

KOOLIO: PATH-PLANNING USING REINFORCEMENT LEARNING ON A REAL
ROBOT IN A REAL ENVIRONMENT

By

LAVI MICHAEL ZAMSTEIN

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2009

© 2009 Lavi Michael Zamstein

To my parents

ACKNOWLEDGMENTS

I thank my parents for raising me and for encouraging me and supporting me through the difficult times. I thank my rabbi and professor, Dr. Tony Arroyo, for always being there to offer advice. Most of all, I thank G-d, who gives me strength and enables me to do all things.

TABLE OF CONTENTS

	<u>page</u>
ACKNOWLEDGMENTS.....	4
LIST OF TABLES.....	8
LIST OF FIGURES	9
ABSTRACT	12
CHAPTER	
1 INTRODUCTION.....	14
Motivation and Problem Statement.....	14
Proposed Solution	14
Research Environment	15
2 BACKGROUND AND LITERATURE REVIEW	16
Reinforcement Learning Overview.....	16
Markov Models	19
Reinforcement Learning Method Groups.....	20
Dynamic Programming Methods.....	20
Monte Carlo Methods.....	23
Temporal Difference Methods.....	28
Sarsa	29
Actor-critic.....	30
R-learning	31
Q-learning	31
3 ROBOT PLATFORM.....	43
Brief History of Koolio.....	43
Physical Platform	43
Integrated System.....	45
Atmega128	46
Single Board Computer.....	47
Sensors	47
Sonar	47
Encoders	49
Compass	50
Cameras	51
Proximity	52

	PID (Proportional Integral Derivative)	52
	Electrical Hardware.....	53
	Software.....	54
	SBC (Single Board Computer)	54
	Atmega128	54
	Obstacle avoidance.....	55
	Navigation in the Machine Intelligence Lab	55
	Hallway navigation	56
	Door detection	56
	Ordering Website.....	56
4	THE ADVANTAGES OF REINFORCEMENT LEARNING	58
	Learning through Experience	58
	Experience Versus Example.....	60
	Hybrid Methods.....	61
	Learning in Humans.....	62
5	SIMULATOR.....	64
	Reason for Simulation.....	64
	First Simulator	64
	Other Considered Simulators	65
	Simulator.....	66
	Considerations.....	66
	New Simulator	67
	Arena Environment.....	67
	Robot	71
	Sensors.....	71
	Compass.....	71
	Sonar	72
	Cameras	78
	Movement.....	83
	Displays and Controls.....	89
	Learning Algorithm	92
	Q-Table Implementation	94
	Runtime	96
6	EXPERIMENTAL SIMULATION	99
	Unsuccessful Trials	99
	Successful Trials	99
	Reward Schemes.....	99
	Simulated Results	102
	Future Work.....	103
7	EXPERIMENTATION ON THE REAL ROBOT	105

APPENDIX

A	GLOSSARY OF TERMS.....	107
B	LEARNING CURVES	109
C	SIMULATOR CODE	119
D	ROBOT CODE.....	292
	LIST OF REFERENCES	317
	BIOGRAPHICAL SKETCH	320

LIST OF TABLES

<u>Table</u>	<u>page</u>
2-1 Rhetorical dialogue in reinforcement learning.....	17
2-2 The general Q-learning algorithm.....	32
3-1 Atmega128 port assignments.....	47
3-2 Sonar I ² C addresses.....	48
3-3 Port and pin assignments for encoders on Atmega board.....	50
3-4 Compass bearings.....	50
3-5 Files that must be included to enable website ordering.....	57
5-1 Simulators considered but not used.....	66
5-2 Variables used in <i>LineIntersectDistance</i>	73
5-3 Conversion of (Ix,Iy) to polar coordinates (PolarR,PolarTheta).....	74
5-4 Checking whether the intersection is within the arc of the sensor.....	75
5-5 Range values used for Koolio's sonar sensor encoding.....	77
5-6 Range values used for Koolio's camera encoding.....	82
5-7 Status variables in the simulator.....	93
6-1 Reward scheme when goal is not seen.....	100
6-2 Reward scheme when <i>CanSeeGoal</i> is true.....	101
6-3 Reward scheme when <i>GoalInFront</i> is true.....	101
6-4 Reward scheme when <i>GoalFrontCenter</i> is true.....	102

LIST OF FIGURES

<u>Figure</u>	<u>page</u>
2-1	Flowchart of general reinforcement learning. 17
2-2	Tradeoffs of Exploration versus Exploitation. 18
2-3	SMITH's visual model of an object and its input when identifying objects from a scene. 23
2-4	Actor-Critic flowchart. 30
2-5	Fuzzy Reinforcement Logic architecture. 33
2-6	An optimal HDG policy. 40
3-1	Koolio. 44
3-2	Koolio's backpack. 45
3-3	Block Diagram of Integrated System. 45
3-4	MAVRIC-IIB board. 46
3-5	SRF08 board. 48
3-6	S1-50 encoder wire diagram. 49
3-7	CMPS03 circuit board. 50
3-8	Motor control system block diagram. 53
3-9	Koolio's toggle switches. 54
5-1	The first simulator in action. 65
5-2	The TJ Pro. 65
5-3	A simple square arena. 69
5-4	Scale model of the hallway environment. 70
5-5	The locations of the sonar sensors on the agent. 73
5-6	Configuration that will result in a sensor reading of <i>ReallyLongDistance</i> due to the intersection being outside of the sensor's viewing arc. 76
5-7	Multiple detected walls. 77

5-8	Goal (left) and End of the World (right) markers used in the real environment.....	79
5-9	Goal and End of the World markers in the simulator.	79
5-10	Two examples of line of sight to a Goal point.	81
5-11	Sensor displays.....	90
5-12	The sensor HUD.....	90
5-13	Simulation Statistics.....	90
5-14	Q-Value display.....	91
5-15	The main control panel.	92
5-16	The control panel for manual mode.	92
B-1	Size of Q File for first 100 episodes.	109
B-2	Size of Q File for first 1000 episodes.	109
B-3	Size of Q File for first 10000 episodes.	110
B-4	Size of Q File for all experimental episodes.	110
B-5	Percentage of the first 100 episodes that ended at the goal.	111
B-6	Percentage of the first 1000 episodes that ended at the goal.	111
B-7	Percentage of the first 10000 episodes that ended at the goal.....	112
B-8	Percentage of all experimental episodes that ended at the goal.	112
B-9	Average reward per episode for the first 100 episodes.....	113
B-10	Average reward per episode for the first 1000 episodes.....	113
B-11	Average reward per episode for the first 10000 episodes.....	114
B-12	Average reward per episode for all experimental episodes.	114
B-13	Average number of steps per episode for the first 100 episodes.....	115
B-14	Average number of steps per episode for the first 1000 episodes.....	115
B-15	Average number of steps per episode for the first 10000 episodes.....	116
B-16	Average number of steps per episode for all experimental episodes.....	116

B-17	Bump rate per episode for the first 100 episodes.....	117
B-18	Bump rate per episode for the first 1000 episodes.....	117
B-19	Bump rate per episode for the first 10000 episodes.....	118
B-20	Bump rate per episode for all experimental episodes.	118

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

KOOLIO: PATH-PLANNING USING REINFORCEMENT LEARNING ON A REAL
ROBOT IN A REAL ENVIRONMENT

By

Lavi Michael Zamstein

May 2009

Chair: A. Antonio Arroyo
Major: Electrical Engineering

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.

The two primary method groups of machine learning are Supervised Learning, methods by which the supervisor provides training data in order to help the agent learn, and Reinforcement Learning, which requires only a set of rewards for certain choices. Of the three categories of Reinforcement Learning, Dynamic Programming, Monte Carlo, and Temporal Difference, the Temporal Difference method known as Q-Learning was chosen.

Q-Learning is a Markov method which uses a weighted decision table to determine the best choice for any given set of sensor inputs. The values in this Q-table are calculated using the Q-formula, which weighs the expected value of a decision based on the known reward and uses a discounting factor to give more recent choices a greater effect on the values than older choices. The Q-table also allowed the learning to be modular, as a learning agent would only need the file containing the table to be able to use the learned policy generated by a different agent.

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.

After a policy was found in simulation, the Q-table was transferred into Koolio, a refrigerator robot, to allow it to navigate the hallways with the experience gathered in simulation. This Q-table was then further refined through more learning on the real robot.

CHAPTER 1 INTRODUCTION

Motivation and Problem Statement

The Machine Intelligence Lab (MIL) at the University of Florida has been focused on designing and building intelligent mobile robots since 1993. Since then, MIL has designed and built robots as simple as the Talrik Jr. (TJ) or as complex as the SubjuGator underwater submarine.

For years, the lab has had a refrigerator of drinks and snacks available for MIL members. However, this refrigerator had always been immobile. In order to fit with the theme of the lab, the original Koolio robot was created as a mobile refrigerated vending machine. His purpose was to bring drinks to the offices of the professors and graduate students. However, the lab recently moved to a different building, so the software that might have allowed Koolio to move from the old lab to a professor's old office was no longer valid. After the move, Koolio would have to be reprogrammed for the new environment.

Proposed Solution

A new Koolio, a Koolio that could learn for himself and adjust to a new environment, was needed. Therefore, it was proposed to install a reinforcement learning algorithm on Koolio so he could find his way from the lab to the offices even if the building was changed again. Reinforcement learning can enable a robot to write its own path-planning behavior, greatly reducing the need for a human to manually code a new behavior algorithm to adapt to a new environment. After exploring the various categories and specific types of reinforcement learning, Q-learning was the method chosen to use for Koolio.

Research Environment

Because of the repetitive and time-consuming nature of reinforcement learning, as well as the size of Koolio and the potential damage he could cause to himself and others, the initial learning will be performed in simulation. Once an acceptable simulated behavior has emerged, that policy will be ported into the real robot, where learning can continue. Since Koolio will have a learned behavior set from which to start, the learning process on the actual robot should proceed much more quickly and with a reduced possibility of causing damage.

CHAPTER 2 BACKGROUND AND LITERATURE REVIEW

Reinforcement Learning Overview

Reinforcement learning is a method of learning by way of rewards and punishments [29]. The learning agent will seek those choices that result in high rewards and avoid those actions that result in low rewards or punishments (negative rewards). In this way, an agent will learn to follow the decision path that results in the best possible reward.

Reinforcement learning is different than supervised learning. In supervised learning, examples of goal behavior are given to the learning agent before the learning process begins. Reinforcement learning, on the other hand, supplies no prior knowledge to the learning agent of what is good or bad. The agent learns as it goes, being told as it makes action choices whether a behavior is desired or not.

One application of reinforcement learning is in the behavior of robots. As a generalized example, this chapter will explain reinforcement learning methods using a robot that is learning to wall-follow while avoiding obstacles and not touching the wall itself, using simple infrared (IR) emitter/detector pairs and bump sensors (switches).

In the general reinforcement learning model, an agent receives information from the environment via sensor input and interacts with the environment with an action, or possible output [12]. An agent is defined as the self-contained whole of what is learning. An agent has complete and perfect control over everything that happens within itself. Therefore, a robot is not an agent, since the robot does not have perfect control over such problems as sensor noise or imperfect movement. The agent in a mobile robot is the artificial intelligence itself. The robot platform is considered part of the environment, since it cannot be completely controlled by the agent.

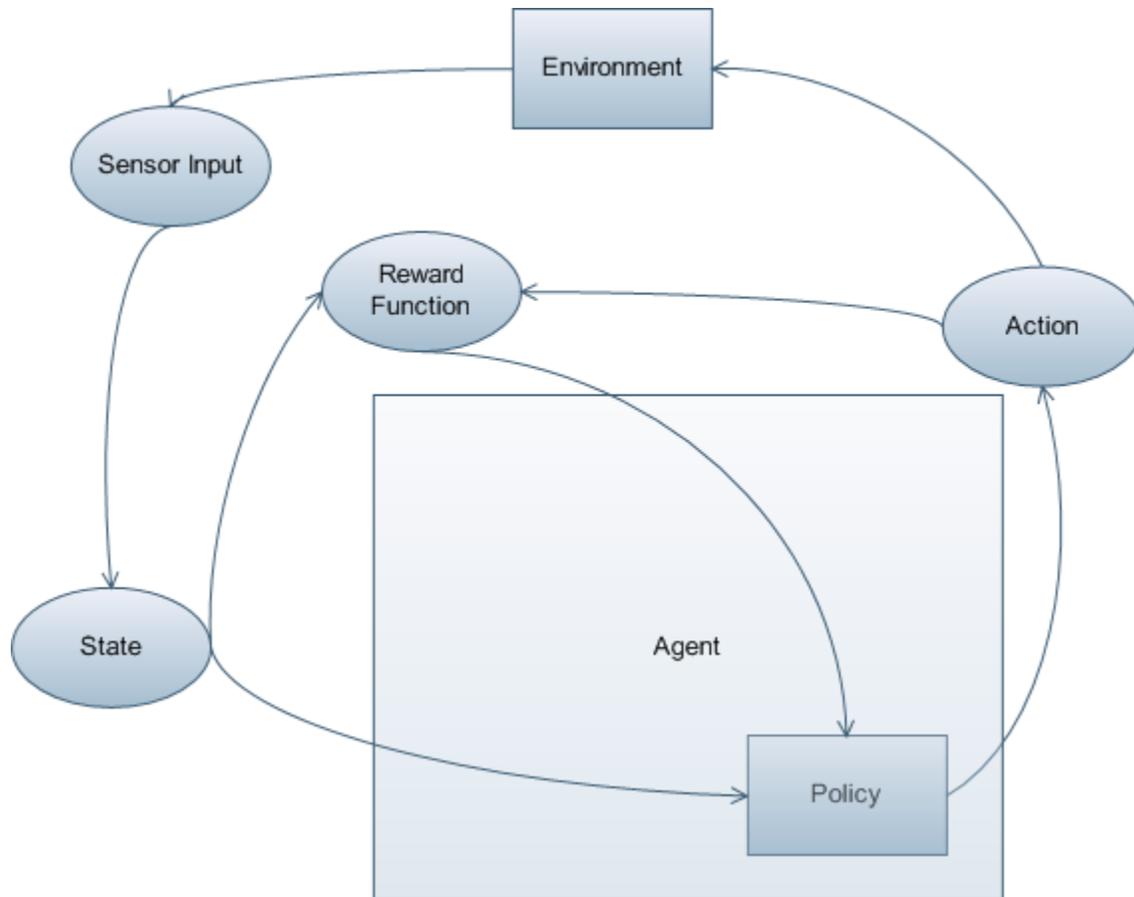


Figure 2-1. Flowchart of general reinforcement learning.

In the case of a mobile robot, the state is the set of all sensor inputs. The learning agent takes that state and is given a reward based on the state and the action taken to reach that state. The reward is used to update the policy. Based on the current state, the policy decides the best action and takes it, resulting in a new state. Figure 2.1 summarizes the reinforcement learning structure.

Table 2-1. Rhetorical dialogue in reinforcement learning.

Environment:	You are in state 74. There are 5 possible action choices.
Agent:	I take action 3.
Environment:	The reward for action 3 in state 74 is 12. You are now in state 45 There are 4 possible action choices.
Agent:	I take action 1.
Environment:	The reward for action 1 in state 45 is -7. You are now in state 97. There are 6 possible action choices. [12]

The rhetorical dialogue in Table 2-1 presents a simplified form of the interaction between agent and environment.

One of the fundamental balances in reinforcement learning is between exploration and exploitation. Since the agent wants to maximize its reward, it will often make the choice that offers the highest immediate reward for that state. However, if this short-term best choice is always made, then many other possible choices are never encountered. It is very possible that some of these choices never observed may have an even better long-term return than the current best choice, just hidden beneath lesser rewards. To avoid the skipping of such cases, there must be some exploration of non-optimal states, in addition to exploitation by picking the current best choice.

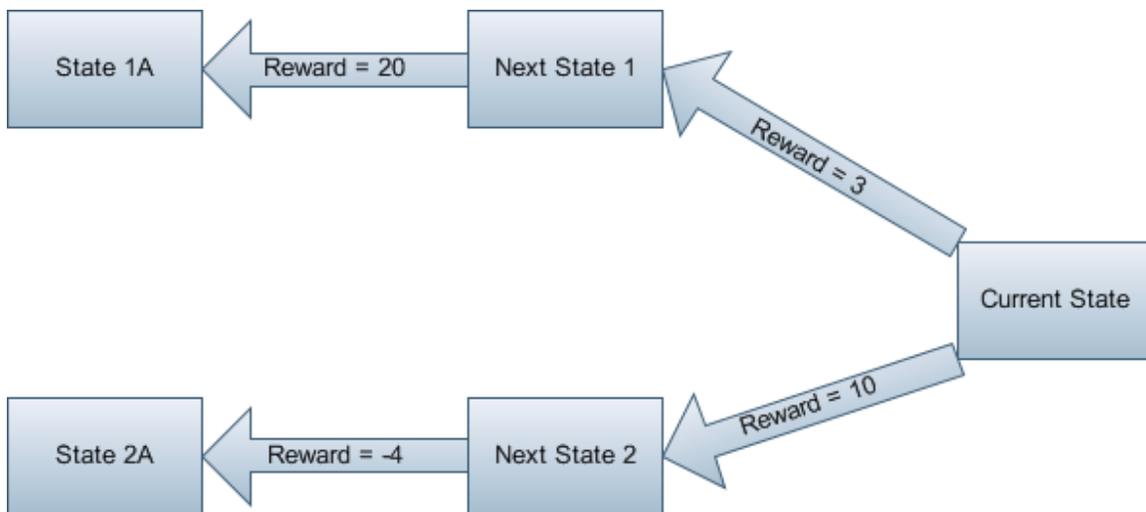


Figure 2-2. Tradeoffs of Exploration versus Exploitation.

Figure 2-2 displays the difference between exploration and exploitation. From the current state, the agent can choose from two different actions. The second action, leading to *Next State 2*, yields a higher reward of 10. The policy may therefore decide that this choice is currently the best. Following from that choice leads to a reward of -4 on the next action, for a total reward of 6 for the choices. This is exploitation, following the current best choice.

Once in a while, the agent makes a random choice that may not be optimal in the short term. If it decides to explore, it randomly selects an action and chooses the action that leads to *Next State 1*, giving a reward of 3. The following state comes with a reward of 20, for a total reward of 23 for those two states. The agent has found a choice that gives a higher reward in the long run, even though the initial reward is less. The policy is therefore updated to choose *Next State 1* as the best choice.

If there were never a chance of picking a random action, *Next State 1* would never be encountered. Exploration allows a chance for that state to be chosen at random, thereby leading to an overall higher reward. If the learning agent always exploited its current best choice, then choices such as *Next State 1*, with greater rewards available in later states, would not be chosen, and the optimal reward would never be reached.

Most reinforcement learning methods use an algorithm called ϵ -greedy. In this algorithm, ϵ is some small number that describes the probability of choosing a random action. For many forms of reinforcement learning, an ϵ of 0.1 is selected. Whenever the time comes to select an action, there is a $1-\epsilon$ chance of taking the current best choice as dictated by the policy. This is exploitation. There is a ϵ chance of taking a random choice, or exploration. This allows for choices which are not currently the best, but which may lead to policies that prove optimal. Using an ϵ of 0.1 allows for selecting the current best choice 90% of the time and randomly selecting an action choice 10% of the time.

Markov Models

A Markov process is any process that uses only current values of inputs to determine the outputs. All past input values are ignored in Markov processes. This makes Markov processes very useful in the realm of robots, since the outputs and all calculations will rely only on the

current input of the sensors, rather than any past inputs. Not only does this remove the necessity of memory for past sensor values, but it also greatly reduces the necessary calculations, as there are fewer variables; only the current sensor values instead of the current and many past sensor values.

Reinforcement Learning Method Groups

There are three major groups of methods in reinforcement learning, though each of the methods may be further divided into more specific methods. These three main methods are differentiated from one another by two main factors: models of the environment and bootstrapping.

Dynamic Programming (DP) methods bootstrap and require a complete and accurate model of the environment.

Monte Carlo (MC) methods do not require a model of the environment and do not bootstrap.

Temporal Difference (TD) methods do not require a model of the environment but do bootstrap.

Dynamic Programming Methods

Because DP methods require an accurate model of the environment, they are not suitable for reinforcement learning with robots. This is because the environment is the real world around the robot, and therefore is very difficult, if not impossible, to model perfectly. In addition, if the environment were to change, the entire model must be changed, which will severely hamper the learning process.

On the other hand, since MC and TD methods do not require an accurate model, but only a general idea of the environment, changing that environment has much less impact on their methods of learning.

Because DP methods are not efficient for use in reinforcement learning for robots, they are not used in favor of MC or TD methods.

Dynamic Programming methods have been used successfully when it is safe to assume that a perfect model of the environment can be made. A recursive DP method was used to optimally schedule autonomous aerial refueling (AAR) of multiple non-identical unmanned aerial vehicles (UAVs) [10]. If there was only a single aerial refueling tanker with multiple UAVs in need of fuel, a task schedule was required for order of refueling. This sort of scheduling is classified as an NP-hard problem, making it difficult to come up with an efficient solution. In addition to choosing the order of refueling based on need, the algorithm also had to deal with the limitations of the UAVs. Unlike many scheduling problems, UAVs were not able to repeatedly change order in midair. Such a shuffle of order involved complex maneuvers on the part of all UAVs in the queue. Therefore, the algorithm sought to make an efficient schedule with a minimum number of order reshufflings. The algorithm used assumes that only one UAV is shuffled at a time and that there is not a dynamic cost for different UAV maneuvers. Each of the UAVs had its own parameters, including the maximum time it can wait given its fuel level, refueling time, priority, and current location in the refueling queue. Because the time that a UAV was able to remain in the air changes as time goes by, the DP scheduling had to be dynamic to allow for changing fuel levels. Therefore, while return-to-field priority was always important, it was more important for the schedule to keep a UAV from crashing than to allow higher priority vehicles to be finished sooner. Since there had to be a minimum number of reshuffles, a metric was calculated for the similarity between different possible UAV sequences, which became the queue reconfiguration cost. This algorithm can also be used for other scheduling problems that seek to minimize the cost of re-ordering the queue of items to be serviced.

DP has also been used for vision algorithms to identify objects by shape [15]. In order to minimize the computation time of shape identification, an algorithm was developed that was not model-driven. In other words, it did not take the input and compare it to a database of examples. Instead of doing an exhaustive comparison search, a hybrid between data-driven and model-driven methods was used. First, the presence of one or more identifiable objects was hypothesized. This reduced the number of models that had to be retrieved from the database, since only those objects hypothesized were taken for the search. Then, these selected models were used to do a more thorough analysis. This indexing mechanism reduced the time needed for the recognition phase. This mechanism was used in the two-dimensional object recognition system called SMITH (Shape Matching Utilizing Indexed Hypothesis Generation and Testing). Models for the database were made by taking pictures of objects in grayscale, running a binary filter to remove noise, and making a polygonal approximation of the outline of the object. This polygonal approximation became the model.

In Figure 2-3, SMITH detected that the string of line segments from vertex 6 to vertex 21 compose a shape very similar to the model for a wrench. Therefore, it made a hypothesis that the scene contained the wrench object and pulled the wrench model from the database of models for further comparison. SMITH computed that the top object in the scene was indeed the wrench, with a scaling factor of 1.10, a rotation of 15.94° , and a translation of 12.36 in the x direction and -17.75 in the y direction. In addition to wrenches, pliers were also used in the testing. However, no tests were shown with dissimilar opening angles of the pliers, so conclusions cannot be drawn on how SMITH identified a pair of pliers closed fully or open significantly beyond the model. It is possible that dynamic objects such as pliers had multiple models for the multiple shapes they could take on depending on how much they were opened.

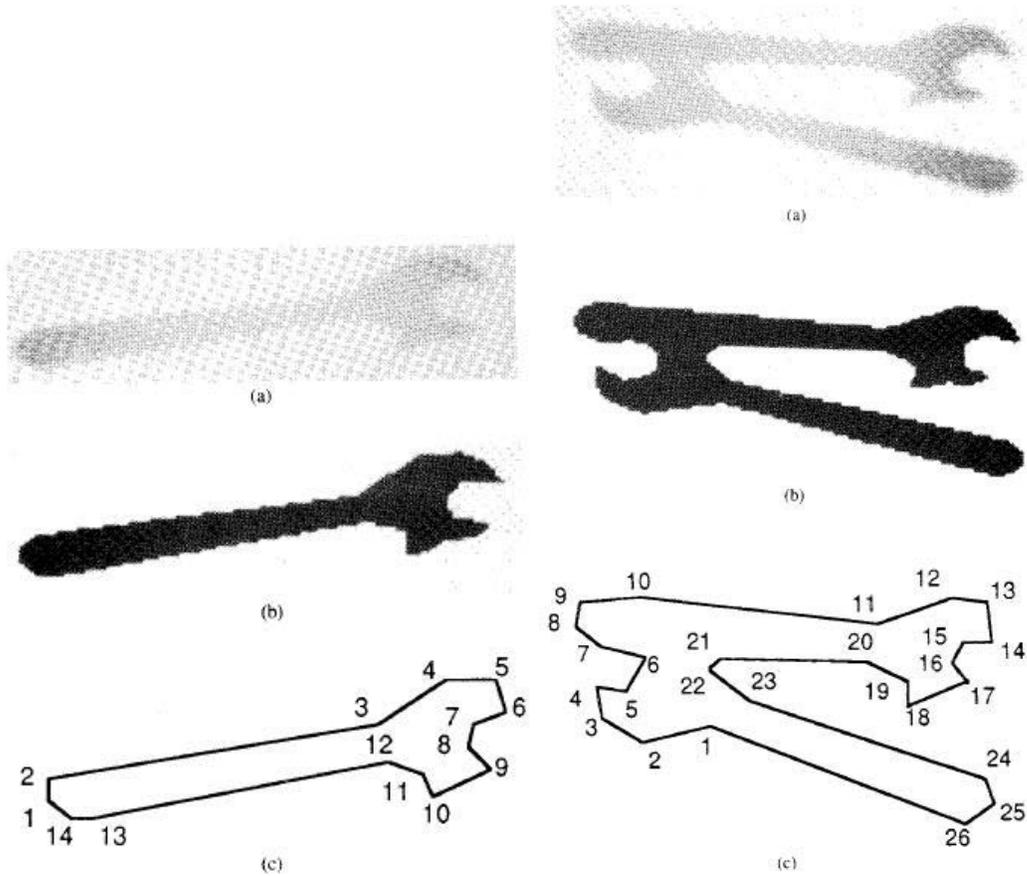


Figure 2-3. SMITH's visual model of an object and its input when identifying objects from a scene. a) Gray-level image. b) Threshold image. c) Polygonal approximation.

Both of these systems were able to use Dynamic Programming because the environment for the learning did not change. A UAV with enough fuel remaining for ten minutes of flying will have that amount of time airborne each time it is encountered. A UAV that takes ten seconds to be refueled will always take ten seconds to be refueled. A wrench or a pair of pliers encountered on a flat surfaced and photographed with the same camera will always have the same type of shape. If wrenches were dynamic objects that changed shape significantly, then the environmental model would be dynamic and would not be usable with DP methods.

Monte Carlo Methods

Because MC methods do not require a complex model of the environment, they are much more fit for use in robot learning. The environment model for MC need only generate sample

state transitions. In other words, the model only needs to define the states and their possible actions, and how the agent can move from one state to another.

This in itself makes MC methods usable for robot learning, as well as minimizing the effects of a changing environment. In the example of wall following and obstacle avoidance, for example, the model must have states defined for the various possible conditions in which the agent may find itself. Some of these may be *nothing*, where neither the IR nor the bump sensors are active, showing the robot is in a clear area, *wall*, where the IR sensors detect some obstacle on one side, and *contact*, where the bump sensors indicate the robot has touched an object. For a more complicated model, one could add states for corners, dead ends, and tight spaces. Using the basic states of *nothing*, *wall*, and *contact*, however, a robot can successfully learn to follow walls without bumping into them.

Because, unlike DP methods, MC and TD methods do not require a model of the environment, they are much more flexible. If a robot were to learn the wall-following and obstacle avoidance behavior as desired, then was moved into another room with similar makeup (walls and solid objects), either in the middle of the learning process or after the learning is complete, it would be able to perform with the same policy.

However, if the substance of the environment were changed, there would be a problem. For example, if an obstacle was introduced that could be seen with IR sensors but was light enough to be pushed by the robot without triggering the bump sensors, that would represent a fundamental change in the state makeup. Since the ‘normal’ wall/object definition is something that is solid and causes the bump sensors to activate when hit, this new object is completely alien to the agent’s knowledge and will serve to make the learning process more difficult, if not impossible, in the changed environment.

Because MC methods do not bootstrap and use average sample returns, they are defined only for episodic tasks, since the value estimates and policy are updated only at the termination of each episode. Thus non-episodic tasks must use TD methods for learning.

MC methods use a cycle of evaluation and improvement to implement policy iteration. For each instance of this cycle, the state-action functions are evaluated for the current policy. The best of these actions for each state are selected to make the new policy. This new policy can then be evaluated next episode to repeat the cycle. The greedy method is often used in the policy improvement, resulting in a new policy that is either of equal or greater value than the previous policy. Given an infinite number of episodes and an assumption of exploring starts, this method is guaranteed to converge to an optimal policy. Realistically, however, there will never be an infinite number of episodes. To bypass this assumption, the evaluation step of each evaluation and improvement cycle may be changed so the value function approaches the state-action value function but does not necessarily reach it immediately. In other words, the policy need not immediately jump to the new better value, but may simply proceed toward it. If the next policy is better and has the same state-action pair, then the improvement will approach it further. If not, there is much less backtracking needed to move toward a new choice.

Since a good model of the environment is never assumed for MC methods, state-action values become more important than state values, since they are possible to estimate without values of future events, whereas state values require knowledge of the following states in order to estimate them. Because of this, though, it is possible for direct policy evaluation to miss some actions in a state. In order to avoid this, exploration must be maintained in a method called exploring starts. Exploring starts gives every state-action pair a non-zero probability of being the first step in each iteration.

In order to use exploring starts in our example of a robot learning the wall-following behavior, the robot must be placed in a random location in the environment at the start of each episode. This also means that it must sometimes be placed near a wall to fulfill the state of sensing a wall nearby, as well as touching a wall, to explore the state of bumping into a wall. From these random positions, a random action must be picked of the possible actions from those states. All this will ensure that exploration is maintained and no state-action pairs are skipped.

This method of ensuring exploring starts, however, is tedious and not likely to fully explore all the possible states and actions as initial steps. There are two other methods used to ensure full exploration: on-policy MC and off-policy MC.

On-policy MC methods use first-visit methods, which use the average of returns after the first instance of a given state to determine the state-action pair's value function, to estimate the current policy's state-action value functions. Because a full greedy algorithm for improving the policy would miss some state-action pairs without the assumption of exploring starts, ϵ -greedy is used instead. This ϵ ensures that policy improvement approaches an optimal policy while still maintaining exploration.

Off-policy MC methods separate the functions of control and policy evaluation into two separate functions: the behavior policy to govern the decisions of the current policy and the estimation policy to be evaluated and improved. Because the policy functions are separated in this way, ϵ -greedy is not needed, and greedy is sufficient for exploring all possible state-action pairs, since the estimation policy may continue to explore while the behavior policy acts. The tradeoff for not needing ϵ -greedy, however, is that these methods learn faster for selecting non-greedy actions than for selecting greedy actions, resulting in slow learning for some states, especially early states in long episodes.

With our wall-following robot example, off-policy MC would remove the random aspect of on-policy; however, it would see a decrease in learning speed, especially for longer episodes.

Monte Carlo methods have been used in estimating the probability of a rigid polyhedral object landing in different stable positions when dropped [9]. This was used to find the probability of each possible stable position of a part with known center of mass on an assembly line to help automation. The MC method used was compared to two other models: a quasistatic motion model and a dynamic stability measurement based on perturbation. These estimators were to be used for the design of a parts feeder so a robot arm with a gripper could easily move to pick up the part in the correct orientation. In order to perform the calculations for the MC method, a mechanics simulator package called Impulse was used [17]. This simulator package used a method to determine position based on natural frictional collisions. The experiments conducted using the three estimators found that the MC estimator with impulse-based mechanics gave the most accurate predictions for all of the parts tested. However, it was concluded that it was unsuitable for use, since its intensive calculations took a much longer amount of time than the other two methods: up to two hours compared to under a second for the perturbed quasistatic estimator. The MC dynamic estimator would be more useful in analysis of the robot design, but was not suitable for use in an assembly line environment, where the robot arm would require the estimation as quickly as possible.

When a robot is introduced to a new environment, a map of that environment can always be useful. Creating that map, however, is a difficult task. A probabilistic topological map (PTM), made of nodes of important locations connected by pathways, can be made using Markov-chain Monte Carlo (MCMC) sampling methods [20]. By passing a Markov chain over the state space of the map choices, the MC method is guaranteed to converge to the topological

node distribution desired for the map. By giving a negative reward scaled to the difference in the topological maps, the MCMC method encourages the map to converge into a single set of correct node configurations. Various reward magnitudes were attempted, and it was discovered that a negative reward of large enough magnitude would outweigh the odometry measurements, effectively swamping them out and creating absurd topological maps. Therefore it was important to not make the negative rewards for non-convergence too large in comparison to the rewards for proper odometry measurements. Through the use of these MCMC methods, the robot was able to construct a PTM using a minimal amount of prior information and scanning. The methods showed that, while there was no ideal mapping algorithm, using PTMs allowed for probabilistically reasoning and flagging any errors or uncertainty encountered during mapping.

Temporal Difference Methods

Like MC methods, TD methods also do not need a model of the environment, and also make predictions in the same way, using sample state transitions. Unlike MC methods, TD bootstraps, using estimates to update other estimates. Because of this, TD methods update values after each time step, rather than after each episode. Bootstrapping allows for TD methods to learn faster than MC methods during exploration, since the policy update each time step means the method does not need to wait until the end of an episode to determine if that choice was better or worse than the choice indicated by the previous policy. TD methods can then know whether an explored choice is good or bad right away, rather than waiting until the end of the episode.

The use of bootstrapping is the major difference between TD and MC methods. Many other parts are the same, such as the use of on-policy and off-policy methods, though the results may be different because of the quicker learning under exploration with TD.

TD learning also has several special types of methods, both on and off policy, which are useful in certain cases.

Sarsa

Sarsa, an on-policy TD method, updates after each transition from a non-terminal state. Because of this, Sarsa methods always converge to optimal policies so long as all the state-action pairs are visited an infinite number of times and the policy is able to converge to the greedy function. The name Sarsa is formed from an acronym that shows the order in which a Sarsa method views values: State, Action, Reward, State, Action.

Unlike MC methods, Sarsa and other TD methods are able to determine during an episode whether the policy is good or bad, and change policies if the current one is determined to be bad. This proves very useful in episodes where the current policy may never finish, from an inability to reach the goal.

In the wall-following robot example, if the goal was to spend more time along a wall than in the open, some policies may never reach that goal. For example, a policy that turns away from a wall as soon as it is spotted will always spend more time away from the wall than against it, resulting in a never-ending episode. Using Sarsa or other TD methods avoid this by determining in the middle of the episode that the choices defined by the policy are not able to reach the goal.

Comparisons have been made between Sarsa methods and the more popular Q-Learning methods under two different policy architectures: GLIE and RRR [26]. Policies that are GLIE (Greedy in the Limit with Infinite Exploration) make two primary assumptions. First, GLIE policies assume that, if a state is visited an infinite number of times, then every action in that state will also be chosen an infinite number of times. GLIE policies also assume that the learning policy becomes greedy with a probability of 1 as the steps become infinite in number.

This second assumption must be made for Sarsa to converge, since it is an on-policy TD method. It was shown that the difference between the current and the optimal policy will always converge to 0 under GLIE learning policies. RRR (Restricted Rank-based Randomized) learning policies set the probabilities of choosing a given action based on ranks of the current value functions, with greedy being given the highest probability and the choice with the lowest Q value receiving the lowest probability of being chosen. This takes the place of ϵ -greedy by making tier-based probabilities rather than random selections. RRR learning policies were also proven to converge to optimal policies under Sarsa methods. Although the inclusion of probability ranks in an RRR policy makes the action selection process more computationally complex, it still converges to an optimal policy.

Actor-critic

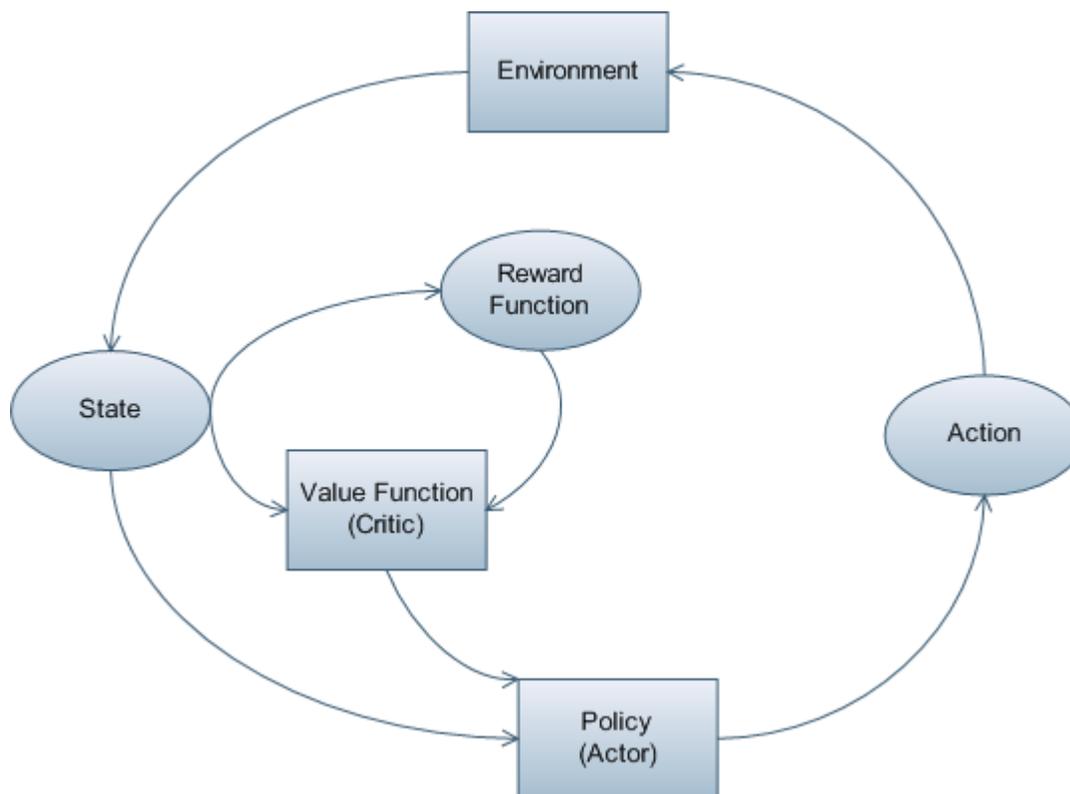


Figure 2-4. Actor-Critic flowchart.

Actor-critic, a group of on-policy TD methods, separates the policy and the value function into independent memory structures. The policy structure, or actor, is used to decide which action to pick in each state. The estimate value function, or critic, determines whether the actions of the actor are good or bad, and whether they should be encouraged or discouraged.

These methods are useful because they do not require much computation to select actions, due to the policy and value functions being stored and operated independently. Because of the division between decision making and decision critiquing, actor-critic methods are also useful in modeling psychological or biological behavior, as such behavior functions under the same general separation structure. Figure 2-4 illustrates the actor-critic method.

R-learning

R-learning, an off-policy TD method, does not discount past experiences, unlike most other learning methods. R-learning also does not divide experiences into episodes, instead learning from a single, extended task. This shifts the priority for the optimal policy to optimizing each time step, rather than optimizing each episode. R-learning methods use relative values, state-action functions that determine their value based on the average of all other state-action values for the current policy. With relative values, the value of each action chosen is compared to the overall average of all other values in the policy. If the chosen action is better than the average, it is considered good and is incorporated into the policy. If the action returns a value worse than average, it is considered bad and discarded from the policy.

Q-learning

In Q-learning, the action-value function, denoted as Q , is able to approximate the optimal action-value function, Q^* . Since Q-learning is an off-policy method, the Q function can grow closer to Q^* independent of the current policy that is being followed by the agent. This simplifies policy evaluation and updating while only making the assumption that all state-action

pairs continue to be updated. Unlike Sarsa, Q-learning does not need to assume that state-action pairs are visited an infinite number of times, making it more likely to reach the optimal policy than Sarsa given a finite number of episodes. With this assumption, Q will always converge to the optimal policy Q^* .

The Q-learning algorithm consists of only one major formula, used to update the learned action-value function. To begin Q-learning, first initialize the value functions for every state-action pair to some arbitrary value. The algorithm is repeated for each episode. For each step in the episode, the agent chooses its action from the set of possible states using its current policy and a method such as ϵ -greedy to ensure exploration. Once the action is taken, the reward and the next state are used in the Q-learning equation to update the value function for that state-action pair. The updating process is repeated for each step until a terminal state is reached.

Table 2-2. The general Q-learning algorithm.

```

Initialize all  $Q(s,a)$ 
For each episode:
    Initialize  $s$ 
    Repeat for each step in the episode:
        Choose  $a$  from  $s$  using the policy
        Observe  $r$  and  $s'$  after taking action  $a$ 
         $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$ 
         $s \leftarrow s'$ 

```

The general Q-learning algorithm can be seen in Table 2-2. In this algorithm, $Q(s,a)$ is the state-action value function for taking action a when in state s . Using ϵ -greedy and the current policy, the action a is chosen from the possible actions to take from state s . This yields a reward r and leads to the next state s' . The value function is then updated by adding a factor to it. This factor comes from looking ahead to the estimated return from the next action taken. From the next state s' , the maximum state-action value $Q(s',a')$ is taken and multiplied by the discounting factor γ . The recent reward r is then added and the recent value function $Q(s,a)$ is subtracted.

This sum is then multiplied by the step size parameter α and added to $Q(s,a)$ to get the updated state-action value function $Q(s,a)$. The state is advanced and the cycle continues until the terminal state at the end of the episode is reached.

$Q(\lambda)$ -learning, a modified Q-learning technique using eligibility traces, utilizes fuzzy logic in reinforcement learning systems. These fuzzy reinforcement learning (FRL) methods allowed for a blending of reinforcement learning and expert systems using prior knowledge [8]. Instead of updating the state-action value function only from the reward earned by the choice of action, the FRL algorithm also had an order-0 Takagi-Sugeno FIS feed information to the update function. The FRL learned choice was taken along with prior knowledge to make the decision on which action to take in each state. Figure 2-5 displays the architecture of the FRL system.

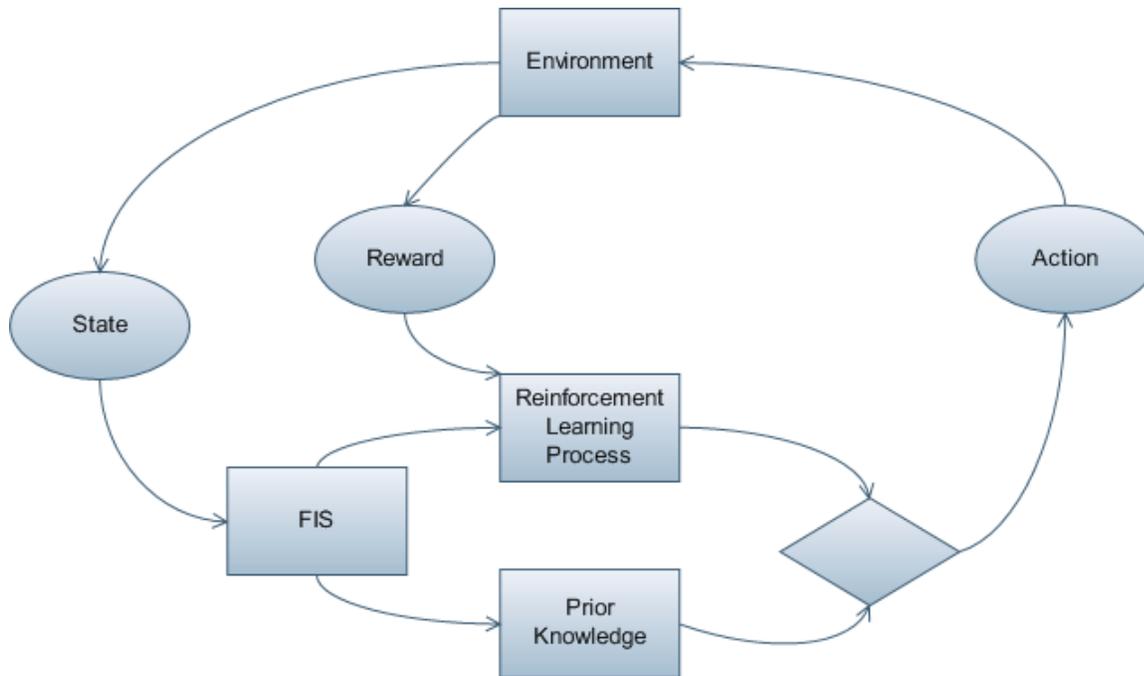


Figure 2-5. Fuzzy Reinforcement Logic architecture.

This FRL algorithm was used to enable a robot to learn obstacle avoidance and pathing behaviors as it moved toward a goal area. Because the reward system in the robot used was based on obstacle avoidance while moving toward a known goal, with a negative reward as a

penalty whenever the robot moved away from the goal, such obstacles as concave walls became effective traps. In order to escape such a trap, a wall-following behavior was added to the set of actions. When the robot recognized that it was in a concave trap by finding two key points, the FRL indicated that it was in such a trap and the robot would be rewarded for using wall-following behavior until the concave wall had been bypassed. By learning when to switch behaviors between tracking toward the goal and wall-following, the robot was able to navigate a wide variety of obstacles and mazes to reach the goal.

Q-learning has also been used on the TJ Pro robots that became a model for the simulation program explained in Chapter 3. The rewards were set so the robot would prefer moving straight forward and be penalized for bumping into an obstacle. A lesser negative reward was given for having to go into reverse [3]. With the relatively simple task of obstacle avoidance and the simple sensor suite of only two IR sensors and a bump ring, Charm, the TJ Pro robot used, showed obvious learned behavior within ten minutes of running the learning process. During the learning process, Charm obtained certain behavior quirks. For example, the preference when choosing direction when rotating to face away from an obstacle seemed to ‘stick’ with it throughout the three-hour run until it was reset. Also, Charm had a tendency to follow up a soft turn to avoid an obstacle with an additional pivot. In looking at the sum of the Q-table values versus time, it became apparent that the learning process was not able to converge completely within each three-hour run. It is possible that the policy would have converged given more time, but longer runs were not performed. This is one reason why simulation can be important to the learning process of a real robot. Simulations are not limited to real-time, so an agent in simulation can run the learning algorithm for extended periods of time that can translate into real robot experiments that would take orders of magnitude longer to perform.

A second learning type, H-learning, was also attempted with the TJ Pro. H-learning, while more aggressive in its learning approach, was shown to have one major flaw in the use of real robots. Because it had no discounting factor, it was possible to become stuck in an area of either very high positive rewards or very large negative rewards. This resulted in increased computational complexity. While the Q-learning was able to be performed using only the 256 bytes of RAM on a TJ Pro, H-learning demanded more memory. When comparing the two learning methods, it was shown that H-learning does converge to an optimal policy more quickly than Q-learning. However, because of the issue of memories and scaling, H-learning was shown to not effectively scale to agents requiring more complex controllers.

In order to allow the agent to function best in a real environment, it must operate within a real robot and learn in a real environment. While simulation can be useful, it is impossible to perfectly model and simulate the real world. As such, porting a learned policy from a simulated learning agent directly into a real robot with no further learning is unlikely to work optimally. A hybrid reinforcement learning system was created [34] to allow a learning agent to use both the more accurate real environment and the much faster and less computationally costly simulated environment. This hybrid process was used to allow a robot to learn different tasks involving interaction with a ball, such as following the ball and pushing it along. By allowing the simulated agent and the real-time agent to cooperate in the learning process, this hybrid method covers the weaknesses of both agents. The simulated agent can not account for real world irregularities such as uneven motors, imperfect surfaces, and unexpected friction, while the real world agent deals with such things. On the other hand, the real robot is extremely slow relative to the simulated agent, so the simulation adds to the speed of learning. In addition to the hybrid system, sub-tasks were set up. Each sub-task had its own sub-reward, so completing a single

sub-task gave a higher reward than working toward several sub-tasks but not completing any. In later experiments [33], the simulated agent and the real robot cooperated through the process of adaptive mimetism. This process helped the sharing of learned behaviors between the slower real robot and the much faster simulated agent. Adaptive mimetism allowed select behaviors to be propagated from the simulation to the real robot, which allowed for accelerated learning in the desired behaviors of the real robot. This also prevented the real robot from becoming over-specialized from the propagated behavior of the simulated agent.

Q-learning has also been used in multiple robot systems. It was implemented in a multi-agent system seeking to minimize the communication necessary between the individual robots [25]. If communication is kept to a minimum, then less overhead is required for things such as handshaking and data verification and error checking. With each agent independently using Q-learning methods to move about its environment, a global coordination behavior began to emerge even without the explicit or implicit transmission of data. Four agents operating simultaneously in a grid world learned to move toward the goal point. The agents were given small positive rewards for each step that moved them closer to the goal and small negative rewards for each step that moved them away from the goal. Relatively larger magnitude negative rewards were given for a collision with another agent. Because of these collision rewards, the agents learned to avoid one another while moving through the grid world without needing to communicate their presence with one another. A classifier-based learning method was also used. This classifier-based method was able to produce results which were, although not as good as the optimal Q-learning policy, near-optimal, and required less time to find these near-optimal policies compared to Q-learning's optimal policies.

Other implementations of multi-agent systems have been implemented with Q-learning. A team of robot soccer players was able to learn both offensive and defensive behavior through global and local reinforcement learning [4]. Each individual robot ran its own Q-learning algorithm, but the rewards were based both on that individual robot's actions and on the actions of the team as a whole. For the global reward, each agent on the team was given a reward of 1 when its team scored a goal and a reward of -1 when the opposing team scored a goal. Local rewards gave a reward of 1 if the agent was the closest one to the ball when a goal was scored and a reward of -1 if it was the closest to the ball when the opposing team scored a goal. In this way, the entire team is rewarded for a goal and given negative rewards for giving up a point, and individual agents are given positive and negative rewards for being directly responsible for scoring a goal and giving up a goal, respectively. For initial testing purposes, a hand-coded control team was made to play against the learning team for 100 10-point games. When using only the global rewards, the learning team averages 6 points to 4 for the control team. However, when using only individual rewards, the learning team averages 4 points to the control team's 6. When only individual rewards were used, the agents each tended to learn the same behavior, resulting in a homogeneous team. With the global rewards, the individual agents began to specialize into a heterogeneous team, with one becoming a goal defender and another becoming a primary scorer. The local reward methods had one advantage over the global reward method in that it converged much more quickly to optimal policies. While the global reward method resulted in better performance, the agents were often still altering their policies and had not yet reached an optimal policy by the end of the 100 games.

A similar comparison of independently-emerging heterogeneous behaviors in multi-robot learning systems was done using multi-foraging [5]. Multiple color-coded types of objects, or

attractors, were present for the robots to search for and bring back to a goal area, dependent on the type of attractor. Three different reward functions were used. In the global reward function, all agents were given a positive reward whenever one robot returned an attractor to its goal area. In the local reward function, each agent was rewarded individually for the attractors it returned. The local shaped reward method gave rewards to each agent individually as that agent performed segments of the required task, such as picking up an attractor, moving to the goal area, and dropping the attractor in the correct goal area. After trials on several randomly-generated environments, it was found that the global reward system for this task performed poorly compared to the others. The local reward and the local shaped reward resulted in very similar performance, with both of them having very few difference in performance from the control group of hand-coded robots. When comparing the diversity of the agents' individual performances, the global reward method produced the most heterogeneous agents. Compared with the team of soccer robots [4], multiple foraging robots gave opposite results. When learning a team activity, such as playing soccer, there is a direct correlation between diversity of individual agents to make a heterogeneous team. However, when performing individual tasks such as foraging, diversity proved detrimental and even had an inverse correlation between diversity and performance. It was shown that whether a team of robots should be homogeneous or heterogeneous depended on whether the task to be completed required a team effort or if individual agents acting alone would perform better.

Q-learning backed by learning from demonstration was used on balancing an inverted pendulum on a cart movable in two dimensions [24]. It was discovered that Q-learning allowed the balancing problem to converge to an optimal policy within 120 seconds. However, in order to begin learning at all, an initial stabilizing policy had to be given. When implemented on a real

robot arm, the robot was able to learn to balance the pendulum in the first trial if a ‘priming’ model was given. Without any prior policy or model, the robot learning only by basic Q-learning took over ten trials before reaching the performance of the primed model.

Variations of Q-learning, $Q(\lambda)$ and Modified Connectionist Q-Learning (MCQ-L) have been used in an attempt to more efficiently arrive at an optimal policy [22]. MCQ-L was made using a hybrid between Q-learning and a generic Temporal Difference learning model, resulting in a value function that updates the same as Q-learning when a greedy choice is made, but learns more quickly when a non-greedy choice allows for exploration. Standard Q-learning, $Q(\lambda)$, and MCQ-L were compared on a simulated robot tasked with traveling toward a goal while avoiding obstacles. All three learning methods were also tested using on-line updates rather than waiting until the end of the episode to update the value functions. Without the use of on-line updates, both $Q(\lambda)$ and MCQ-L converged to an optimal policy much sooner than standard Q-learning. Adding the on-line updates improved performance for all three methods. By using both on-line updates and either $Q(\lambda)$ or MCQ-L, the weight of the parameters in the value functions is lessened, requiring less experimentation to find optimal step size parameter values. While MCQ-L and $Q(\lambda)$ both performed similarly well, MCQ-L did so with lower computational complexity.

Other methods have evolved from modified Q-learning algorithms. The HDG (Hierarchical Distance to Goal) algorithm was formed using Q-learning as a base [11]. The HDG algorithm was modified for use in a landmark network environment. For each state, the HDG algorithm found the nearest landmark to the current state and the nearest landmark to the goal state. If the nearest landmark to the current state and the nearest one to the goal are the same, the agent moved toward that landmark. Otherwise it moved to the next landmark along

the shortest path toward the goal. Rather than dealing with only states, the HDG algorithm used these landmarks as guides, resulting in a sub-task style of learning. Although the agent rarely arrived in any of the actual landmarks, they served to guide it in the direction of other landmarks until it finally reached the goal. Figure 2-6 illustrates an optimal HDG policy. In testing, the HDG method learned faster in a low number of trials than the non-hierarchical DG method, but the HDG performed less well over time than the DG when there were more than fifty runs; in this case, the DG method was able to reach an optimal policy.

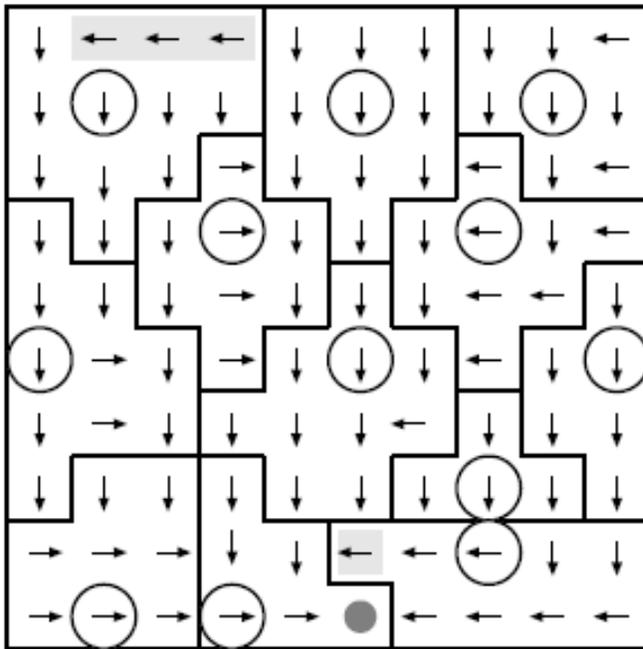


Figure 2-6. An optimal HDG policy. The dark circle in the bottom center is the goal state, and the open circles are landmarks. Boxes indicate regions of states with their nearest landmark.

A hybrid method, Dyna-Q, was developed by blending Q-learning with a modified form of dynamic programming [28]. Another method, Dyna-PI, used policy iteration in order to learn. Dyna-Q used the idea of Q-learning with the Dyna method of generating hypothetical experience in an environment based on a model. Dyna’s planning architecture allowed for unexplored states to be reached more directly during exploration choices of Q-learning, ensuring that all states

would be visited even if the likelihood of discovering upon them through random exploration was very low. Dyna-Q was further split into Dyna-Q+, which adds an exploration bonus, and Dyna-Q-, which lacks the exploration bonus. All three methods were tested in attempting to move around a blocked area in a simulated grid world. Dyna-Q+ performed much better than the other two methods, with Dyna-Q- reaching similar results after a larger number of time steps. Dyna-PI did not perform as well as either of the two Q-learning hybrid methods. A second test was performed with an environment that changed after a period of time. Once all three Dyna methods had an optimal path to the goal, a shortcut through a previously blocked area was opened. Only the Dyna-Q+ method, which gave an increased reward for exploration, found the shortcut. Since Dyna-Q- was computationally less complex, it performed better in static environments than Dyna-Q+. However, if the environment changed, Dyna-Q+'s exploration bonus allowed it to find a new optimal path more quickly than Dyna-Q- or Dyna-PI.

When shortcomings in a robot's sensors, such as noise, limits the field of view, or occlusion prevents an agent from seeing all possible next states, a hidden state problem may arise. A method called instance-based state identification was developed to allow the agent to see these hidden states and proceed with the reinforcement learning algorithms [14]. The agent used memory to recall past states similar to the hidden one in order to estimate its associated values in a process called Nearest Sequence Memory (NSM). Each encountered state was recorded, along with the action taken and reward given, and added to memory in a chain of states. When it encountered a new state, the agent looked through its memory for states similar to its current sensor readings. By averaging the values from the states closest to the current state, the agent updated the value of its current state, as well as the similar states that also indicated the same action chosen. This resulted in a majority vote of similar states at each decision. This

NSM process was shown to help overcome sensor noise and resulted in convergence to an optimal policy in much fewer time steps than the comparison methods of Perceptual Distinction Approach, Recurrent-Q, and Utile Distinction Memory.

There are some situations in which Q-learning methods may fail in real robots. A primary cause of this is through overestimation of values due to noise errors [31]. These overestimation errors primarily stem from the function approximator and the discounting factor. After experimentation on failure to converge to an optimal policy, several methods to counter this overestimation problem were presented. Rather than using instance-based approximators, using an approximator with unbounded memory could reduce the overestimation error to zero. Using methods with eligibility traces, like $TD(\lambda)$, also reduced the error. Removing the discounting factor altogether or adding psuedo-costs reduced the overestimation error, as did using approximators that tended toward underestimating through biasing.

CHAPTER 3 ROBOT PLATFORM

In order to allow the agent to learn outside of simulation, a real robot must be used. The robot chosen for this purposes is a mobile refrigerator delivery robot named Koolio.

Brief History of Koolio

The original Koolio was created in the Machine Intelligence Laboratory by Brian Pietrodangelo and Kevin Phillipson. Their intention was to create a mobile refrigerated vending machine to provide service throughout the various labs and offices in the hallway of the third floor in Benton Hall at the University of Florida. Koolio has undergone several changes since his original inception.

Physical Platform

Koolio consists of a circular base, a refrigerator body, and an LCD screen supported by aluminum pipes (Figure 3-1).

The circular base has a diameter of twenty inches and is supported by two wheels and two casters. To maximize stability of the robot, many of the heavy components are mounted in the base to keep the center of gravity as low as possible. The base contains the batteries, motors, motor drivers, and the docking circuitry. Originally, Koolio had an aluminum skirt around the base, but the current platform keeps it open to allow for ease of changing batteries and access to many of the electronic components. The majority of the sonar sensors are mounted around Koolio's base.

The central body of Koolio is a refrigerator mounted on the base. It is used to hold the packages and cans that Koolio carries to perform his primary function of bringing drinks and food to other offices on the floor. The refrigerator also serves as a means to support the aluminum poles and the backpack. Koolio's backpack, shown in Figure 3-2, holds the

Atmega128 and the single board computer that serve as his primary means of control. A front-facing sonar sensor sits on top of the refrigerator.



Figure 3-1. Koolio.

The LCD monitor serves as Koolio's 'head.' It is mounted on a pair of aluminum pipes running from the base and supported by the refrigerator. Also on his head are side-facing sonar sensors, cameras, and speakers.

The speakers are not currently used for the present setup of Koolio. The monitor, although it can be used for displaying information, is not currently in use.



Figure 3-2. Koolio's backpack.

