Developing a robust system for occupancy detection in the household

Ash Tyndall

This report is submitted as partial fulfilment of the requirements for the Honours Programme of the School of Computer Science and Software Engineering, The University of Western Australia, 2015

Abstract

This is the abstract.

Keywords: keyword, keyword
CR Categories: category, category

A digital copy of this document and supporting files can be found at `http://github.com/atyndall/honours`.

Acknowledgements

These are the acknowledgements.

Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

The proportion of elderly and mobility-impaired people is predicted to grow dramatically over the next century, leaving a large proportion of the population unable to care for themselves, and also reducing the number of human carers available [8]. With this issue looming, investments are being made in technologies that can provide the support these groups need to live independent of human assistance.

With recent advance in low cost embedded computing, such as the Arduino and Raspberry Pi, the ability to provide a set of interconnected sensors, actuators and interfaces to enable a low-cost 'smart home for the disabled' that takes advantage of the Internet of Things (IoT) is becoming increasingly achievable.

Sensing techniques to determine occupancy, the detection of the presence and number of people in an area, are of particular use to the elderly and disabled. Detection can be used to inform various devices that change state depending on the user's location, including the better regulation energy hungry devices to help reduce financial burden. Household climate control, which in some regions of Australia accounts for up to 40% of energy usage [5] is one area in which occupancy detection can reduce costs, as efficiency can be increased with annual energy savings of up to 25% found in some cases [7].

While many of the above solutions achieve excellent accuracies, in many cases they suffer from problems of installation logistics, difficult assembly, assumptions on user's technology ownership and component cost. In a smart home for the disabled, accuracy is important, but accessibility is paramount.

The goal of this research project is to devise an occupancy detection system that forms part of a larger 'smart home for the disabled', and intergrates into the IoT, that meets the following qualitative accessibility criteria;

- *Low Cost*: The set of components required should aim to minimise cost, as these devices are intended to be deployed in situations where the serviced user may be financially restricted.

- *Non-Invasive*: The sensors used in the system should gather as little information as necessary to achieve the detection goal; there are privacy concerns with the use of high-definition sensors.

- *Energy Efficient*: The system may be placed in a location where there is no access to mains power (e.g. roof), and the retrofitting of appropriate power can be difficult; the ability to survive for long periods on only battery power is advantageous.

- *Reliable*: The system should be able to operate without user intervention or frequent maintenance, and should be able to perform its occupancy detection goal with a high degree of accuracy.

To create a picture of what options there are in this sensing area, a literature review of the available sensor types and wireless sensor architectures is needed. From this list, proposed solutions will be compared against the aforementioned accessibility criteria to determine their suitability.

CHAPTER 2

Literature Review

To achieve the accessibility criteria, a wide variety of sensing approaches must be considered. It can be difficult to approach the board variety of sensor types in the field, so a structure must be developed through which to evaluate them. Teixeira, Dublon and Savvides [24] propose a 5-element human-sensing criteria which provides a structure through which we may define the broad quantitative requirements of different sensors.

These quantitative requirements can be used to exclude sensing options that clearly cannot meet the requirements before the more specific qualitative accessibility criteria will be considered for those remaining sensors.

The quantitative criteria elements are;

1. *Presence*: Is there any occupant present in the sensed area?
2. *Count*: How many occupants are there in the sensed area?
3. *Location*: Where are the occupants in the sensed area?
4. *Track*: Where do the occupants move in the sensed area? (local identification)
5. *Identity*: Who are the occupants in the sensed area? (global identification)

At a fundamental level, this research project requires a sensor system that provides both Presence and Count information. To assist with the reduction of privacy concerns, excluding systems that permit Identity will generally result in a less invasive system also. The presence of Location or Track are irrelevant to our project's goals, but overall, minimising these elements should in most cases help to maximise the energy efficiency of the system also.

Teixeira, Dublon and Savvides [24] also propose a measurable occupancy sensor taxonomy (see Figure 2.1 on the following page), which categorises different

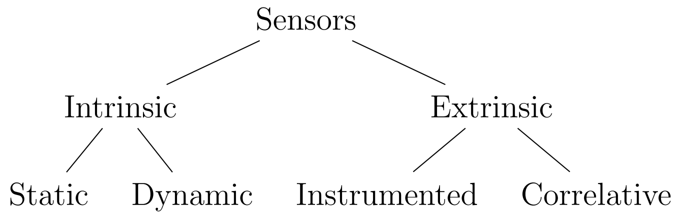


Figure 2.1: Taxonomy of occupancy sensors

sensing systems in terms of what information they use as a proxy for human-sensing. We use this taxonomy here as a structure through which we group and discuss different sensor types.

2.1 Intrinsic traits

Intrinsic traits are those which can be sensed that are a direct property of being a human occupant. Intrinsic traits are particularly useful, as in many situations they are guaranteed to be present if an occupant is present. However, they do have varying degrees of detectability and differentiation between occupants. Two main subcategories of these sensor types are static and dynamic traits.

2.1.1 Static traits

Static traits are physiologically derived, and are present with most (living) occupants. One key static trait that can be used for occupant sensing is that of thermal emissions. All human occupants emit distinctive thermal radiation in both resting and active states. The heat signatures of these emissions could potentially be measured with some apparatus, counted, and used to provide Presence and Count information to a sensor system, without providing Identity information.

Beltran, Erickson and Cerpa [7] propose Thermosense, a system that uses a type of thermal sensor known as an Infrared Array Sensor (IAR). This sensor is much like a camera, in that it has a field of view which is divided into "pixels"; in this case an 8×8 grid of detected temperatures. This sensor is mounted on an embedded device on the ceiling, along with a Passive Infrared Sensor (PIR), and uses a variety of classification algorithms to detect human heat signatures within the raw thermal and motion data it collects. Thermosense achieves Root Mean Squared Error ≈ 0.35 persons, meaning the standard deviation between Thermosense's occupancy predictions and the actual occupancy number was $\approx$

0.35.

Another static trait is that of CO_2 emissions, which, like thermal emissions, are emitted by human occupants in both resting and active states. By measuring the buildup of CO_2 within a given area, one can use a variety of mathematical models of human CO_2 production to determine the likely number of occupants present. Hailemariam et al. [14] trialled this as part of a sensor fusion within the context of an office environment, achieving a $\approx 94\%$ accuracy. Such a sensing system could provide both the Presence and Count information, and exclude the Identity information as required. However, a CO_2 based detection mechanism has serious drawbacks, discussed by Fisk, Faulkner and Sullivan [10]: The CO_2 feedback mechanism is very slow, taking hours of continuous occupancy to correctly identify the presence of people. In a residential environment, occupants are more likely to be moving between rooms than an office, so the system may have a more difficult time detecting in that situation. Similarly, such systems can be interfered with by other elements that control the CO_2 buildup in a space, like air conditioners, open windows, etc. This is also much more of a concern in a residential environment compared to the studied office space, as the average residence can have numerous such confounding factors that cannot easily be controlled for.

Visual identification can be, achieved through the use of video or still-image cameras and advanced image processing algorithms. Video can be used in occupancy detection in several different ways, achieving different levels of accuracy and requiring different configurations. The first use of video, POEM, proposed by Erickson, Achleitner and Cerpa [9] is the use of video as a “optical turnstile”; the video system detects potential occupants and the direction they are moving in at each entrance and exit to an area, and uses that information to extrapolate the number of occupants within the turnstiled area; this system has up to a 94% accuracy. However, the main issue with such a system applied to a residential environment is the system assumes that there will be wide enough “turnstile areas”, corridors of a fairly large area that connect different sections of a building, to use as detection zones. While such corridors exist in office environments, they are less likely to exist in residential ones.

Another video sensor system is proposed by Serrano-Cuerda et al. [22], that uses ceiling-based cameras and advanced image processing algorithms to count the number of people in the captured area. This system achieves a specificity of $TP/(TP+FP) \approx 97\%$ and a sensitivity $TP/(TP+FN) \approx 96\%$ (TP = true positives, FP = false positives, FN = false negatives). Such a system could be successfully applied to the residential environment, as both it and the “optical turnstile” model provide Presence and Count information. However, these systems also allow Identity to be determined, and thus are perceived as privacy-invasive. This perception leads to adoption and acceptance issues, which work against the ideal system's goals.

2.1.2 Dynamic traits

Dynamic traits are usually products of human occupant activity, and thus can generally only be detected when a human occupant is physically active or in motion.

Ultrasonic systems, such as Doorjamb proposed by Hnat et al. [15], use clusters of such sensors above doorframes to detect the height and direction of potential occupants travelling between rooms. This acts as a turnstile based system, much like POEM [9], but augments this with an understanding of the model of the building to error correct for invalid and impossible movements brought about from sensing errors. This system provides an overall room-level tracking accuracy of 90%, however to achieve this accuracy, potential occupants are intended to be tracked using their heights, which has privacy implications. The system can also suffer from problems with error propagation, as there are possibilities of "phantom" occupants entering a room due to sensing errors.

Solely PIR based systems, like those used by Hailemariam et al. [14], involve the motion of the sensor being averaged over several different time intervals, and fed into a decision tree classifier. This PIR system alone produced a $\approx 98\%$ accuracy. However, such a system, due to only motion detection capabilities, can only provide Presence information, and is unable to provide Count information, nor detect motionless occupants.

2.2 Extrinsic traits

Extrinsic traits are those which are actually other environmental changes that are caused by or correlated with human occupant presence. These traits generally present a less accurate picture, or require the sensed occupants to be in some way "tagged", but they are generally also easier to sense in of themselves. The sensors in this category have been divided into two subcategories.

2.2.1 Instrumented traits

One extrinsic trait category is instrumented approaches; these require that detectable occupants carry with them some device that is detected as a proxy for the occupant themselves.

The most obvious of these approaches is a specially designed device. Li et al. [19] use RFID tags placed on building occupant's persons and a set of transmitters to triangulate the tags and place them within different thermal zones for the use of the HVAC system. For stationary occupants, there was a detection accuracy of $\approx$ 88%, and for occupants who were mobile, the accuracy was $\approx$ 62%. Such a system could be re-purposed for the residence, however, these systems raise issues in a residential environment as it requires occupants to be constantly carrying their sensors, which is less likely in such an environment. Additionally, the accuracy for this system is not necessarily high enough for a residential environment, where much smaller rooms are used.

To make extrinsic detection more reliable, Li, Calis and Becerik-Gerber [16] leverage a common consumer device; wifi enabled smart phones. They propose the *homeset* algorithm, which uses the phones to scan the visible wifi networks, and from that information estimate if the occupants are at home or out and about by "triangulating" their position from the visible wifi networks. This solution does not provide the fine-grained Presence data that we need, as it is only able to triangulate the phone's position very roughly with the wireless network detection information.

Balaji et al. [6] also leverage smart phones to determine occupancy, but in a more broad enterprise environment: Wireless association logs are analysed to determine which access points in a building a given occupant is connected to. If this access point falls within the radio range of their designated "personal space", they are considered to be occupying that personal space. This technique cannot be applied to a residential environment, as there are usually not multiple wireless hotspots.

Finally, Gupta, Intille and Larson [13] use specifically the GPS functions of the smartphone to perform optimisation on heating and cooling systems by calculating the "travel-to-home" time of occupants at all times and ensuring at every distance the house is minimally heated such that if the potential occupant were to travel home, the house would be at the correct temperature when they arrived. While this system does achieve similar potential air-conditioning energy savings, it is not room-level modular, and also presupposes an occupant whose primary energy costs are from incorrect heating when away from home, which isn't necessarily the case for this demographic.

2.2.2 Correlative traits

The second of these subcategories are correlative approaches. These approaches analyse data that is correlated with human occupant activity, but does not require a specific device to be present on each occupant that is tracked with the system.

The primary approach in this area is work done by Kleiminger et al. [17], which attempts to measure electricity consumption and use such data to determine Presence. Electricity data was measured at two different levels of granularity; the whole house level with a smart meter, and the consumption of specific appliances through smart plugs. This data was then processed by a variety of classifiers to achieve a classification accuracy of more than 80%. Such a system presents a low-cost solution to occupancy, however it is not sufficiently granular in either the detection of multiple occupants, or the detection of occupants in a specific room.

2.3 Analysis

From these various sensor options, there are a few candidates that provide the necessary quantitative criteria (Presence and Count); these are thermal, CO_2 , Video, Ultrasonic, RFID and WiFi association and triangulation based methods. All sensing options are compared on Table 2.1 on the next page.

In the context of our four qualitative accessibility criteria, CO_2 sensing has several reliability drawbacks, the predominant ones being a large lag time to receive accurate occupancy information and interference from a variety of air conditioning sources which can modify the CO_2 concentration in the room in unexpected ways.

Video-based sensing methods suffer from invasiveness concerns, as they by design must have a constant video feed of all detected areas.

Ultrasonic methods suffer from reliability concerns when a user falls outside the prescribed height bounds of normal humans. Wheelchair bound occupants, a core demographic of our proposed sensing system, are not discussed in the Doorjamb paper. Their wheelchair may also interfere with height measurement results. Ultrasonic methods also provide weak Identity information through height detection.

RFID sensing also has several drawbacks; it is difficult value proposition to get residential occupants to carry RFID tags with them continuously. Another drawback is that the triangulation methods discussed are too unreliable to place occupants in specific rooms in many cases.

	Requires		Excludes	Irrelevant	
	Presence	Count	Identity	Location	Track
<u>Intrinsic</u>					
Static					
Thermal	✓	✓	✓	✓	
CO_2	✓	✓	✓		
Video	✓	✓	✗	✓	✓
Dynamic					
Ultrasonic	✓	✓	✗		✓
PIR	✓	✗	✓		
<u>Extrinsic</u>					
Instrumented					
RFID	✓[1]	✓	✓	✓	
WiFi assoc.[2]	✓[1]	✓	✗	✓	
WiFi triang.[2]	✓[1]	✓	✗		
GPS[2]	✓[1]	✗	✓	✓	
Correlative					
Electricity	✓[1]	✗	✓		

[1]Doesn't provide data at required level of accuracy for home use.
[2]Uses smartphone as detector.

Table 2.1: Comparison of different sensors and project requirements

WiFi association is not granular enough for residential use, as the original enterprise use case presupposed a much larger area, as well as multiple wireless access points, neither of which a typical residential environment have.

WiFi triangulation is a good candidate for residential use, as there are most likely neighbouring wireless networks that can be used as virtual landmarks. However, it suffers from the same granularity problems as WiFi association, as these signals are not specific enough to pinpoint an occupant to a specific room.

For approaches presupposing smartphones being present on each occupant, it is more difficult to ensure that occupants are carrying their smartphones with them at all times in a residential environment. Another issue with smart phones is that they represent an expense that the target markets of the elderly and the disabled may not be able to afford.

Finally, we have thermal sensing. It provides both Presence and Count information, as it uses occupants' thermal signatures to determine the presence of people in a room. It does not however provide Identity information, as thermal signatures are not sufficiently unique with the technologies used to distinguished between occupants. Such a sensor system is presented as low-cost and energy efficient within Thermosense [7], is non-invasive by design and can reliably detect occupants with a very low root mean squared error. For our specific accessibility criteria, thermal sensing appears to be the best option available.

2.4 Thermal sensors

Our analysis (Subsection 2.3 on page 8) concluded that thermal sensors are the best candidates for this project. In this section we discuss the thermal sensing field in more detail.

A primary static/dynamic sensor fusion system in this field is the Thermosense system [7], a Passive Infrared Sensor (PIR) and Infrared Array Sensor (IAR) used to subdivide an area into an 8×8 grid of sections from which temperatures can be derived. This sensor system is attached to the roof on a small embedded controller which is responsible for collecting the data and transmitting it back to a larger computer via low powered wireless protocols.

The Thermosense system develops a thermal background map of the room using an Exponential Weighted Moving Average (EMWA) over a 15 minute time window (if no motion is detected). If the room remains occupied for a long period, a more complex scaling algorithm is used which considers the coldest points in the room empty, and averages them against the new background, then performs EMWA with a lower weighting.

This background map is used as a baseline to calculate standard deviations of each grid area, which are then used to determine several characteristics to be used as feature vectors for a variety of classification approaches. The determination of the feature vectors was subject to experimentation, since the differences at each grid element too susceptible to individual room conditions to be used as feature vectors. Instead, a set of three different features was designed; the number of temperature anomalies in the space, the number of groups of temperature anomalies, and the size of the largest anomaly in the space. These feature vectors were compared against three classification approaches; K-Nearest Neighbors, Linear Regression and an a feed-forward Artificial Neural Network of one hidden later and 5 perceptions. All three classifiers achieved a Root Mean Squared Error (RMSE) within 0.38 ± 0.04. This final classification is subject to a final averaging process over a 4 minute window to remove the presence of independent errors from the raw classification data.

The Thermosense approach presents the state of the art in the field of sensing with IAR technology. Using a similar IAR system along with those types of classification algorithms should yield useful sensing results which can be then integrated into the broader sensor system.

2.5 Research Gap

Throughout this review of the area of wireless occupancy sensors within the Internet of Things (IoT) it can be seen that there is a clear research gap within the area of occupancy. No group could be found who has assembled an occupancy sensor that optimises these ares of Low Cost, Non-Invasiveness, Energy Efficiency and Reliability into a architected software and hardware package that can be integrated like any other Thing into the IoT.

This is a key research area, because, as we have previously mentioned, the true "disruptive level of innovation"[4] the IoT provides can only be realised once a novel idea has been properly packaged as a Thing, rather than as a research curiosity. Packaging something as a Thing requires careful consideration of the best sensing systems, the best hardware to run those systems on, the best protocols to allow these Things to communicate, and the best device architecture to enable that communication. The state of the art in all these areas have been discussed throughout this literature review.

2.6 Conclusion

Several criteria were identified through which the spectrum of occupancy sensing could be examined; a quantitative criteria by Teixeira, Dublon and Savvides [24] to examine the different functionality offerings of sensor systems and a qualitative criteria derived from the aims of the project to examine how those sensors fit within the project's parameters.

Occupancy research performed with different sensor types was examined methodically through a set of taxonomic categories also originally proposed by Teixeira, Dublon and Savvides [24], but modified to better suit the specifics of occupancy sensors. These sensor types included Thermal, CO_2 , Video, Ultrasonic, Passive Infrared Sensor (PIR), RFID, various WiFi based methods, GPS and electricity consumption. Through an examination of these sensing systems quantitative and qualitative characteristics, it was determined that the Thermosense Infrared Array Sensor (IAR) system [7] was the most suitable to the project's aims.

A key part of enabling the "smart home for the disabled" is creating a set of Things that can improve quality of life for those people. We believe our proposed Thing has clearly demonstrated this potential.

CHAPTER 3

Prototype Design

As discussed in the Literature Review, using an Infrared Array Sensor (IAR) appear to be the most viable way to achieve the high-level goals of this project. Thermosense [7], the primary occupancy sensor in the IAR space, used the low-cost Panasonic Grid-EYE sensor for this task. This sensor, costing around $30USD, appears to be a prime candidate for use in this project, as it satisfied low-cost criteria, as well as being proven by Thermosense to be effective in this space. However, while still available for sale in the United States, we were unable to order the sensor for shipping to Australia due to export restrictions outside of our control. While such restrictions would be circumventable with sufficient effort, using a sensor with such restrictions in place goes against an implicit criteria of the parts used in the project being relatively easy to acquire.

This forced us to search for alternative sensors in the space that fulfill similar criteria but were more broadly available. The sensor we settled on was the Melexis MLX90620 (*Melexis*) [20], an IAR with similar overall qualities that differed in several important ways; it provides a 16×4 grid of thermal information, it has an overall narrower field of view and it sells for approximately $80USD. Like the Grid-EYE , the *Melexis* sensor communicates over the 2-wire I^2C bus, a low-level bi-directional communication bus widely used and supported in embedded systems.

