# Real-World Robot Localization & Kidnapping

## Team Members

Taylor Apgar - tapgar3@gatech.edu

Sven Koppany - skoppany3@gatech.edu

Eric Leschinski - eleschinski3@gatech.edu

Gautam Salhotra - gautam.salhotra@gatech.edu

## Introduction

In line with our scheduled deliverables from the proposal, we have successfully implemented the Monte-Carlo Localisation (particle filter) on two robots. Furthermore, we have also implemented additional features, namely

1. Adaptive Monte-Carlo Localisation (AMCL)
2. Path Planning using Markov Decision Processes and A*
3. Kidnapped robot problem: Recognizing whether the robot has been kidnapped
4. Moving the robot under high uncertainty, using a local occupancy grid and sensor readings until uncertainty reduces.

Summary Video

## Table of Contents

## Custom Robot Theory of Operation

In order for the robot to go from a completely unknown position to the goal state in a highly symmetric environment it performs a multi-step localization process. Initially the particles are spread uniformly throughout the entire map. Since the robot has no idea where it is it decides which direction to drive in based on a local occupancy grid map. As it is doing this it updates the KLD based AMCL algorithm. In order to get a measure of the KL distance we insert the particles into a histogram during the resampling step. We also use this histogram for a hierarchical clustering algorithm that determines how many unique modes the robot is tracking. Once the clustering algorithm finds a bimodal distribution we extract the mean position estimate of dominant particle cluster. At this point we switch to the bimodal MDP policy which is a velocity vector field that guides the robot to one of the unique features on the map. Then once the clustering algorithm determines that there is a single cluster we switch to the unimodal MDP policy which loads a velocity vector field to the goal state. A separate passive algorithm is also estimating the probability that the robot has been kidnapped. If the robot determines it has been kidnapped it uses its local measurement history and a genetic algorithm to search the map for the most probable locations.

## Hardware

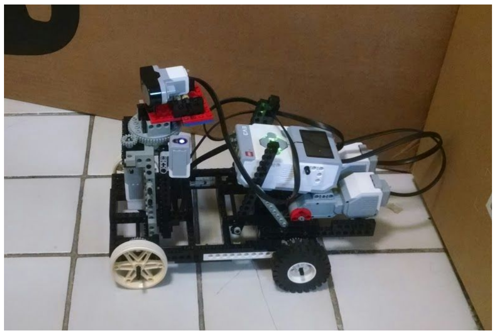
We implemented localisation on two robots in two different environments. The LEGO EV3 localised in a room environment, whereas our custom robot localised in a nearly-symmetric maze. Note that for the maze we had to switch over from the GoPiGo to the custom robot in our proposal because of the GoPiGo's hardware limitations.

### LEGO Mindstorms EV3

#### Robot

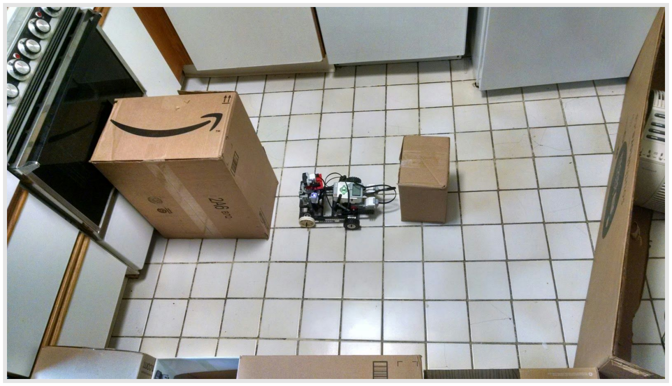
The skid steer style robot is constructed of legos from the EV3 mindstorms kit. The EV3 brick, runs on Debian Jessie Linux and has 4 ports each for actuators and sensors. There are two motors controlling movement for the right and left sets of wheels allowing the robot to move forward, backward, or rotate in place. The mounted sensor is an ultrasonic proximity sensor that is mounted on a rotating turret so the sensor can take samples without rotating the entire vehicle.

The programs that implement Probabilistic Robotics and principles are written in Python 2.7.

