Collision-Free Swarm Motion Planning with Control Barrier Functions

Aaron Ray

May 15, 2019

1 Abstract

We demonstrate collision avoidance based on the safety barrier function paradigm with a fleet of Duckiebots. As robots' size and cost have shrunk and their capabilities expanded, swarms of robots have become viable solutions to a variety of problems such as exploration and delivery by exploiting the parallelism inherent in these tasks. Regardless of the high-level autonomy required for a task, the swarm of robots must navigate without its agents colliding. In some cases, such collision-free trajectory planning can be incorporated into the higher-level algorithm. However, testing and deployment of new algorithms is easier if collision-free motion can be guaranteed by a lower-level set of control software. We demonstrate that safety barrier functions can provide a locally-optimal collision-free trajectory that is agnostic to the higher-level autonomy guiding the system.

2 Introduction

Decreases in size and cost of hardware have made large scale distributed autonomous systems a feasible solution to an increasing number of problems. Some of these systems are fixed, such as long-term environmental monitoring, but others are dynamic, such as drone swarms for mapping disaster sites. Often large numbers of small agents can accomplish a task more efficiently than an individual agent. In these cases, the agents must coordinate their navigation so that they will not run into each other. For systems with slow dynamics, this problem is relatively simple. As long as each robot does not direct its current velocity directly at the position of another robot, collision can be avoided. However, as the dynamics of the robots become faster or more complicated, a naive [1] collision avoidance scheme may allow robots to find themselves in states where they do not have the control authority to avoid the collisions. For example, an autonomous car that realizes there is a stopped car in front of it when it is only a foot away does not give it enough time to slow down and avoid collision. A naive collision avoidance algorithm also may not find the "best" safe trajectory. An algorithm that prevents agents from moving successfully prevents collisions, but it would not be very useful. We need an algorithm that prevents collisions while altering the commanded velocities sent to all agents in the swarm as little as possible. In other words, at each time step, we want the safe set of velocity commands sent to the whole swarm to be as close as possible to the original input.

While the original scope of this project included extending the capabilities of existing collision-avoidance algorithms, slow progress toward the hardware demonstration resulted in a narrowing of the project's scope. The project's goal has been modified to focus on implementing existing algorithms for multi-robot collision avoidance on a group of robots.

We use the control barrier function (CBF) paradigm to ensure safe operation of general robot control algorithms and higher-level autonomy. We demonstrate this algorithm on a four Duckietown Duckiebots. We show that the control barrier function allows us to use a very naive proportional controller (P controller) to generate desired trajectories for the robots that are successfully modified by the control barrier function to prevent collision. The trajectories from the naive P controller would result in inter-robot collisions. The CBF

[1] Naive in the sense that the sole objective is preventing collision in a local/greedy manner

filters these commands to prevent the collisions without having to explicitly design any other inter-robot coordination.

The control pipeline for these Duckiebots is first validated in simulation with a small group of four agents, and a larger group of 32 agents. While only four Duckiebots are available for physical testing, it is important to verify that the control solution extends to larger swarms. After showing in simulation that the CBF approach leads to collision-free motion, we demonstrate the algorithm on four real Duckiebots. Further improvement and tuning of the control algorithms are needed before useful performance information can be extracted from the Duckiebots, but we show that the Duckiebots can operate in a collision-free manner.

3 Related Work

A swarm of cooperating agents needs to ensure there is no collision between any pair. We want to filter all commands to the swarm so that the resulting state is still in the collision-free "safe set". This filter is called a "Control Barrier Function" (CBF). At the same time, there is some desired trajectory that each agent is trying to achieve. Thus the CBF should provide a set of control outputs that is as close as possible to the original inputs, while maintaining safety. At least two formulations of this constrained optimization problem have been presented in the literature [1], [2]. In [1], the authors demonstrate how sets of velocities that would result in imminent collision can be identified and treated as obstacles in the velocity space when motion planning. When considering a system modeled with double integrator dynamics (e.g. the system's acceleration is controlled, but there is no direct control over velocity), the agents can never enter a state where their control authority is insufficient to avoid collision. The algorithm from [2] is similar, but it focuses on a single integrator dynamics model[2] (the system has direct control over its velocity). For small, lightweight robots the single integrator model is a reasonable simplification, as we can assume the low-level motor controller can achieve any desired velocity on a very short timescale. The algorithm from [2] has been used in the "Robotarium" at Georgia Tech, a fleet of internet-connected robots that allows remote users to test multi-robot control algorithms.

