# Extension of SCS Platform to Run Robotito

David Ehrenhaft

University of South Florida

## Introduction

This report details the work done to extend the SCS platform to control a physical robot, including the precise means by which the robot and software are interfaced. The report will also explain the issues discovered during testing, and suggested improvements to improve overall capability and efficiency of future experiments.

## Objective

SCS v2 branch was extended to integrate the Robotito robot design as a valid controllable robot. This allows the lab to run completed cognitive mapping models on a physical robot, as opposed to relying exclusively on virtual robots.

## Integration Details

The Spatial Cognition Simulator (SCS) is an application designed to test cognitive mapping models onto multiple different types of robots, both virtual and real. As such, SCS is effectively an interface between any such model and any compatible robot. The goal for this project was to extend the code to enable a real robot to be used by the system, as all previously compatible robots were virtual.

Many basic components of this integration already existed, but not in a useable form. The work performed consisted of revising code to function as intended (where it was crashing or otherwise ineffective) and adding a significant volume of code to interface the Robotito's firmware with the SCS platform, including changes to pose detection, wall detection, wall collision avoidance, action affordance calculation, platform placement, and motion control.

All of the testing performed used the Multiscale Actor Critic Morris Maze model, using exclusively platforms as the goal. As such, the robot should work with minimal or no changes for other affordance-based models. It may require more significant work where action selection is not handled by affordances.

### Localization and Communication

The Robotito is equipped with an Xbee communication module – a wireless packet sender and receiver that runs using a Zigbee-based architecture. The firmware for the Robotito waits for control signals to engage or release the motors, to modify k values, and to set the current velocity. Although the Robotito sends back packets itself, the contents of these packets are not currently used by the SCS receiver.

The Robotito is outfitted with a dozen IR distance sensors, but none of these are used for localization in the SCS system. In fact, the Robotito does not have any awareness of its own position, the position of the goal, the position of walls, or any other local information. It simply follows the control signals sent by the SCS plataform, without any modification or independent action taking.

As for SCS, it is aware of the position of the Robotito using a global camera, which must be set to see the entire maze. The camera system runs independently using ROS, specifically 4 major nodes – one to run the camera, one to parse the video stream for the position and orientation of multiple Alvar AR markers, and a pair to filter the positions of markers onto a stream that the SCS threads can access.

Within the SCS application, a pair of listeners read the robot pose and wall rostopic data streams to update the universe in simulation. The robot pose listener simply waits for any updates to the marker representing value 0, using the new pose of the marker to update the robot's position and orientation. The wall listener thread uses the markers' values to identify the type of object (values 1-4 marks the maze's outer walls, 5-10 marks internal walls), and uses constants to set the size of the object. The object is placed such that the marker is the middle of the object, and for walls and obstacles, this also includes an additional, parallel line segment a constant distance away.

## Flow of execution

There are 4 major components to running the Robotito using SCS:

1) The SCS application, which runs the cognitive model and selects the next action for the robot to take.
2) The Robotito firmware, which uses control signals from SCS to move the robot. It is connected to SCS platform via a control packets through a pair of Xbee modules.
3) A Stingray camera, which sends video streams to a AR marker detector for processing; the resulting stream is sent to the pose detector and wall detector ROS nodes.
4) The pose detector and wall detector ROS nodes, which filter the results from the AR detector and send a stream of the position of the respective objects to SCS via rostopic data streams.

