Tracking with Constraints in a Web of Sensors

by

Brian Kerry Dunagan

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2004 [June 2004]

Author ....
Department of Electrical Engineering and Computer Science
May 20, 2004

Certified by................
Trevor Darrell
Associate Professor
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Tracking with Constraints in a Web of Sensors

by

Brian Kerry Dunagan

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2004, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

With the dramatic fall in price of electronics over the past several years, large-scale networks of sensors are steadily becoming more feasible. The goal of this research project was to deploy a sensor network for data collection of human trajectories and to develop and test a method for taking walls into account using a penalty function in an ongoing research project in trajectory tracking model. There were two deployed sensor networks, one with distance sensors and one with cameras, and the camera network was used to collect data for the two papers. The trajectory tracking model was modified to incorporate wall constraints to enable exploration of more realistic scenarios, with encouraging preliminary results.

Thesis Supervisor: Trevor Darrell
Title: Associate Professor

Acknowledgments

I'd like to thank Ali Rahimi for letting me add to his doctoral work and giving me the opportunity to help publish the research. He provided me with all the help and direction I needed. My thesis is a small piece of his. I'd also like to thank Trevor Darrell for overseeing my contributions, Ron Wiken for building ceiling mounts for the iPaqs, and Project Oxygen for its hardware donations. Finally, I'd like to thank my family and my friends for their support.

Contents

1 Introduction **11**

1.1 Motivation . . . 11

1.1.1 Traffic Patterns . . . 12

1.1.2 Security . . . 12

1.1.3 Network Self-Calibration . . . 12

1.2 Goal . . . 13

1.3 Related Works . . . 13

1.3.1 Sensor Networks . . . 13

1.3.2 Obstacle Avoidance . . . 14

1.4 Structure . . . 14

2 Sensor Networks **15**

2.1 Network Design . . . 15

2.2 Mote Network . . . 16

2.2.1 Overview . . . 17

2.2.2 Adding a Distance Sensor . . . 18

2.2.3 Programming in TinyOS . . . 19

2.2.4 Discussion . . . 20

2.2.5 Future Work . . . 21

2.3 iPaq Network . . . 22

2.3.1 Setup . . . 23

2.3.2 Discussion . . . 25

2.4 Comparison of Networks . . . 26

2.5 Results 26

2.5.1 Measuring Bearing 27

2.5.2 Measuring Location 27

3 Wall Constraints **29**

3.1 Mathematical Programming Overview 29

3.2 Linear Constraints 30

3.3 Objective Function 30

3.3.1 Dot Products 31

3.3.2 The penalty 32

3.4 Implementation 33

3.4.1 First Version 33

3.4.2 Second Version 33

3.5 Results 34

3.6 Discussion 34

3.7 Future Work 35

4 Conclusion **41**

A TinyOS Distance Module **43**

B Matlab Code **47**

List of Figures

2-1 A Mote: the battery and processing unit (right) and the sensor board with distance sensor attached (left). 17

2-2 Two Motes with distance sensors attached. 18

2-3 The configuration module for the distance sensor in TinyOS. 19

2-4 Five Motes, one base station board, and one detached distance sensor. . . 20

2-5 A circuit diagram for a Mote sensor board connector. The distance sensor was connected to the circled pins. 21

2-6 The distance sensor produced noisy data. This is a typical snapshot of one hundred seconds. The middle line represents the "true" value, four feet in this case. 22

2-7 Compaq iPaq. 23

2-8 iPaq with Backpaq attached. 23

2-9 iPaq network viewing traffic horizontally. 24

2-10 iPaq network viewing traffic vertically. 24

2-11 A result with data collected from the iPaq network with horizontal cameras. The thin line is the actual trajectory, and the thick line is the perceived one. The grey cones are the fields of view. The black bars are the walls. Notice that the generated trajectory goes directly through three walls. 27

2-12 A result with data collected from the iPaq network with vertical cameras. The dotted line is the recovered trajectory. The grey boxes are the recovered locations and orientations of the sensors, and the dotted boxes are the actual locations and orientations of the sensors. 28

3-1 The three subdomains of the constraint problem. 31
3-2 A scenario with one wall and two observations. 36
3-3 A scenario with one wall and four observations. 37
3-4 A scenario with two walls and three observations. 38
3-5 A scenario with three walls and four observations. 39
3-6 A scenario with five walls and five observations. 40

Chapter 1

Introduction

With the dramatic fall in price of electronics over the past several years, large-scale networks of sensors are steadily becoming more feasible. One use for these is person tracking. Here, the goal is to observe a person at various locations and collect the observations.

One use for that collected data is generating a complete trajectory for a moving person. This entails looking at where the person was when he was observed by the network, noting the time, and reconstructing a likely trajectory, or path, that he could have taken, one that includes those observed locations. This problem of trajectory tracking is essentially an inference problem. The goal is to look at the observed points in a person's trajectory and interpolate the other points that went unseen.

1.1 Motivation

There are a number of applications for this sort of infrastructure and many scenarios where this research would prove valuable. Each of the examples below highlights usefulness of a sensor network that tracks trajectories.

1.1.1 Traffic Patterns

A university just finished a building on campus, and the administration and the architect are interested in how traffic patterns evolve over time. They considered hiring a consulting firm to study the traffic, but for their long time frame, that solution is financially infeasible. Instead, the administration invests in a large number of sensors equipped with cameras. The cameras are installed at main throughways and take snapshots of the current traffic. After identifying the people in each snapshot [6, 9, 10], the network is able to calculate the trajectory of every person and aggregate the data into a general pattern.

1.1.2 Security

A security company would like to keep track of where its employees are at all times, so they install a network of radio frequency identification (RFID) readers and an RFID tag give each of their employees. The reader senses any tag that passes by, acting as a proximity detector. Using these observations and their time stamps, the network can infer a coarse likely trajectory for each person and build up a database of where each employee has been.

1.1.3 Network Self-Calibration

There is a newly-installed network of a thousand sensors in an electronics fabrication factory that monitor temperature, humidity, and human presence. The problem is the individual sensors do not know where they are located with respect to the rest of the network. Measuring the exact three-dimensional coordinates for every one of the thousand sensors would be expensive and time-consuming. A better solution is to leverage the presence sensor and use its data to let the network self-calibrate [12]. A person could walk around the building until all the sensors had seen him several times, and the sensor network could infer the specific locations of each sensor to varying degrees of accuracy.

1.2 Goal

The goal of this research project was to deploy a sensor network for data collection of human trajectories and to develop and test a method for taking walls into account using a penalty function in an ongoing research project in trajectory tracking model. There were two deployed sensor networks, one with distance sensors (§2.2) and one with cameras (§2.3). The camera network was used to collect data for the two papers (§2.5). The trajectory tracking model was modified to incorporate wall constraints to enable exploration of more realistic scenarios (§3).

1.3 Related Works

Given the dual goals of this project, there is a distinct body of work that relates to each.

1.3.1 Sensor Networks

U.C. Berkeley has been a major contributor to the field. Research groups there have designed and implemented a modular sensor [14] and a functional operating system [4], both of which are now used extensively in the sensor network community and are packaged as a commercial product [2]. Culler and Whitehouse have explored localization techniques within a sensor network [15]. Their goal was to generate a local (x, y) coordinate system for the sensor network and link that to a known coordinate system. In addition, Culler deployed a network on Great Duck Island to monitor weather and nesting behavior [3]. These are sensors that need to run for months or years without needing maintenance. They are also looking beyond the macroscopic implementations to the design of microscopic sensors [5].

Balakrishnan et al. [11] set up a location-support system for in-building, mobile, location-dependent applications. There are a set of sensors, beacons, that broadcast their two-dimensional coordinate, and another set of sensors, listeners, triangulate their own position based on the broadcast messages.

1.3.2 Obstacle Avoidance

T. Schouwenaars et al. have done work in obstacle avoidance, a superset of the wall avoidance problem [7]. Their goal was to build a helicopter that could be fed a start point and an end point and could then autonomously fly from one to the other without colliding with an obstacle. It is important to note that their research question, "Where should the helicopter go?" differs slightly from this project's question, "Where did the person go?"

Their approach to the obstacle avoidance problem was to set up four linear constraints, forming a solid box around the obstacle that the object's trajectory points could not touch, and ensured that all the trajectory points satisfy those. The pitfall of this approach was the boundary case when the discrete points of where the helicopter was planning to fly did not overlap with the box but the actual trajectory of the helicopter clipped a corner of the box. To avoid this, they expanded each obstacle's box by a certain amount related to the distance the helicopter could travel in a time step.

On the other hand, their approach is trivially scalable to three dimensions, while the approach described in this project is not (§3) . For their calculations, they used mixed integer linear programming and a piecewise linear approximation for their nonlinear cost function.

1.4 Structure

Section 2 describes the configuration of a Mote network (§2.2) and a iPaq network (§2.3). Section 3 details the problem of wall constraints, this project's solution to it, and a set of preliminary results. Section 4 concludes with a discussion on feasibility and practicality of these sensor networks and the wall constraint problem.

Chapter 2

Sensor Networks

This project was part of an ongoing investigation into trajectory tracking. The research had already produced an algorithm that generated likely trajectories given a set of observation. It had been tested on simulated data from Matlab, so the next step was to generate trajectories for the real world using experimental data. Two testbeds were created on which to collect this experimental data: a network of Motes (§2.2) and a network of iPaqs (§2.3).

2.1 Network Design

There were several design goals that guided the creation of both sensor networks. The research project was specifically directed at sparse sensor networks, or networks where the sensors' views do not overlap. In previous work, the networks have been intentionally made dense [1], with views overlapping, so that sensors could hand off tracked objects from one to another; the design constraint greatly simplified the task of inferring position and allowed for other avenues of research. However, the goal of this research project was tracking a human in a sparse network. One advantage to the sparse network is the incremental cost of deployment. There is no lower bound to the number of sensors needed for the algorithm to run, but adding them incrementally increases the trajectory's accuracy.

The central goal of the network was to provide the maximum number of observations per second. Thus, this project focused on that aspect instead of implementing any data routing protocols or on-sensor filtering mechanisms. Both of those are necessary for large-scale deployment of either sensor network, especially with Motes, but they were not a high priority for this small-scale testbed; the base station that collected all the data was in broadcast range of all of the deployed sensors, and the amount of data, on the order of kilobytes, was not the bottleneck.

