USE OF WIRELESS SIGNAL CHARACTERISTICS FOR MOBILE ROBOTICS LOCALIZATION AND ACCESS POINT MAPPING

by

JOSHUA DAVIES

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT ARLINGTON

December 2008

ACKNOWLEDGEMENTS

I owe a deep debt of gratitude to my thesis advisor, Dr. Kamangar, for constantly steering me in the right direction and reminding me to focus on what was important. I would also like to thank Dr. Zaruba for his constant patience and assistance – all the more appreciated considering that I was not his student at the time, and Dr. Huber for agreeing to sit on my committee even though I spent the semester having impromptu hallway meetings outside his class. Special thanks go to Dr. Roger Lamprey for cheering me on the whole time.

Of course, I would like to thank my family for putting up with my absence while I worked on this thesis and my employers at Travelocity.com for their understanding around my whole schedule throughout the process.

Finally, I'd like to thank my sister-in-law, Liliana Mejia, for asking me for help with her homework so many years ago – if not for that request, I may never have found an interest in returning to graduate school.

November 24, 2008

ABSTRACT

USE OF WIRELESS SIGNAL CHARACTERISTICS

FOR MOBILE ROBOTICS LOCALIZATION

AND ACCESS POINT MAPPING

Joshua Davies, M.S.

The University of Texas at Arlington, 2008

Supervising Professor: Farhad Kamangar

Mobile robotics is an interesting and challenging emerging field in computer science. To properly achieve true autonomy, a robot must be able to determine where it is in relationship to a global frame of reference and it must be able to do so with a minimum of interaction from human operators and impose as few constraints on its surroundings as possible. The uncertain nature of mobility and perception requires that advanced probabilistic inference techniques be applied to minimize error.

This work examines the applicability of radio-frequency signals to the mobile robot localization problem to localize quickly over a wide area. Particle filtering techniques are employed to adjust for anticipated errors in both the motion model and the perception model. Radio hardware designed to capture time of flight and received signal-strength indicators (RSSI) is used to infer relative distances and triangulate the most likely position of a mobile node, taking into account a priori knowledge about past poses.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Mobile Robot Localization

A robot is an electromechanical automaton that performs repetitive functions that are too dangerous, too tedious, or too error-prone for a human to perform. The term “robot” describes a wide variety of actual devices. For example, a stationary mechanical arm that assists with a manufacturing assembly line can be considered a robot. For the purposes of this work, however, the term robot will be used exclusively to refer to mobile robots – that is, those that are capable of moving from one place to another under their own power.

Many factors need to be taken into account to accomplish this, including optimal path planning, safety and power management. However, the nonintuitive problem of localization has been described as “the most fundamental problem in robotics”. This problem arises due to the fact that vehicular motion of any kind is imperfect, no matter how great the precision. When applying a motion control, there is a non-zero “drift” associated with every motion. In order to account for this drift and to compensate for it, the robot must periodically determine where it actually is in relation to where it thinks it is. This process is referred to in mobile robotics literature as “localization”.

The difficulty in solving this problem is compounded by the fact that any form of sensing its location is, in turn, subject to error in the same way that motion is subject to error. No sensor is perfect. Various methods have been applied to the localization problem including sonar, laser range finders (which bounce a highly focused beam of light off of a nearby object such as a wall and measure the time it takes to return) and stereoscopic camera. In recent years, however, a

growing interest has been shown in the use of radio-frequency signals to accomplish localization.

The benefits of using radio-frequency signals for localization include the ability to localize regardless of the positioning of walls or furniture, as well as the ability to use relatively inexpensive hardware to solve a problem that heretofore required highly specialized, difficult-to-obtain components.

1.1.1. Simultaneous Localization and Mapping

Although it is important, and challenging, to properly localize a robot moving on its own volition, it's equally important for that same robot to be localized within some map. It does no good for a robot to know, for example, that it is three meters to the north and east of a global origin, if it does not have any information on where walls, doors and other obstacles are located. In the simplest case, this knowledge is provided out of band to the robot before self-guided motion is attempted.

A more complex, but more useful, scenario is one where the robot does not know anything about its surroundings but instead uses probabilistic inference to try to discover the characteristics of its environment as it moves. This process is referred to as simultaneous localization and mapping or "SLAM" and moves robotics one step further toward complete independence.

One of the benefits of "immediate" localization techniques such as sonar or laser range finders is that they, by their very nature, build a map as they localize. RF signals, on the other hand, do not. However, the requirement that the locations of the base stations be known beforehand in order to properly localize is a significant drawback of the use of radio frequency for the purposes of localization. This work explores the possibility of using SLAM techniques to discover the locations of the unknown base stations while moving.

1.2 Probabilistic Inference

A common general robotics problem, which localization is just one instance of, is the problem of state estimation. Given a mathematical, finite state model of a system (that is, the environment in which the robot or automaton finds itself, also called the process model) and a means of measuring the states of the system (the measurement model), what is the actual state at any given time? In the absence of uncertainty, this is a trivially simple problem to solve – in fact, measurements don't even need to be taken, since the model itself would be guaranteed to be 100% correct all the time. However, in real-world scenarios, uncertainty is always present and must be accounted for.

The robot in this scenario now has two conflicting sets of data about its position – first, the prediction of where it should be based on where it was and the motion model and second, the data returned from its sensors. To further complicate the problem, measurements report distances to features, rather than absolute positions. Therefore, a specific measurement can be equally likely in many different actual locations.

One naïve approach might be to average the two data sets and assume that "reality" is the average. Fortunately, there is provides a much better general framework for integrating these two sets of conflicting data, assuming that the "inverse probabilities" are available.

1.2.1. Baye's Rule

One of the most fundamental principles of probability is "Bayes Rule" which states a precise construct for taking "prior knowledge" (usually referred to as "a priori") into account when formulating a probabilistic total. To speak of probability in mathematical terms, it is necessary to assign a numeric value to events. By convention, this numeric value ranges from 0 to 1 with 1 identifying a certain event and 0 identifying an impossible event and higher values indicating more likely events.

When combining events with known probabilities, the probabilities of each event are multiplied. For example, if a fair coin is flipped once, the probability of observing the event

"heads" is ½, and the probability of observing the event "tails" is ½. If a fair coin is flipped twice, the probability of observing the event "heads" followed by another event "heads" is ½ * ½ = ¼. As more events are introduced into such a "chain", the probability of observing a specific sequence of events gets smaller (which makes intuitive sense). This is part of the reason that probabilities are always represented as fractions, since multiplication by a fraction always results in a smaller quantity.

Most interesting applications of probability do not deal with random events such as coin flips, but instead events whose probabilities are predicated on known conditions – these are referred to as "conditional probabilities" and are denoted as P(X|Y). This is easy to translate into a robotics theme – X is the state, or pose, and Y is, for example, the most recent movement. In other words, given that the robot was issued a control command to move 1 meter to the east, what is the probability that the robot's new position is precisely 1 meter east of its previous position? What is the probability that it was off by 1 degree and is instead 0.99 meters east but 0.01 meters north?

Conditional probabilities are defined more precisely as $P(X|Y) = \frac{P(X \& Y)}{P(Y)}$ – however, this definition sidesteps how to compute P(X & Y) (the probability that both X and Y are simultaneously true). In general, this probability can't be precisely, mathematically defined but must instead be determined through experimentation.

In many applications of probability theory, P(X|Y) is not known or cannot be directly measured, but P(Y|X) is. Since $P(X|Y) = \frac{P(X \& Y)}{P(Y)}$ and $P(Y|X) = \frac{P(Y \& X)}{P(X)}$, P(X|Y) != P(Y|X) unless P(Y) = P(X), so "inverting" conditional proabilities this way is not as straightforward as it may appear. In fact, expanding the definitions makes the correct formulation obvious:

$$P(Y|X) = \frac{P(Y \& X)}{P(X)}$$

$$P(X|Y) = \frac{P(X \& Y)}{P(Y)} = \frac{\frac{P(X \& Y)}{P(X)}P(X)}{P(Y)} = \frac{P(Y|X)P(X)}{P(Y)}$$

Which is, in fact, “Bayes Rule”. Bayes rule states a mathematically sound means of converting from one form to the other, assuming a few additional probabilities are also known. Specifically, P(X) (irrespective of Y) must be known, and the P(Y) (irrespective of X) must also be known. If they are, Bayes rule states that $P(X|Y) = \frac{P(Y|X)P(X)}{P(Y)}$. Although this doesn't appear to be progress (we now have to obtain P(X) and P(Y) in addition to P(Y|X) just to solve for the one required quantity P(X|Y)), in many applications, this information is available or can be easily computed.

1.2.2. Random Variables

Ordinarily, when quantities of interest are modeled probabilistically, they are known to follow a distribution of values – some values are more likely than others, and the likelihood of each value is known. This likelihood model is referred to as the “probability density function” and quantities that take on values with frequencies associated with a known probability density function are called “random” variables. The study of random variables is fundamental to the formal study of probability.

Often, a variable is known to be random, but its associated probability density function is not known. In this case, statistical inference must be performed to estimate the probability density function by means of “sampling”. The simplest form of statistical sampling is to measure the variable of interest several times, keep track of how often various values are observed in the form of a histogram, and assume that the random variable being estimated takes on probabilities according to the density given by the histogram.

1.2.3. Confidence Intervals

One question that arises in statistical inference is that of the number of samples that need to be taken to ensure confidence that the resulting histogram accurately represents the density to be modeled. Put another way, how confident can an experimenter be, given n samples, that the recorded histogram correctly models the underlying probability density function within a specific error tolerance?

According to the central limit theorem, as the number samples n → ∞, the sum of the expected values of a set of random samples is normally distributed. When estimating such a normally distributed underlying random variable, the question arises as to how many samples must be drawn in order to establish confidence that the results estimate the underlying distribution within a specific tolerance. This confidence level, and the tolerance, can be made arbitrarily high, but the number of samples n that must be drawn grows without bounds relative to these levels.

It can be shown (Baron 2007) that, given a confidence α and a tolerance ϵ, the maximum number n of samples that must be drawn to estimate the underlying distribution is given by: $n=z(\alpha/2)/\epsilon$ Where z(α/2) is the inverse standard normal value. However, in actual practice, this tends to overestimate n by a large amount. A more practical approach (Jain 1991) is to run a small but non-insignificant number of tests (15, for example), compute the average and the standard deviation of the test, and use that to compute the standard error $\epsilon = \frac{\sigma}{\sqrt{n}}$ and then adjust the *n* estimate if necessary, repeating until the standard deviation follows a normal distribution.

1.2.4. Normal Distributions

The discussion above of confidence intervals made a reference to "normal" distributions. This refers to a probability density function whose probabilities are given by:

$$\frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

This distribution is completely defined by its two parameters μ and σ (this is one reason that it is referred to as a "parametric" distribution). When μ and σ, which represent the mean and standard deviation of the distribution, are known, the probability of any x value can be computed by simply evaluating the standard normal equation. This is also known as a "Gaussian" distribution (after its discoverer) and plays an important role in probabilistic robotics.

One of the main benefits of assuming that a random variable follows a normal distribution is that the equation above is very precise and well studied. However, it is important to establish that the random variable being modeled does, indeed, follow the normal distribution. In particular, many random variables of interest are referred to as "multi-modal" – that is, they have more than one "most likely" value which are not centered around. As shown in Figure 1, below, the normal distribution has a limiting property that, regardless of the choice of μ and σ, there is one single peak (most likely) value, centered around μ (and spread out around this value according to σ – which is exactly the sort of behavior one would tend to expect for the parameters mean and standard deviation).

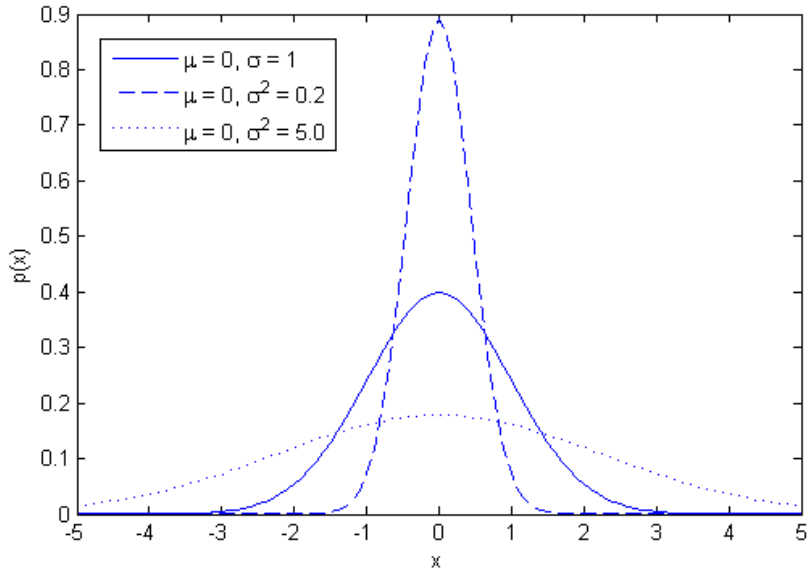

Figure 1 - Normal Distribution

In particular, measurement error is usually assumed to be Gaussian (that is, described by the parameters μ and σ as described above) due to the central limit theorem's assertion that the sum of many random variables drawn from any distribution tend to follow a normal distribution. It remains in these cases, however, to determine (usually through experimentation) the actual values μ and σ.

1.3 Radio Frequency Localization

Most localization techniques rely on measuring the time taken to perform a specific action. Sonar, for example, emits a sound and then listens for the echo of that same sound. The time taken for the echo to be returned is directly proportional to the distance from an object capable of reflecting sound. Laser range finders work in the same way.

Since radio frequency signals are not significantly reflective (at least not with regards to objects that a mobile robot would find itself interested in), radio frequency localization relies on two components – the active radio and one or more passive ones (the "bases"). The active radio sends a signal to a base which is expected to respond – the time taken to get a response can be used to calculate the distance from the mobile station to the base. The use of multiple bases can be used to infer a location relative to a fixed origin, assuming that the locations of the base stations themselves are also known.

1.3.1 Received Signal Strength

Since wireless signals are electromagnetic phenomena, the sender naturally induces standard electromagnetic properties upon the receiver. One of the most important of these is "power", which is usually measured in dB for radio frequency transmission. In the context of wireless localization, this received power is dubbed "Received Signal Strength" (RSS) or "Received Signal Strength Indication" (RSSI). One important characteristic of the received signal strength is that it is a function of the distance between the sender and the receiver. In fact, if the effects of noise and multipath fading could be ignored, the received signal strength (power) would be inversely proportional to the distance between the two:

$$RSSI = \frac{P}{d^2}$$

where P is the transmitting power of the sender. Although real-world effects cause this to vary (sometimes significantly) when applied, it still provides a useful baseline for the evaluation of distances.

1.3.2. Time of Flight

Another well-known and well-studied characteristic of wireless signal transmissions is their speed. The propagation of wireless transmissions through a vacuum is the same as the speed of light and is 299,792,458 meters per second. The speed of wireless transmission through free space ("air") is slightly less, but negligible for computational purposes. If the time between the transmission of a signal and the reception of that same signal (the "time of flight" or "time of arrival") can be measured, it can be divided by this known constant and a distance can be computed.

Unfortunately, this requires absolute synchronization between the sender and the receiver. Both nodes have to have such precise notions of the passage of time as to make this impractical for real-world scenarios. Methods of taking advantage of time-of-flight without requiring absolute synchronization are explored in Chapter 3.

1.3.3. Wireless Perception Model

The use of received signal strength (RSS) and time-of-flight (TOF) as a perception model has a few advantages over the more traditional SLAM model where sonar or laser range finders are used. Since sonar or laser range finders can only sense a limited range, a "map-fitting" must be performed to determine (probabilistically) where in the given map the robot may be. Since, from the perspective of a range finder, many locations look identical (consider the case of a long hallway), many range finding localizations may be needed to localize with any level of confidence – each from a different pose. This, in turn, adds risk, as moving from an unknown pose is in itself inherently dangerous.

Wireless signal strength and time-of-flight indicators, on the other hand, report an exact location (with some noise, whose effects and compensation will be examined below) as long as two or more signaling stations are used. (One signaling station can be used for localization, but the problem is much harder in this case, as the localizer only knows, with some varying degree of confidence, where it is located on a radius around the signaling station.)

One of the down sides of using wireless signal strength data for localization is that wireless signals do not provide any usable information on obstacles in the immediate vicinity, whereas by their nature, laser range finders and sonar do. This means that, if wireless signal strength is used for motion planning, either an accurate map (of a static environment) must be available, or a range finder must be used in tandem with the wireless signal strength readings to dynamically build a map.

Also, although wireless data does a good job of predicting the location respective to a given reference frame, it is much more difficult to extract orientation from wireless data, which is important since errors in orientation are multiplicative during motion.

802.11 is a common choice for this purpose. One of the drawbacks of 802.11, however, is the high power consumption inherent in the protocol, with implementations averaging about 7W during use. (Atheros Communications 2003) This can pose a significant problem for devices that need to operate for long periods of time, solely on battery power.

CHAPTER 2

BACKGROUND AND PREVIOUS WORK

2.1 Probabilistic State Estimation

Typically, elements of interest such as the position of the robot or the position of obstacles are not known with absolute precision. Instead, there is a "belief" associated with each such "state element". Initially, that belief may be practically unusable – it may span the entire range of possibilities uniformly. However, as more data is gathered, it can be integrated using Bayesian inference techniques as described in Chapter 1 and the belief can be refined according to the accepted laws of probabilities. Over time, as more and more data is discovered and integrated, the belief will become narrower and narrower until it is centered around a specific area of high probability.

Two of the most important issues when formulating such a probabilistic state estimator is how to represent the belief and how to update it. Techniques for both are explored below.

2.1.1. Markov Chain Models

One of the most straightforward probabilistic estimators is the "Markov Chain Model". An important characteristic of the Markov Chain Model is that the state at time k+1 be dependent only on the state at time k. The system being modeled can exhibit as many or as few states as required, as long as the probability of a transition from one state to the next. These transitions can then be represented as a matrix M of state transition probabilities. Some desirable properties fall out of systems that exhibit this behavior. One such property is that the most likely estimate at time p is given by M^p. Also useful is that any such Markov Chain Model which is "regular" (defined as any state transition being possible at any time) will eventually reach a "steady state" as $p \rightarrow \infty$, which can be used to make predictions in the distant future.

Two characteristics need to be in place for a Markov Chain Model system to be applicable – first, the state transitions must be discrete. It must be clear whether a state transition has or has not taken place. Second, the probability of each state transition must be known, and those probabilities must remain constant.

2.1.2. Hidden Markov Models

Often a system is known to follow a probabilistic transition matrix where the transitions themselves are known, but the probabilities of each transition are not. In this case, it is useful to view the problem as a "hidden" markov chain model, where the transition probabilities are hidden from view. Learning techniques, such as the Viterbi algorithm (Rabiner 1989) and the Baum-Welch algorithm (L. Welch 2003), have been developed to give a set of "training" data (with known outcomes) and allow it to infer the probabilities of state transitions.

2.1.3. Recursive Bayesian Estimation

Unfortunately, for robotics applications, the probabilities of state transitions do change over time, and they do so in a continuous manner, making markov chain models and hidden markov models impractical.

A "Bayes Filter" (or recursive bayesian estimator) represents a general framework for recovering an unknown probability density function from observations, assuming that the probability of observations given measurements is known (this can and usually must be determined experimentally). The Bayes Filter is intractable in the general case, but assumptions and simplifications can be made in order to make the computations feasible. In many cases, these assumptions and simplifications track closely enough with real-world behavior that they can be safely used to model real-world phenomena.

The Bayes filter operates recursively and is different than the Markov Chain Model in that all past estimates are accounted for in the estimation of the state at time k+1, however, these past estimates must be "rolled up" into the estimate at time k.

