The Robotarium

Python Guide

Author:
Renato Maynard Etchepare
Sean Wilson [1]

Last Update:
November 20, 2024

Georgia Institute of Technology

[1]If you have a question, please contact Sean Wilson at Sean.Wilson@gtri.gatech.edu

Contents

List of Figures

List of Listings

1 The Robotarium

1.1 The Robotarium Project

The Robotarium project provides a remotely accessible swarm robotics research and education platform that remains freely accessible to anyone. Currently, robotics research and education requires significant investments in terms of manpower and resources to competitively participate. However, we believe that anyone with new, amazing ideas should be able to see their algorithms deployed on real robots, rather than purely simulated. In order to make this vision a reality, the Robotarium team has created a remote-access, robotics lab where anyone can upload and test their ideas on real robotic hardware.

1.2 How to Use the Robotarium Simulator

To execute your experiments on the Robotarium robots, download one of the simulators, prototype your code in simulation, and submit the exact same code for execution on actual robots through the Robotarium's web interface. You can get started right away using the Robotarium simulator provided here. A set of well-documented examples (that are also part of the provided simulators) will help you get started immediately. In a nutshell, completing the following three steps is all that is required to run your experiment on the Robotarium:

1. Download the Robotarium's MATLAB or Python simulator. (For the Robotarium MATLAB's guide please click here)
2. Write and verify your script functions correctly in the simulator.
3. Upload your scripts through your account on the Robotarium website.

1.3 How to use the Robotarium Hardware

Now that you tried your code in the simulator, if it does not output any potential errors, it is time to try to run your code on real robots!

1. Sign up for a Robotarium account and wait to get approved by one of the Robotarium's admins. This can take 1-2 business days!
2. Create your experiment in the web interface, upload your code, and submit it for execution.

1.3.1 Create a Robotarium Account

Creating a Robotarium account can be done in four (4) easy steps :

1. Go to the Robotarium website and navigate to the right corner, click in Sign Up

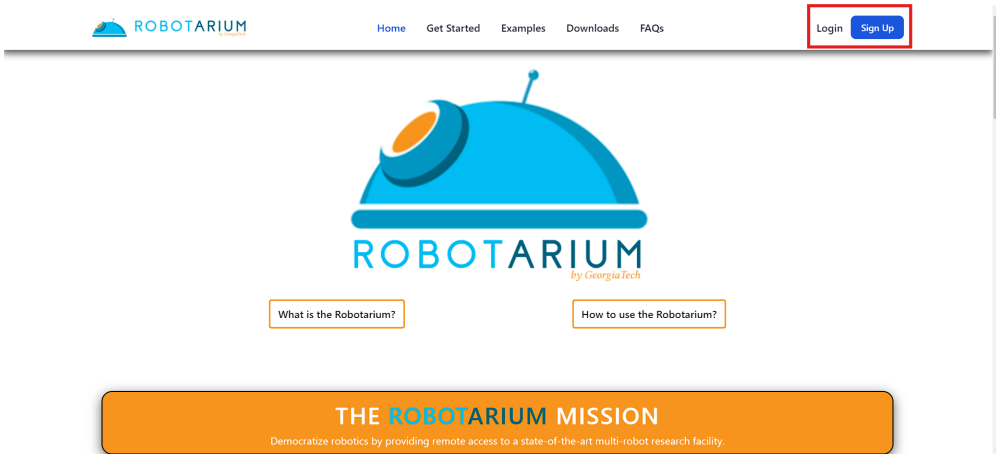
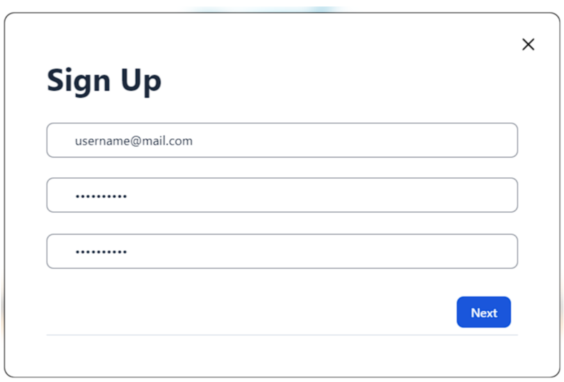
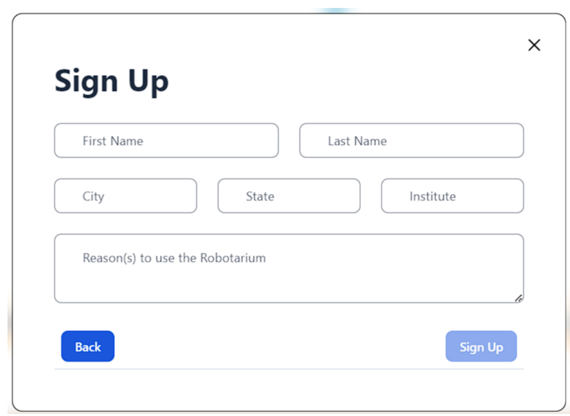


Figure 1: The landing page of the Robotarium website.

2. Complete the form with your email, password, and confirmation. Then, add your name, city, state, institute, and the reason you want to use the Robotarium. Please do not provide a fake e-mail as this is the way the Robotarium staff and system will contact you about your experiment status. We ask for your demographic information to keep track of *who* is using the Robotarium so it can be better developed to serve its user community. After creating your personal profile, you should see the message in Figure 2b, thanking you for signing up. If you do not receive a confirmation email within a few minutes, make sure to click on the "Resend Verification Link."



(a) Create your log in information by providing an e-mail and creating a password.



(b) Demographic information to complete.

Figure 2: Process to create a Robotarium user account.

3. In your inbox, you should receive a message asking you to verify your account. Click on the link to verify.

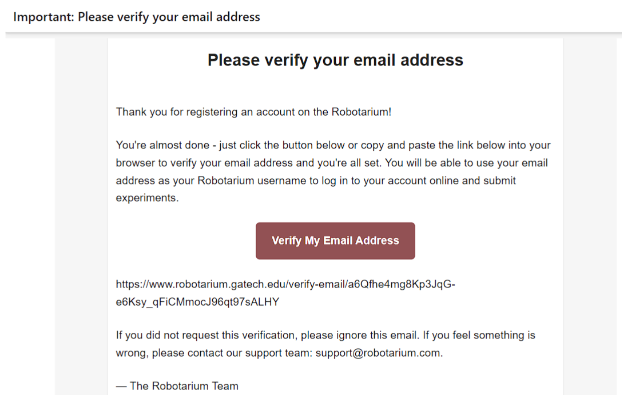


Figure 3: Verification Message

4. Wait for approval by the Robotarium admins (this should take about 1-2 business days) and then you will be able to submit code to the Robotarium using your personal user account!