All components must be set up before the SCS can be run, and each runs independently of each other. SCS is aware of whether the Robotito is receiving packets, but otherwise no component is aware of whether the others are actually running or correct. The general flow of execution proceeds as follows:

1) The Stingray camera begins recording a video stream and send the frames to the Alvar AR detector ROS node.
2) Once the AR detector node identifies the positions of all markers in the frames, it sends them to the pose detector and wall detector.
3) The pose detector and wall detector filter the data if needed (currently, they don't do anything to the data. These were originally designed to combine data from two camera, but due to errors in the code, a simplified version is currently used), then send the data to SCS.
4) A listening thread in SCS read in the pose and orientation of the robot. Another listening thread reads in any wall markers and generates a wall with a constant size such that the marker is placed at the center of the wall.
5) SCS determines an action to perform based on the Robotito's position, orientation, proximity to walls, and the learned policy, then selects an action and sends a control signal to the Robotito firmware using its Xbee module. The main SCS thread waits until the action is completed to continue.
6) The Robotito responds to the control signals. In the current design, SCS will send a signal to move at a desired velocity, then sends a stop signal after the estimated time-to-completion is reached.

It is important to note that (5) and (6) occur in every cycle in SCS, but the ROS interactions occur independently, and there are no stalling operations to ensure that the robot pose has been updated before deciding the next action. Thus, it is possible that the robot's position was not updated for a given cycle. Additionally, the simulator does not update the robot pose or orientation in any other way than by the pose detector data stream. It does not predict the robot's next position based on the control signal given.

## Robot motion

SCS itself runs in a series of cycles. For each cycle, there is a main thread of execution that updates the simulator and ends when an action for the robot is selected and performed. These cycles take roughly 200-300 milliseconds altogether, depending on the motion speed (as defined by constants), the size of steps, and the sleep time for visualization (which can be increased in real-time during operation).

The robot has 4 distinct possible actions: step forward, rotate slightly left, rotate slightly right, or eat. In the current state, feeding is not implemented, so the robot can only elect to move forward or rotate in either direction.

The action that SCS selects in the Multiscale Actor Critic Morris Maze Model is based on two major factors: affordance – whether the robot can actually perform the action – and votes – how desirable the model's learned policy views each action. The Robotito can afford to move forward if it doesn't get too close to any walls when doing so, and it can afford to turn if it cannot step forward or turning doesn't prevent forward motion on the next step when a forward is possible. Votes are determined by the model used. The action selected is simply whichever action has the non-negative highest number of votes among actions that have affordance. Although it is not possible for all actions to be "impossible," it is possible for all possible actions to have a negative or zero number of votes. In such cases, the robot will move forward if possible, otherwise it will randomly elect to turn left or right for that given cycle. There is currently no code in place to ensure that the robot rotates in a constant direction.

Once an action is selected by the model, SCS executes the action – for the Robotito, this means that a control signal to move forward or to rotate is sent. Once complete, the Robotito is sent a signal to completely stop. The Robotito's integration is designed using a stop-and-wait approach, and as such will perform a single step each cycle then stop until a new action is selected. The motion is not continuous.

Currently, forward and rotate actions are handled using an estimation approach. This was done for two reasons. First, all other approaches required significant modification to the firmware or a drastically faster pose detection system to be effective. Secondly, using estimation, the motions are completed quickly even if some are missed. A PID approach was hampered both by a slow update rate for feedback and a much slower rate of completion, which begins to affect the feasibility of testing multiple individuals.

