A Maze Bot

Leonid Keselman*[1], Joshua Born†[2], Negin Heravi‡[2], Jess Moss§[2], and Margaret Coad¶[2]

[1]Department of Computer Science, Stanford University
[2]Department of Mechanical Engineering, Stanford University

June 12, 2017

Abstract

Our group found inspiration in the labyrinth maze toys. In them, the user must move a ball through the maze to reach a desired end position by tilting the maze in different directions to navigate the ball around the walls. The goal of our project was to program the Puma robot to mimic this behavior. We wanted to incorporate both vision and real time feedback in our project so we planned on having a camera above the robot to first solve the maze and then provide the current ball position as the robot moved the ball to it's goal position.

The initial challenges of this project were creating a real time vision system that would both find the solutions for the maze ahead of time and then while solving the maze provide feedback of the current ball position with minimal latency so that it could be used in a controller for the Puma. We also needed to design a control law that would move the ball to a target position. Before running this on the Puma we also wanted to test our controller in simulation which required modifying the simulator, visualizer, and controller code to accept two "robots", one for the Puma and another that represented the ball.

In addition, we hoped to be able to solve any viable maze we were given using computer vision. To achieve this, we first acquired blue and red Lego pieces, which allowed us to easily reconfigure the maze as well as provide contrast for the vision algorithm. The red Legos represented possible positions the ball could go and the blue Lego pieces represented the outline of the maze or the walls. Additionally, one green square represented the final desired position and the ball was a black marble. Using this color schematic, we were then able to process the maze and solve it using a breadth-first-search algorithm. Since the camera was mounted above the robot using a tripod we also needed to track the orientation of the maze, which was done using AprilTags and allowed us to create an undistorted view of the maze where we could then get the position of the ball.

We had to create a viable controller that worked for our system. To do this, we decided to keep the Puma end effector position fixed, and only change its orientation. While we initially hoped to do this using proportional derivative control, we realized delays in the camera sensory information resulted in the the derivative term creating large oscillations. Hence, instead we used a proportional controller based off the balls current and desired position calculated from the maze solver.

Before controlling the Puma, we precomputed the solution for each new maze. This yielded a desired position for the ball at every point on the maze. So during runtime, when we needed to control the Puma to solve the maze we simply found the current position of the ball and then looked up the desired position to send to our controller. This allowed for real time feedback with low latency since the maze solution was only computed once. Using this design and control strategy we were able to consistently solve the maze and navigate the ball to the goal position.

*leonidk@stanford.edu
†jcborn@stanford.edu
‡nheravi@stanford.edu
§jlmoss@stanford.edu
¶mmcoad@stanford.edu

1 Final Implementation

A video example of our final prototype operating successfully is available in full or in part.

1.1 Hardware

To facilitate the change of the maze design for users, Lego pieces were used to build the maze. For the walls standard blue 2X1 legos were used, and to provide a flat surface for the ball to roll on we used flat red 2X2 legos. A black marble was used as the rolling ball. Four 5 mm holes were drilled at the center of the base plate with 30 mm by 30 mm spacing. To account for the error and for better alignment of the drilled holes with Puma's end effector holes, the holes can be widened to about 7 mm in diameter. The maze plate was screwed onto the Puma's end effector and four pieces of blue foam were used to cover the screw heads in order to avoid confusion in the vision module. This direct mounting method to the Puma minimized the mass of the end effector which allowed for better performance when controlling orientation. A piece of green foam was used to cover the final goal tile. These Legos allowed us to try many different maze configurations to ensure that both our maze solving algorithm and controller were effective. A picture of the built Lego maze is shown in Figure 1.

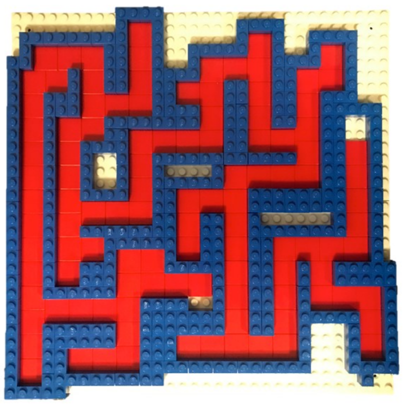
Figure 1: Maze built from of Legos.

1.2 Vision

We initially planned to track the maze directly using an RGBD camera to estimate position; color to track the maze and depth to perform maze segmentation from the background. However, challenges handling camera noise, accuracy in the depth data in tracking a maze our size, and inconsistent lighting led us to abandon that approach. Instead we moved to a marker-based tracking system using a standard RGB camera. After some review, we chose to use AprilTags, a set of identifiable QR codes which are easy to detect and uniquely identifiable (regardless of orientation). With four unique AprilTags printed and placed on the corners of the maze, we can reliably detect and track any maze of any size with a standard webcam. We initially solved for a homography [1] between the four detected AprilTags [2] in the corners and a standard reference plane, which allowed us to undistort the maze into a canonical view easily (see video1). We further extended this by using the corners of each tag as reference locations, allowing us to use anywhere from 1 to 4 detected tags in order to perform undistortion (see video2) although the tracking is more jittery as undetected corners are only weakly constrained. The maze can be segmented into valid (red and black (due to marble color)) and invalid (blue and white) in order to determine valid paths. Color-based thresholds work sufficiently well in our conditions.

