Simulated Q-Learning in Preparation for a Real Robot in a Real Environment

Dr. Lavi M. Zamstein
University of Florida
(561) 713-2181
dino-man@ieee.org

Dr. A Antonio Arroyo
University of Florida
(352) 392-2639
arroyo@mil.ufl.edu

ABSTRACT

There are many cases where it is not possible to program a robot with precise instructions. The environment may be unknown, or the programmer may not even know the best way in which to solve a problem. In cases such as these, intelligent machine learning is useful in order to provide the robot, or agent, with a policy, a set schema for determining choices based on inputs. Reinforcement Learning, and specifically Q-Learning, can be used to allow the agent to teach itself an optimal policy. Because of the large number of iterations required for Q-Learning to reach an optimal policy, a simulator was required. This simulator provided a means by which the agent could learn behaviors without the need to worry about such things as parts wearing down or an untrained robot colliding with a wall.

Keywords

Reinforcement Learning, Q-Learning, Simulator

1. INTRODUCTION

Koolio is part butler, part vending machine, and part delivery service. He stays in the Machine Intelligence Laboratory at the University of Florida, located on the third floor of the Mechanical Engineering Building A. Professors with offices on that floor can access Koolio via the internet and place an order for a drink or snack. Koolio finds his way to their office, identifying it by the number sign outside the door.

Koolio learns his behavior through the reinforcement learning method Q-learning.

2. ADVANTAGES OF REINFORCEMENT LEARNING

2.1 Learning Through Experience

Reinforcement learning is a process by which the agent learns through its own experiences, rather than through some external source [1]. Since the Q-table is updated after every step, the agent is able to record the results of each decision and can continue learning through its own past actions.

This experience recorded in the Q-table can be used for several purposes. So long as the Q-table still exists and continues to be updated, the agent can continue to learn through experience as new situations and states are encountered. This learning process never stops until a true optimal policy has been reached. Until then, many policies may be close enough to optimal to be considered as good policies, but the agent can continue to refine these good policies over time through new experiences. Once a true optimal policy has been reached, the Q-table entries along that policy will no longer change. However, other entries in the Q-table for non-optimal choices encountered through exploration may still be updated. This learning is extraneous to the optimal policy, but the agent can still continue to refine the Q-table entries for these non-optimal choices.

Another advantage to continued learning through experience is that the environment can change without forcing the agent to restart its learning from scratch. If, for example, a new obstacle appeared in a hallway environment, so long as the obstacle had the same properties as previously-encountered environmental factors (in this case, the obstacle must be seen by the sensors the same way as a wall would be seen), the agent would be able to learn how to deal with it. The current optimal policy would no longer be optimal, and the agent would have to continue the learning process to find the new optimal policy. However, no external changes would be necessary, and the agent would be able to find a new optimal policy.

By comparison, any behavior that is programmed by a human would no longer work in this situation. If the agent was programmed to act a certain way and a new obstacle is encountered, it likely would not be able to react properly to that new obstacle and would no longer be able to function properly. This would result in the human programmer needing to rewrite the agent's code to allow for these new changes in the environment. This need would make any real robot using a pre-programmed behavior impractical for use in any environment prone to change. Calling in a programmer to rewrite the robot's behavior would be prohibitive in both cost and time, or it may not even be possible, as many robots are used in environments inaccessible or hazardous to humans.

In addition, an agent that learns through experience is able to share those experiences with other agents. Once an agent has found an optimal policy, that entire policy is contained within the Q-table. It is a simple matter to copy the Q-table to another agent, as it can be contained in a single file of reasonable size. If an agent knows nothing and is given a copy of the Q-table from a more experienced agent, that new agent would be able to take the old agent's experiences and refine them through its own experiences. No two robots are completely identical, even those built with the same specifications and parts, due to minor errors in assembly or small changes in different sensors. Because of this, some further refining of the Q-table would be necessary. However, if the agent is able to begin from an experienced robot's Q-table, it can learn as if it had all the same experiences as the other agent had. The ability to copy and share experiences like this makes reinforcement learning very useful for many

applications, including any system that requires more than one robot, duplicating a robot for the same task in another location, or replacing an old or damaged robot with a new one.

2.2 Experience Versus Example

While Reinforcement Learning methods are ways for an agent to learn by experience, Supervised Learning methods are ways by which an agent learns through example. In order for Supervised Learning to take place, there must first be examples provided for the agent to use as training data. However, there are many situations where training data is not available in sufficient quantities to allow for Supervised Learning methods to work properly.

Most methods of Supervised Learning require a large selection of training data, with both positive and negative examples, in order for the agent to properly learn the expected behavior. In instances such as investigating an unknown environment, however, this training data may not be available. In some cases, training data is available, but not in sufficient quantities for efficient learning to take place.

There are other cases when the human programmer does not know the best method to solve a given problem. In this case, examples provided for Supervised Learning may not be the best selection of training data, which will lead to improper or skewed learning.

These problems do not exist to such a degree in Reinforcement Learning methods, however. Since learning by experience does not require any predetermined knowledge by the human programmer, it is much less likely for a mistake to be made in providing information to the learning agent. The learning agent gathers data for itself as it learns.

Because of this, there is no concern over not having enough training data. Because the data is collected by the learning agent as part of the learning process, the amount of training data is limited only by the amount of time given to the agent to explore.

2.3 Hybrid Methods

In some cases, pure Reinforcement Learning is not viable. At the beginning of the Reinforcement Learning process, the learning agent knows absolutely nothing. In many cases, this complete lack of knowledge may lead to an exploration process that is far too random.

This problem can be solved by providing the agent with an initial policy. This initial policy can be simple or complex, but even the most basic initial policy reduces the early randomness and enables faster learning through exploration. In the case of a robot trying to reach a goal point, a basic initial policy may be one walk-through episode from beginning to end. On subsequent episodes, the learning agent is on its own.

Without this initial policy, the early episodes of exploration become an exercise in randomness, as the learning agent wanders aimlessly through the environment until it finally reaches the goal. With the basic initial policy, however, the early episodes become more directed. Whether it decides to follow the policy or to explore, the learning agent still has a vague sense of which direction the goal lies. Even if it decides to explore a different action choice, it will still have some information about the goal.

Because of the discounting factor inherent in Q-Learning, this initial policy does not have a major effect on the final policy. The effect of the policy given by the human program is soon overridden by the more recent learned policies.

Since the weight of the initial policy is soon minimized due to the discounting factor, it is more useful to give only a simple initial policy. A more elaborate initial policy would take more time on the part of the human programmer to create, but would be of questionable additional use. In this case, the purpose of the initial policy is only to focus the learning agent in the early episodes of learning, and not to provide it with an optimal, or even nearly optimal, policy.

2.4 Learning in Humans

When considering the different types of learning available for robots, the methods in which human beings learn should also be considered. Although there are many other methods by which a human can learn, they are capable of learning both by experience and by example.

As with a learning agent undergoing Reinforcement Learning methods, humans learning by experience often have a difficult time. There are penalties for making a mistake (negative rewards), the scope of which varies widely depending on what is being learned. For example, a child learning to ride a bicycle may receive a negative reward by falling off the bicycle and scraping a knee. On the other hand, if the same child can correctly learn how to balance on the bicycle, a positive reward may come in the form of personal pride or the ability to show off the newly learned skill to friends.

Humans can also learn by example. A friend of the child learning to ride a bicycle, for instance, might observe the first child falling after attempting to ride over a pothole. This friend then already knows the consequences for attempting to ride over the pothole, so does not need to experience it directly to know that it would result in the negative reward of falling off the bicycle.

This illustrates the principle of one agent using Reinforcement Learning to discover the positive and negative rewards for a task, then copying the resulting Q-table to another agent. This new agent has no experiences of its own, yet is able to share the experiences of the first agent and avoid many of the same mistakes which lead to negative rewards.

Learning by example in the method of Supervised Learning is also possible for humans. For instance, a person can be shown several images of circles and several images that are not circles. With enough of this training data, the person should be able to identify a shape as a circle or not a circle.

This can also be applied in a more abstract sense. Just by seeing many pictures of dogs and pictures of animals that are not dogs, a person may be able to infer what characteristics are required to make something a dog and what characteristics must be omitted. Even if the person is not told what to look for, given enough of this training data, a human should be able to decide what defines a dog and what defines something that is not a dog. The higher the similarity between dogs and things that are not dogs, the more training data is required to avoid mistaking something similar for a dog. In the same way, agents undergoing Supervised Learning must be given a large number of both positive and negative examples in order to properly learn to identify something.

When applying Reinforcement and Supervised Learning to humans, it becomes more obvious what the advantages and

disadvantages are of each method. Reinforcement Learning requires comparatively less data, but costs more in terms of both time and pain (negative rewards) or some other penalty. Supervised Learning requires much less time and minimizes the negative feedback required of Reinforcement Learning, but in exchange requires much more data, both positive and negative examples, to be presented for learning to take place. There are both advantages and disadvantages for using each method, but the costs are different for the two.