The approach in [2] can be seen as a generalization of what is presented in [1]. The formulation of [2] is more directly applicable to our case of filtering desired commands, as [1] focuses on setting up a set of constraints that can be used to explicitly plan a full trajectory for the swarm. We choose to implement [2], as it better reflects our desire for a lightweight safety filter for use on top of higher-level autonomy algorithms.

4 Technical Approach

In order to ensure collision-free motion, we must first develop the motion model of the robots. The control barrier function formulation uses these dynamics to maintain safe distances between the robots. The specific equations necessary to implement the CBF are presented, followed by the full control pipeline run on the Duckiebots.

4.1 System Dynamics

For the sake of simplicity, the Duckiebots have been modeled with single-integrator dynamics. This means that inputs to the Duckiebot directly affect their velocity. Formally,

$$\dot{x}_i = u_i \tag{1}$$

$x_i \in \mathbb{R}^2$ represents the position in the plane of agent i, $\dot{x}_i \in \mathbb{R}^2$ represents its velocity, and $u_i \in \mathbb{R}^2$ represents the input. This formulation deviates from reality in two important ways. First, real systems follow double-integrator dynamics. Inputs are really forces (which control acceleration), so velocity cannot be adjusted instantaneously. However, for small, lightweight systems like the Duckiebots, velocity can be adjusted very quickly so single-integrator dynamics are a reasonable approximation and reduce the state space. The more important deviation from reality is the assumption in Equation 1 of holonomic control.

[2]The specific safety barrier function formulation presented in [2] focuses on single integrator dynamics, but their technique is extensible to more sophisticated dynamics models

A robot with holonomic drive can directly control each of its degrees of freedom. However, the Duckiebot can only drive forward and turn. Maintaining the single-integrator simplification but accounting for non-holonomic constraints, the system dynamics become

$$\begin{cases} \dot{x} = \langle v\cos\theta, v\sin\theta \rangle \\ \dot{\theta} = \omega \end{cases} \tag{2}$$

In this formulation, $v \in \mathbb{R}$ and $\omega \in \mathbb{R}$ represent the longitudinal and angular velocity commands. When a v and ω are both nonzero, the robot turns in an arc. As the inputs are given in discrete time, the robot moves in a series of arcs with finite length. Each of these motion can be approximated by its tangent *at the arc's midpoint*[3]. Thus we can consider the robot as able to move in any direction within some cone in front of the robot. In other words, while the instantaneous lateral velocity of the robot is 0 at all times and restricts the motion to a line, if we consider the motion achieved by finite time slices it can move in a two-dimensional subspace of the plane. Even considering this simplification, there are some velocities that are not achievable. A purely lateral velocity cannot be achieved, for example. In order to account for these seemingly unachievable trajectories, a second layer of controller is used to map desired velocities from $u \in \mathbb{R}^2$ to the arcs defined by v and ω that approximate the desired velocity. When the desired arc is not achievable, v is set to 0, and the robot rotates in place until its longitudinal axis aligns closely enough with the desired velocity that it can approximate the desired velocity with an arc. A more detailed description of this process is presented in Section 4.3.

We might worry that the robot deviating from the CBF command by turning in place will lead to a crash, as the system is not following the collision-free inputs that have been generated. However, when a robot turns in place, it cannot collide with another robot. Any robot that would be at risk of collision with the stopped robot would be guided around the stopped robot by the CBF algorithm as the distance between them shrank.

4.2 CBF

Consider of set of N agents with index set $\mathcal{M} = (1, 2, \ldots, N)$. Each agent follows the dynamics outlined above. We define the function $h_{ij}(x)$ as the squared distance between the safety bubbles of agent i and agent j. Formally,

$$h_{ij}(x) = \|x_i - x_j\|^2 - D_s^2 \tag{3}$$

D_s is the buffer radius that must be maintained between all pairs of agents. As long as $h_{ij}(x)$ is greater than zero, there are no collisions between agents. We want to find a sequence of control inputs that adhere as closely as possible to the original inputs while keeping this function positive.