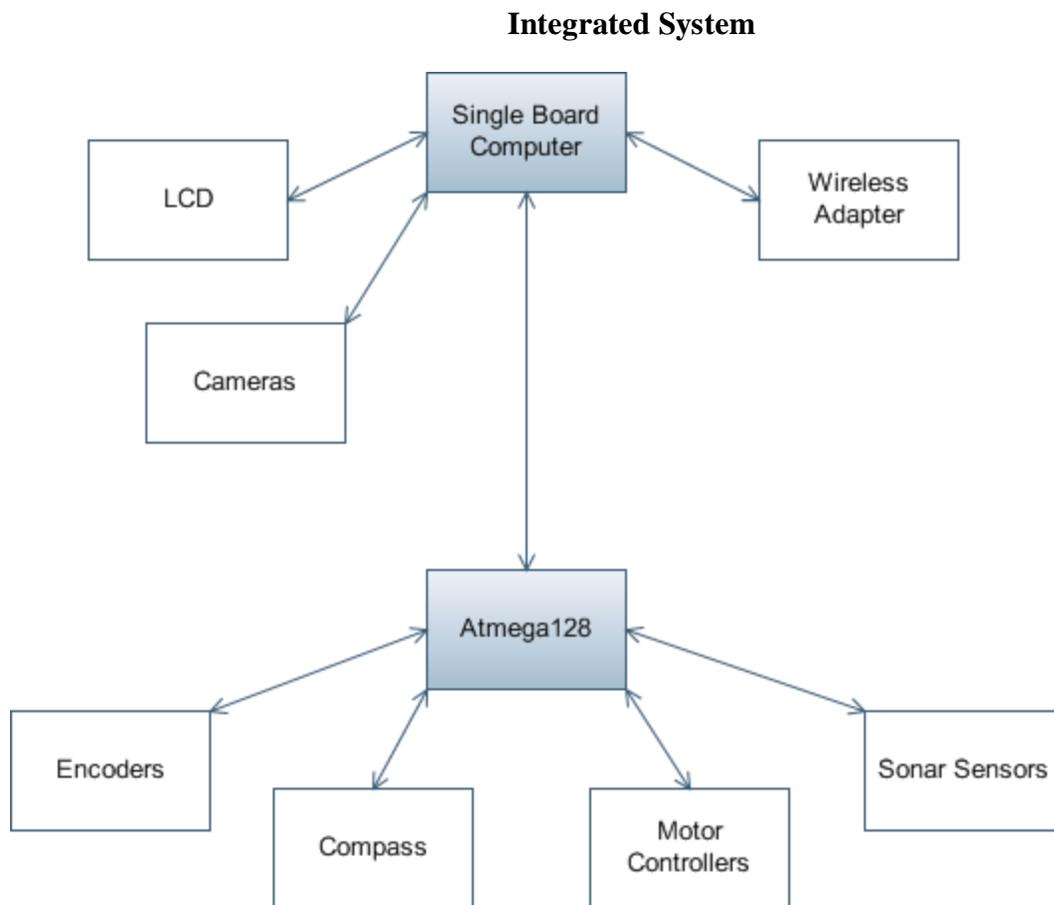


Figure 3-3. Block Diagram of Integrated System.

Koolio’s integrated system is shown in Figure 3-3. The single board computer controls the LCD, takes input from the cameras, uses the wireless adapter to transfer and receive data to and from other computers, and communicates with the Atmega 128. The Atmega 128 handles the encoder, compass, and sonar sensors, as well as driving the motor controls.

Atmega128

An Atmega128 microprocessor on a MAVRIC-IIB board [13] (Figure 3-4) takes care of the input analysis and decision making processes that aren’t from the reinforcement learning process. This allows Koolio to operate under programmed intelligence instead of a learned policy. See Table 3-1 for the Atmega 128 port assignments.

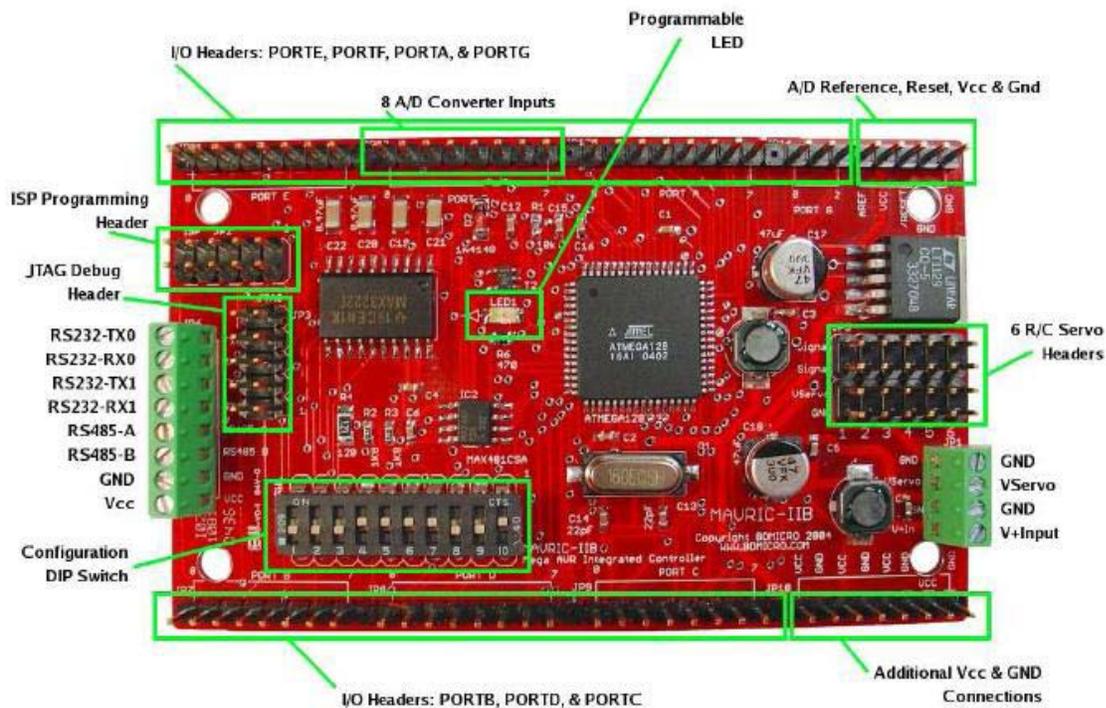


Figure 3-4. MAVRIC-IIB board.

An I²C interface of the Atmega128 connects the processor to the motor control (MD03), the compass (CMPS03), and the sonar sensors (SRF08). The I²C is on Port D of the board (pins 0 and 1). Because the I²C needs only one ground, power, clock, and serial data line per

connected device, and since each I²C address has its own unique 8-bit address, multiple devices can be connected to a single bus. The processor can place the address on the data line, and only the indicated device will be given the commands that follow.

The encoders for the wheels are connected to the Input Capture pins of the Atmega. Each rising edge of the encoder signal triggers an Input Capture interrupt, which enables the processor to determine the wheel position.

The MAVRIC-IIB board is connected to the single board (x86) computer serially through UART0.

Table 3-1. Atmega128 port assignments.

Port/Pins	Assignment
PORTA	LCD
PORTB	not used
PORT C	not used
PORTD, Pins 0, 1	SDL, SCA
PORTD, Pin 4	IC1 – RIGHT – CHA
PORTE, Pin 7	IC3 – LEFT – CHA
PORTF	not used
PORTG	not used

Single Board Computer

Koolio’s Single Board Computer (SBC) is a NOVA 7896 [18] running Ubuntu 6.06 LTS Linux. The SBC has a USB hub, which links the two cameras and the wireless router into a single USB port. The LCD monitor that serves as Koolio’s head also connects to the SBC. It controls the ‘face’ image that is displayed on the monitor. The SBC also connects serially with the Atmega128 to allow sensor information to pass from the microprocessor to the computer.

Sensors

Sonar

Koolio’s primary sensors are sonars mounted at several points on the robot. Koolio has eight sonars, designated Back, Back Left, Back Right, Center Left, Center Right, Top Center,

Top Left, and Top Right. The first five are used for obstacle avoidance and are mounted around the base. The Top Center sonar is also used for obstacle avoidance and is mounted on top of the refrigerator. Sonars Top Left and Top Right are mounted on the side of the head and are used for checking turns, range finding for the cameras, doorway detection, and can also be used for wall-following. Table 3-2 shows the Sonar I²C addresses.

All sonars are SRF08 Ultra Sonic Range Finders [27] (Figure 3-5), chosen because of their compatibility with the I²C interface on the MAVRIC board.

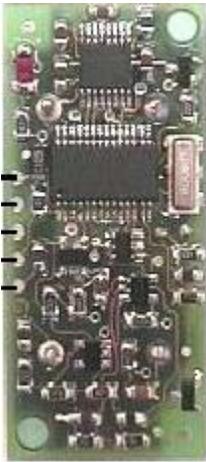


Figure 3-5. SRF08 board.

Table 3-2. Sonar I²C addresses.

Sonar Sensor	Address
Back Left	0xE0
Center Left	0xE2
Center Right	0xE4
Back Right	0xE6
Back	0xE8
Top Left	0xEA
Top Right	0xEC
Top Center	0xEE

The SRF08 sonars can only ping one at a time, so each sensor pings approximately once per second. Experimentation was conducted with simultaneous pinging and lowering of the analog gain, but that produced too much interference between the sensors. Lowering the Range register and increasing the I²C bus frequency also did not increase the ping rate of the sonars.

Encoders

Optical Shaft Encoders are used to count the number of rotations for each wheel. The US Digital S1-50 encoders [23] (Figure 3-6) have two-channel quadrature and contain fifty counts per revolution. The two motors are geared down to make 416 counts per rotation of each wheel. Although there are two channels, only Channel A is used. Channel B's purpose is to lag or lead Channel A in order to determine which direction the wheel is turning. For the purposes of Koolio, this was not necessary.

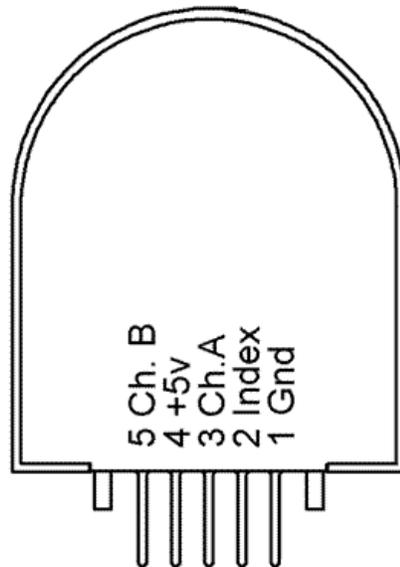


Figure 3-6. S1-50 encoder wire diagram.

Channel A from each encoder is connected to an Input Capture pin on the Atmega processor. This triggers an interrupt each time the encoder sends a rising edge. At a rate of 10 Hz, PID calculations use *RIGHT_ENC_VALUE* and reset the counter to 0. *RIGHT_ENC_CNT* is used to count total distance traveled. Since the circumference of each wheel is 16π inches and there are 416 counts per revolution, Koolio is able to travel $\frac{16\pi}{416}$ inches/count. To calculate distance traveled, the total counts from each encoder are averaged and multiplied by $\frac{16\pi}{416}$.

Distance traveled is calculated by the functions *Start_Distance_S* and *Get_Distance_S*. Table 3-3 shows the port and pin assignments for the encoders' input capture channels.

Table 3-3. Port and pin assignments for encoders on Atmega board.

Port	Pin	Assignment
PORTD	4	Input Capture 1 – Right Encoder Channel A
PORTE	7	Input Capture 3 – Left Encoder Channel A

Compass

Koolio uses a CMPS03 [7] electronic compass module (Figure 3-7) to determine heading.

The CMPS03 is compatible with the I²C used by the Atmega128 microprocessor.

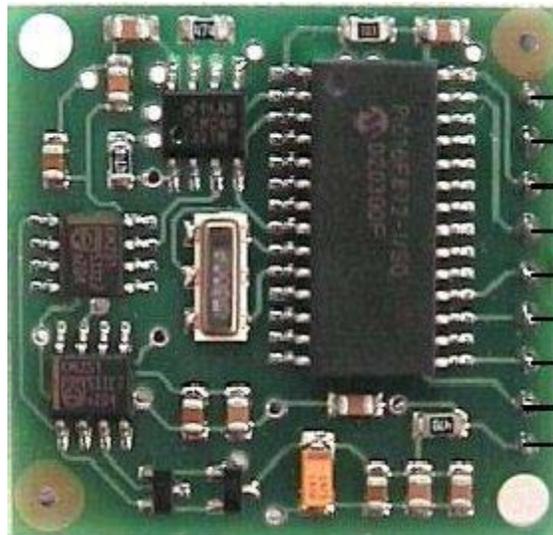


Figure 3-7. CMPS03 circuit board.

Table 3-4. Compass bearings.

Cardinal Bearing	8-bit Bearing Range
North	185-200
South	55-80
East	235-255, 0-15
West	130-140

The compass outputs an 8-bit number from 0-255 corresponding to its current bearing (Table 3-4 for bearing ranges). The function *bearing_8* returns the 8-bit bearing value from the compass to the I²C address C0. The compass is mounted on top of the refrigerator to minimize

noise interference from the motors. However, the reading is inaccurate with a large degree of variability. As such, the compass should not be used for anything but general direction-finding, and not for pinpointing the heading of the robot.

Cameras

Two Creative Video Blaster II Webcams [32] are mounted on the sides of Koolio's head. These cameras are used to identify the room numbers on the walls. These cameras are interfaced to the SBC.

The cameras use color recognition in order to determine the distance to a goal or end of the world point. Each of these points is marked by a square (Figure 5-8) of a color unlikely to be found anywhere else in the normal environment.

The first step of the range-finding process is to search the camera's captured image for that color. If it is not found, then the point is not seen. If a blob of color matching the goal or end of the world points is found, then that point is seen and distance calculation can begin.

In order to calculate distance, the colored blob is measured in individual horizontal pixel lines. The longest of these horizontal lines of colored pixels is considered to be the size of the color blob.

This horizontal size is then used to calculate the distance between the camera and the point. This is possible because all of the goal and end of the world markers are of uniform size and because they are square of shape. Since their actual size is the same for any chosen horizontal cross-section line, it doesn't matter which of the pixel lines in the color blob are selected. In addition, since the point markers are all the same size, there is no need to worry about differences in the markers affecting the length of the horizontal pixel line.

Because of problems with the camera properly focusing on objects that are too close, there is a minimum distance at which this distance calculation will work. This is not a serious issue,

however, since the distance designated as ‘close’ in the simulator and learning software is longer than this minimum accurate distance.

Proximity

Currently, Koolio does not have any means of telling if he bumps into a wall or object. Three possibilities have been considered for this.

A bump ring might be constructed to allow for sensing a bump on any side. While this would be a robust system for detecting bumps, it would be mechanically complex.

A series of whiskers might be added around Koolio’s base. While whiskers work well to detect bumps, they are fragile.

A series of short-range sonar or infrared sensors might be mounted around the base of Koolio. These would be distinct from the current sonar sensors and need only determine if there is something within a certain tolerance range from the base. If Koolio gets within that short tolerance range, it will be read as a bump.

PID (Proportional Integral Derivative)

The PID, or Proportional Integral Derivative, is used to smoothly change the speed of the motors without any jerking motions that may damage them.

Because no two motors are alike, a gradient for each motor had to be found separately. A linear relationship between the encoder counts and the motor command values was approximated to give Equation 3-1 and Equation 3-2.

$$Speed(Left) = 2.81 * encoder_cnt + 13 \quad (3-1)$$

$$Speed(Right) = 2.94 * encoder_cnt + 13 \quad (3-2)$$

The motor speed input into the PID loop is converted to encoder counts for the set point. The Atmega128 then finds the error between the set point and actual motor speed, the encoder counts per 0.1 second interval. This proportional term makes the acceleration of the motor much

more smooth. The motors were found to work best with the K_p coefficient set to 0.35, but it can be adjusted to change the effect of the proportional term. Figure 3-8 shows a block diagram of the motor control system.

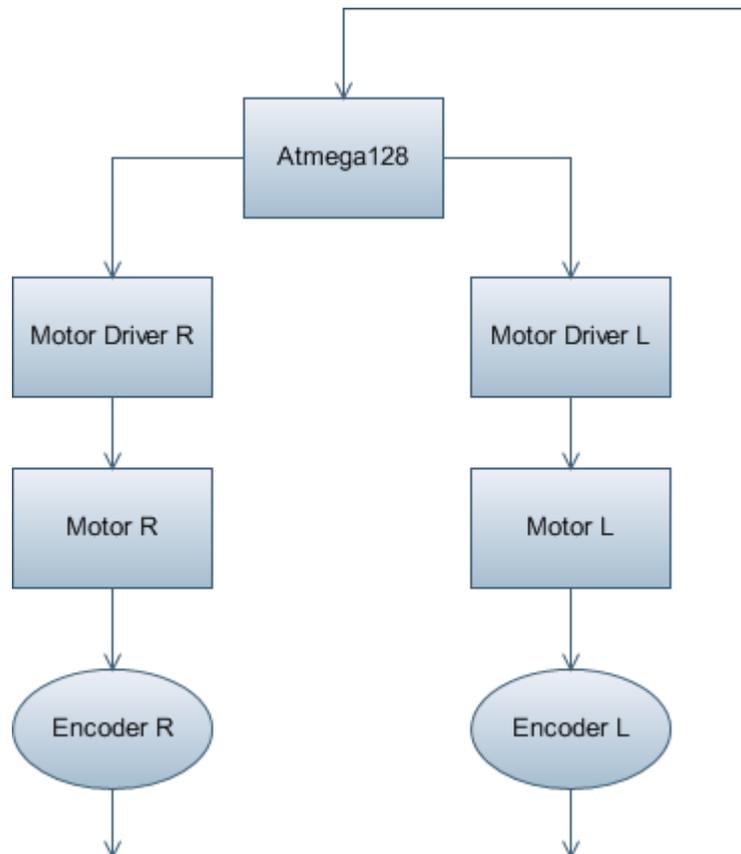


Figure 3-8. Motor control system block diagram.

Electrical Hardware

To keep the processor and computer from being subjected to current spikes, the refrigerator and motors are powered by their own battery. Since the motors may change directions or speed quickly and cause current changes, this is the simplest and most effective solution. The two main batteries share a common ground, but are otherwise isolated from one another and do not affect one another. Since the LCD monitor uses more current than the AVR battery can supply, it has its own separate battery.

Koolio has four external switches and an emergency stop (E-stop) button. The red E-stop button sits on top of the refrigerator and shuts off or reactivates motor power. Three upper switches arranged in a row, from left to right, toggle the power for the electronics, the motors, and the refrigerator (Figure 3-9). The bottom switch toggles the power to the Atmega128 microprocessor.

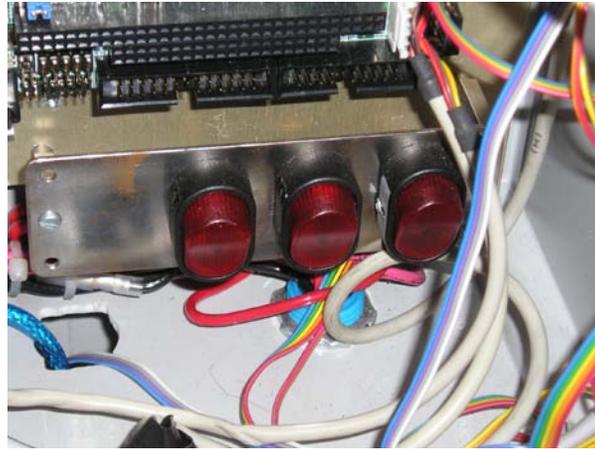


Figure 3-9. Koolio's toggle switches.

Software

SBC (Single Board Computer)

The SBC software oversees the cameras, wireless adaptor, and the LCD monitor, as well as communicating with the Atmega128 processor. Images are taken from the camera are taken using the programs *gqcam* and *streamer*.

Koolio's 'face' is an image displayed on the LCD monitor using the *qiv* program. The face image displays are outputs of the current behavior of Koolio. The face looks forward, left, right, or turns to face away depending on the direction Koolio is moving.

Atmega128

The Atmega128 microprocessor is programmed with several methods to maneuver Koolio without the use of the policy developed through the reinforcement learning process.

Obstacle avoidance

Only the bottom five and top center sonar sensors are used for the obstacle avoidance process; the two top side sensors are not. The sensors were calibrated by making a grid with nine inch squares and placing an object into each of the squares. The values of the lower sensors were recorded for each grid square. These values were compiled to make a set of states of where an obstacle is located: close in front, far in front, close to either side, and far to either side. For each category of states, an equation defines the motor speed for the reaction to seeing the obstacle, with the readings from the left sonars controlling the right motor and the readings from the right sonars controlling the left motor. If an object is close to the side, the opposite motor slows down to turn away from the obstacle or reverse to rotate in place. If the top center sonar detects an obstacle, both motors reverse regardless of the other sonar readings. This top center sonar sensor detects objects like tables that have few or no visible components at the level of the lower sonars around the base. The sonar in back comes into play if there is an object directly behind Koolio. If that sonar detects an obstacle, Koolio will not reverse. The function *Decision_Eq* oversees this process.

Navigation in the Machine Intelligence Lab

In order to exit the lab and get into the hallway, Koolio must follow a specific set of directions. The first step after Koolio receives a command from the SBC to go to a room is to leave his docking station and face West. Distance calculation begins at this point as a global encoder counter is reset. *Arbiter_Lab*, the behavior arbiter, checks the obstacle avoidance function and the bearing, if Koolio is not turning or reversing. If the robot is not within 180 degrees of West, it turns until it is facing that direction. Koolio then moves forward the approximate distance from his docking station to the first turn in the lab and checks his top right sonar. If no objects are detected, then the way is clear and Koolio turns toward the South.

Checking the side sonar allows for extra redundancy with the encoders in case the distance measurements from the encoders is off. Koolio proceeds toward the South until neither left nor right top sonar sensors do not detect a nearby object, signaling that he has cleared the doorway and is now in the hallway.

Hallway navigation

Once Koolio is in the hallway, he turns East or West depending on the office that summoned him. As Koolio travels down the hallway, he observes the obstacle avoidance and wall-following functions through the *Arbiter_Hall* function. Using the functions *Guide_Hallway_Left* and *Guide_Hallway_Right*, Koolio remains a constant distance from the wall to move in a nearly straight path until he has reached his destination.

Door detection

Using the encoders to determine the distance, Koolio stops once he reaches the approximate location of the target door. He turns to face the doorway using the compass and moves forward into the office, using obstacle avoidance and the same processes used in exiting the lab. Koolio then rotates 180 degrees and exits back into the hallway.

Ordering Website

Koolio is meant to take orders from the website to tell him toward which doorway he should move. Text can also be entered into the website telling Koolio what to say once he is at his goal, and remote control of Koolio can be given to the orderer. In order to enable website ordering, the files in Table 3-5 must be included.

The socket function is used to create the socket and bind it to a port. Since Koolio has no way of knowing when an order might come in from a client program, he must run an infinite loop to search for clients trying to connect. This loop only checks for incoming data. Once data is read, it is put into the echo buffer.

Table 3-5. Files that must be included to enable website ordering.

```
#include <sys/types.h> /* for Socket data types */
#include <sys/socket.h> // for socket(), connect(), send(), and recv()
#include <netinet/in.h> /* for IP Socket data types */
#include <arpa/inet.h> /* for sockaddr_in and inet_addr() */
#include <stdlib.h> /* for atoi() */
#include <string.h> /* for memset() */
#include <unistd.h> /* for close() */
```

CHAPTER 4 THE ADVANTAGES OF REINFORCEMENT LEARNING

Learning through Experience

Reinforcement learning is a process by which the agent learns through its own experiences, rather than through some external source. 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.

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.

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.

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 vary 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.

CHAPTER 5 SIMULATOR

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.

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 [1]. 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 [16] (Figure 5-2). 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.

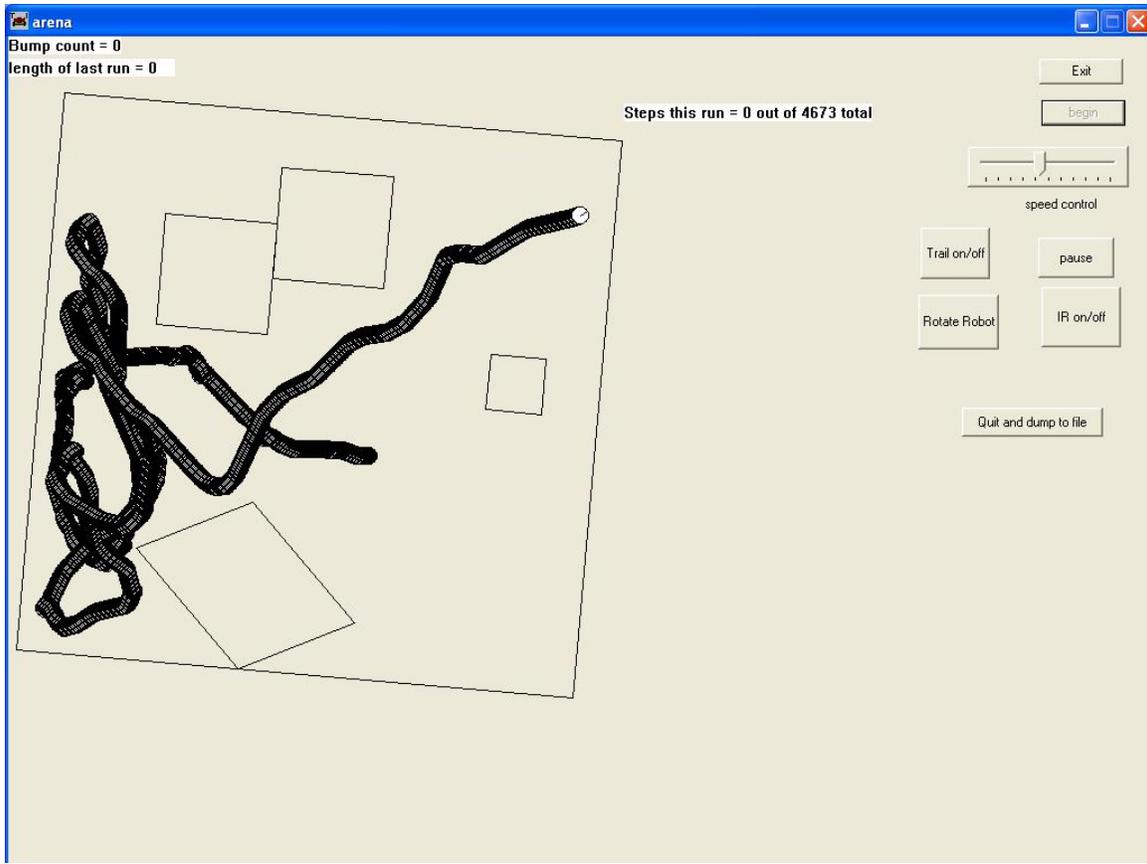


Figure 5-1. The first simulator in action.

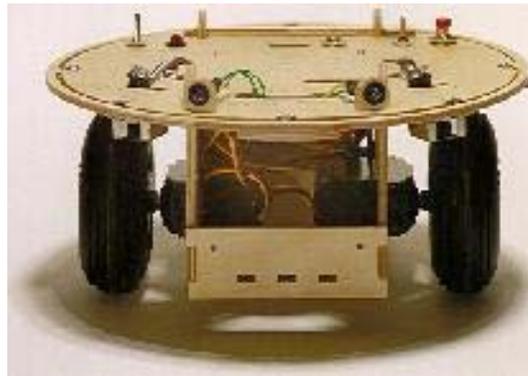


Figure 5-2. The TJ Pro.

Other Considered Simulators

Several other simulation programs were considered as well. The primary considerations were ease of use and editing, the ability to run in a Windows environment, and simplicity. Some simulators add extra features like three dimensional construction that do not add to the Koolio

project but add unnecessary complexity. Table 5-1 lists several simulators that were considered but ultimately not used.

Table 5-1. Simulators considered but not used.

Player [19]
TeamBots [30]
Robot3D [21]
Camelot [6]
Amrose A/S [2]

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.

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.

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 5-3). The goal point was placed on the center of one of the walls. There were no end of the world points present.

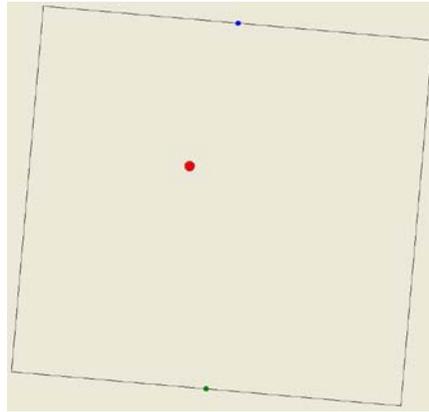


Figure 5-3. A simple square arena.

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

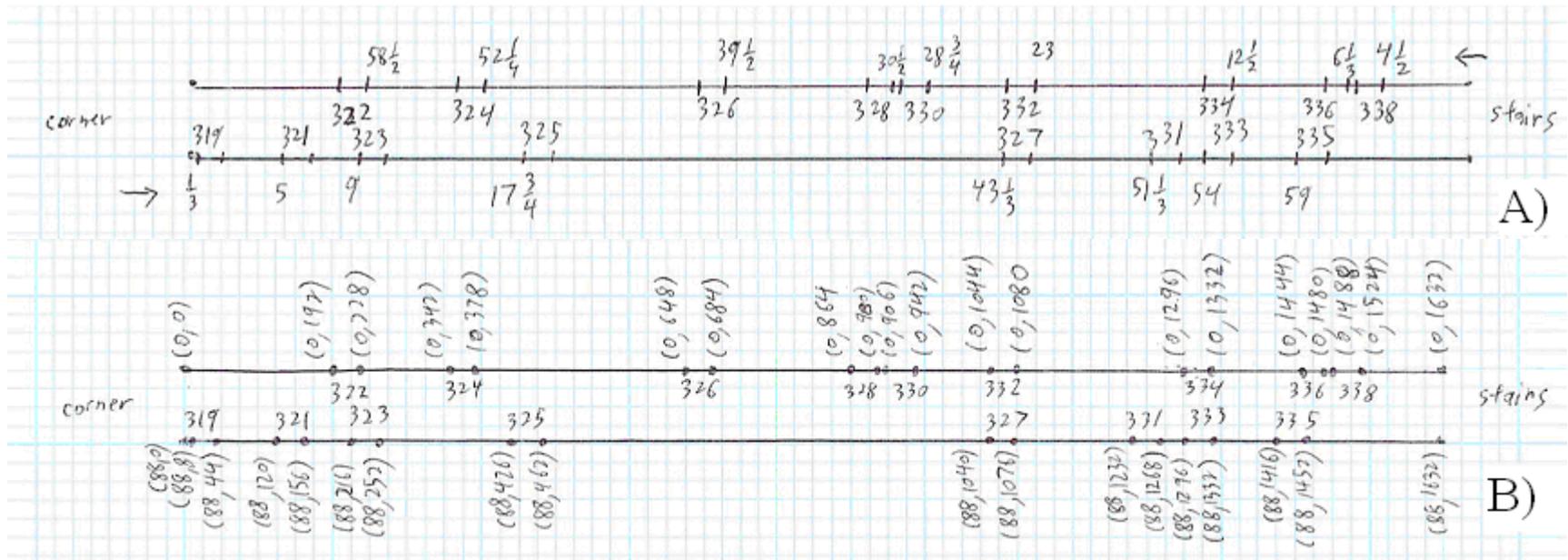


Figure 5-4. Scale model of the hallway environment. Room numbers are indicated in the hallway; measurements are indicated outside. A) Tile measurement. The measurement is a count of the tiles from the end of the hallway in the direction of the arrow indicators to the leading edge of the doorway. The even side rooms begin counting tiles from the stairs side of the hall, while the odd side rooms begin counting tiles from the corner side of the hall. B) Coordinate measurement. Using 36 pixels per tile side, this is an absolute coordinate scale of the door edges.

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. Each doorway has a plate on either the right or the left side to identify the room by name and number. Figure 5-4 shows the hallway environment without the T-shaped areas on the end with room numbers measured in both number of tiles and pixel coordinates.

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.

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.

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.

Sonar

The sonar sensors are defined by their location on the robot in relation to the front (Figure 5-5). 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.

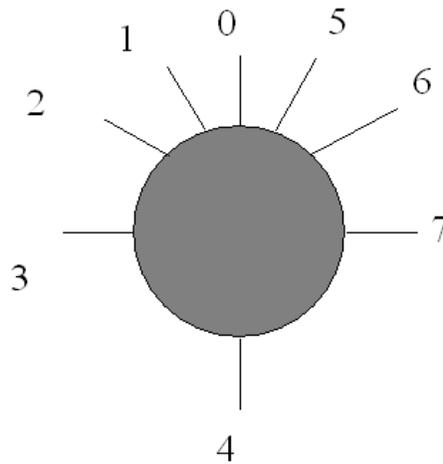


Figure 5-5. The locations of the sonar sensors on the agent.

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. Table 5-2 shows the variables used in this function.

Table 5-2. Variables used in *LineIntersectDistance*.

Variable	Use
<i>Slope1</i>	slope of the sonar line segment
<i>b1</i>	Y-intercept of the sonar line segment
<i>Xa</i>	X-coordinate of one end of the wall
<i>Ya</i>	Y-coordinate of one end of the wall
<i>Xb</i>	X-coordinate of the other end of the wall
<i>Yb</i>	Y-coordinate of the other end of the wall
<i>SensorTheta</i>	angle between the front of the robot and the center of the sensor arc

Using the endpoints for the wall, the equation of the line containing that wall is found (Equations 5-1 and 5-2). These two equations are used to find the intersection point (Ix,Iy) between the two lines (Equations 5-3, 5-4, 5-5, and 5-6).

$$Slope2 = \frac{Ya - Yb}{Xa - Xb} \quad (5-1)$$

$$b2 = Ya - Slope2 * Xa \quad (5-2)$$

$$Y1 = Y2 \quad (5-3)$$

$$Slope1 * Ix + b1 = Slope2 * Ix + b2 \quad (5-4)$$

$$Ix = \frac{b2 - b1}{Slope1 - Slope2} \quad (5-5)$$

$$Iy = Slope1 * Ix + b1 \quad (5-6)$$

Table 5-3. Conversion of (Ix,Iy) to polar coordinates (PolarR,PolarTheta).

$$PolarX = LocationX - Ix$$

$$PolarY = LocationY - Iy$$

$$PolarR = \sqrt{PolarX^2 + PolarY^2} - RobotRadius$$

if $PolarX > 0$

$$then \ PolarTheta = \arctan\left(\frac{PolarY}{PolarX}\right) + \pi$$

else if $(PolarX < 0) \& (PolarY > 0)$

$$then \ PolarTheta = \arctan\left(\frac{PolarY}{PolarX}\right) + 2\pi$$

else if $(PolarX < 0) \& (PolarY \leq 0)$

$$then \ PolarTheta = \arctan\left(\frac{PolarY}{PolarX}\right)$$

else if $(PolarX = 0) \& (PolarY > 0)$

$$then \ PolarTheta = \frac{3\pi}{2}$$

else if $(PolarX = 0) \& (PolarY < 0)$

$$then \ PolarTheta = \frac{\pi}{2}$$

Once the intersection point is found, it must be converted from Cartesian coordinates to polar coordinates in relation to the location of the robot (Table 5-3).

Although the intersection point is known, a check must be made to ensure that it is within the viewable angle of the sensor (Table 5-4). 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 5-6 shows an example of a situation in which *ReallyLongDistance* will be returned as the sensor reading.

Table 5-4. Checking whether the intersection is within the arc of the sensor.

$$ArcCheckMax = RobotOrientation + SensorTheta + (SensorArc / 2)$$

$$ArcCheckMin = RobotOrientation + SensorTheta - (SensorArc / 2)$$

if ($PolarTheta \leq ArcCheckMax$) & ($PolarTheta \geq ArcCheckMin$)
 then $distance = PolarR$
 else $distance = ReallyLongDistance$

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 5-7).

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.

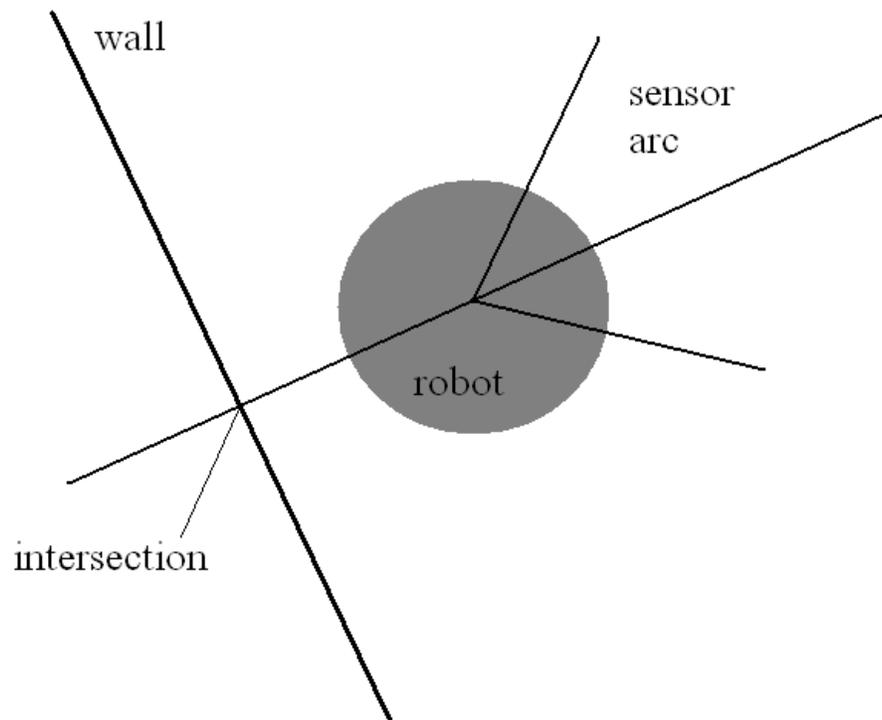


Figure 5-6. Configuration that will result in a sensor reading of *ReallyLongDistance* due to the intersection being outside of the sensor's viewing arc.

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.

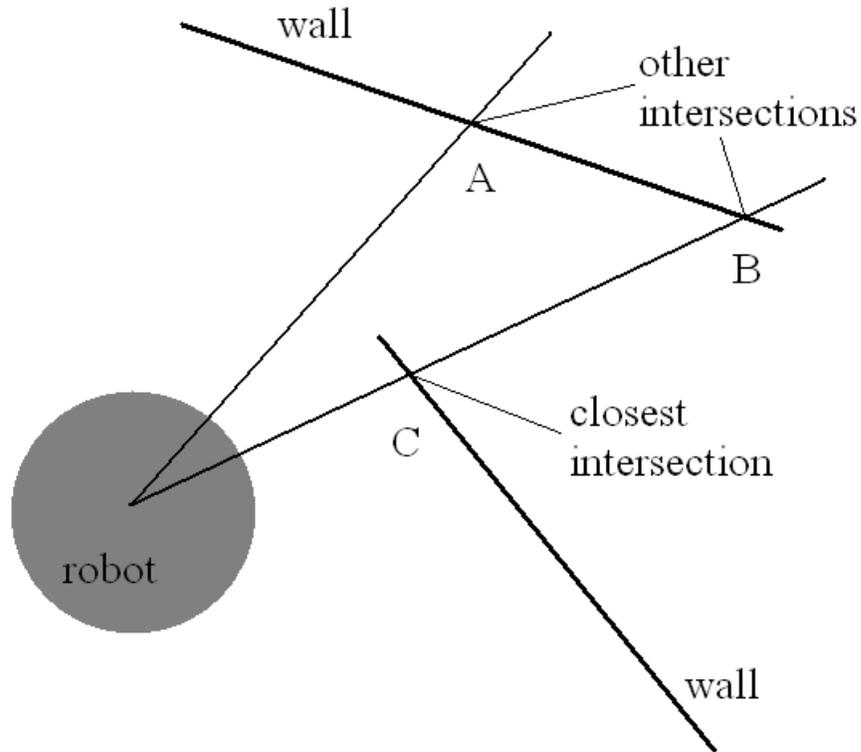


Figure 5-7. Multiple detected walls. Only the closest wall will matter. Intersection B will be removed during *LineIntersectDistance* because intersection C is closer on that direction line, making C the *MinSegmentDistance* for that smaller arc. Since intersection C is closer than intersection A, C will also become the *MinDistance* for the entire sensor reading.

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 5-5.

Table 5-5. 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.

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 5-8). In the simulator, these markers are indicated to the user as colored dots (Figure 5-9).

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 5-8. Goal (left) and End of the World (right) markers used in the real environment.

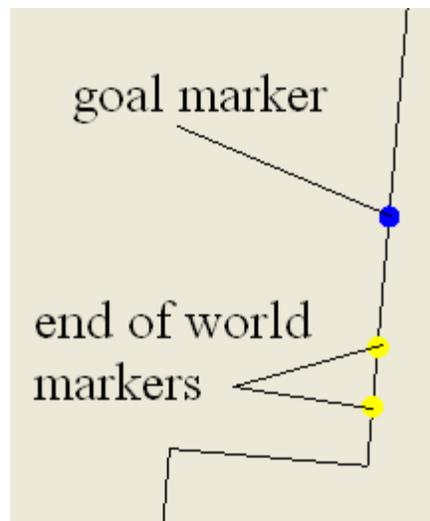


Figure 5-9. Goal and End of the World markers in the simulator.

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.

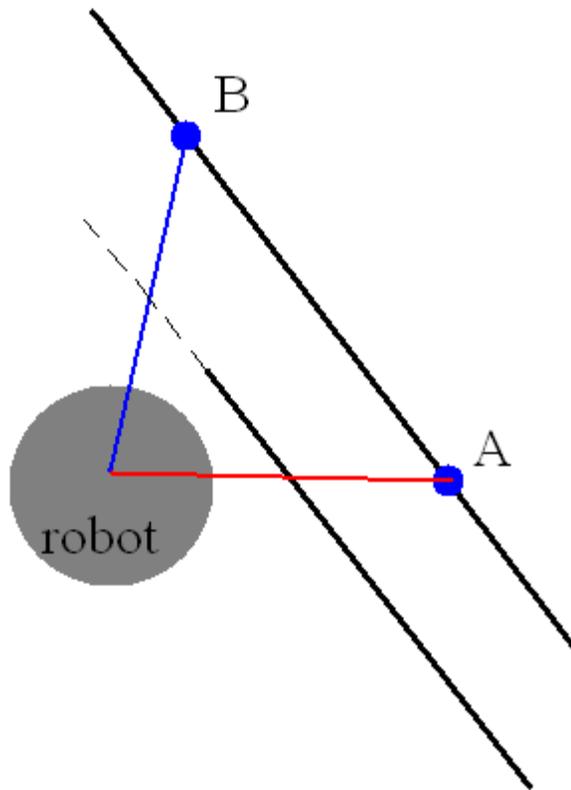


Figure 5-10. Two examples of line of sight to a Goal point. Point A is blocked by the wall. The full line that contains the wall lies between the robot and point B, but the point is considered visible after checking the wall endpoints.

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 5-10). 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.

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 5-6.

Table 5-6. 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.

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 is 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.

Given the equation of a circle (Equation 5-7) and the equation of a line (Equation 5-8), a quadratic equation (Equation 5-9) can be made describing the intersections between the two. The coefficients of this quadratic equation (Equations 5-10, 5-11, and 5-12) and its discriminant (Equation 5-13) can be used to find the x-coordinates (Equations 5-14, 5-15) and the y-coordinates of the intersection (Equations 5-16, 5-17).

$$(x - LocationX)^2 + (y - LocationY)^2 = radius^2 \quad (5-7)$$

$$y = Slope * x + Intercept \quad (5-8)$$

$$\begin{aligned} (Slope^2 + 1)x^2 + 2(Slope(Intercept - LocationY) - LocationX)x \\ + (LocationX^2 + LocationY^2 + Intercept^2 - radius^2 - 2 * Intercept * LocationY) = 0 \end{aligned} \quad (5-9)$$

$$quadraticA = Slope^2 + 1 \quad (5-10)$$

$$quadraticB = 2(Slope * (Intercept - LocationY) - LocationX) \quad (5-11)$$

$$quadraticC = LocationX^2 + LocationY^2 + Intercept^2 - radius^2 - 2 * Intercept * LocationY \quad (5-12)$$

$$discr = quadraticB^2 - 4 * quadraticA * quadraticC \quad (5-13)$$

$$IntersectX1 = \frac{-qB + \sqrt{discr}}{2 * qA} \quad (5-14)$$

$$IntersectX2 = \frac{-qB - \sqrt{discr}}{2 * qA} \quad (5-15)$$

$$IntersectY1 = Slope * IntersectX1 + Intercept \quad (5-16)$$

$$IntersectY2 = Slope * IntersectX2 + Intercept \quad (5-17)$$

If the discriminant is negative, the robot is not touching the wall. If it is 0, the robot is tangent to the wall, touching at one point. If it is positive, the robot intersects the wall at two points. If the discriminant is negative and the robot is not touching that wall, nothing is done. Otherwise, the quadratic formula is performed to find the point or points of intersection.

In the rare case that there is actually only one intersection, both of these two intersection points will be identical. As in the case with the sonar and camera sensors, the intersection points may not be on the line segment that contains the wall.

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 5-18) is less than or equal to 1 and either *Check1* or *Check2* (Equation 5-19) 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.

$$Check1 = \frac{IntX1 - WallEndX1}{WallEndX2 - WallEndX1} \quad (5-18)$$

$$Check2 = \frac{IntX2 - WallEndX1}{WallEndX2 - WallEndX1} \quad (5-19)$$

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.

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 5-11 shows some examples of the sensor displays.

The encoded readings for the sensors can also be displayed (Figure 5-12). 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 if any of the sonar sensors read a Bump state.

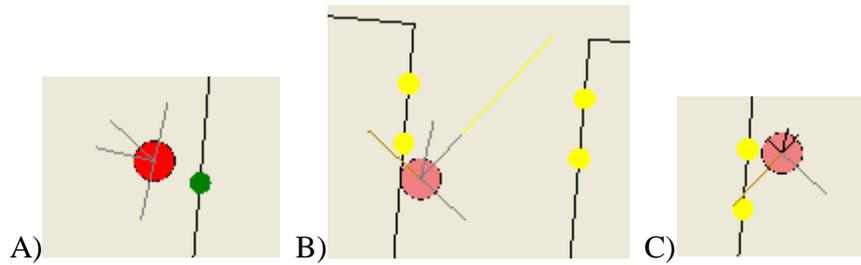


Figure 5-11. Sensor displays. A) The robot near the start area. B) The robot in a Bump state seeing the End of the World markers in the front camera (Medium) and the left camera (Close). C) The robot in a Bump state seeing an End of the World marker in the left camera (Close).

HUD	BUMP		
Front Sonar:		Close	
Front Left Sonar:		Close	
Front Right Sonar:		Far	
Left Sonar:		Close	
Right Sonar:		Far	
Back Sonar:		Far	
Front Goal Camera:		Medium	RIGHT
Left Goal Camera:		Far	
Right Goal Camera:		Far	
Front End of World Camera:		Far	
Left End of World Camera:		Far	
Right End of World Camera:		Far	

Figure 5-12. The sensor HUD.

A statistics display for the simulation can be toggled with the *Show Stats* button (Figure 5-13). 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.

Simulation Statistics	
Current Episode Reward:	-25000
Current Episode Steps:	105
Current Episode Bumps:	25
Number of Episodes:	67689
Total Number of Goals:	21708
Average Reward per Episode:	-5289046.46318397
Average Steps per Episode:	555.167370878147
Average Bumps per Episode:	525.548162155774

Figure 5-13. Simulation Statistics.

A display of values from the current Q-table location can also be toggled with the *Display Q Values* button (Figure 5-14). 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.

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.

Q Table Values	
Random Choice	No Random
Forward Reward	16.6666660043928
Reverse Reward	14.6666660043928
Stop Reward	-13.3333339956071
Left Reward	15.6666660043928
Right Reward	15.6666660043928
Choice	FORWARD
Current Reward	4

Figure 5-14. Q-Value display.

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 5-15). 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.

In addition, there is a control panel for manual controls (Figure 5-16). 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.

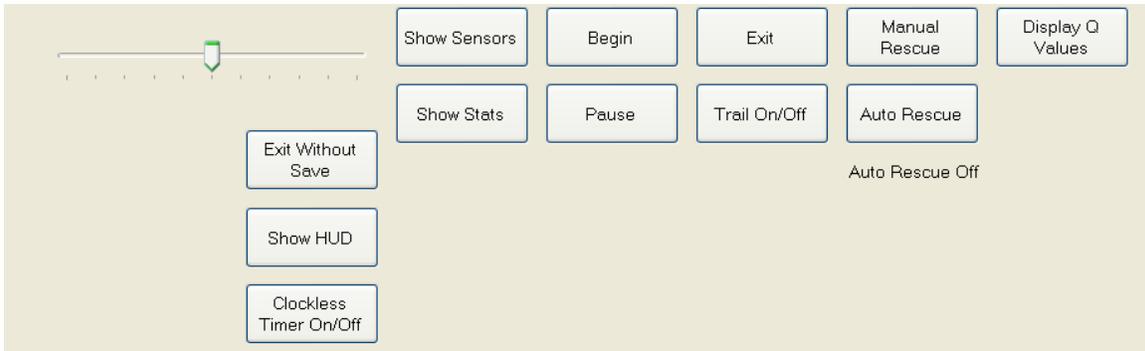


Figure 5-15. The main control panel.

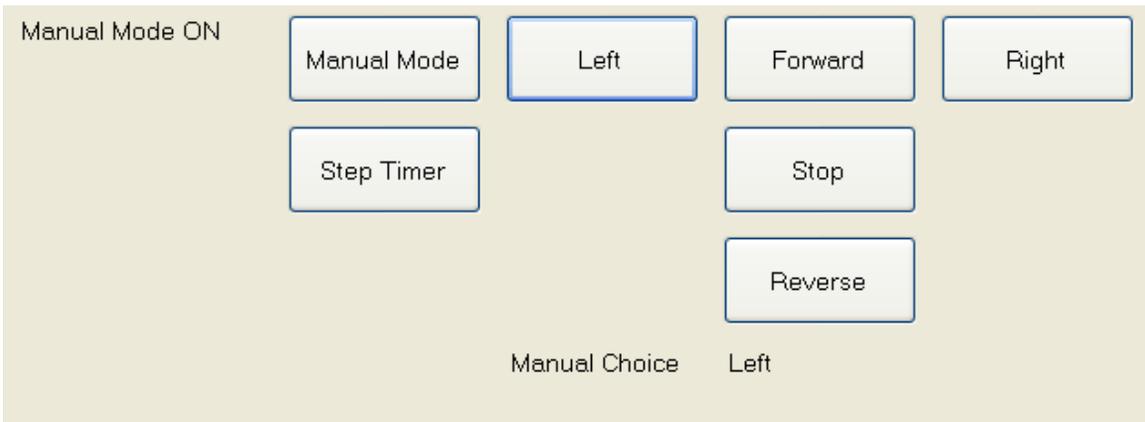


Figure 5-16. The control panel for manual mode.

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 $1 - \epsilon$, 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 5-7. Status variables in the simulator.

Status Variable	True When
<i>AtGoal</i>	Goal is seen at distance <i>Dist_Close</i> in the front camera
<i>AtEndOfWorld</i>	Agent has gone past the end of the world
<i>BumpDetected</i>	Any sonar sensor reads a distance of <i>Dist_Bump</i>
<i>CenterOfHall</i>	Sonar sensors on the left and right read the same distance
<i>CanSeeGoal</i>	Any camera sensor can see the goal at distance <i>Dist_Close</i> or <i>Dist_Medium</i>
<i>GoalInFront</i>	<i>CanSeeGoal</i> is true and the goal is in either the right or left focus of the front camera
<i>GoalFrontCenter</i>	<i>CanSeeGoal</i> 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 5-7). 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 5-20). 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.

$$NewQ = CurrentQ + \alpha * (reward + (\gamma * \max(NextQ)) - CurrentQ) \quad (5-20)$$

The final step is to move to the next state.

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 led 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 (Equation 5-21). 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.

$$QKeyword = GoalFoc2*2^{31} + GoalFoc1*2^{29} + GoalFoc0*2^{27} + GoalCam0*2^{25} + GoalCam1*2^{23} + GoalCam2*2^{21} + EoWCam0*2^{19} + EoWCam1*2^{17} + EoWCam2*2^{15} + Sonar0*2^{13} + Sonar1-2*2^{11} + Sonar3*2^9 + Sonar4*2^7 + Sonar5-6*2^5 + Sonar7*2^3 + Action \quad (5-21)$$

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.

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 $1 - \epsilon$, 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 5-22) and is added to the Q-table.

$$NewQ = CurrentQ + \alpha(reward + (\gamma * NextQMax) - CurrentQ) \quad (5-22)$$

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.

CHAPTER 6 EXPERIMENTAL SIMULATION

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 so 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.

Successful Trials

Reward Schemes

After much experimentation and the addition of the status variables (Table 5-7) to determine the proper values, a new reward scheme was created (Table 6-1). 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 6-1). 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 6-1. Reward scheme when goal is not seen.

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

If *CanSeeGoal* is true, but *GoalInFront* and *GoalFrontCenter* are not, then the robot can see the goal from the side cameras (Table 6-2). 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.

Table 6-2. Reward scheme when *CanSeeGoal* is true.

Reward Name	Reward Amount
<i>Reward2_Goal</i>	2000
<i>Reward2_Bump</i>	-5000
<i>Reward2_End_of_World</i>	-30000
<i>Reward2_Stop</i>	-25
<i>Reward2_Forward</i>	2
<i>Reward2_Reverse</i>	1
<i>Reward2_Turn</i>	2
<i>Reward2_Big_Turn</i>	2
<i>Reward2_Center_Forward</i>	2
<i>Reward2_Center_Reverse</i>	1
<i>Reward2_Center_Turn</i>	2
<i>Reward2_Center_Big_Turn</i>	2

Table 6-3. Reward scheme when *GoalInFront* is true.

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

If *GoalInFront* is true, then the robot can see the goal in either the left or right focus of the front camera (Table 6-3). 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.

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 6-4). 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.

Table 6-4. Reward scheme when *GoalFrontCenter* is true.

Reward Name	Reward Amount
<i>Reward3_Goal</i>	4000
<i>Reward3_Bump</i>	-2500
<i>Reward3_End_of_World</i>	-20000
<i>Reward3_Stop</i>	-25
<i>Reward3_Forward</i>	10
<i>Reward3_Reverse</i>	-1
<i>Reward3_Turn</i>	7
<i>Reward3_Big_Turn</i>	7
<i>Reward3_Center_Forward</i>	10
<i>Reward3_Center_Reverse</i>	-1
<i>Reward3_Center_Turn</i>	7
<i>Reward3_Center_Big_Turn</i>	7

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.

Future Work

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 an 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.

CHAPTER 7 EXPERIMENTATION ON THE 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.

APPENDIX A GLOSSARY OF TERMS

ACTION. The output at any given time. Actions are usually associated with the states from which they are performed, resulting in a state-action pair. These state-action pairs are made up of an available action from the given state. Any state may have multiple state-action pairs. In the case of a robot, the action is often movement.

AGENT. What is learning. In the cases of reinforcement learning used for this project, the agent is Koolio with his sensor inputs and motor control outputs.

BOOTSTRAPPING. The method of updating estimates based on other estimates. In the realm of reinforcement learning, bootstrapping is used to update the estimates of state values and action values based on the estimates of the state values and action values of later states.

DISCOUNTING. The process of assigning less value to older experiences than to newer experiences. Most reinforcement learning methods use discounting to ensure that the policy is updated from the most recent discoveries and mistakes from many episodes before do not block current policies from becoming optimal.

EPISODE. A single instance of a task that is performed many times with a definite beginning and end to each episode. An example of these episodic tasks is a game, where each game played represents a different episode.

EXPLOITATION. Following the current policy toward the known best action choice.

EXPLORATION. Choosing a random action rather than following the policy. This allows for all possible state-action choices to be taken.

EXPLORING STARTS. A method that gives every state-action pair a non-zero probability of being the first step in each iteration

FIRST-VISIT. Methods that use the average of returns after the first instance of a given state to determine the state-action pair's value function

GREEDY. An algorithm in which the best choice is always selected. ϵ -greedy is a commonly-used variation in which ϵ is some small number. The best choice is selected with $(1-\epsilon)$ probability and a random choice is made with ϵ probability.

OFF-POLICY. Methods that separate the functions of control and policy evaluation into two separate functions: the behavior policy to govern the decisions of the current policy and the estimation policy to be evaluated and improved

ON-POLICY. Methods that use first-visit to estimate the current policy's state-action value functions

POLICY. A set of state-action pairs that define the behavior of the agent. The policy is, in the end, what reinforcement learning methods attempt to maximize.

POLICY EVALUATION. The value functions for a given policy. Iterative policy evaluation is the process of repeating the policy value calculation until it converges to the actual value of that policy. In performing iterative policy evaluation, the value of a particular state is replaced with a new state value based on the old values of the following states and the expected immediate rewards in a process called full backup. The policy values obtained through policy evaluation are used as benchmarks in order to compare other policies and determine which is better.

POLICY ITERATION. The process of evaluating a policy and comparing it to the previous best policy. If the current policy is better than the previous best, it replaces the best. This process is repeated until an optimal policy is reached.

POLICY ITERATION, GENERALIZED (GPI). A two-step circular process in which policy evaluation is used to determine the values of the available policies, and a greedy algorithm picks the best of these to become the new current policy. This policy is then evaluated to determine the next set of value functions.

POLICY, OPTIMAL. A policy that yields a reward greater than or equal to all other policies. Optimal policies are the end goal of any reinforcement learning algorithm.

RETURN. The sum of rewards received for a given policy.

REWARD. The basic element of reinforcement learning. By giving rewards at certain points, a learning method can let the agent know what choices are good and what choices are bad. Rewards can be either positive or negative.

STATE. The current logical position of the decision tree. States in Markov processes are usually defined as the set of inputs of all the sensors. The state tells the agent all the available information about where it is in regards to the environment and its sensors.

VALUE, STATE. The expected return from that state onward.

VALUE, STATE-ACTION. The expected return from that state onward given that action is selected. Also known as the action value.

APPENDIX B
LEARNING CURVES

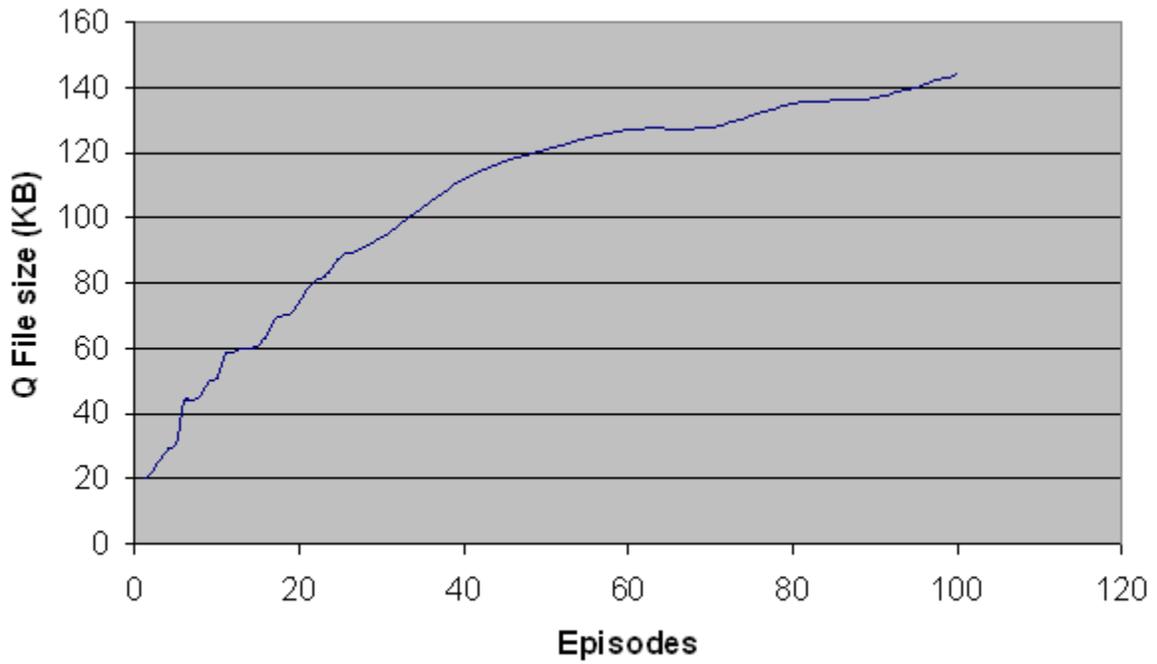


Figure B-1. Size of Q File for first 100 episodes.

