

University of Cincinnati

College of Engineering and Applied Science

Implementing Monte Carlo Localization with the Lego Mindstorm

Senior Design Project  
Winter-Spring 2011

Seniors

Andrew Larger  
Zihao Lu  
Trevor Plassman  
Matt Schmucki  
Alicia Zwiebel

Advisor

Professor Dan Humpert

# Executive Summary

[insert text]

**Table of Contents**

Executive Summary 2

**1.** **Introduction** 4

**2.** **Background** 4

**3.** **Development** 4

3.1 Constructing the Course 4

3.2 Constructing the Robot 6

3.3 Programming the Robot 7

3.4 A\* Navigation 9

3.5 Programming Monte Carlo Localization 11

3.5.1 Theory 11

3.5.2 Programming MCL 15

3.6 Difficulties 20

3.7 Testing 20

3.8 Future Work 20

**4.** **Conclusions** 21

**5.** **References** 22

**6.** **Appendix** 23

1. **Introduction**

The objective of the project is to program the Lego Mindstorm to use Monte Carlo Localization (MCL) to allow the robot to determine where it is within a given environment, with known boundaries and landmarks. Once the robot has localized itself within the environment, it will be able to autonomously determine the most efficient and quick path to navigate to the end destination, avoiding the boundaries and landmarks.

1. **Background**

The capabilities of the Lego Mindstorm will be explored for programming a robot to autonomously navigate through the given environment. The robot is able to run a version of Java instead of the provided Lego Mindstorm software, allowing more versatile programming of the MCL algorithms.

The project focuses on using only the Lego Mindstorm, necessary Lego Mindstorm sensors, and the MCL algorithms. The MCL algorithms are developed specifically for sensors being used in this project. The project will stick to using sensors available specifically for the Lego Mindstorm, which ensures a cost effective development. Third party sensors can be cost prohibitive and cause the project to focus on specifying the optimal sensors as opposed to optimizing the programming and imploring creative ways to use the sensors. Focusing on less important aspects of the robot can lead to scope creep which the group aims to eliminate while staying focused on the primary objective and means of obtaining the goals in place for the project.

1. **Development**
   1. Constructing the Course

The robot needed to exist in a world of known exterior boundaries and variable interior boundaries. To achieve this, a large piece of plywood was offered and used as the base of this course. Interior dimensions were taken that would allow for maximum possible area while having perfect 90 degree corners. The dimensions allowed for the total length of the course to be 140 cm and 63 cm tall with a notch cut in one of the long ends 20 cm deep, 20 cm long and centered along the length of the plywood. Once the dimensions were marked off on the plywood base, a band saw was used to cut the piece to the exact dimensions.

The robot needed to be able to get the distances to these edges to accurately perform its task. To accomplish this, walls were measured out, cut with the band saw and screwed in place on these edges standing roughly 10 cm above the base level.

During testing of the robot, it was noticed that the marble on the robot used for turning was picking up quite a bit of resistance from the plywood due to its uneven surface. To smooth this out without creating an entirely new base piece, three pieces of poster board were cut up, laid on the course and then taped together.

With the exterior wall dimensions set, several smaller movable walls were put together to allow for some degree of variability in the course layout. These walls were made at 10, 15, and 20 cm long and stand slightly higher than the other walls due to the bases nailed on the bottom of them which add stability. These bases for the walls do not add additional length to the walls but do add approximately ½ cm of width on either side.

To complete the course, it was decided that it should be made easier to determine the actual location of the robot visually to compare with the results on screen. The poster board pieces were marked up with a square pattern of 5 cm by 5 cm to more accurately allow the group to determine visually the robot position on the map as well as determine the distances the robot should be reading and displaying. Along the interior walls, labels were then put in place showing the total length of each wall.



Figure 3.1.1: Completed Course



Figure 3.1.2: Mobile Walls

* 1. Constructing the Robot

The robot was designed and built using the Lego Mindstorm and compatible pieces. The major components are the control brick, two motors to control the two wheels, the compass sensor to get direction and allow for accurate 90 degree turns and the distance sensor to determine the distance to the nearest wall approximately straight out from the robot. A non-Lego component was used to allow the robot to turn with the least resistance possible. This piece is a marble held in place by an inverted wheel well. The marble will rotate in place and is located on the back end to balance the robot on three points, including the two wheels.

To design the robot, we first had to account for the two motors we had which would drive the robot, the control brick which would have to be situated somewhere above the motors, and the distance sensor. Our group decided to put the distance sensor beneath the control brick so as to not put it too high, which would have required taller walls as well as creating more potential variance in the readings if the sensor was angled too far down. Many different methods were attempted to create a rear wheel or wheels which would both balance the robot as well as allow for smooth turning in place. After several weeks, it was decided that a small marble could be held in place by one of the available wheel wells and that this would suffice for both turning smoothly and balance of the robot.

The compass sensor was acquired much later to better facilitate accurate turns of 90 degrees. However, it was discovered that the motors actually created some interference with the compass readings and the solution was to have to the sensor located at least 4 to 6 inches away from the motors. From the control brick, support beams were built up to a height sufficient to alleviate these interference issues.



Figure 3.2.1: Completed Robot



Figure 3.2.2: Marble Wheel Base Design

* 1. Programming the Robot

The Lego Mindstorm came with custom preinstalled software. This software did not meet our needs. It used a block GUI to program the brick and was not robust enough for Monte Carlo Localization. After some research, a Java operating system, Lejos, was found that could be installed on the brick. This system erased what the brick previously had and installed a version of Java. A regular Java programming suite could now be used to program the brick. It is important to note that no one in this group is a computer science major. The group is made up of mechanical engineers and learning how to program at this level was its own challenge.

