CMPU-250: Final Project Write-up

Autonomous Robotics Design Competition Simulation

George Witteman

May 2018

Abstract

For my final project I chose to design and implement an agent-based model of Vassar's Autonomous Robotics Design Competition class in the Cognitive Science department. This model simulates two robots in the arena using a slightly modified version of the rules. The model is validated by testing individual functions for correctness, as well as visually testing that the behavior is correct. Finally, the model is evaluated using hypothesis testing to test assumptions about the competition, such that the addition of a camera will improve the efficiency of the robots or that increasing the velocity or rotation speed of the robot will increase it's score.

1 Introduction

Vassar's Autonomous Robotics Design Competition is a class in the Cognitive Science department that gives students the opportunity to create robots designed to complete a specific task. In the course, students work in teams and, using Arduinos, foam core, 3D printers, Pixy camera, and various other electronics, build a robot that competes with the other team's robot to fulfill a certain task as best as possible within a given time frame.

This year, the goal of the competition is for teams to create a robot that gains the best score by collecting blocks of it's own quadrant's color into it's home quadrant, and removing blocks of other colors from it's home quadrant. There is also a sabotage element that can be implemented by moving blocks that are not the opponents color into the opponents quadrant.

During the actual competition, three of the four teams used a strategy that collected one block at a time and moved it to an appropriate quadrant. My team decided that a collection strategy would prove more beneficial, since it could rely less heavily on the camera if that proved to not be easy to implement. Since we were not able to fully finish completing our robot, I wanted to build this model in order to test some of the theories we had about why our robot would perform well. I tried to build this model as closely as possible to the version that we created, including the camera function.

This model includes a visualization of the arena that can be seen over time. See Figure 1 for an example of this. In this figure, you're able to see the arena including the outer and quadrant boundary lines, blocks, and robots. The two robots are represented by rectangles with a triangle in the middle indicating their direction. The robots camera is also represented by drawing it's field of view as a mostly transparent semi-circle in front of the robot. The time that the simulation has been running for is indicated in the title of the plot, and when the simulation finishes the points for each team and winner is displayed in the MATLAB command window.

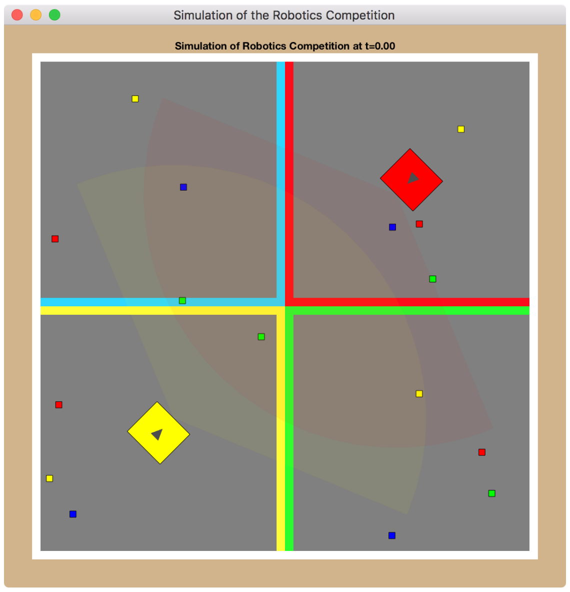
Simulation of the Robotics Competition
Simulation of Robotics Competition at t=0.00

Figure 1: An example starting simulation state. You can see two robots (red and yellow) and twelve blocks spread out evenly through the four quadrants. The direction the robot is facing as well as it's camera's field of view is also represented.

2 Simplifying Assumptions

- Robots can reach their full velocity and come to a stop instantly.
- All robots are identical in their software and hardware implementations.
- Blocks cannot be pushed, only collected or ignored.
- Placing blocks into the quadrant of the opponent or removing other color blocks from the home quadrant is insignificant. That is, being able to collect home color blocks and bringing them back to the home quadrant is the most important thing.
- The robot's cameras are perfect accurate within their field of view.
- The sorting mechanism is perfect. That is, blocks are either completely ignored (not moved at all) or collected into the chamber with 100% accuracy.
- The rotation of the blocks is insignificant.

