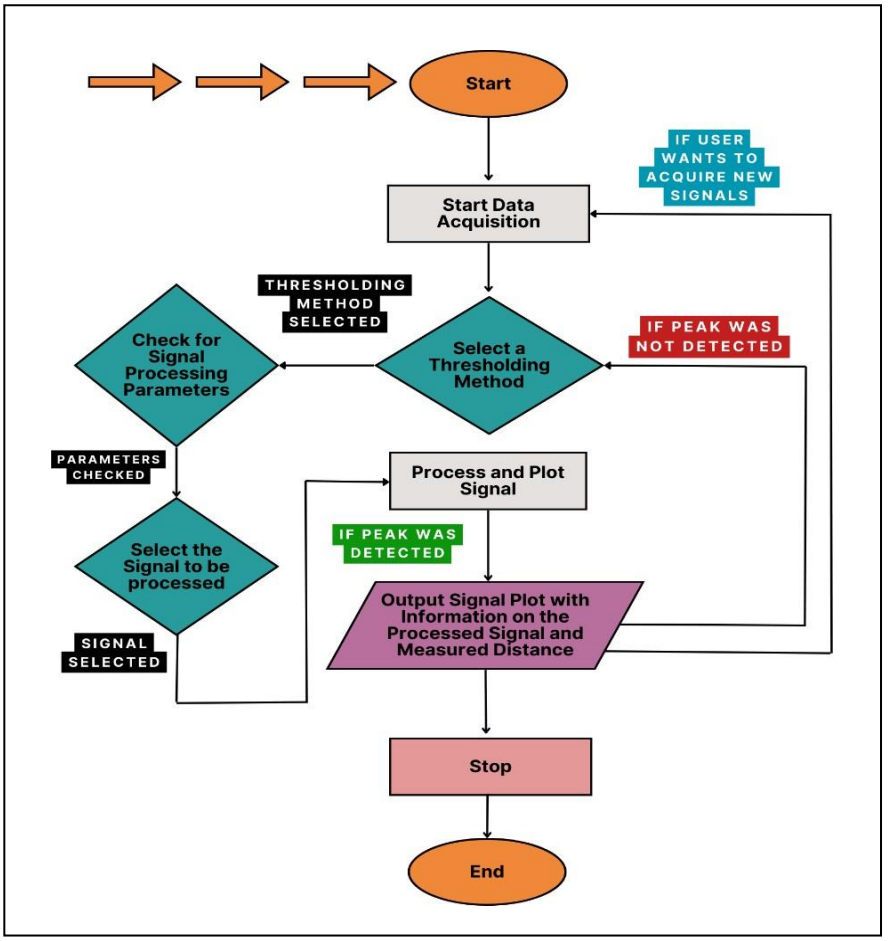


Figure 3.8: Flowchart of the real-time robust ultrasonic distance measuring Classical GUI.

- Imports and Initializations

```
from tkinter import *
from tkinter.scrolledtext import ScrolledText
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
import numpy as np
import matplotlib.pyplot as plt
from ultrasonic_data_extractor import RedPitayaSensor
```

Figure 3.9: The required libraries and class RedPitayaSensor imported in Classical_GUI.py.

The ***EnhancedRedPitayaSensor*** class extends the normal reading functionality developed for the extended data storage capabilities. In the_init_method, it initializes an empty list called data_storage for holding retrieved data and assigns a limit (2) indicating that only two data sets will be stored at a time. The entire method called get_data_from_server replaced the parent class with doing a lot of if-else through one ask function and then another until we got what we want at most once per minute into our data storage (data_storage) that can hold as many entries specified by the variable named data_limit. In case of success, the data is concatenated to data_storage, and it informs us about the process. The loop will break if the response data is not returned (here data variable being set as None). Next, it will return the container stored data. There is also a reset_data_storage method to clear the data, which sets data_storage = []. This class provides the ability to allow controlled accumulation and retrieval of data from the server with a possible reset when needed.

```
class EnhancedRedPitayaSensor(RedPitayaSensor):
    def __init__(self):
        super().__init__()
        self.data_storage = []
        self.data_limit = 2

    def get_data_from_server(self):
        while len(self.data_storage) < self.data_limit:
            data = super().get_data_from_server()
            if data is not None:
                self.data_storage.append(data)
                print(f"Data set {len(self.data_storage)} stored.")
            else:
                break
        return self.data_storage

    4 usages (4 dynamic)
    def reset_data_storage(self):
        self.data_storage = []
```

Figure 3.10: Extends the RedPitayaSensor class with data storing capabilities. get_data_from_server: Overrides the basic function for storing several pieces of data. reset_data_storage: Clears the stored data.

- Signal processing functions

The Kalman Filter is a math formula applied to predict the condition of a linear dynamic system when dealing with uncertain or inaccurate data. It is commonly utilized in a range of industries to generate predictions or estimates from noisy data, offering optimal estimates through the fusion of historical measurements with a mathematical system model (Zhang et al., 2022)(Auger et al., 2022). Here is the Python code along with the explanation of how I applied it to generate a smoothed signal for subsequent processing with the selected thresholding method.

```python
def kalman_filter(signal, process_variance=1e-5, measurement_variance=0.5):
    n_iterations = len(signal)
    smoothed_signal = np.zeros(n_iterations)
    error_covariance = 0.0
    current_estimate = signal[0]

    for t in range(n_iterations):
        error_covariance += process_variance
        kalman_gain = error_covariance / (error_covariance + measurement_variance)
        current_estimate += kalman_gain * (signal[t] - current_estimate)
        error_covariance *= (1 - kalman_gain)
        smoothed_signal[t] = current_estimate

    return smoothed_signal
```

Figure 3.11: Kalman filter function representation.

Here, signal is the input array, process_variance represents the variance of the process noise, and measurement_variance represents the variance of the measurement noise. The filter initializes the current estimate with the first value of the signal and sets the initial error covariance to zero. In each iteration, it updates the error covariance to account for the process variance, computes the Kalman gain (which determines the weight given to the new measurement), updates the current estimate using the Kalman gain and the new measurement, and adjusts the error covariance. The smoothed signal is stored in smoothed_signal, which is returned at the end of the function. This process helps to reduce the noise in the signal, providing a smoother output ("pykalman." GitHub)(Dhanoop Karunakaran, 2018).

```python
def yanowitz_bruckstein_thresholding(signal, window_size=100, threshold_init_factor=0.5, min_peak_prominence=0.1):
    threshold = threshold_init_factor * np.max(signal)

    for i in range(window_size, len(signal) - window_size):
        window = signal[i - window_size:i + window_size]
        local_max = np.max(window)

        if local_max > threshold:
            threshold = local_max - min_peak_prominence

        if signal[i] >= threshold and signal[i] == local_max:
            if all(signal[i] >= signal[j] for j in range(i - window_size, i + window_size) if j != i):
                return i

    return None
```

Figure 3.12: Yanowitz-Bruckstein adaptive local thresholding function representation.

The Yanowitz-Bruckstein adaptive local thresholding algorithm is primarily designed for image processing applications and to adapt the Yanowitz-Bruckstein method for ADC signal peak detection, the signal must be considered as a one-dimensional image where the intensity of each pixel corresponds to the signal amplitude at each point in time (Yanowitz & Bruckstein, 1989). The goal is to dynamically adjust a threshold to efficiently identify the first significant peak in the signal. The function def ***yanowitz_bruckstein_thresholding()*** is used for detecting the first peak in a signal array through adaptive local thresholding. The global threshold is established by multiplying the maximum signal value with a factor called threshold_init_factor. It iterates through every signal point in a specified window size through a loop process.

The function calculates a peak in the window for every point. If the local peak is higher than the threshold, the threshold decreases by a specific min_peak_prominence (Nixon & Aguado, 2019). A peak is recognized at a specific point in the signal if it matches the highest point in its surroundings and is not smaller than any other points in the area, effectively singling out the most significant peak nearby (Otsu, 1979). The function will return the index of the initial peak found, or will return None if no peak meets the specified criteria across the entire signal.

```
def niblacks_local_thresholding(signal, window_size=2000, k=2.7):
    n = len(signal)
    for i in range(window_size, n - window_size):
        local_segment = signal[i - window_size:i + window_size]
        local_mean = np.mean(local_segment)
        local_std = np.std(local_segment)
        threshold = local_mean + k * local_std

        if signal[i] > threshold:
            if all(signal[i] >= signal[j] for j in range(i - window_size, i + window_size) if j != i):
                return i

    return None
```

Figure 3.13: Niblack's local thresholding function representation.

The function def ***niblacks_local_thresholding()*** applies local thresholding to the input signal, this is a technique often used in image processing and is adapted in this project for a one-dimensional data array (Niblack, 1986). The purpose is to identify first peak in the signal that significantly stands out from its local surroundings based on statistical characteristics, that is through the computation of local mean and local standard deviation. The above function shown in Figure 3.12 takes three parameters: the signal (a list or array of numerical values), window_size (default is 2000), and k (a scaling factor for the standard deviation, default is 2.7). The function iterates through each point in the signal, except for the first and last sections defined by the window_size. For each point, it calculates the local mean and standard deviation of the values within the window centered at that point. It then computes a threshold using the formula: local mean plus k times the local standard deviation. The function checks if the current signal value exceeds this threshold and if it is the local maximum within its window. If both conditions are met, the function returns the index of that peak (Wang et al., 2008). If no such peak is found after processing the entire signal, the function returns None.

```
def calculate_distance(peak_index, temperature, sampling_frequency):
    speed_of_sound = 331.45 + 0.606 * temperature
    time_delay = peak_index / sampling_frequency
    distance = 0.5 * speed_of_sound * time_delay
    return distance
```

Figure 3.14: Distance calculation function representation.

The def ***calculate_distance()***, is used to calculate the distance based on the position of the detected first peak in the signal and the ambient temperature (Piercy et al., 1977). We know that the distance 'd' can be computed using the formula: d = 1/2 * v * t where,

- 'v' is the speed of sound in air, which depends on the ambient temperature. The formula for 'v' (in meters per second) based on temperature 'T' (in degrees Celsius with default is 20) is: v = 331.45 + 0.606 * T (Jiao, 2015),
- 't' is the time corresponding to the first peak in the signal. Assuming the sampling rate (samples per second) of the Red Pitaya is known in our case 1953125 Hz (Pech et al., 2019). we can calculate 't' as t = peak_index / sampling_rate.

- GUI functions

```
def update_parameters_display(method_var, yanowitz_frame, niblack_frame):
    if method_var.get() == 'Yanowitz-Bruckstein Thresholding':
        yanowitz_frame.pack(side=TOP, fill=X)
        niblack_frame.pack_forget()
    elif method_var.get() == 'Niblack\'s Local Thresholding':
        niblack_frame.pack(side=TOP, fill=X)
        yanowitz_frame.pack_forget()
```

Figure 3.15: Adapts the displayed parameter input fields dynamically according to the chosen peak detection technique.

```
def clear_plot(master):
    for widget in master.pack_slaves():
        if isinstance(widget, Canvas):
            widget.destroy()
```

Figure 3.16: Clears the current plot from the GUI.

The function def ***process_and_plot_signal()*** has been developed to analyze and plot a signal from a chosen sensor using a tkinter generated GUI. The function begins by erasing any existing plots and writes the status of the data processing to the output text area. It then gets and checks the temperature and sampling frequency entered by the user. The function helps in retrieving the raw data of the selected signal from the data storage of the sensor. This raw signal is then filtered using a Kalman filter. Based on the selected peak detection method by the user from the two options of Yanowitz-Bruckstein Thresholding or Niblack's Local Thresholding, the code determines the index of the peak in the filtered signal. The detected peak, if any, is then employed to compute the distance from the given temperature and the sampling frequency. A graph showing the original signal and filtered signal is displayed; if a peak is found, it is indicated. The plot is then displayed in the graphical user interface (GUI), and data is written to the text area, including the peak index and the computed distance. From the start of the process to its conclusion, the GUI is altered to display the results and progress.

```
def process_and_plot_signal(temperature_var, sampling_frequency_var, peak_detection_method,
                            yanowitz_frame, niblack_frame, master, output_text, selected_signal, sensor):
    clear_plot(master)
    output_text.delete('1.0', END)
    output_text.insert(END, "Starting data processing...\n")
    master.update()

    try:
        temperature = float(temperature_var.get())
        sampling_frequency = int(sampling_frequency_var.get())
    except ValueError:
        output_text.insert(END, "Invalid temperature or sampling frequency. Please enter valid numbers.\n")
        return

    signal_index = int(selected_signal.get().split()[-1]) - 1
    if signal_index >= len(sensor.data_storage):
        output_text.insert(END, f"Selected signal {signal_index + 1} is not available.\n")
        return

    raw_adc = sensor.data_storage[signal_index]
    output_text.insert(END, f"Processing data for signal {signal_index + 1}...\n")
    output_text.insert(END, f"Length of Extracted Ultrasonic Data = {len(raw_adc)}\n")
    master.update()

    filtered_signal = kalman_filter(raw_adc)
```

*Figure 3.17: Function def **process_and_plot_signal()** computes the distance, shows the findings, performs peak detection and filtering to the chosen signal. updates the GUI with findings and progress.*

```
def extract_data(sensor, output_text):
    output_text.delete('1.0', END)
    output_text.insert(END, "Starting data acquisition...\n")
    sensor.reset_data_storage()
    start_time = time.time()
    sensor.get_data_from_server()
    end_time = time.time()
    output_text.insert(END, "Data acquisition completed.\n")
    output_text.insert(END, f"Data acquisition time: {end_time - start_time:.2f} seconds\n")
```

Figure 3.18: Data extraction from the sensor begins, and the GUI is updated as it progresses.

- Main Function and GUI Setup

The function def ***main()*** with the help of the GUI of the tkinter, processes and plots the ADC signals to determine the first peak and calculate the distance based on the ambient temperature. The GUI consists of several input fields and buttons: The fields that can be edited are the temperature and sampling frequency inputs, the set of radio buttons by which one can select the preferred peak detection method, Yanowitz-Bruckstein Thresholding or Niblack's Local Thresholding and the signal to be processed, either Signal 1 or Signal 2. It also has specific parameters that are related to each one of the peak detection methods and they are shown in a dynamic way according to the selected method at the time. For the processing of the sensor data an ***EnhancedRedPitayaSensor*** object is defined to work with the data. These are the buttons to get the data, to process it, to plot the signal and the stop button of the application. The output_text area is for status/result information output. The main purpose of the program is to open the windows based graphical user interface and then goes into the main event loop to await the user's commands.

```
def main():
    root = Tk()
    root.title("Classical Tool for ADC signal processing to detect the first peak and measure the distance w.r.t ambient temperature")

    temperature_var = StringVar(value="20")  # Default temperature in Celsius
    sampling_frequency_var = StringVar(value="1953125")  # Default sampling frequency in Hz

    config_frame = Frame(root)
    config_frame.pack(side=TOP, fill=X)

    Label(config_frame, text="Temperature (°C):").pack(side=LEFT)
    Entry(config_frame, textvariable=temperature_var).pack(side=LEFT)

    Label(config_frame, text="Sampling Frequency (Hz):").pack(side=LEFT)
    Entry(config_frame, textvariable=sampling_frequency_var).pack(side=LEFT)

    peak_detection_method = StringVar(value="Yanowitz-Bruckstein Thresholding")
```

Figure 3.19: Configures the primary GUI window, initializes variables, frames, and widgets, and specifies button actions. Executes the Tkinter main event loop.

*Figure 3.20: Sample snippet of the GUI when **Classical_GUI.py** is launched in Pycharm IDE.*

```
C:\Users\PrajwalPraveenAthkar\PycharmProjects\Test\.venv\Scripts\python.exe C:\Users\PrajwalPraveenAthkar\PycharmProjects\Test\Classical_GUI.py
Waiting to Connect with RedPitaya UDP Server!
Sensor Connected Successfully at ('192.168.128.1', 61231)!
Total Received: 32832 Bytes.
Length of Header: 16
Length of Ultrasonic Data: 16384
Data set 1 stored.
Sensor Connected Successfully at ('192.168.128.1', 61231)!
Total Received: 32832 Bytes.
Length of Header: 16
Length of Ultrasonic Data: 16384
Data set 2 stored.
```

*Figure 3.21: Sample snippet of the run-time console when **Classical_GUI.py** is launched in Pycharm IDE.*

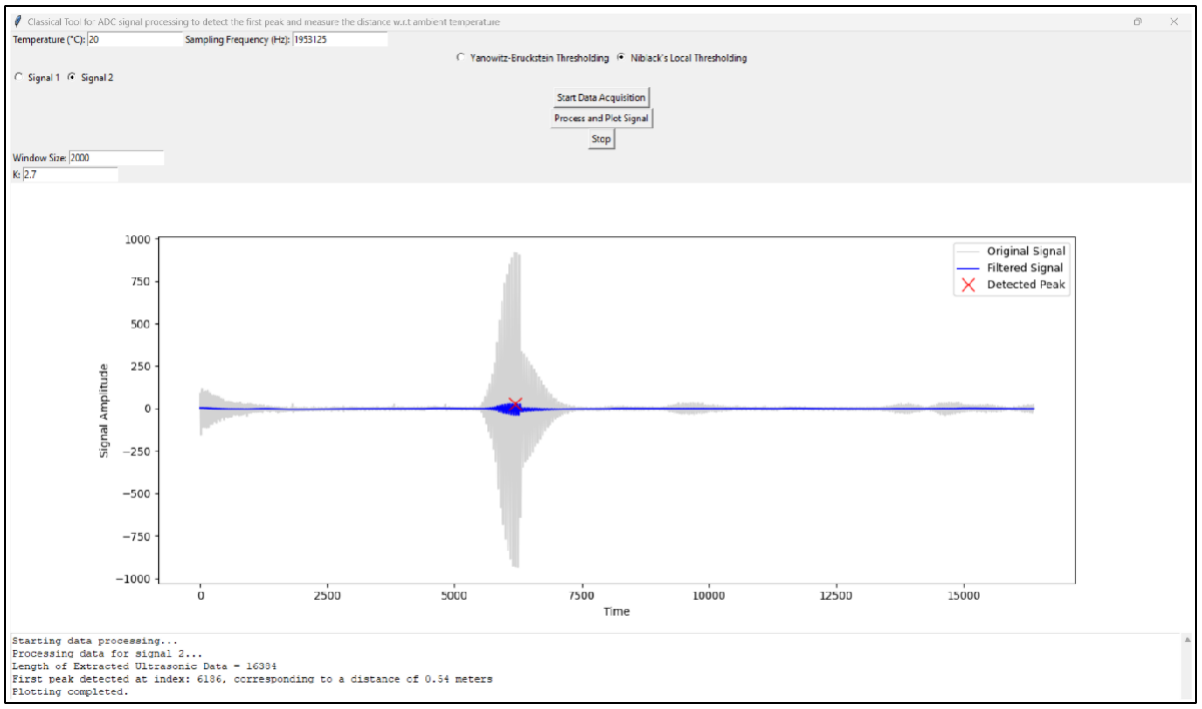


Figure 3.22: Sample snippet of the GUI after detecting the first peak index in signal 2 with Niblack's local thresholding, and computing the distance based on the detected first peak index, ambient temperature and sampling frequency of the Red Pitaya ultrasonic sensor.