One important facet of the trajectory algorithm is its processing mode. It was designed to process all the data en masse, not incrementally as new observations occurred. This relaxation allowed for two possibilities: a constant stream of data or bursts of data at regular intervals. Both options were explored but the final implementation of both networks sent data as the first, a constant stream.

Finally, there was an urgent need for a deployed network, so designing a sensor and fabricating a set of them was not a viable option. Fortunately, there are a number of sensors on the market or in development. Three fit the majority of the design requirements, though only one came equipped with a distance sensor. The first choice was Stacks, developed at the MIT Media Lab. They featured a modular platform and a high bandwidth of over one hundred kilobytes per second, though there was no high-level operating system written for it. Unfortunately, there were no spare sensors, and there was no time to assemble more. The second choice was Motes, developed by U.C. Berkeley and commercialized by Crossbow Technology. They also featured a modular design and were the de facto choice in the wireless sensor network research community. The third choice was iPaqs, sold by Compaq. They were small handheld computers. The first network was built out of Motes, because of their low cost and extensibility. The second network was build out of iPaqs, because of their availability and ease of configuration.

2.2 Mote Network

The Mote network required hardware and software engineering to add the distance sensor and to resolve several issues with time stamps. The sensor programming was

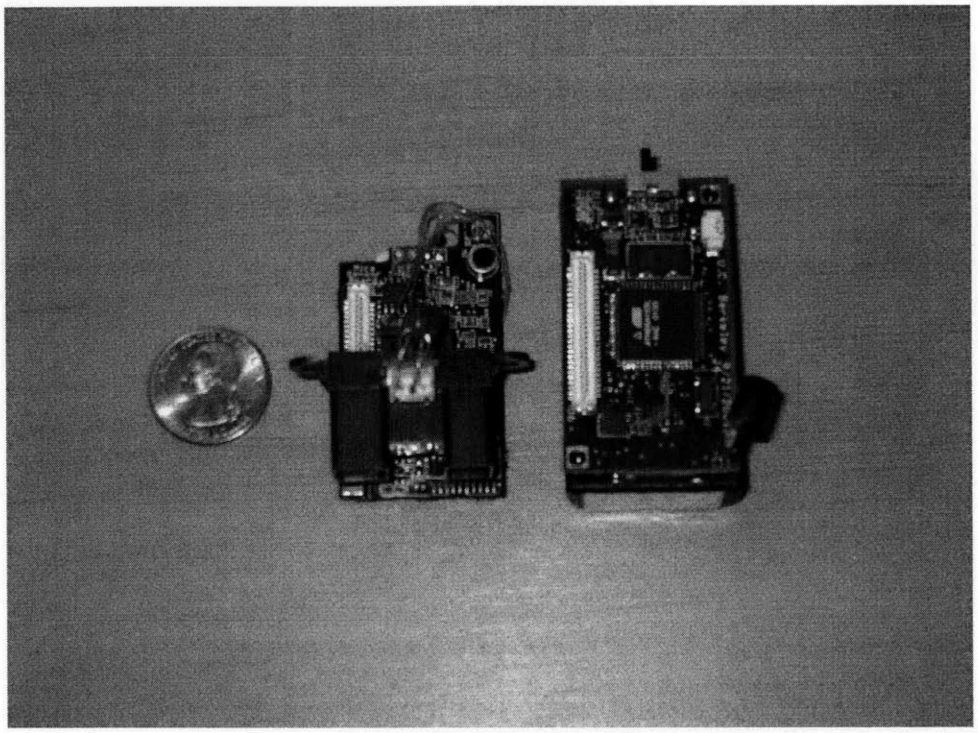


Figure 2-1: A Mote: the battery and processing unit (right) and the sensor board with distance sensor attached (left).

in a C variant, and interface and data manipulation on the PC was in Java.

2.2.1 Overview

A Mote was two inches long, one inch wide, and one inch tall. It was comprised of two pieces: a battery pack and processing unit and a sensor board (Figure 2-1). The central unit had a dual purpose. It could accept data from a sensor board and interact with other sensors over radio at 916 MHz. It could also be plugged into a peripheral board attached to a computer and act as a base station, broadcasting messages to the network and relaying data from the network to a computer. The sensor board was equipped with an array of sensors: a magnetic field sensor, thermometer, accelerometer, a microphone, and photoresistor. In addition, it included a "sounder" that produced a high-pitch squeal.

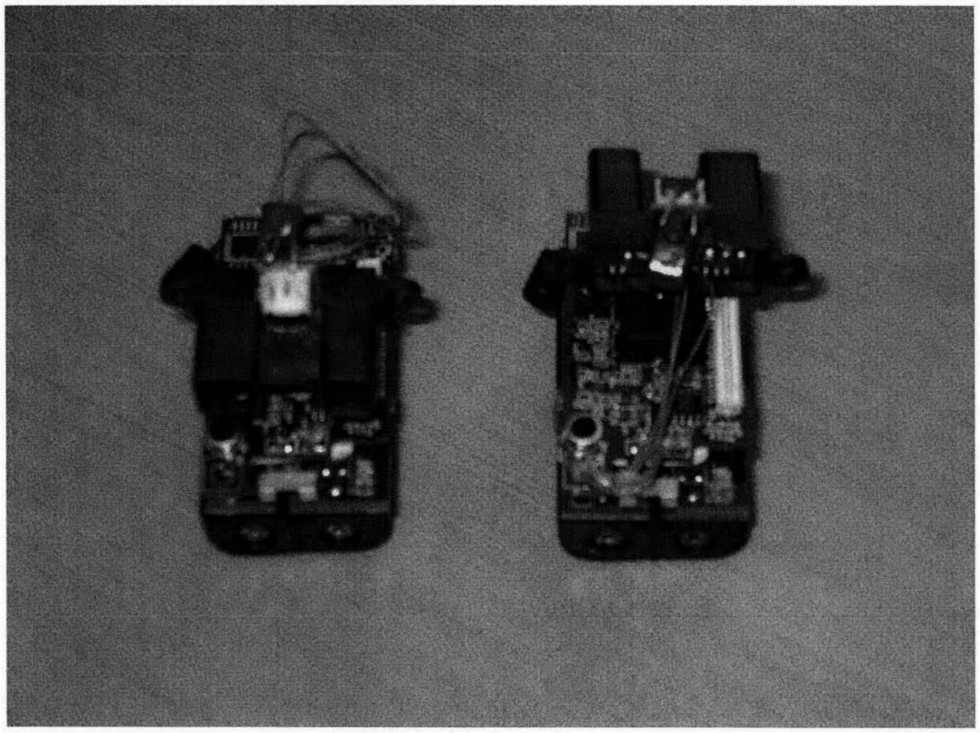


Figure 2-2: Two Motes with distance sensors attached.

2.2.2 Adding a Distance Sensor

Because the Mote did not come with a distance sensor, the first task was to find one to add. One popular compact distance sensor was Sharp's infrared sensor (GP2Y0A02YK). It had a range of twenty centimeters to 150 centimeters (or eight inches to six feet) and observed the world every forty milliseconds, or at twenty-five hertz. The key feature was its size as it needed to sit on top of a Mote.

The next step was to add it to the Mote. Luckily, most of the sensor boards did not have an accelerometer, so that ADC pin was free on the board. A connection was soldered between the ADC6 pin and the sensor's output pin, along with the necessary connections between the power and ground on both (Figure 2-5). Using an oscilloscope verified that all three connections were correct, though two capacitors were placed between the power and ground pins on the sensor for regulation. For convenience, the infrared sensor was attached to the top of the Mote with hot glue (Figure 2-2).

```
includes BrianMsg;
configuration Snet_stream {
  provides interface ProcessCmd;
}
implementation {
  components Main, Snet_streamM, LedsC, Accel as IR;
  components TimerC, ClockC;
  components GenericComm as Comm;
  Main.StdControl -> Snet_streamM;
  Snet_streamM.Leds -> LedsC;
  ProcessCmd = Snet_streamM.ProcessCmd;
  Snet_streamM.CommControl -> Comm;
  Snet_streamM.ReceiveMsg -> Comm.ReceiveMsg[AM_BRIANRECEIVEMSG];
  Snet_streamM.SendMsg -> Comm.SendMsg[AM_BRIANSENDMSG];
  Snet_streamM.Timer_sample -> TimerC.Timer[unique("Timer")];
  Snet_streamM.Timer_send -> TimerC.Timer[unique("Timer")];
  Snet_streamM.TimerControl -> TimerC;
  Snet_streamM.ADC -> IR.AccelX;
  Snet_streamM.SensorControl -> IR;
  Snet_streamM.Clock -> ClockC;
}
```

Figure 2-3: The configuration module for the distance sensor in TinyOS.

2.2.3 Programming in TinyOS

Researchers at U.C. Berkeley wrote a tiny operating system for Motes: TinyOS [4]. Implemented in NesC, a variant of C, it required only 176 bytes of memory, making it an ideal operating system for small sensors.

For this research project, the initial programming was straight-forward. The distance sensor needed a software module to interpret its signals and relay them to the networking module. There were a number of modules already designed for performing the same function for the other sensors, so the module for the distance sensor was modeled after those. To convey the simplicity of programming for TinyOS, the distance sensor's configuration module has been included (Figure 2-3), and the implementation of this module is in Appendix A.