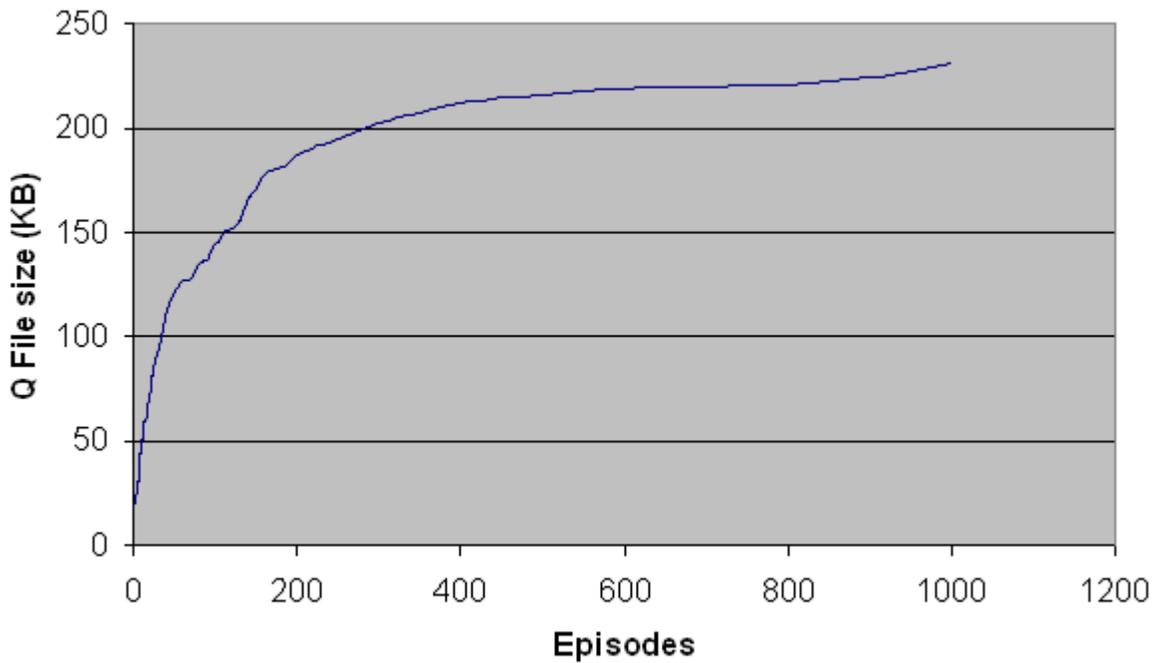


Figure B-2. Size of Q File for first 1000 episodes.

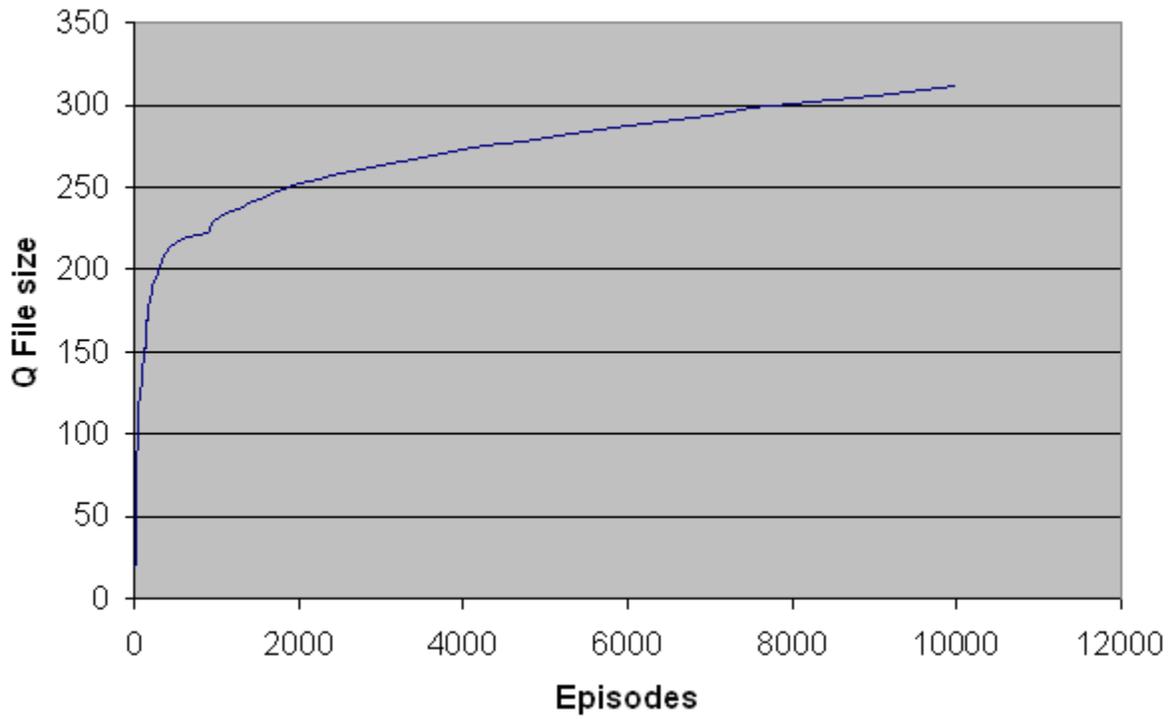


Figure B-3. Size of Q File for first 10000 episodes.

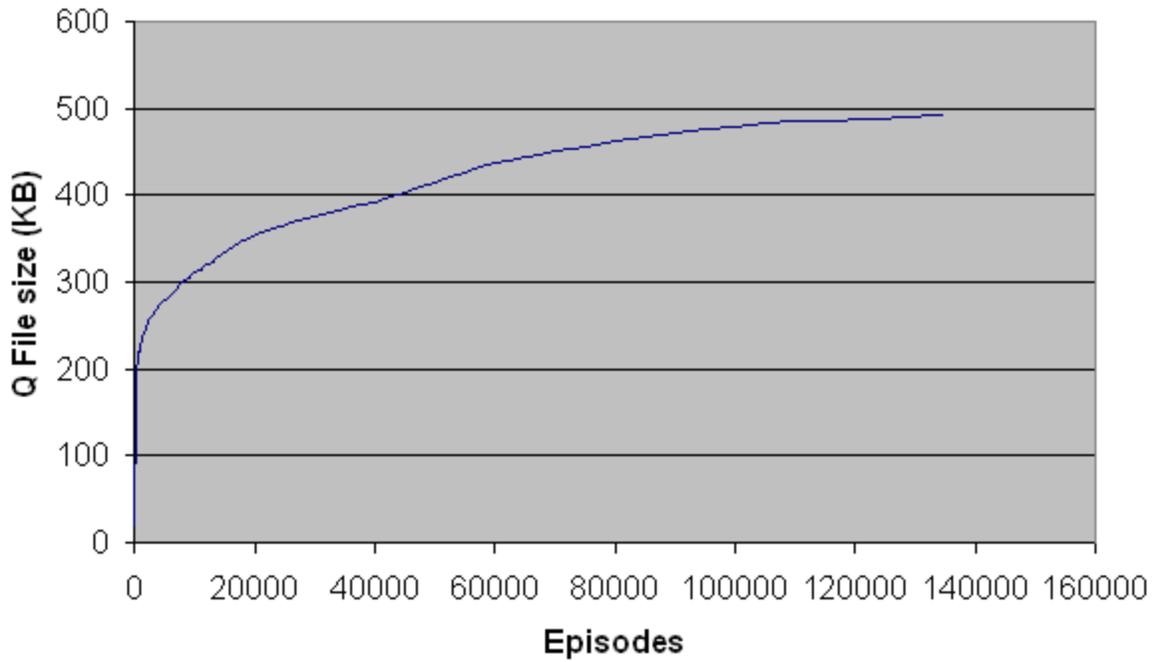


Figure B-4. Size of Q File for all experimental episodes.

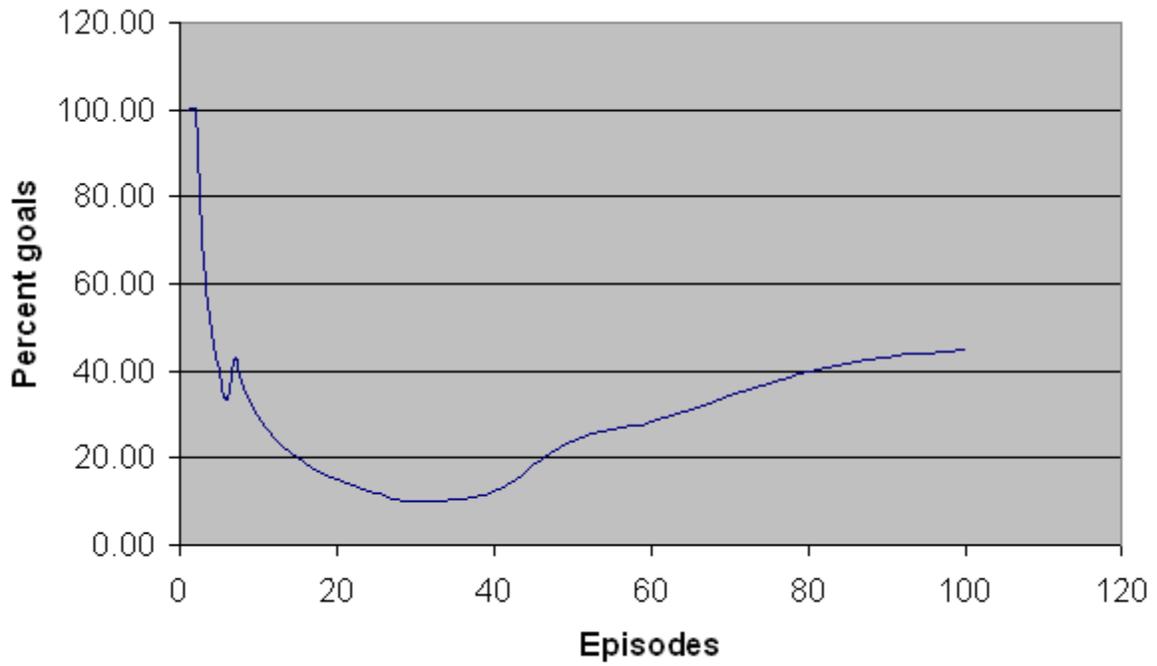


Figure B-5. Percentage of the first 100 episodes that ended at the goal.

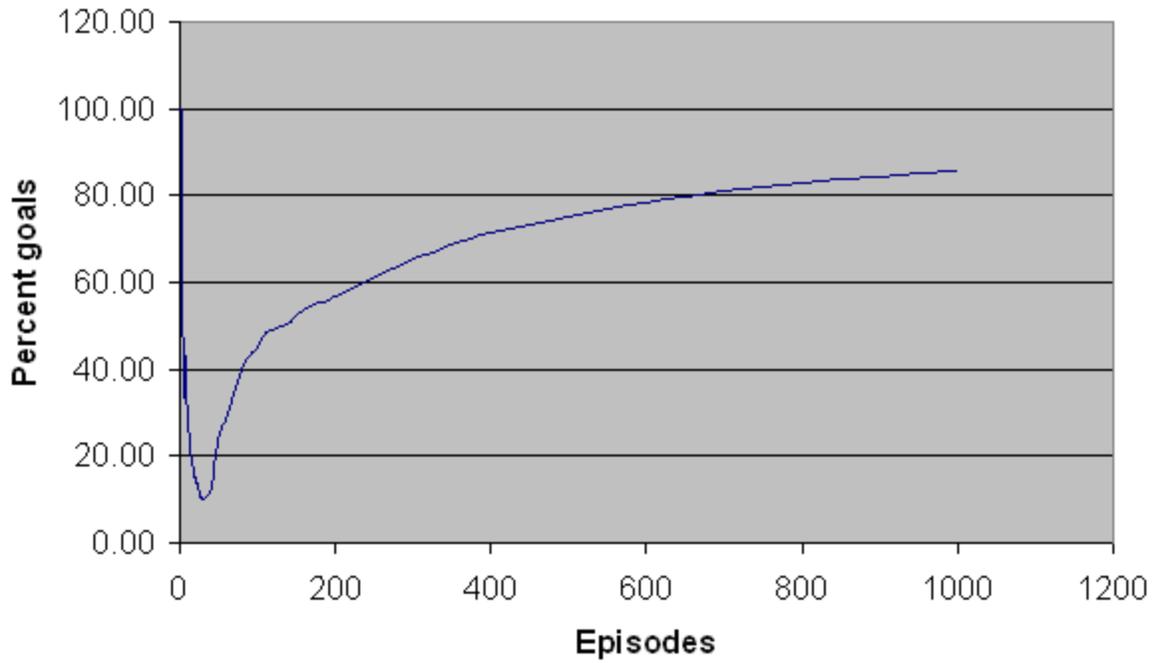


Figure B-6. Percentage of the first 1000 episodes that ended at the goal.

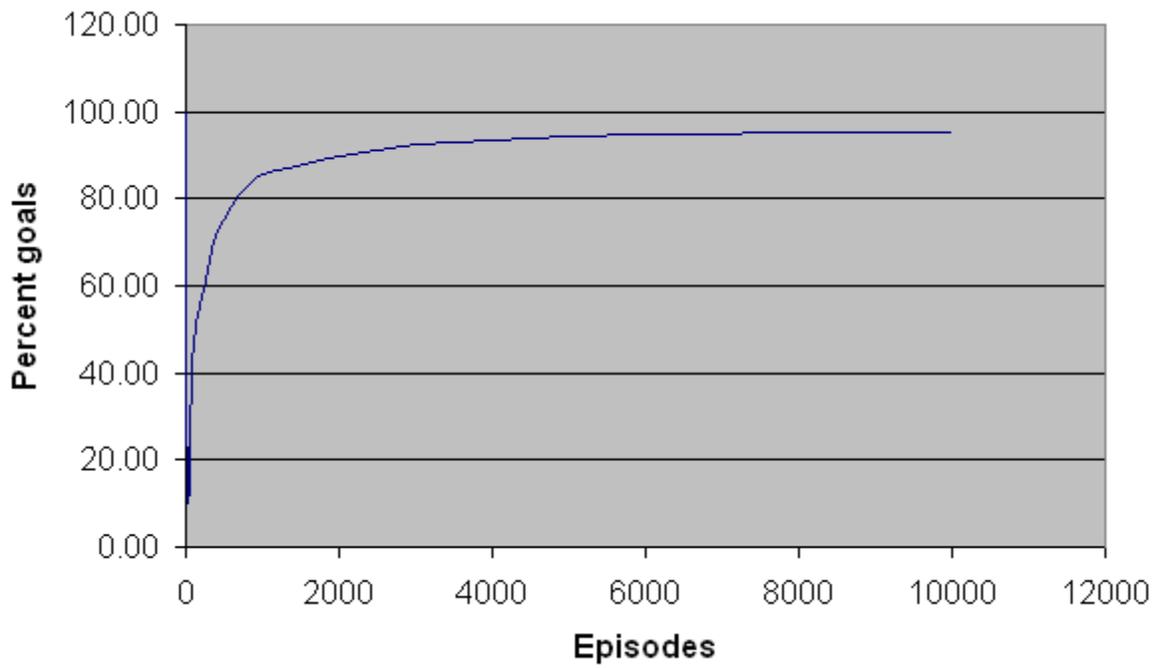


Figure B-7. Percentage of the first 10000 episodes that ended at the goal.

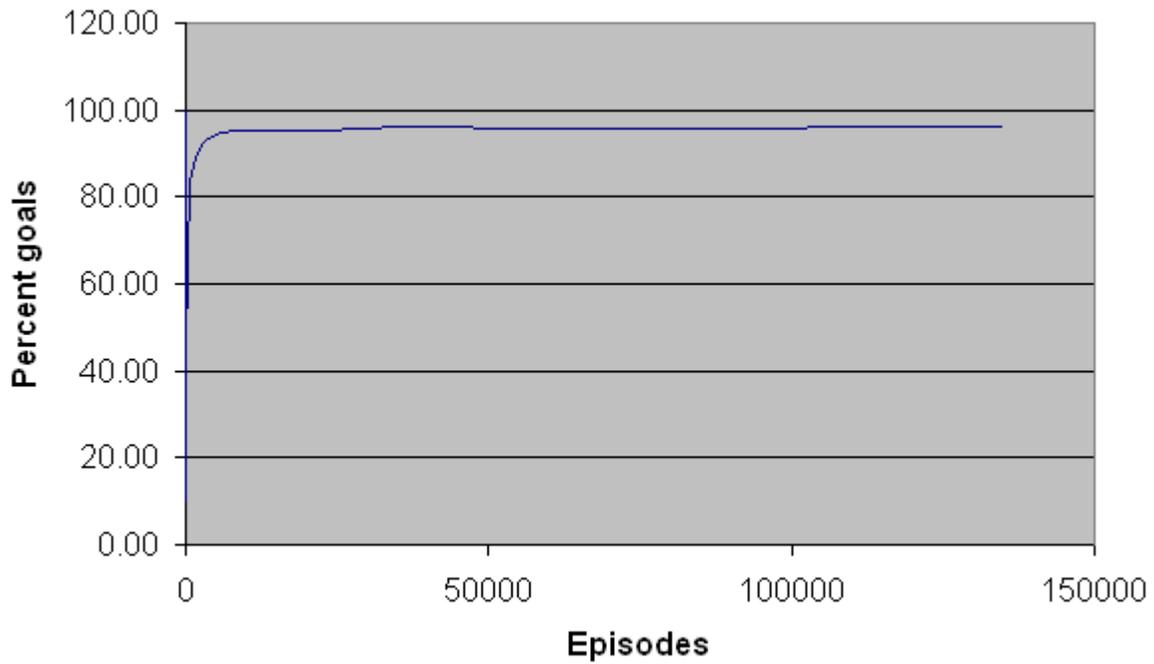


Figure B-8. Percentage of all experimental episodes that ended at the goal.

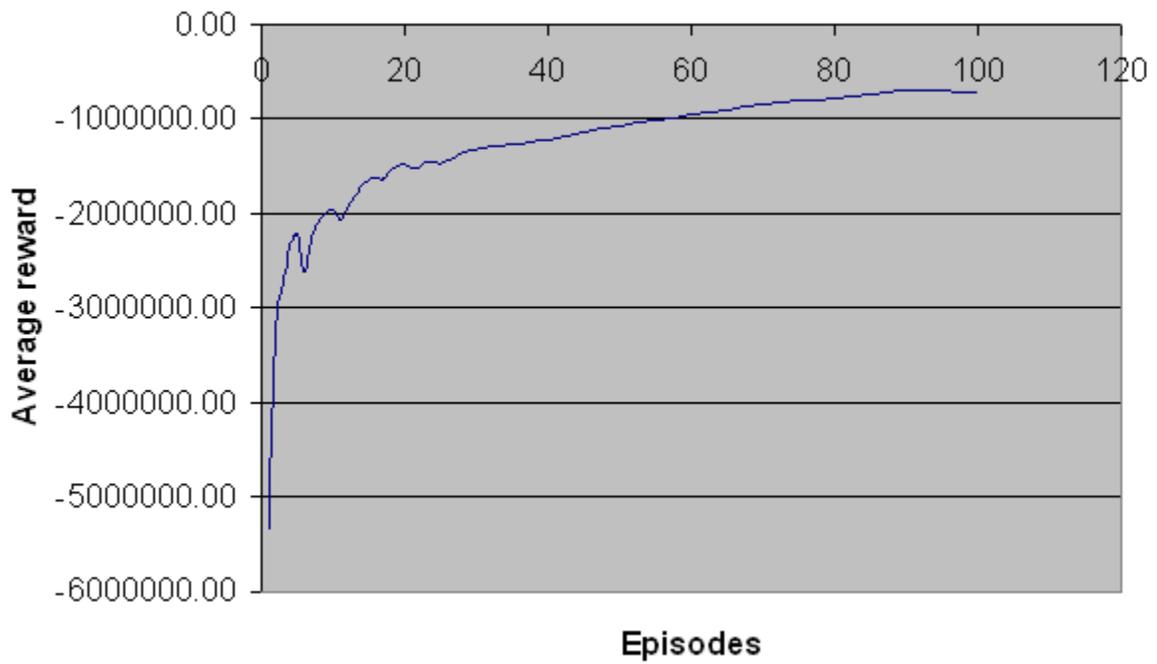


Figure B-9. Average reward per episode for the first 100 episodes.

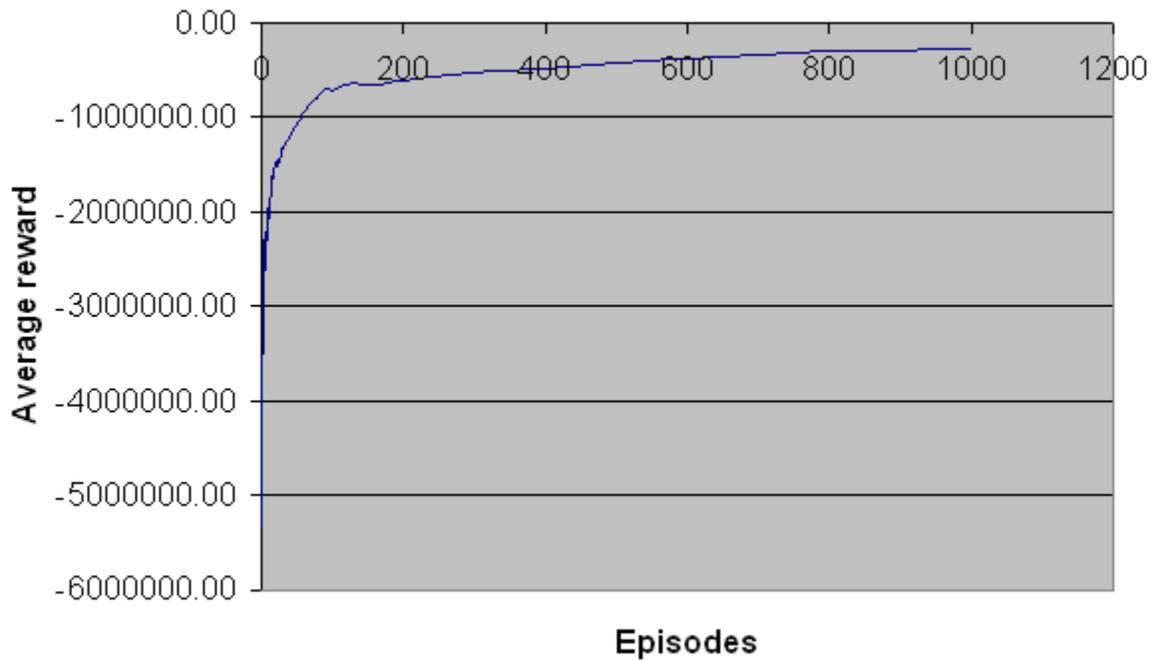


Figure B-10. Average reward per episode for the first 1000 episodes.

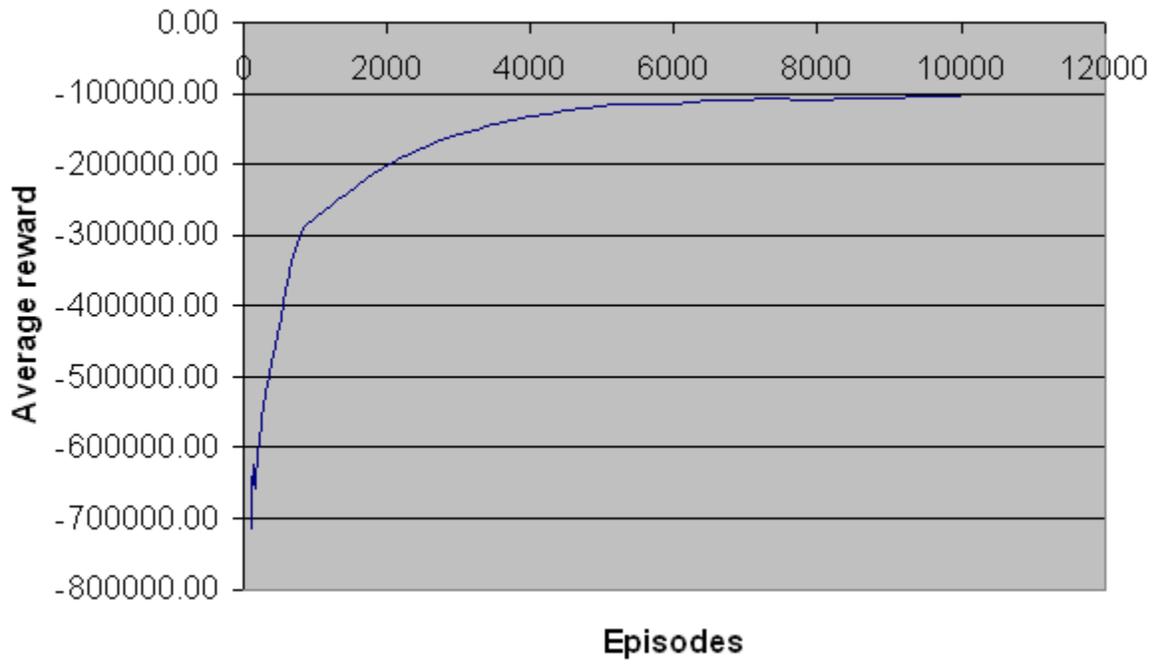


Figure B-11. Average reward per episode for the first 10000 episodes.

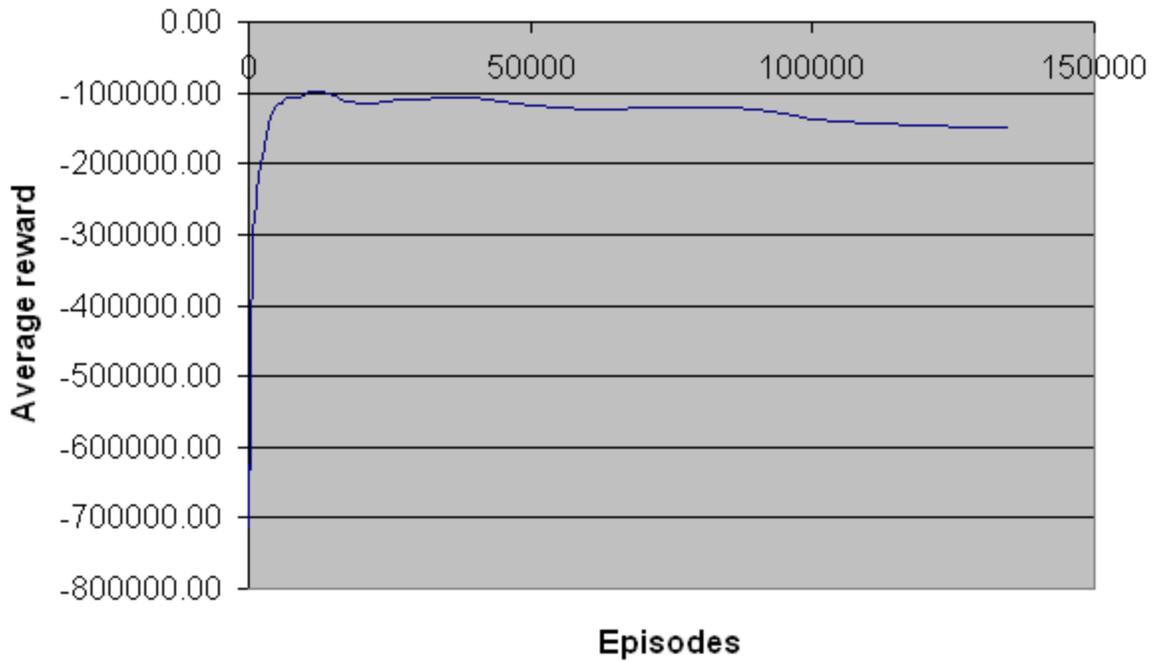


Figure B-12. Average reward per episode for all experimental episodes.

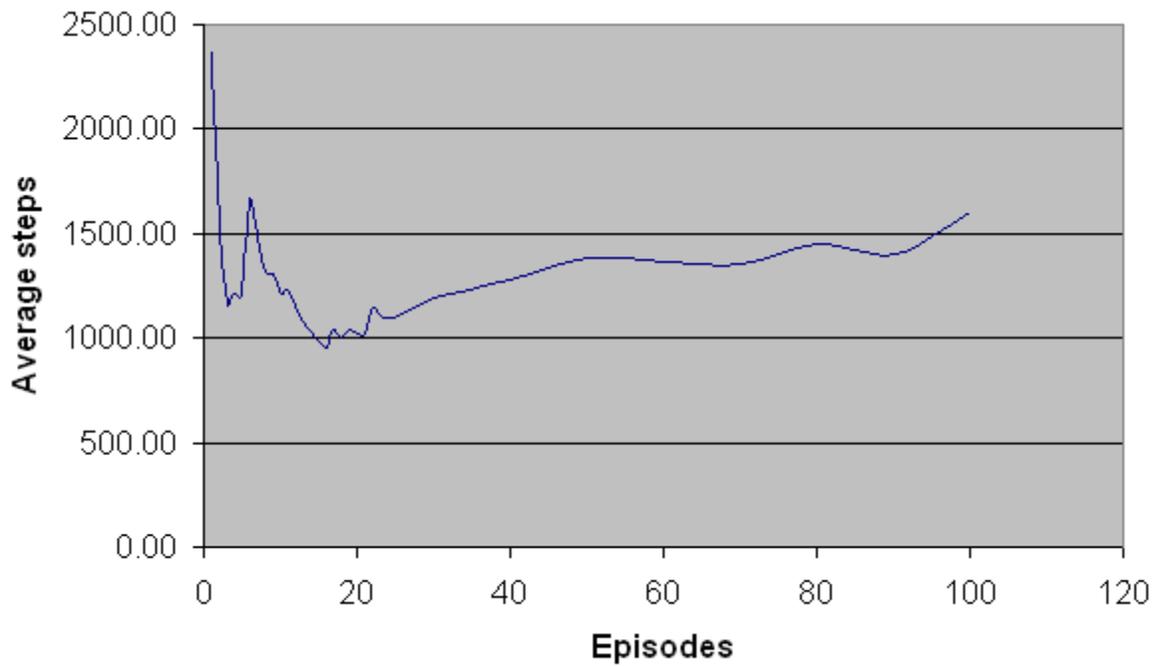


Figure B-13. Average number of steps per episode for the first 100 episodes.

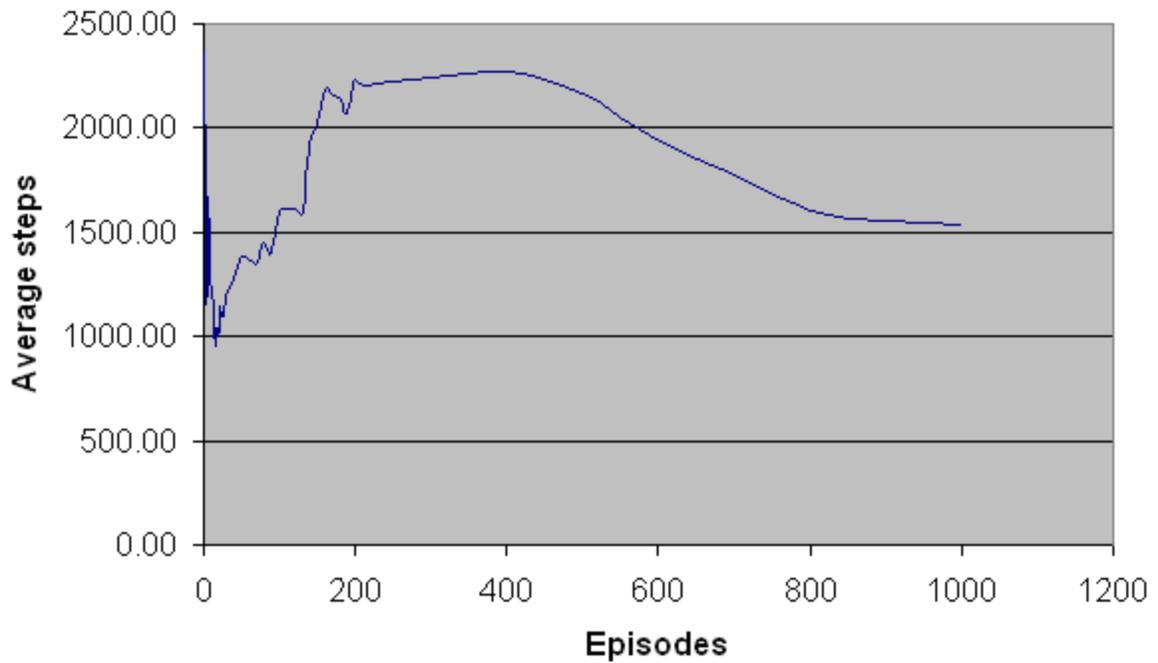


Figure B-14. Average number of steps per episode for the first 1000 episodes.

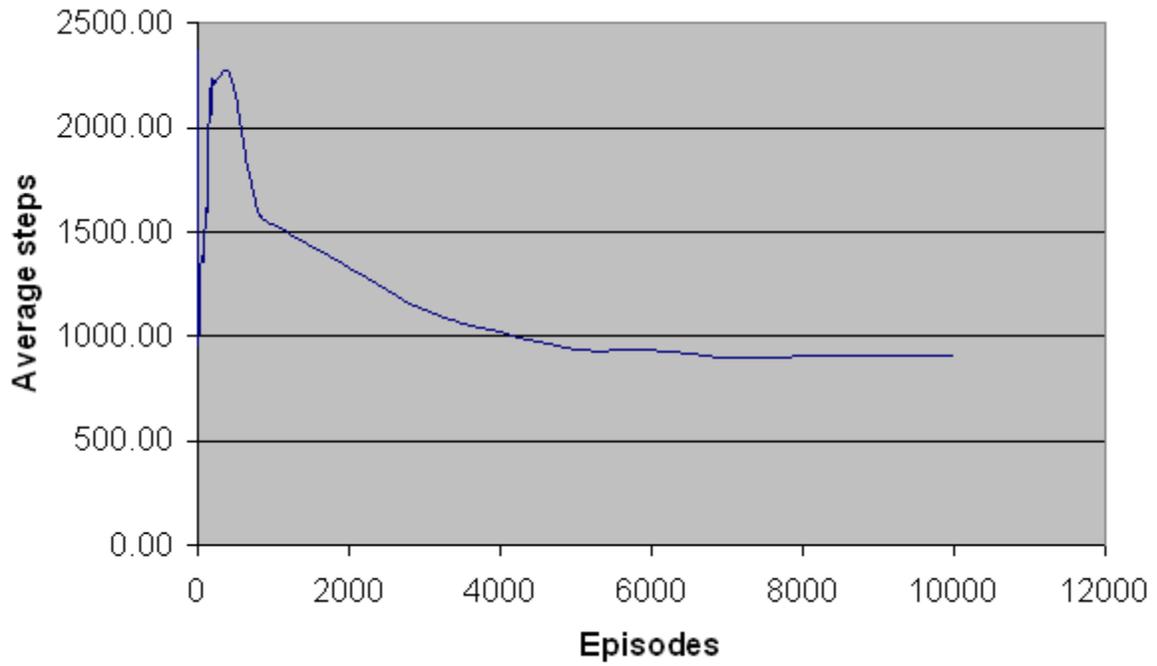


Figure B-15. Average number of steps per episode for the first 10000 episodes.

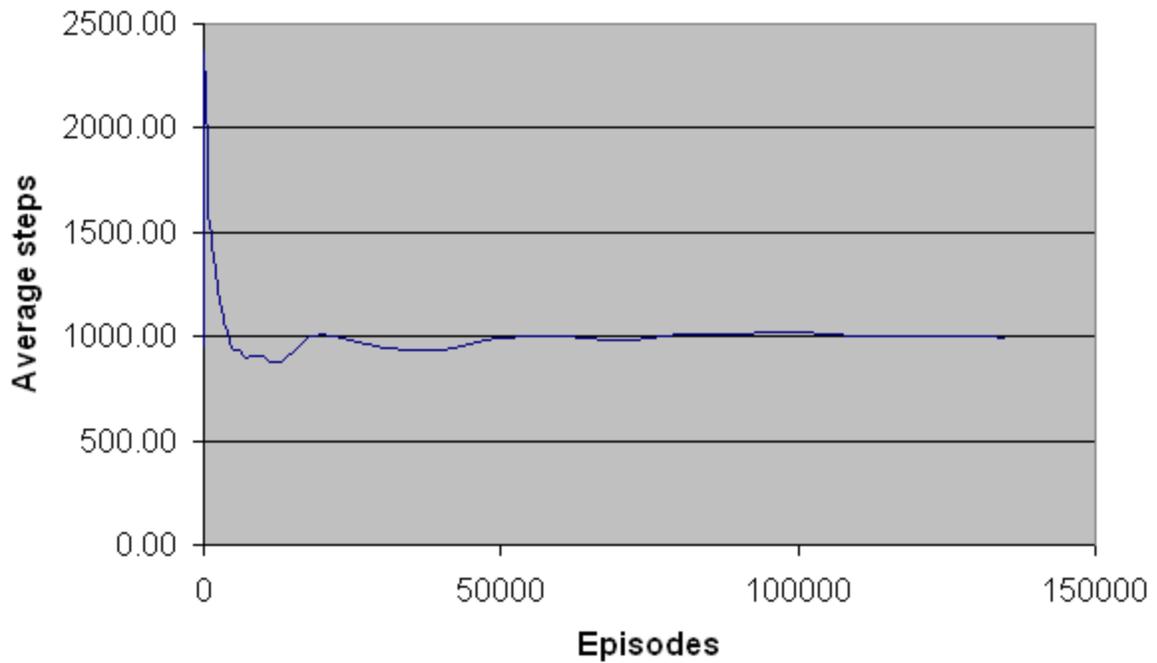


Figure B-16. Average number of steps per episode for all experimental episodes.

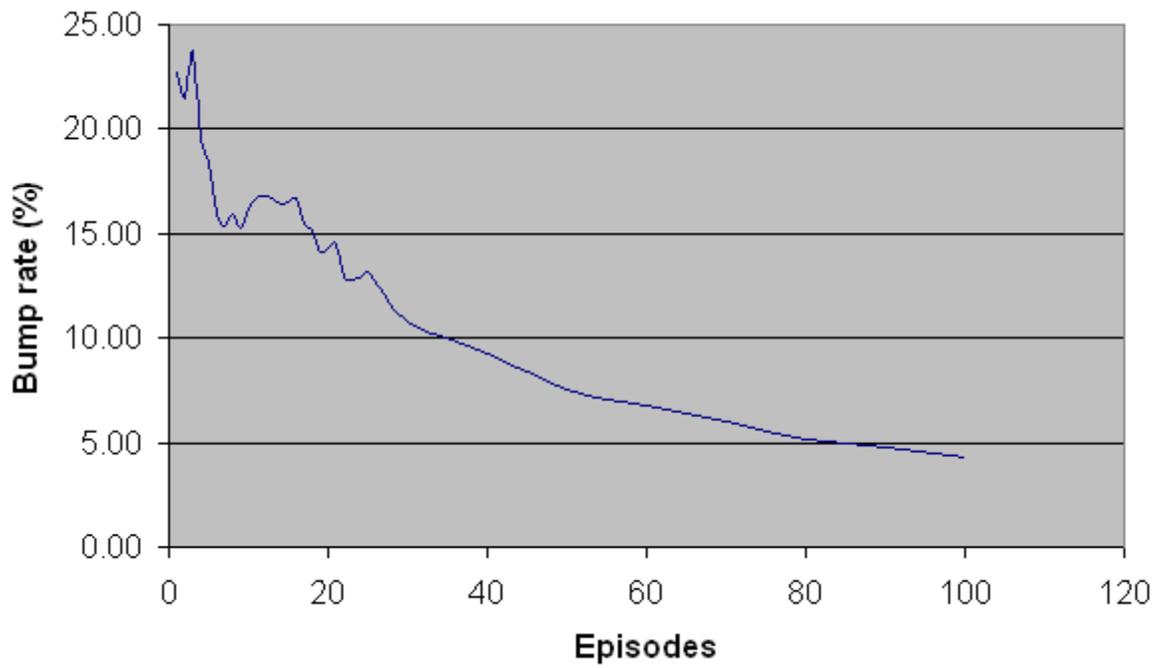


Figure B-17. Bump rate per episode for the first 100 episodes.

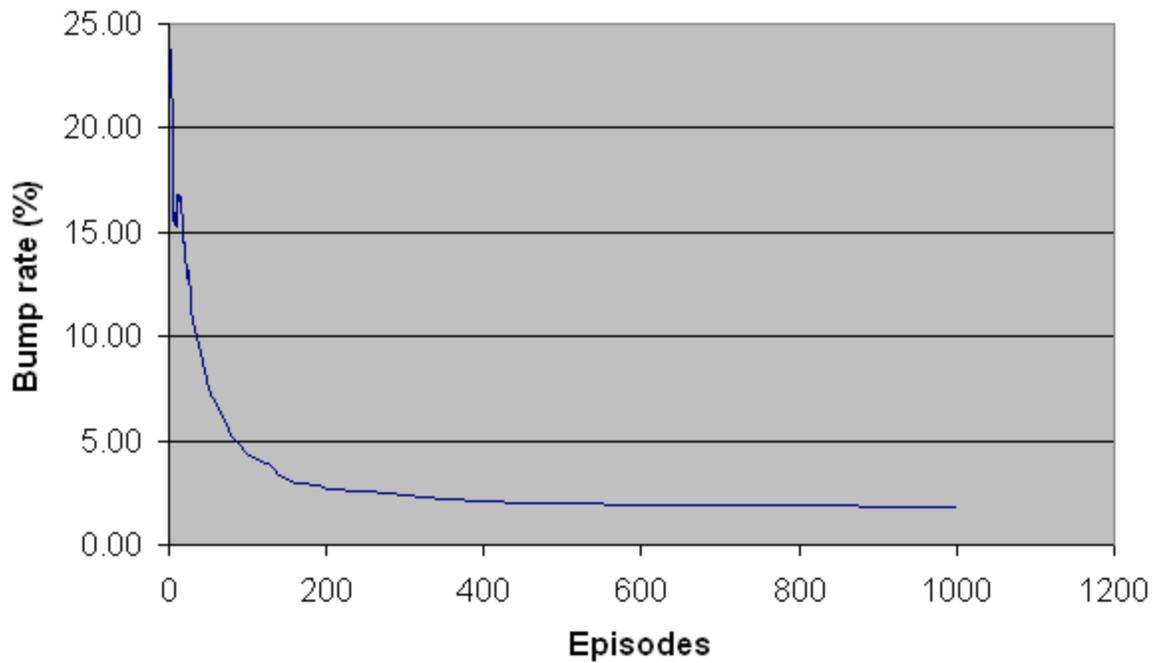


Figure B-18. Bump rate per episode for the first 1000 episodes.

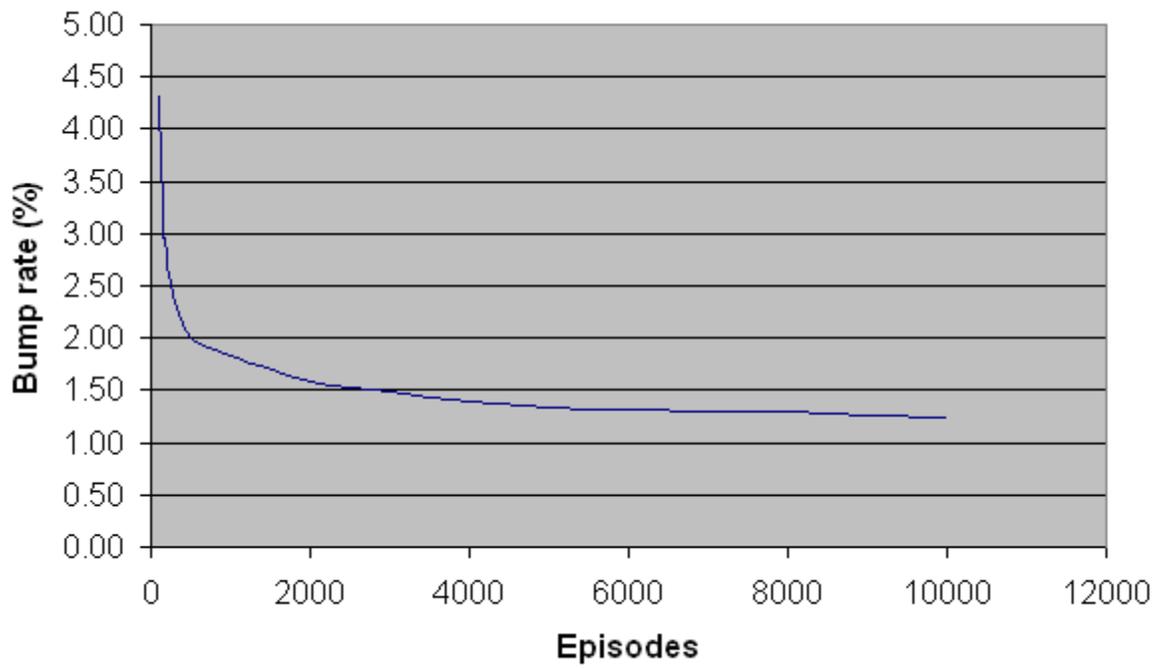


Figure B-19. Bump rate per episode for the first 10000 episodes.

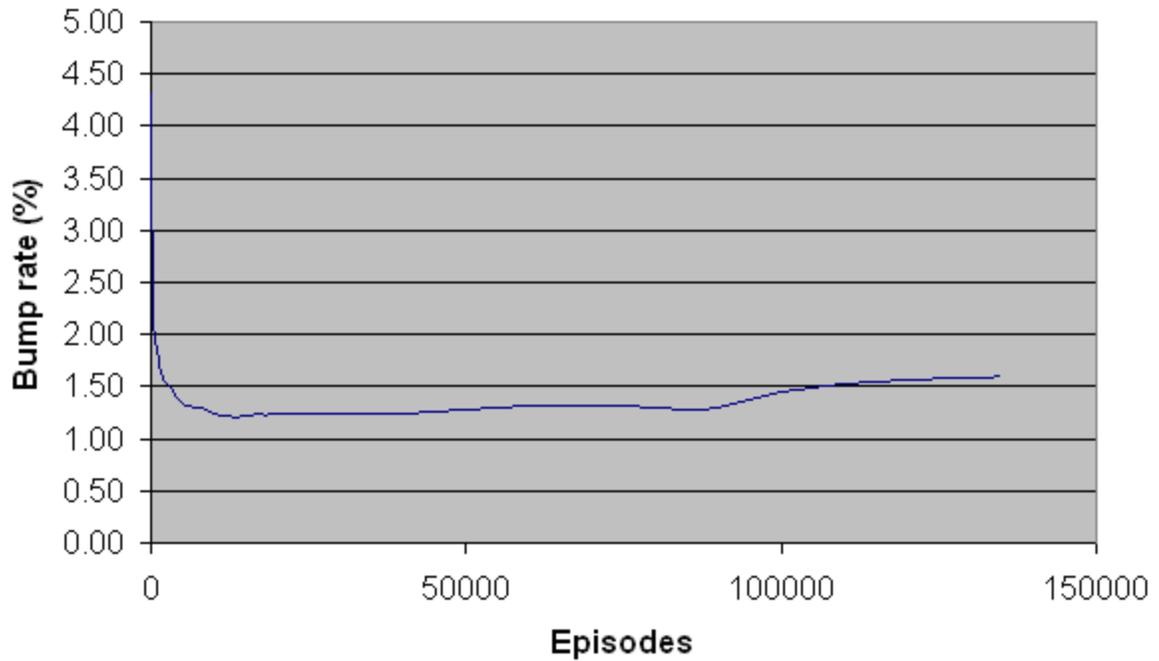


Figure B-20. Bump rate per episode for all experimental episodes.

APPENDIX C SIMULATOR CODE

```
// NOTE:
// Look for comment SMALL and ENDSMALL to note things commented out to make this a
small version.
// Look for comment DoubleQ for things changed to make a double Q-File
// Look for QFocus for the additions/changes made to add the front camera focus to the Q-table.
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.IO;
using System.Windows.Forms;

namespace ArenaSim
{
    public partial class ArenaSim : Form
    {
        // five degrees in radians for rotating the arena to prevent vertical lines
        float cosfive = (float)Math.Cos(Math.PI/36);
        float sinfive = (float)Math.Sin(Math.PI/36);
        float fivedegrees = (float)Math.PI / 36;
        // twopi is used for all instances of 2*Pi
        // NOTE: All calculations in this program use the range 0 to 2*Pi for a circle, and not -Pi to
Pi.
        float twopi = 2 * (float)Math.PI;
        // The following set x and y to permanent values so the use of x and y coordinates is more
logical
        int x = 0;
        int y = 1;

        // The following definitions should be changed to customize the arena and robot
        // RobotRadius is the radius of the robot
        float RobotRadius = 10;
        // BumpRadius is the number of inches past the edge of the robot you want to be registered
as a 'bump'
        float BumpRadius = 6;
        // The following list stores the endpoints for each line segment wall.
        // IMPORTANT NOTE: The function RobotAtEndOfWorld() currently requires that the
arena be a single straight hallway.
```

// If the arena is changed to another makeup, then RobotAtEndOfWorld must also be changed.

```
float[,] SegmentEndpoint = new float[,] { { 200, 250 }, { 200, 1000 }, { 1000, 1000 }, { 1000, 250 } };
```

// SMALL

```
// float[,] SegmentEndpoint = new float[,] { { 200, 200 }, { 200, 300 }, { 300, 300 }, { 300, 1932 }, { 200, 1932 }, { 200, 2032 }, { 488, 2032 }, { 488, 1932 }, { 388, 1932 }, { 388, 300 }, { 488, 300 }, { 488, 200 } };
```

// BOOGA

```
// float[,] SegmentEndpoint = new float[,] { { 388, 1000 }, { 388, 500 }, { 100, 500 }, { 100, 1000 } };
```

// NumEndpoints is the number of endpoints in SegmentEndpoint
int NumEndpoints;

// ReallyLongDistance is used in calculating distances for the IR sensors.
// If no wall is in sight, set the sensor value to ReallyLongDistance.
// This value should be at least double the longest dimension in the arena.
// This particular arena is about 2000 long, so ReallyLongDistance is being set to 5000.
float ReallyLongDistance = 5000;

// VeryHighSlope is used to prevent infinite slopes.
// It just needs to be a large number.
float VeryHighSlope = 5000;

// SonarArc is the arc of the sonar sensors in radians
// The arc of the sonar used is 55 degrees (from the SRF08 datasheet).
// NOTE: Use the image (beam.gif) of the sonar beam in the final paper.
float SonarArc = 55 * (float)Math.PI / 180;

// SonarSensors is an array of the sonar sensors.
// The values in SonarSensors are the angle of each sonar (in radians) from the front of the robot.

// Since all sensors are always in use, the order should not matter. All references to SonarSensors occur in loops that go through the entire array.

// The following values were approximated from the Koolio platform.

```
float[] SonarSensors = new float[] { 0, 2 * (float)Math.PI / 9, 4 * (float)Math.PI / 9, (float)Math.PI / 2, (float)Math.PI, -2 * (float)Math.PI / 9, -4 * (float)Math.PI / 9, -(float)Math.PI / 2 };
```

// NumSomars is the length of the array SonarSensors.
// It gets defined in the init function.

```

int NumSonars;

// CameraArc is the visible arc of the cameras
// From measurements, the arc of the cameras was estimated at approximately 45 degrees.
float CameraArc = 45 * (float)Math.PI / 180;

// CameraSensors is an array of the cameras.
// The values in CameraSensors are the angle of each camera (in radians) from the front of
the robot.
// Since all cameras are always in use, the order should not matter. All references to
CameraSensors occur in loops that go through the entire array.
float[] CameraSensors = new float[] { 0, -(float)Math.PI / 2, (float)Math.PI / 2 };

// NumCameras is the length of the array CameraSensors.
// It gets defined in the init function.
int NumCameras;

// NOTE: Remove the following block of comments if the CameraVision1/2/3 system
works
// These constants are used as codes for what the camera sees.
//int CameraNothing = 0;
//int CameraGoal = 1;
//int CameraEndOfWorld = 2;
//
// These are for what the camera currently sees.
// If a different number of cameras are being used, more must be added here.
// IMPORTANT: Since NumCameras can't be defined yet, the length of this array MUST be
changed manually if there are a different number of cameras.
//int[] CameraVision = new int[3] { 0, 0, 0 };

// These are for what the cameras currently see.
// IMPORTANT: More must be manually added or removed if there are a different number
of cameras.
// Format is CameraVision0/1/2[Goal seen, End of world seen]
bool[] CameraVision0 = new bool[] { false, false };
bool[] CameraVision1 = new bool[] { false, false };
bool[] CameraVision2 = new bool[] { false, false };
// These constants are used only as reference coordinates to CameraVision
int SeeGoal = 0;
int SeeEndOfWorld = 1;

// MaxCameraView is the maximum distance (in inches) that the camera can recognize the
points.
// From the data values, the cameras are accurate to approximately 100 inches.
// int MaxCameraView = 100;
// DoubleQ

```

```

int MaxCameraView = 1000;

// GoalPoint is the coordinates of the Goal used for the cameras.

// DoubleQ
float[] GoalPoint = new float[] { 600, 250 };

// float[] GoalPoint = new float[] { 350, 200 };

// SMALL
// float[] GoalPoint = new float[] { 300, 1806 };
// test the following: for a goal not so near the EoW
// float[] GoalPoint = new float[] { 300, 1500 };
// BOOGA
// float[] GoalPoint = new float[] { 100, 600 };

// EndOfWorldPoints is an array of the coordinates for the end of the world points used for
the cameras.
// Here is how they were calculated for the Koolio arena:
// Put 2 at the end of the hallway, evenly spaced. Since the width of the hall is 88, the points
are put 29 in from each direction with 30 between them.
// Near the end of the hall, put 2 more on each wall. The goal point is 126 from the end of
the wall. Make the two points 30 from the corner, then another 30 in. This leaves 66 between
the end of world marker and the goal.
// The order of the points should not matter since the code will always check all of them
without preference.

float[,] EndOfWorldPoints = new float[,] { { 5000, 5000 }, { 5000, 5001 } };

// /* SMALL
// float[,] EndOfWorldPoints = new float[,] { { 329, 2032 }, { 359, 2032 }, { 300, 1902 }, {
300, 1872 }, { 388, 1902 }, { 388, 1872 }, { 300, 360 }, { 300, 330 }, { 388, 360 }, { 388, 330 },
{ 329, 200 }, { 359, 200 } };
// ENDSMALL */

// NumEndOfWorld is the number of end of the world points. It is calculated in the init
function.
int NumEndOfWorld;

// StartLocation is the starting location of the robot.
// Note that this, RobotLocation, and OldLocation must be set to the same thing at the
beginning.
float[] StartLocation = new float[] { 600, 500 };

```

```

// float[] StartLocation = new float[] { 745, 500 };
// float[] StartLocation = new float[] { 455, 500 };
// float[] StartLocation = new float[] { 630, 500 };
// float[] StartLocation = new float[] { 570, 500 };
// float[] StartLocation = new float[] { 900, 340 };
// float[] StartLocation = new float[] { 900, 300 };
// float[] StartLocation = new float[] { 300, 340 };
// float[] StartLocation = new float[] { 300, 300 };
// float[] StartLocation = new float[] { 600, 340 };
// float[] StartLocation = new float[] { 240, 400 };
// float[] StartLocation = new float[] { 328, 744 };
// float[] StartLocation = new float[] { 346, 744 };
// SMALL
// float[] StartLocation = new float[] { 363, 744 };

// RobotLocation is the current location of the center of the robot.
float[] RobotLocation = new float[] { 600, 500 };
// float[] RobotLocation = new float[] { 745, 500 };
// float[] RobotLocation = new float[] { 455, 500 };
// float[] RobotLocation = new float[] { 630, 500 };
// float[] RobotLocation = new float[] { 570, 500 };
// float[] RobotLocation = new float[] { 900, 340 };
// float[] RobotLocation = new float[] { 900, 300 };
// float[] RobotLocation = new float[] { 300, 340 };
// float[] RobotLocation = new float[] { 300, 300 };
// float[] RobotLocation = new float[] { 600, 340 };
// float[] RobotLocation = new float[] { 240, 400 };
// float[] RobotLocation = new float[] { 328, 744 };
// float[] RobotLocation = new float[] { 346, 744 };
// SMALL
// float[] RobotLocation = new float[] { 363, 744 };

// OldLocation is the location of the previous timestep.
float[] OldLocation = new float[] { 600, 500 };
// float[] OldLocation = new float[] { 745, 500 };
// float[] OldLocation = new float[] { 455, 500 };
// float[] OldLocation = new float[] { 630, 500 };
// float[] OldLocation = new float[] { 570, 500 };
// float[] OldLocation = new float[] { 900, 340 };
// float[] OldLocation = new float[] { 900, 300 };
// float[] OldLocation = new float[] { 300, 340 };
// float[] OldLocation = new float[] { 300, 300 };
// float[] OldLocation = new float[] { 600, 340 };
// float[] OldLocation = new float[] { 240, 400 };
// float[] OldLocation = new float[] { 328, 744 };
// float[] OldLocation = new float[] { 346, 744 };

```

```

// SMALL
// float[] OldLocation = new float[] { 363, 744 };

// SMALL
// StartOrientation is a constant.
float StartOrientation = 3 * (float)Math.PI / 2;
// float StartOrientation = 4 * (float)Math.PI / 3;
// float StartOrientation = 5 * (float)Math.PI / 3;
// float StartOrientation = (float)Math.PI;
// float StartOrientation = 0;
// float StartOrientation = -(float)Math.PI / 3;
// float StartOrientation = -2 * (float)Math.PI / 3;
// float StartOrientation = -(float)Math.PI / 2;
// float StartOrientation = (float)Math.PI / 2;
// float StartOrientation = 5 * (float)Math.PI / 8;

// RobotOrientation is the direction the robot is facing in radians
// Start at 180 degrees
float RobotOrientation = 3 * (float)Math.PI / 2;
// float RobotOrientation = 4 * (float)Math.PI / 3;
// float RobotOrientation = 5 * (float)Math.PI / 3;
// float RobotOrientation = (float)Math.PI;
// float RobotOrientation = 0;
// float RobotOrientation = -(float)Math.PI / 3;
// float RobotOrientation = -2 * (float)Math.PI / 3;
// float RobotOrientation = -(float)Math.PI / 2;
// float RobotOrientation = (float)Math.PI / 2;
// SMALL
// float RobotOrientation = 5 * (float)Math.PI / 8;

// OldOrientation is the robot's orientation of the previous timestep.
float OldOrientation = 3 * (float)Math.PI / 2;
// float OldOrientation = 4 * (float)Math.PI / 3;
// float OldOrientation = 5 * (float)Math.PI / 3;
// float OldOrientation = (float)Math.PI;
// float OldOrientation = 0;
// float OldOrientation = -(float)Math.PI / 3;
// float OldOrientation = -2 * (float)Math.PI / 3;
// float OldOrientation = -(float)Math.PI / 2;
// float OldOrientation = (float)Math.PI / 2;
// SMALL
// float OldOrientation = 5 * (float)Math.PI / 8;

// The following are to draw circles on the arena to indicate to the user where the start point
and goal are
float[] StartDot = new float[] { 600, 1000 };

```

```

float[] GoalDot = new float[] { 600, 250 };
// float[] StartDot = new float[] { 350, 500 };
// float[] GoalDot = new float[] { 350, 200 };
// SMALL
// float[] StartDot = new float[] { 388, 744 };
// float[] GoalDot = new float[] { 300, 1806 };
// float[] GoalDot = new float[] { 300, 1500 };
// BOOGA
// float[] GoalDot = new float[] { 100, 600 };

// Trail_Toggle is the toggle for having the trail show for the robot.
bool Trail_Toggle = false;

// VerySmallNumber is used to prevent division by 0 in the RobotContacting function.
float VerySmallNumber = 0.00000000001F;

// Dist_Bump, Dist_Close, Dist_Medium, and Dist_Far are used for the sensors.
// To make the learning look-up table smaller, instead of using every possible value of all
sensors, the sensor readings will be categorized into Close, Medium, and Far. Bump is only used
for the sonars, not the cameras.
// Each sensor will have different boundaries for the three distance categories.
int Dist_Bump = 3;
int Dist_Close = 0;
int Dist_Medium = 1;
int Dist_Far = 2;
// NOTE: commented out for smaller Q-table size
// int Dist_Very_Far = 3;

// The following are values for the different readings from the compass.
// Because the compass is so unreliable, such broad readings are fine.
// The cardinal directions are arbitrary. 'North' refers to 'up' on the map and the others
follow accordingly.
// North is a compass reading from  $\text{Pi}/4$  to  $3*\text{Pi}/4$ 
// East is a compass reading from  $-\text{Pi}/4$  to  $\text{Pi}/4$ 
// South is a compass reading from  $-3*\text{Pi}/4$  to  $-\text{Pi}/4$ 
// West is a compass reading from  $-\text{Pi}$  to  $-3*\text{Pi}/4$  and from  $3*\text{Pi}/4$  to  $\text{Pi}$ 
// NOTE: commented out for smaller Q-table size
// int Heading_North = 0;
// int Heading_East = 1;
// int Heading_South = 2;
// int Heading_West = 3;

// State is an array that indicates the sensor state.
// It uses the following indices for sensors:

```

```

// Camera to goal:
int State_Goal0 = 0;
int State_Goal1 = 1;
int State_Goal2 = 2;
// Camera to end of world
int State_EndOfWorld0 = 3;
int State_EndOfWorld1 = 4;
int State_EndOfWorld2 = 5;
// Sonars
int State_Sonar0 = 6;
// NOTE: commented out for smaller Q-table size
/* int State_Sonar1 = 7;
int State_Sonar2 = 8;
int State_Sonar3 = 9;
int State_Sonar4 = 10;
int State_Sonar5 = 11;
int State_Sonar6 = 12;
int State_Sonar7 = 13;*/
// NOTE: added for smaller Q-table size
int State_Sonar1_2 = 7;
int State_Sonar3 = 8;
int State_Sonar4 = 9;
int State_Sonar5_6 = 10;
int State_Sonar7 = 11;
// QFocus
// Focus of the front camera (State_Goal0)
int State_Focus0 = 12;
int State_Focus1 = 13;
int State_Focus2 = 14;
// State_Focus0 can have 4 values: CameraFocusCenter, CameraFocusRight,
CameraFocusLeft, and CameraFocusNone.
// NOTE NOTE NOTE: this changes the size of the Q-table! Note so in the chapter that
talks about maximum size.
// end QFocus

// Compass
// NOTE: commented out for smaller Q-table size
// int State_Compass = 14;

// NOTE: the following comments invalid for smaller Q-table size
// The format for the state is as follows:
// State[sensor reference]=sensor range value
// There are three cameras. Camera to Goal has 4 possible values: Dist_Close,
Dist_Medium, Dist_Far, and Dist_Very_Far.
// There are three cameras. Camera to End of World has 4 possible values: Dist_Close,
Dist_Medium, Dist_Far, and Dist_Very_Far.