3.6 Work Environment Description and Real-Time Test Conditions

The experiment is conducted in Computational Intelligence Laboratory present in building 8, room 102a, the machine with the GUI software setup is placed on a table and the Red Pitaya device is positioned on the left side of the computer on a tripod stand, where the ultrasonic sensor is pointing to the chair, as shown in Figure 3.23. A chair is purposefully positioned 0.6m - 1m away from the sensor, the complete experimental setup can be observed in Figure 3.23 without person and in Figure 3.24 with person. The artificial lighting in the room was turned on and the window was kept open during the experiment. The Red Pitaya device is powered by connecting it to a power source via the tripod stand.

Figure 3.23 on the left side and Figure 3.24 on the right side: The snapshot of the experimental configuration without person on the left side and with person on the right side.

3.7 GUI Prototype Use Cases

The GUI image shown in Figure 3.20 is used for ADC signal processing to find the first peak and measure the distance with respect to the ambient temperature and sampling frequency of the Red Pitaya sensor. At the top it contains fields to input temperature, which is default as 20 in Celsius and the sampling frequency, by default as 1953125 in Hertz. Further down these fields, there is a set of radio buttons that enables the user to choose between 'Signal 1' and 'Signal 2' for further processing. Users can choose between two thresholding methods: Yanowitz and Bruckstein adaptive thresholding and Niblack's local thresholding are selected in their respective radio buttons. Also, there are input fields for the respective thresholding techniques, for Yanowitz-Bruckstein adaptive thresholding there are window size is 100, threshold initialization factor is 0.5 and minimum peak prominence is 0.1 (refer Figure 3.25) and for Niblack's local thresholding there are window_size is 2000 and k is a scaling factor for the standard deviation, 2.7 by default (refer Figure 3.26). The most prominent features on the interfaceare three buttons, namely 'Start Data Acquisition,' 'Process and Plot Signal,' and 'Stop,' which define the actions with data and signal processing. The bottom of the GUI seems to be designed for showing the result or the plottedsignals if the processing is started.

The GUI now shows the outcome of applying Yanowitz-Bruckstein adaptive thresholding to Signal 2. The input parameters remain the same: a temperature value of 20 °C, a sampling frequency of 1953125 Hz, a window size of 100 and an initial threshold factor of 0.5, and a minimum peak prominence of 0.1. The main panel upon processing displays a plot of the signal amplitude with respect to time. The plot presented in Figure 3.25 contains the original signal in grey color, the Kalman filtered signal in blue color, and the detected peak is marked with an 'X' in the red color. The detected peak is identified to be at an index of 2556 which in terms of distance measurement is equal to 0.22 meters, as presented in the text placed under the plot of the film. The signal analysis results are represented both in the form of image and text where the thresholding method used in the work is described and exemplified on the actual signal data to show its efficiency in identifying the prominent peaks.

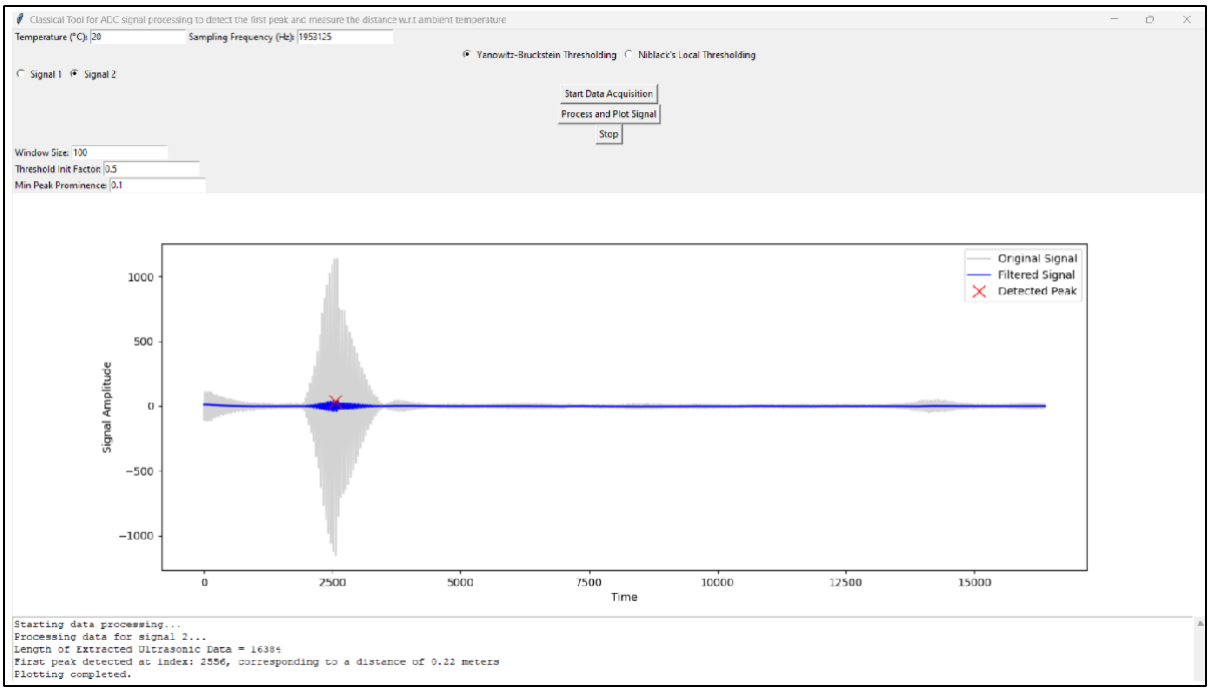


Figure 3.25: Sample snippet of the GUI outcome when Yanowitz-Bruckstein adaptive thresholding to Signal 2 is selected by the user.

In Figure 3.26, GUI shows the outcome of applying Niblack's Local Thresholding on Signal 2. The input parameters with default values: the window size of 2000 and the parameter K equal to 2.7, the temperature at which the experiment is conducted stays to be 20°C and the sampling frequency 1953125 Hz. The plot in the main panel is as before, the signal amplitude is plotted versus time. Here, the main signal is shown in grey color, the filtered signal in blue color and the detected peak is represented by 'X' symbol. The detected peak is at an index of 3091, which when further computed gives a distance 0.27 meters based on the distance calculation logic. The textual output below the plot also validates the processing steps with the identification of the number of data points extracted from the ADC signal being 16384 and the identification of the peak and plotting it. This visualization and data support the use of Niblack's Local Thresholding in locating the prominent peaks of the signal data, and it offers easily interpretable results for further examination.

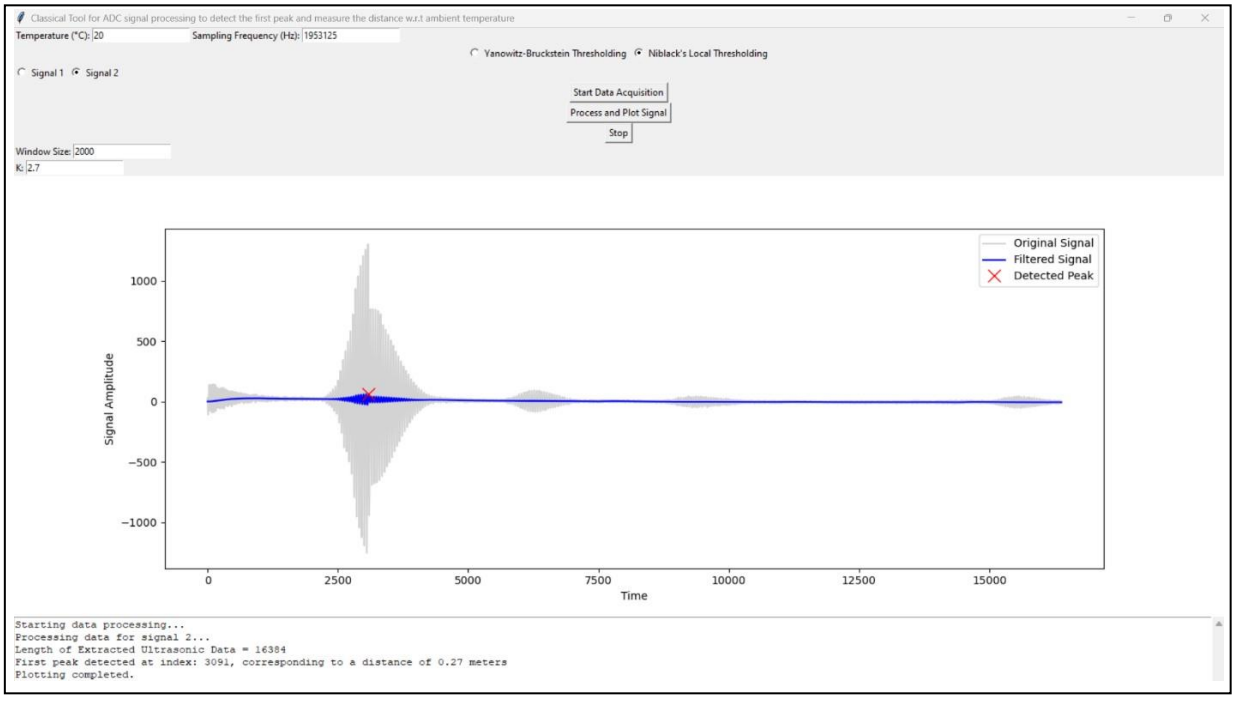


Figure 3.26: Sample snippet of the GUI outcome when Niblack's Local Thresholding to Signal 2 is selected by the user.

3.8 Requirement Clarification and Future Scope for the Upgraded GUI

The GUI software requirements are calculating the distance with the precision depending on the position of the first reflection, ambient temperature, and sampling frequency, allowing for more than two distance measurements per second, creating a simple interface for testing, testing all the implemented methods, comparing the precision, memory usage, repetition rate, and stability of the algorithms. The developed ***Classical_GUI.py*** successfully passes all the requirements for the GUI software, which can acquire signal data of total length 32832 bytes, in which 16 instances contain header information and 16384 instances contain ultrasonic data based on the script ***ultrasonic_data_extrator.py***. As an enhancement to this GUI software, ***Upgraded_GUI.py*** was developed where I have increased the data acquisition capability to receive upto 50020 bytes, containing 5 instances of header information and 75000 instances of ultrasonic data, the ***ultrasonic_data_extrator_upgraded.py*** upgraded script outlines a Python class *RedPitayaSensor* for controlling the RedPitaya over UDP and using SSH. The class defines some pars, for instance, buffer sizes and the details of the network. It creates a UDP client socket for transmission and reception of data packets and Paramiko for the SSH connection for the execution of remote commands in the RedPitaya device. The `give_ssh_command` method is used for connecting to the RedPitaya via SSH and process the output or errors of a given command. The `send_msg_to_server` method encodes the messages and sends them to the UDP server, the `get_data_from_server` method sends the initial message to request data, then it receives a data packet, splits the header and ultrasonic data from all the blocks. The ultrasonic data is then stored in a Pandas DataFrame which is provided for use at a later stage. The class contains methods to set and get status messages, informing about the state of the connection and data transmission (refer Figure 3.27).

```
import socket, struct
import pandas as pd
import time
import paramiko

class RedPitayaSensor:
    def __init__(self):
        self.size_of_raw_adc = 25000
        self.buffer_size = (self.size_of_raw_adc + 5) * 4
        self.msg_from_client = "-i 1"
        self.hostIP = "192.168.128.1"
        self.data_port = 61231
        self.ssh_port = 22
        self.server_address_port = (self.hostIP, self.data_port)
        # Create a UDP socket at client side
        self.sensor_status_message = "Waiting to Connect with RedPitaya UDP Server!"
        print(self.sensor_status_message)
        self.udp_client_socket = socket.socket(family=socket.AF_INET, type=socket.SOCK_DGRAM)
        self.client = paramiko.SSHClient()
        self.client.set_missing_host_key_policy(paramiko.AutoAddPolicy())

    6 usages
    def give_ssh_command(self, command):
        try:
            # Connect to the Redpitaya device
            self.client.connect(self.hostIP, self.ssh_port, username: "root", password: "root")
            self.set_sensor_message(f"Connected to Redpitaya {self.hostIP}")

            # Execute the command
            stdin, stdout, stderr = self.client.exec_command(command)

            # Read the command
            output = stdout.read().decode()
            error = stderr.read().decode()
```

Figure 3.27: Snippet of the upgraded data acquisition script ***ultrasonic_data_extrator_upgraded.py.***

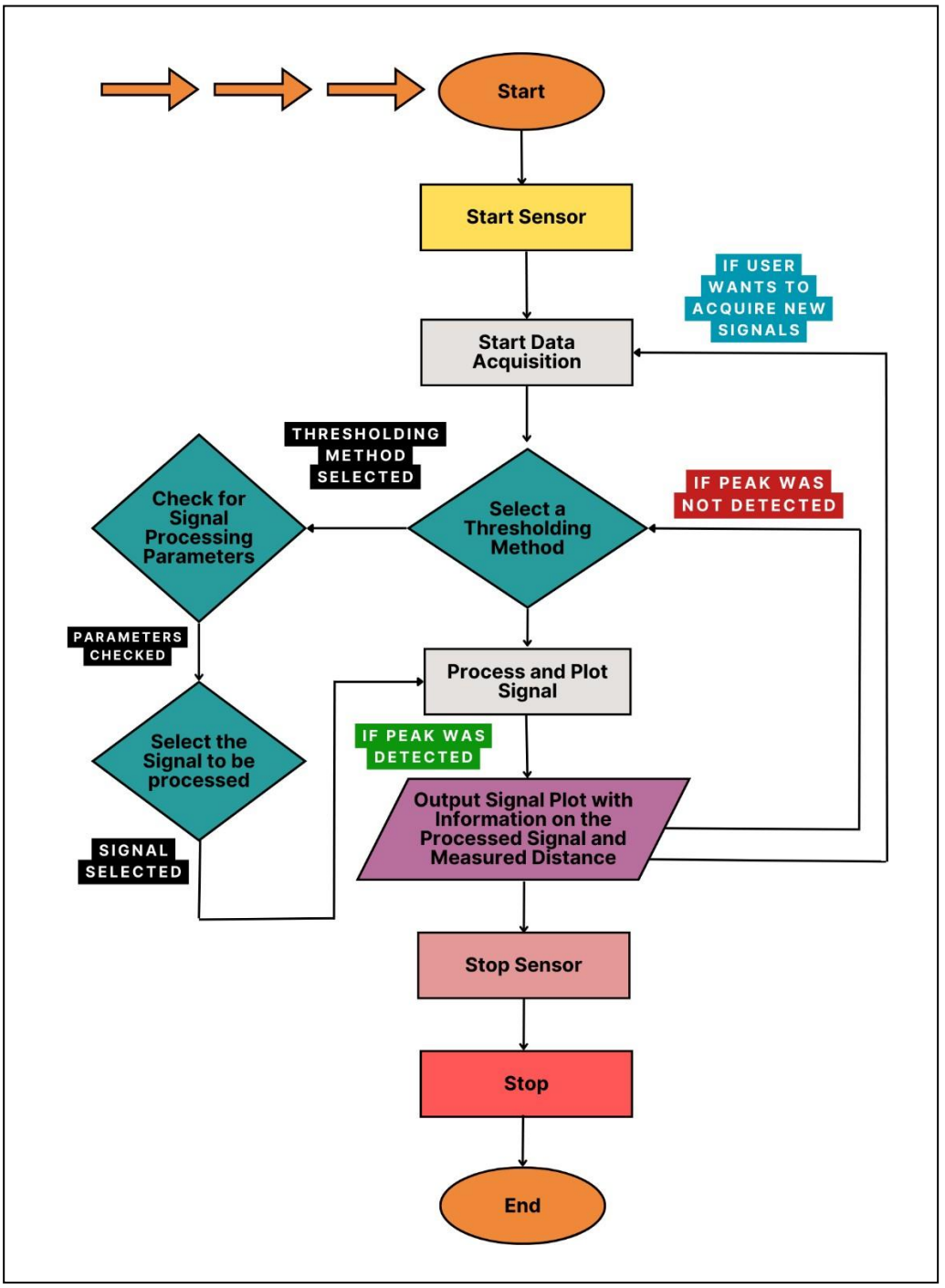


Figure 3.28: Flowchart of the real-time robust ultrasonic distance measuring Upgraded GUI.

3.8.1 Drawbacks of the Upgraded GUI

The ***ultrasonic_data_extrator_upgraded.py*** script upgrade developed to acquire large signal data causes drawbacks in the GUI and makes it buggy. The drawbacks are explained below with real-time test results and snapshots of the GUI.

