MOBILE ROBOT MOTION-PLANNING USING WIRELESS SIGNALS FOR LOCALIZATION

BY

AARON TRENT BECKER

B.S., Iowa State University, 2005

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2008

Urbana, Illinois

Adviser:

Professor Mark Spong

ABSTRACT

Many applications for autonomous agents require the agent to have accurate position data. *Signal-strength based localization* attempts to determine an agent's location in $\mathbb{R}^n$ from a scalar sensor reading z. This is a nontrivial problem because the mapping from $\mathbb{R}^n$ to z is noninvertible.

Indoor Localization attempts to solve this problem in an indoor environment. This adds challenges. Among these are the effects of signal interference; signal dropout due to walls, doors, and humans; and the expense of data collection.

This project implements a localization system on an autonomous system, the *SegMonster*. The SegMonster is a human-sized robot that rides a commercial Segway personal transporter. Local data from wheel-mounted encoders are combined with signal-strength based localization that interprets wireless signal strength using *Gaussian processes* to calculate a global position estimate.

To Laney.

ACKNOWLEDGMENTS

This project would not have been possible without the support of many people. Many thanks to my adviser, Professor Mark Spong, for his guidance and support. Daniel Block, head of the Control Systems Labs, provided expert advice, lab space, and friendship. Thank you to David Johnson and Jan Vervoorst, the designers of the SegMonster, for their robust design; Gohkan Atinc for helping check and explain nonlinear dynamics; and Abdullah Acke, who spent long hours learning localization using GP with the author. Finally, thanks to my wife and parents who endured this long process with me, always offering support and love.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AOA	angle of arrival
AP	access point (bridge between the wired and wireless networks)
CDF	cumulative distribution function
DSP	digital signal processor
GP	Gaussian process
GPS	Global Positioning System
HT	Segway Human Transporter
IR	infrared
K-NNSS	k nearest neighbor(s) in signal space
LAN	local area network
NNSS	nearest neighbor(s) in signal space
PDF	probability distribution function
PMF	probability mass function
RF	radio frequency
RMP	Segway Robotic Mobility Platform
RSS	received signal strength
TDOA	time difference of arrival
WLAN	wireless local area network

LIST OF SYMBOLS

$\mathbf{x}$	system state (vector)
$\hat{x}$	state estimate (vector)
η	normalizer
$\circledcirc$	observability matrix
$\mathbf{P}$	system covariance matrix
$\mathbf{p}_t$	array of particles
θ	angular position
ω	angular velocity
v	speed

CHAPTER 1

INTRODUCTION

> A brave knight returned to his castle after an absence of six months. After being presented with great fanfare, the knight reported his feats, "Oh King, I've spent the last six months pillaging, burning, and looting the villages of your enemies in the east." The king answered, "Wonderful—but we don't have any enemies in the east." The knight hesitated for a moment, then replied, "We do *now*."
>
> —Anon

Location matters. Many actions that are acceptable in one context are inappropriate in others. Most robots require an accurate position value to accomplish their objectives. Industrial robots achieve accurate end-effector positioning through a rigid structure with sensors (usually optical encoders) at every joint, and calculate their position with respect to the fixed world coordinates [1]. Mobile robots are, by definition, not connected to a fixed world coordinate and so must cope with position uncertainty. Mobile robotics can be divided into behavior-based robotics and state-based robotics [2]. Most behavior-based systems are reactive and work without a parameterized version of their world. Examples include very basic cleaning robots: [*If* floor is dirty: stay and vacuum, *else*: move] and light chasing photovores: [*Forever*: Move toward light]. Such robots respond directly to the inputs their sensors provide. State-based mobile robots attempt to describe their position and their world

through state variables. They try to use sensor inputs to refine their state variables. Because they are untethered, state-based mobile robots must either keep track of changes to their position, or extract measurements from the world to calculate their position. This process is called localization.

This thesis presents the challenge of localization with respect to the SegMonster, a human-sized autonomous robot depicted in Fig. 1.1.

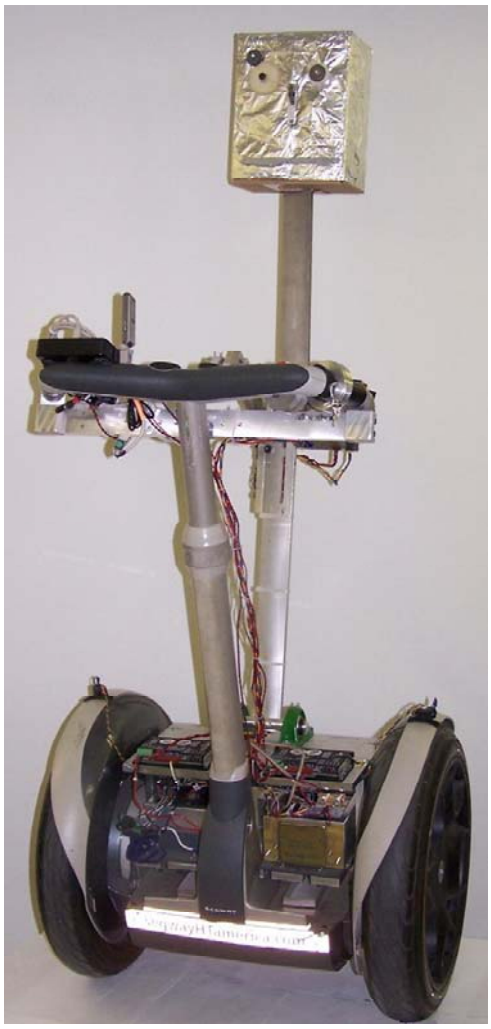


Figure 1.1: The SegMonster, a Segway-riding mobile robot. The SegMonster was made autonomous for this thesis.

After an overview of localization, Chapter 2 presents several basic solutions to the problem. *Dead reckoning*, when internal state measurements are integrated to keep track of the state, is analyzed first and found to be lacking for tracking location, and totally unsuited for state estimation from an unknown starting configuration. Process noise causes any state information to drift, so the robot loses state information as it acts in the environment and the environment acts on it.

Simple observers are often used for state estimation, and are analyzed next. Observers can reject model inaccuracies and correct faulty state information, but only for a narrow class of systems. Most robots fall outside this class. Observers provide a first step towards robust state estimation, but fail for a number of reasons. Many of the problems with these first solutions stem from the fact that they only maintain a single state estimate. It would be better to maintain a probability distribution. Chapter 3 begins to add this extra information to the model. The *Kalman filter* is a recursive procedure to estimate state. Its chief difference from an observer is that the Kalman filter maintains both a state estimate and the estimate error covariance matrix. For linear systems with linear noise, the Kalman filter provides an optimal state estimate.

Kalman filters have several drawbacks. Though they maintain a state estimate and covariance explicitly, they are restricted to linear or linearized systems. *Particle filters*, in contrast, represent the state estimate and covariance implicitly, by using a large number of state estimates. The motion model can propagate this data set of state estimates, which can be linear or nonlinear. The state mean and covariance can be extracted from this data set.

Localization requires global sensors that can be processed to determine position in the world. One source for this global information is wireless signal strength. Wireless

networks are becoming ubiquitous in urban environments. Much research has been and is being done on estimating location from wireless signal strength. This thesis presents a solution that uses *Gaussian processes* (GPs) to model the expected signal strength over a map. GPs are non-parametric models that estimate Gaussian distributions over *functions* based on training data [3]. Chapter 4 discusses strengths of GPs and presents a method to construct a map of the environment.

The SegMonster as a robotic platform is unique, and presents an interesting case study for control techniques. Chapter 5 describes the SegMonster and its dynamics. Some local controllers are presented.

Chapter 6 describes the experimental results from a localization solution implemented on the SegMonster. Finally, the concluding chapter gives suggestions for further work.

CHAPTER 2

LOCALIZATION

> One of Igor's former masters had *made* a tick-tock man, all levers and gearwheels and cranks and clockwork. Instead of a brain, it had a long tape punched with holes. Instead of a heart, it had a big spring. Provided everything in the kitchen was very carefully positioned, the thing could sweep the floor and make a passable cup of tea. If it *wasn't* carefully positioned, or if the ticking, clicking thing hit an unexpected bump, then it'd strip the plaster off the walls and make a furious cup of cat.
>
> —*Thief of Time*, Terry Pratchet

2.1 Classification

There are two main problems in localization. The first is maintaining a position estimate from a known starting point.

This task is called *position tracking*, and is useful when the robot starts from a hardhome configuration or when an outside agent supplies the starting conditions.

$$\mathbf{x}_0 = \begin{bmatrix} 3 & 2 & \pi \end{bmatrix} \Rightarrow \mathbf{x}_t = \begin{bmatrix} ? & ? & ? \end{bmatrix}. \tag{2.1}$$

Localization from an unknown starting position is called *global localization*. Position tracking is a subset of global localization and is less complex.

$$\mathbf{x}_0 = \begin{bmatrix} ? & ? & ? \end{bmatrix} \Rightarrow \mathbf{x}_t = \begin{bmatrix} ? & ? & ? \end{bmatrix}. \tag{2.2}$$

A third and more advanced localization problem is called the *kidnapped robot problem*. In this problem, an outside agent is allowed to freeze the robot's internal state and move the robot to another location at any time without informing the robot. In order to recover, the robot must realize that its current state estimate is incorrect, and start a new global localization.

2.2 Representing a Belief about Location

Probability densities, which map a scalar probability to every point in the configuration space of the robot, offer a way to represent the certainty of a position estimate.

$$p\left(\mathbf{x_t} = \mathbf{x}\right) \forall \mathbf{x} \in \mathbf{C}. \tag{2.3}$$

If the current position is known to be $\mathbf{x_0}$, the probability distribution can be represented as

$$\mathbf{x} = \begin{cases} 1 & \textit{if } \mathbf{x} = \mathbf{x_0} \\ 0 & \textit{otherwise}. \end{cases}$$

Figure 2.1 shows that as uncertainty grows, the probability distribution spreads out. For mobile robotic applications, a useful system state is the robot's x and y position and heading. This information is called the robot's *pose* (position and heading). The control inputs to the SegMonster are a desired speed and desired angular velocity

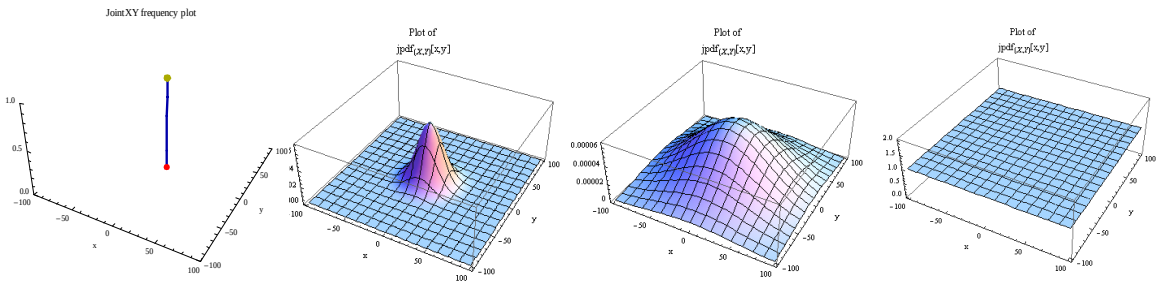


Figure 2.1: Using probability distributions to represent location certainty. A point represents total certainty about position. As certainty decreases, the point spreads into a probability distribution. If all position estimates are equally likely, the probability distribution is uniform. The progression from left to right shows decreasing certainty.

$$\mathbf{x}_t = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} \mathbf{u}_t = \begin{bmatrix} v \\ \omega \end{bmatrix}. \tag{2.4}$$

The robot pose will be represented by the following icon: , a circle centered at (x, y), with a line extending from (x, y) in the direction θ, referenced with 0 radians pointing to the right. Because the input contains an angular velocity, the state update equations are nonlinear $\mathbf{x}_t = f(\mathbf{x}_{t-1}, \mathbf{u}_t)$. The state update equations for the SegMonster (and other unicycle-type robots) are

$$\mathbf{x}_t = \begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix} = \begin{bmatrix} x_{t-1} + v_t \cos(\theta_{t-1})\Delta t \\ y_{t-1} + v_t \sin(\theta_{t-1})\Delta t \\ \theta_{t-1} + \omega_t \Delta t \end{bmatrix}. \tag{2.5}$$

Figure 2.2 depicts the state update in block diagram form.

On the SegMonster, the speed and angular velocity are computed by differentiating the output of optical encoders mounted on each wheel. If the model were precise

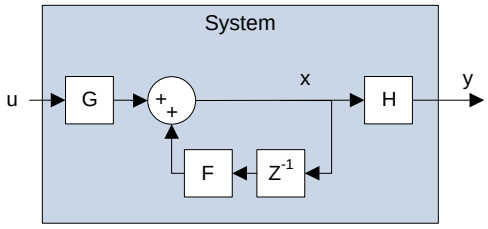


Figure 2.2: Block diagram for a digital system. Time is discrete, and the block z^{-1} is a unit delay. Here, $\mathbf{F}$ is the linearization of f at the given state. The linearization of a noiseless system can have arbitrarily small drift by choosing a small enough step time.

(perfectly calibrated encoders; equal size, perfectly round tires; zero wheel slippage; infinite resolution encoders), then tracking would be a simple procedure. The only errors would stem from the linearization, and could be made arbitrarily small by shrinking the step size.

2.3 Dead Reckoning

Integrating control inputs (such as the desired speed and angular velocity) to update the pose estimate is called *dead reckoning*. Using local sensors such as onboard encoders, rate gyros, or accelerometers to estimate speed and angular velocity also falls under this heading. Dead reckoning works well in simulation and in practice for short distances, but it is an open loop system and suffers from drift. Often in practice, as in Fig 2.3, heading and distance errors increase until the resulting position estimate is useless.

This drift has a number of causes that can be lumped into either model errors or sensor errors. Both can be minimized. The model can be improved by changing the robot to be more precise (exchange balloon tires for precision milled steel wheels), or by measuring the robot constants more accurately. Some of the orientation drift in Fig. 2.3 is due to low tire pressure in the right wheel. The accuracy of the initial

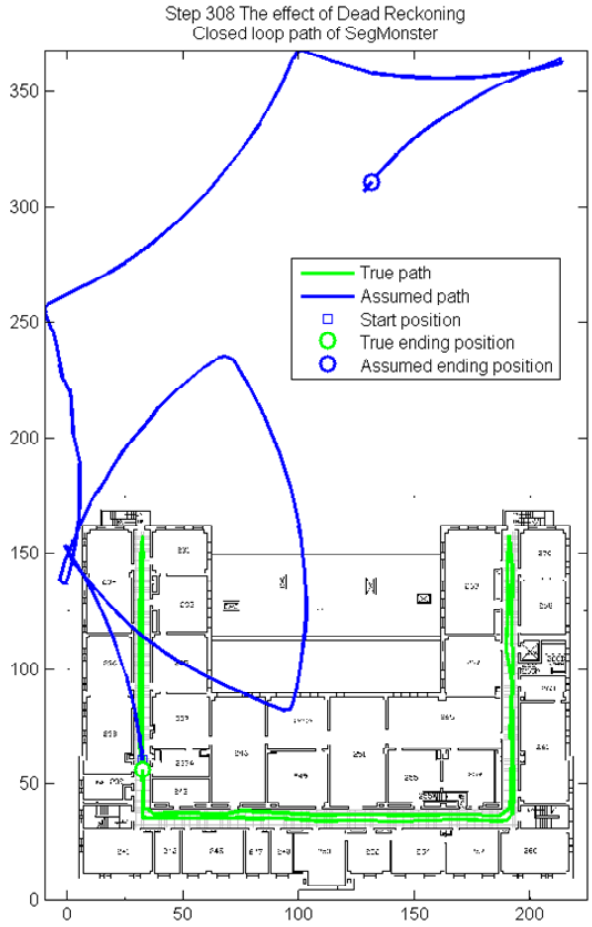


Figure 2.3: Test results showing dead reckoning error. The SegMonster navigated a closed loop path in the hallways of Everitt lab (70×50 m). By the end of the first hallway (20 m) the dead reckoned orientation estimate had accumulated a 30° error. The position error increased to 90 m over the course of the exercise.

position estimate is also important. While position errors may be negligible, the effect of a small initial deviation from the true orientation integrates over time.

To minimize sensor error, the sensors can be replaced with more accurate counterparts. The SegMonster's encoders, with 1250 ticks per revolution, are already near the top end of the range for commercial optical encoders. They are accurate for speed measurement, but perform poorly for measuring angular position change. Precision MEMS rate-gyros perform much better, with measured drift in our lab that is less

than 10 per min. Some commercial systems have drifts in the tenths of degrees per hour in orientation and of less than 0.6 nautical miles per hour in position [4].

Process noise is a blanket term that encompasses all these drifts. The new model for the system is

$$\mathbf{x}_t = \begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix} = \begin{bmatrix} x_{t-1} + v_t \cos(\theta_{t-1})\Delta t \\ y_{t-1} + v_t \sin(\theta_{t-1})\Delta t \\ \theta_{t-1} + \omega_t \Delta t \end{bmatrix} + \begin{bmatrix} processNoise \cdot v_t \cos(\theta_{t-1})\Delta t \\ processNoise \cdot v_t \sin(\theta_{t-1})\Delta t \\ processNoise \cdot \omega_t \Delta t \end{bmatrix}. \quad (2.6)$$

The equation for the first step $\mathbf{x}_{\Delta t}$ must also include the inaccuracy of the initial state estimate. Inaccuracies in θ_0 cause more problems than inaccuracy in the initial x or y coordinates.

$$\begin{aligned} \mathbf{x}_{\Delta t} = \begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix} &= \begin{bmatrix} x_0 + inaccuracy_x + v_t \cos(\theta_{t-1})\Delta t \\ y_0 + inaccuracy_y + v_t \sin(\theta_{t-1})\Delta t \\ \theta_0 = inaccuracy_\theta + \omega_t \Delta t \end{bmatrix} \\ &+ \begin{bmatrix} processNoise \cdot v_t \cos(\theta_{t-1})\Delta t \\ processNoise \cdot v_t \sin(\theta_{t-1})\Delta t \\ processNoise \cdot \omega_t \Delta t \end{bmatrix}. \end{aligned} \quad (2.7)$$

2.3.1 Model

The motion model for a differential drive robot is simple [5]. Such a robot has two wheels aligned on a common axis that can turn independently. If the wheel radii are both r and are separated by distance L as shown in Fig. 2.4, and u_ℓ and u_r are the

velocities in radians per second of the left and right wheels, then the state updates are

$$\begin{aligned} \dot{x} &= \tfrac{r}{2}\left(u_\ell + u_r\right)\cos\theta \\ \dot{y} &= \tfrac{r}{2}\left(u_\ell + u_r\right)\sin\theta \\ \dot{\theta} &= \tfrac{r}{L}\left(u_r - u_\ell\right). \end{aligned} \tag{2.8}$$

The angular velocities u_ℓ and u_r can be approximated by a simple derivative of the wheel encoder measurements θ_ℓ and θ_r, where the encoder measurements do not roll over. On the SegMonster, a low pass filter is used to smooth these derivatives.

$$\begin{aligned} u_\ell &= \theta_t^{[\ell]} - \theta_{t-\Delta t}^{[\ell]} \\ u_r &= \theta_t^{[r]} - \theta_{t-\Delta t}^{[r]}. \end{aligned} \tag{2.9}$$

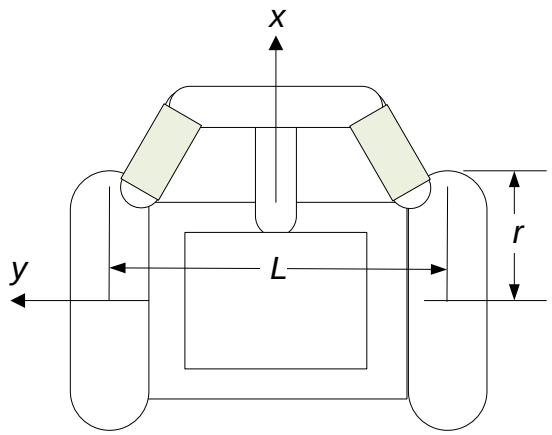


Figure 2.4: Parameters for a differential drive robot. Differential robots that cannot balance themselves dynamically must include a caster wheel for support.

2.3.2 Results

In practice, dead reckoning exhibits a drift from the true estimate, with increasing variance. On the SegMonster, the drift does not have a zero mean due mainly to

the difference in tire pressure. In simulation, when the state update equations are perturbed by noise, the dead reckoned estimate also diverges. Figure 2.5 shows the effects of process noise on the state.

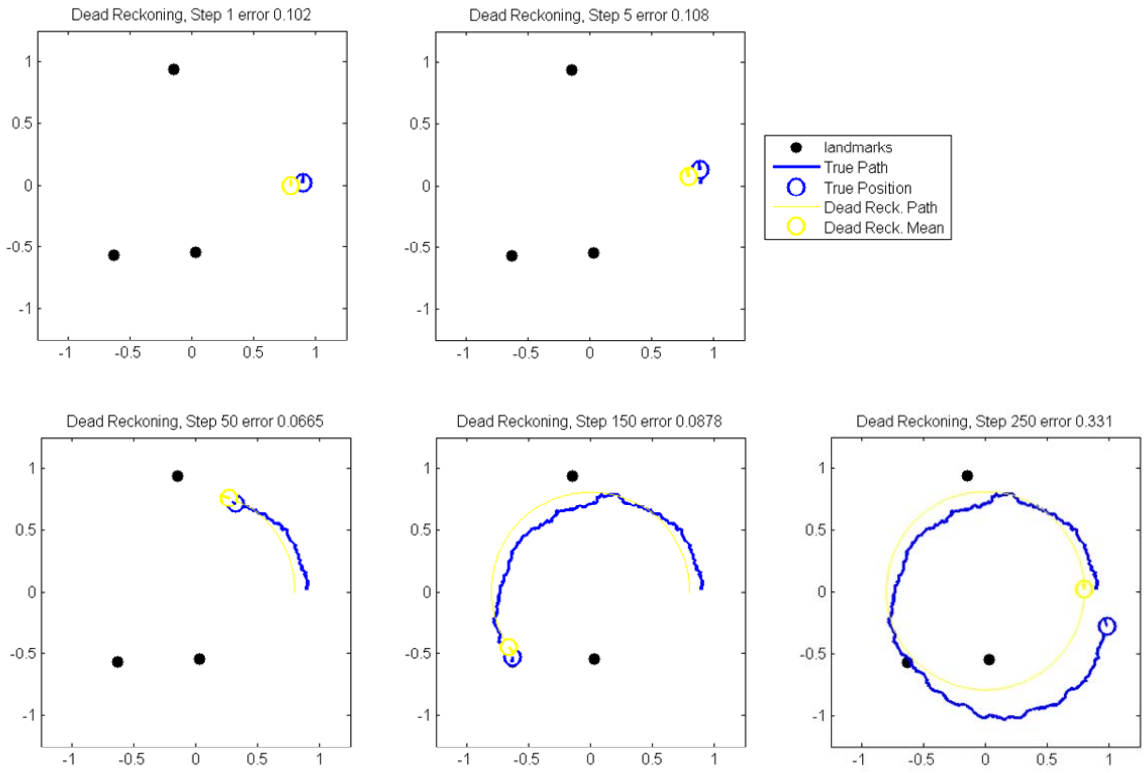


Figure 2.5: Simulation showing effects of dead reckoning. The simulated mobile robot had control inputs $[Speed, AngularVelocity]$ corrupted by random noise with variance $1e^{-2}$. The robot was commanded to travel in a unit circle, and the initial pose was corrupted by $1e^{-2}$ random noise as well. The true position is shown in blue, while the dead reckoned estimate is in yellow. The dead reckoned estimate does not, in general, converge to the true position.

2.4 Observers

The inadequacies of dead reckoning require additional sensors. Global sensors return measurements $\mathbf{y}_t$ that correspond to the true state $\mathbf{x}_t$. The mapping is done though a matrix $\mathbf{H}$ so that $\mathbf{y}_t = \mathbf{H}\mathbf{x}_t$. If $\mathbf{H}$ is invertible (has full rank), then the state $\mathbf{x}_t$ can be found directly from the current measurement. Often $\mathbf{y}_t$ is a list of distances

to landmarks, and **H** is not full rank. In this case, **H** cannot be inverted. Figure 2.6 shows range to landmark measurement readings for a mobile robot. In order to use the range estimates, the robot must have a map listing the true locations of each landmark. Notice that the measurements do not overlap, as they should with no measurement noise. The range measurements are corrupted by noise (here with variance $1e^{-2}$) and the true position cannot be calculated by simple triangulation.

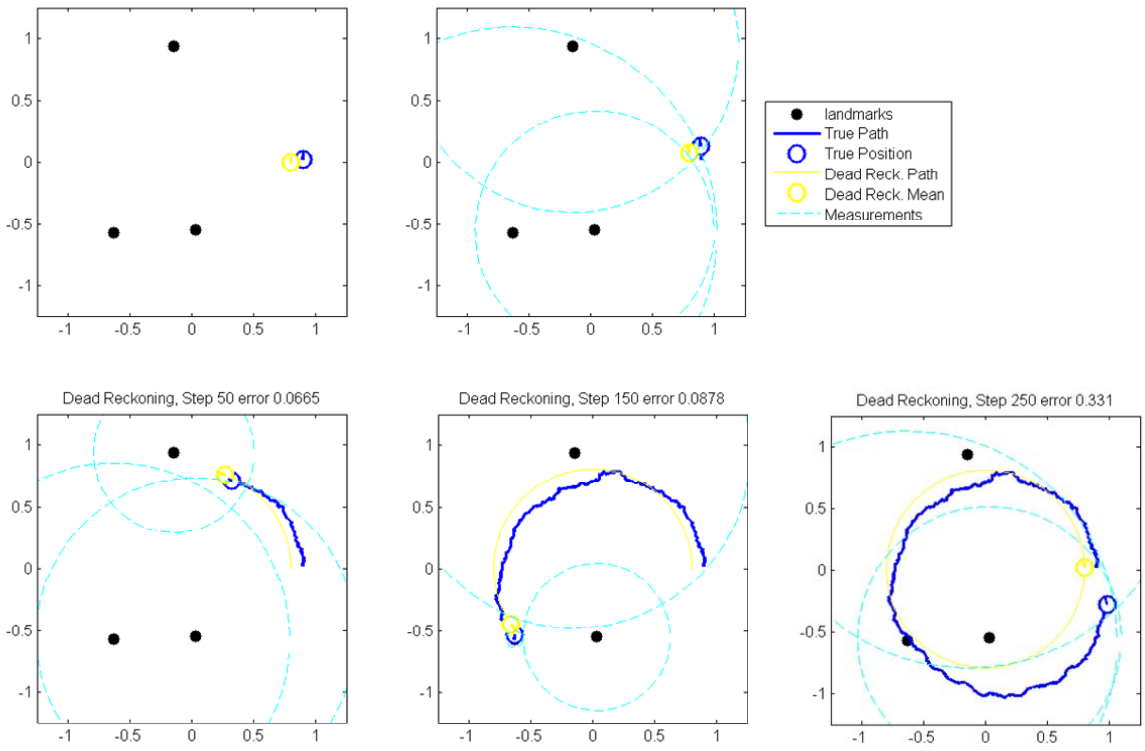


Figure 2.6: Simulation showing range to landmark readings. Here the simulation is run with the same random seed value as before. This time the robot uses a sensor to measure range to each landmark. The landmarks are denoted as black points, and their range measurement is drawn in cyan. In order to use the range estimates, the robot must have a map listing the true locations of each landmark. *Note:* the range measurements are corrupted by noise (here with variance $1e^{-2}$) and the true position cannot be calculated by simple triangulation.

In control system design, having fewer sensors than states, or having sensors that do not return our states directly, is a common problem. The standard technique for

dealing with this is to design an *observer*, a mathematical model that maintains a state estimate $\hat{\mathbf{x}}$ of its own. This state estimate is mapped to an output estimate $\hat{\mathbf{y}}$, through $\mathbf{H}$, and compared to the actual output $\mathbf{y}_t$. The resulting error signal,

$$\mathbf{e}_t = \mathbf{y}_t - \mathbf{y}_{t+\Delta t}, \tag{2.10}$$

is used to correct the next state estimate $\mathbf{x}_t + \Delta t$. One way to do this is to construct the pseudo-inverse of $\mathbf{H}$:

$$\tilde{\mathbf{H}}^{-1} = \mathbf{H}^T(\mathbf{H}\mathbf{H}^T)^{-1}. \tag{2.11}$$

The pseudo-inverse is used to calculate a correction factor to update the state estimate. Figure 2.7 shows this in a block diagram. In practice, a matrix $\mathbf{L}$ is used to scale the correction factor $\tilde{\mathbf{H}}^{-1}$. $\mathbf{L}$ is designed to cause the estimated state to decay gradually to the true state. For linear systems, $\mathbf{L}$ is chosen so that the matrix **F-LH** has reasonably fast eigenvalues. The MATLAB command `L = place(F',H',desired_poles)'` calculates the appropriate $\mathbf{L}$ for any desired poles of the observer. In practice, we rarely have a perfect model of the plant **(F,G,H)**. Even if the plant is not completely accurate, $\mathbf{L}$ can still be chosen to keep the error acceptably small, under certain conditions.