```

```
// There are eight sonars. Sonar has 5 possible values: Dist_Close, Dist_Medium, Dist_Far, Dist_Very_Far, and Dist_Bump.
```

```
// There is one compass. Compass has 4 possible values: Heading_North, Heading_East, Heading_South, and Heading_West.
```

```
// NOTE: If the total number of sensors changes, the size of this will have to change as well.
```

```
// NOTE: the following comments used for smaller Q-table size
```

```
// The format for the state is as follows:
```

```
// State[sensor reference]=sensor range value
```

```
// There are three cameras. Camera to Goal has 3 possible values: Dist_Close, Dist_Medium, and Dist_Far.
```

```
// There are three cameras. Camera to End of World has 3 possible values: Dist_Close, Dist_Medium, and Dist_Far.
```

```
// There are six sonars. Sonar has 4 possible values: Dist_Close, Dist_Medium, Dist_Far, and Dist_Bump.
```

```
// QFocus
```

```
// Comment out the following:
```

```
/*
```

```
int[] State = new int[12];
```

```
// NextState is the next state.
```

```
// It is used for the Q-learning formula.
```

```
int[] NextState = new int[12];
```

```
*/
```

```
// Because of the addition of State_Focus0, State and NextState need to be 1 bigger.
```

```
// Edit: 2 more bigger because of State_Focus being on all three cameras.
```

```
int[] State = new int[15];
```

```
int[] NextState = new int[15];
```

```
// Focus has 4 possible values: CameraFocusCenter, CameraFocusLeft, CameraFocusRight, and CameraFocusNone.
```

```
// end QFocus
```

```
int NumStates = 15;
```

```
// Reward is the value of the reward for the current state and action choice.
```

```
int Reward;
```

```
// The following are rewards for different states.
```

```
/* int Reward_Goal = 1000;
```

```
int Reward_Bump = -10000;
```

```
int Reward_Stop = -25;
```

```
int Reward_Turn = 2;
```

```
int Reward_Reverse = 1;
```

```
int Reward_Forward = 3;
```

```
int Reward_End_of_World = -50000;
```

```
int Reward_Center_Forward = 5;
```

```

int Reward_Center_Reverse = 3;
int Reward_Center_Turn = 4;

// Q2
int Reward2_Goal = 2000;
int Reward2_Bump = -5000;
int Reward2_Stop = -50;
int Reward2_Turn = 4;
int Reward2_Reverse = 2;
int Reward2_Forward = 6;
int Reward2_End_of_World = -30000;
int Reward2_Center_Forward = 10;
int Reward2_Center_Reverse = 6;
int Reward2_Center_Turn = 8;

// Q3
// The tier 3 rewards are only for when the goal can be seen in the Center fous of the Front
camera.
int Reward3_Goal = 4000;
int Reward3_Bump = -2500;
int Reward3_Stop = -10;
int Reward3_Turn = 20;
int Reward3_Reverse = 2;
int Reward3_Forward = 30;
int Reward3_End_of_World = -20000;
int Reward3_Center_Forward = 50;
int Reward3_Center_Reverse = 6;
int Reward3_Center_Turn = 30;
*/
// Following rewards were used in 'start from nothing 5 percent epsilon new rewards' folder
/*
// New reward set testing!
int Reward_Goal = 1000;
int Reward_Bump = -10000;
int Reward_Stop = -25;
int Reward_Turn = -2;
int Reward_Reverse = -3;
int Reward_Forward = -1;
int Reward_End_of_World = -50000;
int Reward_Center_Forward = -1;
int Reward_Center_Reverse = -3;
int Reward_Center_Turn = -2;
// Q2
int Reward2_Goal = 2000;
int Reward2_Bump = -5000;
int Reward2_Stop = -25;
int Reward2_Turn = 2;

```

```

int Reward2_Reverse = 1;
int Reward2_Forward = 3;
int Reward2_End_of_World = -30000;
int Reward2_Center_Forward = 3;
int Reward2_Center_Reverse = 1;
int Reward2_Center_Turn = 2;
// Q3
// The tier 3 rewards are only for when the goal can be seen in the Center fous of the Front
camera.
int Reward3_Goal = 4000;
int Reward3_Bump = -2500;
int Reward3_Stop = -25;
int Reward3_Turn = 3;
int Reward3_Reverse = 2;
int Reward3_Forward = 5;
int Reward3_End_of_World = -20000;
int Reward3_Center_Forward = 5;
int Reward3_Center_Reverse = 2;
int Reward3_Center_Turn = 3;
*/
/* // Following used in 'start from nothing 5 percent epsilon new rewards 2' 3 and 4 folder
int Reward_Goal = 1000;
int Reward_Bump = -10000;
int Reward_Stop = -25;
int Reward_Turn = -2;
int Reward_Big_Turn = -2;
int Reward_Reverse = -3;
int Reward_Forward = -1;
int Reward_End_of_World = -50000;
int Reward_Center_Forward = -1;
int Reward_Center_Reverse = -2;
int Reward_Center_Turn = -1;
int Reward_Center_Big_Turn = -1;
// Q2
int Reward2_Goal = 2000;
int Reward2_Bump = -5000;
int Reward2_Stop = -25;
int Reward2_Turn = 3;
int Reward2_Big_Turn = 3;
int Reward2_Reverse = 1;
int Reward2_Forward = 3;
int Reward2_End_of_World = -30000;
int Reward2_Center_Forward = 3;
int Reward2_Center_Reverse = 1;
int Reward2_Center_Turn = 3;
int Reward2_Center_Big_Turn = 3;

```

```

// Q3
// The tier 3 rewards are only for when the goal can be seen in the Center fous of the Front
camera.
int Reward3_Goal = 4000;
int Reward3_Bump = -2500;
int Reward3_Stop = -25;
int Reward3_Turn = 4;
int Reward3_Big_Turn = 4;
int Reward3_Reverse = 2;
int Reward3_Forward = 5;
int Reward3_End_of_World = -20000;
int Reward3_Center_Forward = 5;
int Reward3_Center_Reverse = 2;
int Reward3_Center_Turn = 4;
int Reward3_Center_Big_Turn = 4;
*/
/* // Following used in 'start from nothing 5 percent epsilon new rewards 5' folder
int Reward_Goal = 1000;
int Reward_Bump = -10000;
int Reward_Stop = -25;
int Reward_Turn = -2;
int Reward_Big_Turn = -2;
int Reward_Reverse = -3;
int Reward_Forward = -1;
int Reward_End_of_World = -50000;
int Reward_Center_Forward = -1;
int Reward_Center_Reverse = -2;
int Reward_Center_Turn = -2;
int Reward_Center_Big_Turn = -2;
// Q2
int Reward2_Goal = 2000;
int Reward2_Bump = -5000;
int Reward2_Stop = -25;
int Reward2_Turn = 2;
int Reward2_Big_Turn = 2;
int Reward2_Reverse = 1;
int Reward2_Forward = 2;
int Reward2_End_of_World = -30000;
int Reward2_Center_Forward = 2;
int Reward2_Center_Reverse = 1;
int Reward2_Center_Turn = 2;
int Reward2_Center_Big_Turn = 2;
// Q3
// The tier 3 rewards are only for when the goal can be seen in the Center fous of the Front
camera.
int Reward3_Goal = 4000;

```

```

int Reward3_Bump = -2500;
int Reward3_Stop = -25;
int Reward3_Turn = -1;
int Reward3_Big_Turn = -1;
int Reward3_Reverse = -1;
int Reward3_Forward = 5;
int Reward3_End_of_World = -20000;
int Reward3_Center_Forward = 5;
int Reward3_Center_Reverse = -1;
int Reward3_Center_Turn = -1;
int Reward3_Center_Big_Turn = -1;
*/
/* // Following used in '5 percent epsilon new rewards'
int Reward_Goal = 1000;
int Reward_Bump = -10000;
int Reward_Stop = -25;
int Reward_Turn = -2;
int Reward_Big_Turn = -2;
int Reward_Reverse = -3;
int Reward_Forward = -1;
int Reward_End_of_World = -50000;
int Reward_Center_Forward = -1;
int Reward_Center_Reverse = -2;
int Reward_Center_Turn = -2;
int Reward_Center_Big_Turn = -2;
// Q2
int Reward2_Goal = 2000;
int Reward2_Bump = -5000;
int Reward2_Stop = -25;
int Reward2_Turn = 2;
int Reward2_Big_Turn = 2;
int Reward2_Reverse = 1;
int Reward2_Forward = 2;
int Reward2_End_of_World = -30000;
int Reward2_Center_Forward = 2;
int Reward2_Center_Reverse = 1;
int Reward2_Center_Turn = 2;
int Reward2_Center_Big_Turn = 2;
// Q3
// The tier 3 rewards are only for when the goal can be seen in the Center fous of the Front
camera.
int Reward3_Goal = 4000;
int Reward3_Bump = -2500;
int Reward3_Stop = -25;
int Reward3_Turn = 2;
int Reward3_Big_Turn = 2;

```

```

int Reward3_Reverse = -1;
int Reward3_Forward = 5;
int Reward3_End_of_World = -20000;
int Reward3_Center_Forward = 5;
int Reward3_Center_Reverse = -1;
int Reward3_Center_Turn = 2;
int Reward3_Center_Big_Turn = 2;
*/
/* // Following used in 'newer rewards'
int Reward_Goal = 1000;
int Reward_Bump = -10000;
int Reward_Stop = -25;
int Reward_Turn = -2;
int Reward_Big_Turn = -2;
int Reward_Reverse = -3;
int Reward_Forward = -1;
int Reward_End_of_World = -50000;
int Reward_Center_Forward = -1;
int Reward_Center_Reverse = -2;
int Reward_Center_Turn = -2;
int Reward_Center_Big_Turn = -2;
// Q2
int Reward2_Goal = 2000;
int Reward2_Bump = -5000;
int Reward2_Stop = -25;
int Reward2_Turn = 2;
int Reward2_Big_Turn = 2;
int Reward2_Reverse = 1;
int Reward2_Forward = 2;
int Reward2_End_of_World = -30000;
int Reward2_Center_Forward = 2;
int Reward2_Center_Reverse = 1;
int Reward2_Center_Turn = 2;
int Reward2_Center_Big_Turn = 2;
// Q3
// The tier 3 rewards are only for when the goal can be seen in the Center fous of the Front
camera.
int Reward3_Goal = 4000;
int Reward3_Bump = -2500;
int Reward3_Stop = -25;
int Reward3_Turn = 4;
int Reward3_Big_Turn = 4;
int Reward3_Reverse = -1;
int Reward3_Forward = 5;
int Reward3_End_of_World = -20000;
int Reward3_Center_Forward = 5;

```

```

int Reward3_Center_Reverse = -1;
int Reward3_Center_Turn = 4;
int Reward3_Center_Big_Turn = 4;
*/
/* // Following used in 'newer rewards 2'
int Reward_Goal = 1000;
int Reward_Bump = -10000;
int Reward_Stop = -25;
int Reward_Turn = -2;
int Reward_Big_Turn = -2;
int Reward_Reverse = -3;
int Reward_Forward = -1;
int Reward_End_of_World = -50000;
int Reward_Center_Forward = -1;
int Reward_Center_Reverse = -2;
int Reward_Center_Turn = -2;
int Reward_Center_Big_Turn = -2;
// Q2
int Reward2_Goal = 2000;
int Reward2_Bump = -5000;
int Reward2_Stop = -25;
int Reward2_Turn = 3;
int Reward2_Big_Turn = 3;
int Reward2_Reverse = 1;
int Reward2_Forward = 2;
int Reward2_End_of_World = -30000;
int Reward2_Center_Forward = 2;
int Reward2_Center_Reverse = 1;
int Reward2_Center_Turn = 3;
int Reward2_Center_Big_Turn = 3;
// Q3
// The tier 3 rewards are only for when the goal can be seen in the Center focus of the Front
camera.
int Reward3_Goal = 4000;
int Reward3_Bump = -2500;
int Reward3_Stop = -25;
int Reward3_Turn = 4;
int Reward3_Big_Turn = 4;
int Reward3_Reverse = -1;
int Reward3_Forward = 5;
int Reward3_End_of_World = -20000;
int Reward3_Center_Forward = 5;
int Reward3_Center_Reverse = -1;
int Reward3_Center_Turn = 4;
int Reward3_Center_Big_Turn = 4;
*/

```

```

/*    // Following used in 'newer rewards 3' and 'newer rewards 6'
int Reward_Goal = 1000;
int Reward_Bump = -10000;
int Reward_Stop = -25;
int Reward_Turn = -2;
int Reward_Big_Turn = -2;
int Reward_Reverse = -3;
int Reward_Forward = -1;
int Reward_End_of_World = -50000;
int Reward_Center_Forward = -1;
int Reward_Center_Reverse = -2;
int Reward_Center_Turn = -2;
int Reward_Center_Big_Turn = -2;
// Q2
int Reward2_Goal = 2000;
int Reward2_Bump = -5000;
int Reward2_Stop = -25;
int Reward2_Turn = 2;
int Reward2_Big_Turn = 2;
int Reward2_Reverse = 1;
int Reward2_Forward = 2;
int Reward2_End_of_World = -30000;
int Reward2_Center_Forward = 2;
int Reward2_Center_Reverse = 1;
int Reward2_Center_Turn = 2;
int Reward2_Center_Big_Turn = 2;
// Q3
// The tier 3 rewards are only for when the goal can be seen in the Center fous of the Front
camera.
int Reward3_Goal = 4000;
int Reward3_Bump = -2500;
int Reward3_Stop = -25;
int Reward3_Turn = 4;
int Reward3_Big_Turn = 4;
int Reward3_Reverse = -1;
int Reward3_Forward = 5;
int Reward3_End_of_World = -20000;
int Reward3_Center_Forward = 5;
int Reward3_Center_Reverse = -1;
int Reward3_Center_Turn = 4;
int Reward3_Center_Big_Turn = 4;
*/
/*    // Following used in 'newer rewards 4' and 'newer rewards 5'
int Reward_Goal = 1000;
int Reward_Bump = -10000;
int Reward_Stop = -25;

```

```

int Reward_Turn = -2;
int Reward_Big_Turn = -2;
int Reward_Reverse = -3;
int Reward_Forward = -1;
int Reward_End_of_World = -50000;
int Reward_Center_Forward = -1;
int Reward_Center_Reverse = -2;
int Reward_Center_Turn = -2;
int Reward_Center_Big_Turn = -2;
// Q2
int Reward2_Goal = 2000;
int Reward2_Bump = -5000;
int Reward2_Stop = -25;
int Reward2_Turn = 2;
int Reward2_Big_Turn = 2;
int Reward2_Reverse = 1;
int Reward2_Forward = 2;
int Reward2_End_of_World = -30000;
int Reward2_Center_Forward = 2;
int Reward2_Center_Reverse = 1;
int Reward2_Center_Turn = 2;
int Reward2_Center_Big_Turn = 2;
// Q3
// The tier 3 rewards are only for when the goal can be seen in the Center focus of the Front
camera.
int Reward3_Goal = 4000;
int Reward3_Bump = -2500;
int Reward3_Stop = -25;
int Reward3_Turn = 7;
int Reward3_Big_Turn = 7;
int Reward3_Reverse = -1;
int Reward3_Forward = 10;
int Reward3_End_of_World = -20000;
int Reward3_Center_Forward = 10;
int Reward3_Center_Reverse = -1;
int Reward3_Center_Turn = 7;
int Reward3_Center_Big_Turn = 7;
*/
/* // Following used in 'newer rewards 7'
int Reward_Goal = 1000;
int Reward_Bump = -10000;
int Reward_Stop = -25;
int Reward_Turn = -2;
int Reward_Big_Turn = -2;
int Reward_Reverse = -3;
int Reward_Forward = -1;

```

```

int Reward_End_of_World = -50000;
int Reward_Center_Forward = -1;
int Reward_Center_Reverse = -2;
int Reward_Center_Turn = -2;
int Reward_Center_Big_Turn = -2;
// Q2
int Reward2_Goal = 2000;
int Reward2_Bump = -5000;
int Reward2_Stop = -25;
int Reward2_Turn = 4;
int Reward2_Big_Turn = 4;
int Reward2_Reverse = 1;
int Reward2_Forward = 2;
int Reward2_End_of_World = -30000;
int Reward2_Center_Forward = 2;
int Reward2_Center_Reverse = 1;
int Reward2_Center_Turn = 4;
int Reward2_Center_Big_Turn = 4;
// Q3
// The tier 3 rewards are only for when the goal can be seen in the Center fous of the Front
camera.
int Reward3_Goal = 4000;
int Reward3_Bump = -2500;
int Reward3_Stop = -25;
int Reward3_Turn = 7;
int Reward3_Big_Turn = 7;
int Reward3_Reverse = -1;
int Reward3_Forward = 10;
int Reward3_End_of_World = -20000;
int Reward3_Center_Forward = 10;
int Reward3_Center_Reverse = -1;
int Reward3_Center_Turn = 7;
int Reward3_Center_Big_Turn = 7;
*/
// Following used in 'newer rewards 8'
int Reward_Goal = 1000;
int Reward_Bump = -10000;
int Reward_Stop = -25;
int Reward_Turn = -2;
int Reward_Big_Turn = -2;
int Reward_Reverse = -3;
int Reward_Forward = -1;
int Reward_End_of_World = -50000;
int Reward_Center_Forward = -1;
int Reward_Center_Reverse = -2;
int Reward_Center_Turn = -2;

```

```

int Reward_Center_Big_Turn = -2;
// Q2
// The tier 2 rewards are for when the goal is seen (CanSeeGoal)
int Reward2_Goal = 2000;
int Reward2_Bump = -5000;
int Reward2_Stop = -25;
int Reward2_Turn = 2;
int Reward2_Big_Turn = 2;
int Reward2_Reverse = 1;
int Reward2_Forward = 2;
int Reward2_End_of_World = -30000;
int Reward2_Center_Forward = 2;
int Reward2_Center_Reverse = 1;
int Reward2_Center_Turn = 2;
int Reward2_Center_Big_Turn = 2;
// Q2.5
// Tier 2.5 are for when the goal is seen in the front.
int Reward25_Goal = 4000;
int Reward25_Bump = -2500;
int Reward25_Stop = -25;
int Reward25_Turn = 5;
int Reward25_Big_Turn = 5;
int Reward25_Reverse = 1;
int Reward25_Forward = 5;
int Reward25_End_of_World = -20000;
int Reward25_Center_Forward = 5;
int Reward25_Center_Reverse = 1;
int Reward25_Center_Turn = 5;
int Reward25_Center_Big_Turn = 5;
// Q3
// The tier 3 rewards are only for when the goal can be seen in the Center focus of the Front
camera. (GoalFrontCenter)
int Reward3_Goal = 4000;
int Reward3_Bump = -2500;
int Reward3_Stop = -25;
int Reward3_Turn = 7;
int Reward3_Big_Turn = 7;
int Reward3_Reverse = -1;
int Reward3_Forward = 10;
int Reward3_End_of_World = -20000;
int Reward3_Center_Forward = 10;
int Reward3_Center_Reverse = -1;
int Reward3_Center_Turn = 7;
int Reward3_Center_Big_Turn = 7;

// GOAL PROXIMITY

```

```

// int Reward_Goal_Side = 10;
// int Reward_Goal_Front = 10;

// There are five possible actions that the robot can take.
// For simplicity of the simulator, there is no 'arc forward left' or 'arc forward right' action.
// Excluding these also minimizes the size of the Q-table.
int Action_Forward = 0;
int Action_Reverse = 1;
int Action_Rotate_Left = 2;
int Action_Rotate_Right = 3;
int Action_Stop = 4;
int Action_Big_Rotate_Left = 5;
int Action_Big_Rotate_Right = 6;

// NumActions is the number of possible different actions that can be taken.
int NumActions = 7;

// Action is the action choice.
int Action;

// MoveActionDistance is the distance the robot moves when given a movement action
(forward or reverse).

// NOTE: Trying it at 9 instead of 5.
// 10 was too much and the robot could potentially go through the wall if it was
perpendicular to and touching a wall when given a move action.
// float MoveActionDistance = 9F;
// For now, set it to half the radius. Change this later.
// Setting back to 5
// float MoveActionDistance = 5F;
// SMALL
float MoveActionDistance = 8F;

// RotateActionArc is the angle which the robot turns during a rotation action (rotate left or
rotate right).
// For now, set it to 15 degrees. Change this later based on the minimum rotation arc for the
robot.
// float RotateActionArc = 30 * (float)Math.PI / 180;
// SMALL
// float RotateActionArc = 90 * (float)Math.PI / 180;
float RotateActionArc = 15 * (float)Math.PI / 180;

// BigRotateActionArc is 60 degrees: enough for the goal point to go from one edge on a
side camera to the nearest edge on the front camera.

```

```

float BigRotateActionArc = 60 * (float)Math.PI / 180;

// NOTE: NextAction is not used or needed. Keep it commented for now, delete it later.
// NextAction is the next action choice. It is used for the Q-learning formula.
//   int NextAction;

// NOTE: comment out the following for smaller Q-table size
/*   // QTable is an array of the values in the Q-table.
   // At the beginning of each run, QTable is read from a textfile. At the end, it is written to
   the textfile.
   // Size of QTable is [# of cameras,# of goal states per camera,# of end of world states per
   camera,# of sonars,# of states per sonar,# of compass states]
   // NOTE: For now, say there are 4 different compass readings. THIS MAY CHANGE!
   // NOTE: Re-do this declaration.
   double[, , , ,] QTable = new double[3, 4, 4, 8, 5, 4];*/

// QTableOld is an array of the values in the Q-table.
// At the beginning of each run, QTableOld is read from a textfile. At the end, it is written
to the textfile.
// Format of QTableOld is an array with each reference being the reading from the sensor.
// QTableOld is as follows: [Goal0, Goal1, Goal2, EoW0, EoW1, EoW2, Sonar0, Sonar1-2,
Sonar3, Sonar4, Sonar5-6, Sonar7]
// As an example, QTableOld[2,1,1,0,0,2,0,3,2,0,1,1,3] indicates the Q-value when:
// Goal Camera 0 reads Far
// Goal Camera 1 reads Medium
// Goal Camera 2 reads Medium
// End of World Camera 0 reads Close
// End of World Camera 1 reads Close
// End of World Camera 2 reads Far
// Sonar 0 reads Close
// Sonar 1-2 reads Bump
// Sonar 3 reads Far
// Sonar 4 reads Close
// Sonar 5-6 reads Medium
// Sonar 7 reads Medium
// Action taken is Rotate Left
// The size of QTableOld is [3,3,3,3,3,3,4,4,4,4,4,5] where each camera reading has 3
different values (Close, Medium, Far), each sonar reading has 4 different values (Close, Medium,
Far, Bump), and there are 5 possible actions (Forward, Reverse, Stop, Rotate_Left,
Rotate_Right).
//   double[, , , , , , , , , ,] QTableOld = new double[3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 5];

/*   // NOTE
   // REMOVE THIS whole paragraph; defined inside the only function that uses it
   // possible makeup for Dictionary keyword:
   // Convert string of matrix coordinates into binary.

```

```

// If the order of the table is as follows:
// GoalCam0,GoalCam1,GoalCam2,EoWCam0,EoWCam1,EoWCam2,Sonar0,Sonar1-
2,Sonar3,Sonar4,Sonar5-6,Sonar7,Action
// Then the number of possible readings are:
// 3 3 3 3 3 3 4 4 4 4 4 4 5
// And the number of bits necessary are:
// 2 2 2 2 2 2 2 2 2 2 2 2 3
// Then to make the keyword, multiply each coordinate by a power of 2 and add them
together to get an integer. This integer in binary would list the coordinates in the Q matrix.
// So the keyword would be: GoalCam0*2^25 + GoalCam1 *2^23 + GoalCam2*2^21 +
EoWCam0*2^19 + EoWCam1 *2^17 + EoWCam2*2^15 + Sonar0*2^13 + Sonar1-2*2^11 +
Sonar3*2^9 + Sonar4*2^7 + Sonar5-6*2^5 + Sonar7*2^3 + Action
// The maximum value of this is 89,489,404, well within the maximum limit of an int.
// Set the initial value to -1, a value that will never be used as a keyword.
int QKeyword = -1;
*/

// The declaration should look like this:
// Dictionary<int,double> QTable = new Dictionary<int,double>();
// Search under help 'Dictionary' for examples.
// Whenever it is used, if the key doesn't exist, have the exception handler make a new entry
with that key if writing it, or use some pre-defined 'unknown' constant value if only looking at it.
Dictionary<Int64, double> QTable = new Dictionary<Int64, double>();

// Q2
// Dictionary<int, double> QTable2 = new Dictionary<int, double>();

// GenericQValue is used whenever an attempt to look up in a QTable entry that isn't there.
// Change the value as appropriate.
double GenericQValue = 0;
// SMALL
//double GenericQValue = -99999;

// NOTE: keep this commented out, remove it later. I don't think I'm using it.
// NextQMax is used for part of the Q-learning formula. (max a' Q(s',a'))
// It stores the values of the different actions for a particular state so they can be sorted.
// NOTE: The size of this is the number of possible action choices. If the number of action
choices changes, the size must change.
// NOTE: think about a better variable name for this
// double[] NextQMax = new double[5];

```

```

// NOTE: remove the following comments for the smaller Q-table size
// The following are the values for sensor distances.
// Sonar_Bump and Camera_Far are defined in the init function since their value will be the
same as BumpRadius and MaxCameraView, respectively. This prevents having to change two
values if they change.
// These are the maximum values for their range. So the 'Medium' range for cameras are all
values between Camera_Close and Camera_Medium.

// The following are the values for sensor distances.
// Sonar_Bump is defined in the init function since its value will be the same as
BumpRadius. This prevents having to change the value in two places if it changes.
// Dist_Close will be any distance between Sonar_Bump and Sonar_Close.
// Dist_Medium will be any distance between Sonar_Close and Sonar_Far.
// Dist_Far will be any distance beyond Sonar_Far.
// The width of the hallway is 88 inches, so 'close' has to be less than half of 88-2*radius
(34) inches.
// Bump distance for the sonars is anything within BumpRadius (6 inches).
// Close distance for the sonars is anything within 2 feet (24 inches).
// Far distance for the sonars is anything beyond 5 feet (60 inches).
// Medium distance for the sonars is anything between 2 feet and 5 feet.
// NOTE: These are just random for now. Decide actual numbers for these later.
float Sonar_Bump;
float Sonar_Close = 24;
// NOTE: commented out for smaller Q-table size
// float Sonar_Medium = 20;
float Sonar_Far = 60;

// Dist_Close will be any distance less than Camera_Close.
// Dist_Medium will be any distance between Camera_Close and Camera_Far.
// Dist_Far will be any distance beyond Camera_Far.
// Close distance for the cameras is anything within 3 feet (36 inches).
// Far distance for the cameras is anything beyond 8 feet (96 inches).
// Medium distance for the cameras is any distance between 3 feet and 8 feet.
float Camera_Close_Inches = 36;
// NOTE: commented out for smaller Q-table size
// float Camera_Medium = 20;
float Camera_Far_Inches = 96;

// These will be calculated in the init function using the conversion from inches to camera
readings.
float Camera_Close;
float Camera_Far;

// The following are the values for compass headings.

```

```

// These were calculated by running the eight directions through the CalculateCompass
function.
// NOTE: Comment out the following for smaller Q-table size:
/*   int Compass_North = 139;
   int Compass_East = 75;
   int Compass_South = 11;
   int Compass_West = 203;
   int Compass_Northeast = 107;
   int Compass_Southeast = 43;
   int Compass_Southwest = 235;
   int Compass_Northwest = 171;*/

// AtGoal is true if the robot receives the Goal reward.
// This is used for the automatic restart of the simulaton once the goal is reached.
bool AtGoal = false;

// AtEndOfWorld is true if the robot has passed the end of the world.
// This is used for the automatic restart of the simulation once the robot passes the end of the
world.
bool AtEndOfWorld = false;

// alpha and gamma are used for the Q-learning formula.
// alpha is the step size parameter.
// gamma is the discounting factor.
// As gamma gets closer to 0, the agent is more concerned about immediate rewards.
// As gamma approaches 1, it seeks to maximize overall reward and ecomes more
farsighted in its learning.
// For now, set alpha to 0.4 and gamma to 0.7. These could change.
float alpha = 0.4F;
float gamma = 0.7F;

// epsilon is used for the epsilon-greedy choice method.
// epsilon of the time, a random action will be taken.
// 1 - epsilon of the time, the current best (greedy) action will be taken.
float epsilon = 0.05F;
//float epsilon = 0.05F;
// RAR
//float epsilon = 0.1F;

// commented - made a global variable
// AtEndOfWorld is a boolean variable that is true if the robot is in an area designated as the
End of the World.
//   bool AtEndOfWorld = false;

// BumpDetected is true if any of the sensors indicate a bump.

```

```

bool BumpDetected = false;

// Show_Sensors_Toggle is used for one of the buttons.
bool Show_Sensors_Toggle = false;

// Show_Stats_Toggle is used for one of the buttons.
bool Show_Stats_Toggle = false;

// Show_Q_Toggle is used for one of the buttons.
bool Show_Q_Toggle = false;

// QFile holds the name of the textfile containing the Q-table.
const string QFile = "QFile.data";

// BackupQFile holds the name of the textfile containing the backup of the Q-table.
const string BackupQFile = "BackupQFile.data";

// BigBackupQFile is used to dynamically create a name for the bigger backups.
string BigBackupQFile = "";

// Q2
// All the various statistical data will remain stored in QFile. QFile2 only has QTable2.
/*  const string QFile2 = "QFile2.data";
    const string BackupQFile2 = "BackupQFile2.data";
    string BigBackupQFile2 = "";
*/

// NumRepetitions is a number read from the QFile.
// It holds the total number of times that the robot has run through from the start point to
either the Goal or the End of the World.
// Start it at 0 by default. It will be overwritten with whatever number is in the file.
int NumRepetitions = 0;

// BackupFreq is how often the QFile will be backed up to BackupQFile
// For example: if BackupFreq is 100 then the file will be backed up every 100 iterations.
//  int BackupFreq = 1000;
int BackupFreq = 1;

// BigBackupFreq is how often the QFile will be backed up to a unique file.
//  int BigBackupFreq = 50000;
int BigBackupFreq = 10000;

// CurrentReward keeps a running total of the Rewards for the current episode.
int CurrentReward = 0;

// CurrentSteps keeps a running total of the number of steps in the current episode.

```

```

int CurrentSteps = 0;

// NumberGoals is the number of episodes ending in a goal state.
int NumberGoals = 0;

// CurrentWallHits keeps a running total of the number of bumps in the current episode.
int CurrentWallHits = 0;

// AvgWallHits is the average number of bumps per episode.
double AvgWallHits = 0;

// AvgReward is the average of all the total rewards for each episode.
double AvgReward = 0;

// AvgSteps is the average number of steps per episode.
double AvgSteps = 0;

// MaxStepsBeforeRescue is the point where the auto-rescue function starts looking at the
robot.
int MaxStepsBeforeRescue = 100000;

// AutoRescueToggle is the toggle for the Auto Rescue button.
bool Auto_Rescue_Toggle = false;

// Rescue_Max_Backup is the maximum number of times the robot will back up in the
rescue function.
int Rescue_Max_Backup = 3;

// Rescue_Chance is the percent of time that the Auto Rescue will toggle if all the
conditions are met.
double Rescue_Chance = .1;

// Has_Been_Rescued is true if the robot had to be rescued in the current episode.
bool Has_Been_Rescued = false;

// Show_HUD_Toggle is the toggle for the Show HUD button.
bool Show_HUD_Toggle = false;

// Clockless_Timer_Toggle is the toggle for the clockless timer.
bool Clockless_Timer_Toggle = false;

// Clockless_Loop_Count is the number of episodes the program will execute each timer
tick if the clockless timer is activated.
int Clockless_Loop_Count = 500;

// remove later, used for movement test

```

```

//bool movingRight = true;
//bool movingDown = true;

// CenterOfHall is true if the left and right sonars (Sonar_3 and Sonar_7) are equal and are
not bumps.
bool CenterOfHall = false;

// Table_Values are used to hold the Q-table values for display.
double[] Table_Values = new double[] { -1, -1, -1, -1, -1, -1, -1, -1 };
// The following are constants used as labels for Table_Values
int TableTestForward = 0;
int TableTestReverse = 1;
int TableTestLeft = 2;
int TableTestRight = 3;
int TableTestStop = 4;
int TableTestBigLeft = 5;
int TableTestBigRight = 6;
int TableTestRandom = 7;
double TableRandomForward = 0;
double TableRandomReverse = 1;
double TableRandomLeft = 2;
double TableRandomRight = 3;
double TableRandomStop = 4;
double TableRandomBigLeft = 5;
double TableRandomBigRight = 6;
double TableRandomNone = 7;
int TableChoice = -1;

// Q2
// CanSeeGoal is true if the goal is seen in any of the cameras in Close or Medium distance.
// It is also true if the goal is seen in a camera Far distance, but less than half of the 'can't
see' value ReallyLongDistance, a certain percentage of the time (MyopicPercent).
bool CanSeeGoal = false;
float MyopicPercent = 0F;

// These are used for manual timer progression.
bool Manual_Mode_Toggle = false;
bool Manual_Timer_Step = false;

// These are used for manual control mode.
bool Manual_Forward_Toggle = false;
bool Manual_Left_Toggle = false;
bool Manual_Right_Toggle = false;
bool Manual_Stop_Toggle = false;

```

```

bool Manual_Reverse_Toggle = false;
bool Manual_Big_Right_Toggle = false;
bool Manual_Big_Left_Toggle = false;
int Manual_Choice = 7;
int Manual_None = 7;
int Manual_Forward = 0;
int Manual_Reverse = 1;
int Manual_Left = 2;
int Manual_Right = 3;
int Manual_Stop = 4;
int Manual_Big_Left = 5;
int Manual_Big_Right = 6;

// These are to allow the cameras to differentiate left, right, or center views.
// Look for comment CameraFocus for parts that use this.
// CameraFocus are the direction values. FocusCam are the camera numbers.
// As with other parts involving cameras, this will have to be manually changed if more
cameras are added.
int[] FocusInCamera = new int[3] { 3, 3, 3 };
int TempFocusInCamera = 3;
int CameraFocusNone = 3;
int CameraFocusLeft = 0;
int CameraFocusRight = 1;
int CameraFocusCenter = 2;
int FocusCamFront = 0;
int FocusCamRight = 1;
int FocusCamLeft = 2;
float OneThirdCameraArc;

// Q3
// Similar to CanSeeGoal above, this is true if the goal is seen in the Center Focus of the
Front camera.
bool GoalFrontCenter = false;

// Q2.5
// Similar to CanSeeGoal and GoalFrontCenter, GoalInFront is true if the goal is seen in the
Front camera but not in the center focus.
bool GoalInFront = false;

Int64 GlobalKeyword;
Int64 TempKeyword;

int GlobalKeywordA;
int GlobalKeywordB;
int TempKeywordA;
int TempKeywordB;

```

```

/*****
*****/

public ArenaSim()
{
    InitializeComponent();
}

/*****
*****/

private void Arena_Init(object sender, EventArgs e)
{
    float temp_x;
    float temp_y;
    Random randomseeder = new Random();
    float spinrandom;

    // For now, comment out the control panel, since it does not work yet.
//    ArenaSim_Control_Panel_Load(sender, e);

    // CameraFocus
    OneThirdCameraArc = CameraArc / 3;

    NumEndpoints = (SegmentEndpoint.Length) / 2;
    NumSonars = SonarSensors.Length;
    NumCameras = CameraSensors.Length;
    NumEndOfWorld = (EndOfWorldPoints.Length) / 2;

    // NOTE: comment out the following for smaller Q-table size
//    Camera_Far = (float)MaxCameraView;

    Sonar_Bump = BumpRadius;

    // Convert Camera_Close and Camera_Far from inches into camera reading values.
    Camera_Close = CameraInchesToReading(Camera_Close_Inches);
    Camera_Far = CameraInchesToReading(Camera_Far_Inches);

    for (int i = 0; i < NumEndpoints; i++)
    { // This look rotates all the points in the arena map by 5 degrees
        temp_x = SegmentEndpoint[i, x];
        temp_y = SegmentEndpoint[i, y];
    }
}

```

```

SegmentEndpoint[i, x] = temp_x * cosfive - temp_y * sinfive;
SegmentEndpoint[i, y] = temp_x * sinfive + temp_y * cosfive;
}

```

```

// Rotate the starting location 5 degrees also.
temp_x = StartLocation[x];
temp_y = StartLocation[y];
StartLocation[x] = temp_x * cosfive - temp_y * sinfive;
StartLocation[y] = temp_x * sinfive + temp_y * cosfive;

```

```

// Do the same with RobotLocation.
temp_x = RobotLocation[x];
temp_y = RobotLocation[y];
RobotLocation[x] = temp_x * cosfive - temp_y * sinfive;
RobotLocation[y] = temp_x * sinfive + temp_y * cosfive;

```

```

// Do the same with OldLocation.
temp_x = OldLocation[x];
temp_y = OldLocation[y];
OldLocation[x] = temp_x * cosfive - temp_y * sinfive;
OldLocation[y] = temp_x * sinfive + temp_y * cosfive;

```

```

// Rotate the goal point for the camera
temp_x = GoalPoint[x];
temp_y = GoalPoint[y];
GoalPoint[x] = temp_x * cosfive - temp_y * sinfive;
GoalPoint[y] = temp_x * sinfive + temp_y * cosfive;

```

```

// Rotate the end of world points
for (int i = 0; i < NumEndOfWorld; i++)
{
    temp_x = EndOfWorldPoints[i, x];
    temp_y = EndOfWorldPoints[i, y];
    EndOfWorldPoints[i, x] = temp_x * cosfive - temp_y * sinfive;
    EndOfWorldPoints[i, y] = temp_x * sinfive + temp_y * cosfive;
}

```

```

// Rotate the start and goal marker dots
temp_x = StartDot[x];
temp_y = StartDot[y];
StartDot[x] = temp_x * cosfive - temp_y * sinfive;
StartDot[y] = temp_x * sinfive + temp_y * cosfive;

```

```

temp_x = GoalDot[x];
temp_y = GoalDot[y];
GoalDot[x] = temp_x * cosfive - temp_y * sinfive;
GoalDot[y] = temp_x * sinfive + temp_y * cosfive;

// Rotate the orientation 5 degrees
RobotOrientation = RobotOrientation + fivedegrees;

// Sets the timer to its slowest setting.
// Further timer changes are handled with the Sim_Scroll trackbar during runtime.
RobotTimer.Interval = 250;

// Create the QTable file if it doesn't exist.
CreateQFile();

// Create the backup Q-table file if it doesn't exist.
CreateBackupQFile();

// Q2
// Create the files for the QTable2 and its backup.
/*   CreateQFile2();
   CreateBackupQFile2();
*/

// Read the file to the QTable.
ReadQTable();

// Increase the number of repetitions by 1.
NumRepetitions++;

// Reset the Has_Been_Rescued toggle;
Has_Been_Rescued = false;

// Set the current reward and steps and wall hits to 0;
CurrentReward = 0;
CurrentSteps = 0;
CurrentWallHits = 0;

// SMALL
//Now spin the robot to a random starting orientation to ensure Exploring Starts.
//   spinrandom = (float)((360 * randomseeder.NextDouble()) * Math.PI / 180);
//   RobotOrientation = spinrandom;
//   ENDSMALL */

```

```

// comment these out to restore GUI
/* RobotTimer.Interval = 1;
Auto_Rescue_Toggle = true;
Clockless_Timer_Toggle = true;
RobotTimer.Enabled = true;
RobotTimer.Interval = 1;
*/

}

/*****
*****/

private void Rescue_Robot()
{ // This function will 'rescue' the robot by forcing it to move backwards then spin
randomly.

    Random randomseeder = new Random();
    int backuprandom;
    float spinrandom;
    double myopicrandom;

    // NOTE: Add more here if more sensors are added.
    int[] SonarReadings = new int[NumSonars];
    int[] CameraGoalReadings = new int[NumCameras];
    int[] CameraEndOfWorldReadings = new int[NumCameras];

    // backuprandom becomes a random integer between 0 and Rescue_Max_Backup
    backuprandom = (int)(Rescue_Max_Backup * randomseeder.NextDouble());
    for (int i = 0; i < backuprandom; i++)
    { // Go through this loop backuprandom times.
        // Move the robot straight backwards each time.
        PerformAction(Action_Reverse);
        // RobotContacting makes sure that this doesn't shoot the robot outside of the arena.
        RobotContacting();
    }

    // SMALL
    // Now spin the robot in a random direction.
    spinrandom = (float)((360 * randomseeder.NextDouble()) * Math.PI / 180);

    RobotOrientation = spinrandom;
// ENDSMALL */

```

```

// Read new sensor values.
// These are the same as in RobotTimer_Tick
// Cameras to goal and end of world

// DoubleQ
// Set CanSeeGoal to false. If it can be seen, it will be set true inside of the
CalculateCameraToGoal functions.
CanSeeGoal = false;
// Q3
// Also set GoalFrontCenter to false for the same reason.
GoalFrontCenter = false;
// Q2.5
// Also set GoalInFront to false.
GoalInFront = false;

for (int i = 0; i < NumCameras; i++)
{
    CameraGoalReadings[i] = CalculateCameraToGoal(i);
    NextState[State_Goal0 + i] = SensorEncodeCamera(CameraGoalReadings[i]);

    // CameraFocus
    // This is just a test so far. It should only set it for the front camera when the clockless
timer is not active
    // If it's the front sensor and the distance is Close or Medium (i.e., not Dist_Far), set
FocusInCamera to TempFocusInCamera.
    // Since FocusInCamera now works for all three cameras, no more need to check for
i==0.
    // if (i == 0)
    // {
    /* if (NextState[State_Goal0] != Dist_Far)
    {
        FocusInCamera[i] = TempFocusInCamera;
    }
    else
    {
        FocusInCamera[i] = CameraFocusNone;
    }
    // QFocus */
    NextState[State_Focus0 + i] = FocusInCamera[i];
// }
// end CameraFocus

if (NextState[State_Goal0 + i] == Dist_Far)

```

```

    {
        NextState[State_Focus0 + i] = CameraFocusNone;
        //FocusInCamera[i] = CameraFocusNone;
    }
    // end CameraFocus

    // DoubleQ
    if ((NextState[State_Goal0 + i] == Dist_Close) || (NextState[State_Goal0 + i] ==
Dist_Medium))
    { // If the camera reading just found (NextState[State_Goal0+i]) is Dist_Close or
Dist_Medium, set CanSeeGoal to true.
        CanSeeGoal = true;
    }
    else if ((NextState[State_Goal0 + i] == Dist_Far) && (CameraGoalReadings[i] <
(ReallyLongDistance / 2)))
    { // If the camera reading is Dist_Far but the distance to the goal is less than
ReallyLongDistance/2, generate a random number.
        myopicrandom = randomseeder.NextDouble();
        if (myopicrandom < MyopicPercent)
        { // If that random number is within MyopicPercent, set CanSeeGoal to true.
            CanSeeGoal = true;
        }
    }

    // Q3
    if (CanSeeGoal && (FocusInCamera[FocusCamFront] == CameraFocusCenter))
    { // If the goal can be seen and is in the Center focus of the front camera, set
GoalFrontCenter true.
        GoalFrontCenter = true;
    }
    // end Q3

    // Q2.5
    if (CanSeeGoal && ((FocusInCamera[FocusCamFront] == CameraFocusLeft) ||
(FocusInCamera[FocusCamFront] == CameraFocusRight)))
    { // If the goal can be seen and is in the front camera but not in the center, set
GoalInFront true.
        GoalInFront = true;
    }
    // end Q2.5

    CameraEndOfWorldReadings[i] = CalculateCameraToEndOfWorld(i);
    NextState[State_EndOfWorld0 + i] =
SensorEncodeCamera(CameraEndOfWorldReadings[i]);

```

```

    }
    // Sonars
    for (int i = 0; i < NumSonars; i++)
    {
        SonarReadings[i] = CalculateSonar(i);
    }
    NextState[State_Sonar0] = SensorEncodeSonar(SonarReadings[0]);
    NextState[State_Sonar1_2] = SensorEncodeSonar((SonarReadings[1] +
SonarReadings[2]) / 2);
    NextState[State_Sonar3] = SensorEncodeSonar(SonarReadings[3]);
    NextState[State_Sonar4] = SensorEncodeSonar(SonarReadings[4]);
    NextState[State_Sonar5_6] = SensorEncodeSonar((SonarReadings[5] +
SonarReadings[6]) / 2);
    NextState[State_Sonar7] = SensorEncodeSonar(SonarReadings[7]);
    // Set CenterOfHall
    if ((NextState[State_Sonar3] != Sonar_Bump) && (NextState[State_Sonar3] ==
NextState[State_Sonar7]))
    { // If the left and right sonar readings (3 and 7) are the same and aren't reading bumps,
then set CenterOfHall to true.
        CenterOfHall = true;
    }
    else
    { // otherwise set it false
        CenterOfHall = false;
    }

    // Calculated BumpDetected.
    // BumpDetected is true if any of the sonars detect a bump.
    if ((NextState[State_Sonar0] == Dist_Bump) || (NextState[State_Sonar1_2] ==
Dist_Bump) || (NextState[State_Sonar3] == Dist_Bump) || (NextState[State_Sonar4] ==
Dist_Bump) || (NextState[State_Sonar5_6] == Dist_Bump) || (NextState[State_Sonar7] ==
Dist_Bump))
    { // If any of the sonar sensors detects a bump, then BumpDetected is true.
        BumpDetected = true;
    }
    else
    { // Otherwise, BumpDetected is false.
        BumpDetected = false;
    }

    Has_Been_Rescued = true;

    return;
}

```

```
/**
*****/
```

```
private void CreateBackupQFile()
{ // Create an empty BackupQFile if it doesn't already exist.

    if (File.Exists(BackupQFile))
    { // If the file already exists, do nothing.
        return;
    }

    // Otherwise, create it.
    FileStream fs = new FileStream(BackupQFile, FileMode.CreateNew);
    fs.Close();

    return;

}
```

```
/**
*****/
```

```
private void CreateQFile()
{ // Create an empty QFile if it doesn't exist already.

    if (File.Exists(QFile))
    { // If the file already exists, do nothing.
        return;
    }

    // Otherwise, create it.
    FileStream fs = new FileStream(QFile, FileMode.CreateNew);

    fs.Close();

    return;

}
```

```
/**
*****/
```

```
    // Q2
    /* private void CreateQFile2()
    { // Create an empty QFile2 if it doesn't exist already.
```

```

// Aside from filename, this is identical to CreateQFile.
if (File.Exists(QFile2))
{
    return;
}
FileStream fs = new FileStream(QFile2, FileMode.CreateNew);
fs.Close();
return;
}
*/

/*****/
// Q2
/* private void CreateBackupQFile2()
{ // Create an empty BackupQFile2 if it doesn't exist already.
// Aside from filename, this is identical to CreateBackupQFile.
if (File.Exists(BackupQFile2))
{ // If the file already exists, do nothing.
return;
}
// Otherwise, create it.
FileStream fs = new FileStream(BackupQFile2, FileMode.CreateNew);
fs.Close();
return;
}
*/

/*****/

private void WriteQTable()
{ // WriteQTable writes the QTable back into QFile

    FileStream fs = new FileStream(QFile, FileMode.Open, FileAccess.Write);
    BinaryWriter BinWriter = new BinaryWriter(fs);

    // First write the number of repetitions back to the file.
    BinWriter.Write((Int32)NumRepetitions);
    BinWriter.Write((Int32)NumberGoals);
    BinWriter.Write(AvgReward);
    BinWriter.Write(AvgSteps);
    BinWriter.Write(AvgWallHits);

    // Then go through all the QTable entries and write each one to the file.

```

```

foreach (KeyValuePair<Int64, double> kvp in QTable)
{ // This goes through all entries in QTable.
  BinWriter.Write((Int64)kvp.Key);
  BinWriter.Write(kvp.Value);
}

BinWriter.Close();
fs.Close();

// Q2
// Same for QFile2, only no statistics.
/*  FileStream fs2 = new FileStream(QFile2, FileMode.Open, FileAccess.Write);
    BinaryWriter BinWriter2 = new BinaryWriter(fs2);
    foreach (KeyValuePair<Int64, double> kvp2 in QTable2)
    {
      BinWriter2.Write((Int32)kvp2.Key);
      BinWriter2.Write(kvp2.Value);
    }
    BinWriter2.Close();
    fs2.Close();
// end Q2
*/

return;

}

/*****
*****/

private void WriteBackupQTable()
{ // This writes the QTable to BackupQFile

  FileStream fs = new FileStream(BackupQFile, FileMode.Open, FileAccess.Write);
  BinaryWriter BinWriter = new BinaryWriter(fs);

  // First write the number of repetitions back to the file.
  BinWriter.Write((Int32)NumRepetitions);
  BinWriter.Write((Int32)NumberGoals);
  BinWriter.Write(AvgReward);
  BinWriter.Write(AvgSteps);
  BinWriter.Write(AvgWallHits);
}

```

```

// Then go through all the QTable entries and write each one to the file.
foreach (KeyValuePair<Int64, double> kvp in QTable)
{ // This goes through all entries in QTable.
  BinWriter.Write((Int64)kvp.Key);
  BinWriter.Write(kvp.Value);
}

BinWriter.Close();
fs.Close();

// Q2
// Same for BackupQFile2, without the stats.
/*
  FileStream fsQ2 = new FileStream(BackupQFile2, FileMode.Open, FileAccess.Write);
  BinaryWriter BinWriterQ2 = new BinaryWriter(fsQ2);
  foreach (KeyValuePair<Int64, double> kvp2 in QTable2)
  {
    BinWriterQ2.Write((Int32)kvp2.Key);
    BinWriterQ2.Write(kvp2.Value);
  }
  BinWriterQ2.Close();
  fsQ2.Close();
// end Q2
*/

// Now check to see if another backup is needed. Every BigBackupFreq episodes, write
to a unique file.
if ((NumRepetitions % BigBackupFreq) == 0)
{ // Every BigBackupFreq episodes, do this.
  BigBackupQFile = NumRepetitions.ToString() + BackupQFile;
  if (!File.Exists(BigBackupQFile))
  { // If the file doesn't exist, create it.
    // It should never exist but the if statement is here just in case.
    FileStream fsmake = new FileStream(BigBackupQFile, FileMode.CreateNew);
    fsmake.Close();
  }
  FileStream fs2 = new FileStream(BigBackupQFile, FileMode.Open,
FileAccess.Write);
  BinaryWriter BinWriter2 = new BinaryWriter(fs2);
  BinWriter2.Write((Int32)NumRepetitions);
  BinWriter2.Write((Int32)NumberGoals);
  BinWriter2.Write(AvgReward);
  BinWriter2.Write(AvgSteps);
  BinWriter2.Write(AvgWallHits);
  foreach (KeyValuePair<Int64, double> kvp in QTable)
  { // This goes through all entries in QTable.

```

```

        BinWriter2.Write((Int64)kvp.Key);
        BinWriter2.Write(kvp.Value);
    }
    BinWriter2.Close();
    fs2.Close();

    // Q2
    // Same for BackupQFile2, without stats.
    /*    BigBackupQFile2 = NumRepetitions.ToString() + BackupQFile2;
    if (!File.Exists(BigBackupQFile2))
    {
        FileStream fsmake2 = new FileStream(BigBackupQFile2, FileMode.CreateNew);
        fsmake2.Close();
    }
    FileStream fsBQ2 = new FileStream(BigBackupQFile2, FileMode.Open,
FileAccess.Write);
    BinaryWriter BinWriterBQ2 = new BinaryWriter(fsBQ2);
    foreach (KeyValuePair<Int64, double> kvp2 in QTable2)
    {
        BinWriterBQ2.Write((Int32)kvp2.Key);
        BinWriterBQ2.Write(kvp2.Value);
    }
    BinWriterBQ2.Close();
    fsBQ2.Close();
    // end Q2
    */

}

return;

}

/*****
*****/

private void ReadQTable()
{ // ReadQTable reads values from the QFile into the QTable.

    // FirstLine is used to identify if the file is on the first line.
    bool ReadingStats = true;
    // IsKeyword is used to identify if the file is reading the keyword (true) or table entry
(false).

```

```

bool IsKeyword = true;

// ReadKeyword is initialized to 0 to prevent errors. It should be overwritten before it is
ever used.
Int64 ReadKeyword = 0;
double ReadQValue;
bool EndOfFile = false;

// These two lines set up to read data from the QFile:
FileStream fs = new FileStream(QFile, FileMode.Open, FileAccess.Read);
BinaryReader BinReader = new BinaryReader(fs);

// Now read the data.
while (!EndOfFile)
{ // Until it reaches the end of the file.

    try
    { // If there are lines still left to read.
      if (ReadingStats)
      { // If this is the first line, then assign it to NumRepetitions.
        NumRepetitions = BinReader.ReadInt32();
        NumberGoals = BinReader.ReadInt32();
        AvgReward = BinReader.ReadDouble();
        AvgSteps = BinReader.ReadDouble();
        AvgWallHits = BinReader.ReadDouble();
        ReadingStats = false;
      }
      else
      { // Otherwise each line has the key and value to the lookup Q-table.
        if (IsKeyword)
        { // If it's reading the keyword.
          ReadKeyword = BinReader.ReadInt64();
          IsKeyword = false;
        }
        else
        { // Otherwise it's reading the value.
          ReadQValue = BinReader.ReadDouble();
          // Now try to add the ReadQValue to the QTable under keyword
ReadKeyword.
          // If it already exists, overwrite it.
          // It should never already exist but this is here for catching a possible error.
          try
          {
            QTable.Add(ReadKeyword, ReadQValue);
          }
          catch

```



```

        }
        IsKeyword = true;
    }
}
catch
{
    EndOfFile = true;
}
}
BinReader2.Close();
fs2.Close();
// end Q2
*/

return;
}

```

```

/*****
*****/

```

```

private void CalcAvgReward()
{ // This function factors the current reward in to calculate the average total reward over all
episodes.

```

```

// Formula for finding new average:
// CurrentAvg = OldTotal/(NumRepetitions-1)
// OldTotal = CurrentAvg * (NumRepetitions-1)
// NewAvg = (OldTotal + Current)/NumRepetitions
// Then:
// NewAvg = (CurrentAvg*(NumRepetitions-1) + Current)/NumRepetitions

```

```

AvgReward = (AvgReward * (NumRepetitions - 1) + CurrentReward) / NumRepetitions;

```

```

return;
}

```

```

/*****
*****/

```

```

private void CalcAvgSteps()
{ // This function factors the current step count in to calculate the average total number of
steps over all episodes.