The packets sent out by the networking component each contained three obser-

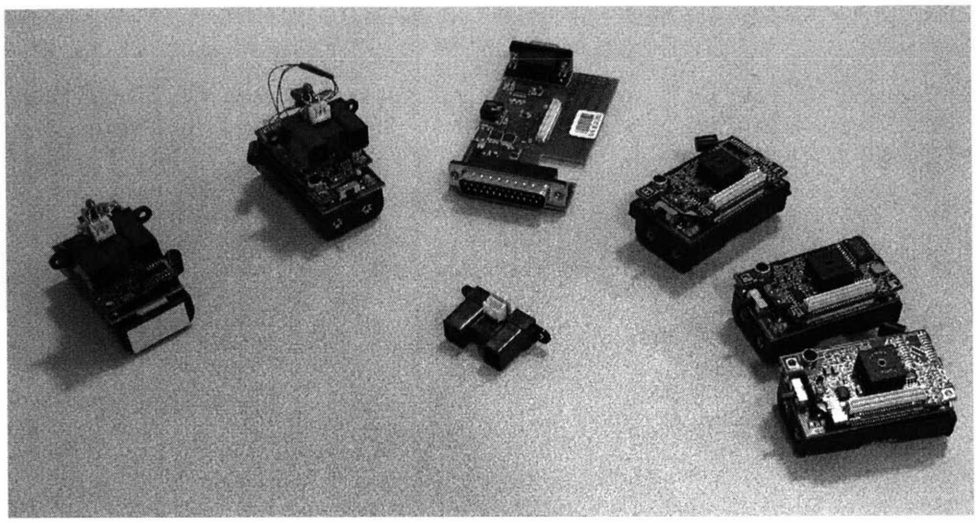


Figure 2-4: Five Motes, one base station board, and one detached distance sensor.

vations, with one byte for each distance sensor's reading and two bytes for each time stamp. The two-byte time stamp allowed the continuous stream of data to last for ten minutes at a clock rate of one hundred hertz, without help.

There were two caveats to consider with the time stamp. First, the network needed to operate continuously for days, not minutes. Because the timer ran out of space every ten minutes, the base station was programmed to broadcast a heartbeat that zeroed out the timers on all the Motes. Second, this broadcast had the added benefit of minimizing timer drift. Individual timers began to vary after a significant amount of time. These inaccuracies led to errors in the observations and eventually in the calculated trajectory. Repeatedly resetting them to a global value ensured the short-term drift was small and the long-term drift was zero.

2.2.4 Discussion

The Mote network was a success in a few ways. It was able to record twenty-five observations per second. In addition, while the data required a small amount of post processing, it was simple, though not necessarily accurate. Most of all, Motes had very small footprints.

However, there were more drawbacks to this network. Foremost, the sensor were

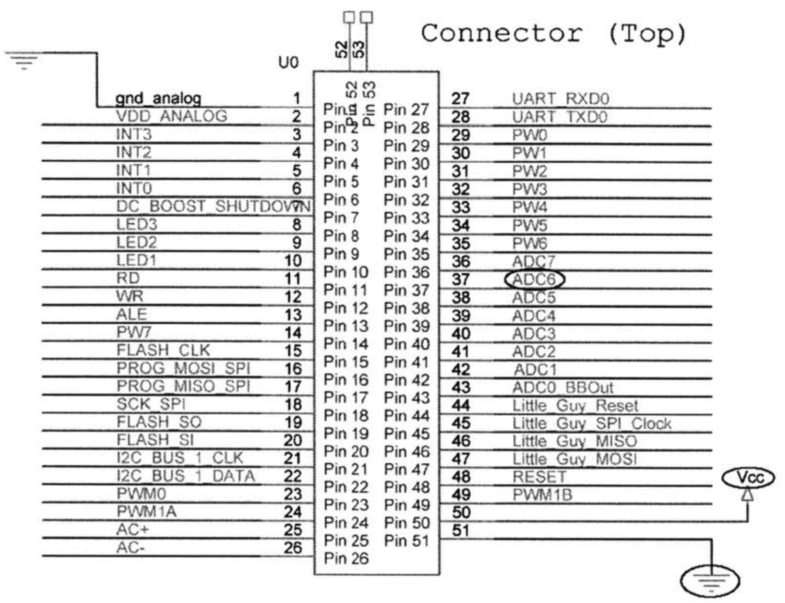


Figure 2-5: A circuit diagram for a Mote sensor board connector. The distance sensor was connected to the circled pins.

inconsistent. The data from the sensor was noisy (Figure 2-6), and post processing didn't necessarily result in an accurate reading. The broadcast range was relatively limited, thirty feet for line-of-sight and twenty feet if obscured. Finally, Motes did not provide much feedback for debugging. Each had three LEDs on-board to convey information on a small number of aspects of its operation, but they did not allow for very much information on the status of the Mote.

2.2.5 Future Work

The Mote network failed to meet the requirements of the project primarily because of the distance sensor's inaccuracy. The sensor generated very noisy data that was difficult to tease apart into measurements that were consistently accurate to within five feet. While the sensor could simply be used as a proximity detector, relaxing those requirements in the project's tracking algorithm would have increased its complexity and limited the accuracy in the resulting trajectories. Finding a better, albeit more expensive, sensor or increasing the accuracy of the current one would enable serious work with a Mote network in the future. We designed the second-generation network

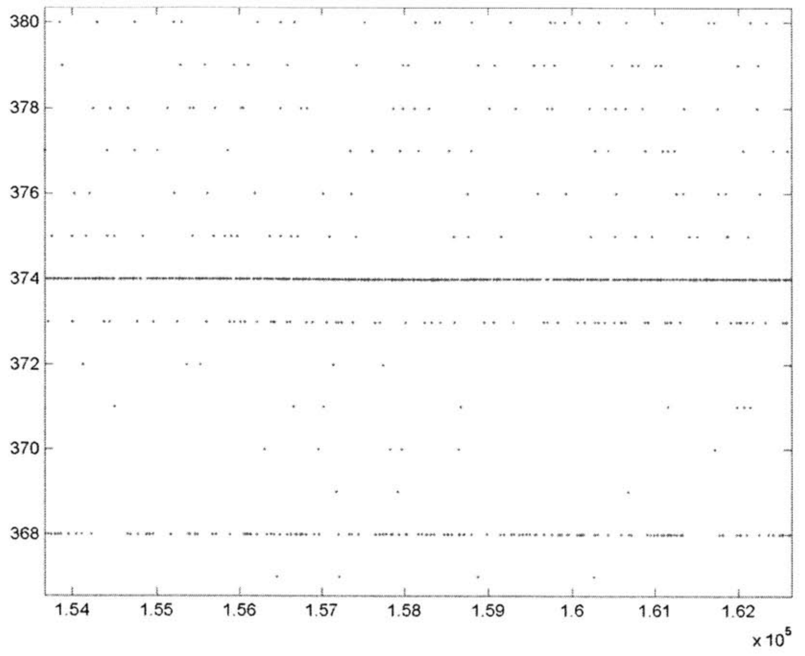


Figure 2-6: The distance sensor produced noisy data. This is a typical snapshot of one hundred seconds. The middle line represents the "true" value, four feet in this case.

---

using nodes with more powerful sensors.

2.3 iPaq Network

The iPaq network provided a higher level of abstraction for data collection. An iPaq (Figure 2-7) was a small handheld computer with an StrongARM processor sold by Compaq. The iPaq could be docked with an external module, a Backpaq (Figure 2-8), equipped with a 802.11b wireless network card and a tiny digital camera. Ten iPaqs and accompanying Backpaqs were supplied to the project through MIT's Project Oxygen, with ARM Linux 2.4.18-rmk3 installed. Unfortunately, the camera was limited to four frames a second, producing very few observations of a passerby.

Like the Mote network, the goal of the iPaq network was to collect experimental data on people walking past each sensor. The observations and timestamps were fed into the trajectory-tracking algorithm, and a likely trajectory was produced.

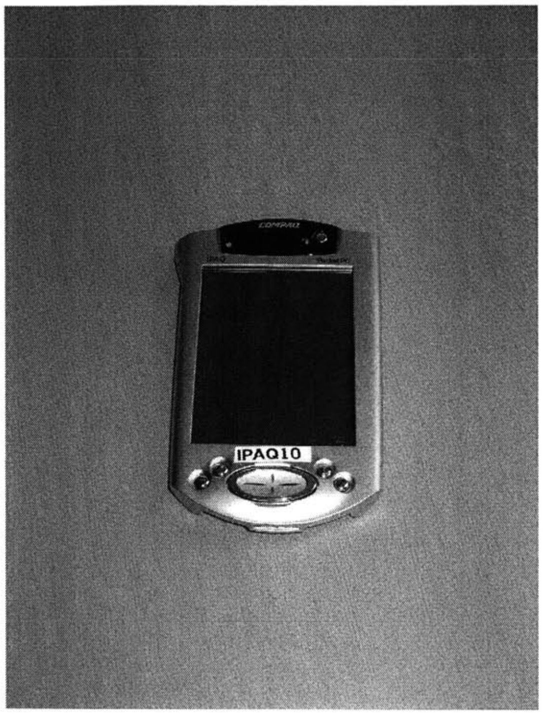


Figure 2-7: Compaq iPaq.

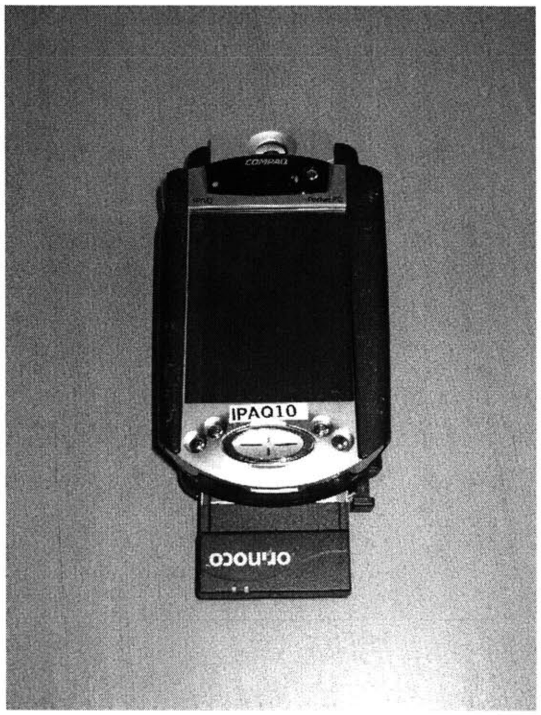


Figure 2-8: iPaq with Backpaq attached.

2.3.1 Setup

There were two main aspects of the iPaq to manage: physical placement and internal operation. The research project required multiple physical setups for the iPaq network, as there were multiple types of data desired, both horizontal (Figure 2-9) and vertical (Figure 2-10). Mounts were used used to secure the iPaqs to locations.

One prevalent issue was power. iPaqs are designed to run three hours by battery, so an alternate power source was necessary. The most practical solution was to run power cables to each of them.

In the vertical setup, the floor needed to represent ground truth. This constraint allowed the algorithm to determine a local coordinate system in only two dimensions, instead of three. However, it meant the iPaqs had to be facing directly downwards.