One important aspect of the general filter – in most cases, multiple measurements can and will be gathered within the span of a single control update (e.g. movement). The Bayes

filter allows these measurements to be integrated all at once – in fact, most implementations rely on this characteristic.

Since the Bayes Filter is a framework, it leaves certain aspects unspecified. One of the most important is how the state itself is actually represented. Since the model is designed around probabilities, the state is never represented as an absolute, but rather as some probabilistic representation of possible states. In its purest form, the Bayes filter will iterate over every possible "posterior" state - which, clearly, is impossible, since the range of possible posterior states is infinite (although some are exponentially more likely than others).

Algorithm Bayes_filter($bel(x_{t-1}),u_t,z_t$)
for all x_t do
$bel'(x_t) = \int p(x_t \mid u_t, x_{t-1})\, bel(x_{t-1})\, dx_{t-1}$
$bel(xt) = \frac{p(zt \mid xt)\, bel'(xt)}{p(z_t)}$
end
return $bel(x_t)$

Figure 2 - Bayes Filter

Shown above is the general outline that all Bayes' filters follow (Thrun, Burgard and Fox 2005). This is difficult to implement in directly due to the difficulty in representing the associated probability densities as integrable functions. As a result, implementations seek to employ shortcuts to simplify representation. Two such implementations are outlined below.

2.1.4. Kalman Filtering

The Kalman filter represents the state as a simple "Gaussian" – that is, an average (or expected) value and a standard deviation or "variance". In the case of multivariate state (which is almost always the case), the average is a vector of the same dimension N as the overall state and the variance is a covariance vector whose dimension is NxN. If the degree of uncertainty is known and can be expressed mathematically, it can be accounted for and the measurements can be combined with the models prediction in an optimal way. The Kalman filter is an optimal (that is, provably best) solution to this problem when the uncertainty can be expressed as a Gaussian, or normal, form. Specifically:

$$\det(2\pi\Sigma)^{\frac{1}{2}} e^{-\frac{1}{2}(x-\mu)'\Sigma^{-1}(x-\mu)}$$ (Thrun, Burgard and Fox 2005)

This is a multi-dimensional extension of the normal distribution outlined in section 1.2.4 that allows for dependencies to exist between different state elements.

The Kalman filter exploits the predictable nature of the uncertainty or "noise" in the signal to optimize the estimation based on the prediction of the model as well as the update from the measurement. In this case, the Bayes Filter problem is simplified by assuming that the probability density function to be recovered is known – it is the Gaussian function above. What remains unknown are the exact values of the expectation and covariance, which the Kalman filter seeks to recover from measured observations.

Unfortunately, this relies on two very restrictive conditions. The first is that the noise in the process model and the measurement model are Gaussian, with known covariances. The second is the both the process model and the measurement model are linear – that is, can be expressed as:

$x[t] = Ax[t-1] + Bu[t] + \delta$

$z[t] = C[t]x[t] + \varepsilon$

where:

- x is an n x 1 column matrix with one row for each state variable
- A is an n x n state transition matrix relating state x[t-1] to x[t]
- u is an m x 1 column matrix representing the controls (if present) in the system
- B is an n x m transition matrix relating the control u to current state
- z is an l x 1 column matrix representing the measurement
- C is an l x n transition matrix relating the state to it's measurements
- δ and ϵ are zero-mean gaussian random variables whose covariance matrices Q and R are known.

The salient feature of the Kalman filter is the "Kalman Gain" which is given by (Welch and Bishop 2001):

$K = \hat{p}[t] C'(C \hat{p}[t] C' + R)^{-1}$

With:

$\hat{p}[t] = A p[t - 1] A' + Q$

And Q & R the covariance matrices of δ and ϵ, above. Once this gain has been computed, it can be used to "normalize" the influence of the sensor data on the predicted motion model – in other words, the larger the gain, the more faith should be put into the measured sensor data.

Many interesting and important estimation problems do not satisfy the tight constraints required to employ the Kalman filter. The Extended Kalman Filter (EKF) seeks to generalize the restricted Kalman filter by allowing the process and measurement models to be any general function and then linearizing them by means of a Taylor series expansion. This, of course, implies that the state transition and measurement functions be at least differentiable.

2.1.5. Particle Filtering

The Kalman filter was derived prior to the advent of modern computers, when computations were often carried out on pencil and paper, and the existence of a differentiable function was a computational necessity. However, with the massive computing power now cheaply and readily available, it is now possible to "emulate" functions by sampling over a large distribution until enough samples have been taken as to approach a continuous function. As the number of samples approaches infinity, the recovered probability density function becomes the actual function; in reality, however, it's possible to stop far short of infinity before the granularity is such that a working approximation of the probability density function has been obtained. This is similar to a computer graphics display emulating a curve, or a circle, by representing it as a large but finite number of discrete points, sampled according to a well-specified function.

The particle filter approach, rather than representing the state as an average/variance, represents state by a large (but still finite) number of samples (called “particles”, hence the name) of the same dimension as the state. In each iteration of the particle filter, each sample is moved probabilistically according to the process model (that is, the process model is applied to it and some random variance is added artificially) and then compared to the measurement probability to determine its likelihood (called the “importance weight” in particle filter literature). Finally, particles of low likelihood are eliminated and the current estimated state is given by the remaining particles.

```
Algorithm ParticleFilter( χ_{t-1}, u_t, z_t ):
        χ̂_t = χ_t = {}
        for m = 1 to M do
                sample x_t ~ p(x_t | u_t, x_{t-1})
                w = p(z_t | x_t)
                χ̂_t += { x_t, w_t }
        end
        for m = 1 to M do
                draw i with probability w_t
                χ_t += x_t
        end
        return χ_t
```

Figure 3 - Particle Filter

Shown above is the outline of the particle filter algorithm (Thrun, Burgard and Fox 2005).

Another important consideration in the use of particle filtering is making use of the data itself. The output of a particle filter is a (large) vector of hypothetical states, all clustered around the most likely actual state. Whereas with a Kalman filter, the output is an expected state with its corresponding covariance, additional computational work needs to be done by the implementer of the particle filter to determine where the particles are actually clustered to make use of the resulting data.

Three approaches to extracting a usable pose from particle filter output have been suggested. (Rekleitis 2003) The simplest is to use the weighted average of all of the particles. The primary problem here is that this can fail when presented with a multi-modal pdf – the

precise problem that particle filters were designed to avoid! Another is to use the "best particle" (that is, the one with the highest weight). Finally, a mixture of the two approaches called the "robust mean" approach can be used where a weighted mean is computed around the best particle.

The particle filter approach is relatively new – the EKF is far more prevalent in robotics literature at the time of this writing. Although the particle filter is conceptually simple, there are quite a few intricacies in implementing one effectively. One primary reason is the probabilistic nature of the output itself – it can be difficult to determine if an inaccurate response is received due to an implementation error or due to the noise inherent in the process.

2.1.5. Hybrid Approaches

Alternatively, as in the case of the Rao-Blackwellized particle filter approach, particle filters can be combined with Kalman filters. Here, the representations of the particles include EKF-managed gaussian distributions. Although this technique was originally formulated for the "FastSlam" implementation of the SLAM problem, it has since been generalized and applied in other contexts.

2.2 Kinematics

2.2.1. Motion Model

Although the real world is three-dimensional, most of its participants, being constrained by gravity, operate conceptually in two dimensions. Although there are exceptions even in the field of robotics (for example, flight and underwater navigation), this work will restrict itself to applications which, at least conceptually, are constrained to just two dimensions and whose position can therefore be described with two variables (x and y), relative to a base coordinate frame.

A simple model would be one where the robot under consideration was given a velocity in both the x and the y direction. To compensate for unexpected noise, a normally distributed random noise variable could be added to each velocity at each discrete time step.

Unfortunately, this model is too brittle to be applied in real life, as it doesn't account for orientation. If the motion error caused a shift in the orientation of the robot (which is very likely), a "forward" motion, even with some additive noise, could be in entirely the wrong direction. This error would accumulate over time, and the model would be unable to compensate for it.

This means that any realistic motion model must account for both translational (forward/backward, left/right) movement as well as angular movement. A simple, but realistic model of robot motion in two dimensions is one where the robot has two programmable velocities – a motion (translational) velocity and an angular (rotational) velocity. By adjusting these two velocities, the robot can be made to traverse any possible path. In this case, the robots pose p' [x' y' θ'](T) relative to a starting pose p [x y θ](T) after some time t is given by:

$\theta' = \theta + (\omega t)$

$x' = x + (vt)\cos\theta'$

$y' = y + (vt)\sin\theta'$

Where θ represents the robots orientation (relative to a fixed coordinate frame), ω represents the rotational velocity (how fast θ changes), and v is the translational velocity.

If the robot is moving with a translational velocity of v on a circle (caused by a constant rotational velocity w), the arc length traveled in one unit of time must be v. Likewise, if the rotational velocity is given by w, then the time taken to make a complete circle is (2 * pi)/w. As a result, the circumference of the given circle must be given by (2 * pi/w) * v. This relates v & w to the radius of the circle as v/w (it's important to note here that w is given in radians, not degrees).

In actual practice, the translational and rotational models are applied continuously, but when modeling mathematically, discretization must be performed. This has the potential to introduce error – it's important to ensure that the time steps are not too large, or the path modeled will be unrealistic. Figure 1, below, illustrates the difference in computed paths when the time steps are made too large relative to the diameter of the path being taken.

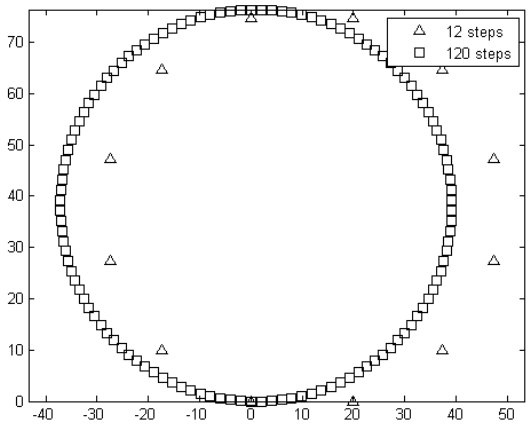

Figure 4 - Discretization Error

Here it is clear that although the point returns to the correct place in the end, the path taken is actually quite a bit different than the actual circle described by the applied velocity. There's a point of diminishing returns here, however – increasing the number of modeled steps from 120 to 360 does not introduce a significant correction:

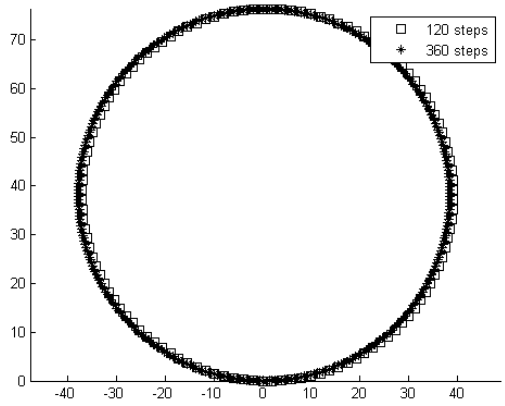

Figure 5 - Additional Steps

Minimizing this error means that the discrete time steps modeled must be a function of the current velocities (v/w) – as illustrated above, a good balance between computational efficiency and accuracy is (v/w) * π.

This is a real-world, workable process model. Unfortunately for Kalman filter applications, it doesn't lend itself to strict linearization because of the inclusion of the trigonometric functions – and trying to remove them would restrict the applications to those where a robot cannot make arbitrary turns. To see why this model can't be linearized, imagine that the robots initial pose is given by [x y θ]' = [0 0 π/2]'. Now the robot applies a control of [1 π/4]' (that is, it moves 1 unit of distance per time unit at a 45-degree angle). In the absence of noise, the pose should now be [-0.707 0.707 (3π / 4)]'. So far, so good – the motion model above does yield the correct result (again, in the absence of noise or error).

Now try to account for error. Assume the initial pose and the amount of uncertainty are given – so, at t=0, the pose is [0 0 π /2]' and the covariance matrix is given by [0 0 0; 0 0 0; 0 0 0] (that is, there is no uncertainty in this pose). For example, assume that the motion model [v w]' itself has an uncertainty of 0.5, π / 16 (that is, there is a 0.5 variance in the application of the translational velocity and a π /16 variance in the rotational velocity). The change to θ can range from (3π)/16 to (5π) / 16 (which can be accounted for), but since the final x & y coordinates are dependent on both the variance in v as well as the variance in w, the resulting uncertainty is no longer continuous. x, for example, can range from [0.9 * cos (5π) / 16, 0.9 * cos (3π) /16] = [0.278, 0.415] or from [1.1 * cos (5π)/16, 1.1 * cos (3π)/16] = [0.833, 1.24]… however, the range [0.415, 0.833] is no longer likely. This uncertainty has (quickly!) devolved into multiple hypotheses (a "multi-modal" posterior) and can no longer be represented as a simple gaussian.

This example is somewhat extreme (this represents a 50% uncertainty in the motion model – this is unlikely to be workable in practice regardless of the compensating controls), but even in the face of less uncertainty, the problem remains – some posterior positions are more likely than others due to the multiple independent sources of uncertainty. The problem here is not that the motion model is non-linear, but that the resulting probability density model is non-linear.

For this reason, any realistic motion model will need to be accounted for by either an extended Kalman filter or a particle filter.

2.2.2. Measurement Model

The process model is constrained by the controls of the robot, and the measurement model is constrained by the sensors of the robot. The simplest measurement model tries to distinguish "features" from observations (e.g. captured sensor data). It is important that these features "partially constrain the robot pose in the full space of robot poses." (Folkesson, Jensfelt and Christensen 2007) One complicating factor in applying measurements back to the localization problem is that measurements probabilistically identify features – which probabilistically constrain robot poses. In other words, unlike the process model, the measurement model includes an extra "level of indirection" between the measurement data and its interpretation in the overall problem.

This fact is in part due to the low resolution of the sensors. Human perception models are very detailed, and very abstract. We perceive, for example, walls, doors, doorknobs, signs, paint colors, etc., all of which help us "localize" ourselves. Robotic perception models, on the other hand, are constrained to very simple objects such as lines, curves, and planes. This means that, even with a perfect, noise-free measurement (which doesn't exist), multiple scans, taken across multiple poses, may be necessary to reconcile a pose with a map.

The measurement model assumes the existence of a map – in other words, when a measurement is taken, it can be matched against something. A simple, but workable, model for a map is a collection of distinguishable "features". The sensors would then report range (distance) and bearing (angular offset relative to the robot's angular orientation) to the known feature. Having recognizable features (that is, known "correspondences") simplifies the application of the measurement model, but isn't strictly necessary – a "maximum likelihood" estimation can be performed if a measurement might refer to one of several features.

Of course, in the true “SLAM” problem (simultaneous localization and mapping), the map is not previously known and must be built from sensor readings (which again report back range, bearing and some distinguishable “signature” for each recognized feature) which are then subsequently used to fix location.

2.3 Localization

The localization problem has been widely studied outside the context of robotics. There is significant interest in using RSSI and TOF data to localize a wireless node that is in control of a (mobile) human user, (Cavalieri 2007), (Ladd, et al. 2005), (Bahl and Padmabhan 2000). Some of the applications of such functionality include hazard notifications, print-to-nearest-printer, etc. It has not escaped the attention of researchers, however, that this same technique can be effectively used to localize mobile robots (Zaruba, et al. 2007), (Haeberlen, et al. 2004).

Wireless localization techniques fall into two broad categories – the first is the pattern-matching or “fingerprint” method. Here, the area in which nodes may range is calibrated by recording signal strengths associated with various access points at many positions in the ranging area, generating a “wireless signal strength map”. Once this calibration has been performed, a node scans the signal strengths received against those captured during calibration and assumes that its position is the one that most closely matches.

Pattern matching localization has been shown to work; however, its accuracy even in the best circumstances is in the resolution of a few meters. Unfortunately, the drift encountered by mobile robots is much lower than a few meters and pattern-matching localization is unsuitable for correcting such drift. A more precise method must be used for robotics applications.

Additionally, pattern-matching localization requires that the base stations themselves remain in fixed locations, or that calibration be re-performed if any of the base stations move. Although this is not as significant a problem as it may seem in actual use (base stations tend to

remain in fixed locations for very long periods of time), pattern-matching localization will also become unstable if any of the base stations suddenly goes offline, which is a more common, expected occurrence.

The second wireless localization technique makes use of known physical characteristics of radio-frequency signals and uses timing data to compute actual distances to base stations (Nanotron Technologies 2006). These computed distances can be used to very accurately triangulate a nodes physical location, in relation to the base stations.

This method, however, also requires that the base stations remain in fixed locations and remain powered up. An additional complicating factor here is that the locations of the base stations, relative to a global coordinate system, must be known with great precision – the accuracy of the localization computation can only be as accurate as the precision of the base station's known positions.

2.4 Simultaneous Localization and Mapping

In order for a robot to achieve true autonomy, it must be able to operate with a minimal amount of data about its surroundings. Specifically, it must be able to operate without an accurate map, building it's own internal map as it moves based on its observations about its environment. This problem is referred to as simultaneous localization and mapping, since the robot is localizing itself against a map even while it is building that map. Although this sounds like an intractable problem, it has been proven rigorously to be solvable (Dissanayake, et al. 2001), given enough time.

The essence of SLAM is to represent not only the position of the robot being tracked as a state element with associated uncertainty, but to also represent the position of each feature as a state element in a large, multidimensional state matrix (Riisgard and Blas 2005). If each feature has three state elements (e.g. x, y and θ), then the associated state matrix will be of size $(3s + 3) \times (3s + 3)$ where s is the number of features observed thus far. It's important to note

that as new features are observed, this matrix can grow in dimensionality, and the associated algorithms must be prepared to take this into account.

One important aspect of any SLAM implementation is feature recognition. It does no good for the robot to recognize, for example, that there is a wall 2 meters to the left, if it is incapable of recognizing that this is part of the same wall (or even the exact same section of wall) that it perceived in its last scan. This feature extraction is difficult to perform precisely and, like everything else in robotics, has an associated uncertainty which must be dealt with by the SLAM algorithm.

Also important is how the map is represented. Three common approaches are "grid-based", "feature-based" and "topological" (Zunino 2002). Each has associated strengths and weaknesses. The grid-based method divides the entire reachable range into a fine-grained mesh of squares (or cubes in a three-dimensional environment), each with a Boolean indicator "free" or "occupied". This is a useful representation for path planning algorithms (that is, how to get from point A to point B), but difficult to keep accurate in a dynamic environment (for example, one with people). The "feature based" representation keeps track of the coordinates (relative to a global coordinate system) of each observed feature, just as the position of the robot itself is tracked. This makes it easy to update the position of moving feature over time, but complicates path planning somewhat. Finally, the topological representation attempts to track and represent relationships between features rather than absolute positions. The primary drawback of the topological representation is its inability to correlate features that are physically far away – in actual practice, topological maps are combined into larger feature-based maps.

Most of the early work in SLAM used an Extended Kalman Filter (EKF) to update the associated state matrix. However, due to the size of the matrix, and the requirement that the matrix be inverted (a non-trivial operation) for each update step, considerable interest has been shown in speeding up this process. The FastSlam algorithm (Thrun, Burgard and Fox 2005) represents a combination of particle filtering and kalman filtering that speeds up the processing

considerably by only requiring the feature-associated state matrices to be updated when that feature is actually observed (as opposed to the EKF approach where every feature was part of a large state matrix).

Most SLAM research is focused on the use of “immediate” perception models such as acoustic or laser-range scanners, which return data about the area directly surrounding the robot to be localized. This information is important to a robot moving under its own power, as it must not allow itself to crash into any obstacles. However, considerable computational effort if expended in integrating this data into a larger map for localization purposes, since the data itself is immediate and has no global context.

This work examines the possibility of using “global” perception models such as wireless signal strength and time-of-flight data that span the entire range to be localized to speed up the localization process. At the same time, a local map can be built from immediate scanners and matched to the global map to assist in path planning.