In an idealized version of this occupancy system, much like Thermosense this system would include wireless networking and a very small form factor. However, due to time and resource constraints, the scope of this project has been limited to a minimum viable implementation. Appendix Chapter B on page 44 discusses in detail how the introduction of new open standards in the Wireless Personal Area Network space could be used in future systems to provide robust, decentralized networking of future occupancy sensors. This prototype architecture has been designed such that a clear path to the idea system architecture discussed therein is available.

Analysis Tier	Raspberry Pi B+
Preprocessing Tier	Arduino Uno R3
Sensing Tier	Melexis MLX90620 & PIR

Table 3.1: Hardware tiers

3.1 Hardware

As reliability and future extensibility are core concerns of the project, a three-tiered system is employed with regards to the hardware involved in the system (Table 3.1). At the bottom, the Sensing Tier, we have the raw sensor, the Melexis MLX90620 (*Melexis*), which communicate over I^2C . Connected to these devices via those respective protocols is the Preprocessing Tier, run an embedded system. The embedded device polls the data from these sensors, performs necessary calculations to turn raw information into suitable data, and communicates this via Serial over USB to the third tier. The third tier, the Analysis Tier, is run on a fully fledged computer. In our prototype, it captures and stores both video data, and the Temperature and Motion data it receives over Serial over USB.

While at a glance this system may seem overly complicated, it ensures that a sensible upgrade path to a more feature-rich sensing system is available. In the current prototype, the Analysis Tier merely stores captured data for offline analysis, in future prototypes this analysis can be done live and served to interested parties over a RESTful API. In the current prototype, the Analysis and Sensing Tiers are connected by Serial over USB, in future prototypes, this can be replaced by a wireless mesh network, with many Preprocessing/Sensing Tier nodes communicating with one Analysis Tier node.

Due to low cost and ease of use, the Arduino platform was selected as the host for the Preprocessing Tier, and thus the low-level I^2C interface for communication to the *Melexis*. Initially, this presented some challenges, as the *Melexis* recommends a power and communication voltage of 2.6V, while the Arduino is only able to output 3.3V and 5V as power, and 5V as communication. Due to this, it was not possible to directly connect the Arduino to the *Melexis*, and similarly due to the two-way nature of the I^2C 2-wire communication protocol, it was also not possible to simply lower the Arduino voltage using simple electrical techniques, as such techniques would interfere with two-way communication.

A solution was found in the form of a I^2C level-shifter, the Adafruit "4-channel I2C-safe Bi-directional Logic Level Converter" [1], which provided a cheap method to bi-directionally communicate between the two devices at their own preferred voltages. The layout of the circuit necessary to link the Arduino

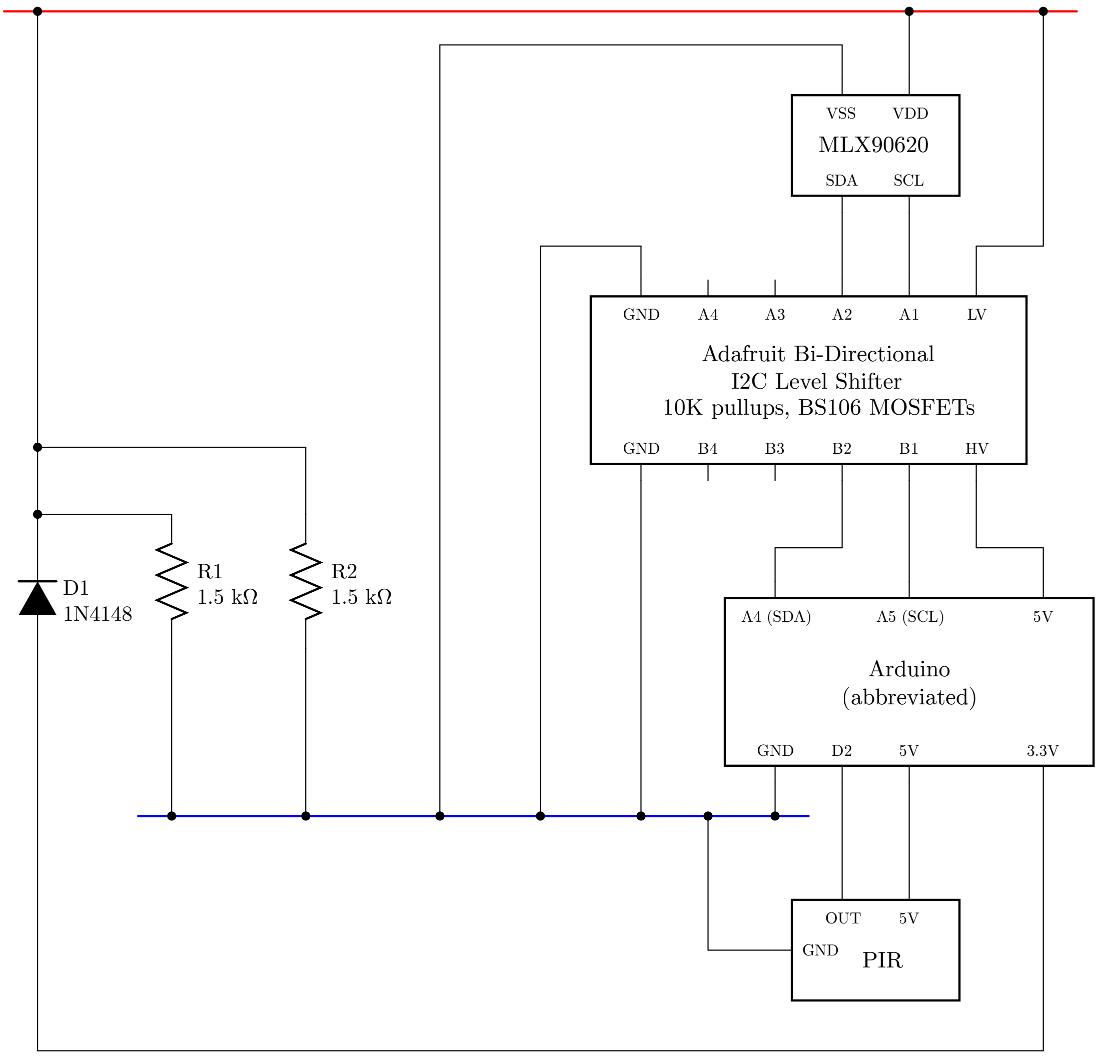


Figure 3.1: MLX90620, PIR and Arduino integration circuit

and the *Melexis* using this converter can be seen in Figure 3.1 on the previous page.

Additionally, as used in the Thermosense paper, a Passive Infrared Sensor (PIR) motion sensor [2] was also connected to the Arduino . This sensor, operating at 5V natively, did not require any complex circuitry to interface with the Arduino . It is connected to digital pin 2 on the Arduino , where it provides a rising signal in the event that motion is detected, which can be configured to cause an interrupt on the Arduino . In the configuration used in this project, the sensor's sensitivity was set to the highest value and the timeout for re-triggering was set to the lowest value (approximately 2.5 seconds). Additionally, the continuous re-triggering feature (whereby the sensor produces continuous rising and falling signals for the duration of motion) was disabled using the provided jumpers.

For the Analysis Tier, the Raspberry Pi B+ was chosen, as it is a powerful computer capable of running Linux available for an extraordinarily low price. The Arduino is connected to the Raspberry Pi over USB, which provides it both power and the capacity to transfer data. In turn, the Raspberry Pi is connected to a simple micro-USB rechargeable battery pack, which provides it with power, and subsequently the Arduino and sensors.

3.2 Software

At each layer of the described three-tier software architecture (pictured in greater detail in Figure 3.2 on page 18), there must exist software which governs the operation of that tier's processing concerns. Software in this project was written in two different languages.

At the Sensing Tier, it was not necessary for any software to be developed, as any software necessary came pre-installed and ready for use on the aforementioned sensors.

At the Preprocessing Tier, the Arduino, the default C++ derivative language was used, as careful management of memory usage and algorithmic complexity is required in such a resource-constrained environment, thus limiting choice in the area.

Finally, at Analysis Tier, a computer running fully-fledged Linux, choice of language becomes a possibility. In this instance, Python was settled on as the language of choice, as it is a quite high-level language with excellent library support for the functions required of the Analysis Tier, including serial interface, the use of the Raspberry Pi's built in camera, and image analysis. The 2.x branch of Python was chosen over the 3.x branch, despite its age, due a greater maturity in support for several key graphical interface libraries.

3.2.1 Pre-processing: `mlx90620_driver.ino`

On the Arduino, once large program was developed, termed `mlx90620_driver.ino`. This program's purpose was to take simple commands over serial to configure the Melexis MLX90620 (*Melexis*) and to report back the current temperature values and Passive Infrared Sensor (PIR) motion information at either a pre-set interval, or when requested.

To calculate the final temperature values that the *Melexis* offers, a complex initialization and computational process must be followed, which is specified in the sensor's datasheet [20]. This process involves initializing the sensor with values attained from a separate on-board I^2C EEPROM, then retrieving a variety of normalization and adjustment values, along with the raw sensor data, to compute the final temperature result.

The basic algorithm to perform this normalization was based upon the provided datasheet [20], as well as code by users "maxbot", "IIBaboomba", "nseidle" and others on the Arduino Forums [3] and was modified to operate with the newer Arduino "Wire" I^2C libraries released since the authors' posts. In pursuit of the project's aims to create a more approachable thermal sensor, the code was also restructured and rewritten to be both more readable, and to introduce a set of features to make the management of the sensor data easier for the user, and for the information to be more human readable.

Additionally, support for the PIR's motion data was added to the code, with the PIR configured to perform interrupts on one of the Arduino's digital pinsnd the code structured to take note of this information and to report it to the user in the "MOTION" section of the next packet.

The first of the features introduced was the human-readable format for serial transmission. This allows the user to both easily write code that can parse the serial to acquire the serial data, as well as examine the serial data directly with ease. When the Arduino first boots running the software, the output in Figure 3.3 on page 19 is output. This specifies several things that are useful to the user; the attached sensor ("DRIVER"), the build of the software ("BUILD") and the refresh rate of the sensor ("IRHZ"). Several different headers, such as "ACTIVE" and "INIT" specify the current millisecond time of the processor, thus indicating how long the execution of the initialization process took (33 milliseconds).

Once booted, the user is able to send several one-character commands to

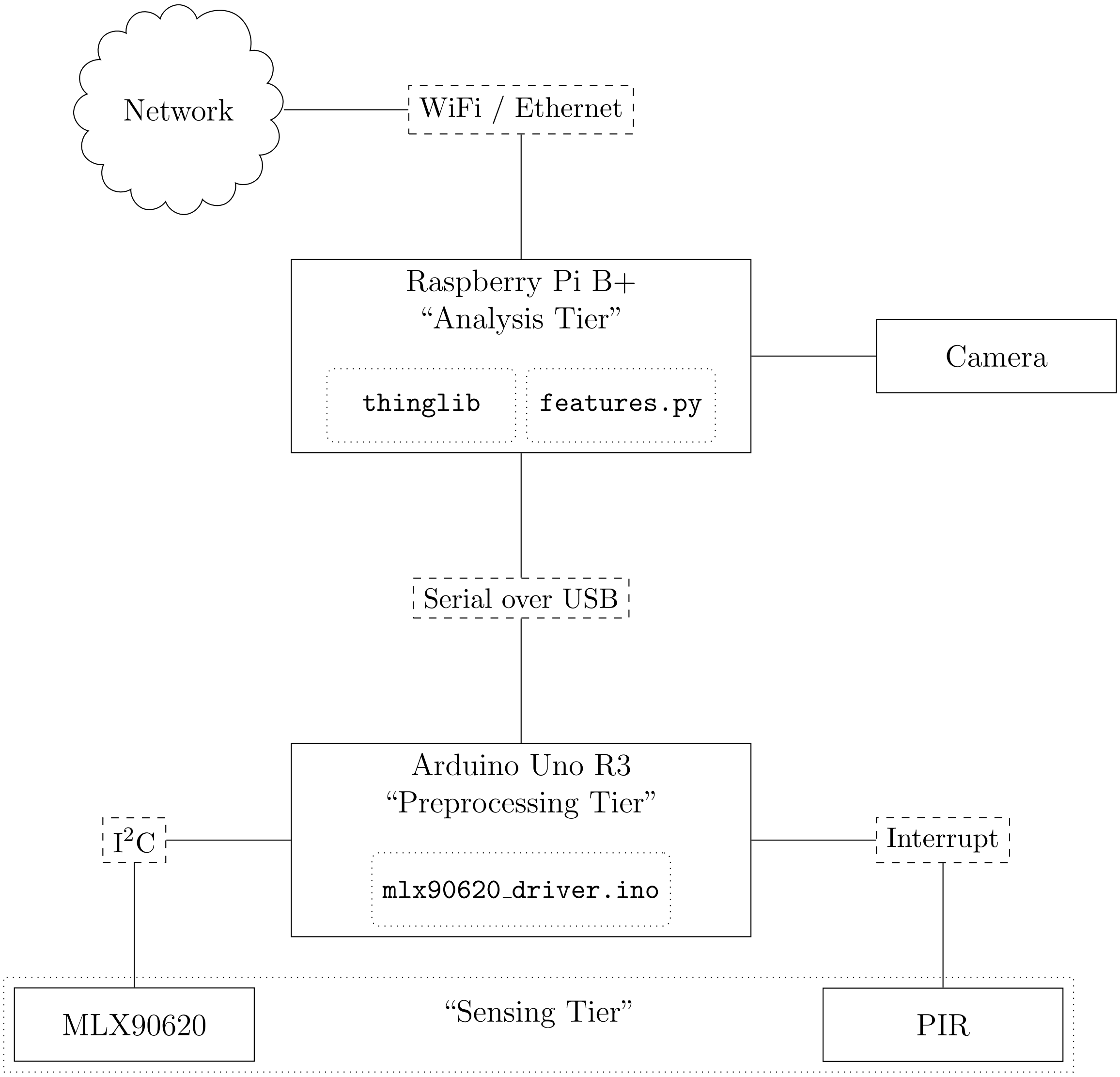


Figure 3.2: Prototype system architecture

```
INIT 0
INFO START
DRIVER MLX90620
BUILD Feb  1 2015 00:00:00
IRHZ 1
INFO STOP
ACTIVE 33

START 34
MOVEMENT 0
1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
STOP 97
```

Figure 3.3: Initialisation sequence and thermal packet

the sensor to configure operation, which are described in Table **??** on page ??. Depending on the sensor configuration, IR data may be periodically output automatically, or otherwise manually triggered. This IR data is produced in the packet format described in Figure 3.3. This is a simple, human readable format that includes the millisecond time of the processor at the start and end of the calculation, if the PIR has seen any motion for the duration of the calculation, and the 16x4 grid of calculated temperature values.

3.2.2 Analysis: `thinglib`

On the analysis tier a set of Python libraries and accompanying utility scripts were developed to interface with the Arduino, parse and interpret its data, and to provide data logging and visualization capabilities. Most of this functionality was split into a reusable and versatile Python module called `thinglib`.

`thinglib` provides 4 main feature sets across 3 files; the `Manager` series of classes, the `Visualizer` class, the `Features` class and the `pxdisplay` module.

3.2.2.1 `Manager` classes

The Manager series of classes are the direct interface between the Arduino and the Python classes. They implement a multi-threaded serial data collection and parsing system which converts the raw serial output of the connected Arduino into a series of Python data structures that represent the collected temperature and motion data of each captured frame. Several different versions of the `Manager` class exist to perform slightly different functions. When initializing these classes the sample rate of the *Melexis* can be configured, and it will be sent through to the Arduino for updating.

`BaseManager` is responsible for the implementation of the core serial parsing functions. It also provides a threaded interface through which the *Melexis*'s continuous stream of data can be subscribed to by other threads. The primary API, the `subscribe_` series of functions, return a thread-safe queue structure, through which thermal packets can be received by various other threads when they become available.

`Manager`, the primary class, provides access the *Melexis*'s data at configurable intervals. When initializing this class, you may specific 0.5, 1, 2, 4 or 8Hz, and the class will configure the Arduino to both set the *Melexis* to this sample rate, and to automatically write this data to the serial buffer whenever it is available. This serial interface is multi-threaded, as at higher serial baud rates if data was not polled continuously enough the internal serial buffer would fill and some data would be discarded. By ensuring this process cannot be blocked by other parts of the running program this problem is mostly eliminated.

`OnDemandManager` operates in a similar way to `Manager`, however instead of using a non-blocking threaded approach, the user's scripts may request thermal/motion data from the class, and it will poll the Arduino for information and block until this information is parsed and returned.

Finally, `ManagerPlaybackEmulator` is a simple class which can take a previously created thermal recording from a file, and emulate the `Manager` class by providing access to thread-safe queues which return this data at the specified Hz rate. This class can be used as a means to playback thermal recordings with the same visualization functions.

3.2.2.2 `pxdisplay` functions

The `pxdisplay` module is a set of functions that utilize the `pygame` library to create a simple live-updating window containing a thermal map representation of the thermal data. One can generate any number of `pxdisplay` objects, which leverage the `multithreading` library and `multithreading.Queue` to allow thermal data to be sent to the display.

The class also provides a set of functions to set a "hotest" and "coldest" temperature and have RGB colors assigned from red to blue for each temperature that falls between those two extremes.

3.2.2.3 `Visualizer` class

The `Visualizer` class is the natural compliment to the `Manager` series of classes. The functions contained within can usually be provided with a Queue object (generated by a `Manager` class) and can perform a variety of visualization and storage functions.

From the recording side, the `Visualizer` class can "record" a thermal capture by saving the motion and thermal information to a simple `.tcap` file, which stores the sample rate, timings, thermal and motion data from a capture in a very straightforward format. The class can also read these files back into the data structures `Visualizer` uses internally to store data. If `Visualizer` is running on a Raspberry Pi, it can also leverage the `picamera` library an the `OnDemandManager` class to synchronously capture both visual and thermal data for ground truth purposes.

From the visualization side, `Visualizer` can leverage the `pxdisplay` module to create thermal maps that can update in real-time based on the thermal data provided by a `Manager` class. The class can also generate both images and movie files from thermal recordings using the PIL and ffmpeg libraries respectively.

3.2.2.4 `Features` class

In Thermosense [7], an algorithm was demonstrated that allowed the separation of "background" information from "active" pixels, and from that information, the extraction of the features necessary for a classifier to correctly determine the number of people in an 8×8 thermal image. This algorithm involved calculating the average and standard deviations of each pixel while it is guaranteed that the image would be empty, and then when motion is detected, considering any pixel "active" that reaches a value more than 3 standard deviations above the pixel when there was no motion.

From these "active" pixels, it was established that a set of three feature vectors were all that were required to correctly classify the number of people in the thermal image. These feature vectors were;

1. **Number of active pixels**: The total number of pixels that are considered "active" in a given frame

2. **Number of connected components**: If each active pixel is represented as an node in an undirected graph where adjacent active pixels are connected, how many connected components does this graph have?

3. **Size of largest connected component**: The number of active pixels contained within the largest connected component

In accordance with the pseudo-code outlined in the Thermosense paper, the algorithm described in Figure 1 on the following page was created to extract these figures. The portion of this code dealing with scaling the thermal background for rooms without motion was not implemented, as in all experiments tested, there exists a significant interval of time during which the no motion is guarenteed and the thermal background can be generated. The `networkx` library was used to generate the connected components information.

```python
import networkx, itertools

nomotion_wgt = 0.01
n_rows = 4
n_cols = 16
background = first_frame
means = first_frame
stds = [ [0]*16 ]*4
stds_post = [ [None]*16 ]*4

def create_features(new_frame, is_motion):
  active = []
  g = networkx.Graph()

  for i, j in itertools.product( range(n_rows), range(n_cols) ):
    prev = background[i][j]
    cur = new_frame[i][j]
    cur_mean = means[i][j]
    cur_std = stds[i][j]

    if not is_motion:
      background[i][j] = nomotion_wgt * cur + (1 - nomotion_wgt) * prev
      means[i][j]      = cur_mean + (cur - cur_mean) / n
      stds[i][j]       = cur_std + (cur - cur_mean) * (cur - means[i][j])
      stds_post[i][j]  = math.sqrt(stds[i][j] / (n-1))

    if (cur - background[i][j]) > (3 * stds_post[i][j]):
      active.append((i,j))
      g.add_node((i,j))

      # Add edges for nodes that have already been computed as active
      for ix, jx in [(-1, -1), (-1, 0), (-1, 1), (0, -1)]:
        if (i+ix, j+jx) in active:
          g.add_edge((i,j), (i+ix,j+jx))

  comps = list(networkx.connected_components(g))
  num_active = len(active)
  num_connected = len(comps)
  size_connected = max(len(c) for c in comps) if len(comps) > 0 else None

  return (num_active, num_connected, size_connected)
```

Listing 1: Core feature extraction code

3.3 Sensor Properties

In order to best utilize the Melexis MLX90620 (*Melexis*), we must first understand the properties it exhibits, and their potential affects on our ability to perform person related measurements. These properties can be broadly separated into three different categories; bias, noise and sensitivity. A broad range of data was collected with the sensor in a horizontal orientation using various sources of heat and cold to determine these properties. This experimental setup is described in Figure 3.4 on the next page.

