State Estimation for a Holonomic Omniwheel Robot Using a Particle Filter

by

Donald S. Eng

S.B., E.E.C.S. M.I.T., 2009

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

[June 2010]
May 2010

Author_____
Department of Electrical Engineering and Computer Science
May 21, 2010

Certified by_____
Dr. Leslie Kaelbling Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by_____
Dr. Christopher J. Terman
Chairman, Department Committee on Graduate Theses

State Estimation for a Holonomic Omniwheel

Robot Using a Particle Filter

by

Donald S. Eng

Submitted to the
Department of Electrical Engineering and Computer Science

May 21, 2010

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

The holonomic robot platform designed for the Opera of the Future must perform continuously on stage in a 10 meter by 20 meter world for one hour. The robot interacts with twelve other robots, stage elements, and human performers. Fast, accurate, and continuous state estimation for robot pose is a critical component for robots to safely perform on stage in front of a live audience. A custom robot platform was designed to use a Particle Filter to estimate state. The motor controller was developed to control robot vectoring and report odometry, and noise analysis on an absolute positioning system, Ubisense, was performed to characterize the system. High frequency noise confounds the Ubisense measurement of θ, but the Particle Filter acts as a low pass filter on the absolute positions and mixes the high frequency components of the odometry to determine an accurate estimate of the robot pose.

Thesis Supervisor: Dr. Leslie Kaelbling
Title: Professor of Computer Science and Engineering

Table of Contents

Section Page

1 Introduction
- 1.1 Fast Real-time State Estimation 7
- 1.2 Influence of Robot Aesthetics 8
- 1.3 Overview of Design Work 8
- 1.4 Designing Experiments 8

2 Robot Drive System Design
- 2.1 System Overview 12
- 2.2 Python Joystick Control 12
- 2.3 Hardware Overview 14
  - *2.3.1 Arduino Mega, Atmega 1280* 14
  - *2.3.2 Encoders* 15
  - *2.3.3 H-bridges* 17
- 2.4 Motor Controller Firmware 18
  - *2.4.1 Designing Motor Controller Firmware* 18
  - *2.4.2 Speed Calculator* 20
  - *2.4.3 PID velocity control* 20
  - *2.4.4 Velocity Transform Equations* 21
  - *2.4.5 Final Event Timing Configuration* 22
- 2.5 Odometry Results 23
  - *2.5.1 Adjusting Odometry Constants* 23
  - *2.5.2 Path plots* 24
  - *2.5.3 Cross comparisons* 25
  - *2.5.4 Analysis of Odometry* 25

3 Robot Absolute Positioning
- 3.1 Introduction to Ubisense 26
  - *3.1.1 Selecting an absolute positioning system* 26
  - *3.1.2 Ubisense Technology* 28
  - *3.1.3 Ubisense Tag Update Slots* 29
- 3.2 Sensor Network Topology 30
  - *3.2.1 Sensor Network Layout* 30
  - *3.2.2 Tag Locations on the Robot* 31
- 3.3 Noise Characteristics 32
  - *3.3.1 Stationary Tags* 32
  - *3.3.2 Moving Tags* 33
- 3.4 Filtering Tags 35
- 3.5 Synchronizing Odometry and Absolute Positioning 38
  - *3.5.1 Adding Offsets* 38
  - *3.5.2 Warping Time* 39

3.6 Preliminary Combined Results 40

4 Particle Filter for State Estimation
4.1 Particle Filter or Extended Kalman Filter 41
4.1.1 Advantages and Disadvantages 42
4.1.2 Particle Filter Implementation 42
4.2 Characterizing the filter 43
4.2.1 Representation of Belief 43
4.2.2 Representation of State 44
4.2.3 Selection of Particles 45
4.3 Particle Filter Update 45
4.3.1 Updating cloud from odometry measurements 45
4.3.2 Updating cloud from Absolute Positioning measurements 47
4.3.3 Reselecting Particles 49
4.3.4 Behavior summary 50
4.4 Particle Filter Results 51
4.4.1 Animation of Filter 51
4.4.2 Results Representation 52
4.4.3 Comparing Non-holonomic and Holonomic 54
4.4.4 Analysis of Directionality 57
4.4.5 Analysis of Random Track 59
4.4.6 Issues from Manual Drive 60

5 Conclusions
5.1 Particle Filter Performance 60
5.2 Particle Filter Improvements 61
5.3 Implementation in Real time 61

6 Acknowledgements 62

7 References 62

8 Appendices 63
8.1 Python Joystick Control Code 63
8.2 Firmware Code 68
8.2.1 Main Module 68
8.2.2 PID Module 75
8.2.3 Encoder Class Module 78
8.2.4 Motor Assembly Module 81
8.3 Matlab data processing code 83
8.4 Matlab Particle filter code 92

Table of Figures

Figure	Title	Page
1.1	Picture of the Opera Robots	7
1.2	Robot Base and Omniwheel.	8
1.3	World and Tracks	9
1.4	Experiment Conditions	10
1.5	Illustrations of Experiments	10
2.1	Robot Controls System Overview	12
2.2	Mapping Joystick Position to Robot Velocity	13
2.3	Velocity Mapping Example and Python Serial Interface	13
2.4	Odometry Data Packet Protocol	14
2.5	The Arduino Mega	15
2.6	Image of Encoder and Disc	15
2.7	Interrupts for Encoder Sampling Illustrating Counting Techniques	16
2.8	H-bridge and Servo Control Signal.	18
2.9	Overview of Motor Controller Firmware	19
2.1	Feedback Control Diagram	19
2.11	PID Response Curve	20
2.12	Vector Transform Diagram for Holonomic Drive	21
2.13	Final Event Timing Diagram	22
2.14	Rescaling of Angular Rate	23
2.15	Odometry Path Plots for Experiments 1-9	24
3.1	Ubisense Noise Model and Reflections	27
3.2	Error Analysis for UWB Signals	27
3.3	Ubisense Operation illustration	28
3.4	Solution Sets	29
3.5	Tag and Sensor Geometries	30
3.6	Linear Tag Interpolation	32
3.7	Ubisense x, y, θ Histograms for Stationary Tag Noise	33
3.8	Ubisense Absolute Positioning Path Plots for Experiments 1-9	34
3.9	Ubisense Measurement of θ for Holonomic Circles Unfiltered	35
3.1	Comparison of Ubisense Measurements for Robot Vectors	35
3.11	Histogram for Ubisense Robot Diameter Measurements	36
3.12	Ubisense Robot Vectors Filtered and Removed for Holonomic CCW Circles	37
3.13	Histograms for Filtered and Removed Ubisense Measured θ for Holonomic CCW Circles	37
3.14	Pre and Post Offsets for Combining Ubisense and Odometry Data	38
3.15	Removing the Odometry Time Warping	39
3.16	Scaled and Offset Ubisense and Odometry Time Courses for Rectangular CW Track	40
3.17	Cross Section of Sin(θ) Cos(θ) Time Courses (100-200s) for Rectangular CW Track	41
4.1	Belief Representation with Particle Cloud	43
4.2	Transition Update with Odometry	45
4.3	Adding Process Noise to Odometry Measurements	46
4.4	Observation Model Based on Measured Robot Diameter	48

4.5	Creating a New Cloud by Reselecting Particles	49
4.6	Illustration of Reselection and Addition of Initial Process Noise	50
4.7	Typical Evolution of Belief Distribution	51
4.8	Animations of Odometry, Ubisense, and the Particle Filter.	52
4.9	Cross Comparison of Path Plots for Experiment 1, Rectangle CW	53
4.1	Cross Comparison of Time Courses for Experiment 1, Rectangle CW	53
4.11	Cross Section of Sin(θ) Cos(θ) time course (100-200s).	54
4.12	Cross Comparison of Path Plots for Experiment 4, Non-holonomic CCW Circles	54
4.13	Cross Comparison of Time Courses for Experiment 4, Non-holonomic CCW Circles (100-150s).	55
4.14	Cross Comparison of Path Plots for Experiment 6, Holonomic CCW Circles	56
4.15	Cross Comparison of Time Courses for Experiment 6, Holonomic CCW Circles (100-150s).	56
4.16	Cross Comparison of Path Plots for Experiment 7, Switchback Suboptimal	57
4.17	Cross Comparison of Time Courses for Experiment 7, Switchback Suboptimal (100-150s).	57
4.18	Cross Comparison of Path Plots for Experiment 8, Switchback more Optimal	58
4.19	Cross Comparison of Time Courses for Experiment 8, Switchback more Optimal (100-150s).	58
4.20	Cross Comparison of Path Plots for Experiment 9, Random Track.	59
4.21	Cross Comparison of Time Courses for Experiment 9, Random Track	59
5.1	Path Plots for Experiment 6 Varying Particles, Holonomic CW Circles	61

1 Introduction

The Opera of the Future group at the MIT Media lab is conducting a robot opera to be performed in Monaco in early September 2010. There are many technical components for the visualization of the opera including: robotic interactive walls, voice modifying devices, a giant musical chandelier, and a multiagent robot system. The multiagent robot system is composed of twelve 7ft tall robots. This research concerns the design and implementation of a single robot for this multiagent system.

1.1 Fast Real-time State Estimation

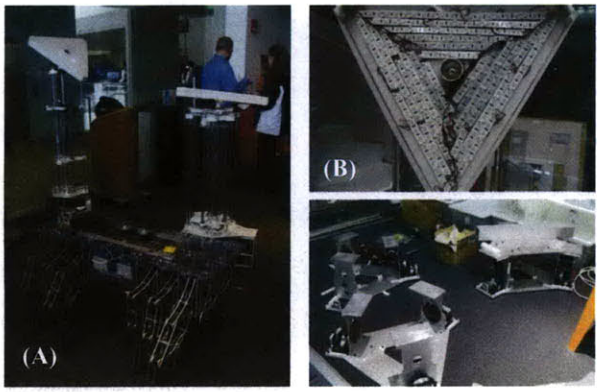


Figure 1.1: Picture of the Opera Robots. (A) The Opera of the Future will feature walking robots and several triangular holonomic robots. (B) The control and state estimation of the holonomic robots are the focus of this research

The robots will perform on stage for roughly an hour during the opera. Because they will be operating simultaneously and sharing the stage with actors, robotic walls, and other stage pieces, accurate localization is a critical component in designing the robots. It is imperative that the robots maintain a safe distance from the stage boundaries and the audience. There is an obvious safety concern for the audience and the performers with twelve 300 pound robots dancing around the stage.

Another issue is the speed of the robots. To maintain the proper aesthetics the robots need to operate gracefully and quickly around the stage. The robots will drive at roughly walking speed around the stage. This aspect is another challenge for accurate state estimation. Fast, accurate state estimation for an extended period of time can be achieved with a combination of odometry and absolute positioning. Therefore the design of the robots was conducted from the bottom up.

1.2 Influence of Robot Aesthetics

At the request of the opera director, the robot mechanics were designed for a holonomic drive system. This design choice allows the robots to glide around the stage translating in any direction while performing graceful spins. While aesthetically pleasing, the holonomic drive mechanics provide another challenge for accurately estimating the robot's location on stage.

Figure 1.2: Robot Base and Omniwheel. (A) The triangular robot based used for experimentation. (B) A close up of one of the omniwheels

A triangular wheel configuration was implemented using a set of three omniwheels. The triangular configuration was selected for stability on stage to keep the robot in three points of contact with the ground at all times. A previous non-holonomic two wheel drive system caused the robot to oscillate dramatically because of its height. It was decided that the two wheel system was mechanically unstable and aesthetically undesirable. However, omniwheels are designed to slip during translation which can cause inconsistencies in the robot odometry.

1.3 Overview of Design Work

The design of the robot drive hardware was carefully considered to report odometry readings. Custom firmware for controlling the robot drive configuration was designed, written, and implemented. An appropriate absolute positioning system was researched and selected for robot

localization. Finally, these components were integrated into a localization engine for post processing with a custom Particle Filter for the robot's state estimation.

Because this research serves as a proof of concept for accurate real-time state estimation, post-processing the odometry and absolute positioning data is an acceptable solution. The success of this research will support the implementation of a similar state estimation algorithm onboard each robot to be run in real time. Once implemented onboard, the state estimator should be directly compatible with the robot odometry hardware and the absolute positioning system chosen.

1.4 Designing Experiments

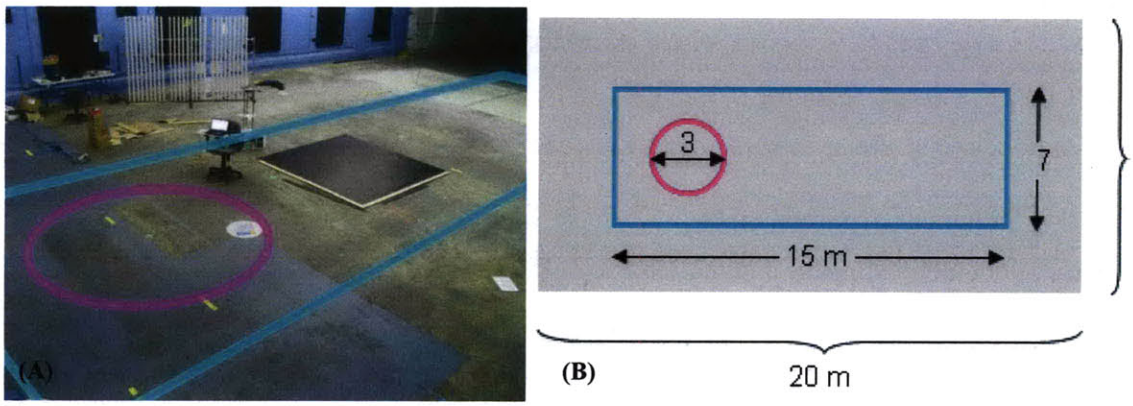


Figure 1.3: World and Tracks. (A) Image of the warehouse used for running experiments with the experimental tracks outlined. (B) Illustration of the world and tracks from an aerial view.

The robot was tested in a large empty warehouse. Its world is roughly defined by a 10m by 20m rectangle. Two tracks were carefully measured and laid out in the robot world: a 7m by 15m rectangle and a 3m in diameter circle. The robot was run on bare concrete, which represents a lower bound on the condition of the actual stage. The bare floor is pitted and contains a number of inclusions and separations where old tiling has not been removed. These floor features provided a handful of challenges for the robot odometry because of the inevitable wheel slippage over the uneven surface.

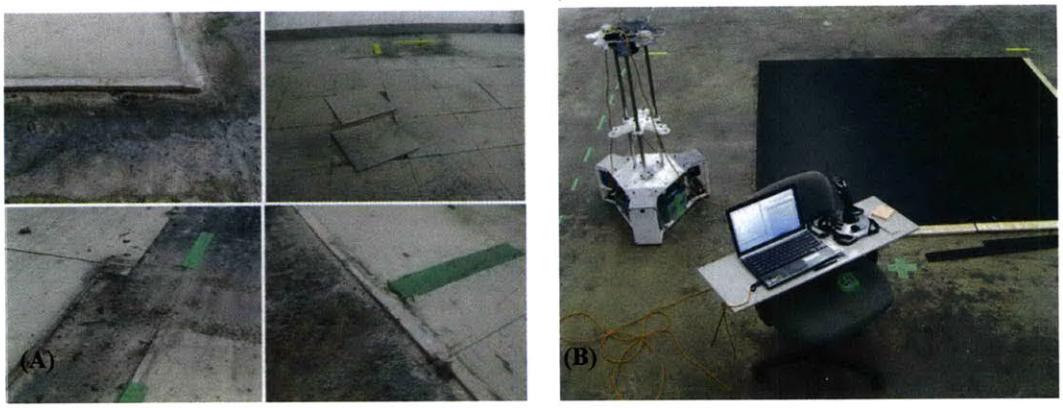


Figure 1.4: Experiment Conditions. (A) Documentation of floor features. (B) The robot base tethered to the computer via USB for joystick control.

While several test runs were made setting up the localization engine, nine base line experiments were chosen for post processing. The robot was run tethered to a computer for odometry data collection and joystick control. The robot was human controlled for the experiments. To compensate for human error running over the track, several laps were made on each track to see the effects of odometry slippage and the accuracy of the absolute positioning system over an extended period of time. While the human error was significant at times during the experimentation, the tracks were large enough that this should appear as high frequency noise.

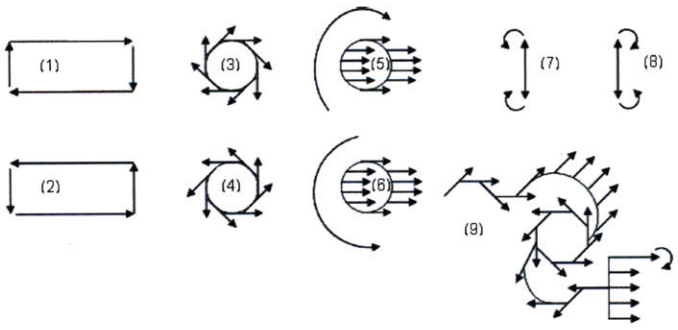


Figure 1.5: Illustrations of Experiments. The robot was driven in experiments 1-9 are as follows 1) Rectangle CW 2) Rectangle CCW 3) Circle CW 4) Circle CCW 5) Holonomic circle CW 6) Holonomic circle CCW 7) Switchbacks in suboptimal location 8) Switchbacks in more optimal location 9) Random track with irregular input.

The nine base line experiments are shown in the figure 1.5. For experiments 1 and 2 the robot was driven for 10 laps on the rectangular track clockwise (1) and counter clockwise (2) with the typical non-holonomic drive. For experiments 3 and 4, several laps were run on the circular track clockwise (3) and counter clockwise (4) as if the robot were a non-holonomic system. For experiments 5 and 6, several laps were run on the circular track clockwise (5) and counter clockwise (6) using the robot's holonomics having the robot maintain a constant heading. For experiments 7 and 8, several quick switch backs were made on the left (7) and again on the right (8) as if the robot were non-holonomic. Finally for experiment 9, the robot was driven randomly around the world with a series of non-holonomic and holonomic motions, small rings, and short switchbacks.

These experiments were chosen because for the variety of motions they capture. Experiments were run clockwise and counter clockwise to compare any skew in directionality. The rectangular experiments 1 and 2 test for a mix of long and short distances over a straight line and sharp turns. Experiments 7 and 8 test repetition, short distances over a long period of time, a high frequency of change, and extremely sharp turns.

A comparison of non-holonomic and holonomic drives was performed across experiments 3-6. Again, directionality was considered for analyzing angular drift. The ring experiments also test longevity with a low frequency input. Finally, experiment 9 combined the motions captured in each individual experiment. Experiment 9 tests the effectiveness of the state estimator to transition between non-holonomic and holonomic movements and irregular inputs.

For each experiment, the odometry data was collected from the robot at roughly 10Hz and written to a data file on the robot control computer with the elapsed time between odometry measurements. A server running the absolute positioning system recorded robot positions at exactly 4Hz with time stamps. These two data files were post-processed with the state estimator. Because these two computers were running independently, there was a combination of clock skew from the two systems and a slight inaccuracy in the elapsed time between odometry measurements. Hand pre-processing of the two data files was conducted to line up the two data sets to run simultaneously. The effects of this hand processing are apparent and discussed more fully in section 3.5

2 Robot Drive System Design

2.1 System Overview

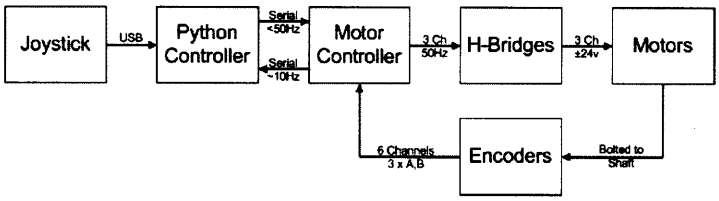


Figure 2.1: Robot Controls System Overview. The joystick input is parsed by a Python program, x, y, θ velocities are sent to the motor controller, control signals to the H-bridges change the voltages applied to each motor, quadrature shaft encoders provide speed measurements for feedback control.

The high level overview of the robot drive system is shown in figure 2.1. The joystick is used for user input. The joystick then feeds into a Python joystick module to process the joystick positions. A short Python script was written to convert these joystick positions into x, y, θ velocities. These velocities are converted into serial commands and using a Python serial module these desired velocity commands are sent to the robot motor controller for execution at a data rate less than 50Hz

The robot motor controller is developed on an Arduino Mega, which is running an ATMEGA 1280. The motor controller takes in x, y, θ velocities and converts them into the three corresponding motor velocities. A PID control loop is run on the motors for speed control. The output of the PID control is converted to a servo signal and passed to the three motor H-bridges. An encoder is attached to each motor and provides a speed measurement for the Arduino motor controller. Finally, odometry readings calculated by the motor controller are sent back to the Python joystick control program at roughly 10Hz via serial communication.

2.2 Python Joystick Control

The joystick represents position as a floating point number in the range [-1, 1]. The position on the x-y axis of the joystick controls the robot velocity along its x-y axis respectively in proportion to the joystick's position. Twisting the joystick imparts a rotational velocity onto the

robot proportional to the position. Throttling the joystick controls the maximum speed of the robot attenuating the proportionality. The Python module calculates the appropriate x, y, θ velocities based on the position of the joystick and the throttle. These velocities are converted into serial commands to be sent to the motor controller.

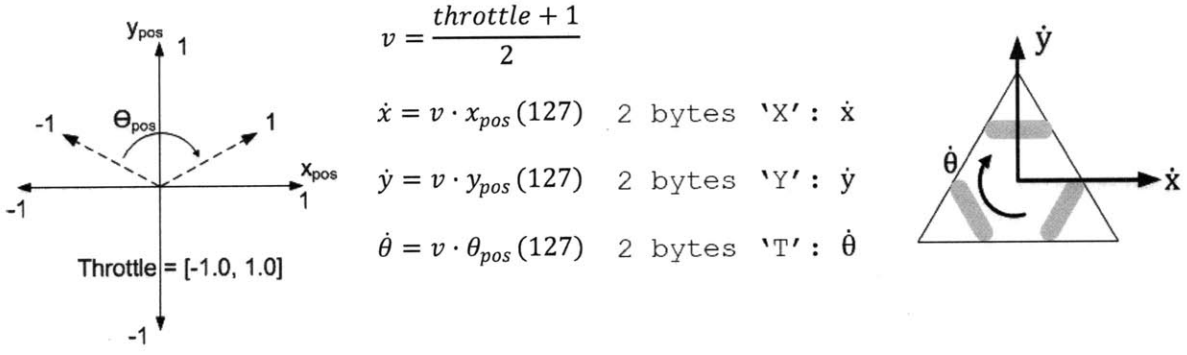
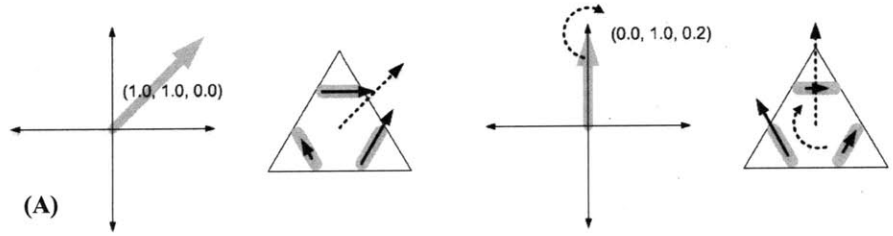


Figure 2.2: Mapping Joystick Position to Robot Velocity. A 3axis joystick maps axis positions to values [-1, 1], these values correspond to x, y, θ velocities and are attenuated by a throttle knob, converted into serial commands, and sent to the robot. The robot has bounded velocities between [-2, 2] m/s and [-pi/2, pi/2] rad/s.



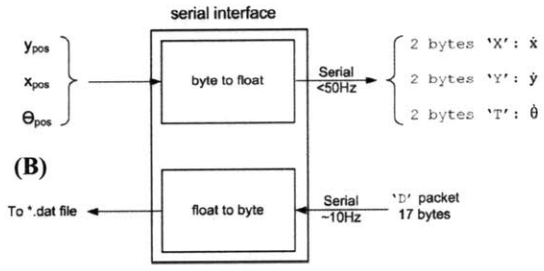


Figure 2.3: Velocity Mapping Example and Python Serial Interface. (A) Example mappings from joystick position to desired robot x, y, θ velocities (B) serial interface module converts joystick positions into serial commands (1byte precharacter, 1 byte value). The interface also parses data packets from the motor controller.

The serial protocol was decided arbitrarily because the motor controller firmware is a custom program. Two bytes are sent to communicate a desired velocity. A velocity pre-character is sent in front of a 1 byte velocity command. The precharacter denotes which velocity x, y or θ is to be

set. The motor controller program checks this pre-character before setting the desired velocity to verify a valid serial command.

'D'	dt (10^6)	$\{\dot{A}, \dot{B}, \dot{C}\}$ (10^3)	$\{\dot{x}, \dot{y}, \dot{\theta}\}$ (10^3)
Precharacter	Elapsed time since last packet (μs)	Measured velocity for each motor (mm/s)	Desired velocity from Python (mm/s)
1 byte	4 bytes	6 bytes 3x(2 bytes)	6 bytes 3x(2 bytes)

Figure 2.4: Odometry Data Packet Protocol. A 'D' is sent as the packet precharacter to validate serial transmission followed by 4 bytes for the elapses time (in microseconds) since the last packet was sent, three 2 byte signed integers for the velocity (in mm/s) for each of the 3 motors, and three 2byte signed integers for the desired x, y, θ velocity at the time this packet was sent.

In addition to sending desired velocities to the motor controller, the Python joystick control program receives odometry data from the motor controller. The motor controller sends data back to the Python program at semi-regular intervals at roughly 10Hz. Data is sent in 17 byte packets. The packet break down is shown in figure 2.4. A packet pre-character is sent as 1 byte , then the time elapsed in micro seconds from the last packet is sent in 4 separate bytes, followed by 2 bytes for each of the 3 measured motor velocities, and 2 bytes for each of the 3 desired velocities in terms of x, y, θ for that time step. Python writes these values to a data file in semi-regular intervals. These odometry measurements can be integrated during post processing to determine the robot position based on the measured velocities and elapsed time (dt).

2.3 Hardware Overview

2.3.1 Arduino Mega, Atmega 1280

The Arduino Mega was chosen for the motor controller for its 54 GPIO. The Arduino Duelmilove has only 12 GPIO. The motor controller requires 6 IO for the encoders (3-AB signal channels), 6 IO for the H-bridges (3-servo signal, 3-safety signal), and more IO for miscellaneous safety and status LEDs.

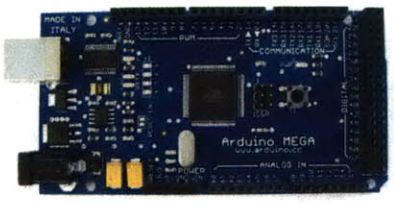


Figure 2.5: The Arduino Mega. Off the shelf microcontroller used for the motor controller.. Image taken from http://arduino.cc/en/Main/ArduinoBoardMega

The native clock on the Arduino Mega can run scheduled interrupts at 32kHz for sampling the encoder signals. However, at 32kHz the interrupts run too frequently for the Arduino to do any useful work. Instead the interrupts are run at 15kHz and a lower resolution encoder was chosen to meet the aliasing requirements.

In general, the Arduino platform was selected to develop the motor controller because of its flexibility and practical debugging. An off the shelf solution for a 3 motor omniwheel controller board was not cost effective, and did not give enough freedom for control and data collection. The available motor control boards were for mixing 4 motors not the desired 3, and these could not perform the motor vectoring calculations needed for the triangular configuration.

2.3.2 Encoders

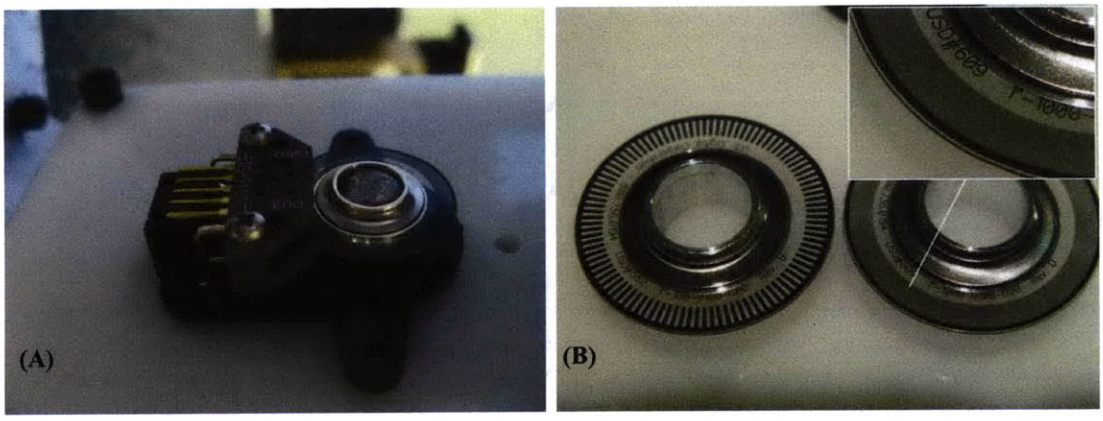


Figure 2.6: Image of Encoder and Disc. (A) The encoder mounted on a gear box shaft. (B) a cross comparison of encoder resolutions at 100ticks/in and 1000ticks/in. The disc with 100ticks/in was used because of limitations on sampling frequency.

