Kiosks for Non-Contact Vital Sign Detection

by

Ivan Goryachev

B.S.M.E., Northeastern University (2013)

Submitted to the Department of Mechanical Engineering
in partial fulfillment of the requirements for the degree of

Master of Science in Mechanical Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2021

Author . . . .
Department of Mechanical Engineering
August 31, 2021

Certified by . . . .
Brian W. Anthony
Principal Research Scientist
Thesis Supervisor

Accepted by . . . .
Nicolas Hadjiconstantinou
Chairman, Department Committee on Graduate Theses

Kiosks for Non-Contact Vital Sign Detection

by

Ivan Goryachev

Submitted to the Department of Mechanical Engineering
on August 31, 2021, in partial fulfillment of the
requirements for the degree of
Master of Science in Mechanical Engineering

Abstract

Motivated by the COVID-19 Pandemic and its effect on individuals' baseline vital signs, this work presents a modular hardware and software platform for contact-less vital sign detection. The kiosk is intended collect users' vital sign data to track changes over time and try to identify periods of illness. It is modular and transportable, able to be deployed and moved quickly, and reconfigured with different sensors if needed. In this implementation, it is instrumented with an infrared thermal imaging camera, FMCW radar sensors, motion and height detection, and ambient climate sensors. The kiosk collects body temperature data and radar-based chest displacement data for offline processing. The algorithms described here show good performance in measuring respiration rate when used with a mechanical chest simulator, but require additional work to properly measure heart rate, due to the low signal-to-noise ratio.

Thesis Supervisor: Brian W. Anthony
Title: Principal Research Scientist

Acknowledgments

I'd like to thank Brian for his guidance and willingness to accommodate my interests and tie them into relevant projects, as well as his coordination of the many stakeholders involved in this project effort. I'd like to thank Shawn Zhang for brainstorming and helping me get settled within the lab. Greg and Anne for their incredible patience in helping me get up to speed with MATLAB and image processing. Alex Jaffe and the computations group for listening and their feedback during minigroup meetings. I'd also like to thank my friends for their support and late nights spent helping me with various aspects of debugging, writing and homework - Danyal, Trang, Turga. Finally, Arushi and my family for bearing with me during my absence and encouraging me to push ahead.

Contents

List of Figures

List of Tables

Nomenclature

Acronyms

AC	Alternating Current
ADC	Analog to digital converter
AUC	Area Under the ROC Curve
API	Application Programming Interface
bpm	beats per minute
BR	Breathing Rate
brpm	breaths per minute
CW	Continuous Wave
DC	Direct Current
DRL	Device Realization Laboratory
ECG	Electrocardiogram
FFT	Fast Fourier Transform
FMCW	Frequency Modulated Continuous Wave
FPGA	Field Programmable Gate Array
GUI	Graphical User Interface
HR	Heart Rate
HROS	Heart rate over steps
HRV	Heart Rate Variability
I	In-phase
IF	Intermediate Frequency
IMU	Inertial Measurement Unit
IR	Infrared
JSON	JavaScript Object Notation
MA	Motion Artifacts
NEMA	National Electrical Manufacturers Association
PCR	Polymerase Chain Reaction
PPG	Photoplethysmogram
PWM	Pulse Width Modulation
Q	Quadrature
RBW	Random Body Movement
RFID	Radio Frequency Identification
RHR	Resting Heart Rate
ROC	Receiver Operating Characteristic
RMSSD	Root Mean Square of Successive Differences
SAR	Secondary Attack Rate
SARS-CoV2	Severe Acute Respiratory Syndrome Coronavirus 2
SD	Standard Deviation
SNR	Signal-to-Noise Ratio
SpO_2	Peripheral oxygen saturation percentage
VI	Virtual Instrument

Chapter 1

Background and motivation

1.1 COVID-19 and Baseline Vital Signs

To date, effective management of the COVID-19 pandemic consists of social distancing, mask usage, testing, and quarantining symptomatic individuals. However, due to the variation in symptom severity and onset, it may not be obvious when to quarantine. It has been shown that pre-symptomatic transmission can be more frequent than symptomatic transmission. Asymptomatic transmission is less understood, due to differences in symptom definition and misclassification of pre-symptomatic cases as asymptomatic [1]. Therefore, the ability to isolate patients during the pre-symptomatic phase could further reduce spread.

Throughout 2020, several large scale studies found possible links between wearable device vital sign data and instances of COVID-19 infection. By tracking and analyzing subjects' vital signs over a period of time, they were able to identify deviations from normal prior to symptom onset in some people.

To help reduce infection spread indoors, institutions implement some form of access control, which can include temperature screening, self-attestations, contact tracing, and testing. These methods can be effective in identifying acutely ill individuals, but, due to the variation in "normal" body temperatures and symptom severity, may miss pre- and asymptomatic individuals. Using and building upon insights from the vital sign studies, it could be possible to create a more effective access control protocol by collecting a broader array of vital signs and tracking them over time. Users could then receive warnings to get tested and/or quarantine earlier, which in turn would lower chances of spreading the virus.

1.1.1 COVID-19 and Vital Signs

For early detection of respiratory illnesses such as COVID-19, body temperature, heart rate, respiration rate, blood oxygen level are the vital signs of interest. During the course of an infection, heart rate and temperature are elevated and correlated (every 1°C increase in temperature corresponds to a 7-8.5 beat per minute (bpm) increase in heart rate [2, 3]. When lung function is affected, such as during pneumonia caused by SARS-CoV2, breathing rate can be elevated and blood oxygen levels reduced. In onne study, 1,095 COVID patients had a mean respiratory rate of 23 brpm and mean oxygen saturation of 91%, with only 10%

and 25% of them experiencing shortness of breath and cough, respectively [4]. This further highlights the importance of tracking vital sign metrics in the detection of COVID-19.

Mishra et al. of Stanford [5] analyzed wearable data from 3,325 individuals using Fitbit devices over a period of time from January to September 2020. The researchers retrospectively analyzed heart rate and activity level (step count) data together with symptom reporting COVID test results to to show that some individuals exhibited changes in their vital signs prior to symptoms occurring. Using this data, they proposed an algorithm for detecting anomalies in users' vital signs after collecting sufficient data to characterize normal baseline levels for each individual.

In their retrospective data analysis, the authors used two resting heart rate (RHR) based metrics, RHR difference (RHR-Diff) and heart rate over steps (HROS). RHR-Diff tracked the standardized residual of the daily RHR average and the previous 28-day sliding window average. The HROS metric is RHR divided by number of daily steps, with the assumption that infected individuals would exhibit lower activity levels, leading to a more prominently elevated parameter than increased RHR alone. Results were similar, with 2 and 1 events being detected by only RHR-Diff and only HROS, respectively. Of the 32 COVID-19 infected subjects, 16 had distinct elevated values before symptoms, 10 had symptom-associated peaks, and 6 did not have a significant deviation (2 with respiratory conditions and 1 with allergies). The median RHR increase was 7 bpm.

The predictive method of identifying COVID-19 onset used the previous 28-day RHR deviation values to build a null distribution, and used adjustable significance levels to generate an initial alarm when RHR deviation p-value fell within that level. If that continued for 24 hours, the alarm was converted to a "positive event". The authors speculate that, because the predictive method only looks at the RHR data directly preceding an event instead of the entire data set, the number of subjects with anomalies detected is lower, 63% versus 81%.

In a study conducted by Fitbit Research, data from 2,745 COVID-19-positive wearable users showed elevated respiration rate and heart rate, and lowered heart rate variability (HRV, calculated as the root mean square of successive differences (RMSSD)), around the onset of symptoms. The team used the data for classifying users as sick or healthy, producing a model that yielded an area under ROC curve of 0.77 [6].

A study by the Huami Corporation used heart rate and sleep data collected from wearable devices to predict the onset of COVID-19 in different cities. Like the Stanford Study, the researchers used both retrospective and predictive approaches. In the former, they defined anomalies in RHR as being at least 1.5 standard deviations (SD) larger the average, and anomalies in daily sleep duration as being longer than average minus 0.5 SD, both observed over a period of at least five days. In the latter, they combined historical anomaly data with with temporal categorical data (holidays, etc) and anomaly data of the past few days to predict rises in COVID-19 infection rates before official numbers were reported by authorities [7].

Another wearable manufacturer, Whoop Inc, used respiratory rate data from 271 subjects (81 tested positive for COVID-19) to build a machine learning model to retrospectively identify periods of infection. 20% of COVID-19 positive subjects in their validation set were identified before symptom onset, and 80% were identified by the third day of symptoms [8].

The studies show potential of using vital signs other than body temperature in the detection of COVID-19 and other respiratory illnesses. The data has been collected from wearable devices, and is subject to issues arising from wearing and charging habits. Applying these insights to design an early warning system for users in an organization could reduce spread and allow for more in-person interaction, but would be met with challenges due to the high cost of devices and user adherence and proper usage. As an alternative, a single device collecting daily vital sign data from all users of a particular space is proposed.

1.2 Vital sign detection techniques

Vital sign detection is routinely performed in clinical and everyday settings with varying degrees of accuracy depending on the method used. The main vital signs include heart rate, breathing rate, body temperature, blood oxygen saturation, and blood pressure, with the focus being on the first four in the context of respiratory illness detection. Contact-based vital sign measurement techniques are described and compared to non-contact techniques below.

1.2.1 Contact Methods

Electrocardiography

The clinical standard of measuring heart rate is electrocardiography, which works by measuring the potential difference of at least two points on a subject's body. This measurement signal is called the electrocardiogram (ECG) and is used to monitor heart rate, diagnose heart conditions, and estimate breathing rate. ECG systems range from 2-lead portable ambulatory devices to 12-lead bedside systems, depending on the application, where a "lead" is the potential difference between two electrodes, or one electrode and an average potential value of multiple electrodes.

There are several factors that influence the accuracy of ECG measurements - more electrodes, correct electrode placement and adhesive type, as well as motion artifacts. Wet electrodes are more accurate due to their low contact impedance, but rely on conductive gels and adhesives between themselves and the skin, which can cause irritation and allergic reactions. Dry electrodes have higher impedance but are better suited to long-term wear. Motion artifacts occur as a result of other muscle movements, usually associated with respiration [9]. Respiration causes three types of modulation in the ECG signal - (1) baseline drift and (2) amplitude modulation caused by changes in the heart's position relative to electrodes and changes in thoracic impedance, and (3) frequency modulation caused by increased and decreased HR during inhalation and exhalation, respectively. By analyzing one or more of these effects, it is possible to estimate breathing rate [10].

Photoplethysmography

Another broadly applied vital sign measurement technique is photoplethysmography (PPG), which relies on the principle that light is absorbed differently by various tissues in the body. The system works by shining a light onto skin and measuring either reflected light intensity on the same side (usually green light, as in a smartwatch), or transmitted intensity on the opposite side (red light, as in a finger clip pulse oximeter). The measured waveform, shown in Figure 1-1, corresponds to the blood volume in the tissue, with the AC component being affected by the cardiac cycle, and the DC offset relating to respiration [11]. Green light is highly absorbed by blood and is therefore a better choice for reflectance methods, while red and infrared light passes through tissue and is typically used for transmittance methods [12].

In addition to heart rate and breathing rate, PPG can be used to measure peripheral blood oxygen concentration, abbreviated as SpO_2. When oxygenated, blood absorbs more infrared light and less red light than when it is oxygenated [13],[11]. Devices therefore use two emitters and convert the measured ratio of infrared to red light detected into an estimate of oxygen saturation. PPG is affected by motion artifacts like movement, tremors, coughing or sudden respiratory changes [11], which makes it difficult for consumer wearables to extract the more subtle respiration signal while not at rest.

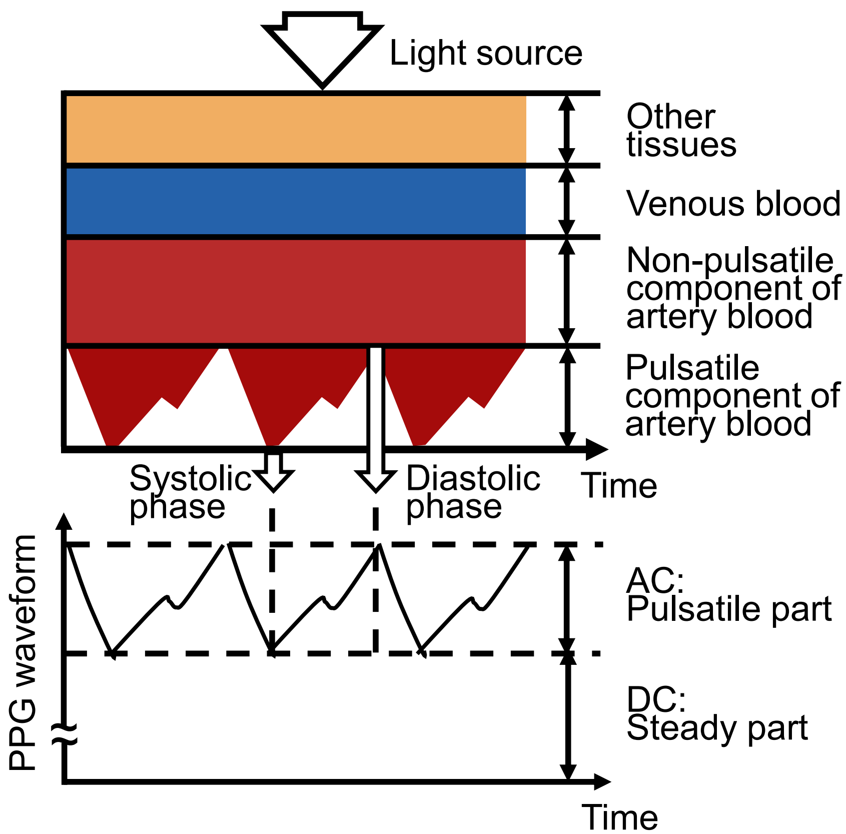


Figure 1-1: Illustration of typical PPG waveform [11].

Surface Temperature

Measuring human body temperature presents a number of challenges. Typically, it is measured on the surface of the skin, rectally, or orally, and is a proxy for core temperature, which can be measured with a surgical probe of inner organs [14]. Standard medical literature defines a fever as an oral temperature above 37.5°C, with a normal range of 36.4°C to 37.2°C [15]. There are many caveats to this which introduce offsets and errors, including circadian rhythm, medical conditions, menstrual cycles, and medications.

With skin or tissue contact methods, there are many factors to consider. A systematic review [16] of many contact-based skin temperature studies, highlighted the following:

- The influence of the sensor on the skin temperature. Sensor serves to carry heat away from the skin, based on its thermal properties and geometry. See Figure 1-2.
- The environment around the sensor. Higher temperature difference between skin and the environment typically showed a larger absolute mean difference. Air movement tends to have a negligible effect when the environmental difference is low.
- The pressure applied to the skin. Increased contact pressure causes higher temperature readings, but too low of a contact pressure causes poor conduction between the probe and the skin, so a "sweet spot" needs to be maintained.
- The attachment method of the sensor. Using patches or plain contacts and presence of sweat can influence temperature readings, either by insulating or cooling, respectively.

Because skin temperature cannot be measured directly, the studies of skin temperature are all comparative - they report the mean absolute difference of two measurements. Comparing sensor types, attachments, environments, etc. The review points out that it is practical to consider and each of the aforementioned factors and work to minimize their effects to obtain the best measurement. In their analysis the authors considered only mean absolute differences above 0.5°C as "practically meaningful".

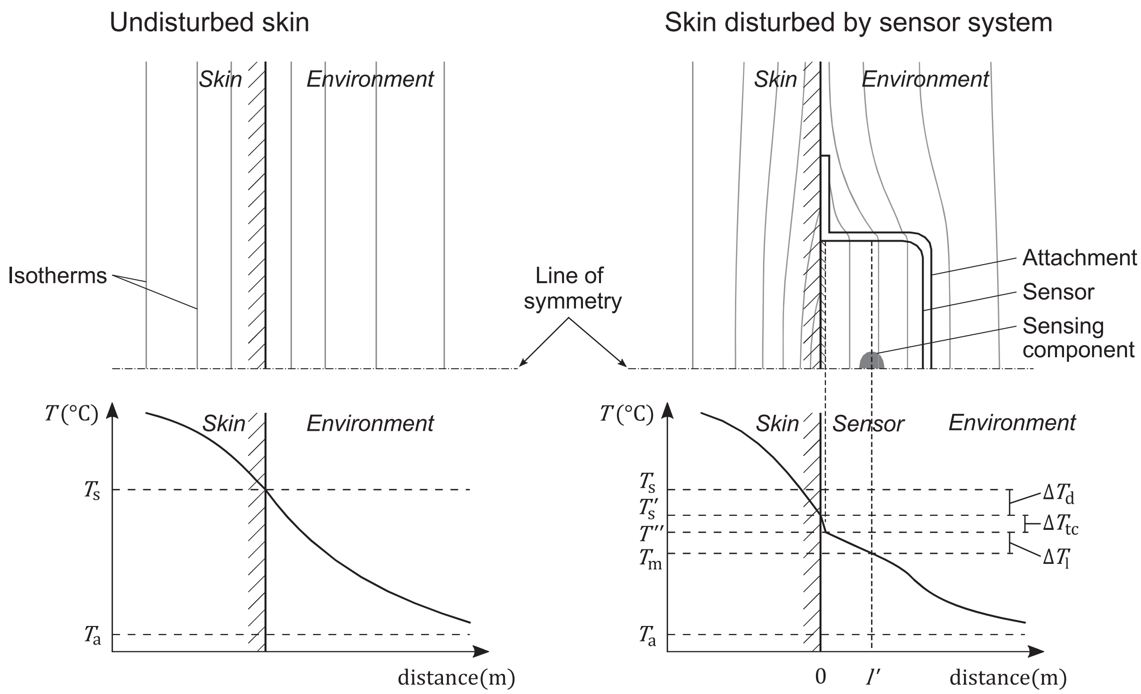


Figure 1-2: Temperature profile of skin-sensor interface [16].

Chest Displacement and Air Analysis

Two other respiration measurement methods are worth mentioning - chest displacement tracking and air flow analysis [9]. Chest displacement can be measured by sensing strain using resistive strain gauges, front-to-back capacitance meters (changes when lungs fill up), impedance measurement (same as ECG), and inertial measurement units (IMU, a combination of accelerometer, gyroscope, and magnetometer). Inhaled and exhaled air have different properties, like CO_2 content, temperature, and amount of water vapor. These can be measured by capturing air using a mask or nasal cannula and sensing the air properties. Exhaled air will have more CO_2 (6% vs 0.04%), up to 15°C higher temperature, and 20% to 60% higher humidity. Air-based methods are less susceptible to motion artifacts than chest displacement tracking, but are more invasive.

1.2.2 Non-contact Measurement Methods

Infrared Temperature

A commonly used non-contact vital sign measurement to date is temperature, using infrared sensing. Examples of such devices include handheld IR forehead thermometers, tympanic (in-ear) thermometers, and IR cameras. Infrared sensing relies on detecting emitted infrared light intensity, which is related to an object's temperature via the Stefan-Boltzmann law, 1.1. The law assumes the object to be a perfect emitter, called a blackbody which absorbs all incoming radiation, such that all of its emitted radiation depends only on its temperature, and does not include reflections. In practice, objects are characterized via their emissivity ε, or how much energy they radiate relative to an ideal blackbody. Emissivity for human skin is roughly 0.98, making it easier to determine accurate temperature versus highly reflective materials like gold and aluminum that have emissivities of less than 0.05 [17].

$$\ddot{Q} = \varepsilon \sigma T^4 \tag{1.1}$$

IR thermometers use thermopile sensors such as the MLX90614 (Melexis NV, Ypres, Belgium) in Figure 1-3, with accuracy of 0.3°C [18]. These sensors are a combination of many thermocouples in series, which output a potential difference directly proportional to the temperature change caused by the incident radiation, as described by the Seebeck effect. In medical and screening applications, such thermometers are used in close proximity to the subject, either by measuring inside the ear or a few centimeters from the forehead.

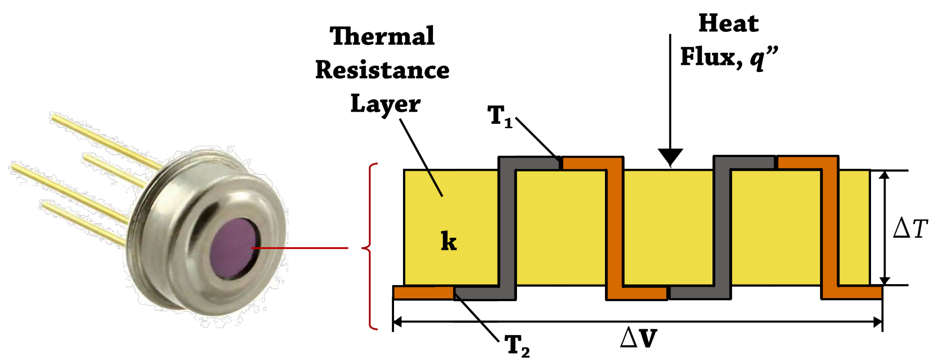


Figure 1-3: Photo [18] and schematic [19] of a thermopile sensor.

IR cameras, among other applications, are increasingly used to screen people for elevated skin temperatures. Consumer and medical applications use microbolometer sensors, which are an array of elements whose resistance changes when they are heated by focused infrared radiation. A schematic in Figure 1-4, shows the conversion of incident IR radiation to a matrix of temperature values. In common configurations for applications such as surface and body temperature measurements, the sensors are designed to detect IR wavelengths of 8-14 μm, to avoid interference from water vapor, CO_2, and other gases [17]. Due to the high cost of these sensors, more affordable low resolution versions are often paired with visible light cameras in a single device. The camera is used for either enhancing the final image by overlaying the two signals, or for locating the area of interest to be measured, using human face detection, for example.

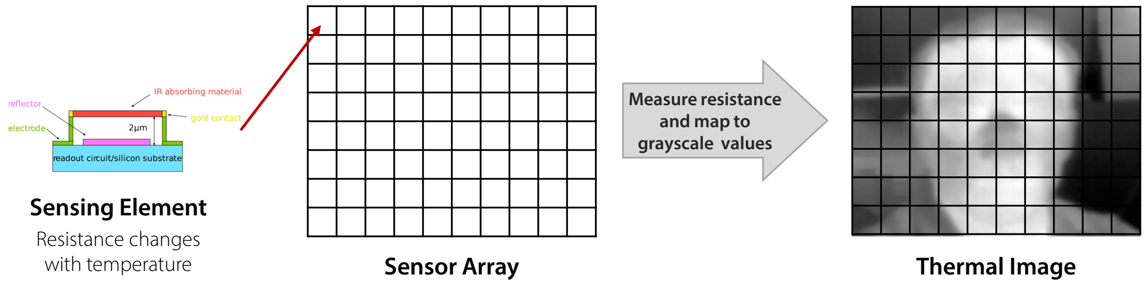


Figure 1-4: Photo [18] and schematic [20] of a microbolometer-based IR camera,

IR cameras can be calibrated using a series of near-blackbody emitters set to a range of temperatures, to form a relationship between object temperature and detected radiation [21]. Due to calibration drift over time, instead of periodic calibration, many end-user solutions place a known temperature reference emitter in the field of view close the object of interest. Software then compares measured values to determine the object temperature.

In light of the increased use of IR cameras for COVID-19 screening, Dell'Isola et. al. [22] presented a literature analysis of IR measurement uncertainty and developed a screening approach to address it. Their uncertainty sources are summarized below:

- **Measurand:** The individual and their physical and temporal state.
  Physical:
  - Older age corresponds to lower body temperatures by 0.2-0.7°C.
  - Common drugs, like antihistamines, contraceptives, and melatonin can alter temperatures by –0.6°C, +0.6°C, and +0.2°C, respectively.
  - Menstrual cycle, shown in Figure 1-5, decreases body temperature in the pre-ovulatory phase and increases it in the luteal phase, by 0.1-0.9°C [23].
  - Medical and psychiatric conditions such as hypothyroidism and depression can alter body temperatures.

  Temporal:
  - Circadian rhythm, shown in Figure 1-5, alters the body temperature with its lowest value around 5am and its highest around 8pm, varying by as much as 0.9°C.
  - Physical activity increases body temperature (rectal up to 40°C, inner canthus up to 39°C, by transposition), which returns to baseline within 30 minutes of cessation.
- **Instrumental:**
  - Contact thermometers, both mercury and electronic have a maximum permissible error of 0.1°C according to ASTM E 667-86 and ASTM E 1112-86.
  - IR thermometers have a maximum permissible error of 0.2°C and 0.3°C for for ear canal and skin measurements, respectively, according to ASTM E1965-98:2003, EN 12470-5:2003 and EN ISO 80601-2-56:2012.

- Drift of IR thermometers and cameras can be about 0.1°C per year and requires periodic calibration unless a reference emitter is used.
- Instrument resolution depends on its analog to digital converter (ADC) and/or its LCD display.

- **Outside Influences:**
  - Environmental conditions influence the temperature difference between core and peripheral by 0°C to 15°C in hot and cold environments, respectively.
  - Skin emissivity determines the fraction of IR radiation detected that is a result of the temperature vs reflections from the environment. Sweat, oils, and cosmetic products lower the emissivity of the skin, with the latter being able to mask a fever of 1-2°C.
  - Radiant temperature variation can produce measurement differences of 0.7°C between 0°C and 40°C environments, assuming a normal skin emissivity of 0.98.

- **Operator:**
  - Angle of measurement: for best results, handheld IR thermometers should not be held more than 30° from perpendicular to the surface.
  - Distance: IR thermometers have varying focal points, so proper distance to the subject must be maintained to ensure only the area of interest is measured and not other components. In the case of IR cameras with fixed-focus lenses, the subject should be placed at the proper distance.

Taking all of these into consideration, the authors developed a method to calculate the measurement uncertainty of IR temperature, using Equation 1.2 and Table 1.1.

$$u_c^2 = u_{m,ind}^2 + u_{m,env}^2 + u_{i,cal}^2 + u_{i,drift}^2 + u_{i,res}^2 + u_{e,temps}^2 + u_{e,emi}^2 + u_{e,mrt}^2 + u_{o,target}^2 \quad (1.2)$$

Uncertainty Source	Uncertainty Cause	Symbol	Type	Distribution	Measurement Conditions			
					Indoor (After Subject Acclimatisation)		Outdoor (Without Subject Acclimatisation)	
					Expanded Uncertainty	Standard Uncertainty	Expanded Uncertainty	Standard Uncertainty
Measurand	Individual and Spatial	$u_{m,ind}$	A	Normal	0.20 °C	0.10 °C	0.20 °C	0.10 °C
	Temporal	$u_{m,temp}$	A	Rectangular	0.25 °C	0.09 °C	0.50 °C	0.18 °C
	Environmental	$u_{m,\ env}$	B	Rectangular	0.25 °C	0.09 °C	0.50 °C	0.18 °C
Instrument	Calibration	$u_{i,cal}$	B	Normal	0.15 °C	0.08 °C	0.15 °C	0.08 °C
	Drift	$u_{i,drift}$	B	Normal	0.10 °C	0.05 °C	0.10 °C	0.05 °C
	Temperature resolution	$u_{i,res}$	A	Normal	0.10 °C	0.05 °C	0.10 °C	0.05 °C
	Response time	$u_{i,time}$	A/B	Normal	negligible	-	negligible	-
Environmental Influence quantities	Temperature effect [1]	$u_{e,temp}$	B	Rectangular	0.10 °C	0.04 °C	0.20 °C	0.07 °C
	Skin emissivity [2]	$u_{e,emi}$	B	Rectangular	0.05 °C	0.02 °C	0.10 °C	0.04 °C
	Mean radiant temperature [3]	$u_{e,mrt}$	B	Rectangular	0.05 °C	0.02 °C	0.20 °C	0.07 °C
Operator	Target uniformity	$u_{o,target}$	B	Rectangular	0.05 °C	0.02 °C	0.05 °C	0.02 °C
	Angle incidence	$u_{o,angle}$	A/B	Normal	negligible	-	negligible	-
Composed Uncertainty		fixed threshold			0.40 °C	0.20 °C	0.62 °C	0.31 °C
		statistical threshold			0.28 °C	0.14 °C	0.42 °C	0.21 °C

[1] Evaluated on the basis of a device temperature between 20 ± 2 °C (20 ± 5 °C) for indoor (outdoor) conditions. [2] Evaluated on the basis of a skin emissivity between 0.980 ± 0.003 (0.980 ± 0.006) for indoor (outdoor). [3] Evaluated on the basis of a mean radiant temperature between 20 ± 2 °C (20 ± 10 °C) for indoor (outdoor) conditions.

Table 1.1: IR temperature uncertainty sources and values [22].

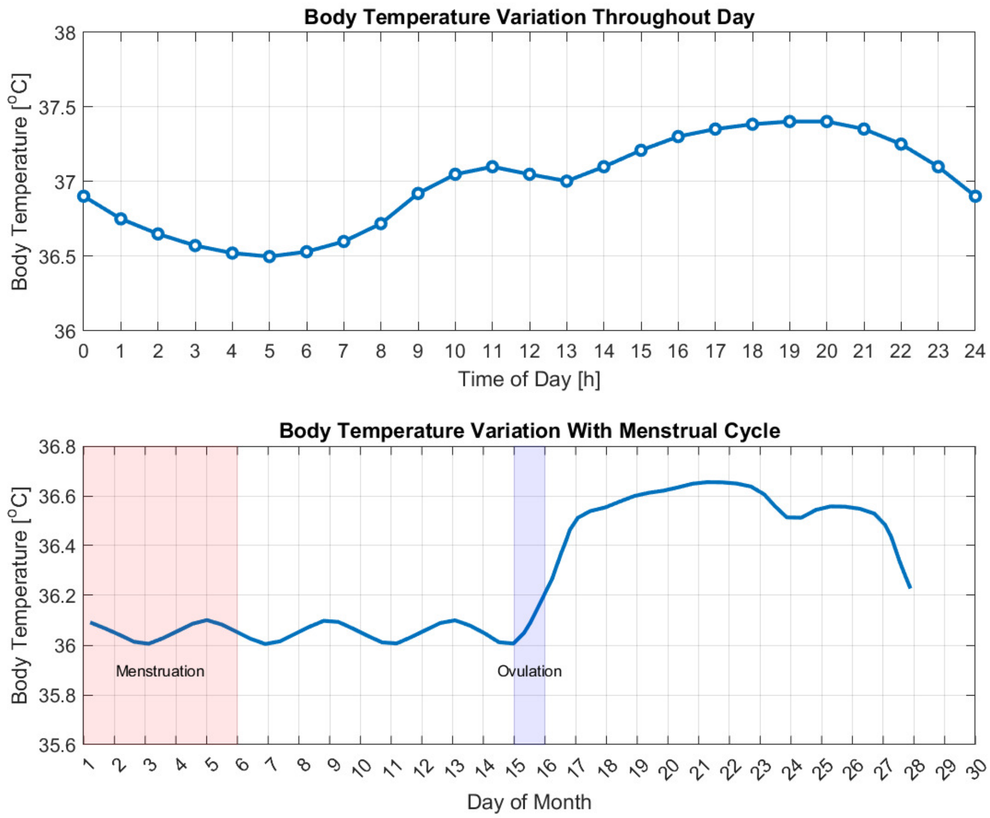


Figure 1-5: Body temperature variation with circadian rhythm and menstrual cycle, adapted from [22].

Radar Sensors

There is significant interest in detecting heart rate and respiration rate at a distance, for applications such as search and rescue and non-invasive monitoring of patients, elderly residents and infants [9]. A person's chest wall motion contains both large amplitude periodic expansion due to respiration and small amplitude displacements due to heart rate, both of which can modulate a reflected radio frequency wave [24]. There are several common radar configurations and corresponding signal processing techniques that have been used for this purpose, and, based on selected research efforts, they are compared in Table 1.2 and described in detail below [9].

Continuous Wave (CW)

One of the most commonly used radar approaches is CW radar, which sends a continuous waveform at a single radio frequency toward a target [9]. In it's simplest implementation, it mixes (multiplies) the returning signal with two copies of the sent signal. One copy is the transmitted signal, and the other is the same signal delayed by 90°. With a few additional components, this produces what are called in-phase (I) and quadrature (Q) waveforms [9]. The basic signal processing approach to extract RR and HR is to take magnitude of the two orthogonal I and Q signals and perform a Fast Fourier Transform (FFT) on the result to extract the contained frequencies. The frequency content will include the respiration rate and its harmonics, the heart rate, as well as noise.

This approach relies on having a well-controlled environment, with a single, still subject in view, in order to make accurate measurements. Many filtering approaches exist to minimize the effects of breathing harmonics and noise, with motion artifacts (MA) and random body movements (RBM) of the subject being the most difficult challenges to address [9].

Frequency Modulated Continuous Wave (FMCW)

A newer type of radar commonly used in the automotive industry, FMCW radar can identify multiple objects located in its path, as well as their motion [9, 25]. It works by sending out repeated "chirps", or short waveforms of linearly increasing frequency. The chirp is reflected off of objects in view and collected by the receiver, after which it is mixed with a copy of the original chirp. A "frame" can contain a one or more chirps. The frame duration is known as fast time, while the frame-to-frame spacing is called slow time. A radar system like this outputs a matrix of voltage readings with slow time on one dimension and fast time on the other [26].

Performing an FFT on a single received chirp frame (along the fast time axis of the matrix) and plotting the absolute values of the results reveals frequency peaks of objects detected. Higher frequencies correspond to objects further away. Converting the frequency axis to distance provides a convenient way to identify different objects in view. This resulting matrix of complex values is known as the Range FFT [26].

Then, assuming the object of interest is not moving significant distances (depending on chirp frequencies) across the frame slow time, in order to remain in the same frequency (distance range) bin, a second FFT (Doppler FFT) operation can then be performed on the phase angle of the Range FFT matrix, across the slow time axis. This corresponds to small motions of an object in that range bin [26]. From this Doppler FFT, the respiration and heart rate can be extracted in a similar way to the CW radar. This is described in more detail in Chapter 4.

Stepped Frequency Continuous Wave (SFCW) & Ultra-wideband Pulsed (UWB)

Two less common radar architectures have been used for vital sign detection. Similar to FMCW radar, SFCW systems send out a series of discrete, constant frequency pulses, increasing in frequency in uniform steps. Signal processing is similar to FMCW with the advantage being lower power consumption [9]. UWB radars reduce power consumption even more by using strobed sampling, where they send periodic broadband pulses, and then only sample echoes based on the time-of-flight to the desired target. These systems have more complex processing algorithms, but show promise in search and rescue applications [9].

RGB Camera

A number of researchers have explored using visible light cameras to detect color changes in subjects' skin to extract oxygen saturation, respiration rate, and heart rate. Unlike the relatively well-controlled, fast-sampling, and narrow wavelength finger PPG sensors, visible light cameras are challenged by unpredictable, broad spectrum lighting, slow frame rate, and subject movement. Guazzi et al. created an algorithm based on the mostly linear re-

Radar	Advantages	Disadvantages
CW	+ Simple operation *(ideal conditions)* + Most common	- No multi-subject capability - Sensitive to RBW - Med power efficiency
FMCW	+ Range information + Many evaluation kits	- Affected by MA - Low power efficiency Affected by Resp. Harmonics
SFCW	+ High sensitivity + Works w/various body orientations	- Affected by MA - Med power efficiency - Affected by Resp. Harmonics
UWB	+ High power efficiency + Detection behind objects + High SNR	- Regulations limit range

Table 1.2: Comparison of radar types used for vital sign detection based on studies reviewed in [9].

lationship between SpO_2 and the log ratio of ratios of AC/DC blue light to AC/DC red light detected by the camera [27]. The blue channel is poorly absorbed by blood and well absorbed by melanin, so it provides information about subject movement (and therefore respiration rate), while the red detects the pulsatile behavior of blood near the surface of the skin. They employed a novel technique to evaluate regions of interest on the subjects' skin for signal to noise ratio (SNR) and only used the highest SNR regions for final data processing.

The researchers used a special chamber to control lighting conditions and a physician operated specialized breathing equipment to lower subjects' oxygen saturation to a clinically relevant 80% and back to normal, for several cycles. Visible light PPG relies on carefully setting up the environment for consistent reflectivity of background surfaces and control of incident ambient lighting (like using Tyvek wrap and uniform LED panels in [27]). Subjects must remain still and identical camera equipment needs to be used for an algorithm to produce consistent results. This effort used 16 frame-per-second video with 12 second averaging windows, which were able to capture saturation changes on the order of 1 minute. Future, faster frame rates would be able to provide better temporal resolution, more in line with finger pulse oximetry [27].

Other notable research efforts involve controlling incident wavelengths to use orange and red light [28], using a smartphone camera and its built-in LED light close to the finger like a traditional oximeter [29], and using two cameras with narrowband filters for red and green wavelengths [30]. So far, patents have been granted, such as [31], but no commercial solution is readily available.

1.3 Proposed Implementation of Non-contact Vital Sign Detection

During the height of the pandemic, multiple groups at MIT had an interest in tracking vital signs, for both COVID-19 research via wearable devices and spread prevention via access control. MIT Lincoln Lab and MIT Clinical Research Center collaborated to collect data from sleep trackers, pulse oximeters, thermometers, and surveys with the goal of exploring early detection methods and gaining an understanding of the prevalence of COVID-19 among the MIT community. MIT Sloan School of Business was interested in a more robust access control system to accelerate the restart of in-person learning.

In the summer of 2020, MIT implemented its campus-wide strategy of access control for allowing on-campus work which consisted of:

- Daily self-attestation of the absence of common COVID-19 symptoms
- Minimum twice-per-week nasal swab polymerase chain reaction (PCR) testing

A self-report of symptoms would prompt a visit to MIT Medical to for symptom verification and evaluation, while a positive test result would mandate a quarantine period of at least 10 days before returning to campus [32]. Adding an IR camera-based skin temperature check immediately before PCR testing was considered, and a device was constructed but never implemented.

MIT Sloan considered an enhanced attestation procedure which would require users to self-report any anomalous vital signs detected by a Garmin Vivoactive 4S smartwatch.

We at Device Realization Laboratory (DRL) proposed an additional solution to augment both efforts - a self-service station for collecting vital sign measurements using non-contact methods. This system could be placed at entryways and common spaces to conduct regular screening of users, both for enhanced attestation and research purposes.

Hardware

The following design requirements were proposed. The system should:

1. Be automated, easy to use, and not require an operator present.
2. Measure temperature in a non-contact way.
3. Measure respiration rate in a non-contact way.
4. Measure heart rate in a non-contact way.
5. Have a modular architecture to allow expansion or replacement of sensors.
6. Be easily transportable by non-technical staff, for flexibility in placement.
7. Be reasonably secure to prevent tampering with equipment.

8. Have a finished appearance for long-term use in visible areas.
9. Minimize cost and assembly time to allow for fast and affordable deployment of replicate units.

Temperature Screening

Using the previously summarized uncertainty information, Dell'Isola et. al. [22] propose a two-tiered screening procedure to address the shortcomings of IR temperature measurement, shown graphically in Figure 1-6. If the temperature measured by the IR device falls within its uncertainty range, a contact-based device would be used to make the final decision, based on a different threshold which takes int account its corresponding uncertainty. Measurements above or below the IR device's uncertainty range would not require a second test, and would be treated as a fail or a pass, respectively.

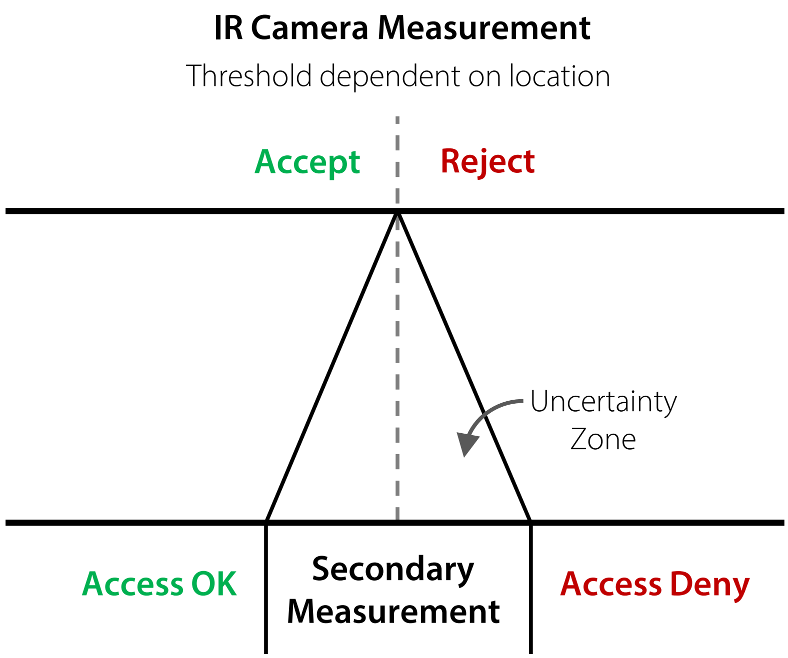


Figure 1-6: Visual representation of two-tiered temperature screening [22].

We suggest the above approach for immediate access control applications where a quick go/no-go decision is required. For research purposes, to develop early detection procedures, temperature should be collected at least daily from users, at the same times, due to the circadian rhythm effect previously mentioned. Ideally, temperature should be measured during the plateau that occurs after 5 pm, or around midday, in order of preference. See Figure 1-5.

Heart Rate and Respiration Tracking

Based on the wearable study results discussed, collecting controlled daily readings of RR and HR could make it possible to generate baseline profiles for each participant. This data, along with surveys of illness, stress, and/or COVID-19 test results could be used to retrospectively search for anomalies in vital sign data points. If a relationship can be established, this system can be used to potentially provide early warning signs to users who may otherwise be pre- or asymptomatic.

The following sections describe the process of designing and constructing a system that addresses these design requirements.

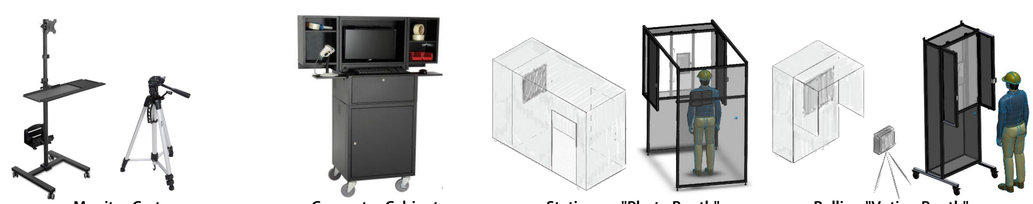
Chapter 2

Kiosk Design and Construction

2.1 Hardware Design

2.1.1 Structure Design

Several directions were considered for the overall embodiment of the system, summarized in Figure 2-1 alongside decision criteria. We ultimately settled for a mobile, modular instrument cart constructed from standard aluminum extrusions and hardware made by 80/20 Inc., Columbia City, IN, USA. With this approach, we prioritized modularity, while keeping cost and assembly difficulty reasonably low, by purchasing pre-machined components and hardware from a single vendor.