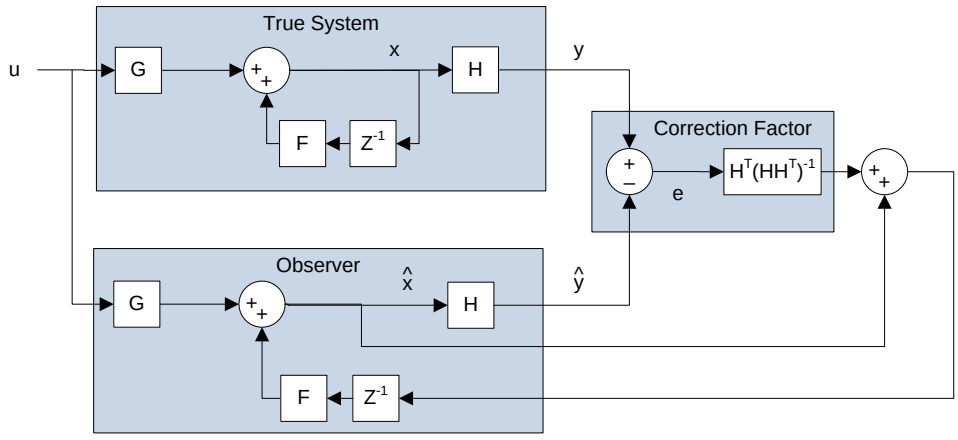


Figure 2.7: Simple observer block diagram. The error $\mathbf{e}$ between the expected measurement and the actual measurement is used to correct the state estimate.

2.4.1 Requirements for observers

For an observer to converge to the true state, the system must be stable, observable, and linear [6]. A *stable* system has all poles in the left-hand plane (the eigenvalues of the system matrix are all negative). Being *observable* is a structural property of the system that indicates whether the complete state can be reconstructed from only the outputs $\mathbf{y}$. A simple mathematical test of observability for linear systems is to construct a matrix called the *observability matrix* and evaluate it to determine if it has independent columns. The observability matrix $\circledcirc$ is calculated using the state update equation $\mathbf{F}$ and the output mapping matrix $\mathbf{H}$:

$$\circledcirc = \begin{bmatrix} \mathbf{H} \\ \mathbf{HF} \\ \mathbf{HF^2} \\ \vdots \\ \mathbf{HF^{n-1}} \end{bmatrix}. \tag{2.12}$$

A *linear system* is a system that can be described as a linear sum of independent components. Linear systems can be written matrix in form $\dot{\mathbf{x}} = \mathbf{Ax} + \mathbf{Bu}$, where $\mathbf{A}$ and $\mathbf{B}$ are scalar matrices. Under these constraints the observer can be designed to converge to the correct state at any rate.

2.4.2 Case study: Reaction wheel pendulum

In ECE 486, a control systems class for upper-level undergraduates and first semester graduate students, the final project involves controlling a reaction wheel pendulum (RWP, see Fig. 2.8). The RWP is a simple pendulum with a motor-powered rotating wheel as the pendulum bob. The pendulum arm is unactuated, making this a nonlin-

ear, under actuated system with two degrees of freedom. The two links have optical encoders to measure position. The system state is composed of the orientation and angular velocity of the two links:

$$\mathbf{x} = \begin{bmatrix} \theta_p & \dot{\theta}_p & \theta_r & \dot{\theta}_r \end{bmatrix}.$$

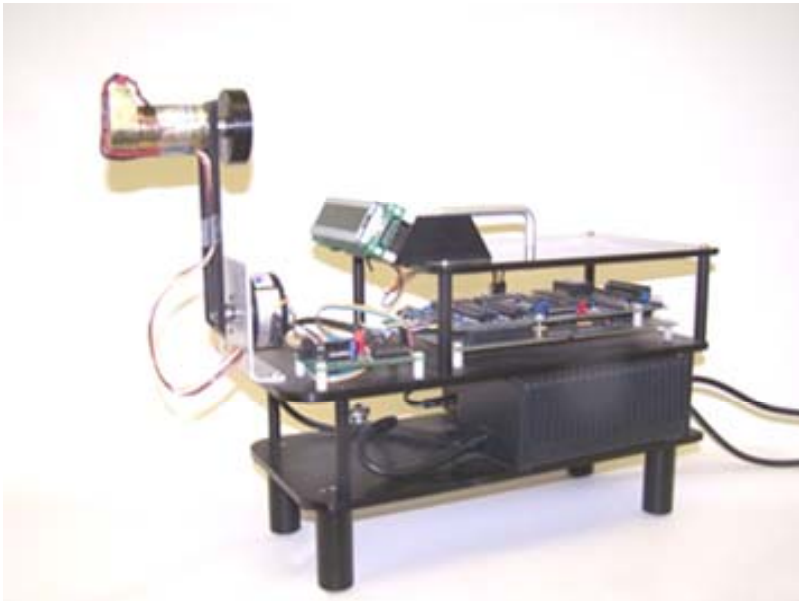


Figure 2.8: The RWP balancing at an unstable equilibrium position. The only motor in the system is equipped with a heavy wheel and then attached to the end of a pendulum. By exerting positive torque on the wheel, an equal and opposite reaction torque is applied to the pendulum. This torque can be used to control and stabilize the system.

Figure 2.9 shows how the orientations are measured. Two optical encoders measure the orientation of the pendulum and motor rotor, but the velocities are unmeasured. The system has two more states than sensors. An observer can be used to recover these hidden states. First, the system must be linearized about an operating point ($\theta_p \in [0, \pi]$). The controller must run a mathematical model alongside the model.

This model is called an observer, and the observer state estimate $\hat{\mathbf{x}}$ tracks the true state variables $\mathbf{x}$, if properly designed. The observer predicts the sensor measurements by multiplying the state estimate by $\mathbf{H}$. This prediction, $\hat{\mathbf{y}}$ is compared to the true measurement and fed back to the observer to continually pull the state estimate to the true value. Figure 2.10 shows a block diagram of the RWP and its observer.

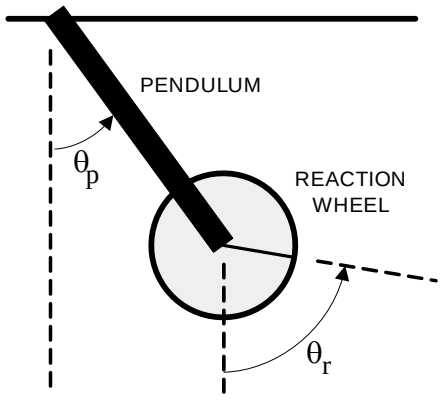


Figure 2.9: RWP diagram. Both θ_p and θ_r are referenced from vertical.

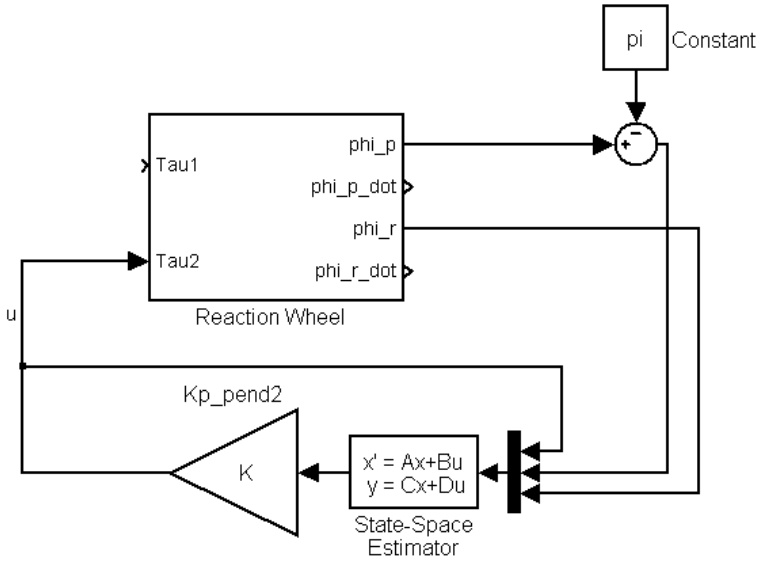


Figure 2.10: Block diagram of the RWP controller. The bottom state space block observes the control input $\mathbf{u}$ and the sensor output $\mathbf{y}$ to create a state estimate, $\hat{\mathrm{x}}$.

The system is robust to faulty initial conditions. Because the observer is linear, it can recover from arbitrarily large initial condition errors. Figure 2.11 shows two unmeasured states being found and tracked by an observer.

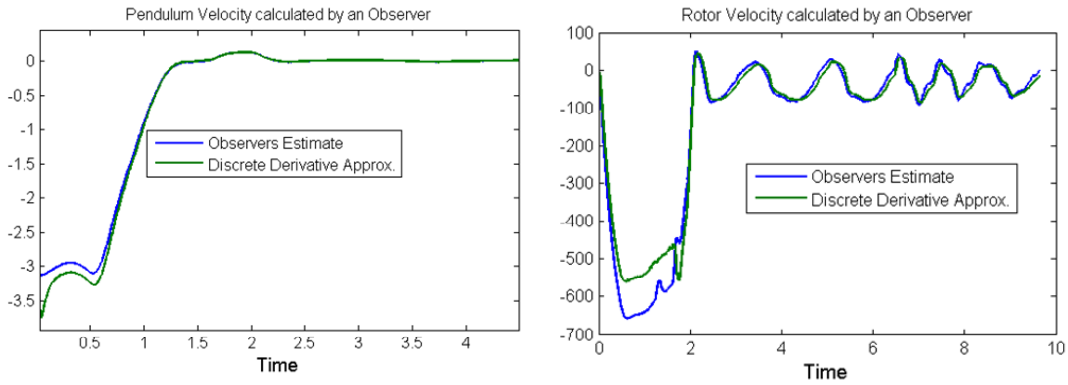


Figure 2.11: Observers can track system states of linearized nonlinear systems. Here the pendulum arm velocity and rotor velocity estimate (blue) converge to the unmeasured true state values (green, calculated by taking the discrete derivative of the orientations).

2.4.3 Observers and robot localization

Observers can be designed to estimate unmeasured system states, and have the remarkable property that their tracking error goes to zero as time goes to infinity. Observers can even estimate the state of a nonlinear system like the RWP. We wish to estimate the state of a mobile robot using sensor measurements that do not directly reveal the state. As a first approach to robot localization, it is very natural to try an observer. Figure 2.12 shows the results of an observer during a mobile robot simulation. The robot was equipped with range to landmark sensors.

In simulation the estimated state usually blows up. As mentioned previously, observers are only guaranteed to converge to the correct state for systems that are stable, observable, and linear. Unfortunately, the motion model for the SegMonster

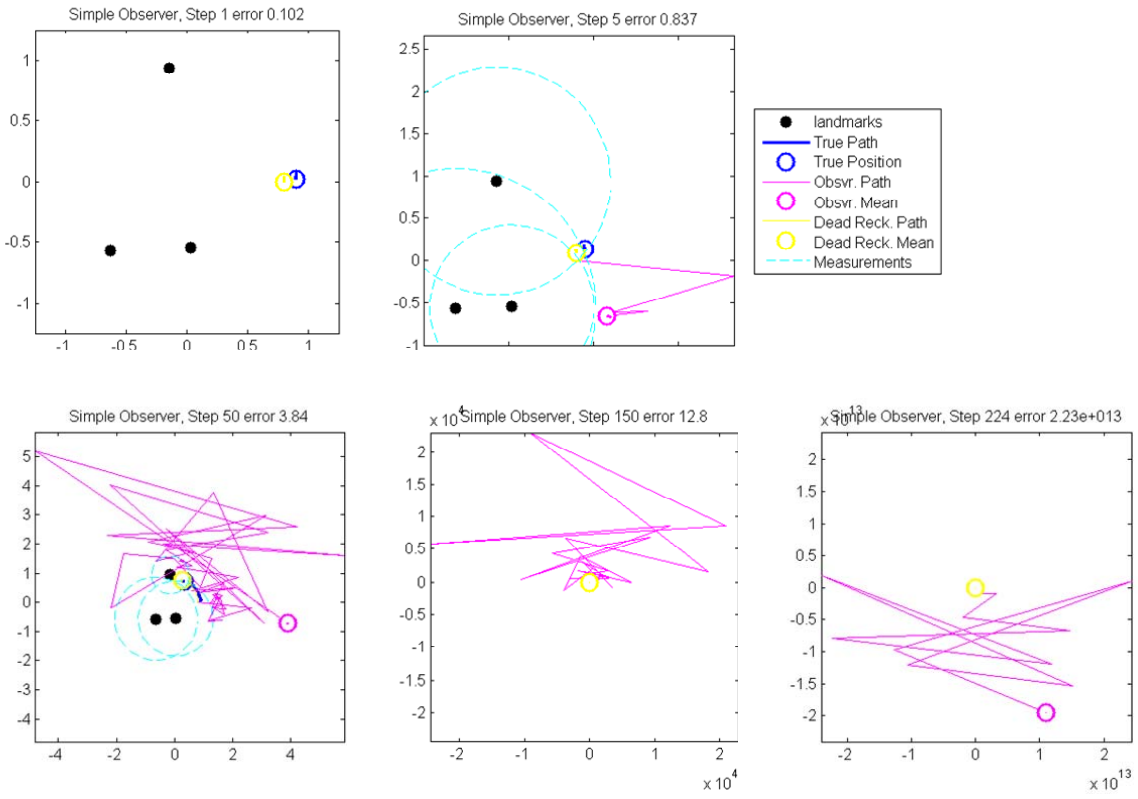


Figure 2.12: A simple observer on a mobile robot usually goes unstable.

violates each constraint. The linearized state update $\mathbf{F}$ has three poles at the origin, and so each mode is only marginally stable. Also, for many configurations, the state matrix $\mathbf{H}$ is not observable. Finally, the trigonometric functions in $\mathbf{F}$ ensure that the system is not linear. In general, the observer can only correct estimate errors visible in the error equation $\mathbf{e}_t = \mathbf{y}_t - \mathbf{y}_{t+\Delta t}$. In simulation the estimated state poorly tracks the true state and often grows unbounded. This is because the psuedo-inverse $\tilde{\mathbf{H}}^{-1}$ becomes singular. This is easily seen in a simple example. A robot that translates without rotating has a state described by $\mathbf{x} = [x, y]$. If it is equipped with an IR sensor pointing directly forward and measures a wall 5 m in front of the robot, the $\mathbf{x}$ estimate can be corrected, but the $\mathbf{y}$ estimate cannot be improved. The error on the x estimate will decay to zero, but the true location cannot be determined.

2.5 Localization, beyond Local Sensors

Clearly, the dead reckoning approach is inadequate for position tracking for any length of time. According to King, current state-of-the-art navigation systems have an inaccuracy "normally fewer than 0.6 nautical miles per hour in position and on the order of tenths of a degree per hour in orientation" [4]. Certainly our estimate could be improved by augmenting the SegMonster's local sensors. The mobile robots used in the Mechatronics class at UIUC use the same DSP as the SegMonster, and are each equipped with a rate gyro, a small microelectromechanical system (MEMS) sensor that measures angular acceleration. By integrating this data, the robot can measure orientation with error rates in the degrees per minute—much better than our dead reckoning estimate, though not state-of-the-art. These mobile robots are four wheeled, and remain nearly parallel to the floor even during fast turns. Using rate gyro measurements on the SegMonster would be complicated by the pitching motion inherent to an inverted pendulum system, but could be done.

All local sensors, such as integrating motion commands, integrating wheel encoder measurements, or integrating rate gyros and accelerometers, suffer from parameter drift. Some (such as the rate gyro and accelerometers) can drift even when the robot is standing still. This causes the position estimate to steadily decay over time. Even when the robot is initialized with its true position, the estimate uncertainty grows with time and distance traveled, as in Fig. 2.1. We can mitigate this by investing in better quality sensors, but we are only slowing down the rate of information loss. To increase position certainty, the robot must incorporate information from the world around it. Two basic approaches to this problem are to add global sensors that directly measure system states, or to use a map to filter out improbable position states and select probable ones.

- *Global Approaches—Triangulation*

  Some global sensors return direct state information, or receive external state estimates from other agents. Several labs at UIUC use ceiling mounted cameras that have dedicated processors to segment out the robot's location and return this pose information to the robot. The Global Positioning System (GPS) developed by the US Department of Defense, uses at least 24 orbiting satellites that transmit precise microwave signals [7]. A robot equipped with a GPS receiver can use these signals for accurate localization. The signal contains the satellite identification number and precise time. By comparing several of these signals to an almanac giving the satellite's expected position, the robot can triangulate its position. Unfortunately, buildings effectively shield GPS microwave signals, limiting their suitability for indoor robotic localization.

  Other signals can be used in much the same way. Infrared (IR) signals are one such type of signal. The first work on IR-based location systems was the *Active Badge Localization System*, which was composed of a network of building mounted sensors, and small badges worn by employees [8]. These badges emitted a unique code every 15 s, which was detected by the network of sensors. A central processor calculated the position of each badge seen and stored this in a database accessible to the users. The system, though accurate at identifying user's room location, had several problems. The system required a dedicated infrastructure that had to be installed and maintained. The IR signals performed poorly in rooms with direct sunlight. It also scaled poorly. While GPS requires 24 satellites in order to function globally, the Active Badge system used 200 sensors to cover four buildings.

Many other signal combinations have been tried. The cricket system uses concurrent ultrasound and RF beacons for localization [9]. In this system, ceiling mounted beacons are placed at known locations throughout a building. Periodically, these beacons simultaneously transmit an RF signal containing location information of the beacon and an ultrasonic pulse. Any listening devices within range can use the difference in signal propagation times to triangulate location information. This system, like GPS, avoids having a central processor that locates each device. Like the Active Badge system, using crickets requires a significant investment for installing and maintaining the beacons. Once more, scaling is an issue. The number of beacons required increases proportionally with the area to be covered.

The prison guard *Duress alarm location system* uses only RF signals to locate prison guards [10]. The system consists of RF badges, worn by the guards, and sensor/relay modules with RF receivers and antennas. When a badge is activated, each sensor/relay module measures the signal strength from the badge and the data are processed to estimate the guard's position. The system is constructed so that at least three sensors are within range of each area to be protected. Unlike IR and, to a lesser degree, the RF/ultrasound combination on the crickets, using only RF data is plagued by signal noise and *multipath effects*, which stem from the same signal being reflected so that several copies arrive at the same destination. The Duress system, like all dedicated localization systems, requires a large investment for setup and maintenance.

Many sensors have seen publication and some, production. Some of the devices used for minimally invasive surgery are robots in their own right, and tracking

their position inside a human body is of paramount importance. Pulsed DC magnetic fields can be used to accurately localize surgical tools [11].

A less exotic magnetic sensor is the common compass. A compass provides a sensor measurement of θ with respect to the world's reference frame. If working correctly with no disturbance, this measurement allows the robot to determine its orientation exactly, except near the magnetic poles of the earth. A perfect compass would reduce the mobile robot's state space estimate by one, an ideal circumstance. In practice indoors, the compass is plagued with large, localized disturbances. These disturbances have many causes, including metal structural beams and magnetic fields from DC motors. The mobile robots used in the Mechatronics class at UIUC use the same DSP as the SegMonster, and each is equipped with a compass. This compass data is rarely used, due to the large disturbances indoors, but using the compass could prove helpful to the SegMonster when it transfers to an outdoor environment.

- *Global Approaches—Map-Based Methods*

  A *map* is a (partial) model of the world. The robot can compare sensor data to the model given by the map to determine possible locations. The dividing line between map-based approaches and global sensors is not well defined. Most of the references cited above had some form of map, but were able to directly triangulate the robot's position. Purely map-based approaches cannot find the current position by triangulation.

  These maps can take many forms. Minerva, a museum tour guide robot, used mosaic images of the ceiling, along with measurements from a laser range finder

to determine location [12]. Both maps—an occupancy map for the laser range finder, and the compiled image of the ceiling—were used by the robot for global localization and position tracking

Light intensities in a building can be measured and stored in a map, keyed with location information. It was demonstrated in [13] that a light sensor mounted on a hat could be used for room-level localization with 90% accuracy. Accuracy can be improved by setting room lights to nonoverlapping illumination levels.

Contrary to popular belief, accurate indoor localization has been demonstrated using the Global System for Mobile communication (GSM). In [14], signal information from up to 35 GSM channels was used for localization. By using wide-area GSM fingerprints, the authors reported an average localization error of about 4 m. This was done by *fingerprinting*, where a map of signal strength was compiled by moving around the map and collecting signal strength data. During the localization stage, this map was searched to find the K signal strength sets that best matched the current reading, and took a weighted average of this data to estimate the true position. This algorithm is called the k-nearest neighbor(s) in signal space (K-NNSS).

In Chapter 4, wireless signal strength information is recorded at different $[x, y]$ positions and used to construct a model of probable sensor measurements throughout the configuration space of the robot.

CHAPTER 3

PROBABILISTIC OBSERVERS

> Probability is expectation founded upon partial knowledge. A perfect acquaintance with *all* the circumstances affecting the occurrence of an event would change expectation into certainty, and leave nether room nor demand for a theory of probabilities.
>
> —George Boole

Estimating position by only calculating and correcting the mean is problematic. It allows no uncertainty in representation - at each time step the observer returns one possible pose value. As shown earlier, a probability distribution does represent certainty. A probability distribution assigns a probability to every possible pose. We need an observer that will maintain a probability distribution at each time step. Correctly updating this probability is known as the *Bayesian filter*.

3.1 Bayesian Filter

To propagate probability functions, we need a few tools. The *theorem of total probability* states that

$$p(x) = \begin{cases} \int_Y p(x|y)p(y)dy & p(y) \neq 0, p(x|y) \neq 0 \\ 0 & otherwise. \end{cases} \tag{3.1}$$

To relate a conditional of the form $p(x|y)$ to its inverse $p(y|x)$, we can use *Bayes' rule*:

$$p(x|y) = \frac{p(y|x)p(x)}{p(y)} = \frac{p(y|x)p(x)}{\int_X p(y|x')p(x')dx'}. \tag{3.2}$$

Note that the denominator does not depend on the x in the numerator and can be generated from any x in the posterior $p(x|y)$. For this reason, the denominator, $p(y)^{-1}$, is often denoted by η and is known as the *normalizer*.

Any probability can be conditioned on an arbitrary random variable, Z. As long as $p(y|z) \neq 0$, Bayes' rule can be expanded as

$$p(x|y,z) = \frac{p(y|x,z)p(x|z)}{p(y|z)} = \eta p(y|x,z)p(x|z). \tag{3.3}$$

The probability that the robot pose $\mathbf{x}_t$ at time t is any pose $\mathbf{x}_a \in \mathbf{X}$ is called the belief, and it is represented by $bel(x_t)$. It is given by the conditional probability

$$\begin{aligned} bel(x_t) &= p\left(x_t|u_{0:t-1}, y_{1:t}, x_0\right) \\ bel(x_t) &= p\left(x_t|u_{0:t-1}, y_{1:t}\right). \end{aligned} \tag{3.4}$$

This term is called the *posterior probability*, the probability distribution over the state x_t at time t, given all the sensor readings and movements. The first equation is for the position tracking problem and the second is for global localization. The robot's movements from the beginning to the present are $u_{0:t-1}$, and $y_{1:t}$ are the sensor readings from the first time step to the current time.

In the following section it will also be necessary to represent the belief before integrating the last sensor measurement y_t. This posterior will be represented as $\overline{bel}(x_t)$.

$$\overline{bel}(x_t) = p\left(x_t | u_{0:t-1}, y_{1:t-1}\right). \tag{3.5}$$

The equation is called the *prediction* because it takes the previous posterior probability $bel(x_{t-1})$ and integrates the last movement u_t. It predicts the effect of the movement on the belief.

To derive the update equations for $bel(x_t)$, we start with Bayes' rule.

$$\begin{aligned} bel(x_t) &= p\left(x_t | u_{0:t-1}, y_{1:t}\right) = \frac{p(y_t | x_t, u_{0:t-1}, y_{1:t-1}) p(x_t | u_{0:t-1}, y_{1:t-1})}{p\left(y_{1:t} | u_{0:t-1}, y_{1:t-1}\right)} \\ &= \eta p\left(y_t | x_t, u_{0:t-1}, y_{1:t-1}\right) p\left(x_t | u_{0:t-1}, y_{1:t-1}\right). \end{aligned} \tag{3.6}$$

Note that $p\left(x_t | u_{0:t-1}, y_{1:t-1}\right)$ is the belief $\overline{bel}(x_t)$, and therefore

$$bel(x_t) = \eta p\left(y_{1:t} | x_t, u_{0:t-1}\right) \overline{bel}(x_t).$$

This equation can be simplified by making three key assumptions:

1. *Independence of sensor readings from previous movement, given current pose*
   We can usually assume that the sensor reading does not depend on the previous position given the current position and therefore

   $$p\left(y_{1:t} | x_t, u_{0:t-1}, y_{1:t-1}\right) = p\left(y_{1:t} | x_t\right).$$

2. *Recursive nature of the motion model*

We can expand $\overline{bel}(x_t)$ using Eq. (3.1) as

$$\begin{aligned}\overline{bel}(x_t) &= p\left(x_t|u_{0:t-1}, y_{1:t-1}\right) \\ &= \int_X p(x_t|x_{t-1}, y_{1:t-1}, u_{0:t-1}) p(x_{t-1}|y_{1:t-1}, u_{0:t-1})\, dx_{t-1}.\end{aligned}$$

Notice that $p(x_{t-1}|y_{1:t-1}, u_{0:t-1})$ is simply the previous posterior. The previous posterior is called the *prior* and is $bel(x_{t-1})$. This reduces the integral to

$$\overline{bel}(x_t) = \int_X p(x_t|x_{t-1}, y_{1:t-1}, u_{0:t-1}) bel(x_{t-1})\, dx_{t-1}.$$

3. *If we know x_{t-1}, the movements $u_{0:t-2}$ and the sensor measurements $y_{1:t-1}$ add no additional information:*

$$\overline{bel}(x_t) = \int_X p(x_t|x_{t-1}, u_{t-1}) bel(x_{t-1})\, dx_{t-1}. \tag{3.7}$$

This step is called the motion model, and it describes how the prior is updated by the movement.

The Bayesian filter provides the current posterior, given the prior $bel(x_{t-1})$, the last movement u_t, and the current measurement y_t. The filter is recursive in that it only depends on the previous state and the current inputs and output. An algorithm in psuedo code is given as Algorithm 1.

The integral on line 3 rarely has a closed form. Because of the difficulty involved in evaluating these integrals, as Fig. 3.1 hints at, different approximations have been attempted. One notable exception is the Kalman filter, which can be solved directly for linear systems with Gaussian normal noise. For systems with non-Gaussian noise, or nonlinear systems, the integral can be approximated by a sum over a discritized

Algorithm 1 Bayesian_filtering($bel(x_{t-1}), u_t, y_t$)

1: $\tau = 0$
2: **for** all $x_t \in X$ **do**
3: $\quad \overline{bel}(x_t) = \int p(x_t|u_t, x_{t-1}) bel(x_{t-1})\, dx_{t-1}$
4: $\quad \gamma = p(y_t|x_t)\overline{bel}(x_{t-1})$
5: $\quad \tau = \tau + \gamma$
6: $\quad unNormalized\_bel(x_t) = \gamma$
7: **end for**
8: $\eta = 1/\tau$
9: **for** all $x_t \in X$ **do**
10: $\quad bel(x_t) = \eta \cdot unNormalized\_bel(x_t)$
11: **end for**
12: **return** $bel(x_t)$

state space. This approximation converges to the Bayesian filter in the limit as the grid spacing goes to zero.