The encoders chosen have 100 ticks per inch, and are relatively low resolution for encoders. To determine encoder counts and direction, the typical algorithm samples only a quarter of the available clock edges. A modified algorithm was used to acquire 4 times the native resolution at a cost of 4 times the computation during interrupts.

This is another reason why the 32kHz interrupt rate could not be implemented. At 32kHz the sample rate would be well above Nyquist for the predicted wheel velocities. While the sample quality would be high, the encoder positions would be extremely coarse. It is also impossible to run the modified algorithm at 32kHz because the interrupts are too short. Operating the interrupts at 15kHz sacrifices some sampling quality, but the resolution is increased for a small trade off in computation time.

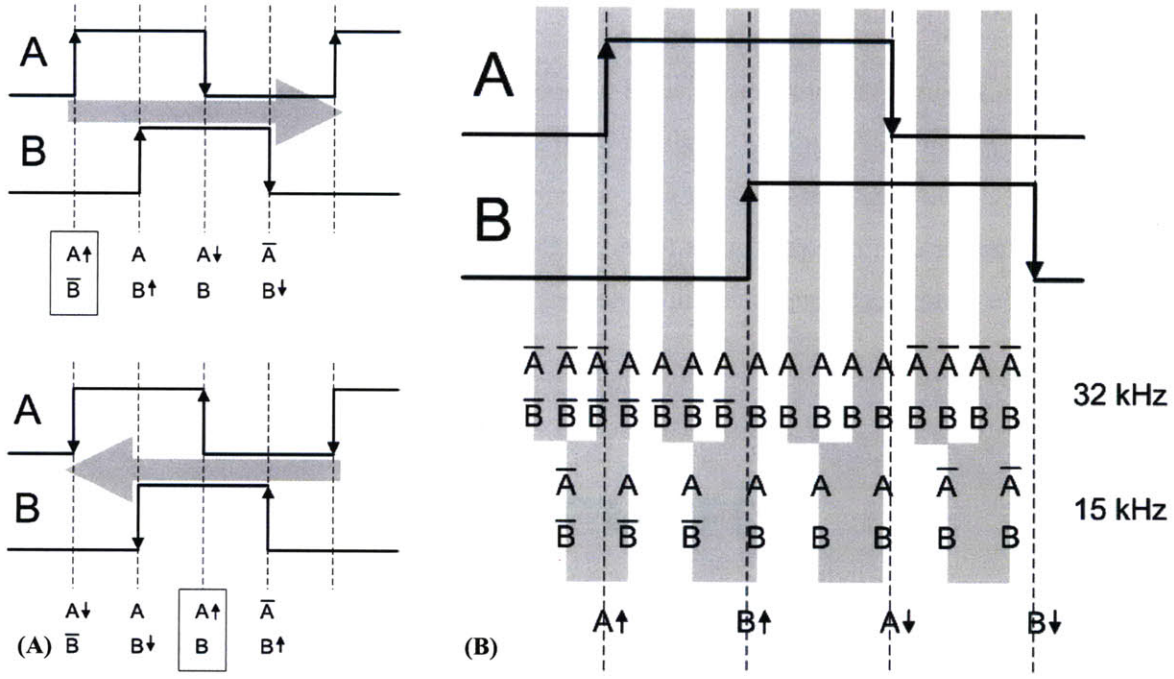


Figure 2.7: Interrupts for Encoder Sampling Illustrating Counting Techniques. (A) corresponding signals for A, B when counting encoder ticks at every edge signal edge. (B) comparison of interrupt frequencies 32kHz and 15kHz, identical counts from signals A, B, but 15kHz frees up twice as much time for other computations.

The typical counting scheme looks only at a single channel for a single type of edge. For example, the interrupt might only look at channel A for rising edges. If A is rising, and B is low

then there is a 'plus' count. If A is rising and B is high then there is a 'minus' count. This is an extremely fast computation, but only a quarter of the clock edges are being used.

The modified scheme uses rising and falling edges of both channels. If A is rising and B is low or if A is falling and B is high then there is a plus count or if B is rising and A is high or if B is falling and A is low then there is also a plus count. The minus count is just a logical negation of the above calculation. This is implemented in code as the following.

```
//edge detection then state detection
if (t_A != e_A){
        if (t_A)
                tick_cur += (t_B ? 1 : -1);
        else
                tick_cur += (t_B ? -1 : 1);
}
if (t_B != e_B){
        if (t_B)
                tick_cur += (t_A ? -1 : 1);
        else
                tick_cur += (t_A ? 1 : -1);
}

e_B = t_B;
e_A = t_A;
```

Here `e_A` and `e_B` are the previous values of the encoder signals A, B from the last sample. The variables `t_A` and `t_B` are the current values of the encoder signal A, B, and `tick_cur` is the current value of the ticks for this encoder. It is reset every time the speed is calculated, and represents the number of ticks that have transpired.

This actually becomes very expensive because of branching in the `if blocks`. There are 4 times as many branches which consume a large amount of processing time. The advantage is an increase in the encoder resolution from 350 ticks per wheel revolution to 1400 ticks per wheel revolution.

2.3.3 H-bridges

The motors are specified are driven at +-24V DC, and can draw at peak loads 30A. To control speed and direction of the drive motors off the shelf H-bridges were selected to drive the robot. The control scheme used for these H-bridges is a standard servo control signal. The typical servo control signal is a single pulse [1-2]ms every 20ms. With a 1ms pulse width, the motor runs full

reverse (-24v), at 1.5ms the motor is completely stopped (0v), and at 2ms the motor runs full forward (+24v). A pulse width in between produces a proportional voltage to the motors in forward or reverse respectively [-24 to +24v]. A convenient servo library can be found with the Arduino, and was used to generate these servo pulse widths with the correct timing. The servo scheme is convenient, because it is low bandwidth, and in a passive on or off state no pulses are being sent to match the servo control signal so the H-bridges default to off.

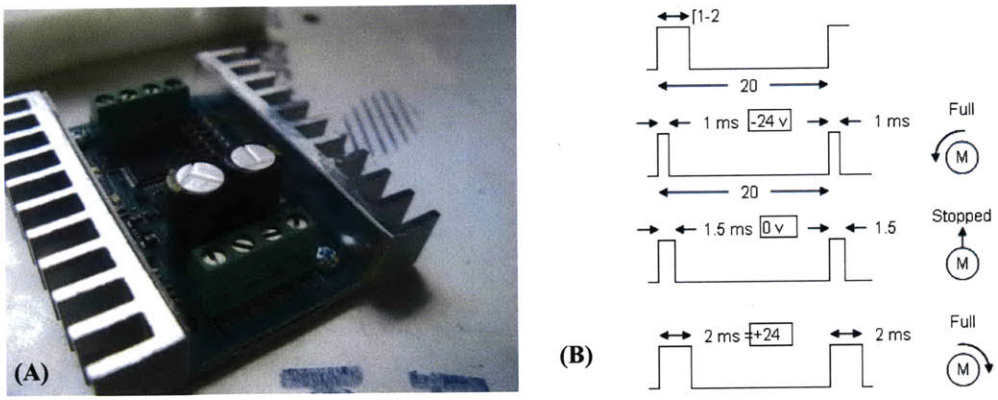


Figure 2.8: H-bridge and Servo Control Signal. (A) a single H-bridge per motor was used to drive the motor speed and direction. (B) standard servo control every 20ms, maps 1ms pulse to -24v in full reverse, 1.5ms pulse to 0v, 2ms to +24v full forward.

The H-bridges also have a number of convenient features to check if they are receiving power which is an important feedback signal to stop the firmware program when the power is removed.

2.4 Motor Controller Firmware

2.4.1 Designing Motor Controller Firmware

The motor controller firmware begins with a serial interface module. When a new desired x, y, θ velocity is received from the Python joystick controller, a vector transform module is called to convert the x, y, θ velocities into the three respective motor velocities. Floating point conversions from single bytes from the serial commands are also performed at this point, and the velocity pre-character check is also made to verify a valid serial command. Once calculated, these three desired floating point motor velocities are sent to the PID module.

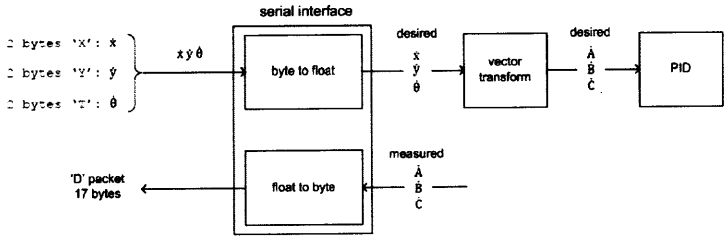


Figure 2.9: Overview of Motor Controller Firmware. Serial commands are parsed by the motor controllers serial interface module, the desired floating point x, y, θ velocities are transformed into motor velocities by a vector transform module, these transformed velocities for motors A, B, C are turned into the set points for the PID controller. The measured motor velocities A, B, C are converted put into a data packet by the serial interface, and returned to the Python joystick controller.

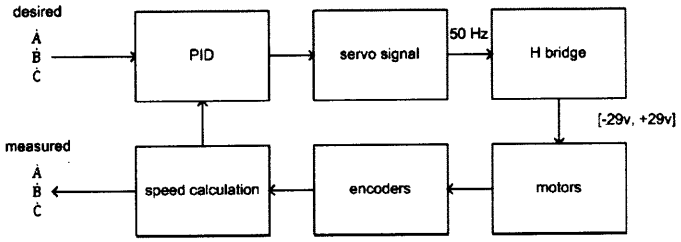


Figure 2.10: Feedback Control Diagram. The desired motor velocities A, B, C are the set points for the PID controller. The PID module calculates the needed motor voltage and changes the servo signal pulse width that maps to this voltage. The H-bridges send this voltage to the motors. Shaft encoders provide position measurements to a shaft speed calculator. Measured motor speeds for A, B, C are sent to the PID for feedback, and to the serial interface to be put into a data packet.

The PID controller makes these new desired motor velocities the set points and controls the voltage to the H-bridges via servo signal to maintain these velocities to the best of its ability. The H-bridges use the servo signal from the PID control module to change the voltage sent to each motor.

A quadrature encoder is attached to the motor shaft to monitor the motor position. The AB signals from the encoder are fed back into the motor controller, and a speed calculator module determines the wheel velocity. This calculated wheel velocity is fed back into the PID controller as the measured wheel velocity. It is also fed back to the serial interface module. Floating point motor speeds are converted back into serial bytes, and a data packet is constructed to be sent back to the Python joystick controller.

2.4.2 *Speed Calculator*

Wheel speed is determined by counting the encoder ticks and direction over a small dt. At walking speed roughly 1.5 m/s the wheels will rotate roughly 5 times per second. With 1400 encoder ticks per wheel revolution this is roughly 7000 ticks per second. Sampling at 15kHz is just slightly above the Nyquist sampling criteria. To this degree, recalculating speed for every count produces and extremely coarse velocity measurement, and is computationally intractable. A suitable velocity sampling dt is chosen to acquire a higher resolution on the velocity measurement. Sampling the encoder counts too quickly creates oscillations in the PID controller because the signal derivative is too high. The velocity measurement approaches a square wave as the dt shrinks to the sampling period. If the velocity measurement dt is too long, the measurements are severely delayed, and the PID controller would not operate with the real time measurements. A suitable dt for calculating speed is every 100 sampling interrupts. This effectively gives a resolution of 100 discrete values for the speed measurement, which is appropriate resolution for PID calculations. If the interrupts sample at 15kHz, the speed calculations occur at 150Hz.

2.4.3 *PID velocity control*

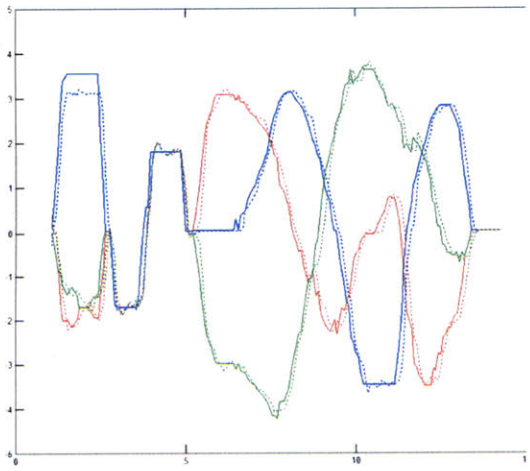


Figure 2.11: PID Response Curve. The solid curve is the desired velocity transformed into motor velocities A, B, C. The dotted line is the measured motor velocity. Oscillations and a small 10ms lag are visible in the response time course.

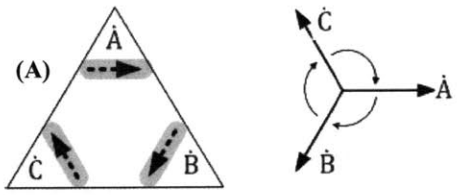
A standard PID control algorithm was implemented to control each motor's speed. Three PID controllers and 3 speed calculators run simultaneously. The PID constants were tuned manually to a desired response minimizing visual oscillation and minimizing settle time. The PID

controller was tested manually with joystick inputs which attenuate the derivative of a step response. The PID response curve, for the constants selected, shows small oscillations on manual step inputs. A small 10ms delay is also visible in the response time for the wheel velocities.

2.4.4 Velocity Transform Equations

The floating point x, y, θ velocities are converted into individual motor velocities by a transformation module. The three motor drive shafts are oriented 120 degrees apart, forming an equilateral triangle. The omniwheels have rollers that spin orthogonally to their axis of rotation. Hence, force can only be transferred orthogonally to the rollers. Applying these three force vectors, a free body diagram can be drawn to represent the robot's motion model.

An input x, y, velocity is transformed into radial coordinates with v and θ. A projection of v is made onto the 3 motor vectors (120 degrees apart) and θ is included as the offset for the projection. This projection transforms the translational x, y velocity into the three motor velocities. Adding the θ velocity as an offset to each of the three motor velocities accounts for the rotational component of velocity.



$$v = \sqrt{\dot{x}^2 + \dot{y}^2}$$

$$\theta = \tan^{-1}\left(\frac{\dot{y}}{\dot{x}}\right)$$

(B) $\dot{A} = v \cdot \cos\left(\frac{\pi}{2} - \theta\right) + \dot{\theta}$ $\dot{B} = v \cdot \cos\left(\frac{11\pi}{6} - \theta\right) + \dot{\theta}$ $\dot{C} = v \cdot \cos\left(\frac{7\pi}{6} - \theta\right) + \dot{\theta}$

Figure 2.12: Vector Transform Diagram for Holonomic Drive. (A) robot free body diagram for the omniwheels. (B) Transform equations for determining desired motor velocities A, B, C

The serial interface on the motor controller is responsible for maintaining communication with the Python joystick controller, or any other interface running the same motor control protocol. While the odometry data packets are sent back at semi-regular intervals (roughly 10Hz), the x, y, θ, velocities set to the serial interface module may be variable. However, if the velocity update packets are sent faster than 50Hz the serial buffer on the Arduino is flooded and over flow occurs. For proper operation, velocity commands must be regulated by the control program.

2.4.5 *Final Event Timing Configuration*

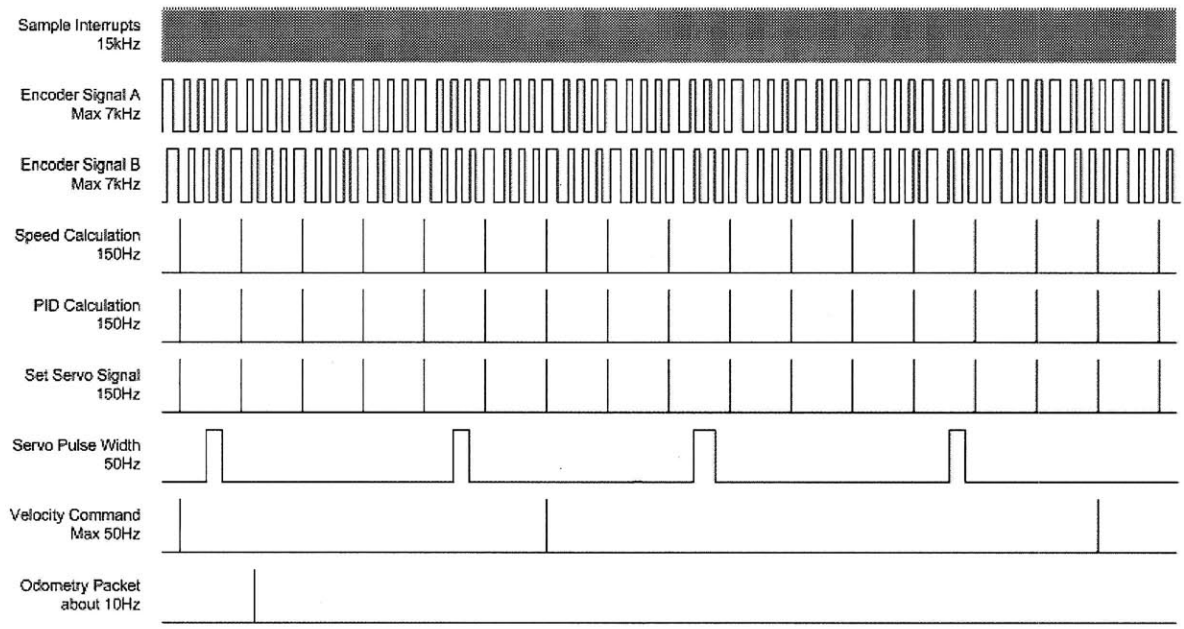


Figure 2.13: Final Event Timing Diagram. Interrupts operate at 15kHz, the encoders signals have a maximum frequency of 7kHz, the speed, PID, and servo pulse width are updated at 150Hz, the servo control signal is at 50Hz, input velocities arrive at < 50Hz, and the odometry data packets are sent at roughly 10Hz.

The final timing schedule is as shown in figure 2.13. Interrupts run on the Arduino at regularly scheduled intervals, sampling at 15kHz. The encoder signals A and B will have variable frequency depending on the motor velocity. The maximum frequency should be around 7kHz. The speed calculator recalculates the rotational speed of each motor every 100 samples. Therefore, speed is recalculated at 150Hz. After recalculating the speed, the PID values are recalculated against the new measured speed, also at 150Hz. The servo pulse width is redefined at this point, again at 150Hz. Finally at 50Hz, the servo signal is sent out to the H-bridges. This timing scheme allows for some buffering around the PID controller depending on when desired velocity updates occur.

A faster response time can be achieved for the desired velocity. The maximum wait time for setting the motor velocity is still 20ms, but the PID controller can settle on a few intermediate measurements before finalizing a servo pulse width to be sent to the H-bridges. If the speed is

recalculated too fast, then the speed measurements will be extremely coarse and the PID controller will have large oscillations attempting to stabilize a high frequency signal.

Finally, the input velocities from the Python joystick controller arrive sporadically at less than 50Hz, and at semi-regular intervals, the motor controller reports the measured velocity for each of the three motors, the current desired x, y, θ velocities, and an elapsed time from the last report. This large data packet is sent infrequently to prevent flooding the serial buffer.

2.5 Odometry Results

2.5.1 Adjusting Odometry Constants

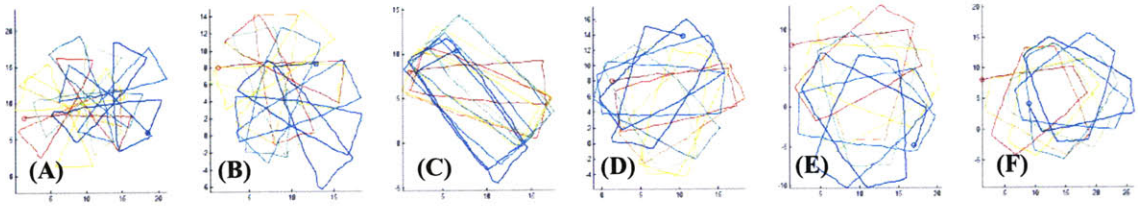


Figure 2.14: Rescaling of Angular Rate. (A-F) Rescaling of Angular rate from 0.5 rad/s to 0.8 rad/s

The odometry velocities reported were not to scale. Small measurement errors in wheel diameter, robot diameter, and turning rates caused the scalars for angular rate and translational rate to be slightly off. In figure 2.14 is a plot of small deviations from the correct angular rate by only a few fractions of rad/s. The odometry quickly twists up when the scalar is slightly off from its true value. Manual tweaking of this value was made to force the odometry plots to somewhat resemble the general path taken. Once determined this angular rate constant was used for all 9 experiments.

For real life implementation, the constant determined in post-processing should be similar to the actual constant. Using this post-processed value should produce desirable results.

2.5.2 *Path plots*

The path plots in figure 2.15 were generated by integrating the robot odometry measurements from the data packets returned by the motor controller. These plots correspond to the 9 experiments laid out in the experimentation description.

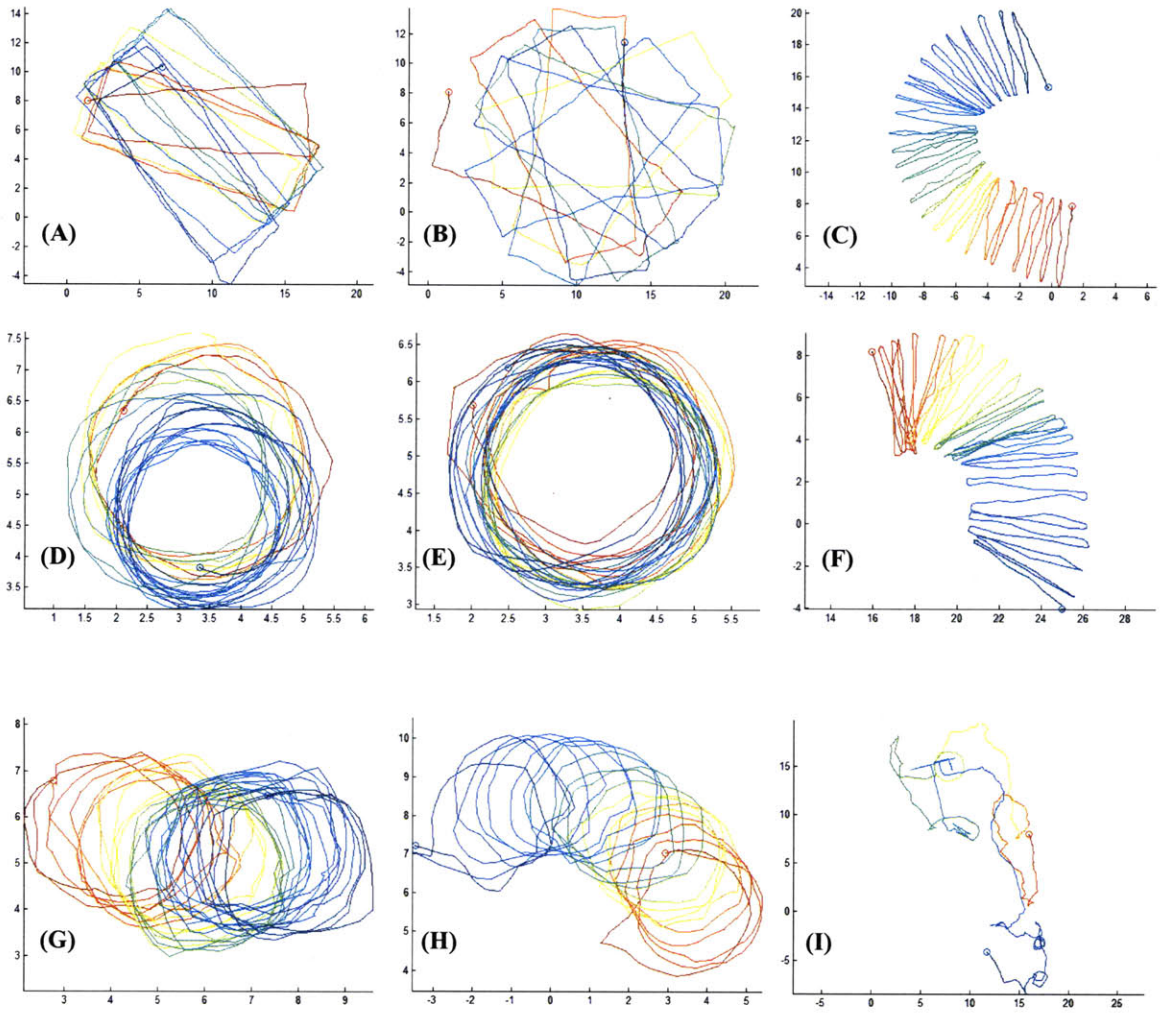


Figure 2.15: Odometry Path Plots for Experiments 1-9. (A) Rectangle CW - 1, (B) Rectangle CCW - 2, (C) Switchback suboptimal - 7, (D) Non-holonomic circles CW - 3, (E) Non-holonomic circles CCW - 4, (F) Switchback optimal - 8, (G) Holonomic Circles CW - 5, (H) Holonomic Circles CCW - 6, (I) Random Track - 9.

2.5.3 *Cross comparisons*

The rectangle in the first experiment (clockwise) is semi-regular with very little angular drift as compared to the counter clockwise rectangle in experiment 2. While part of this phenomenon is due to an adjustment of the angular rate constant, the simple reason for the inconsistency is an asymmetry in the robot mechanics or how the encoders read the wheel velocities. This behavior is, however, consistent with the other odometry plots, the switchback experiments in 7 and 8 have a clockwise skew to them as a result of the odometry considering right turns sharper than left turns.

The non-holonomic circles in experiments 3 and 4 show a tightening of the ring formation for making sharper right turns, and an expansion of the ring for making wider left turns. The holonomic circles in experiments 5 and 6 have a more interesting deviation because with a maintained heading there should be little angular drift hence an asymmetry in the encoders causes the paths to be slightly biased.

2.5.4 *Analysis of Odometry*

In general, the odometry exhibits significant drift. The angular drift is the largest concern because this causes warping for loop closure. Drift in x, y velocities is not immediately apparent over the warping of θ. Experiment 9 is especially skewed and does not fit within the confines of the robot's world. It is apparent that the odometry alone is incapable of accurately representing irregular motions.

However, the odometry is rather smooth. Human errors in driving the robot manually are immediately obvious from the plots, and the deviation from one time step to the next is not severe enough to cause concern. The odometry is accurate over small time steps. There is a low degree of high frequency noise, and only a large amount of drift over an extended period of time.

An immediate solution to solving the issue of drift is to use an absolute positioning system to localize the robot in a global sense in its world.

3 Robot Absolute Positioning

3.1 Introduction to Ubisense

3.1.1 Selecting an absolute positioning system

The shortcomings of the odometry established a need for an absolute positioning system. Several options are available and were considered in the selection process. The most obvious choice would be using GPS.

GPS can be a convenient off the shelf solution for absolute positioning. Depending on the GPS module selected, centimeter accuracy can be achieved at an extremely high bandwidth for a sacrifice in cost. A practical GPS solution for the robot opera would be low bandwidth and likely low accuracy due to cost restrictions and the need to implement 12 modules. However, using GPS indoors in an opera house is of some concern. Localizing with satellites indoors cannot be done reliably and largely depends on the building structure. The opera house in Monaco would need to be tested for compliance. In addition, the more critical issue is accurately determining the robot heading, which GPS alone cannot. Using magnetometers on the GPS for 'true north' may become problematic with 5 large motors onboard each of the 12 robots, and several moving metallic stage elements weighing over 2000lbs.

For the same reasons GPS is impractical, an Inertial Navigation System is also not ideal. An INS typically uses a combination of GPS and magnetometers to update an Extended Kalman Filter. A decent off the shelf INS is also a costly solution. The robot is also holonomic which may need a custom EKF to be written because most INS are designed for UAVs.