# Observations from initial testing

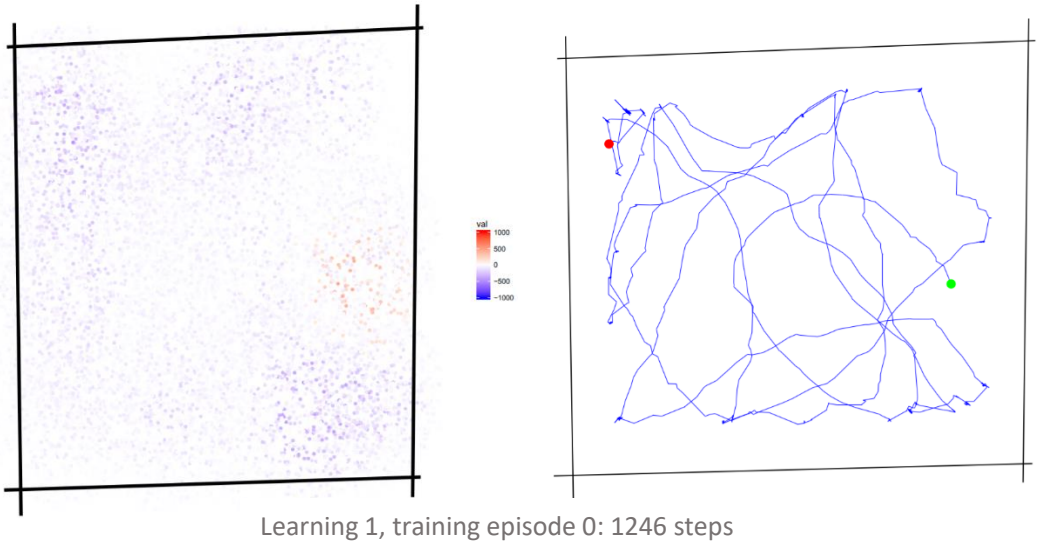
Multiple tests were performed during this study to confirm general accuracy of the firmware, ROS components, turning and forward steps (of the Robotito), and accuracy of marker detection. The final testing was a test of the entire setup, containing 69 proper episodes of execution over 3 individuals (1 learning individual and 2 control individuals) using the Robotito. Below are observations of performance from both the final testing and the individual tests.

As for terminology, an "individual" refers to a single learning entity – each individual learns a policy over multiple instances of trying to reach the platform. Each instance of trying to reach the goal is called an "episode", and there are different numbers of episodes based on the class of individual. In the Multiscale Actor Critic Morris Maze Model, there are two classes (or "groups") of individuals: learning individuals, who have a "trial" of 40 training episodes and a separate trial of 10 recall episodes, and control individuals, who only have a trial of 10 recall episodes. The comparison between the two groups should demonstrate how extra episodes improve the policy by producing a superior path (as determined by the number of cycles taken to reach the platform), or as determined by consistency between different recall episodes.

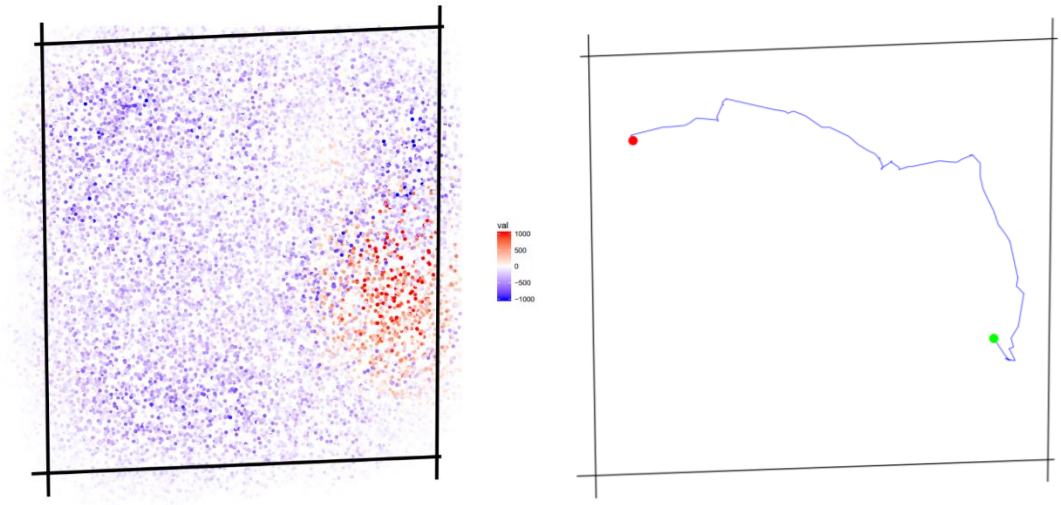
## Proof of learning

Figure 1 contains the values charts for the Learning individual in testing over 5 different points in the Training trial. These charts can be used to infer the robot's policy – namely that it will try to move from the blue regions to the white regions to the red regions. Over time, areas close to the platform will become more valuable (represented by intensity of red), and regions far from it become less valuable (represented by intensity of blue).