On top of our canonical view, the ball is tracked using color thresholds. Originally we'd used a method for background modeling, but the specular highlights from the ceiling lights provided to be hard to model for most background subtraction methods. Instead we use color thresholds, knowing

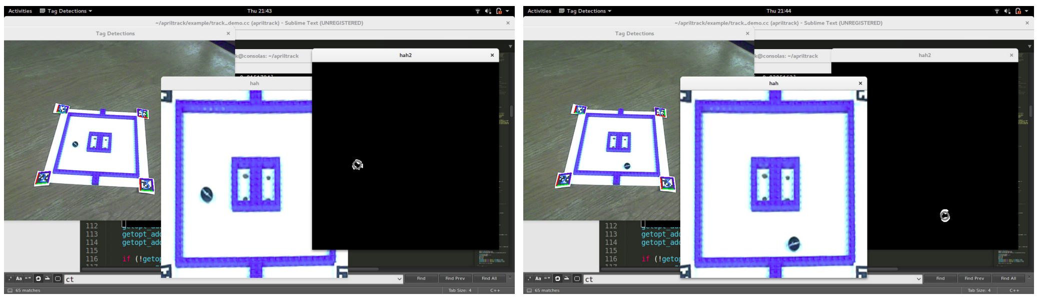
Figure 2: Tracked ball position in canonical view. Output location is drawn with a gray circle on top of our white mask image.

that our walls are blue or white, our maze floor is red, and the ball is black ($s = min(r, g, b), b = s < \tau$ (with τ set to $\frac{1}{10}$ of image brightness)). This image of b values is then filtered with a standard morphological operation, *opening* [3] with a window size of 11×11 in order to remove small, noisy segments. After this, all objects in our canonical image, but the black marble, are removed. The center-of-mass of this image is then returned as our ball position (and the total mass and standard deviation of this signal is used as a confidence marker, when we're correctly detecting only the ball, the distribution and total energy of the ball are known). All of this information, including the ball position, algorithm control parameters, and tracking statistics are output controlled and outputted through *redis*.

Our results are from a standard VGA (640×480) webcam, and we expect better results when we use a high-end webcam. However, our results from the above method are fairly stable under test conditions, but this solution is only tracking-by-detection, so there is plenty of room to explore smoothing and tracking techniques, such as Kalman Filters [4] and Lucas-Kanade [5] tracking respectively.

1.3 Maze solving

We implemented a pixel-level maze solver in Python using a handwritten breadth-first-search from a detected target location. Maze solving is, in fact, the first published use of the BFS algorithm [6]. The maze solver runs on a per-pixel level. Walls are marked as pre-visited and the detected final location (green) is inserted into the queue as the starting location. The BFS solver is written by us in C++, and directly uses the image pixels as the graph specification. The maze is solved by back-tracking (which gives desired directions at every point in the maze). These back-tracks are then transformed into desired locations by following each path until their direction changes. As with the vision component, these are all done in normalized maze coordinate space [0,1], which can be transformed into 3D coordinates using the known orientation of the Puma arm end-effector. An example of this operation working on a randomly generated maze can be seen at Fig. 3.

In solving the maze, our handwritten solver allows for both L_1 and L_2 distance norms, the former only allowing motion down cardinal axes, while the latter enables diagonal motion. While the L_1 (Manhattan distance) solver works correctly in theory, it is not tenable for robotics use due to how ties are forced to be handled. If ties are broken with consistent preference direction, we end up with artifacts where an entire maze corridor has a one pixel "go left" and all other pixels stating "go down" (to get the the "left" pixel). Unfortunately, the real ball position has a fixed size that never gets to this one-pixel boundary and so the ball is stranded. If ties are broken randomly, then the target positions become very jittery, randomly jumping from "go down" to "go left" from pixel to pixel, which isn't desired for control. We originally only had an L_1 solver and had to implement an L_2 solver to resolve this issue. The L_2 solver generates diagonal motions which resolve this problem.