The first step was connecting to the brick using a Bluetooth connection. The only options were a Bluetooth connection or a USB connection. Using USB would mean the brick would always need to be connected to the laptop. This was a limitation that was not acceptable. It took trial and error to try and open the ports needed to send integers to and from the brick. The key was to get the computer and brick to do the opposite of each other. If one was sending data, the other had to be listening for data. The table below shows what calls the computer would make that what calls the brick would make it response. The segment below would ask the brick to use its sonar and record the distance it reads. The brick then returns this value to the laptop and the laptop then prints it out in the form of ‘temp.’ By passing numbers back and forth, the two computers can be regulated and not override each other.

|  |  |
| --- | --- |
| Laptop | Brick |
| *conn* = **new** NXTConnector();  *dos* = *conn*.getDataOut();  *dis* = *conn*.getDataIn();  *dos*.writeInt(5);  *dos*.flush();  **int** temp = *dis*.readInt();  System.*out*.println(temp); | *btc* = Bluetooth.*waitForConnection*(); *dis* = *btc*.openDataInputStream();  *dos* = *btc*.openDataOutputStream();  n = *dis*.readInt();  **int** temp = *uss*.getDistance();  *dos*.writeInt(temp);  *dos*.flush(); |

After the laptop was communicating with the brick, the laptop had to create a grid that represented the environment. While the computers may not need this display, the humans running the program needed it so they could understand what the computers were doing. The grid would be identical to the physical course that was constructed. On the screen, 5 pixels is the equivalent of 1 centimeter. In the screen shot below, each small square represents 1 centimeter. The larger squares are 5 centimeters on each side and match the physical course. Any red square represents a wall. The user can easily click a cell to change it into a wall. This screen is interactive.

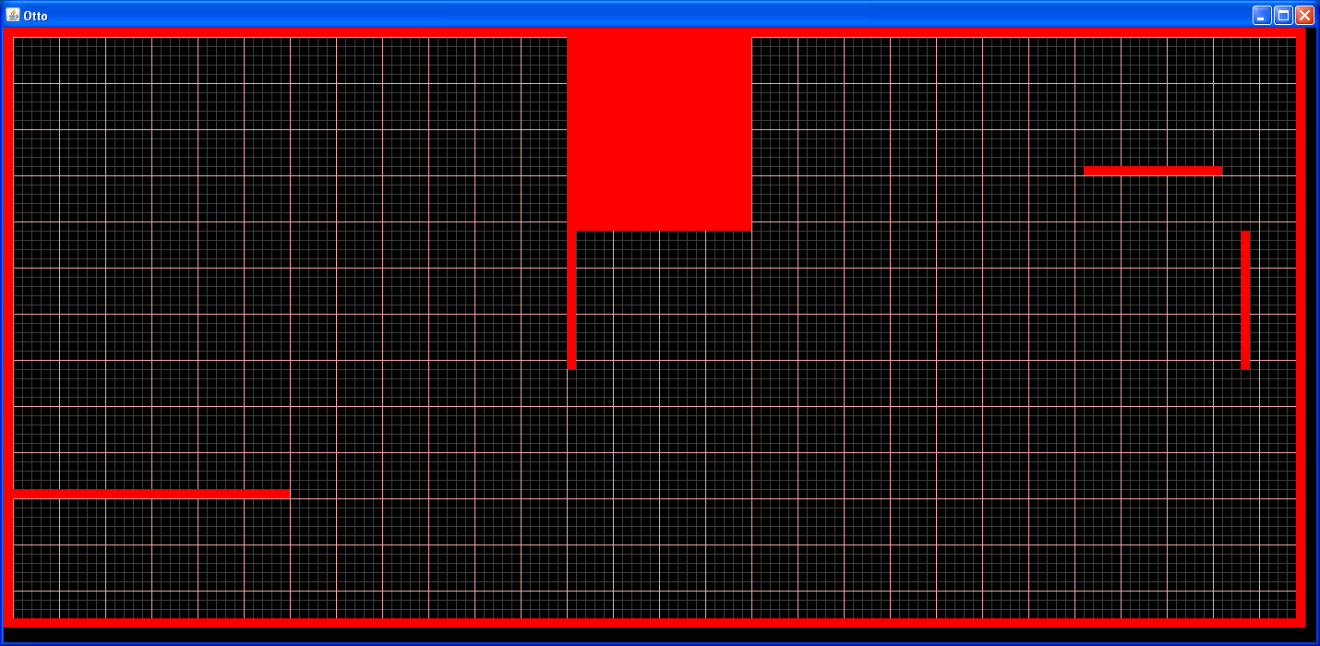


Figure 3.3.1: Screen Shot of User Interface

With the laptop now showing an interface and also being able to communicate with the robot, two major obstacles still existed. The first was localizing itself so it knew where it was. The second was getting it to travel the course once it knew where it was. These two obstacles were divided up among two people on the team.

* 1. A\* Navigation

A\* navigation was chosen to navigate the robot once it knew where it was. This algorithm was first used in 1968 but was based on a 1959 algorithm created by Edsger Dijkstra [1]. A\* uses heuristics to be efficient and effective. Heuristics is often defined as common sense or an educated guess. While it is easy for a human to use common sense, it can be much more difficult to program a robot to use common sense. Heuristics attempts to do this.