1.3.2 Submit Code to Robotarium

After you have created a user account for the Robotarium and it has been approved, you can now submit code to the Robotarium whenever you want for it to be run on real robots. The submission steps are:

1. When you navigate to the dashboard, you will see the "Pending Experiments" and "Completed Experiments" tabs. If you have not run anything, it will look like Figure 4

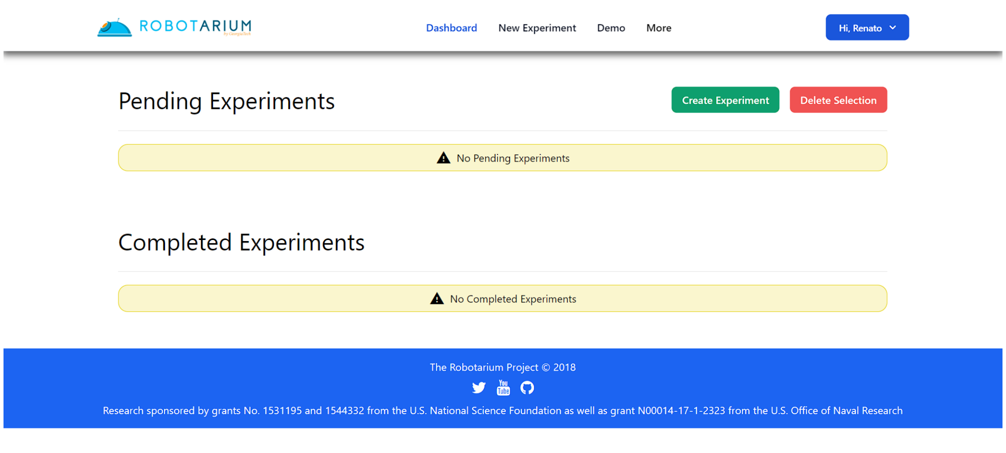


Figure 4: Dashboard

2. Click on the “Create Experiment” button. We will be submitting a file called “experiment.py” as an experiment to run on the Robotarium. You should see the “Define Your Experiment!” (Figure 5) heading at the top of the page. On this page, you will provide an experiment title, experiment description, estimated experiment duration, and number of robots you will need. Again, this information is used to understand what our users are deploying on the Robotarium to inform its development directions. Note, if the submitted experiment takes longer than your estimated duration to execute, we will still generate the video and you can view if there were any issues that occurred.

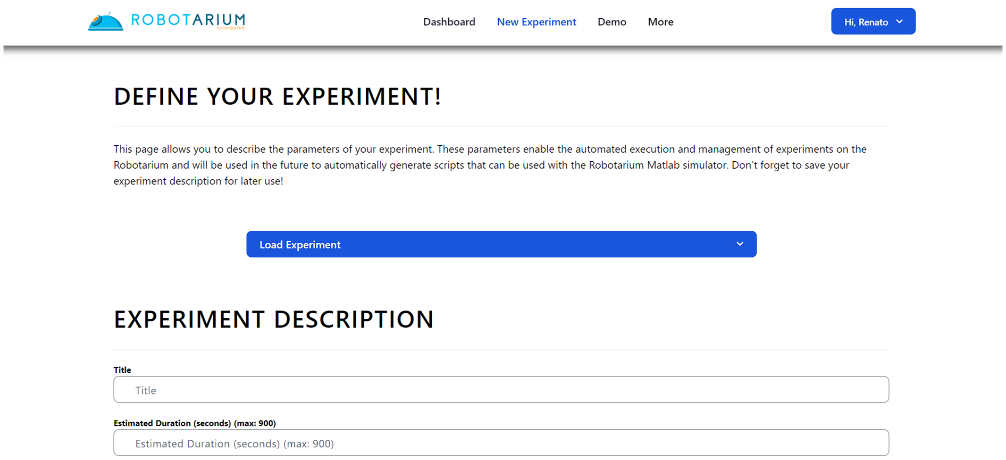


Figure 5: Define your Experiment

3. After you have defined your experiment, upload the associated experiment file(s). Note, do not upload any of the Robotarium files you downloaded with the simulator, only custom ones you have created. If your program required more than one file, please make sure you include all necessary files. For a single file or multiple files, indicate which is the ”main” file by clicking the box next to it as shown in Figure 6.

4. Once you are satisfied with the files you have included and the experiment description, click the green “Submit Experiment” button to submit the experiment. You also have the option to save the experiment and submit it for later and to download the .json file of the simulator parameters based on your input. When you have submitted the experiment, the dashboard should show your experiment in the “Pending Experiments” section, with the “Status” as “submitted” (Figure 7)

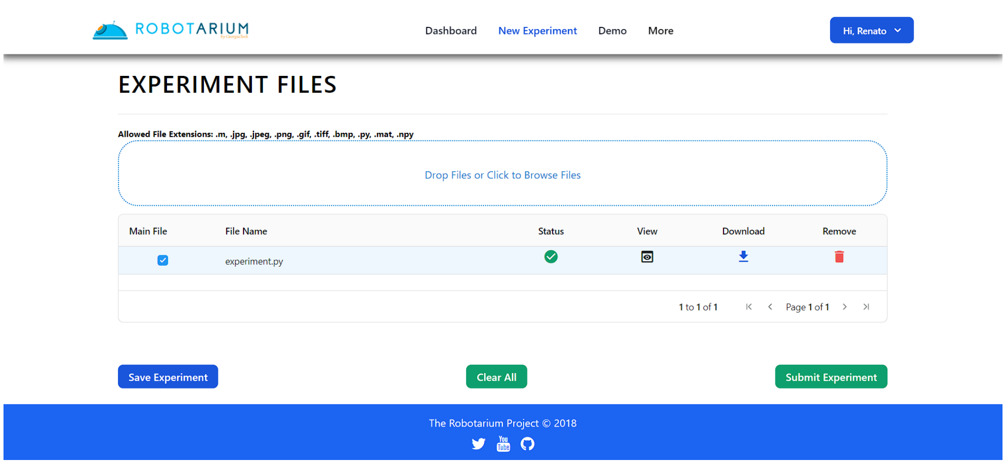


Figure 6: Submission of Experiment

5. Experiments take between 10 minutes and 2 days to execute, depending on the use traffic of the Robotarium. During that time, your experiment will be in the "Pending Experiment" dashboard as shown in Figure 7. You will get an email when your experiment has completed or if any issues were detected with the submission.

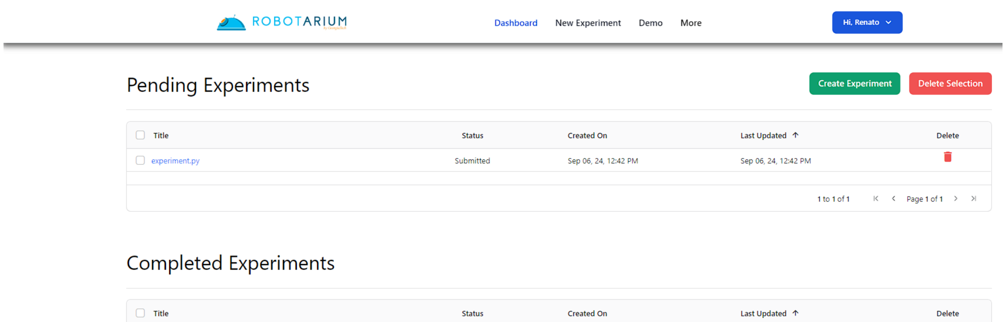


Figure 7: Pending Experiment

6. Once your experiment has been executed, you can view it in the "Completed Experiments" dashboard (Figure 8). By selecting the desired experiment, you will be taken to the results page, where you can find details such as the Experiment Settings, Experiment Video, Scripts, Return Messages, Data Files, and Log Files as shown in Figure 9.