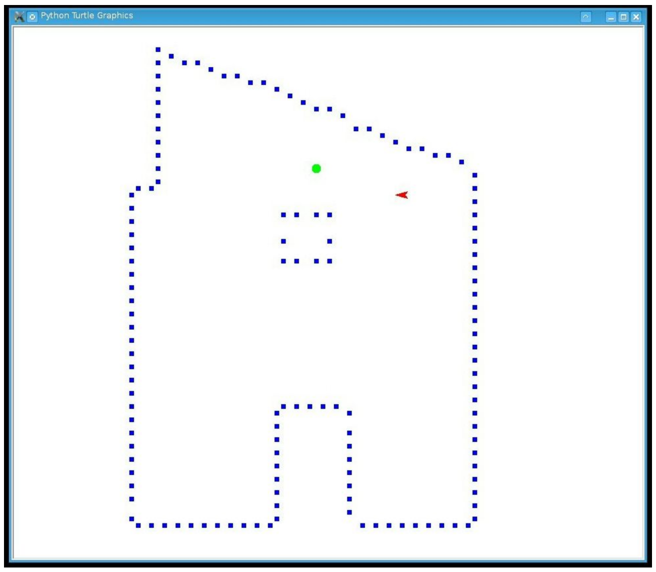


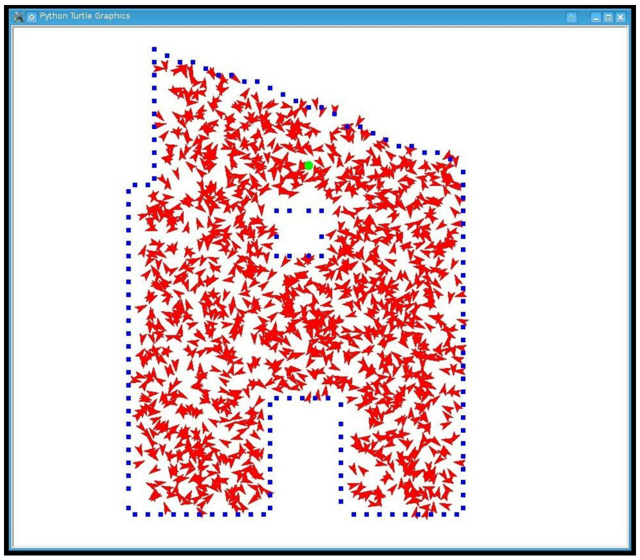
## Environment

The world for the Mindstorms EV3 robot has a width/height of 74" by 61". This photograph corresponds to the 2d representation of the world represented in the python file kitchen_map.py. The distances between objects are estimated to a tolerance of plus or minus one inch of variation.

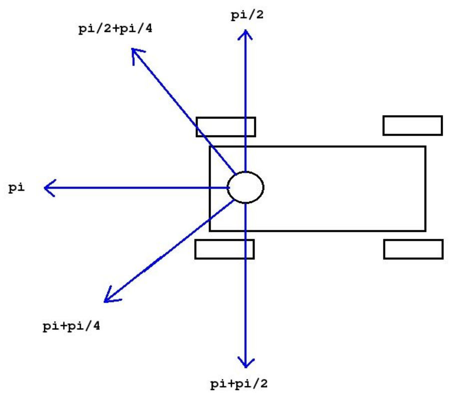


Using Python code and python turtle graphics, we construct a live 2-dimensional map of the world. The red arrow represents a robot particle's location and orientation. The green circle represents a desired target location. Each blue square dot represents a wall landmark that the sensor could read.





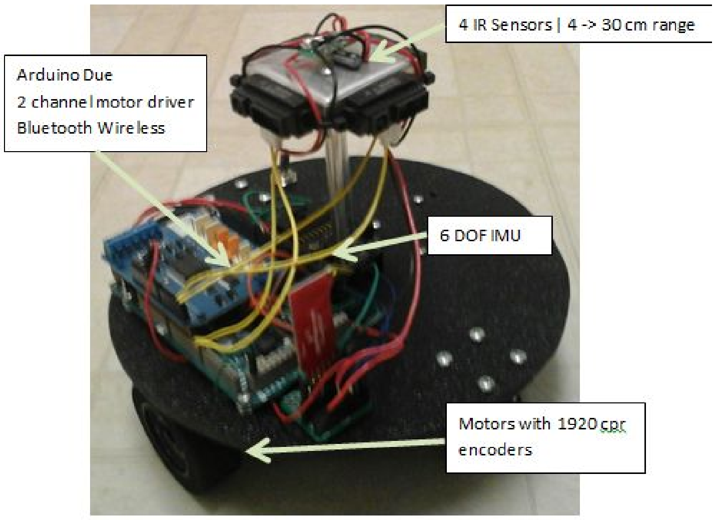
The EV3 robot takes 5 Sensor readings with the turret. The sensor is a distance sensor and takes readings in 5 different directions as illustrated below. Each direction returns a numerical value representing how far away the nearest hard object is.