This process divides the environment into cells called nodes. Each node is then assigned three costs. The table below talks about each cost. Whichever node has the lowest cost is the best solution.

|  |  |  |
| --- | --- | --- |
| A\* Star Costs | | |
| Name | Symbol | Description |
| Exact Cost | G | Exact distance from the starting point |
| Heuristic Cost | H | Estimated distance from the end point |
| Total Cost | F | = G + H |

Gamegardens.com provided an excellent example of pseudo code describing how A\* is implemented. That portion of code is included below. From this outline, code written in Java and customized to our course was created and could solve how to get from one point to another. The code also took into account walls and how much room was needed for the robot to get by. This was added on our own.

create the open list of nodes, initially containing only our starting node

create the closed list of nodes, initially empty

while (we have not reached our goal) {

consider the best node in the open list (the node with the lowest f value)

if (this node is the goal) {

then we're done

}

else {

move the current node to the closed list and consider all of its neighbors

for (each neighbor) {

if (this neighbor is in the closed list and our current g value is lower) {

update the neighbor with the new, lower, g value

change the neighbor's parent to our current node

}

else if (this neighbor is in the open list and our current g value is lower) {

update the neighbor with the new, lower, g value

change the neighbor's parent to our current node

}

else this neighbor is not in either the open or closed list {

add the neighbor to the open list and set its g value

}

}

}

}

* 1. Programming Monte Carlo Localization
     1. Theory

To begin Monte Carlo Localization (MCL) the processing computer must generate a large number of theoretically random possible poses, consisting of location and heading, within a known boundary environment which represent the possible locations that the robot may occupy. These can be thought of as the computer’s random guesses at where the robot might be within the known boundary. In actuality only one of these poses represents the actual location of the robot but at the beginning each pose generated by the computer has an equal chance of being the correct location. In order to eliminate certain poses the robot must move to a new location, update its current pose guesses and then resample the poses in a manner which keeps only those poses which have the highest possibility of being the correct location. This possibility of being the correct location is known as the pose’s weight. This weight is found by monitoring the difference between a sonar reading from the robot to the nearest obstructions and the pose property that represents this reading for each pose. The closer the pose reading to the nearest obstruction is to the robots reading the higher the weight of that pose and the more likely it is the correct guess at the robots location. Repeating this resampling process over the course of several random moves applied to the robot and the poses will eventually converge all of the poses into a single location that, based on the pose weight, has the best chance of representing the location of the robot.