An alternative to GPS and an INS is to use beacons for localization, and run an Extended Kalman Filter over the beacon locations to estimate the robot location and heading. Using beacons requires careful setup and installation of several components. Calibration techniques can be arduous, and depending on the type of beacons being used the implementation of the robot localization engine may change. Most beacon systems for robotics are custom built, or used over large areas and are based on ultrasound. Ultrasound has poor reflectance properties, and is subject to the mechanical noise on stage during the performance.

In a similar category with beacons are Ultra Wide Band sensor networks. Radio frequency sensors that operate in the GHz frequency can send extremely short pulse trains out to avoid issues with reflectance than would cause issues for ultrasound. Using Angle Of Arrival (AOA) and Time Difference Of Arrival (TDOA), the sensors can determine the location of a tag. Advantages for using UWB localization include operation indoors, immunity to magnetic noise, and avoidance of reflectance. The disadvantages are precision setup, cost, high noise levels, low bandwidth, line of sight, and localizing tags still does not solve the issue of determining robot heading.

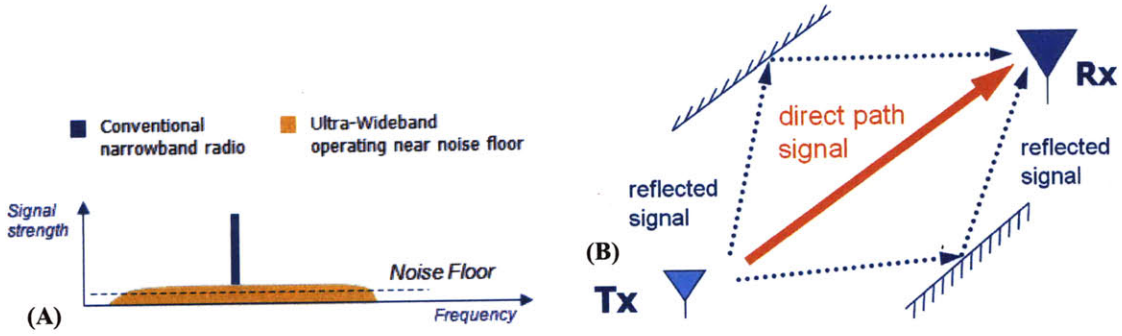
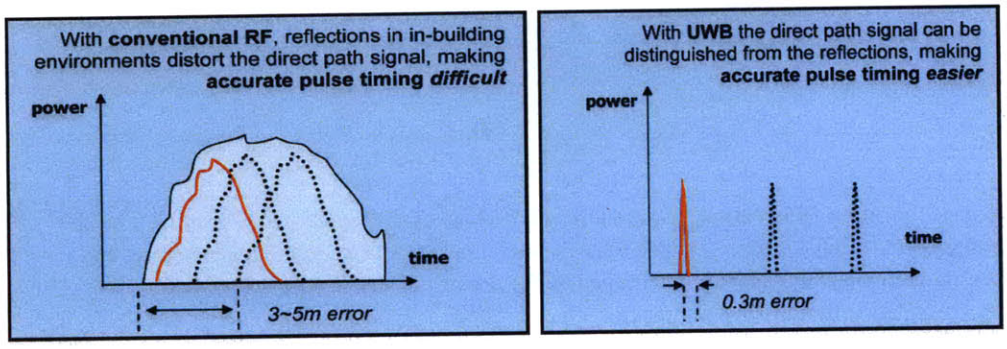


Figure 3.1: Ubisense Noise Model and Reflections. (A) an UWB signal operates just above the noise floor (B) Typical RF communications and Ultrasound are confounded by the effects of reflected signals arriving at the receiver.. Diagrams taken from Ubisense Training Presentation by Chris Doernbrack



(A) (B)

Figure 3.2: Error Analysis for UWB Signals. (A) Typical error associated with conventional RF, (B) Because UWB is high frequency, a very narrow pulse width will capture the full signal. Reflections do not alias with the actual signal because the delay from the reflection is much larger than the pulse width. Diagrams taken from Ubisense Training Presentation by Chris Doernbrack

Ubisense makes an off the shelf UWB tracking system and provides renowned tech support for installation. To fit the strict time constraint, the Ubisense system was chosen because of the rapid installation process. Localization can be accomplished in a day.

3.1.2 Ubisense Technology

Ubisense uses UWB sensors mounted near the ceiling facing into the area of interest. Ubisense tags can be localized while in the line of site of 2 or more sensors. Their location is reported in x, y, z. A master sensor synchronizes the timing around the other slave sensors. Each tag is assigned a slot to send UWB pulses out to the sensors. Using this time slot, two sensors can determine the Time Difference Of Arrival (TDOA) for the pulse. The TDOA is the difference in time of arrival between two sensors for the same tag pulse. The sensor can also determine Angle Of Arrival (AOA) from the tag using internal geometries. Using AOA and TDOA from multiple sensors, the network can determine an accurate tag location, and will then wait for the next tag to send a pulse on the next time slot.

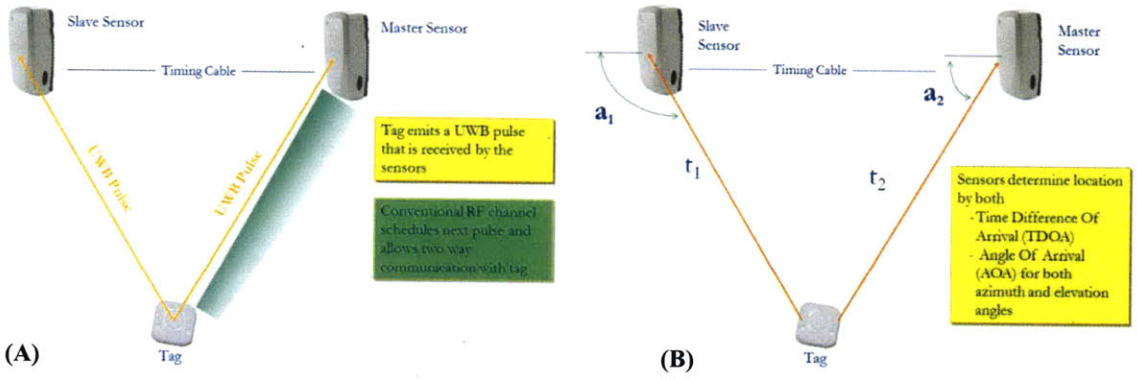


Figure 3.3: Ubisense Operation Illustration. (A) A pulse is broadcast by the tag during a specified time slot and arrives at the master and a slave sensor. (B) The sensors determine AOA using internal geometries, and the signal's TDOA between the two sensors is determined by the timing cable. Diagrams taken from Ubisense Training Presentation by Chris Doernbrack

The TDOA creates a manifold of solutions for the intersection point from the AOA vectors between 2 sensors. Any given sensor produces a set of TDOA manifolds between itself and

neighboring sensors that also picked up the tag pulse. The manifolds from a single sensor are shown below alongside the manifolds for all the sensors. The intersection of these manifolds and the AOA vectors from all the sensors create a least squares solution set.

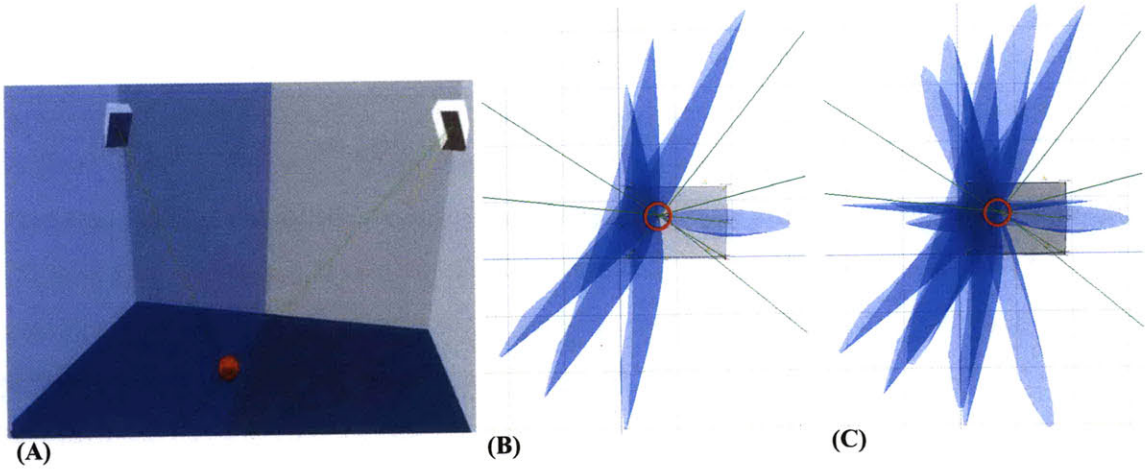


Figure 3.4: Solution Sets. (A) The TDOA creates a manifold between the two sensors. The AOA for each sensor creates an intersection on this manifold that is the location of the tag. (B) The set of manifolds for a single sensor between the other sensors that picked up the tag pulse. (C) The collective group of manifolds for all the sensors for this tag pulse.

3.1.3 Ubisense Tag Update Slots

The Ubisense tags are assigned specific time slots to send out a localizing pulse. There are only 128 time slots per second. The tag will pulse every 4 time slots, but this means the network can only localize 4 tags per second which is impractical considering the number of robots and performers on stage at any given moment. It is more practical to have each tag update every 32 time slots for 4 updates per tag per second, allowing 32 tags on stage at any given time. Updating 4 times per second is a relatively high rate for an absolute positioning system. However, the noise level on the tag is independent of the update rate. A single measurement updating every 4 time slots has the same noise margins as a single measurement every 32 time slots. Averaging measurements over a period of time reduces the standard error, but to accommodate for large number of items on stage at any moment, the rate must be compromised for the number of tags.

3.2 Sensor Network Topology

3.2.1 Sensor Network Layout

The Ubisense network was setup to simulate the conditions in an actual opera house. Optimal placements of the sensors for maximal line of sight could not be made. The sensor's viewing angle is less than 45 degrees. The stage set as shown in figure 3.5 does not provide full coverage for the regions just below the sensors at close range. A more optimal sensor placement would be placing a pair of sensors across the vertical division line to acquire a better line of sight near the edges of the stage.

The rectangle was marked out for experiments 1 and 2 at a location that would pick up a variation in measurement quality. The left side of the rectangle is located in a sub optimal location to test the Ubisense performance. The circle for experiments 3-6 is located in a more optimal location. Experiments 7 and 8 took place on the right and left sides of the rectangle. The left experiment (7) is in a less optimal location. Experiment 9 took place in the world defined by the gray polygon. The robot was temporarily driven to the edges of the sensor network to experiment with the network's performance.

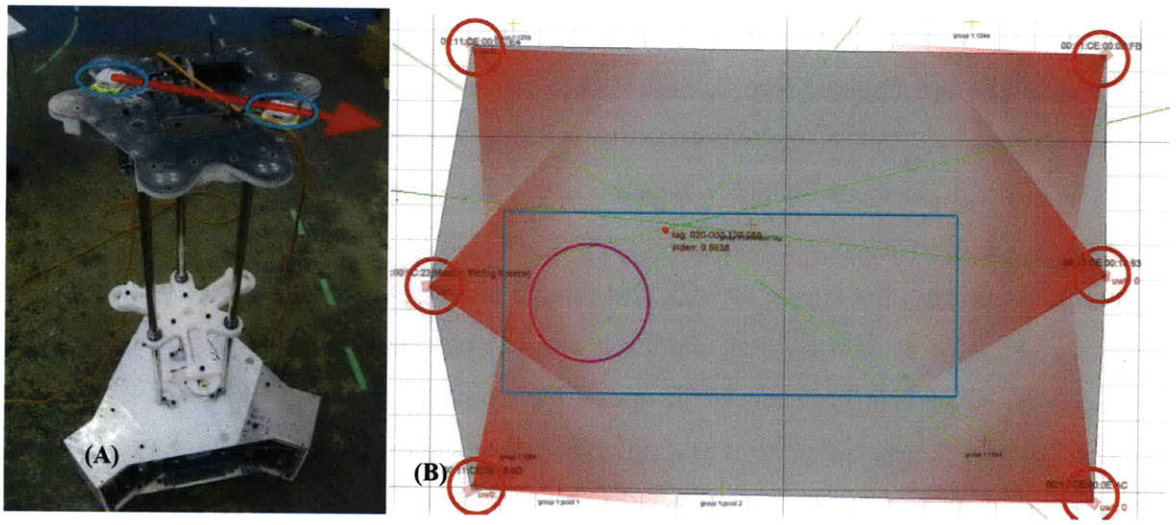


Figure 3.5: Tag and Sensor Geometries. (A) Front and back tag placement on robot canopy, separation is 27cm. (B) Sensor topology in robot world, areas of poor coverage shown in red, portions of experiments were purposefully run on left side of world where there is suboptimal coverage.

3.2.2 Tag Locations on the Robot

To determine heading on the robot, two tags were attached on the front and back of the robot's canopy. The robot orientation can then be calculated by taking the arc tangent of the separation between the two tag locations. The front and back tags are separated by 27cm.

Several issues arise from this configuration. While sound in theory, this two-tag setup has many setbacks in practice. The first obvious issue is the noise margins on both tags confound the angle measurement. There is effectively twice the amount of noise on the heading measurement. In a worst case scenario the front and back tag could even be flipped to produce a completely erroneous heading measurement. In general, this issue can be solved by taking an average of the front and back tags. However, this method causes some lag on the update of the robot heading in proportion to the number of samples used to perform the update.

A more subtle error comes from the tag time-slot assignments. Each tag is assigned a specific time slot to perform and update. This assignment is made as the network comes online. However, the updates for the two tags are not simultaneous; there is a lag between the two measurements equal to the length of the time-slot separation. For one update every 32 time slots this is a separation of 0.25 seconds. If the robot is moving at 1.5 m/s then a 0.25 second delay is an error of nearly 40cm, which is greater than the separation between the front and back tag.

In addition, the tags update at regular intervals, but there is a data association problem. Given a current measurement for the back tag, there is an ambiguity as to whether the corresponding front tag measurement is the measurement just prior or just after this back tag measurement. In fact, neither measurement is appropriate because both are out of sync by up to 0.25 seconds.

To solve this issue the measurements can be interpolated, and an update for this back tag measurement can be performed with a small delay after its arrival. For every new front tag measurement, a simple linear interpolation is run to the last front tag measurement. The midpoint of this interpolation (front' [prime]) and the previous back tag measurement are a delayed data set more closely timed together. This measurement is then used to produce a more accurate measure of heading.

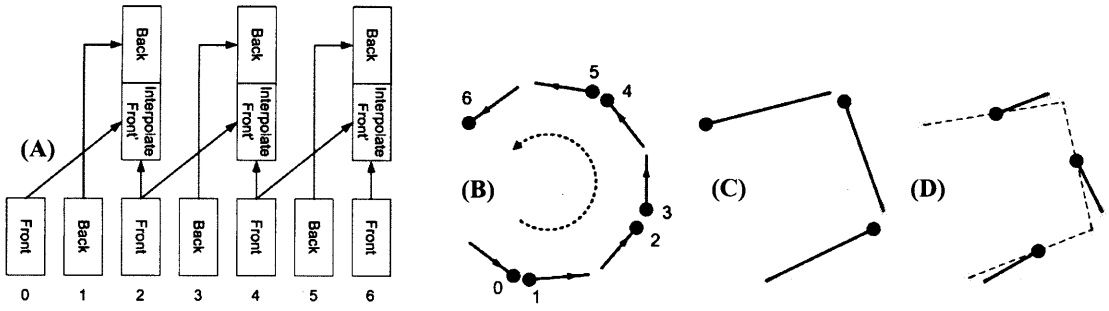


Figure 3.6: Linear Tag Interpolation. (A) Timing sequence of tag interpolation, delayed by one measurement. (B) Illustration of timing diagram for robot moving at constant velocity in a non-holonomic circle. (C) Robot diameters from raw absolute position results. (D) Interpolated robot diameters

3.3 Noise Characteristics

3.3.1 Stationary Tags

The Ubisense noise specifications on tag locations are given as a 15cm error worst case for a still tag. However, it was quickly discovered that this error changes depending on how optimal the tag location is. The 2D histograms in figure 3.7 show a sub optimal location and a more optimal location of the tag standard error. In the sub optimal case, the tag was located in the upper left corner of the rectangle use in experiments 1 and 2. The range of error was nearly 50cm in either direction which is far greater than expected. For localizing position, this is of little consequence, but for a measurement on heading this is far too large for a decent measurement. The front and back tag separation is half of the noise range.

In a more optimal placement of the tag, in the upper right corner of the rectangle used for experiments 1 and 2, the tag showed a significant improvement in the error range. The spread was bounded to roughly 20cm in both x and y.

Unfortunately, the tags will not always be in an optimal location. The histogram below shows the deviation in initial pose angle of the robot while sitting still in a sub optimal location. The standard error is less than 4 degrees which is a decent measurement for a stationary tag. For a stationary tag, the noise margins are largely dependent on the location of the tag in the world. Areas with weak coverage will have larger noise margins. Over a long period of time for a still

tag, an accurate measurement of heading can be produced. However, for high frequency sampling of heading, a single measurement of heading may be severely incorrect.

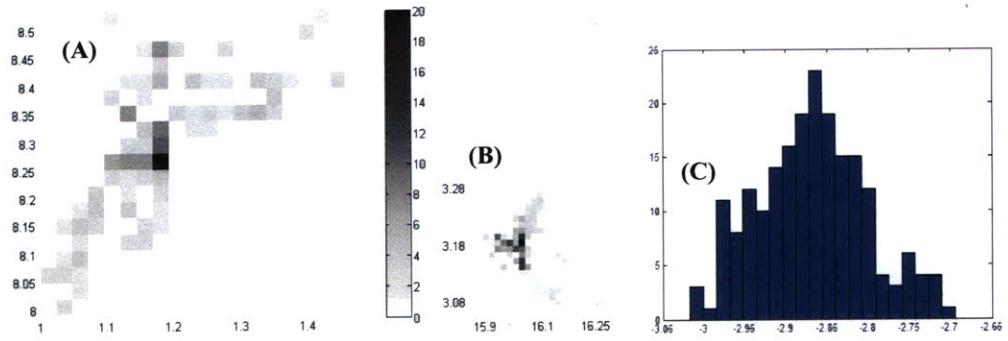


Figure 3.7: Ubisense x, y, θ Histograms for Stationary Tag Noise. (A) Suboptimal tag location with error range of 0.5m. (B) More optimal tag location with error range of 0.2m (C) The standard error for a stationary tag is 3.8 degrees. Stationary measurements have narrow error margins.

3.3.2 Moving Tags

Stationary tags produce a very tight bound on the noise margins. The path plots in figure 3.8 are the results from the 9 experiments using only the Ubisense data.

Over a long period of time and several laps the Ubisense data produces an accurate path taken by the robot around the different tracks. The drift that occurred from the odometry data is no longer an issue. There is a high frequency noise component that is difficult to discern with the path plots shown. Looking carefully at the ring formations, there are a number of jagged lines and a few outlying data points from the Ubisense noise. Most of the deviation in the plots is due to human error driving around the track. This is especially evident in experiment 3 where a learning curve took place to drive the robot in non-holonomic rings.

Averaging each of the experiment paths together would effectively produce the track used to generate the path plots. Unfortunately, there needs to be live state estimation for the robot, and it is unlikely that the robot will repeat a path multiple times to localize itself. A more accurate representation of a typical robot path would be in experiment 9. Close inspection reveals a number of sharp edges due to the high frequency noise produced by the Ubisense localization.

Using a low pass filter on this data set may remove some of these high frequency kinks, but there will be a delay in the localization engine as a result, and some of the sharp edges may not need to be filtered out.

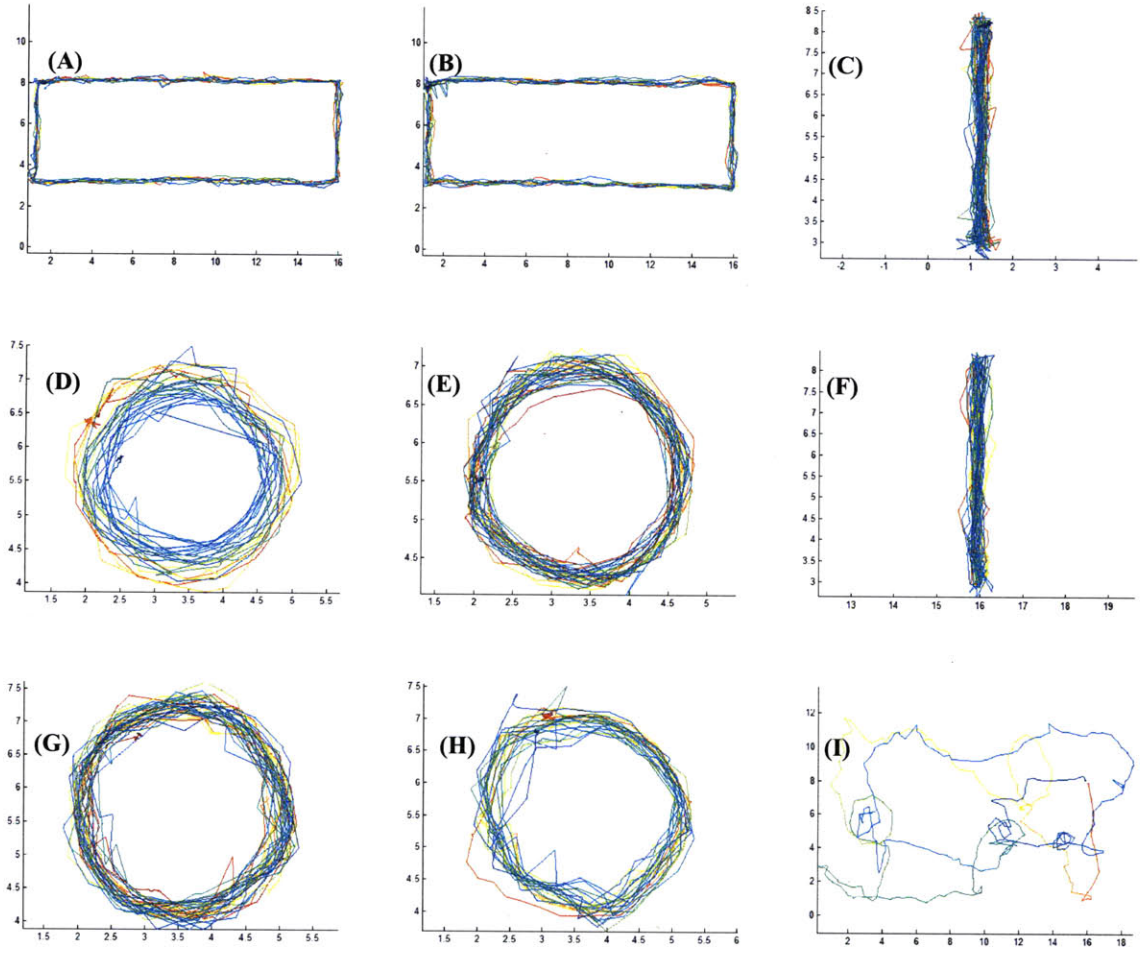


Figure 3.8: Ubisense Absolute Positioning Path Plots for Experiments 1-9. (A) Rectangle CW - 1, (B) Rectangle CCW - 2, (C) Switchback suboptimal - 7, (D) Non-holonomic circles CW - 3, (E) Non-holonomic circles CCW - 4, (F) Switchback optimal - 8, (G) Holonomic Circles CW - 5, (H) Holonomic Circles CCW - 6, (I) Random Track - 9.

The final histogram in figure 3.9 is an estimate of the heading for the robot driving in holonomic circles maintaining a constant heading during experiment 5. Comparing this histogram to the

histogram for the stationary measurement of heading, the standard error is 31 degrees, which is almost a magnitude greater than the standard error of the stationary robot's heading.

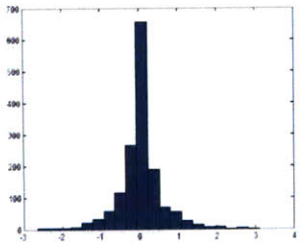


Figure 3.9: Ubisense Measurement of θ for Holonomic Circles Unfiltered. While maintaining a constant heading and driving in holonomic circles, the θ measurement had a standard error of 31.5 degrees.

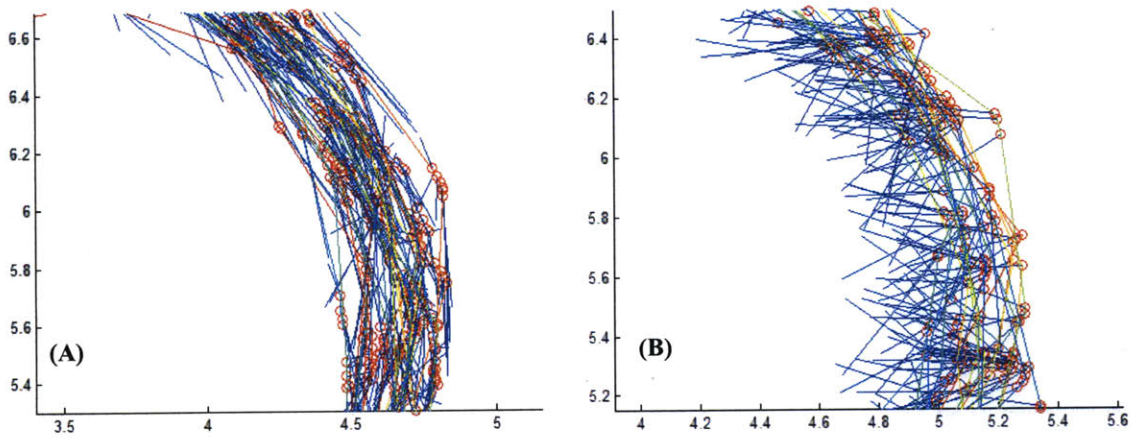


Figure 3.10: Comparison of Ubisense Measurements for Robot Vectors. (A) Non-holonomic CCW circles have robot vectors roughly tangent to velocity. (B) Holonomic CCW circles with maintained heading have robot vectors at 0 degrees with significant noise.

Comparing non-holonomic to holonomic circles the sources of error become clearer. In figure 3.10, the robot is plotted as a vector. The front tag is circled in red, and the back tag extends along the blue line segment. There are several vectors that appear out of angle in the holonomic circle. In the holonomic circle the robot vectors should all be horizontal at a 0 degree heading. While the deviations could be an artifact of the tag interpolation described in 3.2.2, this could also be a more accurate representation of the noise on the robot heading for a moving tag. Position errors in tag measurements are correlated with the robot x,y velocity. For non-holonomic circles, the robot heading is always parallel to the velocity. Errors change the distance

between the tags, while in non-holonomic circles the error can directly affect the angle between tags when the heading is orthogonal to the motion.

3.4 Filtering Tags

A direct solution to the Ubisense noise problem is to filter out poor tag measurements. The criteria for filtering the tag measurements should be a function of the 'goodness' of a measurement. The distance between the front and back tag can be used as a measure of goodness. This known distance of 27cm should be consistent with the distance between the front and back tag measurements reported by the Ubisense network. The histogram in figure 3.11 is the distribution of reported robot diameters (distance between the front and back tag) for all 9 experiments using over 20 000 data points.

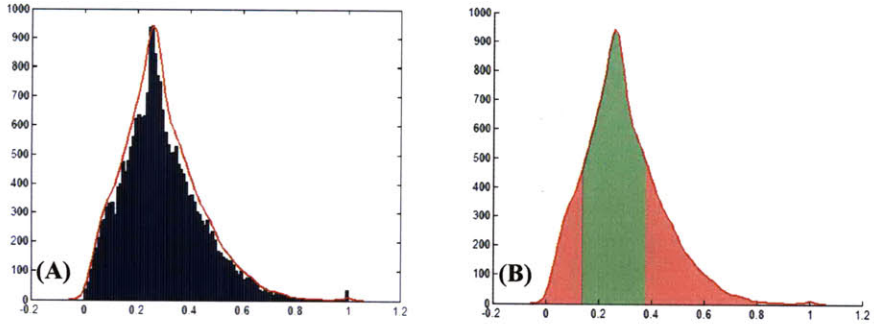


Figure 3.11: Histogram for Ubisense Robot Diameter Measurements. (A) Diameters from experiments 1-9, over 20 000 data points, standard deviation 15cm. (B) Illustration of filtering criteria for 'good' measurements that fall into one standard error from diameter distribution.