	Monitor Cart		Computer Cabinet		Stationary "Photo Booth"		Rolling "Voting Booth"	
Description	Commercial rolling monitor cart. Equipment would be mounted on built-in shelves and a rear tripod. Wires exposed. May require custom brackets.		Commercial locking cabinet provides secure storage for computer equipment. Requires ear tripod for mounting equipment and possibly custom brackets. Wires exposed.		Fully enclosed structure built with standard extrusions and paneling. Provides full customization and ample space for mounting and expansion. Can be made fully private and very secure.		Partially enclosed structure built with standard extrusions and paneling. Front and rear modules can be disconnected for occasional mobility. Folding design for partial prvacy.	
Modularity	1	poor	1	poor	3	easy	3	easy
Cost	3	low	2	med	1	high	2	med
Assembly	3	none	3	none	1	extensive	2	some
Security	1	none	2	some	3	high	2	some
Privacy	1	none	1	none	3	complete	2	some
Mobility	3	mobile w/o assem.	3	mobile w/o assem.	1	not mobile	2	mobile w/some assem.
Appearance	1	industrial	1	industrial	2	functional, clean	2	functional, clean
Totals	13		13		14		15	

Figure 2-1: Decision matrix showing concept directions considered.

The structure of the kiosk is a two-tiered rolling cabinet with privacy doors and connected rear mounting wall behind the user, seen in Figure 2-2. The main cabinet has a compartment for electronics, and a bottom compartment for additional storage or future hardware that might require more space. The frame is constructed with 80/20 15XX-LS series (1.5" square profile) T-slotted black anodized aluminum extrusions. The LS series has a smooth appearance instead of the typical ridge pattern normally seen on aluminum extrusions. The aluminum structure is covered with black HDPE plastic paneling, which is pre-cut into appropriate shapes with fastener clearances by 80/20. Together with the black anodized aluminum, this material provides a finished appearance suitable for everyday use in a highly visible setting. Large, lockable heavy duty rubber casters make the structure easy to move and secure into place.

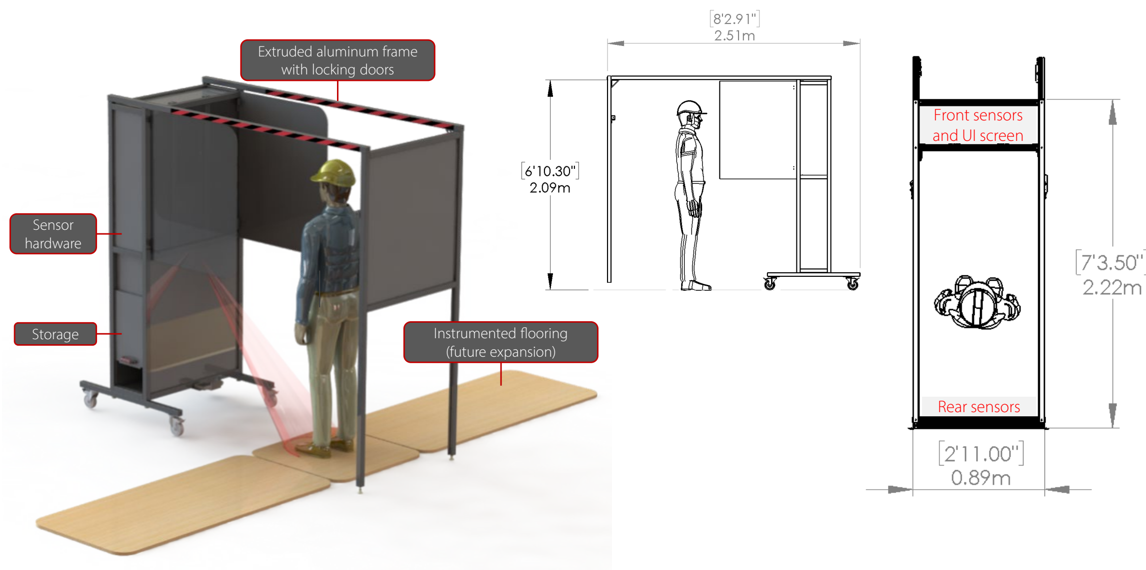


Figure 2-2: Frame Design Overview.

Assembly of the structure happens bottom to top, using anchor t-slot fasteners. Machining features for these are specified when ordering from 80/20. The assembly process is shown in Figures 2-3 and 2-4. When each layer is secured with the anchor fasteners, it is covered by the HDPE panel that resides in the t-slots. The top layer is further covered with extrusions that connect the main cabinet to the rear mounting wall with internal fasteners. This scheme prevents access to the anchor fasteners such that the system can only be disassembled from the top, using a long ball-head $\frac{1}{4}$ inch Allen wrench for the anchor fasteners and a long $\frac{3}{16}$ inch Allen wrench. During storage, the cabinet doors can be shut and secured with a padlock. For added security, the modular design allows for replacing the plastic panels with aluminum, to discourage cutting attacks.

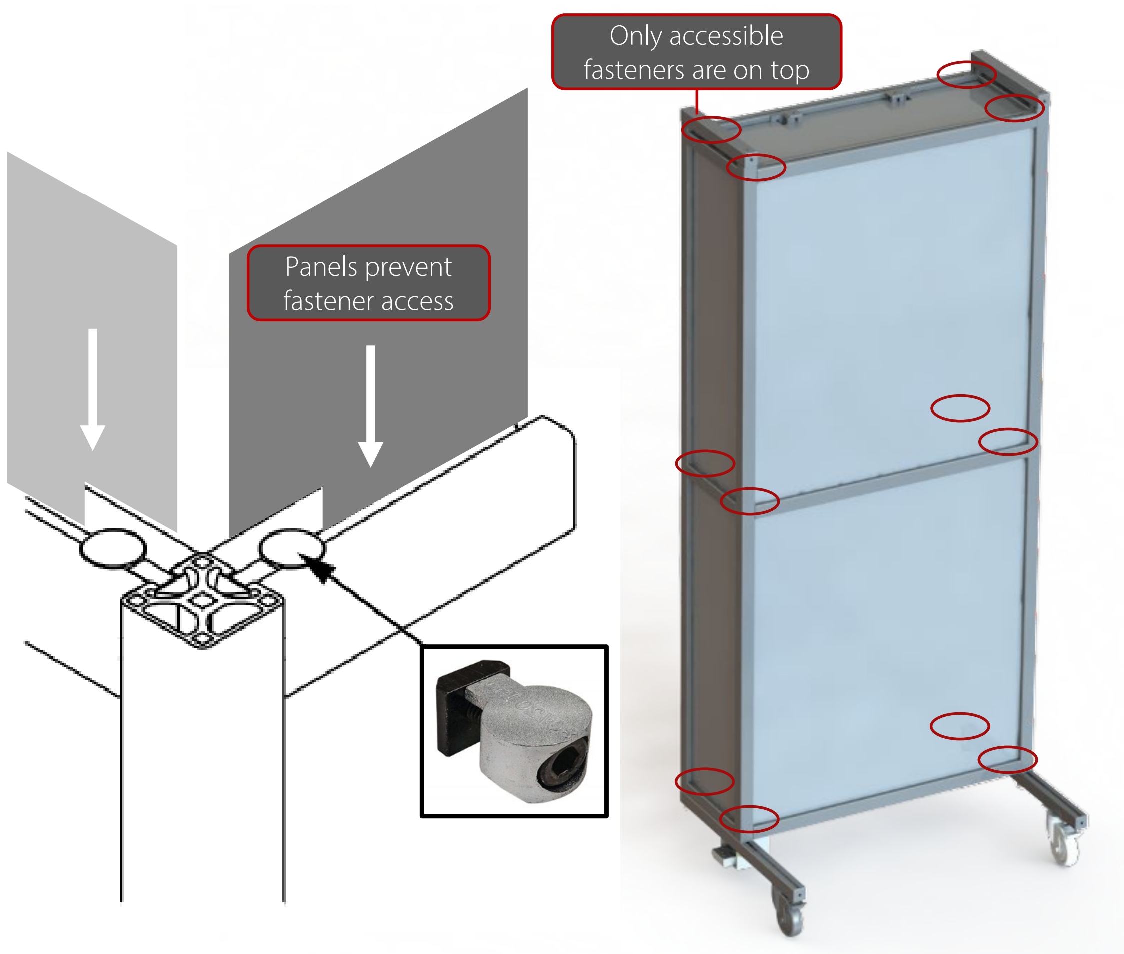


Figure 2-3: Secure frame assembly procedure.

The top compartment has two main vertical extrusion columns for mounting equipment that can be re-positioned left to right as needed. The t-slot profile of the extrusions allows for mounting items anywhere along their length in the vertical direction. In this implementation, the right column supports the all-in-one PC and monitor, while the left column has the sensor plate assembly mounted to it. The compartment is enclosed in a clear polycarbonate plate, with appropriate sensor cut-outs. This prevents users from easily interacting with the computer and tampering with the test software. The rear wall and extrusions connecting it to the main chamber allow for mounting of sensor hardware and running wires. The bottom compartment has a sliding access door and is used for storing the power cable, pedal controllers, and tools. It can be used to house additional equipment as necessary in the future.

Figure 2-4: Frame assembly process.

Figure 2-5: Completed kiosk installed in the Clinical Research Center, MIT Bldg E25, 2nd fl.

2.1.2 Hardware and Sensor Selection

Lenovo ThinkCentre M720 Mini PCs were selected for their easy integration into Lenovo Tiny-in-one 3 24” monitors (Lenovo Group Limited, Beijing, China). The monitors have touchscreens for future applications that might require user interaction. There were three main hardware choices to make - a thermal measurement device, a radar sensor, and an auxiliary sensor platform.

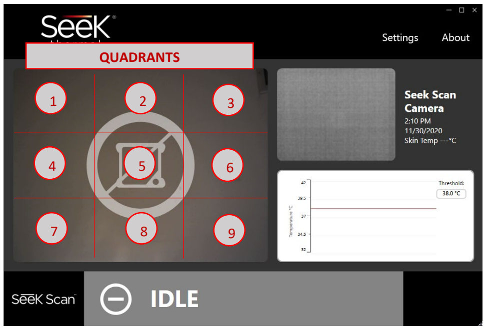
Temperature Measurement

The thermal cameras considered are shown in Figure 2-6. The Seek Scan system (Seek Thermal Inc. Santa Barbara, CA) was chosen for its combination of price, size, and easy availability. It consists of a camera module containing visible and IR cameras side-by-side, and an IR emitter module. The software uses the higher resolution visible camera to detect the square IR reference emitter and then finds the hottest pixel in the thermal camera image to compare to the temperature of pixels inside the detected reference square.

	ICI FM400 P Infrared Cameras Inc. Beaumont, TX, USA		**Seek Scan** Seek Thermal Inc. Santa Barbara, CA, USA		**Flir Lepton** Teledyne FLIR LLC Arlington, VA, USA		**Flir A-500 or similar** Teledyne FLIR LLC Arlington, VA, USA	
Description	Consumer grade thermal camera and blackbody calibrator kit built for human thermal screening applications.		Lower resolution thermal camera and blackbody calibrator kit built for human thermal screening applications.		Small-scale, low resolution camera module intended for fully custom applications.		High performance, multi-purpose thermal camera adapted for human temperature screening.	
Cost	1	~$15,000	2	$2,000	3	$300 + Custom electronics	1	~$10,000
Resolution	2	384x288	2	206x156	1	160x120	3	464x348
Claimed Accuracy	3	±0.3°C (0.54°F)	3	±0.3°C (0.5°F)	1	Greater of ±5°C (9°F) or 5%	3	±0.3°C (0.5°F)
Calibrator Included?	3	blackbody included	3	blackbody included	1	no blackbody	1	no blackbody ($1,700 option)
Ease of integration	2	bulky	3	small, self-contained	1	Custom hardware/software	2	bulky, rugged
API / Flexibility	2	basic functions	2	basic functions	3	fully customizable	2	basic functions
Platforms	1	Windows	1	Windows	3	all	3	all
Totals	14		16		13		15	

Figure 2-6: Decision matrix for thermal camera selection.

The manufacturer claimed an accuracy of ±0.3°C, using their provided temperature reference emitter. The camera was tested by dividing the display into 9 equal quadrants, and repeatedly testing the same subject in each quadrant. The results of the test, in Table 2.1, show that temperature of same subject varied by as much as 1.2°C between the top right and bottom left quadrants of the image. After several inquires to the manufacturer, no useful recommendation was provided to address this. The temperature variation within each quadrant was much lower, however. As a result, the camera would need to be mounted on a vertical linear stage and the position of the subject precisely specified to the center of the frame, to avoid temperature variation as much as possible.



PROCEDURE
1. Placed a non-reflective book in front of the camera
2. Unplugged for a few seconds
3. Plugged back in
4. Started the SeekScan App
5. Collected 25 datapoints in each of the 9 quadrants (blackbody in quadrant 4)
6. Moved the blackbody to quadrant 6
7. Repeated Step 5

* SET indicates the order in which samples were collected. Ex: SET 2 sample 2 was the 7th datapoint collected

BLACK BODY LOCATED IN QUADRANT 4, AFTER RESETTING WITH NON-REFLECTIVE COVER

		Sample No: (within group)									Sample No: (within group)									Sample No: (within group)						
QUAD	SET	1	2	3	4	5	Mean	STDEV	QUAD	SET	1	2	3	4	5	Mean	STDEV	QUAD	SET	1	2	3	4	5	Mean	STDEV
1	12	36.3	36.4	36.4	35.6	36.2	36.18	0.3347	2	1	35.3	35.3	35.4	35.5	35.3	35.36	0.0894	3	9	35	34.6	34.8	34.7	34.7	34.76	0.1517
1	18	36.2	36.2	36.2	36.3	36.2	36.22	0.0447	2	2	35.7	35.6	35.6	35.6	35.8	35.66	0.0894	3	13	34.9	35.2	35	35.2	35.1	35.08	0.1304
1	21	36	36.4	36.1	36	36	36.1	0.1732	2	20	36	35.8	35.8	36	35.8	35.88	0.1095	3	19	35.6	35.7	35.6	35.7	35.6	35.64	0.0548
1	28	36.4	36.3	36.3	36.3	36.5	36.36	0.0894	2	30	35.6	35.7	35.8	36	36.1	35.84	0.2074	3	29	35.1	35.4	35.3	35.1	35.2	35.22	0.1304
1	37	36.4	36.4	36.4	36.4	36.3	36.38	0.0447	2	39	35.6	35.9	35.4	35.5	35.8	35.64	0.2074	3	38	35.4	35.3	35.2	35.2	35.3	35.28	0.0837
						QUADRANT TOTALS:	36.25	0.1939							QUADRANT TOTALS:	35.68	0.2332							QUADRANT TOTALS:	35.2	0.3102
4	7	35.9	36.1	36.3	35.7	35.9	35.98	0.228	5	3	35.8	35.8	35.9	35.8	35.8	35.82	0.0447	6	8	35.2	35.3	35.2	35.1	35.1	35.18	0.0837
4	14	36	35.8	35.8	35.7	35.9	35.84	0.114	5	6	35.7	35.7	35.6	35.7	35.5	35.64	0.0894	6	15	34.5	34.6	34.6	34.9	35	34.72	0.2168
4	22	36.3	36.5	36.4	36.3	36.4	36.38	0.0837	5	24	35.9	36.1	36.2	35.9	36.2	36.06	0.1517	6	23	35.2	35.3	35.4	35.2	35	35.22	0.1483
4	31	36.7	36.6	36.4	36.4	36.4	36.5	0.1414	5	33	35.8	35.6	36.1	35.9	36.1	35.9	0.2121	6	32	35.1	34.9	34.9	35	35	34.98	0.0837
4	40	36	36	35.7	37	35.8	36.1	0.5196	5	42	35.9	35.9	35.9	35.9	35.9	35.9	0	6	41	34.9	35.1	34.9	35.1	35.1	35.02	0.1095
						QUADRANT TOTALS:	36.16	0.3512							QUADRANT TOTALS:	35.86	0.18							QUADRANT TOTALS:	35.02	0.2204
7	11	35.1	36.4	36.4	36.5	36.3	36.14	0.5857	8	4	36.1	36.1	36.1	36.2	36.1	36.12	0.0447	9	10	35.7	35.6	35.5	35.7	35.7	35.64	0.0894
7	16	36.5	36.6	36.5	36.6	36.5	36.54	0.0548	8	5	36.2	36.1	36.1	36.1	36	36.1	0.0707	9	17	35.4	35.3	35.6	35.8	35.5	35.52	0.1924
7	25	36.6	36.8	36.6	36.7	36.8	36.7	0.1	8	27	36.1	36	36.2	36.2	36.1	36.12	0.0837	9	26	35.3	35.5	35.5	35.5	35.7	35.5	0.1414
7	34	36.7	36.6	36.6	36.6	36.4	36.58	0.1095	8	36	36	36.1	36.2	36.1	36.2	36.12	0.0837	9	35	35.8	35.6	35.6	35.6	35.5	35.62	0.1095
7	43	36.4	33.1	36.4	36.6	36.5	35.8	1.5116	8	45	36.1	36.2	36.1	36.3	36.3	36.2	0.1	9	44	35.4	34.9	35.2	35.5	35.5	35.3	0.255
						QUADRANT TOTALS:	36.35	0.7473							QUADRANT TOTALS:	36.13	0.0802							QUADRANT TOTALS:	35.52	0.1972

BLACK BODY LOCATED IN QUADRANT 6, WITHOUT RESET

		Sample No: (within group)									Sample No: (within group)									Sample No: (within group)						
QUAD	SET	1	2	3	4	5	Mean	STDEV	QUAD	SET	1	2	3	4	5	Mean	STDEV	QUAD	SET	1	2	3	4	5	Mean	STDEV
1	1	37.2	37.3	37.1	36.8	36.9	37.06	0.2074	2	3	36.3	36.6	36.9	36.8	36.7	36.66	0.2302	3	2	36.3	36.4	36.7	36.7	36.6	36.54	0.1817
1	4	36.2	36	36.3	36.4	36.4	36.26	0.1673	2	6	36	36.4	36.5	36.5	36.4	36.36	0.2074	3	5	36.3	36.3	36.3	36.3	36.2	36.28	0.0447
1	7	36.2	36.3	36.4	36.6	36.1	36.32	0.1924	2	9	36.2	35.8	36	35.8	35.7	35.9	0.2	3	8	36.1	36.3	36.1	36.1	36.2	36.16	0.0894
1	10	35.8	36.2	35.9	36.1	36.2	36.04	0.1817	2	12	36.3	35.7	35.9	36.3	36.4	36.12	0.3033	3	11	36.2	36.1	36.1	36.1	36.1	36.12	0.0447
1	13	36.1	36.2	36.5	36.1	36.2	36.22	0.1643	2	15	36.3	36.3	36.3	36	36.3	36.24	0.1342	3	14	36.1	36	36.1	36.1	36.1	36.08	0.0447
						QUADRANT TOTALS:	36.38	0.3969							QUADRANT TOTALS:	36.26	0.328							QUADRANT TOTALS:	36.24	0.1912
4	16	37.2	37.1	37.2	37.2	37.2	37.18	0.0447	5	18	36.9	37	36.9	37	37	36.96	0.0548	6	17	35.9	36.3	36.3	36.1	36.4	36.2	0.2
4	19	37	36.8	36.9	37.2	36.9	36.96	0.1517	5	21	36.7	36.9	36.8	36.7	36.7	36.76	0.0894	6	20	36.1	36.1	36.1	35.9	36.1	36.06	0.0894
4	22	36.3	36.8	36.7	36.8	36.8	36.68	0.2168	5	24	36.5	36.9	36.6	36.7	36.8	36.7	0.1581	6	23	35.9	36.1	36	36.1	36.1	36.04	0.0894
4	25	36.8	36.7	36.7	36.7	36.9	36.76	0.0894	5	27	36.5	36.6	36.6	36.8	36.8	36.66	0.1342	6	26	35.8	36	36	36.2	36	36	0.1414
4	28	36.9	36.6	36.6	36.8	36.8	36.74	0.1342	5	30	36.6	36.6	36.6	36.8	36.8	36.68	0.1095	6	29	35.7	36.1	36.1	36	36.1	36	0.1732
						QUADRANT TOTALS:	36.86	0.2271							QUADRANT TOTALS:	36.75	0.1531							QUADRANT TOTALS:	36.06	0.1528
7	31	37.5	37.4	37.5	37.5	37.5	37.48	0.0447	8	33	37.1	37.3	37.3	37.2	37.3	37.24	0.0894	9	32	36.2	36.4	36.3	36.4	36.3	36.32	0.0837
7	34	37.3	37.3	37.1	37.2	37.1	37.2	0.1	8	36	37.1	37.1	37.2	37	37	37.08	0.0837	9	35	36.4	36.3	36.6	36.6	36.5	36.48	0.1304
7	37	37.3	37.2	37.2	36.9	37.2	37.16	0.1517	8	39	37.1	37.2	37.1	37.1	37.2	37.14	0.0548	9	38	36.1	36.3	36.3	36.5	36.3	36.3	0.1414
7	40	37.3	37.4	37.4	37.4	37.4	37.38	0.0447	8	42	37	37	37.1	37	37	37.02	0.0447	9	41	36.4	36.4	36.3	36.5	36.5	36.42	0.0837
7	43	37.5	37.5	37.3	37.3	37.5	37.42	0.1095	8	45	37	37.1	37	37.2	36.9	37.04	0.114	9	44	36.1	36.2	36.2	36.1	36.2	36.16	0.0548
						QUADRANT TOTALS:	37.33	0.1568							QUADRANT TOTALS:	37.1	0.1098							QUADRANT TOTALS:	36.34	0.1469

ANOTHER BOOK RESET

BLACK BODY LOCATED IN QUADRANT 6

		Sample No: (within group)									Sample No: (within group)									Sample No: (within group)						
QUAD	SET	1	2	3	4	5	Mean	STDEV	QUAD	SET	1	2	3	4	5	Mean	STDEV	QUAD	SET	1	2	3	4	5	Mean	STDEV
1	1	36.4	36	36	36.1	35.9	36.08	0.1924	2	3	36.2	36.2	36	36.3	36.1	36.16	0.114	3	2	35.9	36.3	36.1	36.3	36.2	36.16	0.1673
1	4	36.2	36	36.1	36.4	36.2	36.18	0.1483	2	6	36.4	36.2	36.2	36.4	36.4	36.32	0.1095	3	5	36	36.3	36.2	36.3	36.2	36.2	0.1225
1	7	36.3	36	36.2	36.1	36.1	36.14	0.114	2	9	36.4	36.4	36.2	36.4	36.3	36.34	0.0894	3	8	36.2	36.2	36.3	36.3	36.4	36.28	0.0837
1	10	35.6	35.6	36.1	36.1	36.1	35.9	0.2739	2	12	36.3	36.2	36	36.2	36.2	36.18	0.1095	3	11	36.1	36.1	36.1	36.3	36.4	36.2	0.1414
1	13	36.1	36	36	35.9	35.7	35.94	0.1517	2	15	36	36.4	36.2	35.9	36.1	36.12	0.1924	3	14	35.9	36.1	36.3	36	36.3	36.12	0.1789
UADRANT TOTALS:							36.05	0.2023	UADRANT TOTALS:							36.22	0.148	UADRANT TOTALS:							36.19	0.1412
4	16	36.8	37	37	37	36.9	36.94	0.0894	5	18	36.8	36.8	36.7	36.9	36.8	36.8	0.0707	6	17	36.1	36.3	36.3	36.2	36.2	36.22	0.0837
4	19	36.8	36.8	36.9	36.9	36.8	36.84	0.0548	5	21	36.8	36.6	36.6	36.6	36.8	36.68	0.1095	6	20	35.9	36.2	36.1	36.1	36.1	36.08	0.1095
4	22	37	37	37	37.1	36.9	37	0.0707	5	24	36.5	36.8	36.6	36.6	36.7	36.64	0.114	6	23	36.2	36.3	36.5	36.2	36.1	36.26	0.1517
4	25	37.1	37.1	37	37.3	37	37.1	0.1225	5	27	36.9	37.1	37.1	36.8	36.9	36.96	0.1342	6	26	36.1	36.2	36.2	36.4	36.2	36.22	0.1095
4	28	37	37.1	37.1	37.1	36.9	37.04	0.0894	5	30	37	36.9	37	36.9	37.1	36.98	0.0837	6	29	36.4	36.5	36.2	36.4	36.2	36.34	0.1342
UADRANT TOTALS:							36.98	0.1214	UADRANT TOTALS:							36.81	0.1716	UADRANT TOTALS:							36.22	0.1393
7	31	37.5	37.4	37.5	37.4	37.6	37.48	0.0837	8	33	37.1	37.2	37.2	37.2	37	37.14	0.0894	9	32	36.3	36.4	36.4	36.4	36.5	36.4	0.0707
7	34	37.5	37.3	37.3	37.4	37.5	37.4	0.1	8	36	37	37	37.1	37.1	37.1	37.06	0.0548	9	35	36.4	36.6	36.5	36.7	36.5	36.54	0.114
7	37	37.2	37.5	37.3	37.5	37.5	37.4	0.1414	8	39	37.1	37.2	37.2	37.2	37.1	37.16	0.0548	9	38	36.4	36.5	36.5	36.7	36.6	36.54	0.114
7	40	37.4	37.6	37.5	37.5	37.5	37.5	0.0707	8	42	37.2	37.2	37.2	37.2	37.1	37.18	0.0447	9	41	36.1	36.4	36.5	36.4	36	36.28	0.2168
7	43	37.4	37.1	37.9	37.3	37.3	37.4	0.3	8	45	37.3	37.1	37.3	37.3	37.4	37.28	0.1095	9	44	36.3	36.3	36.4	36.6	36.4	36.4	0.1225
UADRANT TOTALS:							37.44	0.1551	UADRANT TOTALS:							37.16	0.0995	UADRANT TOTALS:							36.43	0.16

Table 2.1: Testing regions of thermal camera for measurement consistency.

An additional challenge of the Seek Scan system was its limited API functionality. When connected to a PC, the camera acts as a server that can be queried via a URL request in JSON format. This allows the receipt of temperature data as well as images, but does not allow for the control of the camera. Instead, it operates automatically, and collects a reading every time it identifies a "new" user. This means a new reading can be initiated simply by looking away briefly. In this application, it was important to collect distinct readings, so a shutter mechanism was added to the camera using a servo motor, shown in Figure 2-7. Software written to interface with the camera waits for a new reading to be available, and closes this shutter to prevent further readings. The shutter reopens when a new reading is needed.

Figure 2-7: Seek Scan shutter mechanism.

A rudimentary test was performed to evaluate the performance of the thermal camera. A facial temperature was taken by the camera 18 times, where the hottest spots detected were in the inner corners of the eyes (inner canthus). Then 18 measurements were taken in the same locations (9 in each eye) using a thermistor-based consumer-grade thermometer, Kinsa QuickCare (Kinsa Inc. San Francisco, CA, USA), with a stated accuracy of ±0.11°C (±0.20°F) [33] as shown in figure 2-8. The average absolute difference was 0.26°C. Due to the surface temperature drop effect of contact-based sensors discussed earlier, it is reasonable for the contact thermometer reading to be lower. In the absence of more sophisticated test equipment, this agreement is enough for the IR camera to serve as an early warning tool that can identify users for further evaluation by medical professionals and/or additional temperature screening.

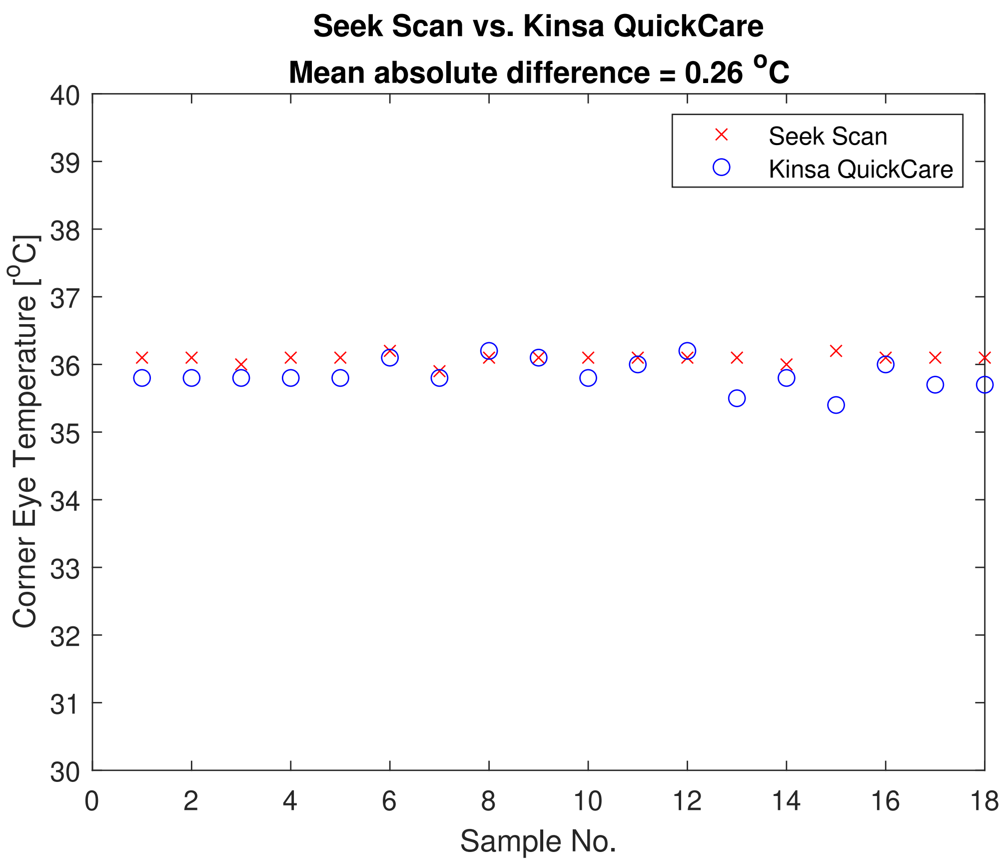


Figure 2-8: Comparing corner-of-eye temperature measurement between Seek Scan IR camera and Kinsa QuickCare thermometer.

Using the uncertainty reference data from Table 1.1, the expanded (>95% of cases), indoor uncertainty parameters were selected, without uncertainty attributed to calibration or drift. The calculated uncertainty for the Seek Scan setup is therefore 0.46°C. The two assumptions are: (1) that by using a blackbody calibration emitter, the system will not drift or require calibration, and (2) that subjects are tested during the middle or end of their work day in campus buildings, which reduces the likelihood of their temperature being affected by vigorous activity or extreme hot or cold environment exposure immediately prior to testing. Based on these values, the proposed two-tier testing criteria is shown in Table 2.2. Dell'isola et. al. recommend a contact thermometer be used for an axillary (underarm) measurement in the second step. The standard threshold value of 37.5°C is lowered by the contact measurement uncertainty of 0.2°C. We think this is a good approach given that this measurement can be taken easily in the field by the subject, and the thermometer sanitized by any trained staff member. If a different method is desired, this value will need to be adjusted based on the body area measured and instrument uncertainty.

First Step Noncontact Temperature *Inner Canthus*	Second Step Contact Temperature *Axillary (Underarm)*	Action
$T \leq 37.0°C$	–	Access OK
$35.5 < T \leq 37.4°C$	$T \leq 37.3°C$	Access OK
	$T > 37.3°C$	Access Deny
$T > 37.4°C$	–	Access Deny

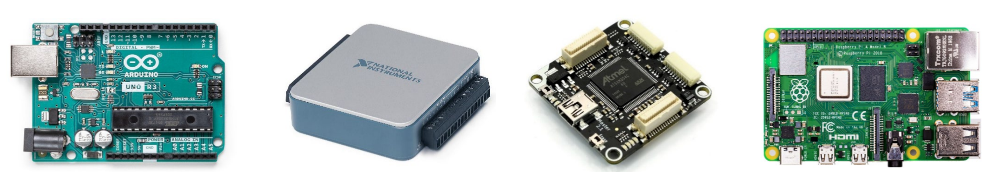
Table 2.2: Proposed MIT IR temperature screening criteria using Seek Scan, adapted from [22].

Radar Sensors

The kiosk uses a Walabot Developer Kit (Vayyar Imaging Ltd., Yehud-Monosson, Israel) FMCW 6.3-8 Ghz radar system. It was chosen for its relative ease of use and fully packaged design. It is compact and connects via a single USB cable. Details about its implementation are in Chapter 4.

Auxiliary Sensors

In addition to the two main sensing devices, the system needed a platform for adding extra sensors and controllers to enable other aspects of user interaction. An overview of considered platforms is shown in Figure 2-9, where Tinkerforge (Tinkerforge GmbH, Schloß Holte-Stukenbrock, Germany) was chosen for its combination of ease-of-use (software API, GUI) and robust packaging (plug-and-play, durable connectors). It consists of larger "Brick" modules, like controllers and I/O boards, and smaller "Bricklets", which contain sensors.



	Arduino Arduino, LLC Boston, MA, USA	**NI DAQ** National Instruments Inc. Austin, TX, USA	**Tinkerforge** Tinkerforge GmbH Schloß Holte-Stukenbrock, Germany	**Raspberry Pi** Raspberry Pi Ltd. Cambridge, England, UK
Description	Low performance hobbyist microcontroller.	High performance DAQ system, would be impractical to implement on a large scale due to cost and software requirements.	Modular instrumentation platform that can be integrated in all the most common languages. Many sensor modules available with robust packaging and wiring. Can be paired with Raspberry Pi for more capability.	Self-contained mini computer and micro-controller and daq with extra add-on shields. Can possibly double as the main computer running the kiosk interface.
Cost	3 low and broadly available	1 costly hardware/software	2 ~$100-150	2 ~$100
Ease of integration	2 messy wiring, difficult PC intfc	1 LabVIEW req'd	3 desktop app, multi-lang. API	2 Req. monitor or remote conn.
Customization	3	3	2	3
Performance	1	3	2	2
Platforms	2 Windows, Linux	2 Windows, Linux	3 Windows, Linux, Ras.pi	1 Self (Linux), others via ethernet
Totals	11	10	12	10

Figure 2-9: Decision matrix for sensor platform selection.

For housing the sensors, a mounting plate was constructed as shown in Figure 2-11. It is a $\frac{1}{4}$" thick Delrin plate that can be quickly adapted to house additional hardware if needed.

In the front, a ballscrew linear stage (FUYU FLS40 Series G1610, FUYU Technology Co., Ltd., Chengdu city, China) with a NEMA 23 stepper motor was mounted to position sensors at the proper height for each test subject. Its 15 kg weight limit provides plenty of overhead for any sensor payload. The thermal camera and radar sensor are mounted on the stage on ball head arms, with more mounting points available. Behind the linear stage is the stack of Tinkerforge components:

- Master Brick - for communicating with PC via USB
- Stepper Brick - for controlling the position of the linear stage
- Servo Bricklet 2.0 - for controlling the shutter on the thermal camera
- Humidity Bricklet 2.0 - for measuring ambient temperature and humidity
- IO-4 Bricklet 2.0 - for homing limit switch of the linear stage, and future inputs
- Motion Detector Bricklet 2.0 - for identifying when a user is present in the kiosk
- Distance IR 20-150cm Bricklet 2.0 - for measuring a user's height, mounted above their head

Figure 2-10 shows the custom designed and 3D-printed mounts used to locate remote components of the kiosk. A mount was also created to house a cross-shaped laser pointer, to mark the appropriate user location on the floor. However, it was not used in the implementation due to brightness of ambient light relative to the emitter. A stronger emitter will need to be used in the future. The rear wall of the kiosk has a t-slot extrusion for mounting the IR reference emitter and barcode scanner for user ID cards.

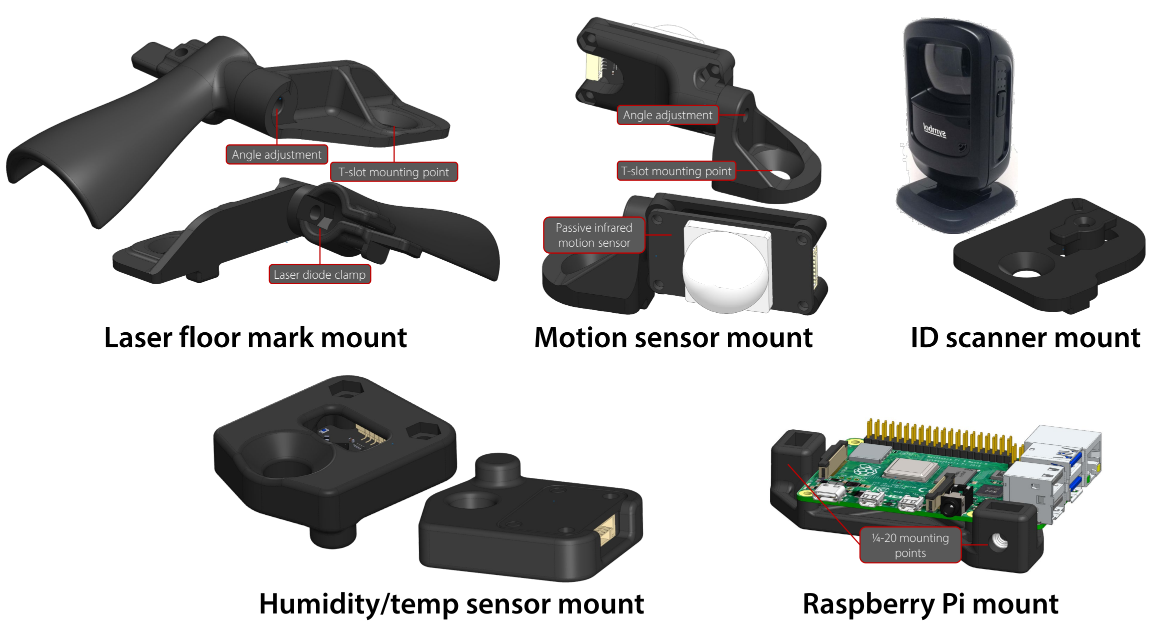


Figure 2-10: Designs for some of the sensor mounts used in the kiosk.

Figure 2-11: Sensor and linear stage controller layout on mounting plate.

2.2 Software Implementation and User Interface

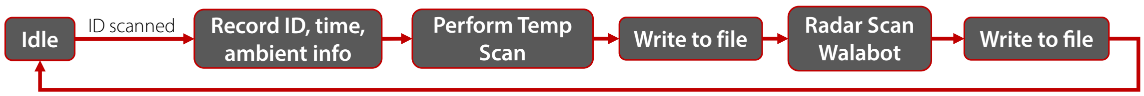


Figure 2-12: Operational flow of the kiosk system.

The operational state flow of the kiosk is outlined in Figure 2-12. The high-level overview of each state is listed below.

1. **Idle State:** Kiosk Information is displayed, and the system is tracking whether a user is present and the tallest height detected.
2. **Initialization State:** Records the user's encrypted MIT ID number, and generates a timestamp for the recording.
3. **Temperature Scan:** Adjusts linear stage to locate thermal camera in line wihh user's face. Performs three successive temperature scans and records as separate entries in a .csv file.
4. **Radar Scan:** Adjusts linear stage to locate radar antenna in line with user's chest. Performs radar scan and record to a binary file.
5. **Exit Instructions:** Information about scans and directions for the user to leave the kiosk.

The software to control the kiosk functions, collect data, and interact with the subject was written in Python 3. A main script runs the user interface and contains the top-level user scanning routines. Modules are then called to create instances of various pieces of hardware attached to the system. The modules are listed below:

- `POD_Main.py` - user interface & main operation flow
- `POD_Configuration.py` - various settings, to be located in a single file
- `POD_encrypt.py` - functions to encrypt user IDs
- `POD_radar_ti` - class and functions to control Ti mmWave radar
- `POD_radar_walabot` - class and functions to control Walabot radar
- `POD_RPi_link` - class and functions to communicate with Raspberry Pi via SSH
- `POD_seekscan.py` - class and functions to interface with Seek Scan thermal camera
- `POD_store_local.py` - functions to read and write collected user data
- `POD_tinkerforge.py` - class and functions to interface with all Tinkerforge hardware
- `RPi_client.py` - a script to run Joybien Ti mmWave board on Raspberry Pi

The user interface of the kiosk consists of only the computer screen, pedal, and barcode scanner, to the possibility of viral transmission via touching surfaces. The barcode scanner is used to begin the scanning process, and the pedal is an optional way for users to skip instructions if they are already familiar with the process. The user interface overview is shown in Figure 2-13. The user interface is built using the Tkinter Python library, which breaks up the screen vertically into several sections. From top to bottom, the sections are:

- **Banner:** Contains title and lab information.
- **Icons:** Shows a visual representations of the current state of the system (idle, temperature scan, radar scan)
- **Status:** Displays text-based instructions, status updates, and timers as necessary for different system states.
- **Info:** Contains user ID entry field, the start button, station information, and ambient temperature and humidity readings.
- **Debug:** An optional section with buttons and quick settings to help prototype changes in the interface. Not visible to the test subjects during normal use.

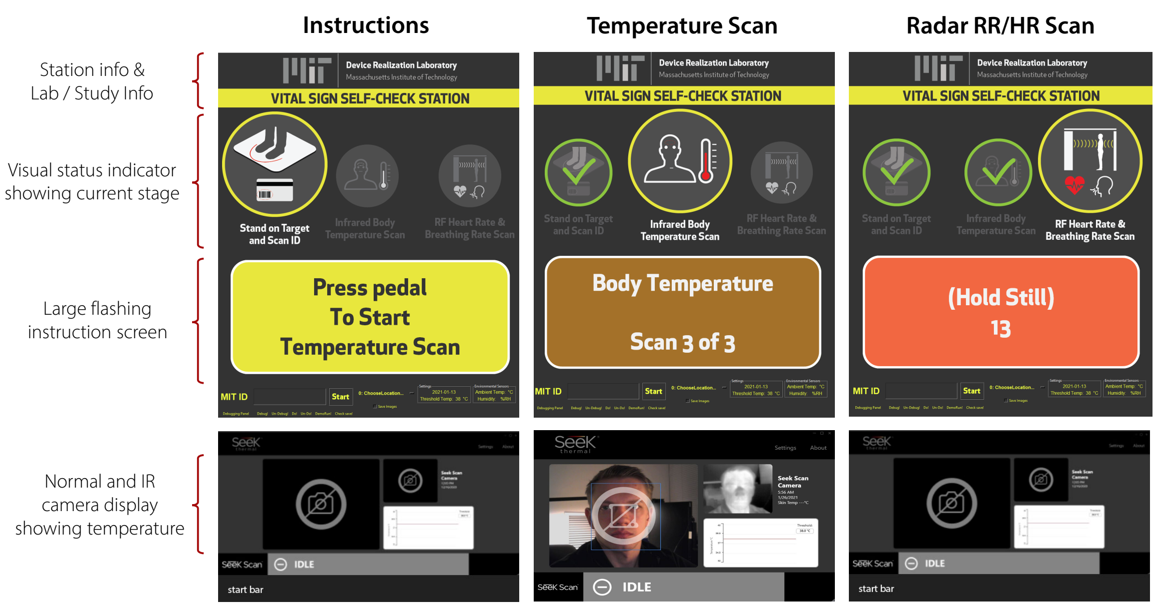


Figure 2-13: User interface overview.