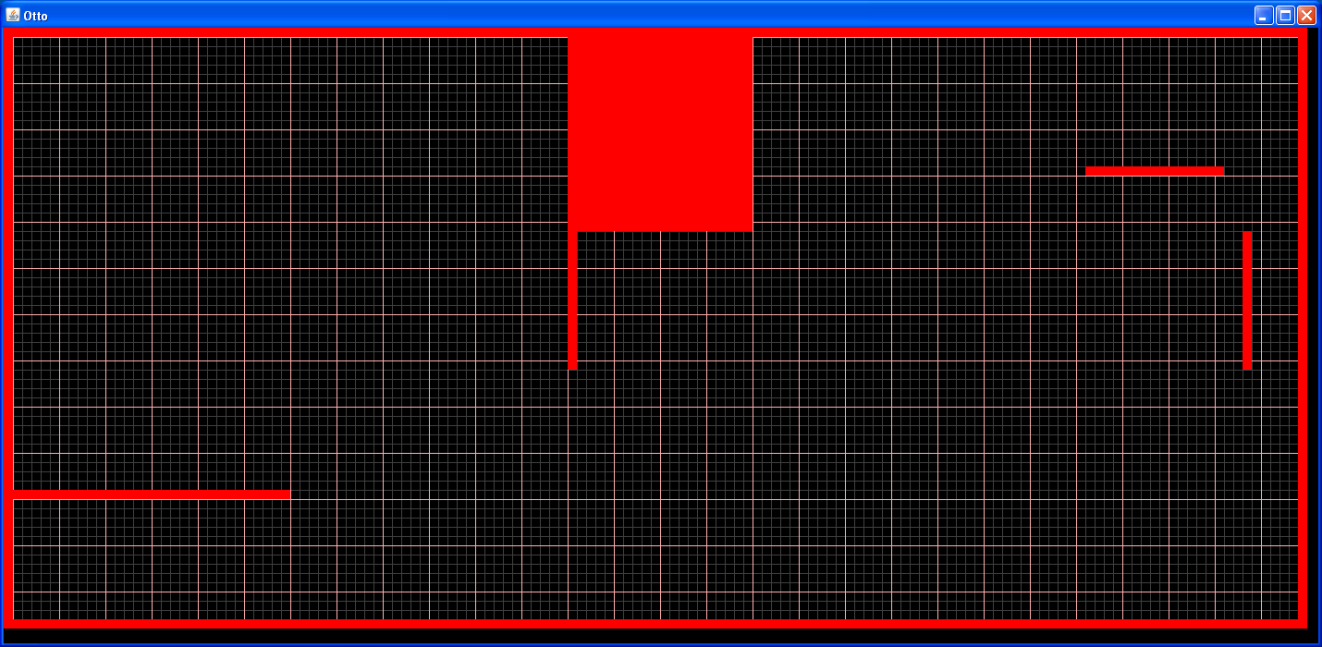


Figure 3.5.1.1: Initial Boundary

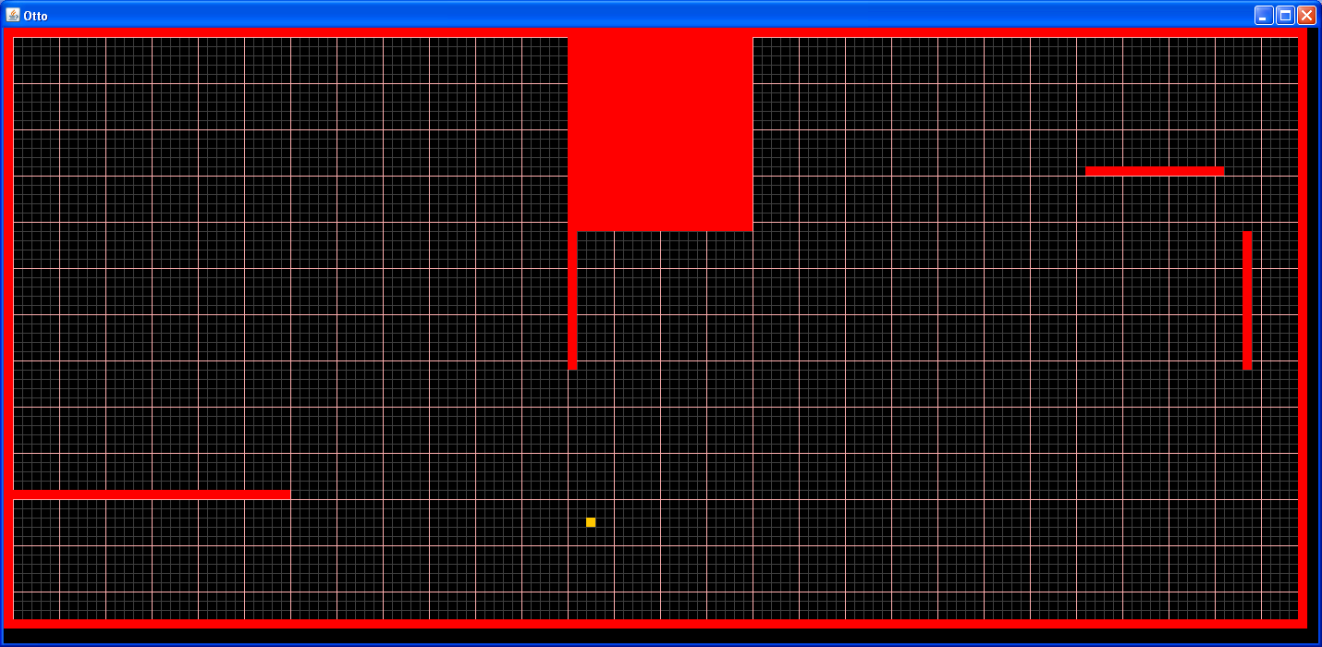


Figure 3.5.1.2: Initial Boundary with End Point Selected

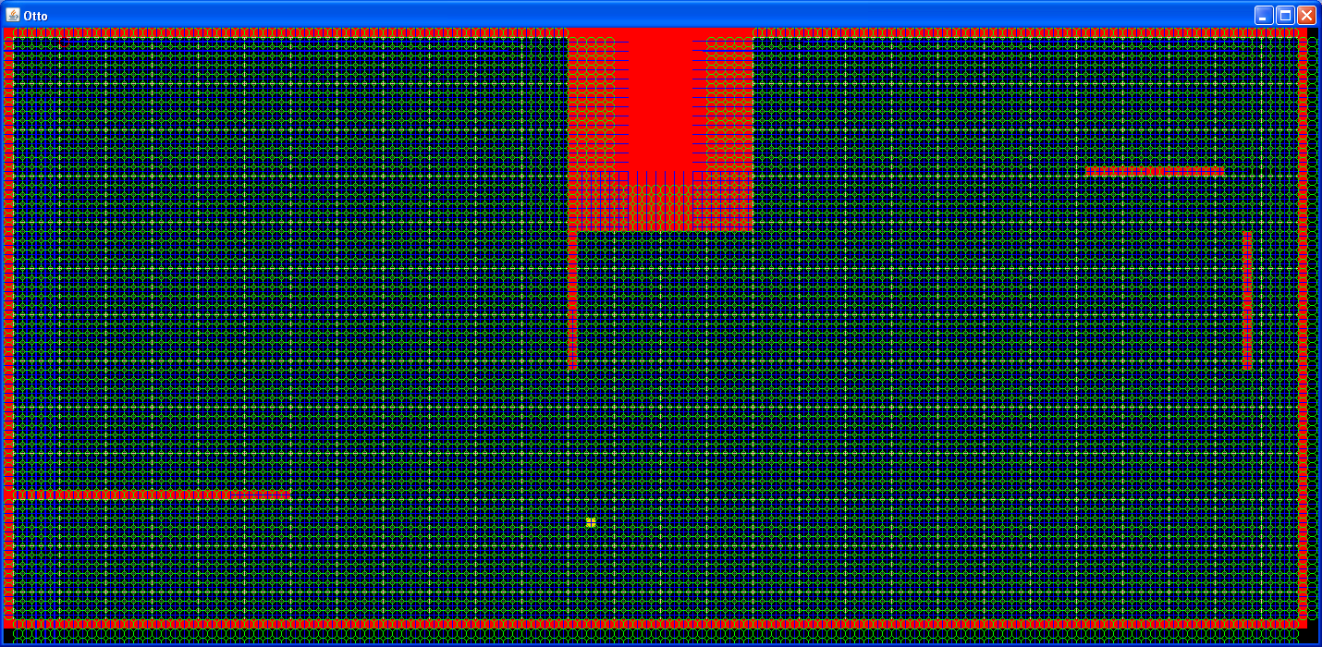


Figure 3.5.1.3: Initial Particle Population

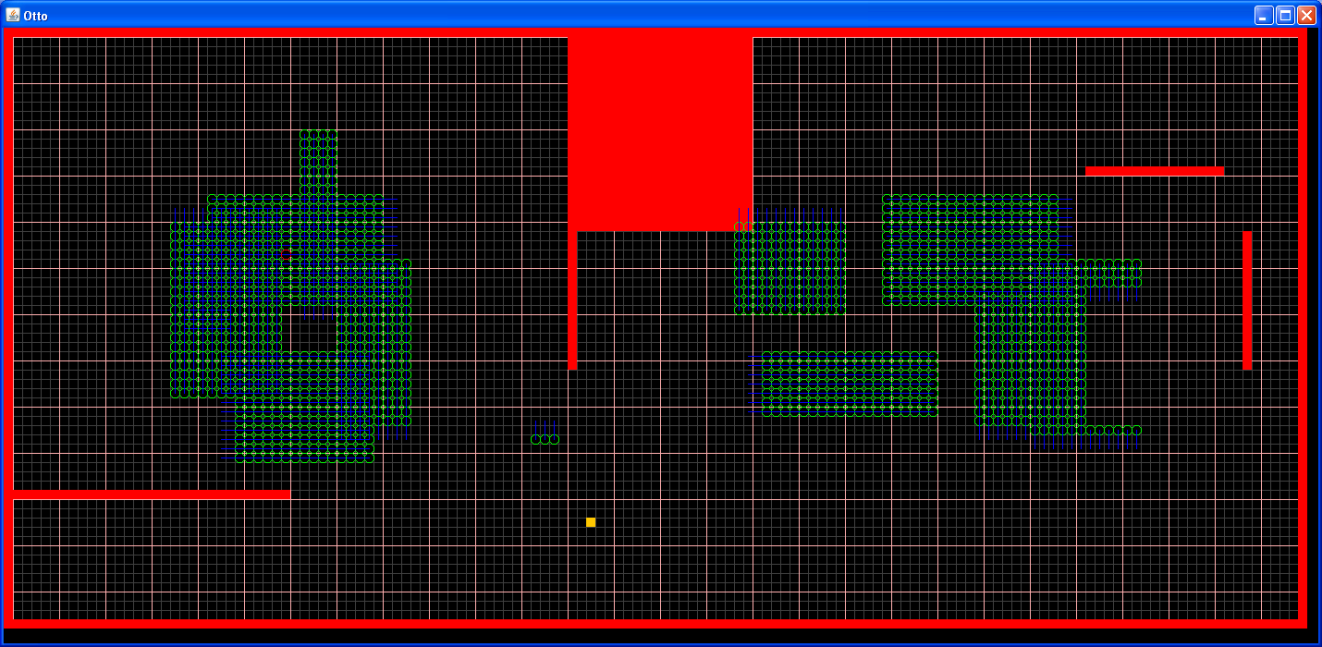


Figure 3.5.1.4: Converging Particles Resample 1

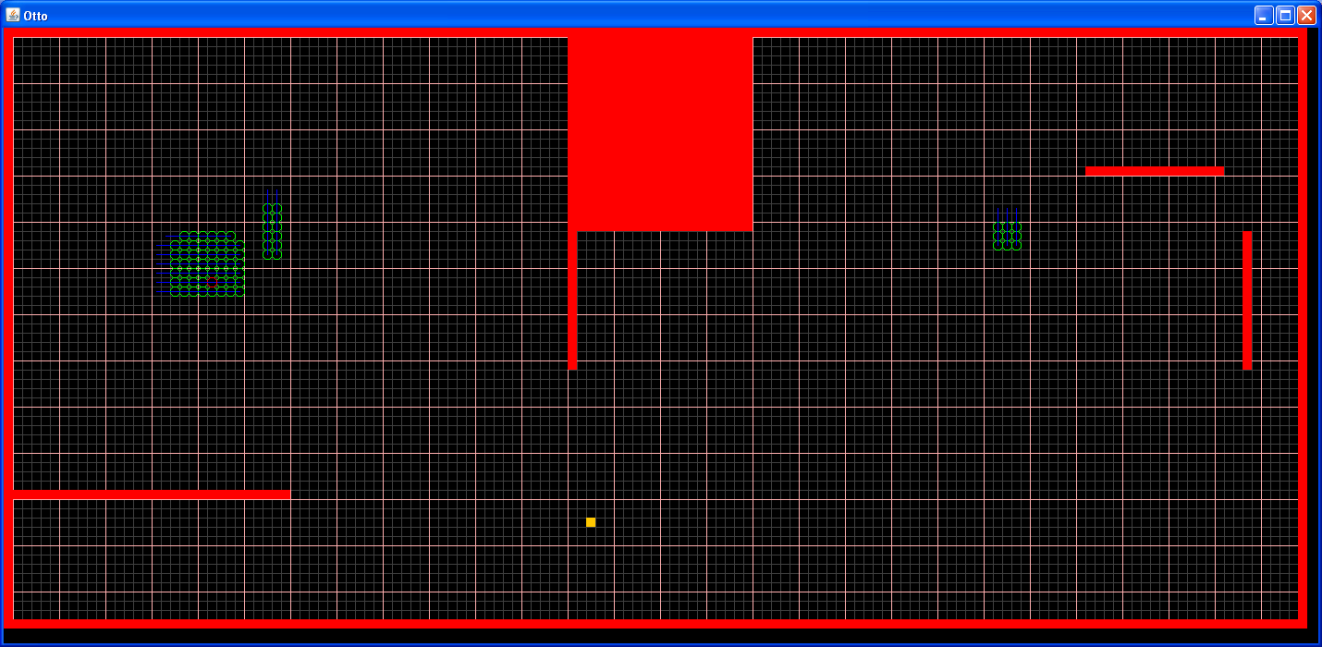


Figure 3.5.1.5: Converging Particles Resample 2

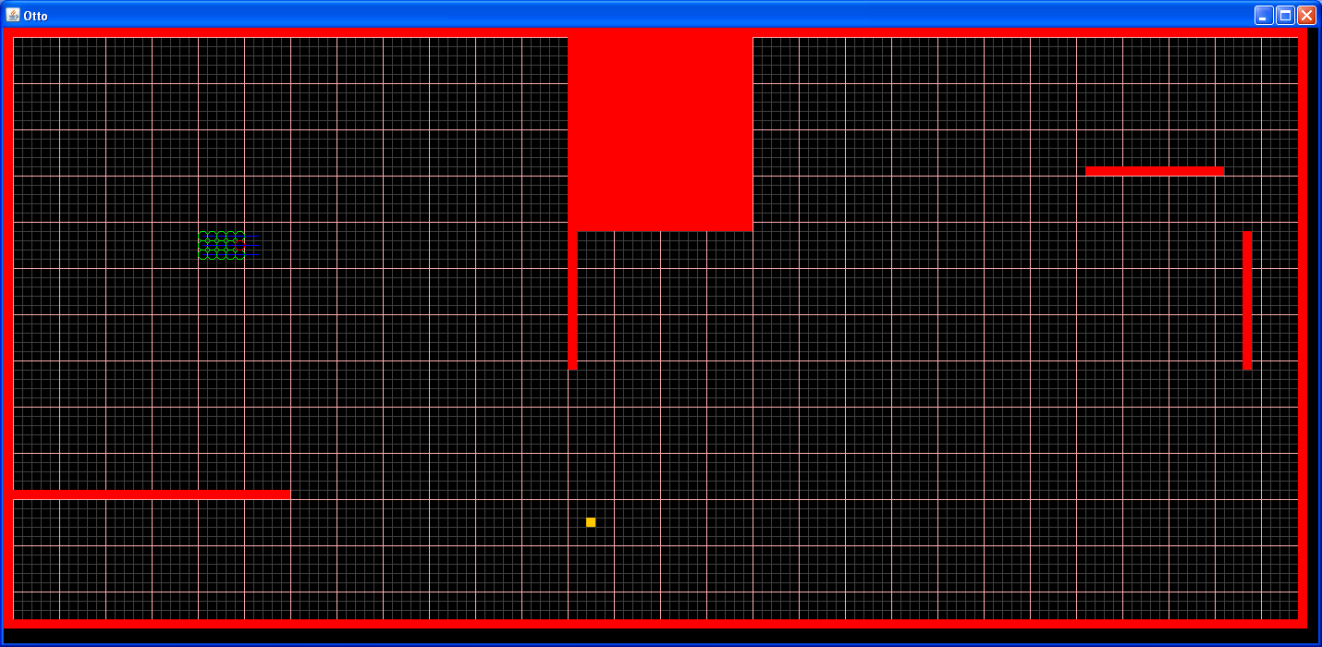


Figure 3.5.1.6: Converging Particles Resample 3

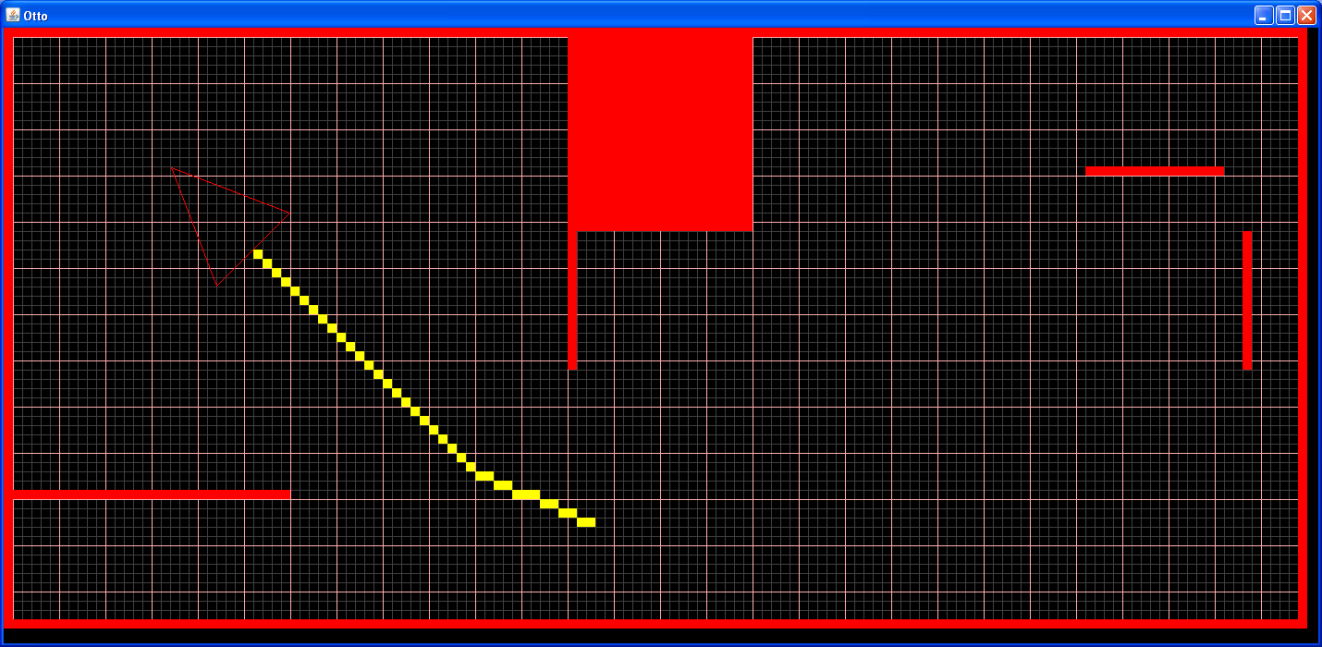


Figure 3.5.1.7: MCL Localization Complete

* + 1. Programming MCL

Programming the Lego NXJ robot very closely followed the MCL theory with a few minor customizations to optimize the process for this specific project. All computation was completed on a laptop PC while the robot simply completed the moves sent to it from the PC code and sent back the sonar readings to the nearest obstruction in the known environment. The PC and robot communicated over a Bluetooth connection with a simple communication protocol uploaded to the robots processing unit (NXJ brick).

The PC program followed a series of simple steps that will be outlined below and are broken down into more detail in their corresponding sections. These steps included:

1. Generate Particles: Generating the pose set to represent the computer’s guess at the possible locations of the robot.
2. Apply Move: Applying a move to the robot and to the poses in order to update the guesses of the computer at the robot’s location.
3. Take Reading: Taking a sonar reading from the robot to be used in the weight calculation for the poses.