- More than the required time taken to acquire the signal data as the GUI requirement was to process 2 signals per second (refer Figure 3.29).
- At instances there is failure observed in data acquisition during GUI run-time (refer Figure 3.31a and Figure 3.31b), and also there is lag (drop in framerate) in the GUI for a few seconds at times when 'Process and Plot Signal' is clicked by the user for signal processing (refer Figure 3.32). Due to these issues the software repetition rate and real-time performance has decreased.
- Memory consumption during run-time of the GUI rendered by ***Upgraded_GUI.py*** was observed to be higher (refer Figure 3.33), compared to the GUI rendered by ***Classical_GUI.py*** with a capability to acquire fixed signal data of total length 32832 bytes and perform further signal processing on it (refer Figure 3.34).
- At instances, it is observed that noisy signals are acquired by the GUI rendered by ***Upgraded_GUI.py*** (refer Figure 3.35 and Figure 3.36).

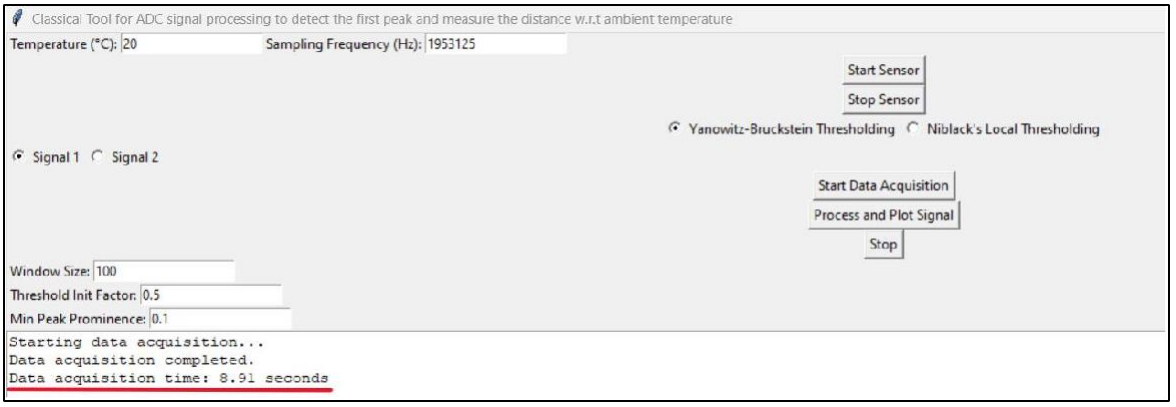


Figure 3.29: Snippet of the upgraded GUI after data acquisition is completed and time taken to acquire the data is displayed as 8.91 seconds.

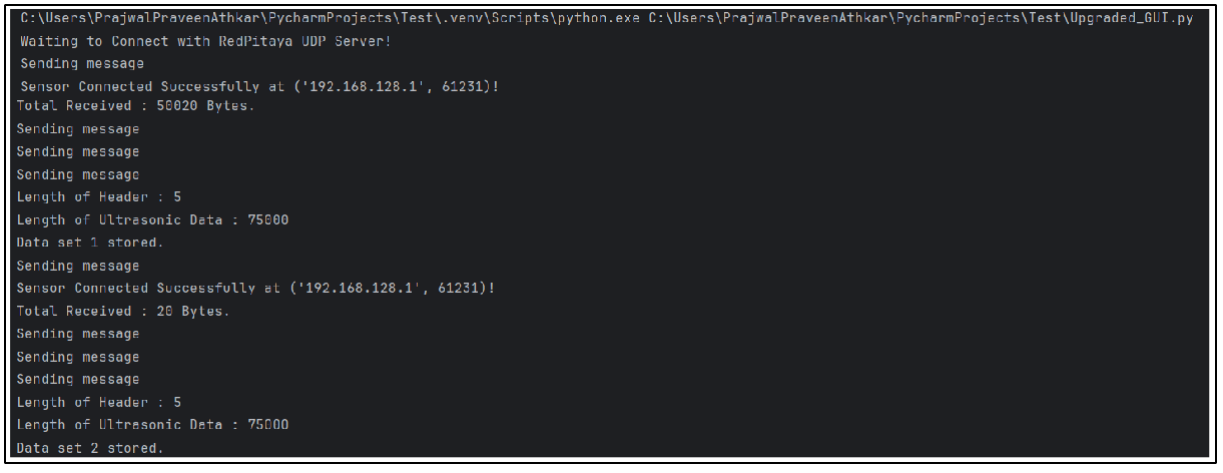


*Figure 3.30: Sample snippet of the run-time console when **Upgraded_GUI.py** is launched in Pycharm IDE.*

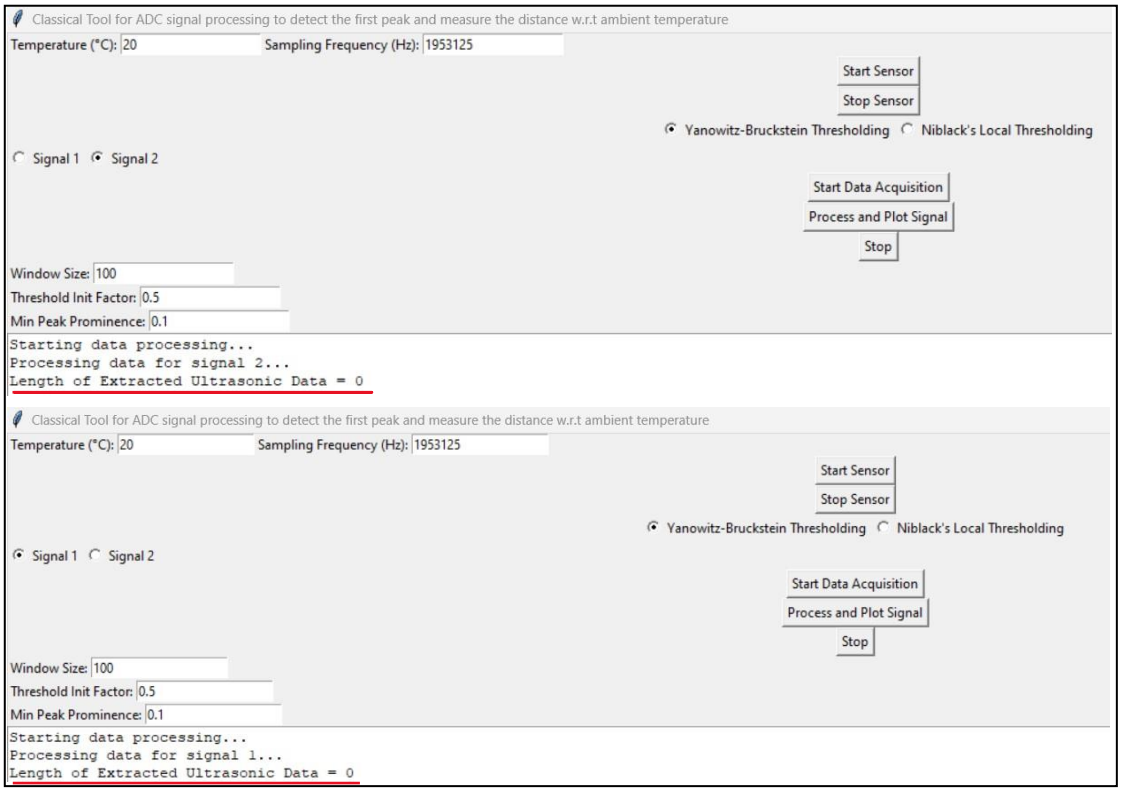


Figure 3.31a and Figure 3.31b*: Snippet of the upgraded GUI after data acquisition is completed and length of the extracted signal data is displayed to be 0 for signal 1 and signal 2.*

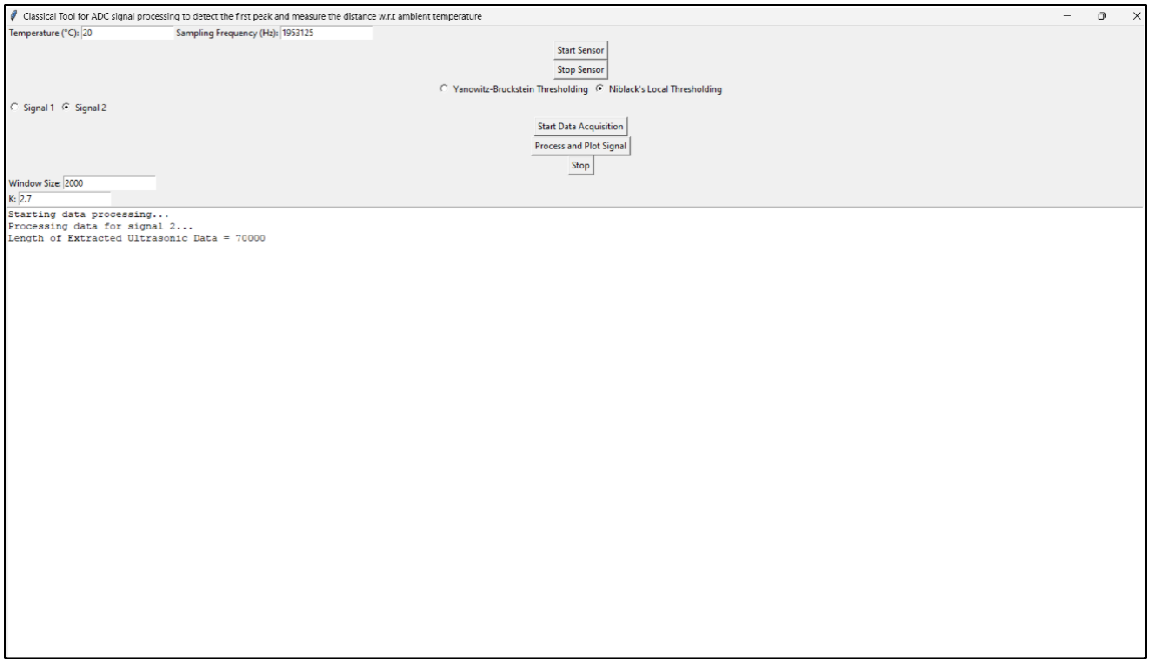


Figure 3.32: Snippet of the upgraded GUI when the user clicks 'Process and Plot Signal' to perform signal processing and a lag is observed while displaying the results along with the signal plot.

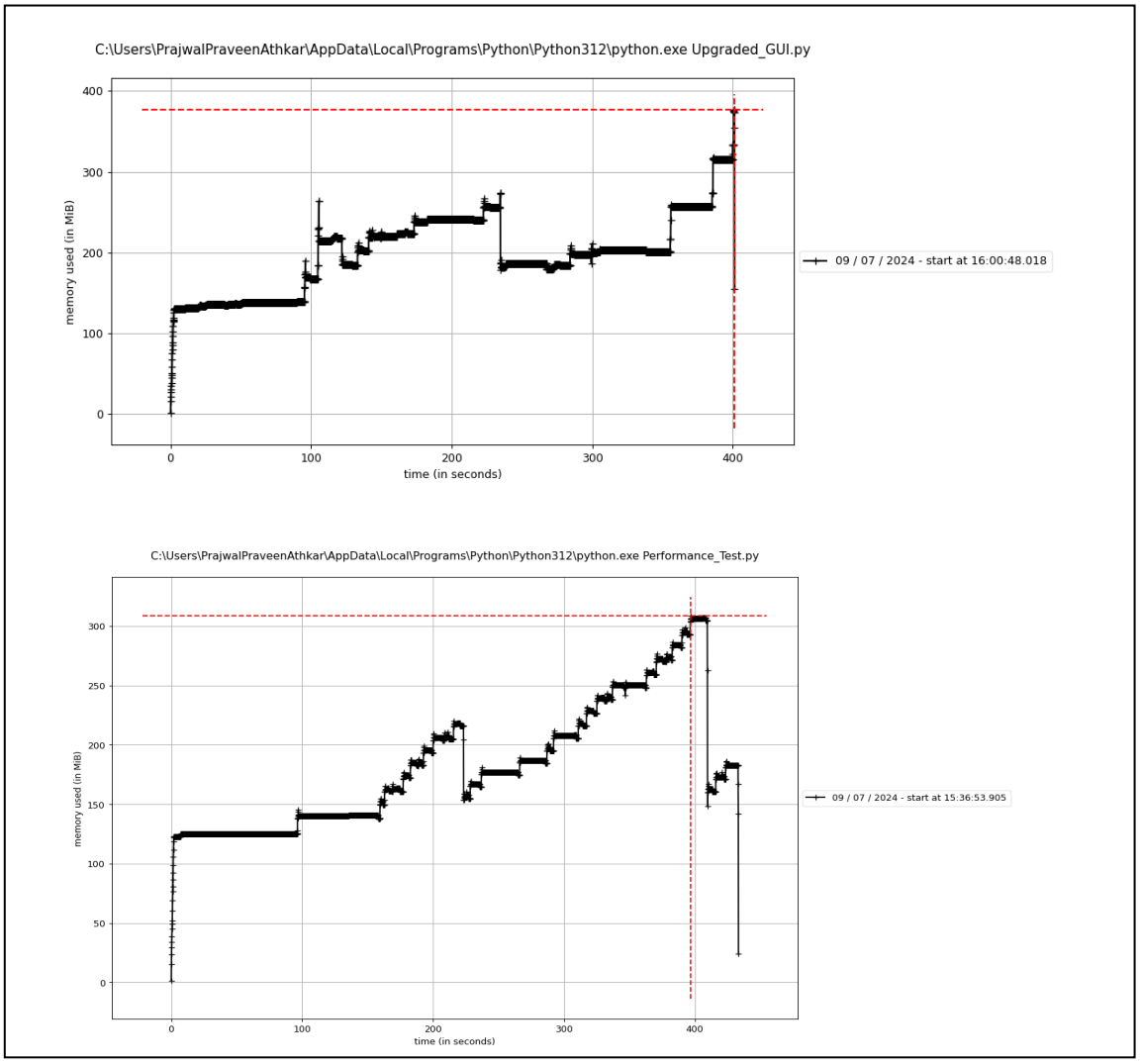


*Figure 3.33 above and Figure 3.34 below: Snippet of the memory profiler plot for **Upgraded_GUI.py** above with peak value 376.555 MiB and **Classical_GUI.py** below with peak value 308.906 MiB.*

3.8.2 Advantages and Future Scope for the Upgraded GUI

- The ***ultrasonic_data_extrator_upgraded.py*** upgraded script provides an enhancement to the GUI to have a data acquisition capability to receive upto 50020 bytes, containing 5 instances of header information and 75000 instances of ultrasonic data. This makes the GUI flexible to receive a higher value of signal data.

- The script ***Upgraded_GUI.py*** has the capability to process the acquired signal data of 50020 bytes, and perform further signal processing on the ultrasonic data to determine the first peak index and accurately measure the distance of the person/object from the ultrasonic sensor based on the ambient temperature and sampling frequency of the ultrasonic sensor (refer Table 3.2).

- To make the ***Upgraded_GUI.py*** work efficiently as the ***Classical_GUI.py***, the ***ultrasonic_data_extrator_upgraded.py*** script needs to be corrected to efficiently acquire large amount of signal data from the ultrasonic sensor within the required time mentioned as 2 signals acquired per second. And also, the GUI script ***Upgraded_GUI.py*** must be corrected to handle and further process large input signal data efficiently without any delay or lag in rendering the results. This will solve the drawbacks mentioned earlier.

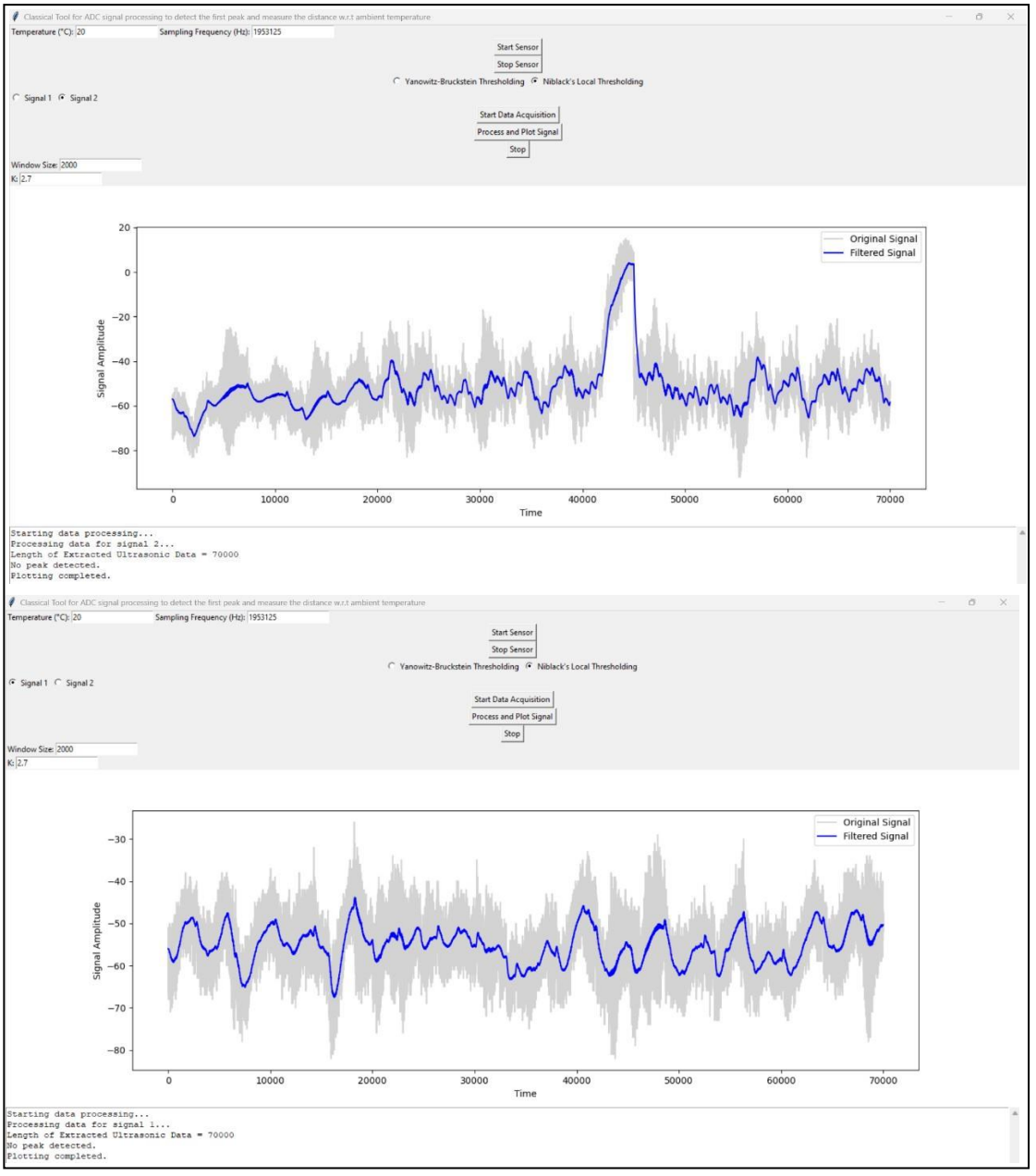


Figure 3.35 above and Figure 3.36 below: Snippet of the upgraded GUI when the user clicks 'Process and Plot Signal' to perform signal processing and there is noisy signal observed in Signal 1 and Signal 2, due to which the peak is not detected.