Images from the actual kiosk scan process are shown in Figure 2-14.

Instructions — ID Scan triggers height detection and adjustment for thermal camera scan

Temperature Scan — Camera shutter activates after every successful acquisition

Radar RR/HR Scan — Height is adjusted lower to align radar sensor with chest

Figure 2-14: Kiosk in use showing interface and height adjustment.

Chapter 3

Chest Displacement Simulator

To aid in the testing radar hardware and development of algorithms, a mechanical simulator was constructed to approximate human chest wall motion. The following section describes its design and functionality.

3.1 Chest Displacement Model

Chest displacement was simplified to 1-dimensional motion normal to the coronal plane and modeled based on the work of [24, 34–36]. It is a sum of two time-varying signals as in 3.1.

$$x(t) = x_{\text{breath}}(t) + x_{\text{heart}}(t) \tag{3.1}$$

Where breathing $x_{\text{breath}}(t)$ is modeled by a series of stepwise functions $x_{\text{bs}}(t)$ in 3.2 consisting of a parabolic inhalation component and an exponential exhalation component and is a result of fitting functions to measurements of pressure induced by respiratory muscles, as collected by [36].

$$x_{\text{bs}}(t) = \begin{cases} \frac{-K_b}{T_i T_e} t^2 + \frac{K_b T}{T_i T_e} t, & t \in [0, T_i] \\ \frac{K_b}{1 - e^{-\frac{T_e}{\tau}}} t^2 (e^{-\frac{(t-T_e)}{\tau}} - e^{-\frac{T_e}{\tau}}), & t \in [T_i, T] \end{cases} \tag{3.2}$$

T_i, T_e and T are inhalation, exhalation, and total breath periods, respectively. τ is the time constant and K_b sets the breathing amplitude. Typical displacement amplitudes are 4-12 mm [37] and typical respiration rates are shown in Table 3.1.

Age Group	**Respiration Rate (RR)** breaths per minute [brpm]
10 years	15-20
Adults	12-20
Adults ≥65	12-28
Adults ≥80	10-30

Table 3.1: Respiration rates for different age groups at rest [38].

Nosrati et al. measured chest wall acceleration due to cardiac cycles, converted it to chest wall position data, and fit a model to it as in Equation 3.3 and as shown in Figure 3-1. It consists of a Gaussian pulse multiplied by a conditioning function, which accounts for the individual's chest wall thickness, respiration amplitude, and heart rate.

$$x_{\text{hs}}(t) = \eta \cos(\omega t + \gamma \sin(\Omega t)) \cdot e^{-\frac{(t-b)^2}{c}} \tag{3.3}$$

Individual pulses with varying beat-to-beat intervals $t_{BB}[j]$ are added together to form the combined displacement due to the cardiac cycle $x_{heart}(t)$ in Equation 3.4. This allows the simulation of heart rate variability (HRV) and amplitude variation.

$$x_{\text{heart}}(t) = \sum_{i=1}^{N_{\text{BBI}}} x_{\text{hs}}(t - \sum_{j=1}^{i} t_{\text{BB}}[j]) \tag{3.4}$$

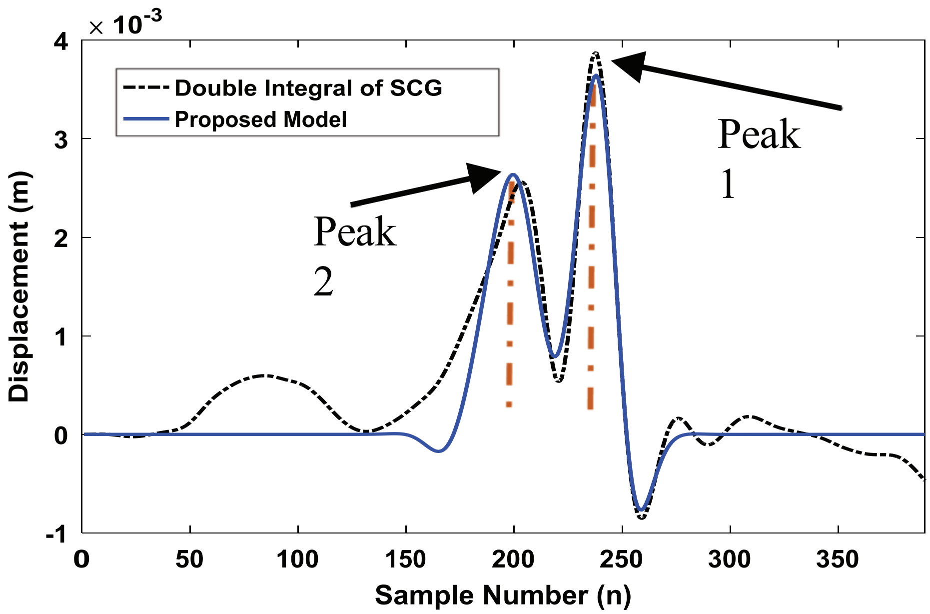


Figure 3-1: Chest displacement due to single cardiac cycle, with fitted model overlaid, from [24].

In our implementation, the model parameters were chosen such that the peaks were combined into a single peak, to simplify initial testing. An example of our simulated displacement waveform is shown in Figure 3-2. The “noisy” waveform includes random variation in HR, RR, and amplitudes. The function to generate chest displacement waveforms is shown in Appendix A.6.

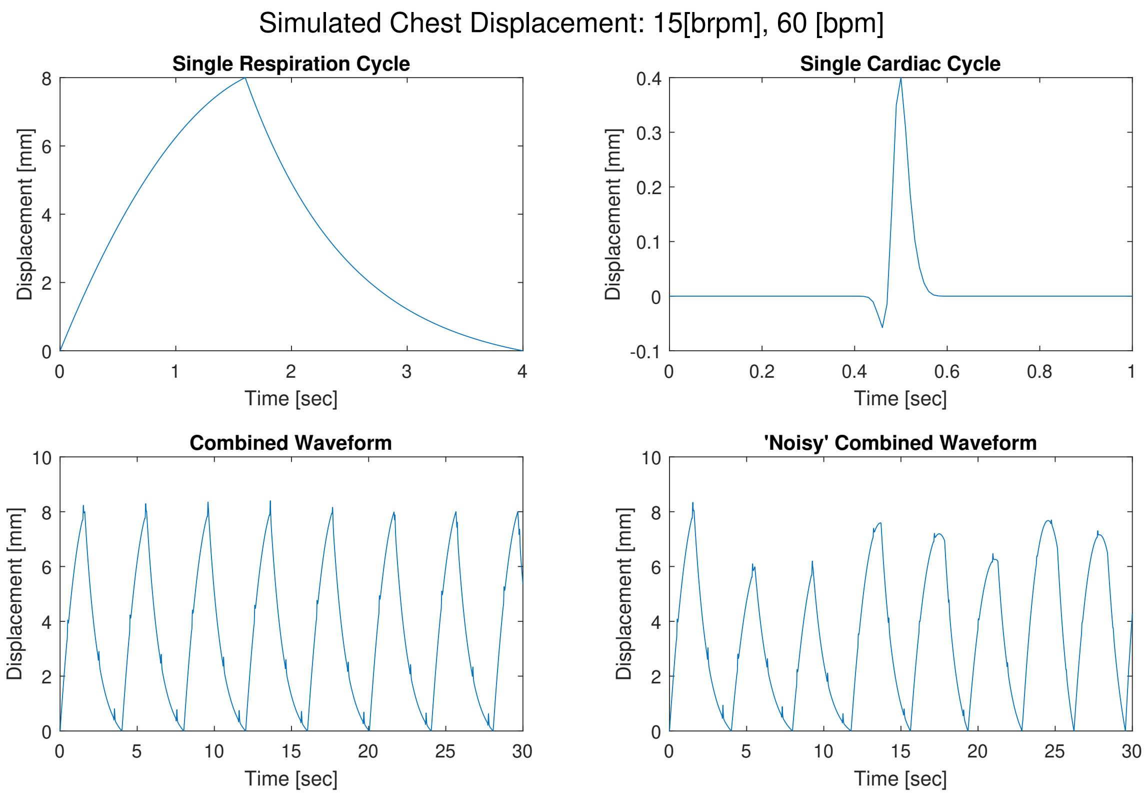


Figure 3-2: Summary of chest displacement waveforms.

3.2 Flat Plate 1-D Simulator

3.2.1 Hardware Construction

The one-dimensional chest wall motion simulator consists of a stiff 5 mm thick carbon fiber plate rigidly attached via an aluminum extrusion to linear ballscrew guide (FUYU FLS40, 10 mm/rev). It is driven by a NEMA 23 stepper motor with integrated encoder. The encoder and stepper motor are driven by a closed loop stepper driver (CL57Y, StepperOnline, OMC Corporation Limited, Nanjing, China), which receives instructions in the form of a PWM signal, which is generated by an FPGA controller (myRIO 1900, National Instruments Inc., Austin, TX, USA). The plate is covered with foil to increase radar reflectivity and convoluted RF absorber foam is used to reduce unwanted reflections. The setup is pictured in Figure 3-3 and a block diagram of the system is shown in Figure 3-4.

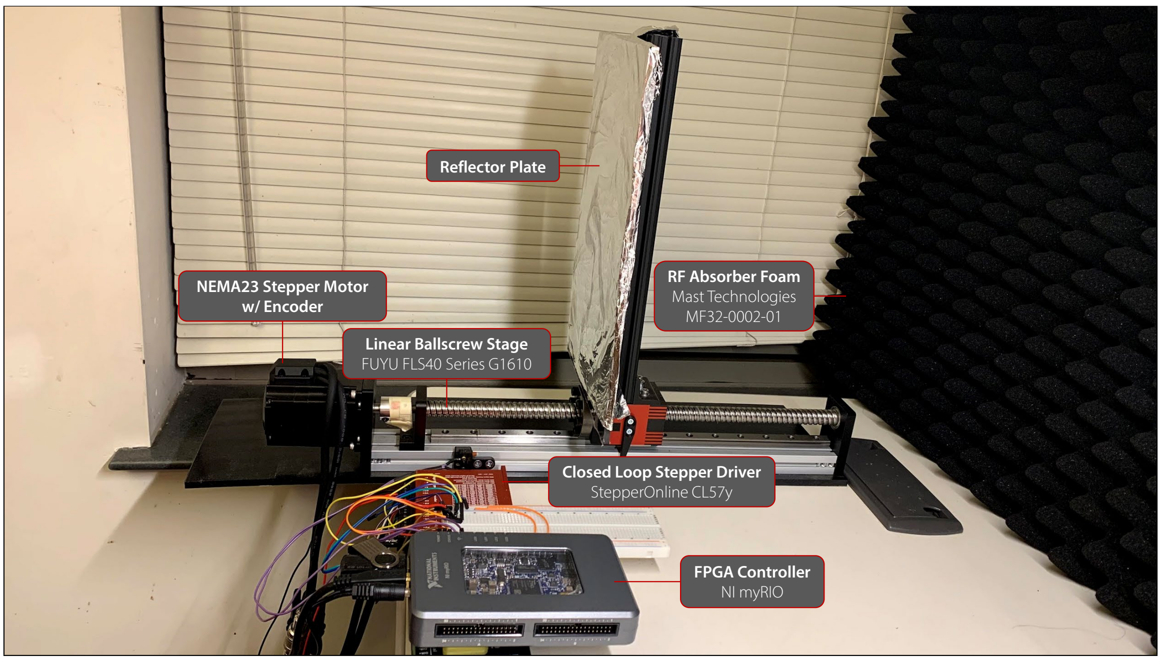


Figure 3-3: 1-D Chest displacement simulator hardware.

The stepper driver has an adjustable "pulses/rev" resolution setting, which allows for coarser, faster movement, or slower, more precise movement, depending on application needs. It has a maximum input frequency of 200 kHz, which, at the lowest resolution setting, corresponds to a theoretical maximum linear stage speed of 5 m/s. The FPGA controller's maximum output frequency is 100 kHz, thus corresponding to 2.5 m/s [39].

The stepper motor torque at lowest speed is 0.78 Nm [40]. Taking into account the inertia of the linear stage payload (J_{load}) and ballscrew (J_B) and the maximum acceleration of a cardiac pulse chest wall displacement, the torque required is 0.02 Nm, which is well within the capability of the motor. The cardiac and respiratory chest wall acceleration profiles are shown in Figure 3-5. The torque requirement is calculated by Equation 3.5 and parameter details are in Appendix A.1.

$$T_{\text{required}} = (J_{\text{load}} + J_B) \cdot \alpha_{\max} \tag{3.5}$$

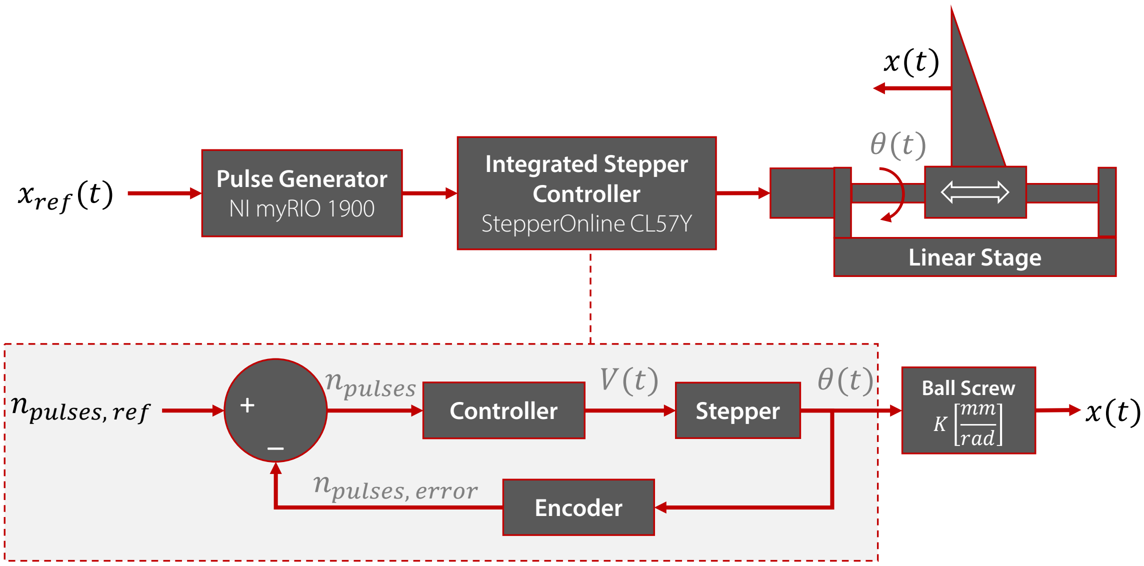


Figure 3-4: Block diagram of chest simulator function.

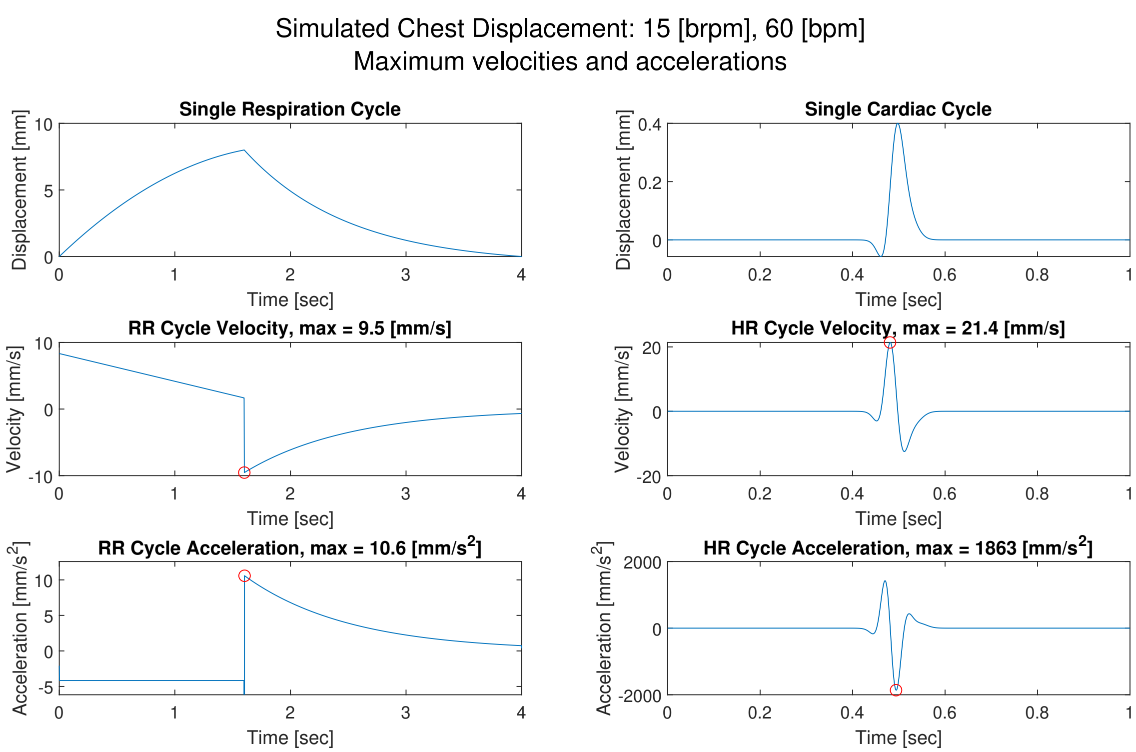


Figure 3-5: Maximum velocities and accelerations of chest motion simulation.

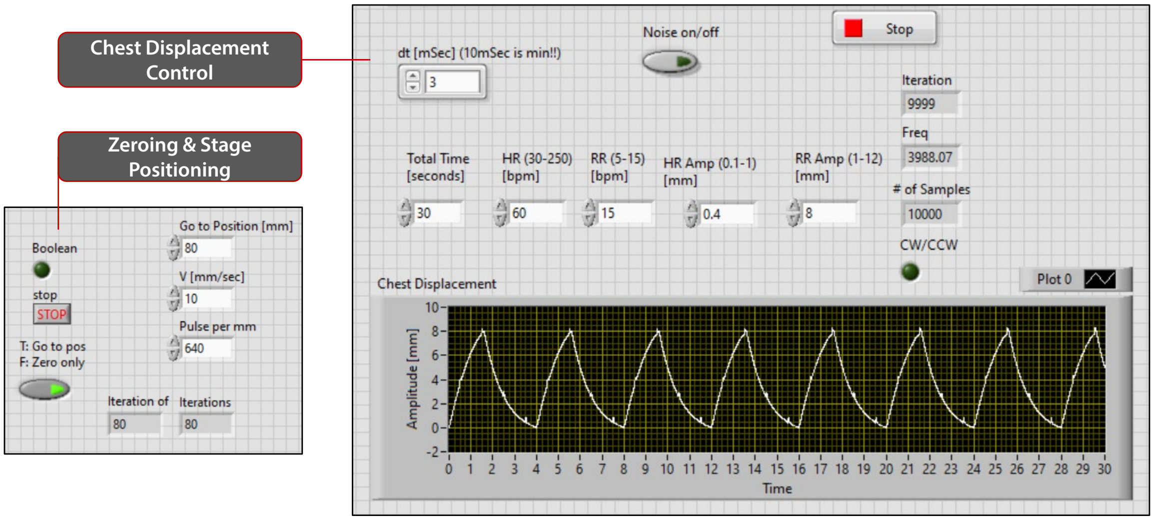


Figure 3-6: LabVIEW VI's: Left – To move the stage to a desired position, Right – To command the the stage to follow the desired chest wall displacement trajectory.

3.2.2 Software Implementation

The code (Appendix A.6) used to generate the waveforms was adapted for LabVIEW (National Instruments Inc., Austin, TX, USA), and an interface was written to convert the resulting displacement vector into discrete blocks of square wave pulses. The frequency of the square wave blocks corresponds to the derivative of the position vector. The square wave is discretized into blocks of time dt. The FPGA module generates the square wave according to this scheme and sends the signal to the stepper motor controller, which counts pulses and converts them to rotary motion based on its “pulses/rev” resolution setting. The LabVIEW VI for this operation is shown in Figure 3-7 and its corresponding control panel is shown in Figure 3-6. The panel has user-selectable settings for duration, frequency, amplitude, and noise. A secondary LabVIEW VI is used to position the stage at various locations, using a homing limit switch installed at one end, and is shown in Figure 3-8.

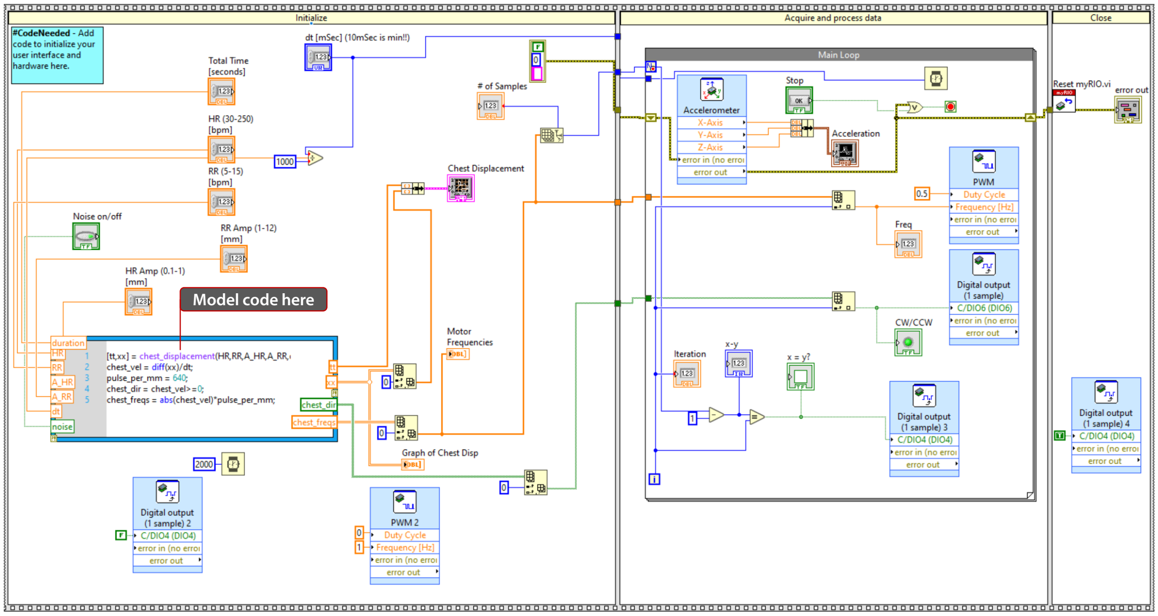


Figure 3-7: LabVIEW VI for waveform generating control scheme.

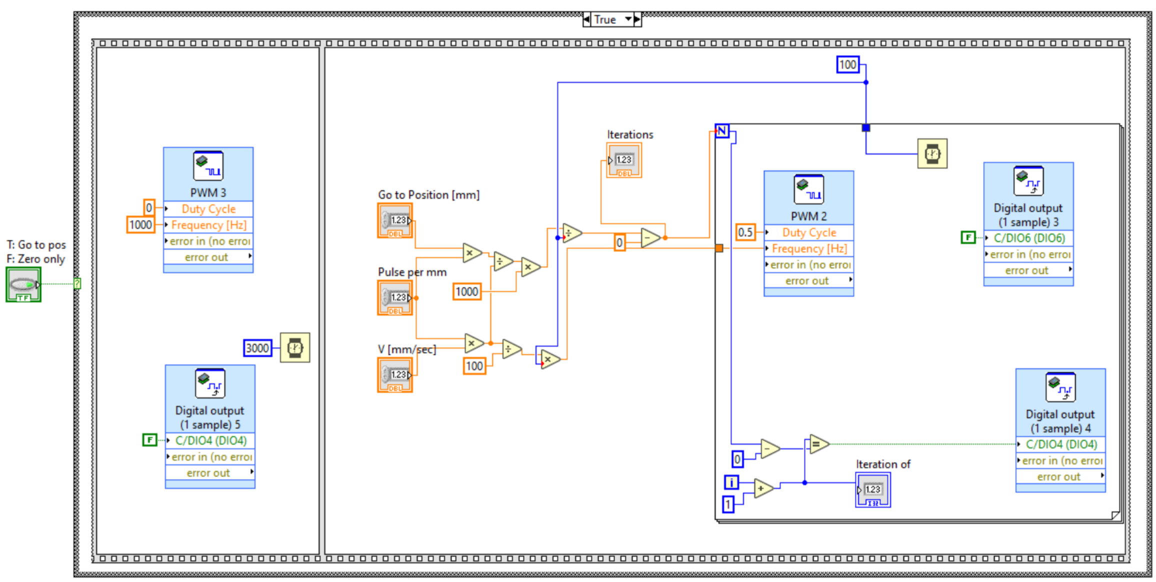


Figure 3-8: LabVIEW VI for positioning and homing the linear stage.

Chapter 4

Radar Sensor Implementation

4.1 FMCW Radar Sensors

An FMCW radar system sends out a "chirp" signal pulse of linearly increasing frequency, which is reflected off of a target and is detected by the receiving antenna. This received chirp is multiplied by the transmitted chirp (mixed), thereby producing a tone of constant frequency which depends on the time delay between the transmitted and received pulses. This mixed signal is known as the intermediate frequency (IF) signal. Every object in view will produce a separate reflection, so the IF signal will contain several frequencies, each corresponding to an object in view. An illustration of sent and received chirps is shown in Figure 4-1.

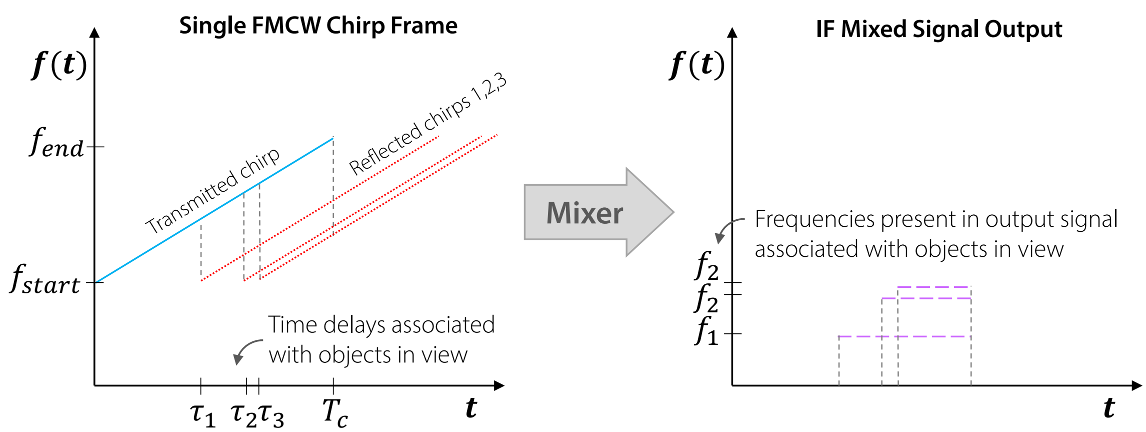


Figure 4-1: Illustration of multiple objects reflecting a chirp signal from an FMCW radar.

There are a few parameters to consider when choosing and configuring the FMCW radar system for a given application [26]. An FMCW radar system has two main "modes" of output: (1) locations of objects in its view (coarse resolution, high range) and (2) small oscillations of the objects in view (high resolution, low range). These parameters depend on the start & end frequencies of the chirp (bandwidth), chirp duration (T_c), chirp-to-chip spacing (T_f), and number of chrips per frame n_c.

Range of Objects

The range resolution of objects detected by an FMCW radar is given by 4.1, and depends on the on frequency spanned by the chirp (bandwidth). The maximum range detected occurs at the maximum possible IF frequency, which occurs if the reflection is received at the end of the chirp, and thus depends only on the duration of the chirp T_c, as in 4.2. For example, in the Walabot system used in the kiosk, this corresponds to a range of approximately 12m. In practice, however, the maximum range is determined by the sampling rate of the device (whether it can unambiguously measure the maximum IF frequency), signal strength, and interference.

$$d_{\text{res}} = \frac{c}{2(f_{\text{end}} - f_{\text{start}})} \tag{4.1}$$

$$f_{\text{IF,max}} = \frac{2([f_{\max} - f_{\min}]/T_c)d_{\max}}{c} \rightarrow d_{\max} = \frac{cT_c}{2} \tag{4.2}$$

Oscillations of Objects

Each frame of the FMCW radar contains frequencies corresponding to objects in view, and by looking at a series of frames, the phase offset of each frequency can be tracked from one frame to the next. In practice, this means transforming each frame into the frequency domain using FFT, and then looking at the phase angle of the resulting complex number frequency peaks. The maximum unambiguous phase difference of the object depends on the wavelength, as in 4.3. Higher frequency radars have a smaller range of measurement, but have higher resolution. This can be overcome using a phase unwrapping algorithm such as MATLAB `unwrap()` function.

$$\Delta\phi = \frac{4\pi\Delta d_{\max}}{\lambda} \rightarrow \Delta d_{\max} = \frac{\lambda(2\pi)}{4\pi} = \frac{\lambda}{2} \tag{4.3}$$

Additionally, if the phase of an object changes more than 180° between subsequent frames, velocity measurement becomes ambiguous. Thus, T_f must be adjusted in order to measure the desired velocity range as in 4.4. Finally, the velocity resolution (ability to distinguish between two oscillating objects of similar velocity) depends on the length of the chirp frame n_cT_c, as in 4.5.

$$v_{\max} = \frac{\lambda}{4T_f} \tag{4.4}$$

$$v_{\text{res}} = \frac{\lambda}{2n_cT_c} \tag{4.5}$$

During the course of this project, several radar systems were compared, and their calculated parameters are summarized in table 4.1. Note that the $v_{\max}$ and v_{res} parameters are standardized using the chirp time T_c of 33 ms and one chirp per frame n_c, which is the default and only possible setting on the Walabot. The Ti radars can be customized to change T_c, n_c, sampling rate, and bandwidth (within the range of f_{start} to f_{end})

Parameter	Unit	Radar Sensors		
		Walabot	**Ti IWR6843**	**Ti IWR1642**
f_{start}	GHz	6.3	60	76
f_{end}	GHz	8	64	81
f_{IF}(900 mm) $f_{\text{IF}} = \frac{2(f_{\text{end}} - f_{\text{start}})d}{t_{\text{chirp}}c}$	MHz	127	300	375
Range Bin Resolution $d_{\text{res}} = \frac{c}{2(f_{\text{end}} - f_{\text{start}}}$	mm	88.2	37.5	30.0
Measurable Position (Phase) Range inside range bin (Closest to Farthest) $\Delta d = \frac{\lambda \Delta \phi}{4\pi}$	mm	23.8 to 18.7	2.5 to 2.3	2 to 1.9
Maximum speed of vibrating object inside farthest range bin $v_{\max} = \frac{\lambda}{4T_c}$ ($T_c = 33$ ms)	mm/s	284	36	28
Velocity Resolution $v_{\text{res}} = \frac{\lambda}{2T_{\text{total}}}$ ($n_c T_c = 33$ ms)	mm/s	0.21	0.03	0.02

Table 4.1: Comparison of FMCW radar unit specifications [26]

4.2 FMWC Radar - Walabot

The Walabot Developer Kit is a self-contained FMCW radar and signal processing unit that connects to PCs via USB. It has an array of 18 transmit/receive antennas that can be used together for two-dimensional multi-object detection. The unit can output both raw FMCW frame signal as well as processed data, such as the location of objects detected. In our application, the unit is set up to output FMCW frames from single pair of antennas, each containing one chirp. The chirp duration is 0.8 ns and its frequency linearly increases from 6.3 GHz to 8 GHz. Due to limitations in the Walabot Python API, the frame-to-frame slow time is limited to 32.3 ms, on average.

When used in the kiosk, this sensor is to be paired with a distance measuring device, such as an ultrasonic or optical distance sensor, to provide the actual location of the subject, and narrow down the peak search in the Range FFT. While the Walabot is the easiest system to interface with that we have found, there are a number of additional limitations. There is no ability to customize the chirp duration or inter-chirp time, or to activate multiple antennas simultaneously to increase transmitted power.

The Walabot radar outputs voltage data in matrix format, as shown in Figure 4-2, where slow time between frames is rows and fast time is columns. Voltage data is the output of the mixer between the transmitted and received chirps.

Slow Time, Frames T_N = start time of Nth frame [sec] $\downarrow$	**Fast Time, Samples** t_n = time of nth sample [sec] $\rightarrow$			
T_0	V_{0,t_0+T_0}	V_{0,t_1+T_0}	$\cdots$	V_{0,t_n+T_0}
T_1	V_{1,t_0+T_1}	V_{1,t_1+T_1}	$\cdots$	V_{1,t_n+T_1}
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
T_N	V_{N,t_0+T_1}	V_{N,t_1+T_1}	$\cdots$	V_{N,t_n+T_N}

Figure 4-2: Voltage data output matrix from walabot FMCW radar.

4.2.1 Signal Processing

The basic signal processing procedure to extract vital signs is as follows:

1. Input signal matrix as in Figure 4-2 and subject distance (m) from radar sensor.

2. **Range FFT**. Convert each frame (from second row onward of the signal matrix) into the frequency domain using FFT.

   (a) Use a hamming window on the signal to reduce edge effects.

   (b) Apply zero padding at the end of the signal to smooth out the frequency plot.

   (c) Trim frequency axis to 6.3 to 8 GHz, since no other frequencies are present in the Walabot output signal.

   (d) Convert the frequency axis of the resulting FFT data into distance. Because of the linearly increasing chirp signal, lower frequencies correspond to reflections of closer objects and vice versa.

3. Using the frequency domain data generated in Step 2, create a plot of the magnitude of the FFT of the first frame and find the peak location that is closest to the known subject distance. This is the peak of interest, which occurs in every subsequent frame FFT, along the same column in the matrix.

4. **Doppler FFT**. Having selected the column of interest in Step 3, perform an FFT on the angle of the complex numbers in that column, to obtain the phase angle variation across the slow time axis. This corresponds to the small displacements occurring in the subject at that particular location.

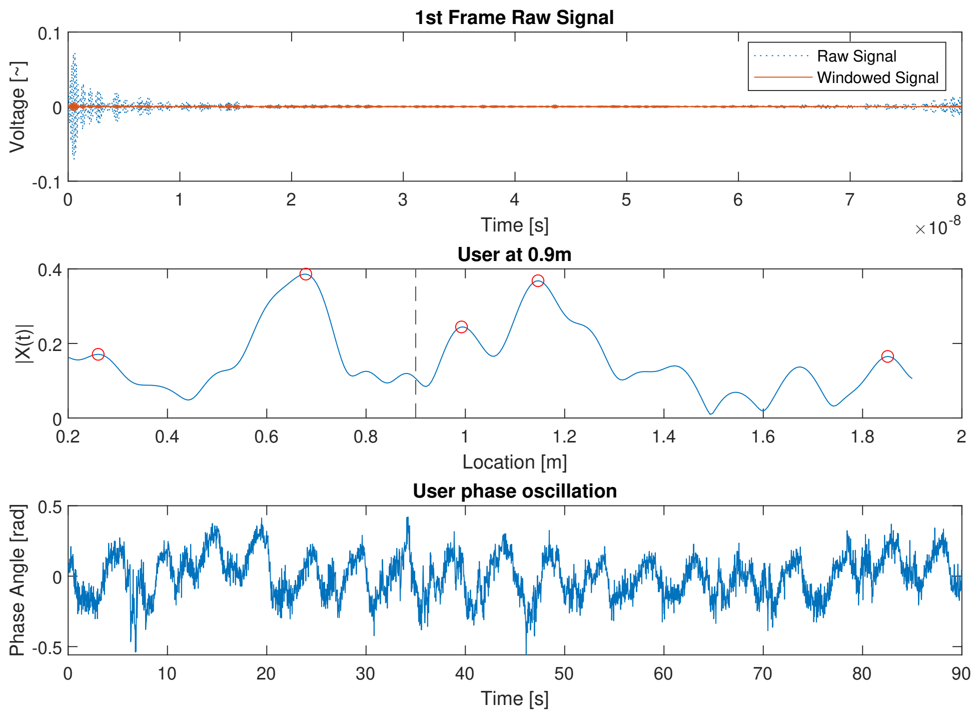


Figure 4-3: Processing steps to extract phase angle. From top: Single frame with Hamming window applied. Range FFT showing peaks corresponding to objects in view, with dotted line indicating approximate plate position. Doppler FFT showing phase angle of the selected frequency bin across the frames.

5. Perform an FFT on on the phase angle values to extract frequencies of oscillation present. Peaks in the frequency domain will represent breathing rate and heart rate. Additional filtering before this step helps narrow down to frequencies of interest - 0.07 to 0.5 Hz for RR and 0.7 to 2 Hz for HR.

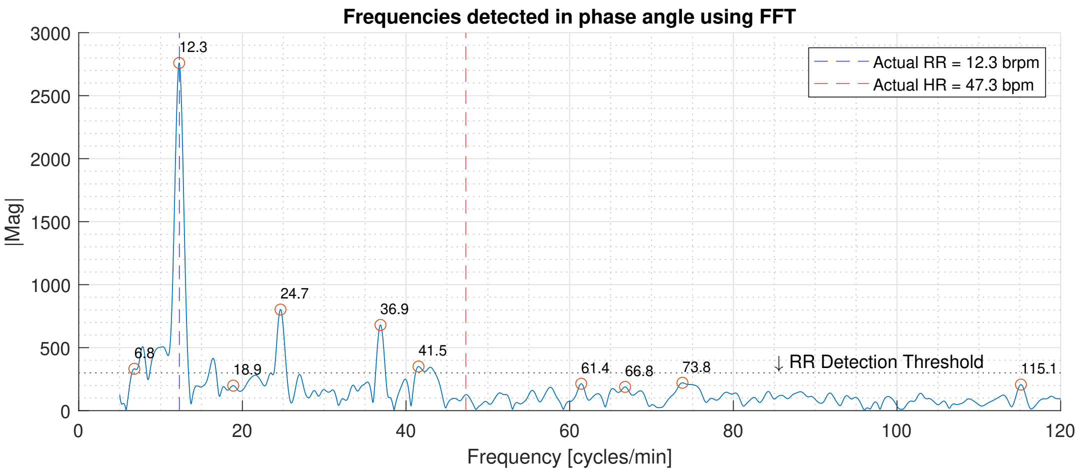


Figure 4-4: Extracting frequencies from phase angle. Dotted lines indicate the frequencies generated by the mechanical chest simulator, with correction factors applied. (Explained in 4.2.2.)

4.2.2 Testing Data

To test and develop the signal processing for Walabot, the experimental setup was used as shown in Figure 4-5. The Walabot is clamped in place a known distance away from the chest simulator and both are operated by the same PC.

For a visual comparison of how the phase angle corresponds to the movement of the chest simulator, Figure 4-6 shows the two waveforms overlaid. Both have been normalized by their means and standard deviations. The phase angle has been de-trended using a high order polynomial fit, and a 1.25 s moving average filter was applied to make the waveform easier to see.

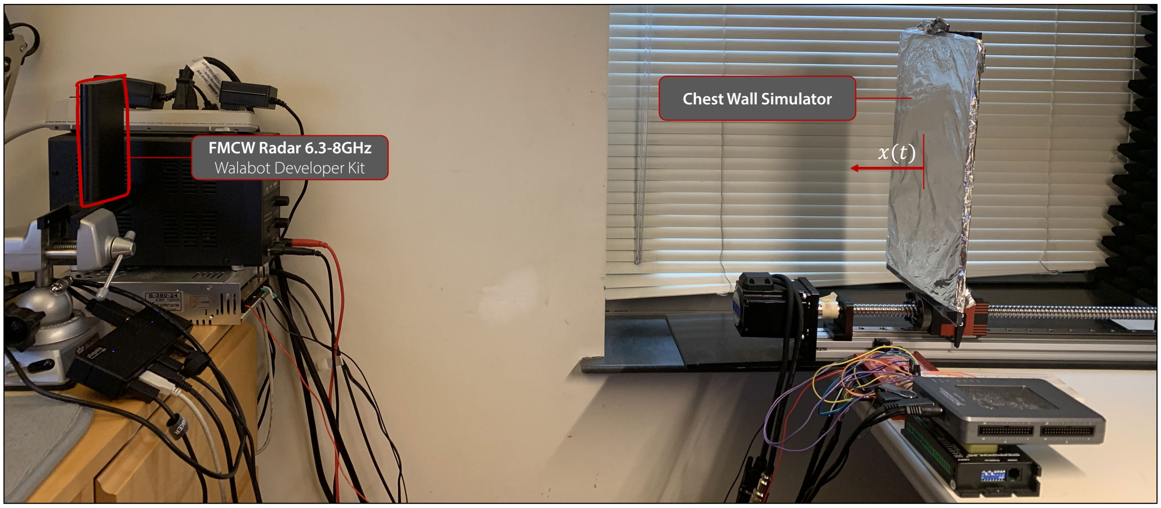


Figure 4-5: Experimental setup with chest simulator facing Walabot FMCW radar sensor.

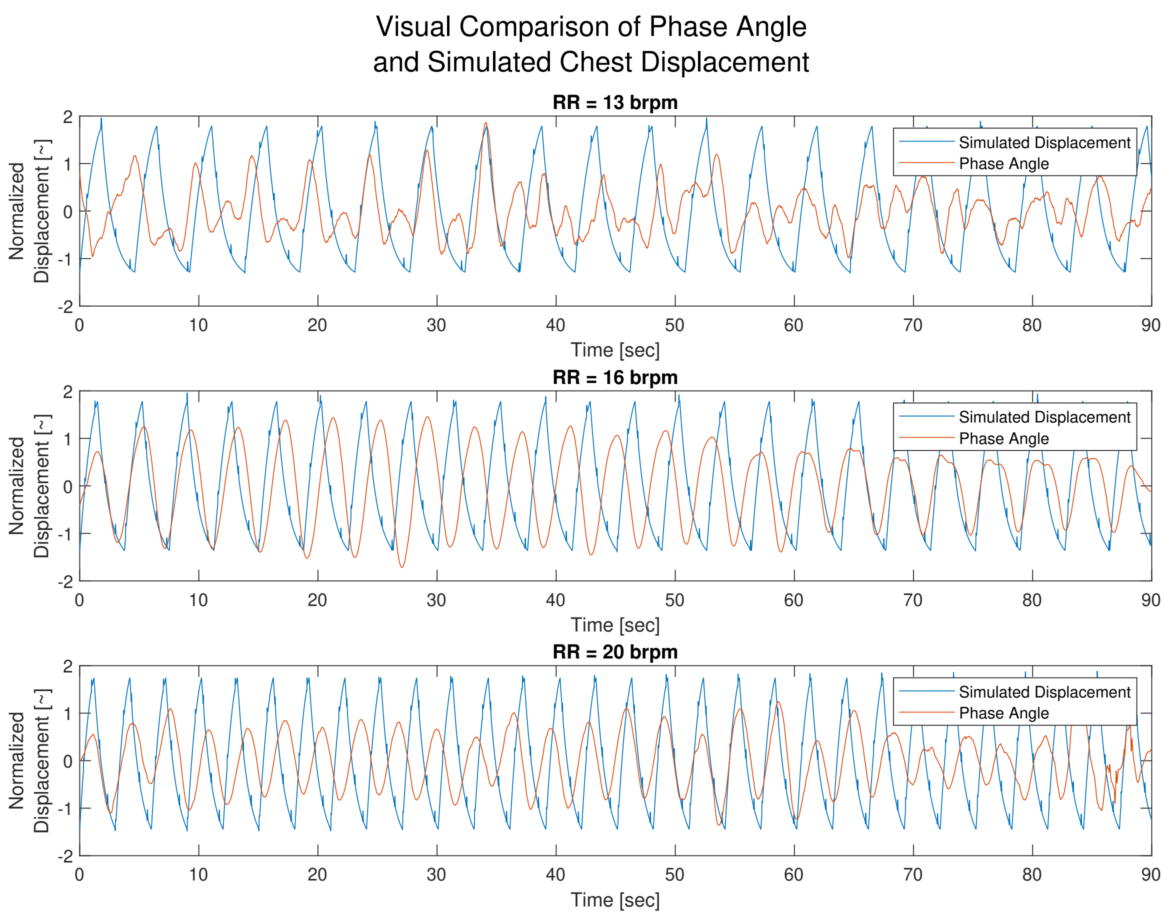


Figure 4-6: Comparison of normalized phase angle to simulated chest displacement. Phase angle has been de-trended and processed with a moving average filter.