3. SIMULATOR

3.1 Reason for Simulation

Reinforcement learning algorithms such as Q-learning take many repetitions to come to an optimal policy. In addition, a real robot in the process of learning has the potential to be hazardous to itself, its environment, and any people who may be nearby, since it must learn to not run into obstacles and will wander aimlessly. Because of these two factors, simulation was used for the initial learning process. By simulating the robot, there was no danger to any real objects or people. The simulation was also able to run much faster than Koolio actually did in the hallway, allowing for a much faster rate of learning.

3.2 First Simulator

The first simulator chosen for the task of Koolio's initial learning was one written by Halim Aljibury at the University of Florida as part of his MS thesis [2]. This simulator was chosen for several reasons. The source code was readily available, so it could be edited to suit Koolio's purposes. This simulator was written in Visual C. Figure 5-1 shows the simulator running with a standard environment.

The simulator was specifically programmed for use with the TJ Pro robot [3] (Figure 1). The TJ Pro has a circular body 17 cm in diameter and 8 cm tall, with two servos as motors. Standard sensors on a TJ Pro are a bump ring and IR sensors with 40KHz modulated IR emitters. Although Koolio is much larger than a TJ Pro, he shares enough similarities that the simulator can adapt to his specifications with no necessary changes. Like the TJ Pro, Koolio has a round base, two wheels, and casters in the same locations. Since the simulation only requires scale relative to the robot size, the difference in sizes between the TJ Pro and Koolio is a non-issue.

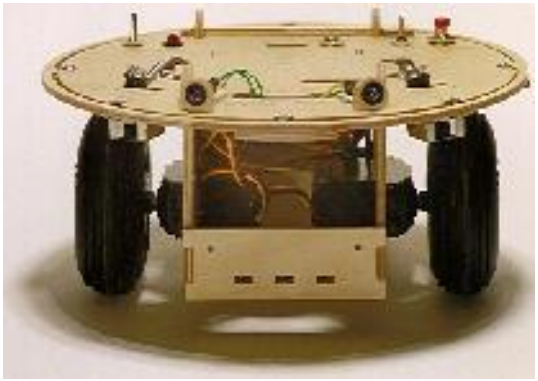

Figure 1. The TJ Pro.

3.3 Simulator Considerations

Simulations have several well-known shortcomings. The most important of these shortcomings is that a simulated model can never perfectly reproduce a real environment. Local noise, imperfections in the sensors, slight variations in the motors, and uneven surfaces can not be easily translated into a simulation. The simplest solution, and the one chosen by this simulator, is to assume that the model is perfect.

Because of differences in the motors, a real robot will never travel in a perfectly straight line. One motor will always be faster than the other, causing the robot to travel in an arc when going 'straight' forward. For the purposes of the simulation, it is assumed that a robot going straight will travel in a straight line. In the long run of robot behavior, however, this simplification of an arc to a straight line will have minimal effect, since the other actions performed by Koolio will often dominate over the small imperfection. Another concession made for the simulator is to discount any physical reactions due to friction from bumping into an object. It is assumed that Koolio maintains full control when he touches an object and does not stick to the object with friction and rotate without the motors being given the turn instructions.

However, the first chosen simulator had several issues which made it unsuitable for the learning task. It was difficult to make an arena with a room or obstacles that were not quadrilaterals, since the base assumption was that all rooms would have four sides. This assumption does not apply for many situations, including the hallway in which the robot was to be learning.

In addition, while the simulator could be edited to add new sensors or new types of sensors, the code did not allow for easy changes. Since several different sensor types were required for the simulated robot, a simulator that allowed for simple editing to add new sensors was required.

The simulator also assumed a continuous running format. However, the task for which reinforcement learning was to be performed is episodic, having a defined beginning and end. The simulator was not made to handle episodic tasks.

Because of these reasons, a new simulator was required to perform the learning procedures necessary for the task.

3.4 New Simulator

A new simulator was made to better fit the needs of an episodic reinforcement learning process. In order to do this, the simulator needed to be able to automatically reset itself upon completion of an episode. It also needed to be as customizable as possible to allow for different environments or added sensors to the robot.

When creating this new simulator, customization was kept in mind throughout. The intention was to make a simulator able to perform learning tasks for Koolio under various situations, as well as to be easily alterable for use with other robots for completely different learning tasks.

3.4.1 Arena Environment

The simulation environment, referred to as the arena from this point, is an area enclosed within a series of line segments. These segments are not limited in length or number. The line segments are defined by their endpoints, which are listed in the *SegmentEndpoint* array in order. When the simulation begins, it constructs the arena by drawing lines between consecutive points in *SegmentEndpoint* until it reaches the end of the array. Once

the end has been reached, the arena is closed by drawing a line from the final point back to the first.

This treatment of the arena walls assumes that the arena environment is fully enclosed. Obstacles that touch the walls can be created by altering the arena walls to include the obstacle endpoints. Obstacles that do not touch the walls are not included in the simulator by default, but they may be added if desired. This can be accomplished by creating an additional array of obstacle endpoints and checking this array at every instance the walls are checked for movement and sensor purposes.

Once the arena walls have been entered, they are rotated five degrees clockwise. This rotation helps to prevent infinite slopes while calculating sensor distances, since there will no longer be perfectly vertical walls in the arena. However, error checks are still in the code to prevent infinite slopes entering any of the calculations.

In addition to the walls, several other landmarks were required for Koolio's episodic learning task. Since Koolio must be able to detect markers for the goal and the end of the world, these markers must also be represented in the simulation.

The simulator assumes a single goal point, defined in *GoalPoint*. If more than one goal is desired, the code must be altered to allow for this.

Multiple markers for the end of the world exist, defined in *EndOfWorldPoints*. There is not a limit to the number of end of the world markers allowed in the simulator.

For preliminary tests with the simulator and learning process, a large, square arena was used (Figure 2). The goal point was placed on the center of one of the walls. There were no end of the world points present.

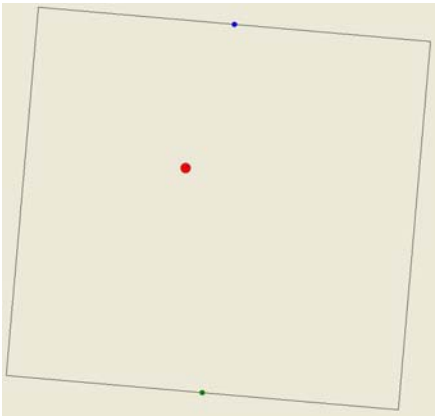

Figure 2. A simple square arena.

This simple arena served as a testbed for all the learning experiments. The reason for choosing such a basic arena is that it was discovered early in the testing process that an arena in the shape of a long hallway had a relatively large percentage of positions in which the agent was in contact with the walls. By comparison, a large square arena has a much larger proportion of open space to area along the walls. This allowed for a greater emphasis on the agent learning without its efforts being frustrated by constant proximity of walls.

After preliminary testing, the simulation for Koolio in the hallway environment was used. The arena was made as a long, narrow rectangle with T-shaped openings on either end to approximate the shape of the hallway. Ceiling tiles in the actual hallway were used as a constant-size measuring tool. Each ceiling tile is approximately twenty-four inches square and is slightly larger than Koolio's base, at twenty inches in diameter. The easiest conversion from reality to simulation was to make each inch one pixel in the simulator. The hallway is 68 tiles long and $3\frac{2}{3}$ tiles wide, giving it simulator dimensions of 1632 pixels long and 88 pixels wide. Each door in the hallway is $1\frac{1}{2}$ ceiling tiles wide, converting to 36 pixels in the simulator.

3.4.2 Robot

Within the simulator, the robot is defined by the location of its center point, *RobotLocation*, its heading, *RobotOrientation*, and its radius, *RobotRadius*. The radius is a constant that cannot be changed by the simulator, but the location and heading change as the code runs. The simulator assumes that the robot base is either circular in shape or can be easily simplified into and represented by a circle.

The two values of *RobotLocation* and *RobotOrientation* are the sole indicators of the robot's position and heading within the arena. These two values are used in determining the movements of the robot, as well as affecting the sensor readings.

3.4.3 Sensors

Koolio has several different types of sensors: a compass, an array of sonar sensors, and three cameras. These three types of sensors were implemented into the simulator.

3.4.3.1 Compass

The compass is the simplest of the sensors implemented in the simulator, since it requires only a single input, the robot's heading. The compass reading is calculated in *CalculateCompass*.

Since *RobotOrientation* is a value between 0 and 2π, it is divided by 2π and multiplied by 256 to get a value between 0 and 255 for the heading. Since the compass used for Koolio is very inaccurate, 25% noise is inserted into the signal.