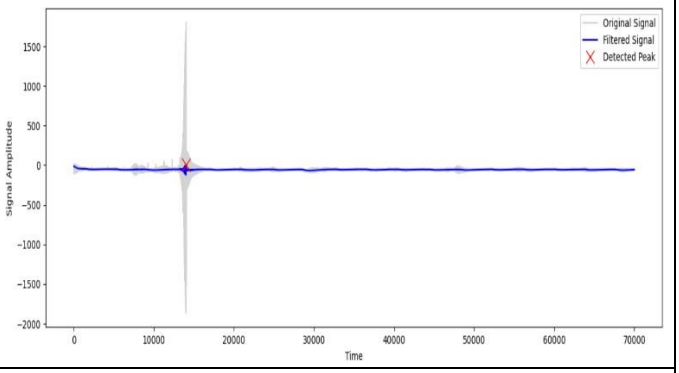
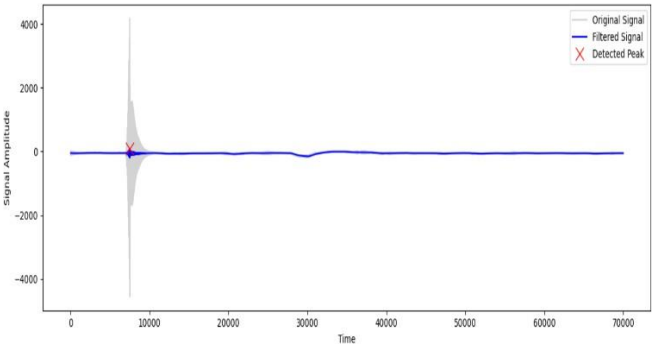
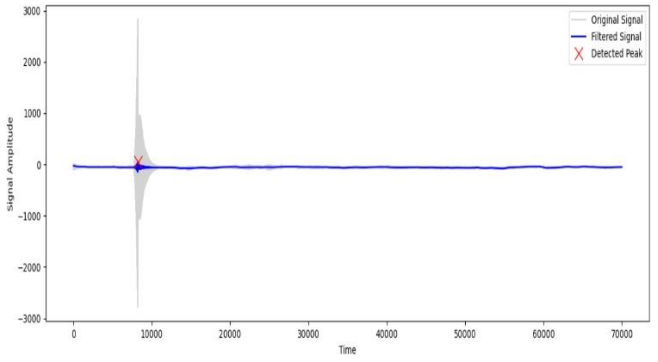
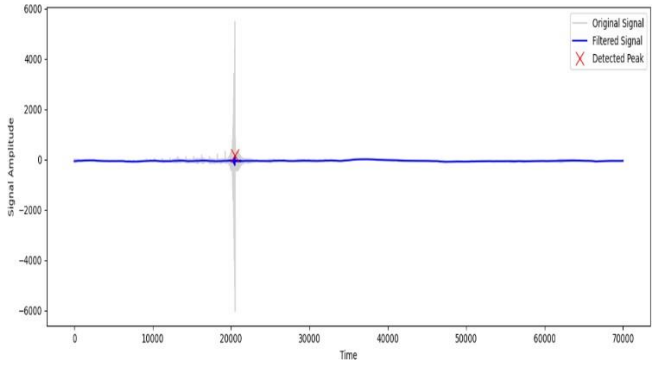
Real-Time Test Instances	Measured distance from the tested signal	Output plot of the tested signal
1	First peak detected at index: 14042, corresponding to a distance of 1.24 meters.	
2	First peak detected at index: 7510, corresponding to a distance of 0.66 meters.	
3	First peak detected at index: 8252, corresponding to a distance of 0.73 meters.	
4	First peak detected at index: 20498, corresponding to a distance of 1.80 meters.	

Table 3.2: Distance measurement results and signal plot rendered by the Upgraded GUI.

4. Realisation

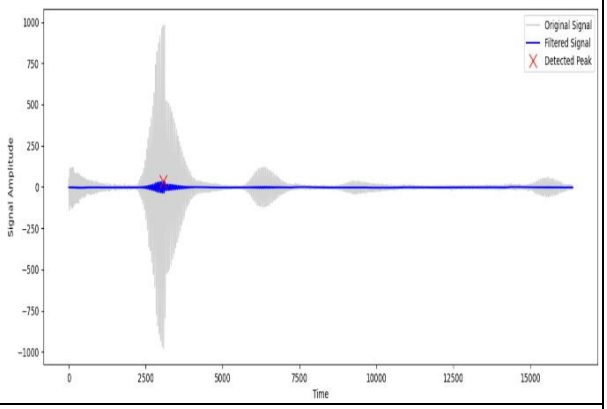
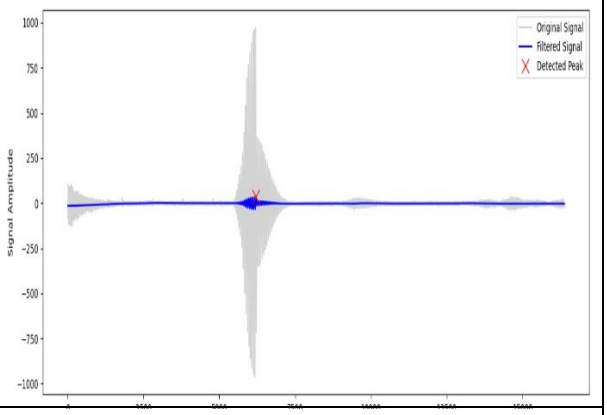
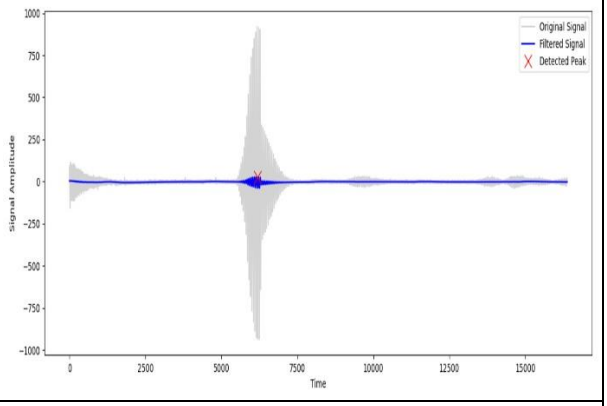
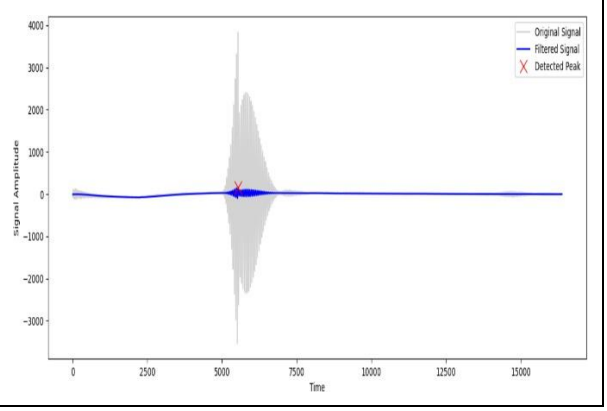
In this research, the primary focus was to develop a real-time robust distance measurement GUI software which measures the distance between the ultrasonic sensor and a person or object based on the position of the first peak when the ultrasound strikes the person or object, the ambient temperature, and the sampling frequency of the ultrasonic sensor. Synchronization between these two systems should also be maintained to retain the accuracy of the distance measurement system. Through this integrated approach, the research endeavors to contribute to the advancement of real-time robust distance measurement technology for prolonged real-world applications.

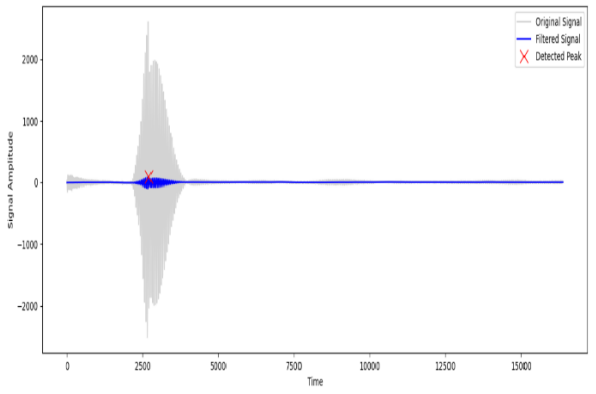
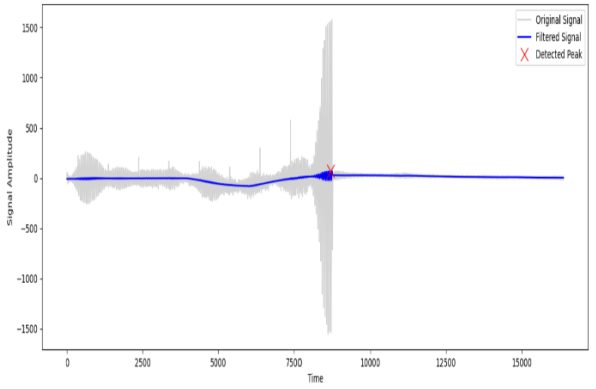
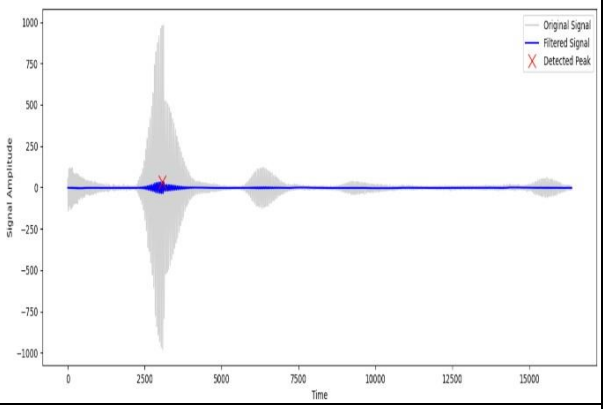
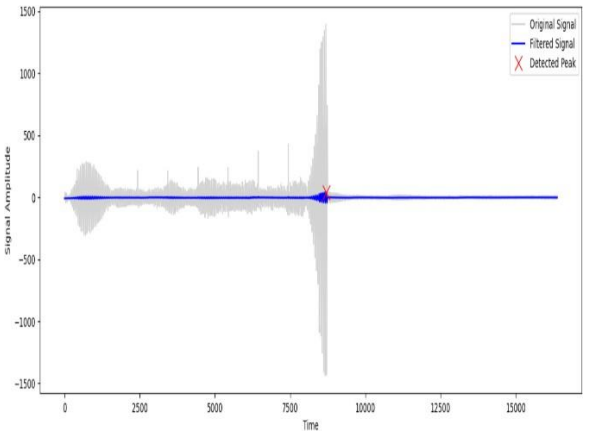
The 2 GUI models developed, the first one being ***Classical_GUI.py*** which successfully passes all the requirements of the performance metrics for the GUI software, and which has a signal data acquiring capability of total length 32832 bytes, in which 16 instances contain header information and 16384 instances contain ultrasonic data based on the script ***ultrasonic_data_extrator.py***. As an enhancement to this GUI software, ***Upgraded_GUI.py*** was developed where I have increased the data acquisition capability to receive up to 50020 bytes, containing 5 instances of header information and 75000 instances of ultrasonic data, the ***ultrasonic_data_extrator_upgraded.py*** upgraded script in this case outlines a Python class RedPitayaSensor for controlling the RedPitaya over UDP to acquire the signal data and using SSH. But the Upgraded_GUI.py GUI has failed to pass the requirements of the performance metrics for the GUI software, as mentioned in Section 3.8.1.

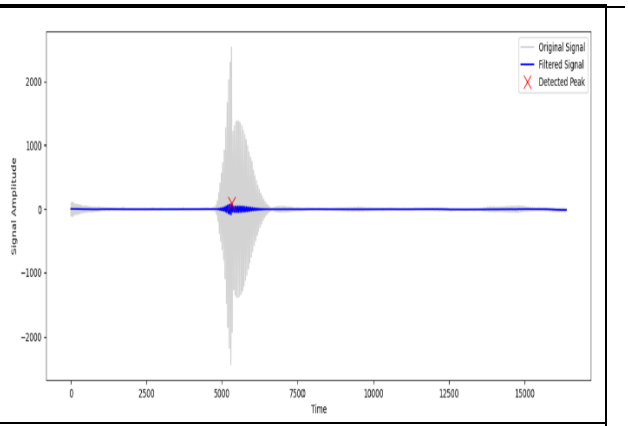
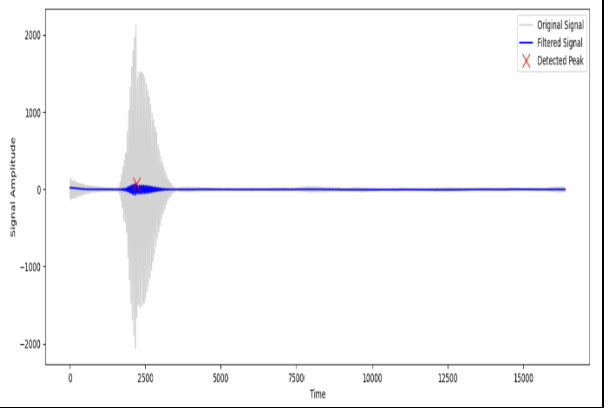
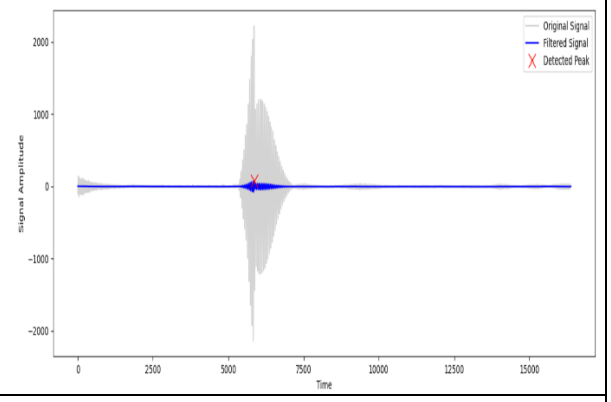
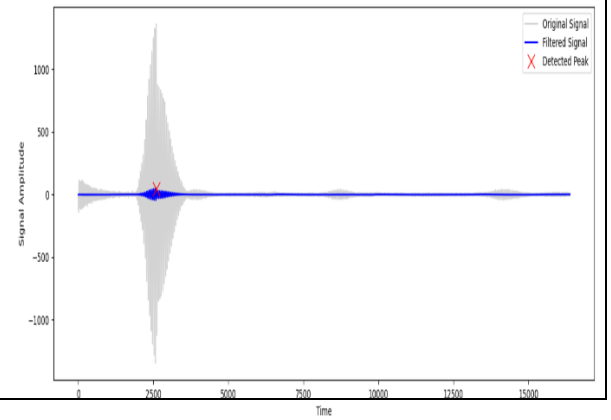
Hence, the GUI rendered by the script ***Classical_GUI.py*** is considered for testing, analysis, and comparison with the best configured LSTM neural network model for occupancy detection in smart office environments developed by (Aishin, 2024). The GUI has undergone extensive and rigorous testing based on the performance metrics for the GUI software. The developed GUI demonstrated real-time robust performance with good accuracy and signal data handling, provided that the devices were connected to a stable power source, marking a significant stride towards a more efficient and user friendly application of ultrasonic distance calculator.