A probability distribution function is fit over the diameter distribution and can be used to filter tag measurements. A good measurement should occur inside one standard deviation (15cm) from the mean (27cm). Diameter measurements outside this standard error could be erroneous because the reported diameter is either too large or too small. Bad measurements are thrown out to avoid incorrectly reporting the robot heading.

The robot vectors shown on the left are the measurements that fell under one standard error of the expected robot diameter. The vectors on the right are the measurements that were thrown out. The filtered holonomic circle appears a little tighter, but only 70% of the measurements are used.

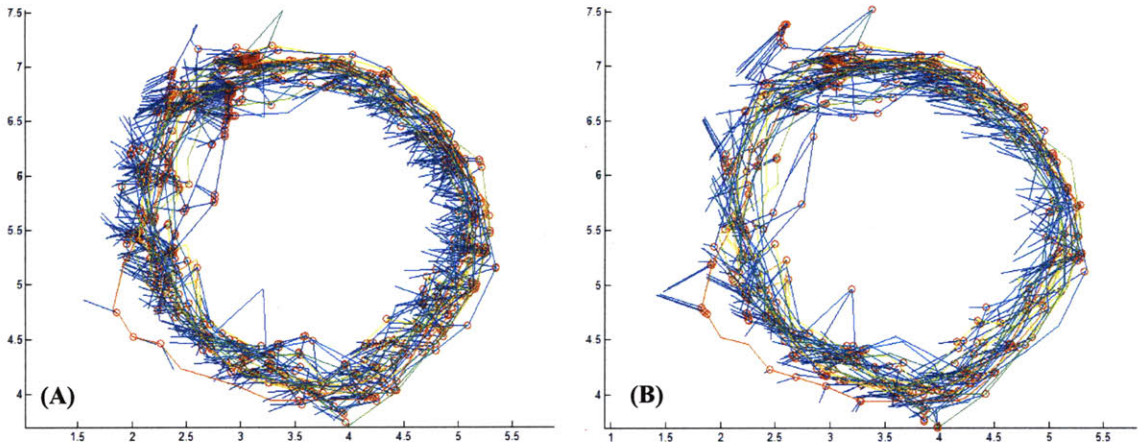


Figure 3.12: Ubisense Robot Vectors Filtered and Removed for Holonomic CCW Circles. (A) Robot vectors that were preserved after filtering corresponding robot diameters by one standard error criteria. (B) Robot vectors that were removed by filtering process

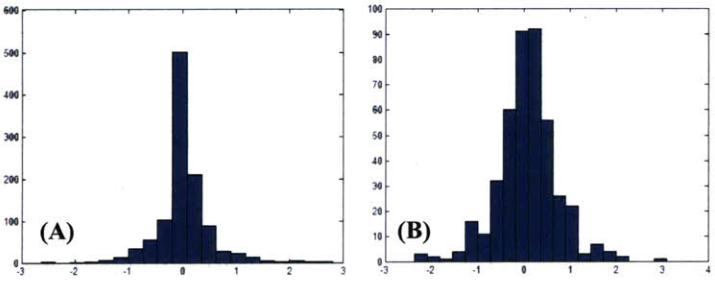


Figure 3.13: Histograms for Filtered and Removed Ubisense Measured θ for Holonomic CCW Circles. (A) The θ measurements filtered by the one standard deviation criteria have a standard error of 28.5 degrees. (B) The θ measurements that were removed by the filtering criteria have a standard error of 37.9 degrees.

The histograms in figure 3.13 show the improvement in the distribution of the robot heading. The standard error for heading using unfiltered data was about 31 degrees. Using only filtered measurements, the standard error shrinks to 28 degrees, and the measurements that were thrown out have a standard error of 38 degrees. The improvement is minor, but can improve the observation model giving the Ubisense measurements a little more weight.

3.5 Synchronizing Odometry and Absolute Positioning

3.5.1 Adding Offsets

A transform must be made to make a cross comparison between the odometry and the Ubisense data. The initial pose for the robot in the odometry setting is unknown. Therefore comparing the position in x and y over time is not possible because the measurement of θ is not correct. The odometry data also has clock skew because it was run on another machine.

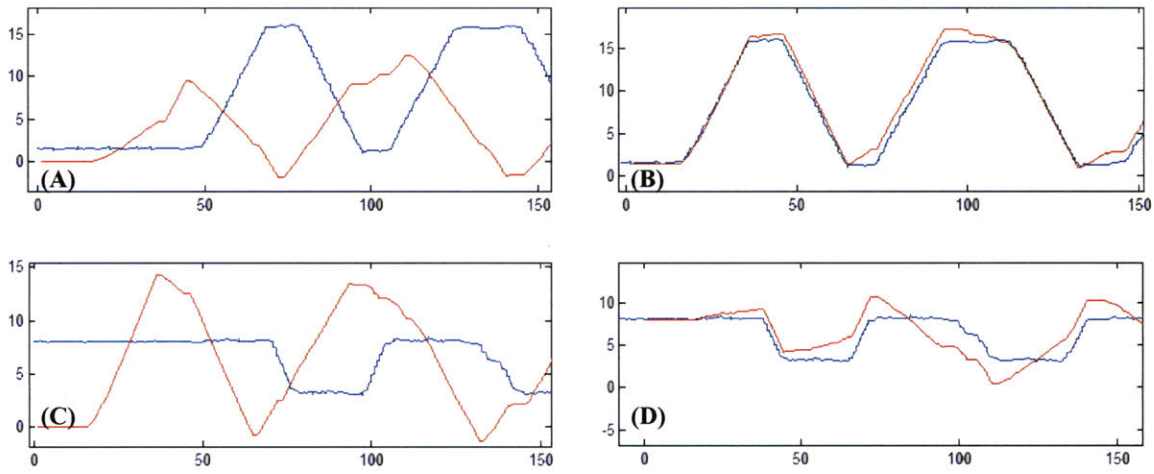


Figure 3.14: Pre and Post Offsets for Combining Ubisense and Odometry Data. (A-B) Plot of x position time course before and after adding initial pose offset for x and θ and adding clock skew. (C-D) Plot of y position time course before and after adding initial pose offset for y and θ and adding clock skew.

To solve these issues, the initial pose from the Ubisense data was used to prime the odometry localization engine. While this made the initial features of the x and y positions similar, there was still clock skew that needed to be added in to line up the features. Manual addition of the clock skew was performed to align the initial data features. The skew was determined by a visual inspection of the initial features and taking the time difference. Because the odometry tends to degrade over time, lining up features later in the time course was not effective for synchronizing the data sets. The plots in figure 3.14 show before and after synchronization of the odometry and Ubisense data for the first few features in the x, y time course.

3.5.2 *Warping Time*

Unfortunately, another error in the odometry was discovered when lining up the data sets. The elapsed time between data packets sent to the Python joystick controller are accumulated as the event time for the odometry data set. There was a small accumulation error for the time it takes to write the data packet that was not properly included in the time stamps. This produces the time warp of the odometry data seen in figure 3.15.

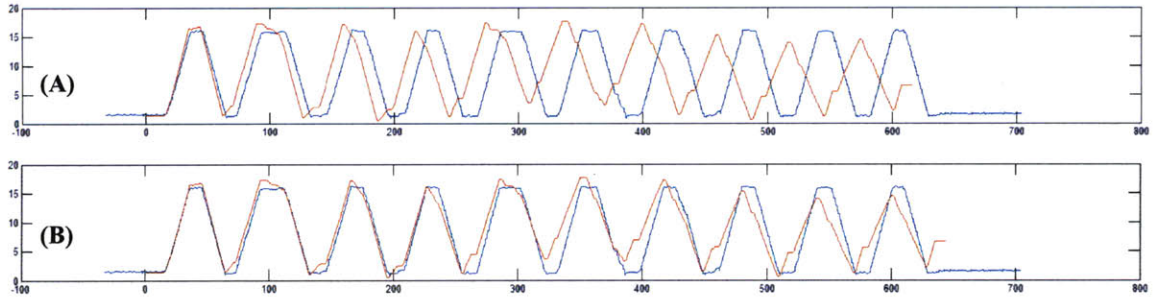


Figure 3.15: Removing the Odometry Time Warping. (A) Plot of warped time course for x position odometry time course faster than Ubisense time course. (B) Rescaling odometry time course by adding 3ms offset for each odometry measurement

The odometry data features line up for the first few features, but the data set falls short of the Ubisense data set features because of the time warp incurred by the accumulation error. In post processing it was determined that this small error in elapsed time is 3ms for each odometry measurement, and was added into the odometry data set.

In a real-life implementation, this time warping would still be present, but the integration of the reported velocities would have just been off by 3ms, which is roughly half a centimeter per odometry update. The data sets would still line up without the need to warp the data because new odometry and new Ubisense data would arrive at the localization engine at the same time and not fall out of sync over time.

3.6 Preliminary Combined Results

The results shown are a cross comparison between the odometry data and the Ubisense absolute positioning data for experiment 1. As mentioned earlier, the first few features of both data sets line up in the x, y time courses. As time progresses, these features tend to slip out of sync because of the accumulated error in the estimation of θ for the robot. As the robot begins to rotate out of phase, the x and y time courses begin to shift, and the peaks do not line up. For experiment 1, this is especially apparent in the time course for the y position of the robot.

A more interesting comparison is in the actual measurement of θ. The plots for sin(θ) and cos(θ) are shown. The Ubisense θ measurement is nearly unintelligible because of the high frequency noise. The odometry measurement appears cleaner, but there is no verification as to whether or not this measurement is becoming more offset as time progresses.

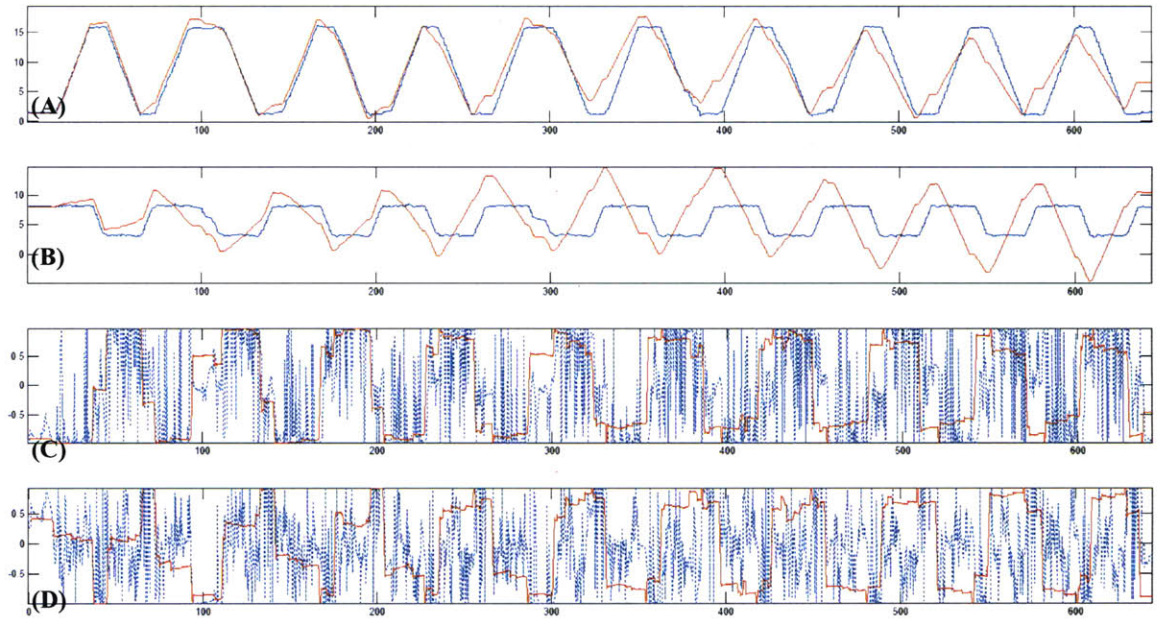


Figure 3.16: Scaled and Offset Ubisense and Odometry Time Courses for Rectangular CW Track. (A) x time course. (B) y time course. (C) sin(θ) time course. (D) cos(θ) time course.

Upon closer inspection for a short time course of 100 seconds, the noise from the Ubisense is cleaned up a small amount, and the profile of what the θ measurement should be becomes

clearer. High frequency spikes litter the measurement of θ from tag measurement noise. The odometry measurement of θ is much cleaner, but already appears to be shifting slightly out of phase with the Ubisense measurement from the scaling error described in section 2.5.1.

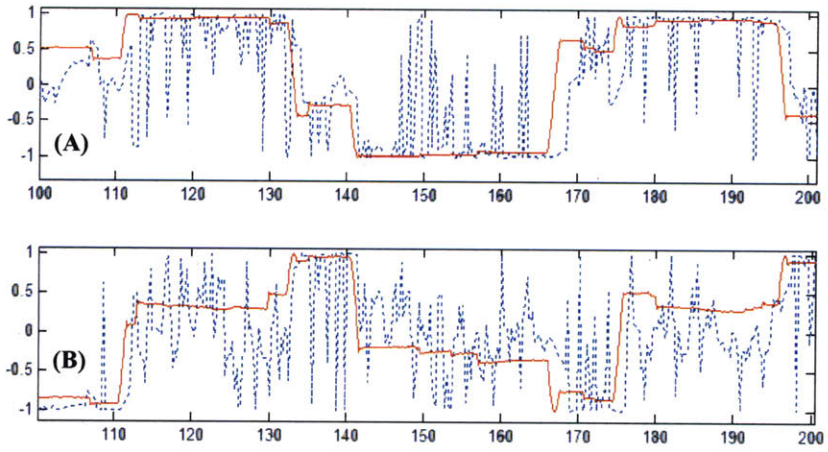


Figure 3.17: Cross Section of Sin(θ) Cos(θ) Time Courses (100-200s) for Rectangular CW Track. (A) sin(θ) time course showing Ubisense high frequency noise. (B) cos(θ) time course

An ideal solution would be to mix the Ubisense absolute positioning data with the odometry data. The goal for mixing would be to remove the high frequency noise from the Ubisense measurements of θ, but use the accuracy of the odometry over short time intervals. The absolute positioning data from the Ubisense would also be able to remove the drift from the odometry due to the accumulation of integration errors.

4 Particle Filter for State Estimation

4.1 Particle Filter or Extended Kalman Filter

The two canonical sensor-fusion algorithms used to solve state estimation are the Kalman Filter and the Particle filter. The obvious choice for solving this particular state estimation problem is the implementation of an Extended Kalman Filter, but without having the system dynamics and

noise characteristics well defined, the EKF can be challenging to implement especially if the state distribution is highly non-Gaussian. The Particle Filter allows for more flexibility in adjusting noise parameters for estimating state, and performs better for non Gaussian belief distributions.

4.1.1 Advantages and Disadvantages

The Particle Filter is a fast straight forward implementation. Increasing the number of particles used to estimate state quickly augments the filter's performance. An EKF lacks the flexibility to increase performance by changing a single parameter. More particles will always produce a more accurate solution. Changing the state model in a Particle Filter is also straight forward. An EKF would need a recalculation of the Jacobians, which must be done analytically by hand. In terms of a proof of concept for estimating state, given the Ubisense and odometry inputs the Particle Filter will perform on par with an EKF with less work. With the time constraints, the Particle Filter is a faster implementation and easier to debug. The Particle Filter also provides a visualization of the non-linear belief distribution for the robot state.

The Particle Filter does have a number of shortcomings. It is computationally expensive in comparison to an EKF. The EKF would only be a handful of states for the estimate of the robot position, velocity, and sensor biases. The Particle Filter requires several hundred particles to calculate its belief state. Propagating several hundred states and sampling the distribution several times per second is computationally expensive. For real life implementation, a Particle Filter may not be as practical, but again it would be faster to implement without the need for external BLAS libraries.

The Particle Filter also has unbounded performance criteria. While changing the number of particles can quickly increase the filter's performance, there is no metric to determine what number of particles is the right number to model the belief distribution. The Particle Filter is also a stochastic model, and from one process to the next the same data may not produce the same traces. An EKF is a far more analytic solution and will produce the same solution set for a given set of inputs. This makes the Particle Filter performance much more subjective in its analysis.

4.1.2 Particle Filter Implementation

The Particle Filter implemented in this research was written in MATLAB for post-processing of the odometry and Ubisense absolute positioning data collected in experiments 1-9. The data updates are slow enough that the filter had no problem running faster than real time even with 3000 particles. This is reassuring if the Particle Filter were to be implemented in real life onboard the robot. The robots run OLPCs (One Laptop Per Child). The computational power of the OLPCs is extremely limited, and the trade off for computational complexity may make the EKF a more practical solution for the final implementation. However, the results from the Particle Filter are promising in terms of successfully localizing the robot given the current odometry system fused with the Ubisense absolute positioning.

4.2 Characterizing the filter

4.2.1 Representation of Belief

The Particle Filter models the uncertainty in its state with a belief distribution. This belief distribution is the cloud of particles. Each particle is a guess for the robot's current state. The particle's state, like the robot's state, contains three fields: x, y, and θ. Instead of one guess for where the robot is located and associating an uncertainty with that single guess, hundreds of guesses are made. These guesses form the particle cloud, which is the belief distribution. The density of the cloud determines how much the filter believes the robot is in that particular state. Higher density regions in the cloud represent a higher probability of having one of those particles' states as the robot's actual state. A low density region, or a single particle one its own away from the distribution's center of mass, would be a less likely estimate of the robot's state.

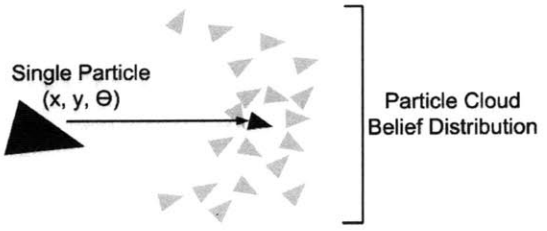


Figure 4.1: Belief Representation with Particle Cloud. A single particle has fields x, y, θ. Several particles represent the belief distribution dense regions correspond to high probability.

With fewer particles, the belief distribution is very coarse. However, as the number of particles increases the particle cloud has a higher resolution and can more accurately represent the robot's belief distribution. With infinite particles, the cloud would be continuous and truly model the

robot's belief distribution as the actual probability distribution function. This is computationally intractable. Only a few hundred particles are typically needed to estimate the belief distribution with enough resolution to properly estimate the robot's state.

$$P_i = (x_i, y_i, \theta_i)$$

$$\mu_x = \frac{1}{k}\sum_{i}^{k} x_i$$

$$\mu_\theta = \tan^{-1}\left(\frac{\sum_i^k \sin\theta_i}{\sum_i^k \cos\theta_i}\right)$$

$$\mu_y = \frac{1}{k}\sum_{i}^{k} y_i$$

By taking the mean of the distribution across the particles' states, the robot's state can be estimated. While the robot's actual state is a distribution across the particle cloud, the mean of the cloud across the particles provides a decent representation for quickly visualizing a summary of the robot's state estimate. These means can be cross referenced with the Ubisense absolute positions and the odometry to compare the Particle Filter's performance.

4.2.2 Representation of State

One design decision in the implementation of the Particle Filter was the representation of the robot state. Because the robot is holonomic its x, y, and θ should be independent states, and could possibly be each represented as 3 separate particle clouds. This keeps the distributions independent. Keeping the distributions independent could increase the performance. The belief distributions on x and y should be tighter than the distribution of θ for the robot. More particles can be dedicated to the estimation of θ, and less computational effort can be spent estimating the robot's x and y.

However, the x, y, and θ are not independent because of the odometry calculations. The robot's position in x and y is determined by its orientation θ, and the integration of the velocity with respect to this θ. The particles must each be a representation of x, y, and θ together because the effects of θ are coupled into the calculation of x and y. To note, a good estimate of θ will also

produce a good estimate of x and y. Conversely, if x and y are well estimated, but θ is a poor estimate, than when propagated this particle will propagate in the wrong direction.

Because the Particle Filter is flexible in its representation of state, both schemes were tested and the decoupling of x, y and θ failed to provide a decent estimate of the robot's state. Having the state with three fields provided a more consistent estimate of the robot's state.

4.2.3 Selection of Particles

As the Particle Filter runs, the number of particles in the filter remains the same. Once the number of particles has been chosen to reflect the resolution of the belief distribution, the number of particles in the filter remains constant. The filter adapts to measurements by choosing to propagate the 'good' particles that accurately represent the belief distribution and throw out 'bad' particles that do not match the measurements. This process of reselection and propagation is critical to the Particle Filter's update.

4.3 Particle Filter Update

4.3.1 Updating cloud from odometry measurements

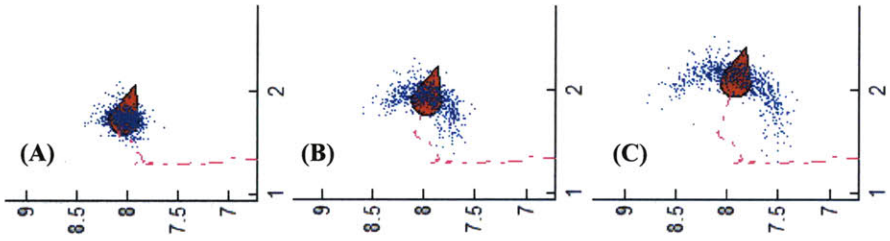


Figure 4.2: Transition Update with Odometry. (A-C) Odometry updates increase the spread of the belief distribution.

An initial cloud of particles represents the robot's initial belief distribution. When an odometry update is received, each particle in the cloud is updated to reflect this transition. Given x, y, and θ velocities and dt from the odometry, the particle will integrate its current state and move forward. When all the particles in the cloud have been integrated with dt, the belief state has performed a transition update.

$$\dot{x} = \dot{A}\cos(0) + \dot{B}\cos\left(\frac{4\pi}{3}\right) + \dot{C}\cos\left(\frac{2\pi}{3}\right) + N(0, \sigma_{\dot{x}})$$

$$\dot{y} = \dot{A}\sin(0) + \dot{B}\sin\left(\frac{4\pi}{3}\right) + \dot{C}\sin\left(\frac{2\pi}{3}\right) + N(0, \sigma_{\dot{y}})$$

$$\dot{\theta} = \frac{\dot{A} + \dot{B} + \dot{C}}{r} + N(0, \sigma_{\dot{\theta}})$$

$$x_t = x_{t-1} + \dot{x}_{t-1}dt$$

$$y_t = y_{t-1} + \dot{y}_{t-1}dt$$

$$\theta_t = \tan^{-1}\left(\frac{\sin(\theta_{t-1} + \dot{\theta}_{t-1}dt)}{\cos(\theta_{t-1} + \dot{\theta}_{t-1}dt)}\right)$$

The odometry data does contain some amount of noise and is far from perfect. To model this noise, some Gaussian noise is added to every measurement before the transition update occurs. For each individual particle the x, y and θ velocities are given a small amount of Gaussian process noise before performing the integration. From the odometry plots it was evident that the noise of θ was larger than the noise from x and y. Therefore, the θ velocity process noise has a larger standard deviation than x and y.

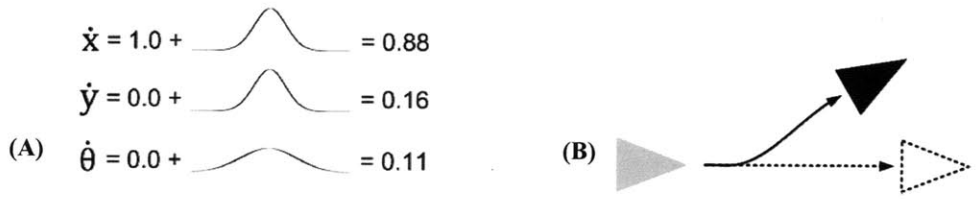


Figure 4.3: Adding Process Noise to Odometry Measurements. (A) Gaussian noise is added to the measured x, y, θ velocity to model the uncertainty in these odometry measurements. (B) Illustration of process noise effects on a single particle

As the particle cloud performs more and more transition updates from the odometry, the belief distribution spreads out. The robot becomes more uncertain about its state. For a small time scale the odometry is a very good estimate for the robot's state, but over time this estimate degrades due to drift. The spread of the belief distribution for each odometry update represents this increase in uncertainty.

The cloud becomes larger due to the process noise. This more accurately reflects the robot's true belief for its state. The longer the odometry runs, the more uncertain its estimate for the robot location becomes. After running for several time steps, the belief distribution becomes quite large. To decrease uncertainty, a measurement must be made.

4.3.2 Updating cloud from Absolute Positioning measurements

The absolute positions from Ubisense give a direct measurement of how well the particles are estimating the robot's belief distribution. As the cloud increases in size from the odometry, it is uncertain which of the particles in the cloud are accurately representing the robot's true position. With an update from Ubisense, the particles that closely match this measurement are the best estimates of the robot's belief state.

$$w(x_{1:k}) = \frac{Pr(Z = z_k | X = x_i)Pr(X = x_i)}{\sum_i^k Pr(Z = z_k | X = x_i)Pr(X = x_i)}$$

$$Pr(X = x_i) = \frac{1}{k}$$

$$w(x_{i=1:k}) = \frac{Pr(Z = z_k | X = x_i)}{\sum_i^k Pr(Z = z_k | X = x_i)}$$

A simplified Bayesian update can be made to give a higher weight to the particles that best represent the robot's belief distribution. Measurements of x, y, and θ are evaluated independently. Each field of the particle state has its own observation model. The weight of a particular x_i (particle's position in x) is determined by a normal distribution with the measurement z_x (measurement of the real robot's x) as the mean and some standard error.

A particle closer to the observation z will receive a higher weight, because it is a better estimate of the robot's belief distribution. Particles that are far away from measurements do not accurately reflect the robot's belief.

One important aspect to the observation model for x, y, and θ is determining the standard error for the weight function. A measurement of x and y is more certain than a measurement of θ. Therefore, it should be probable that a particle's x and y might be close to the measurement.

Particles that are near x and y have accurately modeled the distribution for x, y and θ. However, the measurement of θ is not very accurate as seen in the results from the Ubisense experiments. Particles that have a θ near the measured θ are not necessarily correct due to the high frequency noise on the θ measurements. Good particles should not be assigned low weights because of a bad measurement of θ. To account for this, the standard error on the θ measurement is much larger than for x and y.

Another factor in determining the 'goodness' of a measurement is to take into account the measured robot diameter. If a measurement arrives with the front and back tags a few centimeters apart when they should be 27cm apart, the measurement is not going to be very robust. The measurement can still be used, but it should not be trusted as much as a good measurement with the front and back tags at the expected robot diameter. To model this phenomenon, the standard error for the x, y and θ weight distributions is inversely proportional to the weight given to the diameter measurement.

Using the probability density function from section 3.4 a weight can be assigned to the measured diameter. A large weight implies a good measurement. The standard error on the x, y and θ weight functions should be smaller. Bad diameter measurements are given low weights that correspond to a more uncertain weight assigned to x, y, and θ, and a larger standard deviation.

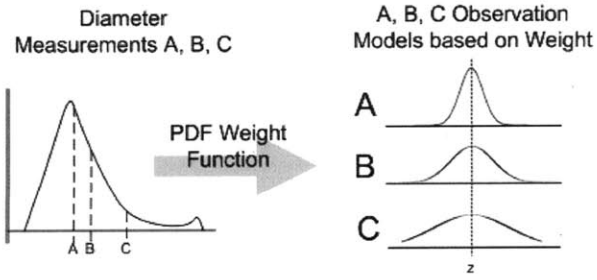


Figure 4.4: Observation Model Based on Measured Robot Diameter. Weight from robot diameter PDF determines standard deviation on observation model. Smaller weight from PDF increases standard deviation on observation model.

Now a new weight is given to each particle as the sum of the x, y, and θ weights. For the Bayesian update, this weight is multiplied by the probability of being in this particular state, but that probability is trivially 1/(number of particles). Normalization of the new weights is similar to the Bayesian update.

4.3.3 *Reselecting Particles*

With new weights, the Particle Filter has a better idea of which particles accurately model the robot's belief distribution. With this information the cloud is sampled, and new particles are selected based on their weights. The process of sampling is straight forward. A cumulative distribution is created from the new weights. Each particle is mapped to a range between 0 and 1 based on this distribution from its weight. A particle with a large weight is mapped to a larger range. Particles with very small weights have a narrow range. Using a uniform distribution between 0 and 1, a random variable is generated and mapped into the cumulative distribution. If the random variable falls into a particle's range, then that particle has been selected for the new cloud. This process is repeated until a new cloud is generated with the same number of particles as the original cloud. Particles with large weights have wider ranges between 0 and 1 and are more likely to be chosen multiple times at random with the uniform distribution. Particles with small weights are likely to be dropped all together and never propagate into the next cloud.