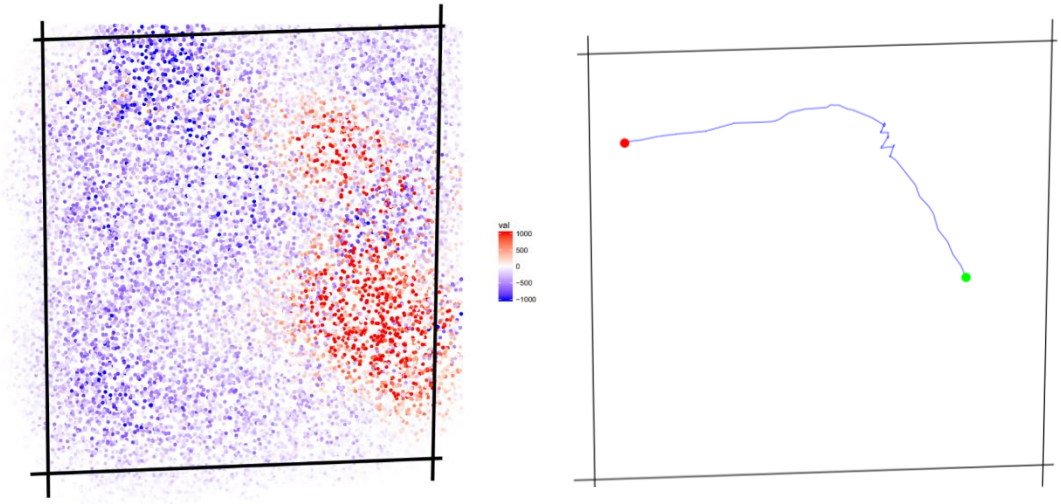
From the paths and charts, it is clear that the individual is slowly identifying a single efficient path to the goal. There is a clear reduction in the length of the path, which coincides with an equally clear contrast in the color concentrations in the values charts. The path for the first trial was made purely by random action (as all regions were equally desirable), and thus spanned the entire maze multiple times until the platform was found by luck. The path for episode 20 contains no loops and has roughly have the total cycles ("steps") in comparison with episode 0. The path for episode 39 is drastically more efficient, only requiring 70 steps total to find the platform, almost a tenth the number of steps required to complete episode 20.