CHAPTER 3

METHODOLOGY

3.1 Nanoloc

In order to capture time-of-flight and received signal strength values, an NA5TR1 (Nanotron n.d.) circuit board, manufactured by Nanotron, was used. This board makes efficient use of the ISM 2.4 GHz frequency band by employing “chirp spread spectrum” technology. The experiments described here use the nanoloc ATMega DK board which includes a programmable ATMega128 microcontroller as the interface between software and the radio circuitry hardware. 48 pins are exposed by the NA5TR1 circuit, although only a few of them are relevant here.

Pin 15, which controls the clock, is generated by the external microcontroller. Pins 17 & 18 are used to transmit and receive, respectively, data in a serial fashion, LSB first (configurable through the “SpiBitOrder” field). Data is transferred from the master (the microcontroller) to the NA5TR1 by outputting first the length of data to be transmitted as a single octet, the starting address as the second octet, and then 1-128 bytes of data, in octet boundaries.

When controlled by the ATMega 128 microcontroller on the ATMega DK board, port D is connected to the SPI (“serial peripheral interface”) interface and used to transfer data in this way. As such, the microcontroller acts as the interface between the nanoloc transceiver chip and the outside world. Most importantly, the microcontroller on the nanoloc DK board is also connected to an RS-232 serial port.

As shipped by the manufacturer, quite a bit of processing is being done by the microcontroller to convert time-of-flight and received signal strength values to distances, measured in meters. However, the microcontroller itself is limited in the amount of processing it

can realistically perform, and does not attempt to do any smoothing or filtering of the data itself, instead trusting the TOF and RSSI data (some simplistic averaging is performed by the client-side data, but since the input from the device is a range measurement in meters, this can only do a certain amount of smoothing).

For the purposes of this study, it was more useful to pass the raw data, as measured by the nanoloc transceiver chip, directly through the microcontroller out through the serial interface so that it can be processed by a more sophisticated software module than the microcontroller environment permits (specifically, the particle filter outlined above).

3.1.1. Serial Peripheral Interface Programming

The microcontroller (the “master”) communicates with the NTRX (the “slave”) through a serial-peripheral interface. Port B on the microcontroller (pins 10-17) is used to transmit and receive to and from the microcontroller. However, although port B is 8-bits wide, the input interface to the NTRX device is only 1 bit, so B0 (master pin 10) is attached to SpiSSN (slave pin 16) which indicates that data is available to send, and the SPSR (an 8-bit shift register) and SPDR (an index register) registers are used to shift data to and from the NA5TR1 hardware. The connection is shown below.

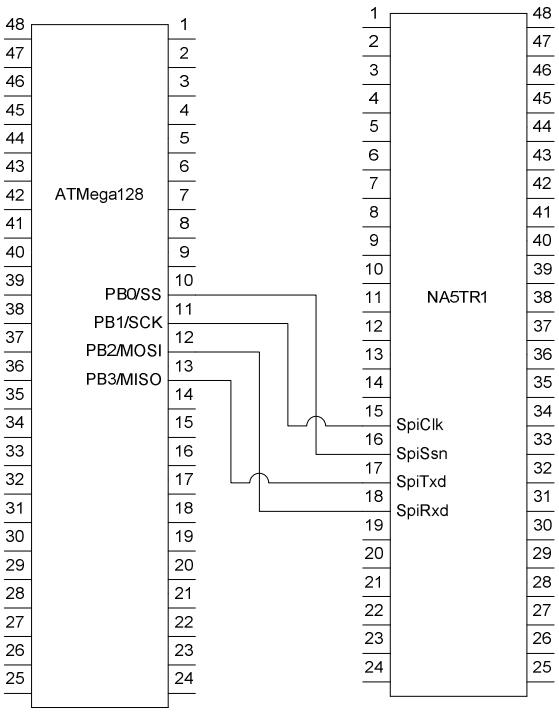

Figure 6 - Nanoloc/ATM128 connectors

The NTRX chip, in turn, expects a specific input format – the first byte is the length of the data that follows, the second is the address to which the data should be written, and the subsequent N bytes are the data itself.

3.1.2. Symmetric Double Sided Two-way ranging

The nanoloc transceiver uses multiple time-of-flight measurements to provide robust ranging capabilities. Specifically, the initiator of the ranging function (the mobile device) initiates a “range” call which is addressed to a specific anchor (identified by a unique MAC address). This is received by the anchor, which then automatically initiates another “range” call back to the mobile device. The mobile device responds, and the anchor returns the processing delay and time-of-flight of both calls to the mobile device. This is what nanotron refers to as “Symmetric double sided two-way ranging” and is responsible for the accuracy of the location information.

The principle benefit of SDS-TWR is that the devices clocks do not have to be synchronized to a high degree of accuracy. By measuring the differences as seen by each devices local clock, a fairly accurate measure of actual elapsed time can be inferred. The measuring device will record the time at the start of the “range” call and record the time that the response is received. One half of this time would be a good estimate of the signal propagation time between the sender and the receiver, except for the fact that the receiver must also spend some time processing the incoming call and preparing an acknowledgment. For this reason, the response packet also includes the processing time at the receiver so that it can be subtracted by the caller. This process is illustrated below.

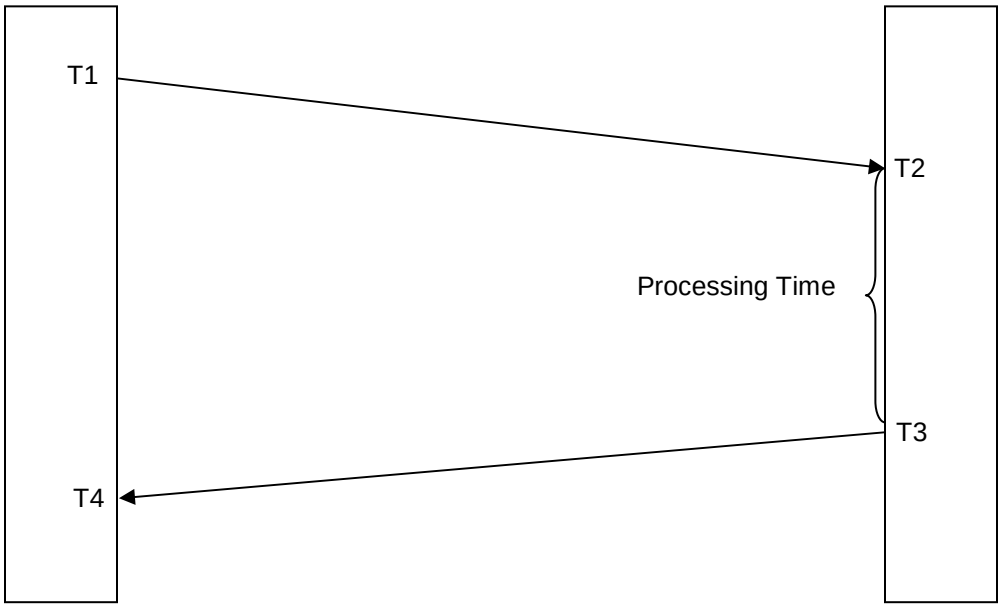

Figure 7 - TOF = (T4 - T1 - (T3 - T2)) / 2

3.1.3. Original Localization Flow

The nanoloc device ships with a few demonstration applications, one of which is a localization application. This application gathers ranging data from 4 or more (fixed-position) access points and uses this data to compute the location of a moving node. The overall

program flow is outlined in appendix A. The location itself is computed from time-of-flight data as follows.

3 16-bit values, TXRespTime t, RxUcSum r, and TxUcSum l, are output over the SPI. The distance is computed as

$$\left(\frac{\frac{t}{4}-\frac{g}{32}-\frac{(r+l)\frac{2000}{244175}}{48}}{2}-c\right)a \qquad \text{Equation 1}$$

Where c is the ranging constant, which is determined by the radio frequency and symbol rate as detailed in Table 1 below, a is the "Speed of air" constant 299.792458 and g is set to 7 if phaseoffsetack = 7, 6-p otherwise. This is calculated once for the data set in the receiver, and again for the dataset in the sender and, as long as the difference is not greater than 6 (meters), the results are averaged and returned to the caller. It's interesting to note that received signal strength is tracked and available in internal register 0x26, but not used in the distance computation above.

Table 1 - Ranging Constants

Radio Frequency/Symbol Duration	Ranging Constant
800 MHz/500 ns	68.929336
80 MHz/1000ns	122.492363
80 MHz/2000ns	229.490053
80 MHz/4000ns	445.584595
22 MHz/4000ns	445.553589

3.1.4. Modified Localization Flow

The results given back by the localization example from Nanotron precompute the location, and suffer from a level of inaccuracy. More useful is the time-of-flight and received signal strength data, which isn't, by default, output. Therefore, a modified version of the location demonstration code was used for the experiments described here. By sending the raw ToF and

RSSI data over the serial port to the host computer, the data itself could be passed through a particle filter implementation and smoothed out for increased accuracy.

These changes primarily consisted of modifications to the tag's app.c file's PollApplication function (see Appendix A for details on the nanoloc flow) as shown below:

```
void  PollApplication(void)
{
  MyDouble32T dist;
  printf_P(PSTR("\n%u 11 "), TCNT1);
  app->dest[ 0 ] = 0x11;
  dist = NTRXRange(app->dest);

  printf_P(PSTR("\n%u 12 "), TCNT1);
  app->dest[ 0 ] = 0x12;
  dist = NTRXRange(app->dest);

  printf_P(PSTR("\n%u 13 "), TCNT1);
  app->dest[ 0 ] = 0x13;
  dist = NTRXRange(app->dest);
}
```

Figure 8 - app.c modifications

```
MyDouble32T dist(MyByte8T *p1, MyByte8T *p2)
{
	MyByte8T rssi;

	MyDouble32T avg = -1.0;
	MyDouble32T speedofmedium = SPEED_OF_AIR;

	/* calculate the one way airtime for local station */
	MyDouble32T distanceD2R = delay(p1);
	/* calculate the one way airtime for remote station */
	MyDouble32T distanceR2D = delay(p2);



	/* calculate the distance in [m] for local station */
	distanceR2D *=(speedofmedium);
	/* calculate the distance in [m] for remote station */
	distanceD2R *=(speedofmedium);

printf_P(PSTR("%7.2f %7.2f"), distanceR2D, distanceD2R);

	/* the difference between the measurement results from local-1
	 * and remote station should not be to large */
	if(distanceR2D>distanceD2R)
	{
		if((distanceR2D-distanceD2R)>MAX_DIFF_ALLOWED) return -1.0;
	}else{
		if((distanceD2R-distanceR2D)>MAX_DIFF_ALLOWED) return -1.0;
	}

	avg  = ((distanceR2D + distanceD2R )/2.0); /* [m] */

	NTRXReadSingleSPI( 0x26, &rssi );
	printf_P(PSTR(" %d"), rssi);

	return avg;
}

MyDouble32T delay(MyByte8T *p)
{
	/* clock period [MHz] */
	const MyDouble32T clk_4MHz  = 4;
	const MyDouble32T clk_32MHz = 32;
	/* Scaled 1:20 divider's clock period [MHz] */
	const MyDouble32T clk_lod20 = (2000.0/244175);

	const MyWord16T PulseDetUcMax = 5;
	const MyWord16T PulseDetUcMax_table[16] = {1, 2, 4, 8, 16, 24, 32,  40,
```

Figure 9 - NTRXRanging.c modifications Part I

```
48, 56, 64, 1, 1, 1, 1, 1};

		MyDouble32T res;

		MyDword32T TxRespTime = (p[TXRESPTIME_H] << 8) |
(p[TXRESPTIME_L]);
		MyLong32T  RxUcSum = (p[TOAOFFSETMEANACK_H]  << 8) |
(p[TOAOFFSETMEANACK_L]);
		MyLong32T  TxUcSum  = (p[TOAOFFSETMEANDATA_H] << 8) |
(p[TOAOFFSETMEANDATA_L]);

		MyLong32T  RxGateOff  = p[PHASEOFFSETACK]  == 7 ? 7 : 6 -
p[PHASEOFFSETACK];
		MyLong32T  TxGateOff  = p[PHASEOFFSETDATA] == 7 ? 7 : 6 -
p[PHASEOFFSETDATA];

		if (settingVal.fdma == 0)
		{
			res = ((TxRespTime)/clk_4MHz
			- ((TxGateOff+RxGateOff))/clk_32MHz
				-
(TxUcSum+RxUcSum)*clk_lod20/(2.0*PulseDetUcMax_table[PulseDetUcMax]))/2.0
				- settingVal.rangingConst;
		}else{
			res = (TxRespTime/clk_4MHz -
(TxGateOff+RxGateOff)/clk_32MHz
				-
(TxUcSum+RxUcSum)/clk_32MHz*(2.0*PulseDetUcMax_table[PulseDetUcMax]))/2.0
				- settingVal.rangingConst;
		}

	printf_P(PSTR("%lu %ld %ld %ld %ld %g "),
			TxRespTime,
			RxUcSum,
			TxUcSum,
			RxGateOff,
			TxGateOff,
			res );

		return res;
	}
```

Figure 10 - NTRXRanging.c modifications Part II

This causes each ranging call to output, in order, the values t, r, l, g (split into two values) as documented in equation 1 on page 30, as well as the computed delay, distance and the RSSI value stored in nanoloc register 0x26.

3.1.5. Triangulation using computed distances

This gives the distance from one node to another, based on time of flight information. To compute a location, a triangulation must be performed. There are two fundamental types of triangulation – lateration and angulation (Liu, et al. 2002). At least one more fixed reference point must be available than the number of dimensions in which localization is being attempted (i.e. to triangulate in two dimensions, three fixed reference points must be available). If only one reference point is available, the returned distance can only be used to fix location on a ring around the reference point, as illustrated in Figure 11, below.

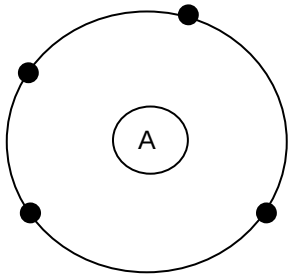

Figure 11 – "Localizing" with one reference point

Adding a second reference point narrows the possibilities to two possible points – the intersection points of the circles around the reference points. These can be computed (in two dimensions) based on the time-of-flight information as illustrated below:

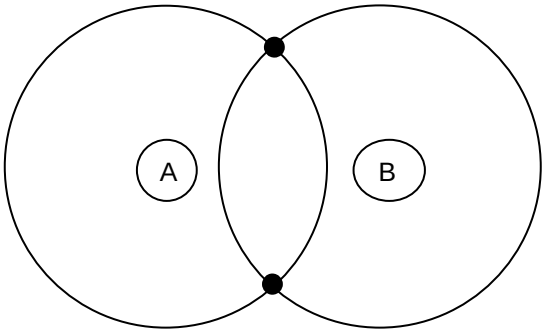

Figure 12 - Localizing with two fixed reference points

Computing the intersection points of two circles with fixed radii is a straightforward algebraic exercise (Borke 1997).

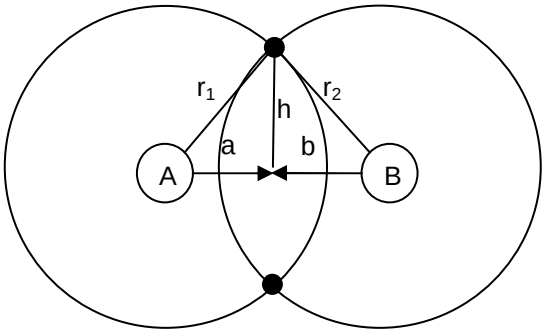

Figure 13 - Computing intersections of circles

a + b, above, is equal to the distance between the two reference points (the centers of the circles) d, which is easily computed given that their positions are known. Since $a^2 + h^2 = r_1^2$, and $b^2 + h^2 = r_2^{2:}$

$$h^2 = r_2^2 - b^2$$

$$a^2 = r_1^2 - h^2$$

$$a^2 = r_1^2 - (r_2^2 - b^2)$$

$$a^2 = r_1^2 - r_2^2 + (d - a)^2$$

$$a^2 = r_1^2 - r_2^2 + d^2 - 2ad + a^2$$

$$2ad = r_1^2 - r_2^2 + d^2$$

$$a = \frac{r_1^2 - r_2^2 + d^2}{2d}$$

This *a* value can then be used to solve for h using $h = \sqrt{r_1^2 - a^2}$. Given a and h, the point where lines a, b and h meet is given by:

$$x_2 = x_0 + dx\,\frac{d}{a}$$

$$y_2 = y_0 + dy\,\frac{d}{a}$$

And the actual intersection points of the two circles are given by:

$$x_3 = x_2 \pm dy\frac{h}{d}$$

$$y_3 = y_2 \pm dx\frac{h}{d}$$

To properly localize with high certainty in two dimensions, three reference points must be known, as illustrated below.

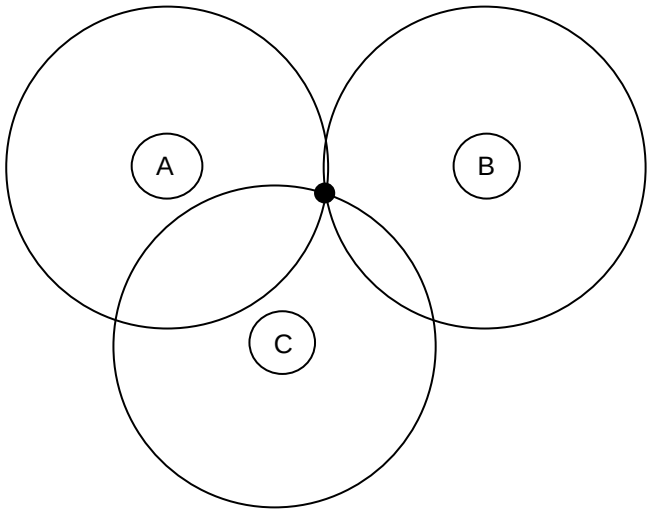

Figure 14 - Proper localization with three reference points

Here, the process is simple – find the intersection points of circles A & B, and the intersection points of circles A & C, and find the one intersection point that appears in both. Note that there is no guarantee that there is a unique intersection point of three arbitrary circles (or even two unique intersection points of two arbitrary circles, since one could be contained within the other, or they could be farther away than their radii allow) – however, in the context of

triangulating from wireless signal data, it is guaranteed that such a point exists, since the radii are the distances from such a point (all that is required of the implementation is to figure out what that point is).

This same concept extends out to higher dimensions – to fix a point in three dimensions, four fixed reference points must be known. However, it is realistic and useful to localize in a three-dimensional environment using only three reference points by assuming that all points are coplanar. This is feasible as long as the differences in the ignored dimension (typically the "z" axis) are not too significant relative to the distances being measured in the other dimensions.

3.2 Particle Filter Implementation

3.2.1. Motion Model Implementation

To evaluate the particle filter, a motion model was simulated. The motion model implementation was a straightforward implementation of the model described in 2.2.1. Motion Model on page 17. This model can be implemented with straightforward matrix calculations where the state vector is represented as $[x\ y\ \theta]'$ and the control vector is modeled as $[v\ \omega]$. Since this was simulated, the simulation needs to be notified of each movement and the distance moved at each step.

3.2.2. Particle Filter Analysis

To establish upper and lower bounds on the realistic performance of the particle filter implementation proposed, various simulated scenarios were run with different noise parameters. Various worst case and best case scenarios were examined. Two important factors are the amount of noise that the implementation can recover from in both the sensor model as well as the motion model, and the number of particles required to achieve tolerable performance, as this has a direct bearing on not only the memory requirements of an

implementation, but on the speed at which the robot can localize and therefore the maximum speed at which it can move.

Throughout this work, data will be derived from first principles, rather than using readily available advanced features of the tools at hand (Matlab, for example). This is done because interim results are usually not made available by the advanced processing tools, and the interim results are those of the most interest here.