In the horizontal case, the iPaq's camera was used to obtain a distance measurement for the person being tracked. To determine that distance, it was straight-forward to use what was already known: the pixel width of the image (I), the focal length of the camera (F), and the actual width of the camera's field of view (FOV). With

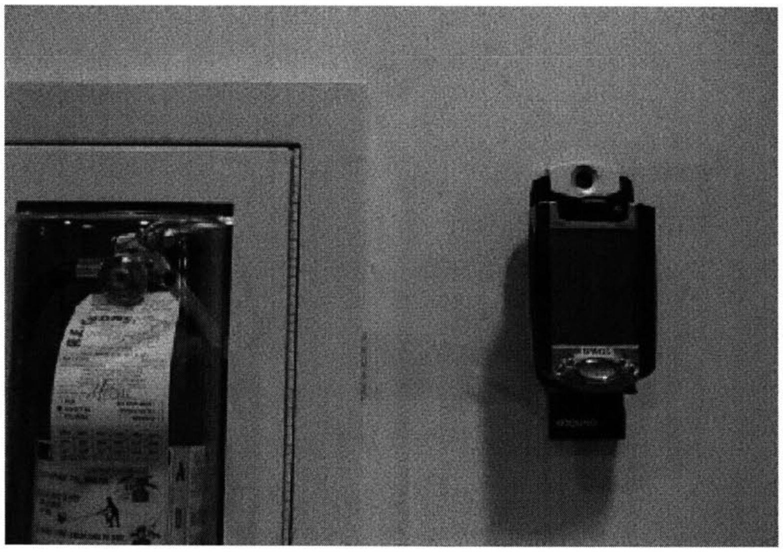


Figure 2-9: iPaq network viewing traffic horizontally.

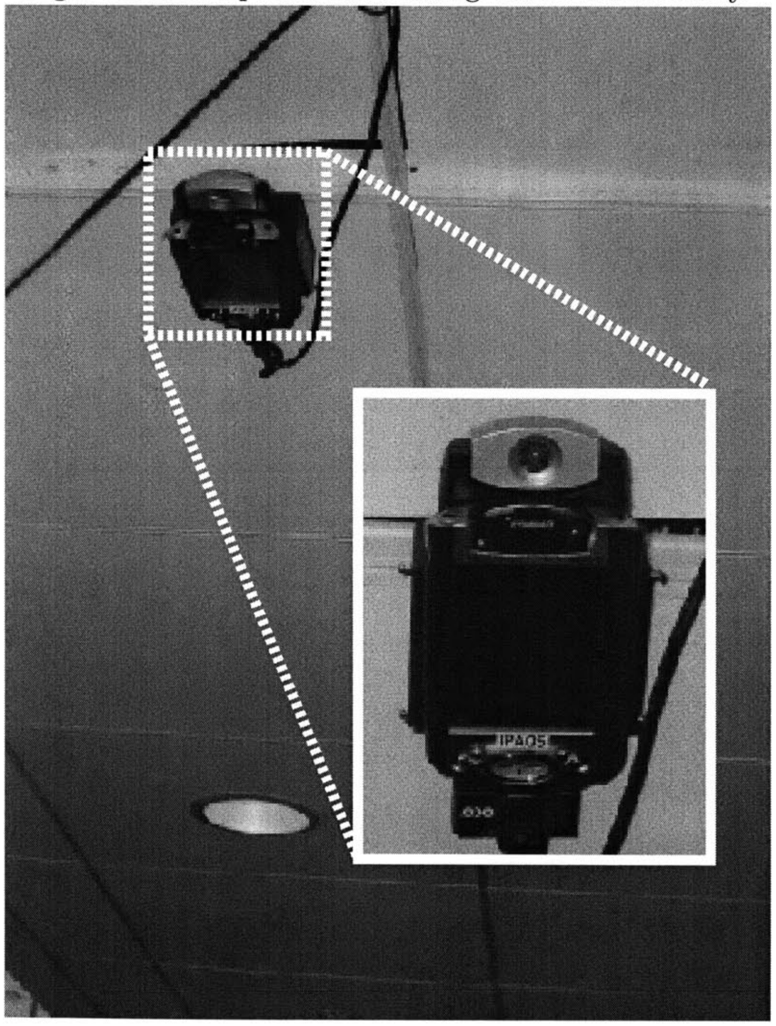


Figure 2-10: iPaq network viewing traffic vertically.

these, a simple ratio could be constructed, $\frac{F}{I} = \frac{x}{FOV}$, where x was the actual distance to the person. However, for this equation to work, the focal length was needed. Calculating that was very simple; fixing a person at a known distance produced the answer. Repeated several times, an approximate focal length was found.

Time stamps were a critical part of the observation data. The original algorithm did not incorporate them because the data had always been simulated. With the transition to experimental data, the C program collecting the data on the iPaq was updated so that time stamps accompanied to each observation.

The most useful part of the iPaq network, aside from the cameras, was the wireless cards accompanying each iPaq. They enabled a Secure Shell (SSH) connection to each of them, allowing the iPaqs to be configured remotely and in bulk.

The last issue was time synchronization. As with the Mote network, the time stamps of separate nodes of the network needed to be as close as possible so that they accurately represented the observed trajectory. A small script was created to aid in this task. It was copied onto each iPaq and used the Network Time Protocol (NTP) to synchronize its time with a global time. Because time drift is relatively small, running this script on each iPaq before every experiment kept the times as synchronized as needed.

2.3.2 Discussion

Using the iPaq provided a couple nice features. They each had an Internet Protocol (IP) address and allowed users to remotely connect with them through SSH. They also provided low-resolution images, allowing the program to accurately estimate the distance or relative location of the object.

On the other hand, because the data were images, they required extra processing to extract the location information. The iPaqs were power hungry, running out of battery power in 3 hours unless plugged in.

2.4 Comparison of Networks

The iPaq network proved to be the more useful one in terms of ease-of-use, distance accuracy, and deployment scale. Though the Motes footprints were smaller and the data was more concise, they were more difficult to interact with. Only the data from the iPaq network was used to determine human trajectories.

2.5 Results

The iPaq network was used for data collection to test the trajectory tracking model. As mentioned previously, there were two configurations of the network: horizontal and vertical. Each was used for observing a person's trajectory in an open space.

To generate a likely trajectory, the original tracking algorithm used two terms in its cost function. The first term, D, took the smoothness of the trajectory into account and penalized any jagged segments or sharp turns, as people typically walk in a smooth curve,

$$D(x) = \sum_{i=1}^{T} ||x_i - Ax_{i-1}||^2_{\Lambda_S}, \tag{2.1}$$

and second term in the cost function was S and focused on keeping the trajectory accurate in terms of the observations,

$$S(x) = \sum_{i \in O} ||Cx_i - y_i||^2_{\Lambda_D}, \tag{2.2}$$

which combine to form the overall cost function $S(x) + D(x)$.

The original model detailed in those did not account for walls.[1] That work was done later (§3).

[1]These results were part of joint work with Ali Rahimi. The figures were published previously.

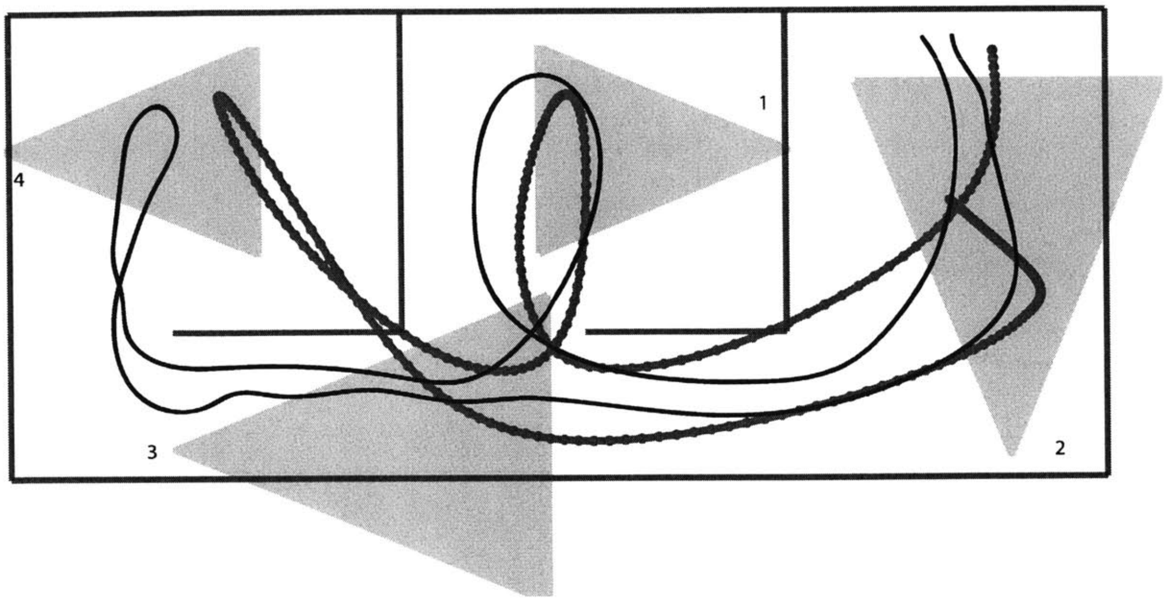


Figure 2-11: A result with data collected from the iPaq network with horizontal cameras. The thin line is the actual trajectory, and the thick line is the perceived one. The grey cones are the fields of view. The black bars are the walls. Notice that the generated trajectory goes directly through three walls.

2.5.1 Measuring Bearing

The first demonstration of the model with experimental data from the network was tracking people using the iPaqs as horizontal bearing sensors (Figure 2-11). The camera was placed horizontally (Figure 2-9), and a person walking by had their bearing and distance calculated. The sensor's exact locations were measured manually so that those parameters were known before calculation. The camera's snapshot was analyzed, as described in Section 2.3.1 to determine the bearing and distance. Using these observations, the trajectory model generated a likely trajectory. The results were accurate, except when walls were incorporated into the layout.