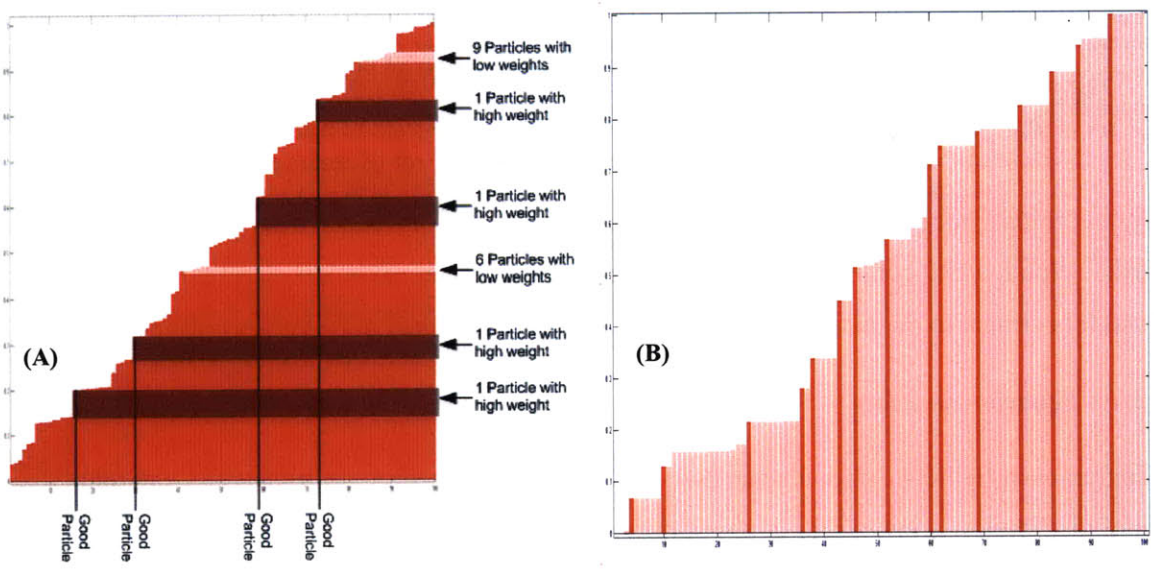


Figure 4.5: Creating a New Cloud by Reselecting Particles. (A) A cumulative distribution maps the particle index (x axis) to a range (y-axis). A uniform distribution is used to sample the y axis, these samples are mapped to the particle index on the x-axis from the cumulative distribution function. Larger ranges on the y-axis as more likely to be chosen by the uniform distribution. (B) A cumulative distribution where only 15 particles compose the more than 90% of the distribution. These 15 have a high probability of being reselected.

Once the new cloud has been generated several of the particles may have been chosen more than once, and several particles that are far from the measurement may have been dropped. The result is that the cloud has shrunk in size around the measurement, and the belief distribution is tighter to reflect a higher certainty in the robot's state. Because many of the particles are the same, some initial process noise is generated to spread the cloud a bit. This initial process noise gives the belief state more continuity, and the distribution is no longer as coarse.

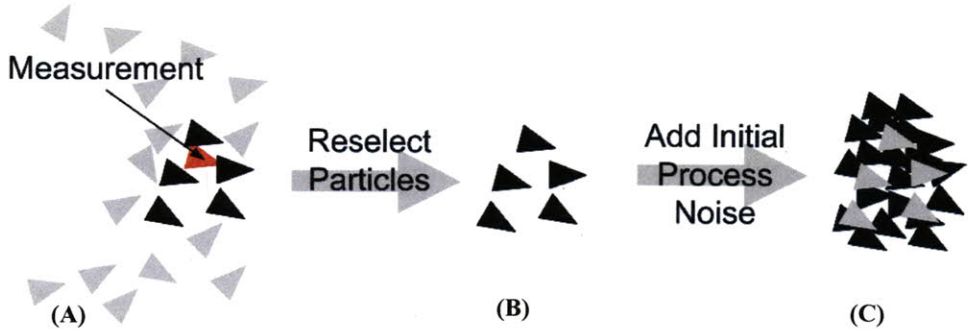


Figure 4.6: Illustration of Reselection and Addition of Initial Process Noise. (A) The Particles near the measurement are more likely to be reselected. The reselected particles are darkened (B) these particles compose the new cloud. Many of them may be repeated. (C) Initial process noise is added to the new particle cloud to distribute the particles for a more continuous belief distribution.

Without initial process noise after sampling, and without process noise from the odometry, the particle cloud would continue to shrink to a single particle and the belief distribution would not exist. It is critical to maintain a distribution and some uncertainty.

4.3.4 Behavior summary

The frames in figure 4.7 depict a summary of the update procedure. In frames 1-3, the robot's belief distribution is increasing in uncertainty. At frame 4, a measurement occurs, but it is not a very 'good' measurement. The belief distribution shrinks a small amount. In frame 5, the distribution grows again from an odometry measurement. In frame 6, a decent measurement is made and the cloud collapses and the belief distribution uncertainty shrinks.

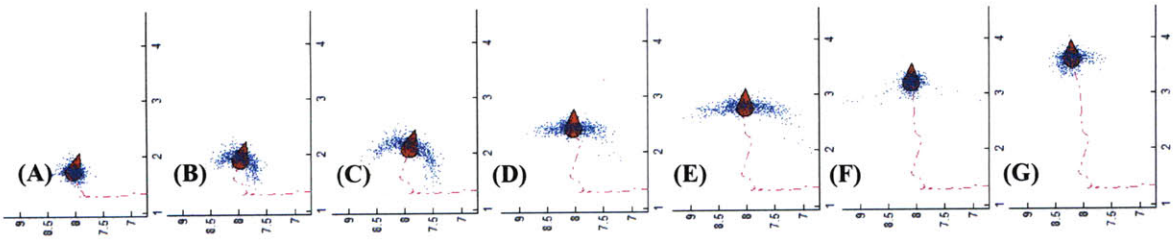


Figure 4.7: Typical Evolution of Belief Distribution. (A-C) Odometry updates increase the uncertainty of the belief distribution, the cloud spreads out. (D) A measurement is made, but is of poor quality, the belief distribution is only slightly more certain. (E) Another odometry measurement spreads the belief distribution out further. (F) A good measurement is made and the belief distribution collapses and the cloud shrinks to represent this increase in certainty. (G) The process continues with the next odometry measurement

This process is very similar to the growing and shrinking of the covariance matrix in an EKF. The covariance increases from process noise and shrinks when a measurement is made. The belief distribution in the Particle Filter is clearly not Gaussian, and the assumptions made in the EKF may not be entirely accurate for this reason.

4.4 Particle Filter Results

4.4.1 Animation of Filter

To visualize the Particle Filter and compare the robot traces simultaneously, animations were created with the robot path history as seen in figure 4.8. The left windows show the path history for the x, y locations of the robot. The odometry is on the top, absolute positions from Ubisense in the middle, and the Particle Filter belief distribution is in the bottom frame.

The right panels show a history of the robot heading. The straight line is the current robot heading at this time step. The arc traced out by the straight line is composed of the previous headings from the last 100 time steps. At the center of the θ path history plots is the orientation from 100 time steps prior. For non-holonomic circles, the path of θ should be a spiral into the center of the graph.

In comparing the animations, some results are immediately obvious. The odometry path history is smooth as expected and free from high frequency noise. However, there is drift. The x, y location of the robot is slightly off from the Ubisense animation. The θ measurement from the

absolute positioning is nearly unintelligible. There is a large amount of high frequency noise, and at the moment the robot is facing backwards. The Particle Filter appears to be decent mix of both. The path history in x, y has a few kinks but is in general smooth. There is no drift on its current location. More importantly, the plot of the θ path history is as smooth as the odometry but accurate without any drift.

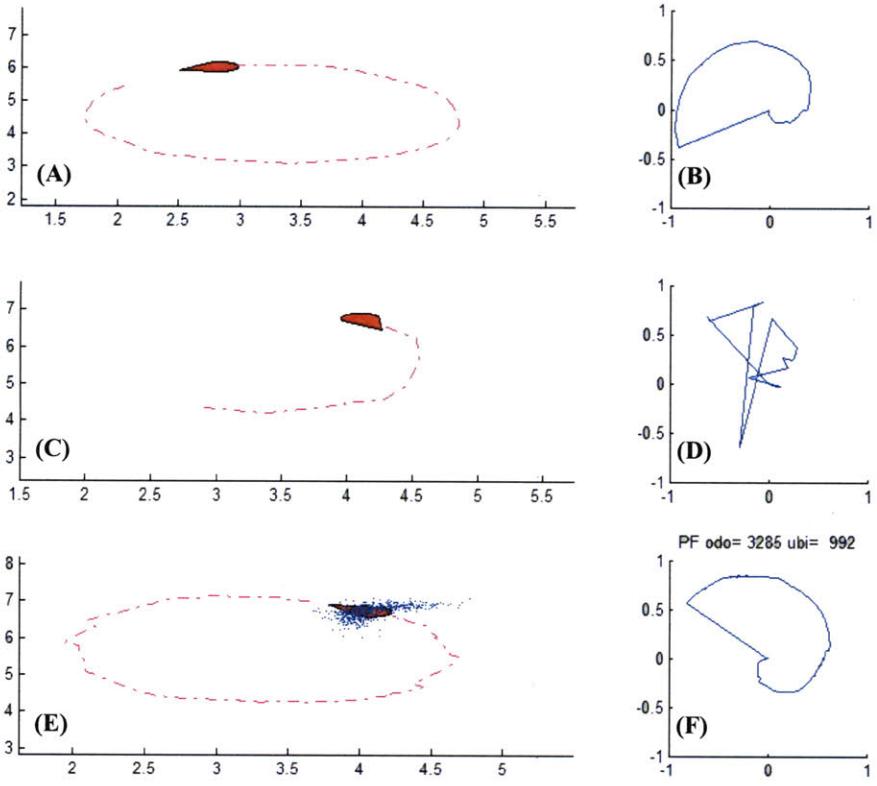


Figure 4.8: Animations of Odometry, Ubisense, and the Particle Filter. (A) x, y path plot for the robot odometry. (B) Heading for the last 100 steps, the most current θ is represented by the radius, previous measurements collapse to the origin. (C) x, y path plot for Ubisense measurements. (D) Heading for the last 100 Ubisense measurements, note high frequency noise. (E) x, y path plot for the Particle Filter, and representation of the belief distribution with particle cloud. (F) Particle Filter heading estimate for the last 100 time steps.

4.4.2 Results Representation

The following results in figure 4.9 are for experiment 1, running the robot clockwise around the rectangular track. Examining the path plots shows promising results.

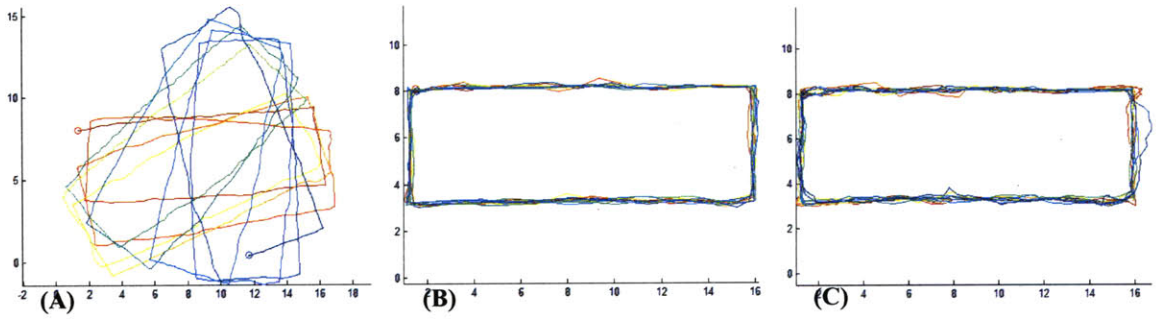


Figure 4.9: Cross Comparison of Path Plots for Experiment 1, Rectangle CW. (A) Odometry path plot contains drift. (B) Ubisense absolute positions. (C) Particle Filter Estimate with some noise.

The odometry on the left has a large amount of radial drift. The Ubisense in the middle appears extremely accurate to the rectangular track, and the Particle Filter on the right has some high frequency noise and bumps where the filter may have become momentarily lost.

A more important comparison is between the time course of x, y and θ. The green trace of the Particle Filter fits snug onto the blue Ubisense traces. The blue Ubisense measurement of θ still confounds the visualization.

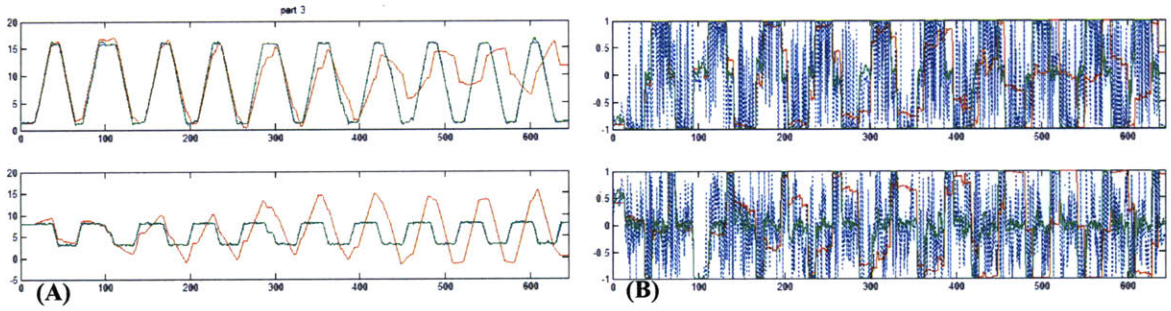


Figure 4.10: Cross Comparison of Time Courses for Experiment 1, Rectangle CW. Ubisense in blue, odometry in red, Particle Filter in green (A) x time course top, y time course bottom. (B) sin(θ) time course top, cos(θ) time course bottom.

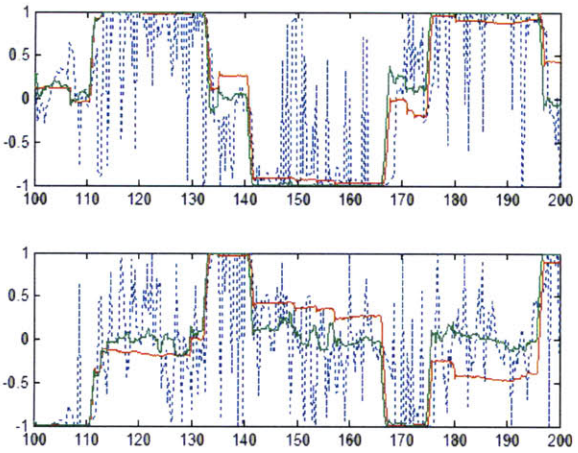


Figure 4.11: Cross Section of Sin(θ) Cos(θ) time course (100-200s). Ubisense in blue, odometry in red, Particle Filter in green. Sin(θ) time course top, cos(θ) time course bottom.

On closer inspection, the Particle filter's estimate for θ has a much lower noise level than the measurement of θ from the absolute positions. There is still a high frequency component to the noise, but it is significantly attenuated. For these initial features, the estimated θ is only slightly out of phase with the odometry θ.

4.4.3 Comparing Non-holonomic and Holonomic

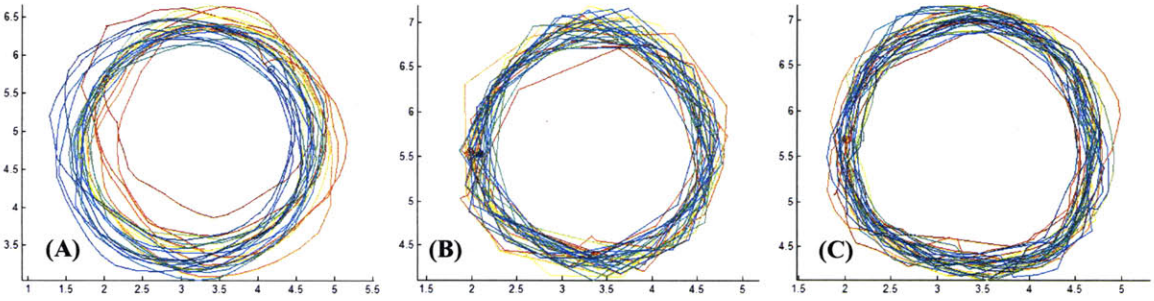


Figure 4.12: Cross Comparison of Path Plots for Experiment 4, Non-holonomic CCW Circles. (A) Odometry path plot. (B) Ubisense path plot. (C) Particle Filter path plot using 700 particles.

For experiment 4, the results are shown in figure 4.12 for the robot making several counter clockwise non-holonomic circles.

It is more evident here that some of the irregular Ubisense measurements have been filtered out. The Particle Filter path is much smoother than the Ubisense path, but without the drift of the odometry.

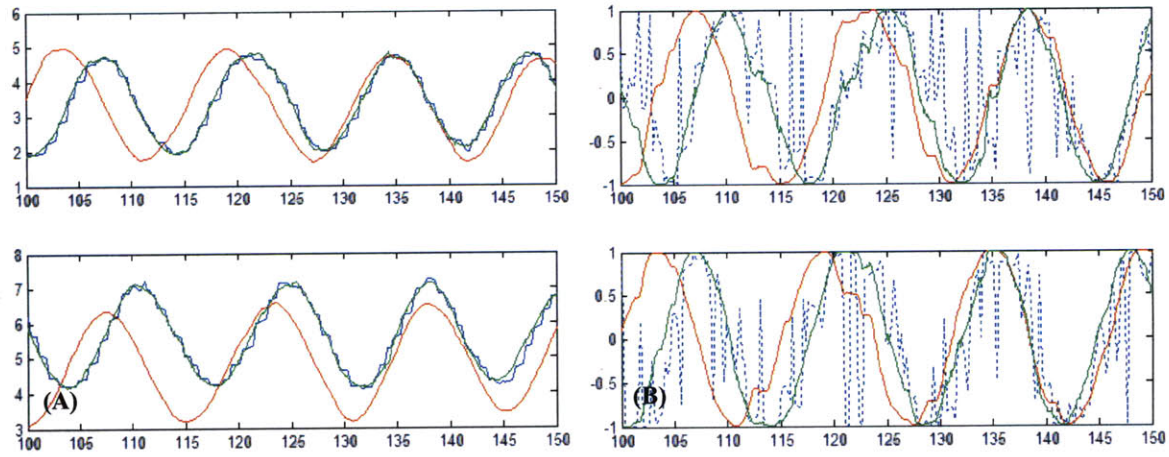


Figure 4.13: Cross Comparison of Time Courses for Experiment 4, Non-holonomic CCW Circles (100-150s). Ubisense in blue, odometry in red, Particle Filter in green (A) x time course top, y time course bottom. (B) sin(θ) time course top, cos(θ) time course bottom.

The smoother fit in the x, y time courses is illustrated in figure 4.13. The accurate estimate of θ is also very clear in a cross comparison of the Ubisense measured θ and the Particle Filter's estimated θ. Phase shifting of the odometry is also apparent in the non-holonomic circles. While smoother, the odometry is slightly out of phase with the measured θ. Small features in the odometry's θ measurement are visible in the Particle Filter's estimate. The high frequency component of the odometry is maintained while the high frequency component of the Ubisense has been filtered out.

The holonomic results in figure 4.14 are similar to the non-holonomic path plots. The Particle Filter path has more high frequency features that are also in the odometry path, but it maintains the general shape as the Ubisense path.

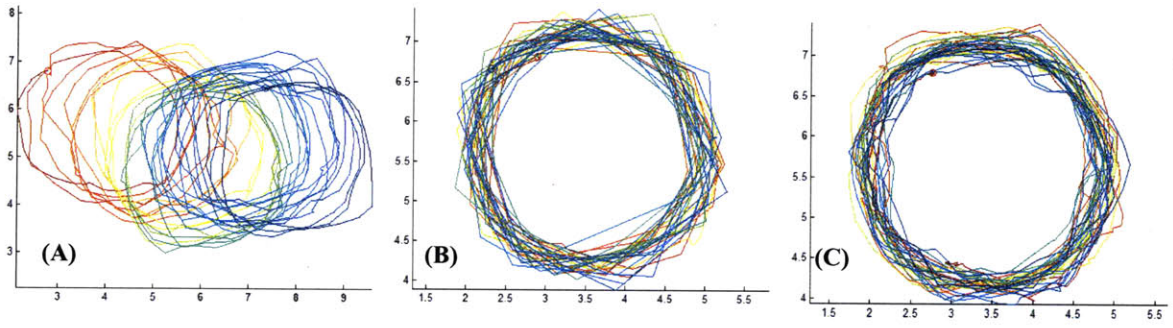


Figure 4.14: Cross Comparison of Path Plots for Experiment 6, Holonomic CCW Circles. (A) Odometry path plot. (B) Ubisense path plot. (C) Particle Filter path plot using 700 particles.

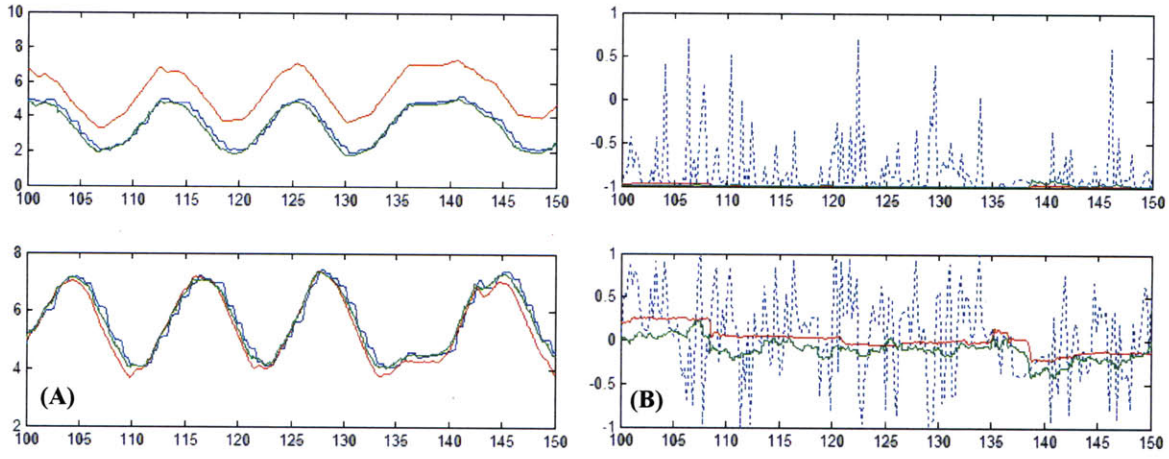


Figure 4.15: Cross Comparison of Time Courses for Experiment 6, Holonomic CCW Circles (100-150s). Ubisense in blue, odometry in red, Particle Filter in green (A) x time course top, y time course bottom. (B) sin(θ) time course top, cos(θ) time course bottom.

Examining just the θ estimation in figure 4.15, the particle filter has a much lower noise level than the Ubisense measurements. The constant θ seems to be maintained as well.

4.4.4 *Analysis of Directionality*

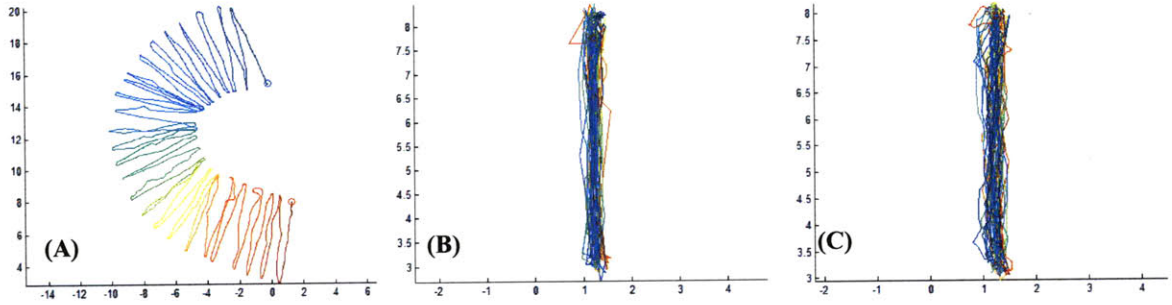


Figure 4.16: Cross Comparison of Path Plots for Experiment 7, Switchback Suboptimal. (A) Odometry path plot. (B) Ubisense path plot. (C) Particle Filter path plot using 700 particles.

Directionality and location do not appear to influence the Particle Filter performance. The path plot in figure 4.16 appears noisier than the Ubisense path, but that is due to the features picked up by the odometry.

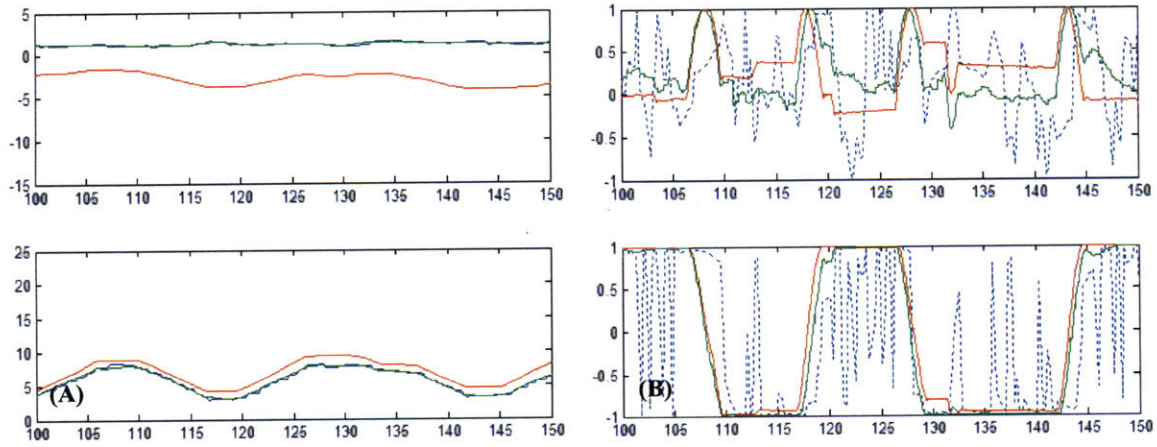


Figure 4.17: Cross Comparison of Time Courses for Experiment 7, Switchback Suboptimal (100-150s). Ubisense in blue, odometry in red, Particle Filter in green (A) x time course top, y time course bottom. (B) sin(θ) time course top, cos(θ) time course bottom.

As the robot odometry measurements begin to rotate, the θ measurements quickly become out of phase, and oscillations begin to appear in the x time course for both experiments 7 and 8. The

Ubisense time courses and Particle filter are still tightly associated. Experiment 7 shows a higher noise level on its measurement of θ than experiment 8. This is likely caused by the sub optimal location of experiment 7 which was chosen purposefully. The Particle Filter results appear unaffected.

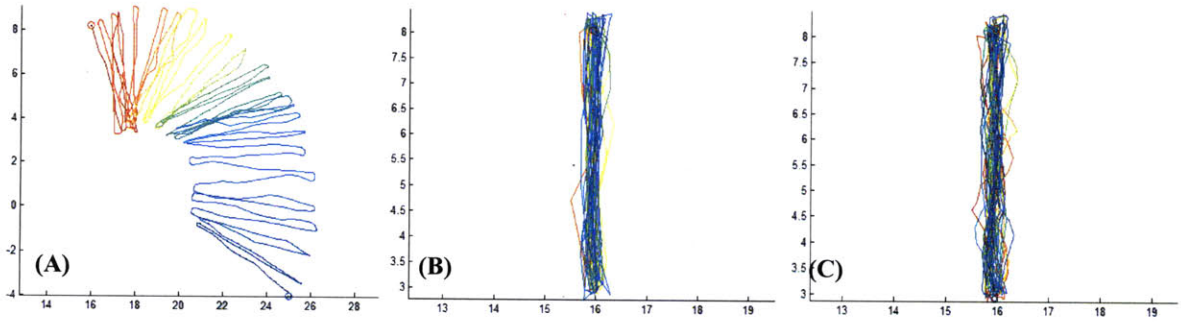


Figure 4.18: Cross Comparison of Path Plots for Experiment 8, Switchback more Optimal. (A) Odometry path plot. (B) Ubisense path plot. (C) Particle Filter path plot using 700 particles.