A metric needs to be established to measure the quality of a particle filter run. Here, two important metrics are examined – the first is how long the implementation takes to stabilize, and the second is how far from "reality" the stabilized prediction is. Stabilization here is defined as the number of iterations that are required before the measured error (the difference between the predicted pose and the actual pose) does not display a standard deviation of greater than 2 for at least three consecutive iterations. After this point, the particle filter has localized itself (erroneously or not) and as implemented, it will not change significantly. This iteration count is important because the robot must move in between each iteration of the algorithm. Since the primary use case here is one of a robot that knows nothing about its position, these moves are made under significant uncertainty and therefore must be minimized as much as possible.

Note that the particle filter will always settle on a location and propagate that location, regardless of how bad the input data is. Even with random sensor data, the particle filter does stabilize, although the stabilized value is invalid and tends to occur at the center of the range of the implementation.

3.2.3. Effects of particle count on behavior

The number of particles used to represent the posterior was measured through a series of simulations and captured within 5% with a 95% confidence level. It was determined that the number of particles had no bearing on how long the algorithm took to stabilize – with fixed sensor and motion noise (discussed below), the algorithm always stabilized after 10 iterations, although the final residual error was impacted by the choice of particle count.

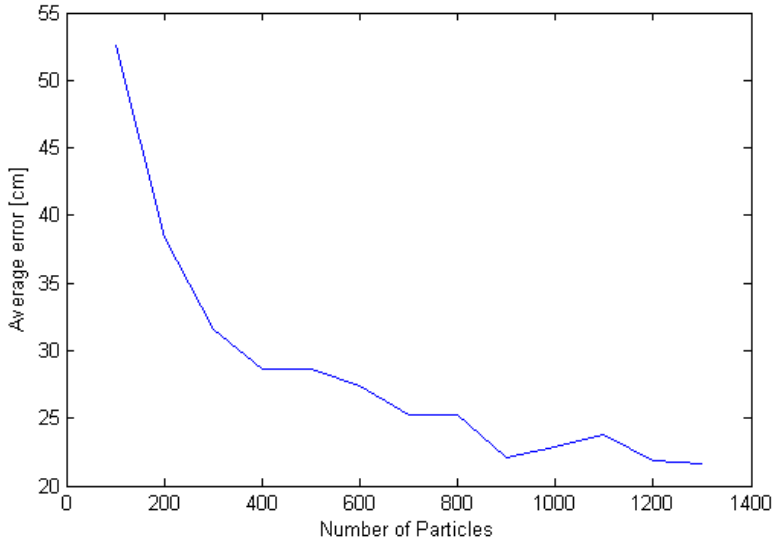

Figure 15 - Effect of particle count on measured error

Figure 15, above, illustrates the effect of particle count on the measured error in a 25 square meter area. Although the error is large with just 100 particles, it declines quickly and stabilizes near 700. 1000 was chosen as the optimal number for remaining studies as a reasonable balance between computational efficiency and error minimization.

3.2.4. Effects of sensor noise and motion noise

Here the effects sensor noise and motion noise on stabilization and final error are examined. In all simulations, sensor noise and motion noise are assumed to be characterized by a zero-mean Gaussian random variable – the variable parameter here is the standard deviation of the noise variable.

One of the parameters of the algorithm is the expected noise, typically determined experimentally. Shown below are the stabilization time (in algorithm iterations) and final error after stabilization when the motion "noise" (rotational and translational drift) match the parameters input to the function.

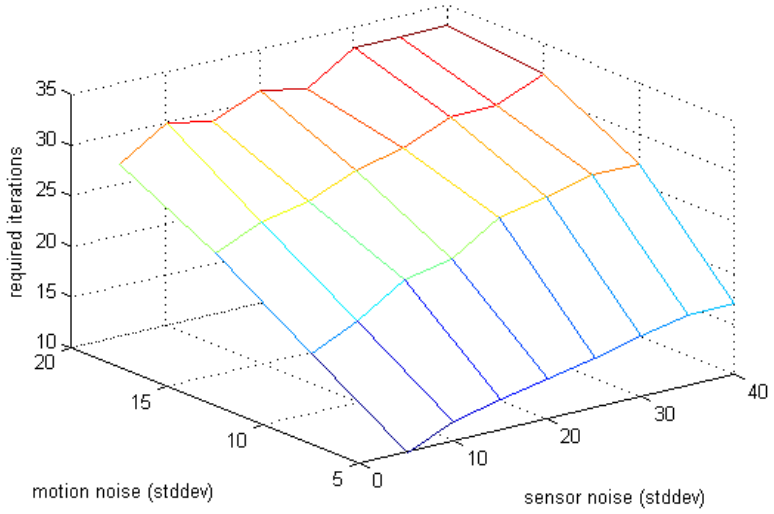

Figure 16 - Required iterations for stabilization

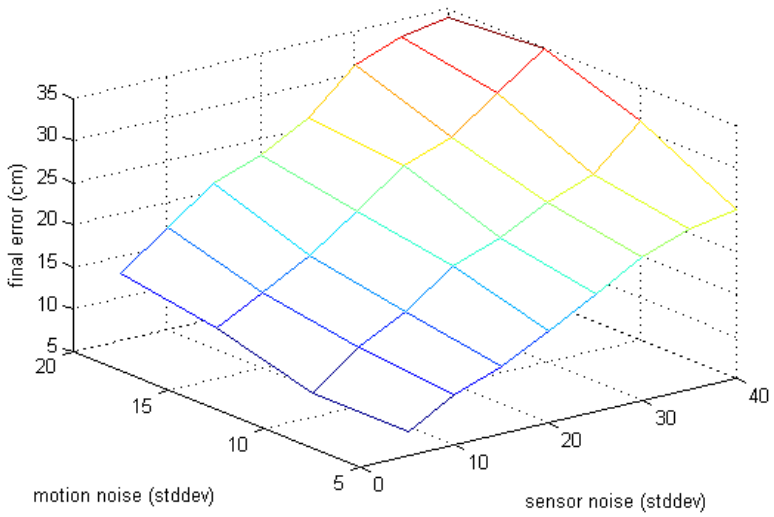

Figure 17 - Residual error after stabilization

The motion noise had a greater effect on the number of iterations that were required to reach a stabilized position, but the sensor noise had a greater impact on the error once stabilization had been achieved.

Also of interest is the more realistic case where the expected noise does not match the actual observed noise. Since it's more likely that noise will be underestimated than

overestimated, the graphs below show the time to stabilize and the resultant error when a standard normal variance is assumed but higher variances are observed. As expected, the motion noise dominates in both required iterations and residual error (since estimated sensor noise is not a parameter to the algorithm). Even in the worst simulated case below, the final error is approximately one meter, although the number of iterations required to reach stabilization is so high as to make its use impractical.

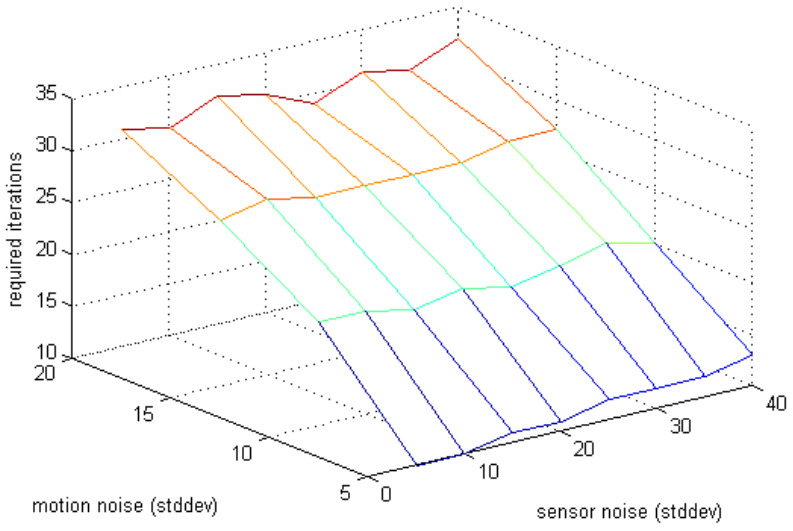

Figure 18 - Required iterations with underestimated noise

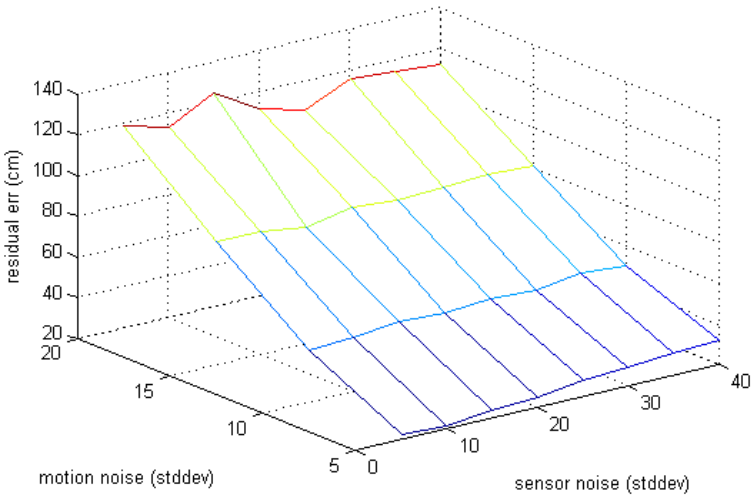

Figure 19 - Residual error with underestimated noise

CHAPTER 4

RESULTS

This section details the performance of the algorithm when faced with real data signals from live hardware. In general, it is found to perform in line with the expectations set by the simulated results.

4.1 Default Implementation

The Nanotron hardware ships with two example applications ready to be flashed onto the master microcontroller – one is the location demo, which uses triangulation to locate a roaming tag in relation to fixed (known) reference anchor points, and the other is the ranging demo that simply computes the distance from one node to another and writes the result through the serial port interface.

Although the Nanotron package ships with a demo application for reading range data, it's not presented in a form useful for offline analysis. Instead, the following simple application code was used to capture the textual data from the serial port.

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <termios.h>
#include <math.h>
#include <iostream>
#include "serialbuffer.h"

// comment below if tag is sending G values
//#define RAW_ACC_VALUES
```

Figure 20 - range.cpp Part I

```
using namespace std;


int open_port()
{
  int fd;

if((fd=open("/dev/ttyS0", O_RDWR | O_NOCTTY | O_NDELAY))==-1)
    cerr<<"Cannot open Serial USB Port"<<endl;
  else

    fcntl(fd,F_SETFL,0);
  return(fd);
}

void replace_r_with_n(char * bf)
{
  int i = 0;
  while(bf[i]!=0)
    {
      if(bf[i]=='\r')
            bf[i]='\n';
      i++;
    }
}

int main(char c, char** v)
{
  int fd;
  struct termios options;

  ssize_t aa;
  char bf[9];

  fd=open_port();
  tcgetattr(fd,&options);
  cfsetispeed(&options , B38400);
  cfsetospeed(&options , B38400);
  options.c_cflag |= (CLOCAL | CREAD);
  //8N1
  options.c_cflag &= ~PARENB;
  options.c_cflag &= ~CSTOPB;
  options.c_cflag &= ~CSIZE;
  options.c_cflag |= CS8;
  //No Flow Control
  options.c_cflag &= ~CRTSCTS;
  //Raw input
  options.c_lflag &= ~(ICANON | ECHO | ECHOE | ISIG);
  //No software flow control
```

Figure 21 - range.cpp Part II

```
    options.c_iflag &= ~(IXON | IXOFF | IXANY);
    //Raw output
    options.c_oflag &= ~OPOST;
    tcsetattr(fd, TCSANOW , &options);

    cout<<"Port:"<<fd<<endl;
    fcntl(fd, F_SETFL , FNDELAY);

    while ( 1 )
    {
      aa = read( fd, bf, 9 );
      if ( aa == 9 )
      {
        bf[8] = 0;
        printf( "%s", bf );
      }
    }
  }
```

Figure 22 - range.cpp Part III

4.1.1. Reported Distances

For this experiment, the default nanoloc "ranging" hex files were loaded onto two nodes labeled "AP1" and "AP4". AP4 was connected to a computer serial port and the code in Figure 20 was run to capture the location information. The table below shows the relationship between measured distances and reported distances. The range reported by the node was inaccurate by a factor of more almost five – 1000 samples were gathered over a period of 14 s, after giving the nodes a few seconds to stabilize. After removing invalid data points (unreadable ranges, likely caused by timeouts, were reported as -1), the average reported distance was 6.16 m with a standard deviation of 0.4596.

Table 2 - Measured vs. Actual Distances

Actual Distance	Reported Distance
0.1 m	0.76759 m
0.3 m	1.18428 m
1 m	3.32806 m

To investigate the feasibility of using time-of-flight data and RSSI for actual location calculation, the ranging code on the microcontroller was modified to output this data along with the precomputed range.

4.1.2. Time-of-flight data

After modifying the firmware, 15 data elements are returned for each anchor point for each scan. The meaning of each data element is described in the table below.

Table 3 – Returned data elements

Element	Origin	Meaning
1	N/A	MAC ID of access point (used to distinguish access points from one another)
2	tag	Transmission response time (ms)
3	tag	TOA Offset mean ack (local propagation delay)
4	tag	TOA Offset mean data (remote propagation delay)
5	tag	Phase offset ack (local)
6	tag	Phase offset delay (remote)
7	tag	Delay (as computed by equation 1)
8	anchor	Transmission response time (ms)
9	anchor	TOA Offset mean ack (local propagation delay)
10	anchor	TOA Offset mean data (remote propagation delay)
11	anchor	Phase offset ack (local)
12	anchor	Phase offset delay (remote)
13	anchor	Delay (as computed by equation 1)
14	tag	Distance (element 7 * speed of medium)
15	anchor	Distance (element 13 * speed of medium)

Elements 14 & 15 should agree or be very close to one another.

To process the data, it's important that accurate timestamps be associated with each record. It was discovered during testing that the resolution of the clocks in the test processing hardware (the laptops) was significantly lower than that of the location hardware (the nodes), so the internal 16-bit timer of the tag hardware had to be enabled to distinguish individual records from one another.

The experimental relationship between time-of-flight and RSSI data between two of the nanoloc devices is summarized here, as a function of actual distance.

Table 4 - average and standard deviation of data points

Measured Distance	RxUcSum (tag)	TxUcSum (tag)	Delay (tag) 10^{-3}	RxUcSum (anchor)	TxUcSum (anchor)	Delay (anchor) 10^{-3}	RSSI
0.1 m	686.34/ 56.2871	684.948/ 56.7429	2.1/ 0.9	680.71/ 58.8009	685.54/ 57.4393	2.9/ 0.9	0/0
0.3 m	679.025/ 61.70598	682.226/ 60.963263	3.204/ 0.834	681.2230/ 59.782382	681.654/ 61.0545	4.237/ 0.881	3.982/ 0.172353
0.5 m	687.903/ 63.550596	679.424/ 63.468847	3.686/ 0.773	678.519/ 62.608973	688.588/ 63.389261	4.830/ 0.775	10.169/ 0.524131
1 m	687.177/ 75.306459	682.019/ 74.443270	8.355/ 0.994	676.847/ 74.140221	678.32/ 75.458290	9.474/ 1.041	12.071/ 0.992443
2 m	692.994/ 92.514501	681.401/ 91.890522	21.967/ 2.853	681.767/ 91.178771	682.483/ 91.256613	23.161/ 6.380	14.329/ 1.683922

In Table 4, above, the average and standard deviation of 1000 samples, drawn over a period of 80 seconds, are shown. The MATLAB code used to compute the data points is illustrated below:

```
s1 = serial('COM4', 'BaudRate', 38400);
fopen(s1);
try
   count_samples = 1;
   max_samples = 3000;
   samples = [];
   first = 1;
   tic();
```

Figure 23 - Offline sampling code Part I

```
            while ( count_samples <= max_samples )
               line = fgetl(s1);
               % discard the first line since it will almost definitely be a partial
               % reading
               if first
                  first = 0;
                  continue;
               end
               samples(count_samples,1) = toc();
               col = 2;
               if ( line )
                  count_samples = count_samples + 1;
               end
               while ( line )
                  [token, line] = strtok(line);
                  if token
                     samples(count_samples, col) = str2num( token );
                  end
                  col = col + 1;
               end
            end

            fclose(s1);

            anchor1_samples = samples(samples(:,3) == 11, :);
            anchor2_samples = samples(samples(:,3) == 12, :);
            anchor3_samples = samples(samples(:,3) == 13, :);