The *SensorEncodeCompass* function takes this reading and compares it to known values for Northwest, Northeast, Southwest, and Southeast. The compass reading is encoded as *North* if it lies between Northwest and Northeast, as *South* if it lies between Southwest and Southeast, as *East* if it lies between Northeast and Southeast, and as *West* if it lies between Northwest and Southwest.

Because the actual compass used on Koolio was very unreliable, it was decided to remove the compass from all calculations in the simulator. The code still exists, however, and can be used or altered for any similar sensor in a future simulation.

3.4.3.2 Sonar

The sonar sensors are defined by their location on the robot in relation to the front (Figure 3). These angles are listed in *SonarSensors*. The order of the sensors in that list is tied to the arbitrarily-assigned sensor numbers later in the code. More sensors can be added and the order of sensors can be rearranged,

but that must also be done in all uses of the sensors. The viewing arc of the sonar sensors is 55 degrees, defined in the constant *SonarArc*.

The sonar sensors used on Koolio return distance values in inches. If different sonar sensors are used, then an additional function will be required to convert real distance into the reading that the sensors output.

Each of the sonar sensors is read individually in *CalculateSonar*. The first step in reading the sensor's value is to split the viewable arc of the sensor into a number of smaller arcs. For each of these smaller arcs, the distance to the nearest wall is calculated. The reading from the sonar is the shortest of these distances.

In order to calculate this distance for each smaller arc, the simulator first finds the equation of the line that passes through the center of that arc. Since straight line can be defined by a point and a slope, all the needed information for the equation of that line is available. The slope of the line is determined by the angle between the center of the arc and the robot's heading. The point needed for the line comes from the robot's location.

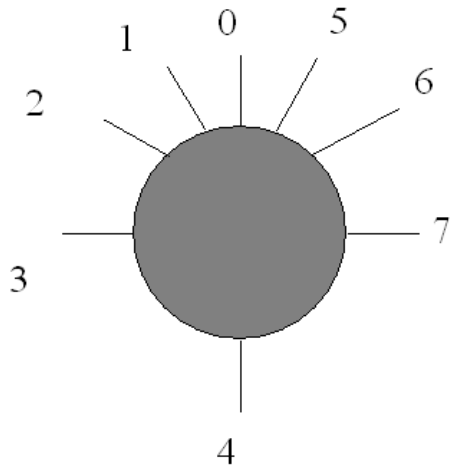

Figure 3. The locations of the sonar sensors on the agent.

For each of these lines through the smaller arcs, the distance to each wall along that line must be checked. This is done in *LineIntersectDistance*, a function that takes the equation of the sensor line and the endpoints of a wall as inputs and returns the distance from the robot to that wall along the sensor line segment.

Using the endpoints for the wall, the equation of the line containing that wall is found. These two equations are used to find the intersection point (Ix,Iy) between the two lines.

Once the intersection point is found, it must be converted from Cartesian coordinates to polar coordinates in relation to the location of the robot.

Although the intersection point is known, a check must be made to ensure that it is within the viewable angle of the sensor. If it is not within the arc of the sensor, then the point of intersection is actually on the opposite side of the robot from the sensor. Because equations deal with lines and not line segments, this check must be made, since the line continues on both sides of the robot. In the case that the sensor cannot see the intersection point, the distance is set to the large constant *ReallyLongDistance*. Figure 4 shows an example of a situation in which *ReallyLongDistance* will be returned as the sensor reading.

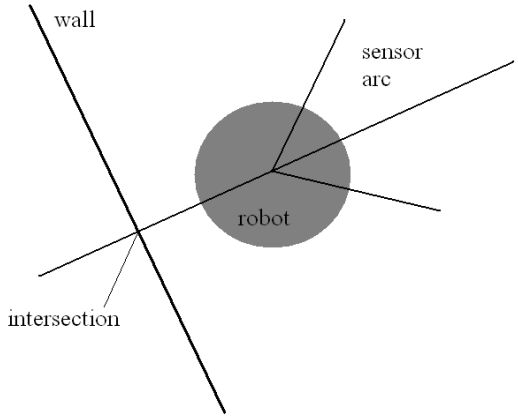

Figure 4. Configuration that will result in a sensor reading of *ReallyLongDistance* due to the intersection being outside of the sensor's viewing arc.

With the distance known, or set to *ReallyLongDistance* if there is no visible wall, that value is returned back to *CalculateSonar* and compared to each of the other distances within the smaller arc. Only the shortest of these distances is kept as the reading from that smaller arc. This is because the sonar sensors are unable to see through walls, and multiple different readings indicate that there is another wall past the closest one that could be seen if that closest wall was not there. This shortest distance is put into the *MinSegmentDistance* for the smaller arc.

Once each of the smaller arcs has a *MinSegmentDistance* set, they are all compared, and the shortest is set into *MinDistance*. This is the distance to the closest wall and is what the sensor actually sees. Only the closest of these readings is needed (Figure 5).

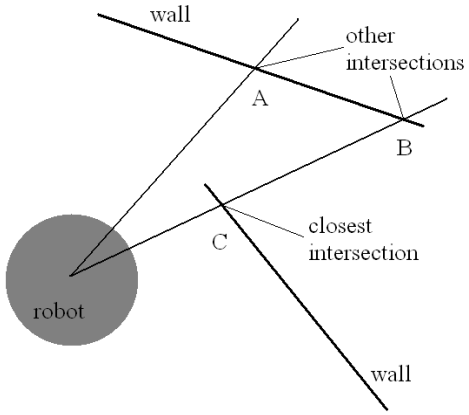

Figure 5. Multiple detected walls. Only the closest wall will matter. Intersection B will be removed during *LineIntersectDistance* because intersection C is closer on that direction line, making C the *MinSegmentDistance* for that smaller arc. Since intersection C is closer than intersection A, C will also become the *MinDistance* for the entire sensor reading.

Once the reading of the sonar sensor is known, the radius of the robot is subtracted, so the sensor reads the distance from the edge of the robot, not from the center. Ten percent noise is then factored into the sensor reading.

Once the sensor reading is calculated, it must be encoded into a set of intervals. *SensorEncodeSonar* takes the readings from all of the sonar sensors and returns encoded interval values. Koolio uses an array of eight sonar sensors. However, as explained below, in order to make the size of the Q-table more manageable, some of the sensors are combined. Sonar sensors 1 and 2, located $\frac{2\pi}{9}$ (40 degrees) and $\frac{4\pi}{9}$ (80 degrees) on the diagonal front-left of Koolio, have their values averaged into a single value before encoding. Likewise, Sonar sensors 5 and 6, located at the same angles on the diagonal front-right of Koolio, are also combined into a single value.

Within *SensorEncodeSonar*, the values are compared to a series of constants for intervals. A sonar sensor can return a value of *Bump*, *Close*, *Medium*, or *Far*, according to the values in Table 1.

Table 1. Range values used for Koolio's sonar sensor encoding.

Distance Classification	Range
Bump	< 6 inches
Close	6 - 24 inches
Medium	24 - 60 inches
Far	> 60 inches

If the reading is within six inches of the robot, it is returned as *Bump*. If the reading is between six inches and two feet, it is classified as *Close*. If it is between two and five feet, it is classified as *Medium*. Any sensor readings beyond five feet from the robot are considered *Far*. All of these values can be changed for sonar sensors with different specifics.

3.4.3.3 Cameras

Like the sonar sensors, the cameras are also defined by their location on the robot in relation to the front. These angles are listed in *CameraSensors*. More cameras can be added and their order can be rearranged, but that must also be done in all uses of the cameras. The viewing arc of the camera is 45 degrees, defined in the constant *CameraArc*.

The cameras on Koolio return values through external code that returns distance value in inches. If cameras or drivers are used which return distance values in some other format, then an additional function will be required to convert real distance into the reading that the sensors actually output.

Unlike the sonar sensors, which have the single function of reading distance to the walls, the cameras perform two different readings and are each treated in the code as two sensors. Each camera returns a distance to the Goal markers and to the End of the World markers. For Koolio, these markers are large colored squares (Figure 6). In the simulator, these markers are indicated to the user as colored dots (Figure 7).

Like the sonar sensors, each camera is read individually. Unlike the sonar sensors, since each camera is checking for two things, there are two functions. The distance to the Goal point is checked in *CalculateCameraToGoal* and the distances to the End of the World points are calculated in *CalculateCameraToEndOfWorld*. The two functions are similar, but different enough to require distinct functions. The simulator allows for only a single Goal point. If more than one Goal point is desired, it must be added and *CalculateCameraToGoal* must be altered to allow for multiple Goal points.