4. Calculate Weight: Comparing the sonar reading from the robot to the known reading assigned to each poses and calculating the possibility that this pose is the correct guess at the robots location by analyzing the difference between these readings.
5. Resample: Updating the pose sets through a resampling process which checks if a given poses has a weight calculation higher than a random number in a specified range and if true, creating a copy of this pose so that its chances of being the correct robot location increase in the next sample.
   * + 1. Generate Particles

The process of standard Monte Carlo Localization (MCL) begins with the computer generating a random set of poses consisting of location and heading to represent the possible locations that the robot may occupy. To customize the project to limitations of the hardware a special pose class was created in Java with specific methods defined for interacting with each pose as well as retrieving information. For the purposes of this project the MCL code required that four poses be created in each cell defined within the environment. These four poses had the same x and y coordinates, however they each had a unique heading in one of the four cardinal directions (East: 0°, North: 90°, West: 180°, South: 270°). Note that this is greatly simplified from standard MCL as the headings of each pose are limited to only four different directions. This was due to the fact that the Lego Mindstorm sonar sensor was only capable of taking readings at locations perpendicular to the environment borders which for this project were all square. The poses were also defined by four integers in each of the cardinal directions representing the distances from each pose to the nearest obstruction. The code defining this process can be found in the Appendix section in the otto.java class file under the method “generateParticles()”.