We need to understand how h changes over time based on what control inputs are given to the system. Equation 4 combines the Jacobian of h with the control input u to give the relationship:

$$\frac{\partial h_{ij}(x)}{\partial t} = -2(x_i - x_j)u_i + 2(x_i - x_j)u_j \tag{4}$$

The control input u is projected along the relevant edge, and determines how quickly that edge grows or shrinks. The critical innovation, as described in [2], is to slow the rate decrease of h as h gets closer to 0. Specifically, we require that

$$-2(x_i - x_j) \cdot u_i + 2(x_i - x_j) \cdot u_j \leq \gamma h_{ij}(x) \text{ for all } i \neq j \tag{5}$$

Here γ is a small positive constant, which we will refer to as the "safety parameter." As h_{ij} goes to zero, the maximum allowable rate of inter-robot distance decrease also goes to zero. When a pair of robots are far apart, $\gamma h_{ij}(x)$ is larger and does not constrain choices of u. When the robots are close, $\gamma h_{ij}(x)$ is small and u cannot be chosen to shrink the inter-robot distance very quickly. γ is a tunable parameter affected by

[3]In the small timestep limit as the discrete dynamics approach continuous dynamics, the tangent of all arcs approach the robot's longitudinal direction and the robot is confined to motion along a line. This is why it's important to consider this approximation in the context of discrete time

controller discretization and motion model uncertainty. In a continuous system, γ could be arbitrarily large, and robots would never collide. However, in a discrete system, a large γ can lead to a large control input sending h_{ij} from a large-magnitude negative number number to a positive number within a single timestep. In the testing for this project, γ was meaningful within a range of $\tilde{0}$-1.

These constraints are placed into a matrix A, such that each row of A represents one pairwise distance within the swarm. Let A_{ij} denote the row of A corresponding to the distance between robots i and j. Then

$$A_{ij} = [0, \dots, -2(x_i - x_j)^T, \dots, 2(x_i - x_j)^T, \dots, 0] \tag{6}$$

We also define

$$b_{ij} = \gamma h_{ij}(x) \tag{7}$$

This is based on the constraint in Equation 5

We can now write the constraints from Equation 5 as

$$Au \leq b \tag{8}$$

As the inputs to the robot must be bounded, we must also express constraints on the maximum possible control input α to the robot

$$\|u_i\|_\infty \leq \alpha, \forall i \in \mathcal{M} \tag{9}$$

Now, we can formulate the collision avoidance problem as a quadratic program minimizing the sum of the square of the differences between the desired and commanded velocities:

$$\begin{aligned} u^* = \underset{u \in \mathcal{R}^{2N}}{\operatorname{argmin}} J(u) = \sum_{i=1}^{N} \|u_i - \hat{u}_i\|^2 \\ \text{s.t.} \qquad Au \leq b \\ \|u\|_\infty \leq \alpha \end{aligned} \tag{10}$$

This optimization problem can be solved very easily with any quadratic program solver, such as quadprog in Python. The speed of solving quadratic programs depends on the dimensionality of the state and the number of constraints. As there is a constraint on each edge in the swarm, the number of constraints grows quadratically with number of agents. While something to be wary of, it is not a dire concern because the solvers are fast enough to solve problems with a relatively large number of constraints. If there are enough agents to slow the solver down too much, the problem can probably be decomposed into smaller collision-avoidance problems as agents that are far from each other cannot run into each other, so the collision avoidance does not need to be run globally over the entire swarm.

As discussed in the previous section, the light weight and low speed of the Duckiebots mean that the single integrator approximation of the robot is reasonable. However, it is worth noting here that in the case of systems with slower dynamics where slow changes in velocity must be taken into account the CBF process can be extended to account for a larger state vector by modifying the safety function $h_{ij}(x)$ and by extension A which relates to the Jacobian of A.

