CalcioBot: a ROSbot pusher

Ugur Y. Yavuz[1], Qiyao Zuo[1], Jeffrey J. Cho[1] and Viney Regunath[1]

***Abstract*—This paper focuses on the problem of a robot being able to detect a target object and a goal in an environment it has been placed in and push the object into the goal while avoiding obstacles. We attempted to solve this problem by implementing a ROS node that directs a ROSbot to perform this task.**

I. Introduction & Problem Statement

One of the classic definitions of robotics revolves around a machine resembling a living creature being capable of executing complex action. We believe that robots serving the needs of humans in the future will need to emulate and perhaps surpass our physical abilities and action. In this context, athletics can be considered the pinnacle of human achievement. Since the 2000s, robotics firms and researchers have been experimenting with the idea of "soccer robots" or robots that resemble humans playing the popular sport that is soccer. In line with this thought process and those fashioning soccer robots, we imagined CalcioBot.

CalcioBot has been designed to solve a simplified version of this task: that of detecting and pushing a target object into a detected goal in an environment, potentially containing obstacles to be avoided. To make the task of detection as straightforward as possible, we have assumed that the ball and the goal are colorwise easily distinguishable from other objects and obstacles present in the environment. Furthermore, the obstacles have been assumed to be static, and the environment has been assumed to provide as much consistent and correct information from its RGB-D sensors as possible. We note that the latter assumption has motivated the use of the Gazebo 3D robotics simulator [1] (and a model of ROSbot in the Unified Robot Description Format [2] in Gazebo) throughout the project, over a real-life environment. A future improvement of the project would involve the testing of our implementation on the actual ROSbot in a carefully designed environment, where the information received by the sensors would not be expected to be as accurate and steady as those generated in the simulation.

[1] Ugur Y. Yavuz, Qiyao Zuo, Jeffrey J. Cho and Viney Regunath are with the Computer Science Department, Dartmouth College, 9 Maynard St. Hanover, NH, 03755 USA.

II. Literature

The simplified problem that we attempted to study mainly falls under the umbrella of the field of robot pushing, and is technically a *dynamic pushing* problem where the pushing agent and the pushed object are both in movement only in the horizontal plane, but their inertial forces are consequential [3]. The specific problem of ball pushing has been studied in depth by Li and Zell [4], and the authors have described a detailed linear state feedback controller for this type of motion.

Although pushing is the core task, there are auxiliary tasks to consider: that of simultaneous localization and mapping (in other words, SLAM), and that of object detection. And it is also important to note that the action of pushing works in conjunction with path finding in our problem, as the goal is to push the object to a certain location, starting from an arbitrary location.

The recently introduced Rao-Blackwellized particle filters constitute an effective means of solving SLAM problems, and this is achieved by using particles that carry individual maps of the environment [5]. The *gmapping* library developed by Grisetti, Stachniss and Burgard [6] (for which there exists a wrapper developed for the *Robot Operating System* (ROS) [7]), uses an implementation of Rao-Blackwellized particle filters that reduce the number of particles for learning grid maps, therefore makes the rapid creation of grid maps from laser range data possible.

For object detection, we note that systems that leverage deep learning techniques to recognize a wide range of objects , such as the "*You only look once"* (YOLO) real-time object detection system [8] which has been trained on the *Common Objects in Context* dataset [9], are widely available for public use. However, especially in consideration of our colorwise distinguishability assumption regarding the objects of interest, these techniques appear to be excessive for our needs. As a result, these techniques can be circumvented by the use of simpler computer vision tools such as the *OpenCV* library [10], which contains implementations of *blob detection* and *connected-component labeling* that are of use for the

detection of uniformly colored regions in images. Specifically, *Blob detection* involves detecting a group of connected pixels in an image that share a common property (e.g. color value). On the other hand, *Connected-component labeling*, or region labeling, applies graph theory over the "connectivity" of groups of connected pixels. The labeling then applies contour properties to quantify the targeted region.

Finally, for the path-planning component of the project, as incorporating heuristics while computing the paths might be of interest, the *A* search algorithm* developed by Hart, Nilsson and Raphael [11] appears to be a good candidate, also owing to its efficient and optimal performance. This algorithm could then be applied on a previously generated grid map. However, in cases where the construction of the grid is not feasible, techniques such as *rapidly-exploring random trees* [12] could be of use given that the environment is not discretized by default.

III. Method

To approach the afore-described problem, we have separated the challenge into three main components: object detection, path finding, and path following/pushing. We created an environment in Gazebo containing a blue cube as the target object and a yellow/orange goal. Both objects are visually distinguishable from the gray surrounding walls in the simulation. The script to launch the Gazebo environment automatically launches *gmapping* which then creates and publishes an occupancy grid of the environment to a map topic in ROS, as well as *rviz,* from which the laser scans and the RGB camera of the robot can be observed.