An additional option in our maze solver is to either use per-pixel target locations with N pixels of lookahead, or to use connected segments where all pixels with the same target path are grouped into the same target location (since our maze is axis-aligned and structured in horizontal and vertical corridors). We found the pixel-based solver was noisy without lookahead and hard to tune for lookahead to work smoothly (sometimes it would look ahead too far, sometimes not enough). We instead use our segmentation-based target path generation, where paths are followed until target direction is changed; this requires no lookahead tuning. This solver, with an L_2 distance

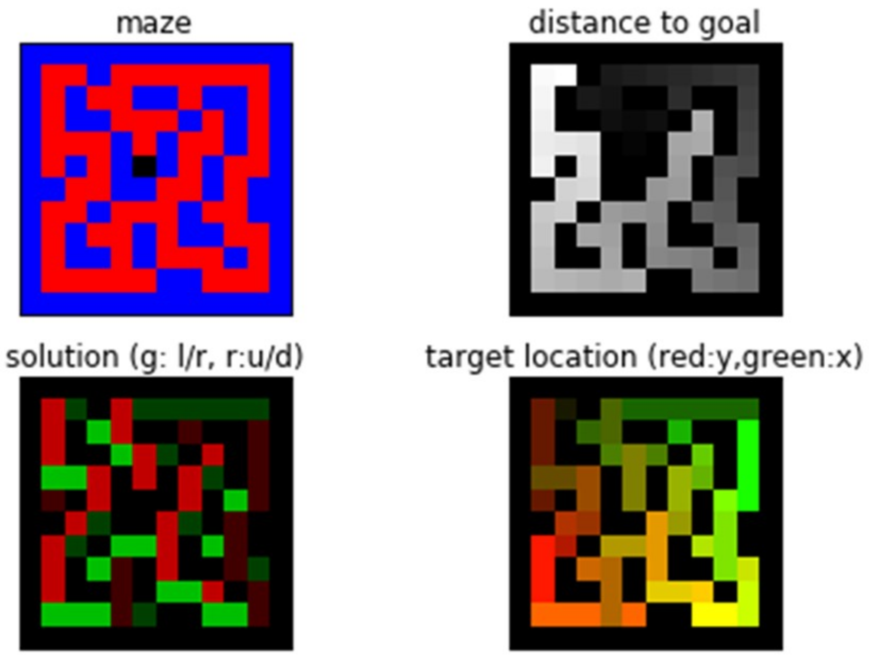

Figure 3: Maze solver

computation is what we ultimately used. The tracking code includes the solver code and outputs these desired positions through *redis*.

A video of our initial maze detection, solving and tracking can be found here. The code is also available.

1.4 Visualization and Simulation

In order to be able to simulate the robot's control of the maze, the URDF file for the Puma was modified such that the end effector has a model of the maze attached to it. The rolling ball on the maze was defined as a PPP robot with a spherical shape with about 50 grams of mass defined as the last link. Since the provided visualizer and simulator library through the CS225 git repository only supported one robot, modifications were made to the code for the simulator and visualizer based on the example codes in the SAI2 library. Pictures of the updated URDF and simulation of a rolling ball on a plate are shown in Figure. 4.

1.5 Control

We've developed a simple control law that outputs the desired orientation that we want the maze to be tilted to based on the position of the ball and then uses orientation control to achieve this desired configuration.

$$\begin{bmatrix}\theta_y \\ \theta_x\end{bmatrix} = -k_p \begin{bmatrix}x - x_{des} \\ y - y_{des}\end{bmatrix} - k_v \begin{bmatrix}\dot{x} - \dot{x_{des}} \\ \dot{y} - \dot{y_{des}}\end{bmatrix} \tag{1}$$

The idea behind our control is that we keep the end effector at a fixed location, but rotate it about either the x or y axes. To do this, we first calculate a θ_x and θ_y based off the equations above. From here, we were able to calculate rotation matrices R_x and R_y based off of these angles. Lastly, we were able to combine these angles to create the final rotation matrix $R_f = R_x * R_y$. We then used the rotation matrix in our orientation control to set the final configuration of the end effector. Note that while we did try to use derivative control, due to the time lag of the camera, and having to calculate velocities based off camera frames, this worsened the characteristics of the

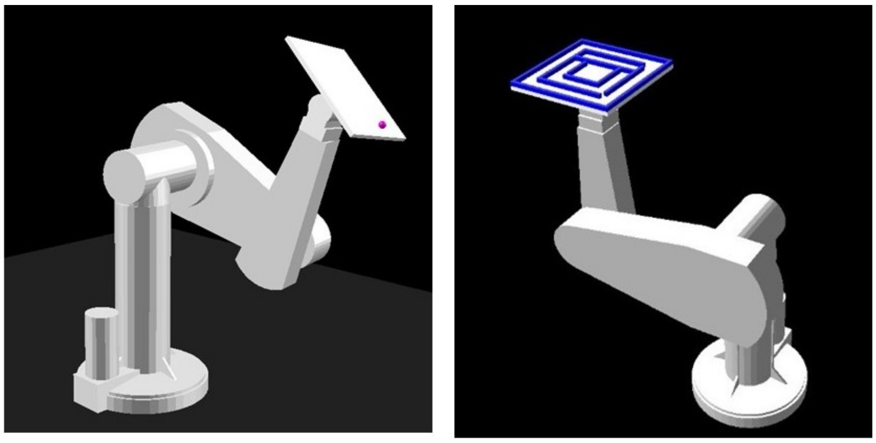
Figure 4: Updated URDF and Simulation of the Ball and the Maze

controller that we hoped to achieve. Namely, it led to less damping and higher overshoot. Hence, our final version of the controller was purely consisted of proportional control.