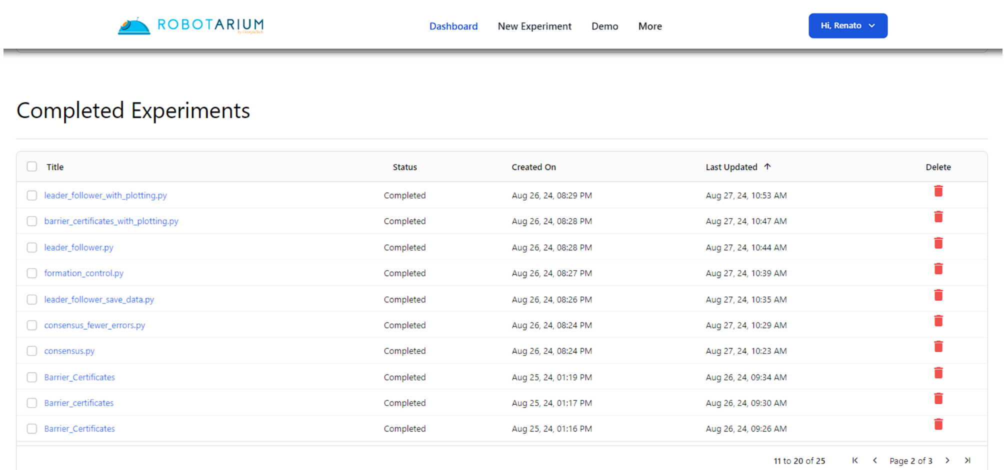
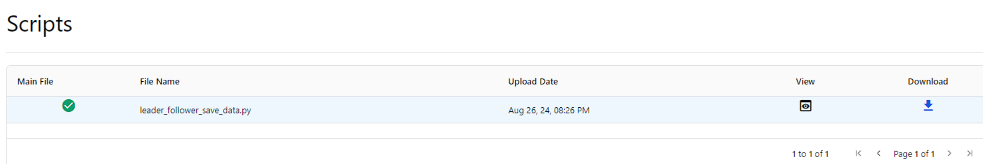


Figure 8: Completed Experiment

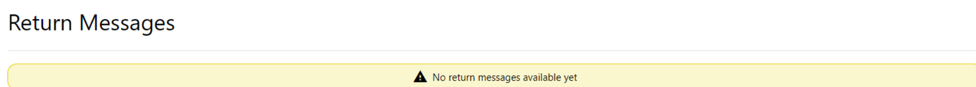
The figures below represent the different outputs you will receive when your experiment is completed. Check each image caption for more details.

(a) Experiment Setting. These are the experiment settings you chose to run your experiment. It is important to be aware of the estimated duration and the number of robots. A duration that is too short may interrupt your experiment, and having fewer or more robots than requested could cause issues in the experiment's execution.

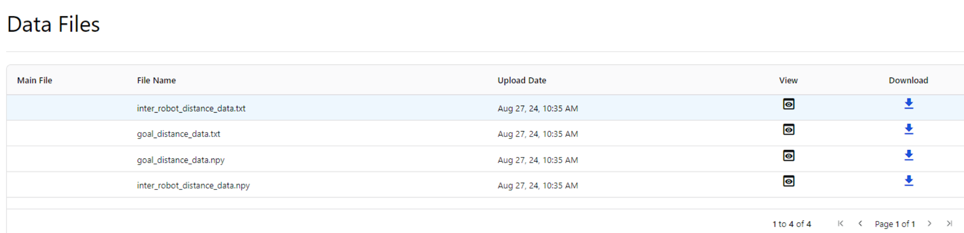
(b) Experiment Video. This is the video of your experiment, which you can watch on the website or download to your local device.



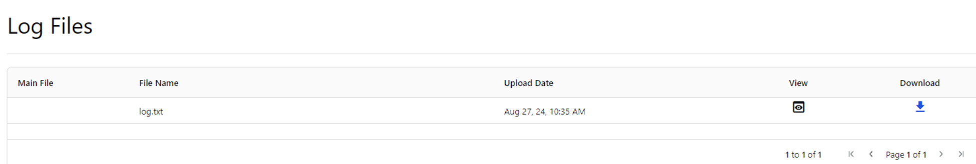
(c) Scripts. This is the code you uploaded to be run by the Robotarium.



(d) Return Messages. Here we can send you feedback regarding your experiment. This is rarely used, but depending on the experiment, we may utilize it.



(e) Data Files. Here we upload any files expected to be returned to you.



(f) Log Files. Any output printed to the console (stdout) will be recorded in the log.txt file of your submission. Error files will also be in this section

Figure 9: Experiment Outputs.

1.4 Robotarium References for Citation

If you use the Robotarium for research purposes and want to mention it in one of your publications, please cite one of the following references. If your research included Robotarium experiments after January 2019, the publication titled `The Robotarium: Globally Impactful Opportunities, Challenges, and Lessons Learned in Remote-Access, Distributed Control of Multirobot Systems` is most appropriate. The other citations are included for your reference.

1. Sean Wilson, Paul Glotfelter, Li Wang, et al. (2020). "The Robotarium: Globally Impactful Opportunities, Challenges, and Lessons Learned in Remote-Access, Distributed Control of Multirobot Systems". In: *IEEE Control Systems Magazine* 40.1, pp. 26–44. DOI: 10.1109/MCS.2019.2949973

2. Sean Wilson, Paul Glotfelter, Siddharth Mayya, et al. (2021). "The Robotarium: Automation of a Remotely Accessible, Multi-Robot Testbed". In: *IEEE Robotics and Automation Letters* 6.2, pp. 2922–2929. DOI: 10.1109/LRA.2021.3062796

3. Sean Wilson and Magnus Egerstedt (2023). "The Robotarium: A Remotely-Accessible, Multi-Robot Testbed for Control Research and Education". In: *IEEE Open Journal of Control Systems* 2, pp. 12–23. DOI: 10.1109/OJCSYS.2022.3231523

4. Daniel Pickem et al. (2017). "The Robotarium: A remotely accessible swarm robotics research testbed". In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1699–1706. DOI: 10.1109/ICRA.2017.7989200.

For information about the current robotic platform on the Robotarium, please refer to the following publication and Github repository.

1. Soobum Kim et al. (2024b). "GTernal: A robot design for the autonomous operation of a multi-robot research testbed". In: *International Symposium on Distributed Autonomous Robotic Systems*

2. Soobum Kim et al. (Sept. 2024a). *GTernal*. URL: https://github.com/robotarium/GTernal

If you have any questions, feel free to contact Sean Wilson.

2 Prerequisites

This section provides intuition and then covers the mathematical principles used in the Robotarium for modeling and control of the robots. While it provides valuable insights, it is not essential to read in detail to use the Robotarium unless you are interested in the mathematical underpinnings.

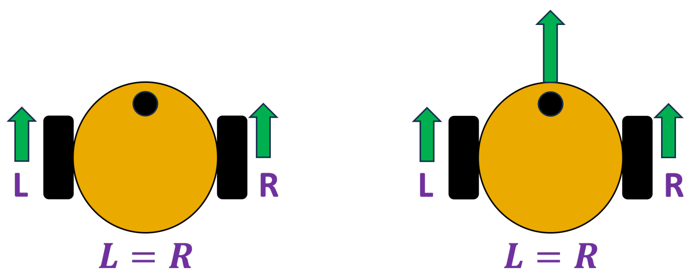
2.1 Differential-Drive Robots

The Robotarium allows for the simultaneous control of up to 20 differential-drive robots called GTernals. These robots are equipped with two parallel wheels that can be moved independently. This design allows the robots to perform a variety of maneuvers, such as moving forward, backward, and rotating in place.

The way to control the robots at the Robotarium is each robot used with a linear and angular velocity to execute.

2.1.1 GTernals Linear Velocity

To move the robot straight, both wheels must move at the same speed in the same direction. When this occurs, the robot ideally drive perfectly straight. We call this a linear velocity, v, as shown in Figure 16.



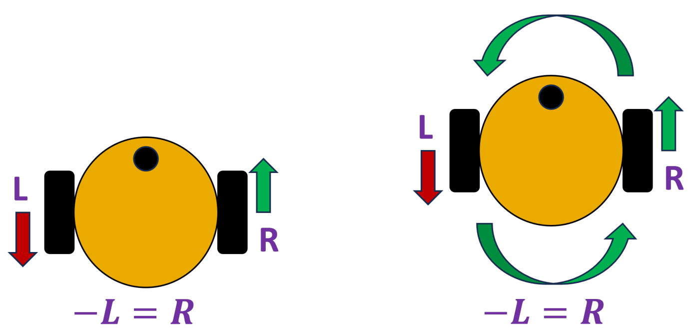
(a) Robot moving both wheels at the same speed in the same direction.

(b) Resulting ideal motion of the robot.

Figure 10: Linear velocity for differential drive robots.