3.3.1 Bias

When receiving no infrared radiation, the sensor should indicate a near-zero temperature. If in such conditions it does not, that indicates that the sensor has some level of bias in its measurement values. We attempted to investigate this bias by performing thermal captures of the night sky. While this does not completely remove the infrared radiation, it does remove a significant proportion of it.

In Table 3.2 on page 26 the thermal sensor was exposed to the night sky at a capture rate of 1Hz for 4 minutes, with the sensing results combined to create a set of means and standard deviations to indicate the pixels at "rest". The average temperature detected was 11.78°C, with the standard deviation remaining less than 0.51°C over the entire exposure period. The resultant thermal map shows that pixels centered around the four "primary" pixels in the center maintain a similar temperature around 9°C, with temperatures beginning to deviate as they became further from the center.

The most likely cause of this bias is related to the physical structure of the sensor. The *Melexis* is a rectangular sensor which has been placed inside a circular tube. Due to this physical arrangement, the sides of this rectangular sensor will be significantly closer to these edges than the center. If these sides are at an ambient temperature higher than the measurement data (as they were in this case) thermal radiation from the sensor package itself could provide significant enough to cause the edges to appear warmer than the observed area of the sky. Such issues with temperature could be controlled for using a device that cools the sensor package to below that of the ambient temperature being measured, however, this is not a concern in this project, as the method of calculating a thermal background will compensate for any such bias as long as it remains constant.

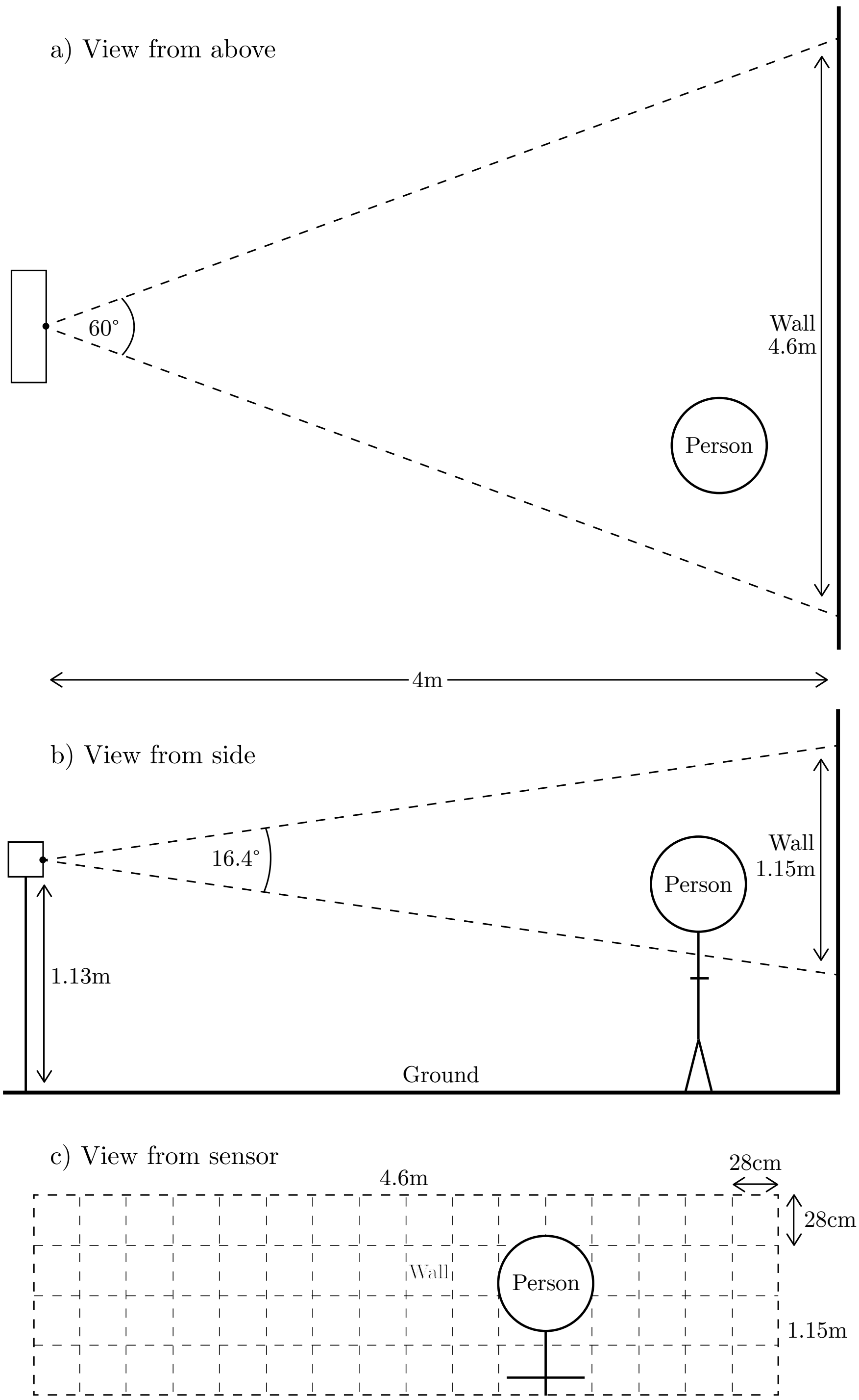


Figure 3.4: Experiment setup to determine sensor properties

14.95 0.51	14.33 0.27	12.34 0.27	8.77 0.33	8.15 0.31	10.84 0.38	9.02 0.26	7.79 0.37	6.67 0.27	9.63 0.29	9.29 0.26	8.24 0.27	9.84 0.25	14.28 0.33	14.92 0.3	13.16 0.25
14.54 0.34	15.62 0.31	12.73 0.23	11.51 0.27	11.79 0.26	11.47 0.27	11.43 0.29	9.02 0.35	8.57 0.23	11.15 0.23	10.64 0.22	10.3 0.24	12.09 0.22	14.49 0.26	14.88 0.31	14.71 0.36
18.25 0.45	16.62 0.31	14.15 0.24	11.97 0.34	13.11 0.3	12.64 0.22	10.66 0.23	9.15 0.24	9.58 0.28	11.95 0.28	11.22 0.24	11.52 0.36	11.11 0.23	12.59 0.25	14.44 0.31	13.35 0.28
16.02 0.28	16.81 0.36	15.0 0.25	11.53 0.28	10.18 0.29	12.2 0.25	11.78 0.29	8.36 0.31	8.15 0.33	10.36 0.32	10.74 0.31	8.25 0.36	9.99 0.35	12.42 0.38	11.39 0.4	11.06 0.34

Table 3.2: Mean and standard deviations for each pixel at rest

3.3.2 Noise

One of the features of the *Melexis* is the ability to sample the thermal data and a variety of sample rates between 0.5Hz and 512Hz. However, it was noted in early experimentation that a higher sample rate resulted in an increase in the noise contained within the resultant images. As our experiments focus on separating objects of interest from a thermal background, it is important to determine the maximum level of noise tolerable before our algorithms are unable to separate the background from the objects of interest.

Figure 3.5 on the next page plots one of the central pixels of the sensor in a scenario where it is merely viewing a background (shown in green), and when it is viewing a person (shown in red), at the 5 different sample rates achievable with the current hardware. We can see in these plots that the data becomes significantly more noisy as the sample rate increases, and we can also determine that the sensor uses a form of data smoothing at lower sample rates, as the variance in data increases with sample rate.

If the sample rate were to increase, it is likely that the ability for the sensing system to disambiguate between objects of interest and the background would diminish. However, in the current project, even the slowest sampling rate of 0.5Hz is sufficient, as occupancy estimations at a sub-second level present little additional value and would require significant reforms in the efficiency of the software used.

3.3.3 Sensitivity

The *Melexis* is a sensor composed of 64 independent non-contact digital thermopiles, which measure infrared radiation to determine the temperature of objects. While they are bundled in one package, Figure 3.6 on page 29 shows that they are in fact wholly independent sensors placed in a grid structure. This has important effects on the properties of the data that the *Melexis* produces.

Figure 3.7 on page 30 shows a graph of the temperatures of the top row of 16 pixels of the *Melexis* as a hot object is moved from left to right at an approximately similar speed. One of the most interesting phenomena in this graph is the apparent extreme variability of the detected temperature of the object as it moves "between" two different pixels; there is a noticeable drop in the objects detected temperature. Further analysis of each of the pixel's lines on the graph shows each pixel exhibiting a bell-curve like structure, with the detected temperature increasing from the baseline and peaking as the object enters the center of the pixel, and the detected temperature similarly decreasing

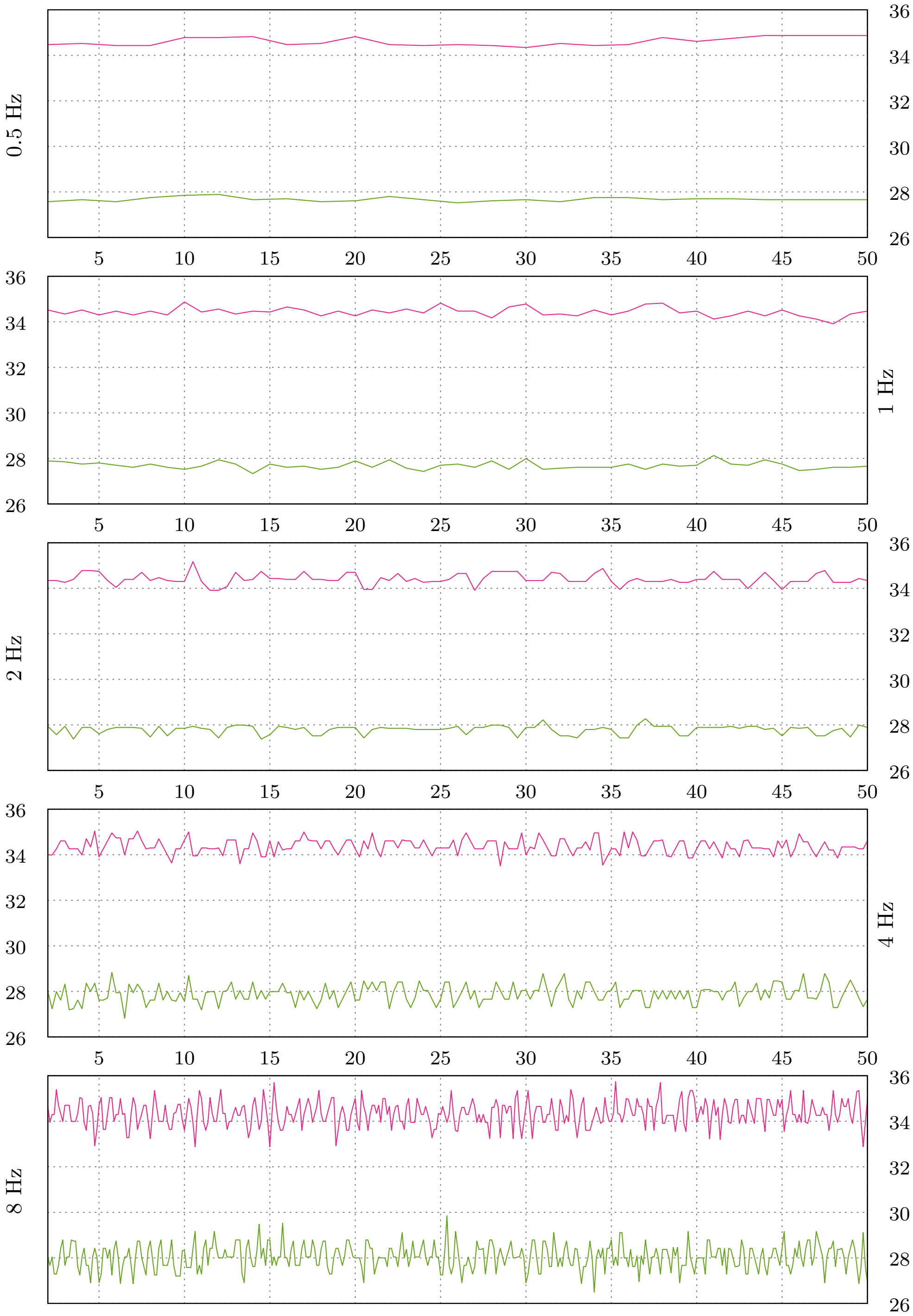


Figure 3.5: Comparison of noise levels at the *Melexis*' various sampling speeds

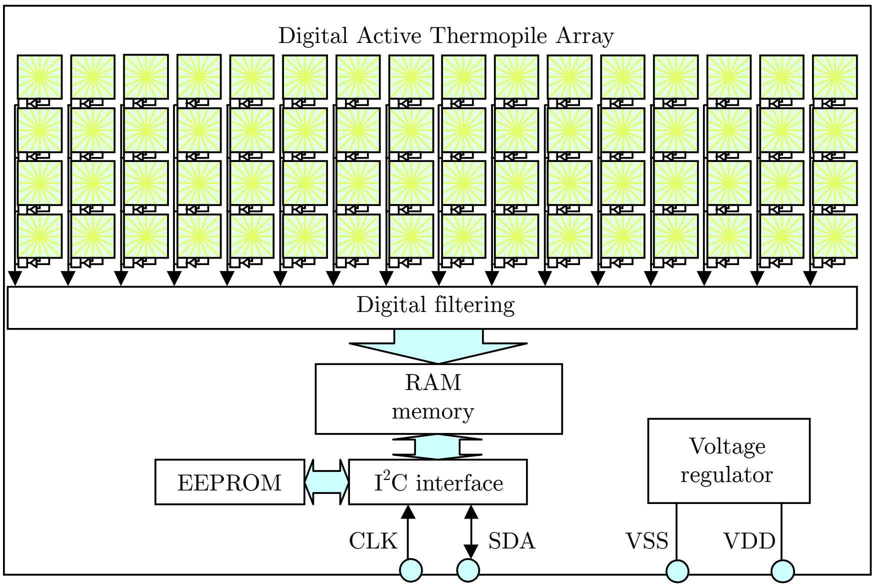


Figure 3.6: Block diagram for the *Melexis* taken from datasheet [20]

as the object leaves the center.

This phenomenon has several possible causes. One likely explanation is that each individual pixel detects objects radiating at less favorable angles of incidence to be colder than they actually are: As the object enters a pixel's effective field of view, it will radiate into the pixel at an angle that is at the edge of the pixel's ability to sense, with this angle slowing decreasing until the hot object is directly radiating into the pixel's sensor, causing a peak in the temperature reading. As the object leaves the individual elements field of view, the same happens in reverse.

While interesting, this phenomenon has little consequence to the effectiveness of the techniques used, as in experimental conditions the sensor will not be sufficiently distant that humans could be detected as single pixels. However, this phenomenon could be leveraged in future work to perform sub-pixel localization, discussed later on.

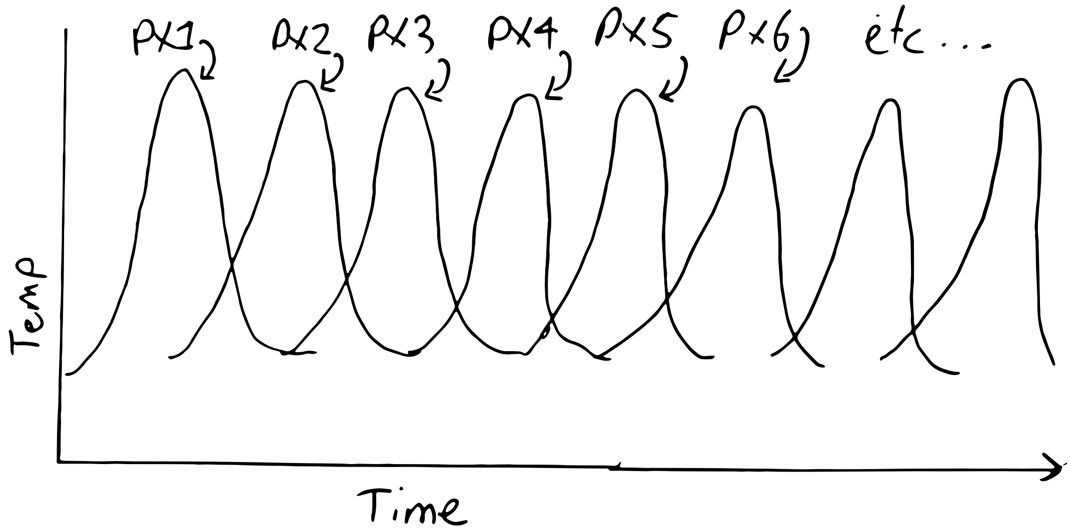


Figure 3.7: Different *Melexis* pixel temperature values as hot object moves across row

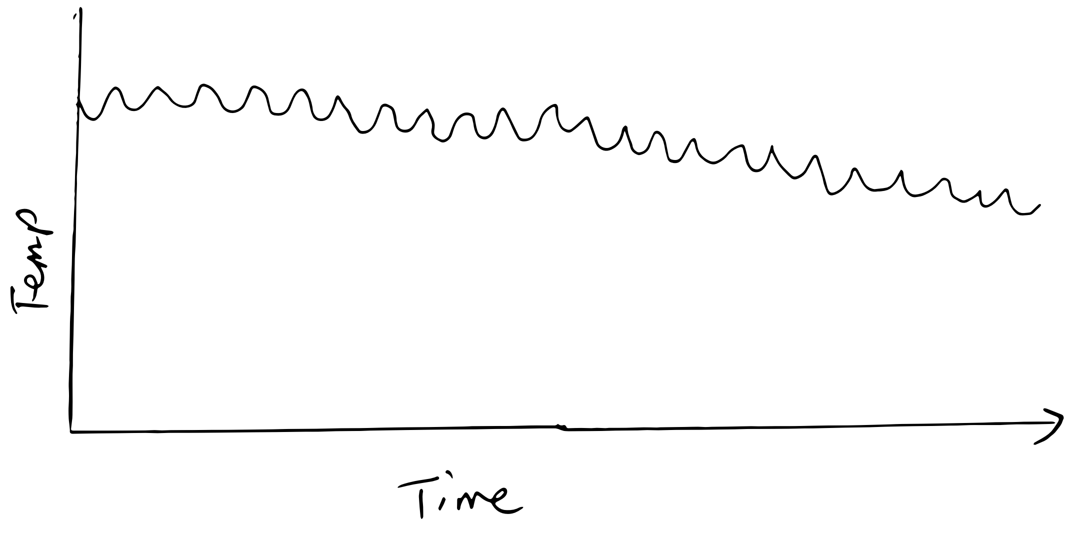


Figure 3.8: Variation in temperature detected for hot object at 1Hz sampling ration

CHAPTER 4

Methods

4.1 Classifier Experiment Set 1 Setup

The first experiment performed with the prototype described in the previous chapter was set out as indicated in Figure 4.1 on page 33. This experiment involved 3 people, who entered the scene in either standing or sitting positions. The following scripts were observed;

1. One person walks in, stands in center, walks out of frame
2. One person walks in, joined by another person, both stand there, one leaves, then another leaves
3. One person walks in, joined by 1, joined by another, all stand there, one leaves, then another, then another
4. Two people walk in, both stand there, both leave
5. (sitting) One person walks in, sits in center, moves to right, walks out of frame
6. (sitting) One person walks in, joined by another person, both sit there, switch chairs, one leaves, then another leaves
7. (sitting) One person walks in, joined by 1, joined by another, all sit there, one leaves, then another, then another
8. (sitting) Two people walk in, both sit there, both leave

These experiments were recorded with a thermal-visual synchronization at 1Hz over approximately 60 second intervals each. Each experiment had 10-15 seconds at the beginning where nothing was within the view of the sensor to allow the thermal background to be calculated. Each frame generated from these experiments was manually tagged with the ground truth value of its occupancy using the script mentioned previously. The resulting features and ground truth were exported to a format allowing the Weka machine learning program to analyze them. This data was analyzed with the feature vectors always being considered numeric data and with the ground truth considered both numeric and nominal (categories 0,1,2,3). The machine learning algorithms J48, Multilayer Perceptron, KStar, IBk, Naive Bayes and SMO were used for the nominal representation, and the algorithms Linear Regression, Multilayer Perceptron and IBk were used for the numeric representation.