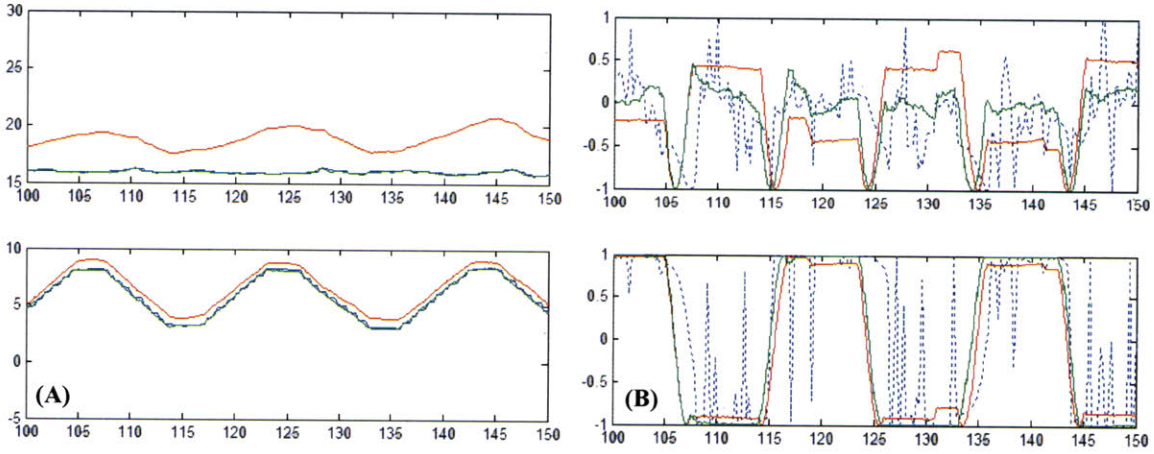


Figure 4.19: Cross Comparison of Time Courses for Experiment 8, Switchback more Optimal (100-150s). Ubisense in blue, odometry in red, Particle Filter in green (A) x time course top, y time course bottom. (B) sin(θ) time course top, cos(θ) time course bottom.

4.4.5 *Analysis of Random Track*

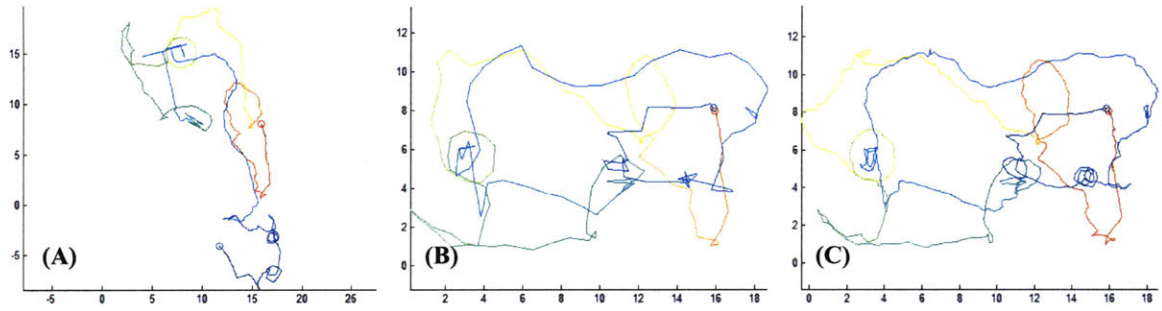


Figure 4.20: Cross Comparison of Path Plots for Experiment 9, Random Track. (A) Odometry path plot. (B) Ubisense path plot. (C) Particle Filter path plot using 700 particles.

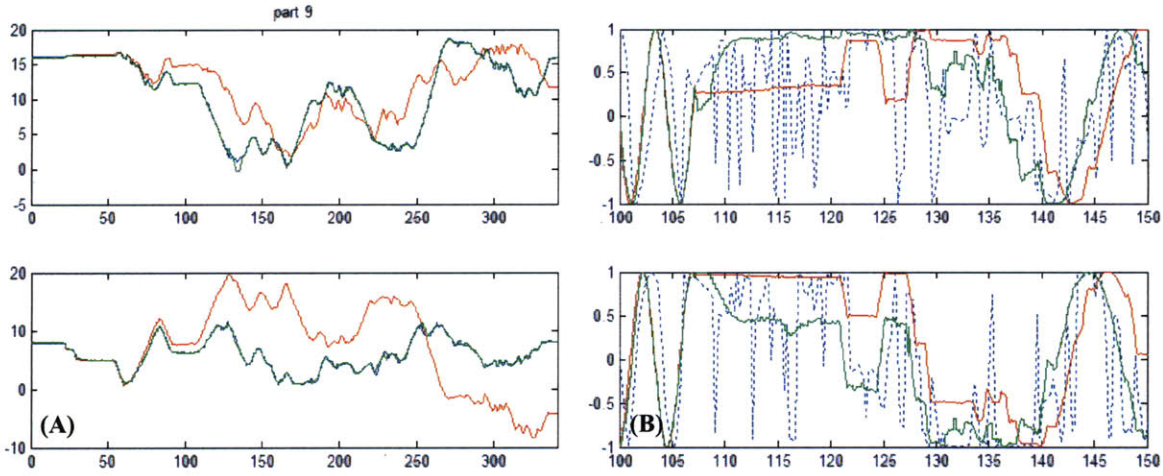


Figure 4.21: Cross Comparison of Time Courses for Experiment 9, Random Track. Ubisense in blue, odometry in red, Particle Filter in green (A) x time course top, y time course bottom. (B) sin(θ) time course top, cos(θ) time course bottom for 100-150s.

The Particle Filter demonstrates robust performance mixing non-holonomic and holonomic drive, mixing directionality, and for longevity. The high frequency features picked up by the odometry are clearly visible on the Particle Filter's path plot in figure 4.20. The rings are clearly rings and less polygonal. In experiments 3-6, the sharp corners were not as visible from the Ubisense measurements because the number of laps occluded these fine details. The hard edges

from the Ubisense are softened by the odometry. The plot of the x, y time course in figure 4.21 illustrates the disastrous drift for the odometry that is removed with the absolute positioning. Additionally, the high frequency noise content on the Ubisense measurement of θ is removed by the filter. The phase shifting and drift in the odometry's θ is also evident, but the hard features in the odometry's θ measurement appear in the Particle Filter's θ estimate.

4.4.6 Issues from Manual Drive

Because the robot was driven manually for the experiments, there are a number of irregularities that appear in the path plots. Jitter in the absolute position and a number of odd outlying points are due to human error when driving. Driving a holonomic robot with non-holonomic constraint is very challenging. A number of the non-holonomic experiments have holonomic jumps in them because of human error.

The most difficult experiment was performing circles while maintaining a constant heading. The standard deviation is mostly due to human error on this distribution.

5 Conclusions

5.1 Particle Filter Performance

The Particle Filter is a suitable solution for the localization of this particular robot with this particular sensor network. As a proof of concept, the Particle Filter performed well and showed that given the current hardware it is possible to localize the robot quickly, accurately, and over a long period of time.

Determining heading is the most critical aspect for all three cases. For odometry, heading was especially difficult because the angular drift was not well defined. For the absolute positioning with Ubisense, heading measurements were subject to the large noise margins from the tags. Finally, the Particle Filter out performed the Ubisense measurements and the odometry for determining heading. Whether or not the Particle Filter estimates for x-y locations are more robust than the Ubisense positions requires more investigation. One advantage to using the Particle Filter is the odometry superimposes the accurate high frequency features onto the

Ubisense data. This gives smoothness to the paths and incorporates small kinks into the paths from human drive error.

5.2 Particle Filter Improvements

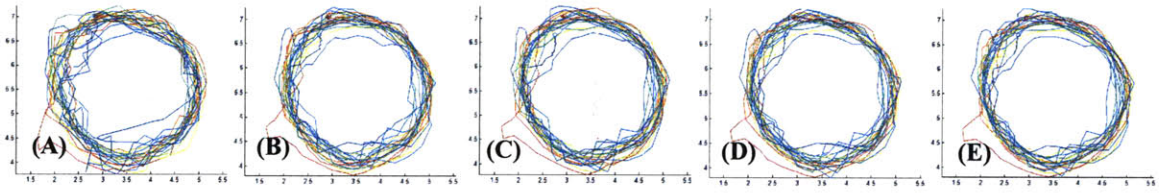


Figure 5.1: Path Plots for Experiment 6 Varying Particles, Holonomic CW Circles. (A-E) Path plots generated with 100, 300, 700, 1000, and 3000 particles respectively.

The Particle Filter is computationally intensive. The plots shown in figure 5.1were from 700 particles. However, the performance can dramatically increase or degrade depending on how many particles are used. One especially difficult data set to estimate is from experiment 6 where the robot ran over a small bolt and caused a number of chain reaction mechanical irregularities. Shown here is a collection of path plots from the particle filter using an increasing number of particles, from left to right: 100, 300, 700, 1000, and 3000. Localization with 100 particles is very fast, but there are several inaccuracies. As the number of particles increase, paths become smoother and the variation in path plots begins to decrease. The plots for 1000 particles and 3000 particles appear very similar. However, because this is a stochastic process, every time the Particle Filter is run a different path plot will result.

There are a number of parameters to tune in the filter to get the desired results. The process noise is actually much higher than originally expected producing a much larger belief distribution. To improve the filter performance, alternative observation models can be tested. The performance also varies by changing the filter parameters on the Ubisense data. Filtering out 30% of the data points makes the Particle Filter more heavily reliant on the odometry, but it might be worth comparing the performance using all the data against filtered data or having a static observation model not weighted by the diameter measurements.

5.3 Implementation in Real time

The next step is to implement the filter in real time, in JAVA for the OLPC robot software modules. The major decision at this point is to continue with the Particle Filter or try an EKF to compare the performance. With a better understanding of the noise model and robot dynamics the EKF may be a more viable solution. For the application, a comparison between computation times and noise level will be critical for determining whether either solution is tractable on the confines of an OLPC.

Another challenge will be synchronizing the Ubisense engine and the odometry readings to arrive simultaneously in the JAVA program. While distributed processing of robot localization is an effective use of resources, it might be more practical to run a localizing engine on a large machine with one particle filter for each robot or one EKF per robot.

6 Acknowledgements

I would like to thank Tod Machover and Bob Hsiung for allowing me to use their robots and equipment for my research in the support of Death and the Powers. I also owe a great deal to Dr Leslie Kaelbling for being patient and understanding my unusual time line for completing my research work.

In addition, I would like to thank friends and family for their support reading revisions of my work. My parents showed an enormous amount of sympathy for my situation, but I would especially like to thank Sophie Wong for encouraging me, and forcing me to stay for a masters.

7 References

http://arduino.cc/en/Main/ArduinoBoardMega

http://eval.ubisense.net/howto/Contents/Contents.html

8 Appendices

8.1 Python Joystick Control Code

```
#joystick_serial.py
import serial
import pygame
import math
import time
import threading

# allow multiple joysticks
joy = []

# Arduino USB port address (try "COM5" on Win32)
usbport = "COM15"

# define usb serial connection to Arduino
ser = serial.Serial(usbport, 19200, timeout = 1)

v_x = 0
v_y = 0
v_t = 0
turn = False

iso_Y = False
iso_X = False

dat_file = open("data1.m", 'w');
dat_file.write("%samples\t dTh\tdX \tdY \tM1 \tM2 \tM3\n robo=[");

command = '-'
throttle = 0
# handle joystick event
def handleJoyEvent(e):
    A_OFF = 0
    B_OFF = 0
    C_OFF = 0

    global v_x
    global v_y
    global v_t
    global turn
    global iso_Y
    global iso_X
    global dat_file
    global throttle
    global command

    #detects change in axis
    if e.type == pygame.JOYAXISMOTION:
        axis = "unknown"
        str = "Axis: %s; Value: %f" % (axis, e.dict['value'])

        if (e.dict['axis'] == 0):
            axis = "X"

        if (e.dict['axis'] == 1):
            axis = "Y"

        if (e.dict['axis'] == 2):
            axis = "Throttle"

        if (e.dict['axis'] == 3):
            axis = "Z"
```

```
        #pump out data to arduino via serial here
        if (axis == "X" or axis == "Y" or axis == "Z" or axis == "Throttle"):
            #print axis, e.dict['value']
            if (axis == "X"):
                if not iso_Y:
                    v_x = e.dict['value']*127*throttle
                    if (abs(v_x) > 10):
                        ser.write("X")
                        ser.write(chr(int(v_x+127)))
                    else:
                        ser.write("X")
                        ser.write(chr(int(127)))
                print 'X: ', int(v_x)
            elif (axis == "Y"):
                if not iso_X:
                    v_y = -e.dict['value']*127*throttle
                    if (abs(v_y) > 10):
                        ser.write("Y")
                        ser.write(chr(int(v_y+127)))
                    else:
                        ser.write("Y")
                        ser.write(chr(int(127)))
                print 'Y: ', int(v_y)
            elif (axis == "Z"):
                if (turn):
                    v_t = e.dict['value']*127*throttle
                else:
                    v_t = 0
                if (abs(v_t) > 10):
                    ser.write("T")
                    ser.write(chr(int(v_t+127)))
                else:
                    ser.write("T")
                    ser.write(chr(int(127)))

                print 'T: ', int(v_t)
            else:
                throttle = -(e.dict['value']-1)/2

            if not turn and v_t != 0:
                ser.write("T")
                ser.write(chr(int(127)))


    #some button logic
    elif e.type == pygame.JOYBUTTONDOWN:
        str = "Button: %d" % (e.dict['button'])
        # uncomment to debug
        #output(str, e.dict['joy'])
        # Button 0 (trigger) to quit
        b = e.dict['button']
        if (b == 0):
            turn = True
        elif (b == 8):
            command = 'H'
            print 'comm H'
            ser.write(command)
            ser.write(command)

        elif (b == 3):
            iso_X = True
            ser.write("Y")
            ser.write(chr(int(127)))
        elif (b == 2):
            iso_Y = True
            ser.write("X")
            ser.write(chr(int(127)))
        else:
```

```
            print b, "Bye!\n"
            ser.close()
            dat_file.write("];")
            dat_file.close()
            quit()


    elif e.type == pygame.JOYBUTTONUP:
        str = "Button: %d" % (e.dict['button'])
        # uncomment to debug
        #output(str, e.dict['joy'])
        # Button 0 (trigger) to quit
        b = e.dict['button']
        if (b == 0):
            turn = False
            ser.write("T")
            ser.write(chr(int(127)))
        elif (b == 3):
            iso_X = False
        elif (b == 2):
            iso_Y = False


    else:
        pass

# print the joystick position
def output(line, stick):
    print "Joystick: %d; %s" % (stick, line)


# wait for joystick input
def joystickControl():
    global turn
    t = time.clock()
    ev = pygame.event.poll()
    while ev.type != pygame.NOEVENT:
        if (ev.type != pygame.NOEVENT and \
            (ev.type == pygame.JOYAXISMOTION or ev.type == pygame.JOYBUTTONDOWN or ev.type ==
pygame.JOYBUTTONUP)):
            handleJoyEvent(ev)
        ev = pygame.event.poll()

    while True:
        """        t = time.clock()
        ser.write("X")

        ser.write(chr(int(127)))
        ser.write("Y")
        ser.write(chr(int(127)))
        ser.write("T")
        ser.write(chr(int(127)))
##        serial_interface('Data')
        print (time.clock() - t)"""
        if (time.clock()-t > .01):
            t = time.clock()
            e_list = pygame.event.get()
            if (len(e_list)):

                #process the entire event queue
                for ev in e_list:
                    #ignore
                    if (ev.type != pygame.NOEVENT and \
                        (ev.type == pygame.JOYBUTTONDOWN or \
                         ev.type == pygame.JOYBUTTONUP or \
                         (ev.type == pygame.JOYAXISMOTION and (turn or ev.dict['axis'] != 3)))):
                            handleJoyEvent(ev)
            serial_interface('Data')
```

```
def serial_interface(command):
    if (command == 'H'):
        b = ser.read(2)
        print b[0] + b[1]
    elif (command == 'Data'):

        #if ser.inWaiting() >= 17:
        #    b = ser.read(17)
        #else:
        #    b = ['Error']
        b = ser.read(17)
        if (len(b) != 17):
            print 'Restart Arduino - Timeout'
        if (len(b) == 17 and b[0] == 'D'):
            write4BytesToFile(b[4],b[3],b[2],b[1])
            for j in range(2, 8):
                write2BytesToFile(b[2*j+2], b[2*j+1])


##                print (i + 2**15) % 2**16 - 2**15  ,
##                i = int(ord(b[13]) + ord(b[12])*256)
##                print (i + 2**15) % 2**16 - 2**15  ,
##                i = int(ord(b[11]) + ord(b[10])*256)
##                print (i + 2**15) % 2**16 - 2**15  ,
##
##                i = int(ord(b[9]) + ord(b[8])*256)
##                print (i + 2**15) % 2**16 - 2**15  ,
##
##                i = int(ord(b[7]) + ord(b[6])*256)
##                print (i + 2**15) % 2**16 - 2**15  ,
##                i = int(ord(b[5]) + ord(b[4])*256)
##                print (i + 2**15) % 2**16 - 2**15  ,
##                i = int(ord(b[3]) + ord(b[2])*256)
##                print (i + 2**15) % 2**16 - 2**15
##
            dat_file.write(';\n');

    ser.flushInput();

def write4BytesToFile(b0, b1, b2, b3):
    i = int(ord(b0) + ord(b1)*(2**8) + ord(b2)*(2**16) + ord(b3)*(2**24))# + ord(b2)<<16 +
ord(b3)<<24)
    #print ord(b0), ord(b1), ord(b2), ord(b3)
    dat_file.write(str(i) + '\t');
    #dat_file.write(' ' + str((i + 2**15) % 2**16 - 2**15) + ';\n')


def write2BytesToFile(low, high):
    i = int(ord(low) + ord(high)*256)
    dat_file.write(' ' + str((i + 2**15) % 2**16 - 2**15) + '\t')

# main method
def main():
    # initialize pygame
    pygame.joystick.init()
    pygame.display.init()
    v_x = 0
    v_y = 0

    if not pygame.joystick.get_count():
        print "\nPlease connect a joystick and run again.\n"
        quit()
    print "\n%d joystick(s) detected." % pygame.joystick.get_count()
    for i in range(pygame.joystick.get_count()):
        myjoy = pygame.joystick.Joystick(i)
        myjoy.init()
        joy.append(myjoy)
        print "Joystick %d: " % (i) + joy[i].get_name()
    print "Depress trigger (button 0) to quit.\n"
```

```
    # run joystick listener loop
    joystickControl()

# allow use as a module or standalone script
if __name__ == "__main__":
    main()
```

8.2Firmware Code

8.2.1 Main Module

```
//motass.pde
#include <PID.h>
#include <Encoder.h>
#include <math.h>
#include <SoftwareServo.h>
#include <Motass.h>

#define cbi(sfr, bit) (_SFR_BYTE(sfr) &= ~_BV(bit))
#define sbi(sfr, bit) (_SFR_BYTE(sfr) |= _BV(bit))

//CHANGE MASK FOR MEGA AND NEW ENCODERS
//REGISTER L on pins 46, 47 and 48,49

#define E1_MASK_A B00000001
#define E1_MASK_B B00000010
#define E2_MASK_A B00000100
#define E2_MASK_B B00001000
#define E3_MASK_A B00010000
#define E3_MASK_B B00100000


//mask for the motor controller ON/OFF check pins
//#define EN_M_MASK B00010101
#define EN_M_MASK B00101010

#define MPIN_3 33
#define MENA_3 32
#define MPIN_2 35
#define MENA_2 34
#define MPIN_1 37
#define MENA_1 36


#define RAD_30  0.523598776
#define RAD_150 2.61799388
#define RAD_270 4.71238898
#define RAD_210 3.6651914//0.523598776
#define RAD_330 5.75958653
#define RAD_90  1.57079633

#define RAD_300 5.23598775
#define RAD_60  1.04719755
#define RAD_180 3.14159265


#define RAD_120  2.09439510
#define RAD_240  4.18879020
#define RAD_0    0

/** Adjust these values for your servo and setup, if necessary **/
Motass MA[3] = {Motass(E1_MASK_A, E1_MASK_B),
                Motass(E2_MASK_A, E2_MASK_B),
                Motass(E3_MASK_A, E3_MASK_B)
               };
#define do3(fun) MA[0]. fun ; MA[1]. fun ;  MA[2]. fun

//#define kP 2.5
//#define kI 1.2
#define kP .5
#define kI 0.1
#define kD 0
#define M_MIN 30
#define M_MAX 150
```

```
#define VEL_SCALE 3.5

//Safety timeouts
#define WD_TIME_OUT 500
#define EN_TIME_OUT 150

#define START_MS 1000
#define START_UP 1900
#define INIT_ZERO 50

#define PY_PRINT_MS 50
unsigned long last_serial;
unsigned long last_encoder;



#define LED_RUN 26
#define LED_SET 24
#define LED_ERR 22

int fail_n = 0;

volatile unsigned int num = 0;

float v_x = 0;
float v_y = 0;
float v_t = 0;

unsigned long py_time;

void setup(){

  //setup the status LEDs
  pinMode(LED_RUN, OUTPUT);
  pinMode(LED_SET, OUTPUT);
  pinMode(LED_ERR, OUTPUT);

  set_LED(LED_SET);

  // Timer2 PWM Mode set to fast PWM
  cbi (TCCR2A, COM2A0);
  sbi (TCCR2A, COM2A1);
  sbi (TCCR2A, WGM20);
  sbi (TCCR2A, WGM21);
  cbi (TCCR2B, WGM22);

  // Timer2 Clock Prescaler to : 2
  cbi (TCCR2B, CS20);
  sbi (TCCR2B, CS21);
  cbi (TCCR2B, CS22);


  sbi (TIMSK2,TOIE2);              // enable Timer2 Interrupt

  //set up motor1 and PID1

  //we need to set the motor enable pins
  //and motor output pins we are on register C 7-0 maps 30-37
  //input enables are odd 33, 35, 37
  //output are 32, 34, 36
  DDRC |= B00010101;
  DDRC &= B11010101;
  //DDRC |= B00101010;  //configure motor outputs
  //register = 0 means input
  //register = 1 means output
  //DDRC &= B11101010;  //configure enable inputs

  MA[0].attach(MPIN_1);
  MA[1].attach(MPIN_2);
```

```
  MA[2].attach(MPIN_3);
  do3(setMinimumPulse(1000)) ;
  do3(setMaximumPulse(2000)) ;
  do3(setPID(kP, kI, kD)) ;
  do3(setlimits(M_MIN,M_MAX)) ;//limit PID range
  do3(fail = false);//initialize the fails to false

  Serial.begin(19200);  // start serial for output

  if ((PINC & EN_M_MASK) != EN_M_MASK){
      fail_n = 4;
      Serial.println("H-bridges not powered");
  }
  else
    Serial.println("H-bridges powered");

  //for(int i = 0; i <300 ; i++){
  //  SoftwareServo::refresh();
  //  delay(10);
  //}



  fail_n = 0;

  num = 0;

  v_x = 0;
  v_y = 0;
  v_t = 0;

  Serial.println("Pulsing Motors");


  while (millis() < START_MS)
    delay(3);
  //put some innocuous values for the initial motor speeds
  do3(write(EST_ZERO)) ;

  //make sure there is a nominal stop value
  do3(setsetpt(0)) ;//set point in ticks per second
  do3(setZeroPt(EST_ZERO)) ;

  SoftwareServo::refresh();
  last_serial = millis();
  last_encoder = millis();
  py_time = millis();
}



void loop(){

  if(num >= 100){//should run 10x per second
    //it is time to calculate the speed
    do3(calc_vel((float) num)) ;

    //we are done
    num = 0;

    if ((PINC & EN_M_MASK) != EN_M_MASK){
      fail_n = 4;
      Serial.println("H-bridges not powered");
    }
    //*********************************************************
    //            FAILURE CHECKS
    //checks failure on each speed calculation
    //*********************************************************
```

```
  //the encoder is stalled if measured vel == 0 and PID is set to a non-zero value
  //or we are requesting a non-zero speed and the robot is significantly stalled
  //this occurs when the motor controllers are not turned on
  //the encoder has become unplugged
  //the motors have become disconnected

  else {
    do3(failruntime()) ;


    if (MA[0].fail || MA[1].fail || MA[2].fail){

      //now we check if we have timed out yet
      if (millis() - last_encoder > EN_TIME_OUT){

        //clear the PID
        do3(clear()) ;
        //write our old zero point
        do3(writezpt()) ;
        //set the failed motor
        fail_n = MA[0].fail ? 1 : (MA[1].fail ? 2 : 3);
      }


    }
    //if we are not stalled don't keep track of the time outs
    else {
      last_encoder = millis();
    }
  }

  //after calc print python

}

//compute the gains for PID
do3(writegain()) ;
//refresh the motor controllers
//if we have not failed, if we fail we cut communication
//and the motor controller will power down

while (fail_n){
  //set_LED(LED_ERR_MASK);
  set_LED(LED_ERR);
  delay(5);
}
SoftwareServo::refresh();

//******************************************************
//          START UP
//gives each motor a kick and sees if the encoders and motors are
//connected properly and working
//can enter initial fail state
//******************************************************
//store the zero point for each motor
//on start up otherwise, run the normal controller loop

if (millis() < START_UP){
  last_serial = millis();

  //check kick
  if (millis() > START_UP-INIT_ZERO){
    //either encoders or motors are not connected
    do3(failstartup()) ;

    //INITIAL FAIL STATE
    //stops communicating with H-bridges
    fail_n = MA[0].fail ? 1 : (MA[1].fail ? 2 : (MA[2].fail ? 3 : 0));
    if (fail_n){
```

```
      Serial.println("Fail: Encoder or Motor not connected");
    }

    //test done save zero points for later

    else {
      //clear PID
      do3(clear()) ;
      do3(adjustzpt()) ;
      Serial.println("Encoder Checked Passed");

    }

    Serial.print(MA[0].gain);
    Serial.print(" ");
    Serial.print(MA[1].gain);
    Serial.print(" ");
    Serial.println(MA[2].gain);
    set_LED(LED_RUN);
  }
}

//*******************************************************
//          NORMAL LOOP
//reads from serial and sets motor speeds appropriately
//*******************************************************
else {
  //if there is new serial data update the wheel velocities
  if (Serial.available() >= 2) {
    switch (Serial.read()){
      case 'X':
        v_x = -((float)(Serial.read()-127))/127;
        last_serial = millis();
        break;
      case 'Y':
        v_y = -((float)(Serial.read()-127))/127;
        last_serial = millis();
        break;
      case 'T':
        v_t = ((float)(Serial.read()-127))/127;
        last_serial = millis();
        break;

      default:

        break;
    }//end switch, still in if

    /*MA[0].setsetpt(v_x);
    MA[1].setsetpt(v_y);
    MA[2].setsetpt(v_t);
    */



    //calculate speeds for each motor

    float v = sqrt(v_x*v_x+v_y*v_y);
    float theta = atan2(v_x, v_y);
    float one_over_denom = VEL_SCALE/(1+fabs(v_t));//might want to just divide v_t

    //set the PID speeds
    MA[0].setsetpt((v*cos(RAD_270  - theta)+v_t)*one_over_denom);
    MA[1].setsetpt((v*cos(RAD_30 - theta)+v_t)*one_over_denom);
    MA[2].setsetpt((v*cos(RAD_150 - theta)+v_t)*one_over_denom);


  }//endif serial available
}
```

```
  //make setpoints 0 if no serial data has been received for roughly half a second
  //comment out if commands not sent that frequently
  /*if (millis()-last_serial > WD_TIME_OUT){
      fail_n = 4;
  }*/

  if (millis()-py_time > PY_PRINT_MS){
    printPython();
    py_time = millis();
  }
  delay(2);
}//end loop


void printPython(){
    /*
    int time = millis();
    uint8_t msb = time && 255 >> ;
    uint8_t lsb = samples & 255;
    Serial.print(msb);
    Serial.print(lsb);

    uint8_t msb = samples >> 8;
    uint8_t lsb = samples & 255;
    Serial.print(msb);
    Serial.print(lsb);
  */
    Serial.print('D');
    print4byte(millis());

    for (int i = 0; i < 3; i++){
      print2byte((int) (MA[i].getSpeed()*1000));
    }
    print2byte((int) (v_x*1000));
    print2byte((int) (v_y*1000));
    print2byte((int) (v_t*1000));
    Serial.println();



}


void print4byte(unsigned long b){
    uint8_t msb = (b >> 24) & 255;
    uint8_t lsb = (b >> 16) & 255;
    Serial.print(msb);
    Serial.print(lsb);

    msb = (b >> 8) & 255;
    lsb = b & 255;
    Serial.print(msb);
    Serial.print(lsb);

}
void print2byte(int b){
    uint8_t msb = b >> 8;
    uint8_t lsb = b & 255;
    Serial.print(msb);
    Serial.print(lsb);
}

void set_LED(uint8_t led){
  digitalWrite(LED_RUN, 0);
  digitalWrite(LED_ERR, 0);
  digitalWrite(LED_SET, 0);
```

```
  digitalWrite(led, 1);
}

//*****************************************************************
// Timer2 Interrupt Service at 62.5 KHz
// here the audio and pot signal is sampled in a rate of:  16Mhz / 256 / 2 / 2 = 15625 Hz
// runtime : xxxx microseconds

ISR(TIMER2_OVF_vect) {

  num++;

  volatile uint8_t tPINL = PINL;

  //count encoder ticks
  do3(callback(tPINL)) ;
}
```