            disp(sprintf( '11: rx = %f/%f, tx = %f/%f, del = %f/%f, rx = %f/%f, tx = %f/%f, del =
%f/%f, rssi = %f/%f', ...
               mean(anchor1_samples(:,5)), std(anchor1_samples(:,5)), ...
               mean(anchor1_samples(:,6)), std(anchor1_samples(:,6)), ...
               mean(anchor1_samples(:,9)), std(anchor1_samples(:,9)), ...
               mean(anchor1_samples(:,11)), std(anchor1_samples(:,11)), ...
               mean(anchor1_samples(:,12)), std(anchor1_samples(:,12)), ...
               mean(anchor1_samples(:,15)), std(anchor1_samples(:,15)), ...
               mean(anchor1_samples(:,18)), std(anchor1_samples(:,18))));
         catch
            fclose(s1);
            rethrow(lasterror);
         end
```

Figure 24 - offline sampling code Part II

4.1.3. Multiple Access Points

The data above captures the behavior of the tag when it is standing still. Here, it's behavior when it is in motion, which is of much more interest to the problem of simultaneous localization and mapping, is exampled.

After the nanoloc hardware was reprogrammed to capture time-of-flight data along with precomputed range measurements, three fixed access points were set up in the General Academic and Classroom Building (GACB) on the UTA campus. The GACB is 2,337.3 cm along its north facing wall (referred to as hallway 1 below) and 2,226 cm along its east-facing wall (referred to as hallway 2). The three access points were located at the far ends of each hall as illustrated below.

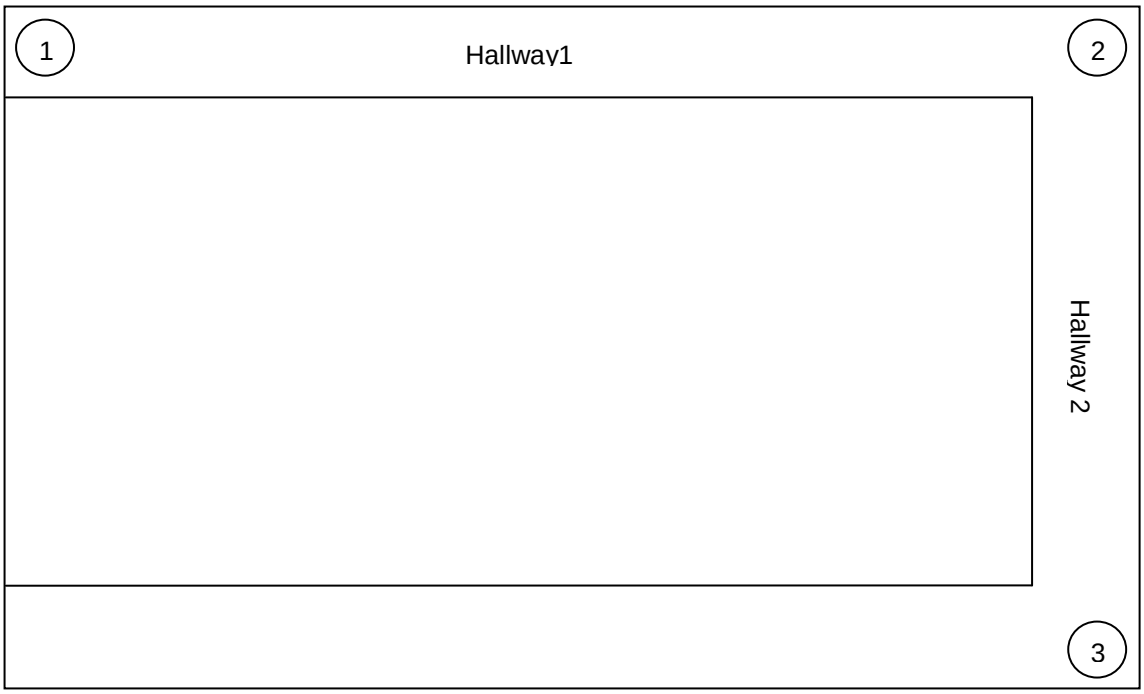

Figure 25 - Access point setup

The "roaming" tag was allowed to move along hallway 1 and hallway 2 at a constant speed, and data from each access point was sampled at the maximum rate allowed by the

hardware (approximately 37 samples/second). The path followed by the roaming tag is illustrated below.

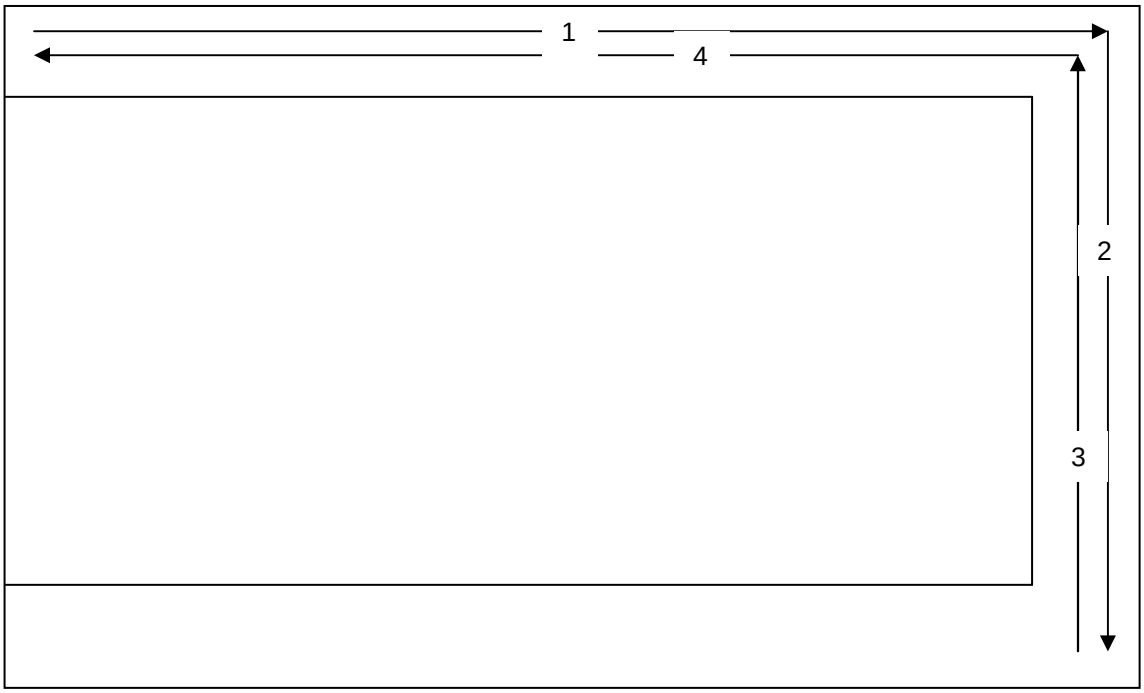

Figure 26 - Roaming tag path

The time for each experiment is summarized below.

Table 5 - GACB experiment elapsed time

Experiment	Elapsed Time (s)	Distance	Average Speed
1	61	2,337 cm	38.31 cm/s
2	50	2,226 cm	44.52 cm/s
3	48	2,226 cm	46.38 cm/s
4	56	2,337 cm	41.73 cm/s

Assuming a (roughly) constant rate of speed, the location during the experiment can be inferred from the elapsed time. The charts below illustrate the recorded delay times for each experiment.

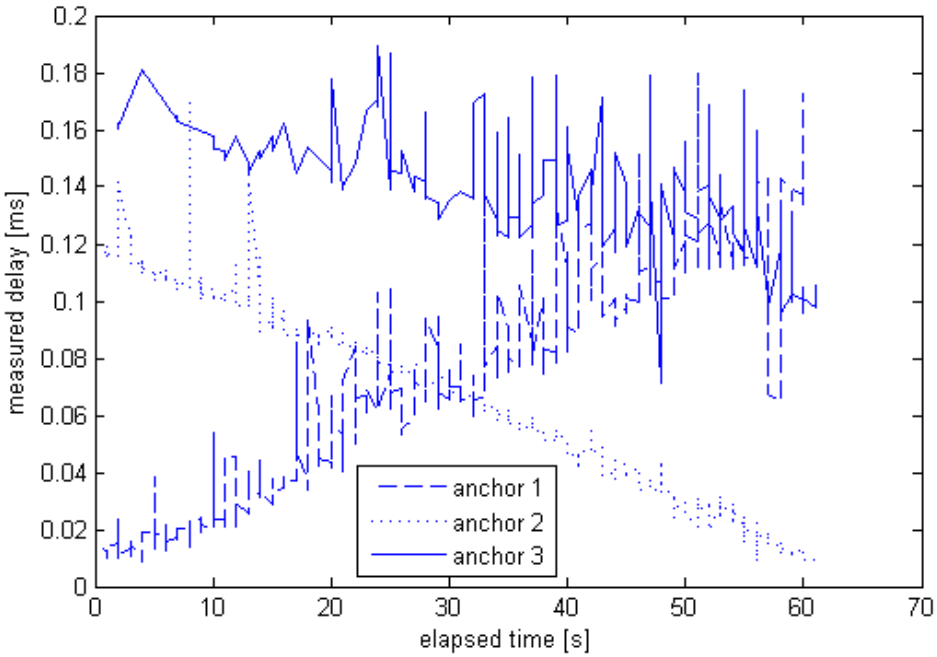

Figure 27 - Hallway1 recorded delay

Here the measured time of flight (delay) between the tag and each anchor point as the tag was moved along hallway 1 is illustrated. Short readings (those that registered as 0 delay) have been omitted from the graph. The erratic curve associated with access point 3 is due to the fact that access point 3 is almost out of range at the start of the experiment. Also note the wild disagreement between sensor points 1 and 3 toward the end. They're on opposite corners of an almost square building – they should be reporting similar information, but they're not, due to the effects of interference.

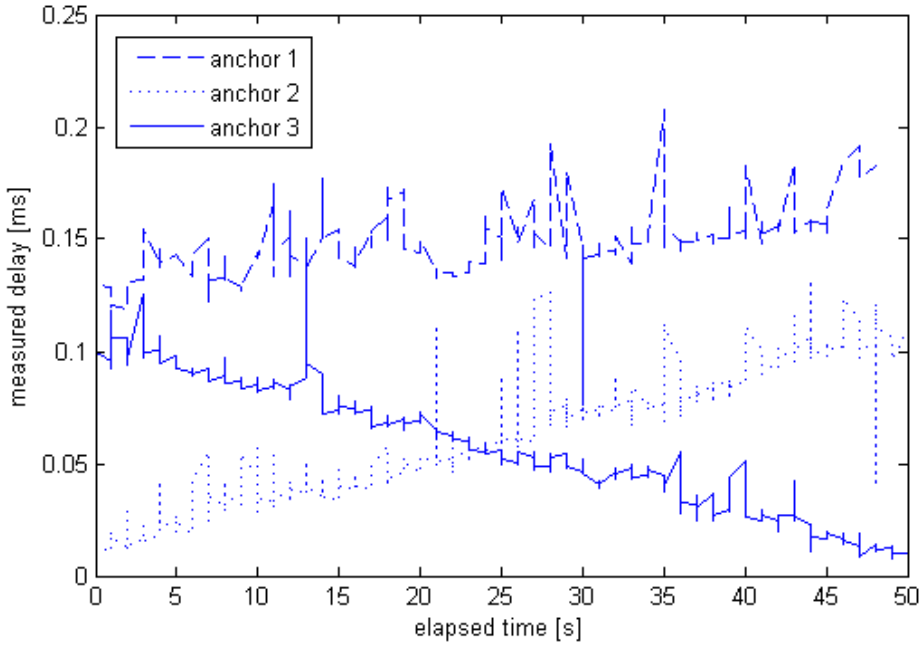

Figure 28 - Hallway2 recorded delay

Here the same data for hallway 2 is illustrated – as can be seen from the chart, access point 1 is beginning to fall out of range toward the end of the experiment. Since hallways 3 and 4 show essentially the same data as 2 and 1, but in reverse, they are not presented here.

Also of not are the RSSI readings recorded by the tag. RSSI readings here are inverted – the lower the value, the closer the tag is.

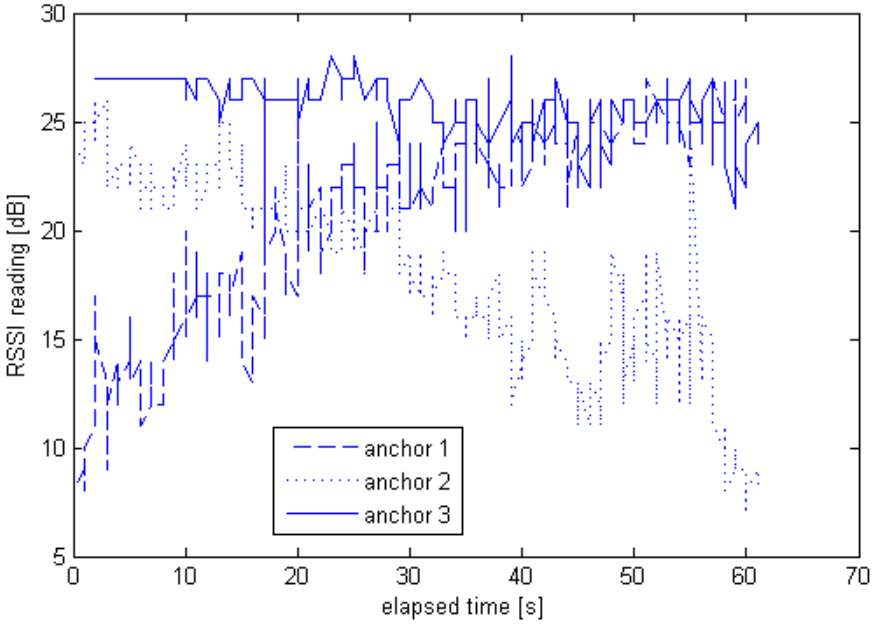

Figure 29 - Hallway1 recorded signal strength

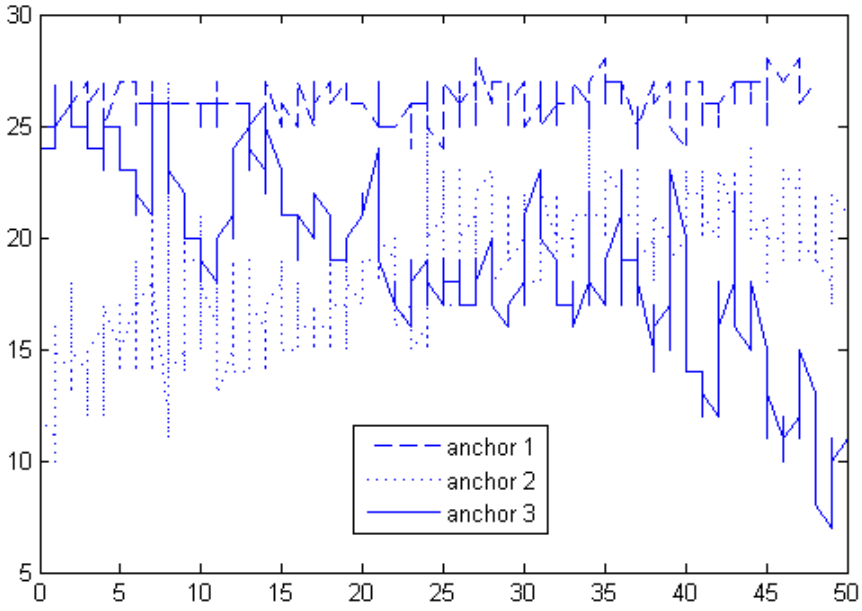

Figure 30 - Hallway2 recorded signal strength

As would be expected, anchor point 3, being essentially out of range for most of the hallway 1 experiment, is at the top of its recorded range (the nanoloc hardware records only 5 bits of data for each RSSI reading) and the same is true for anchor point 1 for most of the hallway 2 experiment.

4.2 Triangulation

Although the triangulation technique suggested in section 3.1.4 is a sound theoretical means of inferring location from a set of reported ranges, it fails in actual practice due to variations in reported data. The average distances for second 3 reported by the remote tags were 293.19 cm, 2,619.44 cm, and 3,227.67 cm, respectively. Unfortunately, at these three radii, the circles themselves never actually overlap.

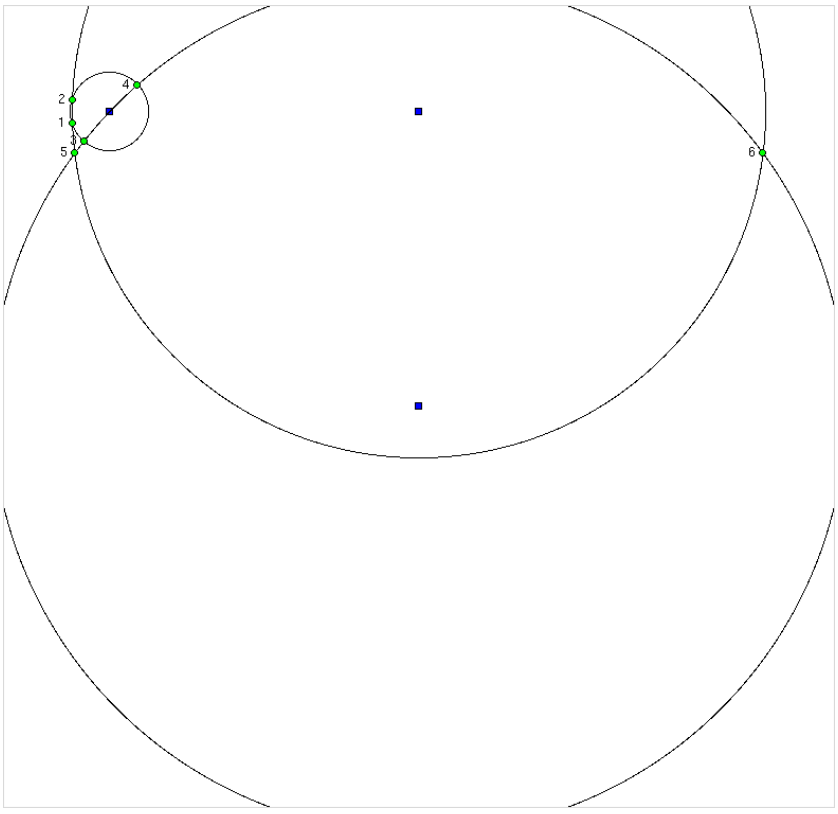

Figure 31 – Non-overlapping ranges

The intersection points are illustrated above. Here, the closest overlap points are 3 & 5, but they're actually 15 cm apart. Although it would be feasible to introduce a tolerance when

comparing for overlap here, it makes more sense compare the distances all 5! = 14 possible combinations of points and accept the two with the smallest distance as the most likely candidate. (Also note that, in the data above, this most likely point actually falls well outside the range of possible area and is quite a distance from the actual measured distance – this will be addressed below).

Additionally, not every sensor returns data in every sensing cycle. Sensors 1 & 3 go out of range in the far corners of the building, and it's necessary to attempt to localize with readings from only two sensors. Although in the general sense this is impossible as illustrated in Figure 12, it's still possible to make use of this data in a particle filter setting. In this case, the sensed point closest to the particles current proposal is assumed to be the correct point. Over time, this will result in proper localization, albeit more slowly than if three reference points are found.

4.3 Particle Filter Performance

The particle filter algorithm, whose source code is detailed in Appendix B, was implemented to smooth out the results from the hardware after performing triangulation as described in section 4.2, above. This particle filter was run in two different modes – first, using the location as computed by the hardware natively, and second, using a more precisely calibrated computation using the data collected in Table 4.

4.3.1. Default Location Computation

The results returned using just the default location computation along hallway 1 are shown in Figure 32, below.

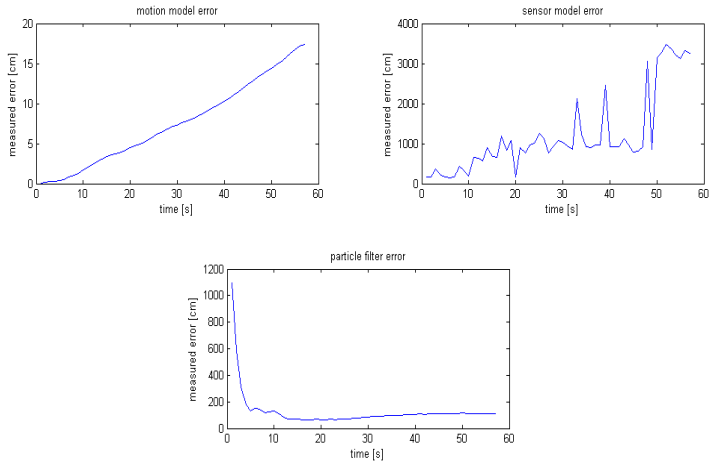

Figure 32 - Measured error, Hallway1

Here, the motion model, as expected, is shown to grow over time, although the drift is fairly small (the motion model here was simulated based on a presumed speed of 38 cm/s with a zero-mean Gaussian error with standard deviation 0.1 for both forward motion and rotation). The sensor model error actually grows over time, by a large amount. This can be explained by looking at Figure 27 – toward the end of the experiment, anchors 1 and 3 are disagreeing significantly on what the actual position is.

However, the particle filter implementation manages to adjust for these wild sensor readings with ease. Although it's very far off at the beginning (note that the robot starts at the extreme upper-left corner of the measurable area, which is the worst case for a particle filter implementation), the algorithm converges quickly.

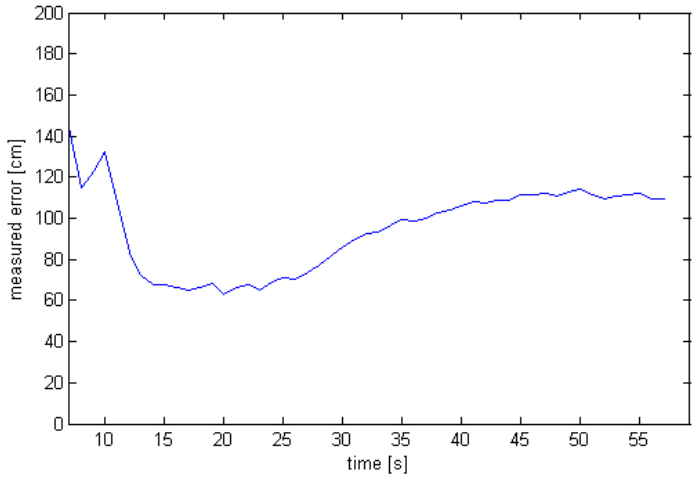

Figure 33 - Particle filter error

Figure 33, above, shows the "steady state" of the particle filter after 7 seconds. Here, the error drops down to almost ½ meter, but climbs again toward the end, due to the wild readings from the sensor.

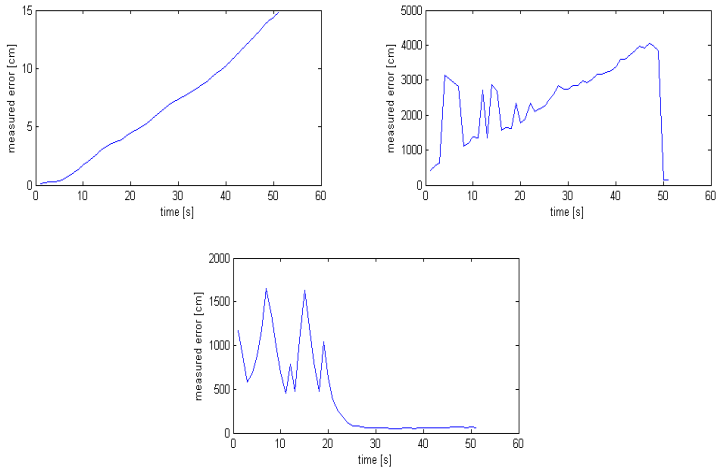

Figure 34 – Measured error, Hallway2

Figure 34, above, shows how the particle filter implementation performs along hallway 2. This time, it takes much longer to localize, because the data returned from the sensor is

much worse. However, after approximately 25 seconds, it has found its proper location, and the error has begun to shrink.

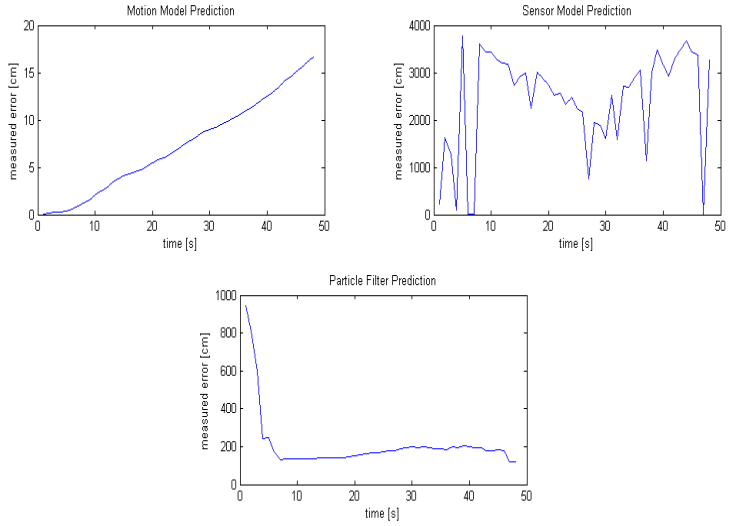

Figure 35 – Measured error, Hallway3

Figure 35, above, illustrates similar data for hallway "3" (as the robot moves back across hallway 2). The localization results here are more disappointing, but still usable (with steady state errors still falling under 2 meters). The high error here was caused by a large number of unusable readings (where only one beacon responded in the given time interval). This could be potentially be compensated for by gathering more readings, at the cost of top speed. Additionally, adding "jitter", as described above, will allow the particle filter to attempt to "re-localize" itself if it gets better data at a later point in time.

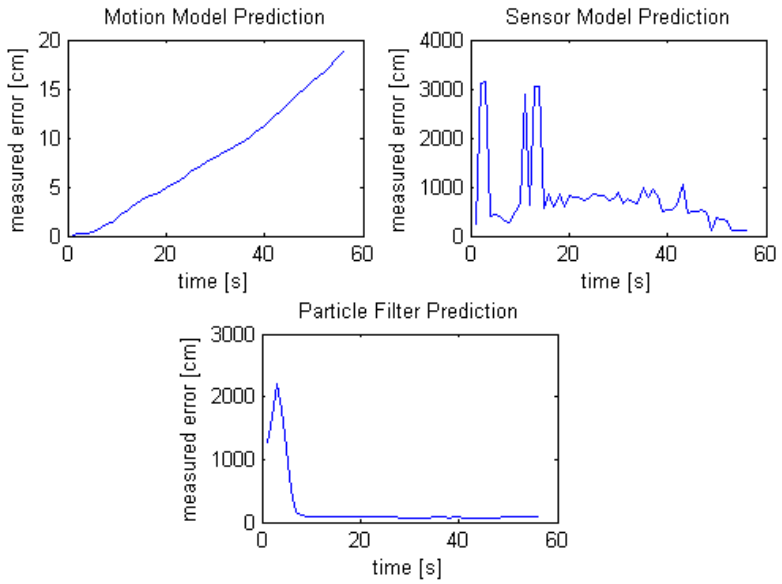

Figure 36 – Measured error, Hallway4

Finally, Figure 36, above, illustrates the results for hallway 4 (as the robot moves back across hallway 1). Here the sensors are much more well behaved and the best behavior of the four experiments is seen.