2.1.2 GTernals Angular Velocity

To change the robot's heading, the wheels need to rotate in opposite directions, with one wheel moving forward and the other backward. If we rotate the motors in opposite directions at the same rate, the robot will spin without translating at all. We call this pure rotation an angular velocity, as shown in Figure 11



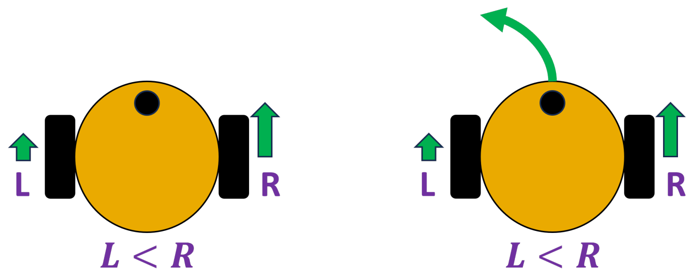
(a) Robot moving both wheels at the same speed in opposite direction.

(b) Resulting ideal motion of the robot.

Figure 11: Angular velocity for differential drive robots.

2.1.3 GTernals Linear and Angular Velocity

When both linear and angular velocities are combined, the robot can perform more complex maneuvers, such as moving in curved trajectories. This is achieved by having the wheels rotate at different speeds, as demonstrated in Figure 12.



(a) Robot moving both wheels at different speeds in the same direction.

(b) Resulting ideal motion of the robot.

Figure 12: Linear and Angular Velocity for GTernals

2.2 Open Loop vs Close Loop

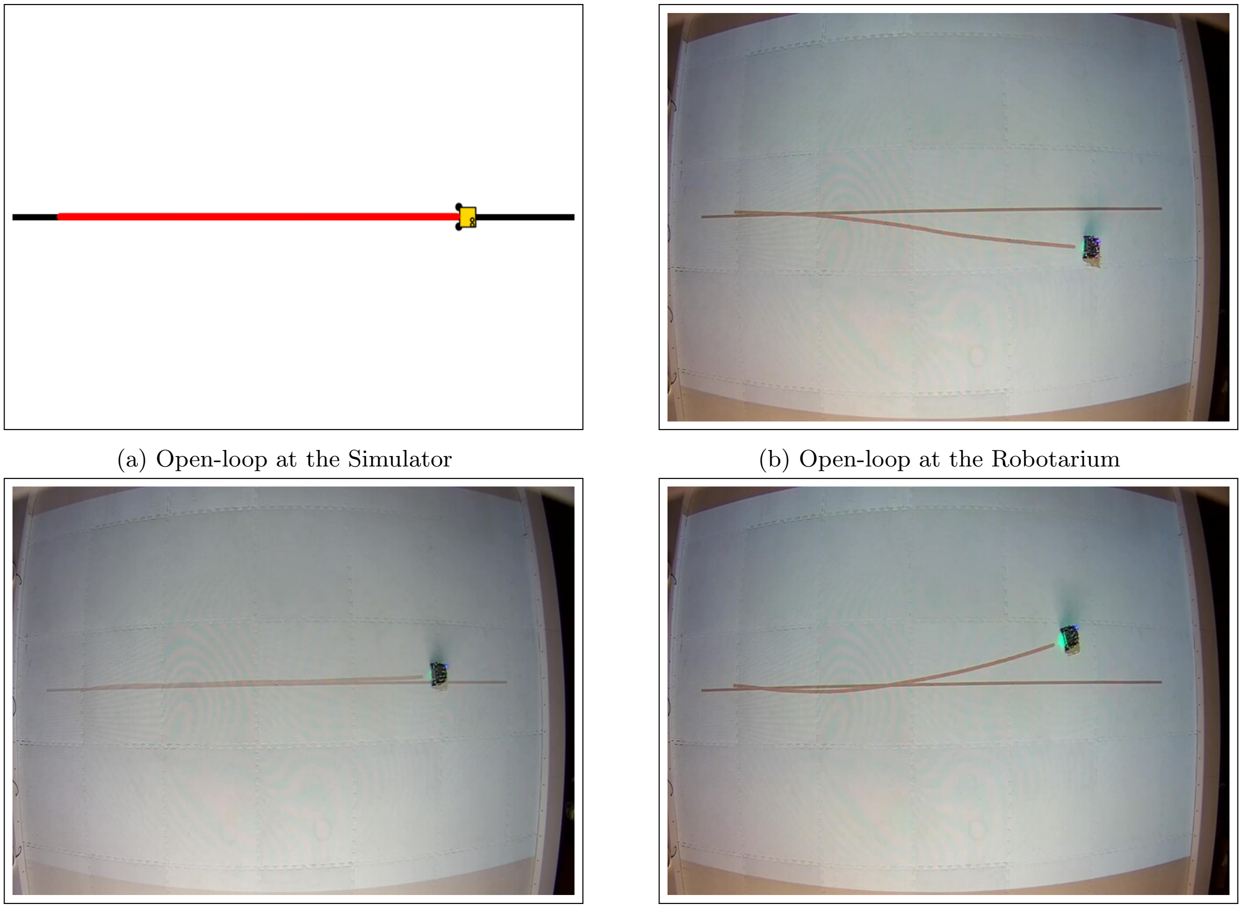
Let's imagine we have a trajectory (e.g. drive in a straight line) that enables a GTernal to reach its goal. To make the robot follow that trajectory, we can approach the problem in two ways:

- **Open-Loop Control**: In this method, the robot follows the path "blindly" by applying pre-computed control inputs and hoping nothing will go wrong. The robot executes predefined commands without considering any feedback from its environment. This approach assumes the robot will follow the path perfectly, without deviations.
- **Closed-Loop Control**: In closed-loop control, the robot follows the path for a short period, then "observes" whether it has deviated at all. Based on this feedback, the robot recomputes and adapts its control inputs to stay on track. This ensures that even if the robot experiences disturbances, it can adjust and stay on its desired trajectory.

Let's explore an example of the implications of each approach. Imagine we have a robot, and we instruct it to follow a straight line by giving it a constant linear velocity in an open-loop control scenario. What do you think would happen?

With this open-loop command, the robot would initially follow the intended straight path. However, if any disturbances occur—such as uneven terrain, wheel slippage, or external forces—the robot will not be able to correct its course. Additionally, there may be effects like actuator dynamic or communication delays that can cause the robot to behave differently than expected. As a result, the robot's trajectory would gradually deviate from the straight line. This is because open-loop control assumes nothing will go wrong, so the robot blindly executes the pre-defined commands. This can be seen in Figure 13. Notice in the perfect simulation, everything works out and the robot drives along the straight line but in reality where there are unexpected complications, the constant linear velocity does not result in straight motion. This is one of the many reasons to use the Robotarium, robots are extremely complex consisting of many interacting systems that can be hard to model mathematically and in simulation. Using the Robotarium exposes your developed algorithms to those complexities and allows you to validate your ideas in hardware.

If you want to try this experiment, you can run the following code on the Robotarium. This script instructs a GTernal to follow a straight-line trajectory by setting a constant linear velocity. In the simulation, the robot will follow the path perfectly since the simulator does not consider real-world disturbances. However, when you execute this code in the Robotarium environment, you'll notice how perturbations, such as friction or slight imperfections in the motors, influence the robot's trajectory, preventing it from following a perfect straight line.