In all three cases the measured phase oscillation frequency is slightly less than the simulated chest displacement. This is most like due to the control scheme used to operate the chest simulator, which is being run on a semi-open loop basis, and depends on the timing circuitry of the FPGA controller. Every time the controller runs through a loop to pass the next frequency to the stepper motor driver, it loses a little bit of time. This can be resolved in future iterations with direct closed loop control by wiring the encoder directly into the FPGA controller. To test this assumption, a 0.56 ms/sample accumulating delay was added to the simulated displacement waveform, and the agreement is much better, as shown in Figure 4-7. In further analysis, this correction factor will be used.

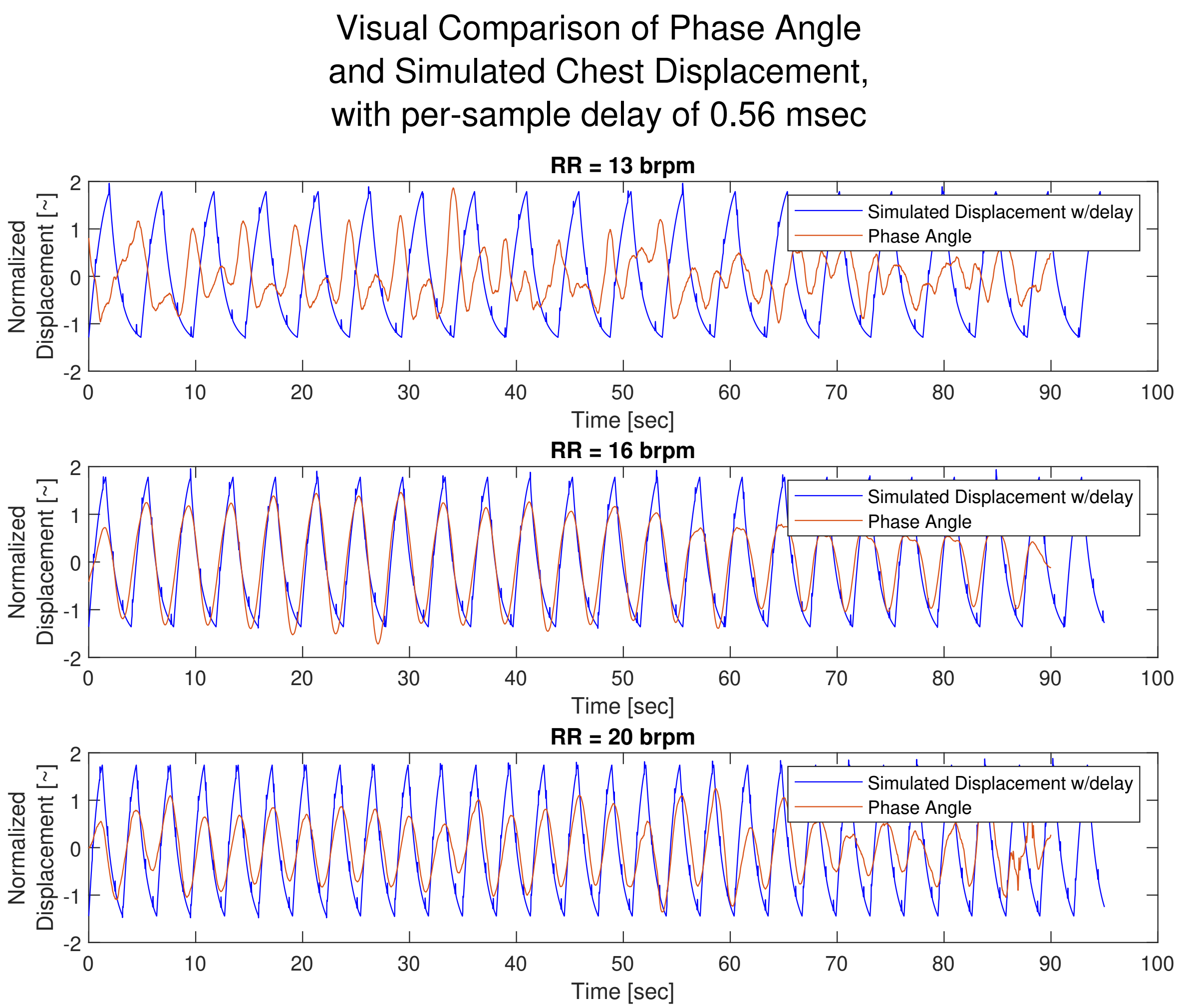


Figure 4-7: Comparison of normalized phase angle to *delayed* simulated chest displacement. Phase angle has been de-trended and processed with a moving average filter.

Extracting respiration rate and heart rate

The phase angle data is processed in two separate separate pathways in Step 5. Figure 4-8 shows the extracted respiration and heart rate frequencies side-by-side. It works well to identify the respiration rate, but struggles with the heart rate using the algorithm presented here, likely due to a low signal-to-noise ratio resulting from the low amplitude of the HR signal.

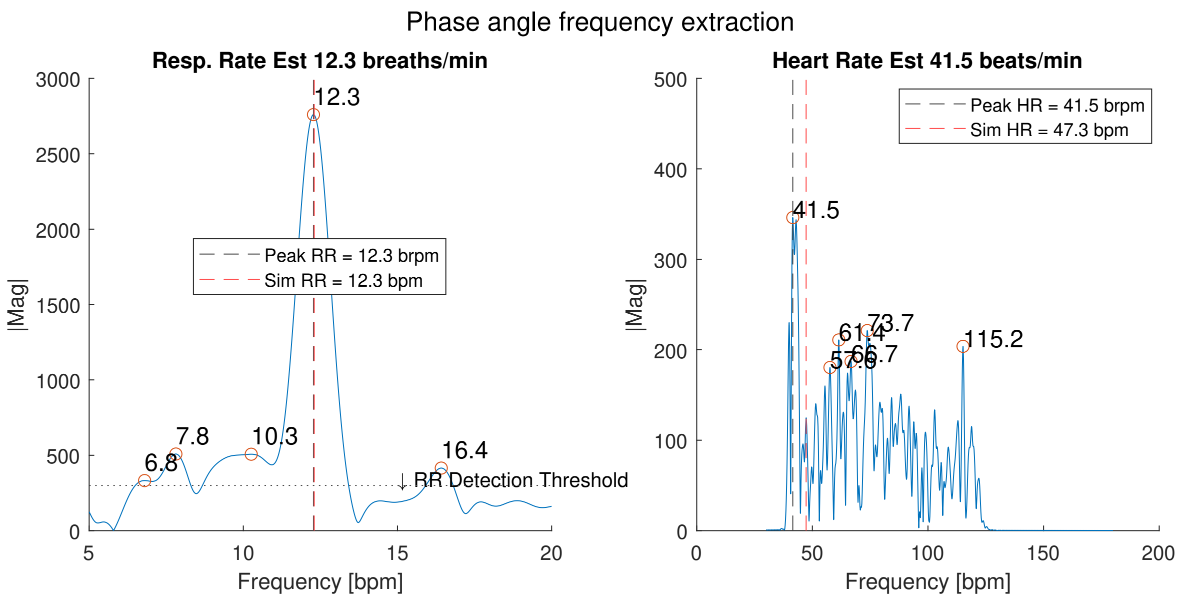


Figure 4-8: Separate phase angle frequency extractions. Chest simulator set up for RR of 12.3 brpm and HR of 47.3 bpm (corrected values).

Respiration sate subprocessing:

1. Apply a Hamming window to the sequence of phase angles to reduce edge effects in Doppler FFT and pad the signal with trailing zeros.
2. Normalize the signal to have zero mean, to eliminate the 0th frequency peak in the Doppler FFT plot.
3. Find the largest peak in the Doppler FFT that is above a certain experimentally determined threshold, which will correspond to the breathing rate. Convert it into brpm.

To test the accuracy of respiration rate measurement using this approach, the chest simulator was driven at RRs of 12 to 20 brpm with amplitude of 8 mm, while the HR was set to 70 bpm with amplitude of 0.5 mm. These amplitude settings are used throughout chest simulator testing. Each test was run for 90 seconds. The plot in Figure 4-9 shows very good agreement between the mechanical chest simulator and measurements extracted from the Walabot radar, with a low error.

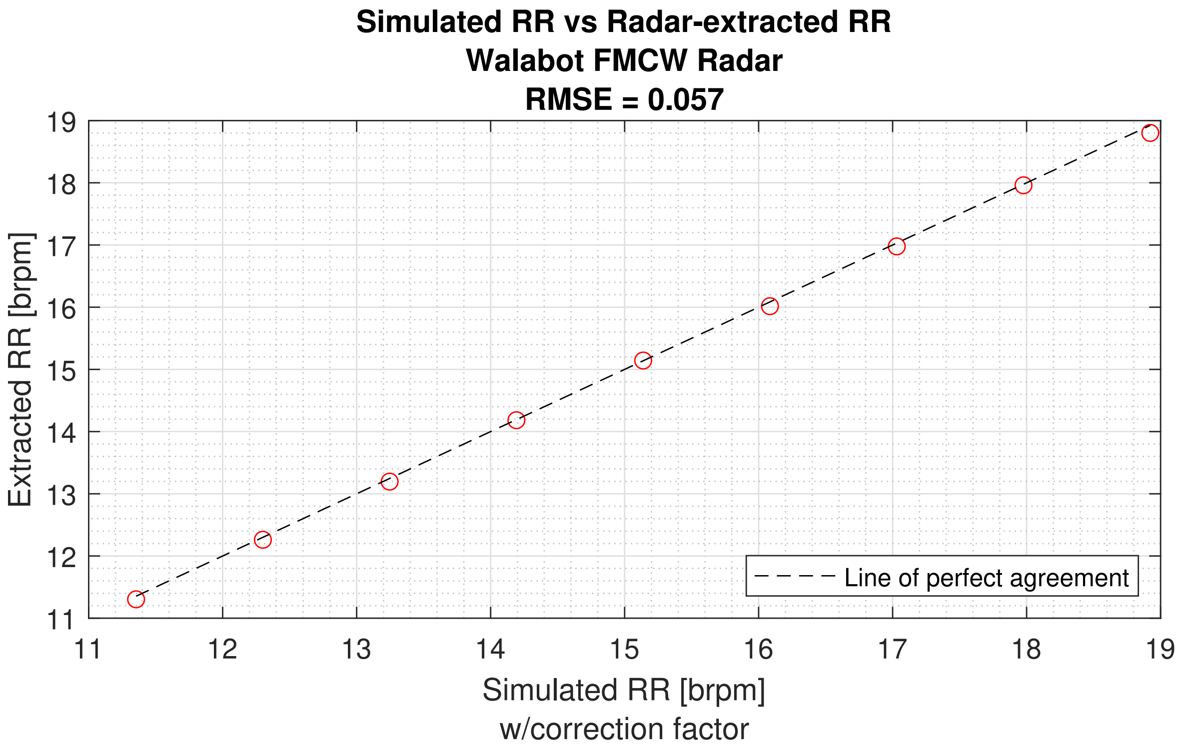


Figure 4-9: Comparison between chest simulator respiration rate (corrected) and Walabot FMCW radar measured respiration rate.

The repeatability of the RR extraction was tested by running 9 trials each of 13 brpm & 50 bpm, 16 brpm & 70 bpm, and 20 brpm & 90 bpm, for a total of 27 trials. The assumption is that people will tend to have higher resting heart rates with increased respiration. The results are shown in figure 4-10 with standard deviations of 0.085, 0.228, and 0.217 for the 13, 16, and 20 brpm trials, respectively.

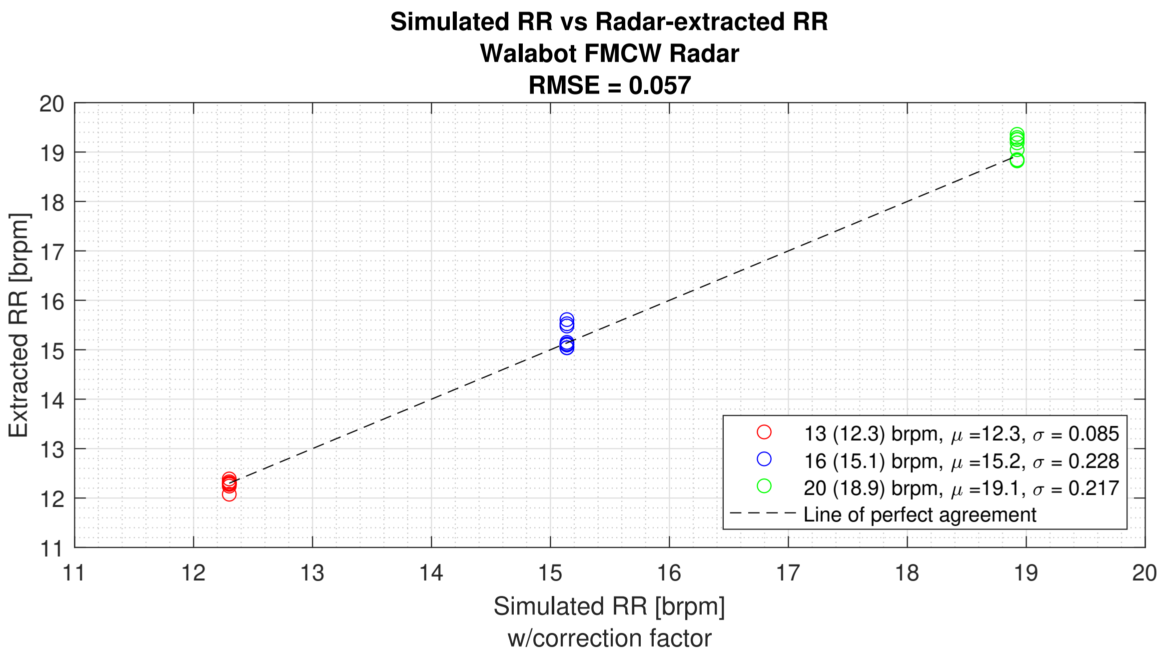


Figure 4-10: Repeatability comparison between chest simulator respiration rate (corrected) and Walabot FMCW radar measured respiration rate.

Heart rate subprocessing:

1. Apply a bandpass filter to the phase angle signal to isolate only the range of frequencies relevant to heart rate measurement: 40 bpm to 120 bpm.

2. Apply a Hamming window to the sequence of phase angles to reduce edge effects in Doppler FFT and pad the signal with trailing zeros.

3. Find peaks within the range of interest, select the most prominent one, and convert it into bpm.

When extracting the frequencies in the heart rate range, it is difficult to select the appropriate peak in the frequency domain. Figures 4-11 and 4-12 show results from the chest simulator running for 90 seconds with the heart rate signal only, with no possible harmonics from respiration to corrupt the signal. Six tests of varying amplitude and frequency are plotted against a stationary simulator, to visualize the effect of oscillation above background noise. Note that in this test, the correction factor has been applied to the chest simulator control loop to compensate for the per-loop delay of 0.56 ms. The left side plots of 4-11 resulted from setting the amplitude to 0.5 mm, while the right side is from 1.0 mm. The red line in the plot is data collected from a stationary plate over the same period of 90 seconds. With the higher 1.0 mm amplitude, the peak heights are noticeably higher than baseline, as opposed to the 0.5mm amplitude plots on the top and middle left side. Interestingly, a high heart rate of 90 bpm and an amplitude of 0.5 mm produces a frequency domain amplitude above baseline, but without an obvious correct peak.

Testing an even higher amplitude of 1.5mm in Figure 4-12 shows peaks that are higher than background noise, and located around the correct frequencies, especially in 50 and 90 bpm. Higher amplitudes were not explored due to the limitation of the chest simulator, which exhibits baseline drift of the plate at high amplitude and frequency. Furthermore, typical chest HR amplitudes would be closer to 0.5 mm in practice. These tests point to a fundamental SNR issue, which would require further effort to address, such as by increasing the measurement time to average out noise or by increasing the transmitted signal power (See also Section 4.2.2 Welch's Method).

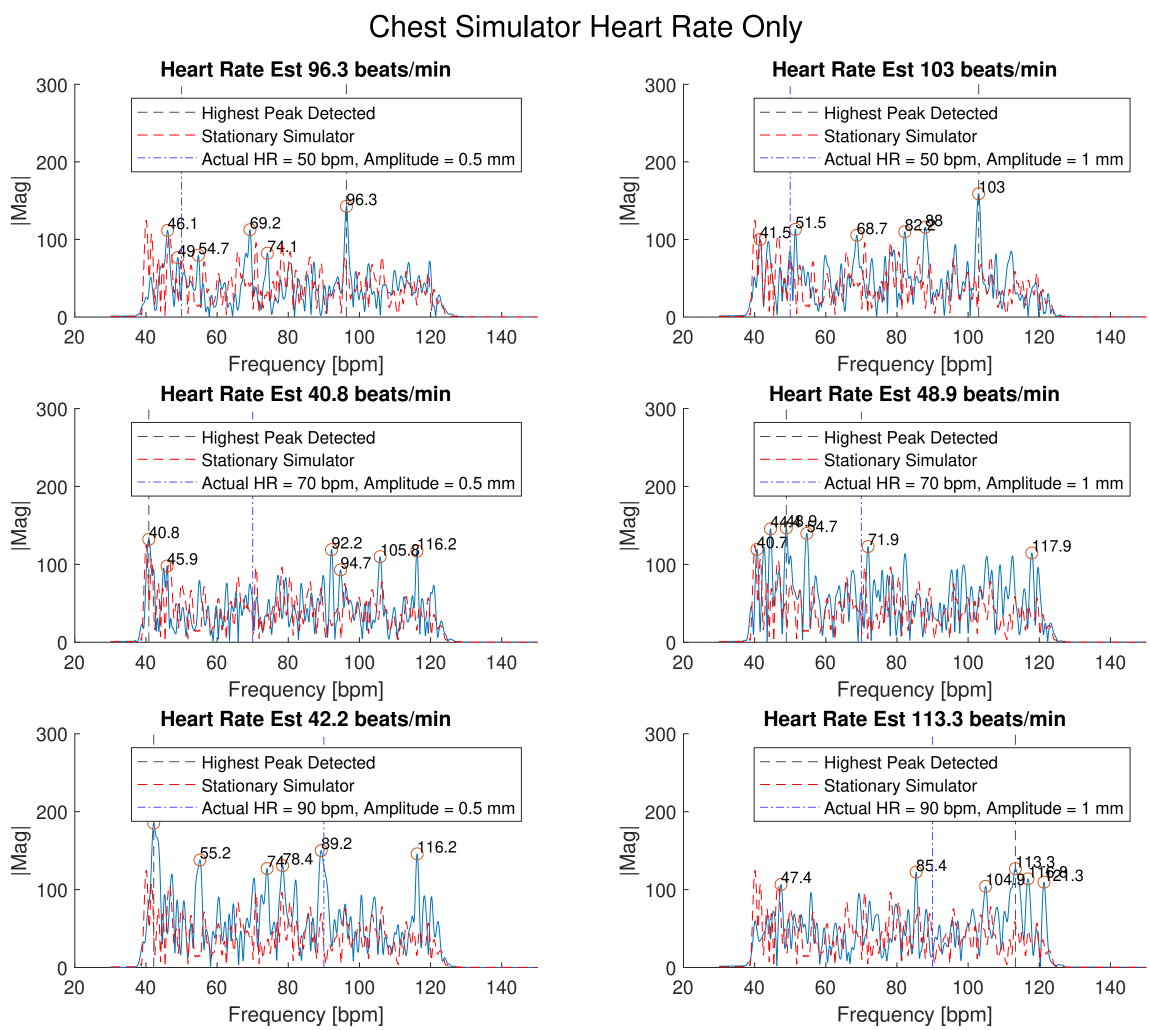


Figure 4-11: Comparison between chest simulator heart rate and Walabot FMCW radar measured heart rate.

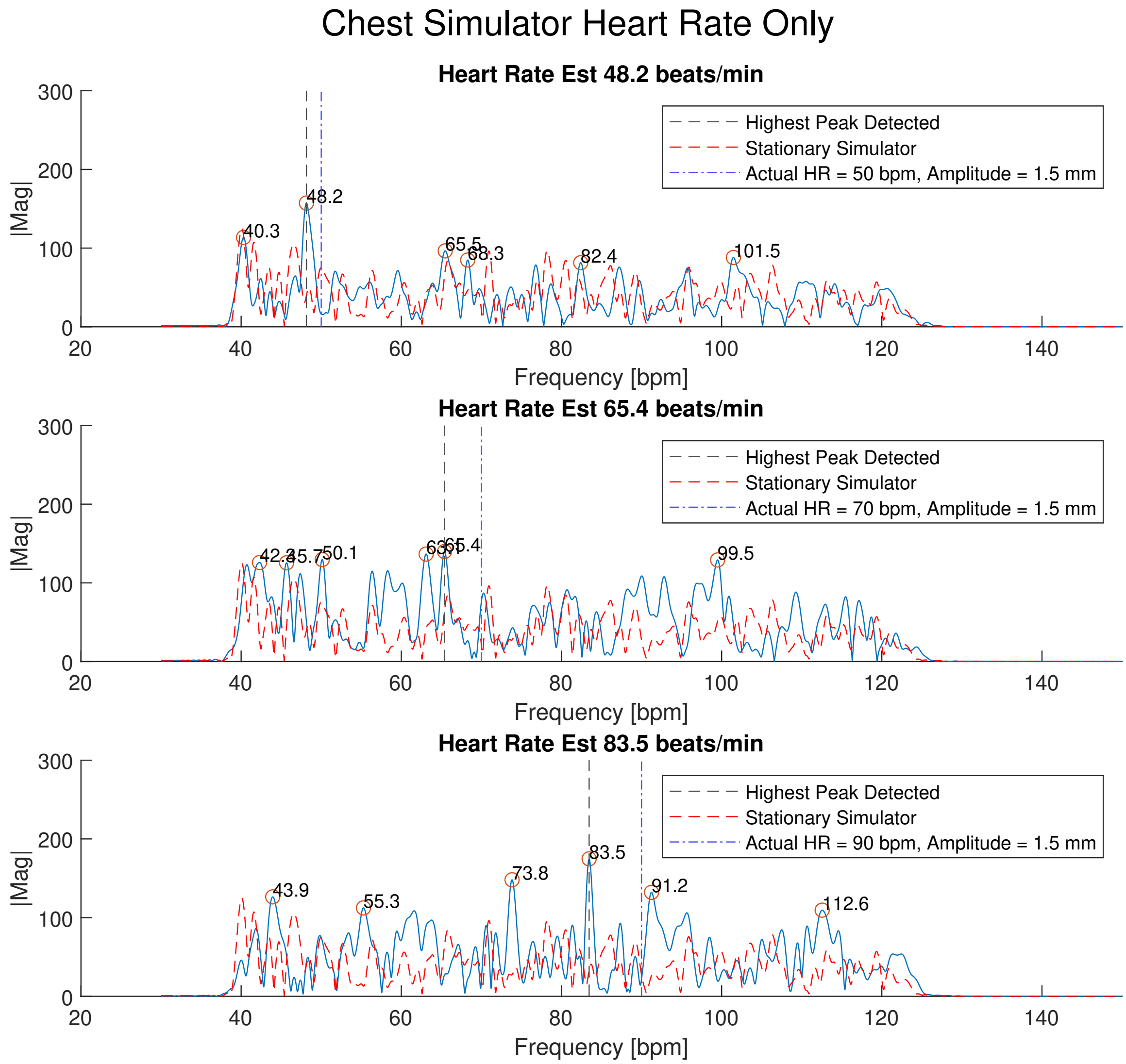


Figure 4-12: Comparison between chest simulator heart rate and Walabot FMCW radar measured heart rate.

We note that due to the operation and processing of the FMCW radar, the subject in the kiosk needs to remain as still as possible, in order to be in the same frequency bin in the Range FFT. If the subject moves too far toward or away from the radar, they will leave the frequency bin whose phase is tracked in the Doppler FFT, and impact the reading accuracy. Thus, a slightly different algorithm would need to be used for a higher frequency FMCW system like the Ti devices which have narrower range bins.

Slicing Phase Angle Data

Additionally, both processing chains for respiration and heart rate can be applied to broken up slices of phase angle data, with the slices including some percentage of overlap. This allows for the observation of evolution of respiration and heart rates over time, and adds robustness against signal irregularities like excessive body movements or interference. Figure 4-13 shows the slicing applied to the phase angle plot, and Figure 4-14 shows this method applied to one of the chest simulator trials, with a window of 20 seconds and an overlap of two-thirds, meaning signal irregularities under the duration of 7 seconds can be recognized and possibly eliminated when calculating the average result. Figure 4-15 shows errors between corrected simulator value and the average of values across the slices, while varying slicing parameters.

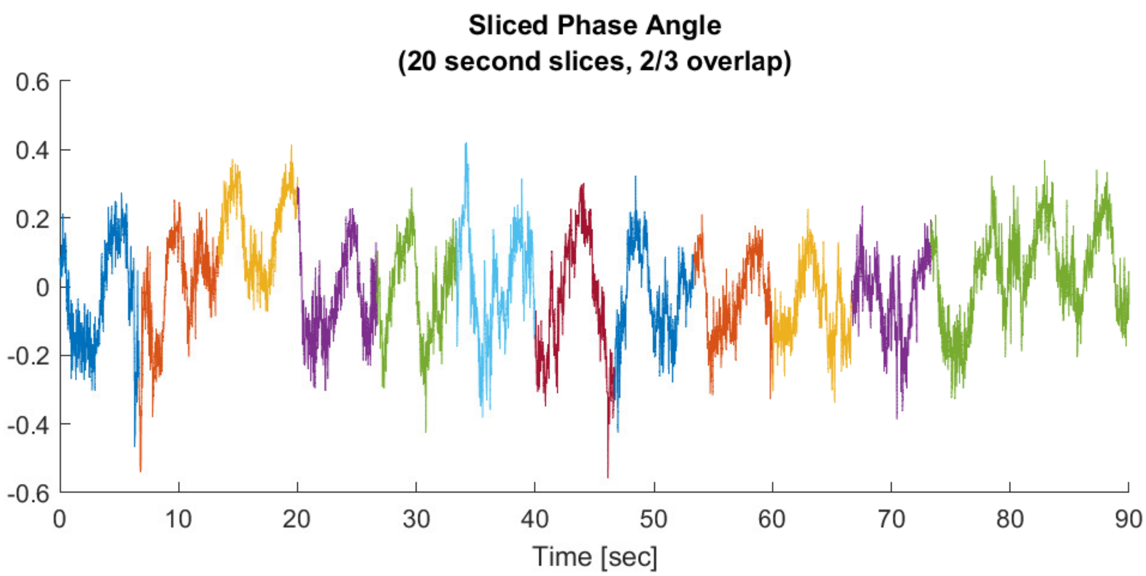


Figure 4-13: Phase angle over time with slices of 20 seconds and a 2/3 overlap.

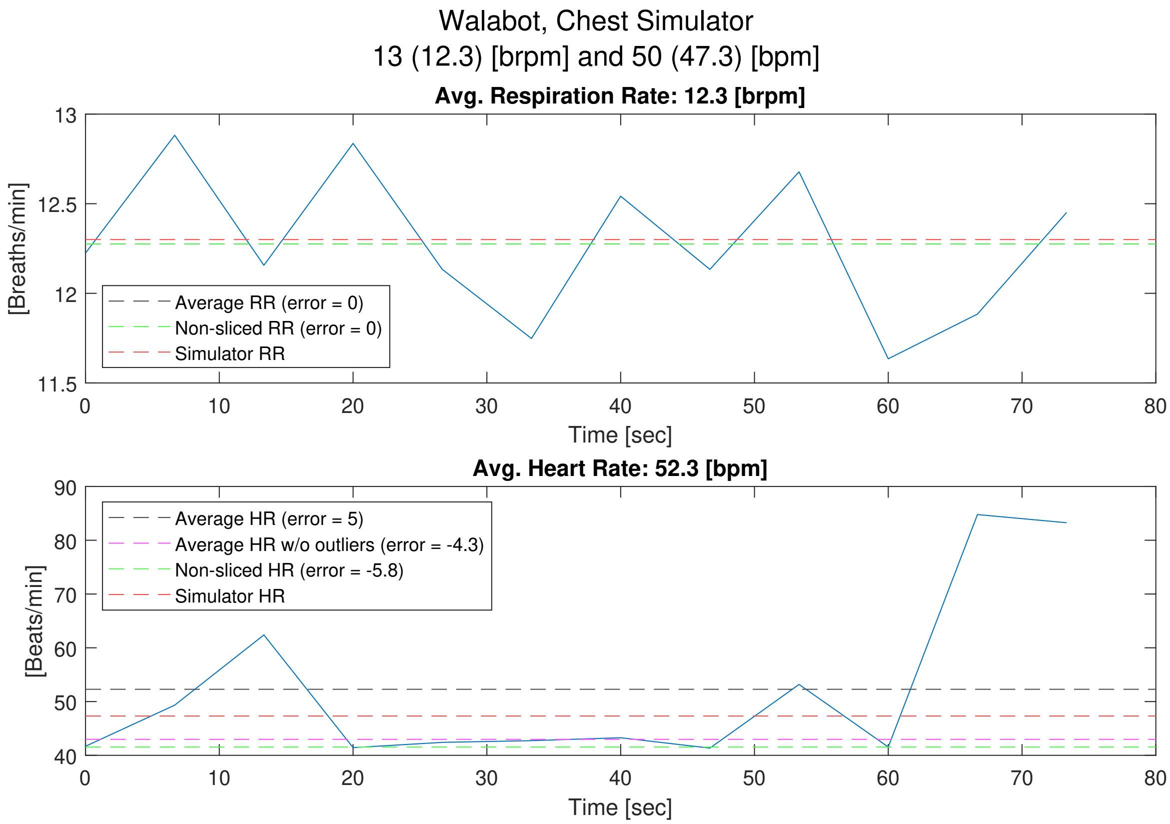


Figure 4-14: Respiration rate and heart rate over time with slices of 20 seconds and a 2/3 overlap.

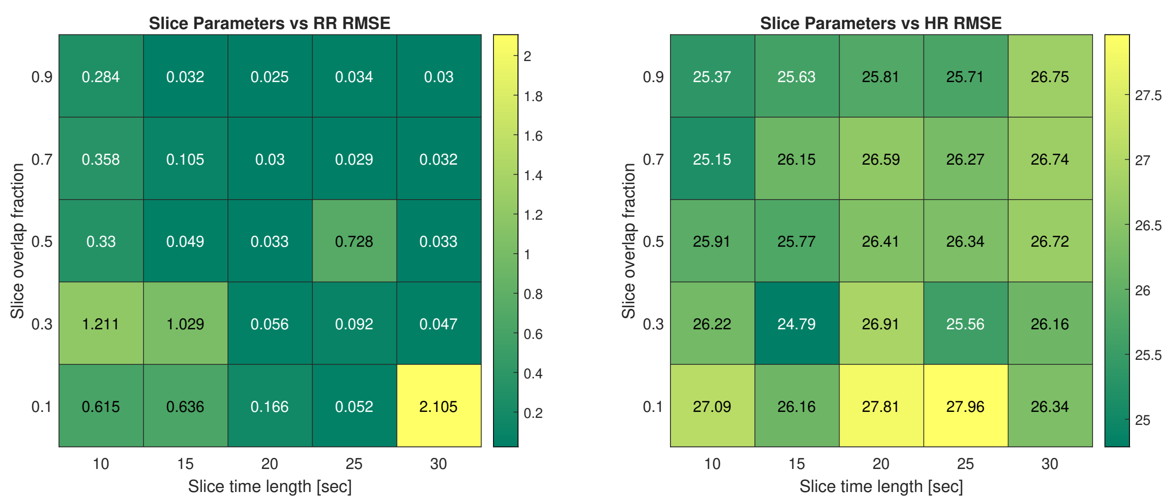


Figure 4-15: Comparing performance of slicing parameters.

Welch's Method

Another way to look at sliced data and possibly improve SNR is Welch's Method, which takes an average of the sliced phase angle data after it has been windowed and converted into the frequency domain. The test data from Figures 4-11 and 4-12 was processed using this method, with a 30 second window and 90% overlap to create the plots in Figure 4-16. Increasing window length and overlap tends to improve results, and peaks are prominent near the actual HR values in the 50 and 90 bpm plots. However, it is not so straightforward to identify the correct peak without knowing the general area to search, especially in the 70 bpm plot. Again, as expected, the higher 1.5 mm amplitude data shows the most prominent peaks around the correct locations.

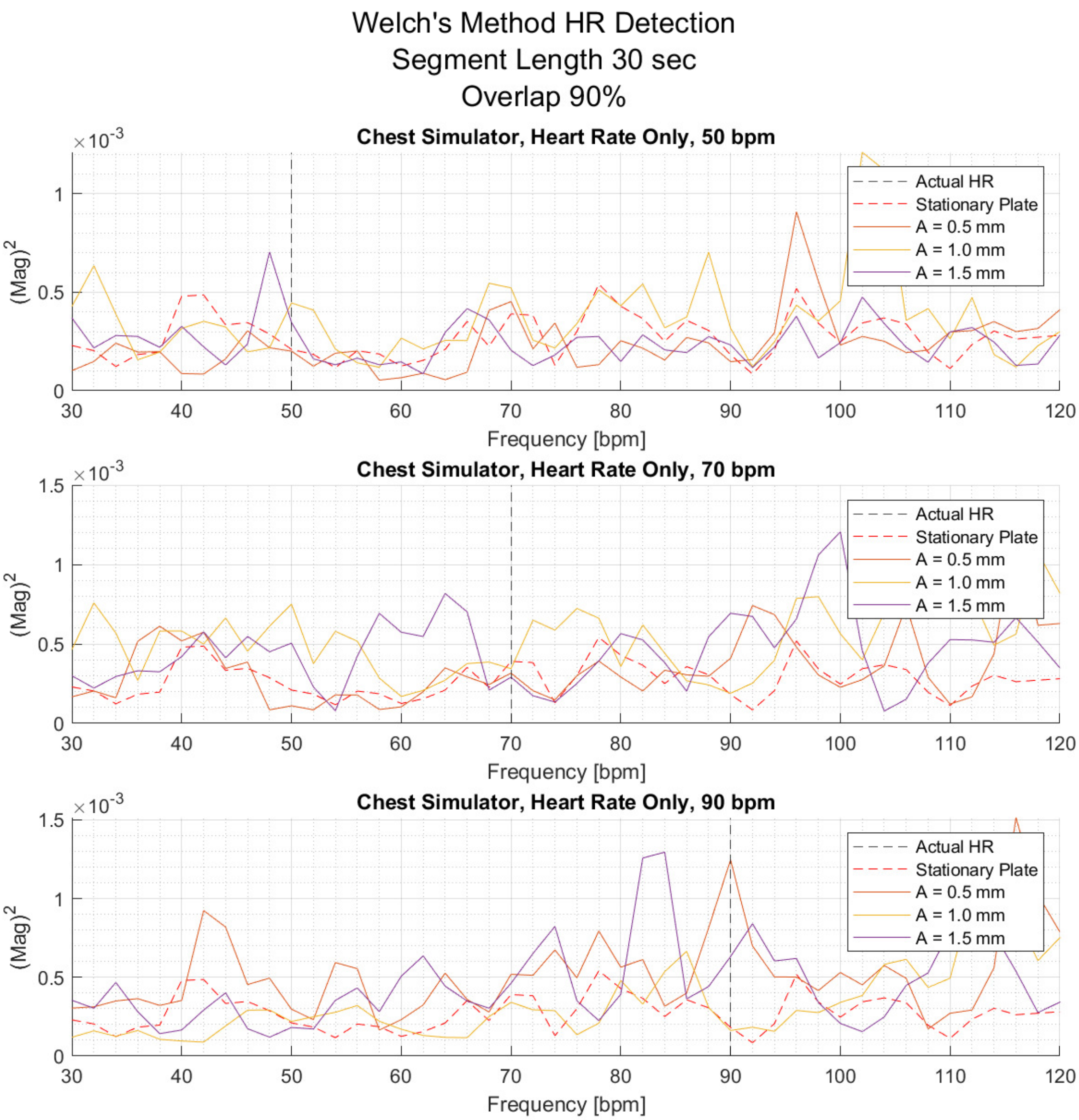


Figure 4-16: Using Welch's Method to detect HR peaks in Walabot FMCW radar data captured from the chest simulator.

4.2.3 Alternative Processing Note - Autocorrelation

As an alternative to using the FFT in Step 5 of the signal processing to extract respiration rate, it is also possible to use the autocorrelation of the phase angle signal, with similar results, which are shown in Figure 4-17. Over the course of 9 measurements the algorithms were timed 16 times (total of 144) using the `timeit()` function in MATLAB, showing average computation times of 115 ms and 16 ms for the FFT method and the autocorrelation method, respectively. This speed increase could be useful when processing large datasets.

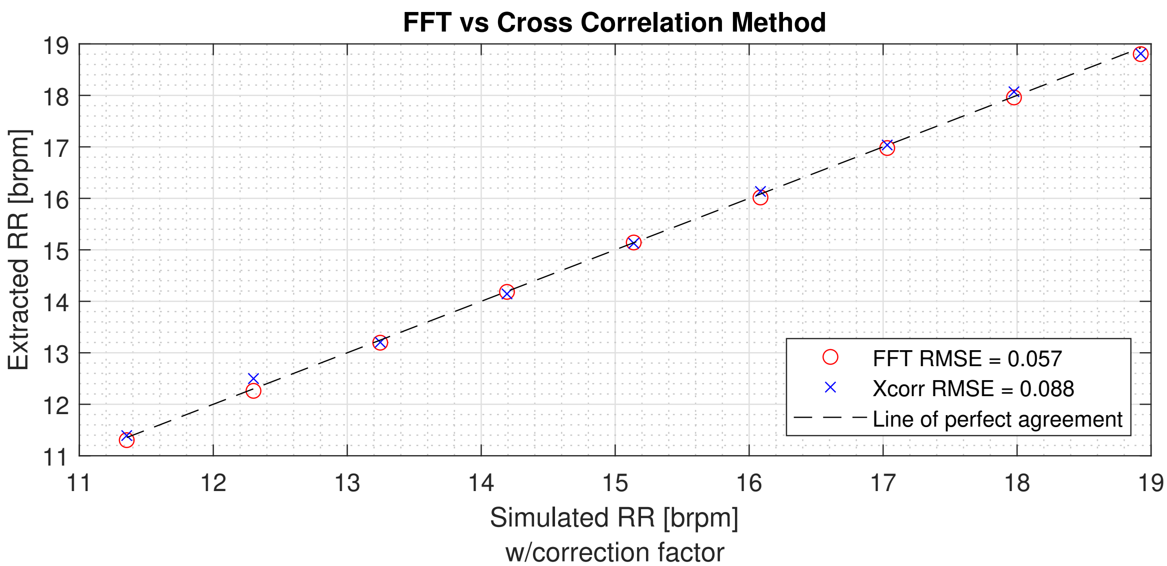


Figure 4-17: Comparison between using FFT and autocorrelation to calculate respiration rate using Walabot FMCW radar.

4.3 FMCW Radar - Ti mmWave on Raspberry Pi

Texas instruments (Dallas, TX, USA) produces several highly customizable FMCW radar systems, which operate on 60-64 GHz and 77-81 GHz chirp bandwidths. The include on-board digital signal processing (DSP) hardware that can allow for real-time processing of signals. Developer boards can be programmed directly to process signals and output specific information, or they can be paired with a separate capture card, Ti DCA1000, for raw signal capture via Ethernet connection to a PC. While it allows for customization in type of data collected, it is too big to integrate into the kiosk, as the complete assembly is too bulky and fragile. Alternatively, it would require significant effort to program the on-board DSP, which would then not allow for retroactive algorithm development. It is therefore preferred to collect raw signal data from test subjects, in order to be able to improve processing offline without having to recollect data.

As part of the provided developer tools, Ti has written an on-board processing routine for vital sign extraction, which attempts to measure both respiration and breathing rates. A separate company, Joybien (Zhonghe City, Taipei County, Taiwan), has taken advantage of this and created a radar board based on the Ti design as an add-on to a Raspberry PI linux computer [41]. This system is more compact than the Ti developer kits and has been added to one of the kiosks as a proof-of-concept. However, since it is based on the vital signs package created by Ti, it runs on the on-board DSP, and outputs only calculated parameters. Ti radar boards, capture card, and the Raspberry PI radar are pictured in Figure 4-18.

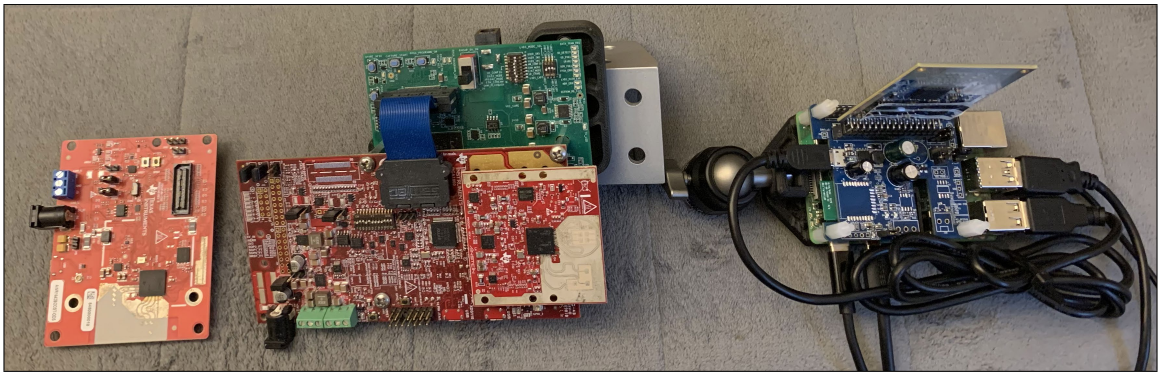


Figure 4-18: From left to right: Ti mmWave AWR1642 (77-81GHz) and IWR6843 (60-64 GHz) development boards, DCA1000 EVM Ethernet capture card for raw signals, Joybien Batman BM201-VSD (based on IWR6843).