4.3 Control Pipeline

Figure 1 outlines all of the distinct pieces involved with running the control barrier functions on the Duckiebot. A position command X^* originates from some higher-level autonomy algorithm. For testing purposes, X^* will be directly supplied by the user. The velocity commander node takes the commanded position and an estimate of the current position, and outputs a desired planar velocity for the robot. The velocity commander node will initially be a proportional (P) controller. It can be extended to a proportional and derivative (PD) controller if oscillations arise in practice. Adding an integral term for a full PID controller would cause problems, as will become evident shortly. The commanded planar velocity V is passed through

the control barrier function to ensure the commanded velocity will result in no collisions, modifying the desired command as necessary. The resulting "safe" velocity that is passed to the Duckiebot is denoted $\tilde{V}$ and is relative to the body-centered reference frame of the robot. The Duckiebot can only move along its x-axis and rotate around its center. Thus it cannot fully realize the commanded velocity $\tilde{V}$. $\tilde{V}$ is passed through a "motion mapper" that turns the planar velocity $\tilde{V}$ into a longitudinal velocity and angular rate, V_x and θ respectively. Algorithm 1 outlines how this transformation takes place. If the angle between the robot's x-axis and the desired velocity is less than θ_{max}, the robot moves forward and turns. The resulting motion approximates the desired velocity, as the robot turns to the desired direction relatively quickly. If the angle between the robot's x-axis and the desired velocity is larger than θ_{max}, the robot turns in the necessary direction without driving forward. Note that while the robot stopping to turn results in its executed velocity not matching the desired "safe" velocity, if it is stationary and not turning then it cannot cause any collisions and is by definition safe. The simple controller presented here is just a proportional controller. It would be possible to extend this to integral and derivative terms if the proportional control is insufficient. The structure of the controller also explains why it would be difficult to add an integral term to the velocity controller. When the motion mapper node commands pure rotation to align the robot's longitudinal axis with the desired velocity, the integral of the velocity error term would grow quite quickly. This integral windup results in a velocity command that is far too large when the vehicle finally aligns itself with the desired velocity direction and stops moving. The V_x and θ commands are passed to a low-level motor control node on the Duckiebot (already implemented as part of the Duckietown software stack). This low-level controller sets the servo motors that drive the wheels to angular velocities ω_1, ω_2. The resulting state X of the Duckiebot is observed by the motion capture rig as $\tilde{X}$ (in a real system an online localization algorithm on the robot can be directly swapped in for the motion tracker), closing the loop back to the velocity commander. When the control barrier function is implemented correctly[4] , it becomes impossible to issue a velocity command V that results in two robots colliding.

Algorithm 1 Motion Mapper

Require: P_v, P_θ: proportional gains for angular and longitudinal acceleration. θ_{max} : Angle errors larger than this value result in pure rotation commanded.

procedure MOTIONMAPPER($\tilde{V}$)
  $(\tilde{v}_x, \tilde{v}_y) \leftarrow \tilde{V}$
  $e_\theta = \text{atan2}(\tilde{v}_y, \tilde{v}_x)$
  if $|e_\theta| < \theta_{max}$ **then**
    $G_t = (\theta_{max} - |e_\theta|)/\theta_{max}$
  else‘
    $G_t = 0$
  $V_x = \tilde{v}_x * G_t * P_v$
  $\omega = e_\theta * P_\theta$
  return (V_x, ω)

5 Evaluation

5.1 Simulation

Before testing the control algorithm from the previous section on real robots, it is verified in simulation. The software outlined in Figure 1 was implemented in Python with simulated robot dynamics. Python's quadprog library solved the quadratic program necessary to optimize control inputs at each step. The most important tests of the algorithm involve trajectories of agents that would collide without intervention from the control barrier function. We arrange the agents with equal spacing in a circle as seen in the first panel of Figures 2 and 3. Each agent is directed to drive to the point at the opposite side of the circle from its

[4]As we have discussed, our dynamics model allowing planar velocities in arbitrary directions does not exactly match the real dynamics of the robot. Thus a "correct" implementation also involves setting the barrier radius around the robots to a large enough values that the slightly-degraded performance of the resulting control inputs does not result in crashes.