To verify that the sensor data does indeed provide benefit to the particle filter, Figure 37 below illustrates the same experiment (using Hallway 1's data) with a random sensor model (that is, each sense returns uniform random data for the x & y values). Here it is shown that the particle filter believes that it has localized itself with confidence, but its steady state is 10 meters away from the actual position of the robot.

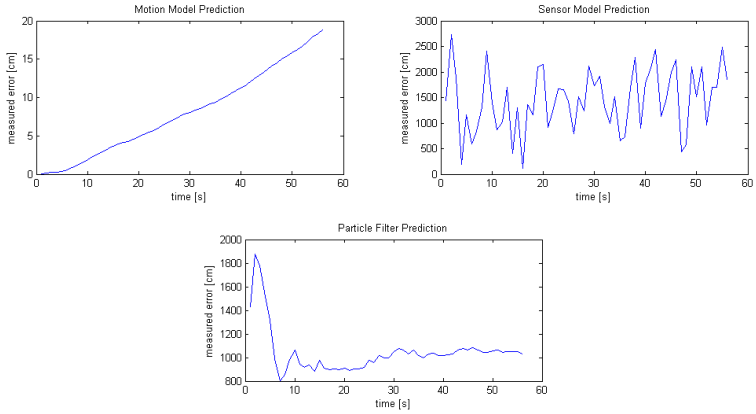

Figure 37 - Random sensor measured error

4.4 Measuring Attenuation to Reduce TOF Error

As is evident from the sensor model predictions above, and Table 4, the locations given by the time-of-flight computations are fairly inaccurate. Surprisingly, the particle filter algorithm performs well in the face of this inaccuracy; as long as the sensors are close fairly often, the algorithm is able to localize itself with a small margin of error. However, it would be desirable to reduce this margin of error further. This section explores the relationship between RSSI (or attenuation) and measured TOF error.

4.4.1. Measuring Attenuation and TOF error

Because radio signals propagate at or near the speed of light through air, tiny changes in the environment can have a huge impact on the distance as computed by the measured time of flight. At such speeds, even the angle of the antennae relative to one another can impact the time that it takes for a signal to be intercepted and processed. However, this difference is (roughly speaking) a function of the power perceived at the receiver.

The Nanoloc hardware doesn't report RSSI (received signal strength indicator), but instead reports attenuation (inverted RSSI). For the purposes of examining the relationship between RSSI and TOF error, it does just as well. The reported attenuation and time-of-flight error was examined for three different nanoloc access points from 0.5 meters to 12.5 meters

(the range of the hardware) in half meter increments by collecting 1000 samples at each distance.

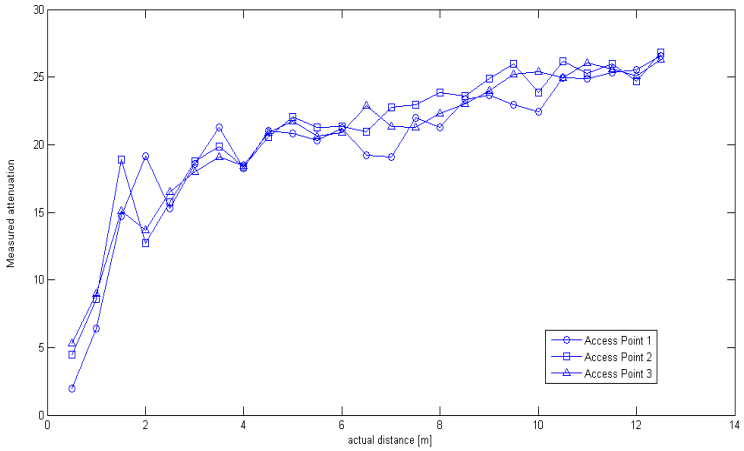

Figure 38 - Distance vs. attenuation

Figure 38, above, illustrates the measured relationship between attenuation as perceived by the roaming tag and distance between the tag and the access point for all three access points. This shows a sharp decline in RSSI (or a sharp increase in attenuation) at small variations in distance, followed by a fairly linear decline thereafter.

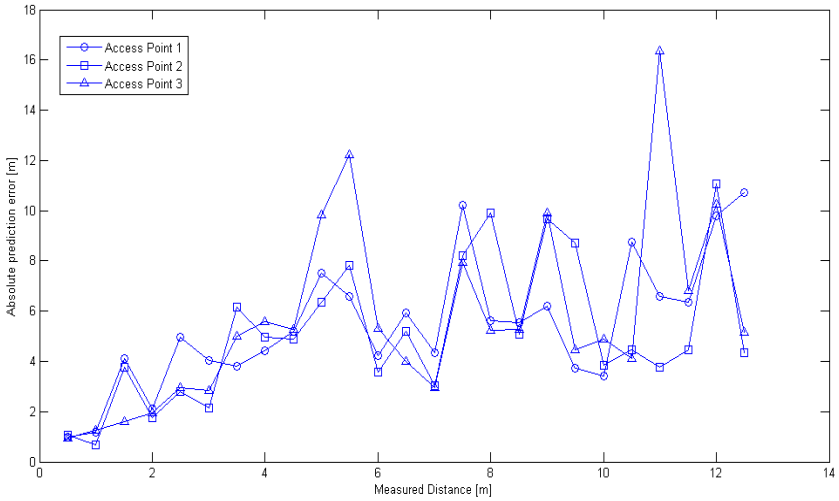

Figure 39 - Distance vs. measurement error

Figure 39, above, plots the distance vs. the absolute measurement error – that is, the difference between what the time-of-flight computations predict as the distance between the access point and the tag and the actual physical distance between the two. As can be seen clearly from the graph above, this relationship is not simply linear. Access point 3 reports an average distance of 20 m when the actual distance is 5.5 m, but 12 m when the actual distance is 6. These spikes are likely caused by Rayleigh fading effects as they appear more pronounced at specific distances and then disappear.

Also note that the TOF computation always overestimates distance. More useful than the absolute error is the magnitude of the error at each distance.

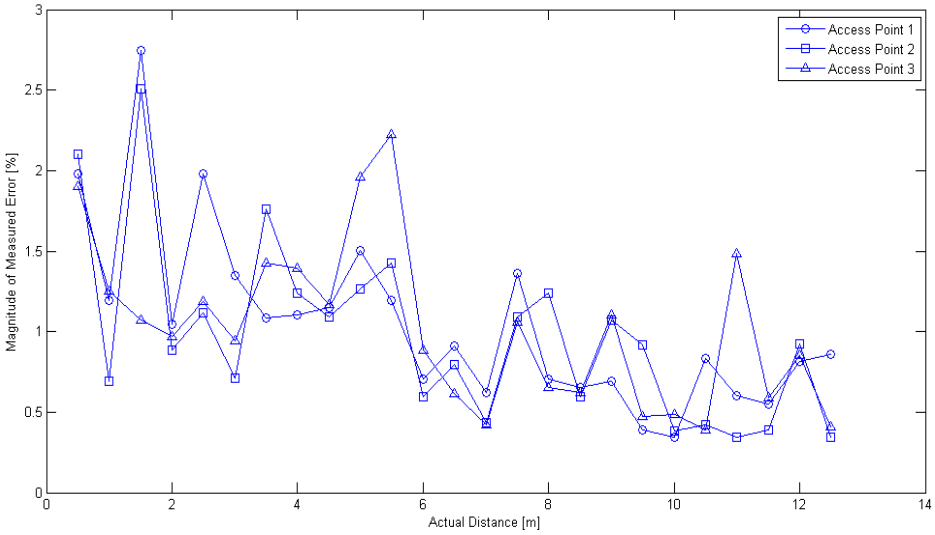

Figure 40 - Distance vs. magnitude of error

Here it is clear that the magnitude of the error doesn't vary as wildly as the absolute error, although the spikes appear in roughly the same places. Is it possible to compensate for this error by taking into account the attenuation observed by the roaming point?

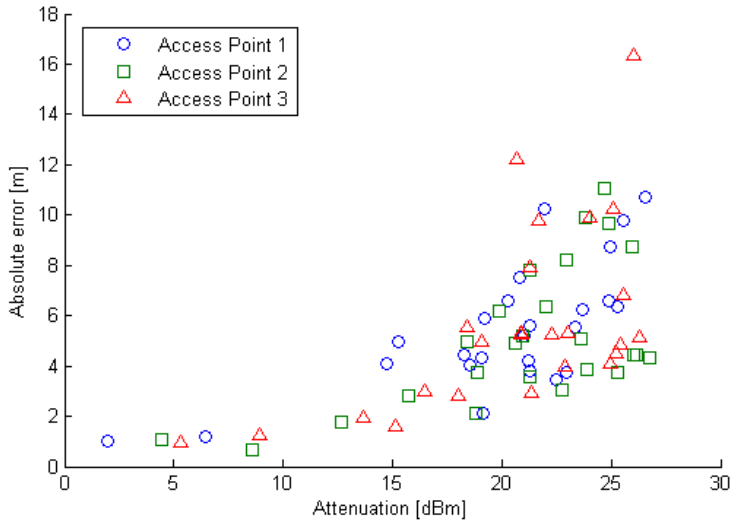

Figure 41 - Attenuation vs. absolute error

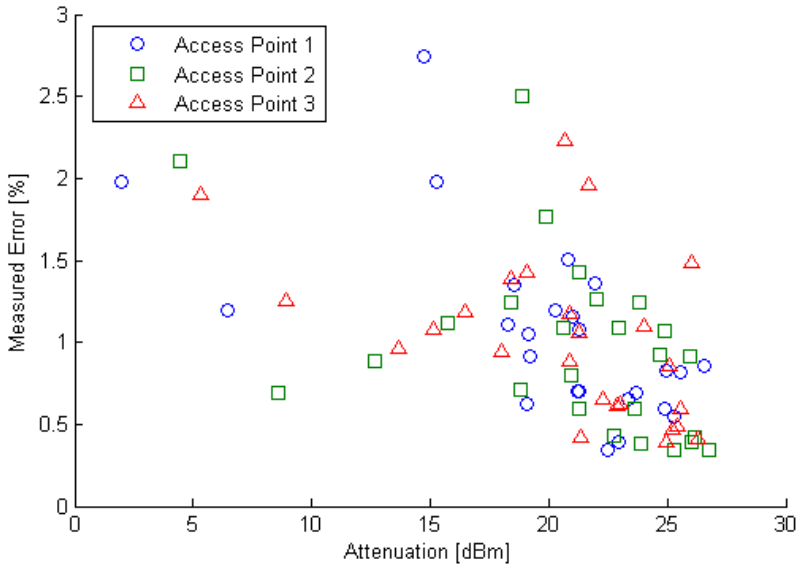

Figure 42 - Attenuation vs. magnitude of error

Figure 41 and Figure 42, above, illustrate the tenuous relationship between measured attenuation and observed TOF computation error (absolute and relative, respectively). The relationship between attenuation and absolute error is roughly quadratic and interpolates to

y = 0.013x^2 – 0.076x + 1.4

$y = 0.00017x^2 – 0.3x + 1.3$

$y = 0.0073x^2 + 0.15x – 0.67$

for access points 1, 2 and 3, respectively. The relationship between the relative error and the attenuation is linear (if there even is one):

$y = -0.061x + 2.3$

$y = -0.047x + 1.9$

$y = -0.046x + 2$

This interpolation can be used to account for the error in the distance as reported by the time of flight computation.

4.4.2. Distribution of error readings

Histogramming the measured error shows a roughly normal distribution of absolute and relative error at each distance. The mean and standard deviation observed at each distance are illustrated below (Data for access point 1 is shown – data for the other two access points is similar and won't be repeated in the interest of space).

Table 6 - Measurement Error Distribution

Distance [m]	Mean Error	Standard Deviation
0.5	0.9884	0.1760
1.0	1.1926	0.1787
1.5	4.1174	0.3154
2.0	2.0932	1.2920
2.5	4.9490	1.7247
3.0	4.0494	1.7444
3.5	3.7921	2.3677
4.0	4.4190	0.4169
4.5	5.1849	0.9201

Table 6 - *Continued*

5.0	7.5166	0.6678
5.5	6.5770	0.5870
6.0	4.2214	0.3276
6.5	5.9202	0.3799
7.0	4.3499	0.2082
7.5	10.2250	0.3947
8.0	5.6293	1.3887
8.5	5.5517	0.3690
9.0	6.2146	2.4458
9.5	3.7232	0.3182
10.0	3.4361	0.2026
10.5	8.7372	1.3285
11.0	6.5965	0.4944
11.5	6.3382	0.4552
12.0	9.7971	0.8230
12.5	10.7304	0.3628

Although the absolute observed error varies widely, the standard deviation of the normal fit does not – the error itself tends to be fairly well concentrated around a specific mean.

4.5 Applicability to the SLAM problem

In general, particle filtering is not applicable to the SLAM problem. The primary reason is the huge number of particles required to represent the potential state space. In most mapping implementations, tens of thousands of features are required for proper localization, making direct particle filtering infeasible, since billions of particles would need to be tracked to realistically model the potential states. Innovative approaches to solving this problem such as

the FastSlam algorithm (Thrun, Burgard and Fox 2005) have been proposed – however, in the case of localization using wireless signals, particle filtering can be directly applied to the SLAM problem.

The key here is in the representation of the map. Generally speaking, a "map" in a SLAM algorithm is a collection of features such as a wall, or a table, or a door as reported by the robot's sensors. However, if the anchor points are instead considered to be features, SLAM techniques can be used to discover the unknown locations of the anchors themselves while simultaneously localizing against them.

Another simplifying factor is that the correspondences between features and sensor readings is known with 100% confidence. Again, in the general SLAM problem, when features are anything that can be reported by a range sensor, it is entirely possible that a range sensor could be sensing the same wall that it sensed three iterations ago. As a result, a maximum likelihood estimator must be applied at each iteration to establish correspondences between sensed data and the features reported by the sensor. When using time-of-flight sensors to capture ranges to sensors, this is not a consideration – each range call includes a header that identifies the "feature" (that is, the wireless anchor point) with absolute surety.

Here, each particle is a nine-element state vector – three elements for the pose of the robot, and another two each for the x and y coordinates of the anchors. The particles themselves are initialized randomly and at each state, the ranges from the three anchors in each particle to the robot pose in the particle are compared to the ranges as reported by the sensors. The greater the difference, the lower the weight of the particle. Specifically, the weight given to each particle is:

$$\frac{1}{\left(r_1 - \sqrt{(t_{1x} - x)^2 + \left(t_{1y} - y\right)^2}\right) + \left(r_2 - \sqrt{(t_{2x} - x)^2 + \left(t_{2y} - y\right)^2}\right) + \left(r_3 - \sqrt{(t_{3x} - x)^2 + \left(t_{3y} - y\right)^2}\right)}$$

Where r_1, r_2, r_3 are the reported distances from the sensors and the particle is represented as:

$$\langle x, y, \theta, t_{1x}, t_{1y}, t_{2x}, t_{2y}, t_{3x}, t_{3y} \rangle$$

Unfortunately, this fails if the individual particles are initialized in a completely random manner. The problem here is that, with only ranges to features being available (as opposed to range and bearing), two or more particles that are identical, but "rotated", can't be effectively distinguished from one another, and therefore are assigned the same particle weight in spite of the fact that one is entirely wrong.