(a) Open-loop at the Simulator

(b) Open-loop at the Robotarium

(c) Open-loop at the Robotarium

(d) Open-loop at the Robotarium

Figure 13: Simulator vs Real Life Open-Loop Control. The black line is the path the robot should take and the red line is what the robot actually drove. You can see this example in our YouTube channel under the video called "Open Loop Control: Simulation vs Reality"

If the robot used closed-loop control, it would continuously monitor its position and correct its trajectory in response to any disturbances. For example, if the robot veers slightly off the straight path, it would detect the deviation and adjust its wheels' speeds to bring it back on track. This makes closed-loop control more robust in dynamic environments or when disturbances are likely. While we do not provide an example to do this in this part of the guide, you are welcome to try to develop your own or use one of the many controllers listed below.

```python
import rps.robotarium as robotarium
from rps.utilities.transformations import *
from rps.utilities.barrier_certificates import *
from rps.utilities.misc import *
from rps.utilities.controllers import *
import numpy as np
import time

N = 1 # Number of Robots

init_pos = np.array([-1.3,0,0]).reshape(3,1) # Initial Positions

iterations = 450 # # Run the simulation/experiment for 450 steps

r = robotarium.Robotarium(number_of_robots=N, show_figure=True, initial_conditions=init_pos, sim_in_real_time=True) # Instantiate the Robotarium

line_width = 5 # Big lines
position_history = np.empty((2,0)) # History to show what the robot does
r.axes.plot([-1.6,1.6],[0,0],linewidth=line_width,color='k',zorder=-1) # Plot reference line

for t in range(iterations):

    x = r.get_poses() # Get the poses of robots

    dxu = np.array([0.15,0]).reshape(2,1) # Define the Speed of the robot

    r.set_velocities(np.arange(N), dxu) # Set the velocities using unicycle commands

    # Plotting the robot's true trajectory.
    position_history=np.append(position_history, x[:2],axis=1)
    if(t == iterations-1):
        r.axes.scatter(position_history[0,:],position_history[1,:], s=1, linewidth=line_width, color='r', linestyle='dashed')

    r.step() # Iterate the simulation

time.sleep(5)

r.call_at_scripts_end() # Call at end of script to print debug information
```

Listing 1: Example code for a robot moving in a straight line.

2.3 Differential-Drive Kinematics

Kinematics is a branch of mechanics that deals with the motion of objects without considering the forces that cause the motion. There are two main problems Kinematics aims to solve *Forward Kinematics* and *Inverse Kinematics*.

- Forward kinematics answers the following question. Given the initial state of the system $\boldsymbol{x_0}$ (e.g. robot pose $\boldsymbol{x_0}=[x(0), y(0), \theta(0)]^T$) and input controls to the system (e.g velocity), find the final state of the system $\boldsymbol{x_t}$ (e.g. robot pose $\boldsymbol{x_t}=[x(t), y(t), \theta(t)]^T$) reached by the robot.

- Inverse kinematics answers the following question. Given the initial state of the system $\boldsymbol{x_0}$ (e.g. robot pose $\boldsymbol{x_0}=[x(0), y(0), \theta(0)]^T$) and a desired final state of the system $\boldsymbol{x_t}$ (e.g. robot pose $\boldsymbol{x_t}=[x(t), y(t), \theta(t)]^T$), find the input controls to the system (e.g velocity),

Note that in general, inverse kinematics is a much harder problem than forward kinematics.

Forward Kinematics for Differential-Drive Robots

A differential-drive robot has two actuators (left wheel speed $\dot{\phi}_l$ and right wheel speed $\dot{\phi}_r$). Through the geometric relation of the wheel locations to the robot body and assuming there is no wheel slip and only motion in the direction the wheel spins, we can determine the entire robot's forward velocity and rotational velocity [2].

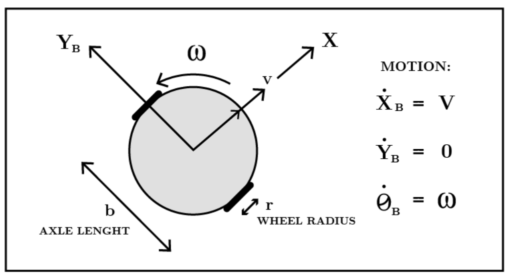


Figure 14: Body Frame Kinematics

Forward Velocity

$$v = \frac{r}{2}(\dot{\phi}_l + \dot{\phi}_r) \quad \text{where } r \text{ is the radius of the wheel}$$

Rotational Velocity

$$w = \frac{r}{d}(\dot{\phi}_r - \dot{\phi}_l) \quad \text{where } d \text{ is the axle length}$$

Then, the motion of the robot in a can be described as

$$\begin{pmatrix} \dot{x}_B \\ \dot{y}_B \\ \dot{\theta}_B \end{pmatrix} = \begin{pmatrix} v \\ 0 \\ w \end{pmatrix}$$

[2] Siegwart, R., Nourbakhsh, I. R., Scaramuzza, D. (2011). Introduction to autonomous mobile robots (2nd ed.). MIT Press.

This is a motion model that describe how the robot state change in its own local coordinate system. We can notice that $y = 0$ since we have assumed the wheels cannot slip. Therefore, we have a motion constraint. This motion constraint is similar to cars and bicycles that cannot immediately drive left or right.

Now, we are going to use this motion model to create a system of equations that describes the robot's state change in a global coordinate (e.g. with respect to the coordinate system of the Robotarium). In other words, given a known control inputs , how does the robot move with respect to a **global coordinate system**?

In order to do that, we will use a rotational matrix $R(\theta)$

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = R(\theta) \begin{pmatrix} \dot{x}_B \\ \dot{y}_B \\ \dot{\theta}_B \end{pmatrix}$$

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = \begin{pmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \dot{x}_B \\ \dot{y}_B \\ \dot{\theta}_B \end{pmatrix}$$

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = \begin{pmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} v \\ 0 \\ w \end{pmatrix}$$

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = \begin{pmatrix} v\ cos(\theta) \\ v\ sin(\theta) \\ w \end{pmatrix}$$

Now, giving inputs u and w and the orientation θ, I can tell how the robot moves in the world frame.

Inverse Kinematics

While forward kinematics are useful. Typically, we are more interested in answering the question, given a desired motion in the global coordinate system, what should the control inputs be to move in that direction?

Deriving the formula used previously, we can invert the previous equations to find the **control inputs**

$$\begin{pmatrix} v \\ 0 \\ w \end{pmatrix} = R^{-1}(\theta) \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix}$$

$$\begin{pmatrix} v \\ 0 \\ w \end{pmatrix} = \begin{pmatrix} cos(\theta) & sin(\theta) & 0 \\ -sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix}$$

$$\begin{pmatrix} v \\ 0 \\ w \end{pmatrix} = \begin{pmatrix} cos(\theta)\dot{x} + sin(\theta)\dot{y} \\ cos(\theta)\dot{x} - sin(\theta)\dot{y} \\ \dot{\theta} \end{pmatrix}$$

Then, we obtain that

$$v = \cos(\theta)\dot{x} + \sin(\theta)\dot{y}$$
$$w = \dot{\theta}$$

and that

$$\dot{\phi}_l = \frac{v}{r} + \frac{wd}{2r} = \cos(\theta)\dot{x} + \sin(\theta)\dot{y} - \frac{\dot{\theta}d}{2r}$$
$$\dot{\phi}_r = \frac{v}{r} - \frac{wd}{2r} = \cos(\theta)\dot{x} + \sin(\theta)\dot{y} + \frac{\dot{\theta}d}{2r}$$

So now we can control the wheel speeds! But, (since our robot is non-holonomic), we have to add a constraint

$$\dot{x}cos(\theta) = \dot{y}\sin(\theta)$$

Being non-holonomic, implies that we cannot change the y_B in the robot's body-frame. If we want to change something in the y_B, we must change something in x_B in the robot body-frame. In order to do that, we must leverage a new trick, $feedback\ linearization$.

2.4 Dynamics in The Robotarium

The robots deployed on the Robotarium are differential drive robots. In simulation the motion of the robots in the global coordinate frame are modeled through the unicycle model.