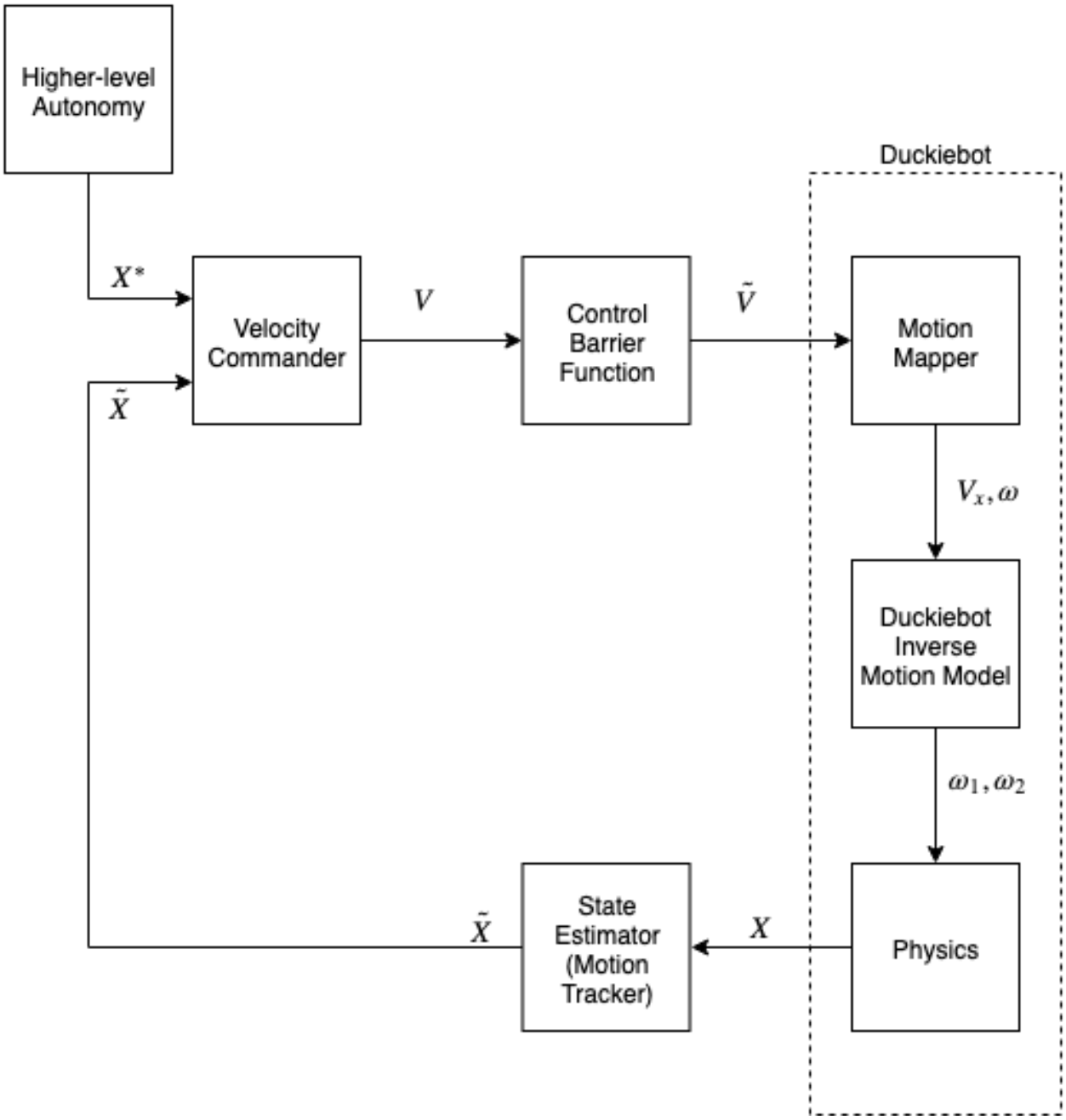

Figure 1: Schematic for the Duckiebot velocity control and control barrier function.