* + - 1. Apply Move

After the pose sets have been created through the particle generation method the MCL algorithm requires that the robot make a series of moves in order to begin the process of converging the poses. For the purposes of this project the move called is a 90° rotation to the right and then if possible a forward move that covers a random integer distance between 1 and 6 centimeters. After each rotation the robot will go through the other steps of taking a reading, calculating the weight of each pose and then resampling the pose set as described in the section below. After the reading to the next obstruction is taken, however, the robot decides whether it has enough room to make a forward move. This is decided by checking to make sure the nearest obstruction is at least 20 centimeters ahead of the robot. If this condition is true then the robot knows it has enough room to make a forward move and then turn again. The purpose of making forward moves in between turns is to help the poses converge quicker by creating greater variance between the robot reading and the pose reading. This process of changing the move type also helps to eliminate local symmetry within the environment.

Applying the move to the robot involves selecting a number between 1 and 4 according to the move protocol defined by the Bluetooth communication file that is uploaded to the robots computing unit (1: Forward, 2: Backward, 3: Left, 4: Right). It is also important to note that the move that is applied to the robot must also be applied to each pose that is contained in the pose set. Applying this move to the poses means not only updating the x and y coordinates based on the pose heading but also updating the integers in the pose representing the distances to the nearest obstruction in each of the four cardinal directions. Updating these is also based on the heading of the poses which is itself changed with each turn move completed. The code for applying moves to the poses and robots can be found in the “applyMove()” method in the otto.java file as well as the “applyMove()” method in the Pose.java class in the Appendix section of this paper.

* + - 1. Take Reading

The take reading method is used for the sole purpose of telling the robot to take a sonar reading to the closest obstruction. It is accomplished by using the communication protocol previously discussed where the integer 5 is sent to the robot and tells the sonar sensor to take the reading and the data out stream flushes the reading from the robot to the PC. This method must also be paired with a while loop in which the reading must be less than 255 to break the loop and continue with the MCL algorithm. The method for taking the reading can be found in the “takeReading()” section of the otto.java file with its implementation being in the “solve()” method of the otto.java file.

* + - 1. Calculate Weight

The calculate weight method is used to determine if a given pose is a good guess at the actual position of the robot. This weight calculation begins with a difference squares difference between the reading that the robot takes from the sonar sensor and the particle reading of the pose as a function of its heading as defined by the equation . This difference is then used in the equation to normalize the value such that the weight will always be a number between 0 and 1 with the smallest differences yielding the highest weights.