2.5.2 Measuring Location

The second demonstration was simultaneous tracking and calibration with the iPaq network as vertical location sensors (Figure 2-12) [12]. The iPaqs were mounted to the ceiling (Figure 2-10) and, using a snapshot from its camera, computed a person's

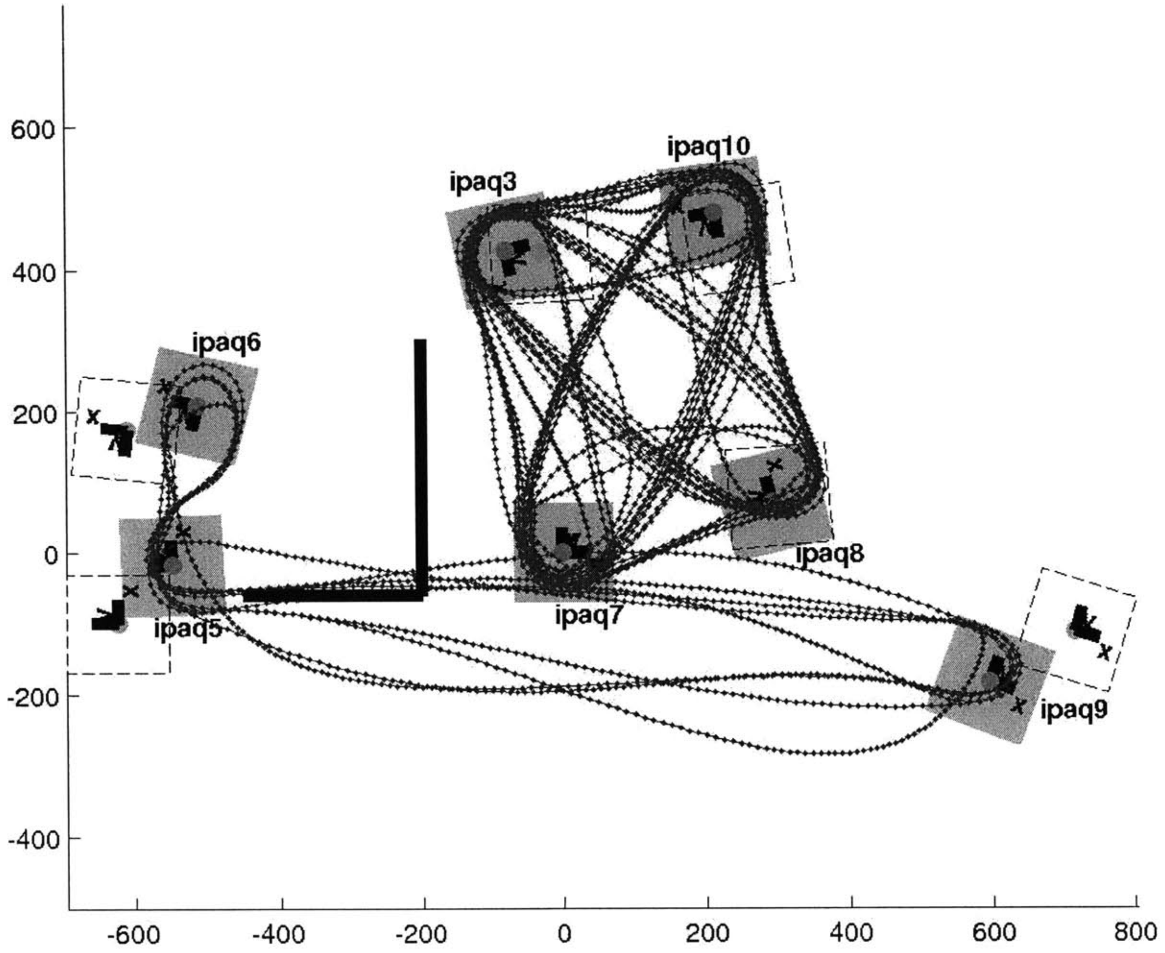


Figure 2-12: A result with data collected from the iPaq network with vertical cameras. The dotted line is the recovered trajectory. The grey boxes are the recovered locations and orientations of the sensors, and the dotted boxes are the actual locations and orientations of the sensors.

relative location within the field of view. Using only this relative location, without knowing any information about each iPaq's location in the network, the model infers both the trajectory of a person and the relative location of the iPaq with each other.

Chapter 3

Wall Constraints

The trajectory tracking algorithm devised for the research project took data points, from either Matlab or the iPaq network, and produced a likely trajectory closely mimicking the real one. Originally, the only other constraint was the smoothness of the trajectory, included because people generally walk smoothly in arcs. The most important constraint missing from this model was walls. The presence of a wall was entirely ignored (Figure 2-11), so while the real trajectory wrapped around a wall, the calculated trajectory continued through it. This constraint severely limited the types of network configurations that would yield accurate experimental results.

Allowing walls opens the door to a large number of two-dimensional layouts. The iPaqs could be arbitrarily placed, and a person could walk smoothly anywhere; the algorithm would automatically wrap around any obstacles it encountered. It was assumed that the locations of the walls were already know.

The focus of the research was to represent the walls mathematically. Although Matlab performed all the calculations, the goal was a mathematical expression for the walls, in some form.

3.1 Mathematical Programming Overview

Since the main algorithm was implemented as a mathematical program [8], a brief overview of it would be useful. There are two high-level aspects to a mathematical program: a set of linear constraints and an objective function. The linear constraints dictate what forms a solution can take. The objective function evaluates a solution, in effect ranking it against other potential solutions and allowing the overall program to find the best solution, either locally or globally.

Originally, the goal was to represent a set of walls as a set of linear constraints. However, because a wall represented a non-convex space, we could not translate it into a linear constraint (§3.2). Instead, the walls were incorporated into the objective function, by heavily penalizing any solution that overlapped with any walls (§3.3). There are no linear constraints in the final program.

3.2 Linear Constraints

For the project, walls cannot be reduced to a set of linear inequalities. The set of linear inequalities must form a convex space within the trajectory model, and walls do not satisfy that. Intuitively, the simplest way to represent a wall is as a very thin box using four inequalities. However, as T. Schouwenaars et al. [7] found, these need to include boolean logic to function correctly. *Either* the point is above the box *or* the point is below the box. As boolean logic is necessary, the space is non-convex and optimization becomes much more complex.

3.3 Objective Function

Because we could not represent walls as linear constraints, we had two options: represent walls as nonlinear constraints that we then linearize, or incorporate walls into the cost function. We chose to pursue the latter first because we had a clearer idea of how to go about it. To incorporate walls in the the cost function, we had to define a function G that could be added to the existing cost function. There were several constraints that our additional term had to satisfy, as seen in Figure 3-1.

These constraints were constructed to test the various domains of the function, to weed out false positives and false negatives. The penalty should be negligible for

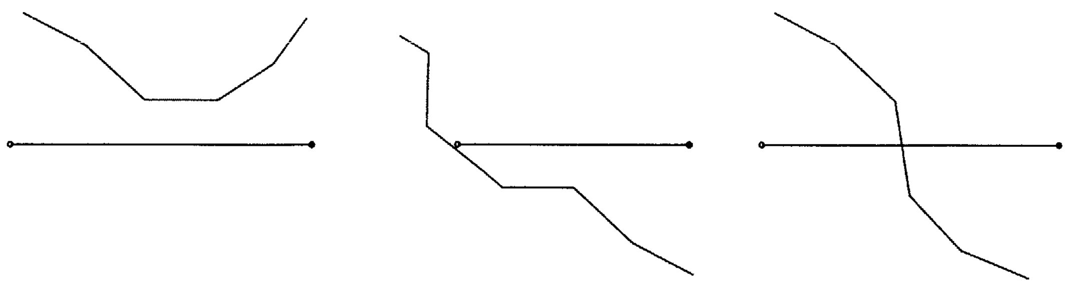

(a) **Valid path.** (b) **Valid path.** (c) **Invalid path.**

Figure 3-1: The three subdomains of the constraint problem.

points on the same side of a wall but large if the points cross a wall. Figure 3-1(a) represents a typical path that does not cross the wall. Figure 3-1(b) is a corner case that should not be penalized. Finally, Figure 3-1(c) is an invalid path that should be heavily penalized.

There were several mathematical tools that proved useful to achieve this precision in the penalty. In essence, the problem broke down into two parts: the polarity of the two points and the magnitude of the penalty. The polarity equation (3.2) and the magnitude equation (3.1) could be found using the dot products.

3.3.1 Dot Products

The dot product is defined as the magnitude of a vector A multiplied by the magnitude of a vector B multiplied by the cosine of the angle between them. We took advantage of this calculation because it allowed us to compute a binary indicator for whether two points in the trajectory were on the same side of a wall or were on opposite sides. This result came from the cosine calculation.

If two points were on the same side, they would produce a dot product of the same sign, so when multiplied together, the result would always be positive. Conversely, the result would always be negative if they were on opposite sides of the wall. This mathematical tool allowed us to coarsely determine whether a solution was good.

3.3.2 The penalty

The wall will be represented by its two endpoints, x_1 and x_2. The trajectory p will be the set of points p_t where

$$p_t = \begin{bmatrix} u_t & v_t \end{bmatrix}^\top, \vec{x} = x_1 - x_2.$$

Let x_m be the midpoint of the wall,

$$x_m = \begin{bmatrix} \frac{1}{2}\left(x_1^1 + x_2^1\right) & \frac{1}{2}\left(x_1^2 + x_2^2\right) \end{bmatrix}.$$

Next, K is the magnitude rating for the trajectory,

$$K = \frac{1}{2}\left[\frac{\|\vec{x}\|}{\min\left(\|p_t - x_m\|, \|p_{t+1} - x_m\|\right)}\right]. \tag{3.1}$$

and H is the polarity of the trajectory, or whether or not it crosses the wall segment,

$$H = \frac{-1}{1 + \exp\left(a\left(\frac{(p_t - x_1)\cdot\left(\frac{\vec{x}}{\|\vec{x}\|}\right)}{\|p_t - x_1\|}\right)\left(\frac{(p_{t+1} - x_1)\cdot\left(\frac{\vec{x}}{\|\vec{x}\|}\right)}{\|p_{t+1} - x_1\|}\right)\right)} + \frac{1}{2}. \tag{3.2}$$

Finally, G was concatenation of the previous expressions, using exponentials to compound small changes,

$$G = \exp\left(-\exp\left(K\right)\left(H\right)\right), \quad \textit{or}$$

$$G = \exp\left(-\exp\left(K\right)\left(\frac{-1}{1 + \exp\left(a\left(\frac{(p_t - x_1)\cdot\left(\frac{\vec{x}}{\|\vec{x}\|}\right)}{\|p_t - x_1\|}\right)\left(\frac{(p_{t+1} - x_1)\cdot\left(\frac{\vec{x}}{\|\vec{x}\|}\right)}{\|p_{t+1} - x_1\|}\right)\right)} + \frac{1}{2}\right)\right). \tag{3.3}$$

(3.3) correctly weighted the polarity and the magnitude to penalize only the paths that crossed the wall. Used in combination with the wall's midpoint, an appropriate penalty could be calculated, given well-chosen constants. Added to the existing cost function, it penalized only pairs of points that straddle the wall line segment. One nice property of this equation was the modularity of it. Given a better method for calculating the polarity or the magnitude, the new expression could be dropped into the full equation instantly.