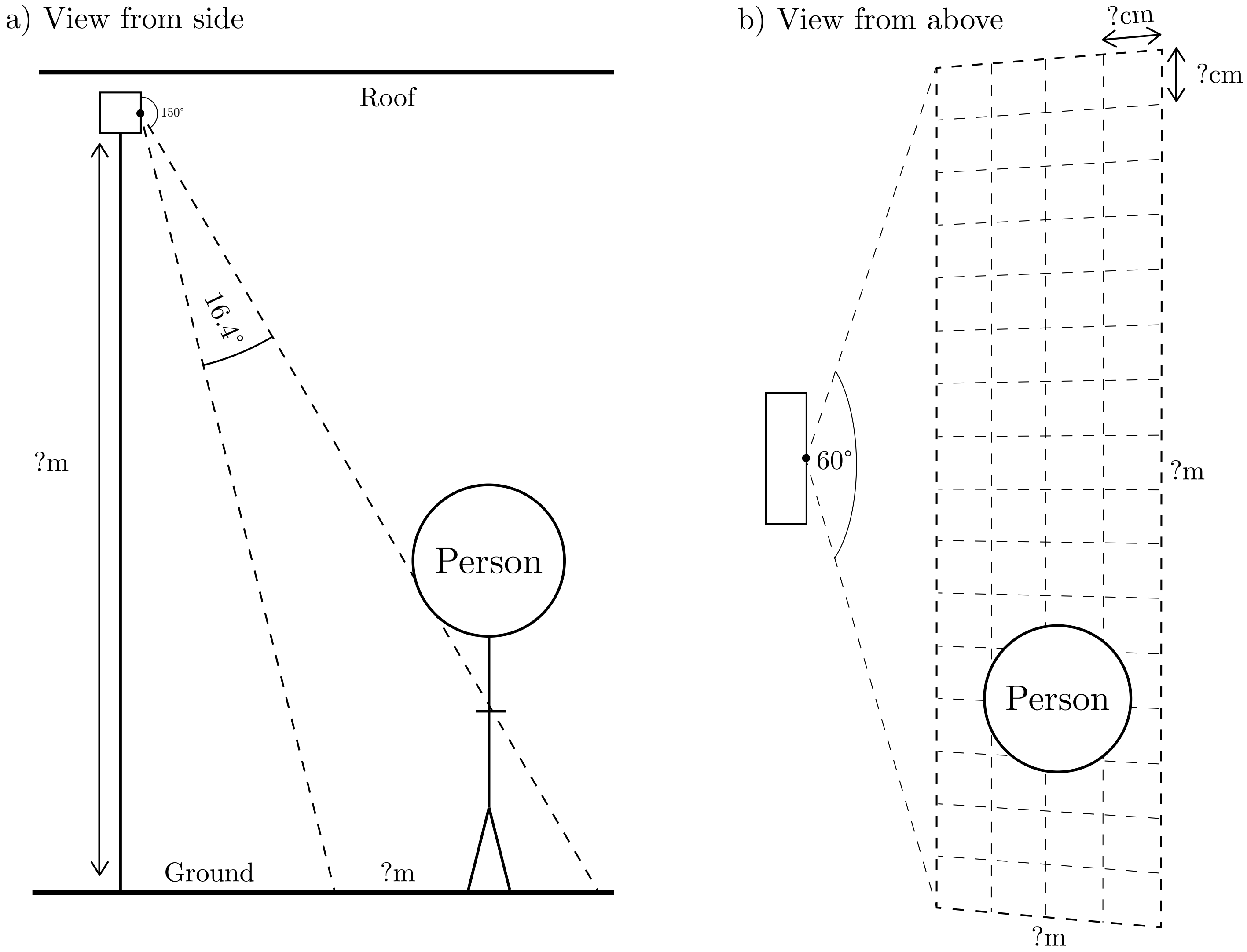


Figure 4.1: Classifier Experiment Set 1 Setup

CHAPTER 5

Results

5.1 Classifier Experiment Set 1

See Section D.1 on page 98.

Classifier	**Correctly Classified**	**RMS Error**	**F-Measure**
J48	82.9%	0.2878	0.824
KStar	82.6%	0.2853	0.818
MLP	78.5%	0.2936	0.777
Naive Bayes	66.2%	0.3516	0.644
IBk	57.6%	0.4245	0.563
SMO	57.5%	0.3795	0.554

Table 5.1: Classifier Experiment Set 1 Nominal Results

Numeric	**Correctly Classified**	**RMS Error**
Linear Regression	63.4%	0.6589
IBk	55.8%	1.1947
Multilayer Perceptron	50.2%	0.7768

Table 5.2: Classifier Experiment Set 1 Numeric Results

CHAPTER 6

Discussion and Conclusion

6.1 Future Directions

- Wireless mesh networking
- Convert into circuit board
- MLX90621
- Lenses
- Rotating the sensor to see wider FOV
- Subpixel localisation (see prev graphs bell curves)

APPENDIX A

Original Honours Proposal

Title:	Developing a robust system for occupancy detection in the household
Author:	Ash Tyndall
Supervisor:	Professor Rachel Cardell-Oliver
Degree:	BCompSci (24 point project)
Date:	October 8, 2014

A.1 Background

The proportion of elderly and mobility-impaired people is predicted to grow dramatically over the next century, leaving a large proportion of the population unable to care for themselves, and consequently less people able care for these groups. [6] With this issue looming, investments are being made into a variety of technologies that can provide the support these groups need to live independent of human assistance.

With recent advancements in low cost embedded computing, such as the Arduino [1] and Raspberry Pi, [2] the ability to provide a set of interconnected sensors, actuators and interfaces to enable a low-cost 'smart home for the disabled' is becoming increasingly achievable.

Sensing techniques to determine occupancy, the detection of the presence and number of people in an area, are of particular use to the elderly and disabled. Detection can be used to inform various devices that change state depending on the user's location, including the better regulation energy hungry devices to help reduce financial burden. Household climate control, which in some regions of Australia accounts for up to 40% of energy usage [3] is one particular area in which occupancy detection can reduce costs, as efficiency can be increased dramatically with annual energy savings of up to 25% found in some cases. [8]

Significant research has been performed into the occupancy field, with a focus on improving the energy efficiency of both office buildings and households. This is achieved through a variety of sensing means, including thermal arrays, [5] ultrasonic sensors, [11] smart phone tracking, [12][4] electricity consumption, [13] network traffic analysis, [15] sound, [10] CO2, [10] passive infrared, [10] video cameras, [7] and various fusions of the above. [16][15]

A.2 Aim

While many of the above solutions achieve excellent accuracies, in many cases they suffer from problems of installation logistics, difficult assembly, assumptions on user's technology ownership and component cost. In a smart home for the disabled, accuracy is important, but accessibility is paramount.

The goal of this research project is to devise an occupancy detection system that forms part of a larger 'smart home for the disabled' that meets the following accessibility criteria;

- *Low Cost*: The set of components required should aim to minimise cost, as these devices are intended to be deployed in situations where the serviced user may be financially restricted.
- *Non-Invasive*: The sensors used in the system should gather as little information as necessary to achieve the detection goal; there are privacy concerns with the use of high-definition sensors.
- *Energy Efficient*: The system may be placed in a location where there is no access to mains power (i.e. roof), and the retrofitting of appropriate power can be difficult; the ability to survive for long periods on only battery power is advantageous.
- *Reliable*: The system should be able to operate without user intervention or frequent maintenance, and should be able to perform its occupancy detection goal with a high degree of accuracy.

Success in this project would involve both

1. Devising a bill of materials that can be purchased off-the-shelf, assembled without difficulty, on which a software platform can be installed that performs analysis of the sensor data and provides a simple answer to the occupancy question, and

2. Using those materials and softwares to create a final demonstration prototype whose success can be tested in controlled and real-world conditions.

This system would be extensible, based on open standards such as REST or CoAP, [9][14] and could easily fit into a larger 'smart home for the disabled' or internet-of-things system.

A.3 Method

Achieving these aims involves performing research and development in several discrete phases.

A.3.1 Hardware

A list of possible sensor candidates will be developed, and these candidates will be ranked according to their adherence to the four accessibility criteria outlined above. Primarily the sensor ranking will consider the cost, invasiveness and reliability of detection, as the sensors themselves do not form a large part of the power requirement.

Similarly, a list of possible embedded boards to act as the sensor's host and data analysis platform will be created. Primarily, they will be ranked on cost, energy efficiency and reliability of programming/system stability.

Low-powered wireless protocols will also be investigated, to determine which is most suitable for the device; providing enough range at low power consumption to allow easy and reliable communication with the hardware.

Once promising candidates have been identified, components will be purchased and analysed to determine how well they can integrate.

A.3.2 Classification

Depending on the final sensor choice, relevant experiments will be performed to determine the classification algorithm with the best occupancy determination accuracy. This will involve the deployment of a prototype to perform data gathering, as well as another device/person to assess ground truth.

A.3.3 Robustness / API

Once the classification algorithm and hardware are finalised, an easy to use API will be developed to allow the data the device collects to be integrated into a broader system.

The finalised product will be architected into a easy-to-install software solution that will allow someone without domain knowledge to use the software and corresponding hardware in their own environment.

A.4 Timeline

Date	Task
Fri 15 August	*Project proposal and project summary due to Coordinator*
August	Hardware shortlisting / testing
25–29 August	*Project proposal talk presented to research group*
September	Literature review
Fri 19 September	*Draft literature review due to supervisor(s)*
October - November	Core Hardware / Software development
Fri 24 October	*Literature Review and Revised Project Proposal due to Coordinator*
November - February	*End of year break*
February	Write dissertation
Thu 16 April	*Draft dissertation due to supervisor*
April - May	Improve robustness and API
Thu 30 April	*Draft dissertation available for collection from supervisor*
Fri 8 May	*Seminar title and abstract due to Coordinator*
Mon 25 May	*Final dissertation due to Coordinator*
25–29 May	*Seminar Presented to Seminar Marking Panel*
Thu 28 May	*Poster Due*
Mon 22 June	*Corrected Dissertation Due to Coordinator*

A.5 Software and Hardware Requirements

A large part of this research project is determining the specific hardware and software that best fit the accessibility criteria. Because of this, an exhaustive list of software and hardware requirements are not given in this proposal.

A budget of up to $300 has been allocated by my supervisor for project purchases. Some technologies with promise that will be investigated include;

Raspberry Pi Model B+ Small form-factor Linux computer
Available from `http://arduino.cc/en/Guide/Introduction`; $38

Arduino Uno Small form-factor microcontroller
Available from `http://arduino.cc/en/Main/arduinoBoardUno`; $36

Panasonic Grid-EYE Infrared Array Sensor
Available from `http://www3.panasonic.biz/ac/e/control/sensor/infrared/grid-eye/index.jsp`; approx. $33

Passive Infrared Sensor
Available from various places; $10–$20

A.6 Proposal References

[1] Ardunio. `http://arduino.cc/en/Guide/Introduction`. Accessed: 2014-08-09.

[2] Raspberry pi. `http://www.raspberrypi.org/`. Accessed: 2014-08-09.

[3] Australian Bureau of Statistics. 4602.2 - household water and energy use, victoria: Heating and cooling. Tech. rep., October 2011.

[4] Balaji, B., Xu, J., Nwokafor, A., Gupta, R., and Agarwal, Y. Sentinel: occupancy based hvac actuation using existing wifi infrastructure within commercial buildings. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems* (2013), ACM, p. 17.

[5] Beltran, A., Erickson, V. L., and Cerpa, A. E. Thermosense: Occupancy thermal based sensing for hvac control. In *Proceedings of the 5th ACM Workshop on Embedded Systems For Energy-Efficient Buildings* (2013), ACM, pp. 1–8.

[6] Chan, M., Campo, E., Estève, D., and Fourniols, J.-Y. Smart homescurrent features and future perspectives. *Maturitas 64*, 2 (2009), 90–97.

[7] Erickson, V. L., Achleitner, S., and Cerpa, A. E. Poem: Power-efficient occupancy-based energy management system. In *Proceedings of the 12th international conference on Information processing in sensor networks* (2013), ACM, pp. 203–216.

[8] Erickson, V. L., Beltran, A., Winkler, D. A., Esfahani, N. P., Lusby, J. R., and Cerpa, A. E. Thermosense: thermal array sensor networks in building management. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems* (2013), ACM, p. 87.

[9] Guinard, D., Ion, I., and Mayer, S. In search of an internet of things service architecture: Rest or ws-*? a developers perspective. In *Mobile and Ubiquitous Systems: Computing, Networking, and Services*. Springer, 2012, pp. 326–337.

[10] Hailemariam, E., Goldstein, R., Attar, R., and Khan, A. Real-time occupancy detection using decision trees with multiple sensor types. In *Proceedings of the 2011 Symposium on Simulation for Architecture and Urban Design* (2011), Society for Computer Simulation International, pp. 141–148.

[11] Hnat, T. W., Griffiths, E., Dawson, R., and Whitehouse, K. Doorjamb: unobtrusive room-level tracking of people in homes using doorway sensors. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems* (2012), ACM, pp. 309–322.

[12] Kleiminger, W., Beckel, C., Dey, A., and Santini, S. Using unlabeled wi-fi scan data to discover occupancy patterns of private households. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems* (2013), ACM, p. 47.

[13] Kleiminger, W., Beckel, C., Staake, T., and Santini, S. Occupancy detection from electricity consumption data. In *Proceedings of the 5th ACM Workshop on Embedded Systems For Energy-Efficient Buildings* (2013), ACM, pp. 1–8.

[14] Kovatsch, M. Coap for the web of things: from tiny resource-constrained devices to the web browser. In *Proceedings of the 2013 ACM conference on Pervasive and ubiquitous computing adjunct publication* (2013), ACM, pp. 1495–1504.

[15] Ting, K., Yu, R., and Srivastava, M. Occupancy inferencing from non-intrusive data sources. In *Proceedings of the 5th ACM Workshop on Embedded Systems For Energy-Efficient Buildings* (2013), ACM, pp. 1–2.

[16] Yang, Z., Li, N., Becerik-Gerber, B., and Orosz, M. A multi-sensor based occupancy estimation model for supporting demand driven hvac operations. In *Proceedings of the 2012 Symposium on Simulation for Architecture and Urban Design* (2012), Society for Computer Simulation International, p. 2.

APPENDIX B

Ideal System Architecture

Beyond specific sensor design and occupancy detection algorithms, a core goal of this project is to create a system that is designed to operate as a useful Thing in a real-world Internet of Things (IoT) environment, as the key advantage of Things is the “disruptive level of innovation”[4] brought about by their ability to be combined in ways unforeseen (yet still enabled) by their creators. This architecture involves careful consideration of the embedded hardware that will drive the system, as well as the communications protocols utilised between the sensor and devices interested in the sensor’s information.

B.1 Protocols

In an ideal smart-home environment, the sensor systems used will communicate with each other wirelessly. As the complete sensor system has low power requirements to enable battery operation, it is important to prioritise those protocols and architectures that minimise power usage while still enabling the necessary wireless communication. The system will also ideally exist in a system with other identical sensors (one for each room in a residence), thus it is important to prioritise those protocols which allow multiple identical sensor systems to coexist on the same network without conflict, and to be uniquely addressable and iden-

REST	
Application	CoAP
Transport	UDP
IP / Routing	IETF RPL
Adaptation	IETF 6LoWPAN
Medium Access	IEEE 802.15.4e
Physical	IEEE 802.15.4-2006

Table B.1: Proposed protocol stack

tifiable. In recent years, many developments have been made in the Internet of Things (IoT) arena, with standards emerging specifically designed for low-power embedded devices to communicate between themselves and bigger systems that address these and other unique needs, across the entire protocol stack.

Palattella et al. [21] propose a protocol stack that aligns with the above requirements, with the key advantage being a wholly standardized implementation of the stack exists. This implementation is based on TCP/IP, uses the latest IEEE and IETF IoT standards, and is free from proprietary protocol restrictions (unlike ZigBee 1.0 devices, for instance). Table B.1 on the preceding page shows the full stack proposed. The key components of this proposal are the introduction of CoAP at the application layer, RPL at the IP / Routing layer and 6LoWPAN at the Adaptation layer.

Above the application layer, Guinard et al. [11] propose the use of Representational state transfer (REST) over Web Services Descriptive Language / Simple Object Access Protocol (WS-*) as a method of exchanging information between sensor systems. Their data suggests that REST is easier to use than WS-*, and the key advantage of a WS-* based approach is its ability to represent much more complex data and abstractions, which are unnecessary in this project's situation.

Constrained Application Protocol (CoAP) [18] is an application layer protocol designed to replace HTTP as a way of transmitting RESTful information between clients. The chief advantage of CoAP over HTTP is it compresses the broadstrokes of the HTTP feature set into a binary language that is much more suitable for transmission over low-bandwidth and low-power links, such as those discussed here.

IPv6 Routing Protocol for Low-Power and Lossy Networks (RPL) [25] is a routing protocol designed for low power environments, allowing low power nodes to create and maintain a mesh network between themselves, allowing, among other things, the routing of packets to a "root" node and back again. RPL is particularly suited to the routing situation of our proposed architecture, as individual sensors do not need to communicate with one another, but rather report back to a larger node (further discussed in Subsection B.2 on the next page).

IPv6 over Low power Wireless Personal Area Networks (6LoWPAN) [23] is a compression and formatting specification to allow IPv6 packets to be sent over an 802.15.4 based network. Optimisations are found in the reduction of the size of 6LoWPAN packets, IPv6 addresses as well as redesigning core Internet Protocol algorithms so that they can run with low power consumption on participating devices.