The object detection module is implemented in *detector.py* and uses the RGBD sensors of the robot to recognize the two objects (the target and the goal) and locate them in the global map. The *Detector* class object subscribes to and synchronizes messages coming from the RGB image, the depth image, and the camera information topics. In a callback function, the object filters the RGB images it receives from the robot by two colors- first blue for the cube, then yellow/orange for the goal. The class object filtrates by using masks specified in the HSV format. Next, the object detects the largest non-background component in the masked images via the "*connected ComponentsWithStats"* helper function available in *OpenCV*, and then computes the average depth of the pixels corresponding to the component using the depth image. The depths varying too little or being too far away is indicative of bad measurements, therefore no location estimates are provided if either of these is the case. Furthermore, we print out statements relating to insignificant depth values in the terminal. Otherwise, the camera information is leveraged (along with the location of the centroid of the component in the image, and the depth of the relevant pixel) to assign the reflection of the centroid on the ground to a location in the map. This reflection of the centroid becomes the estimated location of the object of the color that the detector has applied a mask for.

The second component of the project is the path finding. The path-finding module is implemented in *pather.py,* and leverages the popular graph theory library *NetworkX* [13] to apply A* on the occupancy grid. The module first implements a *Grid* class to work with the occupancy grid data published to the map topic by *gmapping*.

The core of the module is the *Pather* class object, which subscribes to the map topic such that the callback function stores the data in the *Grid* format within a class variable. The object launches two different marker array publishers, and supports a *find_path* method which invokes A* on the grid.

A* is mainly run with the *cautious_dist_to_target* heuristic. The heuristic takes into account the distance of the points along the path to the nearest walls within a given window (*wall_window*) as well as the weight value to be assigned to the inverse of the distance of the closest wall (noted as *k*, which is empirically set to 20 by default), in order to keep the robot as far away as possible from walls.

The *cautious_dist_to_target* function takes in several parameters. one prominent parameter, *start_ignore_window*, allows the object to disregard the occupancy of the cells around the starting point. We note that this parameter is necessary for the computation of the path from the target object to the goal, as the target object blocks the cells around itself in the global map. Besides *start_ignore_window*, *cautious_dist_to_target* passes *wall_window* and *k*. Furthermore, a flag can be passed into the function for the path to be displayed in the form of a marker array, and each invocation of the method will set a new color to the markers being published.

The final section of the project is the path following component. The path-following/pushing module is implemented in *pusher.py*. The file contains a simple *PD controller* object to control the motion of the robot. However the core of the module

is in the *Pusher* class object in which the motion of the robot is governed using a finite-state machine. The *Pusher* object, once prompted to follow a path of given points (the starting point of which is assumed to be in the robot's proximity) publishes *Twist* messages to the velocity command topic in order to make the robot perform the required motion.Furthermore, the object makes the robot rotate towards the start of the path, and then performs a continuous forward motion to follow the path. At the same time, it adjusts for rotational errors with the help of the PD controller. The error fed to the controller is the cross-product of the distance vector from the robot to the closest point on the path, and the local gradient of that point along the path. This controller was developed as an ad hoc solution upon investigating the necessary rotation for error correction based upon a number of possible scenarios,which depended on the direction of the path and the location of the robot. We also empirically selected the parameters of the controller after extensive testing. We also note that our class implements a path extension method, which extends the start of a path by adding the provided number of points, taking into account the gradient of the initial points on the path.

Although the controller suggested by Li and Zell [4] seemed to be quite suitable for the problem we are studying, we were unable to implement this controller on time. Furthermore, their implementation of the controller required constant messages over the cube but our depth sensors could not adequately locate the cube when it was too close to the robot. As a result, we diverted our efforts into a more ad hoc implementation that could push the cube without continuously checking for its location relative to the robot.

We use *driver.py* to coordinate with all the modules mentioned above. The *Driver* class object initializes instances of *Detector, Pather* and *Pusher*. It also provides a dictionary as an argument to the *Detector* in order to receive information about the detected points' locations and whether there is an active detection sequence running. The driver script, once run, registers itself as a ROS node, and then starts communicating with the *Detector* class object. During the detection, the user is expected to teleoperate the robot to aid this detection. We had initially intended this to be achieved by random automated motion; however, the location estimates took a considerably longer time to produce - it took a significant amount of time for the robot to place itself in a spot where it could produce reasonably accurate estimates. As a result, we came to the conclusion that user teleoperation would be the most efficient way to deal with the problem. During the teleoperation phase, the *Detector* prints estimates for the location of the target object (e.g. the blue cube) in the environment. Once the user is satisfied with the estimate, they publish a non-empty string message to the *calcio_driver* topic in order to register this estimate. The estimate is then sent and recorded as the blue cube's location in the map. After this first estimate is registered, the user must teleoperate the robot towards the target goal object. The process is similar for the registration of the estimated location for the goal.