## Custom robot

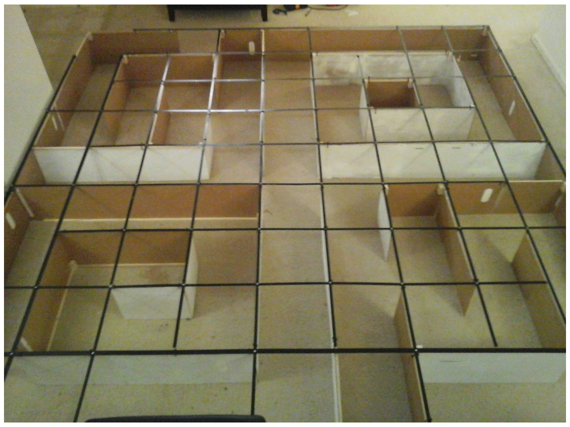
### Robot

We built a custom robot and a symmetric maze (9’ x 9’) for a more challenging localisation problem. The robot can be seen in the image below with all major components:


4 IR Sensors | 4 -> 30 cm range
Arduino Due
2 channel motor driver
Bluetooth Wireless
6 DOF IMU
Motors with 1920 cpr encoders

The robot connects wirelessly to a main C++ application. The main application receives the incremental time, encoder values, gyro data, and the 4 IR distance readings from the robot every 30 ms. The robot then receives left and right wheel rotational velocity targets from the main application. The diameter of the robot is 8”.

### Environment



The symmetric maze environment consists of two exits. However, only one exit is the goal. The only way the robot can determine how to get to the goal state is to first find the unique feature in the map. The unique feature is at the center of the spirals on the top right and bottom left corners of the map. The corridor width is 12” everywhere and it is a 9’x9’ environment.

## State Estimation

In order to estimate the position of the robot we used adaptive monte carlo localization. At its core MCL is an iterative two step process, elegantly represented by the two update equations below:

$$p(x_{k-1}, u_k) \rightleftharpoons p(x_k^+ | z_k,\ x_k^-)$$

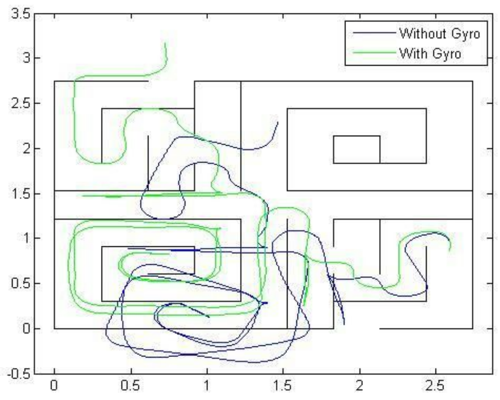
The left hand side of the equation is the motion update where we first update the probability distribution of the state estimate x given its previous distribution and control action u. The right hand side of the equation is the measurement update where we update the probability distribution of the posterior state estimate given the robots measurements z and its previous distribution.

In order to have a high performing MCL algorithm we had to understand the magnitude of our motion and measurement noise.

## Motion Model

### Error Sources

The robot is able to determine its incremental motion using a gyro for rotational velocity and its left and right encoders for incremental wheel position. The encoders are a direct measurement of the wheel position. You can translate wheel position to incremental robot movement as long as the wheels stay in contact with the ground. Wheel slip is the primary reason why position estimates solely based on odometry get less accurate over time. The gyro is an inertial measurement meaning it always measures rotational velocity in the robot frame. This means that the robot doesn't have to have contact with the floor in order for the gyro to get an accurate measurement. The main reason for adding the gyro was to mitigate the effect of wheel slip on the robot's rotation estimate. You can see the huge difference that the gyro makes on the position estimate over time in the plot below:



The main error sources for the gyro are noise and drift. With the MPU6000 the rotational velocity noise is incredibly small. The more significant error source is drift. This gyro does have a relatively small drift rate so the robot is able to run the maze without it becoming a major factor. And whenever the robot process is started the zero offset is recalibrated.

The major error sources in the odometry model are wheel slip and wheel misalignment. Since the robot is driving on carpet it is basically slipping constantly as the carpet fibers move under the robot wheel. Also since it is a custom built robot it was difficult getting the wheels exactly parallel.

### Motion Update

Since the robot is operating in the velocity frame and we get a good measure of linear and rotational velocity we decided to add Gaussian noise to the linear and rotational velocities of the particles in the a priori update step as seen below:

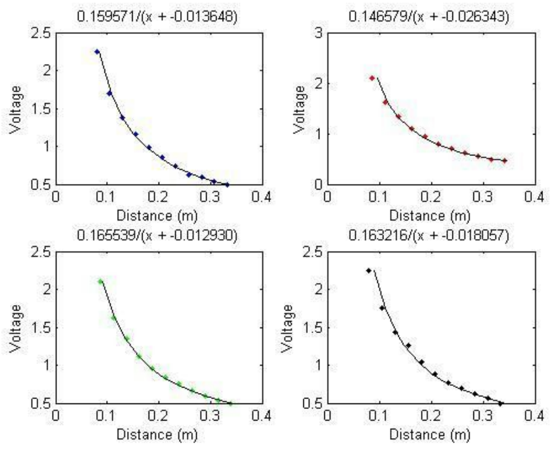
$$v_x = \frac{v_L + v_R}{2} + N(0, 0.015\ mps) \qquad \omega_z = \omega_{gyro} + N(0, 0.008\ rps)$$

$$\varphi_k^- \mathrel{+}= \omega_z * dt \quad X_k^- \mathrel{+}= v_x * \cos(\varphi_k^-) * dt \qquad Y_k^- \mathrel{+}= v_x * \sin(\varphi_k^-) * dt$$