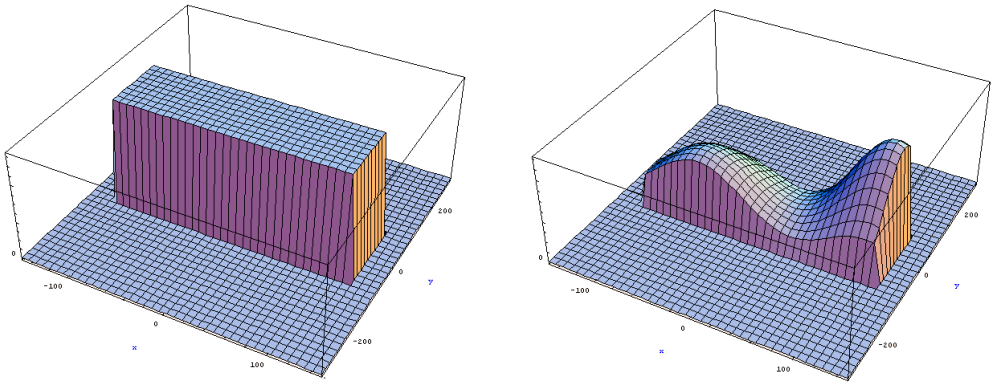


Figure 3.1: Bayesian probability update. A uniform continuous $bel(x_0)$ distribution for $[x, y]^T$ in a rectangular hallway is relatively easy to construct, but updating $bel(t_k)$ becomes increasingly difficult.

3.2 Multidimensional Normal Distributions

Updating most probability distributions requires integration over each degree of freedom, which would often be intractable, except at coarse levels of discritization. The

multidimensional normal distribution is an exception to that rule. Multidimensional normal distributions are used for two main reasons:

- *Simplicity of representation*

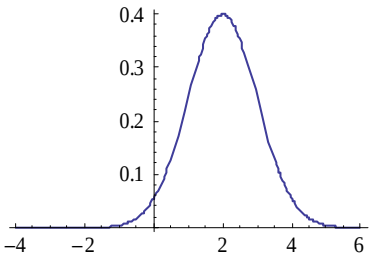
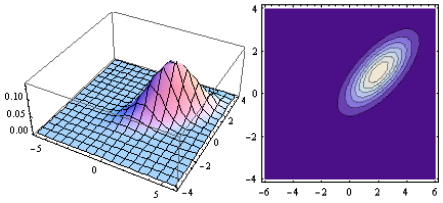
  A normal distribution can be represented by two parameters: a mean (matrix of size n) and a covariance matrix (of size n^2), where n is the dimension of the state vector x.

- *Representational scope*

  Many distributions in life can be represented by a multidimensional normal distribution. The *central limit theorem* states that if $\bar{\mathbf{x}}_n$ is the mean of n samples from any distribution having finite variance σ^2 (and thus a finite mean μ), then $\sqrt{n}(\bar{\mathbf{x}}_n - \mu)/\sigma)$ will converge in distribution to a random variable having a standard normal distribution [15].

Table 3.1 shows a few representative normal distributions.

Table 3.1: Normal Distributions

Dim	Parameters	Example	Representation
1	(μ, σ^2)	$(2, 1)$	
2	(μ_1, Σ_1)	$\left(\begin{bmatrix} 2 \\ 1 \end{bmatrix}, \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix} \right)$	

3.3 Representing Beliefs

Sensor measurements, like state updates, are rarely perfect. Often their readings can be represented by a normal distribution with a mean at the true value. In this case, the measurement we get, $\mathbf{y}_t$, is a random variable from a normal distribution centered at μ, with a covariance matrix inherent to the sensor. A useful theorem is that the product of two normal distributions (μ_1, Σ_1) and (μ_2, Σ_2) is proportional to a normal distribution with mean $\mu_3 = \mu_1 + Q(\mu_2 - \mu_1)$ and covariance matrix $\Sigma_3 = \Sigma_1 - Q\Sigma_1$, where $Q = \Sigma_1 (\Sigma_1 + \Sigma_2)^{-1}$.

3.4 Observing with Probability

We know that as the robot moves it loses information. If we assume that our noise is normally distributed, we can reflect that loss of certainty by propagating a covariance matrix along with the state estimate. Figure 3.2 shows how the observer can be augmented to propagate a covariance matrix $\mathbf{P}$.

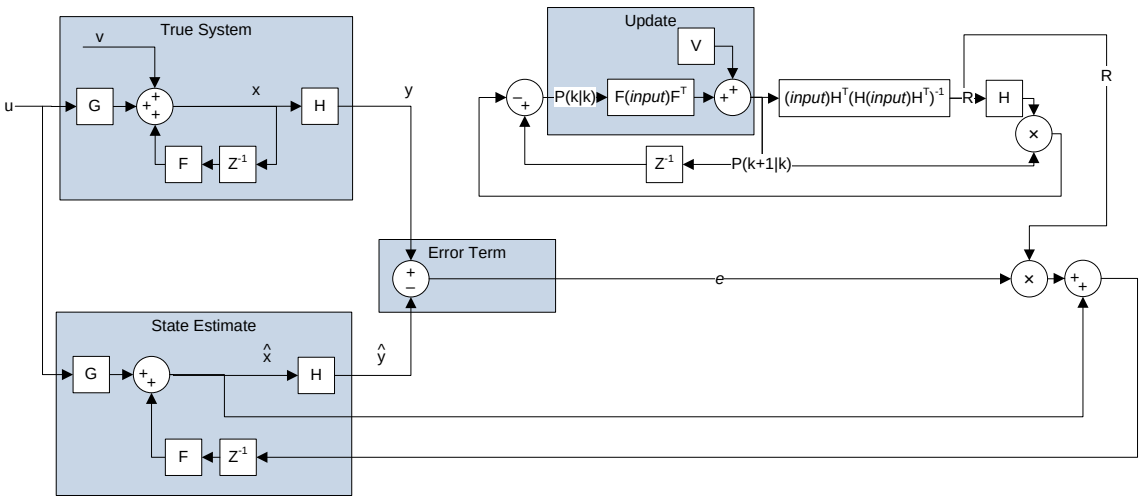


Figure 3.2: Block diagram for an observer that maintains a covariance matrix. It has many similarities with Fig. 2.7. The only difference is the upper right loop. This loop maintains the covariance matrix and uses it to scale the error term.

The problem with this approach, much like the problem with the observer, is that the matrix inversion step $\mathbf{HPH}^{-1}$ becomes singular. Generically, a matrix $\mathbf{A}$ is invertible (nonsingular) if a matrix $\mathbf{A}$ exists such that $\mathbf{AB} = \mathbf{BA} = \mathbf{I}_n$. If the $\mathbf{H}$ matrix ever causes a column of $\mathbf{HPH}$ to go to zero, the matrix becomes singular and is not invertible.

3.5 Kalman Filter

The Kalman filter has much in common with the simple observer. This is particularly obvious in the block diagram, Fig 3.3. The chief difference is that the Kalman filter also maintains a covariance matrix $\mathbf{P}$ along with the state estimate $\hat{\mathbf{x}}$. Since a mean and covariance matrix together describe a normal distribution, the Kalman filter is actually propagating a normal distribution. The filter also assumes that process noise can be described by a normal distribution $\mathbf{V}(t)$. The difference between the Kalman filter and the observer with a covariance matrix is that the Kalman filter adds sensor noise to the equation. This sensor noise is described by a normal distribution $\mathbf{W}(t)$.

The Kalman filter has two stages. In the first, the input and process model are used to predict the next $\hat{\mathbf{x}}$. $\mathbf{P}$ is updated by the process model and process noise. Equation (3.8) shows that the prediction step only adds noise to the prediction $\hat{\mathbf{P}}$. The second stage uses the sensor readings to correct the state and variance prediction.

1. *Predict the state and variance*

$$\begin{aligned} \hat{\mathbf{x}}_{t+1|t} &= \mathbf{F}\hat{\mathbf{x}}_{t|t} + \mathbf{G}\mathbf{u}(t) \\ \hat{\mathbf{P}}_{t+1|t} &= \mathbf{F}\mathbf{P}_{t|t}\mathbf{F}^T + \mathbf{V}(t). \end{aligned} \tag{3.8}$$

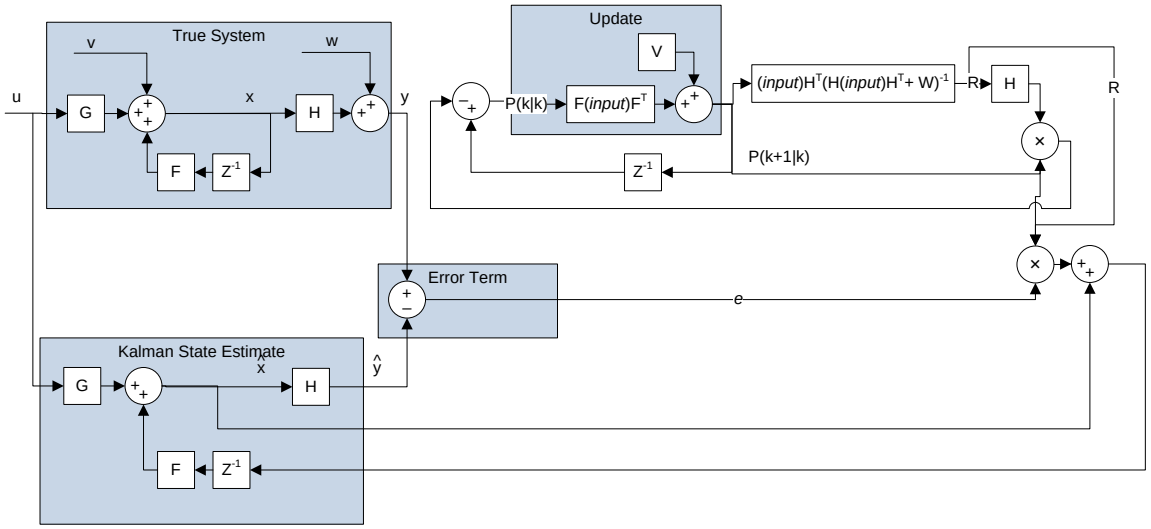


Figure 3.3: Block diagram for the Kalman filter. It has many similarities with Fig. 3.2. The only difference is that now sensor noise $\mathbf{W}$ is added before the matrix is inverted in the upper right.

2. *Correct the state and variance based on sensors*

$$\begin{aligned} \mathbf{x}_{t+1|t+1} &= \mathbf{F}\hat{\mathbf{x}}_{t+1|t} + \mathbf{R}\mathbf{e} \\ \mathbf{P}_{t+1|t+1} &= \hat{\mathbf{P}}_{t+1|t} - \mathbf{R}\mathbf{H}\hat{\mathbf{P}}_{t+1|t}. \end{aligned} \tag{3.9}$$

Here the intermediate variable $\mathbf{e}$ is still the error between the predicted and actual measurements, and $\mathbf{R}$ is a scaling matrix that determines how to correct the covariance $\hat{\mathbf{P}}$ and the state estimate $\hat{\mathbf{x}}$. A large $\mathbf{R}$ (in comparison with the identity matrix) weighs sensor measurements more than the prediction, while a small $\mathbf{R}$ means that sensor measurements are less reliable than state predictions. The necessary equations are

$$\begin{aligned} \mathbf{e} &= \mathbf{y}(t+1) - \mathbf{H}\hat{\mathbf{x}}_{t+1|t} \\ \mathbf{S} &= \mathbf{H}\hat{\mathbf{P}}_{t+1|t}\mathbf{H}^T + \mathbf{W}(t+1) \\ \mathbf{R} &= \hat{\mathbf{P}}_{t+1|t}\mathbf{H}^T(\mathbf{S})^{-1}. \end{aligned} \tag{3.10}$$

In block diagram form, these equations are similar to those of the observer in Chapter 2. The system state estimate $\hat{\mathbf{x}}$ is evolved in parallel with the true state, and the same

error $\mathbf{e}$ is constructed at each step. This $\mathbf{e}$ is now scaled proportionally to $\hat{\mathbf{P}}\mathbf{H}^T$ and inversely to the summation of $(\mathbf{H}\hat{\mathbf{P}}\mathbf{H}^T + \mathbf{W})$, where $\mathbf{W}$ is the assumed sensor noise. Figure 3.4 shows a simulation run of a mobile robot using a Kalman filter for localization. Unlike the dead reckoning and observer localization methods, the Kalman filter converges on the true position.

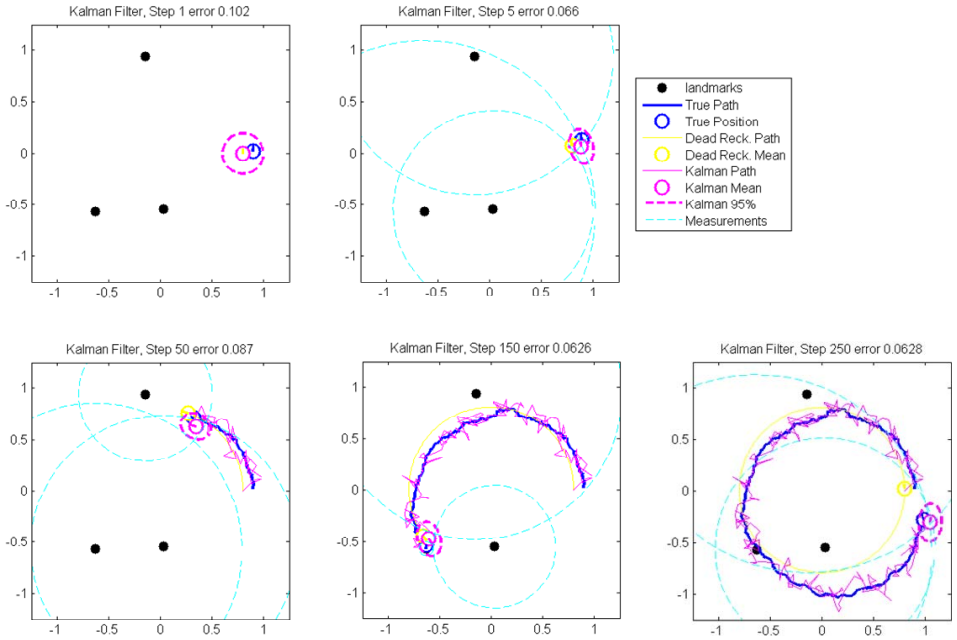


Figure 3.4: Simulation of Kalman filter. This simulation uses the same data as the previous simulations, Figs. 2.5, 2.6, 2.12. The magenta pose is the Kalman estimate, and the 95% confidence ellipse encloses all points with within 2 std. of the mean. Notice that the Kalman estimate tracks the true position (blue) while drift makes the dead reckoning mean (yellow) increasingly poor.

3.5.1 Time and memory constraints

One of the reasons the Kalman filter has remained popular for 40 years is its efficient handling of memory. Since it is a recursive formula that relies only on the past state, current control input, and current sensor readings, the memory constraints are modest. For an n degree of freedom system, n entries are required to record the system state, (necessary for any observer), n^2 entries to record the covariance matrix,

and memory space for the **H**, **R**, and **S** matrices. For most systems the processing is dominated by the inversion of the **S** matrix. The running time of Gauss-Jordan matrix inversion is $O(n^3)$ with storage requirements $O(n^2)$ [16]. The current best matrix inversion algorithm runs in $O(n^{2.376})$ [17], but has large constants hidden by the O notation.

3.5.2 Limitations to Kalman filter

Though the Kalman filter is the optimal linear estimator [18], there are many systems where a nonlinear estimator could outperform it. For the SegMonster, the state update matrix **F** is nonlinear. The **H** used in the simulation is also nonlinear. This is not a showstopper—it is a simple matter to linearize the **F** and **H** matrices at an operating point and use the original Kalman equations. The best operating point would be the true pose, $\mathbf{x}_t$. If our filter is performing well, we can assume that $\hat{\mathbf{x}}$ is near $\mathbf{x}_t$ and linearize at $\hat{\mathbf{x}}$. Now we still have the best linear estimator, but it is probable that there is a better nonlinear estimator. Linearizing at the assumed operating point is called the extended Kalman filter [19].

The Kalman filter is also limited in that it can only maintain one mean. If the sensor data equally supports two means, the only representation possible with a normal distribution is either to pick one over the other, or place the mean between the two possibilities and have a large covariance. Neither option reflects the data well. Techniques exist for propagating multiple Kalman filters, and for better supporting nonlinear systems (the unscented Kalman filter [19]).

3.5.3 Extensions to the Kalman filter

The Kalman filter cannot be implemented directly on the robot because the state update equation $\mathbf{F}$ and the sensor model $\mathbf{H}$ are nonlinear. We will propagate the state by the nonlinear model given as Eq. (2.5), but must linearize $\mathbf{F}$ and $\mathbf{H}$. These two equations can be linearized each time step by taking the Jacobian of the system and sensor matrices at $\hat{\mathbf{x}}$. The resulting equations are

$$\mathbf{F}_t = \frac{\partial f}{\partial x} \mid_{\mathbf{x}=\hat{\mathbf{x}}_{(t|t)}} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix}_{\mathbf{x}=\hat{\mathbf{x}}_{(t|t)}} \tag{3.11}$$

$$\mathbf{H}_{t+1} = \frac{\partial h}{\partial x} \mid_{\mathbf{x}=\hat{\mathbf{x}}_{(t|t)}} = \begin{bmatrix} \frac{\partial h_1}{\partial x_1} & \frac{\partial h_1}{\partial x_2} & \cdots & \frac{\partial h_1}{\partial x_n} \\ \frac{\partial h_2}{\partial x_1} & \frac{\partial h_2}{\partial x_2} & \cdots & \frac{\partial h_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial h_n}{\partial x_1} & \frac{\partial h_n}{\partial x_2} & \cdots & \frac{\partial h_n}{\partial x_n} \end{bmatrix}_{\mathbf{x}=\hat{\mathbf{x}}_{(t|t)}}. \tag{3.12}$$

In the case of our simulation, a differential drive robot with range to landmark readings for sensor measurements, the linearized equations are

$$\mathbf{F} = \begin{bmatrix} 1 & 0 & -v_t \cdot \sin\theta_t \\ 0 & 1 & v_t \cdot \cos\theta_t \\ 0 & 0 & 1 \end{bmatrix} \tag{3.13}$$

and

$$\mathbf{H}_i(t+1, j) = \begin{bmatrix} \frac{\hat{x}_{t+1|t} - x_{\ell j}}{\sqrt{(\hat{x}_{t+1|t} - x_{\ell j})^2 + (\hat{y}_{t+1|t} - y_{\ell j})^2}} & \frac{\hat{y}_{t+1|t} - y_{\ell j}}{\sqrt{(\hat{x}_{t+1|t} - x_{\ell j})^2 + (\hat{y}_{t+1|t} - y_{\ell j})^2}} \end{bmatrix}. \tag{3.14}$$

$\mathbf{H}_i(t+1, j)$ is the sensor map from state $\mathbf{x}$ to sensor reading $\mathbf{y}_j$ for the j^{th} landmark

detected. Here the sensor reading is the distance from the robot to landmark i, where the location of landmark i is $[x_{\ell i}, y_{\ell,i}]$. In the simulation, we detect all j landmarks each step, but this is not necessary for the filter to work.

Linearizing the state and sensor matrices to fit the Kalman filter to a nonlinear model is called the extended Kalman filter (EKF).

3.5.4 Unscented Kalman filter

Other methods for dealing with nonlinearities include propagating the mean along with several carefully selected points, to better represent the nonlinear transformation. This technique is known as the unscented Kalman filter (UKF).

3.6 Discrete Approximations

When the system equations are nonlinear, the Kalman filter can be a poor representation of the belief. Though the Bayesian filter equations may be intractable, it is often easy to discritize the state space by gridding it into nonoverlapping cells, and do a discrete approximation of the Bayesian probability. State transition probabilities must be defined for each state, and the probability of each cell is determined by evaluating a pose at the center of the cell. This allows us to replace the integrals in the Bayesian filter with summations, in exchange for a lower level of accuracy. We have three states for our model and so would need a three-dimensional grid. The system is simpler if we only consider position, as in Fig. 3.5.

Two problems plague grid based methods.

1. *Curse of dimensionality*

   Memory and time increase exponentially with the dimension of the state space.

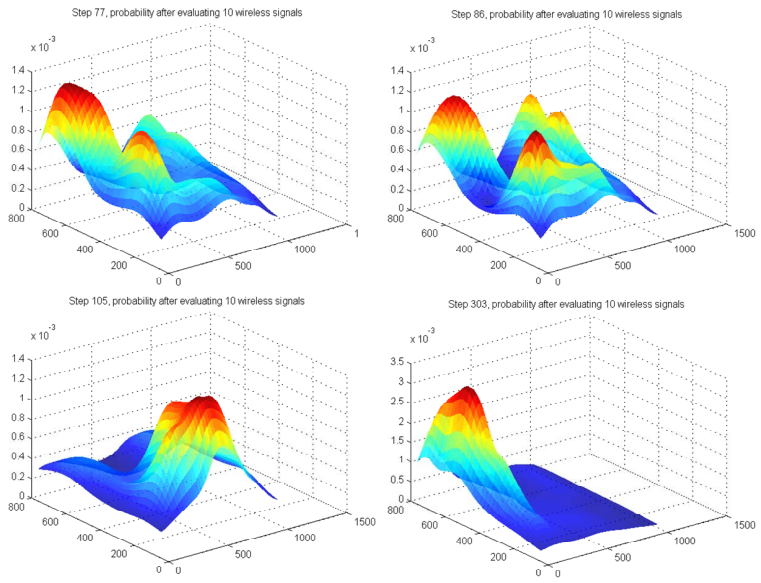


Figure 3.5: Grid-based method to approximate a continuous PDF. Shown here are four time steps reconstructing a walk around Everitt Lab, using sensor data from Wi-Fi APs. The peaks represent higher probability.

A robot that can only translate in a straight line may be adequately approximated with 100 positions. To represent the same level of precision in two dimensions requires 10 000 grid cells (100^2). At 10 dimensions (100^{10} grid cells) the problem is currently intractable for DSP based robots like the SegMonster. Each grid cell requires memory to store and takes time to propagate from one step to the next.

2. *Near zero probabilities*

   Most grid cells at any time have probabilities near zero. Propagating these probabilities with limited resolution datatypes (such as floats or doubles) often runs into problems where the probability is rounded to zero. Also, every grid cell needs to be propagated each time step, even if its probability is negligible. A more efficient method would only keep track of probable grid cells, and not compute the rest of the grid. The particle filter, discussed next, does represent only the probable positions.

3. *Limited resolution*

   The size of the grid cell places a fundamental limit on the resolution of the localization. The system must assume the pose is in the center of the grid cell, since we have no way of representing being in the "upper-right corner" of any given cell.

The particle filter, discussed next, does represent only the probable positions. This efficiently deals with the curse of dimensionality, and allows datatype level resolution.

3.7 Approximating a Probability Distribution Function with Samples

Sampling from a distribution uniformly can approximate any probability distribution. Figure 3.6 shows a contour plot and a corresponding scatter plot of samples drawn from the same probability distribution. Increasing the number of samples decreases the expected error at a rate proportional to $1/\sqrt{n}$ [20].

3.8 Particle Filter

In a particle filter, samples represent the probability distribution. Each sample is called a particle and holds one state estimate. For this localization example, each particle holds one pose as shown in Eq. (3.15). The array $\mathbf{p}_t$ holds all the particles for time t, and represents a probability distribution.

$$\mathbf{p}_{[i]t} = \begin{bmatrix} x_{i,t} \\ y_{i,t} \\ \theta_{i,t} \end{bmatrix}. \tag{3.15}$$

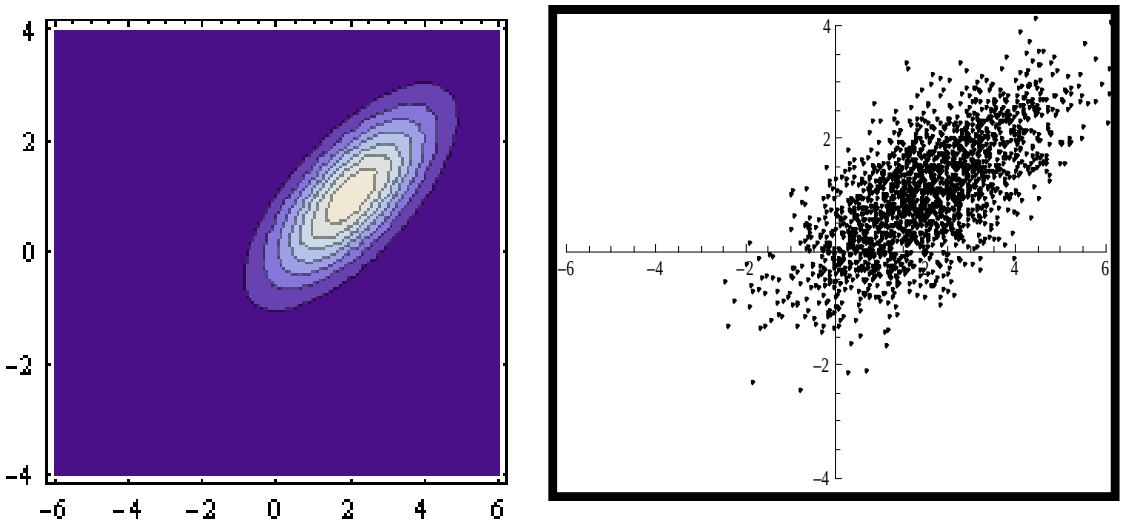


Figure 3.6: Approximating a pdf by sampling. Both images come from the same two-dimensional probability distribution (in this case, a bivariate normal distribution with mean $[2, 1]'$ and variance $[[2, 1]', [1, 1]']$). The left image is a contour plot and the right was made by sampling 2000 points from the distribution. As the number of samples increases, the approximation becomes more exact.

3.8.1 Design of the particle filter

Like the Kalman filter, the particle filter has two stages. In the first, the process model plus additive process noise are used to predict the next $\mathbf{p}_t$. Unlike the Kalman filter, the particle covariance data is propagated implicitly within the particles. Algorithm 2 shows how to calculate the means and covariance statistic from a set of particles. Calculating the means for x and y is trivial, but the mapping for θ is discontinuous at 0 and π. To deal with this issue, we use the function `atan2(y,x)`, the *two-argument arctangent function* that correctly returns θ. The function `atan2(y,x)`is defined for all $(x, y) \neq (0, 0)$ such that

$$\cos\theta = \frac{x}{\sqrt{x^2+y^2}} \quad \sin\theta = \frac{y}{\sqrt{x^2+y^2}}. \tag{3.16}$$

The function *sample_from_motion_model* takes a particle from the previous time step and propagates it according to the control variable with random noise. This function

Algorithm 2 Calculate_particle_mean_and_covariance($\mathbf{p}_{[i]t}$)

1: $x = y = S_{sin} = S_{cos} = 0$
2: $i = 0$
3: **repeat**
4: $\quad x = x + \mathbf{p}_{[i]t}[x]$
5: $\quad y = y + \mathbf{p}_{[i]t}[y]$
6: $\quad S_{sin} = S_{sin} + \sin(\mathbf{p}_{[i]t}[\theta])$
7: $\quad S_{cos} = S_{cos} + \cos(\mathbf{p}_{[i]t}[\theta])$
8: **until** $i > m$
9: $\bar{x} = \mathrm{x/m}$
10: $\bar{y} = \mathrm{y/m}$
11: $\bar{\theta} =$`atan2`(S_{sin}, S_{cos})
12: $S_{XX} = S_{YY} = S_{XY} = 0$
13: $i = 0$
14: **repeat**
15: $\quad S_{XX} = S_{XX} + (\mathbf{p}_{[i]t}[x] - \bar{x})^2$
16: $\quad S_{XY} = S_{XY} + (\mathbf{p}_{[i]t}[x] - \bar{x})(\mathbf{p}_{[i]t}[y] - \bar{y})$
17: $\quad S_{YY} = S_{YY} + (\mathbf{p}_{[i]t}[y] - \bar{y})^2$
18: **until** $i > m$
19: $Cov_{XX} = S_{XX}/m$
20: $Cov_{XY} = S_{XY}/m$
21: $Cov_{YY} = S_{YY}/m$
22: **return** $\langle[\bar{x}, \bar{y}, \bar{\theta}], [Cov_{XX}, Cov_{XY}, Cov_{YY}]\rangle$

creates a new set of particles, $\hat{\mathbf{p}}_{[]t}$, from the old set $\mathbf{p}_{[]t-1}$.

The second stage uses the sensor readings to assign weights, $\mathbf{w}_{[]t}$, to each particle in $\hat{\mathbf{p}}_{[]t}$, with more probable particles receiving larger weights than improbable ones. Next, a new set of particles is constructed, $\mathbf{p}_{[]t}$. Each particle in $\mathbf{p}_{[]t}$ is drawn randomly from $\hat{\mathbf{p}}_{[]t}$, where the particle selected from $\hat{\mathbf{p}}_{[]t}$ is drawn with probability proportional to the weight vector $\mathbf{w}_{[]t}$.