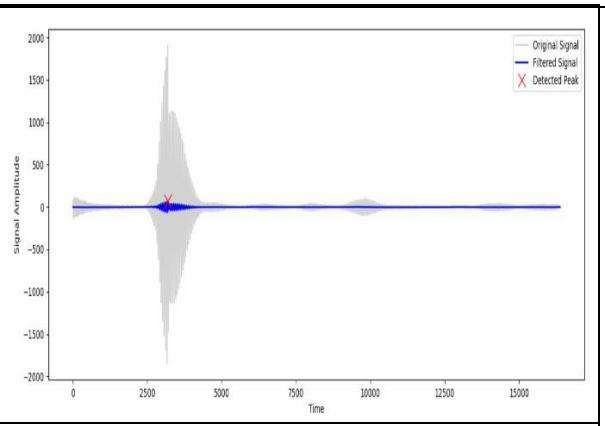
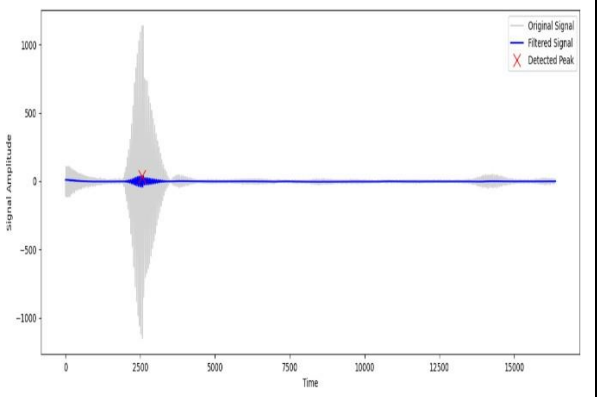
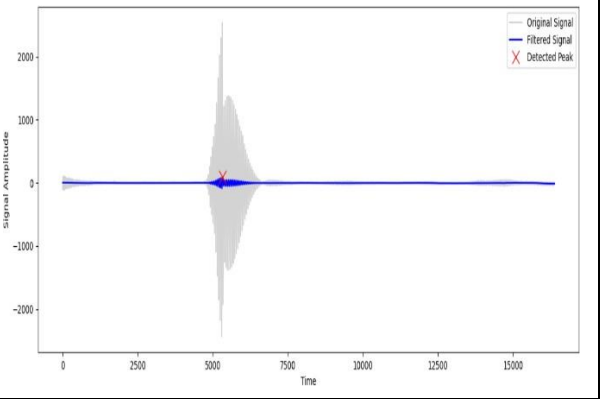
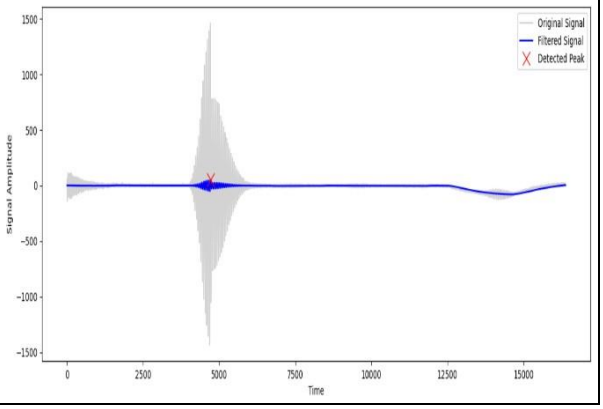
4.1 Evaluation of the Classical GUI based on Performance Metrics

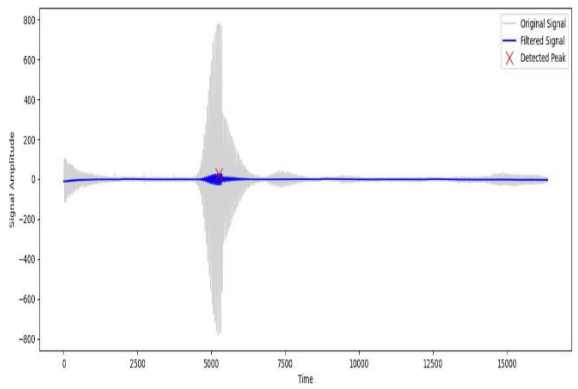
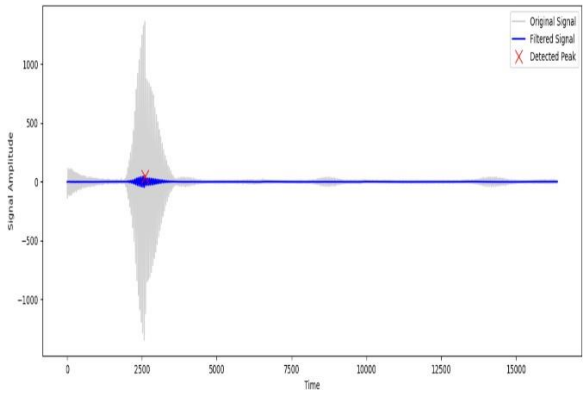
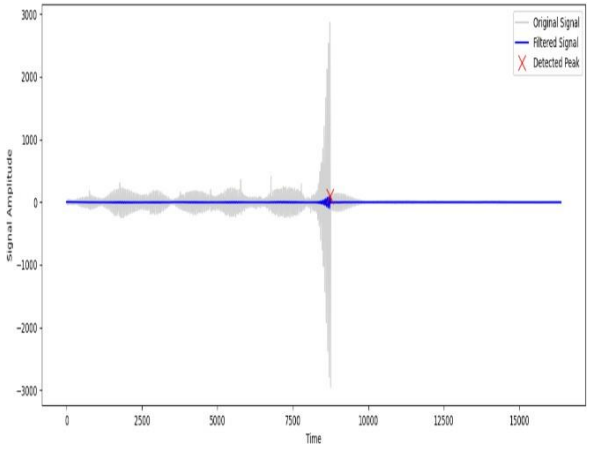
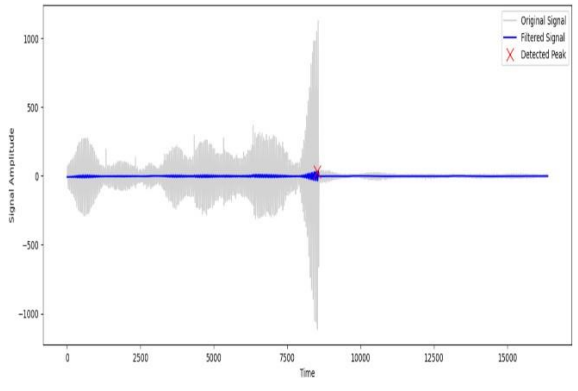
The GUI was tested rigorously for over 100 iterations during real-time based on the performance metrics for the GUI software, accuracy is the extent to which the program is effective in its designated tasks as measured through the difference between the expected and the observed results. Precision relates to the reliability of a software program tested by performing similar operations several times to check whether the results are similar. Stress testing define robustness as the software's capacity to function without problems under unforeseen circumstances. Repetition rate determines how effectively the software is going to perform repetitive tasks, used for frequent or batch, determined by scripting of similar tasks. Memory utilization is how much RAM is consumed and has to be profiled to ensure the utilization of the available resources. Real time performance includes the ability to execute various tasks and operations as fast as possible and if the user interface performance is in question, then response time, frame rate and throughput are areas of interest under different conditions. The sample experimental results and signal plots are shown below in Table 4.1.

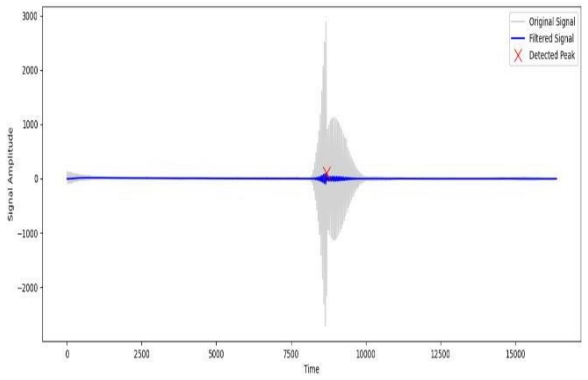
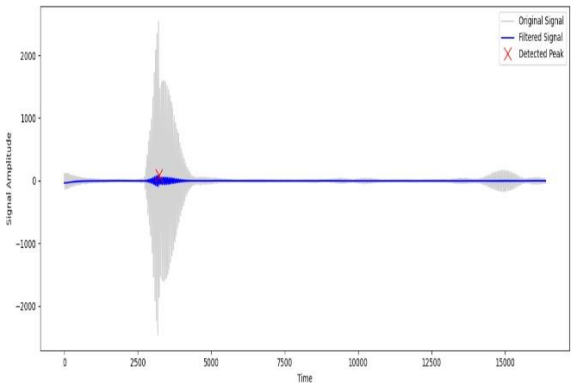
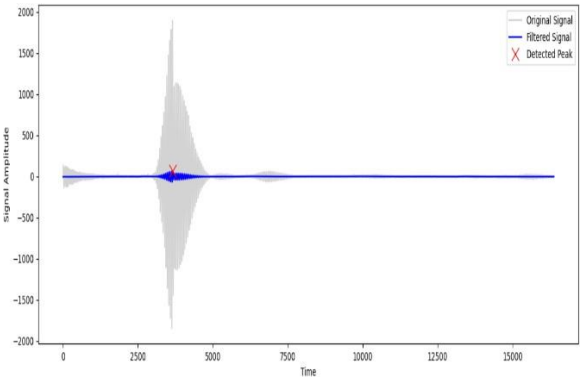
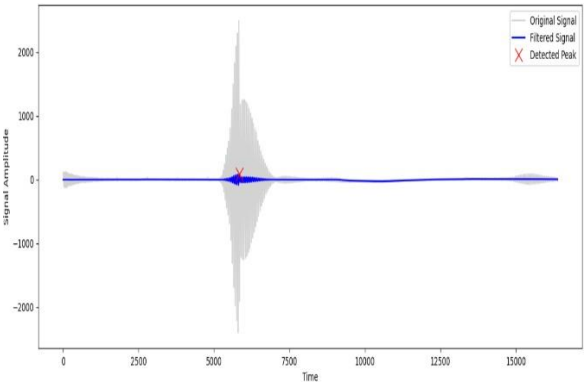
Input Test Signal	Algorithm 1: Yanowitz-Bruckstein Adaptive local Thresholding	Algorithm 2: Niblack's Local Thresholding	Output Plot of the Tested Signal
1	First peak detected at index: 3073, corresponding to a distance of 0.27 meters.	First peak detected at index: 3073, corresponding to a distance of 0.27 meters.	
2	First peak detected at index: 6212, corresponding to a distance of 0.55 meters.	First peak detected at index: 6212, corresponding to a distance of 0.55 meters.	
3	First peak detected at index: 6186, corresponding to a distance of 0.54 meters.	First peak detected at index: 6186, corresponding to a distance of 0.54 meters.	
4	First peak detected at index: 5546, corresponding to a distance of 0.49 meters.	First peak detected at index: 5546, corresponding to a distance of 0.49 meters.	

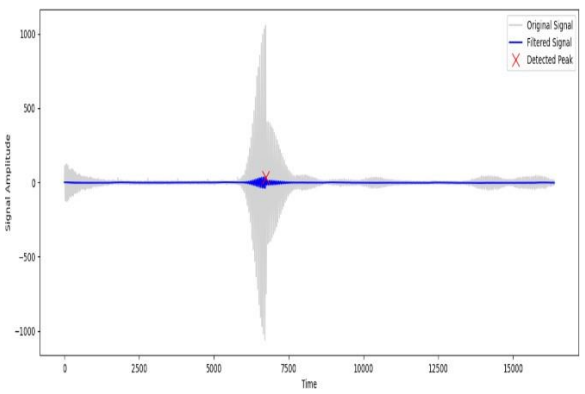
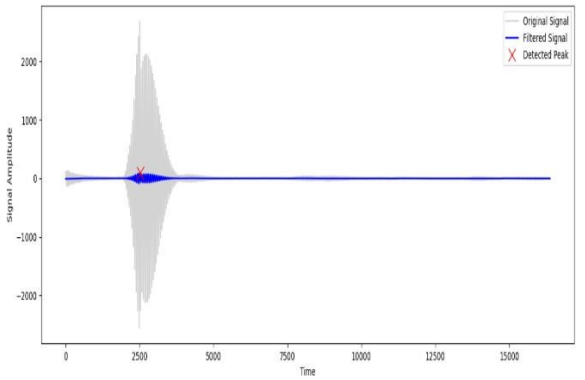
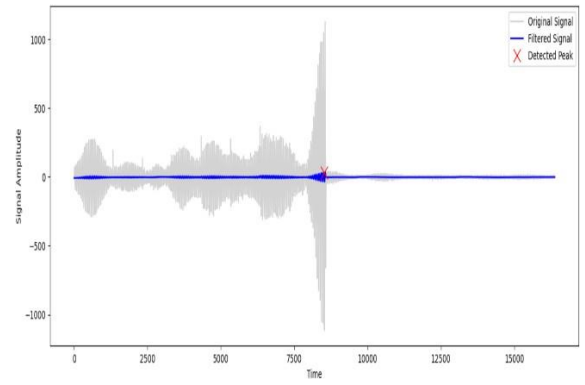
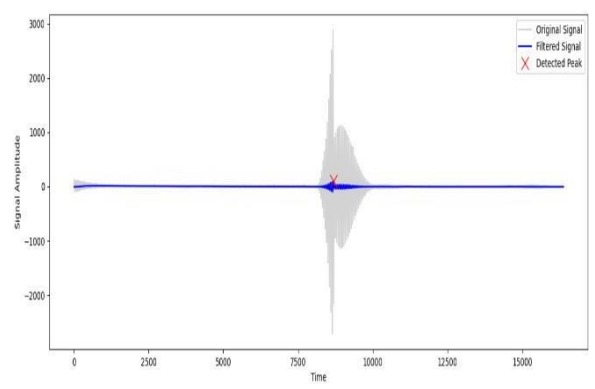
5	First peak detected at index: 2697, corresponding to a distance of 0.24 meters.	First peak detected at index: 2697, corresponding to a distance of 0.24 meters.	
6	First peak detected at index: 8690, corresponding to a distance of 0.76 meters.	No peak detected.	
7	First peak detected at index: 3073, corresponding to a distance of 0.27 meters.	First peak detected at index: 3073, corresponding to a distance of 0.27 meters.	
8	First peak detected at index: 8701, corresponding to a distance of 0.77 meters.	First peak detected at index: 8701, corresponding to a distance of 0.77 meters.	

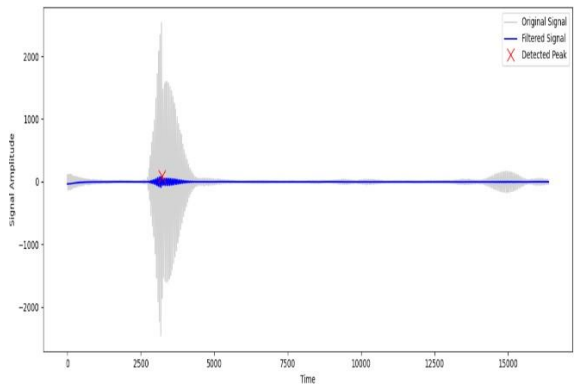
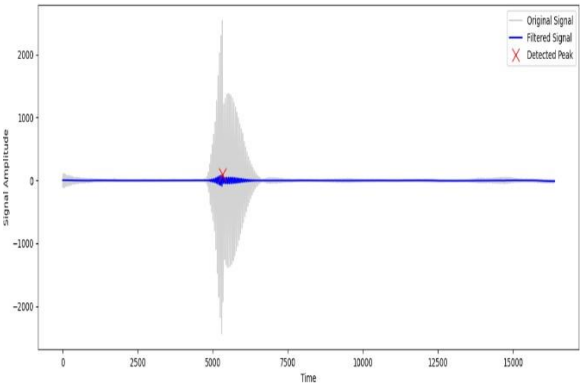
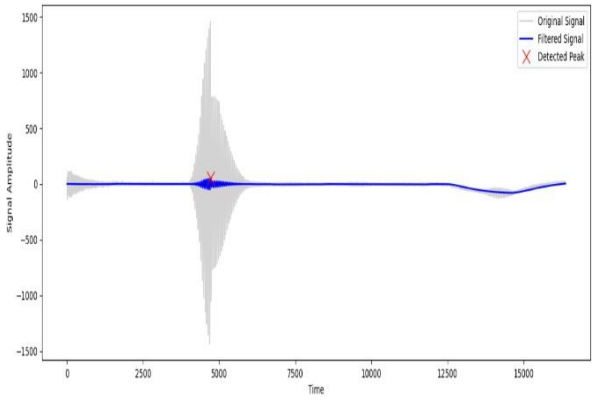
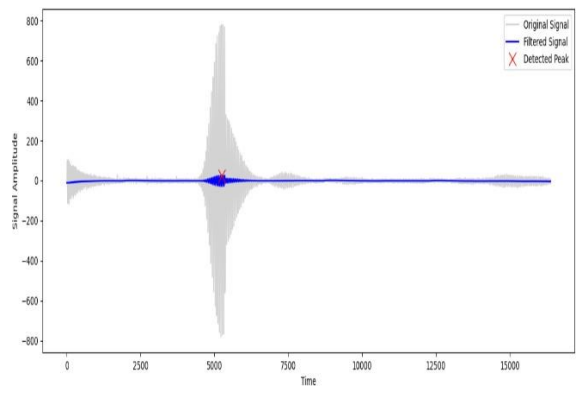
9	First peak detected at index: 5332, corresponding to a distance of 0.47 meters.	First peak detected at index: 5332, corresponding to a distance of 0.47 meters.	
10	First peak detected at index: 2202, corresponding to a distance of 0.19 meters.	First peak detected at index: 2202, corresponding to a distance of 0.19 meters.	
11	First peak detected at index: 5859, corresponding to a distance of 0.52 meters.	First peak detected at index: 5859, corresponding to a distance of 0.52 meters.	
12	First peak detected at index: 2608, corresponding to a distance of 0.23 meters.	First peak detected at index: 2608, corresponding to a distance of 0.23 meters.	