```

```

// Formula for finding new average:
// CurrentAvg = OldTotal/(NumRepetitions-1)
// OldTotal = CurrentAvg * (NumRepetitions-1)
// NewAvg = (OldTotal + Current)/NumRepetitions
// Then:
// NewAvg = (CurrentAvg*(NumRepetitions-1) + Current)/NumRepetitions

AvgSteps = (AvgSteps * (NumRepetitions - 1) + CurrentSteps) / NumRepetitions;

return;
}

/*****
*****/

private void CalcAvgWallHits()
{ // This function factors the current bump count in to calculate the average total number
of bumps over all episodes.

// Formula for finding new average:
// CurrentAvg = OldTotal/(NumRepetitions-1)
// OldTotal = CurrentAvg * (NumRepetitions-1)
// NewAvg = (OldTotal + Current)/NumRepetitions
// Then:
// NewAvg = (CurrentAvg*(NumRepetitions-1) + Current)/NumRepetitions

AvgWallHits = (AvgWallHits * (NumRepetitions - 1) + CurrentWallHits) /
NumRepetitions;

return;

}

/*****
*****/

private void ArenaSim_Control_Panel_Load(object sender, EventArgs e)
{ // This loads the second control panel window

// include parameters for passing here
// help search: 'pass form'

Form f = new ArenaSim_Control_Panel();
f.Show(this);
}

```

```

/*****
*****/

```

```

private float PointToLineDistance(float Cx, float Cy, float Ax, float Ay, float Bx, float By)
{ // This function returns the distance between a point and a line
  // Using basic algebra for the formulas of a line and the distance between two points:
  // For the explanations in comments, A and B are endpoints of the line and C is the point
  // Slope1 = (B.y-A.y)/(B.x-A.x)
  // Slope of perpendicular Slope2 = -1/Slope1
  // Equation of lineAB is Y1 = Slope1*X + b1 where the intercept b1=A.y-Slope1*A.x
  // Repeat to get equation of lineC to be Y2 = Slope2*X + b2 where b2=C.y-Slope2*C.x
  // To find intersection point, set the two lines equal to each other and solve for X:
  // Y1 = Y2
  // Slope1*X + b1 = Slope2*X + b2
  // Gives us I.x = (b2-b1)/(Slope1-Slope2)
  // Then plug I.x into the original line to find the I.y = Slope1*I.x + b1
  // Now we need the distance between point C and point I
  // Distance formula D = SQRT((I.y-C.y)^2 + (I.x-C.x)^2)

```

```

float slope1, slope2, b1, b2, Ix, Iy, distance;

```

```

if (Bx != Ax)
{ // These if functions prevent division by 0 in case there's an infinite slope
  slope1 = (By - Ay) / (Bx - Ax);
}
else
{
  slope1 = VeryHighSlope;
}
if (slope1 != 0)
{
  slope2 = -1 / slope1;
}
else
{
  slope2 = VeryHighSlope;
}
b1 = Ay - slope1 * Ax;
b2 = Cy - slope2 * Cx;
if (slope1 != slope2)
{ // This if statement prevents division by 0
  Ix = (b2 - b1) / (slope1 - slope2);
}
else
{

```

```

    Ix = ReallyLongDistance;
}
Iy = slope1 * Ix + b1;
distance = PointToPointDistance(Ix, Iy, Cx, Cy);

return distance;

}

/*****
*****/

private bool IsBumping()
{ // This returns true if robot is within radius to register a bump, false otherwise
  // This function does NOT say which direction the bump was in!
  // Add directional bump later
  // This function may not be needed once the sonars are working

  bool bump = false;

  for (int i = 0; i < NumEndpoints; i++)
  { // go through all the line segments in the arena
    if (i != NumEndpoints - 1)
    {
      if (PointToLineDistance(RobotLocation[x], RobotLocation[y], SegmentEndpoint[i,
x], SegmentEndpoint[i, y], SegmentEndpoint[i + 1, x], SegmentEndpoint[i + 1, y]) <=
(RobotRadius + BumpRadius))
        { // if the distance between the current location and the wall is less than the bump
radius, signal bump
          bump = true;
        }
      }
    else
    {
      if (PointToLineDistance(RobotLocation[x], RobotLocation[y], SegmentEndpoint[i,
x], SegmentEndpoint[i, y], SegmentEndpoint[0, x], SegmentEndpoint[0, y]) <= (RobotRadius +
BumpRadius))
        { // check the last point connected to the first
          bump = true;
        }
      }
    }
  }
}

```

```

    return bump;

}

/*****
*****/

private bool IsTouchingWall()
{ // This function returns true if the robot is touching a wall
  // This is different than the bump sensor.
  // IsBumping is used as a sensor input for a bump
  // IsTouchingWall is used to prevent the simulator from letting the robot outside the arena
  // The only difference code-wise between IsTouchingWall and IsBumping is that
  touching signals when the robot is RobotRadius from a wall, and bump signals when the robot is
  RobotRadius + BumpRadius from the wall

  bool touching = false;

  for (int i = 0; i < NumEndpoints; i++)
  { // go through all the line segments in the arena
    if (i != NumEndpoints - 1)
    {
      if (PointToLineDistance(RobotLocation[x], RobotLocation[y], SegmentEndpoint[i,
x], SegmentEndpoint[i, y], SegmentEndpoint[i + 1, x], SegmentEndpoint[i + 1, y]) <=
(RobotRadius))
        { // if the distance between the current location and the wall is less than the bump
radius, signal bump
          touching = true;
        }
      }
    else
    {
      if (PointToLineDistance(RobotLocation[x], RobotLocation[y], SegmentEndpoint[i,
x], SegmentEndpoint[i, y], SegmentEndpoint[0, x], SegmentEndpoint[0, y]) <= (RobotRadius))
        { // check the last point connected to the first
          touching = true;
        }
      }
    }
  }

  return touching;
}

```

```
/**
*****/
```

```
private float PointToPointDistance(float X1, float Y1, float X2, float Y2)
{ // This function calculates the distance between two points using the standard distance
formula
// distance = SQRT( (y1-y2)^2 + (x1-x2)^2 )
float distance;

if ((Math.Abs(Y1 - Y2) > ReallyLongDistance) || (Math.Abs(X1 - X2) >
ReallyLongDistance))
{ // If the difference is too far in either the x or y direction, just make the distance
ReallyLongDistance to avoid overflowing float by squaring large numbers
// Any distance farther than ReallyLongDistance doesn't matter anyway.
distance = ReallyLongDistance;
}
else
{
distance = (float)Math.Sqrt(Math.Pow((Y1 - Y2), 2) + Math.Pow((X1 - X2), 2));
}
return distance;
}
```

```
/**
*****/
```

```
private float LineIntersectDistance(float SensorTheta, float Slope1, float b1, float Xa, float
Ya, float Xb, float Yb, float SensorArc)
{ // This function returns the distance from the robot to a line along a line segment (instead
of shortest distance).
// Slope1 and b1 are for the line equation of the sonar line segment.
// Xa, Ya, Xb, Yb are endpoints for a wall segment.
// SensorTheta is the angle from RobotOrientation of the center of the sensor.
// See function PointToLineDistance for comments on the equations.
float Slope2;
float b2;
float IntersectX;
float IntersectY;
float distance;
float PolarTheta;
float PolarX;
float PolarY;
float PolarR;
float ArcCheckMax;
float ArcCheckMin;
```

```

if (Xa != Xb)
{ // These if functions prevent division by 0 in case there's an infinite slope
  Slope2 = (Ya - Yb) / (Xa - Xb);
}
else
{
  Slope2 = VeryHighSlope;
}
b2 = Ya - Slope2 * Xa;
if (Slope1 != Slope2)
{
  IntersectX = (b2 - b1) / (Slope1 - Slope2);
}
else
{ // This prevents division by 0
  IntersectX = ReallyLongDistance;
}
IntersectY = Slope1 * IntersectX + b1;

// Convert the intersect point to polar coordinates in relation to the robot
// PolarR is the r coordinate, polar distance from the robot to the intersect point
PolarX = RobotLocation[x] - IntersectX;
PolarY = RobotLocation[y] - IntersectY;
PolarR = (float)Math.Sqrt(Math.Pow(PolarX, 2) + Math.Pow(PolarY, 2)) - RobotRadius;

// The following set of if statements are to calculate PolarTheta in the range from 0 to
2*Pi
if (PolarX > 0)
{
  PolarTheta = (float)Math.Atan(PolarY / PolarX) + (float)Math.PI;
}
else if ((PolarX < 0) && (PolarY > 0))
{
  PolarTheta = (float)Math.Atan(PolarY / PolarX) + twopi;
}
else if ((PolarX < 0) && (PolarY <= 0))
{
  PolarTheta = (float)Math.Atan(PolarY / PolarX);
}
else if ((PolarX == 0) && (PolarY > 0))
{
  PolarTheta = 3 * (float)Math.PI / 2;
}
else if ((PolarX == 0) && (PolarY < 0))
{

```

```

    PolarTheta = (float)Math.PI / 2;
}
else
{ // This should never happen but is here just in case.
    PolarTheta = RobotOrientation;
}

// Check to make sure that the segment is within the 'viewing' angle of the sensor.
// This makes certain that the sensor doesn't read things behind it.
// If the segment is outside of the arc, then make the distance ReallyLongDistance
// ArcCheckMax and ArcCheckMin are angles of the sonar view.
ArcCheckMax = RobotOrientation + SensorTheta + (SensorArc / 2);
if (ArcCheckMax > twopi)
{ // If the angle comes to more than 2*Pi, subtract 2*Pi
    ArcCheckMax = ArcCheckMax - twopi;
}
if (ArcCheckMax < 0)
{ // If the angle comes to less than 0, add 2*Pi
    ArcCheckMax = ArcCheckMax + twopi;
}
ArcCheckMin = RobotOrientation + SensorTheta - (SensorArc / 2);
if (ArcCheckMin > twopi)
{ // If the angle comes to more than 2*Pi, subtract 2*Pi
    ArcCheckMin = ArcCheckMin - twopi;
}
if (ArcCheckMin < 0)
{ // If the angle comes to less than 0, add 2*Pi
    ArcCheckMin = ArcCheckMin + twopi;
}

if ((PolarTheta <= ArcCheckMax) && (PolarTheta >= ArcCheckMin))
{ // if the segment angle is within the arc of the sensor, then the sensor reading is the
polar distance
    distance = PolarR;
}
else
{ // otherwise, the wall is behind the sensor and can't be seen
    distance = ReallyLongDistance;
}

return distance;
}

```

```
/**
****/
```

```
private float InsertNoise(float signal, float percentnoise)
{ // This function takes a signal and inserts a random amount of noise, within percentnoise
  // most common uses will be 10% for the sonars and 25% for the compass
  Random randomseeder = new Random();
  double randomnumber;
  float tempsignal;

  // Generate a random number between 1 - percentnoise and 1 + percentnoise
  // This number is a decimal (i.e., 1.053 and not 105.3)
  randomnumber = 1 + (percentnoise - (2 * percentnoise * randomseeder.NextDouble())) /
100;

  tempsignal = signal * (float)randomnumber;

  // DoubleQ
  // This cancels the noise.
  // tempsignal = signal;

  return tempsignal;
}
```

```
/**
****/
```

```
private int CalculateSonar(int SensorNum)
{ // This calculates the reading from a sonar sensor
  float MiniArcLength;
  // NumDivisions must be odd
  int NumDivisions = 11;
  // Theta, Slope, and b are used for angle calculations for the sonar
  float Theta;
  float Slope;
  float b;

  /* from an older version of sonar - didn't work
  // SegmentDistance is an array of the lengths of a sonar segment to all the walls
  // This is used for storing all the distances from the robot to each wall along the segment
  being looked at
  float[] SegmentDistance = new float[NumEndpoints];
```

```

// MinSegmentDistance is an array of the first readings for each segment after sorting
// The values here are the 'actual seen' reading from each segment
float[] MinSegmentDistance = new float[NumDivisions + 1];
// MinDistance is the minimum distance from the array MinSegmentDistance
float MinDistance;
// SonarReading is the final reading from the sensor
int SonarReading;
*/

// Added for sonar fix
// SegmentDistance is the length of a sonar segment to a wall.
// This is used for storing the distance from the robot to each wall along the segment
being looked at.
float SegmentDistance = ReallyLongDistance;
// MinSegmentDistance is the shortest reading for each segment.
// It is the shortest distance of all the SegmentDistances. In other words, it is the distance
along the segment from the robot to the nearest wall.
float MinSegmentDistance = ReallyLongDistance;
// MinDistance is the minimum distance for the sensor.
// It is the shortest distance of all the MinSegmentDistances.
float MinDistance = ReallyLongDistance;
// SonarReading is the final reading from the sensor.
int SonarReading;

// This divides the arc of the sonar's 'vision' into segments of interval MiniArcLength
radians
MiniArcLength = SonarArc / NumDivisions;

for (int n = 0; n < NumDivisions; n++)
{ // This loop goes through the divisions of the sonar arc
// Slope and b are respectively the slope and intercept of the line segment from the
robot along the sonar direction
Theta = RobotOrientation + SonarSensors[SensorNum] - (SonarArc / 2) + (n *
MiniArcLength);
if (Theta > twopi)
{
Theta = Theta - twopi;
}
if (Theta != Math.PI / 2)
{ // This if function prevents division by 0 in case there's an infinite slope
Slope = (float)Math.Tan(Theta);
}
else
{
Slope = VeryHighSlope;
}
}

```

```

    }
    b = RobotLocation[y] - Slope * RobotLocation[x];

    for (int i = 0; i < NumEndpoints; i++)
    { // This loop goes through all the arena walls to see where the sonar intersects with
the wall
        if (i != NumEndpoints - 1)
        {
            // from an older version of sonar - didn't work
            // SegmentDistance[i] = LineIntersectDistance(SonarSensors[SensorNum],
Slope, b, SegmentEndpoint[i, x], SegmentEndpoint[i, y], SegmentEndpoint[i + 1, x],
SegmentEndpoint[i + 1, y], SonarArc);

            SegmentDistance = LineIntersectDistance(SonarSensors[SensorNum], Slope, b,
SegmentEndpoint[i, x], SegmentEndpoint[i, y], SegmentEndpoint[i + 1, x], SegmentEndpoint[i +
1, y], SonarArc);
        }
        else
        {
            // from an older version of sonar - didn't work
            // SegmentDistance[i] = LineIntersectDistance(SonarSensors[SensorNum],
Slope, b, SegmentEndpoint[i, x], SegmentEndpoint[i, y], SegmentEndpoint[0, x],
SegmentEndpoint[0, y], SonarArc);

            SegmentDistance = LineIntersectDistance(SonarSensors[SensorNum], Slope, b,
SegmentEndpoint[i, x], SegmentEndpoint[i, y], SegmentEndpoint[0, x], SegmentEndpoint[0, y],
SonarArc);
        }

        if (SegmentDistance < MinSegmentDistance)
        { // If the new segment distance is less than the current shortest one, then make the
new distance the shortest.
            MinSegmentDistance = SegmentDistance;
        }
    }

    // from an older version of sonar - didn't work
    // Now find the shortest of the SegmentDistances
    // This is the reading from that segment of the sonar
    // comment out the next two lines
    // Array.Sort(SegmentDistance);
    // MinSegmentDistance[n] = SegmentDistance[0];

```

```

        if (MinSegmentDistance < MinDistance)
        { // If the shortest distance to that wall is shorter than the current shortest, then make
it the new shortest.
            MinDistance = MinSegmentDistance;
        }

    }

// from an older version of sonar - didn't work
/*
    // Now choose shortest distance from the MinSegmentDistances
    // This is the reading from the sonar
    Array.Sort(MinSegmentDistance);
    MinDistance = MinSegmentDistance[0];

    // subtract the radius from the calculated distance for the reading to the sensor, not
the robot's center
    MinDistance = MinDistance - RobotRadius;

    // insert 10% noise into the signal
    SonarReading = (int)InsertNoise(MinDistance, 10);

    // Now make sure the noise didn't make the value negative
    if (SonarReading < 0)
    {
        SonarReading = 0;
    }

    return SonarReading;
*/

MinDistance = MinDistance - RobotRadius;
SonarReading = (int)InsertNoise(MinDistance, 10);
if (SonarReading < 0)
{
    SonarReading = 0;
}
return SonarReading;

}

```

```
/******  
*****/
```

```
private int CalculateCompass()  
{ // this function calculates the compass sensor reading  
  // Math for this function:  
  // When robot is facing direction Pi, the compass reading is ~203  
  // Therefore, Pi is 204/256 (when 0 is taken into account) around the circle from the 'start'  
point of the compass.  
  // Since Pi is the halfway point, we want a reading of 128 there.  
  // First undo the 5 degree rotation and make sure RobotOrientation is inside the 0-2*Pi  
range (since it normally doesn't matter if it goes over).  
  // Note that, even though the rest of the code uses the range -Pi to Pi, 0 to 2*Pi is  
appropriate here since it is just finding the percentage of the way around the circle, not the actual  
heading.  
  // Then take the Orientation divided by 2*Pi to find the percentage around the circle.  
Multiply that percentage by 256.  
  // Then, since the compass reading is rotated, add 75 (203-128) to the reading.  
  // Feed that number into the InsertNoise function with a 25% noise (since the compass  
has been shown to be very inaccurate).  
  // Finally, make the heading an integer and make sure the value is between 0 and 255 by  
modding it with 256.
```

```
int CompassHeading;  
float TempCompassHeading;  
float RealOrientation;  
// CompassPercentage is for the percentage of the way around the circle that the robot is  
facing  
float CompassPercentage;
```

```
// First, undo the 5 degree rotation  
RealOrientation = RobotOrientation - fivedegrees;  
// Then, make sure orientation is in the 0-2*Pi range  
while ((RealOrientation < 0) || (RealOrientation > twopi))  
{  
  if (RealOrientation < 0)  
  {  
    RealOrientation = RealOrientation + twopi;  
  }  
  if (RealOrientation > twopi)  
  {  
    RealOrientation = RealOrientation - twopi;  
  }  
}
```

```

    }
    // Find the percentage of the way around the circle the robot is facing
    CompassPercentage = RealOrientation / twopi;
    // Multiply the percentage by 256 to get a heading
    TempCompassHeading = CompassPercentage * 256;
    // Add 75 to the heading to match the actual compass output
    TempCompassHeading = TempCompassHeading + 75;
    // Call the noise function with 25% noise and make it an integer
    CompassHeading = (int)InsertNoise(TempCompassHeading, 25);
    // Make sure it's within 0-255
    CompassHeading = CompassHeading % 256;

    return CompassHeading;
}

/*****
*****/

private int CalculateCameraToGoal(int CameraNum)
{ // This function takes the number of a camera (0, 1, or 2) and returns the distance to the
Goal point or if it can't be seen.
    int GoalDistance;
    float PolarTheta;
    float PolarX;
    float PolarY;
    float PolarR;
    float ArcCheckMax;
    float ArcCheckMin;
    float TempGoalDistance = ReallyLongDistance; // Set this initially to prevent errors.
    float DistanceCheck;
    float Slope;
    float b;
    float IntersectX;
    float IntersectY;
    float WallXLow;
    float WallXHigh;
    float WallYLow;
    float WallYHigh;
    bool GoalSeen;
    float Xa;
    float Ya;
    float Xb;
    float Yb;
    float Slope2;
    float b2;

```

```

// CameraFocus
float ArcSplitLeftCenter;
float ArcSplitRightCenter;

// Make sure the line between the two points is within the camera's field of vision, using
polar coordinates.
// First make the polar coordinates.
PolarX = RobotLocation[x] - GoalPoint[x];
PolarY = RobotLocation[y] - GoalPoint[y];
PolarR = (float)Math.Sqrt(Math.Pow(PolarX, 2) + Math.Pow(PolarY, 2)) - RobotRadius;

// The following set of if statements are to calculate PolarTheta in the range from 0 to
2*Pi

if (PolarX > 0)
{
    PolarTheta = (float)Math.Atan(PolarY / PolarX) + (float)Math.PI;
}
else if ((PolarX < 0) && (PolarY > 0))
{
    PolarTheta = (float)Math.Atan(PolarY / PolarX) + twopi;
}
else if ((PolarX < 0) && (PolarY <= 0))
{
    PolarTheta = (float)Math.Atan(PolarY / PolarX);
}
else if ((PolarX == 0) && (PolarY > 0))
{
    PolarTheta = 3 * (float)Math.PI / 2;
}
else if ((PolarX == 0) && (PolarY < 0))
{
    PolarTheta = (float)Math.PI / 2;
}
else
{
    // This should never happen but is here just in case.
    PolarTheta = RobotOrientation;
}

// Now that we have the polar coordinates, check to make sure that this line is within the
viewing angle of the camera.
// If the line segment is outside the arc, make the distance ReallyLongDistance and set the
camera's GoalSeen to false.
// If it's within the arc, set the camera's GoalSeen to true.
// ArcCheckMax and ArcCheckMin are angles of the camera view.

```

```

ArcCheckMax = RobotOrientation + CameraSensors[CameraNum] + (CameraArc / 2);
if (ArcCheckMax > twopi)
{ // If the angle comes to more than 2*Pi, subtract 2*Pi
  ArcCheckMax = ArcCheckMax - twopi;
}
if (ArcCheckMax < 0)
{ // If the angle comes to less than 0, add 2*Pi
  ArcCheckMax = ArcCheckMax + twopi;
}

ArcCheckMin = RobotOrientation + CameraSensors[CameraNum] - (CameraArc / 2);
if (ArcCheckMin > twopi)
{ // If the angle comes to more than 2*Pi, subtract 2*Pi
  ArcCheckMin = ArcCheckMin - twopi;
}
if (ArcCheckMin < 0)
{ // If the angle comes to less than 0, add 2*Pi
  ArcCheckMin = ArcCheckMin + twopi;
}

// CameraFocus
// ArcSplitLeftCenter and ArcSplitRightCenter divide the camera's visible arc into three
segments: ArcCheckMin to ArcSplitRightCenter to ArcSplitLeftCenter to ArcCheckMax
// Like ArcCheckMin and ArcCheckMax, also make sure they're inside the range of 0 to
2*Pi.
ArcSplitLeftCenter = ArcCheckMax - OneThirdCameraArc;
ArcSplitRightCenter = ArcCheckMin + OneThirdCameraArc;
if (ArcSplitLeftCenter > twopi)
{
  ArcSplitLeftCenter = ArcSplitLeftCenter - twopi;
}
if (ArcSplitLeftCenter < 0)
{
  ArcSplitLeftCenter = ArcSplitLeftCenter + twopi;
}
if (ArcSplitRightCenter > twopi)
{
  ArcSplitRightCenter = ArcSplitRightCenter - twopi;
}
if (ArcSplitRightCenter < 0)
{
  ArcSplitRightCenter = ArcSplitRightCenter + twopi;
}
// end CameraFocus

if ((PolarTheta <= ArcCheckMax) && (PolarTheta >= ArcCheckMin))

```

```

    { // If the segment angle is within the arc of the camera view, then the sensor reading is
the polar distance.
    TempGoalDistance = PolarR;
    GoalSeen = true;
    }
else
{ // Otherwise, the wall is outside of the camera's line of sight.
// Set the camera's GoalSeen to false.
    GoalSeen = false;
}

// Now check for intersecting walls.
// For Koolio's run in the hall, this function is not necessarily needed, since the arena
doesn't have many corners to block line of sight. However, it is put in for completeness sake and
to allow for arenas that are not just long, straight hallways.
// This is only done if GoalSeen is true, since it won't matter otherwise.
if (GoalSeen)
{
    // First find the slope and intercept for the line from RobotLocation to GoalPoint.
    if (RobotLocation[x] != GoalPoint[x])
    { // If a slope is possible
        Slope = (RobotLocation[y] - GoalPoint[y]) / (RobotLocation[x] - GoalPoint[x]);
    }
    else
    {
        Slope = VeryHighSlope;
    }
    b = RobotLocation[y] - Slope * RobotLocation[x];

    for (int i = 0; i < NumEndpoints; i++)
    { // This loop goes through all the arena walls to see if there is a wall between the
robot and the goal point
        // Use the line from the robot location to the goal point and see if it intersects with
each wall.
        // Then compare the distance to TempGoalDistance.
        // If this DistanceCheck is more, then continue to the next wall.
        // If it is less, the wall is in the way. The goal point can't be seen, so set SeeGoal to
false and break out of the loop by setting i to NumEndpoints.
        if (i != NumEndpoints - 1)
        {
            Xa = SegmentEndpoint[i, x];
            Ya = SegmentEndpoint[i, y];
            Xb = SegmentEndpoint[i + 1, x];
            Yb = SegmentEndpoint[i + 1, y];
            // Find the distance from the robot to the point where the line from robot to goal
intersects the wall.

```

```

DistanceCheck = LineIntersectDistance(CameraSensors[CameraNum], Slope, b,
Xa, Ya, Xb, Yb, CameraArc);
if (DistanceCheck < (TempGoalDistance - 5))
{ // This is true if the line making the wall is closer to the robot than the goal is.
  // If this is false then the wall is not in the way and there is no need to check the
endpoints.
  // The -5 to TempGoalDistance is necessary because there may be some
rounding error. Since the goal point is on the wall, it is possible that a rounding error may place
the goal just slightly beyond the wall. This prevents that.
  // 5 is an arbitrary distance, but it should be long enough. The only times this
will be a problem is if there is a wall less than 5 inches from the goal. Because this will never
happen (it would make the goal impossible to reach anyway), this is a safe assumption.

  // Use the intersection of two lines to find Ix and Iy.
  // First find the slope and intercept of the line that makes the wall.
  Slope2 = (Ya - Yb) / (Xa - Xb);
  b2 = Ya - Slope2 * Xa;

  // Now use the following:
  // Slope*Ix + b = Slope2*Ix + b2
  // Slope*Ix - Slope2*Ix = b2 - b
  // Ix*(Slope-Slope2) = b2 - b
  // Ix = (b2 - b) / (Slope - Slope2)
  IntersectX = (b2 - b) / (Slope - Slope2);
  IntersectY = Slope * IntersectX + b;

  // Now we need the wall endpoints in the right order.
  // WallXLow is the lower of the two X coordinates of the wall segment and
WallXHigh is the higher.
  // Same with WallYLow and WallYHigh.
  // The X and Y DO NOT need to correlate to each other, since they're being
used just as a range comparison and not as points.
  if (SegmentEndpoint[i, x] > SegmentEndpoint[i + 1, x])
  {
    WallXLow = SegmentEndpoint[i + 1, x];
    WallXHigh = SegmentEndpoint[i, x];
  }
  else if (SegmentEndpoint[i, x] < SegmentEndpoint[i + 1, x])
  {
    WallXLow = SegmentEndpoint[i, x];
    WallXHigh = SegmentEndpoint[i + 1, x];
  }
  else
  { // If WallXLow would be the same as WallXHigh, allow a slight leeway for
rounding errors.
    // RobotRadius is an appropriate allowance.

```

```

        WallXLow = SegmentEndpoint[i, x] - RobotRadius;
        WallXHigh = SegmentEndpoint[i, x] + RobotRadius;
    }
    if (SegmentEndpoint[i, y] > SegmentEndpoint[i + 1, y])
    {
        WallYLow = SegmentEndpoint[i + 1, y];
        WallYHigh = SegmentEndpoint[i, y];
    }
    else if (SegmentEndpoint[i, y] < SegmentEndpoint[i + 1, y])
    {
        WallYLow = SegmentEndpoint[i, y];
        WallYHigh = SegmentEndpoint[i + 1, y];
    }
    else
    {
        WallYLow = SegmentEndpoint[i, y] - RobotRadius;
        WallYHigh = SegmentEndpoint[i, y] + RobotRadius;
    }
    // Now compare the point (IntersectX,IntersectY) to the line segment.
    // If IntersectX is between the X coordinates AND IntersectY is between the Y
coordinates then the line segment crosses the wall.
    if ((IntersectX > WallXLow) && (IntersectX < WallXHigh) && (IntersectY >
WallYLow) && (IntersectY < WallYHigh))
    { // At this point, we know the field of vision is blocked by a wall.
        // Set GoalSeen to false and break out of the loop by setting i to
NumEndpoints
        GoalSeen = false;
        i = NumEndpoints;
    }
}
}
else
{ // See comments above for all the calculations and explanations.
    // This is for checking the final wall between the last point and the first one in
SegmentEndpoint.
    Xa = SegmentEndpoint[i, x];
    Ya = SegmentEndpoint[i, y];
    Xb = SegmentEndpoint[0, x];
    Yb = SegmentEndpoint[0, y];
    DistanceCheck = LineIntersectDistance(CameraSensors[CameraNum], Slope, b,
Xa, Ya, Xb, Yb, CameraArc);
    if (DistanceCheck < (TempGoalDistance - 5))
    {
        Slope2 = (Ya - Yb) / (Xa - Xb);
        b2 = Ya - Slope2 * Xa;
        IntersectX = (b2 - b) / (Slope - Slope2);

```



```

// Call the function to convert distance in inches into the camera reading.
TempGoalDistance = CameraInchesToReading(TempGoalDistance);

// Put in 10% noise
GoalDistance = (int)InsertNoise(TempGoalDistance, 10);

// Make sure the goal point is within the maximum distance the camera can detect an
object.
// This only matters if it can see the goal, but a check would be redundant.
// If the goal can't be seen, GoalSeen is already false.
// If it can be seen and is within the max distance, this statement does nothing.
// If it can be seen and is not within the max distance, this sets GoalSeen to false.
if (GoalDistance > MaxCameraView)
{
    GoalSeen = false;
}

if (!GoalSeen)
{ // If the goal can't be seen, make the distance ReallyLongDistance
    GoalDistance = (int)ReallyLongDistance;
}

// At the end, set the proper camera's SeeGoal to GoalSeen.
// NOTE: If more cameras are added, you MUST also add them manually to this series of
if statements.
if (CameraNum == 0)
{
    CameraVision0[SeeGoal] = GoalSeen;
}
else if (CameraNum == 1)
{
    CameraVision1[SeeGoal] = GoalSeen;
}
else if (CameraNum == 2)
{
    CameraVision2[SeeGoal] = GoalSeen;
}

// CameraFocus
// If the goal is seen (GoalSeen is true) then check which of the three sub-arcs PolarTheta
is in.
// Edit: Since this is no longer only for the front camera, it would happen whether the
goal is seen or not.

```

```

    if (GoalSeen)
    {
        if ((PolarTheta >= ArcCheckMin) && (PolarTheta < ArcSplitRightCenter))
        { // If PolarTheta is between ArcCheckMin and ArSplitRightCenter, it is in the Right
region.
            //      TempFocusInCamera = CameraFocusRight;
            FocusInCamera[CameraNum] = CameraFocusRight;
        }
        else if ((PolarTheta >= ArcSplitRightCenter) && (PolarTheta <= ArcSplitLeftCenter))
        { // If PolarTheta is between ArcSplitRightCenter and ArcSplitLeftCenter, it is in the
Center region.
            //      TempFocusInCamera = CameraFocusCenter;
            FocusInCamera[CameraNum] = CameraFocusCenter;
        }
        else if ((PolarTheta > ArcSplitLeftCenter) && (PolarTheta <= ArcCheckMax))
        { // If PolarTheta is between ArcSplitLeftCenter and ArcCheckMax, it is in the Left
region.
            //      TempFocusInCamera = CameraFocusLeft;
            FocusInCamera[CameraNum] = CameraFocusLeft;
        }
        else
        { // Otherwise set it to None. This should never happen, though.
            //      TempFocusInCamera = CameraFocusNone;
            FocusInCamera[CameraNum] = CameraFocusNone;
        }
    }
    else
    { // Otherwise it is not seen, so set it to None.
//      TempFocusInCamera = CameraFocusNone;
        FocusInCamera[CameraNum] = CameraFocusNone;
    }
// end CameraFocus

    return GoalDistance;
}

```

```

/*****
*****/

```

```

private int CalculateCameraToEndOfWorld(int CameraNum)
{ // This function takes the number of a camera (0, 1, or 2) and returns the distance from
the robot to the nearest End of World marker or if it can't be seen.
    // The algorithm is almost identical to the CalculateCameraToGoal function, with enough
differences that I needed a different function.
    // For comments on the algorithm and calculations, see CalculateCameraToGoal.

```

```

int FinalEndOfWorldDistance;
float PolarTheta;
float PolarX;
float PolarY;
float PolarR;
float ArcCheckMax;
float ArcCheckMin;
float TempEndOfWorldDistance = ReallyLongDistance; // Set this initially to prevent
errors.
float DistanceCheck;
float Slope;
float b;
float IntersectX;
float IntersectY;
float WallXLow;
float WallXHigh;
float WallYLow;
float WallYHigh;
float[] EndOfWorldDistance = new float[NumEndOfWorld];
bool[] CanSeeEndOfWorld = new bool[NumEndOfWorld];
float SortDistance = ReallyLongDistance;
bool SortCanSeeEndOfWorld = false;
float Xa;
float Ya;
float Xb;
float Yb;
float Slope2;
float b2;

// The majority of this is a loop that goes through each End of World point.
for (int CurrentEndOfWorld = 0; CurrentEndOfWorld < NumEndOfWorld;
CurrentEndOfWorld++)
{
    PolarX = RobotLocation[x] - EndOfWorldPoints[CurrentEndOfWorld, x];
    PolarY = RobotLocation[y] - EndOfWorldPoints[CurrentEndOfWorld, y];
    PolarR = (float)Math.Sqrt(Math.Pow(PolarX, 2) + Math.Pow(PolarY, 2)) -
RobotRadius;

    if (PolarX > 0)
    {
        PolarTheta = (float)Math.Atan(PolarY / PolarX) + (float)Math.PI;
    }
    else if ((PolarX < 0) && (PolarY > 0))
    {
        PolarTheta = (float)Math.Atan(PolarY / PolarX) + twopi;
    }
}

```

```

}
else if ((PolarX < 0) && (PolarY <= 0))
{
    PolarTheta = (float)Math.Atan(PolarY / PolarX);
}
else if ((PolarX == 0) && (PolarY > 0))
{
    PolarTheta = 3 * (float)Math.PI / 2;
}
else if ((PolarX == 0) && (PolarY < 0))
{
    PolarTheta = (float)Math.PI / 2;
}
else
{
    // This should never happen but is here just in case.
    PolarTheta = RobotOrientation;
}

ArcCheckMax = RobotOrientation + CameraSensors[CameraNum] + (CameraArc / 2);
if (ArcCheckMax > twopi)
{
    // If the angle comes to more than 2*Pi, subtract 2*Pi
    ArcCheckMax = ArcCheckMax - twopi;
}
if (ArcCheckMax < 0)
{
    // If the angle comes to less than 0, add 2*Pi
    ArcCheckMax = ArcCheckMax + twopi;
}
ArcCheckMin = RobotOrientation + CameraSensors[CameraNum] - (CameraArc / 2);
if (ArcCheckMin > twopi)
{
    // If the angle comes to more than 2*Pi, subtract 2*Pi
    ArcCheckMin = ArcCheckMin - twopi;
}
if (ArcCheckMin < 0)
{
    // If the angle comes to less than 0, add 2*Pi
    ArcCheckMin = ArcCheckMin + twopi;
}

if ((PolarTheta <= ArcCheckMax) && (PolarTheta >= ArcCheckMin))
{
    // If the segment angle is within the arc of the camera view, then the sensor reading
    is the polar distance.
    TempEndOfWorldDistance = PolarR;
    CanSeeEndOfWorld[CurrentEndOfWorld] = true;
}
else
{
    // Otherwise, the wall is outside of the camera's line of sight.

```

```

    // Set the camera's SeeEndOfWorld to false.
    CanSeeEndOfWorld[CurrentEndOfWorld] = false;
}

// Now check for intersecting walls.
// This is only done if CanSeeEndOfWorld is true, since it won't matter otherwise.
if (CanSeeEndOfWorld[CurrentEndOfWorld])
{
    // First find the slope and intercept for the line from RobotLocation to the Ed of
World point.
    if (RobotLocation[x] != EndOfWorldPoints[CurrentEndOfWorld, x])
    { // If a slope is possible
        Slope = (RobotLocation[y] - EndOfWorldPoints[CurrentEndOfWorld, y]) /
(RobotLocation[x] - EndOfWorldPoints[CurrentEndOfWorld, x]);
    }
    else
    {
        Slope = VeryHighSlope;
    }
    b = RobotLocation[y] - Slope * RobotLocation[x];

    for (int i = 0; i < NumEndpoints; i++)
    { // This loop goes through all the arena walls to see if there is a wall between the
robot and the End of World point.
        if (i != NumEndpoints - 1)
        {
            Xa = SegmentEndpoint[i, x];
            Ya = SegmentEndpoint[i, y];
            Xb = SegmentEndpoint[i + 1, x];
            Yb = SegmentEndpoint[i + 1, y];

            // Find the distance from the robot to the point where the line from robot to goal
intersects the wall.
            DistanceCheck = LineIntersectDistance(CameraSensors[CameraNum], Slope,
b, Xa, Ya, Xb, Yb, CameraArc);
            if (DistanceCheck < (TempEndOfWorldDistance - 5))
            { // This is true if the line making the wall is closer to the robot than the goal
is.

                // If this is false then the wall is not in the way and there is no need to check
the endpoints.

                // The -5 to TempEndOfWorldDistance is necessary because there may be
some rounding error. Since the goal point is on the wall, it is possible that a rounding error may
place the goal just slightly beyond the wall. This prevents that.

                // 5 is an arbitrary distance, but it should be long enough. The only times this
will be a problem is if there is a wall less than 5 inches from the goal. Because this will never
happen (it would make the goal impossible to reach anyway), this is a safe assumption.

```

```

Slope2 = (Ya - Yb) / (Xa - Xb);
b2 = Ya - Slope2 * Xa;
IntersectX = (b2 - b) / (Slope - Slope2);
IntersectY = Slope * IntersectX + b;

if (SegmentEndpoint[i, x] > SegmentEndpoint[i + 1, x])
{
    WallXLow = SegmentEndpoint[i + 1, x];
    WallXHigh = SegmentEndpoint[i, x];
}
else if (SegmentEndpoint[i, x] < SegmentEndpoint[i + 1, x])
{
    WallXLow = SegmentEndpoint[i, x];
    WallXHigh = SegmentEndpoint[i + 1, x];
}
else
{ // If WallXLow would be the same as WallXHigh, allow a slight leeway
for rounding errors.
    // RobotRadius is an appropriate allowance.
    WallXLow = SegmentEndpoint[i, x] - RobotRadius;
    WallXHigh = SegmentEndpoint[i, x] + RobotRadius;
}
if (SegmentEndpoint[i, y] > SegmentEndpoint[i + 1, y])
{
    WallYLow = SegmentEndpoint[i + 1, y];
    WallYHigh = SegmentEndpoint[i, y];
}
else if (SegmentEndpoint[i, y] < SegmentEndpoint[i + 1, y])
{
    WallYLow = SegmentEndpoint[i, y];
    WallYHigh = SegmentEndpoint[i + 1, y];
}
else
{
    WallYLow = SegmentEndpoint[i, y] - RobotRadius;
    WallYHigh = SegmentEndpoint[i, y] + RobotRadius;
}
if ((IntersectX > WallXLow) && (IntersectX < WallXHigh) && (IntersectY
> WallYLow) && (IntersectY < WallYHigh))
{
    CanSeeEndOfWorld[CurrentEndOfWorld] = false;
    i = NumEndpoints;
}
}
}

```

```

else
{
  Xa = SegmentEndpoint[i, x];
  Ya = SegmentEndpoint[i, y];
  Xb = SegmentEndpoint[0, x];
  Yb = SegmentEndpoint[0, y];

  DistanceCheck = LineIntersectDistance(CameraSensors[CameraNum], Slope,
b, Xa, Ya, Xb, Yb, CameraArc);
  if (DistanceCheck < (TempEndOfWorldDistance - 5))
  {
    Slope2 = (Ya - Yb) / (Xa - Xb);
    b2 = Ya - Slope2 * Xa;
    IntersectX = (b2 - b) / (Slope - Slope2);
    IntersectY = Slope * IntersectX + b;

    if (SegmentEndpoint[i, x] > SegmentEndpoint[0, x])
    {
      WallXLow = SegmentEndpoint[0, x];
      WallXHigh = SegmentEndpoint[i, x];
    }
    else if (SegmentEndpoint[i, x] < SegmentEndpoint[0, x])
    {
      WallXLow = SegmentEndpoint[i, x];
      WallXHigh = SegmentEndpoint[0, x];
    }
    else
    { // If WallXLow would be the same as WallXHigh, allow a slight leeway
for rounding errors.
      // RobotRadius is an appropriate allowance.
      WallXLow = SegmentEndpoint[i, x] - RobotRadius;
      WallXHigh = SegmentEndpoint[i, x] + RobotRadius;
    }
    if (SegmentEndpoint[i, y] > SegmentEndpoint[0, y])
    {
      WallYLow = SegmentEndpoint[0, y];
      WallYHigh = SegmentEndpoint[i, y];
    }
    else if (SegmentEndpoint[i, y] < SegmentEndpoint[0, y])
    {
      WallYLow = SegmentEndpoint[i, y];
      WallYHigh = SegmentEndpoint[0, y];
    }
    else
    {
      WallYLow = SegmentEndpoint[i, y] - RobotRadius;

```

```

        WallyYHigh = SegmentEndpoint[i, y] + RobotRadius;
    }
    if ((IntersectX > WallXLow) && (IntersectX < WallXHigh) && (IntersectY
> WallyYLow) && (IntersectY < WallyYHigh))
    {
        CanSeeEndOfWorld[CurrentEndOfWorld] = false;
        i = NumEndpoints;
    }
}
}
}
}
if (TempEndOfWorldDistance > MaxCameraView)
{
    CanSeeEndOfWorld[CurrentEndOfWorld] = false;
}
if (!CanSeeEndOfWorld[CurrentEndOfWorld])
{
    TempEndOfWorldDistance = ReallyLongDistance;
}

// Put the distance into EndOfWorldDistance.
EndOfWorldDistance[CurrentEndOfWorld] = TempEndOfWorldDistance;

}

// After checking with each end of world point, pick the closest one.
// Can't just use Array.Sort here since it would dissociate the distances from the
CanSeeEndOfWorld boolean values.
// Instead, use a loop with if/thens.
for (int i = 0; i < NumEndOfWorld; i++)
{ // Go through each of the distances from the previous loop.
    // Check to see if the distance is shorter than SortDistance.
    // If it is, make it the new SortDistance and make its matching CanSeeEndOfWorld the
new SortEndOfWorld.
    if (EndOfWorldDistance[i] < SortDistance)
    {
        SortDistance = EndOfWorldDistance[i];
        SortCanSeeEndOfWorld = CanSeeEndOfWorld[i];
    }
}
// At the end of this loop, we have the shortest of the distances in SortDistance and its
associated CanSeeEndOfWorld in SortCanSeeEndOfWorld.

// Call the function to convert the distance in inches to the camera reading.
SortDistance = CameraInchesToReading(SortDistance);

```

```

FinalEndOfWorldDistance = (int)InsertNoise(SortDistance, 10);

// Set the proper camera's SeeEndOfWorld to SortCanSeeEndOfWorld.
// NOTE: If more cameras are added, you MUST also add them manually to this series of
if statements.
if (CameraNum == 0)
{
    CameraVision0[SeeEndOfWorld] = SortCanSeeEndOfWorld;
}
else if (CameraNum == 1)
{
    CameraVision1[SeeEndOfWorld] = SortCanSeeEndOfWorld;
}
else if (CameraNum == 2)
{
    CameraVision2[SeeEndOfWorld] = SortCanSeeEndOfWorld;
}

return FinalEndOfWorldDistance;

}

```

```

/*****
*****/

```

```

private float CameraInchesToReading(float DistanceInches)
{
    // This function performs an equation to convert actual inches of distance into the reading
that the cameras return.
    // From experimentation, the cameras on the robot return values according to the
following formula:
    // Camera Reading = 366/(distance in feet) +- 10
    // Also, subtract the radius from the reading beforehand so the reading is from the sensor,
not the center of the robot.

    float CameraReading;
    // NOTE: commented out for now, reinstate this later
    // Commenting this fixes a big problem. Right now the comparisons don't work right in
'reading' scale but work fine in inches scale.
    // For now, keep it like this, change to fix later.
// CameraReading = 366 / ((DistanceInches - RobotRadius) / 12);
// NOTE: remove the following line after uncommenting the above line
    CameraReading = DistanceInches;

return CameraReading;
}

```

```
}
```

```
/*  
*****/
```

```
private bool RobotContacting()  
{  
    // This function returns true if the robot is touching a wall.  
    // If the robot has partly crossed over a wall, this function places it fully back into the  
arena tangent to the wall it crossed through.  
    // This is a simulation-only function used to prevent the robot from going through and  
ignoring the walls.  
    // Note that this function is only meant to work with round robots. If the robot is a shape  
other than a circle, the calculations involved here 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 uses the assumption 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 go into tight  
spaces, if they did exist, and should not be expected to squeeze into areas that are exactly as wide  
as (or even less than) its diameter.  
    // The robot acts a bit strange at some of the convex corners at times. This is because it  
attempts to 'cut' across the corner when moving straight toward it and is put back 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 is hugging the wall between  
SegmentEndpoint 4 and 3, approaching the corner formed with the wall between  
SegmentEndpoint 3 and 2, this happens, but not when approaching the same corner from the  
other direction).  
    // If the robot is not hugging the wall and just passes near the corner, this 'stuck'  
situation never happens.  
    // For the time being, this is an acceptable limitation, as the robot will want to avoid  
walls anyway. It may make initial learning a little slow until the robot learns obstacle avoidance,  
but should be a minor factor (or even a complete non-factor) after that.  
    // The speed of learning may ensure the situation never comes up, but it might.  
  
    float Slope;  
    float b;  
    float quadraticA;  
    float quadraticB;  
    float quadraticC;  
    float discriminant;
```

```

float IntersectX1 = RobotLocation[x];
float IntersectX2 = RobotLocation[x];
float IntersectY1 = RobotLocation[y];
float IntersectY2 = RobotLocation[y];
float IntersectXAvg = RobotLocation[x];
float IntersectYAvg = RobotLocation[y];

// Initialize these to -2. They will be changed before they're looked at, but are set to -2
just in case. This value will fail the failsafe check.
float Check1 = -2;
float Check2 = -2;

float Direction = RobotOrientation;
float Overlap;
bool testval = false;
bool foundintersection = false;
bool morecontacts = false;

// Use whichwalls to record which walls the robot is touching.
// The values in whichwalls are the starting segment point number of the intersected wall.
int[] whichwalls = new int[] { -1, -1 };

// First, find the location(s) where the perimeter of the robot crosses with a wall.
for (int i = 0; i < NumEndpoints; i++)
{ // This loop goes through all the arena walls to check for intersections.
  if (i != NumEndpoints - 1)
  {
    // wall endpoints
    SegmentEndpoint[i,x],SegmentEndpoint[i,y],SegmentEndpoint[i+1,x],SegmentEndpoint[i+1,y]
    // Find the intersection between a circle and a line.
    // First find the equation of the line of the wall.
    if (SegmentEndpoint[i, x] != SegmentEndpoint[i + 1, x])
    { // If a slope is possible
      Slope = (SegmentEndpoint[i, y] - SegmentEndpoint[i + 1, y]) /
(SegmentEndpoint[i, x] - SegmentEndpoint[i + 1, x]);
    }
    else
    {
      Slope = VeryHighSlope;
    }
    b = SegmentEndpoint[i, y] - Slope * SegmentEndpoint[i, x];
    // Now find the intersection between line and circle by setting the equations equal to
each other.
    // Circle: (x - LocX)^2 + (y - LocY)^2 = radius^2
    // Line: y = Slope * x + Intercept

```

```

// Expand circle equation:
//  $x^2 - 2*x*LocX + LocX^2 + y^2 - 2*y*LocY + LocY^2 - radius^2 = 0$ 
// Plug line equation in for y:
//  $x^2 - 2*x*LocX + LocX^2 + (Slope*x + Intercept)^2 - 2*(Slope*x +$ 
Intercept)*LocY + LocY^2 - radius^2 = 0
// Simplify:
//  $x^2 - 2*x*LocX + LocX^2 + Slope^2*x^2 + 2*Slope*x*Intercept + Intercept^2 -$ 
2*Slope*x*LocY - 2*Intercept*LocY + LocY^2 - radius^2 = 0
//  $x^2 + Slope^2*x^2 - 2*x*LocX + 2*Slope*x*Intercept - 2*Slope*x*LocY +$ 
LocX^2 + Intercept^2 - 2*Intercept*LocY + LocY^2 - radius^2 = 0
//  $(Slope^2 + 1)*x^2 + 2*(Slope*(Intercept - LocY) - LocX)*x + (LocX^2 +$ 
LocY^2 + Intercept^2 - radius^2 - 2*Intercept*LocY) = 0
// This makes a quadratic equation of the form  $ax^2 + bx + c = 0$ 
// So use the quadratic formula to find the points.
//  $a = Slope^2 + 1$ 
//  $b = 2*(Slope*(Intercept - LocY) - LocX)$ 
//  $c = LocX^2 + LocY^2 + Intercept^2 - radius^2 - 2*Intercept*LocY$ 
quadraticA = (float)Math.Pow(Slope, 2) + 1;
quadraticB = 2 * (Slope * (b - RobotLocation[y]) - RobotLocation[x]);
quadraticC = (float)(Math.Pow(RobotLocation[x], 2) +
Math.Pow(RobotLocation[y], 2) + Math.Pow(b, 2) - Math.Pow(RobotRadius, 2) - 2 * b *
RobotLocation[y]);
// Now find the discriminant.
discriminant = (float)Math.Pow(quadraticB, 2) - 4 * quadraticA * quadraticC;
// If discriminant is negative, there is no intersection. If it is 0, there is one
intersection. If it is positive, there are 2.
// Only bother with the case that discriminant >= 0.
// If it is 0, then both intersect points will be the same. This is fine.
if (discriminant >= 0)
{
// Perform the quadratic formula.
IntersectX1 = (float)(-quadraticB + Math.Sqrt(discriminant)) / (2 * quadraticA);
IntersectX2 = (float)(-quadraticB - Math.Sqrt(discriminant)) / (2 * quadraticA);
// Then insert the x points into the line formula to get the y points.
IntersectY1 = Slope * IntersectX1 + b;
IntersectY2 = Slope * IntersectX2 + b;

// Now check to make sure the intersection points are on the wall segment and not
outside it.
// This check works no matter which point (x1 or x2) is farther left.
// If x1 is farther left and the IntX point is between x1 and x2, Check becomes
positive/positive = positive (negative/positive=negative if IntX is not in the range)
// If x2 is farther left and the IntX point is between x2 and x1, Check becomes
negative/negative = positive (positive/negative=negative if IntX is not in the range)

```

```

        // If a middle-point is between two points, then the distance from one endpoint to
        that middle-point should be less than the distance between the endpoints, so the Check numbers
        should be <=1 if IntX is in the range.
        // Only the x coordinates are checked, since the intersection is between the y
        coordinates if and only if it is between the x coordinates.
        // Since Check1 and Check2 are only used for their values relative to 0 and 1 and
        not their actual numbers, we don't need to also check the y coordinates.
        Check1 = (IntersectX1 - SegmentEndpoint[i, x]) / (SegmentEndpoint[i + 1, x] -
        SegmentEndpoint[i, x]);
        Check2 = (IntersectX2 - SegmentEndpoint[i, x]) / (SegmentEndpoint[i + 1, x] -
        SegmentEndpoint[i, x]);

        // For the intersect point to be valid, ((Check1>=0)&&(Check1<=1)) ||
        ((Check2>=0)&&(Check1<=1))
        // This makes sure that either IntersectX1 or IntersectX2 are valid. The extra
        checks (with 1) are redundant but valid (Hal used them, I will keep them for now).
        if (((Check1 >= 0) && (Check1 <= 1)) || ((Check2 >= 0) && (Check2 <= 1)))
        {
            testval = true;
        }
    }

    if (testval && !foundintersection)
    {
        // Find the average of the two intersect points. If the points are the same then the
        average will be equal to the points.
        IntersectXAvg = (IntersectX1 + IntersectX2) / 2;
        IntersectYAvg = (IntersectY1 + IntersectY2) / 2;
        // Use the atan2 function to find the direction of the line that goes from the robot's
        location to this average intersection.
        if (IntersectXAvg != RobotLocation[x])
        { // Make sure the atan can be taken.
            Direction = (float)Math.Atan2(IntersectYAvg - RobotLocation[y],
            IntersectXAvg - RobotLocation[x]);
        }
        else
        {
            Direction = (float)Math.Atan2(IntersectYAvg - RobotLocation[y],
            VerySmallNumber);
        }

        // whichwalls[0] stores the location of the first wall the robot touches.
        // This is used only in the case of there being two walls touching the robot and
        serves as a double-failsafe check.
        if (whichwalls[0] < 0)
        {

```

```

        whichwalls[0] = i;
    }

    foundintersection = true;
    testval = false;
}

if (testval && foundintersection)
{
    // If this occurs, then the robot is intersecting with more than one wall (another
intersection found in a future run through the loop).
    // Set morecontacts true and find the set of intersects and direction for this wall as
well, then break out of the loop.
    // NOTE: 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 saying
arenas are not allowed to have 'tight spaces' where three or more walls make an alcove that the
robot can't enter.

    // whichwalls[1] stores the location of the second wall the robot touches
    if ((whichwalls[0] >= 0) && (whichwalls[1] < 0))
    {
        whichwalls[1] = i;
    }

    morecontacts = true;
    // No need to change testval and foundintersection here.
    // Break out of the loop.
    i = NumEndpoints;
}
}
else
{
    // wall endpoints
SegmentEndpoint[i,x],SegmentEndpoint[i,y],SegmentEndpoint[0,x],SegmentEndpoint[0,y]
// See comments above. This is for the final wall.
if (SegmentEndpoint[i, x] != SegmentEndpoint[0, x])
{
    Slope = (SegmentEndpoint[i, y] - SegmentEndpoint[0, y]) / (SegmentEndpoint[i,
x] - SegmentEndpoint[0, x]);
}
else
{
    Slope = VeryHighSlope;
}
}
}

```

```

    b = SegmentEndpoint[i, y] - Slope * SegmentEndpoint[i, x];
    quadraticA = (float)Math.Pow(Slope, 2) + 1;
    quadraticB = 2 * (Slope * (b - RobotLocation[y]) - RobotLocation[x]);
    quadraticC = (float)(Math.Pow(RobotLocation[x], 2) +
Math.Pow(RobotLocation[y], 2) + Math.Pow(b, 2) - Math.Pow(RobotRadius, 2) - 2 * b *
RobotLocation[y]);
    discriminant = (float)Math.Pow(quadraticB, 2) - 4 * quadraticA * quadraticC;
    if (discriminant >= 0)
    {
        IntersectX1 = (float)(-quadraticB + Math.Sqrt(discriminant)) / (2 * quadraticA);
        IntersectX2 = (float)(-quadraticB - Math.Sqrt(discriminant)) / (2 * quadraticA);
        IntersectY1 = Slope * IntersectX1 + b;
        IntersectY2 = Slope * IntersectX2 + b;
        Check1 = (IntersectX1 - SegmentEndpoint[i, x]) / (SegmentEndpoint[0, x] -
SegmentEndpoint[i, x]);
        Check2 = (IntersectX2 - SegmentEndpoint[i, x]) / (SegmentEndpoint[0, x] -
SegmentEndpoint[i, x]);
        if (((Check1 >= 0) && (Check1 <= 1)) || ((Check2 >= 0) && (Check2 <= 1)))
        {
            testval = true;
        }
    }
    if (testval && !foundintersection)
    {
        IntersectXAvg = (IntersectX1 + IntersectX2) / 2;
        IntersectYAvg = (IntersectY1 + IntersectY2) / 2;
        if (IntersectXAvg != RobotLocation[x])
        { // Make sure the atan can be taken.
            Direction = (float)Math.Atan2(IntersectYAvg - RobotLocation[y],
IntersectXAvg - RobotLocation[x]);
        }
        else
        {
            Direction = (float)Math.Atan2(IntersectYAvg - RobotLocation[y],
VerySmallNumber);
        }

        if (whichwalls[0] < 0)
        {
            whichwalls[0] = i;
        }

        foundintersection = true;
        testval = false;
    }
}

```

```

    if (testval && foundintersection)
    {
        if ((whichwalls[0] >= 0) && (whichwalls[1] < 0))
        {
            whichwalls[1] = i;
        }

        morecontacts = true;
        i = NumEndpoints;
    }
}

if (foundintersection && !morecontacts)
{
    // If there was a single intersection found.
    // Find Overlap, the distance that the edge of the robot 'overhangs' the wall.
    Overlap = RobotRadius - PointToPointDistance(RobotLocation[x], RobotLocation[y],
IntersectXAvg, IntersectYAvg);

    RobotLocation[x] = RobotLocation[x] - Overlap * (float)Math.Cos(Direction);
    RobotLocation[y] = RobotLocation[y] - Overlap * (float)Math.Sin(Direction);
}

if (foundintersection && morecontacts)
{
    // If there were two intersections found
    // Move away from the first wall that was encountered.
    Overlap = RobotRadius - PointToPointDistance(RobotLocation[x], RobotLocation[y],
IntersectXAvg, IntersectYAvg);
    RobotLocation[x] = RobotLocation[x] - Overlap * (float)Math.Cos(Direction);
    RobotLocation[y] = RobotLocation[y] - Overlap * (float)Math.Sin(Direction);

    // Now check the robot's position in relation to the second wall that was encountered.
    if (whichwalls[1] != NumEndpoints - 1)
    { // If the second wall is not at the end of the list
        // Go through the process of finding the intersections again.
        // The algorithm is the same as the first time this is used above. See there for
comments.
        if (SegmentEndpoint[whichwalls[1], x] != SegmentEndpoint[whichwalls[1] + 1, x])
        {
            Slope = (SegmentEndpoint[whichwalls[1], y] - SegmentEndpoint[whichwalls[1]
+ 1, y]) / (SegmentEndpoint[whichwalls[1], x] - SegmentEndpoint[whichwalls[1] + 1, x]);
        }
        else
        {

```

```

        Slope = VeryHighSlope;
    }
    b = SegmentEndpoint[whichwalls[1], y] - Slope * SegmentEndpoint[whichwalls[1],
x];

    quadraticA = (float)Math.Pow(Slope, 2) + 1;
    quadraticB = 2 * (Slope * (b - RobotLocation[y]) - RobotLocation[x]);
    quadraticC = (float)(Math.Pow(RobotLocation[x], 2) +
Math.Pow(RobotLocation[y], 2) + Math.Pow(b, 2) - Math.Pow(RobotRadius, 2) - 2 * b *
RobotLocation[y]);
    discriminant = (float)Math.Pow(quadraticB, 2) - 4 * quadraticA * quadraticC;
    if (discriminant >= 0)
    {
        IntersectX1 = (float)(-quadraticB + Math.Sqrt(discriminant)) / (2 * quadraticA);
        IntersectX2 = (float)(-quadraticB - Math.Sqrt(discriminant)) / (2 * quadraticA);
        IntersectY1 = Slope * IntersectX1 + b;
        IntersectY2 = Slope * IntersectX2 + b;
        Check1 = (IntersectX1 - SegmentEndpoint[whichwalls[1], x]) /
(SegmentEndpoint[whichwalls[1] + 1, x] - SegmentEndpoint[whichwalls[1], x]);
        Check2 = (IntersectX2 - SegmentEndpoint[whichwalls[1], x]) /
(SegmentEndpoint[whichwalls[1] + 1, x] - SegmentEndpoint[whichwalls[1], x]);
    }
}
else
{ // If the second wall is the last one on the list
    if (SegmentEndpoint[whichwalls[1], x] != SegmentEndpoint[0, x])
    {
        Slope = (SegmentEndpoint[whichwalls[1], y] - SegmentEndpoint[0, y]) /
(SegmentEndpoint[whichwalls[1], x] - SegmentEndpoint[0, x]);
    }
    else
    {
        Slope = VeryHighSlope;
    }
    b = SegmentEndpoint[whichwalls[1], y] - Slope * SegmentEndpoint[whichwalls[1],
x];

    quadraticA = (float)Math.Pow(Slope, 2) + 1;
    quadraticB = 2 * (Slope * (b - RobotLocation[y]) - RobotLocation[x]);
    quadraticC = (float)(Math.Pow(RobotLocation[x], 2) +
Math.Pow(RobotLocation[y], 2) + Math.Pow(b, 2) - Math.Pow(RobotRadius, 2) - 2 * b *
RobotLocation[y]);
    discriminant = (float)Math.Pow(quadraticB, 2) - 4 * quadraticA * quadraticC;
    if (discriminant >= 0)
    {
        IntersectX1 = (float)(-quadraticB + Math.Sqrt(discriminant)) / (2 * quadraticA);
        IntersectX2 = (float)(-quadraticB - Math.Sqrt(discriminant)) / (2 * quadraticA);
        IntersectY1 = Slope * IntersectX1 + b;

```

```

        IntersectY2 = Slope * IntersectX2 + b;
        Check1 = (IntersectX1 - SegmentEndpoint[whichwalls[1], x]) /
(SegmentEndpoint[0, x] - SegmentEndpoint[whichwalls[1], x]);
        Check2 = (IntersectX2 - SegmentEndpoint[whichwalls[1], x]) /
(SegmentEndpoint[0, x] - SegmentEndpoint[whichwalls[1], x]);
    }
}
if (((Check1 >= 0) && (Check1 <= 1)) || ((Check2 >= 0) && (Check2 <= 1)))
{ // Find the new IntersectXAvg, IntersectYAvg, Direction, Overlap, and change
RobotLocation
    IntersectXAvg = (IntersectX1 + IntersectX2) / 2;
    IntersectYAvg = (IntersectY1 + IntersectY2) / 2;
    if (IntersectXAvg != RobotLocation[x])
    { // Make sure the atan can be taken.
        Direction = (float)Math.Atan2(IntersectYAvg - RobotLocation[y], IntersectXAvg
- RobotLocation[x]);
    }
    else
    {
        Direction = (float)Math.Atan2(IntersectYAvg - RobotLocation[y],
VerySmallNumber);
    }
    Overlap = RobotRadius - PointToPointDistance(RobotLocation[x],
RobotLocation[y], IntersectXAvg, IntersectYAvg);
    RobotLocation[x] = RobotLocation[x] - Overlap * (float)Math.Cos(Direction);
    RobotLocation[y] = RobotLocation[y] - Overlap * (float)Math.Sin(Direction);
}

}

return foundintersection;

}

/*****
*****/

private void ArenaMap_Paint(object sender, PaintEventArgs e)
{
    // This function draws the initial arena
    System.Drawing.Graphics g = e.Graphics;
    Pen myPen = new Pen(Color.Black);
    g.PageUnit = GraphicsUnit.Pixel;

    // The following are used for drawing the Show Sensor lines

```

```

// The size must be changed for different sensors.
// The -2 is because 2 sonars are eliminated by combining them.
int tempanglesize = NumSonars - 2 + NumCameras;
double[] tempangle = new double[tempanglesize];

// Used for ShowSensors part.
// IsBumping is true if any of the sonars show Dist_Bump.
bool IsBumping = false;

// Used for Show HUD part.
// HUDBumping is true if any of the sonars show Dist_Bump.
bool HUDBumping = false;

for (int i = 0; i < NumEndpoints; i++)
{
    if (i != NumEndpoints - 1)
    {
        g.DrawLine(myPen, SegmentEndpoint[i, x], SegmentEndpoint[i, y],
SegmentEndpoint[i + 1, x], SegmentEndpoint[i + 1, y]);
    }
    else
    {
        g.DrawLine(myPen, SegmentEndpoint[i, x], SegmentEndpoint[i, y],
SegmentEndpoint[0, x], SegmentEndpoint[0, y]);
    }
}

Brush myStartBrush = new SolidBrush(Color.Green);
Pen myStartPen = new Pen(myStartBrush);
g.DrawEllipse(myStartPen, StartDot[x] + 5, StartDot[y] - 5, -10, 10);
g.FillEllipse(myStartBrush, StartDot[x] + 5, StartDot[y] - 5, -10, 10);

Brush myGoalBrush = new SolidBrush(Color.Blue);
Pen myGoalPen = new Pen(myGoalBrush);
g.DrawEllipse(myGoalPen, GoalDot[x] + 5, GoalDot[y] - 5, -10, 10);
g.FillEllipse(myGoalBrush, GoalDot[x] + 5, GoalDot[y] - 5, -10, 10);

// Draw the end of the world points
// Fix the visuals later since they look off-center.
Brush myEndOfWorldBrush = new SolidBrush(Color.Yellow);
Pen myEndOfWorldPen = new Pen(myEndOfWorldBrush);
for (int i = 0; i < NumEndOfWorld; i++)
{
    g.DrawEllipse(myEndOfWorldPen, EndOfWorldPoints[i, x] + 5, EndOfWorldPoints[i,
y] - 5, -10, 10);
}

```

```

        g.FillEllipse(myEndOfWorldBrush, EndOfWorldPoints[i, x] + 5, EndOfWorldPoints[i,
y] - 5, -10, 10);
    }

```

```

    Brush myRobotBrush = new SolidBrush(Color.Red);
    Pen myRobotPen = new Pen(myRobotBrush);
    g.DrawEllipse(myPen, RobotLocation[x] - RobotRadius, RobotLocation[y] -
RobotRadius, 2 * RobotRadius, 2 * RobotRadius);
    g.FillEllipse(myRobotBrush, RobotLocation[x] - RobotRadius, RobotLocation[y] -
RobotRadius, 2 * RobotRadius, 2 * RobotRadius);

```

```

    Brush myDirectionBrush = new SolidBrush(Color.Black);
    Pen myDirectionPen = new Pen(myDirectionBrush);
    g.DrawLine(myDirectionPen, RobotLocation[x], RobotLocation[y], RobotLocation[x] +
(RobotRadius * (float)Math.Cos(RobotOrientation)), RobotLocation[y] + (RobotRadius *
(float)Math.Sin(RobotOrientation)));

```

```

if (Manual_Mode_Toggle)
{
    Manual_Mode_Label.Text = "Manual Mode ON";
    if (Manual_Choice != Manual_None)
    {
        Manual_Choice_Title.Text = "Manual Choice";
    }
    else
    {
        Manual_Choice_Title.Text = "";
    }
    if (Manual_Choice == Manual_None)
    {
        Manual_Choice_Label.Text = "";
    }
    else if (Manual_Choice == Manual_Forward)
    {
        Manual_Choice_Label.Text = "Forward";
    }
    else if (Manual_Choice == Manual_Left)
    {
        Manual_Choice_Label.Text = "Left";
    }
    else if (Manual_Choice == Manual_Right)
    {
        Manual_Choice_Label.Text = "Right";
    }
    else if (Manual_Choice == Manual_Stop)
    {

```

```

    Manual_Choice_Label.Text = "Stop";
}
else if (Manual_Choice == Manual_Reverse)
{
    Manual_Choice_Label.Text = "Reverse";
}
else if (Manual_Choice == Manual_Big_Left)
{
    Manual_Choice_Label.Text = "Big Left";
}
else if (Manual_Choice == Manual_Big_Right)
{
    Manual_Choice_Label.Text = "Big Right";
}
else
{
    Manual_Choice_Label.Text = "";
}
}
else
{
    Manual_Mode_Label.Text = "Manual Mode OFF";
    Manual_Choice_Title.Text = "";
    Manual_Choice_Label.Text = "";
}
}

if (Auto_Rescue_Toggle)
{ // If the Auto Rescue button is toggled on, show that it is.
    Auto_Rescue_Label.Text = "Auto Rescue On";
}
else
{ // Otherwise, show it's off.
    Auto_Rescue_Label.Text = "Auto Rescue Off";
}
}

if (Has_Been_Rescued)
{ // If the robot has been rescued this episode, display that it has.
    Has_Been_Rescued_Label.Text = "Has Been Rescued";
}
else
{
    Has_Been_Rescued_Label.Text = "";
}
}

if (Show_Stats_Toggle)
{ // If the Show Stats toggle is active, display the statistics in the box.