- Robots can run over each when backing up, but not when going forward. I chose to make this assumption in order to prevent bots from getting stuck when they are backing up. This would mostly happen when dt is low, and does not seem to impact the results of a simulation.

3 Code Validation

There were two main strategies that I used to validate my code. First, I tested individual functions by manually building instances of the classes and running tests on specific functions. Second, I tested the system as a whole by running multiple random simulations using the same and differing parameters and looking at the results to make sure that things were running as expected.

3.1 Individual Function Tests

My first strategy for testing my code was to test individual methods by creating test instances of classes, manually setting parameters, and then testing the output of the methods using inputs where I had manually calculated the output.

For example, I validated the validity of the method `inFieldOfView` on the `Camera` class by creating a new instance of the `Camera` in the command window with the command `c = Camera(5, 135, 'red')`. Then, I tested the function with a number of different points. For example, the camera should be able to see a block 1 foot in front of it, so to test that I used the command `c.inFieldOfView(0,0,0,1,0)` which returned `1`, the expected output. Another test that I ran is for blocks that should not be in the field of view. One command I used to test that was `c.inFieldOfView(0,0,0,-1,0)` which returns `0`, since the block is behind the camera position.

I did a variety of similar tests on many methods as I was developing. I would check that the results were as expected, but I would also check to make sure error cases were also handled in ways that made sense. For example when testing the `Camera.getAngleOffCenter()` method (see Listing 3.2), I tested that the function worked for values behind the camera as well as in front of it.

3.2 Visual System Tests

The second strategy that I used to validate my code's accuracy was by doing visual tests. After developing individual class methods (or even just small blocks of code within a method), I would run the simulation as a whole to see the result.

One place where I used this while developing was the `Camera.getAngleOffCenter()` method (see Listing 3.2). This method is supposed to find the angle that the input point is off the center axis of the camera. In order to find this angle it creates two lines. One line is from the camera origin to 1 unit straight forward, and the other line is from the camera origin to the point of the object. We then calculate the angle between those two lines.

When developing this function, I wanted to make sure that the lines were accurate, since I was using some trigonometry in order to calculate a point. In order to do this, I added a debug flag (`Camera.Debug`) to the constants for the `Camera` class, and then plotted the two lines if the flag is set to `true`. I could then also print out the angle that is getting returned and do a visual check to make sure that the angle matches what I would visually expect for the two lines. This allowed me to catch a subtle error which resulted in an error checking case being added to the function (see lines 17-20 in Listing 3.2).

Another example where I used visual testing was in the block collection code. After adding code to check collect blocks as a robot runs over it, I confirmed the validity of this feature by adding code to display the blocks that a robot has in it's chamber, and testing that when a block got added to the chamber, it was also removed from the arena.

```
 1 function ang = getAngleOffCenter(cam_x, cam_y, cam_R, obj_x, obj_y)
 2   %GETANGLEOFFCENTER
 3   % Calculate the angle of the object relative to the center
 4   % axis of the camera.
 5   cam_x2 = cos(deg2rad(cam_R)) + cam_x;
 6   cam_y2 = sin(deg2rad(cam_R)) + cam_y;
 7 
 8   if Camera.Debug
 9     plot([cam_x cam_x2], [cam_y cam_y2]);
10     plot([cam_x obj_x], [cam_y obj_y]);
11   end
12 
13   % Find the angle between the two lines created by the points
14   ang = rad2deg(atan2(obj_y-cam_y, obj_x-cam_x) - ...
15     atan2(cam_y2-cam_y,cam_x2-cam_x));
16 
17   % Correct for error
18   if ang > 180
19     ang = ang - 360;
20   end
21 end
```

Listing 1: The `Camera.getAngleOffCenter()` method.

4 Hypothesis Tests

The following section describes some different hypothesis that I was able to test using the model.

4.1 Camera Effectiveness

Hypothesis: The addition of the camera allows robots to collect more blocks in a shorter amount of time. In general, the addition of the camera will increase the score for the robot as opposed to roaming randomly.

To test this hypothesis I ran a 5 simulations of the competition with the camera and without the camera and averaged the resulting scores for each robot. The average score across both robots in the simulations without the camera is 6, and the average score across both robots in the simulations *with* the camera is 10. That means that robots with the camera scored the maximum amount of points possible ($(3 \times 10) - (2 \times 1) = 10$) in each round. This shows that there is a significant advantage to having the camera.