Once both estimates are registered, the robot then communicates with the *Pather* to compute a path from the target object to the robot. Next, it computes a path from its current position to a point reasonably close to the cube, or the start of the first path generated. However, the second path's end location will be far enough from the target object so that it can push the object as it starts following the path from cube to goal.

The computation of this specific point is aided by the *path extending* method available to the *Pusher* instance. Both paths are then published as marker arrays, viewable in *rviz*. Finally, the robot is asked to follow the path (via the invocation of the *follow_path* method of the *Pusher*) from its current location to the point preceding the path, and then asked to follow the path from the target object to the goal. The robot will move through the path with the assumption that the target object is being pushed. The script finally indicates the termination of the robot's movement by printing a message in the Terminal.

IV. Individual Contributions

- **Ugur Yavuz:** Design of the workflow, creation of the Gazebo environment, the design and the implementation of the *Detector* and the *Pusher* modules, the design of and some work on the implementation of the *Pather* module, the implementation of the driver, recording and editing of the demonstration video, literature review, and documentation.

- **Qiyao Zuo:** Implementation of the *Pather* module, debugging and improvement of the *Detector* and the *Pusher*, and work on the Gazebo environment configuration.

- **Jeffrey Cho:** Debugging and improvement of the *Detector* and the *Pusher*, and work on experimentation.

- **Viney Regunath:** Debugging and improvement of the *Pusher*, preparation of the demonstration presentation, and documentation.

V. Implementation

Please note our open source code project in Github for insight on our code implementation for CalcioBot (https://github.com/uguryavuz/CalcioBot).

VI. Experiments

To minimize any nonessential background components (other than the cube and the goal) while also mimicking a soccer field, a closed environment world was created in gazebo for the experimental testing of CalcioBot. In the experimental setup, the goal was placed near the corner of the map while the cube was placed much farther away from the goal in the near opposite direction. This specific setup choice was intended to showcase and examine the pushing ability of the robot as well as highlighting the effectiveness of the A* path searching algorithm.

When looking at the objects in the closed environment it is interesting to note the peculiar size and shape of the cube and the goal. The cube was purposefully designed to be on the smaller side to allow for CalcioBot to have better cube control when pushing it around the environment. During experimentation, we found out that having a larger cube could drastically impact the ability of the robot to push the object, but it could also influence the robot to veer off the given path from A* if the robot was too big or too heavy for CalcioBot to handle.

Another interesting experimental design choice is the goal post and how it hovers off the ground. Originally when the bottom half of the goal post was on the ground, the peculiar shape of the goal kept creating the issue of warning the robot about the empty cells in front of the goal which were filled. This phenomena made it hard for CalcioBot to record accurate estimates,if any, when detecting the goal. We also discovered upon much trial and error that removing the bottom half of the goal and having it "float" around allowed the robot to easily detect the goal while getting much better measurements.

Now that we have a better understanding of the experimental testing environment of CalcioBot, let us explore the actual experimentation methodology. In order to account for remaining time while still getting a decent sample size of the experimentation, a total of 20 experimental trials were conducted for CalcioBot. The measures of the outputs were described as followed:

1. Overall success rate (whether CalcioBot was able to seamlessly detect the cube and goal and push the cube to the goal correctly).

2. Average final distance of cube to goal (wherever the cube ended up at the end of the program, what was the distance between that point and the goal).

3. Misses goal (very close to overall success; however robot is not able to get cube in the goal).

4. Loses cube (while pushing the cube, the robot loses the cube).

5. Flips over/misses the cube (what is considered to be a complete failure in that the robot is immobile or does not even push the cube; this data was not included in avg. final distance of cube to goal since robot did not function properly).

trials = 20	**Observed values**
Success rate	40%
Avg. final dist. (cube to goal)	0.79 (m)
Missed goal rate	30%
Lost cube rate	15%
Flip-over/cube-miss rate	15%

Figure 1: Contains the experimental results of 20 trials run on Gazebo (rviz), including the percentage of occurrences for each row, and the average final distance from the cube to the goal.

If we take a look at Figure 1, we can see the breakdown of each of the outputs based on the 20 trials. We can see that the success rate was 40%. This means that the robot was able to deliver the cube to the goal, while detecting both the goal and the cube, and making a path from the cube to the goal and vice versa 8 times out of the 20 total attempts. Although the success rate was not as high as our ideal rate, we can still see that CalcioBot is able to perform its task perfectly almost half of the time.