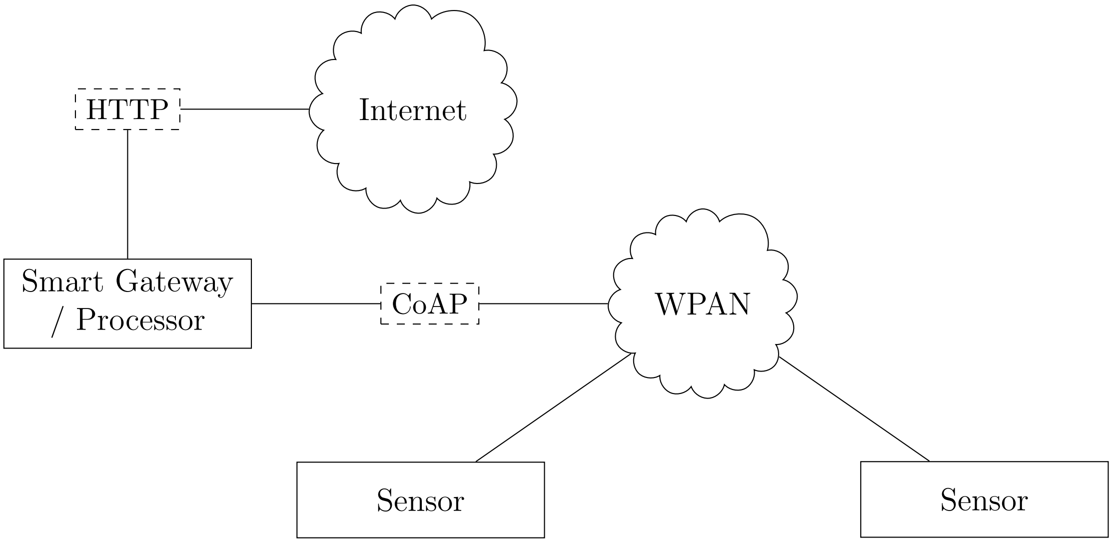


Figure B.1: Proposed system architecture

B.2 Devices

In addition to the protocol stack used, how these nodes relate to each other is also an important consideration. Part of what will inform these decisions are the requisite processing power and internet connectivity required to successfully execute all elements of the sensing system. Kovatsch [18] provides a constructive classification system to consider this, by describing three classes of resource constrained devices that would benefit from Constrained Application Protocol (CoAP), and each can provide different levels of security for an IP stack;

- *Class 0*: "not capable of running an RFC-compliant IP stack in a secure manner. They require application-level gateways to connect to the Internet."
- *Class 1*: Able to connect to the internet with some "integrated security mechanisms". Are unable to employ full HTTP with TLS.
- *Class 2*: Normal Internet nodes, able to use the full HTTP stack with TLS.

The devices that we propose the sensors will connect to are the likes of the Arduino, which can be classified as class 0 or possibly class 1 devices. Due to their insecurity and difficulty running a fully fledged IP stack, Guinard et al. [12] propose the use of a "Smart Gateway" system to bridge the wider internet and these sensor systems. This gateway would be able to communicate with the sensor systems over CoAP and 802.15.4 , as well as receive API requests via HTTP from a traditional TCP/IP network to forward on to these sensors.

The Thermosense paper [7] proposes several different algorithms to process the raw sensing data into the occupancy estimates (further discussed in Section 2.4 on page 10), all of which are fairly computationally expensive. Because of this, it would be non-trivial to implement these algorithms on the embedded sensing devices themselves. This problem is already resolved in our proposed system, as the aforementioned "Smart Gateway" can easily also take on the task of processing the raw sensor data into estimates which it can relay to interested parties over its HTTP-based API. A visualisation of this proposed system is shown in Figure B.1 on the previous page.

APPENDIX C

Code Listings

C.1 ThingLib

C.1.1 cam.py


```
from __future__ import division
from __future__ import print_function

import serial
import copy
import Queue as queue
import time
from collections import deque
import threading
import pygame
import colorsys
import datetime
from PIL import Image, ImageDraw, ImageFont
import subprocess
import tempfile
import os
```

```python
import os.path
import fractions
import pxdisplay
import multiprocessing
import numpy as np
import io



class BaseManager(object):
  driver = None
  build = None
  irhz = None

  tty = None
  baud = None

  hflip = True
  vflip = True

  _temps = None
  _serial_obj = None
  _queues = []

  def __init__(self, tty, hz=8, baud=115200, init=True):
    self.tty = tty
    self.baud = baud
    self.irhz = hz

    if init:
      self._serial_obj = serial.Serial(port=self.tty, baudrate=self.baud, rtscts=True, dsrdtr=True)

  def __del__(self):
    self.close()
```

```
def _reset_and_conf(self, timers=True):
  self._serial_obj.write('r\n') # Reset the sensor
  self._serial_obj.flush()

  time.sleep(2)

  if timers:
    self._serial_obj.write('t\n') # Turn on timers
  else:
    self._serial_obj.write('o\n') # Turn on timers

  self._serial_obj.flush()

def _decode_packet(self, packet, splitchar="\t"):
  decoded_packet = {}
  ir = []

  for line in packet:
    parted = line.partition(" ")
    cmd = parted[0]
    val = parted[2]

    try:
      if cmd == "START":
        decoded_packet['start_millis'] = long(val)
      elif cmd == "STOP":
        decoded_packet['stop_millis'] = long(val)
      elif cmd == "MOVEMENT":
        if val == "0":
          decoded_packet['movement'] = False
        elif val == "1":
          decoded_packet['movement'] = True
      else:
```

```
          ir.append(tuple(float(x) for x in line.split(splitchar)))
      except ValueError:
        print(packet)
        print("WARNING: Could not decode corrupted packet")
        return {}

    if self.hflip:
      ir = map(tuple, np.fliplr(ir))

    if self.vflip:
      ir = map(tuple, np.flipud(ir))

    decoded_packet['ir'] = tuple(ir)

    return decoded_packet

  def _decode_info(self, packet):
    decoded_packet = {}
    ir = []

    for line in packet:
      parted = line.partition(" ")
      cmd = parted[0]
      val = parted[2]

      if cmd == "INFO":
        pass
      elif cmd == "DRIVER":
        decoded_packet['driver'] = val
      elif cmd == "BUILD":
        decoded_packet['build'] = val
      elif cmd == "IRHZ":
        decoded_packet['irhz'] = int(val) if int(val) != 0 else 0.5

```

```
  return decoded_packet

def _update_info(self):
  ser = self._serial_obj

  ser.write('i')
  ser.flush()
  imsg = []

  line = ser.readline().decode("ascii", "ignore").strip()

  # Capture a whole packet
  while not line == "INFO START":
    line = ser.readline().decode("ascii", "ignore").strip()

  while not line == "INFO STOP":
    imsg.append(line)
    line = ser.readline().decode("ascii", "ignore").strip()

  imsg.append(line)

  packet = self._decode_info(imsg)

  self.driver = packet['driver']
  self.build = packet['build']

  if packet['irhz'] != self.irhz:
    ser.write('f{}'.format(self.irhz))
    self._update_info()

def _wait_read_packet(self):
  ser = self._serial_obj
  line = ser.readline().decode("ascii", "ignore").strip()
  msg = []
```

```
    # Capture a whole packet
    while not line.startswith("START"):
      line = ser.readline().decode("ascii", "ignore").strip()

    while not line.startswith("STOP"):
      msg.append(line)
      line = ser.readline().decode("ascii", "ignore").strip()

    msg.append(line)

    return msg

  def close(self):
    return

  def get_temps(self):
    if self._temps is None:
      return False
    else:
      return copy.deepcopy(self._temps)

  def subscribe(self):
    q = queue.Queue()
    self._queues.append(q)
    return q

  def subscribe_multiprocess(self):
    q = multiprocessing.Queue()
    self._queues.append(q)
    return q

  def subscribe_lifo(self):
    q = queue.LifoQueue()
```

```
    self._queues.append(q)
    return q



class Manager(BaseManager):
  _serial_thread = None
  _serial_stop = False
  _serial_ready = False

  _decode_thread = None

  _read_decode_queue = None

  def __init__(self, tty, hz=8, baud=115200):
    super(self.__class__, self).__init__(tty, hz, baud)

    self._serial_thread = threading.Thread(group=None, target=self._read_thread_run)
    self._serial_thread.daemon = True

    self._decode_thread = threading.Thread(group=None, target=self._decode_thread_run)
    self._decode_thread.daemon = True

    self._reset_and_conf(timers=True)

    self._read_decode_queue = queue.Queue()

    self._decode_thread.start()
    self._serial_thread.start()

    while not self._serial_ready: # Wait until we've populated data before continuing
      pass

  def close(self):
```

```python
    self._serial_stop = True

    if self._serial_thread is not None:
      while self._serial_thread.is_alive(): # Wait for thread to terminate
        pass

  def _read_thread_run(self):
    ser = self._serial_obj
    q = self._read_decode_queue
    self._update_info()

    while True:
      msg = self._wait_read_packet()

      q.put(msg)
      self._serial_ready = True

      if self._serial_stop:
        ser.close()
        return

  def _decode_thread_run(self):
    dq = self._read_decode_queue
    while True:
      msg = dq.get(block=True)

      dpct = self._decode_packet(msg)

      if 'ir' in dpct:
        self._temps = dpct

        for q in self._queues:
          q.put(self.get_temps())
```

```
      if self._serial_stop:
        return


class OnDemandManager(BaseManager):
  def __init__(self, tty, hz=8, baud=115200):
    super(self.__class__, self).__init__(tty, hz, baud)

    self._reset_and_conf(timers=False)

    self._update_info()

  def close(self):
    self._serial_obj.close()

  def capture(self):
    self._serial_obj.write('p') # Capture frame manually
    self._serial_obj.flush()

    msg = self._wait_read_packet()
    dpct = self._decode_packet(msg)

    if 'ir' in dpct:
      self._temps = dpct

      for q in self._queues:
        q.put(self.get_temps())

    return dpct



class ManagerPlaybackEmulator(BaseManager):
  _playback_data = None
```

```python
_pb_thread = None
_pb_stop = False
_pb_len = 0

_i = 0

def __init__(self, playback_data=None):
  if playback_data is not None:
    self.irhz, self._playback_data = playback_data
    self._pb_len = len(self._playback_data)

  self.driver = "Playback"
  self.build = "1"

def set_playback_data(self, playback_data):
  self.stop()
  self.irhz, self._playback_data = playback_data
  self._pb_len = len(self._playback_data)

def close(self):
  return

def start(self):
  if self._pb_thread is None:
    self._pb_stop = False
    self._pb_thread = threading.Thread(group=None, target=self._pb_thread_run)
    self._pb_thread.daemon = True
    self._pb_thread.start()

def pause(self):
  self._pb_stop = True

  while self._pb_thread is not None and self._pb_thread.is_alive():
```

```
      pass

    self._pb_thread = None

  def stop(self):
    self._pb_stop = True

    while self._pb_thread is not None and self._pb_thread.is_alive():
      pass

    self._pb_thread = None
    self._i = 0

  def get_temps(self):
    return self._playback_data[self._i]

  def _pb_thread_run(self):
    while True:
      if self._pb_stop:
        return

      for q in self._queues:
        q.put(self._playback_data[self._i])

      time.sleep(1.0/float(self.irhz))

      self._i += 1

      if self._i >= self._pb_len:
        return



class Visualizer(object):
```

```python
_display_thread = None
_display_stop = False
_tmin = None
_tmax = None
_limit = None
_dwidth = None

_tcam = None
_ffmpeg_loc = None

_camera = None

def __init__(self, tcam=None, camera=None, ffmpeg_loc="ffmpeg"):
  self._tcam = tcam
  self._ffmpeg_loc = ffmpeg_loc
  self._camera = camera

def display(self, block=False, limit=0, width=100, tmin=15, tmax=45):
  q = self._tcam.subscribe_multiprocess()
  _, proc = pxdisplay.create(q, limit=limit, width=width, tmin=tmin, tmax=tmax)

  if block:
    proc.join()

def playback(self, filen, tmin=15, tmax=45):
  hz, playdata = self.file_to_capture(filen)

  print(hz)

  q, thread = pxdisplay.create(
    limit=hz,
    tmin=tmin,
    tmax=tmax,
    caption="Playing back '{}'".format(filen)
```

```
  )

  start = datetime.datetime.now()
  offset = playdata[0]['start_millis']

  for n, frame in enumerate(playdata):
    frame['text'] = 'T+%.3f' % ((frame['start_millis'] - offset)/ 1000.0)
    q.put(frame)

def display_close(self):
  if self._display_thread is None:
    return

  self._display_stop = True
  self._display_thread = None

def close(self):
  self.display_close()

def capture_to_file(self, capture, hz, filen):
  with open(filen + '_thermal.hcap', 'w') as f:
    f.write(str(hz) + "\n")

    for frame in capture:
      t = frame['start_millis']
      motion = frame['movement']
      arr = frame['ir']
      f.write(str(t) + "\n")
      f.write(str(motion) + "\n")
      for l in arr:
        f.write('\t'.join([str(x) for x in l]) + "\n")
      f.write("\n")

def capture_to_img_sequence(self, capture, directory, tmin=15, tmax=45, text=True):
```

```
  hz, frames = capture
  pxwidth = 120
  print(directory)

  for i, frame in enumerate(frames):
    im = Image.new("RGB", (1920, 480))
    draw = ImageDraw.Draw(im)
    font = ImageFont.truetype("arial.ttf", 35)

    for k, row in enumerate(frame['ir']):
      for j, px in enumerate(row):
        rgb = pxdisplay.temp_to_rgb(px, tmin, tmax)

        x = k*pxwidth
        y = j*pxwidth

        coords = (y, x, y+pxwidth+1, x+pxwidth+1)

        draw.rectangle(coords, fill=rgb)

        if text:
          draw.text([y+20,x+(pxwidth/2-20)], str(px), fill=(255,255,255), font=font)

    im.save(os.path.join(directory, '{:09d}.png'.format(i)))

def capture_to_movie(self, capture, filename, width=1920, height=480, tmin=15, tmax=45):
  hz, frames = capture
  tdir = tempfile.mkdtemp()

  self.capture_to_img_sequence(capture, tdir, tmin=tmin, tmax=tmax)

  args = [self._ffmpeg_loc,
    "-y",
    "-r", str(fractions.Fraction(hz)),
```

```
      "-i", os.path.join(tdir, "%09d.png"),
      "-s", "{}x{}".format(width, height),
      "-sws_flags", "neighbor",
      "-sws_dither", "none",
      '-vcodec', 'qtrle', '-pix_fmt', 'rgb24',
      filename + '_thermal.mov'
      ]

    subprocess.call(args)

  def file_to_capture(self, filen):
    capture = []
    hz = None
    with open(filen + '_thermal.hcap', 'r') as f:
      frame = {'ir':[]}

      for i, line in enumerate(f):
        if i == 0:
          hz = float(line)
          continue

        j = (i-1) % 7
        if j == 0:
          frame['start_millis'] = int(line)
        elif j == 1:
          frame['movement'] = bool(line)
        elif 1 < j < 6:
          frame['ir'].append(tuple([float(x) for x in line.split("\t")]))
        elif j == 6:
          capture.append(frame)
          frame = {'ir':[]}

    return (hz, capture)
```

```python
def capture(self, seconds, name=None, hcap=False, video=False):
  buff = []
  q = self._tcam.subscribe()
  hz = self._tcam.irhz
  tdir = tempfile.mkdtemp()

  camera = None
  visfile = name + '_visual.h264' #os.path.join(tdir, name + '_visual.h264')

  if video and self._camera is not None:
    self._camera.resolution = (1920, 1080)
    self._camera.framerate = hz
    self._camera.start_recording(visfile)

  start = time.time()
  elapsed = 0

  while elapsed <= seconds:
    elapsed = time.time() - start
    buff.append( q.get() )

  if video and self._camera is not None:
    self._camera.stop_recording()

    #args = [self._ffmpeg_loc,
    #  "-y",
    #  "-r", str(fractions.Fraction(hz)),
    #  "-i", visfile,
    #  "-vcodec", "copy",
    #  name + '_visual.mp4'
    #  ]

    #subprocess.call(args)

```

```
      #os.remove(visfile)


    if hcap:
      self.capture_to_file(buff, hz, name)

    return (hz, buff)

  def capture_synced(self, seconds, name, hz=2):
    cap_method = getattr(self._tcam, "capture", None)
    if not callable(cap_method):
      raise "Provided tcam class must support the capture method"

    if self._camera is None:
      raise "No picamera object provided, cannot proceed"

    camera = self._camera
    camera.resolution = (1920, 1080)

    # TODO: Currently produces black images. Need to fix.
    # Wait for analog gain to settle on a higher value than 1
    #while camera.analog_gain <= 1 or camera.digital_gain <= 1:
    #    time.sleep(1)

    # Now fix the values
    #camera.shutter_speed = camera.exposure_speed
    #camera.exposure_mode = 'off'
    #g = camera.awb_gains
    #camera.awb_mode = 'off'
    #camera.awb_gains = g

    import datetime, threading, time

    dir_name = name
```

```python
frames = seconds * hz

buff = []
imgbuff = [io.BytesIO() for _ in range(frames + 1)]
fps_avg = []
lag_avg = []

try:
  os.mkdir(dir_name)
except OSError:
  pass

def trigger(next_call, i):
  if i % (hz * 3) == 0:
    print('{}/{} seconds'.format(i/hz, seconds))

  t1_start = time.time()
  camera.capture(imgbuff[i], 'jpeg', use_video_port=True)
  t1_t2 = time.time()
  buff.append(self._tcam.capture())
  t2_stop = time.time()

  sec = t2_stop - t1_start
  fps_avg.append(sec)
  lag_avg.append(t2_stop - t1_t2)

  if sec > (1.0/float(hz)):
    print('Cannot keep up with frame rate!')

  if frames == i:
    return

  th = threading.Timer( next_call - time.time(), trigger,
    args=[next_call+(1.0/float(hz)), i + 1] )
```

```
    th.start()
    th.join()

  trigger(time.time(), 0)

  print('Average time for frame capture = {} seconds'.format(sum(fps_avg)/len(fps_avg)))
  print('Average lag between camera and thermal capture = {} seconds'.format(sum(lag_avg)/len(lag_avg)))

  self.capture_to_file(buff, hz, os.path.join(dir_name, 'output'))

  for i, b in enumerate(imgbuff):
    img_name = os.path.join(dir_name, 'video-{:09d}.jpg'.format(i))
    with open(img_name, 'wb') as f:
      f.write(b.getvalue())

  return (hz, buff)
```


C.1.2 pxdisplay.py

```
from __future__ import division
from __future__ import print_function

from multiprocessing import Process, Queue
import colorsys
import time

def millis_diff(a, b):
  diff = b - a
  return (diff.days * 24 * 60 * 60 + diff.seconds) * 1000 + diff.microseconds / 1000.0

def temp_to_rgb(temp, tmin, tmax):
        OLD_MIN = tmin
```

```
    OLD_MAX = tmax

    if temp < OLD_MIN:
      temp = OLD_MIN

    if temp > OLD_MAX:
      temp = OLD_MAX

    v = (temp - OLD_MIN) / (OLD_MAX - OLD_MIN)

    rgb = colorsys.hsv_to_rgb((1-v), 1, v * 0.5)

    return tuple(int(c * 255) for c in rgb)

def create(q=None, limit=0, width=100, tmin=15, tmax=45, caption="Display"):
  if q is None:
    q = Queue()

  p = Process(target=_display_process, args=(q, caption, tmin, tmax, limit, width))
  p.daemon = True
  p.start()

  return (q, p)

def _display_process(q, caption, tmin, tmax, limit, pxwidth):
  import pygame
  pygame.init()
  pygame.display.set_caption(caption)

  size = (16 * pxwidth, 4 * pxwidth)
  screen = pygame.display.set_mode(size)

  background = pygame.Surface(screen.get_size())
  background = background.convert_alpha()
```

```
font = pygame.font.Font(None, 36)

while True:
  for event in pygame.event.get():
    if event.type == pygame.QUIT:
      pygame.quit()
      return

  # Keep the event loop running so the windows don't freeze without data
  try:
    qg = q.get(True, 0.3)
  except:
    continue

  px = qg['ir']

  #lag = q.qsize()
  #if lag > 0:
  #  print("WARNING: Dropped " + str(lag) + " frames")

  for i, row in enumerate(px):
    for j, v in enumerate(row):
      rgb = temp_to_rgb(v, tmin, tmax)

      x = i*pxwidth
      y = j*pxwidth

      screen.fill(rgb, (y, x, pxwidth, pxwidth))

  if 'text' in qg:
    background.fill((0, 0, 0, 0))
    text = font.render(qg['text'], 1, (255,255,255))
    background.blit(text, (0,0))
```

```
      # Blit everything to the screen
      screen.blit(background, (0, 0))

    pygame.display.flip()

    if limit != 0:
      time.sleep(1.0/float(limit))
```