The Joybien Raspberry Pi system was tested using the mechanical chest simulator, by running 6 trials: 3 with 13 brpm and 50 bpm, and 3 with 16 brpm and 70 bpm, with the results shown in Figure 4-19. The points were manually sampled from the output of the built-in algorithm. The algorithm displays live FFT plots while running, and we observed that it had difficulty choosing the correct peak, making the reported values oscillate. Using the average value over all the data points was yielded better results in the faster HR and RR test.

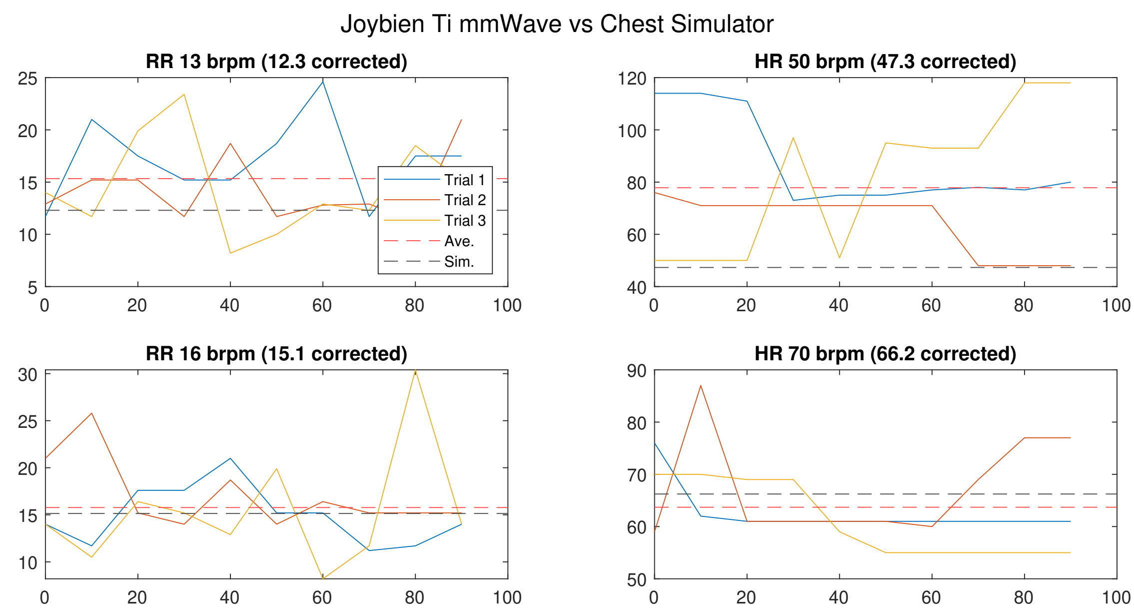


Figure 4-19: Results of testing Joybien Ti mmWave radar using the chest simulator.

Additionally, the Joybien system was testing using the author as a human subject, with results from 5 trials shown below in Figure 4-20. In trials 1 and 3, heart rate was not properly detected and the system defaulted to choosing an incorrect peak in the Doppler FFT.

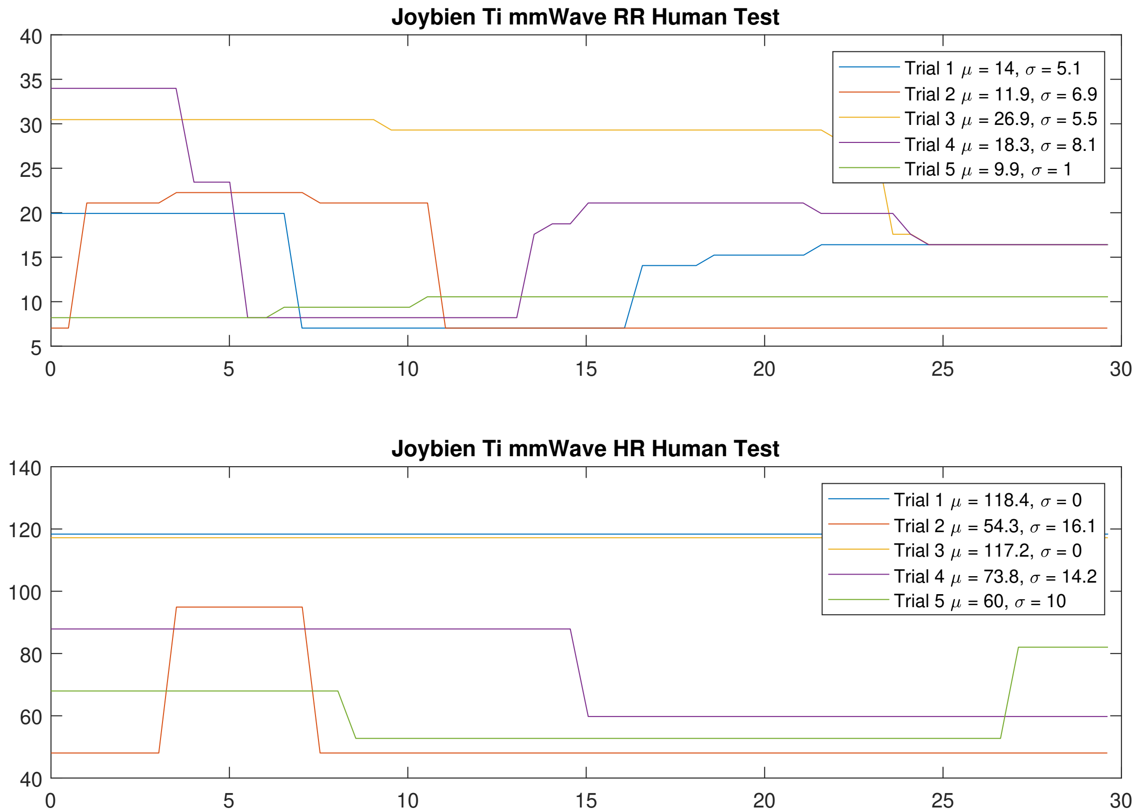


Figure 4-20: Results of testing Joybien Ti mmWave radar using human subject.

Working with these FMCW radar systems has presented many difficulties, and there is still a need to develop an accurate way to measure heart rate. With the research landscape in its present state, to date we know of no commercialized system able to extract accurate heart rates using radar, be it consumer or medical-grade. Currently, our kiosk system can track users' respiration rates well, and allows for further reprocessing and improvement using the raw data stored. It can also store heart rate data captured by the Joybien Raspberry Pi system, but that has poor accuracy with many outliers and does not allow for the collection of raw signal data. Based on data collected so far, continuing work should focus on techniques to improve the SNR of the heart rate signal.

Chapter 5

Conclusions & Future Work

This work presented a device and methodology for automatically collecting vital sign data from subjects in a non-contact way. The device can be easily moved to different locations and has flexible architecture to allow sensor and interface customization. Further work is needed to improve radar data processing - specifically heart rate extraction.

5.1 Applications

The kiosks can be be modified to run various automatic data collection routines, adapted for other research questions. For example, they can be used for broader radar testing - such as taking subjects through a movement routine to characterize common patterns like standing, sitting, and lying down. The kiosks can also be modified to add cameras, lighting, and background materials to allow for RGB camera-based oximetry development. Other types of sensors can be added, like motion tracking cameras and instrumented floor panels, to evaluate gait characteristics.

If further developed and refined, perhaps with the addition of blood pressure monitoring and a weight scale, systems like this could be used in doctors' offices to automate basic patient intake procedures.

5.2 Next Steps and Improvements

Software
If the kiosks are needed for more widespread deployment, it may make sense to improve the software packaging and distribution, such that it can be launched like an executable and updated remotely. Currently, this is done manually via a remote access program such as TeamViewer. File storage could also be improved and made more robust, such as by using Google cloud storage to store and retrieve large amounts of data, instead of the .csv based file system currently in use.

Hardware
Further exploration and algorithm development in the most challenging area of this project is needed - radar signal acquisition and processing. Improving radar-based measurement

accuracy relative to contact-based methods is necessary for more reliable and widespread deployment of non-contact systems.

User Interface

When designing the user interface, the scanning steps and instructions were arranged in a logical way, from the researcher's perspective. When testing with users who had never seen the system, we saw shortcomings in the way instructions were laid out and communicated. For example, users were not aware that their ID cards have a magnetic strip, an RFID chip, and a barcode, all of which can be used for authentication. Therefore, when the screen instructed them to "scan ID to start" they looked for an RFID reader, when instead they needed to turn around and use the barcode reader mounted behind them. This issue requires better signage to make the barcode reader visible, and better screen instructions. Another issue arose when a user did not enter the kiosk, but instead cautiously reached in and scanned their ID before walking in. This initiated the scan procedure, but because they were not yet in the kiosk, their height was not properly detected, and the camera stage was positioned too low. This required adjusting the camera logic to wait longer before making the adjustment.

Other issues were around timing and placement of instructions, such as during the radar scan. In the step preceding it, a few bullet points let the user know to stand still and breathe normally. However, during the "in process" screen there was only a simple countdown to let the user know how much time was remaining in the radar scan. As a result, one user commented that they held their breath the whole time because they weren't sure if they should breathe. This meant that we needed to keep the most critical instructions displayed the entire time, because it wasn't possible to rely on people to remember all the instructions on the first try. As the kiosks see more users, additional tweaks and refinements will need to be made to address inefficiencies in user interaction and confusion or misunderstandings.

Appendix A

Appendix

A.1 Chest Simulator Torque Requirement

Variable	**Value**	**Unit**	**Source**
Max Input Freq	200000	pulses/sec	
Linear stage lead P_B	10	mm/rev	
Controller Setting	400	pulses/rev	
Velocity	5000	mm/s	
$F_{\text{s,torque}}$	35.8		
$T_{\text{max,motor}}$	0.78	Nm	[40]
$a_{\text{max,hrcycle}}$	1867	mm/s^2	
$J_{\text{ballscrew}}$	0.00001488448589	$\text{kg} \cdot \text{m}^2$	$\frac{\pi}{32} \cdot \rho \cdot L_{\text{ballscrew}} \cdot D^4_{\text{ballscrew}}$
J_{load}	0.000003701136187	$\text{kg} \cdot \text{m}^2$	$m \cdot (\frac{P_B}{2\pi})^2$
J_{total}	0.00001858562207	$\text{kg} \cdot \text{m}^2$	$J_{\text{load}} + J_{\text{ballscrew}}$
α_{needed}	1173.07	rad/s^2	Cardiac/respiration waveforms 3-5
T_{needed}	0.022	Nm	$J_{\text{total}} \cdot \alpha$
ρ_{steel}	7900	kg/m^3	
$L_{\text{ballscrew}}$	0.4	m	
$D_{\text{ballscrew}}$	0.0148	m	
$m_{\text{total,Load}}$	1.46115	kg	
m_{cube}	0.405	kg	
$\rho_{\text{cube,Al}}$	2700	kg/m^3	
l_{cube}	0.075	m	
w_{cube}	0.05	m	
h_{cube}	0.04	m	
m_{tower}	0.70875	kg	
$\rho_{\text{tower,Al}}$	2700	kg/m^3	
l_{tower}	0.3	m	
w_{tower}	0.05	m	
h_{tower}	0.025	m	
m_{plate}	0.3474	kg	
ρ_{plate}	1930	kg/m^3	
l_{plate}	0.3	m	
w_{plate}	0.2	m	
h_{plate}	0.005	m	

Table A.1: Calculations of torque needed to generate required acceleration profile of linear stage.

A.2 Hardware Bill of Materials

SUMMARY COSTS	$5,880.79	Comments
STRUCTURE	$1,808.20	Structure, including monitor mount and power strip
FASTENER	$389.20	
EE HARDWARE	$3,656.80	PC, Tinkerforge, Radar, Cameras
MISC	$26.59	Tools, etc

	BOM		Dims [inches]														Blk Price	Wht Price			
Blk Subtotal	Item	Qty	W	L	H	A	8020 Description	Add-on / Machining	Parker PN	Parker Cutting	Parker Machining	C'Bores	Access	TYPE	Vendor	PN		Price	Surcharge	Machining	Subtotal
$234.80	Main Vertical Front	4	1.5	75.5	1.5	75.5	1502 LS BLACK	5/16-18 Hole ea End	10-1592	19-001	19-009 2x			STRUCTURE	8020	1502-LS-Black-FB	$0.70	$0.45	$1.95	$3.90	$159.30
$87.00	Main Depth	6	1.5	10.5	1.5	10.5	1503 LS BLACK	2 C'bores, both side 4	10-1593	19-001	19-039 2x			STRUCTURE	8020	1503-LS-Black-FB	$0.70	$0.45	$1.95	$5.20	$71.25
$236.40	Main Width	8	1.5	32	1.5	32.0	1503 LS BLACK	2 C'bores, both side 4	10-1593	19-001	19-039 2x			STRUCTURE	8020	1503-LS-Black-FB	$0.70	$0.45	$1.95	$5.20	$172.40
$62.10	Main Base Leg	2	1.5	36	1.5	36.0	1504 LS BLACK	2 access holes@ 12, 24 in	10-1594	19-001	19-011 2x			STRUCTURE	8020	1504-LS-Black-FB	$0.70	$0.45	$1.95	$3.90	$44.10
$86.18	Main Vertical Inside Mounting	3	1.5	38.25	1.5	38.3	1515 LS BLACK		10-1515	19-001				STRUCTURE	8020	1515-LS-Black-FB	$0.70	$0.45	$1.95		$57.49
$26.45	Rear Width Mounting	1	1.5	35	1.5	35.0	1515 LS BLACK		10-1515	19-001				STRUCTURE	8020	1515-LS-Black-FB	$0.70	$0.45	$1.95		$17.70
$136.00	Link Top	2	1.5	86	1.5	86.0	1501 LS BLACK	3 access holes, @ 0.75, 12.75, 85.25 in	10-1591	19-001	19-011 3x			STRUCTURE	8020	1501-LS-Black-FB	$0.70	$0.45	$1.95	$5.85	$93.00
$124.93	Rear Vertical	2	1.5	80.875	1.5	80.9	1502 LS BLACK	5/16-18 Hole ea End	10-1592	19-001	19-009 2x			STRUCTURE	8020	1502-LS-Black-FB	$0.70	$0.45	$1.95	$3.90	$84.49
$11.56	Main Panel Front Instrument	1	32.75	36	0.25	8.2	CLEAR PC 2609	4 notches for hinges	26-790-6P		see CAD			STRUCTURE	8020	2609		$8.30	$10.50	$1.06	$79.52
$173.97	Main Panel Front Bottom	4	32.75	36	0.25	8.2	HDPE 2655 2651 blk	2 notches for Anchors	26-410-6P		see CAD			STRUCTURE	8020	2655	$3.90	$5.10	$10.50	$1.06	$213.27
$67.59	Main Panel Side	3	11.25	36	0.25	2.8	HDPE 2655 2651 blk	2 notches for Anchors	26-410-6P		see CAD			STRUCTURE	8020	2655	$3.90	$5.10	$10.50	$1.06	$77.71
$67.80	Main Panel Top	3	11.25	32.75	0.25	2.6	HDPE 2655 2651 blk	4X 7505	26-410-6P		see CAD			STRUCTURE	8020	2655	$3.90	$5.10	$10.50	$2.12	$77.01
$87.36	Door Panel Plain	2	30	38.25	0.25	8.0	HDPE 2655 2651 blk	7179 3" rounded corners	26-410-6P		see CAD			STRUCTURE	8020	2655	$3.90	$5.10	$10.50	$2.10	$106.48
$15.49	Sliding Panel Stationary	1	10.375	17.75	0.25	1.3	HDPE 2655 2651 blk		26-410-6P		see CAD			STRUCTURE	8020	2655	$3.90	$5.10	$10.50		$17.02
$17.35	Sliding Panel Moving	1	11.25	19	0.25	1.5	HDPE 2655 2651 blk	2 notches for Anchors + 2 7598 on top	26-410-6P		see CAD			STRUCTURE	8020	2655	$3.90	$5.10	$10.50	$1.06	$19.13
$17.25	Tslot strip 72.5"	5				1.0	2110							FASTENER	8020	2110	$3.45				$0.00
$74.00	T-Nut 5/16 Roll In	37				1.0	13040							FASTENER	8020	13040	$2.00	$2.00			$74.00
$17.30	T-Nut M5 Roll In	10				1.0	13028							FASTENER	8020	13028	$1.73	$1.73			$17.30
$5.19	T-Nut M4 Roll In	3				1.0	13027							FASTENER	8020	13027	$1.73	$1.73			$5.19
$6.92	T-Nut 1/4-20 Roll In	4				1.0	13037							FASTENER	8020	13037	$1.73	$1.73			
$7.92	Pedal Plate	1				1.0	4365-Black							STRUCTURE	8020	4365-Black	$7.92	$7.92			$7.92
$14.18	Slide Door Mount	2				1.0	2425-Black							FASTENER	8020	2425	$7.09				$0.00
$17.30	Corner Gusset 4336	2				1.0	4336 Black	with fasteners 4x 3320						STRUCTURE	8020	4336 Black	$6.25	$6.25	$2.40		$17.30
$24.80	Plastic Endcap	16				1.0	2030		18-1515					STRUCTURE	8020	2030	$1.55	$1.55			$24.80
$40.96	Single L-Bracket	8				1.0	4302 BLACK	add 2X 3330 bolt only						FASTENER	8020	4302-Black	$3.86	$2.95	$1.26		$33.68
$48.00	Plain Hinge	4				1.0	12084 BLACK	add 2X 13055						FASTENER	8020	12084	$10.00	$10.00	$2.00		$48.00
$10.40	End Fastener	8				1.0	3389 BLACK							FASTENER	8020	3389	$1.30	$1.30			$10.40
$90.91	Monitor Mount	1				1.0	2285							STRUCTURE	8020	2285	$90.91	$90.91			$90.91
$103.50	Anchor Fastener	30				1.0	3359 BLACK		25-004					FASTENER	8020	3359, 3360 (wt)	$3.45	$3.15			$94.50
$4.20	Padlock Plate	1				1.0	Custom Steel Sheetmtl	EST COST 100X						MISC	McMaster		$4.20	$4.20			$4.20
$25.80	Leveling Foot	2				1.0	2190							FASTENER	8020	2190	$12.90	$12.90			$25.80
$11.97	Padlock	1				1.0								MISC	Amazon		$11.97	$11.97			$11.97
$10.42	Slide Door Handle	1				1.0								MISC	McMaster	19950A5	$10.42	$10.42			
$19.88	Door Magnets	2				1.0								FASTENER	McMaster	5537T56	$9.94	$9.94			
$232.32	Caster	4				1.0								STRUCTURE	McMaster		$58.08	$58.08			$232.32
$0.18	Caster Nut replacement	0.04				1.0								FASTENER	McMaster		$4.39	$4.39			$0.18
$5.64	Screws 5/16	12				1.0	3340							FASTENER	8020		$0.47	$0.47			$5.64
$34.36	Master Brick	1				1.0								EE HARDWARE	Mouser		$34.36	$34.36			$34.36
$19.47	Humidity Bricklet	1				1.0								EE HARDWARE	Mouser		$19.47	$19.47			$19.47
$57.28	Stepper Bricklet	1				1.0								EE HARDWARE	Mouser		$57.28	$57.28			$57.28
$41.10	Servo Bricklet	1				1.0								EE HARDWARE	Tinkerforge		$41.10	$41.10			$41.10
$22.90	Distance IR Bricklet	1				1.0								EE HARDWARE	Mouser		$22.90	$22.90			$22.90
$16.03	Motion Bricklet	1				1.0								EE HARDWARE	Mouser		$16.03	$16.03			$16.03
$16.66	Servo Motor	1				1.0								EE HARDWARE	Amazon		$16.66	$16.66			$16.66
$195.00	Linear Stage	1				1.0								EE HARDWARE	Amazon		$195.00	$195.00			$195.00
$1,995.00	Seek Scan Camera	1				1.0								EE HARDWARE	Tequipment		$1,995.00	$1,995.00			$1,995.00
$590.00	Lenovo Mini PC	1				1.0								EE HARDWARE	Lenovo		$590.00	$590.00			$590.00
$220.00	Lenovo Monitor	1				1.0								EE HARDWARE	Lenovo		$220.00	$220.00			$220.00
$449.00	Walabot Developer	1				1.0								EE HARDWARE	Walabot		$449.00	$449.00			$449.00

Table A.2: Bill of Materials for kiosk.

A.3 GUI Software – Python

A.3.1 Main Program

Listing A.1: POD_main.py

```
# -*- coding: utf-8 -*-
"""
Created on Wed Dec 30 17:45:04 2020

MAIN PROGRAM FILE FOR RUNNING POD SOFTWARE


REQUIRED PIP INSTALL MODULES:
        - pip install opencv-python
        - pip install keyboard
        - pip install tinkerforge
        - pip install pyautogui
        - pip install smbus2
        - pip install vl53l1x

@author: IGory
"""
# %% IMPORTS

# POD MODULES
import POD_seekscan as thermal
import POD_store_local as file
import POD_encrypt as encrypt
import POD_RPi_link as rpi
from POD_tinkerforge import Tinkerforge_System
from POD_Configuration import Pod_Configs

#import POD_radar_ti as ti
import POD_radar_walabot as wb

# TODO!!!
# import POD_ui as ui
# import POD_store_googlecloud as cloud
# import POD_events as podfunc


# Other Modules
import os, time, threading, keyboard
import statistics as stats
import tkinter as tk
from datetime import date, datetime
from PIL import ImageTk, Image
import pyautogui
import concurrent.futures

# can be deleted when done
import subprocess, sys, fnmatch, math
import numpy as np
from tkinter.font import Font
from tkinter import filedialog

import matplotlib.pyplot as plt
import matplotlib.image as mpimg


# %% GLOBALS AND SETUPS

# debug_mode = True
# laptop_screen_mode = False

# save_image = False
# units = "C"                 # C of F
# temp_threshold = 38         # deg C, 100.4 deg F
# temp_low_threshold = 36.6   # deg C, if below this value, try 2 more times
# body_temp_offset = 1.2      # deg C, default in camera
# alert_sounds = 0            # turn off sounds
# today = date.today().strftime("%Y-%m-%d")

# num_temp_scans = 3          # 3 Thermal Camera Scans
# radar_duration = 5         # sec
# radar_countdown = 5         # sec

# %% TKINTER MAIN

class ThreadWithReturnValue(threading.Thread):
    def __init__(self, group=None, target=None, name=None,
                 args=(), kwargs={}, Verbose=None):
        threading.Thread.__init__(self, group, target, name, args, kwargs)
```

```
        self._return = None
    def run(self):
        print(type(self._target))
        if self._target is not None:
            self._return = self._target(*self._args,**self._kwargs)
    def join(self, *args):
        threading.Thread.join(self, *args)
        return self._return

class BgFrame(tk.Frame):
    def __init__(self, parent, file_path, width, height, bg_color):
        super(BgFrame, self).__init__(parent, borderwidth=0, highlightthickness=0)

        self.canvas = tk.Canvas(self, width=width, height=height, bg=bg_color)
        self.canvas.pack()

        pil_img = Image.open(file_path)
        self.img = ImageTk.PhotoImage(pil_img)  # .resize((width, height), Image.ANTIALIAS)
        self.bg = self.canvas.create_image(0, 0, anchor=tk.CENTER, image=self.img)

    def add(self, widget, x, y):
        canvas_window = self.canvas.create_window(x, y, anchor=tk.NW, window=widget)
        return widget


class Banner:
    def __init__(self, master, color, path_images):
        self.master = master
        self.path_images = path_images
        self.img_banner = ImageTk.PhotoImage(Image.open(
            self.path_images + "\\UI_Banner.png"))  # must be like this ... https://stackoverflow.com/questions/16424091/why-does-tkinter-image-not-show-up-if-created-in-a-function
        self.label = tk.Label(self.master, image=self.img_banner, compound="center", bg=color["bg"])
        self.label.pack(fill="x", pady=0)
        if pod.laptop_screen_mode:
            self.label.pack_forget()
        # self.label.grid(column=0,row=0,pady=0)


class Icons:
    def __init__(self, master, color, path_images):
        self.master = master
        self.path_images = path_images

        # LOAD IMAGES
        self.img_icons_off = ImageTk.PhotoImage(Image.open(self.path_images + "\\UI_Icons_idle.png"))
        self.img_icons_step1 = ImageTk.PhotoImage(Image.open(self.path_images + "\\UI_Icons_step1.png"))
        self.img_icons_step2 = ImageTk.PhotoImage(Image.open(self.path_images + "\\UI_Icons_step2.png"))
        self.img_icons_step2_error = ImageTk.PhotoImage(Image.open(self.path_images + "\\UI_Icons_step2_error.png"))
        self.img_icons_step3 = ImageTk.PhotoImage(Image.open(self.path_images + "\\UI_Icons_step3.png"))
        self.img_icons_step4 = ImageTk.PhotoImage(Image.open(self.path_images + "\\UI_Icons_all_done.png"))

        # CREATE INITIAL LABEL
        self.label = tk.Label(self.master, image=self.img_icons_off, bg=color["bg"])
        self.label.pack(fill="x", pady=10)
        # self.label.grid(column=0,row=1,pady=10)

        # SET INITIAL STATE
        self.step1()

    def off(self):
        self.label.configure(image=self.img_icons_off)
        self.label.update()

    def step1(self):
        self.label.configure(image=self.img_icons_step1)
        self.label.update()

    def step2(self):
        self.label.configure(image=self.img_icons_step2)
        self.label.update()

    def step2_error(self):
        self.label.configure(image=self.img_icons_step2_error)
        self.label.update()

    def step3(self):
        self.label.configure(image=self.img_icons_step3)
        self.label.update()

    def step4(self):
        self.label.configure(image=self.img_icons_step4)
        self.label.update()

    def error(self):
        self.off()


class Status:
    def __init__(self, master, color, path_images, font):
        self.master = master
        self.path = path_images
```

```
        self.font = font
        self.color = color
        self.pedal_skip = False

        # DEFINE SOME FONT COLORS
        self.color_font_status = "#F2F2F2"
        self.color_font_status_dark = "#414042"
        self.color_font_status_error = "#231F20"
        self.color_font_status_setup = "#333333"

        # STATUS DESCRIPTIONS
        self.status_start_title = "Scanning Procedure Overview:"
        self.status_start_instructions = "Scanning Procedure Overview:\n\n\
        (1) Scan ID & stand on 'T' floor mark as shown, \nunder height sensor                            \n\
        (2) Clear forehead & remove glasses for             \ntemperature scan                                    \n\
        (3) Stand still with arms by your side for             \nheart rate scan ...
                                            \n"#\n\
        #-- Scan ID to begin --        "
        self.status_temp = "Body Temperature\n"  # Infrared Scan
        self.status_temp_guide = "Temperature Scan: \n\n - Expose Forehead\n- Stand Still"
        self.status_radar = "Heart Rate Scan\n ( Hold Still"  # Heart and Breathing Rate\nRF Scan"
        self.status_radar_guide = "Heart Rate Scan: \n\n - Arms by your side\n- Stand Still"
        self.status_working = "Processing..."
        self.status_done = "Scanning Complete\nPlease Exit\nThank You!"
        self.status_idle = "- Idle -\nPress Pedal to\nBegin Operation"
        self.status_setup_id = "Setup:\n\nSelect Station ID"
        self.status_setup_height = "Setup:\n\nEnter camera height"
        self.status_setup_temp_threshold = "Setup: \n\nEnter Temp Threshold"
        self.status_setup_temp_offset = "Setup: \n\nEnter Temp Offset"
        self.status_user_input_temp = "Press pedal\nTo Start\nTemperature Scan"
        self.status_user_input_radar = "Press pedal\nTo Start\n HR & RR Scan"

        self.status_error_temp = "Error in temperature scan"

        # LOAD IMAGES
        self.img_status_start = ImageTk.PhotoImage(Image.open(
            self.path + "\\UI_Status_start.png"))  # must be like this ...
                https://stackoverflow.com/questions/16424091/why-does-tkinter-image-not-show-up-if-created-in-a-function
        self.img_status_working = ImageTk.PhotoImage(Image.open(self.path + "\\UI_Status_working.png"))
        self.img_status_temp = ImageTk.PhotoImage(Image.open(self.path + "\\UI_Status_temp.png"))
        self.img_status_radar = ImageTk.PhotoImage(Image.open(self.path + "\\UI_Status_radar.png"))
        self.img_status_done = ImageTk.PhotoImage(Image.open(self.path + "\\UI_Status_done.png"))
        self.img_status_setup = ImageTk.PhotoImage(Image.open(self.path + "\\UI_Status_setup.png"))
        self.img_status_error = ImageTk.PhotoImage(Image.open(self.path + "\\UI_Status_error.png"))
        self.img_status_idle = ImageTk.PhotoImage(Image.open(self.path + "\\UI_Settings.png"))

        ## CANVAS APPROACH
        # CREATE CANVAS OUTLINE
        # self.canvas = ...
              tk.Canvas(self.master,width=1080,height=480,bg=self.color['bg'],borderwidth=0,highlightthickness=0)
        # self.canvas.pack(fill="both",expand=True)
        # self.status_bg = self.canvas.create_image(540,240,image=self.img_status_start, anchor="center")

        # # CREATE LABELS
        # self.label_top = tk.Label(self.master, text = self.status_start_title, font = ...
              self.font['instructions_title'], bg=color["bg"], fg = self.color_font_status,compound="center")
        # self.label_mid = tk.Label(self.master, text = self.status_start_instructions, anchor=tk.E, font = ...
              self.font['instructions_text'], bg=color["bg"], fg = self.color_font_status, image = ...
              self.img_status_start,compound="center")
        # self.label_bot = tk.Label(self.master, text = self.status_start_instructions, anchor=tk.E, font = ...
              self.font['instructions_text'], bg=color["bg"], fg = self.color_font_status, image = ...
              self.img_status_start,compound="center")

        # # DISPLAY LABELS
        # self.label_top_canvas = ...
              self.canvas.create_text(100,100,text=self.status_start_title,font=self.font['instructions_title'],anchor="w")

        # self.frame = tk.Frame(self.master,bg=self.color["gray"])
        # self.frame.pack(fill="x",expand=True)

        # CREATE INITIAL LABEL
        self.label = tk.Label(self.master, text=self.status_start_instructions, font=self.font['instructions_text'],
                              bg=color["bg"], fg=self.color_font_status, image=self.img_status_start, ...
                                    compound="center")
        self.label.pack(expand=True)
        # self.idle()

    def idle(self):
        message = "\nUser not detected... \n\nStand on floor mark to activate.\n\n"
        self.label.configure(image=self.img_status_idle, text=message, font=self.font['medium'],
                             fg=self.color_font_status)
        self.label.update()

    def start(self):
        self.label.configure(image=self.img_status_start, text=self.status_start_instructions,
                             font=self.font['instructions_text'], fg=self.color_font_status)
        self.label.update()

    def temp_instructions(self):
        message = "Temperature Scan\n\n\
         - Stand still on floor mark                            \n\
         - Clear forehead and remove glasses         \n\
```

```python
            until screen changes color                      \n"
        self.label.configure(image=self.img_status_temp, text=message, font=self.font['medium'],
                             fg=self.color_font_status_dark)
        self.label.update()

    def temp_instructions_countdown(self, duration):

        for second in range(duration + 1):
            # message = "- - - Temperature Scan - - -\n\nStand still on floor mark.\n\nPedal to skip countdown: " + str(
            #     duration - second)
            message = "Temperature Scan\n\n\
                     - Stand still on floor mark                                         \n\
                     - Clear forehead and remove glasses                      \n\
                     until screen changes color                                   \n\
                     Pedal to skip countdown: " + str(duration - second) + "                    "
            self.label.configure(image=self.img_status_temp, fg=self.color_font_status_dark, font=self.font['medium'],
                                 text=message)
            self.label.update()
            time.sleep(1)
            if self.pedal_skip:
                print("Saw pedal press!")
                self.pedal_skip = False  # RESET PEDAL WATCHER
                break

    def temp_scan(self, scan_number, scans):
        message = self.status_temp + "\n" + "Scan " + str(scan_number) + " of " + str(scans)
        self.label.configure(image=self.img_status_temp, fg=self.color_font_status_dark, font=self.font['status'],
                             text=message)
        self.label.update()

#- Stand still on floor mark                           \n\
#- Move to X floor mark                           \n\


    def radar_instructions(self):
        message = "Prepare for Heart Rate Scan\n\n\
        - Move to X floor mark                            \n\
        - Arms by your side                                  \n\
        - Breathe normally                                    \n"
        self.label.configure(image=self.img_status_radar, text=message, font=self.font['medium'],
                             fg=self.color_font_status_dark)
        self.label.update()

    def radar_instructions_countdown(self, duration):

        for second in range(duration + 1):
            # message = "Prepare for Heart Rate Scan\n\nStand still on floor mark.\nBreathe normally.\n\nPedal to skip countdown: " + str(
            #     duration - second)
            message = "Prepare for Heart Rate Scan\n\n\
                    - Move to X floor mark                                            \n\
                    - Arms by your side                                                 \n\
                    - Breathe normally                                                  \n\
                    Pedal to skip countdown: " + str(duration - second) + "                    "
            self.label.configure(image=self.img_status_radar, fg=self.color_font_status_dark, font=self.font['medium'],
                                 text=message)
            self.label.update()
            time.sleep(1)
            if self.pedal_skip:
                print("Saw pedal press!")
                self.pedal_skip = False  # RESET PEDAL WATCHER
                break

    def radar_countdown(self, duration):
        for count in range(duration + 1):
            message = self.status_radar_guide + "\n\nStarting in " + str(duration - count)
            self.label.configure(image=self.img_status_radar, fg=self.color_font_status_dark, font=self.font['medium'],
                                 text=message)
            self.label.update()
            time.sleep(1)

    def radar_scan(self, duration):
        for second in range(duration + 1):
            if second == 0:  # ADD SLIGHT DELAY TO ALLOW USER TO BECOME STILL
                message = "Heart Rate Scan\n\nHold Still\nBreathe Normally\n"
                self.label.configure(image=self.img_status_radar, fg=self.color_font_status_dark, font=self.font['status'],
                                     text=message)
                self.label.update()
                time.sleep(2)
            message = "Heart Rate Scan\n\nHold Still\nBreathe Normally\n" + str(duration - second)
            self.label.configure(image=self.img_status_radar, fg=self.color_font_status_dark, font=self.font['status'],
                                 text=message)
            self.label.update()
            time.sleep(1)

        # NOTIFY OF COMPLETED SCAN
        message = "Heart Rate Scan Complete"
        self.label.configure(image=self.img_status_radar, fg=self.color_font_status_dark, font=self.font['status'],
                             text=message)
        self.label.update()
```

```
        time.sleep(4)

    def radar_results_text(self,t_vec,hr_vec,rr_vec):
        HR = int(stats.mean(hr_vec))
        RR = int(stats.mean(rr_vec))
        message = "Estimated results:\n\n\
        Heart rate: "+str(HR)+" beats/min           \n\
        Breathing rate: "+str(RR)+" breaths/min                \n"
                   #(Experimental results)                                       \n"
        self.label.configure(image=self.img_status_radar, text=message, font=self.font['medium'],
                             fg=self.color_font_status_dark)
        self.label.update()

    def radar_results_graph(self,t_vec,hr_vec,rr_vec):
        pass

    def radar_calibration(self):
        message = "Calibrating...\n\nPlease do not enter kiosk."
        self.label.configure(image=self.img_status_error, fg=self.color["red"], font=self.font['status'], text=message)
        self.label.update()

    def done(self):
        self.label.configure(image=self.img_status_done, fg=self.color["light_green"], font=self.font['status'],
                             text=self.status_done)
        self.label.update()

    def working(self):
        self.label.configure(image=self.img_status_working, fg=self.color_font_status, font=self.font['status'],
                             text=self.status_working)
        self.label.update()
        # self.canvas.itemconfigure(self.label_top_canvas, args=(100,200), text="testing!!!!")
        # self.canvas.itemconfigure(self.status_bg, image=self.img_status_working)

    # OLDER FUNCTIONS AFTER THIS

    def temp_guide(self, count_down):

        for count in range(count_down + 1):
            # self.label.configure(image = self.img_status_radar, fg = self.color_font_status, text = ...
                  self.status_radar+" for "+str(duration)+" sec )\n"+"Starting in "+str(count_down-count))
            self.label.configure(image=self.img_status_temp, fg=self.color_font_status_dark, font=self.font['text'],
                                 text=self.status_temp_guide + "\n\nStarting in " + str(
                                     count_down - count) + "\n(Pedal to skip)")
            self.label.update()
            time.sleep(1)

            # self.label.configure(image = self.img_status_temp, fg = self.color_font_status,font = ...
                  self.font['text'], text = self.status_temp_guide)
        # self.label.update()

    def radar(self, duration, count_down):
        # # PREP COUNTDOWN
        # for count in range(count_down+1):
        #     # self.label.configure(image = self.img_status_radar, fg = self.color_font_status, text = ...
             self.status_radar+" for "+str(duration)+" sec )\n"+"Starting in "+str(count_down-count))
        #     self.label.configure(image = self.img_status_radar, fg = self.color_font_status,font = ...
             self.font['status'], text =  "Hold still for Heart Rate Scan\n\n"+"Starting in "+str(count_down-count))
        #     self.label.update()
        #     time.sleep(1)
        # SCAN DURATION COUNDOWN
        for second in range(duration + 1):
            # self.label.configure(image = self.img_status_radar, fg = self.color_font_status, text = ...
                  self.status_radar+" )\n"+str(duration-second)+" Seconds Remaining..")
            self.label.configure(image=self.img_status_radar, fg=self.color_font_status, font=self.font['status'],
                                 text="(Hold Still)\n\n" + str(duration - second))
            self.label.update()
            time.sleep(1)

        # self.label.configure(image = self.img_status_radar, fg = self.color_font_status, text = ...
             self.status_radar+" )\n"+" Complete")
        self.label.configure(image=self.img_status_radar, fg=self.color_font_status, font=self.font['status'],
                             text="Heart Rate Scan Complete")
        self.label.update()
        time.sleep(2)

    def radar_guide(self, count_down):
        # count_down = 5
        # PREP COUNTDOWN
        for count in range(count_down + 1):
            # self.label.configure(image = self.img_status_radar, fg = self.color_font_status, text = ...
                  self.status_radar+" for "+str(duration)+" sec )\n"+"Starting in "+str(count_down-count))
            self.label.configure(image=self.img_status_radar, fg=self.color_font_status, font=self.font['text'],
                                 text=self.status_radar_guide + "\n\nStarting in " + str(
                                     count_down - count) + "\n(Pedal to skip)")
            self.label.update()
            time.sleep(1)

            # self.label.configure(image = self.img_status_temp, fg = self.color_font_status,font = ...
                  self.font['text'], text = self.status_radar_guide)
        # self.label.update()

    def user_input_temp(self):
```

```
        self.label.configure(image=self.img_status_setup, fg=self.color_font_status_setup, font=self.font['status'],
                             text=self.status_user_input_temp)
        self.label.update()

    def user_input_radar(self):
        self.label.configure(image=self.img_status_setup, fg=self.color_font_status_setup, font=self.font['status'],
                             text=self.status_user_input_radar)
        self.label.update()

    def setup_id(self):
        self.label.configure(image=self.img_status_setup, fg=self.color_font_status_setup, font=self.font['status'],
                             text=self.status_setup_id)
        self.label.update()

    def setup_height(self):
        self.label.configure(image=self.img_status_setup, fg=self.color_font_status_setup, font=self.font['status'],
                             text=self.status_setup_height)
        self.label.update()

    def setup_temp_threshold(self):
        self.label.configure(image=self.img_status_setup, fg=self.color_font_status_setup, font=self.font['status'],
                             text=self.status_setup_temp_threshold)
        self.label.update()

    def setup_temp_offset(self):
        self.label.configure(image=self.img_status_setup, fg=self.color_font_status_setup, font=self.font['status'],
                             text=self.status_setup_temp_offset)
        self.label.update()

    def error_temp(self):
        self.label.configure(image=self.img_status_error, text=self.status_error_temp, font=self.font['status'],
                             fg=self.color_font_status_error)
        self.label.update()

    def error_radar(self, error_instance):
        pass

    def error_unknown(self):
        pass


class DebugFrame:
    def __init__(self, master, color, path_images, font, info_frame, icons_frame, status_frame, settings_frame):
        # self.info = info_frame
        self.icons = icons_frame
        self.status = status_frame
        self.info = info_frame
        self.settings = settings_frame
        self.master = master
        self.frame = tk.Frame(self.master, bg=color["gray"])
        self.frame.pack(fill="x", expand=True)
        if pod.laptop_screen_mode:
            self.frame.pack_configure(before=self.icons.label)
        # self.frame.grid(column=0,row=4)

        self.button = tk.Button(self.frame, relief='flat', bg=color["bg"], fg=color["yellow"], text="Debug!",
                                command=lambda: threading.Thread(target=self.status.radar_results_text, args=([1,2,3],[2,3,4.987],[3,4,5])).start())
        self.button.grid(column=1, row=0, padx=1)
        self.button = tk.Button(self.frame, relief='flat', bg=color["bg"], fg=color["yellow"], text="Radar Cal",
                                command=lambda: [self.icons.off(), self.status.radar_calibration()])
        self.button.grid(column=2, row=0, padx=1)
        self.label_debug_query = tk.Label(self.frame, text="Debugging Panel", bg=color["gray"], fg=color["yellow"])
        self.label_debug_query.grid(column=0, row=0, padx=20)
        self.button = tk.Button(self.frame, relief='flat', bg=color["bg"], fg=color["yellow"], text="Shutter open",
                                command=lambda: tfg.shutter_open())
        self.button.grid(column=3, row=0, padx=1)
        self.button = tk.Button(self.frame, relief='flat', bg=color["bg"], fg=color["yellow"], text="Radar GO",
                                command=lambda: [self.icons.step3(), self.status.radar_scan(5)])
        self.button.grid(column=4, row=0, padx=1)
        self.button = tk.Button(self.frame, relief='flat', bg=color["bg"], fg=color["yellow"], text="DemoRun!",
                                command=lambda: start_scan_demo(info_frame, status_frame, icons_frame))
        self.button.grid(column=5, row=0, padx=1)
        self.button = tk.Button(self.frame, relief='flat', bg=color["bg"], fg=color["yellow"], text="Start",
                                command=lambda: [self.status.start(), self.icons.off()])
        self.button.grid(column=6, row=0, padx=1)
        self.button = tk.Button(self.frame, relief='flat', bg=color["bg"], fg=color["yellow"], text="Temp Inst",
                                command=lambda: [self.icons.step2(), self.status.temp_instructions()])
        self.button.grid(column=7, row=0, padx=1)
        self.button = tk.Button(self.frame, relief='flat', bg=color["bg"], fg=color["yellow"], text="Temp Ct",
                                command=lambda: [self.icons.step2(), self.status.temp_instructions_countdown(5)])
        self.button.grid(column=8, row=0, padx=1)
        self.button = tk.Button(self.frame, relief='flat', bg=color["bg"], fg=color["yellow"], text="Processing",
                                command=lambda: self.status.working())
        self.button.grid(column=9, row=0, padx=1)
        self.button = tk.Button(self.frame, relief='flat', bg=color["bg"], fg=color["yellow"], text="Radar Inst",
                                command=lambda: [self.icons.step3(), self.status.radar_instructions()])
        self.button.grid(column=10, row=0, padx=1)
        self.button = tk.Button(self.frame, relief='flat', bg=color["bg"], fg=color["yellow"], text="Radar Ct",
                                command=lambda: [self.icons.step3(), self.status.radar_instructions_countdown(5)])
        self.button.grid(column=11, row=0, padx=1)
        self.button = tk.Button(self.frame, relief='flat', bg=color["bg"], fg=color["yellow"], text="Radar",
```

```python
                                command=lambda: [self.icons.step3(), self.status.radar_countdown(5)])
        self.button.grid(column=12, row=0, padx=1)
        self.button = tk.Button(self.frame, relief='flat', bg=color["bg"], fg=color["yellow"], text="Settings On",
                                command=lambda: [self.status.label.pack_forget(),
                                                 self.settings.frame.pack_configure(after=self.icons.label,
                                                                                    fill=tk.BOTH, expand=1,
                                                                                    side=tk.TOP),
                                                 self.info.entry_disable()])
        self.button.grid(column=13, row=0, padx=1)
        self.button = tk.Button(self.frame, relief='flat', bg=color["bg"], fg=color["yellow"], text="Settings Off",
                                command=lambda: [self.settings.frame.pack_forget(),
                                                 self.status.label.pack_configure(after=self.icons.label, expand=1)])
        self.button.grid(column=14, row=0, padx=1)

    def check_save(self):
        print(self.info.save_images.get())

    def seek_scan_open(self):
        # my = filedialog.askopenfilename()

        iconX, iconY = pyautogui.locateCenterOnScreen(
            'C:\Dropbox (MIT)\RESEARCH\TESTINGPOD\POD_GITHUB\POD_ui\SeekScan_Icon.png')
        pyautogui.doubleClick(iconX, iconY)

        # path_SeekScan = "C:\Program Files\Seek Thermal\Seek Scan\SeekScan.exe"
        # os.system('"%s"' % my)
        # subprocess.run(path_SeekScan)
        # os.startfile(path_SeekScan)

    def seek_scan_close(self):
        os.system("taskkill /im SeekScan.exe")


class SettingsFrame_old:
    def __init__(self, master, color, path_images, font, info_frame, icons_frame, status_frame):
        self.master = master
        self.info = info_frame
        self.icons = icons_frame
        self.color = color
        self.imagepath = path_images + "\\UI_Status_done.png"  # "ettings.png"

        # self.frame = tk.Frame(self.master,bg=color["gray"])
        self.frame = BgFrame(self.master, self.imagepath, 980, 400, self.color["bg"])
        # self.frame.pack_configure(after=self.icons.label)


class SettingsFrame:
    def __init__(self, master, color, path_images, font, info_frame, icons_frame, status_frame):
        self.master = master
        self.info = info_frame
        self.icons = icons_frame
        self.status = status_frame
        self.color = color
        self.imagepath = path_images + "\\UI_Status_done.png"  # "settings.png"

        sym_deg = u"\N{DEGREE SIGN}"

        # CREATE FRAME OUTLINE
        self.frame = tk.LabelFrame(self.master, fg=self.color["yellow"], bg=self.color["gray"], text="Settings",
                                   height=400, width=980)

        # GET STATION INFO FROM .txt FILE & READ/SAVE SETTINGS
        with open(os.getcwd() + "\\POD_station_IDs.txt", "r") as stations_file:
            self.stations = stations_file.readlines()
            self.stations = [sub.replace('\n', '') for sub in self.stations]
        self.station = tk.StringVar(self.frame)
        self.station.set(self.stations[0])  # MAKE DEFAULT STATION
        self.station_menu = tk.OptionMenu(self.frame, self.station, *self.stations)
        self.station_menu.config(width=20, justify="left", relief="groove", activeforeground=self.color["green"],
                                 highlightthickness=0, bg=self.color["bg"], activebackground=self.color["bg"],
                                 fg=self.color["yellow"])  # font=("Arial", 12, "bold"),
        self.station_menu.grid(column=3, row=1, rowspan=1)
        self.station_id = self.station.get().split(": ")[0]
        self.station_loc = self.station.get().split(": ")[1]

        tk.Label(self.frame, text="Choose Station Location:", bg=self.color["bg"], fg=self.color["yellow"]).grid(
            column=3, row=0, pady=(30, 2), sticky=tk.W)
        # self.station_menu2 = tk.ttk.Combobox(self.frame,values=self.stations)
        # self.station_menu2.grid(column=1,row=2,sticky=tk.W)

        # TRACK WHEN STATION IS CHANGED
        self.station.trace("w", self.callback_station)

        # SYSTEM SETTINGS LABEL
        # self.label_checkboxes = tk.Label(self.frame,text="System Options",bg=self.color["bg"],fg=self.color["yellow"])
        # self.label_checkboxes.grid(column=0,row=0,sticky=tk.W)
        tk.Label(self.frame, text="System Options", bg=self.color["bg"], fg=self.color["yellow"]).grid(column=0, row=0,
                                                                                                        padx=(20, 0),
                                                                                                        pady=(30, 2),
                                                                                                        sticky=tk.W)