In addition, we can see that even though the cube and the goal start off quite far from each other (almost greater than 4 m apart in the simulation), the average final distance of the cube to the goal in the 20 trials (minus the 3 failures) is around 0.79 m, which is commendable.

CalcioBot performs all of its tasks well except for pushing the cube in the goal 30% of the trials, meaning that CalcioBot was close to performing its tasks perfectly 6 out of 20 times, indicating that maybe some tuning or other adjustments could have resulted in the CalcioBot netting the cube in the goal an additional 6 times.

Out of the 20 trials, CalcioBot had the same number of times it lost the cube as it had failures .Failures included the robot flipping over or simply missing the cube from the initial position. These 6 occurrences could primarily be due to the miscalculation of rotation, leading to the robot to initially contact the cube in such a way that losing the cube would be inevitable.For example, if the robot contacted the cube on the right side while it is about to turn left, then that would result in the robot more often than not missing the cube. In the same way, the wrong angular rotation could result in the robot's wheel being stuck on the cube (which is quite small), leading the wheel to climb the cube and causing the robot to flip over. Lastly, overshooting and undershooting angular rotations more often than not can cause CalcioBot to fall off the path and miss the initial contact with the cube. Therefore, one of the major reasons that can explain mishaps with the success of the CalcioBot can be attributed to tuning as well as some potential issues with the rotation of the robot.

Overall, these experiments indicate that the proposed method was quite effective. With some further tuning the initial 40% success rate could also include the 30% missed goal rate, which would lead to a 70% success rate solely based on 20 trials. Therefore, with the addition of more trials and possibly more optimization when regarding tuning, the effectiveness of the proposed method could be much higher than the observed 40% from the 20 trials.

VI. Conclusion

In this project, the CalcioBot, we aimed to program to detect a targeted object and push it into a specified goal point. Throughout the implementation of the problem, we had to make many compromises and design changes to solve the problem. We went from a fully automated system to one requiring initial user input. Throughout the process, we uncovered many discoveries over the Husarion ROSBot, Gazebo and modern robotics as a whole.

We have many potential extensions with this project as a form of future work. For instance, we would implement a different PID controller (such as the Li-Zell controller) instead of the current PD controller in order to get better movement and responses from the robot both in the simulation and real life. We could also try to generalize the color detection messages from the RGB-D sensors in hopes of getting the robot to better capture its real life surroundings. Of course, greater optimization in tuning may lead to more accurate responses from the robot. Finally, the use of more elaborate sensors such as LIDAR may lead to more interesting implementations and solutions for the soccer problem.

References

[1] N. Koenig and A. Howard, "Design and use paradigms for Gazebo, an open-source multi-robot simulator," in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems*, vol. 3, pp. 2149–2154, 2004.

[2] "URDF" in *ROS Wiki*, May 2020.

[3] J. Stüber, C. Zito and R. Stolkin, "Let's Push Things Forward: A Survey on Robot Pushing," in *Frontiers in Robotics and AI*, vol. 7, no. 8, pp. 24–41, 2020.

[4] X. Li and A. Zell, "Path following control for a mobile robot pushing a ball," in *IFAC Proceedings Volumes*, vol. 39, no. 15, pp. 49–54, 2006.

[5] F. Mustiere, M. Bolic and M. Bouchard, "Rao-Blackwellised Particle Filters: Examples of Applications." *2006 Canadian Conference on Electrical and Computer Engineering*, 2006, pp. 1196-1200

[6] G. Grisetti, C. Stachniss and W. Burgard, "Improving Grid-Based SLAM with Rao-Blackwellized Particle Filters by Adaptive Proposals and Selective Resampling." *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, 2005, pp. 2432–2437.

[7] P. Fankhauser and M. Hutter, "A Universal Grid Map Library: Implementation and Use Case for Rough Terrain Navigation," in *Robot Operating System (ROS)*, pp. 99–120, 2016.

[8] M. Bjelonic, "YOLO ROS: Real-Time Object Detection for ROS" on *GitHub*, 2016–2018.

[9] T.-Y. Lin et al., "Microsoft COCO: Common Objects in Context" in *Computer Vision – ECCV 2014*, pp. 740–755, 2014.

[10] G. Bradski, "The OpenCV Library," in *Dr. Dobb's Journal of Software Tools*, pp. 120–123, 2000.

[11] P. E. Hart, N. J. Nilsson and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," in *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[12] S. LaValle and J. J. Kuffner. "Rapidly-Exploring Random Trees: Progress and Prospects," in *Algorithmic and Computational Robotics: New Directions*, pp. 293–308, 2000.

[13] A. A. Hagberg, D. A. Schult and P. J. Swart, "Exploring Network Structure, Dynamics and Function using NetworkX," in *Proceedings of the 7th Python in Science Conference*, pp. 11–15, 2008.