Using a difference square difference method was shown to be superior to simply using a percentage difference method defined by the equation . This is due to the fact that the squared difference method copies more poses with smaller difference and less poses with larger difference in an exponential fashion while the percentage difference method is mostly linear in its selection of poses. This can be seen in the excel plots below. Another advantage to using this method is that is helps to compensate for compounding physical movement errors accrued in the process of having the robot make multiple moves before convergence. One important realization in the project was the idea to use modular math in reference to the angles when selecting the particle reading to be used in comparison with the robot reading. This is because the angle or heading of the robot and thus the particles may exceed 360 degrees depending on how many turns it takes to converge the pose set. Using modular math with respect to the heading will always keep this heading between -360 degrees and 360 degrees which can be handled more easily by the MCL computations. The specific code used for this section can be found in the “calculateWeidght()” method of the Pose.java class found in the Appendix of this paper.

* + - 1. Resample

The resample method in the MCL algorithm uses the weights of each pose to decide what particles from the original pose set will be copied into the new pose set that will define the next round of moves and resampling. The logic of this resampling is as follows:

* The original pose set is copied into a new temporary pose set that will be used as a reference to copy poses and is then cleared
* A while looped is created and is set to break only when the new pose set that will be created is of the same size as the original pose set.
* A random number between 0.5 and 1 is selected and then used as the lower limit of the weight filter used to screen each particle in the temporary set.
* If the particle being analyzed has a weight that is greater than the random number a copy of this particle is placed into the new set and the loop continues to the next particle until the size of the new pose set if the same as the original pose set.

This method causes multiple instances of high weight poses to be created in a pose set of a constant size while the poses with too low of a weight based on their particle readings compared to the robot reading will be eliminated from the pose set representing the guesses at the robots location. The code for this section can be found in the “resample()” method of the otto.java file in the Appendix section.

Note that this entire process for converging the location guesses is run until the number of poses that are the same is at least 25% of the total pose set size. This number was determined after multiple trials of MCL where the solution was reduced down to a reasonable cluster but failed to converge to one specific location due to low weights as a result of errors in the robots physical movements. 25% of same particles is a high enough factor of the total pose set size to constitute a majority of particles.

Some of the limitations of using this method are that the more moves needed to reduce the possible locations of the poses, the more error that will be introduced into the readings based on physical errors involved in the robots movement. For this reason the convergence criteria can be altered to optimize the solution to limit the number of moves needed to reduce the number of possible locations and thereby increase the number of same poses in the pose set.

* 1. Difficulties
  2. Testing
  3. Future Work

One of the largest potential sources of error at this point is the sensor reading the distances to the nearest wall. In theory, it should be reading the distances straight out from the robot. In actuality, the sensor will return the distance to the nearest object it finds out to a fairly wide angle. So for example, the robot may be 30 cm away from the nearest wall straight out from it while there is another wall 20 cm away but off to either side. The sensor will read the distance as 20 cm and update the localization routine accordingly, thus introducing an increasingly significant amount of error every time it takes a reading like that. To combat this, one possible solution is to have the robot travel towards a wall until it is nearly in contact with it, possibly within 5 cm. However, the robot could then find itself in a position where there are obstacle walls on either side of it. At this point the robot would either have to backtrack its path completely or would have to have sensors on the left and right side to read distances and determine if it is possible for the robot to turn in its current position or backtrack itself.

Once the robot localizes itself, the current programming has it travel to a pre-specified point on the course. It accomplishes this by creating a line of travel from its localized position to this point by the shortest route possible. But, as it travels this path, it currently takes no position/distance readings to verify it is remaining on path. As such, it may occasionally run into a wall or obstacle in its path and continue driving the wheels as if it were still on course. To solve this problem, the distance and compass sensors would need to take readings as it travels the path to confirm it is still on target. Reprogramming the route drawing parameters may help as well if instead of travelling at diagonals it traveled only in straight lines perpendicular to the walls. As the robot currently takes all readings while perpendicular to the walls during localization already, this solution would make it far easier for the robot to stay on course and travel to the destination much more accurately.

As the robot attempts to travel in a straight line for more prolonged periods of time, it tends to drift. This may be a function of the robot not turning a perfect 90 degrees, one motor possibly being either more powerful or not calibrated as accurately as the other, the wheels being slightly bowed under the weight of the robot, or more likely, a combination of all of the above. While the robot is in motion, the programming assumes that it is traveling in a straight line from point A to point B. What actually happens, though, is the robot travels from point A to point C and it believes it is at point B, thus adding an escalating degree of error in the robot’s localization routine. To correct this, several steps need to be taken to correct each issue. The motors have to be accurately calibrated to make sure they are turning at the exact same rate for the exact same amount of time. Further calibration of the compass sensor, used for the 90 degree turns, needs to happen along with further shielding of it from interfering elements in the motor and control brick. The base supporting the wheels needs to be strengthened and the possibility of adding two rear wheels to replace the marble but add additional weight distribution should not be ignored. These wheels would have to be able to turn in place to facilitate zero point turning radius. Alternatively, a second marble could be added with their locations being shifted to the rear outside edge instead of centered at the rear of the robot.

1. **Schedule and Tasks**

A timeline had been laid out for the development of the robot. This timeline was followed during the development and different tasks assigned to the group members.

1. **Conclusions**
2. **References**

[1] "A\* Search Algorithm." *Wikipedia, the Free Encyclopedia*. Web. 07 June 2011. <http://en.wikipedia.org/wiki/A\*\_search\_algorithm>.

[2] "Path Finding Tutorial - Gardenwiki." Main Page - Gardenwiki. Web. 07 June 2011. <http://wiki.gamegardens.com/Path\_Finding\_Tutorial>.

[3] Lester, Patrick. "A\* Pathfinding for Beginners." Almanac of Policy Issues. 18 July 2005. Web. 07 June 2011. <http://www.policyalmanac.org/games/aStarTutorial.htm>.

1. **Appendix**