Learning 1, training episode 0: 1246 steps

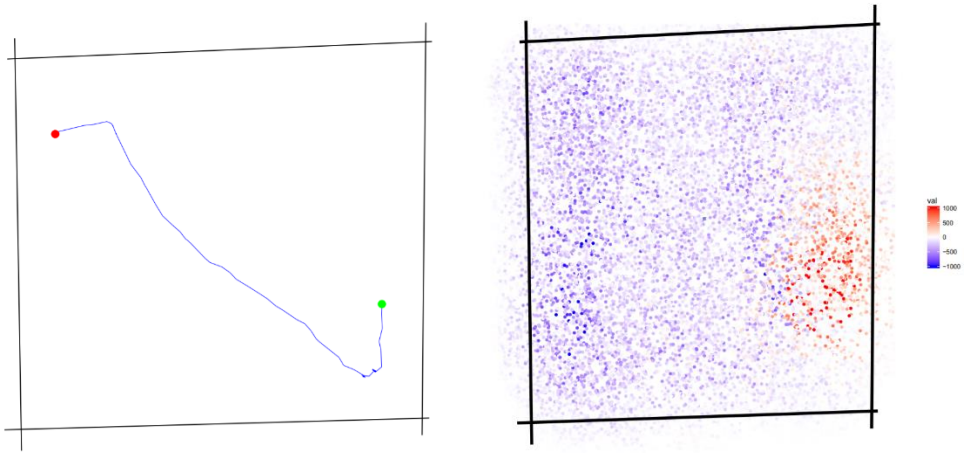



Learning 1, training episode 20: 671 steps

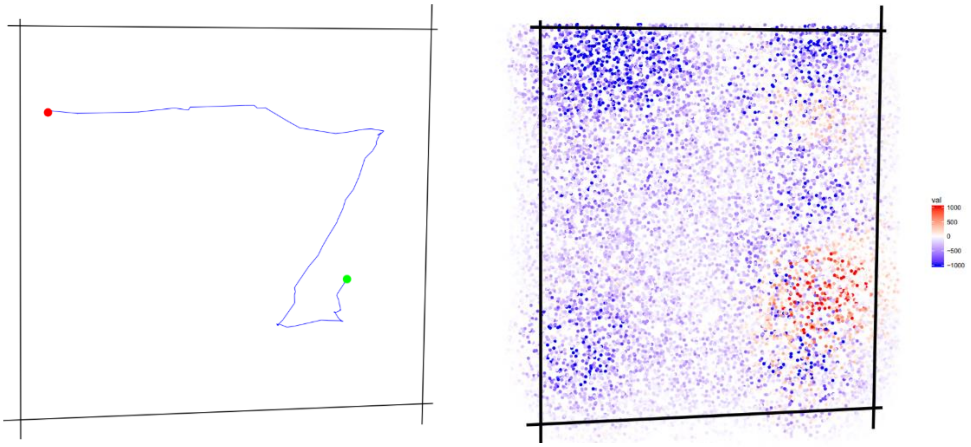



Learning 1, training episode 39: 70 steps

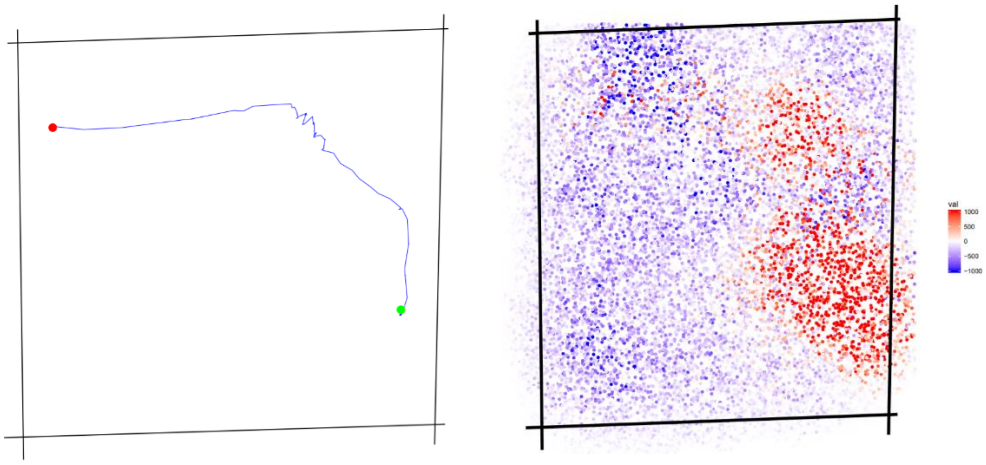
Figure 1: The values charts and paths for Learning 1 individual. In the paths, the red point is the starting position and the green the platform (goal) position.



Control 1, recall episode 9: 117 steps



Control 2, recall episode 9: 118 steps



Learning 1, recall episode 9: 114 steps

**Figure 1: The final recall episodes' paths and activation values charts. From the top down: Control 1, Control 2, and Learning 1. In the paths, red is the starting point and green is the platform. In the activation charts, blue signifies the degree of undesirability, and red signifies desirability. The robot seeks to move from blue to white to red. All individuals faced the right wall when initially placed for all episodes.**

From the results in Figures 2, Control 1 has the least strongly learned policy, owing to few regions of intense color and large concentrations of white. Its path is simply a result of following the white band between bands of blue, then turning toward the platform once the blue concentration becomes weaker.