2.4.1 Unicycle Dynamics

The unicycle model has two input quantities, the linear velocity (v) and the angular velocity (ω). The linear velocity has units m/s and the angular velocity has units radians/s. The set of equations that fully describes the velocities of the global pose (x, y, θ) are

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = \begin{pmatrix} v\ cos(\theta) \\ v\ sin(\theta) \\ \omega \end{pmatrix}$$

2.4.2 Single-Integrator Dynamics

The single integrator model is a simplified way to represent the robot's motion, treating its state as a point in space. In this model, the robot's position is represented by a point $p = [x, y] \in \mathbb{R}^2$, where x and y are the coordinates of the robot in a 2D plane. The control input $u \in \mathbb{R}^2$ consists of the x-velocity (v_x) and y-velocity (v_y), such that $u = [v_x, v_y]$.

The single integrator model's dynamics can be described by the following equations:

$$\dot{p} = u$$
$$y = p$$

In this model:

- $\dot{p}$ (the derivative of p) is directly controlled by u, meaning u represents the instantaneous velocities in the x and y directions.
- Since the output is directly the state p, this model abstracts away any orientation or turning dynamics, focusing solely on point-to-point motion.

This is called the single integrator model because the input u is the derivative of the state p. Integrating u over time gives the position of the robot, allowing us to control the robot's motion by specifying its desired velocities directly. This model simplifies control because it disregards complex dynamics and only considers how the velocities (v_x, v_y) affect the robot's position over time.

Near-Identity-Diffeomorphism[3] between the unicycle control model and single integrator control model

A *diffeomorphism* is a smooth, invertible map between two manifolds that has a smooth inverse. When we talk about a diffeomorphism between the unicycle control model and the single integrator control model, we're discussing a transformation that allows us to relate the two control models in a way that preserves the structure of each.

1. Unicycle Control Model (Section 2.4.1)
   - The robot has two control inputs: **linear velocity** v and **angular velocity** ω.
   - The state of the robot is given by its **position** (x, y) and **orientation** θ in the plane.

   The dynamics of the unicycle model are given by:

$$\dot{x} = v\cos(\theta)$$

$$\dot{y} = v\sin(\theta)$$

$$\dot{\theta} = \omega$$

2. Single Integrator Control Model (Section 2.4.2)
   - In contrast, the single integrator model is simpler. Here, we assume the control inputs directly affect the rate of change of the robot's position:

$$\dot{x} = u_x$$

$$\dot{y} = u_y$$

     where u_x and u_y are the control inputs in the x- and y-directions, respectively. This model is linear, as the controls u_x and u_y directly change x and y without any nonlinearities.

To control a unicycle-like robot as if it were a single integrator (where you can control x and y directly), we need a *near-identity-diffeomorphism* — a transformation that allows us to map the unicycle's controls (v, ω) to the equivalent controls (u_x, u_y) of a single integrator model.

We are trying to answer the following question. Given desired velocities in the single integrator model, how do we convert these into control inputs for a unicycle robot?

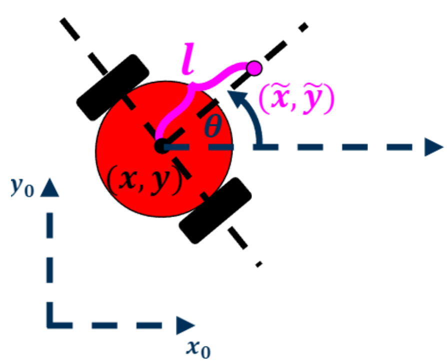


Figure 15: GTernal body-frame with a point with a distance l along the perpendicular bisector of the robot axle

[3] R. Olfati-Saber, "Near-identity diffeomorphisms and exponential /spl epsi/-tracking and /spl epsi/-stabilization of first-order nonholonomic SE(2) vehicles," Proceedings of the 2002 American Control Conference (IEEE Cat. No.CH37301), Anchorage, AK, USA, 2002, pp. 4690-4695 vol.6, doi: 10.1109/ACC.2002.1025398

In order to get the expressions, consider a point a distance l along the perpendicular bisector of the robot axle (Figure 15).

$$\tilde{x} = x + lcos(\theta)$$
$$\tilde{y} = y + lsin(\theta)$$

deriving the point with respect the time we obtain

$$\frac{\mathrm{d}\tilde{x}}{\mathrm{d}t} = \dot{\tilde{x}} = \dot{x} - l\theta sin(\theta)$$
$$\frac{\mathrm{d}\tilde{y}}{\mathrm{d}t} = \dot{\tilde{y}} = \dot{y} + l\theta sin(\theta)$$

replacing the unicycle dynamics previously calculated we have

$$\dot{\tilde{x}} = vcos(\theta) - l\omega sin(\theta)$$
$$\dot{\tilde{y}} = vsin(\theta) + l\omega cos(\theta)$$

These equations can be written as

$$\begin{pmatrix} \dot{\tilde{x}} \\ \dot{\tilde{y}} \end{pmatrix} = R(\theta) \begin{pmatrix} 1 & 0 \\ 0 & l \end{pmatrix} \begin{pmatrix} v \\ w \end{pmatrix}$$

where $R(\theta) = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix}$ is the rotational matrix.

The matrix is nearly always invertible, we get

$$\begin{pmatrix} v \\ w \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & \frac{1}{l} \end{pmatrix} R(-\theta) \begin{pmatrix} \dot{\tilde{x}} \\ \dot{\tilde{y}} \end{pmatrix}$$
$$\begin{pmatrix} v \\ w \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & \frac{1}{l} \end{pmatrix} \begin{pmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{pmatrix} \begin{pmatrix} \dot{\tilde{x}} \\ \dot{\tilde{y}} \end{pmatrix}$$
$$\begin{pmatrix} v \\ w \end{pmatrix} = \begin{pmatrix} \cos(\theta)\dot{\tilde{x}} + \sin(\theta)\dot{\tilde{y}} \\ \frac{-\sin(\theta)\dot{\tilde{x}} + \cos(\theta)\dot{\tilde{y}}}{l} \end{pmatrix}$$

Thus, we obtain the unicycle control inputs as:

$$v = \cos(\theta)\dot{\tilde{x}} + \sin(\theta)\dot{\tilde{y}}$$
$$w = \frac{-\sin(\theta)\dot{\tilde{x}} + \cos(\theta)\dot{\tilde{y}}}{l}$$

These equations allow us to convert the single integrator velocity commands (u_x, u_y) into the unicycle model's linear and angular velocities (v, w). The diffeomorphism between the unicycle and single integrator models allows us to treat a complex, nonlinear unicycle model as if it were a simpler, linear single integrator model for control purposes.

2.5 Velocity Clipping for the Unicycle Model

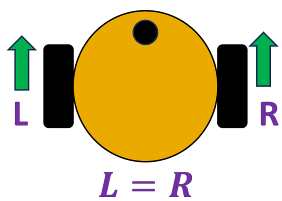
In expanding the diffeomorphism between the unicycle control model and the single integrator model, we address the concept of *velocity clipping*, which helps manage velocity constraints for a unicycle robot. This approach ensures that the robot stays within feasible motion limits while maintaining its intended trajectory shape.

1 **Maximum Linear and Angular Velocities**

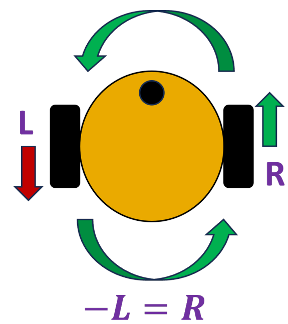
When controlling a unicycle robot, there are distinct maximums for **pure linear** and **pure angular** velocities:

- $v_{\max}$: Maximum linear velocity when only forward or backward motion is supplied.
- $w_{\max}$: Maximum angular velocity when only rotational motion is supplied.

These limits stem from the physical constraints of the motors that spin the robot's wheels, which restrict the achievable speeds regardless of motion type.



(a) GTernal pure linear



(b) GTernal pure angular

Figure 16: Pure linear vs Pure Angular for GTernals.