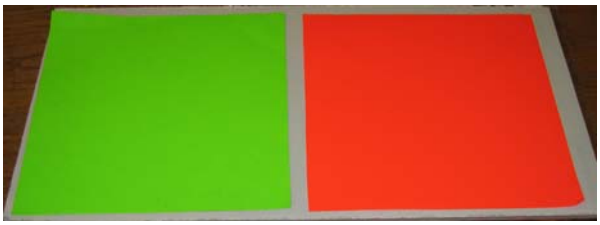

Figure 6. Goal (left) and End of the World (right) markers used in the real environment.

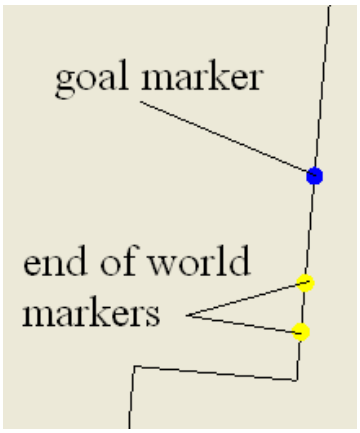

Figure 7. Goal and End of the World markers in the simulator.

There are two main conceptual differences between the camera reading functions and the sonar reading function. The first is that, while a sonar sensor will always see a wall, the cameras may not see any of the Goal or End of the World points. The camera functions have a finite list of points to check for distances and must also ensure that there is no wall between the robot and these marker points. There is always a possibility of none of these points being in the viewing angle of any of the cameras.

The second conceptual difference is that each camera is divided into three focus areas. The 45-degree arcs of each camera are split into three even 15-degree arcs to determine whether the point seen is in focus in the center of the camera, on the left side, or on the right side.

The first step in *CalculateCameraToGoal* is to convert the line between the robot and the Goal point into polar coordinates in relation to the robot's current location. After the polar coordinates of this line are obtained, a check must be made to ensure that this line is within the viewing angle of the camera. The polar coordinate angle is compared to the outer edges of the viewing arc. If that angle is in between the two arc edges, then the Goal point is within the viewing angle and the Boolean variable *GoalSeen* is set to true. Otherwise, the Goal point is outside of the viewing arc, and the distance is set to *ReallyLongDistance* and *GoalSeen* becomes false.

If the Goal point is inside the camera's viewing arc, then another check must be made to ensure that there are no walls in between the robot and the Goal point. If *GoalSeen* is false, this step is bypassed, since it would be a redundant check. The formula for the line between the robot and the Goal point is calculated using Cartesian coordinates. This line is then compared to each of the walls in the arena. The distance from the robot to each wall is calculated using *LineIntersectDistance*, the same function used for determining the readings of the sonar sensors. If this distance is less than the distance to the Goal point, minus 5, then the line of which the wall is a segment is between the robot and the Goal.

The reason the distance to the Goal point must have 5 subtracted for the comparison here is to allow for rounding errors. Because the Goal point is on a wall itself, it is possible for the distance functions to calculate that the Goal is actually a fraction of an inch behind the wall. In this case, the wall would normally be in the way. However, if 5 is subtracted from that distance, it removes the possibility of error. While 5 is more than necessary, since the error is usually on the order of a hundredth of an inch or less, it allows for errors in the case of very large arenas with very large numbers used. This allowable error of 5 will also never be too high, because there will never be a wall 5 inches or less directly in front of the goal, as that would make the goal unreachable in any case.

Since lines are infinite but the walls are only segments, the endpoints of the wall must be checked to determine if the intersection point lies on the line segment that composes the wall (Figure 8). If the x-coordinate of the intersection point is between the x-coordinates of the wall endpoints and the y-coordinate of the intersection point is between the y-coordinates of the wall endpoints, then the intersection point lies on the wall. In this case, *GoalSeen* is set to false and no more walls need to be checked. Otherwise, the rest of the walls are checked.

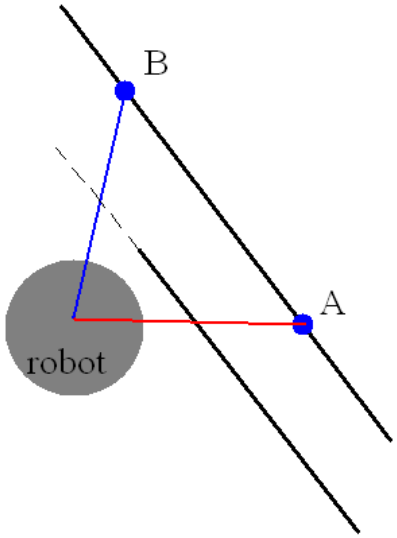

Figure 8. Two examples of line of sight to a Goal point. Point A is blocked by the wall. The full line that contains the wall lies between the robot and point B, but the point is considered visible after checking the wall endpoints.

Ten percent noise is then inserted into the distance. A final check is performed, to ensure that the goal point is within the maximum distance at which the camera can detect. If the distance is greater than *MaxCameraView*, a constant holding this maximum distance, then *GoalSeen* is set to false.

If *GoalSeen* is false at the end of the function, the distance is set to *ReallyLongDistance*. *GoalSeen* is then assigned to the camera's global settings.

If *GoalSeen* is true, then the focus region for that camera is checked. *ArcSplitRightCenter* and *ArcSplitLeftCenter* are used as the angles where the camera arc is split between right and center focus and between center and left focus, respectively. If the angle of the line to the goal point is in between the right edge of the camera arc and *ArcSplitRightCenter*, then the goal is seen in the right focus. If it is in between *ArcSplitRightCenter* and *ArcSplitLeftCenter*, then it is in the center focus. If it is in between *ArcSplitLeftCenter* and the left edge of the camera arc, then it is in the left focus.

Finally, the focus direction is set to *FocusInCamera* and the distance is returned.

Like the sonar sensors, these distance readings must then be encoded into a set of intervals. *SensorEncodeCamera* takes the readings from all the cameras and returns encoded interval values. Within *SensorEncodeCamera*, the values are compared to a series of constant intervals. A camera can return a value of *Close*, *Medium*, or *Far*, according to the values in Table 2.

Table 2. Range values used for Koolio's camera encoding.

Distance Classification	Range
Close	< 36 inches
Medium	36 - 96 inches
Far	> 96 inches

If the reading is within three feet of the robot, it is classified as *Close*. If the reading is between three and eight feet, it is classified as *Medium*. If the reading is more than eight feet from the robot, it is considered *Far*. These values can be changed for cameras and drivers with different minimum and maximum identification ranges.

Calculating the distance to an End of the World marker is similar to calculating the Goal marker distance. However, since there is only one Goal marker and several End of the World markers, the method is slightly different. *CalculateCameraToEndOfWorld* contains a loop which checks the distance from the robot to each of the End of the World points and returns the distance to the closest one. After checking the distances to each End of the World marker using the same method as in *CalculateCameraToGoal*, the distances are stored in an array. Once all End of the World distances have been calculated, the shortest one is chosen as the sensor reading. Noise is added and the reading encoded using *SensorEncodeCamera*. The division into three focus areas is not used for the End of the World markers.

3.4.4 Movement

The position of the robot in the simulated arena environment is defined by the location of its center, *RobotLocation*, and the

direction it is facing, *RobotOrientation*. These are changed when the robot moves in the *PerformAction* function, which takes an action choice as input. There are seven possible action choices, though more can be added if desired. The action choices are *Action_Stop*, *Action_Rotate_Left*, *Action_Rotate_Right*, *Action_Big_Rotate_Left*, *Action_Big_Rotate_Right*, *Action_Forward*, and *Action_Reverse*. For the sake of simplicity, the robot can only move forward or back, wait, or rotate. If an action that incorporates both movement and turning is desired, it can be added to this function and to the list of actions in the constant list.

If *Action_Stop* is chosen, nothing is changed. If *Action_Rotate_Left* is chosen, the global constant *RotateActionArc* is added to *RobotOrientation*. If *Action_Rotate_Right* is chosen, *RotateActionArc* is subtracted from *RobotOrientation*. Similarly, if *Action_Big_Rotate_Left* or *Action_Big_Rotate_Right* are chosen, *BigRotateActionArc* is added to or subtracted from *RobotOrientation*. If *Action_Forward* is chosen, the location is moved by the global constant *MoveActionDistance* in the direction of the current orientation. If *Action_Reverse* is chosen, the location is moved *MoveActionDistance* in the direction opposite of the current orientation.

After the movement is made, the simulator must check to make sure that the robot did not pass through the walls of the arena, since the real environment is made of solid walls. The *RobotContacting* function detects whether or not the robot has passed through an arena wall when performing a movement action and, if it has, places the robot back inside the arena tangent to the wall. This function is only meant to work with round robots. If the robot is a shape other than a circle, or cannot at least be represented roughly by a circle, the calculations in *RobotContacting* may not work. This is an acceptable limitation, however, as most robots capable of this sort of movement are round or can be approximated by a circle. This function also assumes that there are no tight spaces in the arena. In other words, there are no locations the robot can be where it touches more than two walls. This is an acceptable assumption, since the robot should never try to move into tight spaces, if they did exist, and should not be expected to squeeze into areas that are exactly as wide as, or even narrower than, its diameter.