C.1.3 features.py

```
from __future__ import division
from __future__ import print_function

import threading
import pxdisplay
import time
import math
import copy
import networkx as nx
import itertools
import collections
#import matplotlib.pyplot as plt

def tuple_to_list(l):
  new = []

  for r in l:
    new.append(list(r))

  return new

```

```
def min_temps(l, n):
  flat = []
  for i, r in enumerate(l):
    for j, v in enumerate(r):
      flat.append(((i,j), v))
  flat.sort(key=lambda x: x[1])

  ret = [x[0] for x in flat]
  return ret[:n]


def init_arr(val=None):
  return [[val for x in range(16)] for x in range(4)]

class Features(object):
  _q = None
  _thread = None

  _background = None
  _means = None
  _stds = None
  _stds_post = None
  _active = None

  _num_active = None
  _connected_graph = None
  _num_connected = None
  _size_connected = None

  _lock = None

  _rows = None
  _columns = None
```

```
motion_weight = None
nomotion_weight = None

motion_window = None

hz = None

display = None

_exit = False

def __init__(self, q, hz, motion_window=10, motion_weight=0.1, nomotion_weight=0.01, display=True, rows=4, columns=16):
  self._q = q
  self.hz = hz
  self.motion_weight = motion_weight
  self.nomotion_weight = nomotion_weight
  self.display = display
  self.motion_window = motion_window

  self._active = []

  self._rows = rows
  self._columns = columns

  self._thread = threading.Thread(group=None, target=self._monitor_thread)
  self._thread.daemon = True

  self._lock = threading.Lock()

  self._thread.start()

def get_background(self):
  self._lock.acquire()
```

```
    background = copy.deepcopy(self._background)
    self._lock.release()
    return background

  def get_means(self):
    self._lock.acquire()
    means = copy.deepcopy(self._means)
    self._lock.release()
    return means

  def get_stds(self):
    self._lock.acquire()
    stds = copy.deepcopy(self._stds_post)
    self._lock.release()
    return stds

  def get_active(self):
    self._lock.acquire()
    active = copy.deepcopy(self._active)
    self._lock.release()
    return active

  def get_features(self):
    self._lock.acquire()
    num_active = self._num_active
    num_connected = self._num_connected
    size_connected = self._size_connected
    self._lock.release()
    return (num_active, num_connected, size_connected)

  def close(self):
    self._exit = True

    if self._thread is not None:
```

```
      while self._thread.is_alive(): # Wait for thread to terminate
        pass

  def __del__(self):
    self.close()

  def _monitor_thread(self):
    bdisp = None
    ddisp = None

    freq = self.hz * self.motion_window
    mwin = collections.deque([False] * freq)

    n = 1
    while True:
      fdata = None

      if self._exit:
        return

      try:
        fdata = self._q.get(True, 0.3)
      except:
        continue

      if self.display and bdisp is None:
        bdisp, _ = pxdisplay.create(caption="Background", width=80)
        ddisp, _ = pxdisplay.create(caption="Deviation", width=80)

      frame = fdata['ir']

      mwin.popleft()
      mwin.append(fdata['movement'])
      motion = any(mwin)
```

```
self._lock.acquire()

self._active = []

g = nx.Graph()

if n == 1:
  self._background = tuple_to_list(frame)
  self._means = tuple_to_list(frame)
  self._stds = init_arr(0)
  self._stds_post = init_arr()
else:
  weight = self.nomotion_weight
  use_frame = frame

  # Not currently working
  #if motion:
  #  indeces = min_temps(frame, 5)
  #  scalepx = []
  #
  #  for i, j in indeces:
  #    scalepx.append(self._background[i][j] / frame[i][j])
  #
  #  scale = sum(scalepx) / len(scalepx)
  #  scaled_bg = [[x * scale for x in r] for r in frame]
  #
  #  weight = self.motion_weight
  #  use_frame = scaled_bg

  for i in range(self._rows):
    for j in range(self._columns):
      prev = self._background[i][j]
      cur = use_frame[i][j]
```

```
      cur_mean = self._means[i][j]
      cur_std = self._stds[i][j]

      if not motion: # TODO: temp fix
        self._background[i][j]   = weight * cur + (1 - weight) * prev

        # maybe exclude these from motion calculations?
        # n doesn't change when in motion, so it'll cause all sort of corrupted results, as they use n?
        self._means[i][j] = cur_mean + (cur - cur_mean) / n
        self._stds[i][j]  = cur_std + (cur - cur_mean) * (cur - self._means[i][j])
        self._stds_post[i][j] = math.sqrt(self._stds[i][j] / (n-1))

      if (cur - self._background[i][j]) > (3 * self._stds_post[i][j]):
        self._active.append((i,j))

        g.add_node((i,j))

        x = [(-1, -1), (-1, 0), (-1, 1), (0, -1)] # Nodes that have already been computed as active
        for ix, jx in x:
          if (i+ix, j+jx) in self._active:
            g.add_edge((i,j), (i+ix,j+jx))

active = self._active

self._num_active = len(self._active)

components = list(nx.connected_components(g))

self._connected_graph = g
self._num_connected = nx.number_connected_components(g)
self._size_connected = max(len(component) for component in components) if len(components) > 0 else None

self._lock.release()
```

```python

      if self.display:
        bdisp.put({'ir': self._background})

        if n >= 2:
          std = {'ir': init_arr(0)}

          for i, j in active:
            std['ir'][i][j] = frame[i][j]

          ddisp.put(std)

      #print(n)
      #if n > 30:
      #  nx.draw(g)
      #  plt.show()


      if not motion:
        n += 1
```


C.2 Arduino Sketch

```c
/**
 * MLX90260 Arduino Interface
 * Based on code from http://forum.arduino.cc/index.php/topic,126244.0.html
 */
//#define __ASSERT_USE_STDERR

//#include <assert.h>
#include <math.h>
#include <Wire.h>
```

```
#include <EEPROM.h>
#include "SimpleTimer.h" // http://playground.arduino.cc/Code/SimpleTimer

// Configurable options
const int POR_CHECK_FREQ    = 2000; // Time in milliseconds to check if MLX reset has occurred
const int PIR_INTERRUPT_PIN = 0;    // D2 on the Arduino Uno

// Configuration constants
#define PIXEL_LINES     4
#define PIXEL_COLUMNS   16
#define BYTES_PER_PIXEL 2
#define EEPROM_SIZE     255
#define NUM_PIXELS      (PIXEL_LINES * PIXEL_COLUMNS)

// EEPROM helpers
#define E_READ(X)       (EEPROM_DATA[X])
#define E_WRITE(X, Y)   (EEPROM_DATA[X] = (Y))

// Bit fiddling helpers
#define BYTES2INT(H, L)     ( ((H) << 8) + (L) )
#define UBYTES2INT(H, L)    ( ((unsigned int)(H) << 8) + (unsigned int)(L) )
#define BYTE2INT(B)         ( ((int)(B) > 127) ? ((int)(B) - 256) : (int)(B) )
#define E_BYTES2INT(H, L)   ( BYTES2INT(E_READ(H), E_READ(L)) )
#define E_UBYTES2INT(H, L)  ( UBYTES2INT(E_READ(H), E_READ(L)) )
#define E_BYTE2INT(X)       ( BYTE2INT(E_READ(X)) )

// I2C addresses
#define ADDR_EEPROM   0x50
#define ADDR_SENSOR   0x60

// I2C commands
#define CMD_SENSOR_READ         0x02
#define CMD_SENSOR_WRITE_CONF   0x03
#define CMD_SENSOR_WRITE_TRIM   0x04
```

```c

// Addresses in the sensor RAM (see Table 9 in spec)
#define SENSOR_PTAT              0x90
#define SENSOR_CPIX              0x91
#define SENSOR_CONFIG            0x92

// Addresses in the EEPROM (see Tables 5 & 7 in spec)
#define EEPROM_A_I_00              0x00 // A_i(0,0) IR pixel individual offset coefficient (ends at 0x3F)
#define EEPROM_B_I_00              0x40 // B_i(0,0) IR pixel individual offset coefficient (ends at 0x7F)
#define EEPROM_DELTA_ALPHA_00      0x80 // Delta-alpha(0,0) IR pixel individual offset coefficient (ends at 0xBF)
#define EEPROM_A_CP                0xD4 // Compensation pixel individual offset coefficients
#define EEPROM_B_CP                0xD5 // Individual Ta dependence (slope) of the compensation pixel offset
#define EEPROM_ALPHA_CP_L          0xD6 // Sensitivity coefficient of the compensation pixel (low)
#define EEPROM_ALPHA_CP_H          0xD7 // Sensitivity coefficient of the compensation pixel (high)
#define EEPROM_TGC                 0xD8 // Thermal gradient coefficient
#define EEPROM_B_I_SCALE           0xD9 // Scaling coefficient for slope of IR pixels offset
#define EEPROM_V_TH_L              0xDA // V_TH0 of absolute temperature sensor (low)
#define EEPROM_V_TH_H              0xDB // V_TH0 of absolute temperature sensor (high)
#define EEPROM_K_T1_L              0xDC // K_T1 of absolute temperature sensor (low)
#define EEPROM_K_T1_H              0xDD // K_T1 of absolute temperature sensor (high)
#define EEPROM_K_T2_L              0xDE // K_T2 of absolute temperature sensor (low)
#define EEPROM_K_T2_H              0xDF // K_T2 of absolute temperature sensor (high)
#define EEPROM_ALPHA_0_L           0xE0 // Common sensitivity coefficient of IR pixels (low)
#define EEPROM_ALPHA_0_H           0xE1 // Common sensitivity coefficient of IR pixels (high)
#define EEPROM_ALPHA_0_SCALE       0xE2 // Scaling coefficient for common sensitivity
#define EEPROM_DELTA_ALPHA_SCALE   0xE3 // Scaling coefficient for individual sensitivity
#define EEPROM_EPSILON_L           0xE4 // Emissivity (low)
#define EEPROM_EPSILON_H           0xE5 // Emissivity (high)
#define EEPROM_TRIMMING_VAL        0xF7 // Oscillator trimming value

// Config flag locations
#define CFG_TA    8
#define CFG_IR    9
#define CFG_POR   10
```

```
// Arduino EEPROM addresses
#define AEEP_FREQ_ADDR 0x00

// Global variables
unsigned int PTAT;              // Proportional to absolute temperature value
int CPIX;                       // Compensation pixel

int IRDATA[NUM_PIXELS];         // Infrared raw data
byte EEPROM_DATA[EEPROM_SIZE];  // EEPROM dump

float ta;                       // Absolute chip temperature / ambient chip temperature (degrees celsius)
float emissivity;               // Emissivity compensation
float k_t1;                     // K_T1 of absolute temperature sensor
float k_t2;                     // K_T2 of absolute temperature sensor
float da0_scale;                // Scaling coefficient for individual sensitivity
float alpha_const;              // Common sensitivity coefficient of IR pixels and scaling coefficient for
 ↪  common sensitivity

int v_th;                       // V_TH0 of absolute temperature sensor
int a_cp;                       // Compensation pixel individual offset coefficients
int b_cp;                       // Individual Ta dependence (slope) of the compensation pixel offset
int tgc;                        // Thermal gradient coefficient
int b_i_scale;                  // Scaling coefficient for slope of IR pixels offset

float alpha_ij[NUM_PIXELS];     // Individual pixel sensitivity coefficient
int a_ij[NUM_PIXELS];           // Individual pixel offset
int b_ij[NUM_PIXELS];           // Individual pixel offset slope coefficient

char hpbuf[2];                  // Hex printing buffer
int res;                        // Error code storage

float temp[NUM_PIXELS];         // Final calculated temperature values in degrees celsius
```

```
SimpleTimer timer;              // Allows timed callbacks for temp functions

void(* reset_arduino_now) (void) = 0;   // Creates function to reset Arduino

// Stores references to the 3 timers used in the program
int ir_timer;
int ta_timer;
int por_timer;

// Stores refresh frequency, read out of the EEPROM
short REFRESH_FREQ;

volatile bool pir_motion_detected = false;

/*
// Send assertion failures over serial
void __assert(const char *__func, const char *__file, int __lineno, const char *__sexp) {
    // transmit diagnostic informations through serial link.
    Serial.println(__func);
    Serial.println(__file);
    Serial.println(__lineno, DEC);
    Serial.println(__sexp);
    Serial.flush();
    // abort program execution.
    abort();
}*/

void reset_arduino() {
  Serial.flush();
  reset_arduino_now();
}

// Basic assertion failure function
void assert(boolean a) {
```

```
  if (!a) Serial.println("ASSFAIL");
}

// Takes byte value and will output 2 character hex representation on serial
void print_hex(byte b) {
  hpbuf[0] = (b >> 4) + 0x30;
  if (hpbuf[0] > 0x39) hpbuf[0] +=7;

  hpbuf[1] = (b & 0x0f) + 0x30;
  if (hpbuf[1] > 0x39) hpbuf[1] +=7;

  Serial.print(hpbuf);
}

// Will read memory from the given sensor address and convert it into an integer
int _sensor_read_int(byte read_addr) {
  Wire.beginTransmission(ADDR_SENSOR);
  Wire.write(CMD_SENSOR_READ);
  Wire.write(read_addr);
  Wire.write(0x00); // address step (0)
  Wire.write(0x01); // number of reads (1)
  res = Wire.endTransmission(false); // we must use the repeated start here
  if (res != 0) return -1;

  Wire.requestFrom(ADDR_SENSOR, 2); // technically the 1 read takes up 2 bytes

  int LSB, MSB;
  int i = 0;
  while( Wire.available() ) {
    i++;

    if (i > 2) {
      return -1; // Returned more bytes than it should have
    }
```

```
    LSB = Wire.read();
    MSB = Wire.read();
  }

  return UBYTES2INT(MSB, LSB); // rearrange int to account for endian difference (TODO: check)
}

// Will read a configuration flag bit specified by flag_loc from the sensor config
bool _sensor_read_config_flag(int flag_loc) {
  int cur_cfg = _sensor_read_int(SENSOR_CONFIG);
  return (bool)(cur_cfg & ( 1 << flag_loc )) >> flag_loc;
}

// Reads Proportional To Absolute Temperature (PTAT) value
int sensor_read_ptat() {
  return _sensor_read_int(SENSOR_PTAT);
}

// Reads compensation pixel
int sensor_read_cpix() {
  return _sensor_read_int(SENSOR_CPIX);
}

// Reads POR flag
bool sensor_read_por() {
  return _sensor_read_config_flag(CFG_POR); // POR is 10th bit
}

// Read Ta measurement flag
bool sensor_read_ta_measure() {
  return _sensor_read_config_flag(CFG_TA);
}

```

```
// Read IR measurement flag
bool sensor_read_ir_measure() {
  return _sensor_read_config_flag(CFG_IR);
}

// Reads all raw IR data from sensor into IRDATA variable
boolean sensor_read_irdata() {
  int i = 0;

  // Due to wire library buffer limitations, we can only read up to 32 bytes at a time
  // Thus, the request has been split into multiple different requests to get the full 128 values
  // Each pixel value takes up two bytes (???) thus NUM_PIXELS * 2
  for (int line = 0; line < PIXEL_LINES; line++) {
    Wire.beginTransmission(ADDR_SENSOR);
    Wire.write(CMD_SENSOR_READ);
    Wire.write(line);
    Wire.write(0x04);
    Wire.write(0x10);
    res = Wire.endTransmission(false); // use repeated start to get answer

    if (res != 0) return false;

    Wire.requestFrom(ADDR_SENSOR, PIXEL_COLUMNS * BYTES_PER_PIXEL);

    byte PIX_LSB, PIX_MSB;

    for(int j = 0; j < PIXEL_COLUMNS; j++) {
      if (!Wire.available()) return false;

      // We read two bytes
      PIX_LSB = Wire.read();
      PIX_MSB = Wire.read();

      IRDATA[i] = BYTES2INT(PIX_MSB, PIX_LSB);
```

```
      i++;
    }
  }

  return true;
}

// Will send a command and the provided most significant and least significant bit
// with the appropriate check bit added
// Returns the Wire success/error code
boolean _sensor_write_check(byte cmd, byte check, byte lsb, byte msb) {
  Wire.beginTransmission(ADDR_SENSOR);
  Wire.write(cmd);          // Send the command
  Wire.write(lsb - check);  // Send the least significant byte check
  Wire.write(lsb);          // Send the least significant byte
  Wire.write(msb - check);  // Send the most significant byte check
  Wire.write(msb);          // Send the most significant byte
  return Wire.endTransmission() == 0;
}

// See datasheet: 9.4.2 Write configuration register command
// See datasheet: 8.2.2.1 Configuration register (0x92)
// Check byte is 0x55 in this instance
boolean sensor_write_conf() {
  byte cfg_MSB = B01110100;
  //             ||||||||
  //             |||||||*--- Ta measurement running (read only)
  //             ||||||*---- IR measurement running (read only)
  //             |||||*----- POR flag cleared
  //             ||||*------ I2C FM+ mode enabled
  //             ||**------- Ta refresh rate (2 byte code, 2Hz hardcoded)
  //             |*--------- ADC high reference
  //             *---------- NA