```

```python
        # DEBUG CHECKBOX
        self.debug_mode = tk.BooleanVar()
        self.option_debug = tk.Checkbutton(self.frame, text="Debug Mode", variable=self.debug_mode, bg=self.color["bg"],
                                           activebackground=self.color["bg"], activeforeground=self.color["green"],
                                           fg=self.color["white"], selectcolor=self.color["bg"])
        self.option_debug.grid(column=0, row=1, rowspan=1, padx=(20, 0), sticky=tk.W)

        # PEDAL SKIP CHECKBOX
        self.option_pedal_skip = tk.BooleanVar()
        self.option_pedal_skip = tk.Checkbutton(self.frame, text="Allow Pedal Skip", variable=self.option_pedal_skip,
                                                bg=self.color["bg"], activebackground=self.color["bg"],
                                                activeforeground=self.color["green"], fg=self.color["white"],
                                                selectcolor=self.color["bg"])
        self.option_pedal_skip.grid(column=0, row=2, rowspan=2, padx=(20, 0), sticky=tk.W)

        # SEEKSCAN SETTINGS LABEL
        self.label_checkboxes = tk.Label(self.frame, text="SeekScan Options", bg=self.color["bg"],
                                         fg=self.color["yellow"])
        self.label_checkboxes.grid(column=0, row=6, pady=(30, 2), padx=(20, 0), sticky=tk.W)

        # SAVE IMAGES CHECKBOX
        self.save_images = tk.BooleanVar()
        self.option_save_images = tk.Checkbutton(self.frame, text="Save Temperature Images", variable=self.save_images,
                                                 bg=self.color["bg"], activebackground=self.color["bg"],
                                                 activeforeground=self.color["green"], fg=self.color["white"],
                                                 selectcolor=self.color["bg"])
        if not self.debug_mode.get():
            self.option_save_images.config(state=tk.DISABLED)
        self.option_save_images.grid(column=0, row=12, rowspan=1, columnspan=2, padx=(20, 0), pady=(15, 0), sticky=tk.W)

        # SEEKSCAN SOUNDS CHECKBOX
        self.seekscan_sounds = tk.BooleanVar()
        self.option_sounds = tk.Checkbutton(self.frame, text="Temperature Pass/Fail Sounds",
                                            variable=self.seekscan_sounds, bg=self.color["bg"],
                                            activebackground=self.color["bg"], activeforeground=self.color["green"],
                                            fg=self.color["white"], selectcolor=self.color["bg"])
        self.option_sounds.grid(column=0, row=13, rowspan=1, columnspan=2, padx=(20, 0), sticky=tk.W)

        # SEEKSCAN & SYSTEM UNITS
        self.unit = tk.StringVar()
        self.units = [("Celcius", "C"), ("Fahrenheit", "F")]
        self.unit.set(self.units[0][1])
        tk.Label(self.frame, text="Temperature Units:", bg=self.color["bg"], fg=self.color["white"]).grid(column=0,
                                                                                                          row=11,
                                                                                                          padx=(20, 0),
                                                                                                          pady=(15, 2),
                                                                                                          sticky=tk.W)

        self.radio_celcius = tk.Radiobutton(self.frame, bg=self.color["bg"], selectcolor=self.color["bg"],
                                            fg=self.color["white"], text="Celcius", variable=self.unit,
                                            value=self.units[0][1], activebackground=self.color["bg"],
                                            activeforeground=self.color["green"], command=lambda: self.SetUnits())
        self.radio_celcius.grid(column=1, row=11, padx=(0, 0), pady=(15, 2), sticky=tk.W)
        # self.radio_celcius.select()
        self.radio_celcius = tk.Radiobutton(self.frame, bg=self.color["bg"], selectcolor=self.color["bg"],
                                            fg=self.color["white"], text="Fahrenheit", variable=self.unit,
                                            value=self.units[1][1], activebackground=self.color["bg"],
                                            activeforeground=self.color["green"], command=lambda: self.SetUnits())
        self.radio_celcius.grid(column=2, row=11, padx=(0, 0), pady=(15, 2), sticky=tk.W)

        # SEEKSCAN No OF SCANS
        tk.Label(self.frame, text="Number of Temp Scans", bg=self.color["bg"], fg=self.color["white"]).grid(column=0,
                                                                                                            row=7,
                                                                                                            padx=(
                                                                                                                20, 0),
                                                                                                            pady=(2, 2),
                                                                                                            sticky=tk.W)
        self.temp_scan_count = tk.IntVar()
        self.temp_scan_count.set(3)  # Default 3 scans
        self.spinbox_temp_scan_count = tk.Spinbox(self.frame, from_=1, to=5, bg=self.color["light_gray"],
                                                  fg=self.color["yellow"], activebackground=self.color["bg"])
        self.spinbox_temp_scan_count.grid(column=2, row=7, padx=(5, 40), sticky=tk.W)

        # SEEKSCAN BODY TEMP OFFSET
        tk.Label(self.frame, text="Body Temp Offset  [" + sym_deg + "C] :", bg=self.color["bg"],
                 fg=self.color["white"]).grid(column=0, row=8, padx=(20, 0), pady=(2, 2), columnspan=2, sticky=tk.W)
        self.temp_offset = tk.DoubleVar()
        self.temp_offset.set(1.2)  # Default offset
        self.spinbox_temp_offset = tk.Spinbox(self.frame, from_=-5, to=5, bg=self.color["light_gray"],
                                              fg=self.color["yellow"], activebackground=self.color["bg"])
        self.spinbox_temp_offset.grid(column=2, row=8, padx=(5, 40), sticky=tk.W)

        # TEMP THRESHOLD - HIGH (SeekScan also uses this)
        tk.Label(self.frame, text="Body Temp Threshold HIGH  [" + sym_deg + "C] :", bg=self.color["bg"],
                 fg=self.color["white"]).grid(column=0, row=9, padx=(20, 0), pady=(2, 2), columnspan=2, sticky=tk.W)
        self.temp_threshold_high = tk.DoubleVar()
```

```python
        self.temp_threshold_high.set(38)  # Default offset
        self.spinbox_temp_threshold_high = tk.Spinbox(self.frame, from_=36, to=40, bg=self.color["light_gray"],
                                                      fg=self.color["yellow"], activebackground=self.color["bg"])
        self.spinbox_temp_threshold_high.grid(column=2, row=9, padx=(5, 40), sticky=tk.W)

        # TEMP THRESHOLD - LOW
        tk.Label(self.frame, text="Body Temp Threshold LOW  [" + sym_deg + "C] :", bg=self.color["bg"],
                 fg=self.color["white"]).grid(column=0, row=10, columnspan=2, padx=(20, 0), pady=(2, 2), sticky=tk.W)
        self.temp_threshold_low = tk.DoubleVar()
        self.temp_threshold_low.set(35)  # Default offset
        self.spinbox_temp_threshold_low = tk.Spinbox(self.frame, from_=30, to=36, bg=self.color["light_gray"],
                                                     fg=self.color["yellow"], activebackground=self.color["bg"])
        self.spinbox_temp_threshold_low.grid(column=2, row=10, padx=(5, 40), sticky=tk.W)

        # RADAR OPTIONS
        tk.Label(self.frame, text="Radar Options", bg=self.color["bg"], fg=self.color["yellow"]).grid(column=3, row=6,
                                                                                                      padx=(0, 0),
                                                                                                      pady=(30, 2),
                                                                                                      sticky=tk.W)
        tk.Label(self.frame, text="Radar Scan Duration  [sec]:", bg=self.color["bg"], fg=self.color["white"]).grid(
            column=3, row=7, padx=(0, 0), pady=(2, 2), sticky=tk.W)
        self.radar_duration = tk.IntVar()
        self.radar_duration.set(30)  # Default 30 sec
        self.spinbox_radar_duration = tk.Spinbox(self.frame, from_=1, to=60, bg=self.color["light_gray"],
                                                 fg=self.color["yellow"], activebackground=self.color["bg"])
        self.spinbox_radar_duration.grid(column=4, row=7, padx=(20, 0), sticky=tk.W)

        tk.Label(self.frame, text="Radar Scan Countdown  [sec]:", bg=self.color["bg"], fg=self.color["white"]).grid(
            column=3, row=8, padx=(0, 0), pady=(2, 2), sticky=tk.W)
        self.radar_countdown = tk.IntVar()
        self.radar_countdown.set(5)  # Default 30 sec
        self.spinbox_radar_countdown = tk.Spinbox(self.frame, from_=1, to=60, bg=self.color["light_gray"],
                                                  fg=self.color["yellow"], activebackground=self.color["bg"])
        self.spinbox_radar_countdown.grid(column=4, row=8, padx=(20, 0), sticky=tk.W)

        # SAVE & CLOSE
        self.button_exit = tk.Button(self.frame, text="Save & Close", padx=10,
                                     command=lambda: [self.frame.pack_forget(),
                                                      self.status.label.pack_configure(after=self.icons.label,
                                                                                       expand=1),
                                                      self.info.entry_enable()], relief="groove", bg=color["bg"],
                                     fg=color["yellow"], activebackground=self.color["light_gray"],
                                     activeforeground=self.color["yellow"])
        self.button_exit.grid(column=3, row=20, padx=(0, 0), pady=(30, 10), columnspan=1, sticky=tk.W)

        # CHECK IF CONFIG FILE EXISTS, IF SO, USE IT, IF NOT, CREATE IT
        # if os.path.exists(os.getcwd() + "\\POD_config_station_"+str(self.station_id)+".txt"):
        #     with open(os.getcwd() + "\\POD_station_IDs.txt", "r") as station_config:
        #         self.config = station_config.readlines()
        #         self.stations = [sub.replace('\n', '') for sub in self.stations]

    def SetUnits(self):
        print("Units set to: " + self.unit.get())

    def callback_station(self, *args):
        print("Station ID changed to: " + self.station.get())
        self.station_id = self.station.get().split(": ")[0]
        self.station_loc = self.station.get().split(": ")[1]
        # self.label_settings_station_id.configure(text=self.station.get().split(": ")[0])
        # self.label_settings_station_loc.configure(text=self.station.get().split(": ")[1])

    def setup_station(self):
        # USER SETUP IF NO CONFIG FILE, AND WRITE TO CONFIG FILE
        self.status.setup_id()
        self.status.setup_height()
        self.status.setup_temp_threshold()
        self.status.setup_temp_offset()

        # self.button = tk.Button(self.frame, text="Test")
        # self.button.grid(column=0,row=0)
        # self.frame.pack_configure(after=self.icons.label)


class InfoFrame:
    def __init__(self, master, color, path_images, font, icons_frame, status_frame):
        self.master = master
        self.font = font
        self.status = status_frame
        # self.icons = icons2
        self.icons = icons_frame
        self.color = color
        self.id_MIT = None

        sym_deg = u"\N{DEGREE SIGN}"

        self.frame = tk.Frame(self.master, bg=color["bg"])
        self.frame.pack(fill="x", expand=True)
        if pod.laptop_screen_mode:
            self.frame.pack_configure(before=self.icons.label)
        # self.frame.grid(column=0,row=3)

        # CREATE ID ENTRY BOX & BUTTON
```

```
        label_entry_id = tk.Label(self.frame, text="MIT ID  ", font="Arial 24 bold", fg=color["yellow"], bg=color["bg"])
        label_entry_id.grid(column=0, row=0, padx=(12, 0), rowspan=2)
        self.entry_id = tk.Entry(self.frame, width=10, relief='groove', bg=color["bg"], fg=color["white"],
                                 disabledforeground=color["font_gray"], disabledbackground=color["bg"],
                                 font="Arial 34 bold", justify="center")
        self.entry_id.grid(column=1, row=0, rowspan=2)
        self.entry_id.focus_set()

        # START BUTTON
        self.button_id = tk.Button(self.frame, text="Start", font="Arial 18 bold", command=self.press_enter,
                                   relief="groove", bg=color["bg"], fg=color["yellow"],
                                   activebackground=self.color["light_gray"], activeforeground=self.color["yellow"])
        self.button_id.grid(column=2, row=0, padx=(10, 2), rowspan=2)

        self.keyboard_activate()
        # self.master.bind('<Return>', self.press_enter)

        # PEDAL BUTTON
        self.master.bind('<F9>', self.press_pedal)

        # GET STATION INFO FROM .txt FILE & READ/SAVE SETTINGS
        with open(os.getcwd() + "\\POD_station_IDs.txt", "r") as stations_file:
            self.stations = stations_file.readlines()
            self.stations = [sub.replace('\n', '') for sub in self.stations]
        self.station = tk.StringVar(self.frame)
        self.station.set(self.stations[0])  # MAKE DEFAULT STATION
        self.station_menu = tk.OptionMenu(self.frame, self.station, *self.stations)
        self.station_menu.config(width=20, font=("Arial", 12, "bold"), justify="left", relief="flat",
                                 activeforeground=color["green"], highlightthickness=0, bg=color["bg"],
                                 activebackground=color["bg"], fg=color["yellow"])
        self.station_menu.grid(column=3, row=0, rowspan=1)
        self.station_id = self.station.get().split(": ")[0]
        self.station_loc = self.station.get().split(": ")[1]

        # CHECK IF CONFIG FILE EXISTS, IF SO, USE IT, IF NOT, CREATE IT
        # if os.path.exists(os.getcwd() + "\\POD_config_station_"+str(self.station_id)+".txt"):
        #     with open(os.getcwd() + "\\POD_station_IDs.txt", "r") as station_config:
        #         self.config = station_config.readlines()
        #         self.stations = [sub.replace('\n', '') for sub in self.stations]

        # TRACK WHEN STATION IS CHANGED
        self.station.trace("w", self.callback_station)

        # CREATE SAVE IMAGES CHECKBOX (don't display)
        self.save_images = tk.BooleanVar()
        self.option_save_images = tk.Checkbutton(self.frame, text="Save Images", variable=self.save_images,
                                                 bg=color["bg"], activebackground=color["bg"],
                                                 activeforeground=color["yellow"], fg=color["yellow"],
                                                 selectcolor=color["bg"])
        # self.option_save_images.grid(column=3, row=1, rowspan=1)

        # CREATE SETTINGS FRAME
        self.frame_settings = tk.LabelFrame(self.frame, text="Settings", fg=color["white"], bg=color["bg"])
        self.frame_settings.grid(column=4, row=0, padx=(3, 3), rowspan=2)

        self.today = date.today().strftime("%Y-%m-%d")  # %b would be "Aug"

        self.label_settings_date = tk.Label(self.frame_settings, text=self.today, bg=color["bg"], fg=color["yellow"],
                                            font="20", justify="left")
        self.label_settings_date.grid(column=0, row=0)

        self.label_settings_temp_threshold = tk.Label(self.frame_settings, text="Threshold Temp:  " + str(
            pod.temp_threshold) + "  " + sym_deg + "C", bg=color["bg"], fg=color["yellow"], font="20", justify="left")
        self.label_settings_temp_threshold.grid(column=0, row=1, padx=(5, 5))

        # STATION ID INFO, SUPERCEDED BY SELECTION BOX
        # self.label_settings_station_id = tk.Label(self.frame_settings, text="Station ID:  " + str(station_id), bg=color["bg"],fg=color["yellow"], font="20")
        # self.label_settings_station_id.grid(column=1, row=0)
        # self.label_settings_station_loc = tk.Label(self.frame_settings, text="Location:  " + str(station_loc), bg=color["bg"], fg=color["yellow"], font="20")
        # self.label_settings_station_loc.grid(column=1, row=1)

        # CREATE ENVIRONMENT FRAME
        self.frame_environment = tk.LabelFrame(self.frame, text="Environmental Sensors", fg=color["white"],
                                               bg=color["bg"])
        self.frame_environment.grid(column=5, row=0, padx=(3, 3), rowspan=2)

        self.label_environment_temperature = tk.Label(self.frame_environment, text="Ambient Temp:  " + sym_deg + "C",
                                                      bg=color["bg"], fg=color["yellow"], font="20")
        self.label_environment_temperature.grid(column=0, row=0, padx=(5, 5))

        self.label_environment_humidity = tk.Label(self.frame_environment, text="Humidity:  " + "  %RH", bg=color["bg"],
                                                   fg=color["yellow"], font="20")
        self.label_environment_humidity.grid(column=0, row=1)

    def keyboard_activate(self):
        self.master.bind('<Return>', self.press_enter)
        self.master.bind('<a>', self.press_a)
        print("Keyboard Enabled")
```

```python

    def keyboard_deactivate(self):
        self.master.unbind('<Return>')
        self.master.unbind('<a>')
        print("Keyboard Disabled")

    def press_a(self, _event=None):
        print("Yay! Pressed a.")

    def press_enter(self, _event=None):  # underscore prevents warnings
        print('Enter Pressed\n\n')
        self.keyboard_deactivate()
        self.id_MIT = encrypt.encrypt_id(self.entry_id.get())
        start_scan_test(self, self.status, self.icons)


    def press_pedal(self, _event=None):
        print('Pedal Pressed\n\n')
        self.status.pedal_skip = True

    def callback_station(self, *args):
        print("Station ID changed to: " + self.station.get())
        self.station_id = self.station.get().split(": ")[0]
        self.station_loc = self.station.get().split(": ")[1]
        # self.label_settings_station_id.configure(text=self.station.get().split(": ")[0])
        # self.label_settings_station_loc.configure(text=self.station.get().split(": ")[1])

    def setup_station(self):
        # USER SETUP IF NO CONFIG FILE, AND WRITE TO CONFIG FILE
        self.status.setup_id()
        self.status.setup_height()
        self.status.setup_temp_threshold()
        self.status.setup_temp_offset()

    def start_scan(self):
        # DATE UPDATE
        self.today = date.today().strftime("%Y-%m-%d")  # %b would be "Aug"

        print("hooray!")
        print(self.station.get())

    def date_update(self):
        self.today = date.today().strftime("%Y-%m-%d")  # %b would be "Aug"
        self.label_settings_date.config(text=self.today)
        # self.label_settings.update()

    def station_update(self):
        self.station_id = self.station.get().split(": ")[0]
        self.station_loc = self.station.get().split(": ")[1]

    def entry_enable(self):
        self.entry_id.config(state="normal")
        self.button_id.config(state="normal")

    def entry_disable(self):
        # entry_id.insert(0, "---------")
        self.entry_id.delete(0, 'end')
        self.entry_id.config(state="disabled")
        self.button_id.config(state="disabled")


# def start_scan():
#     start_scan_demo(info,status,icons)

def start_scan_demo(info, status, icons):
    pass

def start_scan_demo1(info, status, icons):
    pass

def start_scan_test(info, status, icons):
    today = date.today().strftime("%Y-%m-%d")  # GET TODAY'S DATE
    time_stamp = datetime.now().strftime("%Y-%m-%d_%H_%M_%S")
    # STEP 1 - ID PROCESSING
    icons.off()  # TURN OFF ICONS
    info.entry_disable()  # GRAY OUT ID ENTRY BOX
    status.working()  # DISPLAY "PROCESSING" STATUS
    time.sleep(1)  #

    # STEP 2 - TEMPERATURE
    icons.step2()  # SET ICONS TO TEMP SCAN
    status.temp_instructions()  # DISPLAY INSTRUCTIONS (non-skippable)
    tfg.stepper_goto_user_height(  # ADJUST CAMERA HEIGHT TO EYES
        pod.stepper_cam_height_from_ground_mm,
        pod.sensor_height_from_ground_mm,
        pod.user_dist_top_to_eyes_mm)

    if not thermal.query_cameras():  # CHECK IF THERE ARE RECORDS FOR TODAY OR NOT *potentially can be simplified
        print("No records today!")
        query_last = thermal.query_cameras()
        print(query_last)
        first_scan = True
```

```
else:
    print("Already have records toady!")
    query_last = thermal.query_cameras()[-1]
    first_scan = False

query = query_last  # SET CURRENT QUERY FOR COMPARISON

time.sleep(1)  # TIME FOR HEIGHT ADJUSTMENT
status.temp_instructions_countdown(  # DISPLAY INSTRUCTIONS (skippable)
    pod.temp_instruction_time)

# BEGIN 3 TEMPERATURE SCANS
for j in range(pod.num_temp_scans):
    tfg.shutter_open()
    status.temp_scan(j + 1, pod.num_temp_scans)  # SHOW WHICH x of X SCANS WE ARE ON
    while query == query_last:
        try: query = thermal.query_cameras()[-1]
        except: query = []
        #
        # if first_scan:
        #     query = thermal.query_cameras()[-1]
        #     query = thermal.query_cameras()
        # else:
        #     query = thermal.query_cameras()[-1]
    if first_scan:
        first_scan = False  # REMOVE first_scan flag
    tfg.shutter_close()
    status.working()  # DISPLAY "PROCESSING" STATUS WHILE RECORDING DATA
    print(query)            # DICTIONARY FROM SEEKSCAN
    params = {'humidity': tfg.humidity,
              'temperature': tfg.temperature,
              'time_stamp': time_stamp}  # QUERY tinkerforge for ambient values
    data_loc = file.save_data(query, info, today, params,  # WRITE DATA TO CSV
                   all_time_log=True, excel=False)
    time.sleep(1)  # TIME FOR SAVING DATA
    query_last = query  # SET QUERY FOR COMPARISON
print(data_loc)


# time.sleep(12)

# STEP 3 - RADAR SCANNING
icons.step3()  # SET ICONS TO RADAR SCAN
status.radar_instructions()  # DISPLAY INSTRUCTIONS (non-skippable)
tfg.stepper_goto_user_height(  # ADJUST CAMERA HEIGHT TO CHEST
    pod.stepper_radar_height_from_ground_mm,
    pod.sensor_height_from_ground_mm,
    pod.user_dist_top_to_chest_mm)
time.sleep(5)  # TIME FOR HEIGHT ADJUSTMENT
status.radar_instructions_countdown(  # DISPLAY INSTRUCTIONS (skippable)
    pod.radar_instruction_time)
# status.radar_countdown(pod.radar_countdown_time)# LET USER PREPARE TO HOLD STILL


# id_MIT = encrypt.encrypt_id(info.entry_id.get()) # DELETE!!!

print("Starting Scan...")

threading.Thread(  # RECORD DATA FROM RADAR
    target=radar.scan_user, args = (info.id_MIT, data_loc, time_stamp, pod.radar_scan_time)).start()

thd_ti = ThreadWithReturnValue(
    target=ti.get_vitals, args =  (info.id_MIT, data_loc, time_stamp))
thd_ti.start()


# radar_thd.start()
print("Starting Timer")
status.radar_scan(pod.radar_scan_time)
# threading.Thread(
#     target=status.radar_scan(pod.radar_scan_time + 1)).start()  # RUN TIMER


#status.radar_scan(pod.radar_scan_time + 1)  # RUN TIMER
#time.sleep(pod.radar_scan_time + 1)

# DISPLAY RESULTS TO USER
status.working()
time.sleep(5)


ti_radar_result = thd_ti.join()
t_vec = ti_radar_result[0]
hr_vec = ti_radar_result[1]
rr_vec = ti_radar_result[2]

status.radar_results_text(t_vec, hr_vec, rr_vec)
time.sleep(5)

print(ti_radar_result)
```

```python
    # STEP 4 - COMPLETE & RADAR BACKGROUND SCAN
    icons.step4()  # SET ICONS TO ALL DONE CHECKMARKS
    status.done()  # SET STATUS TO DONE
    time.sleep(5)
    icons.off()  #
    status.radar_calibration()  # TELL PEOPLE TO STAY OUT OF POD
    time.sleep(5)  # TIME TO CLEAR POD
    threading.Thread(  # PERFORM BACKGROUND RADAR SCAN
        target=radar.scan_calibration, args = (info.id_MIT, data_loc, time_stamp, pod.radar_calibration_time)).start()
    time.sleep(pod.radar_calibration_time + 3)  # PAUSE WHILE SCAN IS HAPPENING



    tfg.clear_user_height()   # Reset presence detection

    # GO BACK TO IDLE READY STATE
    status.start()
    icons.step1()
    info.entry_enable()
    info.keyboard_activate()


# %% MAIN SECTION


def main():
    root = tk.Tk()
    root.title("MIT POD VITAL SIGN STATION")
    # root.iconbitmap('c:/gui/codemy.ico')
    # color_bg = "#333333",

    colors = {
        'bg_entry_id': "#ebebeb",
        'bg': "#333333",
        'green': "#39B54A",
        'light_green': "#8DC63F",
        'yellow': "#eeff41",
        'white': "#f2f2f2",
        'red': "#ED1C24",
        'gray': "#333333",
        'light_gray': "#58595B",
        'font_gray': "#3d3d3d"}

    # get screen width and height
    ws = root.winfo_screenwidth()  # width of the screen
    # hs = root.winfo_screenheight() # height of the screen
    w = 1080
    h = 1300

    # calculate x and y coordinates for the Tk root window
    x = (ws / 2) - (w / 2)
    # y = (hs/2) - (h/2)
    y = 0

    # set the dimensions of the screen and where it is placed
    root.geometry('%dx%d+%d+%d' % (w, h, x, y))

    # root.geometry("1080x1300")
    root.configure(bg=colors["bg"])

    # MIT FONT IS "ApexNew-Bold"

    fonts = {
        'status': tk.font.Font(family="Arial Bold", size=50, weight="normal"),
        'info': tk.font.Font(family="Arial Bold", size=20, weight="normal"),
        'banner': tk.font.Font(family="ApexNew-Bold", size=48, weight="normal"),
        'instructions_title': tk.font.Font(family="Arial Bold", size=40, weight="normal"),
        'instructions_text': tk.font.Font(family="Arial Bold", size=28, weight="normal"),
        'medium': tk.font.Font(family="Arial Bold", size=38, weight="normal"),
        'text': tk.font.Font(family="Arial Bold", size=28, weight="normal")}

    font_status = tk.font.Font(family="ApexNew-Bold", size=60, weight="normal")
    font_info = tk.font.Font(family="ApexNew-Bold", size=20, weight="normal")
    # print(str(font_status.actual()))

    humidity = tk.DoubleVar(value=tfg.humidity)
    temperature = tk.DoubleVar(value=tfg.humidity)

    # UI CREATE
    path_images = os.getcwd() + "\\POD_ui"

    banner = Banner(root, colors, path_images)
    icons = Icons(root, colors, path_images)
    status = Status(root, colors, path_images, fonts)

    if pod.debug_mode:
        buffer_line = tk.Frame(root, bg=colors["bg"]).pack(fill="x", pady=10)
        info = InfoFrame(root, colors, path_images, fonts, icons, status)
        settings = SettingsFrame(root, colors, path_images, fonts, info, icons, status)
        debug = DebugFrame(root, colors, path_images, fonts, info, icons, status, settings)


```

```
    else:
        # ADD A BUFFER LINE
        # buffer_line = tk.Frame(root,bg=colors["bg"]).pack(fill="x",pady = 14)
        info = InfoFrame(root, colors, path_images, fonts, icons, status)
        settings = SettingsFrame(root, colors, path_images, fonts, info, icons, status)

    # SEND CAMERA TO LOWEST POSITION
    # stp.goto_height(0,0)

    # root.bind('<Return>', enter_id)

    root.mainloop()


if __name__ == '__main__':
    tfg = Tinkerforge_System()
    pod = Pod_Configs()
    # radar = ti.RadarTi()
    radar = wb.Walabot(pod.radar_chest_distance_mm, pod.radar_chest_circle_r_mm)
    ti = rpi.RPi_ssh('192.168.33.31')
    time.sleep(2)    # Walabot time to connect
    main()
```

A.3.2 Tinkerforge Module

Listing A.2: POD_tinkerforge.py

```
# -*- coding: utf-8 -*-
"""
Created on Sun Mar 14 18:44:32 2021

POD Software Module for Interfacing with Tinkerforge

Runs:
    Temp & Humidity Bricklet
    IO Bricklet to sense limit switch
    Stepper Linear Stage
    Servo bricklet to control camera shutter
    Distance bricklet to measure distance


@author: IGory
"""


# %% IMPORTS

from tinkerforge.ip_connection import IPConnection
from tinkerforge.bricklet_humidity_v2 import BrickletHumidityV2
from tinkerforge.bricklet_io4_v2 import BrickletIO4V2
from tinkerforge.brick_stepper import BrickStepper
from tinkerforge.bricklet_distance_ir_v2 import BrickletDistanceIRV2
from tinkerforge.bricklet_servo_v2 import BrickletServoV2
from tinkerforge.bricklet_distance_us_v2 import BrickletDistanceUSV2

import time

# %% TINKERFORGE FUNCTIONS

class Tinkerforge_System:
    HOST = "localhost"
    PORT = 4223

    def __init__(self):
        # BRICKLET NAMES
        self.hum = None
        self.stepper = None
        self.servo = None
        self.distance= None
        self.io = None
        self.distance_us = None

        self.tinkerforge_initialized = False

        self.temperature = None
        self.humidity = None
        self.stepper_home_limit_sw = False              # Is linear stage at home position?
        self.stepper_height = None                      # What is the current height set at?
        self.stepper_calibrated = False                 # Has stage been calibrated?
        self.mm_per_step = None                         # Based on stepper initialization
        self.sec_per_steps = None                       # Based on stepper initialization
        self.user_detected = False                      # Is there something in front of the distance sensor?
        self.user_detected_time = 0                     # How many times has user been detected
```

```python
        self.user_detection_rate = 4                     # [scans/second]
        self.user_detected_timeout = 5*60                # [seconds] 5 Minute height reset timer
        self.distance_raw_mm = None                      # [mm] Current sensor distance
        self.distance_to_user_mm = None                  # [mm] Distance to user's chest

        # Create IP Connection
        self.ipcon = IPConnection()

        # Register IP Connection callbacks
        self.ipcon.register_callback(IPConnection.CALLBACK_ENUMERATE, self.cb_enumerate)
        self.ipcon.register_callback(IPConnection.CALLBACK_CONNECTED, self.cb_connected)
        self.ipcon.register_callback(IPConnection.CALLBACK_DISCONNECTED, self.cb_disconnected)

        # Connect to brickd, will trigger cb_connected
        self.ipcon.connect(Tinkerforge_System.HOST, Tinkerforge_System.PORT)
        self.ipcon.enumerate()          # GO THROUGH ENUMERATION PROCESS
        time.sleep(3)                   # GIVE TIME FOR BRICKLET DETECTION
        self.initialize()               # SET CONFIGURATIONS FOR BRICKLETS


    # Callback handles device connections and configures possibly lost
    # configuration of lcd and temperature callbacks, backlight etc.
    def cb_enumerate(self, uid, connected_uid, position, hardware_version,
                     firmware_version, device_identifier, enumeration_type):

        if enumeration_type == IPConnection.ENUMERATION_TYPE_CONNECTED or \
           enumeration_type == IPConnection.ENUMERATION_TYPE_AVAILABLE:

            # Enumeration is for io4_v2 Bricklet
            if device_identifier == BrickletIO4V2.DEVICE_IDENTIFIER:
                self.io = BrickletIO4V2(uid, self.ipcon) # Create device object
                # Register input value callback to function cb_input_value
                self.io.register_callback(self.io.CALLBACK_INPUT_VALUE, self.cb_input_value)
                # Set period for input value (channel 0) callback to 0.5s (500ms), "True" means call only when changed
                self.io.set_input_value_callback_configuration(0, 50, False)

            # Enumeration is for Stepper Bricklet
            if device_identifier == BrickStepper.DEVICE_IDENTIFIER:
                self.stepper = BrickStepper(uid, self.ipcon) # Create device object

            # Enumeration is for Distance v2 Bricklet
            if device_identifier == BrickletDistanceIRV2.DEVICE_IDENTIFIER:
                self.distance = BrickletDistanceIRV2(uid, self.ipcon) # Create device object
                # Register distance callback to function cb_distance
                self.distance.register_callback(self.distance.CALLBACK_DISTANCE, self.cb_distance)
                # Set period for distance callback to 1s (1000ms) without a threshold
                self.distance.set_distance_callback_configuration(1000/self.user_detection_rate, False, "x", 0, 0)

            # Enumeration is for Ultrasound v2 Bricklet
            if device_identifier == BrickletDistanceUSV2.DEVICE_IDENTIFIER:
                self.distance_us = BrickletDistanceUSV2(uid, self.ipcon) # Create device object
                # Register distance callback to function cb_distance_us
                self.distance_us.register_callback(self.distance_us.CALLBACK_DISTANCE, self.cb_distance_us)
                # Set period for distance callback to 1s (1000ms) without a threshold
                self.distance_us.set_distance_callback_configuration(1000/self.user_detection_rate, False, "x", ...
                    0, 0)

            # Enumeration is for Servo v2 Bricklet
            if device_identifier == BrickletServoV2.DEVICE_IDENTIFIER:
                self.servo = BrickletServoV2(uid, self.ipcon) # Create device object

            # Enumeration is for BrickletHumidityV2 Bricklet
            if device_identifier == BrickletHumidityV2.DEVICE_IDENTIFIER:
                # Create BrickletHumidityV2 device object
                self.hum = BrickletHumidityV2(uid, self.ipcon)
                # Register callback functions
                self.hum.register_callback(self.hum.CALLBACK_TEMPERATURE, self.cb_temperature)
                self.hum.register_callback(self.hum.CALLBACK_HUMIDITY, self.cb_humidity)
                self.hum.set_temperature_callback_configuration(1000, True, "x", 0, 0)
                self.hum.set_humidity_callback_configuration(1000, True, "x", 0, 0)

    # Callback handles reconnection of IP Connection
    def cb_connected(self, connected_reason):
        # Enumerate devices again. If we reconnected, the Bricks/Bricklets
        # may have been offline and the configuration may be lost.
        # In this case we don't care for the reason of the connection
        self.ipcon.enumerate()
        print("enumerating! cb_connected")


    def cb_disconnected(self, disconnected_reason):
        print("Disconnected!")

    def cb_temperature(self, temperature):
        self.temperature = round(temperature / 100, 1)


    def cb_input_value(self, channel, changed, value):
        self.stepper_home_limit_sw = value
        # if value:
        #     print("Triggered")
        # if value == True and home == False:
```

```
        #     stepper.full_brake()


    def cb_humidity(self, humidity):
        self.humidity = round(humidity / 100, 1)


    def cb_distance(self, distance):
        # self.distance_raw_mm = distance
        if self.distance_raw_mm == None or self.distance_raw_mm > distance:
            self.distance_raw_mm = distance
            print("Distance changed to "+str(self.distance_raw_mm))
        else: pass # print("No change.. "+str(self.distance_raw_mm))
        if self.distance_raw_mm ≤ 1420:         # MAX DISTANCE IS 1486 mm
            self.user_detected = True
        else:
            self.user_detected = False
        self.user_detected_time += 1
        if self.user_detected_time == self.user_detected_timeout*self.user_detection_rate: self.clear_user_height()

    def cb_distance_us(self, distance):
        if distance < 2000: self.distance_to_user_mm = distance
        else:
            self.distance_to_user_mm = None
            print("User too far away or not detected.")

    def clear_user_height(self):
        self.user_detected = False
        self.distance_raw_mm = None
        self.user_detected_time = 0


    def initialize(self):
        # Don't use device before ipcon is connected
        # ...
             https://www.amazon.com/ANNIMOS-Digital-Waterproof-DS3218MG-Control/dp/B076CNKQX4/ref=sr_1_1_sspa?dchild=1&keywords=20kg+servo&link_c
        # 1520 microsec / 333 Hz
        # !!! NEED ACTUAL VALUES !!!
        # Servo 1: Connected to port 0, period of 19.5ms, pulse width of 1 to 2ms
        #          and operating angle -100 to 100
        #time.sleep(5)

        if not self.servo: print("Test my sanity?!")

        if self.servo:
            print("Servo Bricklet Connected!")
            self.servo.set_degree(0, -13500, 13500)      # range, 270 deg model
            self.servo.set_pulse_width(0, 1000, 2000)    # channel, min, max
            self.servo.set_period(0, 19500)
            self.servo.set_motion_configuration(0, 300000, 500000, 500000) # Vel 10000 = 100 deg/sec with full ...
                  ac-/deceleration, slow 10k
            self.servo.set_enable(0, True)
            self.shutter_close()

        # DISTANCE MEASURE - Set moving average to 1 second
        if self.distance:
            print("Distance IR Bricklet Connected!")
            self.distance.set_moving_average_configuration(100)

        # STEPPER SETTINGS INITIALIZE
        if self.stepper:
            print("Stepper Bricklet Connected!")
            self.stepper.set_motor_current(2000)  # 2000 mA  ...
                  https://www.amazon.com/FUYU-Linear-Actuator-Motorized-Stepper/dp/B077QLVRRB/ref=sr_1_3?dchild=1&keywords=fuyu+300&qid=160677323
            self.stepper.set_step_mode(8)  # 1/8 step mode
            step_angle = 1.8  # degrees
            lead = 1.2  # [mm/rev] lead screw pitch * starts
            self.mm_per_step = lead / (360 / step_angle)  # [mm/step]

            # Velocity & Accel Settings
            self.stepper_settings("cal")

            # Step scaling wait factor, found empirically, maybe later come up with robust way to calc based on v ...
                  & a
            self.sec_per_steps = 0.7 / 6000 # was 0.7 for 60k up/dwn

        self.tinkerforge_initialized = True
        self.stepper_calibrated = False
        # self.shutter_close()


    def shutter_close(self):
        if not self.tinkerforge_initialized:
            self.initialize()
        self.servo.set_enable(0, True)
        self.servo.set_position(0,16000)
        print("Shutter Closed")


    def shutter_open(self):
        if not self.tinkerforge_initialized:
            self.initialize()
```

```
        self.servo.set_enable(0, True)
        self.servo.set_position(0,0)
        print("Shutter Open")


    def stepper_settings(self, mode):
        """ Sets speeds and acceleration for stepper motor.
        mode = "cal" for calibration procedure (slow)\t
        mode = "move" for height adjustment (fast)"""
        if mode == "cal":
            vel_cal = 3000  # 5000 steps/s for zeroing procedure
            accel_cal = 6000
            deccel_cal = 6000
            self.stepper.set_speed_ramping(accel_cal, deccel_cal)
            self.stepper.set_max_velocity(vel_cal)
        elif mode == "move":
            vel_move = 50000  # 60000 steps/s for fast adjustment
            accel_move = 60000
            deccel_move = 5000
            self.stepper.set_speed_ramping(accel_move, deccel_move)
            self.stepper.set_max_velocity(vel_move)


    def stepper_calibrate(self):
        print("\nNeed to calibrate first...")
        self.stepper.enable()
        self.stepper_settings("cal")
        if self.stepper_home_limit_sw:
            self.stepper.drive_forward()
            while self.stepper_home_limit_sw:
                time.sleep(0.1)
        self.stepper.full_brake()
        time.sleep(0.3)
        self.stepper_settings("move")
        time.sleep(0.5)
        self.stepper_calibrated = True
        self.stepper.disable()
        self.stepper_height = 0.0
        print("\nHome Position Reached! Calibration Complete.\n")

    def stepper_goto_user_height(self, stepper_offset_from_ground_mm, sensor_offset_from_ground_mm, ...
      user_offset_from_top_head):
        if not self.user_detected:
            print("No user detected or out of sensor range")
            return False

        user_height = sensor_offset_from_ground_mm - self.distance_raw_mm
        # subtract offset of stepper and of user's feature of interest (eye or chest)
        stepper_height = user_height - stepper_offset_from_ground_mm - user_offset_from_top_head

        # add max/min height constraints here
        if stepper_height > 285:
            stepper_height = 285
            print("Max height is " + str((stepper_height + stepper_offset_from_ground_mm)/1000) + " m")
        elif stepper_height < 0:
            stepper_height = 0
            # home = True
            print("Min height is " + str((stepper_height + stepper_offset_from_ground_mm)/1000) + " m")

        # MOVE LINEAR STAGE
        self.stepper_goto_height(stepper_height)

        height_m = (stepper_height + stepper_offset_from_ground_mm)/1000
        height_ft_in = [(stepper_height + stepper_offset_from_ground_mm)/25.4//12, (stepper_height + ...
            stepper_offset_from_ground_mm)/25.4%12]
        print("\nReached height of " + str(round(height_m,2)) + " m  [" + str(int(round(height_ft_in[0],0))) + "' ...
             " + str(int(round(height_ft_in[1],0))) + "'']")

        print("User height reached!")

    def stepper_goto_height(self, height):
        time.sleep(0.1)  # give time for callback to work

        if not self.stepper_calibrated: self.stepper_calibrate()

        steps = (height - self.stepper_height) / self.mm_per_step
        self.stepper.enable()
        self.stepper.set_steps(-1*steps)
        time.sleep(1.0 + self.sec_per_steps*abs(steps))
        self.stepper.disable()
        self.stepper_height = height


# %% GET FUNCTIONS, DEPRECATED

    def get_user_height_mm(self, sensor_offset_from_ground_mm):
        """sensor_offset_from_ground_mm = how high distance sensor is mounted"""
        if not self.tinkerforge_initialized:
            self.initialize()
        #print("Distance: " + str(distance/10.0) + " cm")
        # Get current distance in mm, subtract from ground offset to get height
        print("\n")
```

```
        print(self.distance.get_distance())
        print("\n")
        return sensor_offset_from_ground_mm - self.distance.get_distance()


    def get_humidity(self):
        # print("Humidity: " + str(humidity / 100.0) + " %RH")
        return round(self.hum.get_humidity() / 100, 1)


    def get_temperature(self):
        # print("Temp: " + str(temp / 100) + " deg C")
        return round(self.hum.get_temperature() / 100, 1)

# %% MAIN LOOP FOR TESTING

if __name__ == "__main__":
    test = Tinkerforge_System()
    time.sleep(2)
    test.shutter_open()
##    time.sleep(1)
##    test.shutter_close()
##    #print(test.get_temperature())
##    #print(test.get_humidity())
##    print(test.get_user_height_mm(2000.0))
##    print("\n")
##    #print(test.humidity)
##    #print(test.temperature)
##    time.sleep(1)
##    print(test.distance_raw_mm)
##    print(test.user_detected)
##    test.stepper_goto_user_height(54*25.4,80.5*25.4,5*25.4)
    # input('Press key to exit\n') # Use input() in Python 3
```