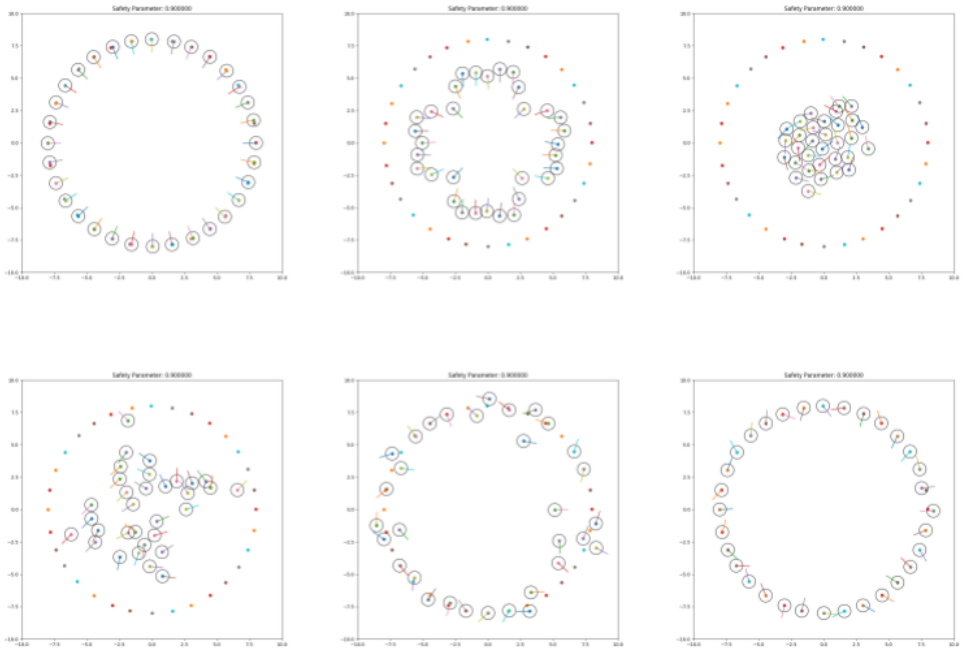

Figure 2: Example of control barrier function preventing collision between 32 simulated agents directed to move from their position on the circle to the position directly opposite. Safety parameter $\gamma = 0.9$.

initial position. As the position controller that decides desired velocity is very naive (at each step the desired velocity is directed directly toward the set point) without intervention all agents would collide at the center of the circle. However, Figures 3 and 2 demonstrate that the control barrier function successfully prevents the collisions and agents still end up at the desired final position. Figure 3 shows a value of $\gamma = 0.45$. This contrasts with the less conservative value of $\gamma = 0.9$ shown in Figure 2. In the former, a larger buffer is kept between the agents. In the latter plots, the control barrier function does not modify directed control inputs until the agents are much closer. In some instances, the robots violate the buffers they are supposed to respect (drawn as black circles). This violation occurs because of the simulation discretization. This shows that for low-rate controllers or uncertain dynamics, the value of γ needs to be tuned to be sufficiently conservative (i.e. low γ).

An interesting phenomenon arises when simulating agents in a symmetric arrangement such as the circle. If the simulated agents are arranged with perfect symmetry, they can get "stuck" sitting next to each other at a buffer-radius away from each other not moving. When agents get close to each other in the perfect-symmetry case, they can no longer make any progress toward their goal. *Any* control input with a component point toward the set point will be invalid. The quadratic program will find that the optimal solution is to have no agents move, because that is the viable control input closest to what is desired. The agents become trapped forever. Fortunately, this is an "artificial" problem because any numerical asymmetry leads to agents successfully navigating past each other. To fix this is simulation, a small amount of noise must be added to the starting positions. In real life, we never see perfect symmetry to machine precision so it is not an issue.

5.2 Real Robots

As testing 32 physical robots was logistically untenable, four Duckiebots were used as the physical test platform. The collision-avoiding Duckiebots provide a platform for testing higher-level distributed autonomy algorithms without having to worry about low-level collision avoidance. This goal can be considered accomplished if the control algorithm maintains a safe separation between robots. The user specifies what buffer radius the safety barrier function should respect, so we can validate that the distance between two