```

```

Show_Stats_Label_Title.Text = "Simulation Statistics";
Show_Stats_Title_CurReward.Text = "Current Episode Reward:";
Show_Stats_Title_CurSteps.Text = "Current Episode Steps:";
Show_Stats_Title_CurBumps.Text = "Current Episode Bumps:";
Show_Stats_Title_NumRep.Text = "Number of Episodes:";
Show_Stats_Title_NumGoals.Text = "Total Number of Goals:";
Show_Stats_Title_AvgReward.Text = "Average Reward per Episode:";
Show_Stats_Title_AvgSteps.Text = "Average Steps per Episode:";
Show_Stats_Title_AvgBumps.Text = "Average Bumps per Episode:";
Show_Stats_Label_CurReward.Text = CurrentReward.ToString();
Show_Stats_Label_CurSteps.Text = CurrentSteps.ToString();
Show_Stats_Label_CurBumps.Text = CurrentWallHits.ToString();
Show_Stats_Label_NumRep.Text = NumRepetitions.ToString();
Show_Stats_Label_NumGoals.Text = NumberGoals.ToString();
Show_Stats_Label_AvgReward.Text = AvgReward.ToString();
Show_Stats_Label_AvgSteps.Text = AvgSteps.ToString();
Show_Stats_Label_AvgBumps.Text = AvgWallHits.ToString();
}
else
{ // Otherwise, show nothing there.
Show_Stats_Label_Title.Text = "";
Show_Stats_Title_CurReward.Text = "";
Show_Stats_Title_CurSteps.Text = "";
Show_Stats_Title_CurBumps.Text = "";
Show_Stats_Title_NumRep.Text = "";
Show_Stats_Title_NumGoals.Text = "";
Show_Stats_Title_AvgReward.Text = "";
Show_Stats_Title_AvgSteps.Text = "";
Show_Stats_Title_AvgBumps.Text = "";
Show_Stats_Label_CurReward.Text = "";
Show_Stats_Label_CurSteps.Text = "";
Show_Stats_Label_CurBumps.Text = "";
Show_Stats_Label_NumRep.Text = "";
Show_Stats_Label_NumGoals.Text = "";
Show_Stats_Label_AvgReward.Text = "";
Show_Stats_Label_AvgSteps.Text = "";
Show_Stats_Label_AvgBumps.Text = "";
}

if (Show_Q_Toggle)
{ // If the Show Q toggle is on, display the Q values.
Show_Q_Label_Title.Text = "Q Table Values";
Show_Q_Title_Random.Text = "Random Choice";
Show_Q_Title_Forward.Text = "Forward Reward";
Show_Q_Title_Reverse.Text = "Reverse Reward";
}

```

```

Show_Q_Title_Stop.Text = "Stop Reward";
Show_Q_Title_Left.Text = "Left Reward";
Show_Q_Title_Right.Text = "Right Reward";
Show_Q_Title_Big_Left.Text = "Big Left Reward";
Show_Q_Title_Big_Right.Text = "Big Right Reward";
Show_Q_Title_Choice.Text = "Choice";
Show_Q_Title_Current.Text = "Current Reward";

if (Table_Values[TableTestRandom] == TableRandomForward)
{
    Show_Q_Label_Random.Text = "Forward";
}
else if (Table_Values[TableTestRandom] == TableRandomReverse)
{
    Show_Q_Label_Random.Text = "Reverse";
}
else if (Table_Values[TableTestRandom] == TableRandomStop)
{
    Show_Q_Label_Random.Text = "Stop";
}
else if (Table_Values[TableTestRandom] == TableRandomLeft)
{
    Show_Q_Label_Random.Text = "Left";
}
else if (Table_Values[TableTestRandom] == TableRandomRight)
{
    Show_Q_Label_Random.Text = "Right";
}
else if (Table_Values[TableTestRandom] == TableRandomBigLeft)
{
    Show_Q_Label_Random.Text = "Big Left";
}
else if (Table_Values[TableTestRandom] == TableRandomBigRight)
{
    Show_Q_Label_Random.Text = "Big Right";
}
else
{
    Show_Q_Label_Random.Text = "No Random";
}

Show_Q_Label_Forward.Text = Table_Values[TableTestForward].ToString();
Show_Q_Label_Reverse.Text = Table_Values[TableTestReverse].ToString();
Show_Q_Label_Stop.Text = Table_Values[TableTestStop].ToString();
Show_Q_Label_Left.Text = Table_Values[TableTestLeft].ToString();
Show_Q_Label_Right.Text = Table_Values[TableTestRight].ToString();

```

```

Show_Q_Label_Big_Left.Text = Table_Values[TableTestBigLeft].ToString();
Show_Q_Label_Big_Right.Text = Table_Values[TableTestBigRight].ToString();

if (TableChoice == Action_Forward)
{
    Show_Q_Label_Choice.Text = "FORWARD";
}
else if (TableChoice == Action_Reverse)
{
    Show_Q_Label_Choice.Text = "REVERSE";
}
else if (TableChoice == Action_Stop)
{
    Show_Q_Label_Choice.Text = "STOP";
}
else if (TableChoice == Action_Rotate_Left)
{
    Show_Q_Label_Choice.Text = "LEFT";
}
else if (TableChoice == Action_Rotate_Right)
{
    Show_Q_Label_Choice.Text = "RIGHT";
}
else if (TableChoice == Action_Big_Rotate_Left)
{
    Show_Q_Label_Choice.Text = "BIG LEFT";
}
else if (TableChoice == Action_Big_Rotate_Right)
{
    Show_Q_Label_Choice.Text = "BIG RIGHT";
}

Show_Q_Label_Current.Text = Reward.ToString();
}
else
{ // Otherwise, show nothing there.
    Show_Q_Label_Title.Text = "";
    Show_Q_Title_Random.Text = "";
    Show_Q_Title_Forward.Text = "";
    Show_Q_Title_Reverse.Text = "";
    Show_Q_Title_Stop.Text = "";
    Show_Q_Title_Left.Text = "";
    Show_Q_Title_Right.Text = "";
    Show_Q_Title_Big_Left.Text = "";
    Show_Q_Title_Big_Right.Text = "";
    Show_Q_Title_Choice.Text = "";
}

```

```

Show_Q_Title_Current.Text = "";
Show_Q_Label_Random.Text = "";
Show_Q_Label_Forward.Text = "";
Show_Q_Label_Reverse.Text = "";
Show_Q_Label_Stop.Text = "";
Show_Q_Label_Left.Text = "";
Show_Q_Label_Right.Text = "";
Show_Q_Label_Big_Left.Text = "";
Show_Q_Label_Big_Right.Text = "";
Show_Q_Label_Choice.Text = "";
Show_Q_Label_Current.Text = "";
}

if (Show_HUD_Toggle)
{ // If the Show HUD toggle is on, display the sensor values.
Show_HUD_Label_Title.Text = "HUD";
Show_HUD_Title_SonarF.Text = "Front Sonar:";
Show_HUD_Title_SonarFL.Text = "Front Left Sonar:";
Show_HUD_Title_SonarFR.Text = "Front Right Sonar:";
Show_HUD_Title_SonarL.Text = "Left Sonar:";
Show_HUD_Title_SonarR.Text = "Right Sonar:";
Show_HUD_Title_SonarB.Text = "Back Sonar:";
Show_HUD_Title_GoalCamF.Text = "Front Goal Camera:";
Show_HUD_Title_GoalCamL.Text = "Left Goal Camera:";
Show_HUD_Title_GoalCamR.Text = "Right Goal Camera:";
Show_HUD_Title_EoWCamF.Text = "Front End of World Camera:";
Show_HUD_Title_EoWCamL.Text = "Left End of World Camera:";
Show_HUD_Title_EoWCamR.Text = "Right End of World Camera:";

// Check to see if the robot is bumping a wall.
// If any of the sonars say bumping then HUDBumping will be true.
for (int i = 0; i < (tempanglesize - NumCameras); i++)
{ // Go through the tempangles for Sonars
  if (State[State_Sonar0 + i] == Dist_Bump)
  { // If the sonar reads Bump distance, set ISBumping to true.
    HUDBumping = true;
  }
}
}
if (HUDBumping)
{ // If any sonars are bumping.
  Show_HUD_Bump.Text = "BUMP";
}
else
{
  Show_HUD_Bump.Text = "";
}
}

```

```

// Check each sonar distance.
// Front Sonar
if ((State[State_Sonar0] == Dist_Close) || (State[State_Sonar0] == Dist_Bump))
{
    Show_HUD_Label_SonarF.Text = "Close";
}
else if (State[State_Sonar0] == Dist_Medium)
{
    Show_HUD_Label_SonarF.Text = "Medium";
}
else
{
    Show_HUD_Label_SonarF.Text = "Far";
}

// Front-Left Sonar
if ((State[State_Sonar1_2] == Dist_Close) || (State[State_Sonar1_2] == Dist_Bump))
{
    Show_HUD_Label_SonarFL.Text = "Close";
}
else if (State[State_Sonar1_2] == Dist_Medium)
{
    Show_HUD_Label_SonarFL.Text = "Medium";
}
else
{
    Show_HUD_Label_SonarFL.Text = "Far";
}

// Front-Right Sonar
if ((State[State_Sonar5_6] == Dist_Close) || (State[State_Sonar5_6] == Dist_Bump))
{
    Show_HUD_Label_SonarFR.Text = "Close";
}
else if (State[State_Sonar5_6] == Dist_Medium)
{
    Show_HUD_Label_SonarFR.Text = "Medium";
}
else
{
    Show_HUD_Label_SonarFR.Text = "Far";
}

// Left Sonar
if ((State[State_Sonar3] == Dist_Close) || (State[State_Sonar3] == Dist_Bump))

```

```

{
    Show_HUD_Label_SonarL.Text = "Close";
}
else if (State[State_Sonar3] == Dist_Medium)
{
    Show_HUD_Label_SonarL.Text = "Medium";
}
else
{
    Show_HUD_Label_SonarL.Text = "Far";
}

// Right Sonar
if ((State[State_Sonar7] == Dist_Close) || (State[State_Sonar7] == Dist_Bump))
{
    Show_HUD_Label_SonarR.Text = "Close";
}
else if (State[State_Sonar7] == Dist_Medium)
{
    Show_HUD_Label_SonarR.Text = "Medium";
}
else
{
    Show_HUD_Label_SonarR.Text = "Far";
}

// Back Sonar
if ((State[State_Sonar4] == Dist_Close) || (State[State_Sonar4] == Dist_Bump))
{
    Show_HUD_Label_SonarB.Text = "Close";
}
else if (State[State_Sonar4] == Dist_Medium)
{
    Show_HUD_Label_SonarB.Text = "Medium";
}
else
{
    Show_HUD_Label_SonarB.Text = "Far";
}

// Now check cameras

// Front GoalCam
if (State[State_Goal0] == Dist_Close)
{
    Show_HUD_Label_GoalCamF.Text = "Close";
}

```

```

}
else if (State[State_Goal0] == Dist_Medium)
{
    Show_HUD_Label_GoalCamF.Text = "Medium";
}
else
{
    Show_HUD_Label_GoalCamF.Text = "Far";
}

// Left GoalCam
if (State[State_Goal2] == Dist_Close)
{
    Show_HUD_Label_GoalCamL.Text = "Close";
}
else if (State[State_Goal2] == Dist_Medium)
{
    Show_HUD_Label_GoalCamL.Text = "Medium";
}
else
{
    Show_HUD_Label_GoalCamL.Text = "Far";
}

// Right GoalCam
if (State[State_Goal1] == Dist_Close)
{
    Show_HUD_Label_GoalCamR.Text = "Close";
}
else if (State[State_Goal1] == Dist_Medium)
{
    Show_HUD_Label_GoalCamR.Text = "Medium";
}
else
{
    Show_HUD_Label_GoalCamR.Text = "Far";
}

// Front EoWCam
if (State[State_EndOfWorld0] == Dist_Close)
{
    Show_HUD_Label_EoWCamF.Text = "Close";
}

```

```

}
else if (State[State_EndOfWorld0] == Dist_Medium)
{
    Show_HUD_Label_EoWCamF.Text = "Medium";
}
else
{
    Show_HUD_Label_EoWCamF.Text = "Far";
}

// Left EoWCam
if (State[State_EndOfWorld2] == Dist_Close)
{
    Show_HUD_Label_EoWCamL.Text = "Close";
}
else if (State[State_EndOfWorld2] == Dist_Medium)
{
    Show_HUD_Label_EoWCamL.Text = "Medium";
}
else
{
    Show_HUD_Label_EoWCamL.Text = "Far";
}

// Right EoWCam
if (State[State_EndOfWorld1] == Dist_Close)
{
    Show_HUD_Label_EoWCamR.Text = "Close";
}
else if (State[State_EndOfWorld1] == Dist_Medium)
{
    Show_HUD_Label_EoWCamR.Text = "Medium";
}
else
{
    Show_HUD_Label_EoWCamR.Text = "Far";
}

// DoubleQ
CanSeeGoalTitle.Text = "Can See Goal";
if (CanSeeGoal)
{
    CanSeeGoalLabel.Text = "Yes";
}

```

```

else
{
    CanSeeGoalLabel.Text = "No";
}

Show_HUD_Title_Location.Text = "Location";
Show_HUD_Title_Orientation.Text = "Orientation";
Show_HUD_Label_Location.Text = "(" + RobotLocation[x].ToString() + "," +
RobotLocation[y].ToString() + ")";
Show_HUD_Label_Orientation.Text = RobotOrientation.ToString();

// CameraFocus
if (State[State_Focus0] == CameraFocusLeft)
{
    CameraFocusFrontLabel.Text = "LEFT";
}
else if (State[State_Focus0] == CameraFocusRight)
{
    CameraFocusFrontLabel.Text = "RIGHT";
}
else if (State[State_Focus0] == CameraFocusCenter)
{
    CameraFocusFrontLabel.Text = "CENTER";
}
else
{
    CameraFocusFrontLabel.Text = "";
}

if (State[State_Focus2] == CameraFocusLeft)
{
    CameraFocusLeftLabel.Text = "LEFT";
}
else if (State[State_Focus2] == CameraFocusRight)
{
    CameraFocusLeftLabel.Text = "RIGHT";
}
else if (State[State_Focus2] == CameraFocusCenter)
{
    CameraFocusLeftLabel.Text = "CENTER";
}
else
{
    CameraFocusLeftLabel.Text = "";
}

```

```

if (State[State_Focus1] == CameraFocusLeft)
{
    CameraFocusRightLabel.Text = "LEFT";
}
else if (State[State_Focus1] == CameraFocusRight)
{
    CameraFocusRightLabel.Text = "RIGHT";
}
else if (State[State_Focus1] == CameraFocusCenter)
{
    CameraFocusRightLabel.Text = "CENTER";
}
else
{
    CameraFocusRightLabel.Text = "";
}
// end CameraFocus

Show_HUD_Title_Keyword.Text = "Keyword";
Show_HUD_Label_Keyword.Text = GlobalKeyword.ToString();
Show_HUD_Title_KeywordA.Text = "Keyword A";
Show_HUD_Label_KeywordA.Text = GlobalKeywordA.ToString();
Show_HUD_Title_KeywordB.Text = "Keyword B";
Show_HUD_Label_KeywordB.Text = GlobalKeywordB.ToString();

}
else
{ // Otherwise, show nothing here.
    Show_HUD_Label_Title.Text = "";
    Show_HUD_Title_SonarF.Text = "";
    Show_HUD_Title_SonarFL.Text = "";
    Show_HUD_Title_SonarFR.Text = "";
    Show_HUD_Title_SonarL.Text = "";
    Show_HUD_Title_SonarR.Text = "";
    Show_HUD_Title_SonarB.Text = "";
    Show_HUD_Title_GoalCamF.Text = "";
    Show_HUD_Title_GoalCamL.Text = "";
    Show_HUD_Title_GoalCamR.Text = "";
    Show_HUD_Title_EoWCamF.Text = "";
    Show_HUD_Title_EoWCamL.Text = "";
    Show_HUD_Title_EoWCamR.Text = "";
    Show_HUD_Label_SonarF.Text = "";
    Show_HUD_Label_SonarFL.Text = "";

```

```

Show_HUD_Label_SonarFR.Text = "";
Show_HUD_Label_SonarL.Text = "";
Show_HUD_Label_SonarR.Text = "";
Show_HUD_Label_SonarB.Text = "";
Show_HUD_Label_GoalCamF.Text = "";
Show_HUD_Label_GoalCamL.Text = "";
Show_HUD_Label_GoalCamR.Text = "";
Show_HUD_Label_EoWCamF.Text = "";
Show_HUD_Label_EoWCamL.Text = "";
Show_HUD_Label_EoWCamR.Text = "";
Show_HUD_Bump.Text = "";

// DoubleQ
CanSeeGoalTitle.Text = "";
CanSeeGoalLabel.Text = "";

Show_HUD_Title_Location.Text = "";
Show_HUD_Title_Orientation.Text = "";
Show_HUD_Label_Location.Text = "";
Show_HUD_Label_Orientation.Text = "";

// CameraFocus
CameraFocusFrontLabel.Text = "";
CameraFocusLeftLabel.Text = "";
CameraFocusRightLabel.Text = "";
// end CameraFocus

Show_HUD_Title_Keyword.Text = "";
Show_HUD_Label_Keyword.Text = "";
Show_HUD_Title_KeywordA.Text = "";
Show_HUD_Label_KeywordA.Text = "";
Show_HUD_Title_KeywordB.Text = "";
Show_HUD_Label_KeywordB.Text = "";

}

if (Clockless_Timer_Toggle)
{ // If the clockless timer is activated, display so here.
  Clockless_Timer_Status.Text = "Clockless Timer Active";
}
else
{
  Clockless_Timer_Status.Text = "";
}

```

```

if (Show_Sensors_Toggle)
{ // If the Show Sensors toggle is active, draw lines showing the sensors.

    Pen mySonarClosePen = new Pen(Color.Black);
    Pen mySonarMediumPen = new Pen(Color.Gray);
    Pen myGoalClosePen = new Pen(Color.DarkCyan);
    Pen myGoalMediumPen = new Pen(Color.Cyan);
    Pen myEoWClosePen = new Pen(Color.DarkGoldenrod);
    Pen myEoWMediumPen = new Pen(Color.Yellow);

    // Sonars first
    // NOTE: If the number of sonars change or a larger Q-table is used that doesn't
combine sensors, this has to be changed!
    // tempangle has the following setup:
    // tempangle[0] is Sonar 0
    // tempangle[1] is Sonars 1 and 2
    // tempangle[2] is Sonar 3
    // tempangle[3] is Sonar 4
    // tempangle[4] is Sonars 5 and 6
    // tempangle[5] is Sonar 7
    // tempangle[6] is Camera 0
    // tempangle[7] is Camera 1
    // tempangle[8] is Camera 2
    tempangle[0] = RobotOrientation + SonarSensors[0];
    tempangle[1] = RobotOrientation + ((SonarSensors[1] + SonarSensors[2]) / 2);
    tempangle[2] = RobotOrientation + SonarSensors[3];
    tempangle[3] = RobotOrientation + SonarSensors[4];
    tempangle[4] = RobotOrientation + ((SonarSensors[5] + SonarSensors[6]) / 2);
    tempangle[5] = RobotOrientation + SonarSensors[7];
    tempangle[6] = RobotOrientation + CameraSensors[0];
    tempangle[7] = RobotOrientation + CameraSensors[1];
    tempangle[8] = RobotOrientation + CameraSensors[2];

    // Now make sure all the angles are between 0 and 2*Pi
    for (int i = 0; i < tempanglesize; i++)
    {
        while (tempangle[i] < 0)
        {
            tempangle[i] = tempangle[i] + twopi;
        }
        while (tempangle[i] > twopi)
        {
            tempangle[i] = tempangle[i] - twopi;
        }
    }
}

```

```

    }

    // Check to see if the robot is bumping a wall.
    // If any of the sonars say bumping then IsBumping will be true.
    for (int i = 0; i < (tempanglesize - NumCameras); i++)
    { // Go through the tempangles for Sonars
      if (State[State_Sonar0 + i] == Dist_Bump)
      { // If the sonar reads Bump distance, set ISBumping to true.
        IsBumping = true;
      }
    }

    // If IsBumping is true, draw the robot as a different colored circle.
    if (IsBumping)
    {

      Brush myBumpBrush = new SolidBrush(Color.LightCoral);
      Pen myBumpPen = new Pen(myBumpBrush);
      g.DrawEllipse(myPen, RobotLocation[x] - RobotRadius, RobotLocation[y] -
RobotRadius, 2 * RobotRadius, 2 * RobotRadius);
      g.FillEllipse(myBumpBrush, RobotLocation[x] - RobotRadius, RobotLocation[y] -
RobotRadius, 2 * RobotRadius, 2 * RobotRadius);

      g.DrawLine(myDirectionPen, RobotLocation[x], RobotLocation[y],
RobotLocation[x] + (RobotRadius * (float)Math.Cos(RobotOrientation)), RobotLocation[y] +
(RobotRadius * (float)Math.Sin(RobotOrientation)));
    }

    // If AtEndOfWorld is true, draw the robot as a different colored circle.
    if (AtEndOfWorld)
    {
      Brush myEoWBrush = new SolidBrush(Color.DarkViolet);
      Pen myEoWPen = new Pen(myEoWBrush);

      g.DrawEllipse(myPen, RobotLocation[x] - RobotRadius, RobotLocation[y] -
RobotRadius, 2 * RobotRadius, 2 * RobotRadius);
      g.FillEllipse(myEoWBrush, RobotLocation[x] - RobotRadius, RobotLocation[y] -
RobotRadius, 2 * RobotRadius, 2 * RobotRadius);

      g.DrawLine(myDirectionPen, RobotLocation[x], RobotLocation[y],
RobotLocation[x] + (RobotRadius * (float)Math.Cos(RobotOrientation)), RobotLocation[y] +
(RobotRadius * (float)Math.Sin(RobotOrientation)));
    }

```

```

// If AtGoal is true, draw the robot as a different colored circle.
if (AtGoal)
{
    Brush myAtGoalBrush = new SolidBrush(Color.Black);
    Pen myAtGoalPen = new Pen(myAtGoalBrush);

    g.DrawEllipse(myPen, RobotLocation[x] - RobotRadius, RobotLocation[y] -
RobotRadius, 2 * RobotRadius, 2 * RobotRadius);
    g.FillEllipse(myAtGoalBrush, RobotLocation[x] - RobotRadius, RobotLocation[y] -
RobotRadius, 2 * RobotRadius, 2 * RobotRadius);

    g.DrawLine(myDirectionPen, RobotLocation[x], RobotLocation[y],
RobotLocation[x] + (RobotRadius * (float)Math.Cos(RobotOrientation)), RobotLocation[y] +
(RobotRadius * (float)Math.Sin(RobotOrientation)));
}

// Now draw the lines.

// Draw the camera lines. Draw the End of World line first and the Goal line second.
This should prioritize the Goal line.
for (int i = tempanglesize - NumCameras; i < tempanglesize; i++)
{ // Go through the tempangles for Cameras
// Draw the End of the World lines first.
if (State[State_EndOfWorld0 + i - (tempanglesize - NumCameras)] == Dist_Close)
{ // If the camera reads Close for End of the World, draw the line in EoWCclosePen
color.
    g.DrawLine(myEoWCclosePen, RobotLocation[x], RobotLocation[y],
RobotLocation[x] + Camera_Close_Inches * (float)Math.Cos(tempangle[i]), RobotLocation[y] +
Camera_Close_Inches * (float)Math.Sin(tempangle[i]));
}
if (State[State_EndOfWorld0 + i - (tempanglesize - NumCameras)] ==
Dist_Medium)
{ // If the camera reads Medium for End of the World, draw the line in
EoWMediumPen color.
    g.DrawLine(myEoWMediumPen, RobotLocation[x], RobotLocation[y],
RobotLocation[x] + Camera_Far_Inches * (float)Math.Cos(tempangle[i]), RobotLocation[y] +
Camera_Far_Inches * (float)Math.Sin(tempangle[i]));
}

// Now draw the Goal lines.
if (State[State_Goal0 + i - (tempanglesize - NumCameras)] == Dist_Close)
{ // If the camera reads Close for Goal, draw the line in GoalClosePen color.

```

```

        g.DrawLine(myGoalClosePen, RobotLocation[x], RobotLocation[y],
RobotLocation[x] + Camera_Close_Inches * (float)Math.Cos(tempangle[i]), RobotLocation[y] +
Camera_Close_Inches * (float)Math.Sin(tempangle[i]));
    }
    if (State[State_Goal0 + i - (tempanglesize - NumCameras)] == Dist_Medium)
    { // If the camera reads MEdium for Goal, draw the line in GoalMediumPen color.
        g.DrawLine(myGoalMediumPen, RobotLocation[x], RobotLocation[y],
RobotLocation[x] + Camera_Far_Inches * (float)Math.Cos(tempangle[i]), RobotLocation[y] +
Camera_Far_Inches * (float)Math.Sin(tempangle[i]));
    }
}

// Draw the sonar lines.
for (int i = 0; i < (tempanglesize - NumCameras); i++)
{ // Go through the tempangles for Sonars
    if (State[State_Sonar0 + i] == Dist_Close)
    { // If the sonar reads Close distance, draw the line in SonarClosePen color.
        g.DrawLine(mySonarClosePen, RobotLocation[x], RobotLocation[y],
RobotLocation[x] + (Sonar_Close / 2) * (float)Math.Cos(tempangle[i]), RobotLocation[y] +
(Sonar_Close / 2) * (float)Math.Sin(tempangle[i]));
    }
    if (State[State_Sonar0 + i] == Dist_Medium)
    { // If the sonar reads Medium distance, draw the line in SonarMediumPen color.
        g.DrawLine(mySonarMediumPen, RobotLocation[x], RobotLocation[y],
RobotLocation[x] + (Sonar_Far / 2) * (float)Math.Cos(tempangle[i]), RobotLocation[y] +
(Sonar_Far / 2) * (float)Math.Sin(tempangle[i]));
    }
}
}

/*
// Trail doesn't work yet. Keep this commented out.
// Draw the trail
if (Trail_Toggle)
{
    g.DrawEllipse(myPen, OldLocation[x] - RobotRadius, OldLocation[y] - RobotRadius,
2 * RobotRadius, 2 * RobotRadius);
    g.FillEllipse(myRobotBrush, OldLocation[x] - RobotRadius, OldLocation[y] -
RobotRadius, 2 * RobotRadius, 2 * RobotRadius);
    g.DrawLine(myDirectionPen, OldLocation[x], OldLocation[y], OldLocation[x] +
(RobotRadius * (float)Math.Cos(OldOrientation)), RobotLocation[y] + (RobotRadius *
(float)Math.Sin(OldOrientation)));
}
}

```

```

    */

}

}

/*****
*****/
// NOTE: comment out this whole function for smaller Q-table
/* private int SensorEncodeCompass(int CompassReading)
{ // This function takes in the reading from the compass and returns an encoded range
value from 0 to 3.
    // Compass_Northwest, Compass_Northeast, Compass_Southwest, and
Compass_Southeast are all defined in the globals.

    // Compass_Range has to be initialized. This value should be changed to one of the
four headings, but it is initialized at North by default.
    int Compass_Range = Heading_North;

    if ((CompassReading <= Compass_Northwest) && (CompassReading >=
Compass_Northeast))
    { // If the reading is between Northwest and Northeast, then the reading is North.
        Compass_Range = Heading_North;
    }
    else if ((CompassReading <= Compass_Southwest) && (CompassReading >=
Compass_Northwest))
    { // If the reading is between Northwest and Southwest, then the reading is West.
        Compass_Range = Heading_West;
    }
    else if ((CompassReading <= Compass_Southeast) || (CompassReading >=
Compass_Southwest))
    { // If the reading is between Southwest and Southeast, then the reading is South.
        // Note that this one uses OR instead of AND. This is because the reading 0
occurs between Southwest and Southeast.
        Compass_Range = Heading_South;
    }
    else if ((CompassReading <= Compass_Northeast) && (CompassReading >=
Compass_Southeast))
    { // If the reading is between Southeast and Northeast, then the reading is East.
        Compass_Range = Heading_East;
    }

    return Compass_Range;
}

```

```

    }
    */
/*****
*****/

private int SensorEncodeCamera(int CameraReading)
{ // This function takes in the reading from a camera and returns an encoded range value
from 0 to 3.
    // Camera_Close and Camera_Far are all defined in the globals.

    int Camera_Range;

    if (CameraReading <= Camera_Close)
    { // If the reading is less than Camera_Close then the reading is Close.
        Camera_Range = Dist_Close;
    }
    else if (CameraReading <= Camera_Far)
    { // Otherwise, if the reading is less than Camera_Far, then the reading is Medium.
        Camera_Range = Dist_Medium;
    }
    else
    { // Otherwise, if the reading is more than Camera_Far, then the reading is Far.
        Camera_Range = Dist_Far;
    }
    // NOTE: commented out for smaller Q-table size
//    else
//    { // Otherwise, the reading is Very Far.
//        Camera_Range = Dist_Very_Far;
//    }

    return Camera_Range;

}

/*****
*****/

private int SensorEncodeSonar(int SonarReading)
{ // This function takes in the reading from a sonar and returns an encoded range value
from 0 to 4.
    // Sonar_Close, Sonar_Far, and Sonar_Bump are all defined in the globals.

```

```

int Sonar_Range;

if (SonarReading <= Sonar_Bump)
{ // If the reading is less than Sonar_Bump then the reading is Bump.
  Sonar_Range = Dist_Bump;
}
else if (SonarReading <= Sonar_Close)
{ // Otherwise, if the reading is less than Sonar_Close, then the reading is Close.
  Sonar_Range = Dist_Close;
}
else if (SonarReading <= Sonar_Far)
{ // Otherwise, if the reading is less than Sonar_Far, then the reading is Medium.
  Sonar_Range = Dist_Medium;
}
else
{ // Otherwise, if the reading is more than Sonar_Far, then the reading is Far.
  Sonar_Range = Dist_Far;
}
// NOTE: comment out the following for smaller Q-table size
// else
// { // Otherwise, the reading is Very Far.
//   Sonar_Range = Dist_Very_Far;
// }

return Sonar_Range;

}

/*****
*****/

private void RobotAtEndOfWorld()
{ // This function returns true if the robot is located beyond the end of the world.
  // This function is for simulation only, since it is coordinate-based. It will have to be
  changed for learning on the actual robot.
  // IMPORTANT NOTE: This function currently will ONLY work right in a long straight
  hallway situation.
  // If any other shape of hallway is required, this function MUST be rewritten.

  // Summary for this function.
  // The hallway is a rectangle, making the ends of the hallway parallel.
  // If the robot is located in between these parallel lines then it is not at the End of the
  World.
  // Otherwise, it is at the End of the World.

```

```

bool PastEndOfWorld;

PastEndOfWorld = false;
// BOOGA

/* SMALL

// SlopeA, SlopeB, bA, and bB are used for the equations of the parallel lines marking the
ends of the hallway.
float SlopeA;
float SlopeB;
float bA;
float bB;

// HallEndA1 and HallEndA2 are two points at one end of the hallway.
// HallEndB1 and HallEndB2 are two points at the other end.
// NOTE: HallEndA is the 'low' line and B is the 'high' line.
// Initialize them to (0,0).
float[] HallEndA1 = new float[] { 0, 0 };
float[] HallEndA2 = new float[] { 0, 0 };
float[] HallEndB1 = new float[] { 0, 0 };
float[] HallEndB2 = new float[] { 0, 0 };

// HigherThanA and LowerThanB are used to check where the location is relative to lines
A and B.
// If both are true then the robot is NOT past the End of the World.
// If EITHER is false then the robot IS past the End of the World.
// Set both to true by default.
bool HigherThanA = true;
bool LowerThanB = true;

// Points were found manually.
HallEndA1[x] = SegmentEndpoint[2, x];
HallEndA1[y] = SegmentEndpoint[2, y];
HallEndA2[x] = SegmentEndpoint[9, x];
HallEndA2[y] = SegmentEndpoint[9, y];
HallEndB1[x] = SegmentEndpoint[3, x];
HallEndB1[y] = SegmentEndpoint[3, y];
HallEndB2[x] = SegmentEndpoint[8, x];
HallEndB2[y] = SegmentEndpoint[8, y];

// Slope is the slope between the two A endpoints or the two B endpoints.
// Since these two lines are parallel then it doesn't matter which is picked.
// Do an error check for x-coordinates being the same just in case.
if (HallEndA1[x] == HallEndA2[x])
{

```

```

    SlopeA = VeryHighSlope;
}
else
{
    SlopeA = (HallEndA1[y] - HallEndA2[y]) / (HallEndA1[x] - HallEndA2[x]);
}

// Because lines A and B are parallel, they have the same slope.
SlopeB = SlopeA;

// bA is the intercept for the line passing through the A endpoints.
bA = HallEndA1[y] - SlopeA * HallEndA1[x];
// bB is the intercept for the line passing through the B endpoints.
bB = HallEndB1[y] - SlopeB * HallEndB1[x];

// HallEndA makes the 'low' line and HallEndB makes the 'high' line.
// The robot is between the lines if its location is higher than A and lower than B.
if ((RobotLocation[y] > HallEndA1[y]) && (RobotLocation[y] > HallEndA2[y]))
{ // If Loc[y] > A1[y] and > A2[y] then it is > A.
    HigherThanA = true;
}
else if ((RobotLocation[y] < HallEndA2[y]) && (RobotLocation[y] < HallEndA2[y]))
{ // If Loc[y] < A1[y] and < A2[y] then it is < A.
    HigherThanA = false;
}
else
{ // Even though the lines ARE NOT horizontal, if the location is between A1 and A2
vertically, let it count as not being in the End of the World.
    // This gives it a small amount of leeway.
    HigherThanA = true;
}
if ((RobotLocation[y] > HallEndB1[y]) && (RobotLocation[y] > HallEndB2[y]))
{ // If Loc[y] > B1[y] and > B2[y] then it is > B.
    LowerThanB = false;
}
else if ((RobotLocation[y] < HallEndB1[y]) && (RobotLocation[y] < HallEndB2[y]))
{ // If Loc[y] < B1[y] and < B2[y] then it is < B.
    LowerThanB = true;
}
else
{ // Give the same leeway as with A.
    LowerThanB = true;
}
}

```

```

        // If the robot is both HigherThanA AND LowerThanB then it is NOT past the End of the
World.
        // Otherwise, if either HigherThanA OR LowerThanB are false, it IS past the End of the
World.
        if (HigherThanA && LowerThanB)
        {
            PastEndOfWorld = false;
        }
        else
        {
            PastEndOfWorld = true;
        }

// ENDSMALL */

        AtEndOfWorld = PastEndOfWorld;

        return;
    }

/*****
*****/

    // QFocus - added GoalFoc0 to input list.
    private Int64 GenerateQKeyword(int GoalCam0, int GoalCam1, int GoalCam2, int
EoWCam0, int EoWCam1, int EoWCam2, int Son0, int Son12, int Son3, int Son4, int Son56, int
Son7, int GoalFoc0, int GoalFoc1, int GoalFoc2, int Action)
    {
        // This function takes the sensor and action values and generates a keyword for the
QTable Dictionary.

        Int64 Keyword;

        // Convert string of matrix coordinates into binary.
        // If the order of the table is as follows:
        // GoalCam0,GoalCam1,GoalCam2,EoWCam0,EoWCam1,EoWCam2,Sonar0,Sonar1-
2,Sonar3,Sonar4,Sonar5-6,Sonar7,Action
        // Then the number of possible readings are:
        // 3 3 3 3 3 4 4 4 4 4 5
        // And the number of bits necessary are:

```

```

// 2 2 2 2 2 2 2 2 2 2 2 3
// Then to make the keyword, multiply each coordinate by a power of 2 and add them
together to get an integer. This integer in binary would list the coordinates in the Q
matrix.
// So the keyword would be: GoalCam0*2^25 + GoalCam1*2^23 + GoalCam2*2^21 +
EoWCam0*2^19 + EoWCam1*2^17 + EoWCam2*2^15 + Sonar0*2^13 + Sonar1*2^11 +
Sonar3*2^9 + Sonar4*2^7 + Sonar5*2^5 + Sonar7*2^3 + Action
// The maximum value of this is 89,489,404, well within the maximum limit of an int.

// QFocus
// Change all the above statements to account for GoalFoc0.
// GoalFoc0 has 4 possible values, so it needs 2 bits.
// To avoid changing the entire line of code, just add it first. So it will be multiplied by
2^27.
// Do the same for GoalFoc1 and 2. They get 2^29 and 2^31.
Keyword = (int64)((GoalFoc2 * Math.Pow(2, 31)) + (GoalFoc1 * Math.Pow(2, 29)) +
(GoalFoc0 * Math.Pow(2, 27)) + (GoalCam0 * Math.Pow(2, 25)) + (GoalCam1 * Math.Pow(2,
23)) + (GoalCam2 * Math.Pow(2, 21)) + (EoWCam0 * Math.Pow(2, 19)) + (EoWCam1 *
Math.Pow(2, 17)) + (EoWCam2 * Math.Pow(2, 15)) + (Son0 * Math.Pow(2, 13)) + (Son12 *
Math.Pow(2, 11)) + (Son3 * Math.Pow(2, 9)) + (Son4 * Math.Pow(2, 7)) + (Son56 *
Math.Pow(2, 5)) + (Son7 * Math.Pow(2, 3)) + Action);
// Now comment out the old one below:
// Keyword = (int)((GoalCam0 * Math.Pow(2, 25)) + (GoalCam1 * Math.Pow(2,
23)) + (GoalCam2 * Math.Pow(2, 21)) + (EoWCam0 * Math.Pow(2, 19)) + (EoWCam1 *
Math.Pow(2, 17)) + (EoWCam2 * Math.Pow(2, 15)) + (Son0 * Math.Pow(2, 13)) + (Son12 *
Math.Pow(2, 11)) + (Son3 * Math.Pow(2, 9)) + (Son4 * Math.Pow(2, 7)) + (Son56 *
Math.Pow(2, 5)) + (Son7 * Math.Pow(2, 3)) + Action);
// end QFocus

TempKeyword = Keyword;

TempKeywordA = (int)((GoalFoc2 * Math.Pow(2, 16)) + (GoalFoc1 * Math.Pow(2, 14))
+ (GoalFoc0 * Math.Pow(2, 12)) + (GoalCam0 * Math.Pow(2, 10)) + (GoalCam1 *
Math.Pow(2, 8)) + (GoalCam2 * Math.Pow(2, 6)) + (EoWCam0 * Math.Pow(2, 4)) +
(EoWCam1 * Math.Pow(2, 2)) + (EoWCam2));
TempKeywordB = (int)((Son0 * Math.Pow(2, 13)) + (Son12 * Math.Pow(2, 11)) + (Son3
* Math.Pow(2, 9)) + (Son4 * Math.Pow(2, 7)) + (Son56 * Math.Pow(2, 5)) + (Son7 *
Math.Pow(2, 3)) + Action);

return Keyword;

}

```

```

/*****
*****/

```

```

private void PerformAction(int Actionchoice)
{ // This function takes Actionchoice and changes RobotLocation and RobotOrientation
according to the action

    if (Actionchoice == Action_Stop)
    { // If the action is Stop, then do not change anything and just end the function.
        return;
    }
    else if (Actionchoice == Action_Rotate_Left)
    { // If the action is Rotate Left, add RotateActionArc to RobotOrientation.
        RobotOrientation = RobotOrientation + RotateActionArc;
        if (RobotOrientation > twopi)
        { // If this makes the orientation greater than 2*Pi, subtract 2*Pi.
            RobotOrientation = RobotOrientation - twopi;
        }
        return;
    }
    else if (Actionchoice == Action_Rotate_Right)
    { // If the action is Rotate Right, subtract RotateActionArc from RobotOrientation.
        RobotOrientation = RobotOrientation - RotateActionArc;
        if (RobotOrientation < 0)
        { // If this makes the orientation less than 0, add 2*Pi.
            RobotOrientation = RobotOrientation + twopi;
        }
        return;
    }
    else if (Actionchoice == Action_Forward)
    { // If the action is Forward, move the robot MoveActionDistance along the
RobotOrientation direction.
        RobotLocation[x] = RobotLocation[x] + MoveActionDistance *
(float)Math.Cos(RobotOrientation);
        RobotLocation[y] = RobotLocation[y] + MoveActionDistance *
(float)Math.Sin(RobotOrientation);
        return;
    }
    else if (Actionchoice == Action_Reverse)
    { // If the action is Reverse, move the robot MoveActionDistance along the direction
opposite RobotOrientation.
        RobotLocation[x] = RobotLocation[x] + MoveActionDistance *
(float)Math.Cos(RobotOrientation + Math.PI);
        RobotLocation[y] = RobotLocation[y] + MoveActionDistance *
(float)Math.Sin(RobotOrientation + Math.PI);

        // Comment out the testing. This is for reverse being double the distance of forward.
/*
        // NOTE: For testing.

```

```

        // For a Reverse action, move the robot back twice.
        // In order to prevent running through the walls, call RobotContacting after the first
one.
        // It is already called after the entire PerformAction function so only needs to be done
here once.
        RobotContacting();
        RobotLocation[x] = RobotLocation[x] + MoveActionDistance *
(float)Math.Cos(RobotOrientation + Math.PI);
        RobotLocation[y] = RobotLocation[y] + MoveActionDistance *
(float)Math.Sin(RobotOrientation + Math.PI);
        */

        return;
    }
    else if (Actionchoice == Action_Big_Rotate_Left)
    { // If the action is Big Rotate Left, add BigRotateActionArc to RobotOrientation.
        RobotOrientation = RobotOrientation + BigRotateActionArc;
        if (RobotOrientation > twopi)
        { // If this makes the orientation greater than 2*Pi, subtract 2*Pi.
            RobotOrientation = RobotOrientation - twopi;
        }
    }
    else if (Actionchoice == Action_Big_Rotate_Right)
    { // If the action is Big Rotate Right, subtract BigRotateActionArc from
RobotOrientation.
        RobotOrientation = RobotOrientation - BigRotateActionArc;
        if (RobotOrientation < 0)
        { // If this makes the orientation less than 0, add 2*Pi.
            RobotOrientation = RobotOrientation + twopi;
        }
    }
    else
    { // This should never be reached but is here as a safety catch.
        // Just end the function.
        return;
    }
}

```

```

/*****
*****/

```

```

private void RobotTimer_Tick(object sender, EventArgs e)
{ // This is the function executed every time the timer ticks.
    // First, read the sensor values.
    // Make those sensor values into the State array.

```

```

// ... do more stuff ...
// ... use RobotContacting() in there somewhere ...

bool RobotTouching;

// GreedyAction and GreedyQ are used for the greedy algorithm in action selection.
int GreedyAction;
double GreedyQ;

// NextQMax is used for part of the Q-learning formula (max over next-actions Q(s',a' )
double NextQMax;

// CurrentQ is the Q value of the current state and current action.
double CurrentQ;

// NewQ is the new value for the current state and current action.
double NewQ;

// QKeyword is the keyword for the QTable dictionary.
// Set it to -1 initially.
Int64 QKeyword0 = -1;
Int64 QKeywordi = -1;
Int64 QKeywordAction = -1;

// QTemp is used as a temporary storage for a Q value.
double QTemp;

// NOTE: comment this out for smaller Q-table
// int CompassReading;

int[] SonarReadings = new int[NumSonars];
int[] CameraGoalReadings = new int[NumCameras];
int[] CameraEndOfWorldReadings = new int[NumCameras];

double randomrescue;
double myopicrandom;

// The following is to set the manual choice.
if (Manual_Forward_Toggle)
{
    Manual_Choice = Manual_Forward;
}

```

```

else if (Manual_Left_Toggle)
{
    Manual_Choice = Manual_Left;
}
else if (Manual_Right_Toggle)
{
    Manual_Choice = Manual_Right;
}
else if (Manual_Stop_Toggle)
{
    Manual_Choice = Manual_Stop;
}
else if (Manual_Reverse_Toggle)
{
    Manual_Choice = Manual_Reverse;
}
else if (Manual_Big_Left_Toggle)
{
    Manual_Choice = Manual_Big_Left;
}
else if (Manual_Big_Right_Toggle)
{
    Manual_Choice = Manual_Big_Right;
}
else
{
    Manual_Choice = Manual_None;
}

// Check for manual timer mode.
if (Manual_Mode_Toggle && !Manual_Timer_Step)
{ // If manual mode is on and the Step Timer button was not pushed, exit the timer loop
(i.e., do nothing and proceed to the next tick)
    ArenaMap.Invalidate();
    return;
}
else
{ // Otherwise, set Manual_Timer_Step false.
// If Manual_Mode_Toggle was true, this advances the timer exactly one step.
Otherwise, this does nothing.
    Manual_Timer_Step = false;
// Now clear the toggles.
    Manual_Forward_Toggle = false;
    Manual_Left_Toggle = false;
    Manual_Right_Toggle = false;
    Manual_Stop_Toggle = false;
}

```

```

Manual_Reverse_Toggle = false;
Manual_Big_Left_Toggle = false;
Manual_Big_Right_Toggle = false;
}

// Q-learning formula:
// Choose an action.
// 1-epsilon to choose the current best.
// epsilon to make a random choice.
// Find reward for choice.
// Read the NextState.
// Q formula.
// State <- NextState.

// RAR denotes things that use Table_Values for the Q Display.

// Q-learning formula: Choose an action.

// random action selection
// epsilon of the time, robot will select a random action
// 1-epsilon of the time, robot will select the current best option
Random randomseeder = new Random();
double epsilonrandom;
double actionrandom;

// Generate a random number between 0 and 1.
epsilonrandom = randomseeder.NextDouble();

if (epsilonrandom > (1 - epsilon))
{ // If the random number is in the top epsilon of the range, select an action at random.
  // Generate a random number between 0 and NumActions.
  actionrandom = NumActions * randomseeder.NextDouble();
  // If actionrandom is between 0 and 1, do action Forward.
  // If actionrandom is between 1 and 2, do action Reverse.
  // If actionrandom is between 2 and 3, do action Rotate Left.
  // If actionrandom is between 3 and 4, do action Rotate Right.
  // If actionrandom is between 4 and 5, do action Stop.
  // The actual order of these doesn't matter, as there should be an equal random chance
of each.
  // NOTE: If more actions are added, they must be added here as well!
  if (actionrandom <= 1)
  {
    Action = Action_Forward;
    // RAR
    Table_Values[TableTestRandom] = TableRandomForward;
  }
}

```

```

else if (actionrandom <= 2)
{
    Action = Action_Reverse;
    // RAR
    Table_Values[TableTestRandom] = TableRandomReverse;
}
else if (actionrandom <= 3)
{
    Action = Action_Rotate_Left;
    // RAR
    Table_Values[TableTestRandom] = TableRandomLeft;
}
else if (actionrandom <= 4)
{
    Action = Action_Rotate_Right;
    // RAR
    Table_Values[TableTestRandom] = TableRandomRight;
}
else if (actionrandom <= 5)
{
    Action = Action_Stop;
    // RAR
    Table_Values[TableTestRandom] = TableRandomStop;
}
else if (actionrandom <= 6)
{
    Action = Action_Big_Rotate_Left;
    Table_Values[TableTestRandom] = TableRandomBigLeft;
}
else
{
    Action = Action_Big_Rotate_Right;
    Table_Values[TableTestRandom] = TableRandomBigRight;
}
}

else
{ // This is the greedy algorithm
  // Go through all possible action choices and select the one with the highest value.

  // RAR
  Table_Values[TableTestRandom] = TableRandomNone;

  // First, set GreedyQ to the value of action choice 0 for the current state and
  GreedyAction to 0.

```

```

// To do this, find the QKeyword for this action.
QKeyword0 = GenerateQKeyword(State[State_Goal0], State[State_Goal1],
State[State_Goal2], State[State_EndOfWorld0], State[State_EndOfWorld1],
State[State_EndOfWorld2], State[State_Sonar0], State[State_Sonar1_2], State[State_Sonar3],
State[State_Sonar4], State[State_Sonar5_6], State[State_Sonar7], State[State_Focus0],
State[State_Focus1], State[State_Focus2], 0);
// Now GreedyQ is the entry of QTable corresponding to QKeyword.
// *** insert code so that whenever QTable[keyword] is looked up, first check that the
keyword exists; otherwise replace with a generic value
// If there is no entry there for that keyword, set GreedyQ to be GenericQValue.

// Q2
// The if statement is new for Q2. Contents of else were there already.
// They are marked with comments.
/*
    if (CanSeeGoal)
    {
        try
        {
            GreedyQ = QTable2[QKeyword0];
        }
        catch (KeyNotFoundException)
        {
            GreedyQ = GenericQValue;
        }
    }
else
{ // Contents of the else were there before Q2 changes */
    try
    {
        GreedyQ = QTable[QKeyword0];
    }
    catch (KeyNotFoundException)
    {
        GreedyQ = GenericQValue;
    }
}
/*
// end Q2
*/

GreedyAction = 0;

GlobalKeyword = TempKeyword;
GlobalKeywordA = TempKeywordA;
GlobalKeywordB = TempKeywordB;

// RAR

```

```

// Table_Values[TableTestForward] = GreedyQ;

// Now go through all the other action choices in a loop.
for (int i = 1; i < NumActions; i++)
{ // This loop goes through all the actions beginning with 1 (action 0 was already
checked).
    QKeywordi = GenerateQKeyword(State[State_Goal0], State[State_Goal1],
State[State_Goal2], State[State_EndOfWorld0], State[State_EndOfWorld1],
State[State_EndOfWorld2], State[State_Sonar0], State[State_Sonar1_2], State[State_Sonar3],
State[State_Sonar4], State[State_Sonar5_6], State[State_Sonar7], State[State_Focus0],
State[State_Focus1], State[State_Focus2], i);
    // *** insert code so that whenever QTable[keyword] is looked up, first check that
the keyword exists; otherwise replace with a generic value
    // QTemp becomes the value in the dictionary at QKeywordi. If there is nothing at
that keyword, set QTemp to QGenericValue.

    // Q2
/*     if (CanSeeGoal)
    {
        try
        {
            QTemp = QTable2[QKeywordi];
        }
        catch (KeyNotFoundException)
        {
            QTemp = GenericQValue;
        }
    }
else
{ // contents of the else were there before Q2 */
    try
    {
        QTemp = QTable[QKeywordi];
    }
    catch (KeyNotFoundException)
    {
        QTemp = GenericQValue;
    }
}
/*     }
// end Q2
*/

// RAR
// Table_Values[i] = QTemp;

if (GreedyQ < QTemp)

```

```

        { // If the GreedyQ is less than the reward from the current state taking action i,
then set GreedyQ to that reward and GreedyAction to that Action choice. Otherwise, do nothing.
        GreedyQ = QTemp;
        GreedyAction = i;
        GlobalKeyword = TempKeyword;
        GlobalKeywordA = TempKeywordA;
        GlobalKeywordB = TempKeywordB;
        }
    }
    // Set Action to whatever the GreedyAction choice is.
    Action = GreedyAction;
}

// RAR
// TableChoice = Action;

/* // NOTE
// Comment this section; rewrite using new QTable Dictionary
// replace with the above code segment
else
{ // This is the greedy algorithm.
// Go through all possible action choices and select the one with the highest value.
// First, set GreedyQ to the value of action choice 0 for the current state and
GreedyAction to 0.
    GreedyQ = QTableOld[State[State_Goal0], State[State_Goal1], State[State_Goal2],
State[State_EndOfWorld0], State[State_EndOfWorld1], State[State_EndOfWorld2],
State[State_Sonar0], State[State_Sonar1_2], State[State_Sonar3], State[State_Sonar4],
State[State_Sonar5_6], State[State_Sonar7], 0];
    GreedyAction = 0;
    // Now go through all the other action choices in a loop.
    for (int i = 1; i < NumActions; i++)
    { // This loop goes through all the actions beginning with 1 (action 0 was already
checked).
        if (GreedyQ < QTableOld[State[State_Goal0], State[State_Goal1],
State[State_Goal2], State[State_EndOfWorld0], State[State_EndOfWorld1],
State[State_EndOfWorld2], State[State_Sonar0], State[State_Sonar1_2], State[State_Sonar3],
State[State_Sonar4], State[State_Sonar5_6], State[State_Sonar7], i])
        { // If the GreedyQ is less than the reward from the current state taking action i,
then set GreedyQ to that reward and GreedyAction to that Action choice. Otherwise, do nothing.
            GreedyQ = QTableOld[State[State_Goal0], State[State_Goal1],
State[State_Goal2], State[State_EndOfWorld0], State[State_EndOfWorld1],
State[State_EndOfWorld2], State[State_Sonar0], State[State_Sonar1_2], State[State_Sonar3],
State[State_Sonar4], State[State_Sonar5_6], State[State_Sonar7], i];
            GreedyAction = i;
        }
    }
}

```

```

    }
    // Set Action to whatever the GreedyAction choice is.
    Action = GreedyAction;
}
*/

// OldLocation and OldOrientation are used to store the current position and orientation
of the robot.
OldLocation = RobotLocation;
OldOrientation = RobotOrientation;

if (Manual_Mode_Toggle && !(Manual_Choice == Manual_None))
{ // If Manual_Mode_Toggle is on and Manual_Choice is not None, change the action
based on the manual choice.
    Action = Manual_Choice;
}
PerformAction(Action);

// Comment out the following since it doesn't seem to work right:
/*
    if (!Manual_Mode_Toggle || (Manual_Choice == Manual_None))
    { // If manual mode is off or no action has been chosen, perform the chosen action.
        PerformAction(Action);
    }
    else if (Manual_Choice == Manual_Forward)
    { // If the manual mode is active, perform the manual action.
        PerformAction(Action_Forward);
    }
    else if (Manual_Choice == Manual_Left)
    {
        PerformAction(Action_Rotate_Left);
    }
    else if (Manual_Choice == Manual_Right)
    {
        PerformAction(Action_Rotate_Right);
    }
    else if (Manual_Choice == Manual_Stop)
    {
        PerformAction(Action_Stop);
    }
    else if (Manual_Choice == Manual_Reverse)
    {
        PerformAction(Action_Reverse);
    }
*/
*/

```

```

// Comment out the following for adding manual mode:
// Perform the chosen action.
//   PerformAction(Action);

// *** begin old robot movement
/*

// NOTE: replace the following with the actual actions
if (movingRight)
{
    RobotLocation[x] = RobotLocation[x] + 5F;
}
else
{
    RobotLocation[x] = RobotLocation[x] - 5F;
}
if (movingDown)
{
    RobotLocation[y] = RobotLocation[y] + 5F;
}
else
{
    RobotLocation[y] = RobotLocation[y] - 5F;
}
if (!movingRight && !movingDown)
{
    RobotOrientation = 5 * (float)Math.PI / 4;
}
else if (movingRight && !movingDown)
{
    RobotOrientation = 7 * (float)Math.PI / 4;
}
else if (movingRight && movingDown)
{
    RobotOrientation = (float)Math.PI / 4;
}
else if (!movingRight && movingDown)
{
    RobotOrientation = 3 * (float)Math.PI / 4;
}
// NOTE: replace the following with the actual action choices
double randomnumber;
double randomnumber2;
randomnumber = randomseeder.NextDouble();

```

```

if (randomnumber > (1 - epsilon))
{
    randomnumber2 = 4 * randomseeder.NextDouble();
    if (randomnumber2 < 1)
    {
        movingRight = true;
        movingDown = true;
    }
    else if (randomnumber2 < 2)
    {
        movingRight = true;
        movingDown = false;
    }
    else if (randomnumber2 < 3)
    {
        movingRight = false;
        movingDown = true;
    }
    else
    {
        movingRight = false;
        movingDown = false;
    }
}
// This line is to force the robot into the 'upper' area of the display
// It is being used only for testing corner interaction. Remove it after.
if (RobotLocation[y] > 400) { movingDown = false; }
//if (RobotLocation[y] < 1700) { movingDown = true; }

// *** end old robot movement
*/

// 'Rescue' the robot if it intersected a wall.
// Do this before reading in any sensor values.
RobotTouching = RobotContacting();

// See if the robot is in an End of the World location.
// Normally, this would be grouped down with reading the sensors, but it needs to be
before the reward assignment for the End of World reward to be assigned.
//     RobotAtEndOfWorld();

// Q-learning formula: Find reward for choice.
// There are seven different rewards: Goal, End of World, Bump, Stop, Reverse, Turn,
and Forward.

```

```

// The following statements go through the states to check for the reward.

/* SMALL
if ((State[State_Goal0] == Dist_Close) && (Action == Action_Stop))
{ // If the action is Stop and the front camera reads Close distance to the goal, then the
robot is at the goal!
    Reward = Reward_Goal;
    AtGoal = true;
}
ENDSMALL */
// Instead, remove the STOP requirement.
// if (State[State_Goal0] == Dist_Close)

// NOTE
// REWARDMOVE
// Move the reward calculations down to after finding the next state.
// The reason for this is that the reward calculations rely on the state and the status bools:
GoalFrontCenter, GoalInFront, CanSeeGoal, AtEndOfWorld, BumpDetected.
// If the reward is calculated here, then the status bools are set for the location where the
robot just was, not where it is now.
// We want the reward to be calculated based on the state where it just was and the results
from what it did (action and status bools).
// So, this gets moved to after status bools are calculated, when NextState is read.
/*
if ((State[State_Goal0]==Dist_Close)&&!(CurrentSteps==0))
{
    // Q3
    if (GoalFrontCenter)
    {
        Reward = Reward3_Goal;
    }
    // Q2
    else if (CanSeeGoal)
    { // To get rid of Q3, also remove the else from the previous line.
        Reward = Reward2_Goal;
    }
    else
    { // contents of else are from before Q2
        Reward = Reward_Goal;
    }
    // end Q2
    // end Q3

    AtGoal = true;
}

```

```

}

else if (AtEndOfWorld)
{ // If the robot is in the area designated as the End of the World, then it gets the End of
the World reward.
  // Q3
  if (GoalFrontCenter)
  {
    Reward = Reward3_End_of_World;
  }
  // Q2
  else if (CanSeeGoal)
  { // To get rid of Q3, also remove the else from the previous line.
    Reward = Reward2_End_of_World;
  }
  else
  { // contents of else are from before Q2
    Reward = Reward_End_of_World;
  }
  // end Q2
  // end Q3
}
// GOAL PROXIMITY
//   else if ((State[State_Goal0] == Dist_Medium) && !BumpDetected)
//   { // If the robot sees the goal in front at Medium distance, give it the
Reward_Goal_Front reward.
//     Reward = Reward_Goal_Front;
//   }
//   else if (((State[State_Goal1] == Dist_Close) || (State[State_Goal2] == Dist_Close) ||
(State[State_Goal1] == Dist_Medium) || (State[State_Goal2] == Dist_Medium)) &&
!BumpDetected)
//   { // If the robot sees the goal on the side, give it the Reward_Goal_Side reward.
//     Reward = Reward_Goal_Side;
//   }
// END GOAL PROXIMITY

else if (BumpDetected)
{ // If the robot is in a bump position, then it gets the Bump reward.
  // Q3
  if (GoalFrontCenter)
  {
    Reward = Reward3_Bump;
  }
  // Q2
  else if (CanSeeGoal)
  { // To get rid of Q3, also remove the else from the previous line.