1. *Predict the state and variance*

$$\begin{aligned} &For\ i = 1\ to\ m \\ &\quad \bar{\mathbf{p}}_{[i]t} = sample\_from\_motion\_model\left(\mathbf{u}_t, \mathbf{p}_{[i]t-1}, map\right). \end{aligned} \tag{3.17}$$

2. *Correct the state and variance based on sensors*

$$
\begin{aligned}
&For\ i = 1\,to\ m \\
&\quad \mathbf{w}_{[i]t} = measurement\_model\left(\mathbf{y}_{[i]t}, \bar{\mathbf{p}}_{[i]t}, map\right). \\
&For\ i = 1\,to\ m \\
&\quad draw\ k\ with\ probability \propto \mathbf{w}_{[k]t} \\
&\quad \mathbf{p}_{[i]t} = \bar{\mathbf{p}}_{[k]t}.
\end{aligned}
\tag{3.18}
$$

This process is illustrated in block diagram form in Fig. 3.7.

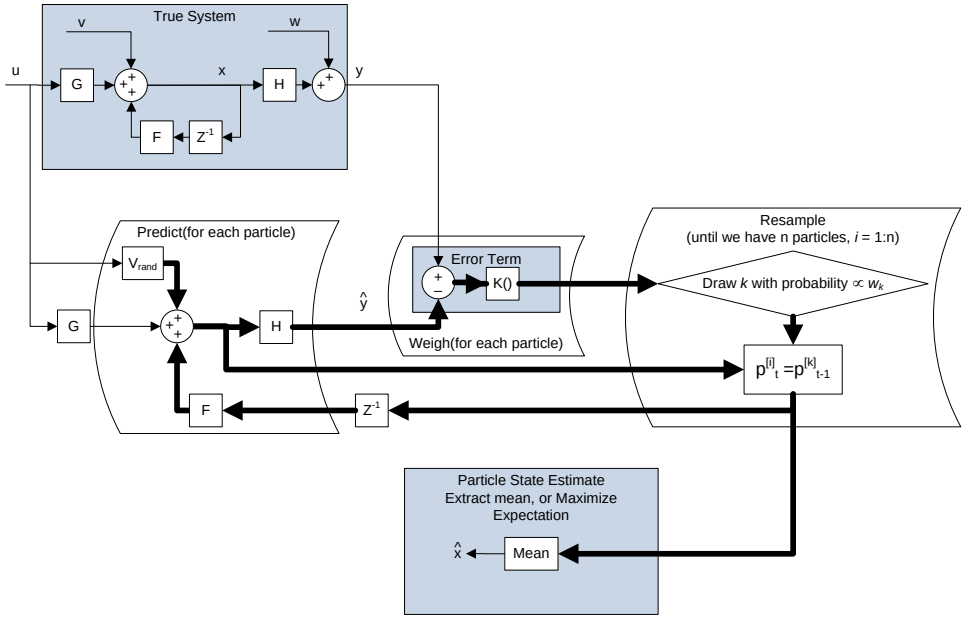


Figure 3.7: Particle filter block diagram. The particle filter has the same structure as the observer, but propagates an array of state estimates.

3.8.2 Simulation results

In simulation the particle filter easily tracks the robot pose using range to landmark data. Figure 3.8 shows that the state estimate converges quickly (within the first 5 steps) to the true location. Just as with the Kalman Filter, the path of the particle

mean is not smooth—if you desired to measure the distance traveled, you would need to run the points through a low pass filter.

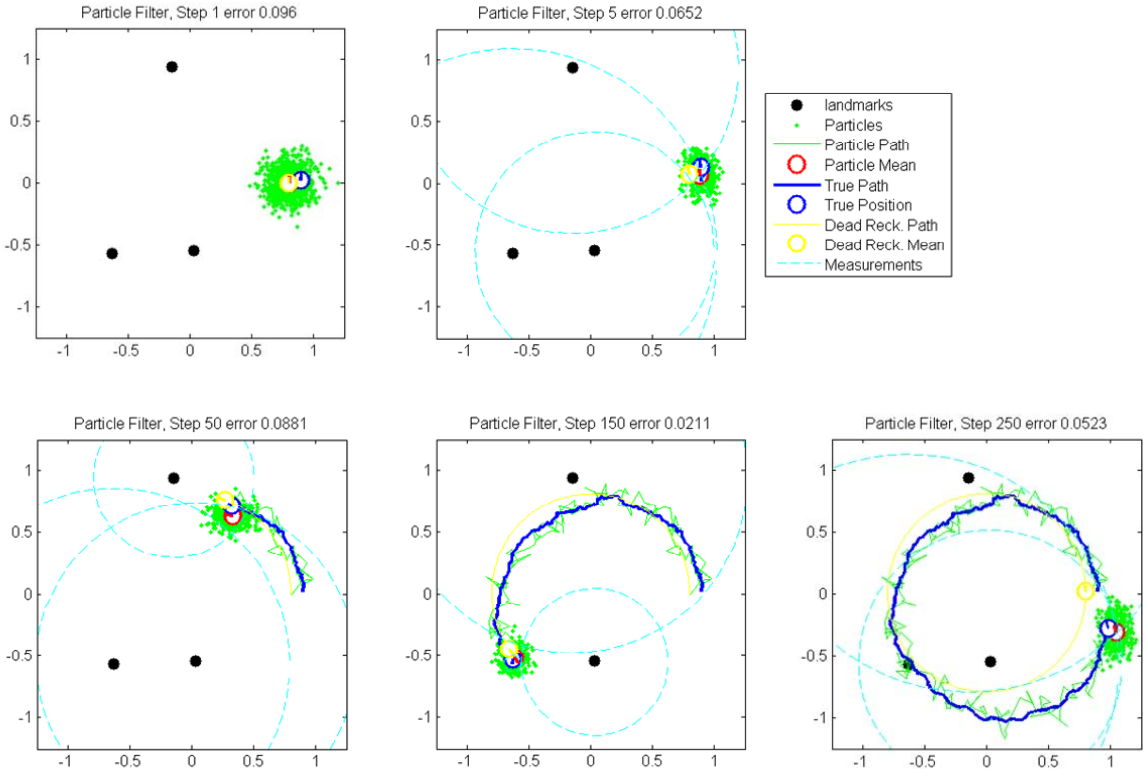


Figure 3.8: Particle filter simulation. The particle filter tracks the true state of the robot with 1000 particles. The particles are shown in green.

3.9 Benefits of the Particle Filter

Unlike the Kalman filter, particles can represent any probability distribution. Figure 3.9 shows a multimodal distribution the SegMonster calculated while it was performing global localization. This distribution cannot be modeled by a Kalman filter.

Particle filters can use nonlinear data, so integrating a map is easy. As a first step, our current pose must be in the free space. We can consult the map and reject any particles that are in obstacles. Figure 3.10 shows a simple map applied to the

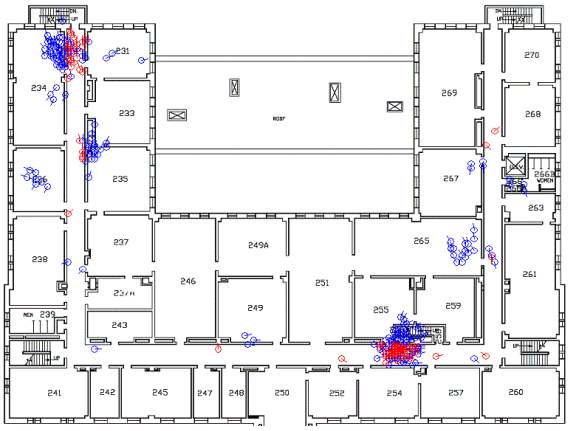


Figure 3.9: Representing a multimodal distribution with particles. Samples can represent a multimodal probability distribution—something the Kalman filter cannot do. In this run of the SegMonster, the robot has almost equal probability of being at the end of the west or south hallway. The mean and covariance of this data would give a wide normal distribution centered in the middle of the roof.

same probability distributions as Fig. 2.1. The map can be used as a binary filter on the data set. Maps can also be used to reject particles based on the path they would have traveled from the previous time step to this time step. If this path at any time intersects an obstacle, it should be rejected. We are currently applying this scheme to a much smaller configuration space in the Robotics Lab. In the lab, the robot is confined to a maze with 0.75-in plywood walls. Modeling these as lines that particles cannot cross is a natural representation and can be computed quickly using line intersection algorithms.

During development on the SegMonster, several other map integration methods were attempted.

1. Sticky walls

   The *sticky walls assumption* truncates any particle path that intersects a wall at the wall boundary. This results in a higher probability mass near walls. Unfortunately, the particle filter on the SegMonster has a slow time constant, and this assumption resulted in the particles being strung out along the hallway

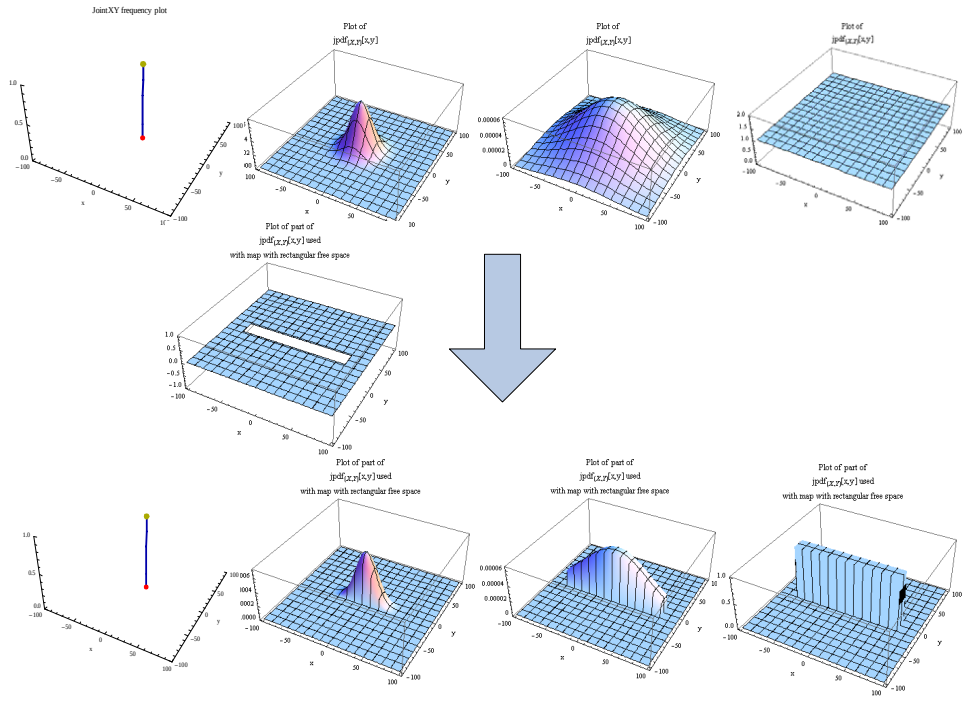


Figure 3.10: Using a map to condition position belief. Top: Position beliefs before conditioning. Center: Map with free space in a long narrow hallway. Bottom: Probability distributions conditioned on the map. A map can be used to limit a probability distribution to the free space. Used in this way, a map is most helpful when the position certainty is low.

behind the true position of the robot as particles hit the walls and stopped. Figure 3.11 shows the disappointing results of this method.

2. Try, try, try... before giving up

   This method iterates on *sample_particle_from_motion_model* until the propagated particle does not collide with the wall or a threshold number of attempts is reached.

   This method worked poorly as well. When the robot reaches the end of a corridor, a high percentage of particles collide with the wall. Resampling on most of these is an exercise in futility and bogs down the particle filter. It also has the disadvantage of placing a higher probability on particles near walls.

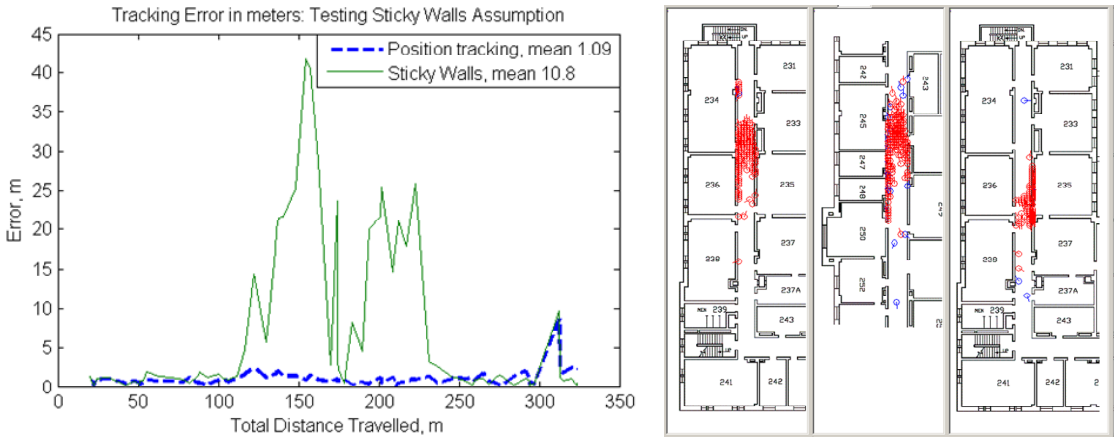


Figure 3.11: Testing the sticky wall assumption. The *sticky wall assumption* behaved poorly in practice. Left: Tracking error compared to zeroing weight if particle collides with wall. Right: Montage showing trails behind the true position of the robot. These grow as particles adhere to the walls.

3. Binary evaluation

   This sets the probabilities of particles not in the free space to 0, but takes a subset of these and places them with uniform probability in the free space of the map. This technique adds extra randomness to the map and is helpful if the particle filter has converged to an incorrect location. In practice the SegMonster adds at most 10 of these random particles each time step. If the particle mass is already at the true position, these extra particles usually are removed in the resampling process because they receive a low weight.

3.9.1 Initializing the particle filter

Both the Kalman filter and the particle filter must be initialized. The particle filter can represent any probability distribution, so its initialization step is more interesting. Two scenarios were tested with the SegMonster. In the first, the probability distribution was initialized to the true pose. This is the position tracking problem. In the second, the robot was given no localization estimate and the initial probability

was uniformly distributed over the map. Figure 3.12 shows the initial uniform distribution. Figure 3.13 shows one time step from an experimental run in the hallway. The two stages of prediction and correction are clearly visible. The robot converged to the true position within the first 30 s, but took longer to establish the correct orientation.

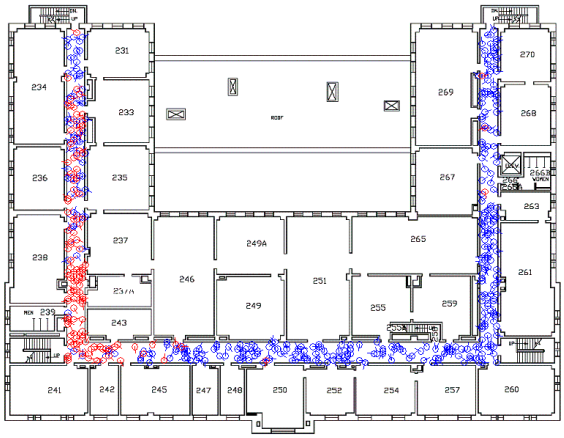


Figure 3.12: Starting the particle filter with a uniform distribution. A uniformly random distribution in the free space represents starting with total uncertainty about position. In the first step the distribution was initialized (blue). A global sensor was used to weigh the particles. Particles in red were selected at least once in the resampling stage.

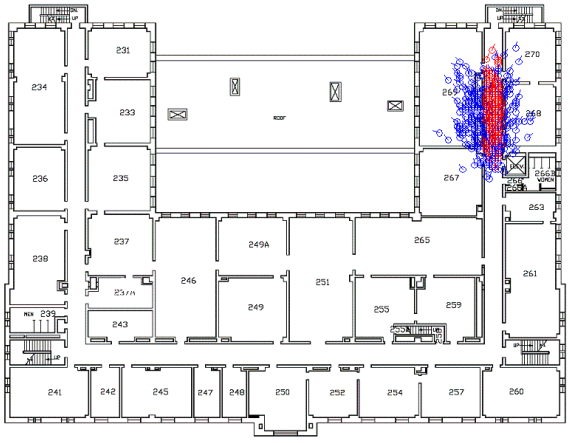


Figure 3.13: Particle filtering: 1 step. Here a distribution is propagated (blue). Global sensors are used to weigh the particles, along with the map, and then the distribution is resampled (red).

3.9.2 Time and memory constraints

The Kalman filter must propagate only the mean and covariance matrix. The particle filter must propagate, weigh, and sample from m particles, taking at least $O(m)$ time. The memory requirements are at least $2m$ because both $\mathbf{p}_{[]t}$ and $\bar{\mathbf{p}}_{[]t}$ must be maintained. The time and memory costs for weighting each sample are application specific. For the SegMonster the weighting step dominates the procedure, and will be discussed in Chapter 4.

3.10 Case Study: Localization on the SegMonster

For this experiment, the SegMonster was given a starting position and asked to track it. The results are shown in Fig. 3.14. This data run is the same one plotted in Figure 2.3. The particle filter kept the average error at 1 m, while the dead reckoning error quickly pushed the dead reckoned estimate off the map.

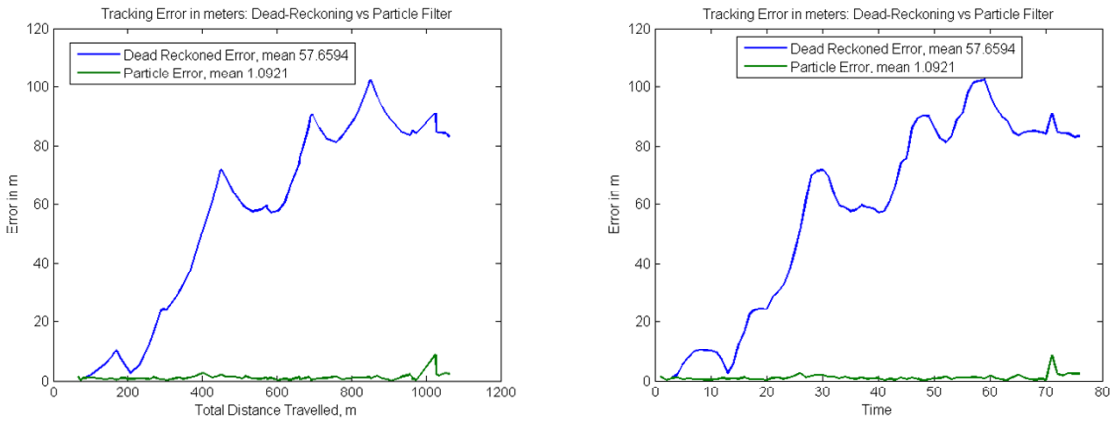


Figure 3.14: Tracking error vs. distance and tracking error vs. time. A plot of position (x, y) error for both dead reckoning and the particle filter estimate, both calculated in real time aboard the SegMonster. 300 particles were used to track the robot.

3.10.1 Motion model

The motion model for our robot is corrupted by noise we assume to be normally distributed. Because our motion model is nonlinear, we cannot simply add this disturbance directly. Algorithm 3 incorporates this uncertainty for each particle. The pose at time t is represented by $pose_t = (x_t, y_t, \theta_t)$. The control is a differentiable set of two pose estimates based on the robot's dead reckoning, $u_t = (\overline{pose}_{t-1}, \overline{pose}_t)^T$, with $\overline{pose}_{t-1} = (\bar{x}_{t-1}, \bar{y}_{t-1}, \bar{\theta}_{t-1})^T$ and $\overline{pose}_t = (\bar{x}_t, \bar{y}_t, \bar{\theta}_t)^T$ [21].

Algorithm 3 sample_motion_model_odometry($u_t, pose_{t-1}$)

1: $\delta_{rot1} =$`atan2`$(\bar{y}_t - \bar{y}_{t-1}, \bar{x}_t - \bar{x}_{t-1}) - \bar{\theta}_{t-1}$
2: $\delta_{trans} = \sqrt{(\bar{y}_t - \bar{y}_{t-1})^2 + (\bar{x}_t - \bar{x}_{t-1})^2}$
3: $\delta_{rot2} = \bar{\theta}_t - \bar{\theta}_{t-1} - \delta_{rot1}$

4: $\hat{\delta}_{rot1} = \delta_{rot1} - \mathbf{sample}(\alpha_1|\delta_{rot1}| + \alpha_2\delta_{trans})$
5: $\hat{\delta}_{trans} = \delta_{trans} - \mathbf{sample}(\alpha_3\delta_{rot1} + \alpha_4(|\delta_{rot1}| + |\delta_{rot2}|)$
6: $\hat{\delta}_{rot2} = \delta_{rot2} - \mathbf{sample}(\alpha_1|\delta_{rot2}| + \alpha_2\delta_{trans})$

7: $x_t = x_{t-1} + \hat{\delta}_{trans}\cos(\theta_{t-1} + \hat{\delta}_{rot1})$
8: $y_t = y_{t-1} + \hat{\delta}_{trans}\sin(\theta_{t-1} + \hat{\delta}_{rot1})$
9: $\theta_t = \theta_{t-1} + \hat{\delta}_{rot1} + \hat{\delta}_{rot2}$

10: **return** $pose_t = (x_t, y_t, \theta_t)^T$

There are four gains on the propagation function $sample\_from\_motion\_model()$. Each gain is proportional to either the measured rotation or translation, and affects either the rotation or translation of the predicted sample. Tuning these gains is very important, as the effect on the probability distributions is significant. The effect of these gains is shown in Fig. 3.15 for the two gains [0.005, 0.001, 0.46, 0.02] and [0.01, 0.005, 0.0016, 0.02]. The first gives a large variance to the translation, while the second gives a noisy rotation estimate. The SegMonster used the propagation gains [0.01, 0.005, 0.16, 0.02]. These were tuned on successive runs. The effect of each gain is listed in Table 3.2.

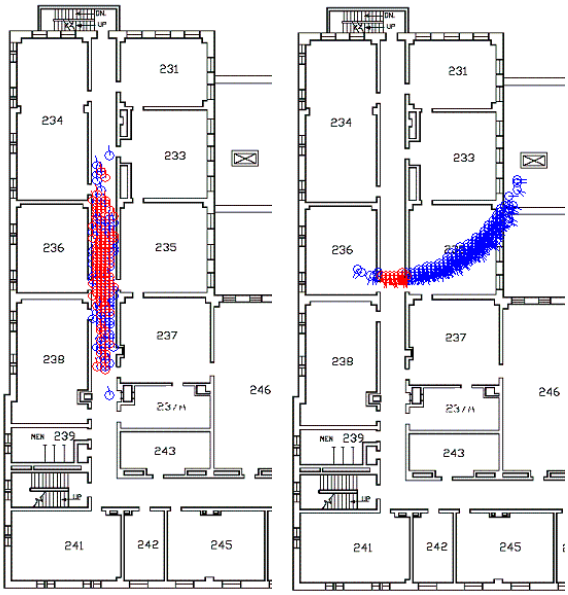


Figure 3.15: Effect of propagation parameters on particle update. Function *sample_from_motion_model*() has four tunable gains, which greatly affect the particle distribution.

Table 3.2: Tunable Gains for Propagation Parameters

Gain	Proportional to	Effects
α_1	rotation	rotation
α_2	translation	rotation
α_3	translation	translation
α_4	rotation	translation

3.10.2 Resampling the particles

After assigning each particle a weight, a new set of particles must be constructed. There are many methods of resampling [20], [21], but all have a similar objective: particles with high weights should have a high probability of being copied into the new array, while particles with low weights should have a low probability of being copied. The SegMonster achieves this by sampling from the *cumulative distribution function* (CDF). The first step is to normalize the weights of the particles in $\bar{\mathbf{p}}$, so that they sum to 1. This turns $\bar{\mathbf{p}}$ into a valid discrete probability space, since all the probabilities are ≥ 0 and the total probability is 1. A plot of the particles with the

probability weight of the particle mapped to the y-axis is a *probability mass function* (PMF). Figure 3.16 shows the PMF and CDF of eight particles.

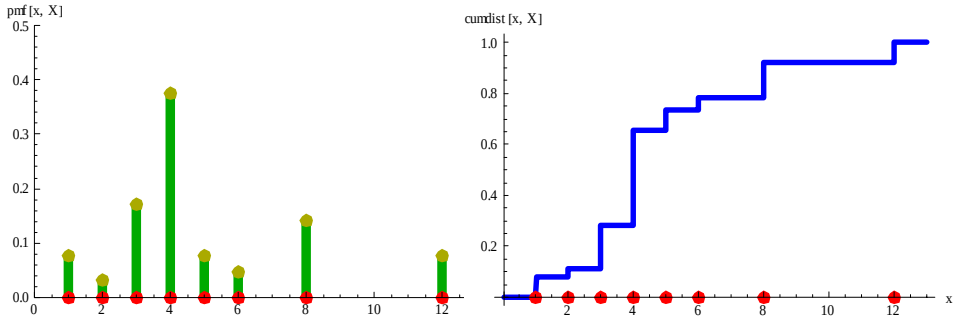


Figure 3.16: A PMF and CDF for eight particles. The probability mass function is on the left and the corresponding cumulative distribution function on the right.

One way to gather the required set of m particles for $\mathbf{p}$ is to generate a random number τ from a uniform distribution on [0,1], and find the last particle whose CDF is less than or equal to τ. By repeating this process a total of m times, $\mathbf{p}$ can be filled. Another way, shown in Fig. 3.17, only requires generating one random number.

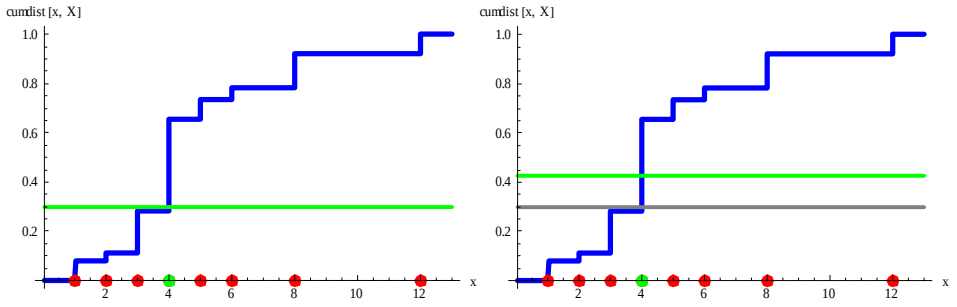


Figure 3.17: Sampling from the CDF. A random number τ is generated from a uniform distribution on [0,1]. Here τ is 0.3. The first particle $\leq \tau$ is selected, and τ is incremented by $1/m$, where m is the total number of particles.

First τ is generated from a uniform distribution on [0,1]. This gives the first particle as before, but now instead of generating a new random number, τ is incremented by $1/m$ so that $(\tau = \tau + 1/m)$. The second particle selected is the last particle

whose CDF is $\leq \tau$. If τ is ever > 1.0, 1 is subtracted from τ and the process continues until m particles have been selected. In Fig. 3.18, the particles selected are $[1, 3, 4, 4, 4, 5, 8, 12]$. Particles 2 and 6 were filtered out by the resampling set, while the most likely particle, 4, was copied three times. The SegMonster uses the second method because it ensures that any particle with probability mass $\geq 1/m$ will not be lost in the resampling stage. Incrementing τ by any number other than $1/m$ removes this property.

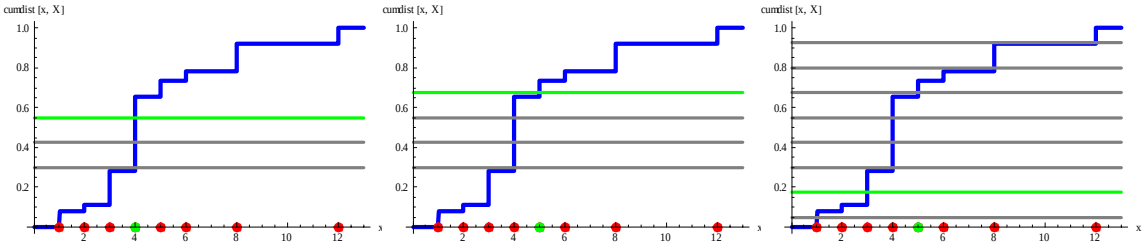


Figure 3.18: Sampling from the CDF (wrapping around at 1.0). To complete the set of new particles $\mathbf{p}$, continue incrementing τ by $1/m$ until the full set of m particles has been collected.

CHAPTER 4

MODELING GAUSSIAN PROCESSES

> "We are coming now rather into the region of guesswork," said Dr. Mortimer.
>
> "Say, rather, into the region where we balance probabilities and choose the most likely. It is the scientific use of the imagination, but we have always some material basis on which to start out speculations." [Sherlock Holmes]
>
> —A. Conan Doyle, *The Hound of the Baskervilles*

4.1 Motivation

The robot localization simulation shown in Figs. 3.4 and 3.8 used noisy range data from uniquely identifiable landmarks. This can and has been implemented. The landmarks used take many forms. Some localization algorithms use artificial beacons as landmarks. One group uses ultrasonic transmitters and receivers that they call crickets to calculate distance from a robot to the transceivers [22]. Other research has focused on using vision algorithms to extract landmark information [23]. The SegMonster uses the strength of wireless access points to determine its position. An *access point* (AP) is a bridge between the wired and wireless networks. These signals cannot be modeled well as a range to landmark sensor, obvious at once from Fig. 4.1.