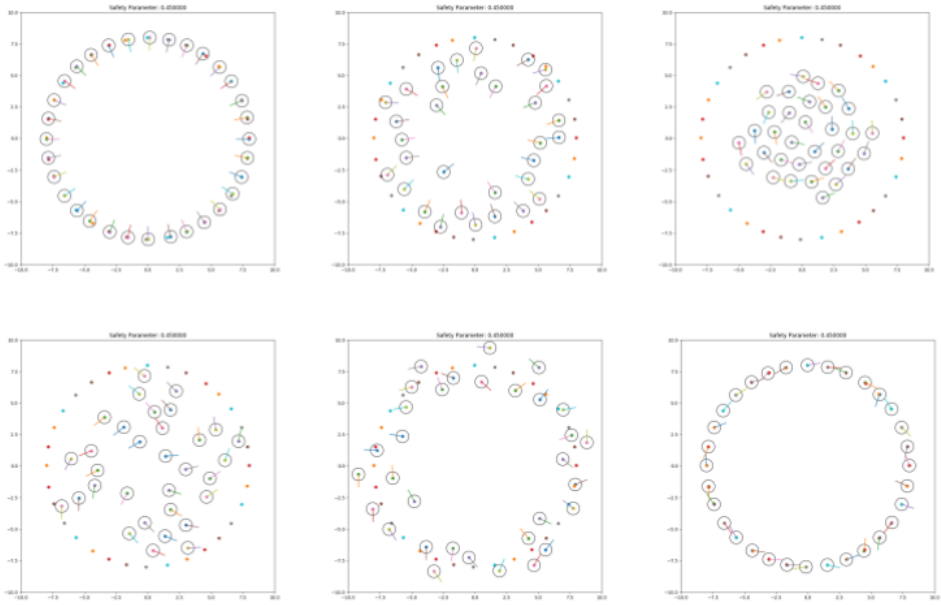

Figure 3: Example of control barrier function preventing collision between 32 simulated agents directed to move from their position on the circle to the position directly opposite. Safety parameter $\gamma = 0.45$.

robots never breaches this threshold. On the other hand, we want to ensure the corrected trajectory is as "efficient" as possible. If the original trajectory would have resulted in collision, then we can expect the optimal collision-free trajectory will result in robots passing each other at the minimum distance allowed by the safety buffer. If the minimum distance between robots is not approximately equal to the safety radius, then the algorithm is not tuned properly. We can also consider the amount of time required to complete the directed task compared to the amount of time a single agent would take in the absence of other obstacles. Inevitably the indirect paths required to avoid collisions will increase the required time for the task, but this increase should be relatively small. While we cannot give any specific bound on what this metric should be, qualitatively we the algorithm should be tuned to minimize this difference. Finally, the safety function should not alter the robot's trajectory in the absence of collision. This is easily tested by designing robot commands that do not result in collisions when executed faithfully.

Unfortunately, a lack of time prevented careful tuning of the Duckiebots, so these proposed quantitative metrics would not be very meaningful. Even in the single-agent case (i.e. one robot being directed to drive to a set point), the motion is not particularly smooth and direct. Thus there is no great baseline to compare with. We can only see that, qualitatively, the control algorithm prevents collisions between the Duckiebots. Figure 4 shows several frames from a successful test of the Duckiebots. As in the simulated example, each Duckiebot is directed to drive to the opposite side of the circle. They do so in a collision-free manner.

6 Conclusion

We have shown that a safety barrier function can augment a simple controller to ensure collision-free swarm behaviors. As this safety mechanism is agnostic to high level controller, it can be used with any higher-level motion or task planning algorithm. The designer of these higher-level algorithms do not have to explicitly worry about collision avoidance, speeding development and testing of new algorithms and behaviors.

In retrospect, the Duckiebots may not have been the best platform for the project. While they did provide a conveniently-packaged set of components that could carry out the required functionality, programming them has a higher overhead than hoped. In fact, it is still unclear to me how you are supposed to run code on them.