```

```

    Reward = Reward2_Bump;
}
else
{ // contents of else are from before Q2
    Reward = Reward_Bump;
}
// end Q2
// end Q3
}
else if (Action == Action_Stop)
{ // If the robot stops, it gets the Stop reward.
// Q3
if (GoalFrontCenter)
{
    Reward = Reward3_Stop;
}
// Q2
else if (CanSeeGoal)
{ // To get rid of Q3, also remove the else from the previous line.
    Reward = Reward2_Stop;
}
else
{ // contents of else are from before Q2
    Reward = Reward_Stop;
}
// end Q2
// end Q3
}
else if (CenterOfHall && (Action == Action_Forward))
{ // If the robot goes forward and is in the center of the hall, it gets the Center_Forward
reward.
// Q3
if (GoalFrontCenter)
{
    Reward = Reward3_Center_Forward;
}
// Q2
else if (CanSeeGoal)
{ // To get rid of Q3, also remove the else from the previous line.
    Reward = Reward2_Center_Forward;
}
else
{ // contents of else are from before Q2
    Reward = Reward_Center_Forward;
}
// end Q2

```

```

    // end Q3
}
else if (CenterOfHall && (Action == Action_Reverse))
{ // If the robot goes in reverse and is in the center of the hall, it gets the Center_Reverse
reward.
    // Q3
    if (GoalFrontCenter)
    {
        Reward = Reward3_Center_Reverse;
    }
    // Q2
    else if (CanSeeGoal)
    { // To get rid of Q3, also remove the else from the previous line.
        Reward = Reward2_Center_Reverse;
    }
    else
    { // contents of else are from before Q2
        Reward = Reward_Center_Reverse;
    }
    // end Q2
    // end Q3
}
else if (CenterOfHall && ((Action == Action_Rotate_Left) || (Action ==
Action_Rotate_Right)))
{ // If the robot rotates left or right and is in the center of the hall, it gets the
Center_Turn reward.
    // Q3
    if (GoalFrontCenter)
    {
        Reward = Reward3_Center_Turn;
    }
    // Q2
    else if (CanSeeGoal)
    { // To get rid of Q3, also remove the else from the previous line.
        Reward = Reward2_Center_Turn;
    }
    else
    { // contents of else are from before Q2
        Reward = Reward_Center_Turn;
    }
    // end Q2
    // end Q3
}
else if (Action == Action_Reverse)
{ // If the robot goes in reverse, it gets the Reverse reward.
    // Q3

```

```

if (GoalFrontCenter)
{
    Reward = Reward3_Center_Reverse;
}
// Q2
else if (CanSeeGoal)
{ // To get rid of Q3, also remove the else from the previous line.
    Reward = Reward2_Reverse;
}
else
{ // contents of else are from before Q2
    Reward = Reward_Reverse;
}
// end Q2
// end Q3
}
else if ((Action == Action_Rotate_Left) || (Action == Action_Rotate_Right))
{ // If the robot rotates left or right, it gets the Turn reward.
// Q3
if (GoalFrontCenter)
{
    Reward = Reward3_Turn;
}
// Q2
else if (CanSeeGoal)
{ // To get rid of Q3, also remove the else from the previous line.
    Reward = Reward2_Turn;
}
else
{ // contents of else are from before Q2
    Reward = Reward_Turn;
}
// end Q2
// end Q3
}
else if (Action == Action_Forward)
{ // If the robot moves forward, it gets the Forward reward.
// Q3
if (GoalFrontCenter)
{
    Reward = Reward3_Forward;
}
// Q2
else if (CanSeeGoal)
{ // To get rid of Q3, also remove the else from the previous line.
    Reward = Reward2_Forward;
}
}

```

```

    }
    else
    { // contents of else are from before Q2
      Reward = Reward_Forward;
    }
    // end Q2
    // end Q3
  }
  else if (CenterOfHall && ((Action == Action_Big_Rotate_Left) || (Action ==
Action_Big_Rotate_Right)))
  { // If the robot rotates left or right and is in the center of the hall, it gets the
Center_Turn reward.
    // Q3
    if (GoalFrontCenter)
    {
      Reward = Reward3_Center_Big_Turn;
    }
    // Q2
    else if (CanSeeGoal)
    { // To get rid of Q3, also remove the else from the previous line.
      Reward = Reward2_Center_Big_Turn;
    }
    else
    { // contents of else are from before Q2
      Reward = Reward_Center_Big_Turn;
    }
    // end Q2
    // end Q3
  }
  else if ((Action == Action_Big_Rotate_Left) || (Action == Action_Big_Rotate_Right))
  { // If the robot rotates left or right, it gets the Turn reward.
    // Q3
    if (GoalFrontCenter)
    {
      Reward = Reward3_Big_Turn;
    }
    // Q2
    else if (CanSeeGoal)
    { // To get rid of Q3, also remove the else from the previous line.
      Reward = Reward2_Big_Turn;
    }
    else
    { // contents of else are from before Q2
      Reward = Reward_Big_Turn;
    }
    // end Q2
  }

```

```

    // end Q3
}

else
{ // This should never happen, but is here just in case
  Reward = -1000;
}

// Update the CurrentReward.
CurrentReward = CurrentReward + Reward;
// Update the number of steps.
CurrentSteps++;

// Q2
if ((Reward == Reward_Bump) || (Reward == Reward2_Bump) || (Reward ==
Reward3_Bump))
{
  CurrentWallHits++;
}
// Comment out the following if statement for Q2
//   if (Reward == Reward_Bump)
//   { // If there was a bump, add to the number of bumps.
//     CurrentWallHits++;
//   }
// end Q2

// After rewards are assigned...
if (AtEndOfWorld || AtGoal)
{ // If the robot has reached the goal or passed the end of the world, reset the simulation.
  SimulationReset();
}

// end REWARDMOVE */

// Q-learning formula: Read Next State.
// Read sensor values and assign the state.
// Compass
// NOTE: comment out for smaller Q-table size
//   CompassReading = CalculateCompass();
//   State[State_Compass] = SensorEncodeCompass(CompassReading);

// DoubleQ

```

```

    // Set CanSeeGoal to false. If it can be seen, it will be set true inside of the
CalculateCameraToGoal functions.
    CanSeeGoal = false;
    // Q3
    // Set GoalFrontCenter false for the same reason.
    GoalFrontCenter = false;
    // Q2.5
    // Also set GoalInFront false.
    GoalInFront = false;

    // Cameras to goal and end of world
    for (int i = 0; i < NumCameras; i++)
    {
        CameraGoalReadings[i] = CalculateCameraToGoal(i);
        NextState[State_Goal0 + i] = SensorEncodeCamera(CameraGoalReadings[i]);

        // CameraFocus
        // This is just a test so far. It should only set it for the front camera when the clockless
timer is not active
        // If it's the front sensor and the distance is Close or Medium (i.e., not Dist_Far), set
FocusInCamera to TempFocusInCamera.
        //   if (i == 0)
        //   {
        /*   if (NextState[State_Goal0] != Dist_Far)
            {
                FocusInCamera[i] = TempFocusInCamera;
            }
            else
            {
                FocusInCamera[i] = CameraFocusNone;
            }
            // QFocus */
            NextState[State_Focus0 + i] = FocusInCamera[i];
        //   }
        // end CameraFocus */

        if (NextState[State_Goal0 + i] == Dist_Far)
        {
            NextState[State_Focus0 + i] = CameraFocusNone;
            //   FocusInCamera[i] = CameraFocusNone;
        }
        // end CameraFocus

        // DoubleQ

```

```

        if ((NextState[State_Goal0 + i] == Dist_Close) || (NextState[State_Goal0 + i] ==
Dist_Medium))
        { // If the camera reading just found (NextState[State_Goal0+i] is Dist_Close or
Dist_Medium, set CanSeeGoal to true.
            CanSeeGoal = true;
        }
        else if ((NextState[State_Goal0 + i] == Dist_Far) && (CameraGoalReadings[i] <
(ReallyLongDistance / 2)))
        { // If the camera reading is Dist_Far but the distance to the goal is less than
ReallyLongDistance/2, generate a random number.
            myopicrandom = randomseeder.NextDouble();
            if (myopicrandom < MyopicPercent)
            { // If that random number is within MyopicPercent, set CanSeeGoal to true.
                CanSeeGoal = true;
            }
        }
    }

    // Q3
    if (CanSeeGoal && (FocusInCamera[FocusCamFront] == CameraFocusCenter))
    { // If the goal can be seen and is in the Center focus of the front camera, set
GoalFrontCenter true.
        GoalFrontCenter = true;
    }
    // end Q3

    // Q2.5
    if (CanSeeGoal && ((FocusInCamera[FocusCamFront] == CameraFocusLeft) ||
(FocusInCamera[FocusCamFront] == CameraFocusRight)))
    { // If the goal can be seen and is in the front camera but not in the center, set
GoalInFront true.
        GoalInFront = true;
    }
    // end Q2.5

    CameraEndOfWorldReadings[i] = CalculateCameraToEndOfWorld(i);
    NextState[State_EndOfWorld0 + i] =
SensorEncodeCamera(CameraEndOfWorldReadings[i]);
}

// Sonars
// NOTE: comment out the following for smaller Q-table size
// for (int i = 0; i < NumSonars; i++)
// {
//     SonarReadings[i] = CalculateSonar(i);
//     State[State_Sonar0 + i] = SensorEncodeSonar(SonarReadings[i]);

```

```

// }

// Sonar
for (int i = 0; i < NumSonars; i++)
{
    SonarReadings[i] = CalculateSonar(i);
}
NextState[State_Sonar0] = SensorEncodeSonar(SonarReadings[0]);
NextState[State_Sonar1_2] = SensorEncodeSonar((SonarReadings[1] +
SonarReadings[2]) / 2);
NextState[State_Sonar3] = SensorEncodeSonar(SonarReadings[3]);
NextState[State_Sonar4] = SensorEncodeSonar(SonarReadings[4]);
NextState[State_Sonar5_6] = SensorEncodeSonar((SonarReadings[5] +
SonarReadings[6]) / 2);
NextState[State_Sonar7] = SensorEncodeSonar(SonarReadings[7]);
// Set CenterOfHall
if ((NextState[State_Sonar3] != Sonar_Bump) && (NextState[State_Sonar3] ==
NextState[State_Sonar7]))
{ // If the left and right sonar readings (3 and 7) are the same and aren't reading bumps,
then set CenterOfHall to true.
    CenterOfHall = true;
}
else
{ // otherwise set it false
    CenterOfHall = false;
}

// Calculated BumpDetected.
// BumpDetected is true if any of the sonars detect a bump.
if ((NextState[State_Sonar0] == Dist_Bump) || (NextState[State_Sonar1_2] ==
Dist_Bump) || (NextState[State_Sonar3] == Dist_Bump) || (NextState[State_Sonar4] ==
Dist_Bump) || (NextState[State_Sonar5_6] == Dist_Bump) || (NextState[State_Sonar7] ==
Dist_Bump))
{ // If any of the sonar sensors detects a bump, then BumpDetected is true.
    BumpDetected = true;
}
else
{ // Otherwise, BumpDetected is false.
    BumpDetected = false;
}

// See if the robot is in an End of the World location.
RobotAtEndOfWorld();

```

```

// REWARDMOVE
// Move the reward-calculating parts to here.
if ((State[State_Goal0] == Dist_Close) && !(CurrentSteps == 0))
{
    // Q3
    if (GoalFrontCenter)
    {
        Reward = Reward3_Goal;
    }
    // Q2.5
    else if (GoalInFront)
    {
        Reward = Reward25_Goal;
    }
    // Q2
    else if (CanSeeGoal)
    { // To get rid of Q3, also remove the else from the previous line.
        Reward = Reward2_Goal;
    }
    else
    { // contents of else are from before Q2
        Reward = Reward_Goal;
    }
    // end Q2
    // end Q2.5
    // end Q3

    AtGoal = true;
}

else if (AtEndOfWorld)
{ // If the robot is in the area designated as the End of the World, then it gets the End of
the World reward.
    // Q3
    if (GoalFrontCenter)
    {
        Reward = Reward3_End_of_World;
    }
    // Q2.5
    else if (GoalInFront)
    {
        Reward = Reward25_End_of_World;
    }
}

```

```

// Q2
else if (CanSeeGoal)
{ // To get rid of Q3, also remove the else from the previous line.
  Reward = Reward2_End_of_World;
}
else
{ // contents of else are from before Q2
  Reward = Reward_End_of_World;
}
// end Q2
// end Q2.5
// end Q3
}
// GOAL PROXIMITY
//     else if ((State[State_Goal0] == Dist_Medium) && !BumpDetected)
//     { // If the robot sees the goal in front at Medium distance, give it the
Reward_Goal_Front reward.
//     Reward = Reward_Goal_Front;
// }
// else if (((State[State_Goal1] == Dist_Close) || (State[State_Goal2] == Dist_Close) ||
(State[State_Goal1] == Dist_Medium) || (State[State_Goal2] == Dist_Medium)) &&
!BumpDetected)
//{ // If the robot sees the goal on the side, give it the Reward_Goal_Side reward.
// Reward = Reward_Goal_Side;
// }
// END GOAL PROXIMITY

else if (BumpDetected && !GoalFrontCenter) // Q3 - added !GoalFrontCenter so bump
doesn't matter if the goal is directly in front.
{ // If the robot is in a bump position, then it gets the Bump reward.
// Q3
if (GoalFrontCenter)
{
  Reward = Reward3_Bump;
}
// Q2.5
else if (GoalInFront)
{
  Reward = Reward25_Bump;
}
// Q2
else if (CanSeeGoal)
{ // To get rid of Q3, also remove the else from the previous line.
  Reward = Reward2_Bump;
}
else

```

```

    { // contents of else are from before Q2
      Reward = Reward_Bump;
    }
    // end Q2
    // end Q2.5
    // end Q3
  }
  else if (Action == Action_Stop)
  { // If the robot stops, it gets the Stop reward.
    // Q3
    if (GoalFrontCenter)
    {
      Reward = Reward3_Stop;
    }
    // Q2.5
    else if (GoalInFront)
    {
      Reward = Reward25_Stop;
    }
    // Q2
    else if (CanSeeGoal)
    { // To get rid of Q3, also remove the else from the previous line.
      Reward = Reward2_Stop;
    }
    else
    { // contents of else are from before Q2
      Reward = Reward_Stop;
    }
    // end Q2
    // end Q2.5
    // end Q3
  }
  else if (CenterOfHall && (Action == Action_Forward))
  { // If the robot goes forward and is in the center of the hall, it gets the Center_Forward
reward.
    // Q3
    if (GoalFrontCenter)
    {
      Reward = Reward3_Center_Forward;
    }
    // Q2.5
    else if (GoalInFront)
    {
      Reward = Reward25_Center_Forward;
    }
    // Q2

```

```

else if (CanSeeGoal)
{ // To get rid of Q3, also remove the else from the previous line.
  Reward = Reward2_Center_Forward;
}
else
{ // contents of else are from before Q2
  Reward = Reward_Center_Forward;
}
// end Q2
// end Q2.5
// end Q3
}
else if (CenterOfHall && (Action == Action_Reverse))
{ // If the robot goes in reverse and is in the center of the hall, it gets the Center_Reverse
reward.
  // Q3
  if (GoalFrontCenter)
  {
    Reward = Reward3_Center_Reverse;
  }
  // Q2.5
  else if (GoalInFront)
  {
    Reward = Reward25_Center_Reverse;
  }
  // Q2
  else if (CanSeeGoal)
  { // To get rid of Q3, also remove the else from the previous line.
    Reward = Reward2_Center_Reverse;
  }
  else
  { // contents of else are from before Q2
    Reward = Reward_Center_Reverse;
  }
  // end Q2
  // end Q2.5
  // end Q3
}
else if (CenterOfHall && ((Action == Action_Rotate_Left) || (Action ==
Action_Rotate_Right)))
{ // If the robot rotates left or right and is in the center of the hall, it gets the
Center_Turn reward.
  // Q3
  if (GoalFrontCenter)
  {
    Reward = Reward3_Center_Turn;
  }

```

```

}
// Q2.5
else if (GoalInFront)
{
    Reward = Reward25_Center_Turn;
}
// Q2
else if (CanSeeGoal)
{ // To get rid of Q3, also remove the else from the previous line.
    Reward = Reward2_Center_Turn;
}
else
{ // contents of else are from before Q2
    Reward = Reward_Center_Turn;
}
// end Q2
// end Q2.5
// end Q3
}
else if (Action == Action_Reverse)
{ // If the robot goes in reverse, it gets the Reverse reward.
// Q3
if (GoalFrontCenter)
{
    Reward = Reward3_Reverse;
}
// Q2.5
else if (GoalInFront)
{
    Reward = Reward25_Reverse;
}
// Q2
else if (CanSeeGoal)
{ // To get rid of Q3, also remove the else from the previous line.
    Reward = Reward2_Reverse;
}
else
{ // contents of else are from before Q2
    Reward = Reward_Reverse;
}
// end Q2
// end Q2.5
// end Q3
}
else if ((Action == Action_Rotate_Left) || (Action == Action_Rotate_Right))
{ // If the robot rotates left or right, it gets the Turn reward.

```

```

// Q3
if (GoalFrontCenter)
{
    Reward = Reward3_Turn;
}
// Q2.5
else if (GoalInFront)
{
    Reward = Reward25_Turn;
}
// Q2
else if (CanSeeGoal)
{ // To get rid of Q3, also remove the else from the previous line.
    Reward = Reward2_Turn;
}
else
{ // contents of else are from before Q2
    Reward = Reward_Turn;
}
// end Q2
// end Q2.5
// end Q3
}
else if (Action == Action_Forward)
{ // If the robot moves forward, it gets the Forward reward.
// Q3
if (GoalFrontCenter)
{
    Reward = Reward3_Forward;
}
// Q2.5
else if (GoalInFront)
{
    Reward = Reward25_Forward;
}
// Q2
else if (CanSeeGoal)
{ // To get rid of Q3, also remove the else from the previous line.
    Reward = Reward2_Forward;
}
else
{ // contents of else are from before Q2
    Reward = Reward_Forward;
}
// end Q2
// end Q2.5

```

```

        // end Q3
    }
    else if (CenterOfHall && ((Action == Action_Big_Rotate_Left) || (Action ==
Action_Big_Rotate_Right)))
    { // If the robot rotates left or right and is in the center of the hall, it gets the
Center_Turn reward.
        // Q3
        if (GoalFrontCenter)
        {
            Reward = Reward3_Center_Big_Turn;
        }
        // Q2.5
        else if (GoalInFront)
        {
            Reward = Reward25_Center_Big_Turn;
        }
        // Q2
        else if (CanSeeGoal)
        { // To get rid of Q3, also remove the else from the previous line.
            Reward = Reward2_Center_Big_Turn;
        }
        else
        { // contents of else are from before Q2
            Reward = Reward_Center_Big_Turn;
        }
        // end Q2
        // end Q2.5
        // end Q3
    }
    else if ((Action == Action_Big_Rotate_Left) || (Action == Action_Big_Rotate_Right))
    { // If the robot rotates left or right, it gets the Turn reward.
        // Q3
        if (GoalFrontCenter)
        {
            Reward = Reward3_Big_Turn;
        }
        // Q2.5
        else if (GoalInFront)
        {
            Reward=Reward25_Big_Turn;
        }
        // Q2
        else if (CanSeeGoal)
        { // To get rid of Q3, also remove the else from the previous line.
            Reward = Reward2_Big_Turn;
        }
    }

```

```

else
{ // contents of else are from before Q2
  Reward = Reward_Big_Turn;
}
// end Q2
// end Q2.5
// end Q3
}

else
{ // This should never happen, but is here just in case
  Reward = -1000;
}

// Update the CurrentReward.
CurrentReward = CurrentReward + Reward;
// Update the number of steps.
CurrentSteps++;

// Q2
if ((Reward == Reward_Bump) || (Reward == Reward2_Bump) || (Reward ==
Reward3_Bump))
{
  CurrentWallHits++;
}
// Comment out the following if statement for Q2
//   if (Reward == Reward_Bump)
//   { // If there was a bump, add to the number of bumps.
//     CurrentWallHits++;
//   }
// end Q2

// After rewards are assigned...
if (AtEndOfWorld || AtGoal)
{ // If the robot has reached the goal or passed the end of the world, reset the simulation.
  SimulationReset();
}

// end REWARDMOVE

// Q-learning formula: Q-formula.

```

```

// trytonext
//   for (int i = 0; i < NumStates; i++)
//   {
//       State[i] = NextState[i];
//   }

//statetonextstate
// First find NextQMax.
// The actual next action chosen is not known yet! That's okay. All we need to know is
the value of whatever next action choice gives the highest reward.
// Start by setting NextQMax to the reward from the next state taking action 0.
QKeyword0 = GenerateQKeyword(State[State_Goal0], State[State_Goal1],
State[State_Goal2], State[State_EndOfWorld0], State[State_EndOfWorld1],
State[State_EndOfWorld2], State[State_Sonar0], State[State_Sonar1_2], State[State_Sonar3],
State[State_Sonar4], State[State_Sonar5_6], State[State_Sonar7], State[State_Focus0],
State[State_Focus1], State[State_Focus2], 0);
// Whenever QTable[keyword] is looked up, first check that the keyword exists;
otherwise replace with a generic value

// Q2
/*   if (CanSeeGoal)
{
    try
    {
        NextQMax = QTable2[QKeyword0];
    }
    catch (KeyNotFoundException)
    {
        NextQMax = GenericQValue;
    }
}
else
{ // contents of else were there previous to Q2 */
    try
    {
        NextQMax = QTable[QKeyword0];
    }
    catch (KeyNotFoundException)
    {
        NextQMax = GenericQValue;
    }
}

```

```

    }
/*    }
    // end Q2
*/

// statetonextstate

// Now go through all the other actions in a loop.
for (int i = 1; i < NumActions; i++)
{ // This loop goes through all actions beginning with 1 (action 0 was already checked).
    QKeywordi = GenerateQKeyword(State[State_Goal0], State[State_Goal1],
State[State_Goal2], State[State_EndOfWorld0], State[State_EndOfWorld1],
State[State_EndOfWorld2], State[State_Sonar0], State[State_Sonar1_2], State[State_Sonar3],
State[State_Sonar4], State[State_Sonar5_6], State[State_Sonar7], State[State_Focus0],
State[State_Focus1], State[State_Focus2], i);
    // Whenever QTable[keyword] is looked up, first check that the keyword exists;
otherwise replace with a generic value

    // Q2
/*    if (CanSeeGoal)
    {
        try
        {
            QTemp = QTable2[QKeywordi];
        }
        catch (KeyNotFoundException)
        {
            QTemp = GenericQValue;
        }
    }
else
{ // contents of else were there before Q2 */
    try
    {
        QTemp = QTable[QKeywordi];
    }
    catch (KeyNotFoundException)
    {
        QTemp = GenericQValue;
    }
/*    }
    // end Q2
*/

    if (NextQMax < QTemp)

```

```
    { // If the NextQMax is less than the reward from the next state taking action i, then
      set NextQMax to that reward.  Otherwise, do nothing.
```

```
        NextQMax = QTemp;
      }
    }
```

```
/*
// Q-learning formula: Q-formula.
```

```
// First find NextQMax.
// The actual next action chosen is not known yet! That's okay. All we need to know is
the value of whatever next action choice gives the highest reward.
```

```
// Start by setting NextQMax to the reward from the next state taking action 0.
NextQMax = QTableOld[NextState[State_Goal0], NextState[State_Goal1],
NextState[State_Goal2], NextState[State_EndOfWorld0], NextState[State_EndOfWorld1],
NextState[State_EndOfWorld2], NextState[State_Sonar0], NextState[State_Sonar1_2],
NextState[State_Sonar3], NextState[State_Sonar4], NextState[State_Sonar5_6],
NextState[State_Sonar7], 0];
```

```
// Now go through all the other actions in a loop.
for (int i = 1; i < NumActions; i++)
{ // This loop goes through all the actions beginning with 1 (action 0 was already
checked).
```

```
    if (NextQMax < QTableOld[NextState[State_Goal0], NextState[State_Goal1],
NextState[State_Goal2], NextState[State_EndOfWorld0], NextState[State_EndOfWorld1],
NextState[State_EndOfWorld2], NextState[State_Sonar0], NextState[State_Sonar1_2],
NextState[State_Sonar3], NextState[State_Sonar4], NextState[State_Sonar5_6],
NextState[State_Sonar7], i])
```

```
    { // If the NextQMax is less than the reward from the next state taking action i, then
      set NextQMax to that reward.  Otherwise, do nothing.
```

```
        NextQMax = QTableOld[NextState[State_Goal0], NextState[State_Goal1],
NextState[State_Goal2], NextState[State_EndOfWorld0], NextState[State_EndOfWorld1],
NextState[State_EndOfWorld2], NextState[State_Sonar0], NextState[State_Sonar1_2],
NextState[State_Sonar3], NextState[State_Sonar4], NextState[State_Sonar5_6],
NextState[State_Sonar7], i];
```

```
    }
  }
```

```
*/
```

```
// statetonextstate
```

```
// Now get CurrentQ
QKeywordAction = GenerateQKeyword(State[State_Goal0], State[State_Goal1],
State[State_Goal2], State[State_EndOfWorld0], State[State_EndOfWorld1],
State[State_EndOfWorld2], State[State_Sonar0], State[State_Sonar1_2], State[State_Sonar3],
```

```
State[State_Sonar4], State[State_Sonar5_6], State[State_Sonar7], State[State_Focus0],  
State[State_Focus1], State[State_Focus2], Action);
```

```
    // Whenever QTable[keyword] is looked up, first check that the keyword exists;  
    otherwise replace with a generic value
```

```
    // Q2  
/*    if (CanSeeGoal)  
    {  
        try  
        {  
            CurrentQ = QTable2[QKeywordAction];  
        }  
        catch (KeyNotFoundException)  
        {  
            CurrentQ = GenericQValue;  
        }  
    }  
    else  
    { // contents of else were there before Q2 */  
        try  
        {  
            CurrentQ = QTable[QKeywordAction];  
        }  
        catch (KeyNotFoundException)  
        {  
            CurrentQ = GenericQValue;  
        }  
    }  
/*  
    // end Q2  
*/
```

```
    // Do the Q-formula
```

```
    NewQ = CurrentQ + alpha * (Reward + (gamma * NextQMax) - CurrentQ);
```

```
    // Write the new Q value back to the array.
```

```
    // First try to add the value NewQ to the keyword location QKeywordAction.
```

```
    // If that element already exists, just overwrite it instead.
```

```
    // Q2  
/*    if (CanSeeGoal)  
    {  
        try  
        {  
            QTable2.Add(QKeywordAction, NewQ);  
        }  
    }  
*/
```

```

    }
    catch (ArgumentException)
    {
        QTable2[QKeywordAction] = NewQ;
    }
}
else
{ // contents of else were there before Q2 */
    try
    {
        QTable.Add(QKeywordAction, NewQ);
    }
    catch (ArgumentException)
    {
        QTable[QKeywordAction] = NewQ;
    }
}
/*
// end Q2
*/

/*
leftover code - done properly above
// Now get CurrentQ.
CurrentQ = QTableOld[State[State_Goal0], State[State_Goal1], State[State_Goal2],
State[State_EndOfWorld0], State[State_EndOfWorld1], State[State_EndOfWorld2],
State[State_Sonar0], State[State_Sonar1_2], State[State_Sonar3], State[State_Sonar4],
State[State_Sonar5_6], State[State_Sonar7], Action];

// Do the Q-formula
NewQ = CurrentQ + alpha * (Reward + (gamma * NextQMax) - CurrentQ);

// Write the new Q value back to the array.
QTableOld[State[State_Goal0], State[State_Goal1], State[State_Goal2],
State[State_EndOfWorld0], State[State_EndOfWorld1], State[State_EndOfWorld2],
State[State_Sonar0], State[State_Sonar1_2], State[State_Sonar3], State[State_Sonar4],
State[State_Sonar5_6], State[State_Sonar7], Action] = NewQ;
*/

/*
NOTE: OLD location for reset check.
if (AtEndOfWorld || AtGoal)
{ // If the robot has reached the goal or passed the end of the world, reset the simulation.
    SimulationReset();
}

```

```

*/

// Check to see if the auto rescue function needs to rescue the robot.
randomrescue = randomseeder.NextDouble();
if (Auto_Rescue_Toggle && (CurrentSteps > MaxStepsBeforeRescue) &&
BumpDetected && (randomrescue < Rescue_Chance))
{ // This requires four conditions:
// The Auto_Rescue_Toggle must be on.
// The CurrentSteps must be higher than MaxStepsBeforeRescue.
// The robot is currently in a Bump state.
// If those three conditions are met, then it has a Rescue_Chance chance to activate.
Rescue_Robot();
}

// Q-learning formula: State <- NextState
// NOTE: If the number of fields in the state changes, the loop count here must also be
changed.

// statetonextstate
// trytonext
for (int i = 0; i < NumStates; i++)
{
    State[i] = NextState[i];
}

// ***** RARRAR
// Do all Table_Values (RAR comment) calculations here.
QKeyword0 = GenerateQKeyword(State[State_Goal0], State[State_Goal1],
State[State_Goal2], State[State_EndOfWorld0], State[State_EndOfWorld1],
State[State_EndOfWorld2], State[State_Sonar0], State[State_Sonar1_2], State[State_Sonar3],
State[State_Sonar4], State[State_Sonar5_6], State[State_Sonar7], State[State_Focus0],
State[State_Focus1], State[State_Focus2], 0);
try
{
    GreedyQ = QTable[QKeyword0];
}
catch (KeyNotFoundException)
{
    GreedyQ = GenericQValue;
}

```

```

    }
    GreedyAction = 0;
    Table_Values[TableTestForward] = GreedyQ;
    for (int i = 1; i < NumActions; i++)
    {
        QKeywordi = GenerateQKeyword(State[State_Goal0], State[State_Goal1],
State[State_Goal2], State[State_EndOfWorld0], State[State_EndOfWorld1],
State[State_EndOfWorld2], State[State_Sonar0], State[State_Sonar1_2], State[State_Sonar3],
State[State_Sonar4], State[State_Sonar5_6], State[State_Sonar7], State[State_Focus0],
State[State_Focus1], State[State_Focus2], i);
        try
        {
            QTemp = QTable[QKeywordi];
        }
        catch (KeyNotFoundException)
        {
            QTemp = GenericQValue;
        }
        Table_Values[i] = QTemp;
        if (GreedyQ < QTemp)
        {
            GreedyQ = QTemp;
            GreedyAction = i;
        }
    }
    TableChoice = GreedyAction;
    // end RARRAR

```

```

// un-comment to restore GUI
ArenaMap.Invalidate();

// Check to see if the clockless timer has been activated.
if (Clockless_Timer_Toggle)
{ // If this is true, the clockless timer has been activated.
    // For the clockless timer, do the following:
    // Perform a loop Clockless_Loop_Count times.
    // In the loop, call the Clockless_Timer_Loop function to execute one episode.
    for (int i = 0; i < Clockless_Loop_Count; i++)
    {
        Clockless_Timer_Loop();
    }
    // When loop is done, refresh the map again.
    // un-comment to restore GUI
    ArenaMap.Invalidate();

```

```

    }

}

/*****
*****/

private void Clockless_Timer_Loop()
{
    // This function runs the clockless timer.
    // It is functionally identical to the RobotTimer_Tick function, except that it does not wait
for the timer to tick over.
    // Since it is the same, see RobotTimer_Tick above for all comments.
    // Any comments in this function have to do with the clockless timer.

    bool RobotTouching;
    int GreedyAction;
    double GreedyQ;
    double NextQMax;
    double CurrentQ;
    double NewQ;
    Int64 QKeyword0 = -1;
    Int64 QKeywordi = -1;
    Int64 QKeywordAction = -1;
    double QTemp;
    int[] SonarReadings = new int[NumSonars];
    int[] CameraGoalReadings = new int[NumCameras];
    int[] CameraEndOfWorldReadings = new int[NumCameras];
    double randomrescue;
    Random randomseeder = new Random();
    double epsilonrandom;
    double actionrandom;
    double myopicrandom;

    epsilonrandom = randomseeder.NextDouble();
    if (epsilonrandom > (1 - epsilon))
    {
        actionrandom = NumActions * randomseeder.NextDouble();
        if (actionrandom <= 1)
        {
            Action = Action_Forward;

```

```

    }
    else if (actionrandom <= 2)
    {
        Action = Action_Reverse;
    }
    else if (actionrandom <= 3)
    {
        Action = Action_Rotate_Left;
    }
    else if (actionrandom <= 4)
    {
        Action = Action_Rotate_Right;
    }
    else if (actionrandom <= 5)
    {
        Action = Action_Stop;
    }
    else if (actionrandom <= 6)
    {
        Action = Action_Big_Rotate_Left;
    }
    else
    {
        Action = Action_Big_Rotate_Right;
    }
}
else
{
    QKeyword0 = GenerateQKeyword(State[State_Goal0], State[State_Goal1],
State[State_Goal2], State[State_EndOfWorld0], State[State_EndOfWorld1],
State[State_EndOfWorld2], State[State_Sonar0], State[State_Sonar1_2], State[State_Sonar3],
State[State_Sonar4], State[State_Sonar5_6], State[State_Sonar7], State[State_Focus0],
State[State_Focus1], State[State_Focus2], 0);

    // Q2
    /* if (CanSeeGoal)
    {
        try
        {
            GreedyQ = QTable2[QKeyword0];
        }
        catch (KeyNotFoundException)
        {
            GreedyQ = GenericQValue;
        }
    }
}

```

```

else
{ // contents of else were there before Q2 */
  try
  {
    GreedyQ = QTable[QKeyword0];
  }
  catch (KeyNotFoundException)
  {
    GreedyQ = GenericQValue;
  }
}
/*
// end Q2
*/

GreedyAction = 0;
for (int i = 1; i < NumActions; i++)
{
  QKeywordi = GenerateQKeyword(State[State_Goal0], State[State_Goal1],
State[State_Goal2], State[State_EndOfWorld0], State[State_EndOfWorld1],
State[State_EndOfWorld2], State[State_Sonar0], State[State_Sonar1_2], State[State_Sonar3],
State[State_Sonar4], State[State_Sonar5_6], State[State_Sonar7], State[State_Focus0],
State[State_Focus1], State[State_Focus2], i);

  // Q2
  /*   if (CanSeeGoal)
  {
    try
    {
      QTemp = QTable2[QKeywordi];
    }
    catch (KeyNotFoundException)
    {
      QTemp = GenericQValue;
    }
  }
  else
  { // contents of else are from before Q2 */
    try
    {
      QTemp = QTable[QKeywordi];
    }
    catch (KeyNotFoundException)
    {
      QTemp = GenericQValue;
    }
  }
  /*
}

```

```

// end Q2
*/

if (GreedyQ < QTemp)
{
    GreedyQ = QTemp;
    GreedyAction = i;
}
}
Action = GreedyAction;
}
OldLocation = RobotLocation;
OldOrientation = RobotOrientation;
PerformAction(Action);
RobotTouching = RobotContacting();

/* SMALL
if ((State[State_Goal0] == Dist_Close) && (Action == Action_Stop))
{
    Reward = Reward_Goal;
    AtGoal = true;
}
ENDSMALL */

// REWARDMOVE
// Move to after Nextstate is found.
/*
if ((State[State_Goal0] == Dist_Close) && !(CurrentSteps == 0))
{
    // Q3
    if (GoalFrontCenter)
    {
        Reward = Reward3_Goal;
    }
    // Q2
    else if (CanSeeGoal)
    { // To get rid of Q3, also remove the else from the previous line.
        Reward = Reward2_Goal;
    }
    else
    { // contents of else are from before Q2
        Reward = Reward_Goal;
    }
}
// end Q2

```

```

    // end Q3
    AtGoal = true;
}

else if (AtEndOfWorld)
{
    // Q3
    if (GoalFrontCenter)
    {
        Reward = Reward3_End_of_World;
    }
    // Q2
    else if (CanSeeGoal)
    { // To get rid of Q3, also remove the else from the previous line.
        Reward = Reward2_End_of_World;
    }
    else
    { // contents of else are from before Q2
        Reward = Reward_End_of_World;
    }
    // end Q2
    // end Q3
}

// GOAL PROXIMITY
//     else if ((State[State_Goal0] == Dist_Medium) && !BumpDetected)
//     { // If the robot sees the goal in front at Medium distance, give it the
Reward_Goal_Front reward.
//         Reward = Reward_Goal_Front;
//     }
//     else if (((State[State_Goal1] == Dist_Close) || (State[State_Goal2] == Dist_Close) ||
(State[State_Goal1] == Dist_Medium) || (State[State_Goal2] ==
Dist_Medium))&&!BumpDetected)
//     { // If the robot sees the goal on the side, give it the Reward_Goal_Side reward.
//         Reward = Reward_Goal_Side;
//     }
// END GOAL PROXIMITY

else if (BumpDetected)
{
    // Q3
    if (GoalFrontCenter)
    {
        Reward = Reward3_Bump;
    }
}

```

```

// Q2
else if (CanSeeGoal)
{ // To get rid of Q3, also remove the else from the previous line.
  Reward = Reward2_Bump;
}
else
{ // contents of else are from before Q2
  Reward = Reward_Bump;
}
// end Q2
// end Q3
}
else if (Action == Action_Stop)
{
  // Q3
  if (GoalFrontCenter)
  {
    Reward = Reward3_Stop;
  }
  // Q2
  else if (CanSeeGoal)
  { // To get rid of Q3, also remove the else from the previous line.
    Reward = Reward2_Stop;
  }
  else
  { // contents of else are from before Q2
    Reward = Reward_Stop;
  }
  // end Q2
  // end Q3
}
else if (CenterOfHall && (Action == Action_Forward))
{
  // Q3
  if (GoalFrontCenter)
  {
    Reward = Reward3_Center_Forward;
  }
  // Q2
  else if (CanSeeGoal)
  { // To get rid of Q3, also remove the else from the previous line.
    Reward = Reward2_Center_Forward;
  }
  else
  { // contents of else are from before Q2
    Reward = Reward_Center_Forward;
  }
}

```

```

    }
    // end Q2
    // end Q3
}
else if (CenterOfHall && (Action == Action_Reverse))
{
    // Q3
    if (GoalFrontCenter)
    {
        Reward = Reward3_Center_Reverse;
    }
    // Q2
    else if (CanSeeGoal)
    { // To get rid of Q3, also remove the else from the previous line.
        Reward = Reward2_Center_Reverse;
    }
    else
    { // contents of else are from before Q2
        Reward = Reward_Center_Reverse;
    }
    // end Q2
    // end Q3
}
else if (CenterOfHall && ((Action == Action_Rotate_Left) || (Action ==
Action_Rotate_Right)))
{
    // Q3
    if (GoalFrontCenter)
    {
        Reward = Reward3_Center_Turn;
    }
    // Q2
    else if (CanSeeGoal)
    { // To get rid of Q3, also remove the else from the previous line.
        Reward = Reward2_Center_Turn;
    }
    else
    { // contents of else are from before Q2
        Reward = Reward_Center_Turn;
    }
    // end Q2
    // end Q3
}
else if (Action == Action_Reverse)
{
    // Q3

```

```

if (GoalFrontCenter)
{
    Reward = Reward3_Reverse;
}
// Q2
else if (CanSeeGoal)
{ // To get rid of Q3, also remove the else from the previous line.
    Reward = Reward2_Reverse;
}
else
{ // contents of else are from before Q2
    Reward = Reward_Reverse;
}
// end Q2
// end Q3
}
else if ((Action == Action_Rotate_Left) || (Action == Action_Rotate_Right))
{
    // Q3
    if (GoalFrontCenter)
    {
        Reward = Reward3_Turn;
    }
    // Q2
    else if (CanSeeGoal)
    { // To get rid of Q3, also remove the else from the previous line.
        Reward = Reward2_Turn;
    }
    else
    { // contents of else are from before Q2
        Reward = Reward_Turn;
    }
    // end Q2
    // end Q3
}
else if (Action == Action_Forward)
{
    // Q3
    if (GoalFrontCenter)
    {
        Reward = Reward3_Forward;
    }
    // Q2
    else if (CanSeeGoal)
    { // To get rid of Q3, also remove the else from the previous line.
        Reward = Reward2_Forward;
    }
}

```

```

    }
    else
    { // contents of else are from before Q2
      Reward = Reward_Forward;
    }
    // end Q2
    // end Q3
  }
  else if (CenterOfHall && ((Action == Action_Big_Rotate_Left) || (Action ==
Action_Big_Rotate_Right)))
  { // If the robot rotates left or right and is in the center of the hall, it gets the
Center_Turn reward.
    // Q3
    if (GoalFrontCenter)
    {
      Reward = Reward3_Center_Big_Turn;
    }
    // Q2
    else if (CanSeeGoal)
    { // To get rid of Q3, also remove the else from the previous line.
      Reward = Reward2_Center_Big_Turn;
    }
    else
    { // contents of else are from before Q2
      Reward = Reward_Center_Big_Turn;
    }
    // end Q2
    // end Q3
  }
  else if ((Action == Action_Big_Rotate_Left) || (Action == Action_Big_Rotate_Right))
  { // If the robot rotates left or right, it gets the Turn reward.
    // Q3
    if (GoalFrontCenter)
    {
      Reward = Reward3_Big_Turn;
    }
    // Q2
    else if (CanSeeGoal)
    { // To get rid of Q3, also remove the else from the previous line.
      Reward = Reward2_Big_Turn;
    }
    else
    { // contents of else are from before Q2
      Reward = Reward_Big_Turn;
    }
    // end Q2
  }

```

```

        // end Q3
    }

    else
    {
        Reward = -1000;
    }
    CurrentReward = CurrentReward + Reward;
    CurrentSteps++;

    // Q2
    if ((Reward == Reward_Bump) || (Reward == Reward2_Bump) || (Reward ==
Reward3_Bump))
    {
        CurrentWallHits++;
    }
    // comment out the following if statement for Q2
//    if (Reward == Reward_Bump)
//    {
//        CurrentWallHits++;
//    }
    // end Q2

    if (AtEndOfWorld || AtGoal)
    {
        SimulationReset();
    }
    // end REWARDMOVE */

    // DoubleQ
    // Set CanSeeGoal to false. If it can be seen, it will be set true inside of the
CalculateCameraToGoal functions.
    CanSeeGoal = false;
    // Q3
    // Set GoalFrontCenter to false for the same reason.
    GoalFrontCenter = false;
    // Q2.5
    // Also set GoalInFront to false.
    GoalInFront = false;

    for (int i = 0; i < NumCameras; i++)
    {
        CameraGoalReadings[i] = CalculateCameraToGoal(i);
        NextState[State_Goal0 + i] = SensorEncodeCamera(CameraGoalReadings[i]);
    }

```

```

// CameraFocus
// This is just a test so far. It should only set it for the front camera when the clockless
timer is not active
// If it's the front sensor and the distance is Close or Medium (i.e., not Dist_Far), set
FocusInCamera to TempFocusInCamera.
//   if (i == 0)
//   {
//   /*   if (NextState[State_Goal0] != Dist_Far)
//       {
//           FocusInCamera[i] = TempFocusInCamera;
//       }
//       else
//       {
//           FocusInCamera[i] = CameraFocusNone;
//       }
//       // QFocus */
//       NextState[State_Focus0 + i] = FocusInCamera[i];
//   }
// end CameraFocus

if (NextState[State_Goal0 + i] == Dist_Far)
{
    NextState[State_Focus0 + i] = CameraFocusNone;
//    FocusInCamera[i] = CameraFocusNone;
}
// end CameraFocus

// DoubleQ
if ((NextState[State_Goal0 + i] == Dist_Close) || (NextState[State_Goal0 + i] ==
Dist_Medium))
{ // If the camera reading just found (NextState[State_Goal0+i]) is Dist_Close or
Dist_Medium, set CanSeeGoal to true.
    CanSeeGoal = true;
}
else if ((NextState[State_Goal0 + i] == Dist_Far) && (CameraGoalReadings[i] <
(ReallyLongDistance / 2)))
{ // If the camera reading is Dist_Far but the distance to the goal is less than
ReallyLongDistance/2, generate a random number.
    myopicrandom = randomseeder.NextDouble();
    if (myopicrandom < MyopicPercent)
    { // If that random number is within MyopicPercent, set CanSeeGoal to true.
        CanSeeGoal = true;
    }
}
}

```

```

    // Q3
    if (CanSeeGoal && (FocusInCamera[FocusCamFront] == CameraFocusCenter))
    { // If the goal can be seen and is in the Center focus of the front camera, set
GoalFrontCenter true.
        GoalFrontCenter = true;
    }
    // end Q3
    // Q2.5
    if (CanSeeGoal && ((FocusInCamera[FocusCamFront] == CameraFocusLeft) ||
(FocusInCamera[FocusCamFront] == CameraFocusRight)))
    { // If the goal can be seen and is in the front camera but not in the center, set
GoalInFront true.
        GoalInFront = true;
    }
    // end Q2.5

    CameraEndOfWorldReadings[i] = CalculateCameraToEndOfWorld(i);
    NextState[State_EndOfWorld0 + i] =
SensorEncodeCamera(CameraEndOfWorldReadings[i]);
}
for (int i = 0; i < NumSonars; i++)
{
    SonarReadings[i] = CalculateSonar(i);
}
NextState[State_Sonar0] = SensorEncodeSonar(SonarReadings[0]);
NextState[State_Sonar1_2] = SensorEncodeSonar((SonarReadings[1] +
SonarReadings[2]) / 2);
NextState[State_Sonar3] = SensorEncodeSonar(SonarReadings[3]);
NextState[State_Sonar4] = SensorEncodeSonar(SonarReadings[4]);
NextState[State_Sonar5_6] = SensorEncodeSonar((SonarReadings[5] +
SonarReadings[6]) / 2);
NextState[State_Sonar7] = SensorEncodeSonar(SonarReadings[7]);
// Set CenterOfHall
if ((NextState[State_Sonar3] != Sonar_Bump) && (NextState[State_Sonar3] ==
NextState[State_Sonar7]))
{ // If the left and right sonar readings (3 and 7) are the same and aren't reading bumps,
then set CenterOfHall to true.
    CenterOfHall = true;
}
else
{ // otherwise set it false
    CenterOfHall = false;
}
}

```

```

    if ((NextState[State_Sonar0] == Dist_Bump) || (NextState[State_Sonar1_2] ==
Dist_Bump) || (NextState[State_Sonar3] == Dist_Bump) || (NextState[State_Sonar4] ==
Dist_Bump) || (NextState[State_Sonar5_6] == Dist_Bump) || (NextState[State_Sonar7] ==
Dist_Bump))
    {
        BumpDetected = true;
    }
    else
    {
        BumpDetected = false;
    }
    RobotAtEndOfWorld();

// REWARDMOVE
// Move the reward-calculating parts to here.
if ((State[State_Goal0] == Dist_Close) && !(CurrentSteps == 0))
{
    // Q3
    if (GoalFrontCenter)
    {
        Reward = Reward3_Goal;
    }
    // Q2.5
    else if (GoalInFront)
    {
        Reward = Reward25_Goal;
    }
    // Q2
    else if (CanSeeGoal)
    { // To get rid of Q3, also remove the else from the previous line.
        Reward = Reward2_Goal;
    }
    else
    { // contents of else are from before Q2
        Reward = Reward_Goal;
    }
    // end Q2
    // end Q2.5
    // end Q3

    AtGoal = true;
}
else if (AtEndOfWorld)

```

```

    { // If the robot is in the area designated as the End of the World, then it gets the End of
the World reward.
    // Q3
    if (GoalFrontCenter)
    {
        Reward = Reward3_End_of_World;
    }
    // Q2.5
    else if (GoalInFront)
    {
        Reward = Reward25_End_of_World;
    }
    // Q2
    else if (CanSeeGoal)
    { // To get rid of Q3, also remove the else from the previous line.
        Reward = Reward2_End_of_World;
    }
    else
    { // contents of else are from before Q2
        Reward = Reward_End_of_World;
    }
    // end Q2
    // end Q2.5
    // end Q3
}
// GOAL PROXIMITY
//     else if ((State[State_Goal0] == Dist_Medium) && !BumpDetected)
//     { // If the robot sees the goal in front at Medium distance, give it the
Reward_Goal_Front reward.
//         Reward = Reward_Goal_Front;
//     }
//     else if (((State[State_Goal1] == Dist_Close) || (State[State_Goal2] == Dist_Close) ||
(State[State_Goal1] == Dist_Medium) || (State[State_Goal2] == Dist_Medium)) &&
!BumpDetected)
//     { // If the robot sees the goal on the side, give it the Reward_Goal_Side reward.
//         Reward = Reward_Goal_Side;
//     }
// END GOAL PROXIMITY
else if (BumpDetected && !GoalFrontCenter) // Q3 - added !GoalFrontCenter so bump
doesn't matter if the goal is directly in front.
{ // If the robot is in a bump position, then it gets the Bump reward.
// Q3
if (GoalFrontCenter)
{
    Reward = Reward3_Bump;
}
}

```

```

// Q2.5
else if (GoalInFront)
{
    Reward = Reward25_Bump;
}
// Q2
else if (CanSeeGoal)
{ // To get rid of Q3, also remove the else from the previous line.
    Reward = Reward2_Bump;
}
else
{ // contents of else are from before Q2
    Reward = Reward_Bump;
}
// end Q2
// end Q2.5
// end Q3
}
else if (Action == Action_Stop)
{ // If the robot stops, it gets the Stop reward.
// Q3
if (GoalFrontCenter)
{
    Reward = Reward3_Stop;
}
// Q2.5
else if (GoalInFront)
{
    Reward = Reward25_Stop;
}
// Q2
else if (CanSeeGoal)
{ // To get rid of Q3, also remove the else from the previous line.
    Reward = Reward2_Stop;
}
else
{ // contents of else are from before Q2
    Reward = Reward_Stop;
}
// end Q2
// end Q2.5
// end Q3
}
else if (CenterOfHall && (Action == Action_Forward))
{ // If the robot goes forward and is in the center of the hall, it gets the Center_Forward
reward.

```

```

// Q3
if (GoalFrontCenter)
{
    Reward = Reward3_Center_Forward;
}
// Q2.5
else if (GoalInFront)
{
    Reward = Reward25_Center_Forward;
}
// Q2
else if (CanSeeGoal)
{ // To get rid of Q3, also remove the else from the previous line.
    Reward = Reward2_Center_Forward;
}
else
{ // contents of else are from before Q2
    Reward = Reward_Center_Forward;
}
// end Q2
// end Q2.5
// end Q3
}
else if (CenterOfHall && (Action == Action_Reverse))
{ // If the robot goes in reverse and is in the center of the hall, it gets the Center_Reverse
reward.
// Q3
if (GoalFrontCenter)
{
    Reward = Reward3_Center_Reverse;
}
// Q2.5
else if (GoalInFront)
{
    Reward = Reward25_Center_Reverse;
}
// Q2
else if (CanSeeGoal)
{ // To get rid of Q3, also remove the else from the previous line.
    Reward = Reward2_Center_Reverse;
}
else
{ // contents of else are from before Q2
    Reward = Reward_Center_Reverse;
}
// end Q2

```

```

        // end Q2.5
        // end Q3
    }
    else if (CenterOfHall && ((Action == Action_Rotate_Left) || (Action ==
Action_Rotate_Right)))
    { // If the robot rotates left or right and is in the center of the hall, it gets the
Center_Turn reward.
        // Q3
        if (GoalFrontCenter)
        {
            Reward = Reward3_Center_Turn;
        }
        // Q2.5
        else if (GoalInFront)
        {
            Reward = Reward25_Center_Turn;
        }
        // Q2
        else if (CanSeeGoal)
        { // To get rid of Q3, also remove the else from the previous line.
            Reward = Reward2_Center_Turn;
        }
        else
        { // contents of else are from before Q2
            Reward = Reward_Center_Turn;
        }
        // end Q2
        // end Q2.5
        // end Q3
    }
    else if (Action == Action_Reverse)
    { // If the robot goes in reverse, it gets the Reverse reward.
        // Q3
        if (GoalFrontCenter)
        {
            Reward = Reward3_Reverse;
        }
        // Q2.5
        else if (GoalInFront)
        {
            Reward = Reward25_Reverse;
        }
        // Q2
        else if (CanSeeGoal)
        { // To get rid of Q3, also remove the else from the previous line.
            Reward = Reward2_Reverse;
        }
    }
}

```

```

}
else
{ // contents of else are from before Q2
  Reward = Reward_Reverse;
}
// end Q2
// end Q2.5
// end Q3
}
else if ((Action == Action_Rotate_Left) || (Action == Action_Rotate_Right))
{ // If the robot rotates left or right, it gets the Turn reward.
  // Q3
  if (GoalFrontCenter)
  {
    Reward = Reward3_Turn;
  }
  // Q2.5
  else if (GoalInFront)
  {
    Reward = Reward25_Turn;
  }
  // Q2
  else if (CanSeeGoal)
  { // To get rid of Q3, also remove the else from the previous line.
    Reward = Reward2_Turn;
  }
  else
  { // contents of else are from before Q2
    Reward = Reward_Turn;
  }
  // end Q2
  // end Q2.5
  // end Q3
}
else if (Action == Action_Forward)
{ // If the robot moves forward, it gets the Forward reward.
  // Q3
  if (GoalFrontCenter)
  {
    Reward = Reward3_Forward;
  }
  // Q2.5
  else if (GoalInFront)
  {
    Reward = Reward25_Forward;
  }
}

```

```

// Q2
else if (CanSeeGoal)
{ // To get rid of Q3, also remove the else from the previous line.
  Reward = Reward2_Forward;
}
else
{ // contents of else are from before Q2
  Reward = Reward_Forward;
}
// end Q2
// end Q2.5
// end Q3
}
else if (CenterOfHall && ((Action == Action_Big_Rotate_Left) || (Action ==
Action_Big_Rotate_Right)))
{ // If the robot rotates left or right and is in the center of the hall, it gets the
Center_Turn reward.
  // Q3
  if (GoalFrontCenter)
  {
    Reward = Reward3_Center_Big_Turn;
  }
  // Q2.5
  else if (GoalInFront)
  {
    Reward = Reward25_Center_Big_Turn;
  }
  // Q2
  else if (CanSeeGoal)
  { // To get rid of Q3, also remove the else from the previous line.
    Reward = Reward2_Center_Big_Turn;
  }
  else
  { // contents of else are from before Q2
    Reward = Reward_Center_Big_Turn;
  }
  // end Q2
  // end Q2.5
  // end Q3
}
else if ((Action == Action_Big_Rotate_Left) || (Action == Action_Big_Rotate_Right))
{ // If the robot rotates left or right, it gets the Turn reward.
  // Q3
  if (GoalFrontCenter)
  {
    Reward = Reward3_Big_Turn;
  }
}

```

```

    }
    // Q2.5
    else if (GoalInFront)
    {
        Reward = Reward25_Big_Turn;
    }
    // Q2
    else if (CanSeeGoal)
    { // To get rid of Q3, also remove the else from the previous line.
        Reward = Reward2_Big_Turn;
    }
    else
    { // contents of else are from before Q2
        Reward = Reward_Big_Turn;
    }
    // end Q2
    // end Q2.5
    // end Q3
}
else
{ // This should never happen, but is here just in case
    Reward = -1000;
}
// Update the CurrentReward.
CurrentReward = CurrentReward + Reward;
// Update the number of steps.
CurrentSteps++;
// Q2
if ((Reward == Reward_Bump) || (Reward == Reward2_Bump) || (Reward ==
Reward3_Bump))
{
    CurrentWallHits++;
}
// Comment out the following if statement for Q2
//     if (Reward == Reward_Bump)
//     { // If there was a bump, add to the number of bumps.
//         CurrentWallHits++;
//     }
// end Q2
// After rewards are assigned...
if (AtEndOfWorld || AtGoal)
{ // If the robot has reached the goal or passed the end of the world, reset the simulation.
    SimulationReset();
}

// end REWARDMOVE

```

```

    QKeyword0 = GenerateQKeyword(State[State_Goal0], State[State_Goal1],
State[State_Goal2], State[State_EndOfWorld0], State[State_EndOfWorld1],
State[State_EndOfWorld2], State[State_Sonar0], State[State_Sonar1_2], State[State_Sonar3],
State[State_Sonar4], State[State_Sonar5_6], State[State_Sonar7], State[State_Focus0],
State[State_Focus1], State[State_Focus2], 0);

```

```

    // Q2
    /*    if (CanSeeGoal)
        {
            try
            {
                NextQMax = QTable2[QKeyword0];
            }
            catch (KeyNotFoundException)
            {
                NextQMax = GenericQValue;
            }
        }
    else
    { // contents of else are from before Q2 */
        try
        {
            NextQMax = QTable[QKeyword0];
        }
        catch (KeyNotFoundException)
        {
            NextQMax = GenericQValue;
        }
    }
    /*
    // end Q2
    */

```

```

    for (int i = 1; i < NumActions; i++)
    {
        QKeywordi = GenerateQKeyword(State[State_Goal0], State[State_Goal1],
State[State_Goal2], State[State_EndOfWorld0], State[State_EndOfWorld1],
State[State_EndOfWorld2], State[State_Sonar0], State[State_Sonar1_2], State[State_Sonar3],
State[State_Sonar4], State[State_Sonar5_6], State[State_Sonar7], State[State_Focus0],
State[State_Focus1], State[State_Focus2], i);

```

```

    // Q2
    /*    if (CanSeeGoal)
        {
            try

```

```

        {
            QTemp = QTable2[QKeywordi];
        }
        catch (KeyNotFoundException)
        {
            QTemp = GenericQValue;
        }
    }
else
{ // contents of else are from before Q2 */
    try
    {
        QTemp = QTable[QKeywordi];
    }
    catch (KeyNotFoundException)
    {
        QTemp = GenericQValue;
    }
}
/*
// end Q2
*/

if (NextQMax < QTemp)
{
    NextQMax = QTemp;
}

QKeywordAction = GenerateQKeyword(State[State_Goal0], State[State_Goal1],
State[State_Goal2], State[State_EndOfWorld0], State[State_EndOfWorld1],
State[State_EndOfWorld2], State[State_Sonar0], State[State_Sonar1_2], State[State_Sonar3],
State[State_Sonar4], State[State_Sonar5_6], State[State_Sonar7], State[State_Focus0],
State[State_Focus1], State[State_Focus2], Action);

// Q2
/*
    if (CanSeeGoal)
    {
        try
        {
            CurrentQ = QTable2[QKeywordAction];
        }
        catch (KeyNotFoundException)
        {
            CurrentQ = GenericQValue;
        }
    }
}

```

```

else
{ // contents of else are from before Q2 */
  try
  {
    CurrentQ = QTable[QKeywordAction];
  }
  catch (KeyNotFoundException)
  {
    CurrentQ = GenericQValue;
  }
}
/*
// end Q2
*/

NewQ = CurrentQ + alpha * (Reward + (gamma * NextQMax) - CurrentQ);

// Q2
/*
  if (CanSeeGoal)
  {
    try
    {
      QTable2.Add(QKeywordAction, NewQ);
    }
    catch (ArgumentException)
    {
      QTable2[QKeywordAction] = NewQ;
    }
  }
else
{ // contents of else are from before Q2 */
  try
  {
    QTable.Add(QKeywordAction, NewQ);
  }
  catch (ArgumentException)
  {
    QTable[QKeywordAction] = NewQ;
  }
}
/*
//end Q2
*/

randomrescue = randomseeder.NextDouble();
if (Auto_Rescue_Toggle && (CurrentSteps > MaxStepsBeforeRescue) &&
BumpDetected && (randomrescue < Rescue_Chance))
{

```

```

        Rescue_Robot();
    }
    for (int i = 0; i < NumStates; i++)
    {
        State[i] = NextState[i];
    }

    return;

}

/*****
*****/

private void SimulationReset()
{ // This function is called if the robot reaches the goal or goes off the end of the world.
  // It 'resets' the simulation to prepare it for the next run.

  int[] SonarReadings = new int[NumSonars];
  int[] CameraGoalReadings = new int[NumCameras];
  int[] CameraEndOfWorldReadings = new int[NumCameras];

  // May not be needed
  // RemainderAnswer is needed for the DivRem function to find the remainder.
  //int RemainderAnswer;

  // Update the average reward and number of steps and number of bumps to date.
  CalcAvgReward();
  CalcAvgSteps();
  CalcAvgWallHits();

  if (AtGoal)
  { // If the episode ended because the robot hit the goal, increase the NumberGoals
    NumberGoals++;
  }

  // First, write the QTable to the QFile as backup.
  WriteQTable();

  // If the number of repetitions is divisible by the backup frequency, also write to the
  backup QFile.
  // try using % for mod
  //if (Math.DivRem(NumRepetitions, BackupFreq, out RemainderAnswer) == 0)
  if ((NumRepetitions % BackupFreq) == 0)
  {

```

```

    WriteBackupQTable();
}

// Now set AtGoal and AtEndOfWorld to false.
AtGoal = false;

// NOTE: should not need to set AtEndOfWorld to false since it's checked every tick.
// AtEndOfWorld = false;

// Change the location back to the starting position.
RobotLocation[x] = StartLocation[x];
RobotLocation[y] = StartLocation[y];
OldLocation[x] = StartLocation[x];
OldLocation[y] = StartLocation[y];

// The RobotOrientation can remain as it is. This will help with the randomness of the
starting state.

// RAR
// We need a spinrandom here so the orientation will not always start pointing at the goal
// float spinrandom;
// Random randomseeder = new Random();
// spinrandom = (float)((360 * randomseeder.NextDouble()) * Math.PI / 180);
// RobotOrientation = spinrandom;
// RAR
RobotOrientation = StartOrientation + fivedegrees;
OldOrientation = RobotOrientation;

// Comment this part out.
// This is if the reset check happens AFTER the new sensor reading. Currently, it is
BEFORE.
/*
// Find the new values for the State.
// First Sonars
for (int i = 0; i < NumSonars; i++)
{
    SonarReadings[i] = CalculateSonar(i);
}
NextState[State_Sonar0] = SensorEncodeSonar(SonarReadings[0]);
NextState[State_Sonar1_2] = SensorEncodeSonar((SonarReadings[1] +
SonarReadings[2]) / 2);
NextState[State_Sonar3] = SensorEncodeSonar(SonarReadings[3]);
NextState[State_Sonar4] = SensorEncodeSonar(SonarReadings[4]);
NextState[State_Sonar5_6] = SensorEncodeSonar((SonarReadings[5] +
SonarReadings[6]) / 2);
NextState[State_Sonar7] = SensorEncodeSonar(SonarReadings[7]);

```

```

// Then cameras.
for (int i = 0; i < NumCameras; i++)
{
    CameraGoalReadings[i] = CalculateCameraToGoal(i);
    NextState[State_Goal0 + i] = SensorEncodeCamera(CameraGoalReadings[i]);
    CameraEndOfWorldReadings[i] = CalculateCameraToEndOfWorld(i);
    NextState[State_EndOfWorld0 + i] =
SensorEncodeCamera(CameraEndOfWorldReadings[i]);
}
// NOTE: If more sensors are added, make sure to add them here in addition to the
Timer_Tick function.
*/

// Increase the number of repetitions by 1.
NumRepetitions++;

// Reset the Has_Been_Rescued toggle.
Has_Been_Rescued = false;

// Reset the current reward and steps and wall hits
CurrentReward = 0;
CurrentSteps = 0;
CurrentWallHits = 0;

return;
}

/*****
*****/

// following are temporary functions for the local control panel
// they will be removed once the floating control panel works

private void Begin_Local_Click(object sender, EventArgs e)
{
    RobotTimer.Enabled = true;
}

private void Exit_Local_Click(object sender, EventArgs e)
{
    RobotTimer.Enabled = false;

// Update the average reward and number of steps and number of bumps to date.
CalcAvgReward();
CalcAvgSteps();
CalcAvgWallHits();

```

```

// First, write the QTable to the QFile as backup.
WriteQTable();

// If the number of repetitions is divisible by the backup frequency, also write to the
backup QFile.
if ((NumRepetitions % BackupFreq) == 0)
{
    WriteBackupQTable();
}

this.Close();
}

private void Pause_Local_Click(object sender, EventArgs e)
{
    RobotTimer.Enabled = !RobotTimer.Enabled;
}

private void Trail_Toggle_Click(object sender, EventArgs e)
{
    Trail_Toggle = !Trail_Toggle;
}

private void Sim_Speed_Scroll(object sender, EventArgs e)
{
    // Sim_Speed.Value can be any integer from 0 to 10.
    // Original settings:
    // Sim_Speed.Value RobotTimer.Interval (milliseconds)
    // 0           250
    // 1           200
    // 2           150
    // 3           100
    // 4           75
    // 5           50
    // 6           30
    // 7           20
    // 8           10
    // 9           5
    // 10          1

    // These values were decided semi-arbitrarily such that the slowest setting had 4 Hz and
the fastest setting was the fastest possible setting for the timer at 1 millisecond intervals.
    // The values were then fed into a least-squares regression best-fit function program.
    // After looking through different best-fit functions (linear, exponential, quadratic), the
quadratic fit the points best.