3.4 Implementation

Since the main algorithm for trajectory tracking was implemented in Matlab, this addition to the cost function was implemented there as well. The resulting cost function was $S(x) + D(x) + G(x)$ using (2.2) and (2.1).

The implementation went through two versions. Originally, differentiability of G was a requirement. Gradients seemed necessary at the time, and although that requirement was relaxed for the revision, it should be required for efficiency reasons (§3.7).

3.4.1 First Version

The first version calculated the magnitude K differently. Instead of using min(), (3.1) was structured as follows,

$$K = \frac{1}{2}\left[\frac{||\vec{x}||}{\frac{1}{2}||p_t - x_m|| + \frac{1}{2}||p_{t+1} - x_m||}\right]. \tag{3.4}$$

This structure caused the optimization algorithm to favor pushing only one point away from the wall, leaving the other very close, because it still decreased the fraction.

3.4.2 Second Version

Instead of averaging, using min() in (3.1) greatly improved the precision of the overall function. Because there was no gradient, the algorithm used Matlab's Optimization Toolbox, specifically the Matlab function fminsearch.

The flow of the program was in two parts. First, the original algorithm from [13, 12] produced a smooth trajectory that matched the observations. Second, the revised algorithm used this trajectory as its initial state and then attempted to push the trajectory away from walls. (See the Matlab code in Appendix B.)

3.5 Results

A number of results were generated with the revised, wall-observant, trajectory-tracking algorithm from simulated data. In each set of figures, there is a trajectory from an unmodified version of the algorithm, where the goal is only smoothness and accuracy, and there are one or more results from the modified version, with walls accounted for. The walls are represented as line segments; observations are represented by circles; and, trajectories are represented by a series of line segments, with points to mark the distance traveled in a time step. Throughout, several constants in the algorithm were varied to achieve the displayed results.

The first set of results (Figure 3-2) reflected the simplest possible scenario for overlap, where a short trajectory crossed one wall once. The final snapshot (Figure 3-2(b)) lacks complete smoothness, but it achieves its objective. The second set of results (Figure 3-3) is slightly more complex, with four observations. Again, the trajectory avoids the wall with negligible impact on smoothness. The third set of results (Figure 3-4) adds a second and third wall to the scenario. The fourth set of results (Figure 3-5) adds a another wall. The final set of results (Figure 3-6), with four walls, shows the limitation in the revised algorithm's ability to scale. If there are too many points it needs to relocate simultaneously, it has trouble deciding which to move and how far.

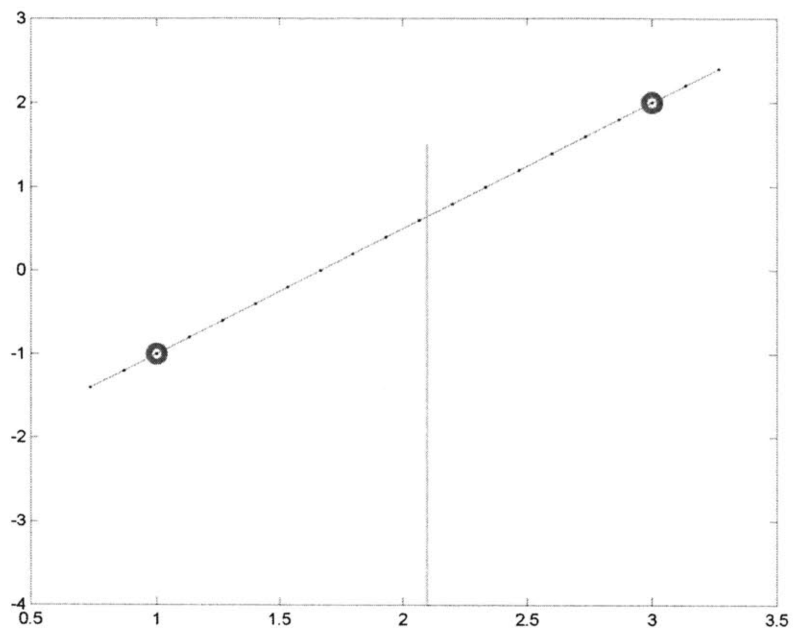
3.6 Discussion

The fundamental drawback is the speed of the revised algorithm. It takes on the order of days to compute a valid trajectory through a realistic maze of walls, such as that found in a typical office layout. The algorithm also suffers from a scaling problem. Increasing the number of walls included in the environment complicates the task for the minimization tool. These bottlenecks restrict the project's usefulness and limit it to research. Finally, there is a heavy reliance on constants to ensure a valid trajectory. There were no obvious choices for the individual numbers that would produce the best result for every scenario.

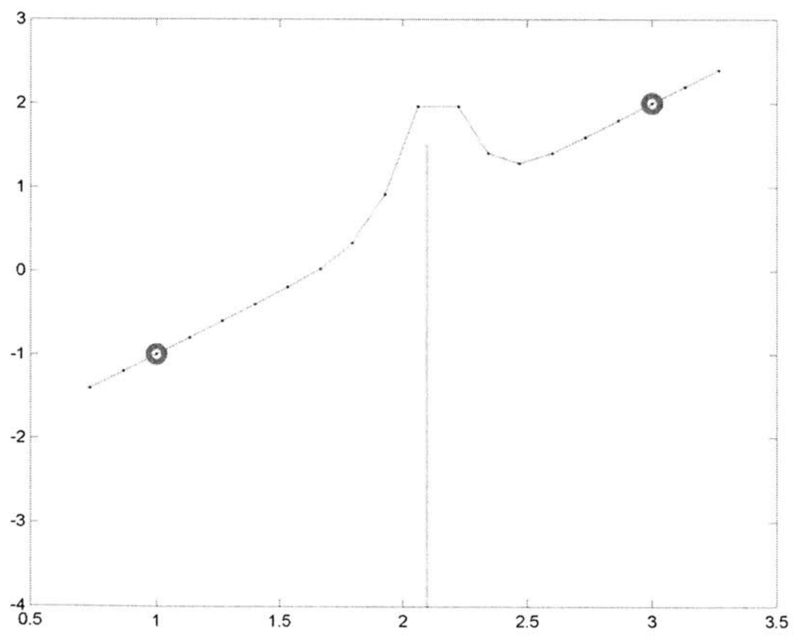
However, the efficiency problem is separate from the theoretical underpinnings described above. Decoupling the polarity and the magnitude enable the mathematics to be worked out in isolation.

3.7 Future Work

Again, the central failure of the wall-constrained algorithm was its speed. There are a couple avenues to pursue to fix this problem. Because (3.3) used an exponential to exaggerate any problem, the number exploded, increasing computation and complicating the work for the Matlab's minimization function. Instead, a polynomial equation or logarithm could be used to penalize incorrect trajectories but without the heavy computations from the multiple exponentials. Furthermore, using min() in (3.1) prevented any derivatives, so creating a differentiable expression would allow gradients and undoubtedly help the minimization algorithm.

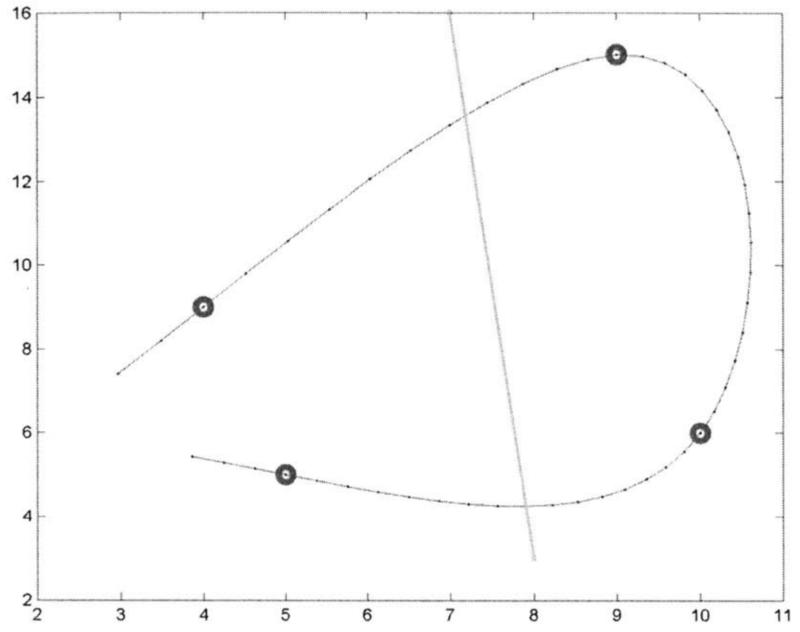


(a) The trajectory without wall constraints.

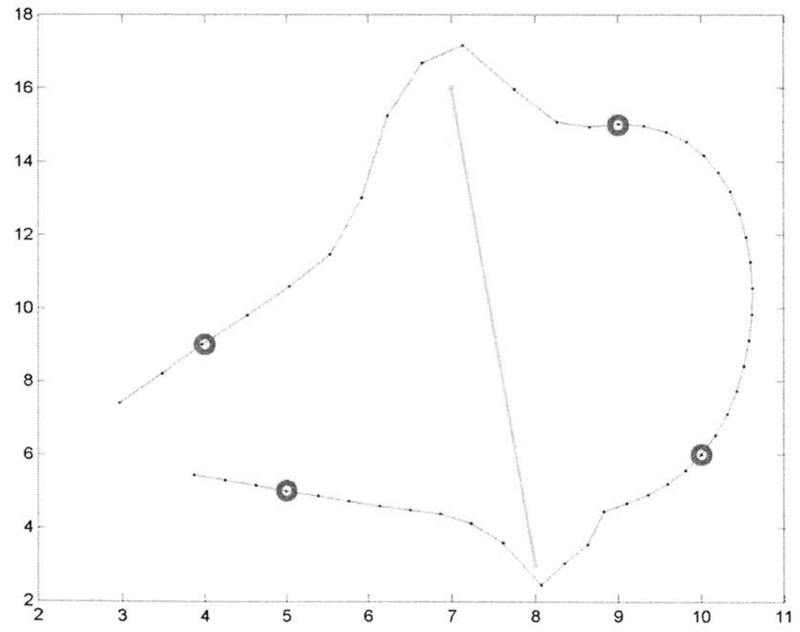


(b) The trajectory with wall constraints.