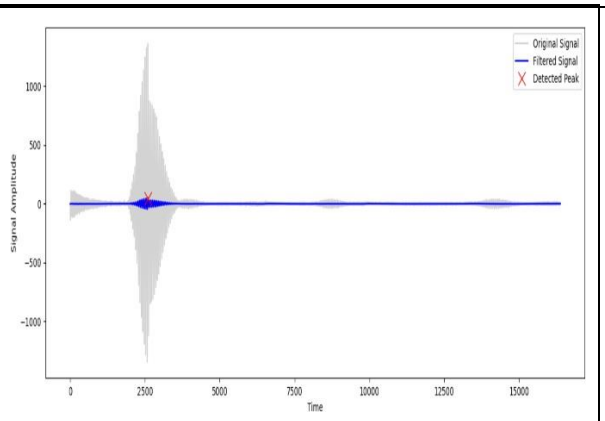
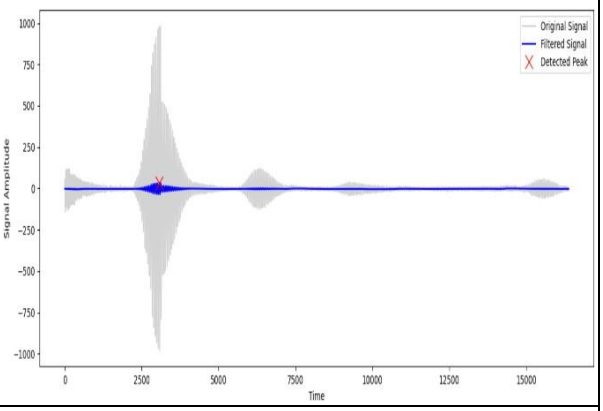
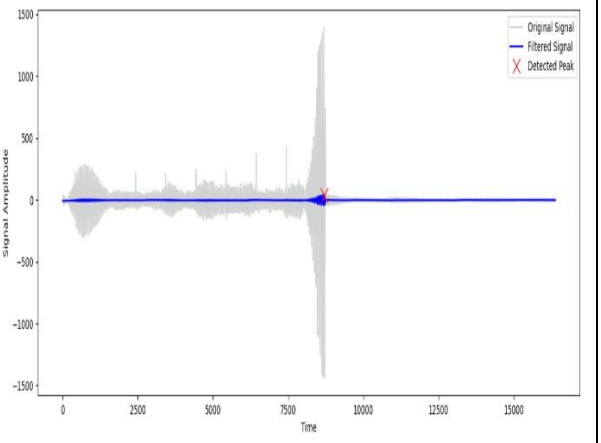
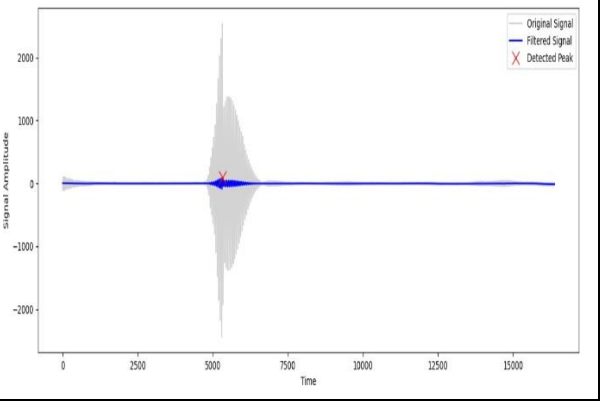
13	First peak detected at index: 3200, corresponding to a distance of 0.28 meters.	First peak detected at index: 3200, corresponding to a distance of 0.28 meters.	
14	First peak detected at index: 2556, corresponding to a distance of 0.22 meters.	First peak detected at index: 2556, corresponding to a distance of 0.22 meters.	
15	First peak detected at index: 5332, corresponding to a distance of 0.47 meters.	First peak detected at index: 5332, corresponding to a distance of 0.47 meters.	
16	First peak detected at index: 4725, corresponding to a distance of 0.42 meters.	First peak detected at index: 4725, corresponding to a distance of 0.42 meters.	

17	First peak detected at index: 5257, corresponding to a distance of 0.46 meters.	First peak detected at index: 5257, corresponding to a distance of 0.46 meters.	
18	First peak detected at index: 2608, corresponding to a distance of 0.23 meters.	First peak detected at index: 2608, corresponding to a distance of 0.23 meters.	
19	First peak detected at index: 8741, corresponding to a distance of 0.77 meters.	First peak detected at index: 8741, corresponding to a distance of 0.77 meters.	
20	First peak detected at index: 8517, corresponding to a distance of 0.75 meters.	First peak detected at index: 8517, corresponding to a distance of 0.75 meters.	

21	First peak detected at index: 8681, corresponding to a distance of 0.76 meters.	First peak detected at index: 8681, corresponding to a distance of 0.76 meters.	
22	First peak detected at index: 3217, corresponding to a distance of 0.28 meters.	First peak detected at index: 3217, corresponding to a distance of 0.28 meters.	
23	First peak detected at index: 3669, corresponding to a distance of 0.32 meters.	First peak detected at index: 3669, corresponding to a distance of 0.32 meters.	
24	First peak detected at index: 5835, corresponding to a distance of 0.51 meters.	First peak detected at index: 2797, corresponding to a distance of 0.25 meters.	

25	First peak detected at index: 6740, corresponding to a distance of 0.59 meters	First peak detected at index: 6740, corresponding to a distance of 0.59 meters	
26	First peak detected at index: 2521, corresponding to a distance of 0.22 meters.	First peak detected at index: 2521, corresponding to a distance of 0.22 meters.	
27	First peak detected at index: 8517, corresponding to a distance of 0.75 meters.	First peak detected at index: 8517, corresponding to a distance of 0.75 meters.	
28	First peak detected at index: 8681, corresponding to a distance of 0.76 meters.	First peak detected at index: 8681, corresponding to a distance of 0.76 meters.	

29	First peak detected at index: 3217, corresponding to a distance of 0.28 meters.	First peak detected at index: 3217, corresponding to a distance of 0.28 meters.	
30	First peak detected at index: 5332, corresponding to a distance of 0.47 meters.	First peak detected at index: 5332, corresponding to a distance of 0.47 meters.	
31	First peak detected at index: 4725, corresponding to a distance of 0.42 meters.	First peak detected at index: 4725, corresponding to a distance of 0.42 meters.	
32	First peak detected at index: 5257, corresponding to a distance of 0.46 meters.	First peak detected at index: 5257, corresponding to a distance of 0.46 meters.	

33	First peak detected at index: 2608, corresponding to a distance of 0.23 meters.	First peak detected at index: 2608, corresponding to a distance of 0.23 meters.	
34	First peak detected at index: 3073, corresponding to a distance of 0.27 meters.	First peak detected at index: 3073, corresponding to a distance of 0.27 meters.	
35	First peak detected at index: 8701, corresponding to a distance of 0.77 meters.	First peak detected at index: 8701, corresponding to a distance of 0.77 meters.	
36	First peak detected at index: 5332, corresponding to a distance of 0.47 meters.	First peak detected at index: 5332, corresponding to a distance of 0.47 meters.	

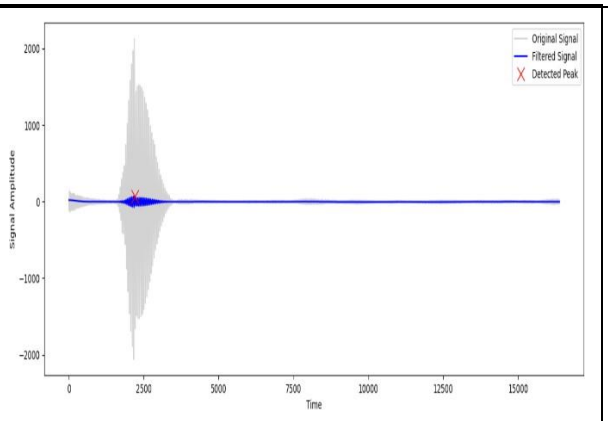
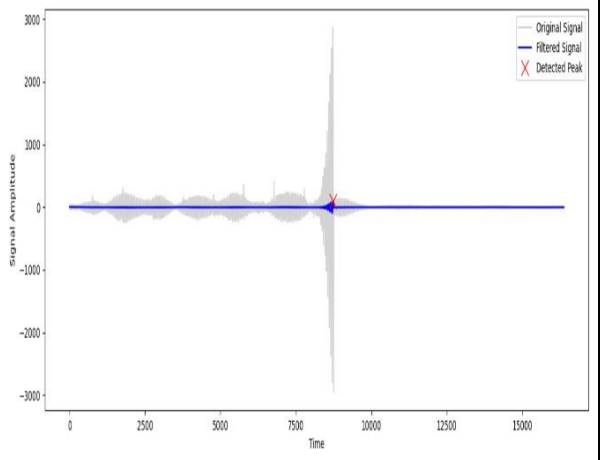
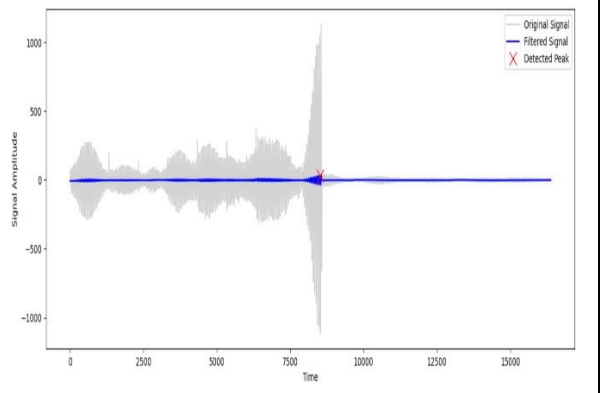
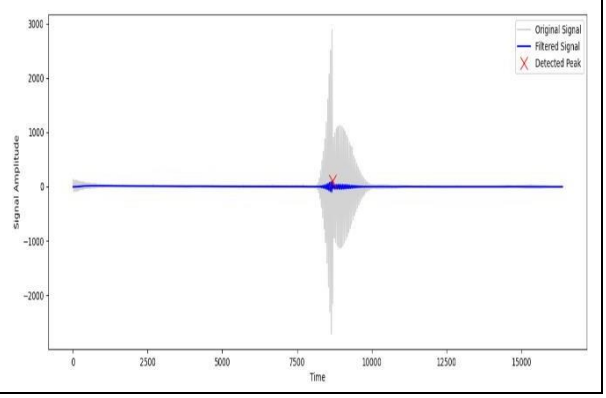
37	First peak detected at index: 2202, corresponding to a distance of 0.19 meters.	First peak detected at index: 2202, corresponding to a distance of 0.19 meters.	
38	First peak detected at index: 8741, corresponding to a distance of 0.77 meters.	First peak detected at index: 8741, corresponding to a distance of 0.77 meters.	
39	First peak detected at index: 8517, corresponding to a distance of 0.75 meters.	First peak detected at index: 8517, corresponding to a distance of 0.75 meters.	
40	First peak detected at index: 8681, corresponding to a distance of 0.76 meters.	First peak detected at index: 8681, corresponding to a distance of 0.76 meters.	

Table 4.1: Distance measurement results and signal plot rendered by the Classical GUI.

4.1.1 Evaluation and Performance Analysis

The evaluation and performance analysis of the GUI software rendered by ***Classical_GUI.py*** is done based on the performance metrics for the GUI software (refer Section 3.8.1), the details and results are as follows:

- Accuracy refers to how successfully the program performs its intended functions and this is evaluated through a python script, it calculates accuracy by determining the proportion of measurements that fall within a **tolerance range of 0.01** from the values. It employs np.isclose to generate an array computes the average and converts it into a percentage (refer Figure 4.1 & Figure 4.2).

- On the other hand precision means the consistency and exactness of the software's responses and actions, this can be evaluated by comparing positives to the total sum of true positives and false positives (refer Figure 4.1 & Figure 4.3). In this context true positives represent measurements falling within the tolerance range while false positives indicate measurements outside this range.

- Memory consumption refers to the quantity of RAM utilized by the software throughout its operation, which may be evaluated using profiling tools to measure and evaluate memory usage patterns, ensuring that the software stays within bounds and does not consume excessive system resources. This can be evaluated by running the script ***Classical_GUI.py*** executed with a memory profiler in Python 3.12, this gives a picture on memory usage plotted as a function of time. Time is represented on the x-axis in seconds while y-axis represents memory usage measured in MiB (Mebibytes). The graph shows step-like increases indicating that the script only allocates memory at certain stages and not continuously. Every step indicates an important change in allocation of memory then it stabilizes for some period of time. The memory usage peaks just above 308.906 MiB, as indicated by the red dashed horizontal line, which represents a memory peak. . On this plot, you can see a red dashed vertical line that delimits around 400 seconds, as if it's something like maximum memory use point followed by a decline in use which may imply stopped executing the script or program (memory discontinuity)(refer Figure 4.4).

- Real-time performance, repetition rate, and robustness of the GUI was evaluated based on the stress testing, user experience and by monitoring the software's performance over time to ensure no latency or loss of responsiveness, if the software can withstand unexpected conditions without crashing or giving incorrect results, and if software can do tasks and respond to inputs in a reasonable time (refer Figure 4.5a & 4.5b). Based on the performance evaluation results provided in this section, it is observed that the application performed very well and turned out to be an application appropriate for real-time robust ultrasonic distance measurement (refer Table 4.1).

```
def calculate_accuracy(actual, calculated):
    accuracy = 100 * np.mean(np.isclose(actual, calculated, atol=0.01))
    return accuracy


1 usage
def calculate_precision(actual, calculated):
    true_positive = np.sum(np.isclose(actual, calculated, atol=0.01))
    false_positive = np.sum(~np.isclose(actual, calculated, atol=0.01))
    precision = true_positive / (true_positive + false_positive) if (true_positive + false_positive) > 0 else 0
    return precision
```

Figure 4.1: Code snippet of the accuracy calculation and precision calculation functions used for the GUI evaluation.

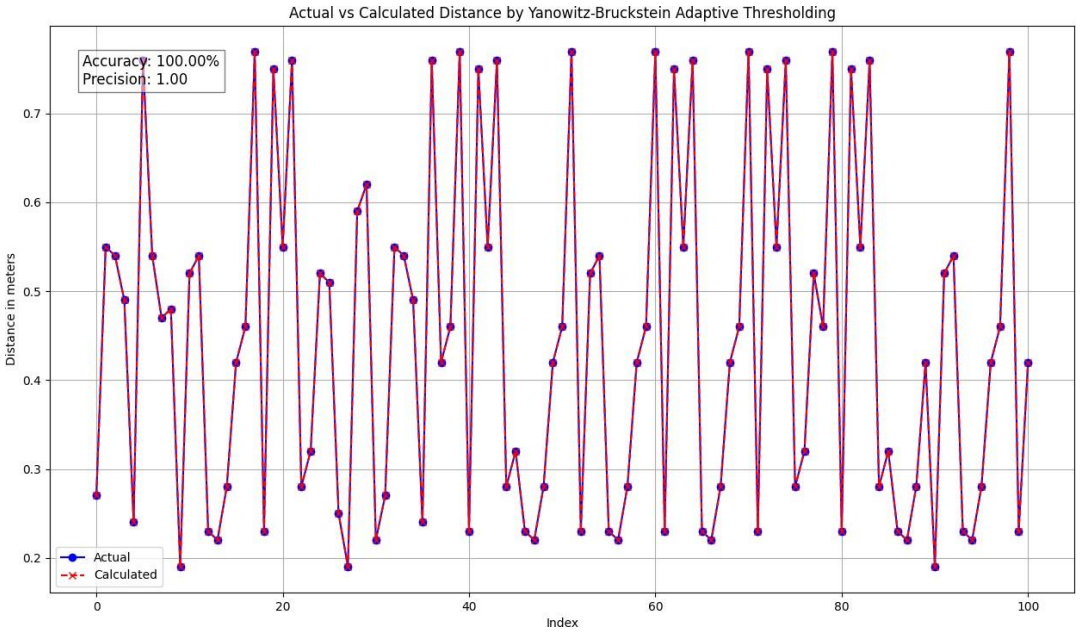


*Figure 4.2: Accuracy and precision plot based on the actual Vs calculated distance by Yanowitz-Bruckstein Adaptive Thresholding, which yields an **accuracy of 100.00%** and a **precision of 1.00**.*

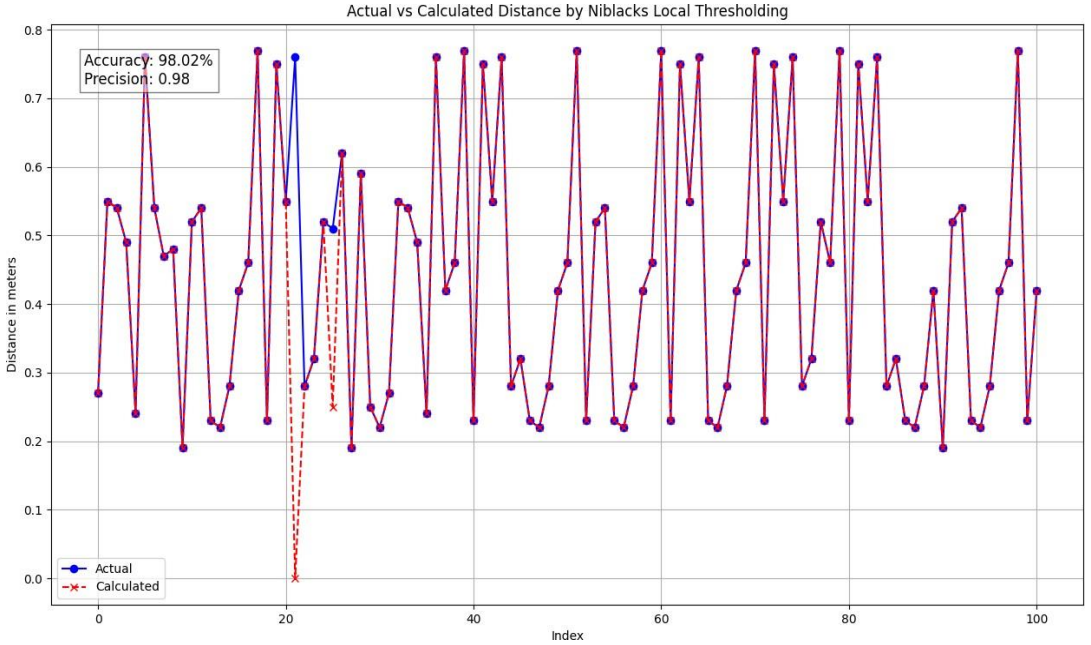


*Figure 4.3: Accuracy and precision plot based on the actual Vs calculated distance by Niblack's Local Thresholding, which yields an **accuracy of 98.02%** and a **precision of 0.98**.*