The robot can still act strangely at some of the convex corners. This is because it attempts to cut across the corner when moving straight toward it and is put back to where it should be. Therefore, if the robot keeps attempting to cut across the corner, it will make no progress. This only happens when the robot is attempting to hug the wall, remaining in tangent contact with it. It is a side effect of the code, as it only happens when the robot is moving counter to the order of the walls. For example, if the robot I hugging the wall between *SegmentEndpoint* 4 and 3, approaching the corner formed with the wall between *SegmentEndpoint* 3 and 2, this effect may occur. However, it does not happen when the robot approaches the same corner from the other direction. If the robot is not hugging the wall and just passes near the corner, this situation never arises.

The *RobotContacting* function uses three Boolean variables as tags to keep track of the status of the robot's interaction with the walls of the arena: *testval*, *foundintersection*, and *morecontacts*. All three are initialized to false. This function also uses a two-dimensional array, *whichwalls*, which stores the starting segment point number of a wall that is intersected by the robot. Both of them are initialized to –1.

The first step in determining whether the robot is intersecting a wall is to go through each of the walls and check for intersections. For each wall, the equation of that wall is found. To check for intersections, the equation of the wall and the equation of the circle that makes up the perimeter of the robot are set equal to each other.

If the first endpoint is farther to the left than the second and the x-coordinate of the intersection is between them, the check becomes a positive over a positive for a positive number. If the intersection is not between them, the check becomes negative over positive for a negative number. If the second endpoint is farther to the left than the first and the x-coordinate of the intersection is between them, the check becomes a negative over a negative for a positive number. If the intersection is not between them, it becomes a positive over a negative for a negative number.

In addition, if a point on a line is in between two points, then the distance from one endpoint to that middle point should be less than the distance between the endpoints. If the x-coordinate of the intersection point is in between the two wall endpoints, then, the check number should be less than or equal to 1.

Only the x-coordinates of the intersection point are checked, since the intersection point is in between the y-coordinates of the wall endpoints if and only if it is also in between the x-coordinates of the wall endpoints. Also, since these check numbers are only used for their values relative to 0 and 1 and not their actual numbers, the y-coordinates do not need to be checked.

Given these properties for the check numbers, at least one of the intersection points are valid if and only if *Check1* (Equation 1) is less than or equal to 1 and either *Check1* or *Check2* (Equation 2) are greater than or equal to 0. While the extra comparison to 1 is not necessary, it is still valid. If these conditions are met and there is a valid intersection point, *testval* is set to true.

$$Check1 = \frac{IntX1 - WallEndX1}{WallEndX2 - WallEndX1} \quad (1)$$

$$Check2 = \frac{IntX2 - WallEndX1}{WallEndX2 - WallEndX1} \quad (2)$$

If *testval* is true and *foundintersection* is false, the average of the two intersection points is taken. The *atan2* function is used to find the direction of the line from the center of the robot to the average of the intersection points, which is stored in *Direction*. If no intersection has been found to this point, then *whichwalls*[0] will still have its initial value of –1. If this is the case, it is instead set to the identifying number of the wall that is being intersected. After this, *foundintersection* is set to true and *testval* is set to false.

If both *testval* and *foundintersection* are true, then the robot is intersecting more than one wall. The *whichwalls* array is checked to make sure that there is a value stored in *whichwalls*[0] and *whichwalls*[1] is still at its initial –1 value, then *whichwalls*[1] is set to the identifying number of the second wall. The Boolean *morecontacts* is set to true and the loop that checks all walls is broken. This means that if the robot is touching three walls at once, it may become stuck. However, such an arena is unlikely to

be made and can be avoided by not allowing arenas with any tight spaces where three or more walls make an alcove that the robot would not be able to physically enter.

The loop to check each of the walls ends when either all the walls have been checked with no more than one intersection or has been ended prematurely by the existence of two intersections.

If *foundintersection* is true and *morecontacts* is false, then a single intersection was found. The distance that the edge of the robot overhangs the wall, *Overlap*, is calculated by subtracting the distance between the robot's center and the intersection point from the robot's radius. Once *Overlap* is found, the robot's location is moved that distance away from the wall in the direction of *Direction*.

If both *foundintersection* and *morecontacts* are true, then there were two intersections found. First, the *Overlap* is calculated for the first wall that was encountered and the robot is moved away from it as above. The robot's position is then checked in relation to the second wall that was encountered, performing a single iteration of the intersection-finding loop on that wall. The new intersection, if any, is calculated, and the robot is moved away from the second wall by a newly-calculated *Overlap* value.

After all the location corrections, if any, are performed, the function ends, returning *foundintersection* to indicate whether one or more intersections were found.

After the simulator ran for several thousand episodes, it was observed that there was a small chance the robot could become stuck against a corner under certain circumstances. To avoid that, another check was added. If the user-indicated *Auto_Rescue_Toggle* is turned on, the episode has run for more than *MaxStepsBeforeRescue* steps, and the sensors indicate that the robot is in a *Bump* condition, then the *Rescue_Robot* function has a *Rescue_Chance* chance of being called. This function forces the robot to execute a random number (between 0 and the constant *Rescue_Max_Backup*) of reverse actions. After each reverse action, *RobotContacting* is called to ensure that it cannot be rescued through a wall to the outside of the arena. After the reverse actions are performed, the robot is spun to face a random direction. After this rescue, all of the sensors are recalculated and *Has_Been_Rescued* is set to true to indicate that a rescue occurred in the current episode.

3.4.5 Displays and Controls

Aside from the arena walls, the Goal and End of the World markers, and the robot's position and direction, the simulator also has several optional displays that can be toggled on or off by the user.

The *Show Sensors* button toggles extra information on the robot itself. When it is activated and the robot receives a Bump signal from one of the sonar sensors, the robot is drawn pink instead of red. When the robot has run off the End of the World, it is displayed in purple. When the robot reaches the goal, it is displayed in black. This toggle also displays lines indicating the sensors, with black lines for the sonar sensors, blue lines for cameras seeing the Goal, and yellow lines for cameras seeing the End of the World. Figure 9 shows some examples of the sensor displays.

The encoded readings for the sensors can also be displayed (Figure 10). This display toggles with the *Show HUD* button. When turned on, a list of all the sensors appears with readings of *Close*, *Medium*, or *Far*, as well as a field that indicates of any of the sonar sensors read a Bump state.

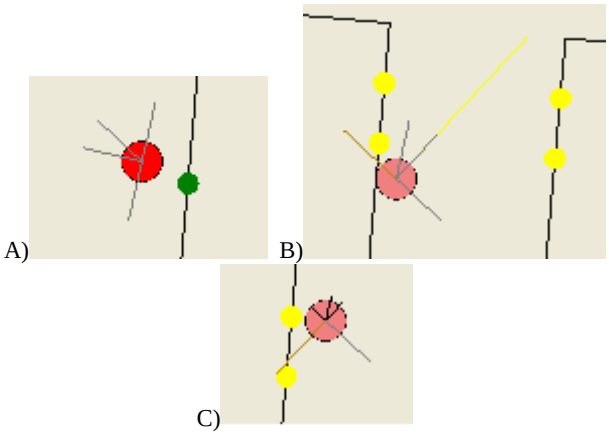

Figure 9. Sensor displays. A) The robot near the start area. B) The robot in a Bump state seeing the End of the World markers in the front camera (Medium) and the left camera (Close). C) The robot in a Bump state seeing an End of the World marker in the left camera (Close).

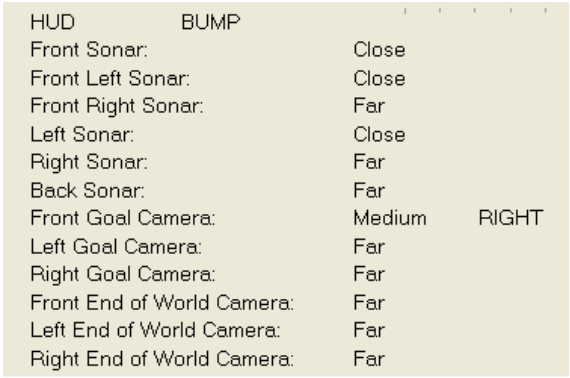

Figure 10. The sensor HUD.

A statistics display for the simulation can be toggled with the *Show Stats* button (Figure 11). This shows the reward, number of steps, and number of bumps for the current episode, as well as the total number of episodes, the total number which ended by reaching the goal, and the average reward, number of steps, and number of bumps per episode.