A.3.3 Radar Module - Walabot

Listing A.3: POD_radar_walabot.py

```
# -*- coding: utf-8 -*-
"""
Created on Mon Mar 22 13:47:40 2021

@author: IGory
"""

from __future__ import print_function
from sys import platform
from os import system
import os
from datetime import datetime
from scipy.fftpack import fft, ifft
import matplotlib as mpl
import matplotlib.pyplot as plt
import WalabotAPI
import math
import time
import numpy as np
import hdf5storage as hdfs
from matplotlib import pyplot as plt
from matplotlib import use as uuse

# %matplotlib inline


class Walabot:

    def __init__(self, dist_to_chest_mm, radius_on_chest_mm):
        '''
        scan_profile:
        CHANGE THESE MANUALLY FOR NOW

        1 = PROF_SHORT_RANGE_IMAGING = 0x00010000
        PROF_SENSOR = 0x00020000
        PROF_SENSOR_NARROW = 0x00020000 + 1
        PROF_TRACKER = 0x00030000
        PROF_WIDE = 0x00040000

        scan filter:
        FILTER_TYPE_NONE = 0
        FILTER_TYPE_DERIVATIVE = 1
        FILTER_TYPE_MTI = 2

        PROF_NARROW ANTENNA PAIRS
```

```
        [AntennaPair(txAntenna=1, rxAntenna=4),
         AntennaPair(txAntenna=1, rxAntenna=17),
         AntennaPair(txAntenna=1, rxAntenna=18),
         AntennaPair(txAntenna=17, rxAntenna=1),
         AntennaPair(txAntenna=17, rxAntenna=4),
         AntennaPair(txAntenna=17, rxAntenna=18),
         AntennaPair(txAntenna=4, rxAntenna=1),
         AntennaPair(txAntenna=4, rxAntenna=17),
         AntennaPair(txAntenna=4, rxAntenna=18),
         AntennaPair(txAntenna=18, rxAntenna=1),
         AntennaPair(txAntenna=18, rxAntenna=4),
         AntennaPair(txAntenna=18, rxAntenna=17)]
        '''

        # self.scan_profile, scan_filter
        self.wlbt = WalabotAPI
        self.wlbt.Init()
        print("passed init")
        # self.wlbt.SetSettingsFolder("C:\ProgramData\Walabot\WalabotSDK")
        self.wlbt.Initialize()
        print("passed initialize")
        self.isConnected = False
        self.isConfigured = False
        self.isTargets = False
        self.boop = 1
        self.tempt = None
        self.tempx = None

        #  do I need to do Disconnect, Stop, and Clean??

        try: self.connect()
        except:
            print("Walabot connection failed, attempting clean... ")
            self.connect_repair()
        self.configure()

    # THESE 2 ARE GENERAL SCAN FUNCTIONS, SAME NAME FOR ALL RADAR CLASSES
    def scan_user(self, id, data_loc, timestamp, duration,save_to_file=True, live_process=False):
        raw_sig_frame_time = 8*(10**-8)
        frame_periodicity_max = 0.003       # [sec]
        sample_rate_sec = 0.03
        frames = int(duration//sample_rate_sec)
        #filename = str(data_loc) + "RADAR_WLBT_" + str(id) + "_" + str(timestamp) + "_" + "SCAN" + ".mat"
        filename = str(data_loc) + "RADAR_WLBT_" + str(id) + ".mat"

        try:
            scan = self.scan_raw_signal(frames,sample_rate_sec)
        except:
            print("Walabot Scan Failed")
        if save_to_file:
            try:
                self.save_raw_signal_bin(scan, filename)
                print("Walabot Scan Success")
            except: print("Walabot Scan Save Failed")

        if live_process:
            try: return scan
            except: print("Could not obtain scan matrix")

        # self.wlbt.Disconnect()
        # self.wlbt.Clean()


    def scan_calibration(self, id, data_loc, timestamp, duration=5, save_to_file=True, live_process=False):
        '''duration is [sec] at 1 fps, default 5'''
        raw_sig_frame_time = 8 * (10 ** -8)
        frame_periodicity_max = 0.003  # [sec]
        sample_rate_sec = 1
        frames = int(duration // sample_rate_sec)
        filename = str(data_loc) + "RADAR_WLBT_" + str(id) + "_" + str(timestamp) + "_" + "CAL" + ".mat"
        print(filename)

        # scan = self.scan_raw_signal(frames, sample_rate_sec)
        # self.save_raw_signal_bin(scan, filename)

        try:
            scan = self.scan_raw_signal(frames, sample_rate_sec)
        except:
            print("Walabot Background Scan Failed")

        if save_to_file:
            try:
                self.save_raw_signal_bin(scan, filename)
                print("Walabot BG Scan Success")
            except: print("Walabot Background Scan Save Failed")

        if live_process:
            try: return scan
            except: print("Could not obtain scan matrix")

    def single_test(self, duration=1):
        '''duration is [sec] at 1 fps, default 5'''
```

```
        raw_sig_frame_time = 8 * (10 ** -8)
        frame_periodicity_max = 0.003  # [sec]
        sample_rate_sec = 1
        frames = int(duration // sample_rate_sec)
        # filename = str(data_loc) + "RADAR_WLBT_" + str(id) + "_" + str(timestamp) + "_" + "CAL" + ".mat"

        # scan = self.scan_raw_signal(frames, sample_rate_sec)
        # self.save_raw_signal_bin(scan, filename)

        try:
            scan = self.scan_raw_signal(frames, sample_rate_sec)
        except:
            print("Walabot Single Scan Failed")
            return None

        time = scan[0:1,1:]
        signal = scan[1:2,1:]

        self.plot_signal(time,signal)

        # self.wlbt.Stop()
        # self.wlbt.Disconnect()
        # self.wlbt.Clean()


    # THESE ARE INTERNAL SCAN FUNCTIONS
    def scan_user_fast(self, id, data_loc, timestamp, duration,save_to_file=True, live_process=False):
        raw_sig_frame_time = 8*(10**-8)
        frame_periodicity_max = 0.003       # [sec]
        sample_rate_sec = 0.03
        frames = int(duration//sample_rate_sec)
        #filename = str(data_loc) + "RADAR_WLBT_" + str(id) + "_" + str(timestamp) + "_" + "SCAN" + ".mat"
        filename = str(data_loc) + "RADAR_WLBT_" + str(id) + ".mat"

        try:
            scan = self.scan_raw_signal_fast(duration)   # change to multi
        except:
            print("Walabot Scan Failed")
        if save_to_file:
            try:
                self.save_raw_signal_bin(scan, filename)
                print("Walabot Scan Success")
            except: print("Walabot Scan Save Failed")

        if live_process:
            try: return scan
            except: print("Could not obtain scan matrix")

        # self.wlbt.Disconnect()
        # self.wlbt.Clean()

    def scan_raw_signal_multi(self, duration_sec):
        self.wlbt.Start()
        antennaPairs = self.wlbt.GetAntennaPairs()
        print("Starting Scan..")
        signals = []
        time_starts = []
        signals2 = []
        time_starts2 = []
        timeout = time.time() + duration_sec
        while time.time() < timeout:
            t_now = time.time()
            self.wlbt.Trigger()
            signal,t = self.wlbt.GetSignal(antennaPairs[6])
            t_now2 = time.time()
            self.wlbt.Trigger()
            signal2,t2 = self.wlbt.GetSignal(antennaPairs[9])

            time_starts.append(t_now)
            time_starts2.append(t_now2)
            signals.append(signal)
            signals2.append(signal2)


        t_starts = np.array([time_starts]).T                    # make a column vector
        t_starts = t_starts - t_starts[[0]]                     # subtract starting time from all subsequent ...
              times to get Δ in seconds
                                                        # replace initial time with 0
        t_starts2 = np.array([time_starts2]).T                    # make a column vector
        t_starts2 = t_starts2 - t_starts[[0]]                     # subtract starting time from all subsequent ...
              times to get Δ in seconds

        t_starts[0] = 0

        # t_starts2[0] = 0
        sigs = np.array(signals, dtype=np.single)
        sigs2 = np.array(signals2, dtype=np.single)
        time_axis = np.array([t], dtype=np.single)
        right_half = np.vstack([time_axis,sigs])                # stack all times series below time axis
        right_half2 = np.vstack([time_axis,sigs2])

        left_half = np.vstack([np.array([[0]]),t_starts])       # add extra zero to top of time starts
```

```python
            left_half2 = np.vstack([np.array([[0]]),t_starts2])
            sig_matrix = np.hstack([left_half,right_half])
            sig_matrix2 = np.hstack([left_half2,right_half2])

            sig_matrix_sum = np.hstack([left_half,right_half+right_half2])

            sig_matrix_stack = np.vstack([sig_matrix,sig_matrix2])

            self.wlbt.Stop()
            return sig_matrix_stack


        def scan_raw_signal_fast(self, duration_sec):
            self.wlbt.Start()
            antennaPairs = self.wlbt.GetAntennaPairs()
            print("Starting Scan..")
            signals = []
            time_starts = []
            timeout = time.time() + duration_sec
            while time.time() < timeout:
                t_now = time.time()
                self.wlbt.Trigger()
                signal, t = self.wlbt.GetSignal(antennaPairs[6])
                signals.append(signal)
                time_starts.append(t_now)

            t_starts = np.array([time_starts]).T                    # make a column vector
            t_starts = t_starts - t_starts[[0]]                     # subtract starting time from all subsequent times to get Δ in seconds
            t_starts[0] = 0                                         # replace initial time with 0
            sigs = np.array(signals, dtype=np.single)
            time_axis = np.array([t], dtype=np.single)
            right_half = np.vstack([time_axis,sigs])                # stack all times series below time axis
            left_half = np.vstack([np.array([[0]]),t_starts])       # add extra zero to top of time starts
            sig_matrix = np.hstack([left_half,right_half])

            self.wlbt.Stop()
            return sig_matrix


        def scan_raw_signal(self, num_frames,t_sample_sec):
            '''
            SensorNarrow: 6.3-8Ghz, 137 points, 140 kHz
            '''
            self.wlbt.Start()
            # self.wlbt.StartCalibration()

            # stat, prog = self.wlbt.GetStatus()

            # while stat == self.wlbt.STATUS_CALIBRATING and prog < 100:
            #     print("Calibrating " + str(prog) + "%")
            #     self.wlbt.Trigger()
            #     stat, prog = self.wlbt.GetStatus()

            antennaPairs = self.wlbt.GetAntennaPairs()                   # Get list of antenna pairs

            # ADD PAUSE TO PUT OBJECT IN VIEW
            print("Cal done, add object to view!")
            time.sleep(2)
            print("Starting scan...")


            # INITIALIZE LISTS
            signals = []
            time_starts = []

            for i in range(num_frames):
                # Perform trigger and record time of trigger
                t_now = time.time()
                self.wlbt.Trigger()

                # Get the signal amplitudes from the trigger
                signal, t = self.wlbt.GetSignal(antennaPairs[6])
                # Append to the list of frame signals
                signals.append(signal)
                time_starts.append(t_now)

                signal, t = self.wlbt.GetSignal(antennaPairs[9])
                # Append to the list of frame signals
                signals.append(signal)
                time_starts.append(t_now)

                time.sleep(t_sample_sec)# - time.time()*8%1/8)

            # CONCATENATE VECTORS INTO NUMPY ARRAY
            # 1st column is start times of frame
            # 1st row is time axis of each frame
            # data is reflected amplitude
            t_starts = np.array([time_starts]).T                    # make a column vector
            t_starts = t_starts - t_starts[[0]]                     # subtract starting time from all subsequent times to get Δ in seconds
            t_starts[0] = 0                                         # replace initial time with 0
```

```
        sigs = np.array(signals, dtype=np.single)
        time_axis = np.array([t], dtype=np.single)
        right_half = np.vstack([time_axis,sigs])                # stack all times series below time axis
        left_half = np.vstack([np.array([[0]]),t_starts])       # add extra zero to top of time starts
        sig_matrix = np.hstack([left_half,right_half])

        self.wlbt.Stop()
        return sig_matrix

    # DATA PROCESSING

    def FFT(self,x):
        """
            A recursive implementation of
            the 1D Cooley-Tukey FFT, the
            input should have a length of
            power of 2.
            """
        N = len(x)

        if N == 1: return x
        else:
            X_even = self.FFT(x[::2])
            X_odd = self.FFT(x[1::2])
            factor = np.exp(-2j * np.pi * np.arange(N) / N)

            X = np.concatenate(
                [X_even + factor[:int(N / 2)] * X_odd,
                 X_even + factor[int(N / 2):] * X_odd])
            return X

    def plot_signal(self,t, x):
        print(self.boop)
        fig, ax = plt.subplots()
        if self.boop == 1:
            self.tempt, self.tempx = t[0], x[0]
        else:
            ax.plot(self.tempt, self.tempx, color="C"+str(self.boop))
            ax.plot(t[0], x[0], color="C"+str(self.boop+1), alpha=0.4)
            fig2, ax2 = plt.subplots()
            ax2.plot(t[0], x[0] - self.tempx, color="C"+str(self.boop))

        print(x[0])
        print(t[0])

        # if self.boop==2: ax.plot(t[0],x[0],color="C"+str(self.boop))
            # print("jack shit") #fig.show()
        self.boop = self.boop+1
        # print(self.boop)
        # fig.savefig('demo.pdf') #, bbox_inches='tight')
        # fig.show()


        print("stop")


    def save_raw_signal_bin(self, matrix, filename):
        print(matrix)
        hdfs.savemat(filename, {'signals': matrix}, version='7.3')

    def save_raw_signal_csv(self, matrix, filename):
        np.savetxt(filename, matrix, delimiter=",")

    def scan_image_2d(self, distance_to_chest_mm, radius_on_chest_mm):
        pass

    def scan_image_3d(self, distance_to_chest_mm, radius_on_chest_mm):
        pass

    def connect_repair(self):
        try: self.wlbt.Stop()
        except: print("Stop() Failed")
        try: self.wlbt.Disconnect()
        except: print("Disconnect() Failed")
        try: self.wlbt.Clean()
        except: print("Clean() Failed")
        self.wlbt.Init()
        self.wlbt.Initialize()
        self.connect()

    def connect(self):
        try:
            self.wlbt.ConnectAny()
            self.isConnected = True
        except self.wlbt.WalabotError as err:
            if err.code != 19:  # 'WALABOT_INSTRUMENT_NOT_FOUND'
                raise err

    def configure(self):
        # self.wlbt.SetProfile(self.wlbt.PROF_SENSOR)
        self.wlbt.SetProfile(self.wlbt.PROF_SENSOR_NARROW)
        # self.wlbt.SetProfile(self.wlbt.PROF_SENSOR_TRACKER)
```

```
        # self.wlbt.SetProfile(self.wlbt.PROF_WIDE)
        # self.wlbt.SetProfile(self.wlbt.PROF_TRACKER)
        # self.wlbt.SetProfile(self.wlbt.PROF_SHORT_RANGE_IMAGING)

        # PROF_SENSOR, PROF_SENSOR_NARROW, PROF_SHORT_RANGE_IMAGING, PROF_TRACKER, PROF_WIDE

        self.wlbt.SetDynamicImageFilter(self.wlbt.FILTER_TYPE_NONE)
        # self.wlbt.SetDynamicImageFilter(self.wlbt.FILTER_TYPE_MTI)
        # self.wlbt.SetDynamicImageFilter(self.wlbt.FILTER_TYPE_DERIVATIVE)

        # rho, theta, phi = calc_spherical(dist, spot, depth)
        # self.wlbt.SetArenaTheta(-0.1, 0.1, 10)
        # self.wlbt.SetArenaPhi(-0.1, 0.1, 10)
        # self.wlbt.SetArenaR(100, 1000, 5)
        # self.wlbt.SetThreshold(100)

        self.isConfigured = True
        print("Configured")

    def calc_spherical(self, d,r,dr):
        phi = math.atan(r/d)*180/math.pi
        theta = math.atan(r/d)*180/math.pi
        rho = d
        return rho, theta, phi


    def start(self):
        self.wlbt.Start()

    def get_targets(self):
        self.wlbt.Trigger()
        return self.wlbt.GetSensorTargets()

    def stop(self):
        self.wlbt.Stop()

    def disconnect(self):
        self.wlbt.Disconnect()


def print_targets(targets):
    system("cls" if platform == "win32" else "clear")  # clear the screen
    if not targets:
        print("No targets")
        return
    d_min = min(targets, key=lambda t: t[2])[2]  # closest target
    d_amp = max(targets, key=lambda t: t[3])[2]  # "strongest" target
    if d_min == d_amp:
        print("THE DISTANCE IS {:3.0f}\n".format(d_min))
    else:
        print("CALCULATING...\n")


def main_old():
    radar = Walabot(1300,200)
    time_stamp = datetime.now().strftime("%Y-%m-%d_%H_%M_%S")
    time_stamp = "" # blank it out for now
    # radar.connect()
    if not radar.isConnected:
        print("Not Connected")
    else:
        print("Connected")

    # uuse('wxAgg')
    radar.single_test()
    time.sleep(4)
    print("carbon time")
    radar.single_test()
    time.sleep(1)
    radar.wlbt.Stop()
    radar.wlbt.Disconnect()
    radar.wlbt.Clean()



    # scan_user(self, id, data_loc, timestamp, duration)
    # radar.scan_user("Sim","C:/Dropbox (MIT)/RESEARCH/TESTINGPOD/MATLAB/",time_stamp,30)
    # print("Get ready to calibrate!")
    # time.sleep(10)
    # radar.scan_calibration("Sim","C:/Dropbox (MIT)/RESEARCH/TESTINGPOD/MATLAB/",time_stamp,5)

def main():
    # Code for running experiment with Walabot - varying location, oscillation, and absorbers
    time_stamp = datetime.now().strftime("%Y-%m-%d_%H_%M_%S")
    time_stamp = ""
    id_distance = input("Enter Distance [mm]")
    id_HR = input("HR [bpm]?")
    id_HRA = input("HR Amp [mm]?")
    id_RR = input("RR [bpm]?")
    id_RRA = input("RR Amp [mm]?")
    id_RF = input("Absorber? [none,egg,flat]")
    id = id_distance+"mm_HR"+id_HR+"-"+id_HRA+"_RR"+id_RR+"-"+id_RRA+"_absorber-"+id_RF
```

```
    # id = "test"
    print(id)
    trial_num = input("Enter Trial No:")
    duration = int(input("Enter duration [1 for 5-frame test, else for seconds"))
    path = os.getcwd()
    save_location = path + "\\" + "WALABOT_TEST\\Trial_" + trial_num

    # path_date_folder = station_folder + "POD_" + str(station_id) + "_" + date_current
    if os.path.isdir(save_location):
            print(f"Trial {trial_num} directory already exists, will append info!\n")
    else:
        os.mkdir(save_location)

    save_location = save_location + "\\"

    plate_wd_mm = 200
    radar = Walabot(float(id_distance),plate_wd_mm)
    if duration == 1:
        # radar.scan_calibration(id,save_location,time_stamp)
        radar.scan_user(id,save_location,time_stamp,duration=0.03)

    else:

        radar.scan_user_fast(id,save_location,time_stamp,duration)
    radar.wlbt.Stop()
    radar.wlbt.Disconnect()
    radar.wlbt.Clean()



if __name__ == '__main__':
    runs = int(input("How many times to scan?"))
    for i in range(runs):
        main()
        # i += 1
    print("Done!")
```

A.3.4 Data Storage Module

Listing A.4: POD_store_local.py

```
# -*- coding: utf-8 -*-
"""
Created on Wed Dec 30 19:22:10 2020

POD Program Module


Stores data as csv or excel

@author: IGory
"""

# %% IMPORTS
import os, cv2
import pandas as pd
import POD_seekscan as thermal
import time
from POD_encrypt import encrypt_id
# import POD_tinkerforge as tfg

# %% CONSTANTS


# %% FUNCTIONS

def create_station_folder(station_id=0):
    print("\ndef_create_station_folder")
    # CREATE STATION NAME FOLDER
    foldername_station = "\\" + "POD_" + str(station_id) + "_Results"
    path = os.getcwd()
    path_station_folder = path + foldername_station
    if os.path.isdir(path_station_folder):
        print(f"Station {station_id} directory already exists, will append info!\n")
    else:
        os.mkdir(path_station_folder)

    return path_station_folder + "\\"

def create_date_folder(station_folder, date_current, station_id=0):
    print("\ndef_create_date_folder")
    # CREATE A DATE FOLDER INSIDE STATION FOLDER
    path_date_folder = station_folder + "POD_" + str(station_id) + "_" + date_current
```

```
    if os.path.isdir(path_date_folder):
            print(f"Today's POD {station_id} directory already exists, will append info!\n")
    else:
        os.mkdir(path_date_folder)

    return path_date_folder + "\\"


def save_images(query_input, info, date_current, count):
    # imageType1 = "FileThermal"  # Use either 'FileVisible', 'FileThermal', or 'FileScreenshot'
    # imageType2 = "FileVisible"  # Use either 'FileVisible', 'FileThermal', or 'FileScreenshot'
    # imageType3 = "FileScreenshot"  # Use either 'FileVisible', 'FileThermal', or 'FileScreenshot'
    image_types = ["FileThermal", "FileVisible", "FileScreenshot"]
    print("\ndef_save_images")
    # MAKE OR GET STORAGE FOLDERS
    folder_station = create_station_folder(info.station_id)
    folder_date = create_date_folder(folder_station,date_current,info.station_id)

    # GET MIT ID
    id_MIT = info.id_MIT
    print("Saving Images for ID: " + str(id_MIT))

    # GET IMAGES FROM SEEKSCAN INTERNAL STORAGE
    image1 = thermal.readImage(query_input, image_types[0])
    image2 = thermal.readImage(query_input, image_types[1])
    image3 = thermal.readImage(query_input, image_types[2])

    # WRITE IMAGES TO FOLDER
    cv2.imwrite(folder_date + date_current + "_POD_" + str(info.station_id) + "_MIT_ID_" + str(id_MIT) + "_" + ...
          image_types[0][4:] + str(count) + ".jpg", image1)
    cv2.imwrite(folder_date + date_current + "_POD_" + str(info.station_id) + "_MIT_ID_" + str(id_MIT) + "_" + ...
          image_types[1][4:] + str(count) +".jpg", image2)
    cv2.imwrite(folder_date + date_current + "_POD_" + str(info.station_id) + "_MIT_ID_" + str(id_MIT) + "_" + ...
          image_types[2][4:] + str(count) +".jpg", image3)


def save_data(input_query, info, date_current, params, all_time_log=True, excel=False):
    """params is a dictionary containing humidity and temperature, from tinkerforge sensors"""
    # global df, df_csv, station_id
    print("\ndef_save_data")
    # MAKE OR GET STORAGE FOLDERS
    folder_station = create_station_folder(info.station_id)
    folder_date = create_date_folder(folder_station,date_current,info.station_id)

    print(info.station_id)
    print(info.station_loc)

    # GET MID ID & encrypt
    id_MIT = encrypt_id(info.entry_id.get())
    # id_MIT = info.entry_id.get()
    print(id_MIT)


    # GET ENVIRONMENTAL SENSOR INFO HUMIDITY & TEMP
    humidity = params['humidity']
    temp_ambient = params['temperature']
    time_stamp = params['time_stamp']

    # CREATE DATAFRAME FROM QUERY REPLY

    # CREATE CSV (& EXCEL) FILENAMES
    name_csv = folder_date + date_current + "_POD_" + str(info.station_id) + "_Results.csv"
    if all_time_log:
        name_csv_all = folder_station + "POD_" + str(info.station_id) + "_All_Time_Results.csv"
    if excel:
        name_spreadsheet = folder_date + date_current + "_POD_" + str(info.station_id) + "_Results.xlsx"

    # CHECK IF CSV (& EXCEL) EXISTS, IF YES, USE THEM TO CREATE DATAFRAMES, IF NOT, CREATE NEW DATAFRAMES
    if all_time_log:
        if os.path.isfile(name_csv_all):
            df_csv_all = pd.read_csv(name_csv_all)
        else:
            df_csv_all = pd.DataFrame()
    if os.path.isfile(name_csv):
        df_csv = pd.read_csv(name_csv, index_col=None)
    else:
        df_csv = pd.DataFrame()
    if excel:
        if os.path.isfile(name_spreadsheet):
            df_excel = pd.read_excel(name_spreadsheet, index_col=None)
        else:
            df_excel = pd.DataFrame()

    # CREATE TEMP DATAFRAME TO APPEND TO EXISTING
    # Need to set index=False to avoid extra column added when saving to csv
    # Useful lines to modify dataframe:
            # df_temp.rename({"CameraName": "Station_ID"}, axis=1, inplace=True)
            # df_temp.columns.values[0] = "Station_ID"
            # df_temp.drop(df_temp.columns[[0,1]],axis=1,inplace=True)
            # df_temp.rename(columns={df_temp.columns[0]: "Station_ID"}, inplace=True)

    # CLEAN UP QUERY
```

```python
    print(input_query)
    # time.sleep(10)
    input_query.pop("FileVisible")
    input_query.pop("FileThermal")
    input_query.pop("FileScreenshot")
    input_query.pop("BodyTempOffsetManual")
    input_query.pop("BodyTempOffsetAdaptive")
    input_query.pop("BodyTempOffsetType")


    df_temp = pd.DataFrame([input_query])
    df_temp["ID"][-1:] = id_MIT                                             # Write to ID column
    df_temp["TimeStamp"] = str(time_stamp)                                  # Write format without :'s
    df_temp.insert(1, "StationID", info.station_id)                         # Insert Station ID column
    df_temp.insert(df_temp.shape[1], "StationLocation", info.station_loc)   # Insert station_loc at end
    df_temp.insert(df_temp.shape[1], "AmbientTempC", temp_ambient)          # Insert Ambient Temp at end
    df_temp.insert(df_temp.shape[1], "Humidity", humidity)                  # Insert Ambient Humidity at end
    # print("\n------\nDATA FRAME LOOKS LIKE:\n")
    # print(df_temp)
    # print("\n-------\n\n")
    df_csv = df_csv.append(df_temp)                                         # Append temp dataframe to created one
    df_csv.to_csv(name_csv, index=False)

    if all_time_log:
        df_csv_all = df_csv_all.append(df_temp)
        df_csv_all.to_csv(name_csv_all, index=False)
    if excel:
        df_excel = df_excel.append(df_temp)
        df_excel.to_excel(name_spreadsheet, sheet_name='Results', index=False)

    # return save location
    return folder_date
```

A.3.5 Thermal Imaging Module

Listing A.5: POD_seekscan.py

```python
# -*- coding: utf-8 -*-
"""
Created on Wed Jan  6 13:48:57 2021

POD Program Module

To Run SeekScan Camera

@author: IGory
"""

# %% IMPORTS

import urllib.request, json, cv2, requests, os
from datetime import date
import numpy as np
import pyautogui

today = date.today().strftime("%Y-%m-%d")

# %% CLASS

class SeekScan():
    def __init__(self, save_function):
        self.poop = None

    def scan(self):
        pass

    def query_cameras(self):
        query = urllib.request.urlopen(
            'http://localhost:16810/query?StartDate=' + today + '&EndDate=' + today + '&Count=1000').read()  # ...
                   Receive and read the HTTP request
        query_string = query  # .decode('utf-8')  # convert the binary data to 8-bit int, I don't think this does ...
              anything
        results = json.loads(query_string)  # sort the string into a dictionary
        # if debug_mode:
        #     debug.label_debug_query.config(text=str(results["Results"]))

        # print(results["Results"])
        return results["Results"]  # data is stored in a nested dictionary named "Results"; see API guide

    def settings_post(self, temp_threshold=30, body_temp_offset=1.2, units="C", save_images=0, alert_sounds=0):
        settings = ""
        url_post = "http://localhost:16810/settings"
        inputs = dict()
```

```python
        inputs = {"TemperatureThreshold": str(temp_threshold) + units,
                  "BodyTemperatureOffset": str(body_temp_offset) + units, "Units": units, "SaveImages": save_images,
                  "EnableAlertSounds": alert_sounds}
        for i in inputs:
            if "None" not in str(inputs[i]):
                settings = settings + i + "=" + str(inputs[i]) + "&"
                # print(i)
                # print(settings+"\n")
        settings = settings[:-1]  # REMOVE LAST '&'
        print(settings)
        # if debug_mode:
        #     print(settings)
        requests.post(url_post, settings)

    def app_open(self):
        # my = filedialog.askopenfilename()
        # for some reason, can't launch .exe, must click icon on desktop
        iconX, iconY = pyautogui.locateCenterOnScreen(
            'C:\Dropbox (MIT)\RESEARCH\TESTINGPOD\POD_GITHUB\POD_ui\SeekScan_Icon.png')
        pyautogui.doubleClick(iconX, iconY)

    def app_close(self):
        os.system("taskkill /im SeekScan.exe")

    def readImage(self, query_input, filePath):
        resp = urllib.request.urlopen(
            'http://localhost:16810/image/' + query_input[filePath])  # receive image through HTTP request
        image_raw = np.asarray(bytearray(resp.read()),
                               dtype="uint8")  # read data from query, convert from binary to uint8, and store in np array
        image = cv2.imdecode(image_raw, cv2.IMREAD_COLOR)  # decode image and convert to color
        return image

    def resizeImg(self, img, scale):
        width = int(img.shape[1] * scale)
        height = int(img.shape[0] * scale)
        dim = (width, height)

        resized = cv2.resize(img, dim, interpolation=cv2.INTER_AREA)
        return resized


# %% FUNCTIONS

def seek_scan_open(self):
    # my = filedialog.askopenfilename()
    # for some reason, can't launch .exe, must click icon on desktop
    iconX, iconY = pyautogui.locateCenterOnScreen('C:\Dropbox (MIT)\RESEARCH\TESTINGPOD\POD_GITHUB\POD_ui\SeekScan_Icon.png')
    pyautogui.doubleClick(iconX, iconY)

def seek_scan_close(self):
    os.system("taskkill /im SeekScan.exe")

def query_cameras():
    query = urllib.request.urlopen(
        'http://localhost:16810/query?StartDate=' + today + '&EndDate=' + today + '&Count=1000').read()  # Receive and read the HTTP request
    query_string = query  # .decode('utf-8')  # convert the binary data to 8-bit int, I don't think this does anything
    results = json.loads(query_string)  # sort the string into a dictionary
    # if debug_mode:
    #     debug.label_debug_query.config(text=str(results["Results"]))

    # print(results["Results"])
    return results["Results"]  # data is stored in a nested dictionary named "Results"; see API guide

def settings_post(temp_threshold=30, body_temp_offset=1.2, units="C", save_images=0, alert_sounds=0):
    settings = ""
    url_post = "http://localhost:16810/settings"
    inputs = dict()
    inputs = {"TemperatureThreshold": str(temp_threshold) + units,
              "BodyTemperatureOffset": str(body_temp_offset) + units, "Units": units, "SaveImages": save_images,
              "EnableAlertSounds": alert_sounds}
    for i in inputs:
        if "None" not in str(inputs[i]):
            settings = settings + i + "=" + str(inputs[i]) + "&"
            # print(i)
            # print(settings+"\n")
    settings = settings[:-1]  # REMOVE LAST '&'
    print(settings)
    # if debug_mode:
    #     print(settings)
    requests.post(url_post, settings)

def readImage(query_input, filePath):
    resp = urllib.request.urlopen(
        'http://localhost:16810/image/' + query_input[filePath])  # receive image through HTTP request
    image_raw = np.asarray(bytearray(resp.read()),
                           dtype="uint8")  # read data from query, convert from binary to uint8, and store in np array
    image = cv2.imdecode(image_raw, cv2.IMREAD_COLOR)  # decode image and convert to color
    return image
```

```python
def resizeImg(img, scale):
    width = int(img.shape[1] * scale)
    height = int(img.shape[0] * scale)
    dim = (width, height)

    resized = cv2.resize(img, dim, interpolation=cv2.INTER_AREA)
    return resized

if __name__ == '__main__':
    print(query_cameras())
```

A.4 Kiosk Configurations

Listing A.6: POD_Configuration.py

```python
# -*- coding: utf-8 -*-
"""
Created on Tue Mar 16 19:26:49 2021

POD_Configuration

Contains default settings for the POD system

@author: IGory
"""

# %% IMPORTS

from datetime import time


# %% FUNCTIONS & VARIABLES

class Pod_Configs:

    def __init__(self, config_file_path=None):
        self.config_file_path = config_file_path

        # HEIGHTS
        self.stepper_cam_height_from_ground_mm = 54*25.4    # 54 to camera lens
        self.stepper_radar_height_from_ground_mm = 50*25.4  # distance to radar sensor from ground
        self.sensor_height_from_ground_mm = 81*25.4
        self.user_dist_top_to_eyes_mm = 112             # distance from top of head to eyes 50% MALE
        self.user_dist_top_to_chest_mm = 300            # distance from top of head to chest 50% MALE was 520

        # SEEKSCAN CAMERA SETTINGS
        self.save_image = False
        self.units = "C"                            # C of F, how temp will be recorded
        self.temp_threshold = 38                    # deg C, 100.4 deg F
        self.temp_low_threshold = 36.6              # deg C, if below this value, try 2 more times
        self.body_temp_offset = 1.2                 # deg C, default in camera
        self.alert_sounds = 0                       # turn off sounds

        # INSTRUCTION COUNTDOWNS
        self.temp_instruction_time = 5             # Temp instructions countdown
        self.radar_instruction_time = 5            # Radar instructions countdown

        # TEMP SCAN PARAMETERS
        self.num_temp_scans = 3

        # RADAR SCAN PARAMETERS
        self.radar_scan_time = 30                    # [sec] Time to scan person
        self.radar_countdown_time = 5               # [sec] Time to prepare for scan
        self.radar_calibration_time = 5             # [frames] Single frame to scan empty pod for baseline
        self.radar_chest_circle_r_mm = 200          # [mm] radius of measurement on chest
        self.radar_chest_distance_mm = 1300         # [mm] distance from radar to chest

        # DEBUG MODES
        self.debug_mode = False
        self.laptop_screen_mode = False

        # STATION INFORMATION
        self.display_units_temperature = "C"        # ["C" or "F"] How temp will be displayed to user
        self.station_id = None                      # [str] need to fill in
        self.station_location = None                # [str] need to fill in

        # LOAD EXTERNAL CONFIGURATION FILE (future use)
        if self.config_file_path is not None:
            self.load_configs()


    def load_config_file(self):
```

```python
        pass

class Pod_Events:

    def __init__(self):
        pass


# %% MAIN TEST

if __name__ == "__main__":
    pass
```

A.4.1 MIT ID Encryption

Listing A.7: POD_Encrypt.py

```python
# -*- coding: utf-8 -*-
"""
Created on Tue Feb  9 12:23:48 2021

@author: IGory

"""

# %% IMPORTS
import hashlib

# %% FUNCTIONS

# ENCRYPT USING SHA 256
def encrypt_id(string):
    # Encodes the string and returns hexadecimal value
    return hashlib.sha256(string.encode()).hexdigest()

if __name__ == '__main__':
    print(encrypt_id("924853348"))
    print(encrypt_id("1753"))
    print(encrypt_id(""))
```

A.4.2 Radar - Ti

Listing A.8: POD_RPi_link.py

```python
# """
# POD Module for Communication with RPi
# """

## IMPORTS
import socket
import time
import pickle
import select
# from pexpect import pxssh
import getpass
import paramiko
import base64
#import wexpect
import json
import hdf5storage as hdfs
import numpy as np

## CLASSES

class RPi_server:

    def __init__(self, ip_address='192.168.33.31', port=12345, header_length=10):
        self.IP = ip_address
        self.PORT = port
        self.HEADER_LENGTH = header_length
        self.clientSocket = None
        self.clientAddress = None
        self.connected = False
        self.s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.s.bind((self.IP, self.PORT))
```

```python
        self.s.listen(5)
        self.connect()

    def connect(self):
        while not self.connected:
            print(str(self.clientSocket))
            if not self.clientSocket:
                self.clientSocket, self.clientAddress = self.s.accept()
                print(str(self.clientSocket))
            else:
                print(f"Connection from {self.clientAddress} has been established.")
                self.connected = True


    def send_vitals(self, msg):
        print('Sending: '+msg)
        msg = f"{len(msg):<{self.HEADER_LENGTH}}" + msg
        self.clientSocket.send(bytes(msg,"utf-8"))

class RPi_client:

    def __init__(self, ip_address='192.168.33.31', port=12345, header_length=10):
        self.IP = ip_address
        self.PORT = port
        self.HEADER_LENGTH = header_length
        self.s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.s.connect((self.IP, self.PORT))

    def connect(self):
        pass
        # while not self.connected:
        #     print(str(self))

    def get_vitals(self, interval=5, samples=6):
        """ 5 second interval, 6 interval samples delivered """
        sample_counter = 0
        print("Trying to listen...")
        time.sleep(1)
        while True:
            full_msg = ''
            new_msg = True
            while True:
                msg = None
                msg = self.s.recv(16)
                if not msg:
                    print("no msg..")
                else:
                    print("msg!")
                if new_msg:
                    print("new msg len:",msg[:self.HEADER_LENGTH])
                    msglen = int(msg[:self.HEADER_LENGTH])
                    new_msg = False

                print(f"full message length: {msglen}")

                full_msg += msg.decode('utf-8')

                print(len(full_msg))

                if len(full_msg)-self.HEADER_LENGTH == msglen:
                    print("full msg recvd")
                    print(full_msg[self.HEADER_LENGTH:])
                    new_msg = True
                    full_msg =''
                    sample_counter += 1
                    if sample_counter == samples:
                        print("All samples received!")
                        return None

class RPi_ssh:
    """ used to send script to RPi and run it """
    def __init__(self, rpi_ip, rpi_port=22,rpi_user='pi', rpi_pass='pi'):
        # self.s = pxssh.pxssh()
        self.hostname = rpi_ip
        self.username = rpi_user
        self.password = rpi_pass
        self.port = rpi_port
        self.logged_in = False
        self.client = paramiko.SSHClient()
        self.client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
        # self.child = wexpect.spawn('ssh.exe', ['%s@%s', '%', "('server1','a.b.c.d')"])


    def login(self):
        # self.s.login(self.hostname, self.username, self.password)
        self.client.connect(self.hostname, port=self.port, username=self.username, password=self.password)
        time.sleep(1)
        self.logged_in = True
        print("Logged into RPi!")

    def logout(self):
        # self.s.logout()
```

```python
        self.client.close()
        self.logged_in = False
        print("Logged out of RPi!")

    def update(self):
        """function to update script on RPi remotely, future"""
        pass

    def waitForExecCommandEnd(self, channel, command):
        """
        Block untill the end of a command executed by Paramiko.ssh.exec_command
            -channel : (channel) channel stdout returned by Paramiko.ssh.exec_command
            -command : (string) command to run
        """
        while not channel.exit_status_ready():
            print(f"Waiting for end of {command}")#.format(command)
            time.sleep(1)


    def run(self):#, dt=0.5, samples=6):
        """ function to get time, HR, RR vectors spaced dt seconds for num smaples"""
        # _,stdout,_ = self.client.exec_command('cd /home/pi/Desktop/VitalSigns_Ti/mmWave-master/VSD')

        # command = "cd /home/pi/Desktop/VitalSigns_Ti/mmWave-master/VSD"
        # _,_,_ = self.client.exec_command(command)
        #self.login()
        command = 'python POD_Ti_vitalsign.py'
        a, stdout, stderr = self.client.exec_command(command)
        self.waitForExecCommandEnd(stdout.channel, command)
        xx = stdout.read().decode('utf-8')
        xx = xx[21:]
        print(xx)

        stdout.close()
        a.close()
        stderr.close()
        xx = json.loads(xx)
        #self.logout()
        return xx

    def get_vitals(self, user_id, data_loc, time_stamp, dt=0.5, samples=60):
        self.login()
        output = self.run()#dt, samples)
        self.logout()
        time = output[0]
        hr = output[1]
        rr = output[2]

        filename = str(data_loc) + "RADAR_Ti-RPi_" + str(user_id) + "_" + str(time_stamp) + "_" + "SCAN" + ".mat"
        matrix = np.array(output)
        try: self.save_bin(matrix, filename)
        except: print("Save ti failed")

        return output #time, hr, rr

    def save_bin(self, matrix, filename):
        hdfs.savemat(filename, {'signals': matrix}, version='7.3')


def main_old():
    print("Sign of life...")
    rpi = RPi_client(port=12346)
    time.sleep(5)
    rpi.get_vitals()
    print("all vitals gotten!")

def main2():
    print("Starting....")
    rpi = RPi_ssh('192.168.33.31')
    rpi.login()
    test = rpi.run()
    rpi.logout()
    print(test[0])
    print(test[1])
    print(test[2])
    print(type(test))

def main():
    rpi = RPi_ssh('192.168.33.31')
    t,tt,tt = rpi.get_vitals()
    print(tt)

## MAIN TEST AREA
if __name__ == '__main__':
    main2()
```