```

```
// y = 3.11189x^2 - 55.2098x + 248.133
// Where x is the Sim_Speed.Value and y is the RobotTimer.Interval
// Convert it all to type int.
// It could have been done with a string of 10 if/then/else statements, but using a least-
squares regression function is nicer and more elegant.
```

```
RobotTimer.Interval = (int)(3.11189 * Math.Pow(Sim_Speed.Value, 2) - 55.2098 *
Sim_Speed.Value + 248.133);
```

```
}
```

```
private void Show_Sensors_Click(object sender, EventArgs e)
{
    Show_Sensors_Toggle = !Show_Sensors_Toggle;
}
```

```
private void Show_Stats_Click(object sender, EventArgs e)
{
    Show_Stats_Toggle = !Show_Stats_Toggle;
}
```

```
private void Manual_Rescue_Click(object sender, EventArgs e)
{
    Rescue_Robot();
}
```

```
private void Auto_Rescue_Click(object sender, EventArgs e)
{
    Auto_Rescue_Toggle = !Auto_Rescue_Toggle;
}
```

```
private void Exit_Without_Save_Click(object sender, EventArgs e)
{
    // This button is to exit without saving the current episode.
    // In other words, it doesn't force a save of a partially-completed episode.
    RobotTimer.Enabled = false;
    this.Close();
}
```

```
private void Show_HUD_Click(object sender, EventArgs e)
{
    Show_HUD_Toggle = !Show_HUD_Toggle;
}
```

```
private void Clockless_Timer_Click(object sender, EventArgs e)
```

```

{
    Clockless_Timer_Toggle = !Clockless_Timer_Toggle;
}

private void Show_Q_Toggle_Click(object sender, EventArgs e)
{
    Show_Q_Toggle = !Show_Q_Toggle;
}

private void Manual_Mode_Click(object sender, EventArgs e)
{
    Manual_Mode_Toggle = !Manual_Mode_Toggle;
}

private void Step_Timer_Click(object sender, EventArgs e)
{
    Manual_Timer_Step = true;
}

private void Manual_Left_Click(object sender, EventArgs e)
{
    Manual_Forward_Toggle = false;
    Manual_Left_Toggle = true;
    Manual_Right_Toggle = false;
    Manual_Stop_Toggle = false;
    Manual_Reverse_Toggle = false;
    Manual_Big_Left_Toggle = false;
    Manual_Big_Right_Toggle = false;
}

private void Manual_Forward_Click(object sender, EventArgs e)
{
    Manual_Forward_Toggle = true;
    Manual_Left_Toggle = false;
    Manual_Right_Toggle = false;
    Manual_Stop_Toggle = false;
    Manual_Reverse_Toggle = false;
    Manual_Big_Left_Toggle = false;
    Manual_Big_Right_Toggle = false;
}

private void Manual_Right_Click(object sender, EventArgs e)
{
    Manual_Forward_Toggle = false;
    Manual_Left_Toggle = false;
    Manual_Right_Toggle = true;
}

```

```

Manual_Stop_Toggle = false;
Manual_Reverse_Toggle = false;
Manual_Big_Left_Toggle = false;
Manual_Big_Right_Toggle = false;
}

private void Manual_Stop_Click(object sender, EventArgs e)
{
Manual_Forward_Toggle = false;
Manual_Left_Toggle = false;
Manual_Right_Toggle = false;
Manual_Stop_Toggle = true;
Manual_Reverse_Toggle = false;
Manual_Big_Left_Toggle = false;
Manual_Big_Right_Toggle = false;
}

private void Manual_Reverse_Click(object sender, EventArgs e)
{
Manual_Forward_Toggle = false;
Manual_Left_Toggle = false;
Manual_Right_Toggle = false;
Manual_Stop_Toggle = false;
Manual_Reverse_Toggle = true;
Manual_Big_Left_Toggle = false;
Manual_Big_Right_Toggle = false;
}

private void Manual_Big_Left_Click(object sender, EventArgs e)
{
Manual_Forward_Toggle = false;
Manual_Left_Toggle = false;
Manual_Right_Toggle = false;
Manual_Stop_Toggle = false;
Manual_Reverse_Toggle = false;
Manual_Big_Left_Toggle = true;
Manual_Big_Right_Toggle = false;
}

private void Manual_Big_Right_Click(object sender, EventArgs e)
{
Manual_Forward_Toggle = false;
Manual_Left_Toggle = false;
Manual_Right_Toggle = false;
Manual_Stop_Toggle = false;
Manual_Reverse_Toggle = false;
}

```

```
Manual_Big_Left_Toggle = false;
Manual_Big_Right_Toggle = true;
}
```

```
/******  
*****/
```

```
/******  
*****/
```

```
}  
}
```

APPENDIX D ROBOT CODE

```
/******  
***  
SBC Code  
Description: Basic send and receive functions between Atmega128 and SBC and  
Lavi's laptop which should be running the AI program.  
All of the internet stuff is commented out.  
Added Vision - initializes colors and detects how far away a colored square  
is.  
  
Some code reformatting, e.g. moving main() to the top.  
  
Date: 8/10/2007  
REV 4  
*****  
***  
*/  
  
#ifdef _CH_  
#pragma package <opencv>  
#endif  
  
#include "cv.h"  
#include "highgui.h"  
#include "cxcore.h"  
  
#include <stdio.h>  
#include <fcntl.h>  
#include <stdlib.h>  
#include <termios.h>  
#include <math.h>  
#include <errno.h>  
#include <sys/socket.h> /* for socket() and bind() */  
#include <arpa/inet.h> /* for sockaddr_in and inet_ntoa() */  
#include <string.h> /* for memset() */  
#include <unistd.h> /* for close() and sleep() */  
  
#define SENSITIVITY 4096 //How close a pixel has to be to the calibrated  
color to be recognized as that color (this number can range from 0(only an  
exact color matches) to 196608(any color matches)  
  
#define DATAPORT "/dev/ttyS0"  
#define LAVIPOINT "/dev/ttyS1"  
#define BUFFER_SIZE 8  
#define ECHOMAX 255 /* Longest string for socket*/  
  
#define ARROYO 1  
#define SCHWARTZ 2  
#define GO_LEFT 3  
#define GO_RIGHT 4  
#define GO_BACKWARD 5  
#define GO_FORWARD 6  
#define STOP 7
```

```

#define LEFT_CAMERA 0
#define FRONT_CAMERA 1
#define RIGHT_CAMERA 2

// Defines for Lavi's added cde to follow Q table
// All additions marked with 'start Q' in comment.
// start Q
// use the MOTOR constants because Koolio's right wheel seems to be running
backwards
#define MOTOR_FORWARD 4
#define MOTOR_REVERSE 3
#define MOTOR_LEFT 5
#define MOTOR_RIGHT 6
#define MOTOR_STOP 7
#define Action_Forward 0
#define Action_Reverse 1
#define Action_Rotate_Left 2
#define Action_Rotate_Right 3
#define Action_Stop 4
#define Action_Big_Rotate_Left 5
#define Action_Big_Rotate_Right 6
#define Bump_Radius 6
#define Sonar_Bump 6
#define Sonar_Close 24
#define Sonar_Far 60
#define Camera_Close 36
#define Camera_Far 96
#define NumActions 7
#define Dist_Bump 3
#define Dist_Close 0
#define Dist_Medium 1
#define Dist_Far 2
#define CameraFocusNone 3
#define CameraFocusLeft 0
#define CameraFocusRight 1
#define CameraFocusCenter 2
// end Q

//some people think they know better...
#define TRUE 1
#define FALSE 0

//Function Prototypes
int open_serial_port(char *port, int baud);
void update_display ();
void speak_text (char *text);
int calibrate_camera(IplImage *imgDeviationFromColor, long color_avg[3]);
short classify_target (int camera, IplImage *imgDeviationFromColor, long
color_target[3]);
// start Q
long GenerateQKeyword(int GoalCam0, int GoalCam1, int GoalCam2, int EoWCam0,
int EoWCam1, int EoWCam2, int Son0, int Son12, int Son3, int Son4, int Son56,
int Son7, int GoalFoc0, int GoalFoc1, int GoalFoc2, int Action);
double QSearch(long Keyword); // This is for searching the QTable
int SensorEncodeCamera(short CameraReading); // to encode the camera
int SensorEncodeSonar(unsigned char SonarReading); // to encode the sonar

```

```

// prototype for sonar classification goes here
// end Q

//Variables used to access the serial ports
int data_ioport;
int lavi_ioport;

//I2C Sensors
unsigned char TR, TL, CR, CL, BR, BL, BK, CMP, DRIVE_CMD;

int msg_num; //1 = Arroyo
              //2 = Schwartz
              //3 = go left
              //4 = go right
              //5 = go backward
              //6 = go forward
              //7 = stop

//Global Variables for GUI code

int maxval;

// start Q
// use a linked list for the QTable
struct QEntry
{
    long QKey;
    double QValue;
    struct QEntry *NextQEntry;
};
struct QEntry *QTable;

// Following are used for I2C sensors. Use these in place of TR, TL, CR, CL,
BR, BL, BK
// CR (moved to front of robot) -> SON0
// BL -> SON12
// TL -> SON3
// BK -> SON4
// BR -> SON56
// TR -> SON7
unsigned char SON0, SON12, SON3, SON4, SON56, SON7;

// end Q

//=====
//
// main()
//=====
//=====
int main(int argc, char *argv[])
{
    int i;
    int sock; // So/home/zardoz/Desktop/Serial/cket */
    struct sockaddr_in echoServAddr; // Local address */

```

```

struct sockaddr_in echoClntAddr; /* Client address */
int cliAddrLen;                /* Length of incoming message */
char echoBuffer[ECHOMAX];     /* Buffer for echo string */
unsigned short echoServPort;  /* Server port */
int recvMsgSize;              /* Size of received message */
int ignore_next_msg;
char *say_it;

long color_avg_goal[3];
long color_avg_worldend[3];

short goal_dist0      = 0;
short worldend_dist0 = 0;
short goal_dist1      = 0;
short worldend_dist1 = 0;
short goal_dist2      = 0;
short worldend_dist2 = 0;
//holds the image used to show the deviation between the actual image and
the
//target RGB value - pixels closest to the target color are white.
IplImage* imgDeviationFromColor;

// start Q
int atgoal=0;
// variables for reading from file
int ReadInt;
double ReadDouble;
long ReadLong;
// initialize the linked list
// struct QEntry *QTable;
struct QEntry *CurrentQEntry;
CurrentQEntry = malloc(sizeof(struct QEntry));
CurrentQEntry->QKey = -1;
CurrentQEntry->QValue = 0;
CurrentQEntry->NextQEntry = 0;
QTable = CurrentQEntry; // QTable points to the beginning.

// Open file here to read
printf("Reading QFile...");
FILE *QFile;
QFile = fopen("/home/zardoz/Desktop/Serial/QFile.data","r");
// Also read in statistic data to move on to the table.  Statistic data is
unneded here.
// Statistic data is 2 ints and 3 doubles.
fread(&ReadInt, sizeof(ReadInt), 2, QFile);
fread(&ReadDouble, sizeof(ReadDouble), 3, QFile);
// Read file in loop, writing new nodes.
// Read first entry first.
fread(&ReadLong, sizeof(ReadLong), 1, QFile);
fread(&ReadDouble, sizeof(ReadDouble), 1, QFile);
CurrentQEntry->QKey = ReadLong;
CurrentQEntry->QValue = ReadDouble;
CurrentQEntry->NextQEntry = malloc(sizeof(struct QEntry));
// QTable = CurrentQEntry; // Set QTable to the first entry as a reference
point.
// Now go through the table, adding new nodes to the list as they are read.
if (CurrentQEntry != 0)

```

```

{
    while(!feof(QFile))
    { // Until the end of file is reached
        CurrentQEntry = CurrentQEntry->NextQEntry; // Go to next
entry.

        fread(&ReadLong, sizeof(ReadLong), 1, QFile);
        fread(&ReadDouble, sizeof(ReadDouble), 1, QFile);
        CurrentQEntry->QKey = ReadLong;
        CurrentQEntry->QValue = ReadDouble;
        CurrentQEntry->NextQEntry = malloc(sizeof(struct QEntry));
    }
}
CurrentQEntry->NextQEntry = 0; // make the last QEntry with a null pointer.
// when done, close the file
fclose(QFile);
// end file reading
printf(" done!\n");
// end Q

// Added Aug 22, 2006 Andrew Chambers
// Run find_ip.sh script to find the ip address and update it on the
// koolio webpage
// system("/home/zardoz/find_ip.sh");

//Display image and play laughter:
// system("aplay /usr/share/sounds/gnibbles/laughter.wav");
// system("qiv -Tfmi ~/Desktop/Serial/Current.jpg &");

// speak_text("If you don't know who it is it's Koolio");
// sleep(5);
// system("mpg123 ~/Desktop/if_you_dont_know_koolio.mp3");
// speak_text("Hello, my name is Koolio. May the Schwartz be with you.");
// speak_text("Hello my name is Koolio and this is my theme song");
// sleep(5);
// system("mpg123 ~/Desktop/theme_song1_koolio.mp3 &");

//1 char buffer for send 1 byte of data to Altera board.
//unsigned char message = 0x00;

//8 byte char buffer for receiving the 10 sensor readings from the Altera
board.
//unsigned char value[BUFFER_SIZE] = "0";
unsigned char input;

//Open connection at 9600 baud
data_ioport = open_serial_port(DATAPORT, 9600);
lavi_ioport = open_serial_port(LAVIPOINT, 9600);

//Read and ignore the initial 0x00 byte that seems to be sent
//by the Mavric board when starting up the UART
//while(read(data_ioport, &value, BUFFER_SIZE)==0);

/* get port number */

// if (argc != 2) // Test for correct number of parameters */
// {
//     printf(stderr, "Usage: %s <UDP SERVER PORT>\n", argv[0]);

```

```

//      exit(1);
//  }

//  echoServPort = atoi(argv[1]); /* First arg:  local port */

/* Create socket for sending/receiving datagrams */

//  if ((sock = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0)
//      printf("socket() failed");

/* Construct local address structure */

//  memset(&echoServAddr, 0, sizeof(echoServAddr)); /* Zero out structure
//  echoServAddr.sin_family = AF_INET; /* Internet address
family
//echoServAddr.sin_addr.s_addr = htonl(INADDR_ANY); /*
Any incoming interface
//  echoServAddr.sin_port = htons(echoServPort); /* Local port

/* Bind to the local address */
//  if (bind(sock, (struct sockaddr *) &echoServAddr, sizeof(echoServAddr)) <
0)
//      printf("bind() failed");

//  ignore_next_msg = 0;

//  start Q
//  at beginning, stop the motors.
input = MOTOR_STOP;
while(write(data_ioport, &input, 1) == -1);
//  end Q

cvNamedWindow("img", CV_WINDOW_AUTOSIZE);
cvNamedWindow("imgDeviationFromColor", CV_WINDOW_AUTOSIZE);
imgDeviationFromColor = cvCreateImage(cvSize(320, 240), IPL_DEPTH_8U, 1);

/* Calibration for the goal and end-of-the-world colors
printf("Please place end of world color in front of the right camera, then
press ESC.\n");
speak_text("Put end of world color in front of the right camera and press
escape");
if(calibrate_camera(imgDeviationFromColor, color_avg_goal) == -1)
return -1;
printf("END OF WORLD CALIBRATION COMPLETE! \n");

printf("Please place goal color in front of the right camera, then press
ESC.\n");
speak_text("Put goal color in front of the right camera and press escape");
if(calibrate_camera(imgDeviationFromColor, color_avg_worldend) == -1)
return -1;
printf("GOAL CALIBRATION COMPLETE! \n");

//  cvReleaseImage(&imgDeviationFromColor);
//  cvDestroyWindow("imgDeviationFromColor");
cvDestroyWindow("img");

```

```

/* END INITIALIZATION STUFF */

printf("Starting to read in serial data.\n");
//speak_text("start");
while(!atgoal)
{
speak_text("Let's begin!");
printf("Reading sensors...");
    //wait for the sensor readings to start with a 0xFF
    input = 0;
    while(input != 0xFF)
        while(read(data_ioport, &input, 1) == 0);
// speak_text("sonar done");
    // start Q
    // Comment out following section, replace with different variable names
for sonars. Compass and CL stay the same, but aren't used.
/* //Full sensor reading from the Atmega128 consists of 8 bytes (7 sonars
+ compass)
    while(read(data_ioport, &BR , 1) == 0);
    while(read(data_ioport, &BL , 1) == 0);
    while(read(data_ioport, &CR , 1) == 0);
    while(read(data_ioport, &CL , 1) == 0);
    while(read(data_ioport, &TL , 1) == 0);
    while(read(data_ioport, &TR , 1) == 0);
    while(read(data_ioport, &BK , 1) == 0);
    while(read(data_ioport, &CMP, 1) == 0);
*/
    while(read(data_ioport, &SON56 , 1) == 0);
    while(read(data_ioport, &SON12 , 1) == 0);
    while(read(data_ioport, &SON0 , 1) == 0);
    while(read(data_ioport, &CL , 1) == 0);
    while(read(data_ioport, &SON3 , 1) == 0);
    while(read(data_ioport, &SON7 , 1) == 0);
    while(read(data_ioport, &SON4 , 1) == 0);
    while(read(data_ioport, &CMP , 1) == 0);
    // note that CL and CMP are not used right now.
    // end Q

    //read and process camera data
    goal_dist0      = classify_target(FRONT_CAMERA, NULL, color_avg_goal);
    worldend_dist0 = classify_target(FRONT_CAMERA, NULL,
color_avg_worldend);
    goal_dist1      = classify_target(LEFT_CAMERA , NULL, color_avg_goal);
    worldend_dist1 = classify_target(LEFT_CAMERA , NULL,
color_avg_worldend);
    goal_dist2      = classify_target(RIGHT_CAMERA, NULL, color_avg_goal);
    worldend_dist2 = classify_target(RIGHT_CAMERA, NULL,
color_avg_worldend);

    //ensure that camera data is not invalid
if(goal_dist0      == -1 ||
    worldend_dist0 == -1 ||
    goal_dist1      == -1 ||
    worldend_dist1 == -1 ||
    goal_dist2      == -1 ||
    worldend_dist2 == -1)

```

```

        return -1;

    // start Q
printf(" done!\n");
// speak_text("sensors read");
    // comment out these next two sections; they'll be coded right here.
/*    //send the data to lavi's computer
    while(write(lavi_ioport, &BR                , 1)==-1);
    while(write(lavi_ioport, &BL                , 1)==-1);
    while(write(lavi_ioport, &CR                , 1)==-1);
    while(write(lavi_ioport, &CL                , 1)==-1);
    while(write(lavi_ioport, &TL                , 1)==-1);
    while(write(lavi_ioport, &TR                , 1)==-1);
    while(write(lavi_ioport, &BK                , 1)==-1);
    while(write(lavi_ioport, &goal_dist0        , 1)==-1);
    while(write(lavi_ioport, &worldend_dist0    , 1)==-1);
    while(write(lavi_ioport, &goal_dist1        , 1)==-1);
    while(write(lavi_ioport, &worldend_dist1    , 1)==-1);
    while(write(lavi_ioport, &goal_dist2        , 1)==-1);
    while(write(lavi_ioport, &worldend_dist2    , 1)==-1);

    //read the command from lavi's computer
    while(read (lavi_ioport, &input, 1) == 0);
*/

double GreedyQ;
int GreedyAction;
double QTemp;
double QTempFocus;
int Action;
long QKeyword0;
long QKeywordi;

// Sensor names
int GoalCameraFront;
int GoalCameraLeft;
int GoalCameraRight;
int EoWCameraFront;
int EoWCameraLeft;
int EoWCameraRight;
int Sonar0;
int Sonar12;
int Sonar3;
int Sonar4;
int Sonar56;
int Sonar7;

// The three SeeGoals are used as shortcut checks
int SeeGoalFront = FALSE;
int SeeGoalLeft = FALSE;
int SeeGoalRight = FALSE;

// * next to parts that are done
// first convert sensor reading to bump/close/medium/far *
GoalCameraFront = SensorEncodeCamera(goal_dist0);
GoalCameraLeft = SensorEncodeCamera(goal_dist1);
GoalCameraRight = SensorEncodeCamera(goal_dist2);

```

```

EoWCameraFront = SensorEncodeCamera(worldend_dist0);
EoWCameraLeft = SensorEncodeCamera(worldend_dist1);
EoWCameraRight = SensorEncodeCamera(worldend_dist2);
Sonar0 = SensorEncodeSonar(SON0);
Sonar12 = SensorEncodeSonar(SON12);
Sonar3 = SensorEncodeSonar(SON3);
Sonar4 = SensorEncodeSonar(SON4);
Sonar56 = SensorEncodeSonar(SON56);
Sonar7 = SensorEncodeSonar(SON7);

// Check the SeeGoals
if (GoalCameraFront != Dist_Far)
{
    SeeGoalFront = TRUE;
}
else
{
    SeeGoalFront = FALSE;
}
if (GoalCameraLeft != Dist_Far)
{
    SeeGoalLeft = TRUE;
}
else
{
    SeeGoalLeft = FALSE;
}
if (GoalCameraRight != Dist_Far)
{
    SeeGoalRight = TRUE;
}
else
{
    SeeGoalRight = FALSE;
}

// generate QKey0(State) *** whenever QKeyword is generated, also
compare to other focus settings, since Koolio camera does not have focus
settings
// *** make sure to check that new QKey exists,
since some combinations of focus settings might not
// *** maybe only check this when goal is seen *
// All the below assumes that the goal can only be seen in at most one
camera at a time. This is a good assumption since the viewing angles of the
cameras do not overlap. If wider angle cameras are ever used, this will
become much more complicated!
// When vision code is updated to also find the focus in the camera,
all these checks won't be needed anymore.
printf("Deciding action...");
// speak_text("deciding action");
if (SeeGoalFront == TRUE)
{
    QKeyword0 =
GenerateQKeyword(GoalCameraFront, GoalCameraRight, GoalCameraLeft, EoWCameraFron
t, EoWCameraRight, EoWCameraLeft, Sonar0, Sonar12, Sonar3, Sonar4, Sonar56, Sonar7, Ca
meraFocusCenter, CameraFocusNone, CameraFocusNone, 0);
    GreedyQ = QSearch(QKeyword0);
}

```

```

        QKeyword0 =
GenerateQKeyword(GoalCameraFront, GoalCameraRight, GoalCameraLeft, EoWCameraFront,
EoWCameraRight, EoWCameraLeft, Sonar0, Sonar12, Sonar3, Sonar4, Sonar56, Sonar7, CameraFocusRight,
CameraFocusNone, CameraFocusNone, 0);
        QTempFocus = QSearch(QKeyword0);
        if (GreedyQ < QTempFocus)
        {
                GreedyQ = QTempFocus;
        }
        QKeyword0 =
GenerateQKeyword(GoalCameraFront, GoalCameraRight, GoalCameraLeft, EoWCameraFront,
EoWCameraRight, EoWCameraLeft, Sonar0, Sonar12, Sonar3, Sonar4, Sonar56, Sonar7, CameraFocusLeft,
CameraFocusNone, CameraFocusNone, 0);
        QTempFocus = QSearch(QKeyword0);
        if (GreedyQ < QTempFocus)
        {
                GreedyQ = QTempFocus;
        }
}
else if (SeeGoalRight == TRUE)
{
        QKeyword0 =
GenerateQKeyword(GoalCameraFront, GoalCameraRight, GoalCameraLeft, EoWCameraFront,
EoWCameraRight, EoWCameraLeft, Sonar0, Sonar12, Sonar3, Sonar4, Sonar56, Sonar7, CameraFocusNone,
CameraFocusCenter, CameraFocusNone, 0);
        GreedyQ = QSearch(QKeyword0);
        QKeyword0 =
GenerateQKeyword(GoalCameraFront, GoalCameraRight, GoalCameraLeft, EoWCameraFront,
EoWCameraRight, EoWCameraLeft, Sonar0, Sonar12, Sonar3, Sonar4, Sonar56, Sonar7, CameraFocusNone,
CameraFocusRight, CameraFocusNone, 0);
        QTempFocus = QSearch(QKeyword0);
        if (GreedyQ < QTempFocus)
        {
                GreedyQ = QTempFocus;
        }
        QKeyword0 =
GenerateQKeyword(GoalCameraFront, GoalCameraRight, GoalCameraLeft, EoWCameraFront,
EoWCameraRight, EoWCameraLeft, Sonar0, Sonar12, Sonar3, Sonar4, Sonar56, Sonar7, CameraFocusNone,
CameraFocusLeft, CameraFocusNone, 0);
        QTempFocus = QSearch(QKeyword0);
        if (GreedyQ < QTempFocus)
        {
                GreedyQ = QTempFocus;
        }
}
else if (SeeGoalLeft == TRUE)
{
        QKeyword0 =
GenerateQKeyword(GoalCameraFront, GoalCameraRight, GoalCameraLeft, EoWCameraFront,
EoWCameraRight, EoWCameraLeft, Sonar0, Sonar12, Sonar3, Sonar4, Sonar56, Sonar7, CameraFocusNone,
CameraFocusNone, CameraFocusCenter, 0);
        GreedyQ = QSearch(QKeyword0);
        QKeyword0 =
GenerateQKeyword(GoalCameraFront, GoalCameraRight, GoalCameraLeft, EoWCameraFront,
EoWCameraRight, EoWCameraLeft, Sonar0, Sonar12, Sonar3, Sonar4, Sonar56, Sonar7, CameraFocusNone,
CameraFocusNone, CameraFocusRight, 0);
        QTempFocus = QSearch(QKeyword0);

```

```

        if (GreedyQ < QTempFocus)
        {
            GreedyQ = QTempFocus;
        }
        QKeyword0 =
GenerateQKeyword(GoalCameraFront, GoalCameraRight, GoalCameraLeft, EoWCameraFron
t, EoWCameraRight, EoWCameraLeft, Sonar0, Sonar12, Sonar3, Sonar4, Sonar56, Sonar7, Ca
meraFocusNone, CameraFocusNone, CameraFocusLeft, 0);
        QTempFocus = QSearch(QKeyword0);
        if (GreedyQ < QTempFocus)
        {
            GreedyQ = QTempFocus;
        }
    }
else
{
    QKeyword0 =
GenerateQKeyword(GoalCameraFront, GoalCameraRight, GoalCameraLeft, EoWCameraFron
t, EoWCameraRight, EoWCameraLeft, Sonar0, Sonar12, Sonar3, Sonar4, Sonar56, Sonar7, Ca
meraFocusNone, CameraFocusNone, CameraFocusNone, 0);
    GreedyQ = QSearch(QKeyword0);
}
GreedyAction = 0;

// for i = 1 to NumActions *
for(i = 1; i < NumActions; i++)
{
    // generate QTemp = QKeyi(State)
    if (SeeGoalFront == TRUE)
    {
        QKeywordi =
GenerateQKeyword(GoalCameraFront, GoalCameraRight, GoalCameraLeft, EoWCameraFron
t, EoWCameraRight, EoWCameraLeft, Sonar0, Sonar12, Sonar3, Sonar4, Sonar56, Sonar7, Ca
meraFocusCenter, CameraFocusNone, CameraFocusNone, i);
        QTemp = QSearch(QKeywordi);
        QKeywordi =
GenerateQKeyword(GoalCameraFront, GoalCameraRight, GoalCameraLeft, EoWCameraFron
t, EoWCameraRight, EoWCameraLeft, Sonar0, Sonar12, Sonar3, Sonar4, Sonar56, Sonar7, Ca
meraFocusRight, CameraFocusNone, CameraFocusNone, i);
        QTempFocus = QSearch(QKeywordi);
        if (QTemp < QTempFocus)
        {
            QTemp = QTempFocus;
        }
        QKeywordi =
GenerateQKeyword(GoalCameraFront, GoalCameraRight, GoalCameraLeft, EoWCameraFron
t, EoWCameraRight, EoWCameraLeft, Sonar0, Sonar12, Sonar3, Sonar4, Sonar56, Sonar7, Ca
meraFocusLeft, CameraFocusNone, CameraFocusNone, i);
        QTempFocus = QSearch(QKeywordi);
        if (QTemp < QTempFocus)
        {
            QTemp = QTempFocus;
        }
    }
else if (SeeGoalRight == TRUE)
{

```

```

        QKeywordi =
GenerateQKeyword(GoalCameraFront,GoalCameraRight,GoalCameraLeft,EoWCameraFron
t,EoWCameraRight,EoWCameraLeft,Sonar0,Sonar12,Sonar3,Sonar4,Sonar56,Sonar7,Ca
meraFocusNone,CameraFocusCenter,CameraFocusNone,i);
        QTemp = QSearch(QKeywordi);
        QKeywordi =
GenerateQKeyword(GoalCameraFront,GoalCameraRight,GoalCameraLeft,EoWCameraFron
t,EoWCameraRight,EoWCameraLeft,Sonar0,Sonar12,Sonar3,Sonar4,Sonar56,Sonar7,Ca
meraFocusNone,CameraFocusRight,CameraFocusNone,i);
        QTempFocus = QSearch(QKeywordi);
        if (QTemp<QTempFocus)
        {
                QTemp = QTempFocus;
        }
        QKeywordi =
GenerateQKeyword(GoalCameraFront,GoalCameraRight,GoalCameraLeft,EoWCameraFron
t,EoWCameraRight,EoWCameraLeft,Sonar0,Sonar12,Sonar3,Sonar4,Sonar56,Sonar7,Ca
meraFocusNone,CameraFocusLeft,CameraFocusNone,i);
        QTempFocus = QSearch(QKeywordi);
        if (QTemp<QTempFocus)
        {
                QTemp = QTempFocus;
        }
}
else if (SeeGoalLeft == TRUE)
{
        QKeywordi =
GenerateQKeyword(GoalCameraFront,GoalCameraRight,GoalCameraLeft,EoWCameraFron
t,EoWCameraRight,EoWCameraLeft,Sonar0,Sonar12,Sonar3,Sonar4,Sonar56,Sonar7,Ca
meraFocusNone,CameraFocusNone,CameraFocusCenter,i);
        QTemp = QSearch(QKeywordi);
        QKeywordi =
GenerateQKeyword(GoalCameraFront,GoalCameraRight,GoalCameraLeft,EoWCameraFron
t,EoWCameraRight,EoWCameraLeft,Sonar0,Sonar12,Sonar3,Sonar4,Sonar56,Sonar7,Ca
meraFocusNone,CameraFocusNone,CameraFocusRight,i);
        QTempFocus = QSearch(QKeywordi);
        if (QTemp<QTempFocus)
        {
                QTemp = QTempFocus;
        }
        QKeywordi =
GenerateQKeyword(GoalCameraFront,GoalCameraRight,GoalCameraLeft,EoWCameraFron
t,EoWCameraRight,EoWCameraLeft,Sonar0,Sonar12,Sonar3,Sonar4,Sonar56,Sonar7,Ca
meraFocusNone,CameraFocusNone,CameraFocusLeft,i);
        QTempFocus = QSearch(QKeywordi);
        if (QTemp<QTempFocus)
        {
                QTemp = QTempFocus;
        }
}
else
{
        QKeywordi =
GenerateQKeyword(GoalCameraFront,GoalCameraRight,GoalCameraLeft,EoWCameraFron
t,EoWCameraRight,EoWCameraLeft,Sonar0,Sonar12,Sonar3,Sonar4,Sonar56,Sonar7,Ca
meraFocusNone,CameraFocusNone,CameraFocusNone,i);
        QTemp = QSearch(QKeywordi);

```

```

    }
    // if GreedyQ<QTemp *
    if (GreedyQ<QTemp)
    {
        // GreedyQ=QTemp *
        GreedyQ = QTemp;
        // GreedyAction=i *
        GreedyAction = i;
    }
}
// Action=GreedyAction *
Action = GreedyAction;
// Check for goal
if (GreedyQ>1000)
{
    atgoal=1;
    input = MOTOR_STOP;
    printf("GOAL FOUND");
    speak_text("Goal found!");
}
// Translate the selected Action to &input
else if ((Action==Action_Forward)&&(GreedyQ== -9999))
{
    input = MOTOR_STOP;
    printf("Q Value not found");
    speak_text("Q value not found");
}
else

    if (Action==Action_Forward)
    {
        input = MOTOR_FORWARD;
        printf("FORWARD\n");
        speak_text("Forward!");
    }
else if (Action==Action_Reverse)
{
    input = MOTOR_REVERSE;
    printf("REVERSE\n");
    speak_text("Reverse!");
}
else if (Action==Action_Rotate_Left)
{
    input = MOTOR_LEFT;
    printf("LEFT\n");
    speak_text("Left!");
}
else if (Action==Action_Rotate_Right)
{
    input = MOTOR_RIGHT;
    printf("RIGHT\n");
    speak_text("Right!");
}
else if (Action==Action_Stop)
{
    input = MOTOR_STOP;
    printf("STOP\n");
}

```

```

        speak_text("Stop!");
    }
    else if (Action==Action_Big_Rotate_Left)
    {
        input = MOTOR_LEFT;
        printf("LEFT\n");
        speak_text("Big left!");
    }
    else if (Action==Action_Big_Rotate_Right)
    {
        input = MOTOR_RIGHT;
        printf("RIGHT\n");
        speak_text("Big right!");
    }
    else
    {
        input = MOTOR_FORWARD;
        printf("DEFAULT\n");
        speak_text("Dunno!");
    }
    // end Q

    //send the command to the Atmega128
    while(write(data_ioport, &input, 1) == -1);
}

// start Q
// after loop breaks, stop the motors.
input = MOTOR_STOP;
while(write(data_ioport, &input, 1) == -1);
// end Q

// This is for ordering website:
/* while(1)
{
    // Set the size of the in-out parameter
    cliAddrLen = sizeof(echoClntAddr);

    // Block until receive message from a client
    if ((recvMsgSize = recvfrom(sock, echoBuffer, ECHOMAX, 0,
        (struct sockaddr *) &echoClntAddr, &cliAddrLen)) < 0)
    {
        printf("error");
        ingnore_next_msg = 1;
    }

    if(ingnore_next_msg == 0)
    {
        //printf("Handling client %s\n",
inet_ntoa(echoClntAddr.sin_addr));
        //printf("Received: %s\n", echoBuffer); // Print the data
received

        //change from char to number
        switch(echoBuffer[0])
        {

```

```

        case 'a':msg_num = ARROYO;
            break;
        case 'b':msg_num = GO_BACKWARD;
            break;
        case 'f':msg_num = GO_FORWARD;
            break;
    case 'l':msg_num = GO_LEFT;
        break;
    case 'r':msg_num = GO_RIGHT;
        break;
    case 's':if (echoBuffer[1] == 'c')
        msg_num = GO_SCHWARTZ;
        else if(echoBuffer[1] == 't')
            msg_num = STOP;
        else
        {
            printf("error does not match any cases");
            msg_num = 0; //0 is an error;
        }
        break;
    case '$':for(i = 0;i < 200;i++)
        say_it[i] = ' ';
        for(i = 3;i < (int)echoBuffer[1] + 3;i++)
            say_it[i - 3] = echoBuffer[i];
        say_it[i - 2] = 0;
        printf(say_it);
        printf("\n");
        speak_text(say_it);
        msg_num = -1;
        break;

        default: printf("error does not match any cases");
            msg_num = 0; //0 is an error;
};
if (msg_num >= 0)
    printf("the number is %d\n", msg_num); // Print the
command being sent

    while(write(data_ioport, &msg_num, 1)==-1);

} // Ends if(ignore_next_msg == 0)

else
{
    ignore_next_msg = 0;
}

    update_display(); // Updates Koolio's face

}*/ // Ends main while(1)

//Close serial port at program end
close(data_ioport);
close(lavi_ioport);

return 0;

```

```

} // end of main()

//=====
===
// open_serial_port()
=====
//=====
===
/*!
 * Opens a serial port with the flags:
 * O_RDWR - read/write mode
 * O_NOCTTY - not the controlling terminal
 * O_NDELAY - do not pay attention to Data Carrier Detect (DCD)
 * \param port a character string of the device
 * \param baud speed to open port
 * \return file descriptor
 */
int open_serial_port(char *port, int baud){
    int fd;
    int speed;
    char msg[70];
    struct termios opts;

    fd = open( port, O_RDWR | O_NOCTTY | O_NDELAY ); //may not need the nodelay
    if( fd <= 0 ){
        sprintf(msg,"Failed: %s:%d - %s",port,baud,strerror(errno));
        perror(msg);
        return(-1);
    }

    // Set the baud rate
    switch(baud){
        case 1200:
            speed = B1200; break;
        case 2400:
            speed = B2400; break;
        case 4800:
            speed = B4800; break;
        case 9600:
            speed = B9600; break;
        case 19200:
            speed = B19200; break;
        case 38400:
            speed = B38400; break;
        case 57600:
            speed = B57600; break;
        default:
            printf("Failed: Invalid speed\n");
    }
}
/*
Contents of termios struct:
    tcflag_t c_iflag;
    tcflag_t c_oflag;
    tcflag_t c_cflag;
    tcflag_t c_lflag;
    cc_t c_cc[NCCS];

```

```

        speed_t c_ispeed;
        speed_t c_ospeed;
*/
//Bzero call throws some kind of warning based on gcc not recognizing it
//so replaced called to bzero below with zero'ing c_ispeed and c_ospeed.
    //bzero( &opts, sizeof( opts ) );
    opts.c_ispeed = 0;
    opts.c_ospeed = 0;

    opts.c_cflag = speed | CS8 | CLOCAL | CREAD /*| CRTSCTS*/;
    opts.c_iflag = IGNPAR;
    opts.c_oflag = 0;
    opts.c_lflag = 0;
    opts.c_cc[VTIME] = 0;
    opts.c_cc[VMIN] = 0;

    tcflush( fd, TCIOFLUSH ); // clean out old crap

    tcsetattr( fd, TCSANOW, &opts );

    return fd;
} // end of open_serial_port()

/*****
7/26/06 Modified to be based on msg_num. Took out
count. No longer need it. Took out cp function.
*****/

//=====
// update_display()
//=====
/*
 * Updates the screen to reflect Koolio's actions
 */
void update_display()
{
    if(msg_num==3)
        //system("cp -f Left.jpg Current.jpg");
        system("qiv -Tfmi ~/Desktop/Serial/Left.jpg &");
    else if(msg_num==4)
        //system("cp -f Right.jpg Current.jpg");
        system("qiv -Tfmi ~/Desktop/Serial/Right.jpg &");
    else if(msg_num==5)
        //system("cp -f Reverse.jpg Current.jpg");
        system("qiv -Tfmi ~/Desktop/Serial/Reverse.jpg &");
    else if(msg_num==6)
        //system("cp -f Forward.jpg Current.jpg");
        system("qiv -Tfmi ~/Desktop/Serial/Forward.jpg &");
    else if(msg_num==7)
        //system("cp -f Stop.jpg Current.jpg");
        system("qiv -Tfmi ~/Desktop/Serial/Stop.jpg &");
    else

```

```

        system("qiv -Tfmi ~/Desktop/Serial/Forward.jpg &");
} // end of update_display()

//=====
===
// speak_text()
//=====
//=====
===
/*
 * Causes Koolio to speak the text string.
 */
void speak_text(char *text)
{
//int i;
//char *text = "hello world";

char festival[240] = "echo ";
//for(i = 0;i <= text_len;i++)
//    festival[i + 7] = text[i];
strcat(festival,text);
strcat(festival," | festival --tts &");

//system("echo '%s ' | festival --tts &", text);
system(festival);
} // end of speak_text()

//=====
===
// calibrate_camera()
//=====
//=====
===
/*
 * Takes in the imgDeviationFromColor image (can be NULL) and an array in
which
 * to put the RGB color average.
 *
 * Loops through each pixel in middle ninth of the source image and
 * calculates the average pixel RGB values.
 *
 * The imgDeviationFromColor receives a greyscale version of the image based
on
 * the deviation of the RGB values from the average.
 *
 * The color_avg[] argument is overwritten with the calculated average RGB
 * values.
 */
int calibrate_camera(IplImage *imgDeviationFromColor, long color_avg[3])
{
    unsigned int    ii, jj;
    unsigned char* ptrSource;
    unsigned char* ptrDev;
    long            pixel_diff[3];
    long            rms_diff;

```

```

CvCapture*    capture = 0;
IplImage*    frame   = 0;
IplImage*    src     = 0;

while(1)
{
    // Open camera for capturing frames:
    capture = cvCaptureFromCAM(RIGHT_CAMERA);
    if(!capture){fprintf(stderr, "ERROR: capture is NULL \n"); return -1;}
    frame = cvQueryFrame(capture);
    if(!frame ){fprintf(stderr, "ERROR: frame is null...\n"); return -1;}
    src = cvCloneImage(frame);

    color_avg[0] = 0;
    color_avg[2] = 0;
    color_avg[1] = 0;

    //initialize the imgDeviationFromColor image to all white
    // this makes the area around the middle always white -
    // since only the middle of the image shows the deviation
    if(imgDeviationFromColor != NULL)
    {
        ptrDev = (unsigned char*) imgDeviationFromColor->imageData;
        for(ii = 0; ii < imgDeviationFromColor->imageSize; ii++)
        {
            *(ptrDev) = 255;
            ptrDev++;
        }
    }

    // loop through each pixel in the middle ninth of the source image to
    retrieve
    // the average color values
    // i.e. if the image looks like this: ###
    //                                     #*#
    //                                     ### then analyze the *
    for(ii = 0; ii < src->height / 3; ii++)
    {
        ptrSource = (unsigned char*) src->imageData + 3 * (src->width * (src-
>height / 3 + ii) + src->width / 3);
        for(jj = 0; jj < src->width / 3; jj++)
        {
            color_avg[0] = color_avg[0] + ptrSource[0];
            color_avg[1] = color_avg[1] + ptrSource[1];
            color_avg[2] = color_avg[2] + ptrSource[2];
            ptrSource += 3;
        }
    }
    color_avg[0] = color_avg[0] / (src->imageSize / 3 / 9);
    color_avg[1] = color_avg[1] / (src->imageSize / 3 / 9);
    color_avg[2] = color_avg[2] / (src->imageSize / 3 / 9);

    // loop through each pixel in the middle ninth of the source image to
    generate
    // the average color values difference
    if(imgDeviationFromColor != NULL)
        for(ii = 0; ii < imgDeviationFromColor->height / 3; ii++)

```

```

    {
        ptrDev = (unsigned char*) imgDeviationFromColor->imageData +
        (imgDeviationFromColor->width * (imgDeviationFromColor->height / 3 + ii) +
        imgDeviationFromColor->width / 3);
        ptrSource = (unsigned char*) src->imageData + 3 * (src->width * (src-
        >height / 3 + ii) + src->width / 3);
        for(jj = 0; jj < imgDeviationFromColor->width / 3; jj++)
        {
            // converts to grayscale based on deviation from average pixel value
            // such that the darker the filtered image, the closer it is to the
            // desired color
            // computes deviation between the source pixel and the goal color. A
            // value of 0 means that the pixel is exactly the same color as the
            // goal color. Bigger or smaller numbers mean that the color is
            farther
            // from the goal color, but if all of the deviations are close to the
            // same value, then the pixel is probably the same color under
            different
            // lighting.
            pixel_diff[0] = ptrSource[0] - color_avg[0];
            pixel_diff[1] = ptrSource[1] - color_avg[1];
            pixel_diff[2] = ptrSource[2] - color_avg[2];

            // Figure out the minimum deviation from the desired color based on
            an
            // RMS approximation which compensates for lighting conditions.
            rms_diff = (2 * pixel_diff[0] - pixel_diff[1] - pixel_diff[2]) *
                (2 * pixel_diff[0] - pixel_diff[1] - pixel_diff[2]) +
                (2 * pixel_diff[1] - pixel_diff[0] - pixel_diff[2]) *
                (2 * pixel_diff[1] - pixel_diff[0] - pixel_diff[2]) +
                (2 * pixel_diff[2] - pixel_diff[0] - pixel_diff[1]) *
                (2 * pixel_diff[2] - pixel_diff[0] - pixel_diff[1]);
            if(rms_diff < SENSITIVITY / 10)
                *ptrDev = 255;
            else if(rms_diff < SENSITIVITY / 3)
                *ptrDev = 200;
            else if(rms_diff < SENSITIVITY)
                *ptrDev = 100;
            else
                *ptrDev = 0;
            ptrDev++;
            ptrSource += 3;
        }
    }

    // show the images if allowed
    if(imgDeviationFromColor != NULL)
    {
        cvShowImage("img", src);
        cvShowImage("imgDeviationFromColor", imgDeviationFromColor);
        printf("color_avg{%u, %u, %u}\n", color_avg[0], color_avg[1],
        color_avg[2]);
    }

    cvReleaseImage(&src);
    cvReleaseCapture(&capture);

```

```

    // breaks if ESC key is pressed, Key=0x10001B under OpenCV 0.9.7(linux
version),
    // remove higher bits using AND operator
    if( (cvWaitKey(10) & 0x0000ff) == 0x1b ) break;
} // Ends while(1)

return 1;
} // end of calibrate_camera()

//=====
===
// classify_target()
=====
//=====
===
/*
 * Takes in a source image, the address of the imgDeviationFromColor image
(can
 * be NULL) and the RGB color of the target rectangle.
 *
 * A pixel is defined as being of the same color as the target if each of its
 * RGB values differ from the RGB values of the target by a similar amount.
 * The technique used is an RMS best fit approximation of a linear
 * representation of each RGB value vs. light intensity (i.e. it finds the
 * color even if that color has more or less light hitting it than the target
 * had).
 *
 * The source image is unaltered, and the imgDeviationFromColor image holds
the
 * grey scale result. color_target[] is unaltered.
 */
short classify_target(int camera, IplImage *imgDeviationFromColor, long
color_target[3])
{
    CvCapture*    capture = 0;
    IplImage*     frame   = 0;
    IplImage*     src     = 0;
    unsigned int  ii, jj, kk;
    short         pixel_count      = 0;
    short         max_pixel_count = 0;
    unsigned char* ptrSource;
    unsigned char* ptrDev;
    long          pixel_diff[3];
    long          rms_diff;

    capture = cvCaptureFromCAM(camera);
    if(!capture){fprintf(stderr, "ERROR: capture is NULL \n"); return -1;}
    frame   = cvQueryFrame(capture);
    if(!frame ){fprintf(stderr, "ERROR: frame is NULL...\n"); return -1;}
    src     = cvCloneImage(frame);
    if(imgDeviationFromColor != NULL)
        cvShowImage("img", src);

    // loop through each pixel checking each row separately
    for(jj = 0; jj < src->width; jj++)
    {

```

```

ptrSource = (unsigned char*) src->imageData + 3 * jj;
for(ii = 0; ii < src->height; ii++)
{
    // computes deviation between the source pixel and the goal color. A
    // value of 0 means that the pixel is exactly the same color as the
    // goal color. Bigger or smaller numbers mean that the color is
farther
    // from the goal color, but if all of the deviations are close to the
    // same value, then the pixel is probably the same color under
different
    // lighting.
    pixel_diff[0] = ptrSource[0] - color_target[0];
    pixel_diff[1] = ptrSource[1] - color_target[1];
    pixel_diff[2] = ptrSource[2] - color_target[2];

    // Figure out the minimum deviation from the desired color based on an
    // RMS approximation which compensates for lighting conditions.
    rms_diff = (2 * pixel_diff[0] - pixel_diff[1] - pixel_diff[2]) *
                (2 * pixel_diff[0] - pixel_diff[1] - pixel_diff[2]) +
                (2 * pixel_diff[1] - pixel_diff[0] - pixel_diff[2]) *
                (2 * pixel_diff[1] - pixel_diff[0] - pixel_diff[2]) +
                (2 * pixel_diff[2] - pixel_diff[0] - pixel_diff[1]) *
                (2 * pixel_diff[2] - pixel_diff[0] - pixel_diff[1]);

    if(rms_diff < SENSITIVITY)
    {
        //if the pixel is the right color, change the pixel on the target to
GREY
        if(imgDeviationFromColor != NULL)
            *ptrDev = 200;
        if(++pixel_count > max_pixel_count)
        {
            max_pixel_count = pixel_count;

            //if the pixel is the right color and one that counts towards
            // max_pixel_count, change the pixel on the target to WHITE
            if(imgDeviationFromColor != NULL)
                for(kk = 0; kk < pixel_count; kk++)
                    *(ptrDev - kk * imgDeviationFromColor->width) = 255;
        }
    }
    else
    {
        //if the pixel is the wrong color, change the pixel on the target to
BLACK
        if(imgDeviationFromColor != NULL)
            *ptrDev = 0;
        pixel_count = 0;
    }

    // increment pointers
    ptrSource += 3 * src->width;
    if(imgDeviationFromColor != NULL)
        ptrDev += imgDeviationFromColor->width;
}
pixel_count = 0;
}

```

```

if(imgDeviationFromColor != NULL)
{
    cvShowImage("imgDeviationFromColor", imgDeviationFromColor);
    printf("%d\n", max_pixel_count);
}

cvReleaseCapture(&capture);
cvReleaseImage(&src);

return max_pixel_count;
} // end of classify_target()

// start Q
//
*****
long GenerateQKeyword(int GoalCam0, int GoalCam1, int GoalCam2, int EoWCam0,
int EoWCam1, int EoWCam2, int Son0, int Son12, int Son3, int Son4, int Son56,
int Son7, int GoalFoc0, int GoalFoc1, int GoalFoc2, int Action)
{ // This generates the Q Keyword
    long Keyword;

    Keyword = ((GoalFoc2 * pow(2, 31)) + (GoalFoc1 * pow(2, 29)) +
(GoalFoc0 * pow(2, 27)) + (GoalCam0 * pow(2, 25)) + (GoalCam1 * pow(2, 23)) +
(GoalCam2 * pow(2, 21)) + (EoWCam0 * pow(2, 19)) + (EoWCam1 * pow(2, 17)) +
(EoWCam2 * pow(2, 15)) + (Son0 * pow(2, 13)) + (Son12 * pow(2, 11)) + (Son3 *
pow(2, 9)) + (Son4 * pow(2, 7)) + (Son56 * pow(2, 5)) + (Son7 * pow(2, 3)) +
Action);
printf("GoalCam0: %d, Goalcam 1: %d, Goalcam 2:
%d\n", GoalCam0, GoalCam1, GoalCam2);
printf("EoWCam0: %d, EoWCam1: %d, EoWCam2: %d\n", EoWCam0, EoWCam1, EoWCam2);
printf("Sonar0: %d, Sonar1-2: %d, Sonar3: %d, Sonar4: %d, Sonar56: %d,
Sonar7: %d\n", Son0, Son12, Son3, Son4, Son56, Son7);
printf("*****\n");
    return Keyword;
}

//
*****
double QSearch(long Keyword)
{ // This searches the Q table for a keyword
    struct QEntry *CurrentQEntry;
    double TempQValue;

    CurrentQEntry = QTable;
    TempQValue = -9999;

    if (CurrentQEntry != 0)
    {
        while ((CurrentQEntry->NextQEntry != 0) && (TempQValue == -9999))
        { // until either the end of the QTable is reached or the entry
is found
            if (CurrentQEntry->QKey == Keyword)
            {
                TempQValue = CurrentQEntry->QValue;
            }
        }
    }
}

```

```

        }
        else
        {
            CurrentQEntry = CurrentQEntry->NextQEntry;
        }
    }
}
if (TempQValue != -9999)
{ printf("Found!\n");}

    return TempQValue;
}
//
*****
int SensorEncodeCamera(short CameraReading)
{
    int TempReading;
    if (CameraReading <= Camera_Close)
    {
        TempReading = Dist_Close;
    }
    else if (CameraReading <= Camera_Far)
    {
        TempReading = Dist_Medium;
    }
    else
    {
        TempReading = Dist_Far;
    }
    return TempReading;
}
//
*****
int SensorEncodeSonar(unsigned char SonarReading)
{
    int TempReading;
    if (SonarReading <= Sonar_Bump)
    {
        TempReading = Dist_Bump;
    }
    else if (SonarReading <= Sonar_Close)
    {
        TempReading = Dist_Close;
    }
    else if (SonarReading <= Sonar_Far)
    {
        TempReading = Dist_Medium;
    }
    else
    {
        TempReading = Dist_Far;
    }
    return TempReading;
}
}

```

//

LIST OF REFERENCES

1. Aljibury, H. (2001). "Improving the Performance of Q-Learning with Locally Weighted Regression." Masters Thesis, University of Florida.
2. Amrose A/S. Amrose Robotics, 2006. <http://www.amrose.dk/>.
3. Bagnell, J. A., Doty, K. L., Arroyo, A. A. (1998). "Comparison of Reinforcement Learning Techniques for Automatic Behavior Programming." Proceedings of the Conference on Automated Learning and Discovery. Pittsburgh, PA.
4. Balch, T. (1997). "Learning Roles: Behavior Diversity in Robot Teams." College of Computing Technical Report. Georgia Institute of Technology.
5. Balch, T. (1999). "Reward and Diversity in Multirobot Foraging." Workshop on Agents Learning About, From, and With Other Agents. Stockholm, Sweden.
6. Camelot. Camelot Robotics, 2006. <http://www.camelot.dk/>.
7. CMPS03 Documentation. <http://www.robot-electronics.co.uk/hfm/cms3doc.shtml>
8. Duan, Y. and Xu, X. (2005). "Fuzzy Reinforcement Learning and Its Application in Robot Navigation." Proceedings on the Fourth International Conference on Machine Learning and Cybernetics, pp. 899-904. Guangzhou, China.
9. Goldberg, K., Mirtich, B. V., Zhuang, Y., Craig, J., Carlisle, B. R., and Canny, J. (1999). "Part pose statistics: estimators and experiments." *IEEE Transactions on Robotics and Automation*, 15:849-857.
10. Jin, Z., Shima, T., Schumacher, C. J. (2006). "Optimal Scheduling for Refueling Multiple Autonomous Aerial Vehicles." *IEEE Transactions on Robotics*, 22:682-693.
11. Kaelbling, L. P. (1993). "Hierarchical Learning in Stochastic Domains: Preliminary Results." Proceedings of the Tenth International Conference on Machine Learning, pp. 167-173. San Mateo, CA.
12. Kaelbling, L. P., Littman, M. L., and Moore, A. W. (1996) "Reinforcement Learning: A Survey." *Journal of Artificial Intelligence Research*, 4:237-285.
13. MAVRIC-IIB Technical Manual. BD Micro. March 19, 2009. <http://www.bdmicro.com/mavric-iib/mavric-iib.pdf>.
14. McCallum, R. A. (1994). "Instance-Based State Identification for Reinforcement Learning." Neural Information Processing Systems 6. Denver, CO.
15. Mehrotra, M. and Grosky, W. I. (1989). "Shape Matching Utilizing Indexed Hypotheses Generation and testing." *IEEE Transactions on Robotics and Automation*, 5:70-77.
16. Mekatronix. Mekatronix, 2009. <http://www.mekatronix.com>.

17. Mirtich, B. and Canny, J. "Impulse-based dynamic simulation." In Goldberg, K., Halperin, D., Latombe, J. C., and Wilson, R., Eds. *The Algorithm Foundations of Robotics*. A.K. Peters, Boston, MA. 1995.
18. NOVA-7896 Datasheet. Tracan Electronics, 2004.
<http://www.tracan.com/products/pdetail.php?v=32&c=56&p=773&x=2>.
19. Player/Stage. The Player Project, 2008. <http://playerstage.sourceforge.net/>.
20. Ranganathan, A., Menegatti, E., and Dellaert, F. (2006). "Bayesian Inference in the Space of Topological Maps." *IEEE Transactions on Robotics*, 22:92-107.
21. Robot3D. Intermedia, 2005.
<http://www.intermedia.sa.it/lia/jroboop/doc/javax/robotics/engine/robots/Robot3D.html>.
22. Rummery, G. and Niranjana, M. (1994). "On-Line Q-Learning Using Connectionist Systems." Technical Report. Cambridge University Engineering Department.
23. S1 Optical Shaft Encoder Data Sheet. US Digital, 2009.
http://www.usdigital.com/assets/general/92_s1_datasheet_1.pdf.
24. Schaal, S. (1994). "Learning from Demonstration." *Control Systems Magazine*, pp. 57-71.
25. Sen, S. and Sekaran, M. (1996). "Multiagent Coordination with Learning Classifier Systems." *Adaptation and Learning in MultiAgent Systems, IJCAI'95 Workshop, Lecture Notes in Artificial Intelligence*, pp. 218-233. Springer, NY.
26. Singh, S., Jaakkola, T., Littman, M. L., and Szepesvári, C. (2000). "Convergence Results for Single-Step On-Policy Reinforcement-Learning Algorithms." *Machine Learning Journal*, 38:287-308.
27. SRF08 Ultra Sonic Range Finger Technical Specifications. Devantech, Ltd., 2001.
<http://www.robot-electronics.co.uk/html/srf08tech.shtml>.
28. Sutton, R. S. (1990). "Integrated Architectures for Learning, Planning, and Reacting Based on Approximate Dynamic Programming." Proceedings of the Seventh International Conference on Machine Learning, pp. 216-224. Palo Alto, CA.
29. Sutton, R. S. and Barto, A. G. Reinforcement Learning: An Introduction. MIT Press, Cambridge, MA. 1998.
30. TeamBots 2.0. TeamBots, 2000. <http://www.cs.cmu.edu/~trb/TeamBots/>.
31. Thrun, S. and Schwartz, A. (1993). "Issues in Using Function Approximation for Reinforcement Learning." Proceedings of the Fourth Connectionist Models Summer School. Hillsdale, NJ.

32. Video Blaster WebCam II User Guide. Creative Labs, Inc., 1998.
http://www.mil.ufl.edu/projects/koolio/Koolio/Datasheets/WCIIUSB_Parallel.pdf.
33. Yamaguchi, T., Masabuchi, M., Fujihara, K., and Yachida, M. (1996). "Propagating Learned Behaviors from a Virtual Agent to a Physical Robot in Reinforcement Learning." Proceedings of IEEE International Conference on Evolutionary Computation, pp. 855-859. Nagoya, Japan.
34. Yamaguchi, T., Masabuchi, M., Fujihara, K., and Yachida, M. (1996). "Realtime Reinforcement Learning for a Real Robot in the Real Environment." Proceedings of the 1996 IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 1321-1328. Osaka, Japan.

BIOGRAPHICAL SKETCH

Lavi Zamstein was born in Chicago, Illinois, and was raised in South Florida. He attended Sar Shalom Hebrew Academy, a Messianic Jewish private school, through eighth grade. In high school, he was in the International Baccalaureate program at Boyd H. Anderson High School in Lauderdale Lakes, where his interest in computers and engineering grew.

He graduated with honors with his bachelor's degree in computer engineering from Tulane University in 2001. That fall, he began graduate school at the University of Florida and earned his master's in electrical engineering in 2003. He earned his PhD in electrical engineering in 2009.