Figure 3-2: A scenario with one wall and two observations.

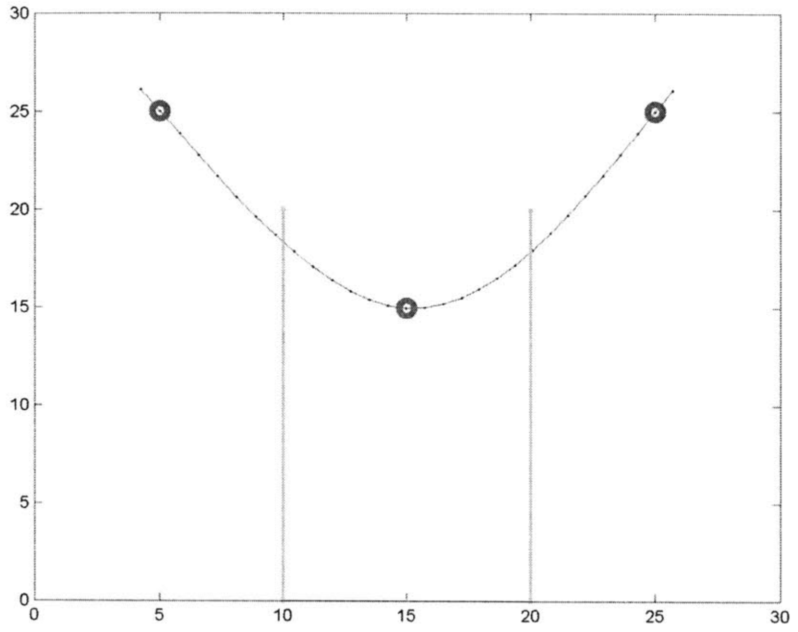


(a) The trajectory without wall constraints.

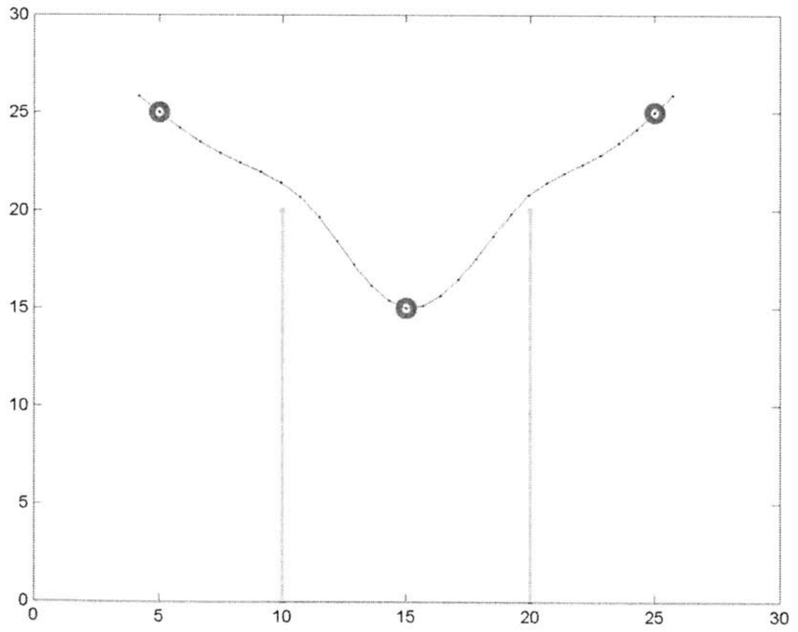


(b) The trajectory with wall constraints.

Figure 3-3: A scenario with one wall and four observations.

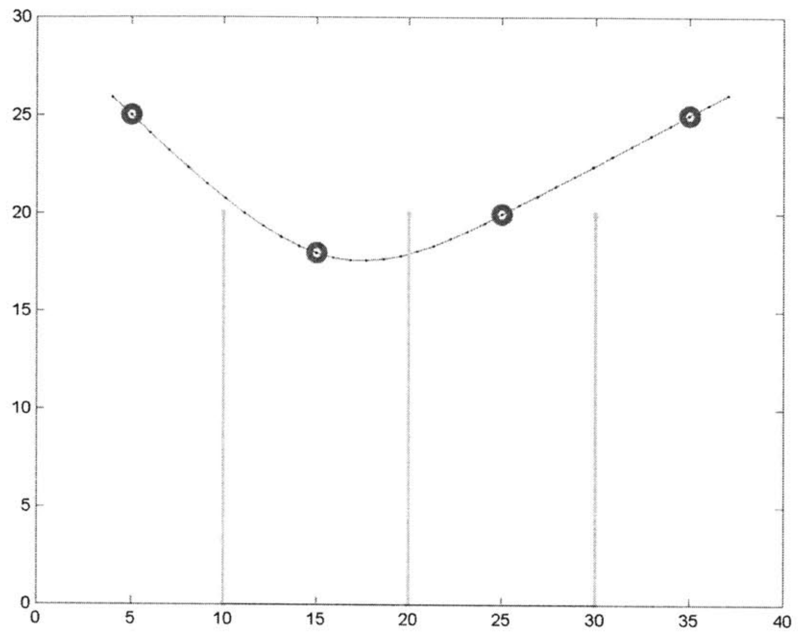


(a) The trajectory without wall constraints.

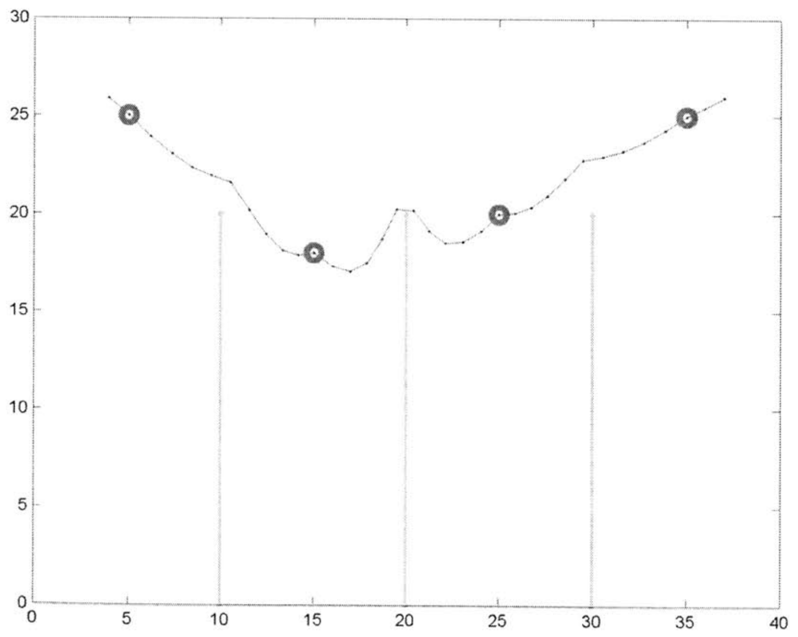


(b) The trajectory with wall constraints.

Figure 3-4: A scenario with two walls and three observations.

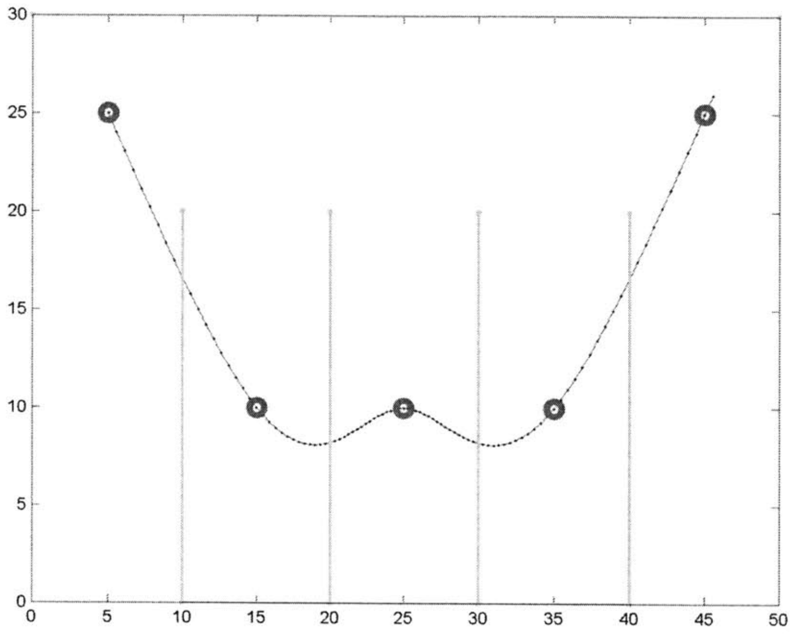


(a) The trajectory without wall constraints.

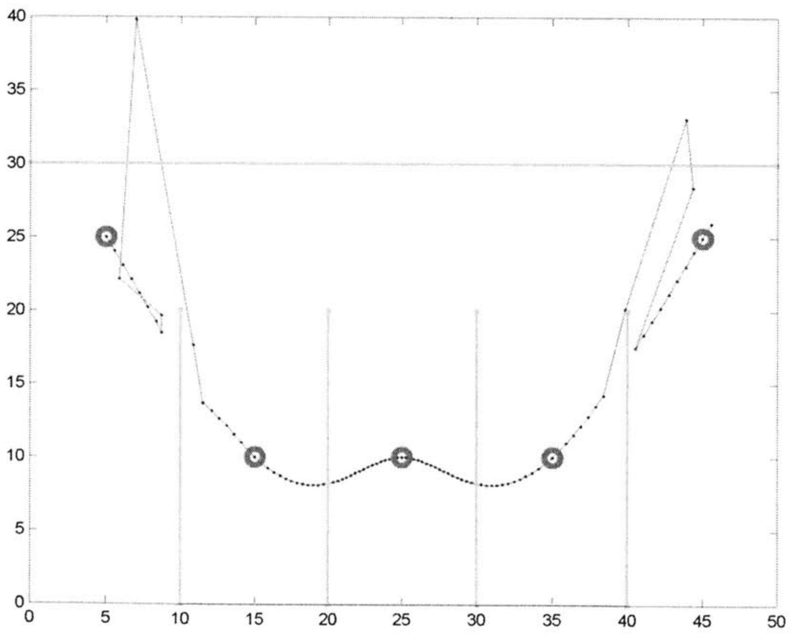


(b) The trajectory with wall constraints.