A.5 Data Processing Scripts

A.5.1 Data processing - Walabot

Listing A.9: MATLAB functions only for brevity.

```

function plot_vitals(time,RRs,HRs,name)
clf; figure;
subplot(2,1,1); plot(time,RRs); title("Avg. Respiration Rate: "+round(mean(RRs),1)+" [brpm]"); xlabel("Time ...
      [sec]"); ylabel("[Breaths/min]");
hold on; yline(mean(RRs),'--k'); hold off;
subplot(2,1,2); plot(time,HRs); title("Avg. Heart Rate: "+round(mean(HRs),1)+" [bpm]"); xlabel("Time [sec]"); ...
      ylabel("[Beats/min]");
hold on; yline(mean(HRs),'--k'); hold off;
sgtitle(name)
end

function [time,RRs,HRs] = ...
      extract_vitals_from_walabot_split(signal,user_distance_m,plots_on,split_time_sec,overlap,plot_title)
% plots_on = false;
params.FMCW_f_range = [6.3e9,8e9];
params.distance_range = [0.1,3];

params_HR.bandpass_filter = true;
params_HR.notch_filter = false;
params_HR.notch_RR = 0;
params_HR.diff_filter = false;
params_HR.minHR = 40;
params_HR.maxHR = 120;
params_HR.multipeak = 1;

[t_s,user_phase_angle] = phase_angle_over_time(signal,user_distance_m,params,plots_on);

% split_time_sec = 10;               % [sec] time to average over
threshold_RR = split_time_sec*10;   % empirical, was 10
% overlap = 2/3;                     % [0-0.9] Fraction of overlap analysis


if split_time_sec == 0
    n_splits = 1;
    split_time_sec = max(t_s);
else
    if overlap == 0
        overlap = 1;
        n_splits = ceil(t_s(end)/split_time_sec);
    else
        n_splits = ceil((t_s(end)/split_time_sec-1)/(1-overlap))+1;
    end
end

time = zeros(1,n_splits);
RRs = zeros(1,n_splits);
% RRsX = RRs;
HRs = zeros(1,n_splits);
clf; figure; hold on
for i = 1:n_splits
    time(i) = (i-1)*split_time_sec*(1-overlap);
    t_s_temp = t_s(time(i) ≤ t_s & t_s < time(i)+split_time_sec);
    user_phase_angle_temp = user_phase_angle(time(i) ≤ t_s & t_s < time(i)+split_time_sec);
    plot(t_s_temp,user_phase_angle_temp); title("Stiched phase angle"); xlabel("Time [sec]");

    t_s_temp = t_s_temp - t_s_temp(1);
    RRs(i) = extract_RR(t_s_temp,user_phase_angle_temp,threshold_RR,plots_on);
%     RRsX(i) = extract_RR_xcorr(t_s_temp,user_phase_angle_temp,false);
    HRs(i) = extract_HR(t_s_temp,user_phase_angle_temp,params_HR,plots_on);
end
hold off

if plots_on
    plot_vitals(time,RRs,HRs,plot_title)
end
% clf; figure
% plot(time,RRsX); hold on; yline(mean(RRsX),'--k');
% title("XCorr Avg. Breathing Rate: "+round(mean(RRsX),1)+" [bpm]"); hold off;

end

function [RR,HR] = extract_vitals_from_walabot_single(signal,user_distance_m,plots_on,interval_sec)

% plots_on = true;
params.FMCW_f_range = [6.3e9,8e9];
params.distance_range = [0.1,2];
[t_s,user_phase_angle] = phase_angle_over_time(signal,user_distance_m,params,plots_on);
```

```
RR_signal = extract_RR(t_s,user_phase_angle,plots_on);

%% Try moving average to smooth RR
% user_phase_angle_filtered_for_RR = movmean(user_phase_angle,100);
% figure, plot(t_s,user_phase_angle); hold on; plot(t_s,user_phase_angle_filtered_for_RR); %hold off
%
% xx = user_phase_angle-user_phase_angle_filtered_for_RR; plot(t_s,xx); hold off;
%
% RR_signal = extract_RR(t_s,user_phase_angle_filtered_for_RR,plots_on)

% figure, plot(t_s,xx)
% extract_HR(t_s,xx,plots_on)

user_phase_angle_filtered = ...
      bandpass(user_phase_angle,[30/60,100/60],1/(t_s(2)),'ImpulseResponse','iir','Steepness',0.5);
% HR_sig = extract_HR(t_s,user_phase_angle,plots_on);
HR_sig = extract_HR(t_s,user_phase_angle_filtered,plots_on);
HR_sig-RR_signal;
% f_temp = fit(t_s',user_phase_angle,'fourier8');
% ff = user_phase_angle-f_temp(t_s);
% HR_sig = extract_HR(t_s,ff,true);
% HR_sig-RR_signal

RR = RR_signal;
HR = HR_sig;
end

function [t_s, phase_angles] = phase_angle_over_time(signal,user_distance,params,plots_on)

t_fast = signal(1,2:end);   % [sec] Fast time vector (across each frame)
t_slow = signal(2:end,1);   % [sec] Slow time vector
L_fast = length(t_fast);    % [--]  Length of fast time vector


% Set min and max range of IF signal frequencies to search
c = physconst("Lightspeed");
t_min = 2*params.distance_range(1)/c;                            % [sec] reflection time to reach closest target
t_max = 2*params.distance_range(2)/c;                            % [sec] reflection time to reach furthest target
f_min = interp1([0,t_fast(end)],params.FMCW_f_range,t_min);      % [Hz]  lowest freq we care about
f_max = interp1([0,t_fast(end)],params.FMCW_f_range,t_max);      % [Hz]  highest freq we care about

% Do a range FFT on the whole signal
Fs = 1/t_fast(2);                           % [samples/sec]
window = hamming(L_fast);                   % create window
% window = hanning(L_fast);
% window = chebwin(L_fast);
signal_bare = signal(2:end,2:end)';
signal_windowed = signal_bare.*window;

L_fast = 2^nextpow2(L_fast*10);                         % Pad FFT Length
range_fft = fft(signal_windowed,L_fast)';               % Outputs rows of FFT'd frames
freqs = Fs*(0:(L_fast/2))/L_fast;                       % [Hz] frequency bins for 1-sided fft

% Trim data to be only in the frequency distance range we care about
[¬,f_min_idx] = min(abs(freqs-f_min)); %f_min_idx = f_min_idx-1;
[¬,f_max_idx] = min(abs(freqs-f_max)); %f_max_idx = f_max_idx+1;

range_fft = range_fft(:,f_min_idx:f_max_idx);
freqs = freqs(f_min_idx:f_max_idx);
distances = linspace(params.distance_range(1),params.distance_range(2),length(freqs));

% Find peaks within this range, using first frame
num_peaks = 5;
[pks,locs] = findpeaks(abs(range_fft(1,:)),'SortStr','descend','NPeaks',num_peaks);

% Find peak closest to user, assuming user doesn't leave range bin
[¬,loc_user_idx] = min(abs(distances(locs)-user_distance(1)));

% Perform interpolation to get uniform slow time axis
interp_steps = 20;                                          % [-] How many steps to subsample
Ts_slow_ave = mean(diff(t_slow));

t_s = 0:Ts_slow_ave/interp_steps:t_slow(end);               % [sec] Create uniform slow time vector
phase_angles = interp1(t_slow,angle(range_fft),t_s);        % Create unformally spaced FFT
phase_angles = detrend(phase_angles(:,locs(loc_user_idx))); % pick out the range_fft peak closest to user (change ...
      later for efficiency)

if plots_on
    % PLOT 1st Frame, compare to windowed
    figure; subplot(3,1,1); plot(t_fast,signal_bare(:,1),':',t_fast,signal_windowed(:,1));
    legend(["Raw Signal","Windowed Signal"]); title("1st Frame Raw Signal"); xlabel("Time [s]"); ylabel("Voltage [¬]");
    % PLOT object distances of first frame
    subplot(3,1,2); plot(distances,abs(range_fft(1,:))',distances(locs),pks,'or');
    title("User at "+round(user_distance(1),2)+"m"); xlabel("Location [m]"); ylabel("|X(t)|");
    hold on; xline(user_distance(1),'k--'); hold off
    % PLOT phase angles of object closest to user distance position reading
    subplot(3,1,3); plot(t_s,phase_angles); title("User phase oscillation"); xlabel("Time [s]"); ylabel("Phase ...
          Angle [rad]")
end
```

```
end

function [t_s, baseband] = extract_baseband(signal,user_distance,params,plots_on)

t_fast = signal(1,2:end);   % [sec] Fast time vector (across each frame)
t_slow = signal(2:end,1);   % [sec] Slow time vector
L_fast = length(t_fast);    % [--]  Length of fast time vector



% Set min and max range of IF signal frequencies to search
c = physconst("Lightspeed");
t_min = 2*params.distance_range(1)/c;                        % [sec] reflection time to reach closest target
t_max = 2*params.distance_range(2)/c;                        % [sec] reflection time to reach furthest target
f_min = interp1([0,t_fast(end)],params.FMCW_f_range,t_min);  % [Hz]  lowest freq we care about
f_max = interp1([0,t_fast(end)],params.FMCW_f_range,t_max);  % [Hz]  highest freq we care about

% Do a range FFT on the whole signal
Fs = 1/t_fast(2);                           % [samples/sec]
window = hamming(L_fast);                   % create window
% window = hanning(L_fast);
% window = chebwin(L_fast);
signal_bare = signal(2:end,2:end)';
signal_windowed = signal_bare.*window;

L_fast = 2^nextpow2(L_fast*10);                         % Pad FFT Length
range_fft = fft(signal_windowed,L_fast)';               % Outputs rows of FFT'd frames
freqs = Fs*(0:(L_fast/2))/L_fast;                       % [Hz] frequency bins for 1-sided fft

% Trim data to be only in the frequency distance range we care about
[~,f_min_idx] = min(abs(freqs-f_min)); %f_min_idx = f_min_idx-1;
[~,f_max_idx] = min(abs(freqs-f_max)); %f_max_idx = f_max_idx+1;

range_fft = range_fft(:,f_min_idx:f_max_idx);
freqs = freqs(f_min_idx:f_max_idx);
distances = linspace(params.distance_range(1),params.distance_range(2),length(freqs));

% Find peaks within this range, using first frame
num_peaks = 5;
[pks,locs] = findpeaks(abs(range_fft(1,:)),'SortStr','descend','NPeaks',num_peaks);

% Find peak closest to user, assuming user doesn't leave range bin
[~,loc_user_idx] = min(abs(distances(locs)-user_distance(1)));

% Perform interpolation to get uniform slow time axis
interp_steps = 20;                                          % [-] How many steps to subsample
Ts_slow_ave = mean(diff(t_slow));

t_s = 0:Ts_slow_ave/interp_steps:t_slow(end);               % [sec] Create uniform slow time vector
baseband = interp1(t_slow,range_fft,t_s);       % Create unformally spaced FFT
baseband = detrend(baseband(:,locs(loc_user_idx))); % pick out the range_fft peak closest to user (change later ...
      for efficiency)

if plots_on
    % PLOT 1st Frame, compare to windowed
    figure; subplot(3,1,1); plot(t_fast,signal_bare(:,1),':',t_fast,signal_windowed(:,1));
    legend(["Raw Signal","Windowed Signal"]); title("1st Frame Raw Signal"); xlabel("Time [s]"); ylabel("Voltage [~]");
    % PLOT object distances of first frame
    subplot(3,1,2); plot(distances,abs(range_fft(1,:))',distances(locs),pks,'or');
    title("User at "+round(user_distance(1),2)+"m"); xlabel("Location [m]"); ylabel("|X(t)|");
    hold on; xline(user_distance(1),'k--'); hold off
    % PLOT phase angles of object closest to user distance position reading
    subplot(3,1,3); plot(t_s,baseband); title("User phase oscillation"); xlabel("Time [s]"); ylabel("Phase Angle ...
          [rad]")
end
end


function RR_bpm = extract_RR(t,x,threshold_height,plots_on)
RR_range = [5,20];

RR_min = RR_range(1);       % [bpm] min expected respiration rate
RR_max = RR_range(2);       % [bpm] max expected respiration rate
f_RR_min = RR_min/60;
f_RR_max = RR_max/60;
if threshold_height == 0
    threshold_height = 300;     % threshold below which respiration is not detected in frequency domain ...
          (experimentally deteremined)
end

L = size(x,1);
w = hamming(L);
x = x.*w;                   % Apply window
L = 2^nextpow2(L*100);      % Add padding
doppler_fft = fft(x-mean(x),L);
Fs = 1/t(2);                % [1/s] Sampling rate
f_RR = Fs*(0:(L/2))/L;      % frequency bins

% Trim to only the respiration rates we care about
[~,f_RR_min_idx] = min(abs(f_RR-f_RR_min));
```

```
[¬,f_RR_max_idx] = min(abs(f_RR-f_RR_max));
f_RR = f_RR(f_RR_min_idx:f_RR_max_idx);
doppler_fft = doppler_fft(f_RR_min_idx:f_RR_max_idx,:);

[tmp_pk,tmp_loc] = ...
    findpeaks(abs(doppler_fft),"NPeaks",6,"SortStr","descend",'MinPeakDistance',150,'MinPeakHeight',threshold_height); %was 200
if ¬isempty(tmp_loc)
    RR_bpm = f_RR(tmp_loc(1))*60;
else
    RR_bpm = 0;
end
if plots_on
%     figure
    hold on
    semilogy(f_RR*60,abs(doppler_fft),f_RR(tmp_loc(:))*60,tmp_pk(:),'o')
    %     ylim([0,2000]);
    yline(threshold_height,':k'); text(15,threshold_height+50,'\downarrow RR Detection Threshold',"FontSize",10);

    xline(RR_bpm,'k--');
    title("Resp. Rate Est "+round(RR_bpm,1)+" breaths/min");
    xlabel("Frequency [bpm]"); ylabel("|Mag|"); hold off

    hs = 0; vs = 130;
    hold on;
    text(f_RR(tmp_loc)*60+hs,tmp_pk+vs,num2cell(round(f_RR(tmp_loc)*60,1)),'FontSize',12); % place peak labels
    hold off;
end
end

function RR_bpm = extract_RR_xcorr(t,x,plots_on)
[r,lags] = xcorr(x);
if plots_on
    plot(lags,r)
end
T = t(2);
[¬,locs] = findpeaks(r,'NPeaks',2,'SortStr',"descend");
RR_bpm = abs(60/((locs(1)-locs(2))*T));
end

function HR_bpm = extract_HR(t,x,parameters,plots_on)
HR_range = [30,180];

HR_min = HR_range(1);       % [bpm] min expected respiration rate
HR_max = HR_range(2);       % [bpm] max expected respiration rate
f_HR_min = HR_min/60;
f_HR_max = HR_max/60;

if exist("parameters.multipeak","var")
    parameters.multipeak = 1;
end

if parameters.notch_filter
    x = ...
        bandstop(x,[(parameters.notch_f-2)/60,(parameters.notch_f+2)/60],1/t(2),"ImpulseResponse","iir","Steepness",0.85);
end


if parameters.bandpass_filter
    x = bandpass(x,[parameters.minHR/60,parameters.maxHR/60],1/(t(2)),'ImpulseResponse','iir','Steepness',0.5);
end



% diff_filter = false;
if parameters.diff_filter
    dx=gradient(x(:))./gradient(t(:));
    d2x=gradient(dx(:))./gradient(t(:));
    x = d2x;
end

% x = diff(x,2);

L = size(x,1);
w = hamming(L);
% w = hann(L);
% w = chebwin(L,1000);
% w = blackmanharris(L);
% w = flattopwin(L);

x = x.*w;                   % Apply window
L = 2^nextpow2(L*100);      % Add padding
doppler_fft = fft(x-mean(x),L);
Fs = 1/t(2);                % [1/s] Sampling rate
f_HR = Fs*(0:(L/2))/L;      % frequency bins

% Trim to only the respiration rates we care about
[¬,f_HR_min_idx] = min(abs(f_HR-f_HR_min));
[¬,f_HR_max_idx] = min(abs(f_HR-f_HR_max));
f_HR = f_HR(f_HR_min_idx:f_HR_max_idx);
doppler_fft = doppler_fft(f_HR_min_idx:f_HR_max_idx,:);

```

```
[tmp_pk,tmp_loc] = findpeaks(abs(doppler_fft),"NPeaks",6,"SortStr","descend",'MinPeakDistance',400);
HR_bpm = f_HR(tmp_loc(1:parameters.multipeak))*60;
if plots_on
%     figure
    hold on



    plot(f_HR*60,abs(doppler_fft),f_HR(tmp_loc(:))*60,tmp_pk(:),'o')
    user_rate = tmp_loc(1);
    xline(f_HR(user_rate)*60,'k--');
    title("Heart Rate Est "+round(f_HR(user_rate)*60,1)+" beats/min");
    xlabel("Frequency [bpm]"); ylabel("|Mag|"); hold off

    hs = 0; vs = 10;
    hold on;
    text(f_HR(tmp_loc)*60+hs,tmp_pk+vs,num2cell(round(f_HR(tmp_loc)*60,1)),'FontSize',12); % place peak labels
    hold off;
end
end

function [freqs, mags] = extract_HR_plain(t,x,parameters,plots_on)
HR_range = [30,180];

HR_min = HR_range(1);       % [bpm] min expected respiration rate
HR_max = HR_range(2);       % [bpm] max expected respiration rate
f_HR_min = HR_min/60;
f_HR_max = HR_max/60;

if exist("parameters.multipeak","var")
    parameters.multipeak = 1;
end

if parameters.notch_filter
    x = ...
        bandstop(x,[(parameters.notch_f -2)/60,(parameters.notch_f+2)/60],1/t(2),"ImpulseResponse","iir","Steepness",0.85);
end


if parameters.bandpass_filter
    x = bandpass(x,[parameters.minHR/60,parameters.maxHR/60],1/(t(2)),'ImpulseResponse','iir','Steepness',0.5);
end



% diff_filter = false;
if parameters.diff_filter
    dx=gradient(x(:))./gradient(t(:));
    d2x=gradient(dx(:))./gradient(t(:));
    x = d2x;
end

% x = diff(x,2);

L = size(x,1);
w = hamming(L);
% w = hann(L);
% w = chebwin(L,1000);
% w = blackmanharris(L);
% w = flattopwin(L);

x = x.*w;                   % Apply window
L = 2^nextpow2(L*100);      % Add padding
doppler_fft = fft(x-mean(x),L);
Fs = 1/t(2);                % [1/s] Sampling rate
f_HR = Fs*(0:(L/2))/L;      % frequency bins

% Trim to only the respiration rates we care about
[¬,f_HR_min_idx] = min(abs(f_HR-f_HR_min));
[¬,f_HR_max_idx] = min(abs(f_HR-f_HR_max));
f_HR = f_HR(f_HR_min_idx:f_HR_max_idx);
doppler_fft = doppler_fft(f_HR_min_idx:f_HR_max_idx,:);

[tmp_pk,tmp_loc] = findpeaks(abs(doppler_fft),"NPeaks",6,"SortStr","descend",'MinPeakDistance',400);
HR_bpm = f_HR(tmp_loc(1:parameters.multipeak))*60;
if plots_on
%     figure
    hold on



    plot(f_HR*60,abs(doppler_fft),'--r');%,f_HR(tmp_loc(:))*60,tmp_pk(:),'o')
    user_rate = tmp_loc(1);
%     xline(f_HR(user_rate)*60,'k--');
%     title("Heart Rate Est "+round(f_HR(user_rate)*60,1)+" beats/min");
    xlabel("Frequency [bpm]"); ylabel("|Mag|"); hold off
%
%     hs = 0; vs = 10;
%     hold on;
%     text(f_HR(tmp_loc)*60+hs,tmp_pk+vs,num2cell(round(f_HR(tmp_loc)*60,1)),'FontSize',12); % place peak labels
%     hold off;
end
```

```
freqs = f_HR*60;
mags = abs(doppler_fft);
end

function plot_raw_FFT(signal,cut_in_half)
t_fast = signal(1,2:end);
L_fast = length(t_fast);
Fs = 1/t_fast(2);
window = hamming(L_fast);
signal_bare = signal(2:end,2:end)';
signal_windowed = signal_bare.*window;

L_fast = 2^nextpow2(L_fast*10);
range_fft = fft(signal_windowed,L_fast)';
freqs = Fs*(0:(L_fast/2-1))/L_fast;

if cut_in_half
    plot(freqs(1:length(freqs)/2)/10^9,abs(range_fft(1,1:length(range_fft)/4)))
else
    plot(freqs/10^9,abs(range_fft(1,1:length(range_fft)/2)))
end

xlabel("Frequency [GHz]"); ylabel("Magnitude of FFT"); grid on; grid(gca,'minor');
end

function [X,freq] = positiveFFT(x,Fs)
N = length(x); %// Taking the length of the input dataset to find the number of samples
k = 0:N-1;
T = N/Fs;   %// Period
freq = k/T; %// Creates the frequency range
X = fft(x)/N;  %// Normalizes the FFT
upperbound = ceil(N/2);
X = X(1:upperbound);   %/* Stores only the first half of the dataset values from the fft to eliminate the ...
    mirrored image of the peaks */
freq = freq(1:upperbound); %/* Stored the x-axis values corresponding to the first half  of the dataset. */
end

function [locs_m,RRs_bpm,HRs_bpm] = vitals(signal, distance_to_user, distance_range, FMCW_f_range, ...
    plots_on,chart_title)
[locs_m,RRs_bpm] = RR(signal,distance_to_user,distance_range,[1,20],FMCW_f_range,plots_on);
[¬,HRs_bpm] = HR(signal,distance_to_user,distance_range,[30,200],FMCW_f_range,plots_on);
sgtitle(chart_title); hold off;
end
```

A.6 Chest Simulation Model - MATLAB

Listing A.10: Code to generate chest displacement waveform in MATLAB.

```
function [t, total_disp] = chest_displacement(HR,RR,A_HR,A_RR,dt,duration,noise)
% CHEST_DISPLACEMENT Something here
%   HR,RR,A_HR,A_RR,dt,duration
%   C = CHEST_DISPLACEMENT(A) adds A to itself.
%
%   C = CHEST_DISPLACEMENT(A,B) adds A and B together.
%
%   See also SUM, PLUS.
%   A_HR = 0.1-0.5 mm
%   A_RR = 1-12 mm
%   dt = 0.001

t = [0:dt:duration];      % time vector
nt = length(t);

RR = zeros(nt,1) + RR;      % Breath per minute (12)
if noise
    RR_noise = .02525;          % add some random walk noise on.. amplitude scaling
    RR = RR+cumsum(RR_noise*randn(size(RR)));
end

Kb = zeros(nt,1) + A_RR; % Amplitude in mm (7)
if noise
    Kb_noise = .005; % add some random walk noise on.
    Kb=Kb+cumsum(Kb_noise*randn(size(RR)));
end

T_ratio = zeros(nt,1) + .4; % ratio between In and out breathing
if noise
    T_noise = .0005; % add some random walk noise on.
    T_ratio=T_ratio+cumsum(T_noise*randn(size(RR)));
end

```

```
tau = zeros(nt,1) + .9; % ratio between In and out breathing
if noise
    tau_noise = .0005; % add some random walk noise on.
    tau=tau+cumsum(tau_noise*randn(size(RR)));
end

% plot(RR)
% ylim([0 30])

RR_signal = make_full_RR_series(t,RR,'Kb',Kb,'T_ratio',T_ratio,'tau',tau);

% HR rates 40-150 ish
% HR displacement .2-.4 mm

HR = zeros(nt,1) + HR; % beats per minutes
if noise
    HR_noise = 0.1; % add some random walk noise on. was 0.1
    HR = HR+cumsum(HR_noise*randn(size(HR)));
end

HR_amp = zeros(nt,1) + A_HR; % [mm] beats per minutes (0.3)
if noise
    HR_noise = 0.0005; % add some random walk noise on. was 0.0005
    HR_amp = HR_amp+cumsum(HR_noise*randn(size(HR_amp)));
end

hr_out=-make_full_HR_series(t,HR,'amp',HR_amp);

total_disp = hr_out+RR_signal;
myFilt = hann(25);
myFilt = myFilt/sum(myFilt);
total_low=filtfilt(myFilt,1,total_disp);

% curLim = [0 duration];
% clf
% % subplot(2,1,1)
% plot(t,total_disp)
% hold on
% hold off
% ylabel('Displacement (mm)')
% xlim(curLim)
% title('Simulated Displacement')

end

function  w_out=make_full_HR_series(t,HR,varargin)

    % handle input arguments
    dt = t(2)-t(1);
    nt = length(t);
    w_out = zeros(size(t));
    amp =  ones(size(t));

    for i = 1:2:length(varargin)
        switch varargin{i}
            case 'amp'
                amp = varargin{i+1};
        end
    end

    cur_idx = 1;
    while cur_idx < nt
        cur_HR = HR(cur_idx);
        cur_T = 60/cur_HR ; % current period in seconds
        cur_amp = amp(cur_idx);
        omega = 30;
        OMEGA = 40;
        % generate HR data.
        [t_single, w_single]=make_single_HR_cycle(cur_T,dt,omega,OMEGA);

        w_single = w_single*cur_amp;
        if cur_idx == 1
%              plot(t_single,w_single)
        end

        targetIdx = cur_idx-1+[1:length(t_single)];
        w_out(targetIdx) = w_single;

        cur_idx = cur_idx + length(t_single);

    end
    w_out = w_out(1:nt);
end
function [t, w]= make_single_HR_cycle(T,dt,omega,OMEGA)

    %T = 1;
    % eta*w, eta assumed to be 1?
    t =[0:dt:T];
    omega = omega/T;
    OMEGA = OMEGA/T;
    gamma = 1;
    b = T/2;
```

```
     c = T*1e-3;

    w = cos(omega*t  +gamma *sin(OMEGA*t));
    w = w.*exp(-(t-b).^2./c);
    w = w/max(abs(w));
    if abs(min(w)) < abs(max(w))
        w = -w;
    end

end
function  w_out=make_full_RR_series(t,RR,varargin)

    % handle input arguments
    for i = 1:2:length(varargin)
        switch varargin{i}
            case 'Kb'
                Kb = varargin{i+1};
            case 'tau'
                tau = varargin{i+1};
            case 'T_ratio'
                T_ratio = varargin{i+1};
        end
    end

    dt = t(2)-t(1);
    nt = length(t);
    w_out = zeros(size(t));
% Set default values.
    if ¬exist('T_ratio','var')    %checks only for variables, if not , put default values
        T_ratio = zeros(size(w_out)) + .4;
    end
    if ¬exist('tau','var')
        tau = zeros(size(w_out)) + .9
    end
    if ¬exist('Kb','var')
        Kb = zeros(size(w_out)) + 8;
    end
    cur_idx = 1;
    while cur_idx < nt
        cur_RR = RR(cur_idx);
        cur_T = 60/cur_RR;      % current period in seconds

        T_ratio_cur = T_ratio(cur_idx);
        Kb_cur = Kb(cur_idx);
        tau_cur = tau(cur_idx);

        % generate respiration data, using current period and parameters,
        % for dt only!
        [t_single, w_single]=make_single_RR_cycle(cur_T,T_ratio_cur,dt,Kb_cur,tau_cur);

        targetIdx = cur_idx-1+[1:length(t_single)];
        w_out(targetIdx) = w_single;

        cur_idx = cur_idx + length(t_single); %advance index by length of cycle

    end
    w_out = w_out(1:nt);

end
function [t, w]=make_single_RR_cycle(T,T_ratio,dt,Kb,tau)

    Ti = T*T_ratio; % time exhale
    Te = T-Ti; % time inhale
    t =[0:dt:T];

    % inhale signal
    w_i = -(Kb/(Ti*Te))*t.^2 +  (Kb*T/(Ti*Te))*t;
    % exhale signal
    w_e =(exp(-(t-Ti)/tau) - exp(-(Te)/tau));
    w_e = Kb/(1-exp(-Te/tau)).*w_e;
    % full signal
    w = w_i;
    w(t>Ti) = w_e(t>Ti) ;

    %plot(t,w)
end
```

Bibliography

[1] Jennifer K. Bender, Michael Brandl, Michael Höhle, Udo Buchholz, and Nadine Zeitlmann. Analysis of asymptomatic and presymptomatic transmission in SARS-CoV-2 outbreak, Germany, 2020. *Emerging Infectious Diseases*, 27(4):1159–1163, 2021. ISSN 10806059. doi: 10.3201/eid2704.204576.

[2] Gregory W. Kirschen, Daniel D. Singer, Henry C. Thode, and Adam J. Singer. Relationship between body temperature and heart rate in adults and children: A local and national study. *American Journal of Emergency Medicine*, 38(5):929–933, 2020. ISSN 15328171. doi: 10.1016/j.ajem.2019.158355. URL https://doi.org/10.1016/j.ajem.2019.158355.

[3] Jouko Karjalainen and Matti Viitasalo. Fever and Cardiac Rhythm. *Archives of Internal Medicine*, 146(6):1169–1171, 1986. ISSN 15383679. doi: 10.1001/archinte.1986.00360180179026.

[4] Neal A. Chatterjee, Paul N. Jensen, Andrew W. Harris, Daniel D. Nguyen, Henry D. Huang, Richard K. Cheng, Jainy J. Savla, Timothy R. Larsen, Joanne Michelle D. Gomez, Jeanne M. Du-Fay-de Lavallaz, Rozenn N. Lemaitre, Barbara McKnight, Sina A. Gharib, and Nona Sotoodehnia. Admission respiratory status predicts mortality in COVID-19. *Influenza and other Respiratory Viruses*, (April):1–4, 2021. ISSN 17502659. doi: 10.1111/irv.12869.

[5] Tejaswini Mishra, Meng Wang, Ahmed A. Metwally, Gireesh K. Bogu, Andrew W. Brooks, Amir Bahmani, Arash Alavi, Alessandra Celli, Emily Higgs, Orit Dagan-Rosenfeld, Bethany Fay, Susan Kirkpatrick, Ryan Kellogg, Michelle Gibson, Tao Wang, Erika M. Hunting, Petra Mamic, Ariel B. Ganz, Benjamin Rolnik, Xiao Li, and Michael P. Snyder. Pre-symptomatic detection of COVID-19 from smartwatch data. *Nature Biomedical Engineering*, 4(12):1208–1220, 2020. ISSN 2157846X. doi: 10.1038/s41551-020-00640-6. URL http://dx.doi.org/10.1038/s41551-020-00640-6.

[6] Aravind Natarajan, Hao Wei Su, and Conor Heneghan. Assessment of physiological signs associated with COVID-19 measured using wearable devices. *npj Digital Medicine*, 3(1), 2020. ISSN 23986352. doi: 10.1038/s41746-020-00363-7. URL http://dx.doi.org/10.1038/s41746-020-00363-7.

[7] Guokang Zhu, Jia Li, Zi Meng, Yi Yu, Yanan Li, Xiao Tang, Yuling Dong, Guangxin Sun, Rui Zhou, Hui Wang, Kongqiao Wang, and Wang Huang. Learning from Large-Scale Wearable Device Data for Predicting Epidemics Trend of COVID-19. *Discrete Dynamics in Nature and Society*, 2020(Cdc), 2020. ISSN 1607887X. doi: 10.1155/2020/6152041.

[8] Dean J. Miller, John V. Capodilupo, Michele Lastella, Charli Sargent, Gregory D. Roach, Victoria H. Lee, and Emily R. Capodilupo. Analyzing changes in respiratory rate to predict the risk of COVID-19 infection. *PLoS ONE*, 15(12 December):1–10, 2020. ISSN 19326203. doi: 10.1371/journal.pone.0243693. URL http://dx.doi.org/10.1371/journal.pone.0243693.

[9] Mamady Kebe, Rida Gadhafi, Baker Mohammad, Mihai Sanduleanu, Hani Saleh, and Mahmoud Al-qutayri. Human vital signs detection methods and potential using radars: A review. *Sensors (Switzerland)*, 20(5), 2020. ISSN 14248220. doi: 10.3390/s20051454.

[10] Peter H. Charlton, Drew A. Birrenkott, Timothy Bonnici, Marco A.F. Pimentel, Alistair E.W. Johnson, Jordi Alastruey, Lionel Tarassenko, Peter J. Watkinson, Richard Beale, and David A. Clifton. Breathing Rate Estimation from the Electrocardiogram and Photoplethysmogram: A Review. *IEEE Reviews in Biomedical Engineering*, 11: 2–20, 2018. ISSN 19411189. doi: 10.1109/RBME.2017.2763681.

[11] John Allen. Photoplethysmography and its application in clinical physiological measurement. *Physiological Measurement*, 28(3), 2007. ISSN 09673334. doi: 10.1088/0967-3334/28/3/R01.

[12] Toshiyo Tamura, Yuka Maeda, Masaki Sekine, and Masaki Yoshida. Wearable photoplethysmographic sensors—past and present. *Electronics*, 3(2):282–302, 2014. ISSN 20799292. doi: 10.3390/electronics3020282.

[13] Prasanna Tilakaratna. How pulse oximeters work explained simply. *howequipmentworks.com*. URL https://www.howequipmentworks.com/pulse_oximeter/.

[14] Abishek Swaminathen. What is core temperature? how is it measured? *ONiO*. URL https://www.onio.com/article/what-is-core-temperature-how-is-it-measured.html.

[15] J. Larry Jameson, Anthony S. Fauci, Dennis L. Kasper, Stephen L. Hauser, Dan L. Longo, and Joseph Loscalzo. *Harrison's Principles of Internal Medicine*. McGraw-Hill Education, New York, NY, USA, 2018. ISBN 1259644030.

[16] Braid A. MacRae, Simon Annaheim, Christina M. Spengler, and René M. Rossi. Skin temperature measurement using contact thermometry: A systematic review of setup variables and their effects on measured values. *Frontiers in Physiology*, 9(JAN):1–24, 2018. ISSN 1664042X. doi: 10.3389/fphys.2018.00029.

[17] John Merchant. Infrared temperature measurement theory and application. *Omega Engineering*. URL https://www.omega.com/en-us/resources/infrared-temperature-measurement-theory-application.

[18] Mlx90614 family datasheet. *Melexis*, . URL https://www.melexis.com/en/documents/documentation/datasheets/datasheet-mlx90614.

[19] FluxTeq. Differential_temperature_thermopile. *Wikipedia*. URL https://commons.wikimedia.org/w/index.php?curid=57734898.

[20] FeuRenard. Cross-sectional view of a microbolometer. *Wikipedia*. URL https://commons.wikimedia.org/w/index.php?curid=46553556.

[21] How do you calibrate a thermal imaging camera? *FLIR Teledyne*, . URL https://www.flir.com/discover/professional-tools/how-do-you-calibrate-a-thermal-imaging-camera/.

[22] Giovanni Battista Dell'isola, Elena Cosentini, Laura Canale, Giorgio Ficco, and Marco Dell'isola. Noncontact body temperature measurement: Uncertainty evaluation and screening decision rule to prevent the spread of covid-19. *Sensors (Switzerland)*, 21(2): 1–20, 2021. ISSN 14248220. doi: 10.3390/s21020346.

[23] Jonathan R. Bull, Simon P. Rowland, Elina Berglund Scherwitzl, Raoul Scherwitzl, Kristina Gemzell Danielsson, and Joyce Harper. Real-world menstrual cycle characteristics of more than 600,000 menstrual cycles. *npj Digital Medicine*, 2(1), 2019. ISSN 2398-6352. doi: 10.1038/s41746-019-0152-7. URL http://dx.doi.org/10.1038/s41746-019-0152-7.

[24] Mehrdad Nosrati and Negar Tavassolian. Accurate Doppler Radar-Based Cardiopulmonary Sensing Using Chest-Wall Acceleration. *IEEE Journal of Electromagnetics, RF and Microwaves in Medicine and Biology*, 3(1):41–47, 2019. ISSN 24697249. doi: 10.1109/JERM.2018.2879452.

[25] Adeel Ahmad, June Chul Roh, Dan Wang, and Aish Dubey. Vital signs monitoring of multiple people using a FMCW millimeter-wave sensor. *2018 IEEE Radar Conference, RadarConf 2018*, (4):1450–1455, 2018. doi: 10.1109/RADAR.2018.8378778.

[26] Introduction to mmwave radar sensing: Fmcw radars. *Texas Instruments*. URL https://training.ti.com/mmwave-training-series.

[27] Alessandro R. Guazzi, Mauricio Villarroel, João Jorge, Jonathan Daly, Matthew C. Frise, Peter A. Robbins, and Lionel Tarassenko. Non-contact measurement of oxygen saturation with an RGB camera. *Biomedical Optics Express*, 6(9):3320, 2015. ISSN 2156-7085. doi: 10.1364/boe.6.003320.

[28] Dangdang Shao, Chenbin Liu, Francis Tsow, Yuting Yang, Zijian Du, Rafael Iriya, Hui Yu, and Nongjian Tao. Noncontact Monitoring of Blood Oxygen Saturation Using Camera and Dual-Wavelength Imaging System. *IEEE Transactions on Biomedical Engineering*, 63(6):1091–1098, 2016. ISSN 15582531. doi: 10.1109/TBME.2015.2481896.

[29] Francesco Lamonaca, Domenico Luca Carni, Domenico Grimaldi, Alfonso Nastro, Maria Riccio, and Vitaliano Spagnolo. Blood oxygen saturation measurement by smartphone camera. *2015 IEEE International Symposium on Medical Measurements and Applications, MeMeA 2015 - Proceedings*, pages 359–364, 2015. doi: 10.1109/MeMeA.2015.7145228.

[30] Lingqin Kong, Yuejin Zhao, Liquan Dong, Yiyun Jian, Xiaoli Jin, Bing Li, Yun Feng, Ming Liu, Xiaohua Liu, and Hong Wu. Non-contact detection of oxygen saturation based on visible light imaging device using ambient light. *Optics Express*, 21(15):17464, 2013. ISSN 1094-4087. doi: 10.1364/oe.21.017464.

[31] Mountain David. (12) Patent Application Publication (10) Pub . No .: US 2017 / 0215756A1. 1(19):2015–2018, 2017.

[32] FAQ: Positive tests: Isolation, quarantine, re-testing, and travel. *MIT Medical*. URL https://medical.mit.edu/faqs/positive-tests-isolation-quarantine-retesting#faq-1.

[33] Sarah. How accurate are Kinsa thermometers? . *Kinsa*. URL https://support.kinsahealth.com/hc/en-us/articles/360048491031-How-accurate-are-Kinsa-thermometers-.

[34] Vladimir L. Petrovic, Milica M. Jankovic, Anita V. Lupsic, Veljko R. Mihajlovic, and Jelena S. Popovic-Bozovic. High-Accuracy Real-Time Monitoring of Heart Rate Variability Using 24 GHz Continuous-Wave Doppler Radar. *IEEE Access*, 7:74721–74733, 2019. ISSN 21693536. doi: 10.1109/ACCESS.2019.2921240.

[35] Antonio Albanese, Limei Cheng, Mauro Ursino, and Nicolas W. Chbat. An integrated mathematical model of the human cardiopulmonary system: Model development. *American Journal of Physiology - Heart and Circulatory Physiology*, 310(7): H899–H921, 2016. ISSN 15221539. doi: 10.1152/ajpheart.00230.2014.

[36] J. S. Mecklenburgh and W. W. Mapleson. Ventilatory assistance and respiratory muscle activity. 2: Simulation with an adaptive active ('aa' or 'a-squared') model lung. *British Journal of Anaesthesia*, 80(4):434–439, 1998. ISSN 00070912. doi: 10.1093/bja/80.4.434.

[37] A. De Groote, M. Wantier, G. Cheron, M. Estenne, and M. Paiva. Chest wall motion during tidal breathing. *Journal of Applied Physiology*, 83(5):1531–1537, 1997. ISSN 87507587. doi: 10.1152/jappl.1997.83.5.1531.

[38] Carina Barbosa Pereira, Xinchi Yu, Michael Czaplik, Vladimir Blazek, Boudewijn Venema, and Steffen Leonhardt. Estimation of breathing rate in thermal imaging videos: a pilot study on healthy human subjects. *Journal of Clinical Monitoring and Computing*, 31(6):1241–1254, 2017. ISSN 15732614. doi: 10.1007/s10877-016-9949-y.

[39] Ni myrio-1900 user guide and specifications. *National Instruments*, . URL https://www.ni.com/pdf/manuals/376047c.pdf.

[40] Pull out torque curve of 23he22-4004d-e1000. *StepperOnline*, . URL https://www.omc-stepperonline.com/download/23HE22-4004D-E1000_Torque_Curve.pdf.

[41] mmwave radar mmwave sensor evaluation solution batman bm201-vsd mmwave evm kit mmwave vital signs detection (bm201-vsd). *Joybien*. URL http://www.joybien.com/product/P_mmWave(Vital%20Signs(BM201-VSD)).html.