2 **Coupling Between Linear and Angular Velocities**

When both max linear and max angular velocities are applied simultaneously (e.g., moving forward while turning), they become **coupled** due to the wheel limits. The relationship is not straightforward, as it's an *under-constrained problem*, meaning multiple solutions could achieve feasible motion within the wheel limits.

3 **Applying Velocity Clipping**

If the combined linear and angular velocities exceed the wheel limits, we apply **velocity clipping**. This involves scaling both the linear and angular components proportionally, to ensure they fit within feasible bounds while preserving the desired path shape. The goal is to keep the arc that the robot would follow with unrestricted velocities.

The velocity clipping process can be summarized as:

$$v_{\text{scaled}} = \alpha\, v,$$
$$w_{\text{scaled}} = \alpha\, w,$$

where α is a scaling factor chosen such that the adjusted velocities v_{scaled} and w_{scaled} do not exceed the wheel constraints. This adjustment maintains the intended trajectory shape, respecting an additional constraint to preserve the "drivable" arc.

The robotarium automatically will threshold your motors commands. It is important to recall that the threshold does not means that your desire speed will be achieve, what the threshold will do it is keep the desire trajectory of the robot.

An example of this below.

2.5.1 Example effect of clipping

Recall the equation to control the wheels.

$$\dot{\phi}_r = \frac{v}{r} + \frac{wd}{2r}$$
$$\dot{\phi}_l = \frac{v}{r} - \frac{wd}{2r}$$

rearranging the terms, we can write the linear velocity v and the angular velocity ω in terms of $\dot{\phi}_r$ and $\dot{\phi}_l$

$$v = \frac{r}{2}(\dot{\phi}_r + \dot{\phi}_l)$$
$$\omega = \frac{r}{d}(\dot{\phi}_r - \dot{\phi}_l)$$

Now, let ϕ_{max} be the maximum angular velocity of the wheels. Thus, we impose the constraint

$$\|\dot{\phi}_r\| \leq \dot{\phi_{max}}$$
$$\|\dot{\phi}_l\| \leq \dot{\phi_{max}}$$

Let $V_{\max}$ be the maximum achievable linear velocity when there is **no rotation** ($\omega = 0$). The relationship between $V_{\max}$ and the maximum wheel speed $\dot{\phi}_{\max}$ is given by:

$$\dot{\phi}_{\max} = \frac{V_{\max}}{r}$$

$$V_{\max} = \dot{\phi}_{\max} \cdot r$$

This indicates that, in this case, we are using **only linear velocity** without any angular component.

Now consider $\omega_{\max}$ when there is **no linear velocity** ($v = 0$):

$$\dot{\phi}_{\max_r} = \frac{\omega_{\max} d}{2r}$$

$$\dot{\phi}_{\max_l} = \frac{-\omega_{\max} d}{2r}$$

This indicates that, in this case, we are using **only angular velocity**.

In these two cases, there is no problem: the robot performs as expected while staying within the limits. Now, let us demonstrate an extreme example.

Let's choose a v that is equal to the maximum linear speed, such that $v = V_{\max}$, and a sufficiently small angular speed ϵ.
If v is really big, and omega is normal, What it going to happen is that the equations

$$\dot{\phi}_r = \frac{V_{max}}{r} + \frac{\epsilon d}{2r}$$
$$\dot{\phi}_l = \frac{V_{max}}{r} - \frac{\epsilon d}{2r}$$

will become

$$\frac{V_{max}}{r} + \epsilon > \dot{\phi}_{max}$$
$$\frac{V_{max}}{r} - \epsilon > \dot{\phi}_{max}$$

where $\epsilon > 0$ is an small number

Since $v = V_{\max}$, the term $\frac{\omega d}{2r}$ will be negligible compared to $\frac{v}{r}$. Thus, we have

$$\dot{\phi}_r \approx \dot{\phi}_l \rightarrow \text{Robot drives straight}$$

if $v > \dot{\phi}_{max} r + \frac{wd}{2r}$, the robot will drive straight, no matter how big you make ω

If the desired linear velocity and angular velocity are exceed the maximum motor speed, then the robot will not move as expected because the robots will be thresholded.

In summary, while maximums $v_{\max}$ and $w_{\max}$ apply to purely linear or angular motion, the combined velocities can be restricted by wheel limits. To manage this, velocity clipping scales both velocities to keep the robot's path feasible, ensuring it stays within physical limits while following the intended trajectory.

2.6 Basic Code Structure

This section gives an example of the minimum code necessary for users to run their algorithms on the Robotarium. The code itself is non-operational (Eventually the robots will leave the test bed or crash between them), but it provides useful function calls and commands that users will need to call in order to run programs successfully. The code is discussed line-by-line below.

```
1  import rps.robotarium as robotarium
2  from rps.utilities.transformations import *
3  from rps.utilities.barrier_certificates import *
4  from rps.utilities.misc import *
5  from rps.utilities.controllers import *
6  import numpy as np
7  import time
8
9  N = 1 # Number of Robots
10
11 iterations = 450 # # Run the simulation/experiment for 450 steps
12
13 r = robotarium.Robotarium(number_of_robots=N, show_figure=True, sim_in_real_time=True) # Instantiate the Robotarium
14
15 # Velocity parameters for the Robots
16 linear_velocity = 0.15  # Constant linear velocity (m/s)
17 angular_velocity = 0  # No angular velocity (to move straight)
18
19 for t in range(iterations):
20
21     x = r.get_poses() # Get the poses of robots
22
23     dxu = np.zeros((2, N))  # Array to hold the velocities for each robot
24     dxu[0, :] = linear_velocity  # Set the linear velocity
25     dxu[1, :] = angular_velocity  # No angular velocity (straight line)
26
27     r.set_velocities(np.arange(N), dxu) # Apply the velocities to the robots
28
29     r.step() # Step the simulation
30
31 r.call_at_scripts_end() # Call this function at the end of the script to properly close the simulation
```

Listing 2: Basic Implementation Code Structure. The functions called and variables declared in this file are meant to familiarize users with the tools available on the Robotarium

Algorithm 1 Basic Implementation Code Structure

1: Import necessary libraries ▷ [1-7]
2: Set the number of robots for the simulation ▷ [9]
3: Determine the number of Iteration. ▷ [11]
4: Instantiate the Robotarium object with the specified parameters ▷ [13]
5: Define linear and angular velocities ▷ [16,17]
6: **for** t < iterations **do** ▷ [19]
7:     Get the current pose of the robot ▷ [21]
8:     Set the velocity of the robot ▷ [27]
9:     Iterate the simulation ▷ [29]
10: **end for**
11: Call the script end function to finalize the simulation ▷ [31]

Each step is explained below. If you need more information about a specific function, please click on ▷ [· - ·] in the Pseudo-Code. Almost all instructions are linked to the function section, where we provide detailed explanations (We use this format through all the guide).

The basic skeleton code must have:

1. We first import all the libraries that we will use in our experiment. You can check the libraries available in Appendix A
2. Declare the number of robots you would like to use. In simulation, this number must be greater than zero and less than fifty. If submitting the code to be run on the Robotarium testbed, the number of robots must be greater than 0 and less than 20.
3. Indicate how many iterations you would like the algorithm to run. If this is replaced with a while loop, make sure the script terminates.
4. Initialize Robotarium object/environment. This allows users to use the functions within the class.
5. In this example, we create two variable for the linear and angular velocity, it is not necessary, and we can add the angular and linear speed directly in the velocity function.
6. Loop over the number of iterations you specify, for instance the expected run-time of your algorithm.
7. Retrieve pose information for all robots. This function returns a matrix of size 3xN, where N is the number of robots and a 3 dimensional pose of the x-coordinate, y-coordinate, and orientation of a robot(θ) in the global coordinate frame.
8. At each iteration, set the desired velocities of the robot. Note: Prior to this function call, your algorithm should have generated a matrix of input velocities of size 2xN, where the velocities are desired linear velocities (v in $\frac{m}{s}$) and angular velocities (ω in $\frac{rad}{s}$) for each robot (We did this in line [23-25]).
9. The step function, sends the velocity inputs previously created to the robots. If you do not call this function, the robots will not move!
10. Remember, make sure the script terminates!
11. Lastly, the debug function is called to make sure that no errors have occurred in the script. It is strongly recommend to implement this function to notify you if any potential errors are found regarding potential robot collisions, actuator saturation, or failure to stay within the testing area. If you submit code that contains errors caught by this debugger, your submitted experiment may be rejected when submitting it to be run on the testbed. After you have run the experiment, you should see a message similar to Figure 18.