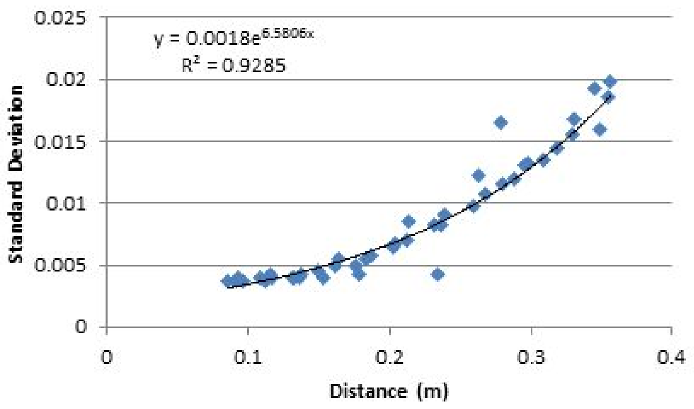
## Sensor Model

### Error Sources

The sensors used to measure the robot's position in relation to an external environment were 4 IR range sensors with a measurement range of 4 cm to 30 cm. These are analog sensors meaning they output a voltage as a function of distance. Our first objective was to understand what this function looked like. For each of the 4 sensors we took measurements from 4 cm to 30 cm and recorded the average voltage output over 10 seconds. Below is a graph of the 4 calibration curves and the best fit curve of the form $V = a/(distance + b)$.

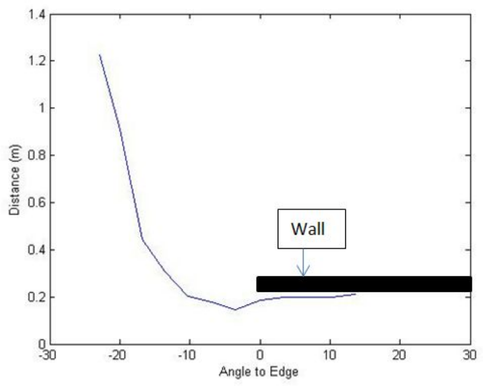


As can be indicated by the calibration curve small deviations in voltage is amplified as the distance grows. So what would be expected is higher deviation as a function of distance. Next, we captured the sensor uncertainty as a function of distance.



The sensor model so far consists of Gaussian noise that increases slightly as a function of distance. Also given a flat surface the sensor had fairly small noise characteristics.

However, after a couple of runs in the actual environment we noticed some odd noise characteristics when the sensor is detecting an edge or a corner. When the sensor was pointed into a corner the beam would never return and the sensor output would be max measurement. When the sensor was passing over an edge the output wouldn't go from the wall distance to max measurement. Instead the sensor output had the characteristics below:



So as the sensor moves past the edge the distance measured is first biased towards the robot then as the robot moves further away it gradually drops off.

## Measurement Update

A popular approach for updating the position estimate using range sensors is to use ray casting. The idea is that for each sensor you propagate the beam out from the robot to the distance measured and find the closest occupied location on the map. You then associate that occupied location with the measurement and use that distance to find the probability of measurement given robot pose. Since we knew our AMCL algorithm would need a very large number of initial particles we tried to do a less computationally expensive measurement update approach called likelihood fields. We pre computed a grid map where each grid cell contained the distance to the nearest occupied grid. Then during the measurement update step all we have to do is compute the global location of the sensor measurement and lookup the grid cell to find the distance used to update our probability. The measurement update has the following form:

$$p(x_k^-) = z_{max} * e^{\frac{-gridDist^2}{2\sigma^2}}$$

Where $z_{max}$ was set to 0.52 and $z_{min}$ was set to 0.48. We could have used higher values but the abnormal effects of corners and edges play a large role in our sensor model.

## Particle Filter

The particle filter algorithm is similar to the algorithm implemented in class, with a few modifications. This video shows the robot going from start to finish with particle filtering, and the visualisation video.

Here is a video of the Mindstorms EV3 robot localizing successfully.  The Mindstorms EV3 robot first calibrates the direction of the turret by rotating the sensor until a known point is reached.  A linux desktop performs the particle filter calculation and Robot Particles are uniformly distributed across the navigable areas of the map.  As the sensors collect data about the distances to the nearest objects, the particles that most fit the sensor data are given higher weights than the particles that do not fit.  On each iteration, the robot gets more and more sure of where it is.  The robot avoids hitting walls before it is localized by gathering information about the objects in its vicinity and travels in a direction without obstacles.