Figure 4.1: Wireless signal strength in Everitt Laboratory. The contour plot shows the highest received wireless signal strength values from a commercial scan in January 2008.

As Fig. 4.2 shows, a lot of noise corrupts signal strength data. Signal dropout is also an issue. During a 15-min scan along the east hallway of Everitt Lab, 40 unique APs were detected, though never more than 18 at one time and once only four. This data is plotted in Fig. 4.3. A definite increasing trend is visible as the Wi-Fi detector moved from south to north along the 40 m hallway, but at first glance nothing else is apparent.

Determining an agent's location in $\mathbb{R}^n$ from a scalar sensor reading y is called *signal-strength based localization*. This is a nontrivial problem because the mapping from $\mathbb{R}^n$ to y is noninvertible. Indoor localization is even more difficult due to an array of challenges. Among these are the effects of signal interference; signal dropout due to walls, doors, and humans; and the expense of data collection. Signal interference is nontrivial. According to [24], a single human body can attenuate an AP signal strength by an average of 3.5 dBm. A linear model poorly reflects these intricacies.

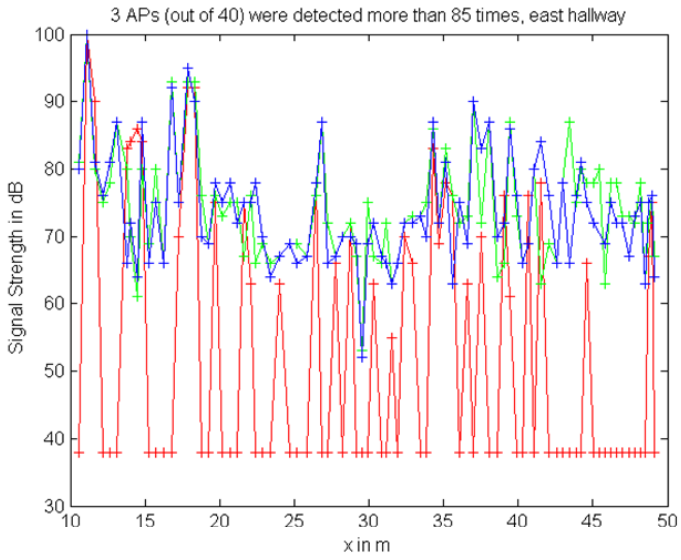


Figure 4.2: Signal strength variation over distance. Forty unique access points were detected during a 15-min scan along the east hallway of Everitt Lab. The signal strength of the signals detected most often does not conform well to a linear model.

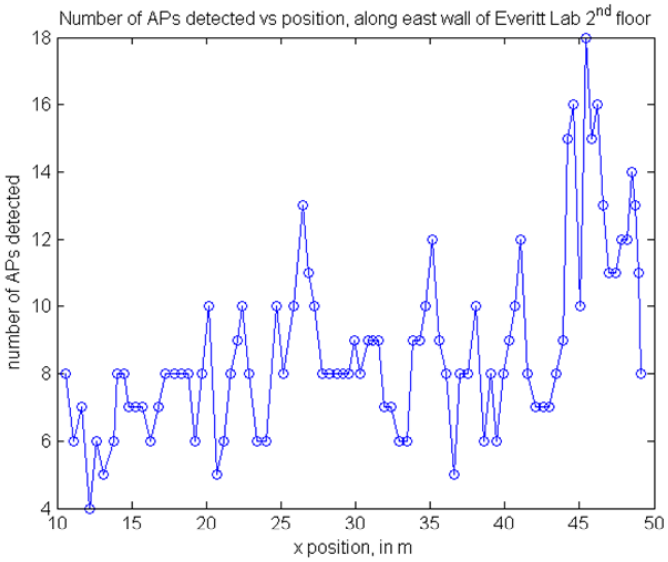


Figure 4.3: Variation in number of APs along a hallway. Forty unique APs were detected. Though a general trend may be interpreted from the data, there is a great deal of noise, even in the number of APs detected. The data was collected over a 15-min period.

Gaussian processes (GPs) are used because they explicitly model both the expected signal strength and the expected signal variance for every point in $\mathbb{R}^n$. GPs model

this relationship through a covariance matrix for all the training values.

4.2 Wi-Fi Signal Strength Data Collection

First, signal strength data must be gathered for different locations on the map. Gathering 4 596 data points, in the form $Raw\_Data = [x, y, AP_{name}, SignalStrength]$, took 2 h.

The Wi-Fi detector on the SegMonster, Fig. 4.4, is mounted above the right handle bar. In order to enforce a uniform bias on the signals, during data gathering the Wi-Fi detector was placed at the same height on a rolling desk chair. The chair was oriented to the north for the entire process.

Figure 4.4: The Wi-Fi sensor used by the SegMonster.

Figure 4.5 shows the setup used for gathering wireless data. A Visual Basic (VB) program was designed to communicate with the Linux processor that ran the wireless sensor. A screenshot is shown in Fig. 4.6. The application runs a map of the building. Clicking on any point of the map prompts the program to request AP data from the Wi-Fi device, and saves [$time$, x_{coord}, y_{coord}, $AP_{MACaddress}$, $SignalStrength$, $SignalNoise$] for each AP returned.

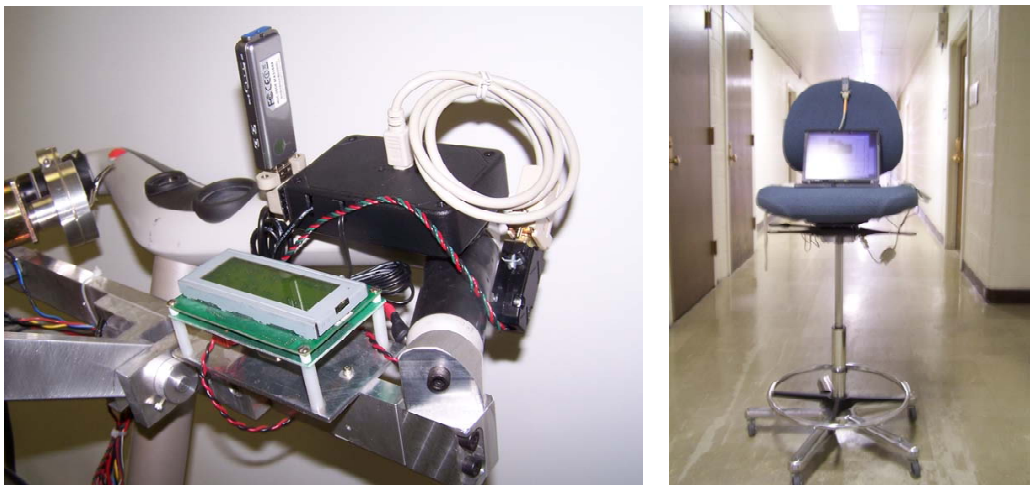


Figure 4.5: Wi-Fi strength data gathering in Everitt Lab. The right image shows the location of the Wi-Fi Detector on the robot. A similar position was arranged for data gathering.

Of the many APs detected during mapping runs, some are easier to find than others. There are three visible APs mounted in the west hallway of Everitt Lab, three in the south hallway, and three more within classrooms on the east wing. Figure 4.7 shows two APs, one mounted in the hallway and another mounted in a conference room.

The scan data was processed to remove APs that appeared less than a threshold of 50 times. This left 32 APs in the data set. The data for each remaining AP was processed to estimate the position of the wireless transmitter, then down-sampled so each AP had 50 evenly spread out samples. These AP locations were stored in an array as $\mathbf{x}_{AP}$. Figure 4.8 shows the data points used for two of the 32 access points.

Finally, GPs assume that the data has a zero mean. Forcing the GP to fit a zero mean approximation to the data would skew our results because Wi-Fi signal strength characteristically decreases with distance from the transmitter. To model this, we fit a linear model to the signal strength, and used this model to predict the sensor

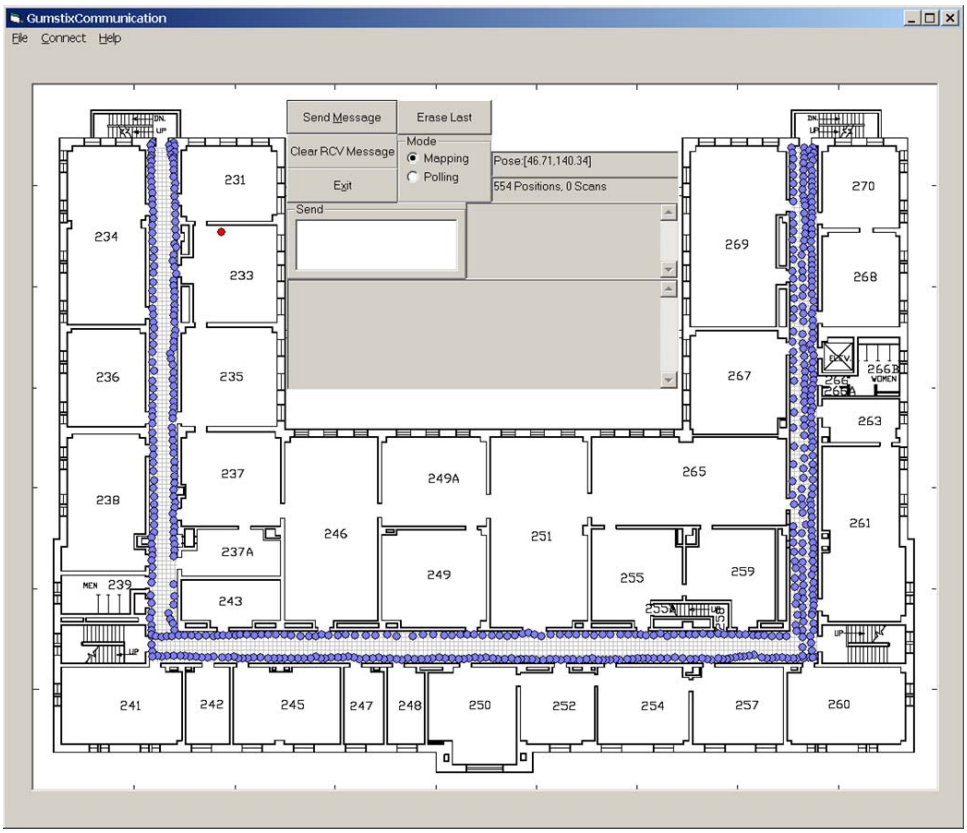


Figure 4.6: VB application used for wireless signal strength mapping. This basic application can be loaded with an arbitrary floor plan image and used to map signal strength.

Figure 4.7: Two access points in Everitt Lab. APs in the hallways are generally protected by metal covers, while those in locked conference rooms are not.

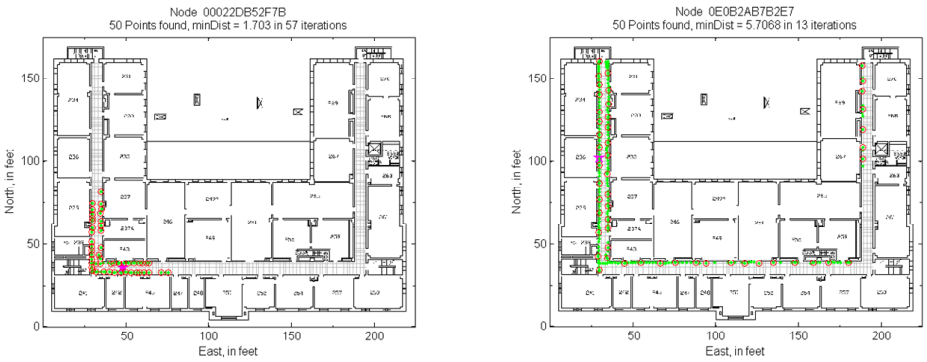


Figure 4.8: Datapoints used to construct GP maps for two APs. Positions where signal strength was recorded for an AP are marked in green, the estimated AP location denoted with a star, and the down-sampled data points in magenta.

measurement $ss_{linear\_prediction}$ at each location.

$$ss_{linear\_prediction} = m\|\mathbf{x} - \mathbf{x}_{AP}\| + b. \tag{4.1}$$

This prediction was subtracted from each signal strength measurement. The resulting signal strength, $\bar{y}_i = y_i - ss_{linear\_prediction,i}$, now has a zero mean.

The MATLAB command `polyfit` can be used to find a linear model of any data set. It fits the data by minimizing the least squares error and returns a vector of coefficients, $[m, b]$, which we stored for each AP. The following code illustrates the method:

```
Distances = sqrt(sum((X-repmat(Model.Xap,size(X,1),1)).^2,2));
coeffs = polyfit(Distances,Model.y,1);
M = coeffs(1);
B = coeffs(2);
Model.LinearFit = [M, B];
```

4.3 Building a Gaussian Process Model

To derive the GP, we followed the function-space view in [3] and [25]. A unique model was constructed for each AP. The first step was to split the down-sampled training data into 32 sets $D_j,\ j \in [1, 32]$. Each $D_j = [(\mathbf{x}_1, \bar{y}_1), (\mathbf{x}_2, \bar{y}_2), \ldots, (\mathbf{x}_n, \bar{y}_n)]$, where $\mathbf{x}_i$ is the datapoint's map coordinate and $\bar{y}_n$ is the recorded signal strength less the linear model. We assume that the data set is derived from a noisy process

$$\bar{y}_i = f(\mathbf{x}_i) + \varepsilon. \tag{4.2}$$

The value ε is additive noise drawn from $\mathcal{N}(0, \sigma_n^2)$, a zero mean Gaussian with known noise σ_n. We stored the $\overline{y}_i$'s in a matrix $\overline{Y}$ and the $\mathbf{x}_i$'s in a matrix $\mathbf{X}$.

The key assumption used by GPs is that function values are correlated. This correlation varies from region to region, but can be described as a function of the input. The GP records this level of correlation in a covariance matrix. When we wish to estimate the function value at a new $\mathbf{x}$, we can use this covariance function to form our prediction. Our covariance equation is the squared exponential

$$k(\mathbf{x}_p, \mathbf{x}_q) = \sigma_f^2 \exp\left(-\tfrac{1}{2\ell^2}|\mathbf{x}_p - \mathbf{x}_q|^2\right). \tag{4.3}$$

The parameter ℓ is the *characteristic length scale* and describes the range of influence one point has over other points. The covariance is proportional to σ_f^2, the signal variance. Since this is a noisy process, we also include the covariance between identical points $k(\mathbf{x}_p, \mathbf{x}_p)$ which is σ_n^2, the signal noise. This makes the covariance of functions

$$\mathtt{cov}\left(\bar{y}_p, \bar{y}_q\right) = k(\mathbf{x}_p, \mathbf{x}_q) + \sigma_n^2 \delta_{pq} \tag{4.4}$$

where δ_{pq} is the Kronecker delta and is 1 if $p = q$ and 0 otherwise. The covariance

for the entire data set can be expressed as the array

$$\mathtt{cov}\left(\bar{\mathbf{y}}\right) = K + \sigma_n^2 I. \tag{4.5}$$

We can use Eq. (4.5) to generate a posterior distribution over functions, given the training data $\mathbf{X}$ and $\overline{\mathbf{y}}$. This posterior over function values is Gaussian and has mean $\mu_{\mathbf{x}_*}$ and variance $\sigma^2_{\mathbf{x}_*}$.

The probability of any given function value $f(\mathbf{x}_*)$ at a location $\mathbf{x}_*$, given the training data, is the probability

$$p(f(\mathbf{x}_*)|\mathbf{x}_*, \mathbf{X}, \overline{\mathbf{y}}) = \mathcal{N}\left(f(\mathbf{x}_*); \mu_{\mathbf{x}_*}, \sigma^2_{\mathbf{x}_*}\right). \tag{4.6}$$

The mean and covariance are functions of the training data

$$\begin{aligned} \mu_{\mathbf{x}_*} &= \mathbf{k}_*^T \left(K + \sigma_n^2 I\right)^{-1} \overline{\mathbf{y}} \\ \sigma^2_{\mathbf{x}_*} &= \sigma_n^2 - \mathbf{k}_*^T \left(K + \sigma_n^2 I\right)^{-1} \mathbf{k}_*. \end{aligned} \tag{4.7}$$

Here $\mathbf{k}_* = k(\mathbf{x}_*, \mathbf{X})$. In practice, the matrix inversion $\left(K + \sigma_n^2 I\right)^{-1}$ is time-consuming. This value is needed twice each iteration of the filter for each particle. Because it is a function only of the training data, it is wise to precompute it once and store the result. $\left(K + \sigma_n^2 I\right)^{-1}$ is a symmetric positive-definite matrix and can be decomposed into LL^T, where L is a lower triangular matrix. This is known as the *Cholesky factorization* and can be computed in MATLAB with the command `L = chol(M)`, where `M` is the matrix to be factored. If we let $\mathbf{L}$ be the Cholesky factorization and set $\alpha = \mathbf{L}^T \backslash (\mathbf{L} \backslash \bar{\mathbf{y}})$, the new algorithm to predict the mean and variance is simplified, as shown in Algorithm 4.

The next step was to estimate the hyperparameters $(\ell, \sigma_n^2, \sigma_f^2)$ by optimizing over the

Algorithm 4 *GP_predict*$(\mathbf{x}_*, \mathbf{X}, \mathbf{L}^{-1}, \alpha)$

1: $\mathbf{k}_* = k(\mathbf{x}_*, \mathbf{X})$
2: $\mu_{\mathbf{x}_*} = \mathbf{k}_*^T \alpha$
3: $\mathbf{v} = \mathbf{L}^{-1}\mathbf{k}_*$
4: $\sigma_{\mathbf{x}_*} = \sigma_n^2 - \mathbf{v}^T\mathbf{v}$
5: **return** $[\mu_{\mathbf{x}_*}, \sigma_{\mathbf{x}_*}]$

log likelihood of the training data, using regression code available at `http://www.gaussianprocess.org/#code`. Finally, a covariance matrix and other intermediate constants were computed and preloaded in the SegMonster.

4.3.1 Using the GP to evaluate particles

The GP provides an expected mean and standard deviation for every point on the map. Figure 4.9 shows the GP maps for one access point. Evaluating the probability of a given measurement y is found by first subtracting the predicted signal strength to create $\bar{y}$. Next, the normal distribution given by the expected mean and standard deviation is evaluated at the point $\bar{y}$. The math for evaluating a normal probability distribution is shown in Eq. (4.8). This returns the probability of the measurement, a scalar value between 0 and 1.

$$p\left(\bar{y}|\mathbf{x}_*\right) = norm\_pdf(\bar{y}, \mu_*, \sigma_*) = \frac{1}{\sigma_*\sqrt{2\pi}} \exp\{-\frac{\bar{y}-\mu_*}{2\sigma_*^2}\}. \tag{4.8}$$

This probability is computed for the k APs detected. By assuming independence, the joint probability of detecting these k signals is the product of each individual probability.

$$p\left(\bar{\mathbf{y}}_{[1:k]}|\mathbf{x}_*\right) = \prod_{i=1}^{k} \left(norm\_pdf(\bar{y}_i, \mu_{i*}, \sigma_{i*})\right). \tag{4.9}$$

This is the weight w for the particle at location $\mathbf{x}_*$.

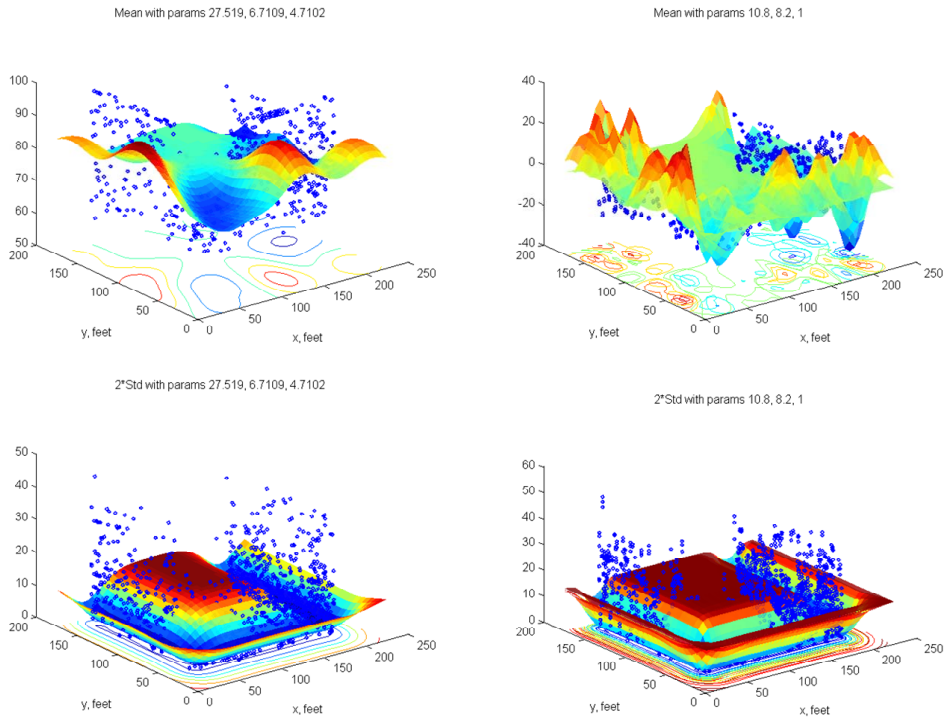


Figure 4.9: GP mean and variance maps for one AP. The GP maps are sensitive to the hyperparameter selection. The figures on the left show the maps used for localization. The figures on the right used a different set of hyperparameters with a shorter length scale ℓ.

4.3.2 Time and memory constraints

While the accuracy of GPs increases with the number of data points, the processing and memory storage loads increase as well. For a particle filter with m particles, if k APs are detected the GP model must be evaluated mk times. If each AP is represented by a data set of n data points, each evaluation takes $O(n^2)$ time. This is due to the $n^2/2$ time for multiplying the triangular system in step 2. The total update time for each step of the particle filter is $O(n^2mk)$. The memory requirements are also nontrivial. For each AP on our map, the system must store the constants listed in Table 4.1, which is dominated by the n^2 matrix L^{-1}. We chose to limit the size of the arrays to 50. We detect an average of 9 APs each scan. Just multiplying the L^{-1} matrix requires $50^2 \cdot 200 \cdot 9 = 4\ 500\ 000$ floating point multiplications, which

is a computational load on our small processor.

Table 4.1: Memory Requirements per AP for Gaussian Process Evaluation

Variable Name	Description	Size/type
AP_{MAC}[1]	MAC address	13 char
$numPoints$	Sample set size = n	1 int
X	sample (x, y) locations	$n \times 2$ float
y	sample signal strength	n floats
Xap	AP (x, y) location	2 floats
m	slope of linear fit	1 float
b	intercept of linear fit	1 float
$\sqrt{\ell}$	length scale hyperparameter	1 float
σ_f^2	signal variance hyperparameter	1 float
σ_n^2	observation noise hyperparameter	1 float
L^{-1}	precomputed Cholesky factorization $(K + \sigma_n^2 I)^2$	$n \times n$ float
α	precomputed matrix $L^T \backslash (L \backslash \hat{y})$	n floats

[1] Though there are 8^{12} possible MAC addresses, in practice there is considerable overlap. Our model treats APs with the same MAC address as identical.

4.4 Procedure

After collecting the signal strength data, the process to build the GP maps is largely automated. Figure 4.10 lists the procedure and the program code for each step. The MATLAB code *GP_makeKLMfiles.m* generates a C file with all the GP constants needed for each AP.

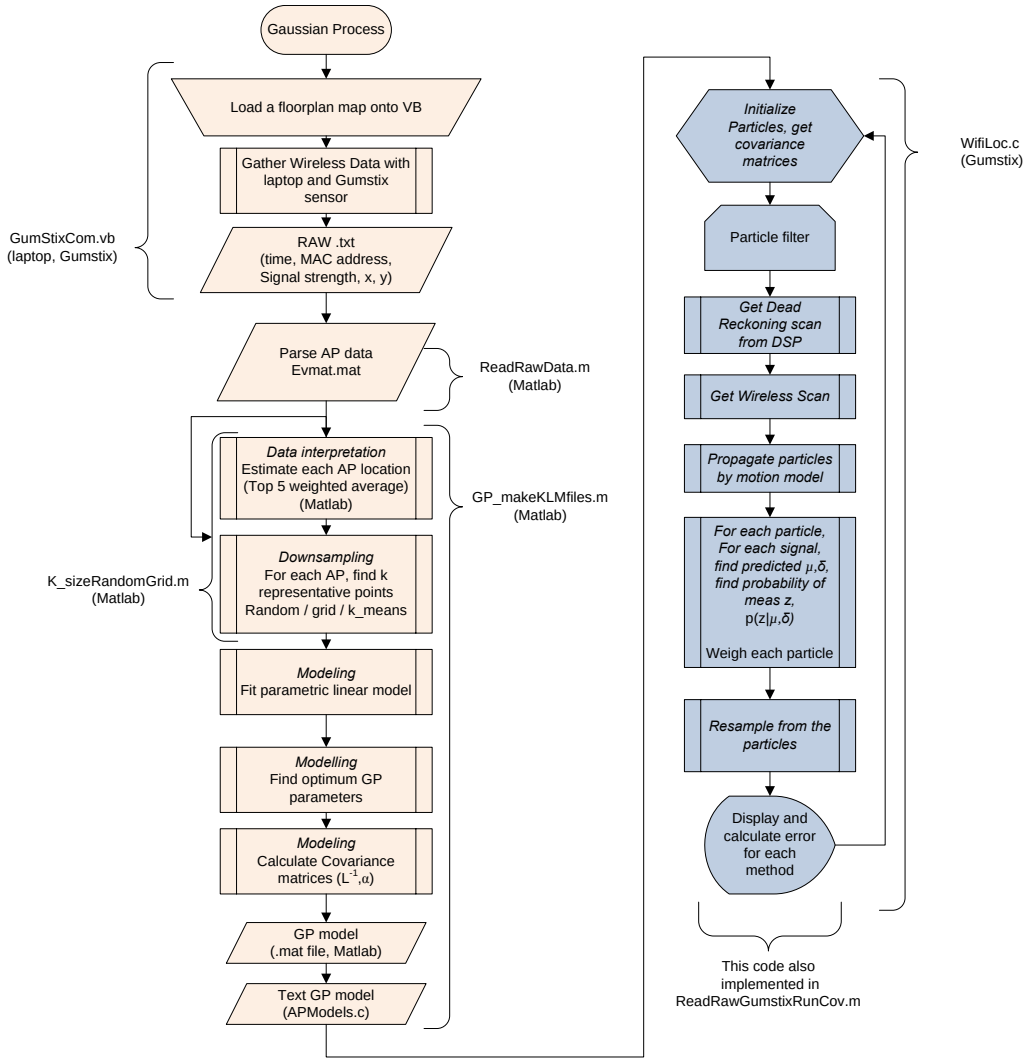


Figure 4.10: Flow chart for Gaussian process localization. Left: Building the map. Right: Using Gaussian processes to weight samples in a particle filter. All the programs referenced are included in the Appendix.

CHAPTER 5

THE SEGMONSTER

Q. Can your robot go down stairs?

A. Only once.

—Anon

5.1 Why a *Segway*?

Our buildings, sidewalks, hallways, and classrooms are built for humans. A humanoid robot, meant to interact and navigate in this environment, must meet certain specifications [26]. Specifically, a humanoid robot must have a small footprint in order to navigate cluttered environments built for humans. In order to manipulate and sense in these environments, the robot needs the ability to place its center of mass high above the ground. These two requirements conflict for statically stable wheeled robots. A humanoid robot must be dynamically stable. Both bipedal legs and dynamically balancing wheel robots meet these criteria. Currently, bipedal walkers are expensive, inefficient, or nonrobust. Dynamically balancing wheeled robots are an attractive alternative. Segway has recognized this market and sells a Segway Robotic Mobility Platform (RMP). The RMP is a balancing scooter engineered to serve as a mobile robot. Unfortunately, its purchase price has prevented widespread adoption of the RMP. The SegMonster (Fig. 5.1) was designed and built as a cost-effective

alternative by David Johnson and Jan Vervoorst [27]. Figure 5.2 illustrates the maneuverability of the Segway platform.