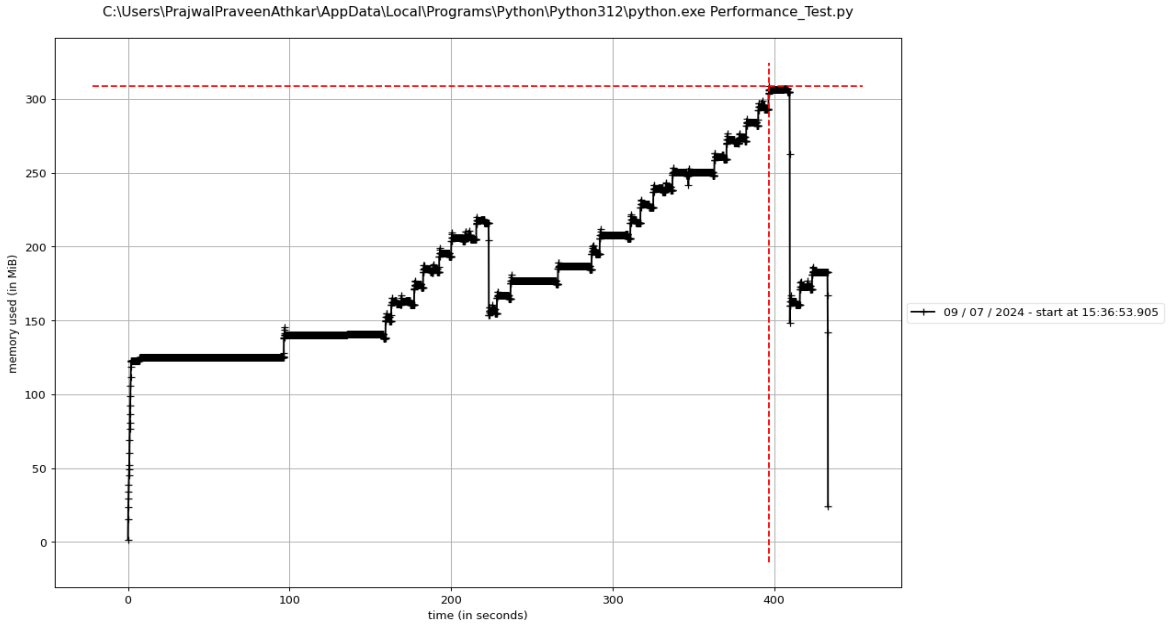


*Figure 4.4: Snippet of the memory profiler plot for **Classical_GUI.py** with a peak value of **308.906 MiB**.*

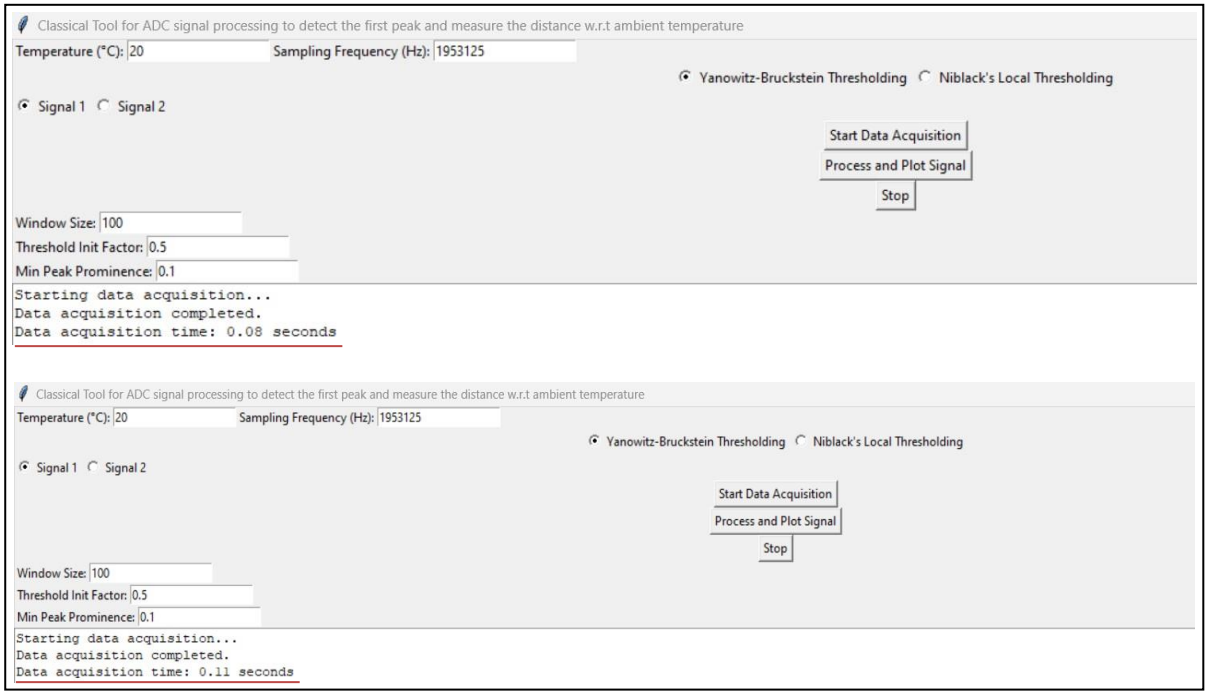


*Figure 4.5a above and Figure 4.5b below: Snippet of the classical GUI after data acquisition is completed and time taken to acquire the data is displayed as **0.08 seconds** and **0.11 seconds**.*

4.1.2 Future Scope of the Classical GUI

- I changed ***Classical_GUI.py*** and its data extraction script ***ultrasonic_data_extrator.py*** to be able to accept up to 50020 bytes of data, which includes 5 instances of header information and 75000 instances of ultrasonic data. This enables the GUI to receive a higher value of signal data. The changed GUI script is ***Upgraded_GUI.py***, and the data acquisition script is ***ultrasonic_data_extractor_upgraded.py***, although the upgrade created to gather huge signal data has downsides and makes the GUI problematic (refer Section 3.8.1).

- To make the upgraded GUI work efficiently as the Classical GUI, the ***ultrasonic_data_extrator_upgraded.py*** script needs to be corrected to efficiently acquire large amount of signal data from the ultrasonic sensor within the required time mentioned as 2 signals acquired per second. And also, the GUI script Upgraded_GUI.py must be corrected to handle and further process large input signal data efficiently without any delay or lag in rendering the results. This will solve the drawbacks mentioned earlier.

- Once the upgraded GUI is stable and working well, it can further be used as a real-time robust ultrasonic data collection and distance measurement software, which saves the captured signal data along with the measured distance between the ultrasonic sensor and the person/object in a .csv file with a label of 'p' if a person was detected or 'o' if an object was detected.

4.2 Analysis and Comparison of the Developed LSTM Neural Network and other RNNs

The experiment performed by (Aishin, 2024) on developing the best RNN for the occupancy detection is used here for comparison with the Classical GUI model developed by me, this section contains the details of the occupancy detection model using various RNN architectures and its evaluation results based on the performance metrics (refer Section 2.8 & 2.9), which includes LSTM, GRU, and BiLSTM. The data for this research were obtained from an auxiliary system (Pongsomboon, 2022) which includes a Red Pitaya with an ultrasonic sensor as the central equipment. This setup, which includes a GUI interface, captured FFT data for two categories: “Person Sitting on Chair or Occupied Seat” and “Empty Chair or Unoccupied Seat.” The system created folders in binary, images, and JSON; however, for this study, the binary files were used. The main operations included using the YOLO model to identify labels in the filenames of FFT information from the ultrasonic sensors saved in the NumPy format, transforming these files into the FFT format, and combining the data into a single CSV file. With regard to data collection, it was possible to obtain a dataset of about 137000 records with 86 features per record. From the filenames, labels were derived as to whether data was about an occupant “p” or a non-occupant “o” or “c” based on the FFT data with timestamps to the visual data. This process served to correctly label into occupant and non-occupant categories the initial NumPy files, to transform them into FFT format and to assemble all the data into a single CSV file. For the best occupancy detection model, several RNN structures, layers, and learning rate were analyzed and compared. The initial step involved evaluating three RNN variants: The different types of the recurrent neural networks used are LSTM, GRU, and BiLSTM with 3 layers, 64 units in each layer and learning rate being 0.001, and a dropout rate of 0.5. These models were evaluated based on the measures like accuracy, ROC AUC, precision, recall, and F1 score. The results are presented in the Table 4.2, showed that LSTM achieved higher results than GRU and BiLSTM in all the aspects, especially in accuracy and ROC AUC. This meant that LSTM was better at capturing temporal dependency in the data and therefore it was the one that was optimized further.

NN Model	Accuracy	ROC AUC	Loss	Precision	Recall	F1 Score
LSTM	0.9817	0.9814	0.0630	0.9802	0.9858	0.9830
GRU	0.9726	0.9721	0.0872	0.9701	0.9790	0.9745
BiLSTM	0.9768	0.9764	0.0765	0.9758	0.9810	0.9784

Table 4.2: Performance Comparison of RNN Architectures (Aishin, 2024).

The details of the final LSTM model architecture for occupancy detection model is as follows: The input sequence is this feature vectors from each of the time steps x_i. There are 3 LSTM layers and each of them is of 64 units; it takes the input sequence as one time step and then passes the hidden state to the next time step. To minimize the degree of overfitting, the model incorporates the dropout layers with dropout rate equals to 0.5 is appended to each LSTM layer where a certain percentage of input units is masked to zero while training. The last layer is a dense layer which, after taking the output of the last LSTM layer, applies the sigmoid function to it and then produces only one output. This model is trained using the Adam optimizer and the learning rate is set to 0.001, with batch size of 32, over 50 epochs, the loss function that was used was binary cross entropy. This allowed to find the best balance between model complexity and performance, which led to the obtaining of a very efficient occupancy detection model. To sum up, based on the comparison of several RNN models and configurations such as layers and learning rate, the best RNN model to predict occupancy detection is LSTM with 3 layers with the learning rate being 0.001 was identified to be the best for the optimization of the model. This procedure ensured that the model was precise and effective in capturing the needed time relations in the data set (refer Table 4.3).

Learning Rate	Accuracy	ROC AUC	Precision	Recall	F1 Score
0.001	0.9817	0.9814	0.9802	0.9858	0.9830

Table 4.3: Final LSTM architecture performance evaluation results with learning rate 0.001 (Aishin, 2024).

4.2.1 Performance Analysis based on Training and Testing Data

The evaluation metrics used for the model include several key performance indicators: The following parameters are used: Accuracy, as the number of correctly predicted occurrences to the total occurrences (refer Figure 4.6); Precision, as the number of correctly predicted occupant instances to the total predicted occupant instances; Recall, as the number of correctly predicted occupant instances to all instances in the actual occupant class; F1 Score, as the harmonic mean of Precision and Recall. Besides, the ROC AUC Score shows the area under the ROC curve, which demonstrates the model's performance in classification differentiation; the Average Precision Score is the arithmetic mean of the precision values at various thresholds (refer Table 4.3). The confusion matrix is a tabular form of presenting the outcomes of the classification model with the actual data findings, which gives a fairly detailed analysis of the model (refer Figure 4.8a & 4.8b).

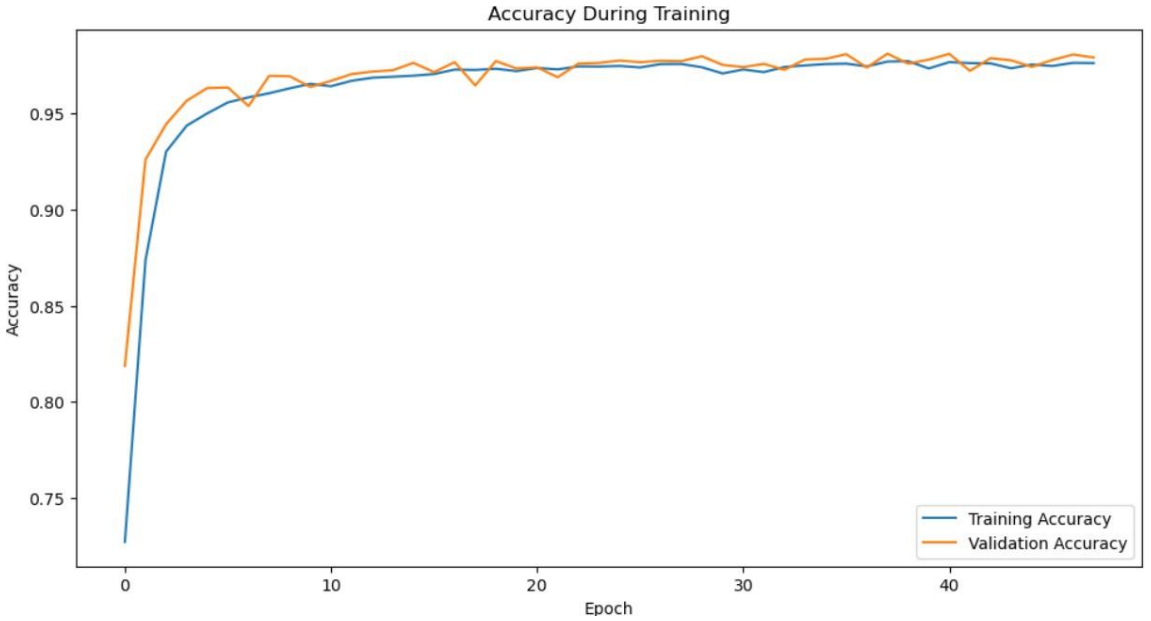


Figure 4.6: Accuracy Vs Epoch plot of the final LSTM architecture during training (Aishin, 2024).

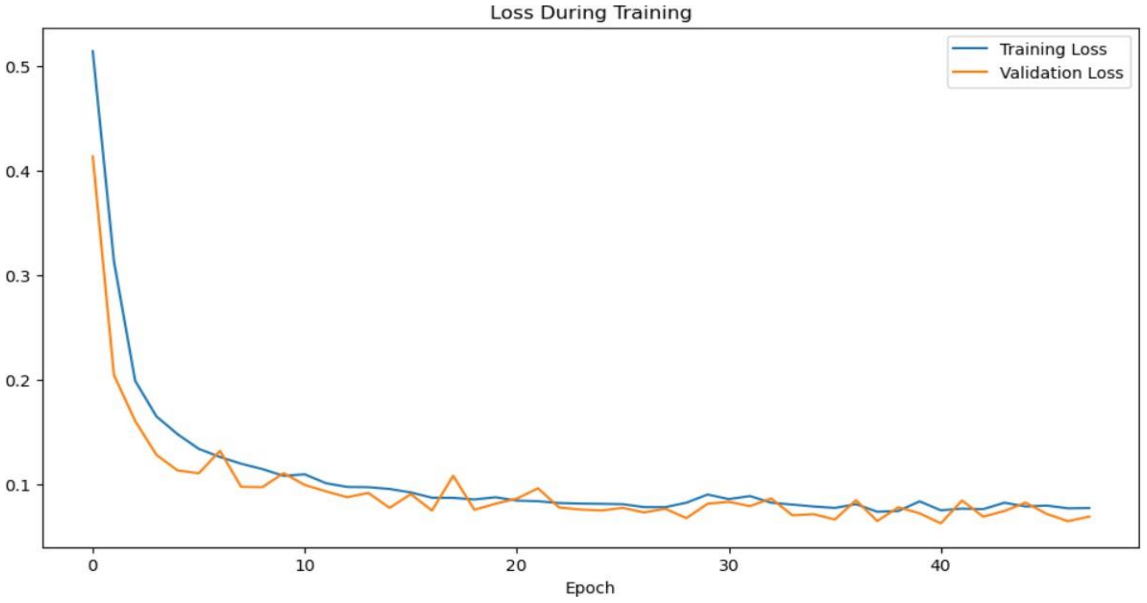


Figure 4.7: Loss function values Vs Epoch plot of the final LSTM architecture during training (Aishin, 2024).

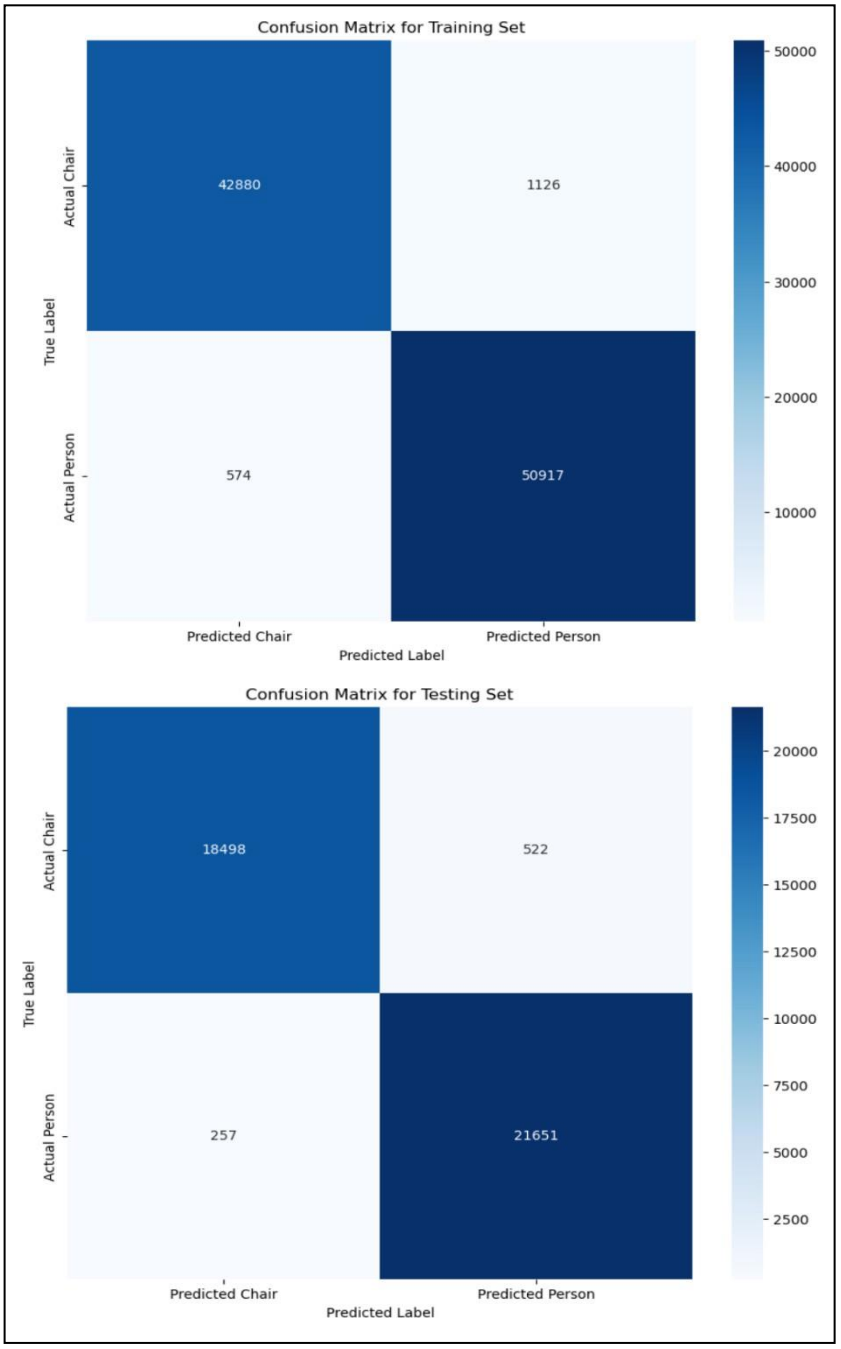


Figure 4.8a above and Figure 4.8b below: Confusion matrix of the training data above and testing data below (Aishin, 2024).