8.2.2 PID Module

```
//PID.cpp
/**
PID class for Arduino
**/
#ifndef PID_CPP
#define PID_CPP
#include "PID.h"

float PID::compute(float input){//find gain based on input
  float err = setpt - input;
  errint += err;


  //bound the error
  if(errint > maxerrint){
       errint = maxerrint;
  }
  if(errint < -maxerrint){
       errint = -maxerrint;
  }

  P = KP*err;
  I = KI*errint;
  D = - KD*(err-lasterr);

  gain = zeropt + P + I + D;//only for these motors, not generally

  //bound the gain
  if(gain > max){
       gain = max;
  }
  if(gain < min){
       gain = min;
  }

  //move on to next run
  lastinput = input;
  lasterr = err;
  return gain;
}

void PID::setPID(float _KP, float _KI, float _KD){
  KP = _KP;
  KI = _KI;
  KD = _KD;
}
void PID::setsetpt(float sp){
  setpt = sp;
}

void PID::setZeroPt(float z){
  gain = z;
  zeropt = z;
}
float PID::getZeroPt(){
  return zeropt;
}


PID::PID(float _KP, float _KI, float _KD, float sp){
  setPID(_KP, _KI, _KD);
  setpt = sp;
  lastinput = 0;
  lasterr = 0;
  errint = 0;
  P = 0;
```

```
  I = 0;
  D = 0;
  zeropt = 90;
}
PID::PID(){
  KP = 0;
  KI = 0;
  KD = 0;
  P = 0;
  I = 0;
  D = 0;
  setpt = 0;
  gain = 0;
}

void PID::setlimits(float lb, float ub){
        min = lb;
        max = ub;
        maxerrint = (ub - lb)/2/KI;
}
void PID::setmaxerrint(float m){
        maxerrint = m;
}

void PID::clear(){
        setpt = 0;
        lastinput = 0;
        lasterr = 0;
        errint = 0;
        P = 0;
        I = 0;
        D = 0;
}
#endif
```

```
//PID.h

#ifndef PID_H
#define PID_H

#include "WProgram.h"

class PID{
  public:
  float KP;
  float KI;
  float KD;
  float P;
  float I;
  float D;
  float setpt;
  float lastinput;
  float lasterr;
  float errint;
  float gain;
  float min;
  float max;
  float maxerrint;
  float zeropt;

  PID(float _KP, float _KI, float _KD, float sp);
  PID();
  float compute(float input);
  void setPID(float _KP, float _KI, float _KD);
  void setsetpt(float sp);
  void setlimits(float lb, float ub);
  void setmaxerrint(float m);
  void clear();
  void setZeroPt(float z);
  float getZeroPt();
};

#endif
```

8.2.3 Encoder Class Module

```
//Encoder.cpp
/**
Encoder class for Arduino
**/
#include "Encoder.h"

Encoder::Encoder(uint8_t m_A, uint8_t m_B){
        mask_A = m_A;
        mask_B = m_B;
        //setup port register L
        DDRL &= !(m_A | m_B);

        e_A = false;
        e_B = false;
        tick_cur = 0;
        tick_pre = 0;
        speed = 0;
}


//call this function from the ISR interrupt based on the Arduino
//TIMER2 which should be running at 62.5kHz 31.25 kHz
//this function will increment the ticks for this encoder as necessary
//the encoder ticks can then be read at a later time to determine the
//wheel speed
void Encoder::callback(uint8_t tPINL){
        bool t_A;
        bool t_B;

        //we should getting roughly 1400 ticks per wheel revolution
        //grab the values from the L register
        t_A = (tPINL & mask_A) == mask_A;
        t_B = (tPINL & mask_B) == mask_B;

        //edge detection then state detection
        if (t_A != e_A){
                if (t_A)
                        tick_cur += (t_B ? 1 : -1);
                else
                        tick_cur += (t_B ? -1 : 1);
        }
        if (t_B != e_B){
                if (t_B)
                        tick_cur += (t_A ? -1 : 1);
                else
                        tick_cur += (t_A ? 1 : -1);
        }

        e_B = t_B;
        e_A = t_A;
}

float Encoder::calc_vel(float samples){
        speed = ((float) (tick_cur - tick_pre)) / samples * SPEED_CONV;
        tick_pre = tick_cur;
        return speed;
}

/*
int8_t Enc_Callback(Enc* encoder, uint8_t tPINL)
{
  uint8_t t_A = (tPINL & encoder->mask_A) == encoder->mask_A;
  uint8_t t_B = (tPINL & encoder->mask_B) == encoder->mask_B;

  if (t_A != encoder->e_A)
  {
```

```
    if (t_A)
      encoder->ticks += (t_B ? 1 : -1);
    else
      encoder->ticks += (t_B ? -1 : 1);
  }

  if (t_B != encoder->e_B)
  {
       if (t_B)
         encoder->ticks += (t_A ? -1 : 1);
       else
         encoder->ticks += (t_A ? 1 : -1);
  }

  return encoder->ticks;
}
*/
```

```
//Encoder.h

#ifndef ENCODER_H
#define ENCODER_H

#include "WProgram.h"
#include "wiring.h"


#define WHEEL_CIR_M 0.37903          //circumference in m
#define SAMPLE_FREQ 31250            //sampling frequency [interrupt clock]
#define ENC_RES_TIC 1400             //encoder resolution ticks per wheel rotation

//converts ticks/samples to m/s
#define SPEED_CONV  8.46055          //WHEEL_CIR_M * SAMPLE_FREQ / ENC_RES_TIC

class Encoder{
       protected:
              //bit masks for fast processing of PINL
              uint8_t mask_A, mask_B;

              //previous encoder values
              boolean e_A, e_B;

              //the current tick count and the previous tick count
              int tick_cur, tick_pre;

              //calculated speed
              float speed;

       public:

              Encoder(uint8_t m_A, uint8_t m_B);

              //callback for the interrupt
              void callback(uint8_t tPINL);
              float calc_vel(float shft);

              inline float getSpeed() const { return speed; }
};
/*
typedef struct Enc
{
  uint8_t mask_A, mask_B;
  uint8_t e_A, e_B;
  int8_t ticks;
};

int8_t Enc_Callback(Enc* encoder, uint8_t tPINL);
*/
#endif
```

8.2.4 Motor Assembly Module

```
//motass.cpp
#include "Motass.h"

Motass::Motass(uint8_t m_A, uint8_t m_B): Encoder(m_A, m_B){
//do nothing else
}


void Motass::writezpt(){
       write(zeropt);
}

void Motass::writegain(){
  compute(speed);
  if(gain != lastpgain){
    write(gain);
    lastpgain = gain;
  }
}

void Motass::failruntime(){

  fail = (speed == 0 && (abs(setpt) > VEL_ZERO ||
                         abs(gain-zeropt) > PID_ZERO));
}

void Motass::failstartup(){
       //not connected
  fail = abs(gain - EST_ZERO) < THR_ZERO;
       //connected in reverse
  fail |= abs(gain - EST_ZERO) > THR_MAX;
}


void Motass::adjustzpt(){
       setZeroPt(gain);
}
```

```
//motass.h

#ifndef MOTASS_H
#define MOTASS_H

#define EST_ZERO 75
//startup zeros to motor from -90 to 90
#define THR_ZERO 3     //an acceptable zero (too small)
#define THR_MAX 30     //too large

//runtime zero's in air
//#define PID_ZERO 8   //from -90 to 90
//#define VEL_ZERO .5  //in m/s

//on ground
#define PID_ZERO 20    //from -90 to 90
#define VEL_ZERO 1.5 //in m/s

#include "WProgram.h"
#include <PID.h>
#include <Encoder.h>
#include <SoftwareServo.h>

//This is a Motor assembly class that includes a motor,
//an encoder, and a PID object.

class Motass: public PID, public SoftwareServo, public Encoder{
  public:
  float lastpgain;
  boolean fail;

  Motass(uint8_t m_A, uint8_t m_B);
  void writezpt();
  void writegain();
  void failruntime();
  void failstartup();
  void adjustzpt();
};

#endif
```

8.3 Matlab data processing code

8.3.1 cross_script.m

```
%plot both odo and ubi for cross reference
%
%close all;
clear all;

exps = 1:12;

plot_odo = 0;
odo_ubi_script;
plot_ubi = 0;
ubi_odo_script;
% plot_part = 0;
% sim_script;

%figure(10);
% title('Odometry');
% figure(20);
% title('Ubisense');
% figure(20+13);
% title('Histogram - Distance between paired tags');
% figure(20+14);
% title('Full Histogram - Distance between paired tags');


for i = exps

    plot_ubi = 0;
    if (plot_ubi)

        figure(20+2*i);
        subplot(2,1,1);
        hold off;
        title(['ubi ' robot_data{i}.odo.fname(17:(end-2)) ' ' num2str(i)]);

        %now connect all of the front points to see the path
        plot(robot_data{i}.ubi.event_time-robot_data{i}.ubi.t_shft,
robot_data{i}.ubi.pos_front(:,1), 'b-');

        subplot(2,1,2);
        hold off;
        title(['ubi ' robot_data{i}.ubi.fname(17:(end-2)) ' ' num2str(i)]);

        %now connect all of the front points to see the path
        plot(robot_data{i}.ubi.event_time-robot_data{i}.ubi.t_shft,
robot_data{i}.ubi.pos_front(:,2), 'b-');

        d = (robot_data{i}.ubi.pos_front(:,1:2)+robot_data{i}.ubi.pos_back(:,1:2))./2;
        pose = mean(d(1:100,:),1);
        size(pose)
        d = (robot_data{i}.ubi.pos_front(:,1:2)-robot_data{i}.ubi.pos_back(:,1:2));
        theta = atan2(d(:,2), d(:,1))-pi/2;

        figure(20+2*i+1);
        subplot(2,1,1);
        hold off;
        title(['ubi ' robot_data{i}.ubi.fname(17:(end-2)) ' ' num2str(i)]);

        %now connect all of the front points to see the path
        plot(robot_data{i}.ubi.event_time-robot_data{i}.ubi.t_shft, sin(theta), 'b:');%,
'MarkerSize', 1);
```

```
        subplot(2,1,2);
        hold off;
        title(['ubi ' robot_data{i}.ubi.fname(17:(end-2)) ' ' num2str(i)]);
        plot(robot_data{i}.ubi.event_time-robot_data{i}.ubi.t_shft, cos(theta), 'b:');%,
'MarkerSize', 1);



        figure(50+3*i+1);
        %subplot(1,3,2);
        hold on;
        x_pos = robot_data{i}.ubi.pos_front(:,1);
        y_pos = robot_data{i}.ubi.pos_front(:,2);
        rainbowplot(x_pos(1:6:end)', y_pos(1:6:end)');
    %    rainbowplot(x_pos', y_pos');
        plot(x_pos(1)', y_pos(1)', 'ro');
        plot(x_pos(end)', y_pos(end)', 'bo');
        %axis([0 20 0 12]);
        axis equal;
    end


    plot_odo = 0;
    if (plot_odo)

        warp_t = cumsum(robot_data{i}.odo.dt);
        figure(20+2*i);
        subplot(2,1, 1);
        hold on;
        title(['odo ' robot_data{i}.odo.fname(17:(end-2)) ' ' num2str(i)]);
        plot(warp_t', robot_data{i}.odo.pose(:, 1)', 'r-');
    %    rainbowplot(x_pos', y_pos');
        subplot(2,1, 2);
        hold on;

        plot(warp_t', robot_data{i}.odo.pose(:, 2)', 'r-');

        figure(20+2*i+1);
        subplot(2,1,1);
        hold on;

        plot(warp_t', sin(robot_data{i}.odo.pose(:, 3))', 'r-');

        subplot(2,1,2);
        hold on;
        plot(warp_t', cos(robot_data{i}.odo.pose(:, 3))', 'r-');


        figure(50+3*i);
        %subplot(1,3,1);
        hold on;
        x_pos = robot_data{i}.odo.pose(:,1);
        y_pos = robot_data{i}.odo.pose(:,2);
        rainbowplot(x_pos(1:6:end)', y_pos(1:6:end)');
    %    rainbowplot(x_pos', y_pos');
        plot(x_pos(1)', y_pos(1)', 'ro');
        plot(x_pos(end)', y_pos(end)', 'bo');
        axis equal;
%       plot(robot_data{i}.odo.pose(1, 1), robot_data{i}.odo.pose(1, 2), 'ro');
%       plot(robot_data{i}.odo.pose(end, 1), robot_data{i}.odo.pose(end, 2), 'bo');
    end

    plot_part = 1;
    if (plot_part)

        figure(20+2*i);
        subplot(2,1, 1);
        hold on;
        title(['part ' num2str(i)]);
```

```
        plot(robot_data{i}.part.event_time, robot_data{i}.part.mean(:, 1)', 'g-');
        a = axis;
        axis([0 robot_data{i}.part.event_time(end) a(3) a(4)]);
        %axis([100 150 a(3) a(4)]);
    %    rainbowplot(x_pos', y_pos');
        subplot(2,1, 2);
        hold on;

        plot(robot_data{i}.part.event_time, robot_data{i}.part.mean(:, 2)', 'g-');
        a = axis;
        axis([0 robot_data{i}.part.event_time(end) a(3) a(4)]);
        %axis([100 150 a(3) a(4)]);
        figure(20+2*i+1);
        subplot(2,1,1);
        hold on;

    %    plot(robot_data{i}.part.event_time, sin(robot_data{i}.part.mean(:, 3))./...
    %                                        cos(robot_data{i}.part.mean(:, 3)), 'g-');
        plot(robot_data{i}.part.event_time, sin(robot_data{i}.part.mean(:, 3)), 'g-');
        a = axis;
%       axis([0 robot_data{i}.part.event_time(end) a(3) a(4)]);
        axis([100 150 a(3) a(4)]);

        subplot(2,1,2);
        hold on;
        plot(robot_data{i}.part.event_time, cos(robot_data{i}.part.mean(:, 3)), 'g-');
        a = axis;
        axis([100 150 a(3) a(4)]);
%

        figure(50+3*i);
        %subplot(1,3,3);
        hold on;
        x_pos = robot_data{i}.part.mean(:,1);
        y_pos = robot_data{i}.part.mean(:,2);
        rainbowplot(x_pos(1:6:end)', y_pos(1:6:end)');
    %    rainbowplot(x_pos', y_pos');
        plot(x_pos(1)', y_pos(1)', 'ro');
        plot(x_pos(end)', y_pos(end)', 'bo');
        axis equal;
        %axis([0 20 0 12]);
    end



end
```

8.3.2 odo_ubi_script.m

```
%open all the files and store their data into the workspace under
%a nice struct for data_01-data_12


if (plot_odo)
    close all;
    clear all;
    plot_odo = 1;
end

%my_path = 'C:\\Documents and Settings\\Donald\\My Documents\\School\\6.021\\dophie\\Exp_1_88-
126\';
my_path = 'C:\\Users\\Donald\\Documents\\Opera\\Particle Filter\\Data\Ubi_with_Odometry\';

robot_data{1}.odo.fname = 'data_01_odo_ubi_2_laps.m';
robot_data{2}.odo.fname = 'data_02_odo_ubi_6_laps.m';
robot_data{3}.odo.fname = 'data_03_odo_ubi_10_laps.m';
robot_data{4}.odo.fname = 'data_04_odo_ubi_10_laps_CCW.m';
robot_data{5}.odo.fname = 'data_05_odo_ubi_CW.m';
robot_data{6}.odo.fname = 'data_06_odo_ubi_CCW.m';
robot_data{7}.odo.fname = 'data_07_odo_ubi_sw_R.m';
robot_data{8}.odo.fname = 'data_08_odo_ubi_sw_L.m';
robot_data{9}.odo.fname = 'data_09_odo_ubi_fun.m';
robot_data{10}.odo.fname = 'data_10_odo_ubi_slip.m';
robot_data{11}.odo.fname = 'data_11_odo_ubi_CCW_head.m';
robot_data{12}.odo.fname = 'data_12_odo_ubi_CW_head.m';


robot_data{1}.odo.init_pose = [1.3760    7.9280    0.1089];
robot_data{2}.odo.init_pose = [1.4456    8.0662   -1.9427];
robot_data{3}.odo.init_pose = [1.3883    7.9931   -1.1729];
%robot_data{3}.odo.init_pose = [0 0 0];
robot_data{4}.odo.init_pose = [1.3508    8.0761   -3.4069];
robot_data{5}.odo.init_pose = [2.1030    6.3474   -0.5358];
robot_data{6}.odo.init_pose = [2.0154    5.6828   -3.3234];
robot_data{7}.odo.init_pose = [1.2656    7.9257   -3.3197];
robot_data{8}.odo.init_pose = [15.9271    8.1824   -3.1329];
robot_data{9}.odo.init_pose = [15.9523    8.0676   -2.8924];
robot_data{10}.odo.init_pose = [1.3229    8.0212   -3.4180];
robot_data{11}.odo.init_pose = [2.7558    6.7934   -1.4680];
robot_data{12}.odo.init_pose = [2.9281    7.0528   -1.5415];

robot_data{1}.odo.opt_axis = [0 16 -10 5];
robot_data{2}.odo.opt_axis = [0 18 -7 10];
robot_data{3}.odo.opt_axis = [-5 18 -10 10];
robot_data{4}.odo.opt_axis = [-1 20 -10 10];
robot_data{5}.odo.opt_axis = [-2 4 -3 3];
robot_data{6}.odo.opt_axis = [-1 4 -3 2];
robot_data{7}.odo.opt_axis = [-5 4 -7 1];
robot_data{8}.odo.opt_axis = [-2 6 -6 2];
robot_data{9}.odo.opt_axis = [-20 10 -15 12];
robot_data{10}.odo.opt_axis = [-1 6 -6 0];
robot_data{11}.odo.opt_axis = [-1 7 -1 5];
robot_data{12}.odo.opt_axis = [-4 3 -6 4];



%theta_dot = linspace(.51, .59, 36);
%theta_scale = .55;
theta_scale = .62;
vel_scale = 6;
%theta_dot = linspace(.5, .8, 6);
for i = exps

    odo = fopen([my_path robot_data{i}.odo.fname]);
```

```
    data = fscanf(odo, ['%i' '%i' '%i' '%i' '%i' '%i' '%i' '%*s' '\n'], [7, inf])';
    fclose(odo);


    data(:,1) = data(:,1)/1000;
    data(:,2) = data(:,2)/1000/vel_scale;
    data(:,3) = data(:,3)/1000/vel_scale;
    data(:,4) = data(:,4)/1000/vel_scale;
    y_dot = (data(:,2)*cos(0)+data(:,3)*cos(120/180*pi)+data(:,4)*cos(240/180*pi));
    x_dot = (data(:,2)*sin(0)+data(:,3)*sin(120/180*pi)+data(:,4)*sin(240/180*pi));
    t_dot = -(data(:,2)+data(:,3)+data(:,4))/theta_scale;

    %now integrate odometry
    dt = diff([0; data(:,1)]);

    x_pos = robot_data{i}.odo.init_pose(1);
    y_pos = robot_data{i}.odo.init_pose(2);
    t_pos = robot_data{i}.odo.init_pose(3);
    for j = 2:length(dt)
%    for j = 2:(length(dt)/3)
        t_pos = [t_pos; t_pos(end)+dt(j)*t_dot(j)];
        x_pos = [x_pos; x_pos(end) + dt(j)*y_dot(j)*cos(t_pos(j)) + ...
                                     dt(j)*x_dot(j)*sin(t_pos(j))];
        y_pos = [y_pos; y_pos(end)+dt(j)*y_dot(j)*sin(t_pos(j)) + ...
                                   -dt(j)*x_dot(j)*cos(t_pos(j))];
    end

    robot_data{i}.odo.dt = dt+.003;
    robot_data{i}.odo.event_time = cumsum(robot_data{i}.odo.dt);
    robot_data{i}.odo.pose = [x_pos y_pos t_pos];
    robot_data{i}.odo.robot_vel = [x_dot y_dot t_dot];

    if (plot_odo)
        figure(10);
        subplot(3,4, i);
        hold on;
        axis equal
        rainbowplot(x_pos(1:6:end)', y_pos(1:6:end)');
    %    rainbowplot(x_pos', y_pos');
        plot(x_pos(1)', y_pos(1)', 'ro');
        plot(x_pos(end)', y_pos(end)', 'bo');
    end
end
```

8.3.3 ubi_odo_script.m

```
%open all the files and store their data into the workspace under
%a nice struct for data_01-data_12

if (plot_ubi)
    close all;
    clear all;
    plot_ubi = 1;
end
%my_path = 'C:\\Documents and Settings\\Donald\\My Documents\\School\\6.021\\dophie\\Exp_1_88-
126\';
my_path = 'C:\\Users\\Donald\\Documents\\Opera\\Particle Filter\\Data\Ubi_with_Odometry\';

robot_data{1}.ubi.fname = 'data_01_ubi_odo_2_laps.txt';
robot_data{2}.ubi.fname = 'data_02_ubi_odo_6_laps.txt';
robot_data{3}.ubi.fname = 'data_03_ubi_odo_10_laps.txt';
robot_data{4}.ubi.fname = 'data_04_ubi_odo_10_laps_CCW.txt';
robot_data{5}.ubi.fname = 'data_05_ubi_odo_CW.txt';
robot_data{6}.ubi.fname = 'data_06_ubi_odo_CCW.txt';
robot_data{7}.ubi.fname = 'data_07_ubi_odo_sw_R.txt';
robot_data{8}.ubi.fname = 'data_08_ubi_odo_sw_L.txt';
robot_data{9}.ubi.fname = 'data_09_ubi_odo_fun.txt';
robot_data{10}.ubi.fname = 'data_10_ubi_odo_slip.txt';
robot_data{11}.ubi.fname = 'data_11_ubi_odo_CCW_head.txt';
robot_data{12}.ubi.fname = 'data_12_ubi_odo_CW_head.txt';


robot_data{1}.ubi.t_shft = 69;
robot_data{2}.ubi.t_shft = 47;
robot_data{3}.ubi.t_shft = 32.5;
%robot_data{3}.ubi.t_shft = 0;
robot_data{4}.ubi.t_shft = 44;
robot_data{5}.ubi.t_shft = 33;
robot_data{6}.ubi.t_shft = 22;
robot_data{7}.ubi.t_shft = 35;
robot_data{8}.ubi.t_shft = 92;
robot_data{9}.ubi.t_shft = 52;
robot_data{10}.ubi.t_shft = 57;
robot_data{11}.ubi.t_shft = 18;
robot_data{12}.ubi.t_shft = 68;

tag_back = '020-000-139-058';
tag_front = '020-000-139-059';
all_dist = [];
for i = exps
    %PARSE THE UBISENSE DATA INTO A STRUCT
    if (i == 6)
        robot_data{i}.ubi.skip = 600;
    else
        robot_data{i}.ubi.skip = 20;
    end

    %get the tags
    ubi = fopen([my_path robot_data{i}.ubi.fname]);
    robot_data{i}.ubi.tags = fscanf(ubi, ['%s' '%*s' '%*s' '%*s' '%*s' '%*s' '%*s' '%*s' '\n'],
[15, inf])';
    fclose(ubi);

    %get the positions for each tag
    ubi = fopen([my_path robot_data{i}.ubi.fname]);
    robot_data{i}.ubi.pos = fscanf(ubi, ['%*s' '%*s' '%f' '%f' '%f' '%*s' '%*s' '%*s' '\n'], [3,
inf])';
    fclose(ubi);

    %filter the data so that we don't have tags that don't match
    robot_data{i}.ubi.pos_back = [];
```

```
    robot_data{i}.ubi.pos_front = [];
    on_front = 1;
    interp_tag = 1;

    if (interp_tag)
        for j = robot_data{i}.ubi.skip:length(robot_data{i}.ubi.tags)-1
            %this is the front tag we will just store it, but we must
            %interp the nearest back tags
            if (strcmp(robot_data{i}.ubi.tags(j,:), tag_front))
                robot_data{i}.ubi.pos_front = ...
                    [robot_data{i}.ubi.pos_front; robot_data{i}.ubi.pos(j,:)];

                %our neighbor tags are back tags which is good
                if (strcmp(robot_data{i}.ubi.tags(j-1,:), tag_back) && ...
                    strcmp(robot_data{i}.ubi.tags(j+1,:), tag_back))
                    robot_data{i}.ubi.pos_back = [robot_data{i}.ubi.pos_back;
                        (robot_data{i}.ubi.pos(j-1,:)+robot_data{i}.ubi.pos(j+1,:))./2];
                else
                    %[robot_data{i}.ubi.tags(j-1,:), robot_data{i}.ubi.tags(j+1,:)]
                    robot_data{i}.ubi.pos_back = [robot_data{i}.ubi.pos_back;
                        robot_data{i}.ubi.pos_back(end,:)];
                end

            end
        end
    else
        for j = robot_data{i}.ubi.skip:length(robot_data{i}.ubi.tags)

            if (on_front && strcmp(robot_data{i}.ubi.tags(j,:), tag_front))
                robot_data{i}.ubi.pos_front = [robot_data{i}.ubi.pos_front;
robot_data{i}.ubi.pos(j,:)];
                on_front = ~on_front;
            elseif (~on_front && strcmp(robot_data{i}.ubi.tags(j,:), tag_back))
                %append back tags
                robot_data{i}.ubi.pos_back = [robot_data{i}.ubi.pos_back;
robot_data{i}.ubi.pos(j,:)];
                on_front = ~on_front;
            end
        end

    end

    %assign a time stamp from 0
    %we will put in an offset later
    %ignore the first 7 entries
    %we get roughly 8 per second so take the length and div by 8
    %we get 2 tag updates at the 'same time' so lets assume they are at the
    %same time step
    samples = length(robot_data{i}.ubi.pos_front);
    robot_data{i}.ubi.event_time = linspace(0, samples/4, samples);
    robot_data{i}.ubi.dt = diff(robot_data{i}.ubi.event_time);




    robot_data{i}.ubi.diameter = [];
    for j = 2:min(length(robot_data{i}.ubi.pos_back),length(robot_data{i}.ubi.pos_front))
        dis = [robot_data{i}.ubi.pos_back(j,1)-robot_data{i}.ubi.pos_front(j,1)
               robot_data{i}.ubi.pos_back(j,2)-robot_data{i}.ubi.pos_front(j,2)];
        robot_data{i}.ubi.diameter(j) = sqrt(sum(dis.*dis));
    end
    all_dist = [all_dist; robot_data{i}.ubi.diameter'];



    if (plot_ubi)
        theta = [];
        if (1)
```

```
            figure(10+i)
            %subplot(3,4,i);
            hold on;
            %plot the robot as a vector
            for j = 2:min(length(robot_data{i}.ubi.pos_back),length(robot_data{i}.ubi.pos_front))
                %if (abs(robot_data{i}.ubi.diameter(j)-.29) < .10)
                    plot([robot_data{i}.ubi.pos_back(j,1) robot_data{i}.ubi.pos_front(j,1)], ...
                         [robot_data{i}.ubi.pos_back(j,2) robot_data{i}.ubi.pos_front(j,2)], 'b-
');
                    plot(robot_data{i}.ubi.pos_front(j,1), ...
                         robot_data{i}.ubi.pos_front(j,2), 'ro');


                     d = (robot_data{i}.ubi.pos_front(j,1:2)-robot_data{i}.ubi.pos_back(j,1:2));
                    theta = [theta; atan2(d(:,2), d(:,1))];
               %end
            end

            %now connect all of the front points to see the path
            rainbowplot(robot_data{i}.ubi.pos_front(:,1)', ...
                 robot_data{i}.ubi.pos_front(:,2)');

            axis equal;
            %keep it all boxed but our fun plot
%             if (i ~= 9)
%                 axis([0 18 2 10]);
%             end
        end
        %plot the histograms on a subplot
%         figure(20+13)
%         subplot(3,4,i)
%         hist(min(robot_data{i}.ubi.diameter, 1), 20)
    end
end

[pdf, bins_k] = ksdensity(min(all_dist, 1), 'width', .02);
for i=1:12
    robot_data{i}.ubi.pdf = [pdf', bins_k'];
end
%plot the entire histogram on a main plot
%plot_ubi = 1;
if (plot_ubi)
%     figure(20+17);
%     hist(theta, 20);
%     [mean(theta) std(theta)]
    figure(20+15)
    hist(min(all_dist, 1), 100);
    [counts, bins_h] = hist(min(all_dist, 1), 100);
    hold on;
    plot(bins_k, pdf*max(counts)/max(pdf), 'r-', 'linewidth', 2);

end



%robot_data{1}.ubi = load([my_path 'data_01_ubi_odo_2_laps.txt']);

%mystr = '  000-000-000-004 ULocationIntegration::Tag   5.73    5.77    1.21    5/1/2010 11:29:32
PM';
%mystr = '  000-000-000-004 ULocationIntegration::Tag   5.73';% 5.77    1.21    5/1/2010 11:29:32
PM'
%tag = sscanf(tags, [], [15, inf])'

%pos = sscanf(mystr, ['%*s' '%*s' '%f' '%f' '%f' '%*s'])
```