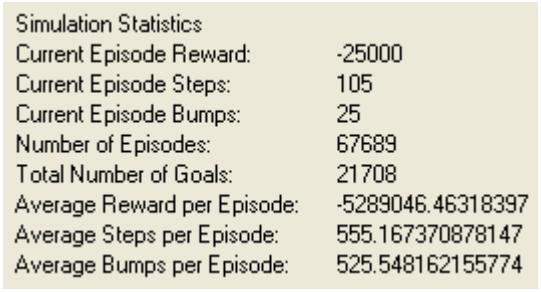

Figure 11. Simulation Statistics.

A display of values from the current Q-table location can also be toggled with the *Display Q Values* button (Figure 12). This shows the expected return for each of the possible action choices,

the choice made from those return values, the current reward for the given action, and whether ε has triggered a random action choice.

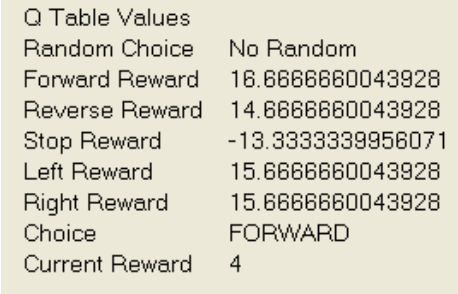

Figure 12. Q-Value display.

Two other special displays are shown, indicating when the auto rescue function and the clockless timer have been turned on. A tag also displays if the robot had to be rescued in the current episode.

Besides the buttons to turn the display toggles and function toggles on and off, there are also buttons to begin and pause the simulation, to manually force *Rescue_Robot* to run, and to exit the simulation (Figure 13). If the normal *Exit* button is pressed, the simulation closes and factors the current episode into the Q-table. If the *Exit Without Save* button is pressed, the simulation closes and discards all data from the current unfinished episode. There is also a slider to control the speed of the clock.

Figure 13. The main control panel.

In addition, there is a control panel for manual controls (Figure 14). These allow for actions to be entered in and override the normal choices from the Q-table. *Manual Mode* toggles the mode on and off, and *Step Timer* advances the timer by a single tick when manual mode is on.

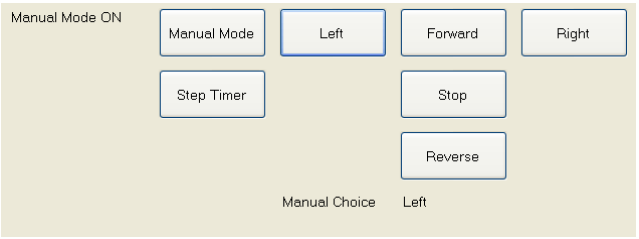

Figure 14. The control panel for manual mode.

3.4.6 Learning Algorithm

The Q-learning formula in the simulator involves five steps: choose an action, find the reward for that action choice, read the next state, update the Q-table, and move to the next state.

The first step is to choose an action. In order for Q-learning to allow for exploration, there must be a small chance ε of selecting an action at random. A random number is generated. If that number is greater than 1-ε, a second random number between 0 and *NumActions*, the number of different possible actions, is generated. This random number determines the action choice, where each choice is an equally likely outcome.

Otherwise, the greedy algorithm is performed. The Q-table values for the current state and all possible action choices are read, and the action choice that yields the highest reward is selected. The robot's current position and direction are stored in *OldLocation* and *OldOrientation*, and *PerformAction* is called with the selected action choice.

After the action has been performed, the next state must be read. Since the state is defined by the sensors, all of the compass, sonar, and camera sensors are read and encoded.

After the new state is read and the status of the robot is found, the reward for that action choice must be determined. There are four categories of rewards, based on the status of the robot after performing the selected action. Each of the four categories has the same set of reward conditions, determined by the action taken, whether the agent has reached the goal, encountered the end of the world, bumped into an obstacle, or is in the center of the hallway. If the action chosen was stop and the front camera reads a Close distance to the goal, then the robot receives the Goal reward for that category. If the robot has run off the end of the world, then it receives the End of World reward. If it is in a bump state, it receives the Bump reward. If none of these three conditions are present, the reward depends solely on the last action taken.

Table 3. Status variables in the simulator.

Status Variable	True When
AtGoal	Goal is seen at distance *Dist_Close* in the front camera
AtEndOfWorld	Agent has gone past the end of the world
BumpDetected	Any sonar sensor reads a distance of *Dist_Bump*
CenterOfHall	Sonar sensors on the left and right read the same distance
CanSeeGoal	Any camera sensor can see the goal at distance *Dist_Close* or *Dist_Medium*
GoalInFront	*CanSeeGoal* is true and the goal is in either the right or left focus of the front camera
GoalFrontCenter	*CanSeeGoal* is true and the goal is in the center focus of the front camera

The reward categories are based on the status of the robot. There are seven Boolean variables used to specify the status of the agent (Table 3). The category with the highest rewards is accessible if *GoalFrontCenter* is true. The category with the next-highest rewards is accessible if *GoalInFront* is true. The next category is accessible if *CanSeeGoal* is true. The category with the smallest

rewards is used when none of those three status variables are true. Within each category, the reward is chosen based on the other status variables (*AtGoal*, *AtEndOfWorld*, *BumpDetected*, and *CenterOfHall*) and the action chosen to get to the state that gives these status readings.

After the reward is chosen, the Q-table must then be updated using the Q-formula (Equation 3). Although the sensor readings of the next state are known, the action choice is not yet known. However, all that is needed for the Q-formula is the value of the highest possible reward. The rewards for all possible action choices for the current sensor state are read, and the highest saved. The reward for the current state is also read, and the Q-formula calculates the new value for the reward of the current state.

$$NewQ = CurrentQ + \alpha * (reward + (\gamma * \max(NextQ)) - CurrentQ) \quad (3)$$

The final step is to move to the next state.

3.4.7 Q-Table Implementation

Initially, the simulator was designed with fifteen sensors: one compass, six cameras (three each for the goal and the end of the world), and eight sonar sensors. The compass could contain four values (North, South, East, and West). Each of the cameras could contain four values (Close, Medium, Far, and Very Far). Each of the sonar sensors could contain five values (Bump, Close, Medium, Far, and Very Far). However, the large number of possible readings let to a combinatorial explosion. With seven sensors containing four possible values each and eight sensors containing five possible values each, there were $6.4*10^9$ possible combinations. Each of these sensor states can have seven different action choices, making a Q-table with nearly $4.5*10^{10}$ different states. Each state requires a 64-bit double type for its value, resulting in a Q-table size of 333.8 GB. This file size is obviously much too large to be of any use.

In order to fix this, the number of states had to be reduced. Since the compass is an erratic and unreliable sensor, it was dropped entirely. Some of the sonar sensors are combined as well. Sonar sensors 1 and 2 are combined into a single reading, and sensors 5 and 6 are also combined into a single reading. Finally, the Very Far value was removed from all sonar and camera sensors.

With these changes, there were six camera sensors with three values each and six sonar sensors with four values each, along with the seven action choices. This made for less than $12.1*10^7$ different states in the Q-table. With this smaller number of states, the Q-table size was reduced to approximately 159.5 MB, more than two thousand times smaller and a much more manageable file size.

When it was later decided to add focus regions for each goal camera, the number of states increased again. Each of the three goal cameras was given a focus reading that could be one of four values: right, left, center, or none. This increased the number of possible states to about $1.3*10^9$. This would result in a maximum Q-table size of approximately 10.0 GB.

This file size is a maximum and will only be reached if the Q-table is completely full. However, because of the nature of Q-learning, the sixteen-dimensional matrix will be extremely sparse. For example, a situation will never arise when all three cameras read the Goal point as being at a Close distance. Because the matrix is sparse, the use of a data structure such as a large array would result in a large amount of wasted space. In order to avoid that, a different data structure is required.

It was decided to implement the Q-table using the Dictionary data structure, a modified hash table quick consists of keywords and values. The function *GenerateQKeyword* takes the sensor readings and the action choice and creates an integer keyword unique to that combination. This is done by converting a list of the sensor values into binary. This gives six cameras with three readings each, which requires two bits each to encode. Three camera focus views with four readings each require 2 bits to encode. Six sonar sensors with four readings each also require two bits each to encode. Five different action choices require three bits to encode. To make the keyword, each component is multiplied by a different power of two and added together to make a single integer. Because the size of this integer exceeds the maximum positive value for a 32-bit integer, the keyword had to be defined using a 64-bit long integer.

Since the file containing the Q-table must store the keywords as well as the Q-values, the maximum file size increases as well. In addition, because a single keyword is used, there are many keywords which are never encountered, such as any keyword with a value for Action that is not in the range of 0 through 4. The theoretical maximum size for the file holding the Q-table is 64.0 GB. However, experiments have shown that, even after millions of episodes, the size of the file has remained significantly less than one megabyte in size.