Learning 1, which has by far the most exploration, has a high concentration of high activation values near the goal, and a clear policy based on the red and white regions. Part of the path taken passes through a blue region, near the wall on the right. On closer inspection, there are also red cell mixed in that region, so it is likely that the robot had learned to go in through this region earlier, then later got stuck by the wall for a significant number of cycles. This couldn't have happen in the opposite order, since red cells are only placed near paths previously taken to the goal. Since every cycle has a negative reward value, getting stuck nearby a wall (a common issue, refer to "Known Issues and Necessary Improvements" points (1) and (2)) can cause a significant reduction in the desirability of the region. This case proves that wasted cycles can cause suboptimal changes in the learned policy, though in at least this case, it seems that enough high-activation cells remained for the path to remain largely unaffected.

Control 2 is the most bizarre case. The intensity of the red regions is similar to Control 1, but the intensity of the blue regions is vastly more extreme – even more dense than the blue concentrations in Learning 1's chart, despite Learning 1 having performed 5 times the number of episodes. The reason is a major issue in the current design – if the Robotito gets too close to a wall, it becomes unable to move forward for long stretches of cycles (see "Known Issues and Necessary Improvements" point (1) for further details). As I recall, this robot suffered from this issue twice, once near the top wall by the starting position, and once on the same wall but on the opposite side. The points where this occurred are relatively easy to guess just based on the high intensity of the blue regions.

Looking very carefully at the pink cells on Control 2's chart, it seems that it began to learn a policy quite similar to Learning 1, where the robot would follow the top wall, curve at the corner, then follow the right wall to the platform. The policy must have later been changed (most likely by getting temporary stuck near the middle of the right wall), since the cells would not be pink anywhere other than on a path to the goal. The actual path the Robotito took simply had it go from the starting position to the pink cells directly in front of it on the right wall, then it would travel along the white cells until the goal was no longer block by dark blue cells.

It is important to note that in the examples in Figure 2, the number of steps is nearly identical. However, looking at the full dataset, there is less consistency in the results for the Control individuals. More testing and more data must be used to make any real determination of whether the Learning group is more efficient than the Control group.

From the above observations, a few overarching considerations can be drawn, though much more data would be necessary to prove any.

1) Any errors that cause the Robotito to spin in place are generally damaging to learning policies, since the resulting negative reward can make a region undesirable in the span of a single episode. Sometimes, this can even overwrite regions of higher than typical values with lower than typical values, which can cause the robot to start adding detours that add steps to initially efficient policies. Perhaps most concerning is as the number of episodes increases, the likelihood of such problems

occurring also increases, meaning that there will be a point where more episodes don't meaningfully improve the policy, but they may cause negative impacts instead.

2) Generally, if the starting position and orientation allow for forward for a few steps, the robot will learn to take a few forward steps, regardless of other potential possibilities. Notice how every robot moves in the same direction, and this direction is the one they were initially placed in. None of the individuals elected to turn initially, despite making a turn initially being an optimal decision in each of their cases.
3) Based on how the robots learn (large reward upon reaching goal, negative reward for each step) and the results above, it seems that the robots won't reach the optimal path. Rather they will identify a somewhat effective, but suboptimal path and refine it as more episodes occur. Looking at Learning 1's policy, it would become increasingly likely that the individual would follow a curve from the start to the goal. It may well reduce the amount of unnecessary turns that cause the zig-zagging pattern, but it would be unlikely that the robot would ever achieve a straight line.

A potential study would be to place the initial position in the center of the Maze, then face the Robotito away from the platform. This would allow us to test considerations (2) and (3), as the robot will either learn a policy where it turns around at the initial point (disproving (2) and (3)), or the robot would move forward and then turn towards the platform later (proving (2) and (3)).

From the individual component testing, a few details of the setup were more comprehensively studied, some of which generated changes to the extension code. As such, details are listed below.