```

```
byte cfg_LSB = B00001110;
//              ||||||||
//              ||||****--- 4 byte IR refresh rate (4 byte code, 1Hz default)
//              ||**------- NA
//              |*--------- Continuous measurement mode
//              *---------- Normal operation mode

switch(REFRESH_FREQ) {
case 0: // 0.5Hz
  cfg_LSB = B00001111;
  break;
case 2:
  cfg_LSB = B00001101;
  break;
case 4:
  cfg_LSB = B00001100;
  break;
case 8:
  cfg_LSB = B00001011;
  break;
case 16:
  cfg_LSB = B00001010;
  break;
case 32:
  cfg_LSB = B00001001;
  break;
case 64:
  cfg_LSB = B00001000;
  break;
case 128:
  cfg_LSB = B00000111;
  break;
case 256:
  cfg_LSB = B00000110;
```

```
    break;
  case 512:
    cfg_LSB = B00000000; // modes 5 to 0 are all 512Hz
    break;
  }

  return _sensor_write_check(CMD_SENSOR_WRITE_CONF, 0x55, cfg_LSB, cfg_MSB);
}

// See datasheet: 9.4.3 Write trimming command
// Check byte is 0xAA in this instance
boolean sensor_write_trim() {
  return _sensor_write_check(CMD_SENSOR_WRITE_TRIM, 0xAA, E_READ(EEPROM_TRIMMING_VAL), 0x00);
}

// Reads EEPROM memory into global variable
boolean eeprom_read_all() {
  int i = 0;
  // Due to wire library buffer limitations, we can only read up to 32 bytes at a time
  // Thus, the request has been split into 4 different requests to get the full 128 values
  for(int j = 0; j < EEPROM_SIZE; j = j + 32) {
    Wire.beginTransmission(ADDR_EEPROM);
    Wire.write( byte(j) );
    res = Wire.endTransmission();

    if (res != 0) return false;

    Wire.requestFrom(ADDR_EEPROM, 32);

    i = j;
    while( Wire.available() ) { // slave may send less than requested
      byte b = Wire.read(); // receive a byte as character
      E_WRITE(i, b);
      i++;
```

```
    }
  }

  if (i < EEPROM_SIZE) { // If we didn't get the whole EEPROM
    return false;
  }

  return true;
}

// Writes various calculation values from EEPROM into global variables
void calculate_init() {
  v_th = E_BYTES2INT(EEPROM_V_TH_H, EEPROM_V_TH_L);
  k_t1 = E_BYTES2INT(EEPROM_K_T1_H, EEPROM_K_T1_L) / 1024.0;
  k_t2 = E_BYTES2INT(EEPROM_K_T2_H, EEPROM_K_T2_L) / 1048576.0;

  a_cp = E_BYTE2INT(EEPROM_A_CP);
  b_cp = E_BYTE2INT(EEPROM_B_CP);
  tgc  = E_BYTE2INT(EEPROM_TGC);

  b_i_scale = E_READ(EEPROM_B_I_SCALE);

  emissivity = E_UBYTES2INT(EEPROM_EPSILON_H, EEPROM_EPSILON_L) / 32768.0;

  da0_scale = pow(2, -E_READ(EEPROM_DELTA_ALPHA_SCALE));
  alpha_const = (float)E_UBYTES2INT(EEPROM_ALPHA_0_H, EEPROM_ALPHA_0_L) * pow(2, -E_READ(EEPROM_ALPHA_0_SCALE));

  for (int i = 0; i < NUM_PIXELS; i++){
    float alpha_var = (float)E_READ(EEPROM_DELTA_ALPHA_00 + i) * da0_scale;
    alpha_ij[i] = (alpha_const + alpha_var);

    a_ij[i] = E_BYTE2INT(EEPROM_A_I_00 + i);
    b_ij[i] = E_BYTE2INT(EEPROM_B_I_00 + i);
  }
```

```
}

// Calculates the absolute chip temperature from the proportional to absolute temperature (PTAT)
float calculate_ta() {
  float ptat = (float)sensor_read_ptat();
  assert(ptat != -1);
  return (-k_t1 +
      sqrt(
        square(k_t1) -
        ( 4 * k_t2 * (v_th-ptat) )
      )
    ) / (2*k_t2) + 25;
}

// Calculates the final temperature value for each pixel and stores it in temp array
void calculate_temp() {
  float v_cp_off_comp = (float) CPIX - (a_cp + (b_cp/pow(2, b_i_scale)) * (ta - 25));

  for (int i = 0; i < NUM_PIXELS; i++){
    float alpha_ij_v = alpha_ij[i];
    int a_ij_v = a_ij[i];
    int b_ij_v = b_ij[i];

    float v_ir_tgc_comp = IRDATA[i] - (a_ij_v + (float)(b_ij_v/pow(2, b_i_scale)) * (ta - 25)) - (((float)tgc/32)*v_cp_off_comp);
    float v_ir_comp = v_ir_tgc_comp / emissivity;
    temp[i] = sqrt(sqrt((v_ir_comp/alpha_ij_v) + pow((ta + 273.15),4))) - 273.15;
  }

}

// Prints all of EEPROM as hex
void print_eeprom() {
  Serial.print("EEPROM ");
```

```
  for(int i = 0; i < EEPROM_SIZE; i++) {
    print_hex(E_READ(i));
  }
  Serial.println();
}

// Prints a serial "packet" containing IR data
void print_packet(unsigned long cur_time) {
  Serial.print("START ");
  Serial.println(cur_time);

  Serial.print("MOVEMENT ");
  Serial.println(pir_motion_detected);

  for(int i = 0; i<NUM_PIXELS; i++) {
    Serial.print(temp[i]);

    if ((i+1) % PIXEL_COLUMNS == 0) {
      Serial.println();
    } else {
      Serial.print("\t");
    }
  }

 Serial.print("STOP ");
 Serial.println(millis());
 Serial.flush();
}

// Prints info about driver, build and configuration
void print_info() {
  Serial.println("INFO START");
  Serial.println("DRIVER MLX90620");
```

```
  Serial.print("BUILD ");
  Serial.print(__DATE__);
  Serial.print(" ");
  Serial.println(__TIME__);

  Serial.print("IRHZ ");
  Serial.println(REFRESH_FREQ);
  Serial.println("INFO STOP");
}

// Runs functions necessary to initialize the temperature sensor
void initialize() {
  assert(eeprom_read_all());
  assert(sensor_write_trim());
  assert(sensor_write_conf());

  calculate_init();

  ta_loop();
}

// Calculates absolute temperature
void ta_loop() {
  ta = calculate_ta();
}

// Checks if the sensor as been reset, and if so, re-runs the initialize functions
void por_loop() {
  if (!sensor_read_por()) { // there has been a reset
    initialize();
  }
}

// Runs functions necessary to compute and output the temperature data
```

```
void ir_loop() {
  unsigned long cur_time = millis();

  assert(sensor_read_irdata());

  CPIX = sensor_read_cpix();
  assert(CPIX != -1);

  calculate_temp();

  print_packet(cur_time);

  pir_motion_detected = false;
}

// Configures timers to poll IR and other data periodically
void activate_timers() {
  float hz = REFRESH_FREQ;

  if (REFRESH_FREQ == 0) {
    hz = 0.5;
  }

  // Calculate how many milliseconds each timer should run for
  // based upon the configured refresh rate of the IR data and
  // absolute temperature data
  long irlen = (1/hz) * 1000;
  long talen = (1/2.0) * 1000;

  if (talen < irlen) {
    talen = irlen;
  }

  ir_timer = timer.setInterval(irlen, ir_loop);
```

```
  ta_timer = timer.setInterval(talen, ta_loop);
  por_timer = timer.setInterval(POR_CHECK_FREQ, por_loop);

  attachInterrupt(PIR_INTERRUPT_PIN, pir_motion, RISING);
}

// Disables timers to poll IR and other data periodically
void deactivate_timers() {
  timer.disable(ir_timer);
  timer.deleteTimer(ir_timer);

  timer.disable(ta_timer);
  timer.deleteTimer(ta_timer);

  timer.disable(por_timer);
  timer.deleteTimer(por_timer);

  detachInterrupt(PIR_INTERRUPT_PIN);
}

void pir_motion() {
  pir_motion_detected = true;
}

void read_freq() {
  byte rd = EEPROM.read(0);

  if (rd > 9) {
    rd = 0;
    EEPROM.write(AEEP_FREQ_ADDR, 0);
  }

  switch(rd) {
  case 1:
```

```
    REFRESH_FREQ = 1;
    break;
  case 2:
    REFRESH_FREQ = 2;
    break;
  case 3:
    REFRESH_FREQ = 4;
    break;
  case 4:
    REFRESH_FREQ = 8;
    break;
  case 5:
    REFRESH_FREQ = 16;
    break;
  case 6:
    REFRESH_FREQ = 32;
    break;
  case 7:
    REFRESH_FREQ = 64;
    break;
  case 8:
    REFRESH_FREQ = 128;
    break;
  case 9:
    REFRESH_FREQ = 256;
    break;
  case 10:
    REFRESH_FREQ = 512;
    break;

  default:
  case 0:
    REFRESH_FREQ = 0;
    break;
```

```
  }
}

void write_freq(int freq) {
  byte wt;

  switch(freq) {
  case 1:
    wt = 1;
    break;
  case 2:
    wt = 2;
    break;
  case 4:
    wt = 3;
    break;
  case 8:
    wt = 4;
    break;
  case 16:
    wt = 5;
    break;
  case 32:
    wt = 6;
    break;
  case 64:
    wt = 7;
    break;
  case 128:
    wt = 8;
    break;
  case 256:
    wt = 9;
    break;
```

```
  case 512: // writing 512 to the config doesn't work for some reason
    wt = 10;
    break;

  default:
  case 0:
    wt = 0;
    break;
  }

  EEPROM.write(AEEP_FREQ_ADDR, wt);
}

// Configure libraries and sensors at startup
void setup() {
  pinMode(2, INPUT);

  Wire.begin();
  Serial.begin(115200);

  Serial.println();
  Serial.print("INIT ");
  Serial.println(millis());

  read_freq();
  print_info();
  initialize();

  Serial.print("ACTIVE ");
  Serial.println(millis());
  Serial.flush();
}

char manualLoop = 0;
```

```
// Triggered when serial data is sent to Arduino. Used to trigger basic actions.
void serialEvent() {
  while (Serial.available()) {
    char in = (char)Serial.read();
    if (in == '\r' || in == '\n') continue;

    switch (in) {
    case 'R':
    case 'r':
      reset_arduino();
      break;

    case 'I':
    case 'i':
      print_info();
      break;

    case 'T':
    case 't':
      activate_timers();
      break;

    case 'O':
    case 'o':
      deactivate_timers();
      break;

    case 'P':
    case 'p':
      if (manualLoop == 16) { // Run ta_loop every 16 manual iterations
        ta_loop();
        manualLoop = 0;
      }
```

```
      ir_loop();

      manualLoop++;
      break;

    case 'f':
    case 'F':
      write_freq(Serial.parseInt());
      reset_arduino();
      break;

    default:
      Serial.println("UNKNOWN COMMAND");
    }
  }
}

void loop() {
  timer.run();
}
```

APPENDIX D

Full Results

D.1 Classifier Experiment Set 1

D.1.1 Nominal Results

D.1.1.1 Multilayer Perceptron

```
=== Run information ===

Scheme:weka.classifiers.functions.MultilayerPerceptron -L 0.3 -M 0.2 -N 500 -V
 ↪  0 -S 0 -E 20 -H a
Relation:     persondata
Instances:    1018
Attributes:   4
              npeople
              numactive
              numconnected
              sizeconnected
Test mode:10-fold cross-validation

=== Classifier model (full training set) ===

Sigmoid Node 0
    Inputs    Weights
    Threshold    -17.82098538043138
    Node 4    10.791969171144421
    Node 5    11.691523214004624
    Node 6    10.27822454007849
Sigmoid Node 1
    Inputs    Weights
    Threshold    -1.7152968701419837
    Node 4    -7.571770467221156
    Node 5    -5.127559825773417
    Node 6    6.476543544185421
Sigmoid Node 2
    Inputs    Weights
```

```
    Threshold    1.9339801770968827
    Node 4    -2.6952562384782275
    Node 5    2.620671306339044
    Node 6    -8.20640522534469
Sigmoid Node 3
    Inputs    Weights
    Threshold    -2.47686769207173
    Node 4    3.378401295716778
    Node 5    0.6306342479203954
    Node 6    -3.925441217557144
Sigmoid Node 4
    Inputs    Weights
    Threshold    3.5799482950612207
    Attrib numactive    3.5468014230351153
    Attrib numconnected    -1.9506325622725589
    Attrib sizeconnected    15.731567321159028
Sigmoid Node 5
    Inputs    Weights
    Threshold    -13.566502805330678
    Attrib numactive    1.4688308541180812
    Attrib numconnected    4.568878889123458
    Attrib sizeconnected    -20.825158179068985
Sigmoid Node 6
    Inputs    Weights
    Threshold    2.782123031368699
    Attrib numactive    -17.96989902500443
    Attrib numconnected    4.340499299171253
    Attrib sizeconnected    15.599045813296243
Class 0
    Input
    Node 0
Class 1
    Input
    Node 1
Class 2
    Input
    Node 2
Class 3
    Input
    Node 3


Time taken to build model: 0.88 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances         799               78.4872 %
Incorrectly Classified Instances       219               21.5128 %
```

```
Kappa statistic                          0.6824
Mean absolute error                      0.153
Root mean squared error                  0.2936
Relative absolute error                 44.3263 %
Root relative squared error             70.6965 %
Total Number of Instances             1018

=== Detailed Accuracy By Class ===

       TP Rate   FP Rate   Precision   Recall  F-Measure   ROC Area  Class
         0.908     0.133      0.822     0.908     0.862      0.926    0
         0.687     0.097      0.7       0.687     0.693      0.863    1
         0.801     0.074      0.812     0.801     0.806      0.903    2
         0.313     0.01       0.667     0.313     0.426      0.864    3
WAvg.    0.785     0.1        0.779     0.785     0.777      0.9

=== Confusion Matrix ===

   a   b   c   d   <-- classified as
 373  32   6   0 |   a = 0
  63 173  16   0 |   b = 1
  11  37 233  10 |   c = 2
   7   5  32  20 |   d = 3
```

D.1.1.2 IBk

```
=== Run information ===

Scheme:weka.classifiers.lazy.IBk -K 1 -W 0 -A
 ↪  "weka.core.neighboursearch.LinearNNSearch -A
 ↪  \"weka.core.EuclideanDistance -R first-last\""
Relation:     persondata
Instances:    1018
Attributes:   4
              npeople
              numactive
              numconnected
              sizeconnected
Test mode:10-fold cross-validation

=== Classifier model (full training set) ===

IB1 instance-based classifier
using 1 nearest neighbour(s) for classification


Time taken to build model: 0 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances         586               57.5639 %
Incorrectly Classified Instances       432               42.4361 %
Kappa statistic                          0.4251
Mean absolute error                      0.2294
Root mean squared error                  0.4245
Relative absolute error                 66.4479 %
Root relative squared error            102.2105 %
Total Number of Instances             1018

=== Detailed Accuracy By Class ===

       TP Rate   FP Rate   Precision   Recall  F-Measure   ROC Area  Class
         0.292     0.063      0.759     0.292     0.422      0.736    0
         0.782     0.307      0.456     0.782     0.576      0.849    1
         0.845     0.087      0.796     0.845     0.82       0.922    2
         0.359     0.101      0.193     0.359     0.251      0.748    3
WAvg.    0.576     0.132      0.659     0.576     0.563      0.818

=== Confusion Matrix ===

   a   b   c   d   <-- classified as
 120 196  16  79 |   a = 0
```

```
33 197  15   7 |    b = 1
 4  31 246  10 |    c = 2
 1   8  32  23 |    d = 3
```

D.1.1.3 Naive Bayes

```
=== Run information ===

Scheme:weka.classifiers.bayes.NaiveBayes
Relation:     persondata
Instances:    1018
Attributes:   4
              npeople
              numactive
              numconnected
              sizeconnected
Test mode:10-fold cross-validation

=== Classifier model (full training set) ===

Naive Bayes Classifier

                  Class
Attribute             0       1       2       3
                  (0.4)  (0.25)  (0.29)  (0.06)
================================================
numactive
  mean            1.9705  10.644 20.4324 31.7871
  std. dev.       4.2009  7.0371  9.8619   10.01
  weight sum         323     252     291      64
  precision       1.0417  1.0417  1.0417  1.0417

numconnected
  mean            0.7864  1.5198  2.2165   2.375
  std. dev.        1.005  1.0214  0.8522  0.7181
  weight sum         323     252     291      64
  precision            1       1       1       1

sizeconnected
  mean             3.481 10.2941 11.4944 19.6742
  std. dev.       4.2277  4.7478  5.2921  7.8351
  weight sum         151     223     282      64
  precision       1.4848  1.4848  1.4848  1.4848



Time taken to build model: 0.01 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances         674               66.2083 %
Incorrectly Classified Instances       344               33.7917 %
```

```
Kappa statistic                          0.4964
Mean absolute error                      0.2087
Root mean squared error                  0.3516
Relative absolute error                 60.4564 %
Root relative squared error             84.6405 %
Total Number of Instances             1018

=== Detailed Accuracy By Class ===

       TP Rate   FP Rate   Precision   Recall  F-Measure   ROC Area  Class
         0.925     0.209      0.75      0.925     0.828      0.889    0
         0.357     0.127      0.481     0.357     0.41       0.746    1
         0.608     0.149      0.621     0.608     0.615      0.826    2
         0.422     0.013      0.692     0.422     0.524      0.914    3
WAvg.    0.662     0.159      0.643     0.662     0.644      0.837

=== Confusion Matrix ===

   a   b   c   d   <-- classified as
 380  16  15   0 |   a = 0
 100  90  60   2 |   b = 1
  27  77 177  10 |   c = 2
   0   4  33  27 |   d = 3
```

D.1.1.4 SMO

```
=== Run information ===

Scheme:weka.classifiers.functions.SMO -C 1.0 -L 0.001 -P 1.0E-12 -N 0 -V -1 -W
 ↪  1 -K "weka.classifiers.functions.supportVector.PolyKernel -C 250007 -E
 ↪  1.0"
Relation:     persondata
Instances:    1018
Attributes:   4
              npeople
              numactive
              numconnected
              sizeconnected
Test mode:10-fold cross-validation

=== Classifier model (full training set) ===

SMO

Kernel used:
  Linear Kernel: K(x,y) = <x,y>

Classifier for classes: 0, 1

BinarySMO

Machine linear: showing attribute weights, not support vectors.

         4.7748 * (normalized) numactive
 +       0.1637 * (normalized) numconnected
 +       1.3126 * (normalized) sizeconnected
 -       1.245

Number of kernel evaluations: 16017 (72.105% cached)

Classifier for classes: 0, 2

BinarySMO

Machine linear: showing attribute weights, not support vectors.

         5.3248 * (normalized) numactive
 +       0.2183 * (normalized) numconnected
 +      -0.9654 * (normalized) sizeconnected
 -       1.2858

Number of kernel evaluations: 6722 (54.43% cached)
```

```
Classifier for classes: 0, 3

BinarySMO

Machine linear: showing attribute weights, not support vectors.

         4.1027 * (normalized) numactive
 +       0.9892 * (normalized) numconnected
 +       1.5031 * (normalized) sizeconnected
 -       2.5291

Number of kernel evaluations: 1053 (69.645% cached)

Classifier for classes: 1, 2

BinarySMO

Machine linear: showing attribute weights, not support vectors.

         6.7667 * (normalized) numactive
 +       1.8327 * (normalized) numconnected
 +      -5.9496 * (normalized) sizeconnected
 -       1.5238

Number of kernel evaluations: 6180 (65.167% cached)

Classifier for classes: 1, 3

BinarySMO

Machine linear: showing attribute weights, not support vectors.

         4.672  * (normalized) numactive
 +      -0.1629 * (normalized) numconnected
 +       0.3922 * (normalized) sizeconnected
 -       2.6961

Number of kernel evaluations: 1687 (59.747% cached)

Classifier for classes: 2, 3

BinarySMO

Machine linear: showing attribute weights, not support vectors.

         0.5273 * (normalized) numactive
 +      -0.4524 * (normalized) numconnected
 +       2.7006 * (normalized) sizeconnected
 -       1.8285
```

```
Number of kernel evaluations: 3332 (53.353% cached)



Time taken to build model: 0.07 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances         585               57.4656 %
Incorrectly Classified Instances       433               42.5344 %
Kappa statistic                          0.3603
Mean absolute error                      0.297
Root mean squared error                  0.3795
Relative absolute error                 86.0431 %
Root relative squared error             91.3677 %
Total Number of Instances             1018

=== Detailed Accuracy By Class ===

       TP Rate   FP Rate   Precision   Recall  F-Measure   ROC Area  Class
         0.74      0.338      0.597     0.74      0.661      0.777    0
         0.353     0.167      0.41      0.353     0.38       0.625    1
         0.649     0.132      0.663     0.649     0.656      0.774    2
         0.047     0.004      0.429     0.047     0.085      0.855    3
WAvg.    0.575     0.216      0.559     0.575     0.554      0.743

=== Confusion Matrix ===

   a   b   c   d   <-- classified as
 304 101   6   0 |   a = 0
 128  89  35   0 |   b = 1
  76  22 189   4 |   c = 2
   1   5  55   3 |   d = 3
```