Before we ran our control law for solving the maze, we first commanded the Puma using joint coordinates to move the end effector to a position in space that was directly under the camera. We also choose the initial position to be away from any singularities to increase the performance of our controller. Once the end effector was in the desired position we switched to operational space control, where we used the above control law to calculate the desired rotation matrix which was then converted into a quaternion and sent to the robot. For the x-y-z position we sent the position that the robot ended up at during the initial configuration and had it remain at that location as the Puma was solving the maze.

2 Challenges

Initially, we attempted to fully define the dynamics of the maze by finding the full set of equations of motion. However, we soon realized that even in a two dimensional system, this led, not only to extremely complicated coupled equations, but also that we still did not know all of the necessary variables such as the coefficient of friction. Instead, we implemented a closed loop controller and were able to tune the gains until they were satisfactory for the physical system.

Over the course of this project, some of the major challenges we encountered were with the simulations. During initial simulations, a strange behavior was observed where the ball didn't fall off the edge of the maze but rather kept rolling into infinity. After many hours of assessing the code, it was found that this was caused by the small value of mass assigned to the ball. If the mass is set to be extremely small, possible numerical errors in the program can lead to strange behaviors where the gravity seems to be not active. This problem was solved by assigning a higher mass value to the ball robot.

Another experienced problem was placing the two robots at correct locations with respect to one another at the initial moment. If the ball bot was placed too close to the end effector it would start in the collision phase, and the program would crash. On the other hand, if the robot ball was placed too far from the plate it would start bouncing on the plate rather than rolling. In addition, occasionally, in the middle of simulation the ball would stop rolling. When we further inquired about these problems, we realized that there could be a variety of bugs either in or out of our control that could cause these. Since the when the ball rolled, the control laws worked as expected, we were advised to move on and try to implement our code directly on the Puma itself instead of trying to fix the bug. We successfully did this, as we had felt we gained all the knowledge we could from the simulation at that point.

In addition, when running our simulation the maze that we defined to be attached to the end effector would sometimes separate from the end effector even though it was defined to be a set distance away. This was an issue that we were not able to determine the cause of but instead used redis keys to track how the end effector was moving.

When solving the maze tuning the gains was also time consuming to create a balance between the speed of solving the maze and creating smooth movements in orientation. This is one area where if we implemented our project again we could look into trajectory generation for orientation control that would allow for smoother paths to send to the Puma compared to the GOTO strategy that we used in our project.

3 Conclusion

Throughout the course of this project, we leaned to create a useful model, simulate it, and apply it to a physical system. In our case, we had to first decide how we wanted to control our system to navigate the ball through the maze. While initially, we decided we would need to know the minute details about the system and create specific equations of motion, we eventually realized we could create similar results by creating a closed loop feedback controller. Next, when it came to simulation, we were able to numerically test our controller and determine if it produced the results we expected from our mathematical calculations. While this worked within some level of accuracy, the simulation fell short in some areas, and we decided to test our controller on the Puma. Tuning the gains until we created a result that was reasonable created our finished product which both worked and we were proud of. In moving from simulation to the Puma we also learned of the differences in implementing code on actual hardware, such as how the gains needed to be adjusted from simulation, and the use of the Puma driver which provided the physical information of the Puma rather then the simulator. Ultimately, going through this process from calculations to finished results was an invaluable experience.

A video example of our final prototype operating successfully is available in full or in part.

References

[1] O. D. Faugeras and F. Lustman, "Motion and structure from motion in a piecewise planar environment," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 2, no. 03, pp. 485–508, 1988.

[2] E. Olson, "AprilTag: A robust and flexible visual fiducial system," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).* IEEE, May 2011, pp. 3400–3407.

[3] J. Serra, *Image analysis and mathematical morphology, v. 1.* Academic press, 1982.

[4] R. E. Kalman *et al.*, "A new approach to linear filtering and prediction problems," 1960.

[5] B. D. Lucas, T. Kanade *et al.*, "An iterative image registration technique with an application to stereo vision," 1981.

[6] E. F. Moore, "The shortest path through a maze," in *Proc. Int. Symp. Switching Theory, 1959*, 1959, pp. 285–292.