Generally, the robot will spend a significant amount of time turning when facing walls and especially corners. After an extreme case with extremely long stretches of pointless turning, I had modified the action selection module (titled "NoExploration.java") to take the first possible forward if the robot had not taken a forward step in 50 cycles. I haven't specifically tested this patch, so it is possible that it merely biases the robot from staying in one place for too long. Regardless of the main cause of the incessant turning, once the robot moves slightly away from the walls, it moves through the maze as normal.

The AR marker system is quite consistent when specifying the position and angle of a marker – it generally is within a few centimeters of accuracy for position and 0.02 radians for angle. However, this accuracy can cause some inconsistency with the wall positions in SCS. As you can see in the videos provided with this documentation, the walls in the simulation window sometimes shift slightly, despite the walls not being moved in the actual camera stream. Slight motion in the paper markers and simple inconsistency in the AR marker detector can cause the orientation or position of the wall to vary, though the extent of these changes was not specifically tested.

Additionally, though the wall markers were detected without issue a majority of the time, the marker for the Robotito was occasionally "lost," forcing the simulator to sometimes rely on old pose values when moving. Though the marker would be found again within a few cycles, this can cause the robot to take unintended actions, sometimes even moving too close to the wall, which causes significant problems [refer to "Known Issues and Necessary Improvements", point (1)]. When performing tests using only the markers, it was obvious that ample white margins around the black marker drastically improved detection but placing a larger marker on the robot increased the likelihood of the marker falling off, the marker being bent or torn, and for air pressure and motion to move the marker around, interfering with accurate pose information.

# Known Issues and Necessary Improvements

Testing has revealed a few notable issues, but due to time constraints, they have been left for future workers to resolve. Below are the most significant issues revealed during testing, or otherwise significant improvements that must be made to continue with experimentation.

## 1 If the Robotito gets too close to a wall, it becomes unable to move forward

If the Robotito ends up too close to any wall, it becomes unable to step away from the wall, making it turn randomly even when it can clearly move forward. This continues until it manages to push itself away from the wall, or the wall detector's readings put the wall further away from the robot.

There could be multiple reasons for this, but I assume that it is principally caused by the function CanRobotMove in "/scs/multiscalemodel/src/edu/usf/ratsim/robot/robotito/Robotito.java". This method identifies whether the robot will move too close to a wall if it moves a specified distance forward with the given change in orientation. This is done by defining three parallel paths of the robot to the goal distance – the center of the robot to the goal position and the left and right flanks of the Robotito moving the same distance. If a wall intersects any of these lines, or if the robot's final position is too close to a wall, the function determines that the robot cannot complete the action.

Since the robot does not move forward even when clearly facing away from this wall, it is most likely that the paths are projected slightly behind the direction the robot will travel in. This explains why the Robotito normally has no issues, but suddenly becomes unable to move forward when too close to the wall.

## 2 Random turning when no possible actions are desirable

The policy of taking a random action if no actions receive a positive number of votes has a tendency to consume a few dozen cycles whenever the Robotito directly faces a wall or corner. Optimally, the robot would turn in a single direction until it can move forward again (whether this should be a bias or learned is a design decision that should be carefully considered), but what generally happens is that the robot turns randomly in either direction for a variable number of cycles until it manages to face a direction from which it can turn forward.

This is a concern because every cycle counts as a negative reward – if the random motions cancel each other for a significant number of cycles, the robot will learn that the particular area is undesirable. The core problem is that the randomness will make the robot's policy about walls inconsistent – where it will randomly learn that some walls or corners are more undesirable than others for no reason other than random chance. This can be particularly damaging when the robot has already learned a policy and it becomes stuck on a wall nearby the path learned – forcing the robot to move around the undesirable area to get back onto the main path, which is drastically less efficient than just learning a new path altogether.

## 3 Feeders must be implemented

The Robotito extension doesn't implement feeders at this time. Though these are not essential to all experiments (indeed they aren't essential for single-goal testing, where platforms are simpler), much of the testing we intend to perform requires feeders, so the Robotito.java code must be updated to interact with feeders.