3.4.8 Runtime

When the simulator is first started, several initializations are made in the *Arena_Init* function. First, all endpoints of the arena walls, the starting location, and the location of all goal and end of the world markers are rotated five degrees. This prevents the arena walls from being vertical lines, which would result in infinite slopes in many of the calculations in the code.

If the Q-table file and the Q-table backup file don't exist in the current directory, empty files are created. The Q-table is then read into a global variable. At the end of the initialization function, the robot's starting orientation is set to a random angle to ensure the Exploring Starts condition required by Q-learning.

The majority of the runtime of the simulator takes place in *RobotTimer_Tick*, which is called with each tick of the timer. First, it checks to see if manual mode is on and if there is a manual action chosen. Then, the Q-learning process is begun.

The first step in the Q-learning process is to choose an action. A random number between 0 and 1 is generated and compared to ε. If that random number is greater than $1-\varepsilon$, then a random action is chosen. For instance, if ε is 10%, then the random action will be chosen whenever the random number is in the top ε percentile.

If a random action is not chosen, the greedy algorithm is performed. The Q-values for each of the possible actions taken in the current state are compared, and the best Q-value is chosen as the action to be performed. The action is then carried out using the *PerformAction* function. After the action is performed, the *RobotContacting* function is called to ensure that it did not pass through any arena walls.

Once the reward for the action is found, it is added to the current reward tally for the episode and the number of steps in the

episode is increased by one. If the robot is in a bump state, the count of wall hits in the current episode is increased by one. After rewards are assigned, if the robot has reached the goal or passed the end of the world, *SimulationReset* is called to end the episode, write the Q-table to its file, and start a new episode.

All the sensors are then read to find the current state, and the *RobotAtEndOfWorld* function determines if the robot has passed the end of the world. The reward is then calculated based on the state, the action taken, and environmental factors such as reaching the goal, reaching the end of the world, or bumping a wall. After this, the new reward is calculated with the Q-formula (Equation 4) and is added to the Q-table.

$$NewQ = CurrentQ + \alpha(reward + (\gamma * NextQMax) - CurrentQ) \quad (4)$$

A check is then made to see if the robot has to be rescued, if the *Auto_Rescue_Toggle* is turned on. The state is advanced to the next state and the graphical display is refreshed.

The final check is for the clockless timer. If this toggle is activated, then the contents of the timer loop are executed *Clockless_Loop_Count* times immediately. *Clockless_Loop_Count* is a global constant initialized to 100, though it can be changed for faster computers. The shortest interval between timer ticks is one millisecond, so a fast computer can run 100 steps per millisecond when this function is turned on.

3.5 Unsuccessful Trials

Many combinations of reward values were attempted in the process of finding a set of rewards that resulted in an optimal or near-optimal policy. A majority of the first unsuccessful attempts resulted in an agent which was able to learn, but did slow at an extremely slow rate. Under these early trials, the learning agent was not able to successfully find its way to the goal until it had been learning for over one million episodes. However, it still had difficulties avoiding the walls. This resulted in a robot which had learned to grind against the arena walls until it stumbled upon the goal. This sort of learned behavior was unacceptable, since any real robot would be damaged or unable to move properly if it attempted to continuously move against a wall.

3.6 Successful Trials

3.6.1 Reward Schemes

After much experimentation and the addition of the status variables to determine the proper values, a new reward scheme was created. This reward scheme resulted in a learning agent that could successfully reach an optimal policy in a short amount of time. These rewards are assigned based on the status of the robot and on the action chosen which resulted in the robot moving to the state that gives that set of status readings.

In most cases, the status variable *CenterOfHall* was not used. It was implemented under the assumption that the agent would be able to learn obstacle avoidance at an accelerated rate. However, after many iterations of reward schemes, it was observed that the agent was able to avoid walls equally well whether it had a different set of rewards for centering itself in the hallway or not.

In addition, some of these rewards, such as *Reward_Goal*, can never be assigned. There is no instance in which the robot can reach the goal without also seeing the goal in the front camera. This reward was put in for completion, as well as to ensure the code for assigning rewards was consistent.

Reaching the goal will always result in the best possible reward, and going past the end of the world will always result in the worst possible reward. Bumping into a wall also gives a large negative reward. The values of these vary with the different status tiers, but a goal is always good, and bumping a wall or running off the end of the world is always bad.

If *CanSeeGoal*, *GoalInFront*, and *GoalFrontCenter* are all false, then the robot is unable to see the goal (Table 4). Any action it takes is given a small negative reward. Stopping is highly discouraged, so it receives a larger negative reward. These small negative rewards for all actions are to ensure that the robot does not favor wandering aimlessly. The robot should learn to travel down the hall and seek the goal as quickly as possible. If these rewards were positive, the robot would be encouraged to take a less than optimal path toward the goal. With these rewards, the robot learns to travel down the hall as quickly as possible, avoiding the walls, until it sees the goal in any of its cameras.

Table 4. Reward scheme when goal is not seen.

Reward Name	Reward Amount
Reward_Goal	1000
Reward_Bump	-10000
Reward_End_of_World	-50000
Reward_Stop	-25
Reward_Forward	-1
Reward_Reverse	-3
Reward_Turn	-2
Reward_Big_Turn	-2
Reward_Center_Forward	-1
Reward_Center_Reverse	-2
Reward_Center_Turn	-2
Reward_Center_Big_Turn	-2

Table 5. Reward scheme when *CanSeeGoal* is true.

Reward Name	Reward Amount
Reward2_Goal	2000
Reward2_Bump	-5000
Reward2_End_of_World	-30000
Reward2_Stop	-25
Reward2_Forward	2
Reward2_Reverse	1
Reward2_Turn	2
Reward2_Big_Turn	2
Reward2_Center_Forward	2
Reward2_Center_Reverse	1
Reward2_Center_Turn	2

Reward2_Center_Big_Turn	2

Reward3_Center_Big_Turn	7

If *CanSeeGoal* is true, but *GoalInFront* and *GoalFrontCenter* are not, then the robot can see the goal from the side cameras (Table 5). The rewards for this tier are similar to when it does not see the goal, except that all actions, aside from stop, which do not result in a bump or running off the end of the world, are given a small positive reward. This is to encourage the robot to not leave the area in which it can see the goal, thus returning to negative rewards.

If *GoalInFront* is true, then the robot can see the goal in either the left or right focus of the front camera (Table 6). The rewards are similar to those given in the *CanSeeGoal* tier, but the magnitudes for turning or moving forward are larger. This gives higher rewards if the robot chooses a turning action which results in seeing the goal in the front camera.

Table 6. Reward scheme when *GoalInFront* is true.

Reward Name	Reward Amount
Reward25_Goal	4000
Reward25_Bump	-2500
Reward25_End_of_World	-20000
Reward25_Stop	-25
Reward25_Forward	5
Reward25_Reverse	1
Reward25_Turn	5
Reward25_Big_Turn	5
Reward25_Center_Forward	5
Reward25_Center_Reverse	1
Reward25_Center_Turn	5
Reward25_Center_Big_Turn	5

Table 7. Reward scheme when *GoalFrontCenter* is true.

Reward Name	Reward Amount
Reward3_Goal	4000
Reward3_Bump	-2500
Reward3_End_of_World	-20000
Reward3_Stop	-25
Reward3_Forward	10
Reward3_Reverse	-1
Reward3_Turn	7
Reward3_Big_Turn	7
Reward3_Center_Forward	10
Reward3_Center_Reverse	-1
Reward3_Center_Turn	7

The category with the highest rewards for movement is reached when the robot can see the goal in the center focus of the front camera, making *GoalFrontCenter* true (Table 7). A turn which results in seeing the goal in the center of the front camera gives a positive reward, and moving straight while seeing the goal in the center of the front camera results in an even larger positive reward.

3.6.2 Simulated Results

The robot was able to learn very quickly to avoid the walls. Within five minutes of simulated runtime, it learned to turn away from walls if it became too close or if it turned to face one. In this time, the simulator ran through nearly fifty episodes.

It took slightly more time for the robot to learn to avoid the end of the world, but that behavior also came very quickly.

The behavior that took longest to learn was seeking the goal directly. The robot learned quickly that it would gain higher rewards if it was near the goal, but it took much more time to learn how to properly approach the goal point. In the initial testing, however, a near-optimal policy was reached in less than one hour of simulated runtime. After a longer period of time spent learning, the agent was able to further refine its policy. Rather than approaching the goal and moving around it, always keeping the goal in sight, the agent learned to approach the goal, turn directly toward it without hesitation, and move directly to the goal point.

The size of the Q file increased steadily, but the growth began to level off after a few hundred episodes. After an experimental run of 130000 episodes, the file containing the Q-table was still under 500 KB (Figure 15).