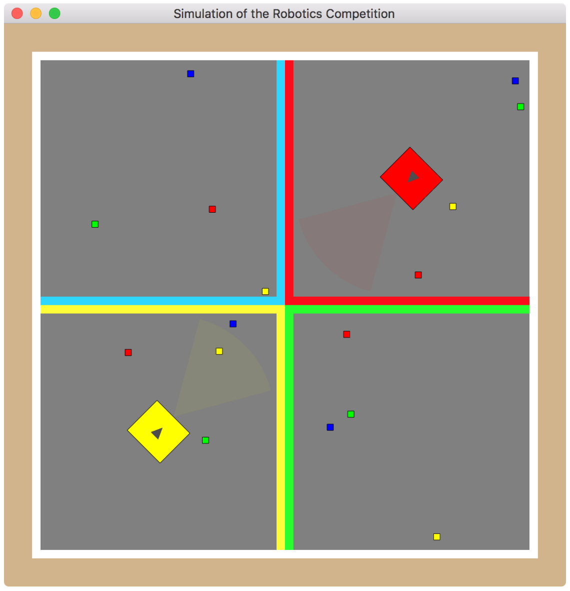

Figure 2: A screenshot at $t = 0$ of the arena where both camera's are configured with a maximum view angle of 60° and maximum viewing distance of 3 ft.

4.2 Camera Field of View

Hypothesis: Because it is able to see more blocks on the arena, a camera with a wider and/or longer field of view will get a higher average score than one with a smaller and/or shorter field of view.

To test this hypothesis I ran 5 simulations for each of the following configurations: a 115° 3 ft. field of view, a 90° 3 ft. field of view, a 60° 3 ft. field of view, and a 60° 2 ft. field of view. The average score for each robot for each configuration, as well as the average score without the camera, is as follows.

- 115° 3 ft. FOV: $(10 + 10 + 10 + 10 + 10 + 10 + 10 + 10 + 10 + 10)/10 = \mathbf{10}$
- 90° 3 ft. FOV: $(10 + 9 + 7 + 10 + 10 + 10 + 4 + 10 + 10 + 10)/10 = \mathbf{9}$
- 60° 3 ft. FOV: $(7 + 10 + 10 + 6 + 10 + 7 + 7 + 7 + 7 + 10)/10 = \mathbf{8.1}$
- 60° 2 ft. FOV: $(7 + 10 + 7 - 2 + 9 + 3 + 4 - 2 + 10 + 4)/10 = \mathbf{5}$
- No camera: $(10 + 9 + 7 - 2 + 1 + 7 + 4 + 9 + 9 + 7)/10 = \mathbf{6}$

As you can see, the camera had a significant impact at higher angles and distances as expected. However, when the camera got down to 60° 2 ft., it actually performed worse than the runs without any camera.

One explanation for why the camera with this configuration did not perform better than the robot without the camera, is that the camera's field of view is almost entirely just directly in front of the robot. That is, if the robot was in a position where the block could be seen by the camera, the robot only had to drive forward to collect the block anyway. This can be seen in Figure 2.

Therefore, this evidence shows that the hypothesis is correct, that robots with a larger field of view are more likely to gain a higher score.

4.3 Spin Angle

Hypothesis: The larger the range of spin angles that the robot is able to spin after hitting an obstacle, the more paths that it will be able to take, and the better score it would get.

This hypothesis is not supported by evidence. I turned the camera off and ran 5 simulations each with the following configurations, shown in tuples representing the minimum and maximum spin angle: (45°, 180°), (0°, 180°), (0°, 90°), and (90°, 180°). The corresponding average scores for these configurations is 4.5, 4.1, 4.6, and 5 respectively. Since all of these values are within the range of 4 to 5 points, it doesn't seem that there is any advantage to having the robots spin with anything other than 90 to 180 degrees.

One type of configuration that I did not test was where the robot always spun at a specific angle (their min and max spin angle is equal). Since I had to use a low value for dt in order to run the simulations in a reasonable amount of time, the robots were not able to spin at exactly the angle that they were supposed to. This would defeat the purpose of testing the spin angle.