To do so, you will need to implement the FeederRobot interface in "/scs/multiscalemodel/src/edu/usf/ratsim/robot/robotito/Robotito.java" and will further need to modify any references to the eat method. You may wish to refer the Virtual Step Robot code, which already implements the FeederRobot interface.

# Suggested Improvements

Below are suggestions that I believe will improve some aspect of the code for using the Robotito. These are not essential, and mainly are comprised of quality-of-life improvements that should be considered to make experiments easier or more convenient to perform.

## 1 Replace the Alvar AR Markers with another detection system

Though the AR markers are convenient and simple, they pose a few issues. Mainly, the markers take too long to update. The marker position stream only updates once every 100 ms on average, but this doesn't always include updated information. As you can see in the RVIZ window of the video recordings, the robot's marker is sometimes "lost," where the pose of the robot isn't being published at all for a few hundred milliseconds. This means that SCS may well give multiple control signals from a single out-of-date pose reading.

This is the principle reason why the robot gets too close to the walls, since it continues to select forward actions (thinking it hasn't moved), while the actual robot is moving forward 2 or 3 steps, possibly hitting a wall in the process.

Another major issue is simply the shape of the markers. Since they are large square shapes, it can be difficult to leave them on top of a wall without covering some portion of the field. If the robot moves too close to these, the robot's marker may become obscured, trapping the robot in an infinite spinning cycle.

## 2 Initial wall detection failures

Although I haven't replicated this issue during the final tests I performed, when I was initially testing the setup, SCS failed to detect internal walls I had added (mid-trial) before the robot began to move. It would be a good idea to make the wall drawer run between episodes, if possible, to make sure all walls are detected before letting the robot move, lest the robot ram into one.

## 3 Automatically move the robot to the starting position

It becomes quite tedious to manually move the robot to the proper starting position after each episode and increases the likelihood of mistakes. It can be difficult to determine if the robot is even in the correct position, as SCS doesn't provide any position or orientation detection during this phase. Furthermore, you will need to manually move the robot 10 times for each Control individual, and 50 for each Learning.

As such, it is highly advised that a method to automatically move the Robotito to the starting position and to turn it to the proper orientation be written after pressing enter.

### 4 Improve the platform and initial position system

Currently, the robot's initial position and the position of the platform are explicitly declared in the experimental XML files. If you wish to change this, you would currently need to read the pose rostopic and use the robot's marker to map the real-world position to the coordinate value. Creating an automatic way of detecting this or creating some way to physically represent the platform and starting position would be reduce the likelihood of errors and simplify the process.

### 5 Fix the camera system to use both Stingray cameras

Currently, only one of the two Stingray cameras is used to identify the pose of the robot and walls. This is because the original code for combining the two camera results failed to compile, so I removed the second camera. It will be desirable to have large arenas, so adding in new code to handle the camera overlaps in the pose detector and the wall detector will be necessary. Refer the FAQ for more details.

### 6 Implement a better obstacle system

The only type of obstacle you can use for the experiment is a rectangle. This is due to both the Robotito's blindness and the complexity of adding different geometric shapes to the marker detection system and the intersection system. As more complex obstacles will need to be used at some point in the future, the means that the of handing obstacles will need to be reconsidered.

Note that this is a non-trivial issue, as it will either require the camera system to detect objects directly using image processing techniques, or it will require the robot to use its own sensors to identify obstacles, and for SCS to be able to interpret these results into boundaries in the arena. Either way, this will require significant changes to SCS and possibly the Robotito's firmware.

## Conclusion

The Robotito robot has been successfully integrated as a viable robot for the SCS Multiscale Actor Critic Morris Maze Model, and with few to no changes, it can be useable for most other models in the system. During testing, it was made clear that the robot is successfully able to learn a policy that reduces the number of steps to reach the goal, but questions remain as to whether the Robotito can create an optimal policy and whether the Learning group has statistically superior path generation in comparison to the Control group. Before additional testing, there are a few critical changes that will be necessary to ensure the validity of the model is not hampered by confounding variables and implementation errors.