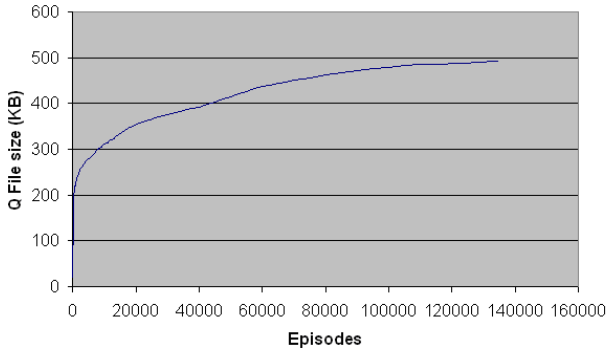

Figure 15. Size of Q-File.

The success rate of the learning process can be measured in several ways. The percentage of episodes ended by reaching the goal and not by running off the end of the world is the first method of measuring success. After 130 episodes, the goal success rate had passed 50%. It was past 85% after 1000 episodes. It passed 90% after 3000 episodes and 95% after 8000 episodes. The final goal success rate after 130000 episodes was 96.1% (Figure 16).

Another measure of success is the number of times the robot bumps into a wall during its run. The agent learned very quickly to avoid hitting walls, due to the large negative reward associated

with a bump. By the end of the first episode, the bump rate was below 25%. By episode 500, the bump rate had fallen below 2% (Figure 17).

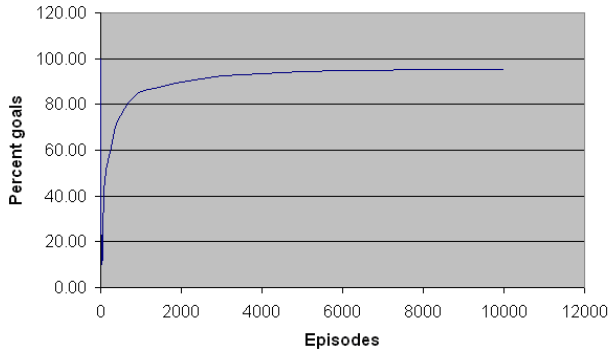

Figure 16. Percentage of episodes ending at the goal.

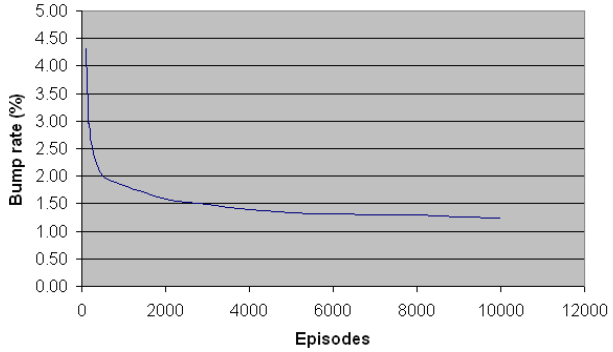

Figure 17. Bump rate per episode.

Another criterion for an optimal policy is to have the highest possible reward per episode. The first few episodes had a very large magnitude negative reward due to a relatively high bump rate. The average reward per episode consistently increased throughout the experimental runs (Figure 18).

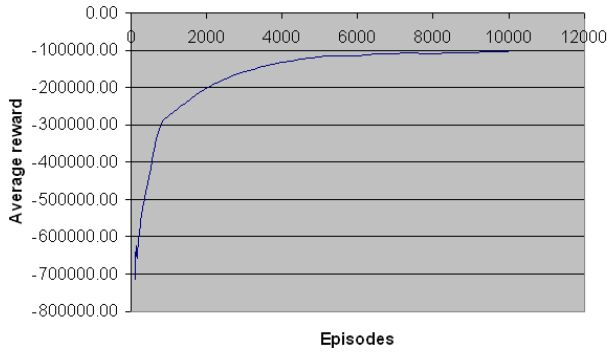

Figure 18. Average reward per episode.

The final measurement of an optimal policy is the length of an episode. The robot should learn to directly approach the goal, not wander aimlessly for a long period of time before finding the goal. The average number of steps per episode decreased steadily and began to level off after 4000 episodes (Figure 19).

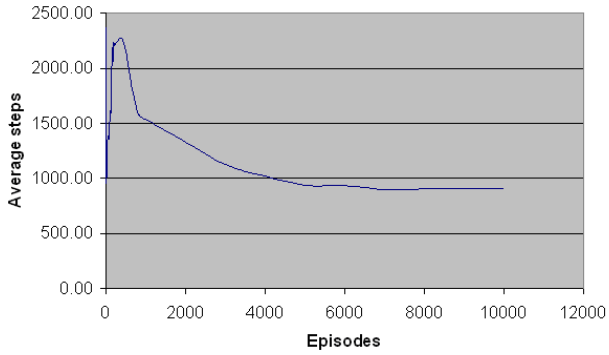

Figure 19. Average number of steps per episode.

4. FUTURE WORK

4.1 Simulator

There are several additions that may be made to the simulator code in order to improve the learning process. While the arena itself is customizable, large arenas make it difficult to observe the robot and the data fields at the same time. A floating control panel containing all the buttons and data readouts would be useful to make the simulator more flexible for various arena map types.

Although it is a purely aesthetic feature, adding functionality to the *Trail On/Off* button would improve the user interface. It would be useful to have a graphical display of the path of the robot during the current episode.

Currently, the learning agent only reacts to the sensor values it sees. If it encounters a set of sensor readings that it has never encountered previously, then that state is treated as an unknown. However, this unknown state may be very similar to another state for which a Q-value is known. For example, two states may be identical save for a single sensor reading. One state may show the goal in the left camera, medium distance on the left sonar, and medium distance on the front-left sonar. Another, similar, state may show the same camera and left sonar reading, but a far distance on the front-left sonar. These states should warrant a similar reaction. A nearness function for using the similarity of two states to adjust the Q-values could improve learning speed. It could also help the agent decide which action to take if it encounters and unknown state that is similar to a known state. This nearness comparison would also likely increase the complexity of the Q-learning algorithm and would require a great deal of experimentation before a it can be determined whether it is a viable and efficient method of learning.

Several other minor additions can be made to the simulator. Adding buttons to allow hypothetical sensor situations to be entered during manual mode would be a good method of checking the fitness of the policy. In addition, adding a function to allow the user to input a position and orientation, either numerically or by way of the mouse on the arena map, would increase the experimental functionality of the simulator.

4.2 Real Robot

Once the Q-table was created by the simulated agent, it was transferred to the real robot. In order to run based solely on the learned experience from the Q-table, the programmed behavior of

the robot was removed. In its place, a decision function based on the Q-table was used.

In general, this decision function was successful. It was consistent with its decisions based on the sensor information. However, some compromises had to be made for the code to be usable on Koolio. Because the vision code only returned the distance to the goal or end of world points and not the location of the points in the camera's view, the new code had to allow for the possibility that any goal or end of the world point might exist in any of the three camera focus divisions. Because of this, any search in the Q-table for a state that includes the goal or end of world points at a close or medium distance had to search the three states which have those three different camera focus sections but otherwise the same sensor information. These three states were then compared, and the state that held the highest value was chosen.

Another difficulty in transferring the Q-table from the simulation to the robot was that the simulator was written in C# while the robot used C. In the simulator code, the Q-table was stored in a data structure called a dictionary, a specialized type of hash table. However, the dictionary data structure does not exist in C. In order to approximate this data structure in C, a linked list was used. Since the matrix formed by the Q-table is always a sparse matrix, there were a manageable number of elements to incorporate into the linked list. In addition, because this code was meant to run on a real robot, speed was less of a factor than it might have been on code meant for use in only software. Searching a linked list was not the most efficient method to implement locating an entry in the Q-table, but it was still accomplished before the mechanical parts of the robot needed the information in order to make its next decision.

The final hurdle was that Koolio had some physical and mechanical problems. At the time of the final testing, the output of the Atmega board did not consistently turn the robot's wheels, making the right wheel move faster than the left wheel. This resulted in Koolio veering to the left when a command was given for it to move forward.

A more difficult error was that the cameras on Koolio began to give many false positive readings. Even though bright colors were used to mark the goal and the end of the world, the cameras still detected up some of those colors, even though they were not present. This led to the distance function believing that the goal or end of the world markers were always visible in all three cameras. Because that is one of the situations which should not exist, there were no entries for it in the Q-table created by the simulator. With no value in the Q-table, there was no learned behavior to follow, and the robot always chose the default action.

5. REFERENCES

[1] Sutton, R. S. and Barto, A. G. Reinforcement Learning: An Introduction. MIT Press, Cambridge, MA. 1998.

[2] Aljibury, H. (2001). "Improving the Performance of Q-Learning with Locally Weighted Regression." Masters Thesis, University of Florida.

[3] Mekatronix. Mekatronix, 2009. http://www.mekatronix.com.