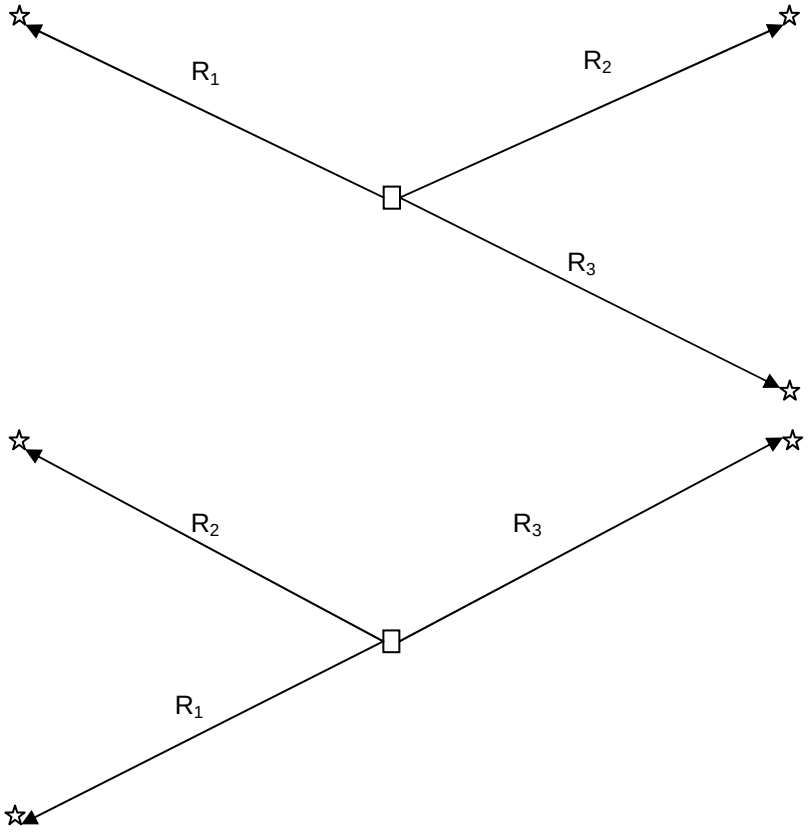

Figure 43 - Two indistinguishable particles

However, this can be made to work with one simplifying assumption – that the locations of the tags are known within some tolerance. In this case, even with somewhat significant

sensor noise, the implementation can discover the correct locations of the tags as well as its own position within a few iterations with a tolerable level of error.

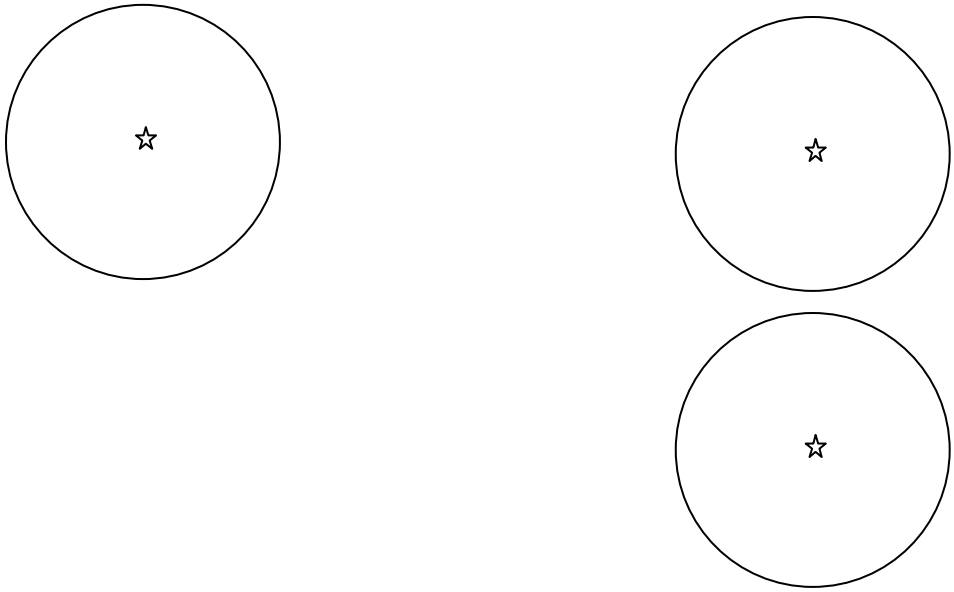

Figure 44 - Inexact knowledge of anchor points

Simulated results indicate that, with modest motion noise assumptions and reasonable sensor noise assumptions, the acceptable tolerance within which the tag positions must be known can be as low as 1/5 of the overall range. This might correspond to knowing, for example, what corner of the building the tag is located in, but not knowing exactly where it is in relation to that corner. Still, the number of particles required to achieve an acceptable error level is high – simulated results showed that 10,000 particles are required to reduce the residual error to under one meter.

4.5.1. Using an Anchor Point as an Origin

One of the problems with the formulation above, when anchor points are not known, is that the particles have six degrees of freedom – in effect, as illustrated in Figure 43, above, there are an infinite number of solutions. However, one of the reasons for this is that the origin

itself is assumed to be fixed. An alternative solution, which doesn't require any a priori knowledge of anchor point distribution, is to treat one of the anchor points as the origin. Although this requires a reformulation of the routines to account for an initially unknown origin point, simulations indicate that this does indeed work.

Another problem with the SLAM algorithm presented above is that the anchor points themselves stay fixed within each particle. Although this is reasonable (since the anchor points in physical space do not move), it constrains the particle filter convergence to its best initial guess. A better approach is to allow the anchor points within each particle to move, slightly, according to a narrowly distributed Gaussian random variable. This speeds convergence and reduces the number of particles that are needed to achieve convergence.

CHAPTER 5

CONCLUSIONS

Although the distances computed based on time-of-flight data by the Nanoloc hardware were wildly inaccurate, they were still effective in localizing after particle filtering techniques were applied. Essentially, for accurate localization, two pieces of information need to be available with at least rough precision – the speed of the node to be localized, and the locations of the sensors themselves. Also, for effective particle filtering, the algorithm needs to know the approximate area in which it may find itself – that is, it must establish upper and lower bounds on each of its state elements. Although in this work, these upper and lower bounds were presumed to be equal to the locations of the sensor nodes, this isn't an inherent limitation in the approach – the sensors themselves may actually be located anywhere.

The initial placement of the robot being localized within its environment (specifically, distance from center) did not significantly impact the speed at which the particle filter implementation converged. The particle filter algorithm is more tolerant of noise in the sensor responses than it is of noise in the motion model. Fortunately, in actual practice, it is reasonable to assume that sensor noise will be greater than motion noise. As was demonstrated with the time-of-flight Nanoloc hardware, sensor noise in actual practice can be very high but still useful.

Although three sensors are ideal for two-dimensional localization, it is feasible to localize with only two. By comparing the intersection points of the circles formed by the returned measurements and assuming that the one closest to the current predicted pose is the best measurement returned from the sensor, localization can be achieved relatively quickly even without the third anchor.

An attempt was made to improve on the range as computed by the nanoloc devices as shipped, but was unsuccessful. However, this was offset by the somewhat surprising result that the particle filter can account for invalid range measurements, as long as the measured ranges are consistently wrong and are somewhat close in at least some cases.

At least 1000 particles need to be tracked in order for the algorithm to perform acceptably. It is notable, as pointed out by (Thrun, Burgard and Fox 2005), that after stabilization has occurred, a very high percentage of the remaining particles are duplicates of one another. The memory requirements of the particle filter implementation could have been improved significantly by eliminating these duplicate particles – however, the upper bounds on memory would have remained the same, since the 1000 particles required for stabilization would still have been required for, on the average, about 10 iterations. Although this seems to preclude the possibility of loading the entire algorithm onto a microcontroller, for example, this might free up resources for path planning or map-building in a real robot implementation.

By treating the anchor points as “features” in a map, the SLAM problem can be applied to recovering both the locations of the anchor points and the mobile node itself from potentially noisy sensor data and motion controls. This problem is intractable in the general case, but with a modest simplifying assumption – that the positions of the anchors are known within some (fairly large) tolerance, SLAM can be effectively applied.

CHAPTER 6

FUTURE WORK

Time to stabilize is an important factor in the operation of the localization algorithm. Here a relatively crude, but still effective, measure was used to find the stabilization point – keeping track of the change in error at each iteration and assuming that three consecutive iterations without a significant variation can safely be considered stabilization. Advanced statistical analysis techniques such as linear regression could be applied to achieve a better sense of where stabilization truly occurs.

Localization is actually occurring in three dimensions – the ideal standard for three-dimensional triangulation is four anchor points. Since only three were available for this study, future work might compare the increased precision of an extra anchor, although as this work has demonstrated, localization can feasibly be performed with just two anchors. It may also be possible to localize with only a single anchor point by measuring the distance from each particle to the circle formed by the anchor measurement and weighting the particle based on that distance rather than weighting each particle based on the distance to the most likely point.

One of the drawbacks of the particle filtering algorithm as proposed is that the orientation of the robot to be localized must be known with some precision, as time-of-flight data offers no direct means of sensing orientation relative to a global coordinate system. However, it may be possible, once the locations of the sensors themselves have been fixed to some precision, to make use of what is known about the anchor points and the robot itself to try to infer orientation as well.

Some information about the anchor points needs to be provided, albeit with a large error tolerance, for proper operation, because at each iteration, very different hypotheses can't be distinguished from one another. However, it may be possible to relax the Markov

assumption and try taking into account past pose estimates when weighting particles to eliminate less likely state estimates.

Of course, the most interesting future work would be the integration of this particle filter and sensor model into a real robotics application with either an odometer or an accelerometer to act as the motion model. Assuming modest and reasonable motion model error, this work demonstrates that there is no reason to think that such an integrated application would fail to correctly localize and operate as expected.

APPENDIX A

NANOTRON LOCALIZATION APPLICATION

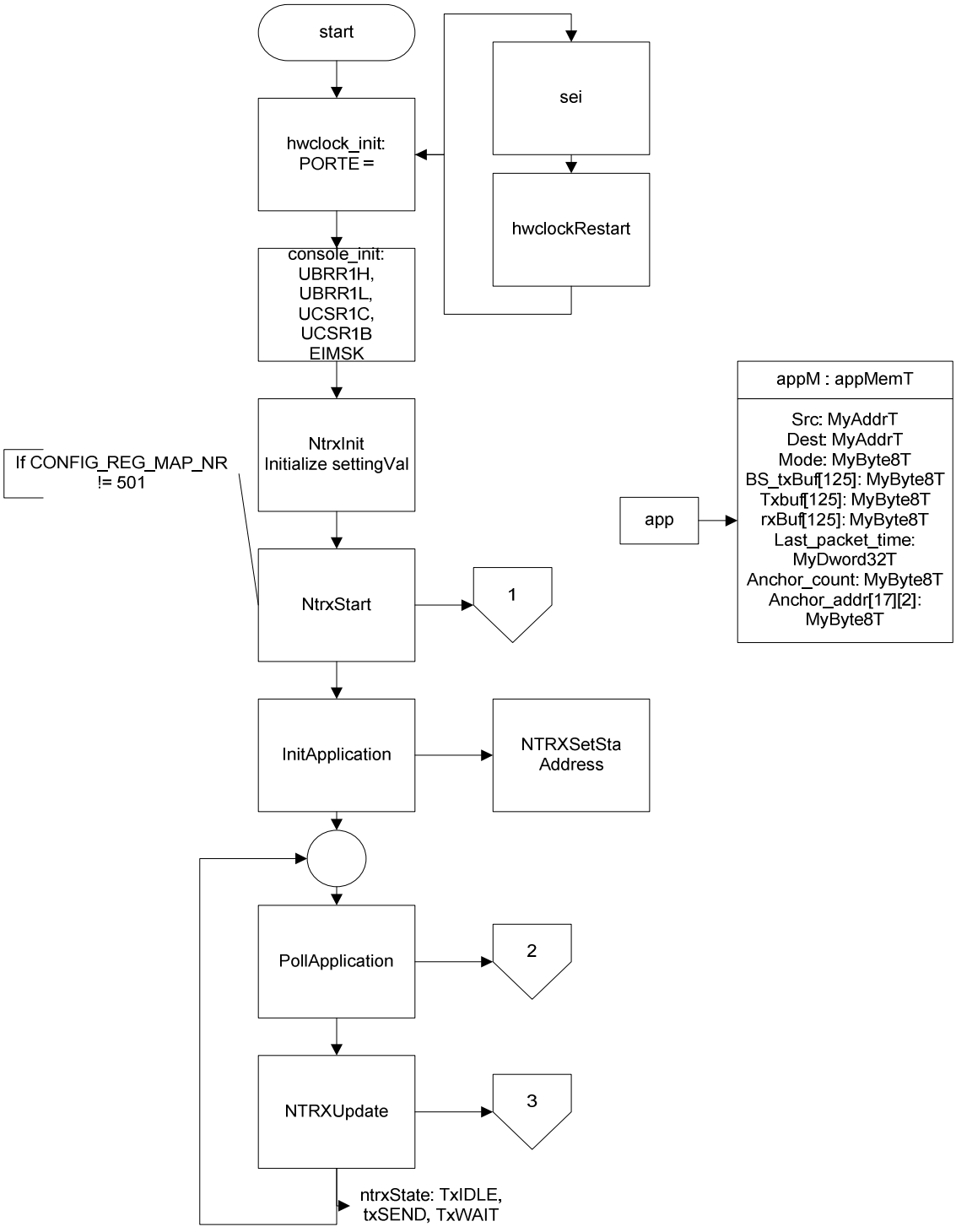

Figure A.1 - Main Flow

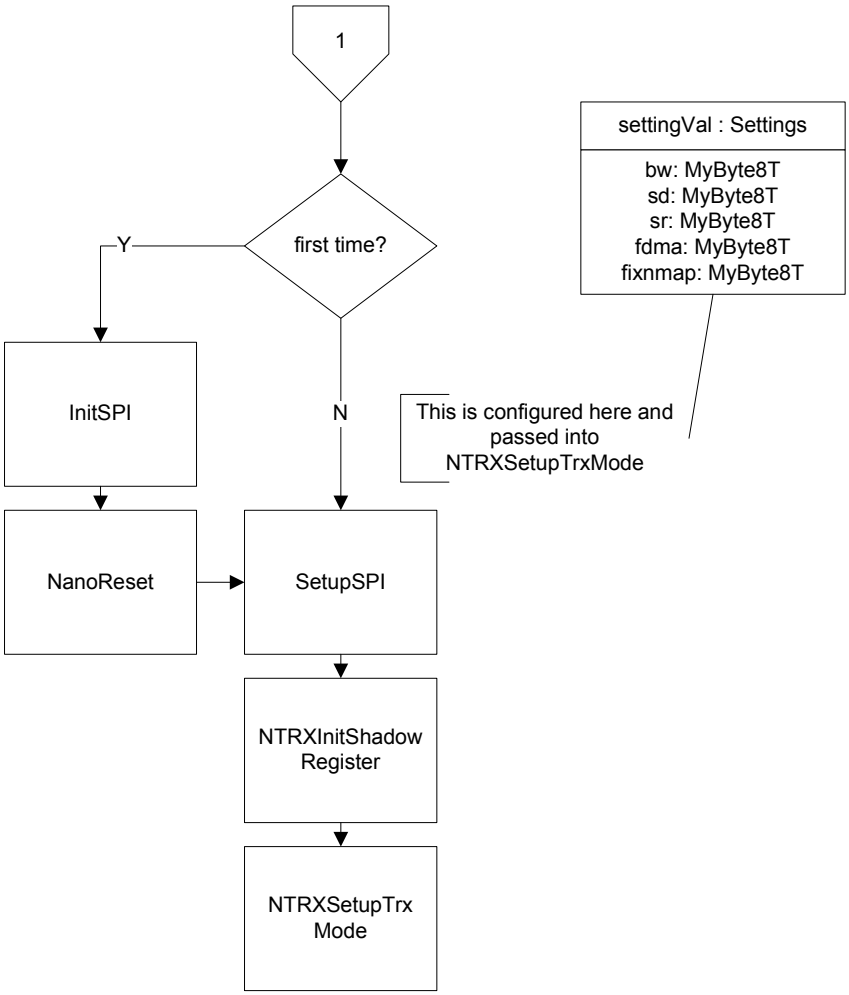

Figure A.2 - NTRXStart

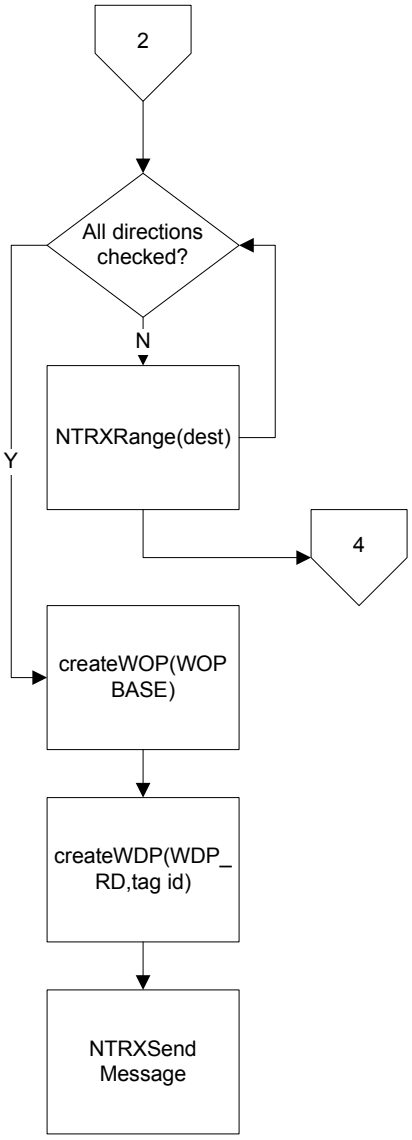

Figure A.3 - Poll Application

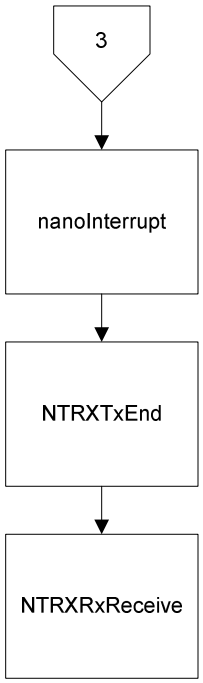

Figure A.4 - NTRXUpdate

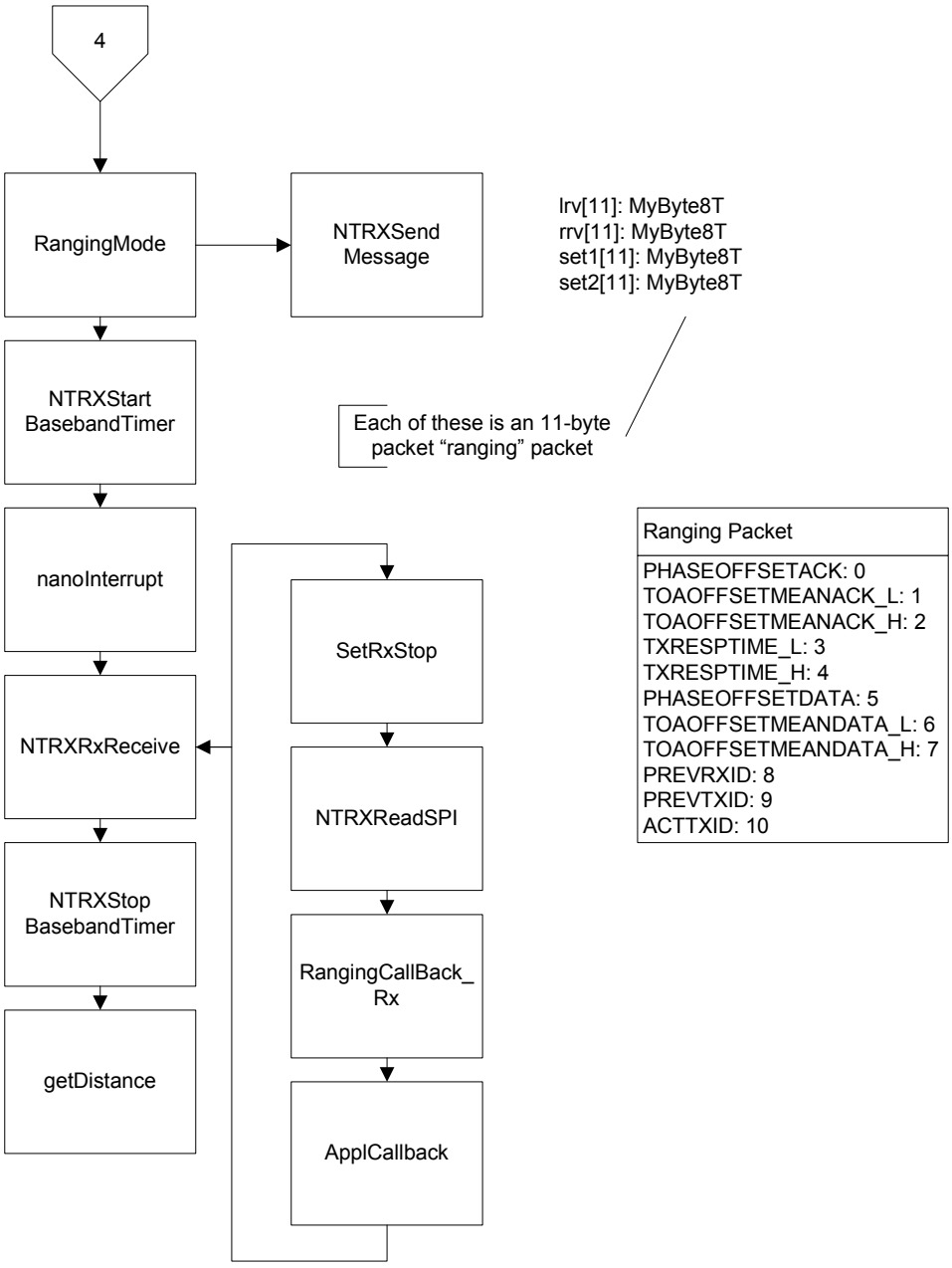

Figure A.5 - NTRXRange

APPENDIX B

PARTCLE FILTER IMPLEMENTATION IN MATLAB