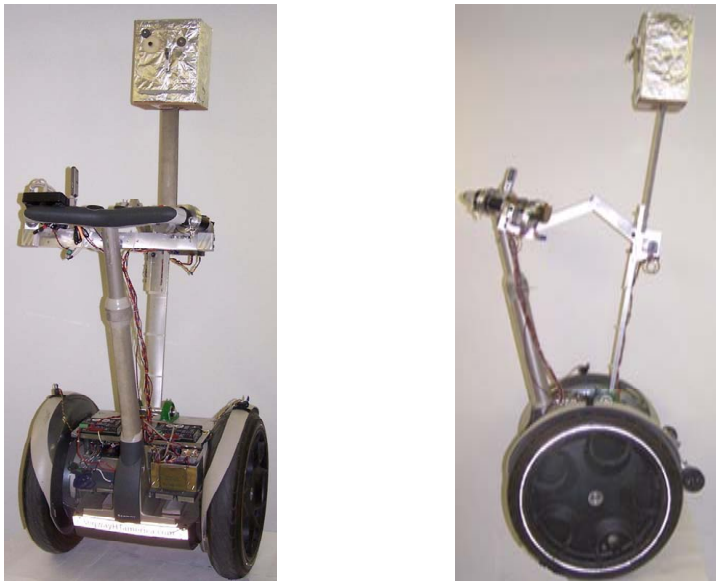


Figure 5.1: Two views of the SegMonster. At 1.7 m (5-ft, 6-in), the SegMonster is a human-sized robot capable of turning in place and moving at 16 km per hour (10 mph).

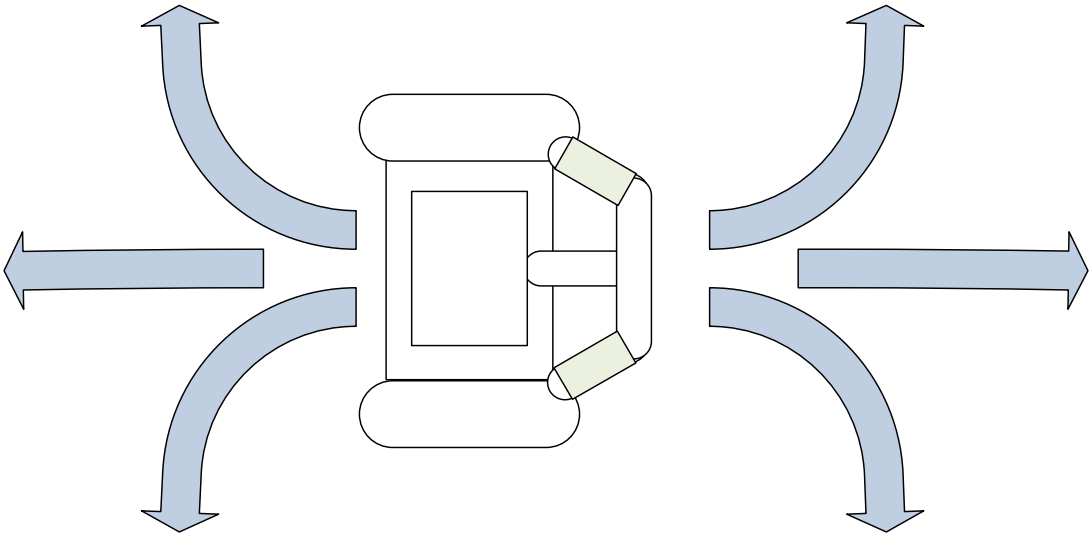


Figure 5.2: Top-down view of Segway. The Segway platform can turn on its own footprint and accelerate in the forward and reverse directions.

5.2 Controlling the SegMonster: Dynamics

Any robot becomes easier to control by crafting a more accurate model of the system. Segways, built as a human transportation solution, are two-wheeled, feedback-stabilized scooters capable of carrying a 96 kg (210 lb) rider at speeds of up to 16 km/h (10 mph). Segways are notable for their small footprint, only 0.6 m wide by 0.5 m deep (24×18 in). Their floor contact area is even smaller due to using only two wheels. Despite this small base, they have remarkable stability, achieved by feedback control [28].

The addition of a rider to the Segway completes an inverted pendulum system. The rider is the pendulum, and the Segway applies control to keep the rider in an upright position. We orient the x-axis to be forward for the Segway, the z-axis to be perpendicular to level ground, the y-axis to extend through the midpoint of each wheel to the left of the Segway, and θ to be the angular difference from vertical to the pendulum center of mass COM.

The Segway runs a control loop that senses the platform tilt 100 times a second (a Δt of 10 ms) and powers the wheels to keep the platform beneath the rider at all times. According to their manual, "The Segway HT works like the human body. If you lean forward, you take a step forward to keep your balance. If you lean back, you step back. Substitute wheels for feet, and you have an empowered pedestrian on a Segway HT" [28].

A human rider can control the Segway's velocity in the x-axis by changing the COM position in the x-axis. Gravity pulls in the negative z-direction, which produces a torque on the Segway. If the Segway counteracts this torque with an acceleration, it

can remain upright. The Segway has a maximum velocity of 16 km/h, so it cannot maintain any nonzero acceleration indefinitely. To counteract this, when the Segway nears its velocity limits it over-accelerates. The front of the base lifts up and the handlebars "push" the rider back. The manufacturers call this effect the *speed limiter*. This riding-up provides an intuitive warning to the user and forces them to slow down. The Segway is also designed to provide audio, visual, and tactile cues when the rider is near falling. The large backlit LED switches from green to red, the system beeps, and the Segway shakes its handle while making growling noises.

5.2.1 Controlling angular velocity

The Segway turns by giving different speed commands to each tire motor. In this way, it can turn over its own footprint, or make gradual turns. The Segway turning speed is controlled by twisting the left handle grip, as shown in Fig. 5.3. The SegMonster has a servo motor cabled to this twist grip.

The original SegMonster controller regulated turning by treating the Segway handle as a black box, and applied a PI control on the error from a desired turning reference. This led to a characteristically oscillatory control. As Fig. 5.3 shows, there is a linear relationship between twist angle and angular velocity in both the forward and reverse directions, but there is a substantial *dead zone*. A dead zone is a nonlinearity that can be removed in code, producing a linear relationship between turn command and angular velocity. Compensating for this improved the SegMonster's response to velocity commands, as seen in Fig. 5.4.

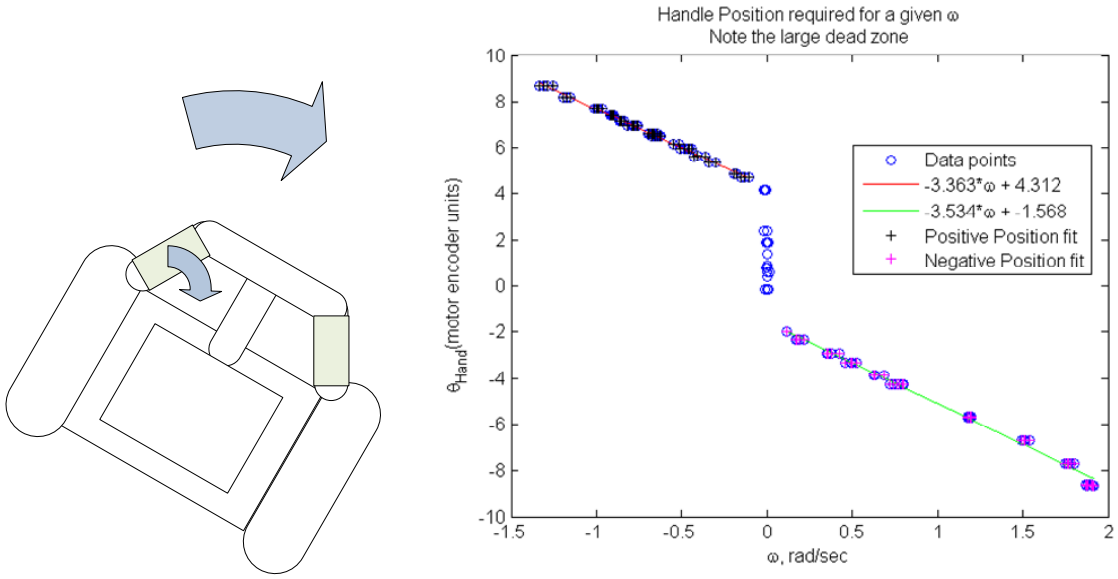


Figure 5.3: Turning the Segway. The Segway turning speed is controlled by twisting the left handle grip. As the plot on the right shows, there is a linear relationship between the twist angle and angular velocity for positive and negative rotation.

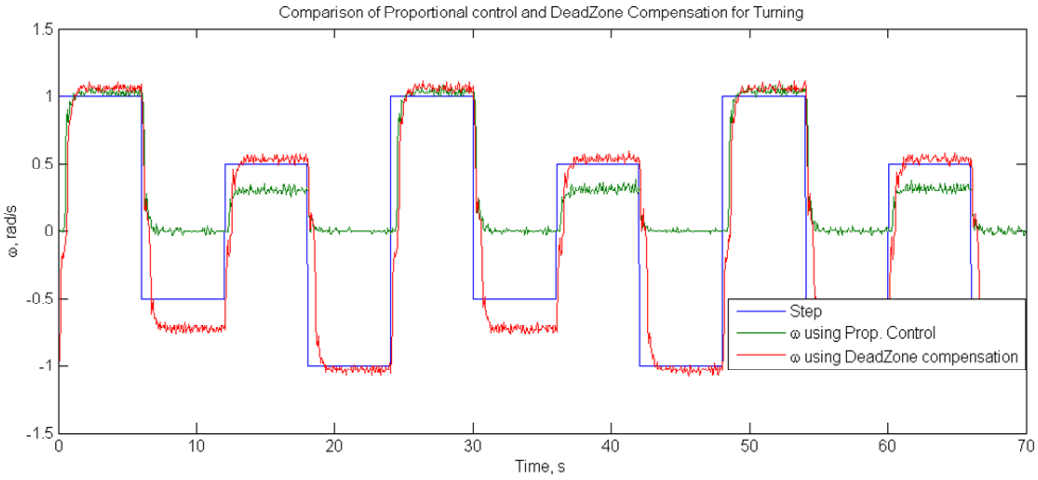


Figure 5.4: Dead zone compensation for turn command. The plot shows angular acceleration response to a pulsed input. Compensating directly for the dead zone is more effective than a proportional gain.

5.2.2 Controlling speed

By modeling the relationship between the wrist motor and angular speed in the controller, the SegMonster's turning control was greatly improved. If a similar step could

be designed for the velocity control, the SegMonster would move more fluidly. Figure 5.5 shows how the SegMonster controls velocity by leaning forwards or backwards.

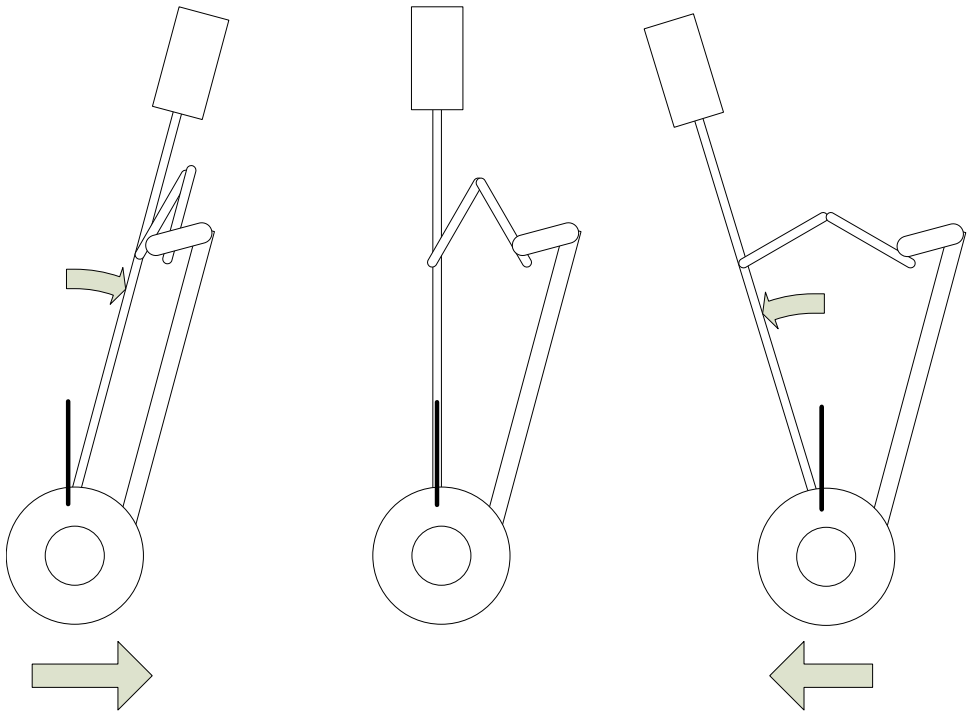


Figure 5.5: Speed control of the SegMonster. The SegMonster controls the Segway's speed in the same way a human rider would, by shifting its center of gravity over the platform. When the Segmonster leans back, the platform accelerates in the negative direction, and it accelerates in the positive direction when the SegMonster leans forward.

Figure 5.6 shows data gathered of speed versus body position. It is obvious that a linear relationship between body position and speed does not exist. To improve our control the SegMonster's speed we will need a better model.

5.3 System Model

A simple model for the SegMonster is a cart-mounted inverted pendulum. Figure 5.7 shows this model, and the relevant variables are listed in Table 5.1.

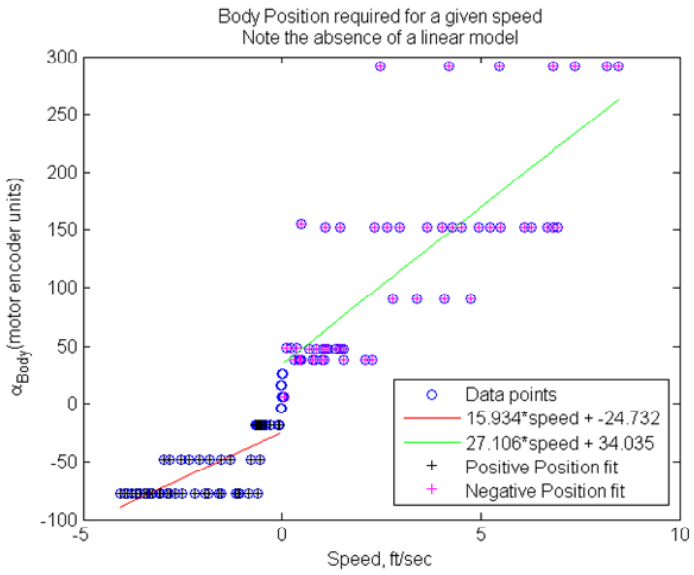


Figure 5.6: Body position vs. velocity. Sadly, there is not a linear relationship between body position and speed. Any positive position can cause any positive velocity, and vice versa.

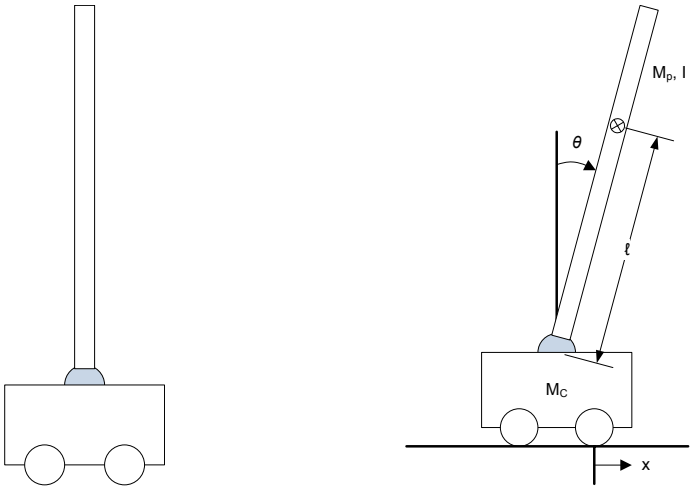


Figure 5.7: The cart-mounted inverted pendulum. Left: The pendulum at equilibrium, $\theta = 0$. Right: The pendulum perturbed from equilibrium.

5.3.1 Calculating COM and inertia

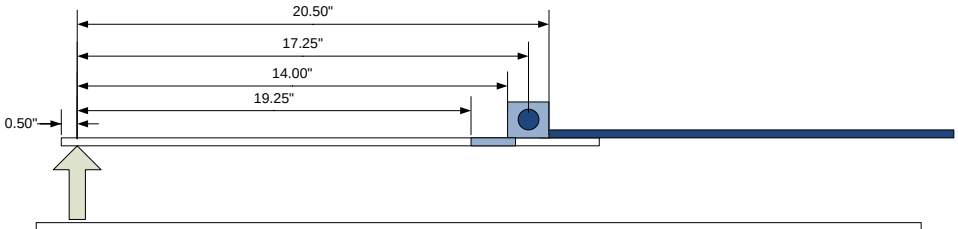
The inertia of the SegMonster is found by integrating

$$I_{zz} = \int_{z_{low}}^{z_{high}} \int_{-b/2}^{b/2} \int_{-a/2}^{a/2} \left(x^2 + y^2\right) \rho \, dx \, dy \, dz$$

Table 5.1: Cart-Mounted Inverted Pendulum Variables

Parameter	Description	value/range
θ	angle from vertical to pendulum COM	$[-15°, 26°]$
ℓ	length from pivot to pendulum COM	0.67 m (26.29 in)
M_c	mass of cart	68 kg (150 lb)
M_p	mass of pendulum	5.1 kg (11.32 lb)
I	inertia of pendulum	1.219×10^3 kg·m^2 (4.1659 lb·in^2)
x	linear displacement of cart	in m

for each component in the body. This integration is considerably simplified because all the components are rectangular. Here a and b are the thickness and width of the component, z_{low} and z_{high} are the min and max distances from the pivot point, and ρ is the density of the material. Steel has a density of 7.87 g/cc (0.284 lb/in^3) and aluminum has a density of 2.6989 g/cc (0.097504 lb/in^3). Using the parameters listed in Fig. 5.8, the I_{zz} is 1.219×10^3 kg·m^2 (4.1659 lb·in^2). The simplified body is shown in Figs. 5.9 and 5.10.



Part	Material	Dim (x,y,z),in	Mass (lb)	Icon
Body Top	Steel	25.25, 2.0, 0.5	7.17	
Body Bottom	Aluminum	21.0, 2.0, 0.5	2.05	
Motor Mount	Aluminum	3.75, 0.5, 2.75	0.50	
Arm Stop	Aluminum	4.50, 0.5, 0.75	0.16	
Motor	Assorted	--	1.43	

Figure 5.8: Arm components for inertia calculation. The body is made of a heavy steel plate attached to a lower structure made of aluminum.

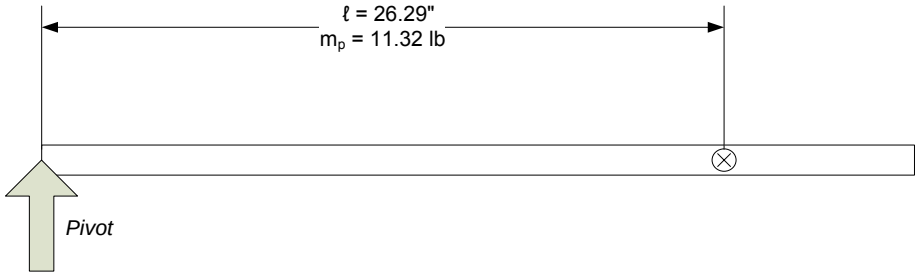


Figure 5.9: Simplified version of the body for controller.

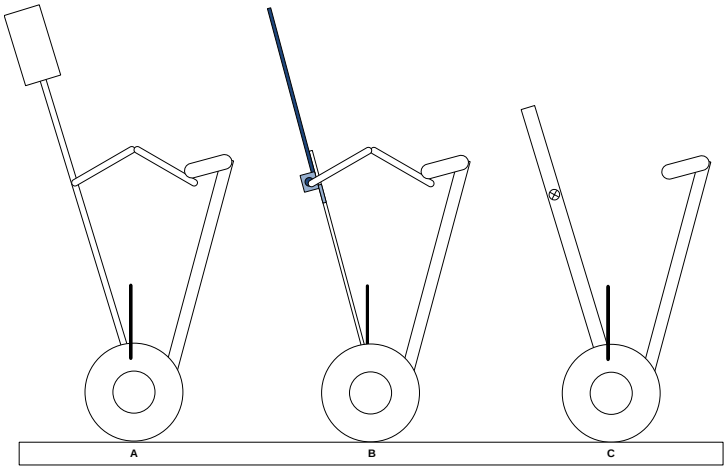


Figure 5.10: Progressive simplification of the SegMonster body. By calculating the mass and inertia of the entire SegMonster body, we can simplify the control calculations.

5.3.2 System dynamics

Using the *Newtonian method*, we can find the equations of motion by summing the forces on each component, starting with the cart. The system's free body diagram is shown in Fig. 5.11. The cart cannot move in the vertical direction, so we will just sum forces on the cart in the horizontal direction:

$$M_c\ddot{x} + b\dot{x} + N = F. \tag{5.1}$$

We can solve for N by summing the forces on the pendulum in the horizontal direction.

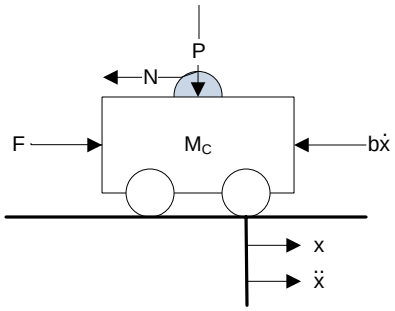


Figure 5.11: Free body diagram of the cart, with forces upon it.

These forces are shown in the free body diagram for the pendulum (Fig. 5.12).

$$N = M_p\ddot{x} - M_p\ell\ddot{\theta}\cos\theta + M_c\ell\dot{\theta}^2\sin\theta. \tag{5.2}$$

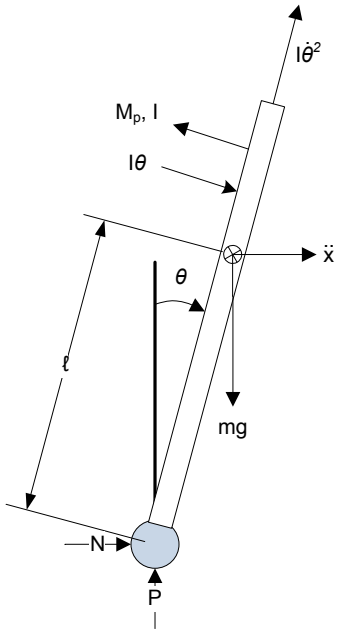


Figure 5.12: Free body diagram of the inverted pendulum and forces upon it.

By substituting Eq. (5.2) for N in Eq. (5.1), we get our first equation of motion:

$$(M_c + M_p)\,\ddot{x} + b\dot{x} - M_p\ell\ddot{\theta}\cos\theta + M_c\ell\dot{\theta}^2\sin\theta = F. \tag{5.3}$$

To solve for the second equation of motion the remaining forces can be summed

perpendicular to the pendulum:

$$-P\sin\theta + N\cos\theta + M_p g\sin\theta = -M_p\ell\ddot{\theta} + M_p\ddot{x}\cos\theta. \tag{5.4}$$

To remove the normal forces, the moments can be summed about the COM of the pendulum, $P\ell\sin\theta - N\ell\cos\theta = I\ddot{\theta}$. This gives

$$-I/\ell\ddot{\theta} + M_p g\sin\theta = -M_p\ell\ddot{\theta} + M_p\ddot{x}\cos\theta. \tag{5.5}$$

The angular accelerations can be grouped, $(-I/\ell + M_p\ell)\,\ddot{\theta} + M_p g\sin\theta = M_p\ddot{x}cos\theta$, and the equation simplified by dividing through by the mass of the pendulum.

$$\left(\frac{-I}{M_p\ell} + \ell\right)\ddot{\theta} + g\sin\theta = \ddot{x}\cos\theta.$$

We can now isolate the horizontal acceleration, $\frac{\left(\frac{-I}{M_p\ell}+\ell\right)\ddot{\theta}+g\sin\theta}{cos\theta} = \ddot{x}$. By exploiting trigonometric identities, the second equation of motion is

$$\frac{\left(\frac{-I}{M_p\ell} + \ell\right)\ddot{\theta}}{\cos\theta} + g\tan\theta = \ddot{x}. \tag{5.6}$$

When $\ddot{\theta}$ is small and $\cos\theta$ large (near the equilibrium) the instantaneous acceleration needed to maintain the pendulum position is

$$g\tan\theta \approx \ddot{x}. \tag{5.7}$$

The approximation is exact when $\ddot{\theta} = 0$. The Segway's control loop provides this acceleration to prevent the system from falling over, if the acceleration is within the Segway's capabilities.

5.3.3 Improving the system model

1. *Adjusting for pivot point offset*

   A few additions to the model are required because the SegMonster's body does not pivot at the center of the wheels, as shown in Fig. 5.13. The SegMonster body is attached slightly back from the center line between the wheels, presumably to counterbalance the handlebars.

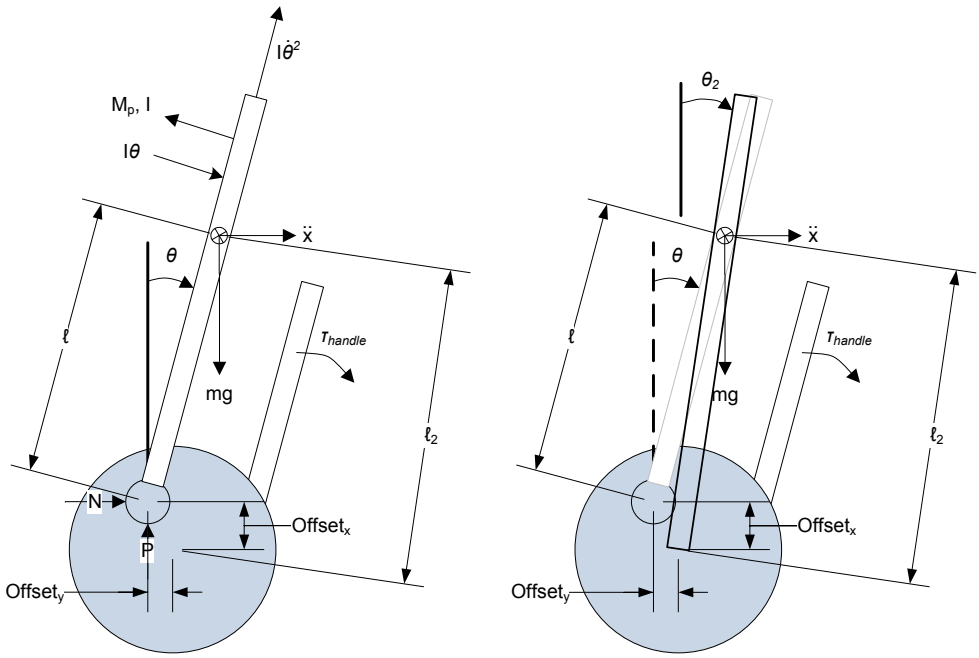


Figure 5.13: Adjusting for pivot point offset. Left: The pivot point for the SegMonster is slightly back from the center line of the robot. Right: a small transformation converts the model position $[\theta_2, \ell_2]$ into body position $[\theta, \ell]$.

The pivot is offset by $offset_x = 5.71$ cm (-2.25 in) and $offset_y = 21$ cm (8.25in). If we define the location of the COM with regard to the offset as (x_2, y_2), the angle from the center of the wheel to the COM as θ_2, and the distance from the COM to the center of the wheel as ℓ_2, the new coordinates are:

$$
\begin{aligned}
x_2 &= \ell \sin\theta - 2.25 \\
y_2 &= \ell \cos\theta + 8.25 \\
\ell_2 &= \sqrt{x_2^2 + y_2^2} = \sqrt{73.12 + \ell^2 + 16.5\ell\cos\theta - 4.6\ell\sin\theta}\,\text{in} \\
\tan\theta_2 &= \tfrac{\ell\sin\theta - 2.25}{\ell\cos\theta + 8.25} \\
\ddot{x} &\approx g\tfrac{\ell\sin\theta - 2.25}{\ell\cos\theta + 8.25}.
\end{aligned}
\tag{5.8}
$$

Fig. 5.14 shows the instantaneous acceleration required of the Segway to maintain the pendulum position. Obviously, this is a nonlinear equation because it requires infinite accelerations as the pendulum reaches a horizontal position. The plot of instantaneous acceleration versus position is shown in Fig. 5.14, along with the linear approximation. Fortunately the SegMonster's body is physically constrained to a small variation in θ. The linear approximation relating the position of the SegMonster's body to the instantaneous acceleration, taking into account the torque from the handlebars and the pivot point offset, is

$$
\begin{gathered}
7.68 \cdot \theta_{(rad)} + 1.06\,\text{m/s}^2 \\
(25.20 \cdot \theta_{(rad)} + 3.48\,\text{ft/s}^2).
\end{gathered}
\tag{5.9}
$$

2. *Converting from motor angle to body position*

   The relationship between motor angle α_B and body angle θ_B can be solved by applying the law of cosines. The relationship is nonlinear, but can be well approximated by the line $\theta_B = -0.324\alpha_B + 55.47°$, with less than 4° error, and that at the endpoints. The relationship and linear approximation are shown in Fig. 5.15.