5. Summary and Perspectives

The focus of this study was to enhance the accuracy and robustness of real-time ultrasonic distance measurement systems, by comparing classical methods with machine learning methods. Red Pitaya based ultrasonic sensor was used to collect data for the study, which used a Python based GUI for data processing. The GUI includes signal processing and peak detection methods to help enhance the accuracy of the ultrasonic distance measurements.

The research made use of classical methods such as Kalman filtering and adaptive and local thresholding methods, like Yanowitz-Bruckstein and Niblack's methods, to filter noise and calculate distances with high accuracy. These methodologies stand out for their straightforward nature and overall practicality in a multitude of situations. The study results indicated that traditional methods supply outstanding accuracy and get precise distance measurements in stable environmental conditions. The Kalman filter proved particularly suitable as a method of smoothing the signal, and the thresholding processes were consistent at detecting the initial peak for calculating the distance.

The study conducted by (Aishin, 2024) made an extensive examination of the application of machine learning models, including Recurrent Neural Networks (RNN), Long Short-term Memory Networks (LSTM) and Bidirectional LSTM (BiLSTM) and Gated recurrent units (GRU). The aim was to ascertain the capability of the machine learning models to capture extensive temporal dependencies and nonlinear interactions in the data. Although the machine learning models showed promise, the results suggested that training LSTM and BiLSTM networks required significant effort and that extensive training and evaluation was needed to achieve comparable results with classical methods using ADC data instead of FFT data.

A comprehensive comparative analysis found that classical methods were more precise and accurate than machine learning models, in respect of real-time ultrasonic distance measurements. More straightforward and reliable solutions were provided by classical methods, with significantly less computational requirements. In contrast, the LSTM model, for all of its sophistication with respect to the architectural configuration of the model was trained and evaluated using FFT data. This study suggests that training and testing the LSTM model with ADC data might result in the model performing better, and achieving similar results to the traditional models. The research concluded that classical methods provide greater accuracy and precision, even though machine learning techniques will advance the development of ultrasonic distance measuring, particularly LSTM networks. Traditional methods are still dependable and effective, seemingly in static environments. Additional studies on the improvement of machine learning models, taking advantage of the ADC data for training and testing to close the gap in accuracy and preserve the adaptability and advanced pattern recognition properties of machine learning should be carried out.

6. Abbreviations

A

ADC	Analog-to-Digital Converter

B

BiLSTM	Bidirectional Long Short-Term Memory

D

DL	Deep Learning
DAC	Digital-to-Analog Converter

F

FFT	Fast Fourier Transform

…

G

GUI	Graphical User Interface
GRU	Gradient Recurrent Unit

L

LSTM	Long Short-Term Memory

M

ML	Machine Learning

R

RNN	Recurrent Neural Network

U

UDP	User Datagram Protocol

Z

…

7. References

1. Tessaro, I., V. Mariani, and L. Coelho, 2020. Machine Learning Models Applied to Predictive Maintenance in Automotive Engine Components. Proceedings, vol. 64, pp. 1-6.

2. Bode, G., S. Thul, M. Baranski, and D. Müller, 2020. Real-world Application of Machine-learning-based Fault Detection Trained with Experimental Data. Energy, vol. 198, p. 117323.

3. Abdelgayed, T. S., W. G. Morsi, and T. S. Sidhu, 2018. Fault Detection and Classification Based on Co-training of Semisupervised Machine Learning. IEEE Transactions on Industrial Electronics, vol. 65, no. 2, pp. 1595–1605.

4. L. H. Arnaldi, 2017. Implementation of an AXI-compliant lock-in amplifier on the redpitaya open source instrument. Buenos Aires, Argentina, IEEE, pp. 1-6.

5. Andreas H. Pech, Peter M. Nauth, Robert Michalik, 11 October 2019. A new Approach for Pedestrian Detection in Vehicles by Ultrasonic Signal Analysis. Novi Sad, Serbia, IEEE, pp. 1-5.

6. Peter M. Nauth, Andreas H. Pech, Robert Michalik, 2019. Research on a new Smart Pedestrian Detection Sensor for Vehicles. Sophia Antipolis, France, IEEE, pp. 1-5.

7. Priya Hosur; Rajashekar Basavaraj Shettar; Milind Potdar, 2016. Environmental awareness around vehicle using ultrasonic sensors. Jaipur, India, IEEE, pp. 1154-1159.

8. DFRobot, 2023. Ultrasonic Sensor Review: Comparing DFRobot URM09, HC-SR04, Devantech SRF02 & Maxbotix MB1040. [Online] Available at: https://www.dfrobot.com/blog-13482.html [Accessed 2024].

9. R.K. Mehra, 1971. On-line identification of linear dynamic systems with applications to Kalman filtering. IEEE Transactions on Automatic Control, 16, pp. 12-21.

10. D. Simon, 2010. Kalman filtering with state constraints: a survey of linear and nonlinear algorithms. IET Control Theory and Applications, 4, pp. 1-17.

11. G.G. Rigatos, 2012. A Derivative-Free Kalman Filtering Approach to State Estimation-Based Control of Nonlinear Systems. IEEE Transactions on Industrial Electronics, 59, pp. 3987-3997.

12. S. Mariani, & A. Ghisi, 2007. Unscented Kalman filtering for nonlinear structural dynamics. Nonlinear Dynamics, 49, pp. 1-20.

13. G. Revach, N. Shlezinger, X. Ni, A.L. Escoriza, R.J. Sloun, & Y.C. Eldar, 2021. KalmanNet: Neural Network Aided Kalman Filtering for Partially Known Dynamics. IEEE Transactions on Signal Processing, 70, pp. 1-15.

14. J. Kuti, I. Rudas, H. Gao, & P. Galambos, 2022. Computationally Relaxed Unscented Kalman Filter. IEEE Transactions on Cybernetics, 53, pp. 1-9.

15. F. Auger, M. Hilairet, J.M. Guerrero, E. Monmasson, T. Orłowska-Kowalska, & S. Katsura, 2013. Industrial Applications of the Kalman Filter: A Review. IEEE Transactions on Industrial Electronics, 60, pp. 1-14.

16. W. Bai, W. Xue, Y. Huang, & H. Fang, 2018. On extended state based Kalman filter design for a class of nonlinear time-varying uncertain systems. Science China Information Sciences, 61, pp. 1-16.

17. Y. Niblack, 1986. An Introduction to Digital Image Processing. Englewood Cliffs, NJ, Prentice-Hall, pp. 1-30.

18. Yu Wang, Xueye Wei, Shuo Xiao, 2008. LBP Texture Analysis Based on the Local Adaptive Niblack Algorithm. 2008 Congress on Image and Signal Processing, IEEE, pp. 777-780.

19. Khairun Saddami, K. Munadi, S. Muchallil, F. Arnia, 2017. Improved Thresholding Method for Enhancing Jawi Binarization Performance. 2017 14th IAPR International Conference on Document Analysis and Recognition (ICDAR), IEEE, pp. 1108-1113.

20. M. Nasri, Zahra Hossein-Nejad, P. Hosseini-Zahmatkesh, 2018. Document Image Binarization Based on Combination of Global and Local Thresholding Methods. International Journal of Imaging and Robotics, vol. 18, pp. 74-87.

21. F. Kurniadi, Desty Septyani, I. Pratama, 2020. Local Adaptive Thresholding Techniques for Binarizing Scanned Lampung Aksara Document Images. 2020 3rd International Conference on Computer and Informatics Engineering (IC2IE), IEEE, pp. 135-139.

22. S.D. Yanowitz, A.M. Bruckstein, 1989. A new method for image segmentation. Computer Vision, Graphics and Image Processing, pp. 82–95.

23. Ilya Blayvas, Alfred M. Bruckstein, Ron Kimmel, 2001. Efficient computation of adaptive threshold surfaces for image binarization. Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001, IEEE, pp. 1-6.

24. H. Yazid, H. Arof, 2013. Gradient based adaptive thresholding. Journal of Visual Communication and Image Representation, Vol. 24, No. 9, pp. 926-936.

25. Min Jiao, 2015. Research of the Ultrasonic Distance Measure. AMEII 2015, Atlantis Press, pp. 1609-1613.

26. A. Sahoo, S. Udgata, 2020. A Novel ANN-Based Adaptive Ultrasonic Measurement System for Accurate Water Level Monitoring. IEEE Transactions on Instrumentation and Measurement, vol. 69, pp. 3359-3369.

27. O. Gultekin, H. Erdol, M. A. Çavuslu, 2017. FPGA based moving objects controller using remote distance sensors. 2017 25th Signal Processing and Communications Applications Conference (SIU), IEEE, pp. 1-3.

28. D. V. Kravets′, A. Tushych, V. Shkapa, & V. Mykolaychuk, 2020. Overview of the standard library for creating a GUI in Python. Connectivity, IEEE, pp. 1-6.

29. A. Gupta, 2021. A Research paper on Typing Speed Tester Game using Python & Tkinter. International Journal for Research in Applied Science and Engineering Technology, IEEE, pp. 1-6.

30. A. Tahat, & M. Tahat, 2011. Python GUI Scripting Interface for Running Atomic Physics Applications. ArXiv, IEEE, pp. 1-6.

31. S. Kurzadkar, A. Dadhe, P. Bhoyar, P. Kumbhare, S. Kadak, & N. Bhore, 2022. Hotel Management System Using Python Tkinter GUI. International Journal of Computer Science and Mobile Computing, IEEE, pp. 1-6.

32. K. D. Lee, & S. Hubbard, 2015. Python Programming 101. IEEE, pp. 1-40.

33. M. T. Hoang, B. Yuen, X. Dong, T. Lu, R. Westendorp, & K. Reddy, 2019. Recurrent Neural Networks for Accurate RSSI Indoor Localization. IEEE Internet of Things Journal, pp. 1-13.

34. S. Gao, Y. Huang, S. Zhang, J. Han, G. Wang, M. Zhang, & Q. Lin, 2020. Short-term runoff prediction with GRU and LSTM networks without requiring time step optimization during sample generation. Journal of Hydrology, pp. 1-22.

35. S. Noh, 2021. Analysis of Gradient Vanishing of RNNs and Performance Comparison. Inf., pp. 1-11.

36. Acroname, n.d. Devantech SRF02 Sonar Rangefinder. [Online] Available at: https://acroname.com/store/sonar-rangefinder-r287-srf02 [Accessed 2024].

37. Paween Pongsomboon, 2022. Using Image Processing for Autonomous Illumination Sensor Optimization, Frankfurt University of Applied Sciences.

38. Wang, Haiyao & Wang, Jianxuan & Cao, Lihui & Li, Yifan & Sun, Qiuhong & Wang, Jingyang, 2021. A Stock Closing Price Prediction Model Based on CNN-BiSLSTM. Complexity, pp. 1-12.

39. Kai Guo, Seungwon Choi, & Jongseong Choi, 2022. Gated Recurrent Unit for Video Denoising. ArXiv, abs/2210.09135.

40. Lu Kuan, Zhao Yan, Wang Xin, Cheng Yan, Pang Xiangkun, Sun Wenxue, Jiang Zhe, Zhang Yong, Xu Nan, & Zhao Xin, 2017. Short-term electricity load forecasting method based on multilayered self-normalizing GRU network. 2017 IEEE Conference on Energy Internet and Energy System Integration (EI2), pp

41. Htet Myet Lynn, S. Pan, & Pankoo Kim, 2019. A Deep Bidirectional GRU Network Model for Biometric Electrocardiogram Classification Based on Recurrent Neural Networks. IEEE Access, pp. 1-11.

42. D2L.ai, 2023. Long Short-Term Memory (LSTM). [Online] Available at: https://d2l.ai/chapter_recurrent-modern/lstm.html [Accessed 2024].

43. JetBrains, n.d. PyCharm: The Python IDE for Data Science and Web Development. [Online] Available at: https://www.jetbrains.com/pycharm/ [Accessed 2024].

44. TutorialsPoint, n.d. PyCharm - Quick Guide. [Online] Available at: https://www.tutorialspoint.com/pycharm/pycharm_quick_guide.htm [Accessed 2024].

45. Dhanoop Karunakaran, 2018. Kalman Filter in Stock Trading. Medium. [Online] Available at: https://medium.com/intro-to-artificial-intelligence/kalman-filter-in-stock-trading-552e1e4b2dfb [Accessed 2024].

46. Zhang, H., Zhu, Y., & Wang, G., 2009. An Improved Algorithm for Adaptive Kalman Filter in Target Tracking. IEEE, pp. 1-12.

47. Auger, F., Hilairet, M., Guerrero, J.M., Monmasson, E., Orłowska-Kowalska, T., & Katsura, S., 2013. Industrial Applications of the Kalman Filter: A Review. IEEE Transactions on Industrial Electronics, 60, pp. 1-12.

48. "pykalman." GitHub. [Online] Available at: https://github.com/pykalman/pykalman. [Accessed 2024].

49. Nixon, M., & Aguado, A., 2019. Feature Extraction and Image Processing for Computer Vision. Academic Press, pp. 77-81.

50. Otsu, N., 1979. A threshold selection method from gray-level histograms. IEEE Transactions on Systems, Man, and Cybernetics, pp. 62-66.

51. Piercy, J. E., Embleton, T. F. W., & Sutherland, L. C., 1977. Review of noise propagation in the atmosphere. The Journal of the Acoustical Society of America, 1403-1418.

52. Ahmed Gad, 2021. KDNuggets. [Online] Available at: https://www.kdnuggets.com/2021/02/evaluating-deep-learning-models-confusion-matrix-accuracy-precision-recall.html [Accessed 2024].

53. B. Dong, D. Yan, Z. Li, Y. Jin, X. Feng, & H. Fontenot, 2018. Modeling occupancy and behavior for better building design and operation—A critical review. Building Simulation.

54. J. Schmidhuber, 2014. Deep Learning in Neural Networks: An Overview. ArXiv, pp. 85-117.

55. J. Zou, Q. Zhao, W. Yang, & F. Wang, 2017. Occupancy detection in the office by analyzing surveillance videos and its application to building energy conservation. Energy and Buildings, pp. 385-398.

56. S. Hochreiter, & J. Schmidhuber, 1997. Long Short-term Memory. Neural computation, 9, pp. 1735-80.

57. K. Greff, R. Srivastava, J. Koutník, B. Steunebrink, & J. Schmidhuber, 2015. LSTM: A search space odyssey. IEEE Transactions on Neural Networks and Learning Systems, pp. 2222-2232.

58. T. Mikolov, M. Karafiát, L. Burget, J. Cernocký, & S. Khudanpur, 2010. Recurrent neural network based language model. Proceedings of the 11th Annual Conference of the International Speech Communication Association, INTERSPEECH 2010, 2, pp. 1045-1048.

59. K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, & Y. Bengio, 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation, pp. 1-15.

60. J. Chung, C. Gulcehre, K. Cho, & Y. Bengio, 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling, pp. 1-9.

61. I. Khan, F. Delicato, E. Greco, M. Guarascio, A. WangEsrafilian-Najafabadi, A. Guerrieri, & G. Spezzano, 2023. Occupancy Prediction in Multi-Occupant IoT Environments Leveraging Federated Learning, pp. 36-43.

62. A. Graves, & J. Schmidhuber, 2005. Framewise phoneme classification with bidirectional LSTM and other neural network architectures. Neural Networks, 18(5–6), pp. 602–610.

63. M. Schuster, & K. K. Paliwal, 1997. Bidirectional recurrent neural networks. IEEE Transactions on Signal Processing, 45(11), pp. 2673-2681.

64. H. Oshima, T. Ishizone, K. Nakamura, & T. Higuchi, 2022. Occupancy Detection for General Households by Bidirectional LSTM with Attention. IECON 2022 – 48th Annual Conference of the IEEE Industrial Electronics Society, Brussels, Belgium, pp. 1-7.

65. M. Sokolova, & G. Lapalme, 2009. A systematic analysis of performance measures for classification tasks. Information Processing & Management, 45(4), pp. 427-437.

66. C. Goutte, & E. Gaussier, 2005. A Probabilistic Interpretation of Precision, Recall and F-Score, with Implication for Evaluation. In: D.E. Losada & J.M. Fernández-Luna (eds) Advances in Information Retrieval. ECIR 2005. Lecture Notes in Computer Science, vol 3408. Springer, Berlin, Heidelberg.

67. D. M. Powers, 2020. Evaluation: From precision, recall and F-measure to ROC, informedness, markedness and correlation, pp. 1-27.

68. T. Fawcett, 2006. An introduction to ROC analysis. Pattern Recognition Letters, 27(8), pp. 861-874.

69. Aishin Abdulla Yoosufali, 2024. Dynamic Smart Office Environment for Occupancy Detection Utilizing Machine Learning. [Online] Available at: https://github.com/aishincp/Master_Thesis_Project

70. Tiniya Vinod Puthanpurayil, 2023. Automatic Labelling System using ML. [Online] Available at: https://github.com/TiniyaVinod/Automatic-Labelling-System-using-ML

71. Bibi, Iram & Akhunzada, Adnan & Malik, Jahanzaib & Iqbal, Javed & Musaddiq, Arslan & Kim, Sung, 2020. A Dynamic DL-Driven Architecture to Combat Sophisticated Android Malware. IEEE Access. pp. 1-13.

72. Open Data Science, 2020. Understanding the Mechanism and Types of Recurring Neural Networks. [Online] Available at: https://opendatascience.com/understanding-the-mechanism-and-types-of-recurring-neural-networks [Accessed 2024].

8. Appendix

Attached CD/DVD content