Figure 3-5: A scenario with three walls and four observations.



(a) The trajectory without wall constraints.



(b) The trajectory with wall constraints.

Figure 3-6: A scenario with five walls and five observations.

Chapter 4

Conclusion

Many organizations have a vested interest in tracking the movements of people through a physical space. However, they dont want a sensor network that requires overlapping fields of view. They prefer not to invade peoples space, and the cost of the minimum number of sensors to canvas a large office is prohibitive. These organizations would benefit from a network designed without this requirement in mind, one that could be expanded incrementally for increasingly accurate results.

Furthermore, they would find it much more helpful if the network would calibrate itself based solely on the trajectories it sees and accounted for walls when given the floor layout. It is a low-cost, scalable solution to the tracking problem.

This project demonstrated the practicality of setting up such a network (§2.3) and using it (§2.5). Furthermore, it showed the feasibility of avoiding walls when computing a person's trajectory (§3). This research was a first step towards moving research out of academic labs and applying it to real-world situations.

Appendix A

TinyOS Distance Module

```
includes BrianMsg;
module Snet_streamM {
  provides  {
    interface StdControl;
    interface ProcessCmd;
  }
  uses {
    interface Leds;
    interface ReceiveMsg as ReceiveMsg;
    interface StdControl as CommControl;
    interface SendMsg as SendMsg;
    interface Timer as Timer_sample;
    interface Timer as Timer_send;
    interface StdControl as SensorControl;
    interface StdControl as TimerControl;
    interface ADC;
    interface Clock;
  }
}

/*
 *  Module Implementation
 */
implementation
{

  enum {
    NUMSAMPLES = 3,
    INTERVAL   = 20,
    ADC_HIGH_THRESH = 380,
    ADC_LOW_THRESH  = 280
  };
  // module scoped variables
  TOS_Msg dmsg;
  TOS_MsgPtr dmsgptr;
  uint16_t timestamp[2];
  uint16_t data_timestamp[NUMSAMPLES*2];
  uint16_t total_samples;
  uint16_t data_sample[NUMSAMPLES];
  TOS_MsgPtr msg;
  int8_t pending;
  TOS_Msg buf;

  command result_t ProcessCmd.startSampling() {
    timestamp[0] = 0;
    timestamp[1] = 0;
  total_samples = 0;
  call Timer_sample.start(TIMER_REPEAT, INTERVAL);
  call Clock.setRate(32, 1); //100 ticks/second
  return SUCCESS;
}

//sends sampled ADC data with timestamps over radio
task void sendDataPacket() {
    struct BrianMsg * time = (struct BrianMsg *) dmsg.data;

    //fill packet with ADC data and time stamps
    time->sourceMoteID = 0x01;
    time->channel = 3;
    time->data[0]  = total_samples;
    time->data[1]  = data_timestamp[0];
    time->data[2]  = data_timestamp[4];
    time->data[3]  = data_sample[0];
    time->data[4]  = data_timestamp[1];
    time->data[5]  = data_timestamp[5];
    time->data[6]  = data_sample[1];
    time->data[7]  = data_timestamp[2];
    time->data[8]  = data_timestamp[6];
    time->data[9]  = data_sample[2];

    //attempt to send packet over radio
    if (call SendMsg.send(TOS_BCAST_ADDR, sizeof(struct BrianMsg),
  &dmsg))
      call Leds.yellowToggle();
    else {
      call Leds.redToggle();
    }
}

//collects data from ADC and stores in a buffer with timestamps
event result_t ADC.dataReady(uint16_t adc_data) {
  dbg(DBG_CLOCK, "adc got data\n");
  //only save valid data points
  if (adc_data >= 280 & adc_data <= 380) {
    //store sampled data and timestamps
    data_sample[total_samples%NUMSAMPLES] = adc_data;
    data_timestamp[total_samples%NUMSAMPLES] = timestamp[0];
    data_timestamp[total_samples%NUMSAMPLES+NUMSAMPLES] = timestamp[1];
    total_samples++;

    //send packet (NUMSAMPLES readings)
    if (total_samples%NUMSAMPLES == 0)
      post sendDataPacket();
  }

  //turns on green LED for data higher than 512
  if (adc_data >= ADC_LOW_THRESH & adc_data <= ADC_HIGH_THRESH)
    call Leds.greenOn();
  else
    call Leds.greenOff();
  return SUCCESS;
}

//updates the timestamp bytes and requests data from ADC
//(called for every timer fire)
command result_t ProcessCmd.updateTimeStamp() {
  //update two-byte timestamp
  timestamp[0]++;
  if (timestamp[0] == 0) {
    timestamp[1]++;
  }

  //request ADC data
  if (timestamp[0]%20)
```

```
    call ADC.getData();

  return SUCCESS;
}

//sampling timer
event result_t Timer_sample.fired() {
  dbg(DBG_CLOCK,"timer_sample fired\n");
  call ProcessCmd.updateTimeStamp();
  return SUCCESS;
}

//radio timer (unused)
event result_t Timer_send.fired() {
  return SUCCESS;
}

event result_t Clock.fire() {
  return SUCCESS;
}

command result_t StdControl.init() {
  total_samples = 0;
  timestamp[0] = 0;
  timestamp[1] = 0;
  call SensorControl.init();
  call CommControl.init();
  return SUCCESS;
}

//turns sampling timer on at 5ms sampling rate
command result_t StdControl.start(){
  call SensorControl.start();
  call CommControl.start();
  return SUCCESS;
}

command result_t StdControl.stop(){
  call Timer_sample.stop();
  call Timer_send.stop();
  call SensorControl.stop();
  call CommControl.stop();
  return SUCCESS;
}

task void cmdInterpret() {
  call Leds.redToggle();
}

/**
 * Posts the cmdInterpret() task to handle the recieved command.
 * @return Always returns <code>SUCCESS</code>
 **/
command result_t ProcessCmd.execute(TOS_MsgPtr pmsg) {
  return SUCCESS;
}

/**
 * Called upon message reception and invokes the ProcessCmd.execute()
 * command.
 * @return Returns a pointer to a TOS_Msg buffer
 **/
event TOS_MsgPtr ReceiveMsg.receive(TOS_MsgPtr pmsg){
    call ProcessCmd.startSampling();
    return pmsg;
}

/**
```

```
   * Signalled when the previous packet has been sent.
   * @return Always returns SUCCESS.
   */
  event result_t SendMsg.sendDone(TOS_MsgPtr sent, result_t success) {
    return SUCCESS;
  }
} // end of implementation
```

Appendix B

Matlab Code

```
function [G] = G(p1,p2,x1,x2)
% p1 and p2 are 4D points on a trajectory [x,y,x',y']
% x1 and x2 are 2D points for the beginning and end of a wall [x,y]
% This function will return G

% modify p's so they are just 2D for math with x's
p1_2D = [p1(1) p1(2)];
p2_2D = [p2(1) p2(2)];

a = 100;
x = x1 - x2;
x_mag = norm(x);
n = 1/x_mag*[x(2) -x(1)];
R = norm(p1_2D - x1);
S = norm(p2_2D - x1);
I = dot(p1_2D - x1,n)/R;
J = dot(p2_2D - x1,n)/S;
x_mid = [.5*(x1(1) + x2(1)) .5*(x1(2) + x2(2))];
ave = min(norm(p1_2D - x_mid),norm(p2_2D - x_mid));
K = 1.7*(.5*x_mag/ave);
H = -1/(1 + exp(a*I*J)) + .5;
G = exp(-exp(K)*H);
```

Bibliography

[1] Q. Cai and J.K. Aggarwal. Tracking human motion in structured environments using a distributed-camera system. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1999.

[2] Crossbow Technology. http://www.xbow.com/.

[3] David E. Culler and Hans Mulder. Smart sensors to network the world. *Scientific American*, 2004.

[4] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer S. J. Pister. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems*, 2000.

[5] J. M. Kahn, R. H. Katz, and K. S. J. Pister. Emerging challenges: Mobile networking for smart dust. In *Journal of Communications and Networks*, 2000.

[6] S. Khan and M. Shah. Consistent labeling of tracked objects in multiple cameras with overlapping fields of view. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2003.

[7] I. Martinos, T. Schouwenaars, J. De Mot, and E. Feron. Hierarchical cooperative multi-agent navigation using mathematical programming. In *Proc. Allerton Conference on Communication, Control and Computing*, 2003.

[8] Mathematical Programming. http://www.mathprog.org/.

[9] A. Mittal and L. S. Davis. M2tracker: A multi-view approach to segmenting and tracking people in a cluttered scene using region-based stereo. In *Proc. European Conference on Computer Vision*, 2002.

[10] H. Pasula, S. J. Russell, M. Ostland, , and Y. Ritov. Tracking many objects with many sensors. In *Proc. IJCAI*, 1999.

[11] Nissanka B. Priyantha, Anit Chakraborty, and Hari Balakrishnan. The cricket location-support system. In *Proc. 6th ACM MOBICOM*, 2000.

[12] A. Rahimi, B. Dunagan, and T. Darrell. Simultaneous calibration and tracking with a network of non-overlapping sensors. In *Proc. Conference on Computer Vision and Pattern Recognition*, 2004.

[13] A. Rahimi, B. Dunagan, and T. Darrell. Tracking people with a sparse network of bearing sensors. In *Proc. European Conference on Computer Vision*, 2004.

[14] University of California at Berkeley Motes. http://webs.cs.berkeley.edu/.

[15] Kamin Whitehouse. The design of calamari: an ad-hoc localization system for sensor networks. Master's thesis, University of California at Berkeley, 2002.