```
% Particle filter to estimate location based on integrated control & sensor
% input.
% PARAM1 prev = a list of matrices of x,y & theta proposals, based on a
% previous invocation of this function.  This is a Nx3 matrix where N = the
% number of particles (default 1000)
% PARAM2 control = a 1x2 matrix of the most recent control applied (that is,
% the translational and rotational velocities of the robot)
% PARAM3 sensor_position = the position (or positions) that were sensed by the
% sensor most recently
% PARAM4 noise = a 2-element array of translational and rotational
% "predicted" noise, respectively.
% PARAM5 plot = a boolean flag to indicate whether the particles themselves
% should be shown (in the active figure).
function particles = particleFilter( prev, control, sensed_position, noise, plot )
    N = length(prev);

    prior = zeros(3,N);
    particles.posterior = zeros(3,N);
    particles.weights = zeros(1,N);
    prior_weights = zeros(1,N);
    for m = 1:N
        prior(:,m) = sampleMotionModel( control, prev(:,m), noise(1), noise(2) );
        prior_weights(m) = sensor_probability( sensed_position, prior(:,m) );
    end

    if plot
        for k = 1:N
            pointx = prior(1,k);
            pointy = prior(2,k);
            line([pointx pointx], [pointy pointy], 'Color', 'Red');
        end
    end



    % Normalize the weights so that they sum to one
    total = sum(prior_weights);
    prior_weights = prior_weights / total;

    % Resample

    % resampling algorithm here from particle filter tutorial
    Q = cumsum(prior_weights);
    %plot(Q);
    % XXX - this needs to be normalized to the range 0,1; the algorithm
    % specification doesn't say this, but the algorithm fails if it's not
```

Figure B.1 – particlefilter.m

```
    t = rand( 1, N + 1 );
    T = sort(t);
    T(N + 1) = 1.0;
    ind_i = 1;
    ind_j = 1;
    while ind_i <= N
      if T(ind_i) < Q(ind_j)
        particles.posterior(:,ind_i) = prior(:,ind_j);
        particles.weights(ind_i) = prior_weights(ind_j);
        ind_i = ind_i + 1;
      else
        ind_j = ind_j + 1;
      end
    end

    % re-normalize the weights so that weighted sum (later) works correctly
    % reset all the weights to uniform weights
    particles.weights(:) = 1 / length( particles.weights );

    if 0
       for k = 1:N
          pointx = particles.posterior(1,k);
          pointy = particles.posterior(2,k);
          line([pointx pointx], [pointy pointy], 'Color', 'Red');
       end
    end
end

% Return the probability of the given sensor readings given a specific
% pose.
function prob = sensor_probability( sens, proposal )
  sensor_size = size(sens);
  if sensor_size(1) == 1
     point = sens;
  else
     % Only got two sensor readings, so I have two candidate points.
     % Choose the one that's closest to my pose proposal
     if sensor_size(1) == 2
        d1 = sqrt( ( sens(1,1) - proposal(1) ) ^ 2 + ( sens(1,2) - proposal(2) ) ^ 2 );
        d2 = sqrt( ( sens(2,1) - proposal(1) ) ^ 2 + ( sens(2,2) - proposal(2) ) ^ 2 );
        if d1 < d2
          point = sens(1,:);
        else
          point = sens(2,:);
        end
     end
```

Figure B.2 – particlefilter.m

```
  end

  distance = sqrt( ( ( point(1) - proposal(1) ) ^ 2 ) + ( ( point(2) - proposal(2) ) ^ 2 ) );
  if distance == 0
    prob = 1.0;
  else
    prob = ( 1 / distance );
  end
end

% control is a matrix containing the translation and rotational velocity
% applied.
% prev is a matrix containing a possible x,y,theta position
% transVar is the estimated variance of the translation velocity (control(1))
% rotVar is the estimated variance of the rotationl velocity (control(2))
function sample = sampleMotionModel( control, prev, transVar, rotVar )
    v = control(1) + sampleNormalDistribution( transVar );
    w = control(2) + sampleNormalDistribution( rotVar );
    g = sampleNormalDistribution( rotVar );

    theta = prev(3);
    sample = prev + [ ( v * cosd( theta ) ) ( v * sind( theta ) ) w ]';
end

function sample = sampleNormalDistribution( stddev )
    sample = randn() * stddev;
end
```

Figure B.3 – particlefilter.m

```
% Initialize a "robot" structure that will be tracked by "updaterobot"
function robot = initrobot( N, maxx, maxy, startX, startY, startTheta, ...
      sensorSource, sensorFile, portNumber, tag1X, tag1Y, tag2X, tag2Y, ...
      tag3X, tag3Y )
   robot.width = 20;
   robot.height = 20;

   robot.pose = [ startX startY startTheta ];
   robot.motionModel = robot.pose;
   robot.predicted_pose = robot.pose;

   %drawrobot(robot.pose, robot.width, robot.height, 'Black');
   %drawrobot(robot.pose, robot.width, robot.height, 'Green');

   % Create a bunch of random particles
   robot.particles = rand( [ 3 N ] );
   % redistribute them according to the boundaries
   robot.particles(1,:) = robot.particles(1,:) * maxx;
   robot.particles(2,:) = robot.particles(2,:) * maxy;
   % Have to initialize the orientation to "reality", since the filter
   % has no way of sensing orientation.
   robot.particles(3,:) = startTheta;

   robot.iterationCount = 0;

   if sensorSource == 3
      robot.sensor = fopen( sensorFile );
   elseif sensorSource == 1
      robot.sensor = serial(strcat( 'COM', portNumber ), 'BaudRate', 38400);
      fopen(robot.sensor);
   end
   % Otherwise simulating, don't open any file

   robot.tags = [ tag1X tag2X tag3X; tag1Y tag2Y tag3Y ];
end
```

Figure B.4 – initrobot.m

```
% Apply a motion control to a "robot" and run an iteration of the particle
% filter algorithm to update its belief.
function robot = updaterobot( robot, control, translationalNoise, rotationalNoise, ...
      freq, jitterAmount, sensorSource, sensorScale, sensorNoise, ...
      predNoiseT, predNoiseR, showParticles, maxx, maxy )
   % move the point that represents the center
   translationalVelocity = control( 1 ) + ( translationalNoise * randn() );
   rotationalVelocity = control( 2 ) + ( rotationalNoise * randn() );

   % actually apply the control itself
   orientation = robot.pose( 3 );
   robot.pose = robot.pose + [ ( translationalVelocity * cosd( orientation ) ), ...
      ( translationalVelocity * sind( orientation ) ), rotationalVelocity ];

   motion_orientation = robot.motionModel(3);
   robot.motionModel = robot.motionModel + [ ( control(1) * cosd( motion_orientation ) ), ...
      ( control(1) * sind( motion_orientation ) ), control( 2 ) ];

   % Insert a percentage of random particles every n iterations if jitter
   % frequency is set higher than 0.
   if ( freq ~= 0 )
      if ( mod( robot.iterationCount, freq ) == 0 )
         pct = jitterAmount / 100;
         % TODO - figure out how to enforce this in the callback
         if pct > 100
            pct = 100;
         end
         for k = 1:( length( robot.particles ) * pct )
            robot.particles(1,k) = rand() * maxx;
            robot.particles(2,k) = rand() * maxy;
            robot.particles(3,k) = robot.predicted_pose(3);
         end
      end
   end

   %sensor_position = sense( handles.pose );
   if sensorSource ~= 2
      sensor_positions = sense( robot.tags, robot.sensor, ...
         sensorScale, sensorSource );
   else
      sensor_positions = [ ( robot.pose(1) + ( randn() * sensorNoise ) ) ...
         ( robot.pose(2) + ( randn() * sensorNoise ) ) robot.pose(3) ];
   end

   % In the case that I got nothing usable back from the sensors, just
   % return the most recent prediction.
```

Figure B.5 – updaterobot.m

```
    if isempty(sensor_positions)
        sensor_positions = robot.predicted_pose;
    end

    filterResponse = particleFilter( robot.particles, control, sensor_positions, ...
        [ predNoiseT predNoiseR ], showParticles );
    robot.particles = filterResponse.posterior;
    weights = filterResponse.weights;
    % estimate the position based on the weighted sum of the particles
    predicted_pose = zeros(1,3);
    for k = 1:size(robot.particles,2)
        predicted_pose(1) = predicted_pose(1) + ( robot.particles(1,k) * weights(k) );
        predicted_pose(2) = predicted_pose(2) + ( robot.particles(2,k) * weights(k) );
        predicted_pose(3) = predicted_pose(3) + ( robot.particles(3,k) * weights(k) );
    end
    % XXX - Since I can't "sense" orientation here, I have to show this as orientation = 0

    sensor_size = size(sensor_positions);
    if sensor_size(1) == 2
      % If I got back two positions from the sensor, out here I don't know which
      % one was more likely.  Find the one closest to the prediction to update
      % sensed pose.
      p1_dist = sqrt( ( ( sensor_positions(1,1) - predicted_pose(1) ) ^ 2 ) + ...
          ( ( sensor_positions(1,2) - predicted_pose(2) ) ^ 2 ) );
      p2_dist = sqrt( ( ( sensor_positions(2,1) - predicted_pose(1) ) ^ 2 ) + ...
          ( ( sensor_positions(2,2) - predicted_pose(2) ) ^ 2 ) );
      if p1_dist < p2_dist
        sensed_pose = [ sensor_positions(1,:) 0.0 ];
      else
        sensed_pose = [ sensor_positions(2,:) 0.0 ];
      end
    else
      sensed_pose = [ sensor_positions 0 ];
    end

    robot.sensed_pose = sensed_pose;

    robot.iterationCount = robot.iterationCount + 1;
    robot.predicted_pose = predicted_pose;
end
```

Figure B.6 – updaterobot.m

```
% Given two 6x2 arrays of points, figure out which point appears in both
function match = find_match(points)
  points_size = size(points);

  if ( points_size(1) == 6 )
     dist(1)  = sqrt( ( ( points(1,1) + points(3,1) ) ^ 2 ) + ( ( points(1,2) + points(3,2) ) ^ 2 ) );
     dist(2)  = sqrt( ( ( points(1,1) + points(4,1) ) ^ 2 ) + ( ( points(1,2) + points(4,2) ) ^ 2 ) );
     dist(3)  = sqrt( ( ( points(1,1) + points(5,1) ) ^ 2 ) + ( ( points(1,2) + points(5,2) ) ^ 2 ) );
     dist(4)  = sqrt( ( ( points(1,1) + points(6,1) ) ^ 2 ) + ( ( points(1,2) + points(6,2) ) ^ 2 ) );
     dist(5)  = sqrt( ( ( points(2,1) + points(3,1) ) ^ 2 ) + ( ( points(2,2) + points(3,2) ) ^ 2 ) );
     dist(6)  = sqrt( ( ( points(2,1) + points(4,1) ) ^ 2 ) + ( ( points(2,2) + points(4,2) ) ^ 2 ) );
     dist(7)  = sqrt( ( ( points(2,1) + points(5,1) ) ^ 2 ) + ( ( points(2,2) + points(5,2) ) ^ 2 ) );
     dist(8)  = sqrt( ( ( points(2,1) + points(6,1) ) ^ 2 ) + ( ( points(2,2) + points(6,2) ) ^ 2 ) );
     dist(9)  = sqrt( ( ( points(3,1) + points(5,1) ) ^ 2 ) + ( ( points(3,2) + points(5,2) ) ^ 2 ) );
     dist(10) = sqrt( ( ( points(3,1) + points(6,1) ) ^ 2 ) + ( ( points(3,2) + points(6,2) ) ^ 2 ) );
     dist(11) = sqrt( ( ( points(4,1) + points(5,1) ) ^ 2 ) + ( ( points(4,2) + points(5,2) ) ^ 2 ) );
     dist(12) = sqrt( ( ( points(4,1) + points(6,1) ) ^ 2 ) + ( ( points(4,2) + points(6,2) ) ^ 2 ) );

     smallest = find(dist == min(dist));

     if smallest <= 4
       match = points(1,:);
     elseif smallest <= 8
       match = points(2,:);
     elseif smallest <= 10
       match = points(3,:);
     else
       match = points(4,:);
     end
  end

  if ( points_size(1) == 4 )
     dist(1)  = sqrt( ( ( points(1,1) + points(3,1) ) ^ 2 ) + ( ( points(1,2) + points(3,2) ) ^ 2 ) );
     dist(2)  = sqrt( ( ( points(1,1) + points(4,1) ) ^ 2 ) + ( ( points(1,2) + points(4,2) ) ^ 2 ) );
     dist(3)  = sqrt( ( ( points(2,1) + points(3,1) ) ^ 2 ) + ( ( points(2,2) + points(3,2) ) ^ 2 ) );
     dist(4)  = sqrt( ( ( points(2,1) + points(4,1) ) ^ 2 ) + ( ( points(2,2) + points(4,2) ) ^ 2 ) );

     smallest = find(dist == min(dist));

     if smallest <= 2
       match = points(1,:);
     else
       match = points(2,:);
     end
  end
end
```

Figure B.7 – find_match.m

```
function i = circle_intersect( p1, p2, r1, r2 )
  dx = p2(1) - p1(1);
  dy = p2(2) - p1(2);
  d = sqrt( ( p1(1) - p2(1) ) ^ 2 + ( p1(2) - p2(2) ) ^ 2 );

  % Check to see if they intersect at all (if not, I'll get back imaginary
  % numbers, which doesn't help much).
  if d > r1 + r2
     i = [];
     return;
  end

  if d < abs( r1 - r2 )
     i = [];
     return;
  end

  a = ( ( r1 ^ 2 ) - ( r2 ^ 2 ) + ( d ^ 2 ) ) / ( 2.0 * d );
  x2 = p1(1) + ( dx * a / d );
  y2 = p1(2) + ( dy * a / d );

  h = sqrt( ( r1 ^ 2 ) - ( a ^ 2 ) );
  rx = -dy * ( h / d );
  ry = dx * ( h / d );
  i = [ [ x2 + rx y2 + ry ]; [ x2 - rx y2 - ry ] ];
end
```

Figure B.8 – circle_intersect.m

REFERENCES

Atheros Communications. "Power Consumption and Engergy Efficiency Comparisons of WLAN Products." 2003. http://www.atheros.com/pt/whitepapers/atheros_power_whitepaper.pdf website.

Bahl, Parmavir, and Vankata Padmabhan. "RADAR: An in-building RD-based User Location and Tracking System." *Infocom.* 2000.

Baron, Michael. *Probability and Statistics for Computer Scientists.* Chapman and Hall, 2007.

Borke, Paul. *Intersection of two circles.* 1997. http://local.wasp.uwa.edu.au/~pbourke/geometry/2circle/.

Cavalieri, Salvatore. "WLAN-based Outdoor Localization Using Pattern Matching Algorithm." *International Journal of Wireless Information Networks*, 2007.

Dissanayake, Gamini, Paul Newman, Steven Clark, and Hugh Durrant-Whyte. "A solution to the Simultaneous Localization and Map Building (SLAM) Problem." *IEEE Transaction on Robotics and Automation, Vol. 17, No. 3*, 2001.

Folkesson, Jensfelt, and Christensen. "The M-Space Feature Representation for SLAM." *IEEE Transactions on Robotics*, 2007.

Haeberlen, Andreas, Algis Rudys, Eliot Flanery, and Dan Wallack. "Practical Robust Localization over Large-scale 802.11 Wireless Networks." *Mobicom.* 2004.

Jain, Raj. *The Art of Computer Systems Performance Analysis.* Wiley, 1991.

Kantor, George, and Sanjiv Signh. "Preliminary results in range-only localization and mapping." *IEEE international conference on robotics and automation.* 2002.

Ladd, Andrew, Kostas Bekris, Algis Rudys, Lydia Kavraki, and Dan Wallach. "Robotics-based Location Sensing Using Wireless Ethernet." *Wireless Networks 11*, 2005: 189-204.

Liu, Hui, Houshang Darabi, Pat Banerjee, and Jing Liu. "Survey of Wireless Indoor Positioning Techniques and Systems ." *IEEE Transactions on sytems, man and cybernetics*, 2002.

Nanotron. *nanoLOC TRX Transceiver (NA5TR1).*

Nanotron Technologies. "Real Time Location Systems." 2006.

Noh, Angela, Woong Jang Lee, and Jin Young Ye. "Comparison of the Mechanisms of the Zigbee's Indoor Localization Algorithm." *Ninth ACS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing.* 2008.

Rabiner, Lawrence. "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition." *Proceedings of the IEEE Vol. 77 No. 2.* 1989. 257-286.

Rekleitis, Ioannis. "A Particle Filter Tutorial for Mobile Robot Localization." *International Conference on Robotics and Automation.* 2003.

Riisgard, Soren, and Morten Rufus Blas. *MIT School of Aeronautics and Astronomics.* 2005. http://ocw.mit.edu/NR/rdonlyres/Aeronautics-and-Astronautics/16-412JSpring-2005/9D8DB59F-24EC-4B75-BA7A-F0916BAB2440/0/1aslam_blas_repo.pdf.

Thrun, Sebastian, Burgard, and Fox. *Probabilistic Robotics.* 2005.

Welch, Greg, and Gary Bishop. "Kalman Filter." *ACM SIGGRAPH.* 2001.

Welch, Lloyd. "Hidden Markov Models and the Baum-Welch Algorithm." *IEEE Information Theory Society Newsletter*, 2003.

Zaruba, Gergely, Manfred Huber, Fahrad Kamangar, and Imrich Chlamtac. "Indoor Location Tracking using RSSI readings from a single wifi access point." *Wireless networks*, 2007.

Zunino, Guido. "Simultaneous Localization and Mapping for Navigation in Realistic Environments." *Royal Institute of Technology*, 2002.

BIOGRAPHICAL INFORMATION

Joshua Davies received a Bachelor of Science in Computer Science degree from Valdosta State University in 1994. He is currently enrolled in the Masters of Computer Science program at the University of Texas at Arlington and works as a principal architect for Travelocity.com.