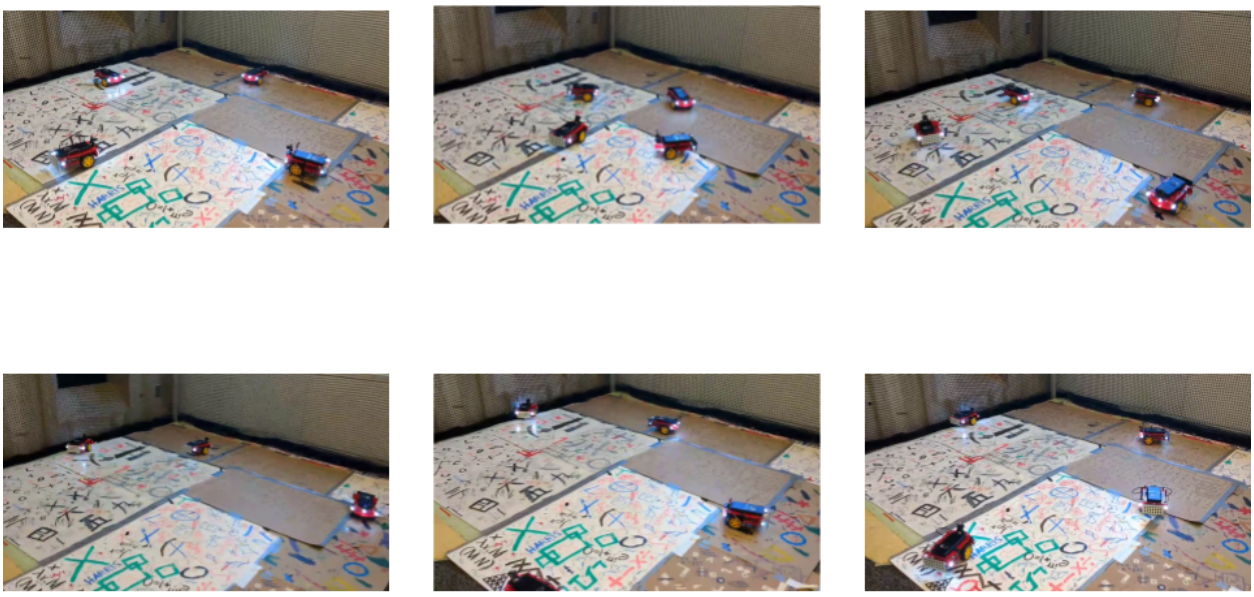

Figure 4: Demonstration of the control barrier function algorithm with a fleet of four duckiebots.

This project leveraged the fact that desired velocity could be sent over ROS, and all the control code was run on a base-station. The mix of Docker and ROS used on the Duckiebots may be beneficial when there are full-time engineering working with a large fleet of robots, but the large learning curve and software development overhead that this paradigm incurs makes is of questionable use for exploratory robotics research.

6.1 Future Work

The most direct extension to this work would be to improve the control algorithms used to determine desired velocities and mapping from commanded velocity to wheel speed/torque. As both of these are currently implemented as proportional controllers, they are not particularly robust. Extending these to PID controllers would improve the individual driving capabilities of each agent and make the system as a whole more compelling. This process would require tuning several parameters, which is why it was skipped for this project.

A downside of the system presented here is that relies on a motion capture system for ground truth position estimates for the Duckiebot fleet. It would be more compelling to see the robots drive around and avoid each other based on on-board state estimates. Each agent could run a SLAM algorithm for state estimation and they could share their state estimates to compute desired trajectories.

As the agents only need an estimate relative to each other (as opposed to a global state estimate) for collision avoidance, the fleet can also make use of a pairwise-distance based collision avoidance system. As described in [3], the relative positions and velocities of a swarm of agents can be estimated based only on measurements of the distance between each pair of agents and a knowledge of the intended motion of each agent. This method has advantages over SLAM-based approaches as the physical measurement of pairwise distance make it immune from the kidnapped robot problem and potentially decreases the computational requirements of the system by removing the need for expensive SLAM updates.

References

[1] Jur van den Berg, Ming C. Lin, and Dinesh Manocha. Reciprocal velocity obstacles for real-time multi-agent navigation. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2008.

[2] D. Pickem, P. Glotfelter, L. Wang, M. Mote, A. Ames, E. Feron, and M. Egerstedt. The robotarium: A remotely accessible swarm robotics research testbed. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1699–1706, May 2017.

[3] Aaron Ray. State estimation for dynamic relative localization systems. Bachelors thesis, Brown University School of Engineering, Providence, RI, 2019.