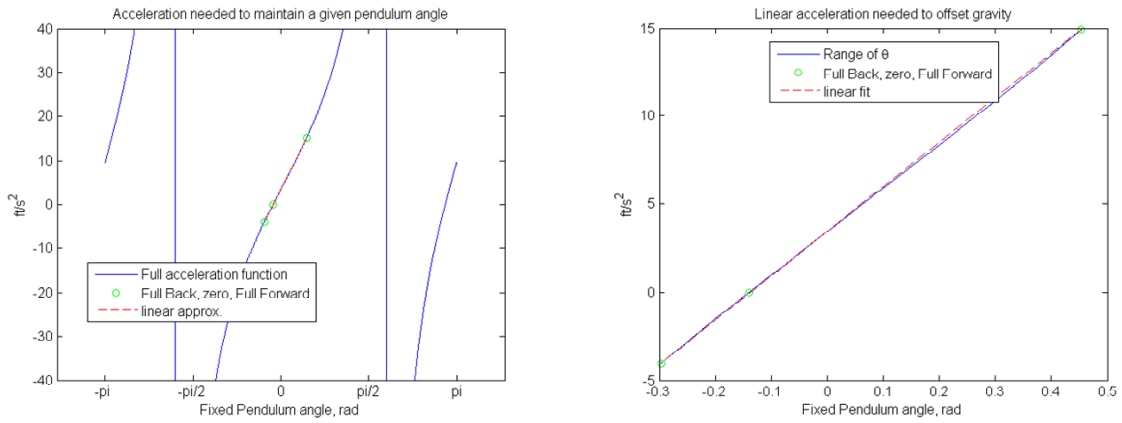


Figure 5.14: Model of acceleration vs. body position. Though the $accel = -g\tan(\theta)$ function grows to $\pm\infty$, the range of our robot position fits neatly on a section that can be modeled linearly.

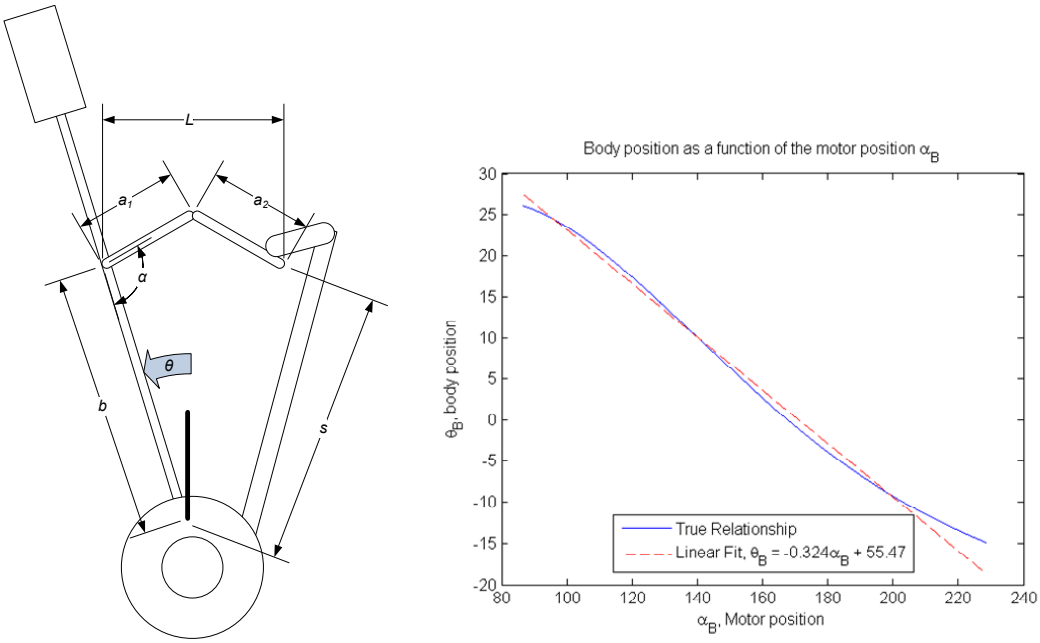


Figure 5.15: Converting from motor angle to body position. The body motor angle α controls the body position θ through a nonlinear relationship. Fortunately for ease of control, this relationship can be well approximated by a first order fit.

5.3.4 Speed control revisited

The RWP connects directly into the Segway's CAN bus and controls the wheels directly [29]. This results in a non-minimum-phase system. When the robot wants

to move forward from a balancing position, it must first accelerate in the reverse direction to tip the COM forward before accelerating in the desired direction. This response can be clearly seen in Fig. 5.16, where the RMP responds to a 6-s pulse between 0 and 1 m/s.

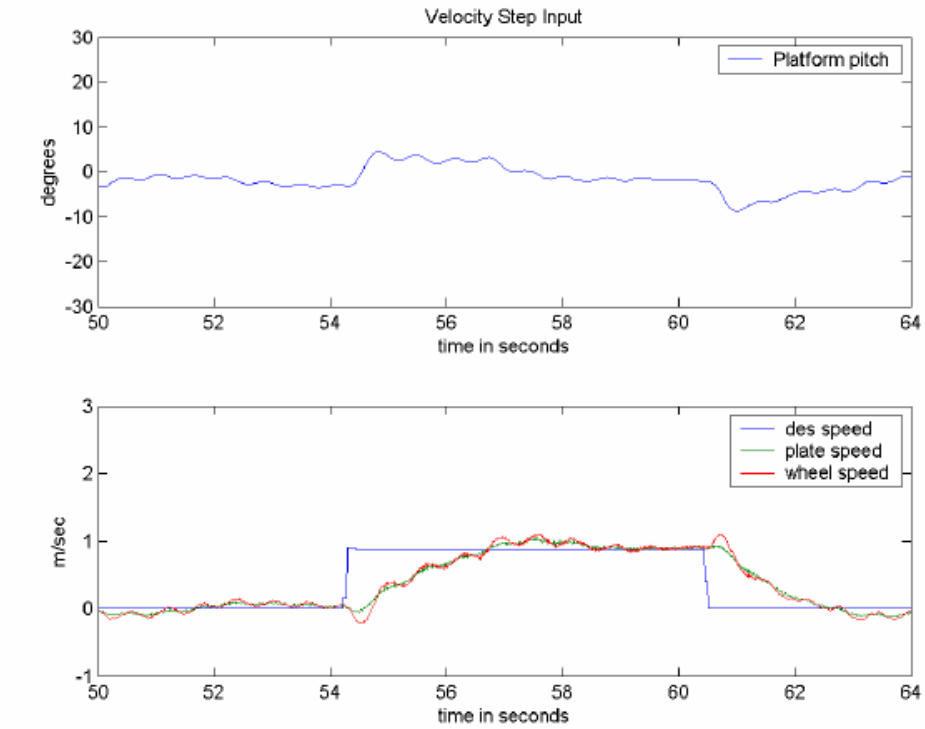


Figure 5.16: RWP response to a step velocity input. The RMP is a non-minimum-phase system, and so must accelerate backwards before going forward. This is much like backing up a truck with an attached trailer. The initial control input must be opposite of the desired direction.

The SegMonster is able to tip its body in the desired direction under power, and the Segway moves to prevent the system from falling. Unfortunately the acceleration $\ddot{\theta}$ exerts a torque that the SegMonster must compensate for, also leading to a non minimum-phase system. Figure 5.17 shows the SegMonster's response to a 6-s pulse between 0 and 1 m/s. The step responses are analyzed in Fig. 5.18.

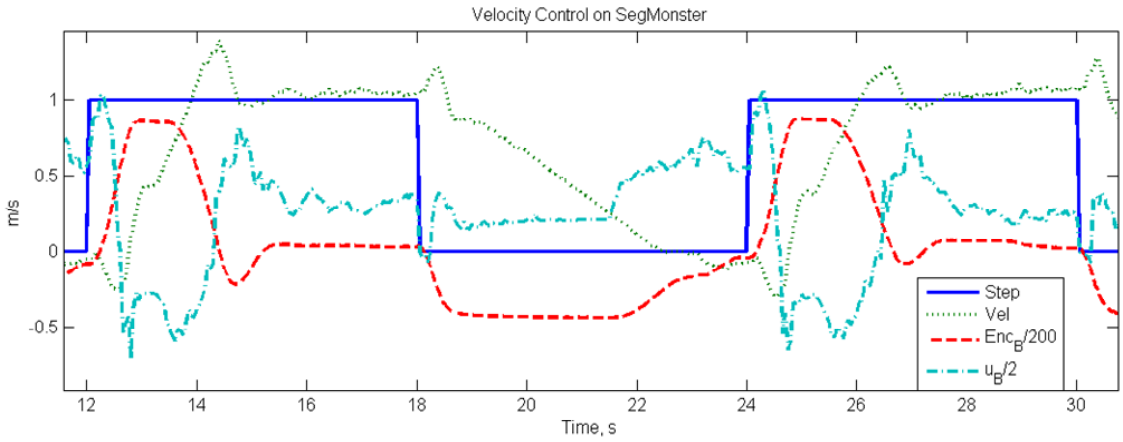


Figure 5.17: SegMonster response to a step velocity input. Using only a PID controller on the SegMonster results in a non minimum-phase system as well. A nonlinear controller is needed to further improve this response.

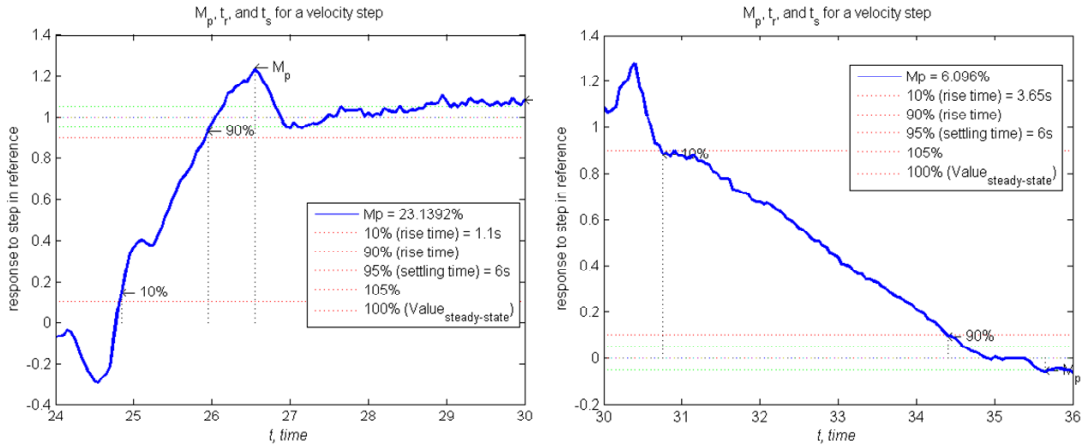


Figure 5.18: Positive and negative step responses for SegMonster velocity. The SegMonster cannot generate as much acceleration in the negative direction as the positive. A longer arm would improve control.

5.3.5 Low level control

We desire to control the SegMonster's location, and can do so by choosing the orientation and speed. Controlling the orientation and speed followed the lead of [27] and was achieved with outer loop controllers that set the desired hand and arm positions, with inner loop controllers to control the motor positions. These controllers are discrete and are calculated 1 000 times a second ($\Delta t = 1$ ms).

- *Hand Outer Loop Control*

  To control the orientation we can twist the grip handle with $motor_H$. The *Hand Outer Loop Control*, shown in Fig. 5.19, creates an *orientation error* signal from the *reference orientation* less the true *orientation* and applies a PI control to it. The error signal summed by the integral incorporates a *forgetting factor* by multiplying the summation by 0.997 each time step. The control effort is pushed through a dead zone compensator as described in Fig. 5.4 and passed into the *Hand Inner Loop Control*.

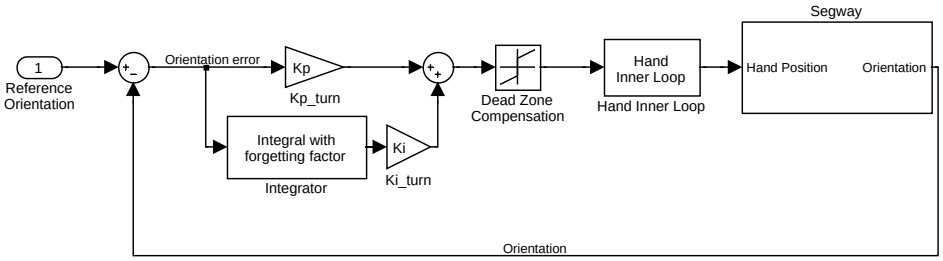


Figure 5.19: Orientation outer control loop block diagram.

- *Hand Inner Loop Control*

  The DC motors on the SegMonster are strong, and are capable of rapid movement. This rapid movement is damaging to the cables connecting the motor to the twist grip. We want a smooth touch at the controls. The *Hand Inner Loop Control*, shown in Fig. 5.20, uses a PID controller, but slowing the response using the derivative is not possible because of the noise inherent in discrete derivatives. Instead, an input shaping function was added that updates a state variable $mH\_desired\_prev$. The current motor position is mH_pos. If the desired motor position is greater than $mH\_desired\_prev$ and mH_pos, $mH\_desired\_prev$ is updated to $max(mH\_pos, mH\_desired\_prev) + \Delta step$. If the desired motor position is less than $mH\_desired\_prev$ and mH_pos,

$mH\_desired\_prev$ is updated to $min(mH\_pos, mH\_desired\_prev) - \Delta step$. Otherwise, $mH\_desired\_prev = desired\_position$. This results in a smooth wrist controller. The next important update was an input saturation to limit control effort to no more than 50% of maximum torque. This amount of torque is sufficient to completely control the wrist. Extra effort merely heats up the motor and, possibly, damages the robot.

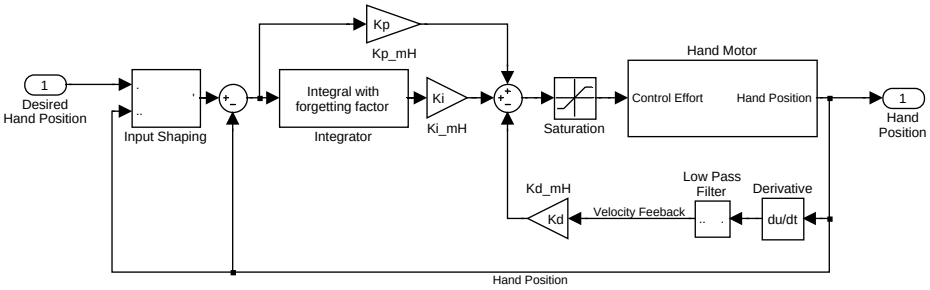


Figure 5.20: Orientation inner control loop block diagram.

- *Arm Outer Loop Control*

  Speed is also controlled with an inner and outer loop. The outer loop, shown in Fig. 5.21, specifies a desired arm position, and the inner loop controls the motor to achieve that position. A simple proportional control with a small position offset is sufficient to control the SegMonster on level ground. Unfortunately, tuning this control to overcome even a 4°, 3-cm elevation change caused the SegMonster to oscillate wildly. Integral control fixes this by ramping up control effort in the presence of disturbances. Unfortunately, integral windup is an issue. To avoid this, a *forgetting factor* was used that allows the SegMonster to grow integral compensation rapidly in the face of disturbances, and quickly diminish that control when the disturbance has passed.

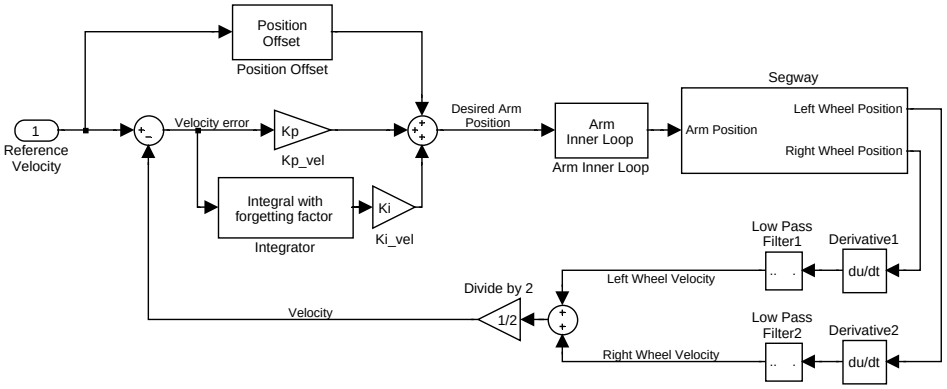


Figure 5.21: Speed outer control loop block diagram.

- *Arm Inner Loop Control*

  The *Arm Inner Loop Control*, shown in Fig. 5.22, also uses a PID controller, and employs a similar input shaping function. The saturation block is very important on the arm motor. One motor burned out during testing before this addition was made. The control circuit also takes advantage of the new limit switches, and will not input positive torque if the forward switch is high, or negative torque if the arm extended switch is high.

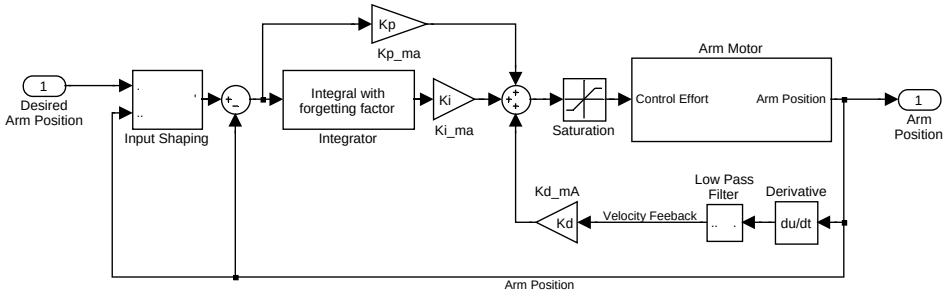


Figure 5.22: Speed inner control loop block diagram.

5.4 Controlling the SegMonster: Local Control

The SegMonster was originally designed with three modes of control. A DIP switch on the DSP board was used to choose the mode.

1. *Remote control*

   This was a basic velocity controller. A *Palm M100* connected to a serial radio was used to update speed and angular velocity commands.

2. *Red LED following*

   The SegMonster used a color camera to detect a red LED light bar. When the light bar was fixed to another Segway, the robot would follow the rider by segmenting the camera image and tracking the centroid of the largest red object in the frame.

3. *Sidewalk following*

   By segmenting images from the color camera, the SegMonster could follow the line between green grass and the sidewalk while maintaining a constant speed.

For this thesis project, the robot needed to be able to autonomously navigate the hallways of Everitt Lab. Wall following gave the robot a level of autonomy, and was a good base for testing localization procedures.

5.4.1 Wall following

The wall following algorithm, shown in Fig. 5.23, takes advantage of two additional IR sensors mounted on the SegMonster. These sensors have a latency of 70 ms, so the control effort is updated about 14 times a second. The SegMonster is able to cleanly navigate the hallways, as long as the doors leading to the stairwells are closed. The SegMonster also has difficulty navigating around tables, whose legs show up poorly on the IR scans.

The IR sensors return a nonlinear range reading that must be individually calibrated for each sensor. Figure 5.24 shows the calibration results for each sensor. The range measurement can be approximated by a fourth-order polynomial.

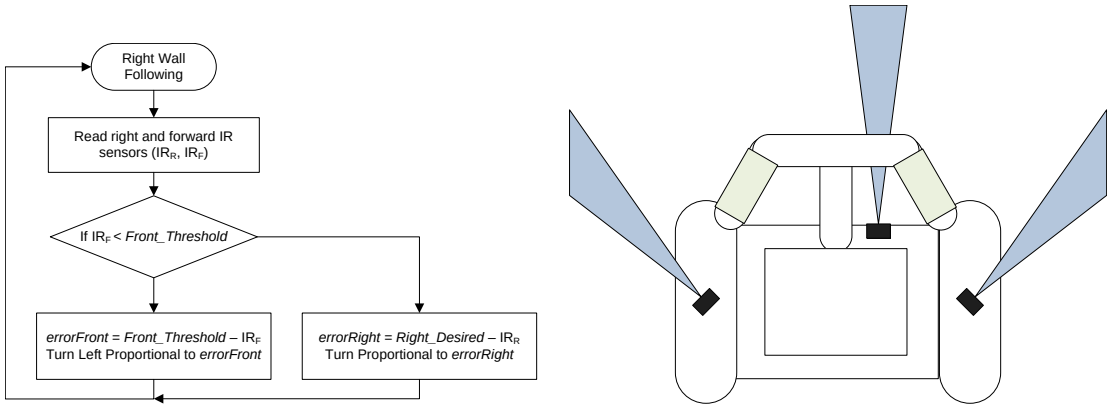


Figure 5.23: Wall following flow chart and IR sensor mounting positions. The black rectangles represent the three IR sensors.

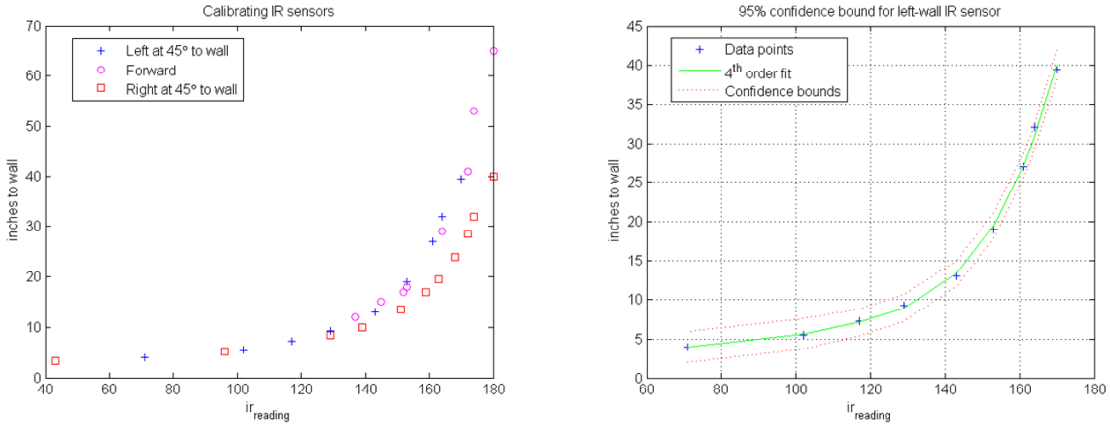


Figure 5.24: Calibrating IR sensors. Each IR must be calibrated individually. The right plot shows that a fourth-order fit adequately models the data.

5.5 Modifications and Improvements to the Robot

The SegMonster is made of a 1.2 m (4-ft) body, a two-link revolute arm pivoted at the shoulder of the body and the handles of the Segway, a wrist motor attached to the twist-grip turning control of the Segway, a base mounted between the two wheels that holds the DSP, and a mechanical foot pad to activate the Segway. It has two actuators, which correspond to the two control inputs of the robot model, $\text{speed}_{desired}$ and angular $\text{velocity}_{desired}$ ($\mathbf{u}_t = [v_{desired}, \omega_{desired}]$).

The actuators are permanent magnet DC motors with integrated optical encoders for angular position feedback. One motor controls the body position θ_B. The motor position varies from 228° to 86°, which translates to a body position from −15° to 26°. This moves the COM approximately 0.46 m (1.5 ft). Since the Segway must exert a constant acceleration to counteract a constant COM position, a mapping between body position and velocity does not exist.

The robot has four optical encoders, one each for the wrist and body motors and one friction-mounted to each wheel. These encoders are relative encoders, meaning they are initialized to a constant (usually zero) at power-up, and count from that initial value. The encoders are *quadrature encoders*, meaning they can identify movement in the forward and reverse directions. The wrist and wheel encoders have 1250 ticks per revolution. The arm position encoder had originally 1250 ticks per revolution as well, but the encoder mounting was custom machined and lacked the required tolerance. The small division marks required for 1250 ticks/revolution resulted in the encoder missing as much as 3% of the body position range at each full oscillation. This inaccuracy made body position difficult to control reliably. The motor has an internal gear ration of 75.11:1, so 1250 ticks/revolution was more resolution than needed for body positioning. A 50 count encoder was installed, which performs well without missing counts.

Relative encoders are fine for wheel position sensors, since setting odometry data to zero at startup is a reasonable assumption. The turning input to the Segway has internal springs that return the grip to its default setting of zero, so the wrist encoder can be safely assumed to start at zero as well. For the body position, a starting position cannot be assumed. For this reason limit switches were added to

the SegMonster. One switch is only activated when the body is in the full forward position, the other only when the body is in the full backward position (see Fig. 5.25). The control architecture was designed to not initialize the encoder until the body is in the full forward configuration, as shown in the SegMonster state machine, Fig 5.26.

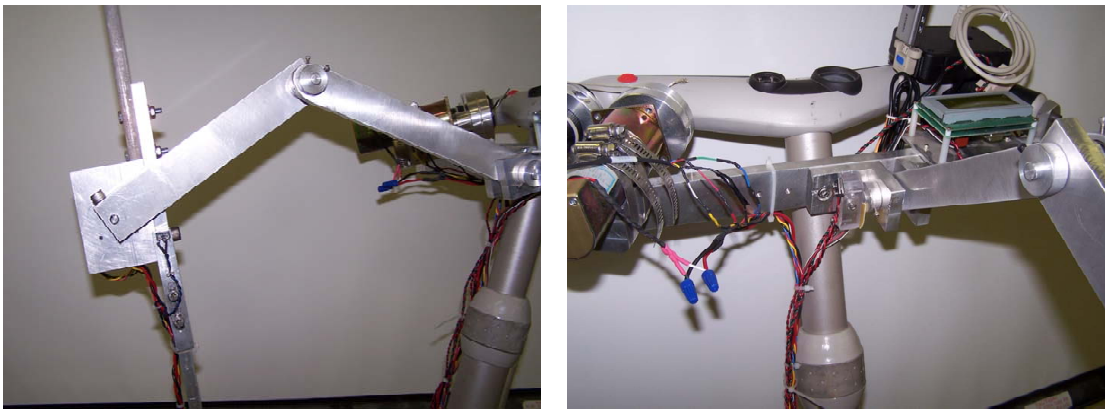


Figure 5.25: Body motion limit switches. The SegMonster uses relative encoders for feedback. To calibrate these, limit switches were attached at the body's range of movement limits.

The original wrist of the SegMonster turned the motor by twisting cables attached to a wheel on the motor. The attachment points (see Fig. 5.27) allowed only $\pm 20°$ of movement. The Segway's twist grip responds proportionally to input in the range of $\pm 90°$. A revision of the wrist allows the SegMonster to use this full range of motion. Occasionally in the past, when the SegMonster rolled down a hill it was unable to apply enough turn command to stay on the sidewalk. Now the SegMonster has as much control as a human rider.