### Initialising Particles

For the EV3 in the room-style map, particles were initialised randomly throughout the free space in the map. In the case of the symmetrical maze, a lot more particles were needed (~100000) and it slowed down the algorithm. Hence, particles were inserted only where the center of the robot could actually be. For example, particles can’t be 1 cm away from the wall, as the custom robot has a radius of about 10 cm. This helped reduce the number of particles required to ~40000.

### Resampling

Stratified resampling was used to promote high variance for multiple possible estimates for the robot location, as is the case in symmetric mazes.

In the symmetric maze, sometimes (potentially) good particles would not be resampled purely due to the probabilistic nature of resampling. This is known as sample impoverishment[1]. Hence, we didn't resample at every step. A common technique is to resample when the weight distribution of the samples is skewed. We resampled when our metric (Effective Sample Size, ESS) was below a threshold.

[1] Particles with large weights are likely to be drawn multiple times during resampling, whereas particles with small weights are not likely to be drawn at all Hence, the diversity of the particles will tend to decrease after resampling. In the extreme case, all particles might “collapse” into a single particle. Source

## Sensor uncertainty

From the sensor model, we got the likelihood of each sensor measurement using a likelihood field (see Sensor Models). However, an incorrect sensor reading would return a low likelihood for a good particle, and will return a high likelihood for a bad particle. Thus, the weight of the bad particle is incorrectly inflated and it has a higher chance of being picked during resampling. This was especially bad when the robot turned. Sometimes the distance sensor wouldn't detect the thin walls around which it is turning, or sometimes it wouldn't give the right readings for corners.

To solve this problem, we update the weights every 10 steps with the average likelihood from the past 10 measurements (with a few exceptions). We assumed each sensor reading to be independent of the other to simplify the likelihood calculation. In reality, if a robot is moving down a corridor, the left and right sensors would give readings relating to the corridor width.

## Motion noise

If a particle is off by a small distance from the true position of the robot, a higher motion noise can account for this offset. For debugging, we added extra motion noise to account for such bad placement of particles, and other unmodeled dynamics.

## Adaptive Monte Carlo Localisation (AMCL)

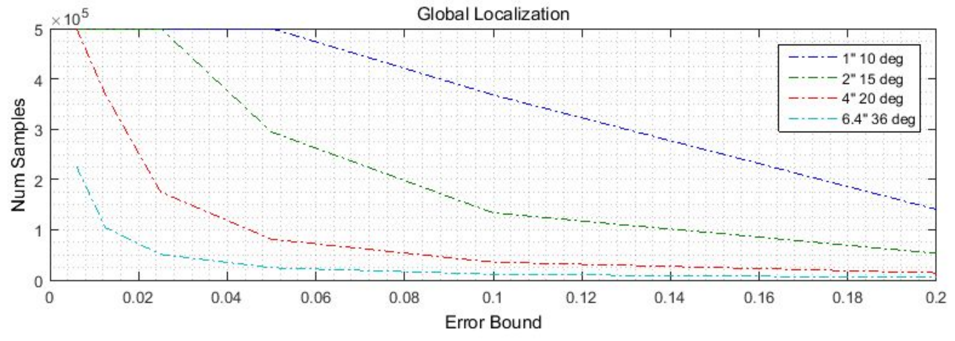
AMCL is a modification of particle filtering, where the number of particles changes every time a resampling occurs. The formula to compute the required number of particles is:

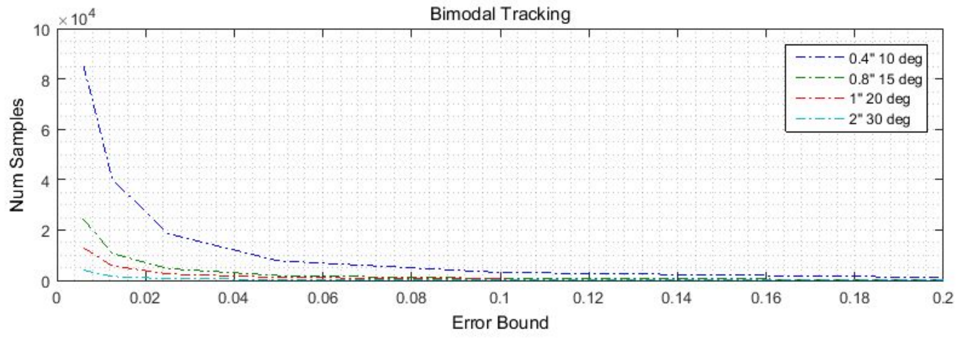
$$N_x = \frac{(k-1)}{2\varepsilon}\left\{1 - \frac{2}{9(k-1)} + z_{1-\delta}\sqrt{\frac{2}{9(k-1)}}\right\}^3$$