8.3.4 sim_script.m

```
%run simulator script

%first run both the scripts but don't plot anything!
% plot_odo = 0;
% odo_ubi_script;
% plot_ubi = 0;
% ubi_odo_script;
% %
% This stores the data into robot_date{i}
%
% now run our simulator on a set of robot data

for i = exps
    robot_data{i}.part = [];
    robot_data{i}.part = simulator(robot_data{i},0);

    if (plot_part)
        figure(20+i);
        subplot(3,1, 1);
        hold on;
        title(['part ' num2str(i)]);
        plot(robot_data{i}.part.event_time, robot_data{i}.part.mean(:, 1)', 'g-');
    %    rainbowplot(x_pos', y_pos');
        subplot(3,1, 2);
        hold on;

        plot(robot_data{i}.part.event_time, robot_data{i}.part.mean(:, 2)', 'g-');

        subplot(3,1, 3);
        hold on;

    %    plot(robot_data{i}.part.event_time, sin(robot_data{i}.part.mean(:, 3))./...
    %                                        cos(robot_data{i}.part.mean(:, 3)), 'g-');
        plot(robot_data{i}.part.event_time, sin(robot_data{i}.part.mean(:, 3)), 'g-');
        plot(robot_data{i}.part.event_time, cos(robot_data{i}.part.mean(:, 3)), 'g:');
    end
end
```

8.3.5 simulator.m

```
%Robot Simulator
%run through the simulated data one point at a time and then update the
%robot position with the odometry and the absolute positioning data.
%Assume the robot data set has been created and parsed into robot_data

%now lets run a simulation on this data. Basically lets grab this robot
%data. So sim_dat = robot_data{i}

%what is in robot_data?
%robot_data{i} is the ith robot run
%robot_data{i}.odo is a reference to all the odometry data for this ith run
%robot_data{i}.ubi is a reference to all the ubisense data for this ith run

%odo.fname      string      1x1     the file name for this odometry set
%odo.init_pose  [x,y,t]     1x3     the initial pose for this odometry set
%odo.dt         [...]'      Nx1     the dt between this odometry
%                                   measurement and the last odometry
%                                   measurement [dt_1, ...dt_n]
%odo.pose       [x,y,t]     Nx3     The raw robot pose as calculated by
%                                   straight odometry integration
%odo.robot_vel  [x_dot y_dot t_dot] The velocity at this current time
%                           Nx3     Measured from the encoders ~8/second

%robot animator draw this robot clear last robot
%clear last particles.

function results = simulator(real_data, animate)
%stores all of our goodies in here
global G;
G = [];
%setup our global variables here
init_globals(real_data, animate);
step_PF(0);  %initializes particles

if (G.Params.animate)
    init_animation();
end



while (G.Odometry.event_time(end-1) > G.Params.last_event)
    ubi_now = next_measurement();
    %if (~ubi_now || ...
    %    ubi_now && abs(G.Ubisense.diameter(G.Ubisense.cur_index)-.28) < .15)
        step_PF(ubi_now);
    %end

end

%clear the zero entries
grab=logical(sum(abs(G.Particles.mean),2) ~= 0);
G.Particles.mean=G.Particles.mean(grab,:);
G.Particles.std=G.Particles.std(grab,:);
G.Particles.event_time=G.Particles.event_time(grab,:);
results = G.Particles;
end


function is_ubi = next_measurement()
    global G;
    o_i=G.Odometry.cur_index;
    u_i=G.Ubisense.cur_index;
    last_time = G.Params.last_time;
    last_event = G.Params.last_event;
```

```
    %increment our index to the next odometry reading
    if (G.Odometry.event_time(o_i) < ...
        G.Ubisense.event_time(u_i)-G.Ubisense.t_shft)
        G.Odometry.cur_index=o_i+1;
        if (~mod(G.Odometry.cur_index, G.Params.skip))
            if (G.Params.animate)
                pause((G.Odometry.event_time(o_i)-last_event-etime(last_time, clock))/...
                    G.Params.time_warp_scale);
                run_animation('odo', 1);
                last_time = clock;
            end

        end
        last_event = G.Odometry.event_time(o_i);

        is_ubi = 0;
    %increment to the next ubisense
    else
        G.Ubisense.cur_index=u_i+1;
        if (~mod(G.Ubisense.cur_index, ceil(G.Params.skip/4)))
            if (G.Params.animate)
                pause((G.Ubisense.event_time(u_i)-G.Ubisense.t_shft -...
                        last_event-etime(last_time,clock))/G.Params.time_warp_scale);
                run_animation('ubi', 2);
                %if (u_i > 961)
                %    pause;
                %end
                last_time = clock;
            end

        end
        last_event = G.Ubisense.event_time(u_i)-G.Ubisense.t_shft;
        is_ubi = 1;
    end


    G.Params.last_time = last_time;
    G.Params.last_event = last_event;

end



function init_globals(real_data, animate)
global G;

figure(1); clf;
%G.Params declared here
G.Params.time_warp_scale = 50;
G.Params.path_plot_hist = 200;
G.Params.last_time = clock;
G.Params.last_event = 0;
G.Params.skip = 8;
G.Params.part_has_init = 0;
G.Params.animate = animate;

%G.Odometry data stored here
G.Odometry = real_data.odo;
G.Odometry.cur_index = 1;       %cur data index
G.Odometry.cur_time = 0;
G.Odometry.FIG = subplot(3,3,[1 2]);
G.Odometry.t_fig = subplot(3,3,3);
G.Odometry.hRobot=[];    %handle to the robot figure
G.Odometry.hPath=[];
G.Odometry.tPath=[];
G.Odometry.path_color = [1 0 1];
G.Odometry.animate = 1;
```

```
%G.Ubisense data stored here
G.Ubisense = real_data.ubi;
G.Ubisense.cur_index = find(G.Ubisense.event_time-G.Ubisense.t_shft>0, 1);       %cur data index
G.Ubisense.cur_time = 0;
G.Ubisense.FIG = subplot(3,3,[4 5]);
G.Ubisense.t_fig = subplot(3,3,6);
G.Ubisense.hRobot = [];
G.Ubisense.hPath=[];
G.Ubisense.tPath=[];
G.Ubisense.path_color = [1 0 1];
G.Ubisense.animate = 1;

%G.FIG stuff for drawing pretty figures
G.FIG.axis=[0 18 0 10];
G.FIG.hRobot=[];    %handle to the robot figure
G.FIG.hObs=[];
G.FIG.hBeacons=[];
G.FIG.hPath=[];
G.FIG.hTitle=[];
G.FIG.path_cur_co = 1;
% G.FIG.clrs.beacons{1}=[0 0 0];
% G.FIG.clrs.beacons{2}=[.4 .4 .4];
% G.FIG.clrs.robot{1}=[0 0 1];
% G.FIG.clrs.robot{2}=[0 1 1];
% G.FIG.clrs.path{1}=[0 1 0];
% G.FIG.clrs.obs{1}=[1 0 0];
G.FIG.prevtime=[];
end

function init_animation()
global G;
%figure(1);
%get(G.Odometry.FIG,'type')
subplot(G.Odometry.FIG),hold on;
set(gcf,'DoubleBuffer','on');
%axis(G.Odometry.opt_axis);
axis(G.FIG.axis);
%axis equal

subplot(G.Odometry.t_fig),hold on;
set(gcf,'DoubleBuffer','on');
axis([-1 1 -1 1]);
axis square

%get(G.Ubisense.FIG,'type')
subplot(G.Ubisense.FIG),hold on;
set(gcf,'DoubleBuffer','on');
axis(G.FIG.axis);
%axis equal

subplot(G.Ubisense.t_fig),hold on;
set(gcf,'DoubleBuffer','on');
axis([-1 1 -1 1]);
axis square

subplot(G.Particles.FIG),hold on;
set(gcf,'DoubleBuffer','on');
%axis(G.Odometry.opt_axis);
axis(G.FIG.axis);
%axis equal

subplot(G.Particles.t_fig),hold on;
set(gcf,'DoubleBuffer','on');
axis([-1 1 -1 1]);
axis square

end
```

8.3.6 run_animation.m

```
%animation functions
%all the things we need to animate our robot is stored in here
function run_animation(animate_what, animate_opt)

animate_title();
if (strcmp(animate_what, 'odo'))

    animate_robot_odometry();
    if (animate_opt)
        animate_path_odometry();
    end

elseif (strcmp(animate_what, 'ubi'))
    animate_robot_ubisense();
    if (animate_opt)
        animate_path_ubisense(animate_opt);
    end

elseif (strcmp(animate_what, 'part'))
    animate_robot_particles(animate_opt);
    animate_path_particles();

end

end



%update the robot animation in the G.FIG.hRobot
%draw with the current pose based on the current index
function animate_robot_odometry()
global G;
%figure(1);
t=pi*(.5+linspace(0,1));
cs=[cos(t);sin(t)];
rr=.15;
rl=.35;
rs=[[rl 0]' rr*cs [rl 0]'];

subplot(G.Odometry.FIG)
pose = G.Odometry.pose(G.Odometry.cur_index,:);
rs=transform_points([pose(1) pose(2) pose(3)+1*pi/2],rs);
if isempty(G.Odometry.hRobot)
    G.Odometry.hRobot=patch(rs(1,:),rs(2,:),'r');
    %get(G.FIG.hRobot);
    %plot([0 10], [0 10]);
else
    set(G.Odometry.hRobot,'XData',rs(1,:));
    set(G.Odometry.hRobot,'YData',rs(2,:));
end

end

function animate_path_odometry()
global G;
%figure(1);
idx=G.Odometry.cur_index;
i0=max([1 idx-G.Params.path_plot_hist+1]);
i1=idx;
% x=G.EKF.X(1,i0:i1);
% y=G.EKF.X(2,i0:i1);
x=G.Odometry.pose(i0:i1,1);
y=G.Odometry.pose(i0:i1,2);
```

```
if isempty(G.Odometry.hPath)
    h=plot(x,y,'LineStyle','-.');
    set(h,'Color',G.Odometry.path_color);
    G.Odometry.hPath{1}=h;
else
    set(G.Odometry.hPath{1},'XData',x);
    set(G.Odometry.hPath{1},'YData',y);
end

subplot(G.Odometry.t_fig)
t = G.Odometry.pose(i0:i1,3);

x =  [linspace(0, 1, length(t)).*cos(t+pi/2)' 0];
y =  [linspace(0, 1, length(t)).*sin(t+pi/2)' 0];
if isempty(G.Odometry.tPath)
    h=plot(x,y,'LineStyle','-');
    set(h,'Color',[0 0 1]);
    G.Odometry.tPath{1}=h;
else
    %x = reshape([x;zeros(1,length(x))], 1, 2*length(x));
    %y = reshape([y;zeros(1,length(y))], 1, 2*length(y));
    set(G.Odometry.tPath{1},'XData',x);
    set(G.Odometry.tPath{1},'YData',y);
end


end



function animate_robot_ubisense()
global G;
%figure(1);
t=pi*(.5+linspace(0,1));
cs=[cos(t);sin(t)];
rr=.15;
rl=.35;
rs=[[rl 0]' rr*cs [rl 0]'];

subplot(G.Ubisense.FIG);
pose = get_ubi_pose(G.Ubisense.pos_front(G.Ubisense.cur_index,:), ...
                    G.Ubisense.pos_back(G.Ubisense.cur_index,:));
rs=transform_points([pose(1) pose(2) pose(3)+1*pi/2],rs);
if isempty(G.Ubisense.hRobot)
    G.Ubisense.hRobot=patch(rs(1,:),rs(2,:),'r');
    %get(G.FIG.hRobot);
    %plot([0 10], [0 10]);
else
    set(G.Ubisense.hRobot,'XData',rs(1,:));
    set(G.Ubisense.hRobot,'YData',rs(2,:));
end

end

function animate_path_ubisense(animate_opt)
global G;
%figure(1);
idx=G.Ubisense.cur_index;
i0=max([1 idx-G.Params.path_plot_hist/8+1]);
i1=idx;
x=G.Ubisense.pos_front(i0:i1,1);
y=G.Ubisense.pos_front(i0:i1,2);

use_i = logical(abs(G.Ubisense.diameter(i0:i1)-.28) < .15);
if isempty(G.Ubisense.hPath)
    h=plot(x,y,'LineStyle','-.');
    set(h,'Color',G.Ubisense.path_color);
    G.Ubisense.hPath{1}=h;
else
```

```
    if (animate_opt == 2)
        x = x(use_i);
        y = y(use_i);
    end

    set(G.Ubisense.hPath{1},'XData',x);
    set(G.Ubisense.hPath{1},'YData',y);
end

subplot(G.Ubisense.t_fig)
p = get_ubi_pose(G.Ubisense.pos_front(i0:i1,:),G.Ubisense.pos_back(i0:i1,:));

x =  linspace(0, 1, length(p)).*cos(p(:,3)+pi/2)';
y = linspace(0, 1, length(p)).*sin(p(:,3)+pi/2)';
if isempty(G.Ubisense.tPath)
    h=plot(x,y,'LineStyle','-');
    set(h,'Color',[0 0 1]);
    G.Ubisense.tPath{1}=h;
else
    if (animate_opt == 2)
        x = x(use_i);
        y = y(use_i);
    end
    %x = reshape([x;zeros(1,length(x))], 1, 2*length(x));
    %y = reshape([y;zeros(1,length(y))], 1, 2*length(y));
    set(G.Ubisense.tPath{1},'XData',[x 0]);
    set(G.Ubisense.tPath{1},'YData',[y 0]);
end
end


function animate_robot_particles(animate_opt)
global G;

t=pi*(.5+linspace(0,1));
cs=[cos(t);sin(t)];
rr=.15;
rl=.35;
rs=[[rl 0]' rr*cs [rl 0]'];

subplot(G.Particles.FIG)
%figure(3);
pose = G.Particles.mean(G.Particles.cur_index,1:3);
rs=transform_points([pose(1) pose(2) pose(3)+1*pi/2],rs);
if isempty(G.Particles.hRobot)
    G.Particles.hRobot=patch(rs(1,:),rs(2,:),'r');
    %get(G.FIG.hRobot);
    %plot([0 10], [0 10]);
else
    set(G.Particles.hRobot,'XData',rs(1,:));
    set(G.Particles.hRobot,'YData',rs(2,:));
end


if (animate_opt)
    x=G.Particles.for_x(:,1);
    y=G.Particles.for_y(:,1);

    if isempty(G.Particles.hPart)
        G.Particles.hPart=plot(x, y, '.','MarkerSize', 4);
        %get(G.FIG.hRobot);
        %plot([0 10], [0 10]);
    else
        set(G.Particles.hPart,'XData',x);
        set(G.Particles.hPart,'YData',y);
    end

end
```

```
end

function animate_path_particles()
global G;

idx=G.Particles.cur_index;
i0=max([1 idx-G.Params.path_plot_hist+1]);
i1=idx;
% x=G.EKF.X(1,i0:i1);
% y=G.EKF.X(2,i0:i1);
x=G.Particles.mean(i0:i1,1);
y=G.Particles.mean(i0:i1,2);

if isempty(G.Particles.hPath)
    h=plot(x,y,'LineStyle','-.');
    set(h,'Color',G.Particles.path_color);
    G.Particles.hPath{1}=h;
else
    set(G.Particles.hPath{1},'XData',x);
    set(G.Particles.hPath{1},'YData',y);
end
%figure(1);
subplot(G.Particles.t_fig)
t=G.Particles.mean(i0:i1,3);

x =  [linspace(0, 1, length(t)).*cos(t+pi/2)' 0];
y =  [linspace(0, 1, length(t)).*sin(t+pi/2)' 0];
if isempty(G.Particles.tPath)
    h=plot(x,y,'LineStyle','-');
    set(h,'Color',[0 0 1]);
    G.Particles.tPath{1}=h;
else
    %x = reshape([x;zeros(1,length(x))], 1, 2*length(x));
    %y = reshape([y;zeros(1,length(y))], 1, 2*length(y));
    set(G.Particles.tPath{1},'XData',x);
    set(G.Particles.tPath{1},'YData',y);
end

end


function animate_title()
global G;

%d=[idx G.EKF.t(idx)  G.EKF.X(1:2,idx)' G.EKF.X(3,idx)*180/pi];
d = [G.Odometry.cur_index G.Ubisense.cur_index];
sFmt='PF odo=%5d ubi=%5d';
sTitle=sprintf(sFmt,d);
if isempty(G.FIG.hTitle)
    figure(1);
    G.FIG.hTitle{1}=title(sTitle);
else
    set(G.FIG.hTitle{1},'String',sTitle);
end

end

function pose = get_ubi_pose(front, back)
    d = (front(:,1:2)+back(:,1:2))./2;
    pose = d;
    d = front(:,1:2)-back(:,1:2);
    theta = awrap(atan2(d(:,2), d(:,1))-pi/2);
    pose = [pose theta];
end


function [a1]=awrap(a)
% function [a1]=awrap(a);
% Wrap angles (radians) to -pi,pi
```

```
% Argument a must be a vector
a1=a;
tran=0;
[n,d] = size(a1);
if and(n==1,d>1)
  tran=1;
  a1=a1';
end
ia=find(~isnan(a1));
a1(ia)=atan2(sin(a1(ia)),cos(a1(ia)));
if tran==1
  a1=a1';
end
end
```

```
%transform the points in p0 by Tab = [a, b, theta]
%used for animating the robot shape into the global
%coordinate frame by the REAL robot pose
function [p1]=transform_points(Tab,p0);
c=cos(Tab(3));s=sin(Tab(3));
M=[c -s;s c];
if size(p0,1)>2
    p1=p0*M';
    p1=p1+repmat([Tab(1) Tab(2)],size(p0,1),1);
else
    p1=M*p0;
    p1=p1+repmat([Tab(1) Tab(2)]',1,size(p0,2));
end
end
```

8.3.7 step_PF.m

```
%particle filter step

function step_PF(ubi_now)
    global G;

    if (~G.Params.part_has_init)
        init_particles();
        return;
    end


    o_i=G.Odometry.cur_index;
    u_i=G.Ubisense.cur_index;
    p_i = G.Particles.cur_index;

    if (ubi_now)
        update_ubisense();
    else
        update_odometry();
    end

    p_i = p_i+1;
    G.Particles.event_time(p_i) = G.Params.last_event;
    G.Particles.mean(p_i,:) = get_mean_state();
    G.Particles.std(p_i,:) = get_std_state();
    G.Particles.cur_index = p_i;

    if (G.Params.animate)
        run_animation('part', 1);
    end


    %increment our index to the next odometry reading

end

function update_ubisense()
global G;
u_i = G.Ubisense.cur_index;

z = [get_ubi_pose(G.Ubisense.pos_front(u_i,:), G.Ubisense.pos_back(u_i,:)),...
     G.Ubisense.diameter(u_i)];
obs_prob = get_obs_prob(z);
%new_weights = get_weights_from_prob(obs_prob);
%make_new_particles(new_weights);

new_weights = get_comb_weights(obs_prob);
make_comb_particles(new_weights);

end

%z is the observation [x, y, t, diameter] diameter is the spacing between the
%two tags. We use this as a measure of how good this measurement is
function obs_prob = get_obs_prob(z)
global G;
d_std = 1/interp1(G.Ubisense.pdf(:,2), G.Ubisense.pdf(:,1), z(4));
%d_std = 1;
x_prob = normpdf(G.Particles.for_x(:,1), z(1), G.Particles.obs_x_std*d_std);
y_prob = normpdf(G.Particles.for_y(:,1), z(2), G.Particles.obs_y_std*d_std);

%we calculate the std for our obs model based on the 'goodness' of our
%observation

t_prob = normpdf(awrap(z(3)-G.Particles.for_t(:,1)), 0, G.Particles.obs_t_std*d_std);
```

```
obs_prob = [x_prob y_prob t_prob];
end

function new_weights = get_weights_from_prob(obs_prob)
new_weights = [obs_prob(:,1)/sum(obs_prob(:,1)),...
               obs_prob(:,2)/sum(obs_prob(:,2)),...
               obs_prob(:,3)/sum(obs_prob(:,3))];
end

function make_new_particles(new_weights)
global G;
old_p = [G.Particles.for_x(:,1) G.Particles.for_y(:,1) G.Particles.for_t(:,1)];
new_p = zeros(G.Particles.num,3);

cum = cumsum(new_weights);%./repmat(sum(new_weights),G.Particles.num,1);
noise = [G.Particles.pro_x_std G.Particles.pro_y_std G.Particles.pro_t_std];
for i = 1:G.Particles.num

    for j = 1:3
        %new_p(cur_i(j):(cur_i(j)+num_new(i,j)-1),j) = repmat(old_p(i,1),num_new(i,j),1);
        %cur_i(j) = cur_i(j)+num_new(i,j);
        ind = find(rand(1)<=cum(:,j),1);
        if (~length(ind)), ind=1; end
        new_p(i,j)=old_p(ind,j);
    end
end
new_p = make_noisy(new_p, noise);
G.Particles.for_x(:,1) = new_p(:,1);
G.Particles.for_y(:,1) = new_p(:,2);
G.Particles.for_t(:,1) = awrap(new_p(:,3));

end



function new_weights = get_comb_weights(obs_prob)
new_weights = sum(obs_prob,2);
new_weights = new_weights./sum(new_weights);
end

function make_comb_particles(new_weights)
global G;
old_p = [G.Particles.for_x(:,1) G.Particles.for_y(:,1) G.Particles.for_t(:,1)];
new_p = zeros(G.Particles.num,3);

cum_w = cumsum(new_weights);%./repmat(sum(new_weights),G.Particles.num,1);
noise = [G.Particles.pro_x_std G.Particles.pro_y_std G.Particles.pro_t_std];
for i = 1:G.Particles.num
    ind = find(rand(1)<=cum_w,1);
    if (~length(ind)), ind=1; end
    new_p(i,:)=old_p(ind,:);
end
new_p = make_noisy(new_p, noise);
G.Particles.for_x(:,1) = new_p(:,1);
G.Particles.for_y(:,1) = new_p(:,2);
G.Particles.for_t(:,1) = awrap(new_p(:,3));

end
```

```
function pose = get_ubi_pose(front, back)
    d = (front(:,1:2)+back(:,1:2))./2;
    pose = d;
    d = front(:,1:2)-back(:,1:2);
    theta = awrap(atan2(d(:,2), d(:,1))-pi/2);
    pose = [pose theta];
end

function update_odometry()
global G;
o_i = G.Odometry.cur_index;

%propagate the particles forward by the odometry readings
%no resampling here


v = repmat(G.Odometry.robot_vel(o_i,:), G.Particles.num,1);
noise = [G.Particles.pro_x_dot_std,...
         G.Particles.pro_y_dot_std,...
         G.Particles.pro_t_dot_std];

%v = make_noisy(v, noise);

dt = G.Odometry.dt(o_i);

G.Particles.for_t = [awrap(G.Particles.for_t(:,1)+v(:,3)*dt), v(:,3)];

G.Particles.for_x = [G.Particles.for_x(:,1)+...
        v(:,2).*dt.*cos(G.Particles.for_t(:,1))+v(:,1).*dt.*sin(G.Particles.for_t(:,1)), v(:,1)];
G.Particles.for_y = [G.Particles.for_y(:,1)+...
        v(:,2).*dt.*sin(G.Particles.for_t(:,1))-v(:,1).*dt.*cos(G.Particles.for_t(:,1)), v(:,2)];


end



%make all of our particles
function init_particles()
global G;
figure(1);
%admin stuff here
G.Particles.cur_index = 1;
G.Particles.num = 100;
G.Particles.t_fig = subplot(3,3,9);
G.Particles.FIG = subplot(3,3,[7 8]);


G.Particles.hRobot = [];
G.Particles.hNoise = [];
G.Particles.hPath = [];
G.Particles.tPath = [];
G.Particles.hPart = [];
G.Particles.path_color = [1 0 1];


%particles for x and x_dot
G.Particles.pro_x_std = .15;
%G.Particles.pro_x_dot_std = .02;
G.Particles.pro_x_dot_std = .05;
G.Particles.obs_x_std = .15;
G.Particles.obs_x_dot_std = .1;
G.Particles.for_x = make_noisy(repmat([G.Odometry.init_pose(1) 0], G.Particles.num, 1),...
                        [G.Particles.pro_x_std, G.Particles.pro_x_dot_std]);
```

```
%particles for y and y_dot
G.Particles.pro_y_std = .15;
%G.Particles.pro_y_dot_std = .02;
G.Particles.pro_y_dot_std = .05;
G.Particles.obs_y_std = .15;
G.Particles.obs_y_dot_std = .1;
G.Particles.for_y = make_noisy(repmat([G.Odometry.init_pose(2) 0], G.Particles.num, 1),...
                        [G.Particles.pro_y_std, G.Particles.pro_y_dot_std]);

%particles for t and t_dot
G.Particles.pro_t_std = .15;
G.Particles.pro_t_dot_std = .15;
G.Particles.obs_t_std = 1.5;
G.Particles.obs_t_dot_std = .1;
G.Particles.for_t = make_noisy(repmat([G.Odometry.init_pose(3) 0], G.Particles.num, 1),...
                        [G.Particles.pro_t_std, G.Particles.pro_t_dot_std]);

%time stamps for these particles at this index
%mean x, y, t and x_dot, y_dot, t_dot
%preallocate the mean for speed
G.Particles.mean = zeros(length(G.Ubisense.pos_front)+...
                         length(G.Odometry.pose)-1, 6);
G.Particles.mean(G.Particles.cur_index,:) = get_mean_state();

G.Particles.event_time = zeros(length(G.Ubisense.pos_front)+...
                         length(G.Odometry.pose)-1, 1);

G.Particles.std = zeros(length(G.Ubisense.pos_front)+...
                         length(G.Odometry.pose)-1, 6);
G.Particles.std(G.Particles.cur_index,:) = get_std_state();

%std x, y, t and x_dot, y_dot, t_dot
%no history of particles just our x, y, t

%we are initialized
G.Params.part_has_init = 1;
end

%we are expecting the input particles to be of the form
% [x0 x_dot0; x1 x_dot1; x2 x_dot2; ...]
function noisy = make_noisy(particles, stds)
global G;
    noisy = particles + ...
        repmat(stds,length(particles),1).*randn(length(particles),length(stds));
end

function all_mean = get_mean_state()
global G;
    x = mean(G.Particles.for_x, 1);
    y = mean(G.Particles.for_y, 1);
    t = atan2(sum(sin(G.Particles.for_t)),sum(cos(G.Particles.for_t)));

    all_mean = [x(1) y(1) t(1) x(2) y(2) t(2)];
end

function all_std = get_std_state()
global G;
    x_s = std(G.Particles.for_x, 1);
    y_s = std(G.Particles.for_y, 1);
    t_s = std(G.Particles.for_t, 1);

    all_std = [x_s(1) y_s(1) t_s(1) x_s(2) y_s(2) t_s(2)];
end

function [a1]=awrap(a)
% function [a1]=awrap(a);
% Wrap angles (radians) to -pi,pi
% Argument a must be a vector
```

```
a1=a;
tran=0;
[n,d] = size(a1);
if and(n==1,d>1)
  tran=1;
  a1=a1';
end
ia=find(~isnan(a1));
a1(ia)=atan2(sin(a1(ia)),cos(a1(ia)));
if tran==1
  a1=a1';
end
end
```

8.4