Calculating the particle filter is computationally intensive, as Chapter 4 discussed. Adding this load to the DSP would make vision processing difficult. Also, the DSP board on the SegMonster is incapable of hosting USB devices. For these two reasons, the SegMonster was given a second processor. The new processor is a Gumstix

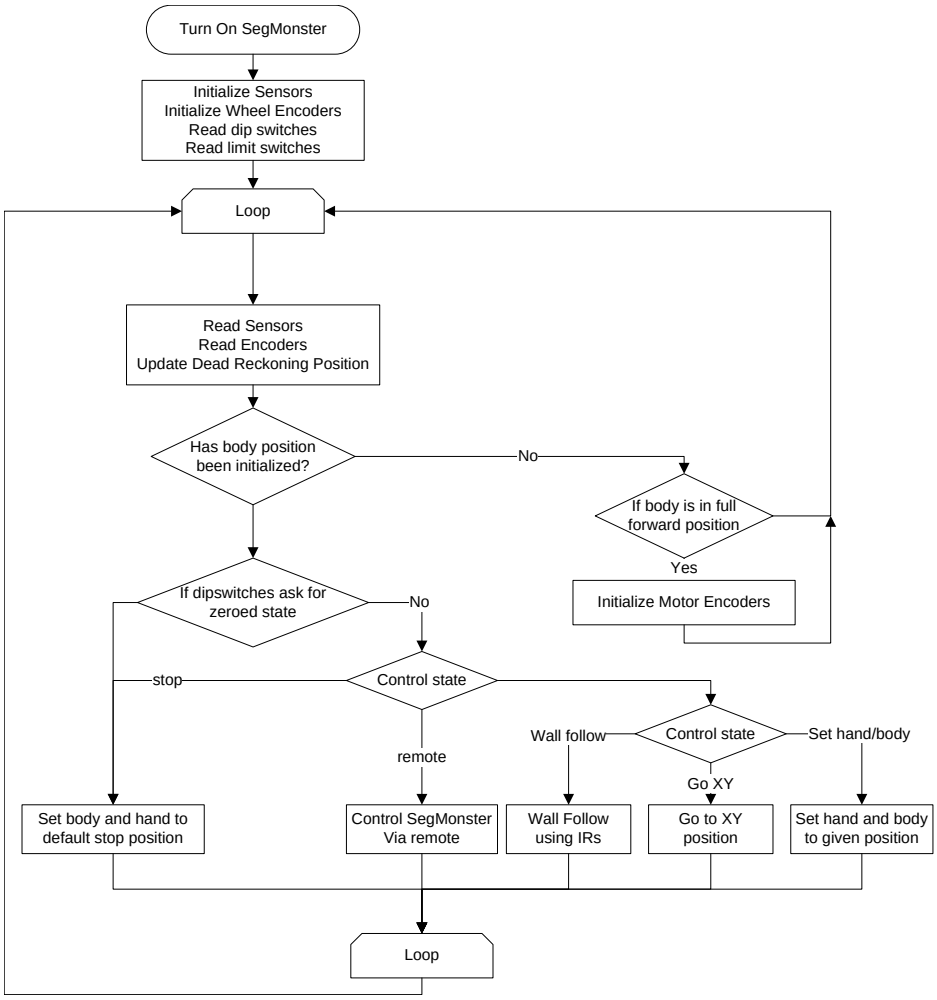


Figure 5.26: Outer loop state machine for the SegMonster.

computer, a small (the size of a stick of chewing gum) 600-MHz processor running a Linux operating system. The Gumstix processor is mounted on the SegMonster's handlebar, giving the Wi-Fi detector clear access to RF signals. Figure 5.28 shows the location of the processor and a closeup of the Gumstix. The Gumstix is a dedicated localization computer, and communicates with the DSP via a serial interface. The control flow for Gumstix is shown in Fig. 5.29.

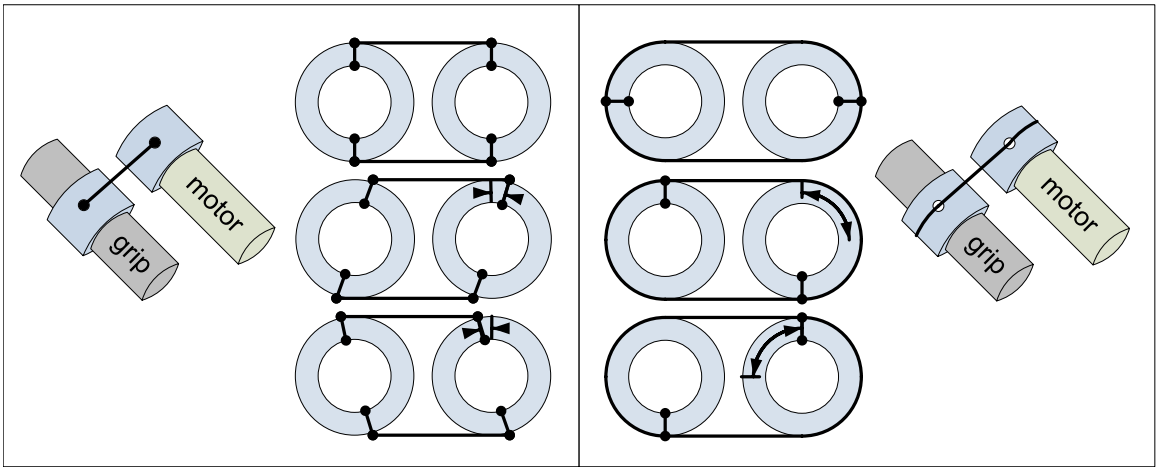


Figure 5.27: Improvement of SegMonster's wrist allows full control of ω_{turn}. The old arrangement is shown on the left and the new grip is shown on the right.

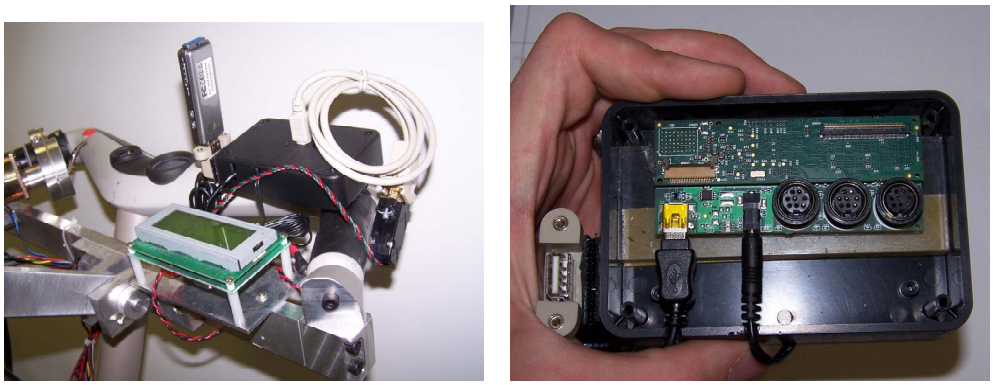


Figure 5.28: Gumstix processor and USB Wi-Fi detector. The Gumstix, a 600-MHz Linux processor is so named because it is the size of a stick of chewing gum. The Gumstix allowed USB peripherals, such as the Wi-Fi sensor, to be integrated, and it provided additional processing power.

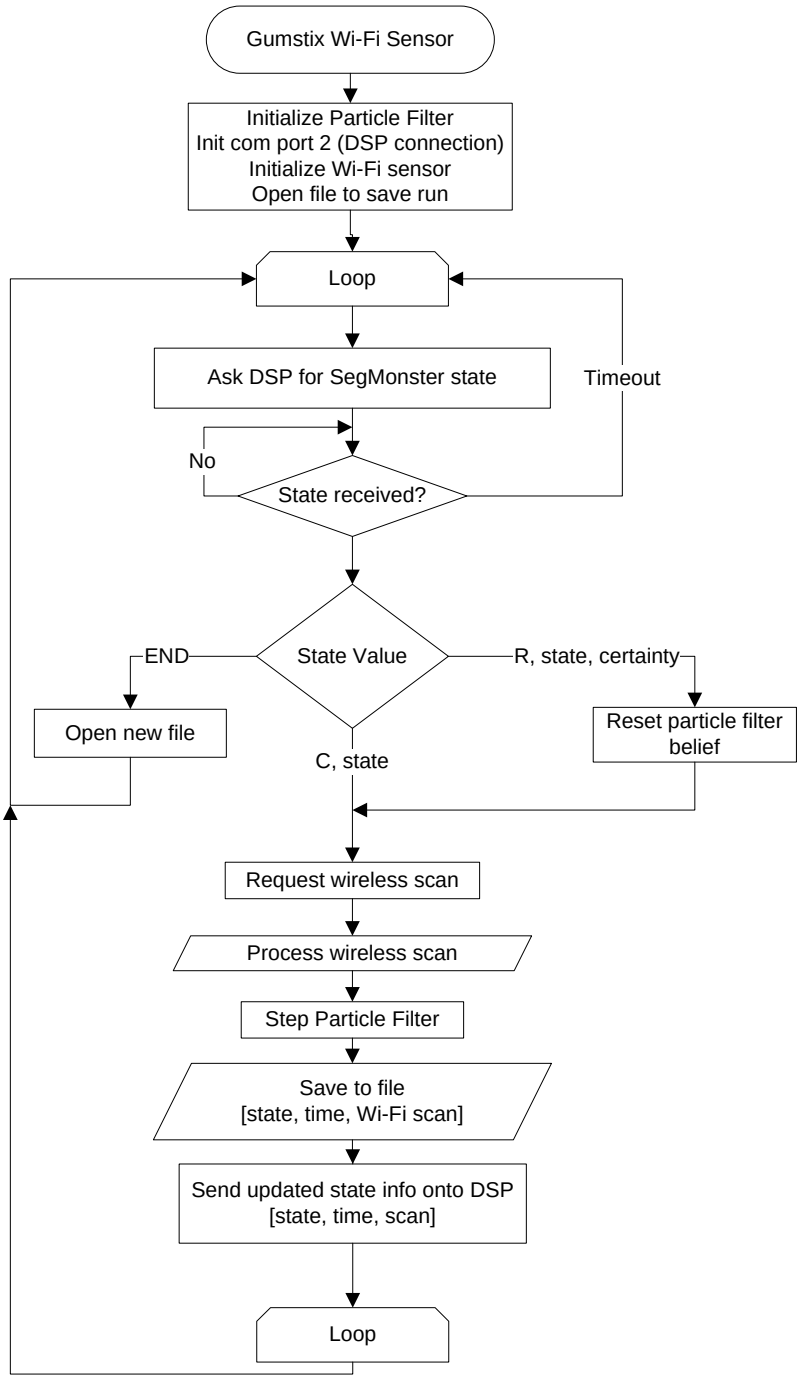


Figure 5.29: Flowchart for Gumstix processor. The Gumstix processor was turned into a dedicated localization device that polled for Wi-Fi data, integrated control inputs from the SegMonster, and calculated the particle fitler, returning accurate location estimates to the DSP.

CHAPTER 6

EXPERIMENTAL RESULTS

6.1 Dead Reckoning

Dead reckoning is our base case for comparing localization results, and was discussed thoroughly in Chapter 2. The error grew quickly, with a mean estimate nearly 58 m from the true position. The results are shown again for convenience in Fig. 6.1.

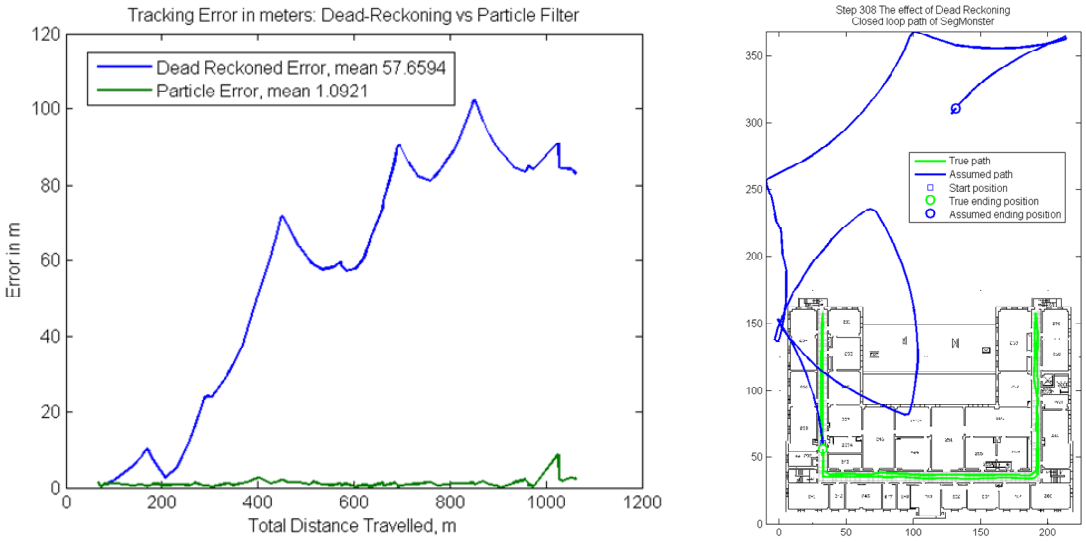


Figure 6.1: Test results showing dead reckoning error. The SegMonster navigated a closed loop path in the hallways of Everitt Lab (70 × 50 m). By the end of the first hallway (20 m) the dead reckoned estimate had accumulated a 30° error. The position error increased to 90 m over the course of the exercise.

6.2 Wi-Fi with No Local Sensors

Localization using Wi-Fi sensors does not require optical encoder data. Many papers have presented results using wireless signals for human tracking and localization. Using GP with no motion input data, our best results could only consistently localize with a 3.5 m average error. Others published better results by making more detailed maps [30] or by using a complex topology for the particle filter and processing larger GPs maps on a laptop computer [3]. When running without motion data, the most basic motion model is to assume that the robot moves with a random acceleration in the x and y axis. This is also easily modeled when the probability is discritized to a grid. Each time step, the previous probability is passed through a moving average filter which flattens probability peaks, then each grid cell is multiplied by the probability of the current measurement being received at that cell. The probability at any time can be represented with a contour map, as in Fig. 6.2.

6.3 Wi-Fi Localization Using Gaussian Processes

6.3.1 Position tracking

Our results on the SegMonster were shown previously in Fig. 6.1. They are compared to published results in Table 6.1.

Table 6.1: Literature Results: Position Tracking Using Wi-Fi Signal Strength

Average Error	Method
1.09 m	SegMonster
2.12 m [3]	Gaussian Processes, Particle Filter
3.88 m [31]	AP Location and Blueprints (no data gathering stage)
2.65 m [24]	Nearest Neighbor in Signal Space

Figure 6.2: Wi-Fi localization with no local sensors. The true position in each case is shown in green, and the sensed APs are represented with boxes. The contour lines denote paths of equal probability.

6.3.2 Global localization

Global localization is more difficult than position tracking. The SegMonster is capable of both. The heading variable is the most sensitive because the GP only provides position data, so heading must be inferred from the route the robot follows. Five global localization runs are compared against a position tracking run in Fig. 6.3. Note that though the particles generally converge to the true position, the heading may not be initially correct, leading to errors further in the process.

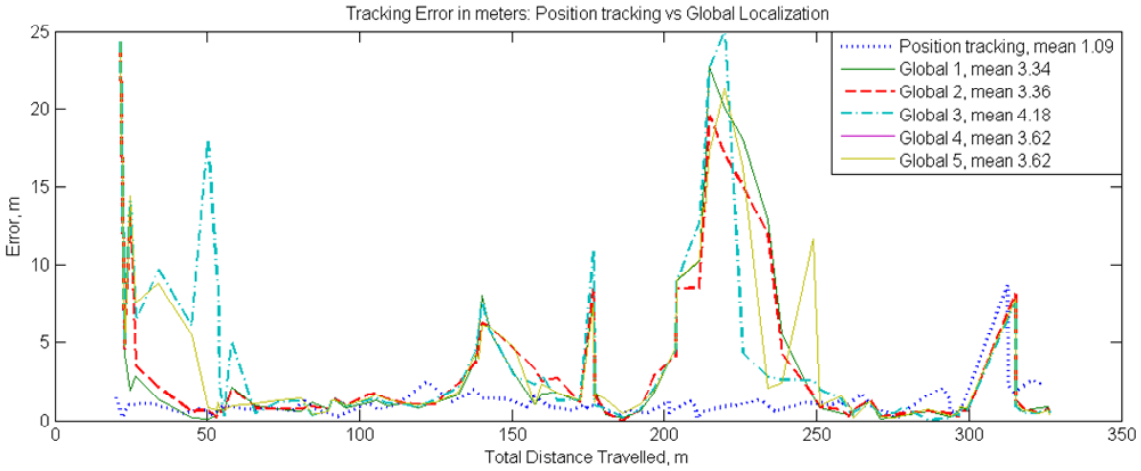


Figure 6.3: Position tracking vs. global localization. Five runs of global localization are compared against the control performing position tracking. Each run used 300 particles and the same sensor readings.

6.3.3 Tuning parameters

One of the biggest parameters to tune is the number of particles to propagate in the model. More particles increases confidence in the localization algorithm. The chance of losing all the *correct* particles during resampling drops as the number of particles increases. Unfortunately, processing time increases linearly with the number of particles maintained. Memory constraints also increase linearly. The number of particles necessary also increases with the dimensionality of the state space for the robot. The effect of additional particles is shown in Fig. 6.4.

Each run used the same data set of motion commands and sensor readings, but different results stem from the number of particles used. A logarithmic set of particle sizes are compared in the plot. Each set performed global localization. Their position error in meters is displayed, along with the error of a 300 particle set performing position tracking. Each run converged to the true *position* rapidly, within the first 4 s. The GP was able to filter out particles not near the true position. Unfortunately, often the particles selected by GP had incorrect orientations. This is particularly

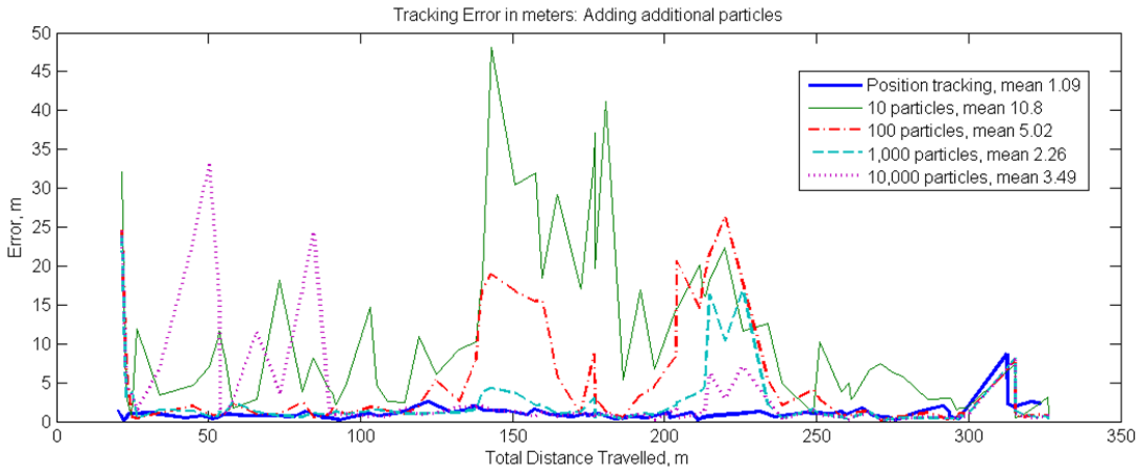


Figure 6.4: Effect of additional particles. Adding additional particles generally improves localization.

obvious for the smaller sample sizes. Using only 10 particles results in a fragile localization. At the 150 m mark, both the 10 and 100 particle runs lost the robot, but the 100 particle sample recovered more quickly. It is noteworthy that the 10 000 sample set took the longest to converge, but was the most accurate from that point on.

CHAPTER 7

CONCLUSIONS

This thesis presented and contrasted methods for localizing a mobile robot. A solution was implemented on the SegMonster for recursive state estimation using a particle filter. Simulation and experimental results were shown, along with a method for signal strength-based localization using Gaussian processes. This thesis also discussed, improved, and implemented controllers for a robot riding a Segway.

7.1 Future Work

Future work will extend this process to a larger, outdoor environment. Outdoors, wall following will no longer be an option, but the SegMonster is capable of sidewalk following using an onboard color camera. Outside, there is much less interference from walls and floors. There is a wealth of wireless signals around campus the robot can use for localization, as Fig. 7.1 shows. Also, GPS and compass data can be integrated into the particle filter for more accurate localization. There is still room for improvement for the robot's velocity control. Though the results are near those of the RMP, the SegMonster should be able to surpass the RMP.

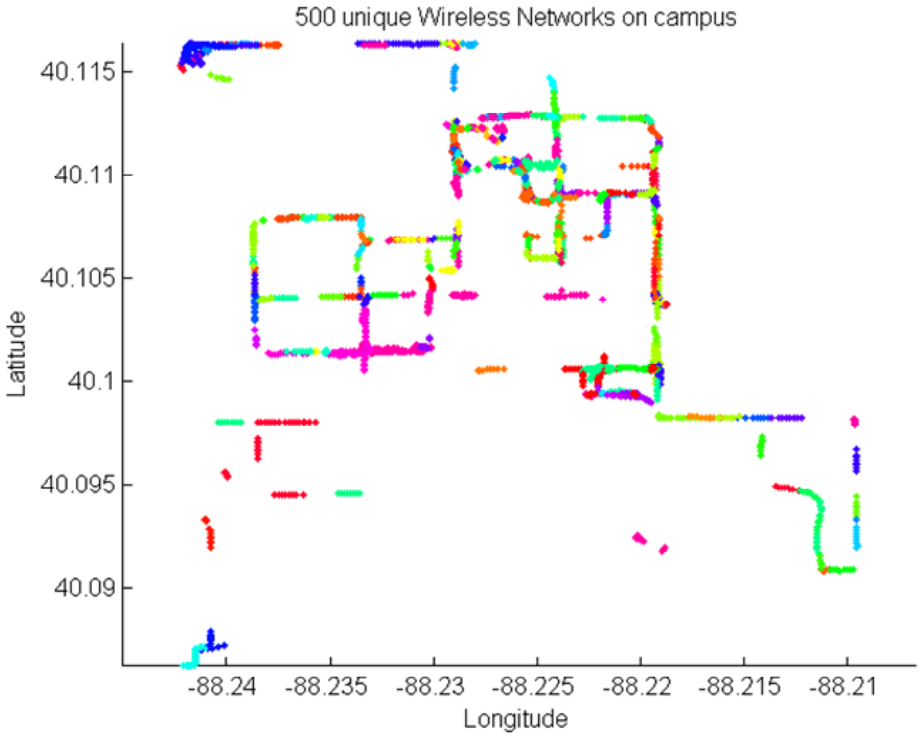


Figure 7.1: 500 unique Wi-Fi access points around campus. During a survey run on the campus buslines, we recorded 1819 unique wireless networks on campus. There are enough signals for GP localization in an outdoor environment.

APPENDIX A

COMPANION CD

The programs used in this thesis project along with videos of the SegMonster in action are included on the CD. Figure A.1 lists the contents of the CD.

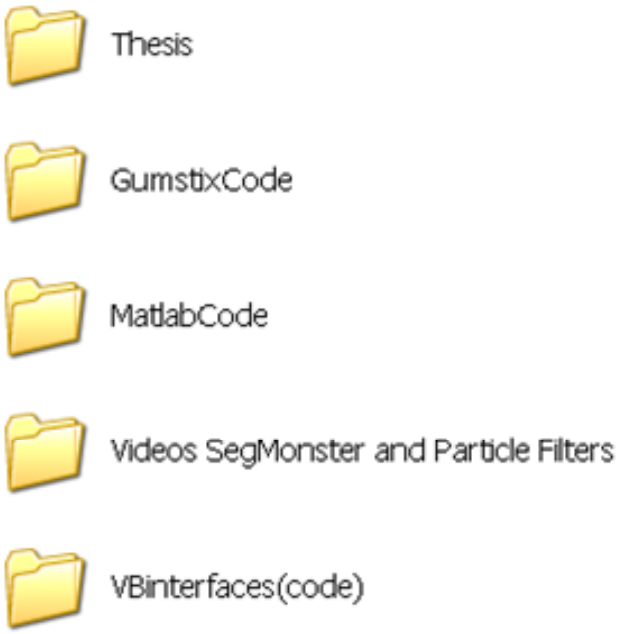


Figure A.1: Files included on the CD. These files include code for MATLAB, the Gumstix processor, the SegMonster's DSP, and Visual Basic interfaces. Also included are videos of the SegMonster and videos of the particle filter localizing the robot

REFERENCES

[1] M. Spong, S. Hutchinson, and M. Vidyasagar, *Robot Modeling and Control*. Hoboken, NJ: John Wiley & Sons, Inc., 2006.

[2] R. Arkin, *Behavior-Based Robotics*. Cambridge, MA: MIT Press, 2006.

[3] B. Ferris, D. Hahnel, and D. Fox, "Gaussian processes for signal strength-based location estimation," *Robotics Proceedings*, vol. rs02, pp. 39–47, 2006.

[4] A. King, "Inertial navigation - forty years of evolution," *GEC REVIEW*, vol. 13, no. 2, pp. 140–149, 1998.

[5] S. M. LaValle, *Planning Algorithms*. Cambridge, U.K.: Cambridge University Press, 2006.

[6] G. Franklin, J. Powell, and A. Emami-Naeini, *Feedback Control of Dynamic Systems*. Upper Saddle River, NJ: Pearson Prentice Hall, 2006.

[7] P. Enge and P. Misra, "Special issue on GPS: The global positioning system," *Proceedings of the IEEE*, vol. 87, no. 1, pp. 3–15, August 1999.

[8] R. Want, A. Hopper, V. Falcao, and J. Gibbons, "The active badge location system," *ACM Transactions on Information Systems*, vol. 10, no. 1, pp. 91–102, January 1992.

[9] N. Prijantha, A. Chakraborty, and H. Balakrishan, "The cricket location-support system," in *MobiCom '00: Proceedings of the 6th Annual International Conference on Mobile Computing and Networking*, August 2000, pp. 32–43.

[10] T. W. Christ, P. Godwin, and R. Lavigne, "A prison guard duress alarm location system," in *Institute of Electrical and Electronics Engineers 1993 International Carnahan Conference*, Oct 1993, pp. 106–116.

[11] M. Schneider and C. Stevens, "Development and testing of a new magnetic-tracking device for image guidance," Ascension Technology Corporation, Milton VT, Tech. Rep., January 2007.

[12] S. Thrun, M. Bennewitz, W. Burgard, A. Cremers, F. Dellaert, D. Fox, D. Hahnel, C. Rosenberg, N. Roy, J. Schulte, and D. Schulz, "Minerva: A second-generation museum tour-guide robot," in *IEEE International Conference on Robotics and Automation*, vol. 3, 1999, pp. 1999–2005.

[13] N. Ravi and L. Iftode, "Fiatlux: Fingerprinting rooms using light intensity," 2007, presented at 5th International Conference on Pervasive Computing, Toronto, Canada.

[14] V. Otsason, A. Varshavsky, A. LaMarca, and E. de Lara, "Accurate GSM indoor localization," in *LNCS 3660*, 2005, pp. 141–158.

[15] R. Hogg, J. McKean, and A. Craig, *Introduction to Mathematical Statistics*. Upper Saddle River, NJ: Pearson Prentice Hall, 2005.

[16] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery, *Numerical Recipes in C*. New York, NY: Cambridge University Press, 1992.

[17] D. Coppersmith and S. Winograd, "Matrix multiplication via arithmetic progressions," *Journal of Symbolic Computation*, vol. 9, no. 3, pp. 251–280, March 1990.

[18] R. E. Kalman, "A new approach to linear filtering and prediction problems," *Journal of Basic Engineering*, vol. 82D, pp. 35–45, 1960.

[19] H. Choset, K. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. Kavraki, and S. Thrun, *Principles of Robot Motion*. Boston, MA: MIT-Press, 2005.

[20] A. Doucet, N. Freitas, and N. Gordon, Eds., *Sequential Monte Carlo Methods in Practice*. New York, New York: Springer-Verlag, 2001.

[21] S. Thrun, W. Burgard, D. Fox, and L. Lamport, *Probabilistic Robotics*. Cambridge, MA: MIT-Press, 2005.

[22] J.Ansari, J. Riihijrvi, and P. Mhnen, "Combining particle filtering with cricket system for indoor localization and tracking services," in *18th Annual IEEE International Symposium on Personal, Indoor and Mobile Radio Communication*, September 2007, pp. 1–5.

[23] G. Adorni, S. Cagnoni, and M. Mordonini, "Landmark-based robot self-localization: a case study for the robocup goal-keeper," in *Information Intelligence and Systems*, 1999, pp. 164–171.

[24] P. Bahl, V. Padmanabhan, and A. Balachandran, "A software system for locating mobile users: Design, evaluation, and lessons," 2000, revised version of Microsoft Research Paper.

[25] C. Rasmussen and C. Williams, *Gaussian Processes for Machine Learning*. Boston, MA: the MIT Press, 2006.

[26] R. Brooks, L. Aryananda, A. Edsinger, P. Fitzpatrick, C. Kemp, U. O'Reilly, E. Torres-Jara, P. Varshavskaya, and J. Weber, "Sensing and manipulating built-for-human environments," *International Journal of Humanoid Robotics*, vol. 1, no. 1, pp. 1–28, January 2004.

[27] D. Johnson, "Mechanical design, construction, and control of an autonomous segway operator," M.S. thesis, University of Illinois at Urbana-Champaign, 2005.

[28] Segway HT Technical Staff, *Segway HT reference manual,i Series, p Series, Segway XT, Segway GT*, Segway HT, 2005.

[29] H. Nguyen, H. Morrell, K. Mullens, A. Burmeister, S. Miles, N. Farrington, K. Thomas, and D. Gage, "Segway robotic mobility platform," *SPIE Proc. 5609*, vol. Mobile Robots XVII, pp. 308–22, October 2004.

[30] A. Ladd, K. Bekris, A. Rudys, D. Wallach, and L. Kavraki, "On the feasibility of using wireless ethernet for indoor localization," *IEEE Transactions on Robotics and Automaton*, vol. 20, no. 3, pp. 555–559, June 2004.

[31] M. Robinson and I. N. Psaromiligkos, "Received signal strength based location estimation of a wireless LAN client," in *Wireless Communications and Networking Conference, New Orleans, LA*, March 2005.