D.1.1.5 J48

```
=== Run information ===

Scheme:weka.classifiers.trees.J48 -C 0.25 -M 2
Relation:     persondata
Instances:    1018
Attributes:   4
              npeople
              numactive
              numconnected
              sizeconnected
Test mode:10-fold cross-validation

=== Classifier model (full training set) ===

J48 pruned tree
------------------

numactive <= 4
|   numactive <= 2: 0 (351.37/45.0)
|   numactive > 2
|   |   numconnected <= 1: 1 (3.28/0.28)
|   |   numconnected > 1
|   |   |   numactive <= 3: 1 (16.42/6.42)
|   |   |   numactive > 3: 0 (15.32/3.0)
numactive > 4
|   numactive <= 20
|   |   sizeconnected <= 7
|   |   |   numactive <= 7
|   |   |   |   sizeconnected <= 5: 2 (18.61/5.61)
|   |   |   |   sizeconnected > 5: 1 (54.73/8.73)
|   |   |   numactive > 7: 2 (88.66/17.66)
|   |   sizeconnected > 7
|   |   |   numconnected <= 1: 1 (95.23/21.23)
|   |   |   numconnected > 1
|   |   |   |   sizeconnected <= 13
|   |   |   |   |   numconnected <= 2
|   |   |   |   |   |   sizeconnected <= 12: 2 (20.8/4.8)
|   |   |   |   |   |   sizeconnected > 12: 1 (13.14/4.14)
|   |   |   |   |   numconnected > 2
|   |   |   |   |   |   numactive <= 16: 0 (22.99/10.0)
|   |   |   |   |   |   numactive > 16
|   |   |   |   |   |   |   numactive <= 17: 0 (2.19/1.0)
|   |   |   |   |   |   |   numactive > 17: 1 (6.57/1.57)
|   |   |   |   sizeconnected > 13: 1 (49.26/14.26)
|   numactive > 20
|   |   numactive <= 36
|   |   |   sizeconnected <= 15: 2 (157.63/30.63)
```

```
|   |   |   sizeconnected > 15
|   |   |   |   numactive <= 29
|   |   |   |   |   sizeconnected <= 23
|   |   |   |   |   |   numactive <= 22
|   |   |   |   |   |   |   sizeconnected <= 17: 0 (2.19/1.0)
|   |   |   |   |   |   |   sizeconnected > 17: 1 (6.57/2.57)
|   |   |   |   |   |   numactive > 22
|   |   |   |   |   |   |   numactive <= 26: 2 (3.28/1.28)
|   |   |   |   |   |   |   numactive > 26: 0 (2.19/1.0)
|   |   |   |   |   sizeconnected > 23: 3 (8.76/1.76)
|   |   |   |   numactive > 29: 2 (44.88/15.88)
|   |   numactive > 36: 3 (33.93/9.93)

Number of Leaves  :         22

Size of the tree :         43


Time taken to build model: 0.06 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances         844               82.9077 %
Incorrectly Classified Instances       174               17.0923 %
Kappa statistic                          0.7487
Mean absolute error                      0.1731
Root mean squared error                  0.2878
Relative absolute error                 50.1407 %
Root relative squared error             69.3014 %
Total Number of Instances             1018

=== Detailed Accuracy By Class ===

       TP Rate   FP Rate   Precision   Recall  F-Measure   ROC Area  Class
         0.929     0.102      0.86      0.929     0.894      0.925    0
         0.71      0.063      0.789     0.71      0.747      0.855    1
         0.873     0.073      0.827     0.873     0.849      0.91     2
         0.453     0.012      0.725     0.453     0.558      0.87     3
WAvg.    0.829     0.078      0.825     0.829     0.824      0.9

=== Confusion Matrix ===

   a   b   c   d   <-- classified as
 382  23   5   1 |   a = 0
  52 179  19   2 |   b = 1
  10  19 254   8 |   c = 2
   0   6  29  29 |   d = 3
```

D.1.1.6 KStar

```
=== Run information ===

Scheme:weka.classifiers.lazy.KStar -B 20 -M a
Relation:     thirdexp-nominal
Instances:    1018
Attributes:   4
              npeople
              numactive
              numconnected
              sizeconnected
Test mode:10-fold cross-validation

=== Classifier model (full training set) ===

KStar Beta Verion (0.1b).
Copyright (c) 1995-97 by Len Trigg (trigg@cs.waikato.ac.nz).
Java port to Weka by Abdelaziz Mahoui (am14@cs.waikato.ac.nz).

KStar options : -B 20 -M a

Time taken to build model: 0 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances         841               82.613  %
Incorrectly Classified Instances       177               17.387  %
Kappa statistic                          0.7441
Mean absolute error                      0.1764
Root mean squared error                  0.2853
Relative absolute error                 51.1162 %
Root relative squared error             68.6796 %
Total Number of Instances             1018

=== Detailed Accuracy By Class ===

       TP Rate   FP Rate   Precision   Recall  F-Measure   ROC Area  Class
         0.915     0.089      0.874     0.915     0.894      0.935    0
         0.746     0.072      0.774     0.746     0.76       0.87     1
         0.88      0.085      0.805     0.88      0.841      0.923    2
         0.328     0.006      0.778     0.328     0.462      0.923    3
WAvg.    0.826     0.078      0.824     0.826     0.818      0.915

=== Confusion Matrix ===

   a   b   c   d   <-- classified as
 376  30   5   0 |   a = 0
```

```
44 188  20   0 |   b = 1
10  19 256   6 |   c = 2
 0   6  37  21 |   d = 3
```

D.1.2 Numeric Results

D.1.2.1 Multilayer Perceptron

```
=== Run information ===

Scheme:weka.classifiers.functions.MultilayerPerceptron -L 0.3 -M 0.2 -N 500 -V
 ↪  0 -S 0 -E 20 -H a
Relation:     persondata
Instances:    1018
Attributes:   4
              npeople
              numactive
              numconnected
              sizeconnected
Test mode:10-fold cross-validation

=== Classifier model (full training set) ===

Linear Node 0
    Inputs    Weights
    Threshold    -0.948400203247411
    Node 1    -0.5404909952916884
    Node 2    1.216178867266227
Sigmoid Node 1
    Inputs    Weights
    Threshold    -0.43529405839714014
    Attrib numactive    -5.375212304536006
    Attrib numconnected    8.485535675559154
    Attrib sizeconnected    10.222854781726667
Sigmoid Node 2
    Inputs    Weights
    Threshold    -1.6694708298582563
    Attrib numactive    20.69453148975731
    Attrib numconnected    -4.263624121611814
    Attrib sizeconnected    -17.140018798993825
Class
    Input
    Node 0


Time taken to build model: 0.27 seconds

=== Cross-validation ===
=== Summary ===

Correlation coefficient                  0.6865
Mean absolute error                      0.5969
Root mean squared error                  0.7768
```

```
Relative absolute error                72.7731 %
Root relative squared error            79.9255 %
Total Number of Instances            1018
```

D.1.2.2 IBk

```
=== Run information ===

Scheme:weka.classifiers.lazy.IBk -K 1 -W 0 -A
 ↪  "weka.core.neighboursearch.LinearNNSearch -A
 ↪  \"weka.core.EuclideanDistance -R first-last\""
Relation:     persondata
Instances:    1018
Attributes:   4
              npeople
              numactive
              numconnected
              sizeconnected
Test mode:10-fold cross-validation

=== Classifier model (full training set) ===

IB1 instance-based classifier
using 1 nearest neighbour(s) for classification


Time taken to build model: 0 seconds

=== Cross-validation ===
=== Summary ===

Correlation coefficient                  0.3194
Mean absolute error                      0.7674
Root mean squared error                  1.1947
Relative absolute error                 93.5545 %
Root relative squared error            122.9183 %
Total Number of Instances             1018
```

D.1.2.3 Linear Regression

```
=== Run information ===

Scheme:weka.classifiers.functions.LinearRegression -S 0 -R 1.0E-8
Relation:     persondata
Instances:    1018
Attributes:   4
              npeople
              numactive
              numconnected
              sizeconnected
Test mode:10-fold cross-validation

=== Classifier model (full training set) ===


Linear Regression Model

npeople =

      0.0783 * numactive +
     -0.0616 * numconnected +
     -0.0331 * sizeconnected +
      0.4923

Time taken to build model: 0.01 seconds

=== Cross-validation ===
=== Summary ===

Correlation coefficient                  0.7339
Mean absolute error                      0.5085
Root mean squared error                  0.6589
Relative absolute error                 61.9941 %
Root relative squared error             67.7949 %
Total Number of Instances             1018
```

D.1.2.4 Decision Stump

```
=== Run information ===

Scheme:weka.classifiers.trees.DecisionStump
Relation:     persondata
Instances:    1018
Attributes:   4
              npeople
              numactive
              numconnected
              sizeconnected
Test mode:10-fold cross-validation

=== Classifier model (full training set) ===

Decision Stump

Classifications

sizeconnected <= 3.5 : 0.14788732394366197
sizeconnected > 3.5 : 1.657439446366782
sizeconnected is missing : 0.15771812080536912


Time taken to build model: 0.01 seconds

=== Cross-validation ===
=== Summary ===

Correlation coefficient                  0.7649
Mean absolute error                      0.4756
Root mean squared error                  0.6249
Relative absolute error                 57.9858 %
Root relative squared error             64.291  %
Total Number of Instances             1018
```

APPENDIX E

Physical Form

To enable the prototype to be easily mounted on the ceiling, the prototype was placed on a flat board with feet that would enable it to be screwed into a pole, and the pole extended to jam the sensor against the ceiling and the floor using the pole (Figure E.2 on the following page, Figure E.1). Due to a wireless module and battery pack being added to the Raspberry Pi, it was feasible for the sensor to operate entirely wirelessly for several hours. However, in most cases it was more convenient to operate using wired power and Ethernet.

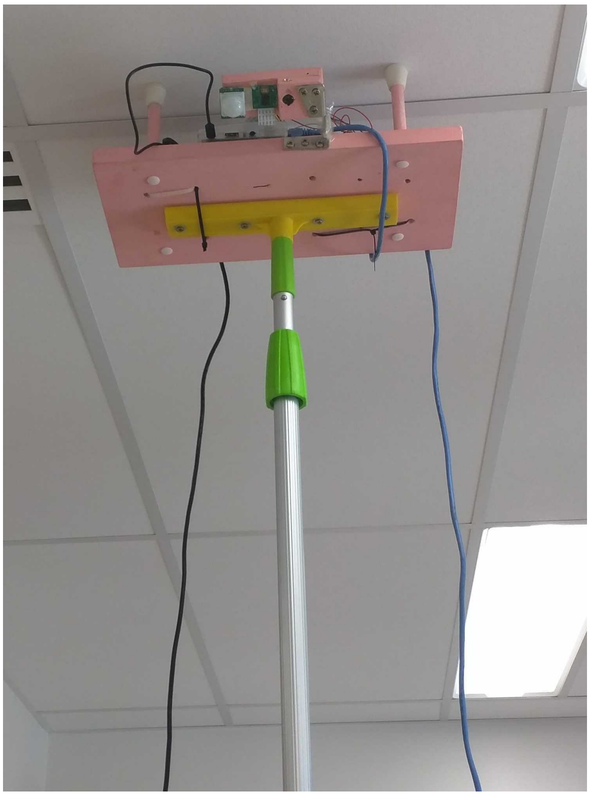
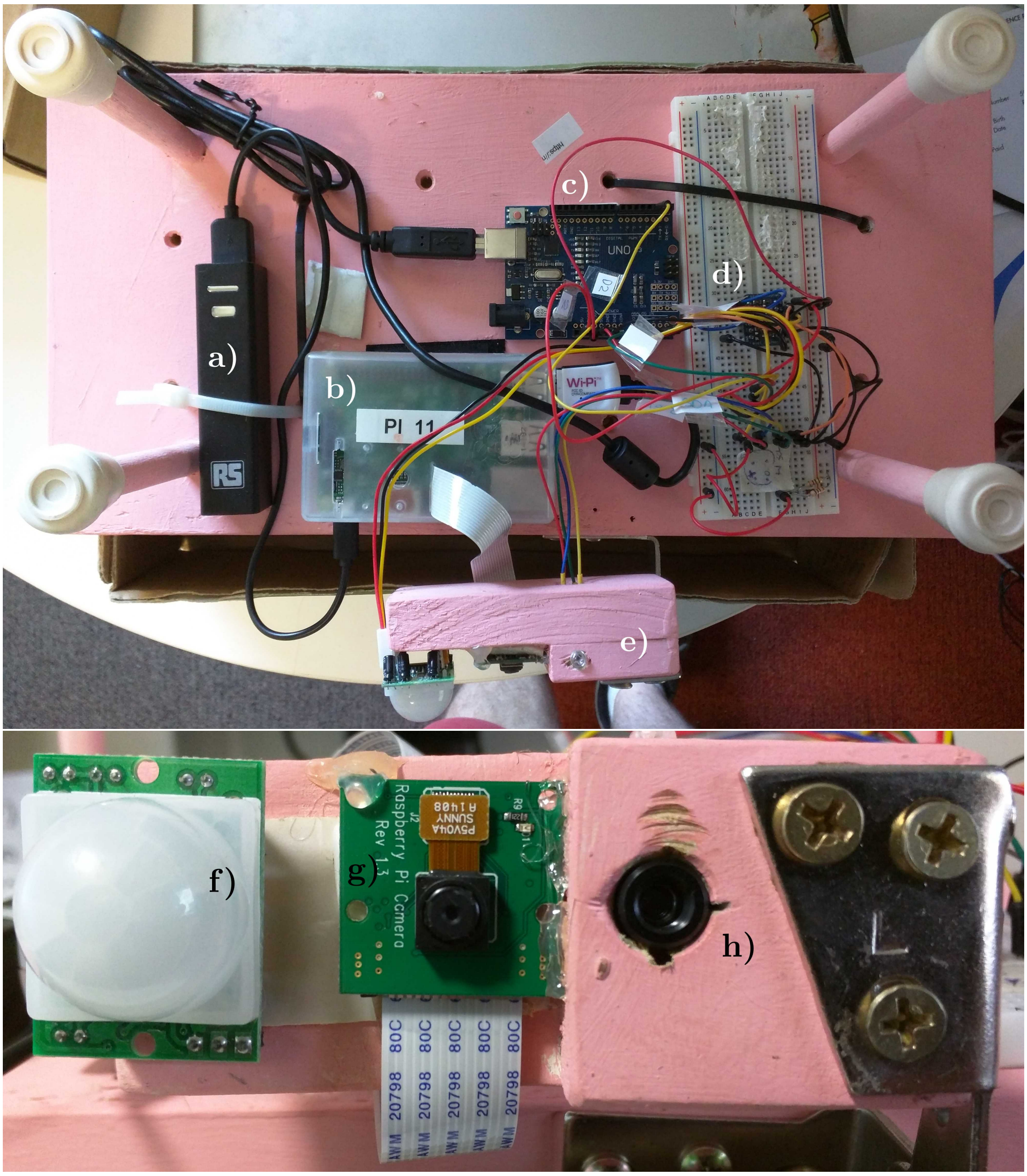


Figure E.1: Prototype in action



a) Battery pack
b) Raspberry Pi
c) Arduino
d) Level-shifting circuitry
e) Movable sensor mount
f) PIR
g) Camera
h) Melexis MLX90620 (*Melexis*)

Figure E.2: Prototype Physical Form

Bibliography

[1] ADAFRUIT. 4-channel I2C-safe bi-directional logic level converter - BSS138 (product ID 757). http://www.adafruit.com/product/757. Accessed: 2015-01-07.

[2] ADAFRUIT. PIR (motion) sensor (product ID 189). http://www.adafruit.com/product/189. Accessed: 2015-02-08.

[3] ARDUINO FORUMS. Arduino and MLX90620 16X4 pixel IR thermal array. http://forum.arduino.cc/index.php/topic,126244.0.html, 2012. Accessed: 2015-01-07.

[4] ATZORI, L., IERA, A., AND MORABITO, G. The internet of things: A survey. *Computer networks 54*, 15 (2010), 2787–2805.

[5] AUSTRALIAN BUREAU OF STATISTICS. Household water and energy use, Victoria: Heating and cooling. Tech. Rep. 4602.2, October 2011. Retrieved October 6, 2014 from http://www.abs.gov.au/ausstats/abs@.nsf/0/85424ADCCF6E5AE9CA257A670013AF89.

[6] BALAJI, B., XU, J., NWOKAFOR, A., GUPTA, R., AND AGARWAL, Y. Sentinel: occupancy based HVAC actuation using existing WiFi infrastructure within commercial buildings. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems* (2013), ACM, p. 17.

[7] BELTRAN, A., ERICKSON, V. L., AND CERPA, A. E. ThermoSense: Occupancy thermal based sensing for HVAC control. In *Proceedings of the 5th ACM Workshop on Embedded Systems For Energy-Efficient Buildings* (2013), ACM, pp. 1–8.

[8] CHAN, M., CAMPO, E., ESTÈVE, D., AND FOURNIOLS, J.-Y. Smart homes - current features and future perspectives. *Maturitas 64*, 2 (2009), 90–97.

[9] ERICKSON, V. L., ACHLEITNER, S., AND CERPA, A. E. POEM: Power-efficient occupancy-based energy management system. In *Proceedings of the 12th international conference on Information processing in sensor networks* (2013), ACM, pp. 203–216.

[10] Fisk, W. J., Faulkner, D., and Sullivan, D. P. Accuracy of CO2 sensors in commercial buildings: a pilot study. Tech. Rep. LBNL-61862, Lawrence Berkeley National Laboratory, 2006. Retrieved October 6, 2014 from `http://eaei.lbl.gov/sites/all/files/LBNL-61862_0.pdf`.

[11] Guinard, D., Ion, I., and Mayer, S. In search of an internet of things service architecture: REST or WS-*? a developers perspective. In *Mobile and Ubiquitous Systems: Computing, Networking, and Services*. Springer, 2012, pp. 326–337.

[12] Guinard, D., Trifa, V., Mattern, F., and Wilde, E. From the internet of things to the web of things: Resource-oriented architecture and best practices. In *Architecting the Internet of Things*. Springer, 2011, pp. 97–129.

[13] Gupta, M., Intille, S. S., and Larson, K. Adding gps-control to traditional thermostats: An exploration of potential energy savings and design challenges. In *Pervasive Computing*. Springer, 2009, pp. 95–114.

[14] Hailemariam, E., Goldstein, R., Attar, R., and Khan, A. Real-time occupancy detection using decision trees with multiple sensor types. In *Proceedings of the 2011 Symposium on Simulation for Architecture and Urban Design* (2011), Society for Computer Simulation International, pp. 141–148.

[15] Hnat, T. W., Griffiths, E., Dawson, R., and Whitehouse, K. Doorjamb: unobtrusive room-level tracking of people in homes using doorway sensors. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems* (2012), ACM, pp. 309–322.

[16] Kleiminger, W., Beckel, C., Dey, A., and Santini, S. Inferring household occupancy patterns from unlabelled sensor data. Tech. Rep. 795, ETH Zurich, 2013. Retrieved October 6, 2014 from `http://eaei.lbl.gov/sites/all/files/LBNL-61862_0.pdf`.

[17] Kleiminger, W., Beckel, C., Staake, T., and Santini, S. Occupancy detection from electricity consumption data. In *Proceedings of the 5th ACM Workshop on Embedded Systems For Energy-Efficient Buildings* (2013), ACM, pp. 1–8.

[18] Kovatsch, M. CoAP for the web of things: from tiny resource-constrained devices to the web browser. In *Proceedings of the 2013 ACM conference on Pervasive and ubiquitous computing adjunct publication* (2013), ACM, pp. 1495–1504.

[19] Li, N., Calis, G., and Becerik-Gerber, B. Measuring and monitoring occupancy with an RFID based system for demand-driven HVAC operations. *Automation in construction 24* (2012), 89–99.

[20] Melexis. Datasheet IR thermometer 16X4 sensor array MLX90620. http://www.melexis.com/Asset/Datasheet-IR-thermometer-16X4-sensor-array-MLX90620-DownloadLink-6099.aspx, 2012. Accessed: 2015-01-07.

[21] Palattella, M. R., Accettura, N., Vilajosana, X., Watteyne, T., Grieco, L. A., Boggia, G., and Dohler, M. Standardized protocol stack for the internet of (important) things. *Communications Surveys & Tutorials, IEEE 15*, 3 (2013), 1389–1406.

[22] Serrano-Cuerda, J., Castillo, J. C., Sokolova, M. V., and Fernández-Caballero, A. Efficient people counting from indoor overhead video camera. In *Trends in Practical Applications of Agents and Multiagent Systems*. Springer, 2013, pp. 129–137.

[23] Shelby, Z., and Bormann, C. *6LoWPAN: The wireless embedded Internet*, vol. 43. John Wiley & Sons, 2011.

[24] Teixeira, T., Dublon, G., and Savvides, A. A survey of human-sensing: Methods for detecting presence, count, location, track, and identity. Tech. rep., Embedded Networks and Applications Lab (ENALAB), Yale University, 2010. Retrieved October 6, 2014 from http://www.eng.yale.edu/enalab/publications/human_sensing_enalabWIP.pdf.

[25] Winter, T., Thubert, P., Cisco Systems, Brandt, A., et al. RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks. RFC 6550, Internet Engineering Task Force, March 2012. Retrieved October 6, 2014 from http://tools.ietf.org/html/rfc6550.