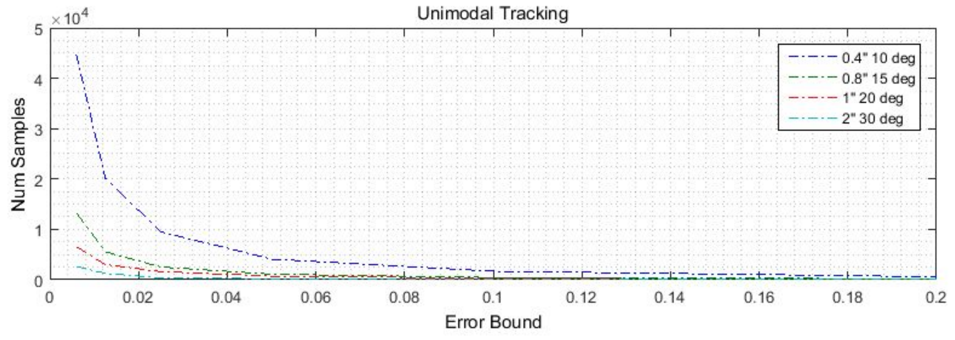
where
k = number of non-empty bins, after creating a histogram of particles
$z_{1-\boldsymbol{\delta}}$ = z-score statistical multiplier for a confidence interval $\boldsymbol{\delta}$

The three parameters we can tweak are $\boldsymbol{\varepsilon}$, the z-score $z_{1-\boldsymbol{\delta}}$, and the bin size (which indirectly affects the number of non-empty bins k). We kept a bin size of 1"x1"x45∘, as coarser bins would not give a good estimate of $N_x$ during tracking. Finer bins increase computational cost. We set a standard $\boldsymbol{\delta}$ of 99%, so $z_{1-\boldsymbol{\delta}}$ = 2.57 from statistical tables. We varied $\boldsymbol{\varepsilon}$ based on how quickly particles converged to a bimodal distribution in the maze, and how many particles were needed to track this successfully. The variation of samples required to localize v/s these three parameters is shown below.

This video shows a full run of AMCL in the symmetric maze, which is very similar to the particle filter video. However, notice the value for NParticles changes every time it resamples. As the robot moves around, unlikely particles keep getting dropped every time resampling occurs. This reduces uncertainty and required number of particles making it computationally cheaper. This makes AMCL a superior algorithm compared to Monte Carlo Localisation (particle filters).







## Clustering

Cluster analysis is used to find clusters of particles. Our implementation of clustering also helps in detecting the number of non-empty bins, k, for the AMCL algorithm. The logic behind clustering is to divide the state-space of the particles in bins. Put particles in their respective bin as per the state(x,y,yaw). Neighbouring bins that aren't empty are grouped together in a cluster. Thus, each particle indirectly belongs to a cluster. The number of clusters is an indicator of uncertainty in the distribution of particles. When there are just two clusters, that means there are only two likely areas in the map where the robot can be - this is a bimodal distribution. This was seen in the maze map due to its symmetrical nature. Once the robot has moved enough, the clustering algorithm should give you just one cluster - this is a unimodal distribution, and the robot has been accurately localised. This can be seen in the video with AMCL and clustering.

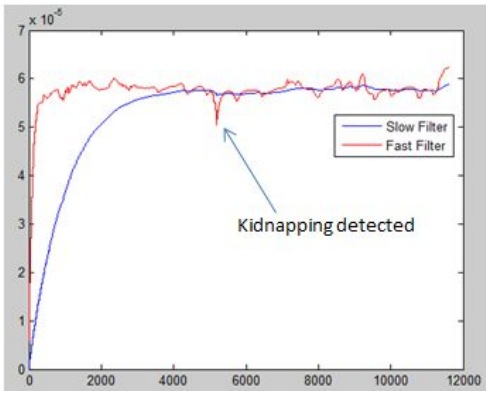
Once the robot is localised, it can implement path planning to go to the target, or even be kidnapped.

## Kidnapped Robot Problem

The kidnapped robot problem is have a robot that is already localized moved to a completely new position. The objective of the robot is two fold. First it must quickly determine that it has been kidnapped and then it must quickly find its new position.

### Kidnap Detection

The algorithm we used for detecting whether the robot has been kidnapped uses two exponential filters for the average particle weight before normalization. One filter has a much larger decay rate than the other this we call the fast filter and the other is the slow filter. These two filters can be seen below:



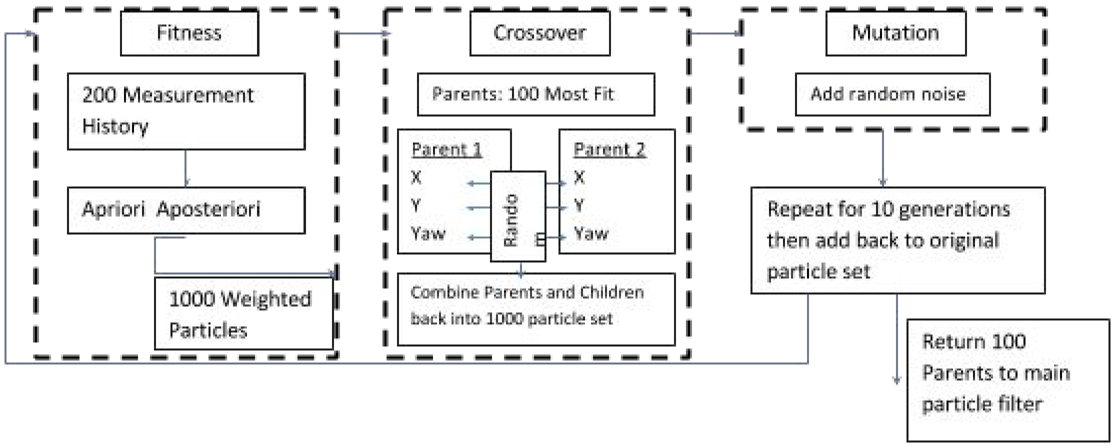
The robot then uses the following condition to determine if it has been kidnapped: $1 - \frac{w_{fast}}{w_{slow}} > 0.9$.

### Searching New Position Estimate

Through our testing we found that it would take at least 200 measurements (30 ms sampling) before we are able to detect the robot being kidnapped. So as the robot is running in normal operation we keep a history of the most recent 200 measurements. Once the kidnap detection is triggered we initialize a genetic algorithm to search for 100 new particle locations.

### Genetic Algorithm (GA) Search

We use a population of 1000 individuals and set their initial position to random positions on the map. For each iteration of the GA we first find the fitness of each of the individuals by performing the normal a priori and posteriori particle filter updates without resampling over the 200 measurements. After we determine fitness we take the 100 individuals with largest particle weights as the parents for the next generation. The 100 parents survive to the next generation so we use crossover to generate the other 900 children. The crossover algorithm creates two children from the two parents where a random element of the state (X, Y, Yaw) is swapped between the two parents to create the two new children. The mutation step then adds a small amount of random noise to each of the elements in the state vector. The children are then recombined with the parents into the initial particle set and the GA repeats for 10 iterations. The 100 parents are then added back to the original particle filter; replacing its 100 lowest weighted particles. The diagram below shows the whole process:


Fitness
200 Measurement History
Apriori Aposteriori
1000 Weighted Particles
Crossover
Parents: 100 Most Fit
Parent 1
X
Y
Yaw
Rando
Parent 2
X
Y
Yaw
Combine Parents and Children back into 1000 particle set
Mutation
Add random noise
Repeat for 10 generations then add back to original particle set
Return 100 Parents to main particle filter

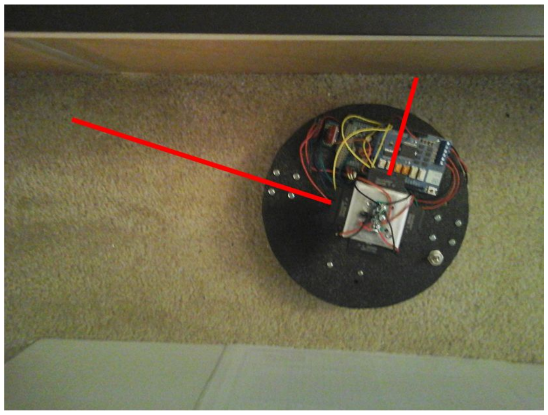
Refer to video showing the kidnapping detection and GA search.

The reason why GA is a good algorithm to use is because it allows the robot to search over the individual elements. Meaning if one particle has a good X estimate it can share that X estimate with another particle that may have a better Y estimate. The main drawback of using this algorithm is that it is a greedy search and 1000 particles isn't enough to cover the entire search space. For the symmetric maze this causes the robot to find only 1 of the 2 possible positions.

# Control

## Path Planning with Unknown Position

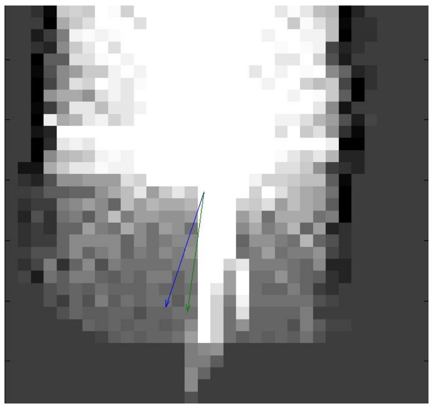
There are many algorithms that can be used to choose an optimal action in a completely unknown environment. However, we found that using immediate sensor values led to very poor action selection. Even simple algorithms like wall following simply do not work for our robot because of how the sensors are positioned on the robot. This is because the robot would get the same sensor values if it were flipped 180 degrees in the other direction so you can't determine which way you should rotate. Also the robot can rotate about a point without the sensor values changing.

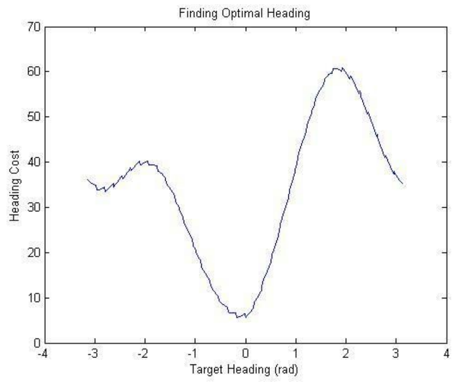


So after some time it became apparent that in order to choose our actions efficiently we needed to estimate the robot's position in relation to its immediate environment.

### Vector Field Histogram

The approach we took was to create a local occupancy grid map in the robot's local coordinate frame. We essentially treated the robots incremental movements as the true position of the robot and updated a 16”x16” occupancy grid map around the robot. So if the robot moves greater than 1 grid all of the grids are shifted and a new row of unexplored grid cells are added.





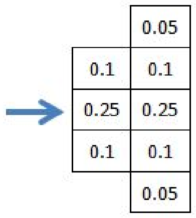
The plot on the left is the local occupancy grid map that the robot and the plot on the right is the target heading cost. The darker the grids the more likely they are occupied and the lighter the grid the more likely it is free. The dark gray is unexplored and has a 50% probability of occupied. The robot calculates the target heading by combining a couple heuristics. First the robot finds the closest obstacle for each heading and assigns the cost for that heading as the probability occupied divided by the distance to the object. The robot then combines that heuristic with a heuristic that gives preference to headings closest to its current heading. It then uses a Gaussian filter to smooth out the cost curve and give a single minimum cost heading direction. Here is a video of the robot performing vector field histogram and following the target velocity vector.

## Path Planning with Known Position

### Markov Decision Process

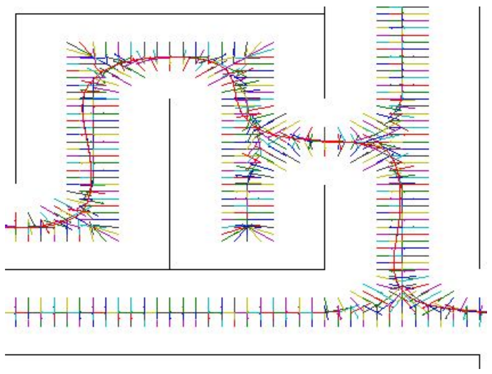
The approach taken for path planning when the robot's position is known was to use Markov Decision Processes to create a global velocity vector profile that would guide the robot from any point on the map to either the unique feature or the goal state.

The MDP used 4924 states which represented the x and y positions that the robot could drive in and it used 4 actions to represent the 4 cardinal directions the robot could travel. Ideally we would have liked to add more actions and to encompass the yaw in the state but this was already near the maximum size that matlab could handle. A positive reward was assigned to the goal state which was the top left opening in the unimodal policy and two goal states at the unique features was used for the bimodal policy. A negative reward was given to the state which represented all positions where part of the robot would be touching occupied space. A very simple transition model was used which can be seen below:



Meaning whichever direction the robot chooses it will transition to the next state with probabilities stated above. After running policy iteration we get a Q value for each state action pair. We then combine the vectors according to value to get a velocity vector. Since the states are still pretty coarse we run an additional

Gaussian smoothing step which interpolates Q values for 10x higher resolution. So when the robot has a position estimate it is able to look up the velocity vector in whichever 0.1” grid cell it is closest to. Below is the coarse Q vectors plotted on the map along with a sample path that the robot would take following this policy. Here is a video of the robot following this MDP policy at max speed while updating the particle filter.



### A* for Mindstorms EV3 Robot

The mindstorms EV3 robot performs an A* search after the robot has localized. Navigable points on the map are defined manually in kitchen_map.py. The method `hasRobotLocalized()` returns true when the position of the robot is known. When the position is known, the A* search takes the world map defining the navigable areas and finds the shortest path between the robot and the target. The current angle and position of the robot is determined with the methods `getCenterMassOfParticles()` and `getAverageOrientation()`.

The Mindstorms EV3 brick knows the current direction via the average particle direction. The robot can be pointed in the direction along the path toward the target. The robot moves forward a small distance. The position of the robot will be re estimated after the robot is re-localized, and the A* search will repeat until the target is reached.

The A* algorithm chooses the shortest distance to the target, this causes the problem of the robot bumping into walls since the robot is wide. This problem was solved by enforcing that navigable points are only ones at least 3 inches away from a non navigable point. The problem of the current robot location being in a non-navigable point was to take the straightest route to the nearest navigable point.

## Acknowledgements

1) FSMA - Fixed Size Multidimensional Arrays Library (in C++): https://code.google.com/p/fsma/

2) Library interfacing the mindstorms sensors and motors with the EV3 robot https://github.com/topikachu/python-ev3

3) Team GitHub repo: https://github.gatech.edu/OrderOfChaos/CS8803