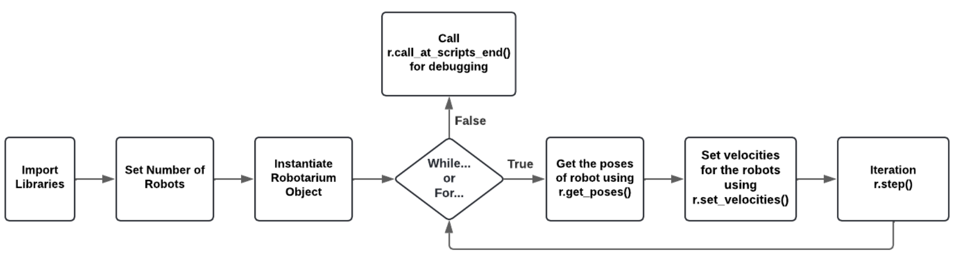


Figure 17: Basic Implementation Code Structure. The functions called and variables declared in this file are meant to familiarize users with the tools available on the Robotarium

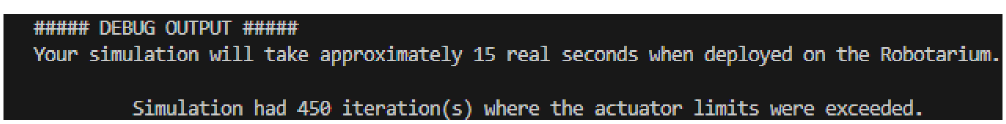
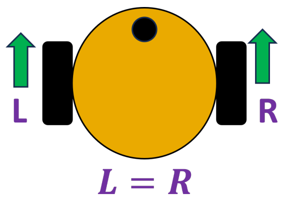


Figure 18: This message will show if you run the debug command after your code has ran. If you receive the no errors message, it improves your chances that your code is within acceptance parameters!

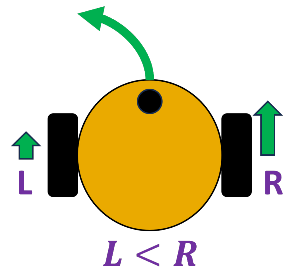
2.7 Specification of the Robotarium (FAQ)

2.7.1 How fast do the robots move?

The robots in the Robotarium have a minimum linear speed of 0.03 cm/s, and a maximum linear speed of 20 cm/s and a maximum rotational speed of approximately 3.6 rad/s (which is roughly 1/2 rotation per second). However, to protect the motors from erratic behavior at high speeds, the motors are thresholded to rotate at a maximum speed of 12.5 rad/s. Consequently, any combination of high linear and rotational commands within these maximum bounds will be adjusted to ensure the rotational speed of the motors does not exceed this limit.



(a) GTernal moving with linear speed

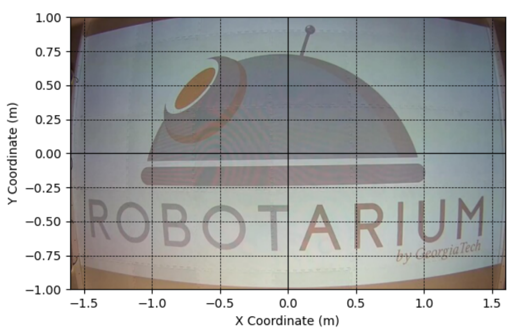


(b) GTernal moving at Angular Speed.

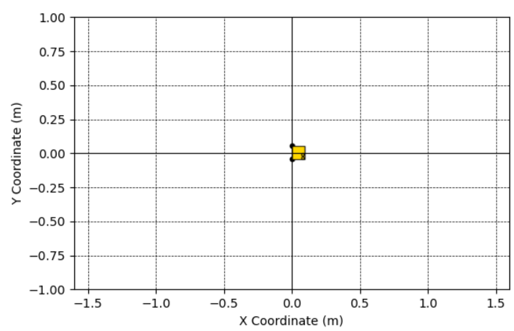
Figure 19: Linear and Angular Velocity for GTernals

2.7.2 What is the dimension of the arena?

The robots operate within a 3.2m x 2m area, which corresponds to the coverage area of the projector used in the Robotarium. This space provides enough room for the robots to maneuver while still being constrained enough to observe interactions effectively.



(a) Dimension of Robotarium



(b) Dimension of Simulation

Figure 20: Dimension of Arena.

2.7.3 How many robots can I use in my experiment? Can I use only a single robot?

In the simulator you can use as few as one robot or as many as 50 robots. In the Robotarium, you can use as few as one robot or as many as 20 robots.

2.7.4 How big are the robots?

Each robot in the Robotarium is 11 cm wide, 10 cm long, and 9.5 cm tall. However, when considering the antenna and tracking markers, the effective height of the robots is taller. These dimensions are crucial for ensuring that the robots can navigate the arena without colliding with each other or the boundaries.

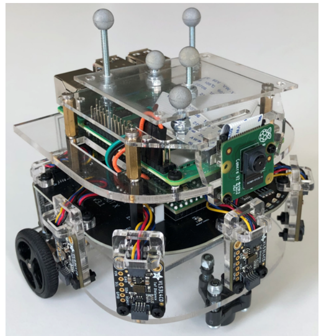


Figure 21: GTernal Robot

2.7.5 What is the minimum distance allowed between robots to avoid collision?

The robots have a diameter of approximately 11 cm. However, to ensure that the robots do not collide, it is recommended to maintain a minimum distance of 15 cm between them. This spacing helps avoid accidental collisions, especially when the robots are in motion. To further reduce the risk of collisions, you can use barrier certificates, which are built-in functions designed to keep the robots from colliding during experiments. More information about barrier certificates and barrier certificate functions can be found in subsection 4.5.

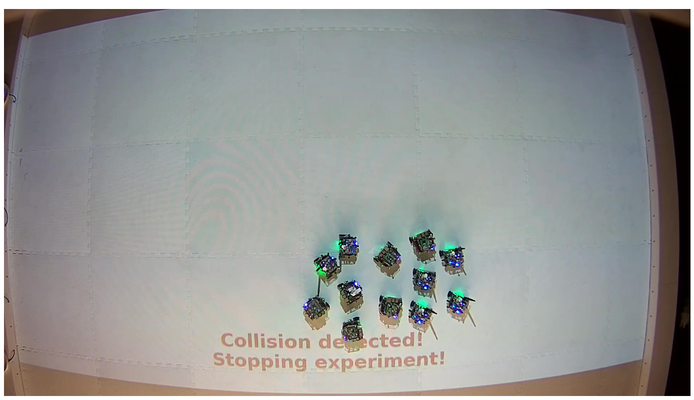


Figure 22: Example of Collision in the Robotarium

2.7.6 Times at the Robotarium

In the robotarium, each iteration is approximately to 0.033 seconds, but it has its limitations. If no new commands are received within 0.5 seconds, the robots will effectively pausing themselves until they receive a new control command again. An example of this would be add time.sleep() between commands.

2.7.7 My experiment works well in the simulator, but behaves differently in the actual testbed. Is this normal?

Yes! The full complexities of a robotic system can not always be faithfully simulated. This is why ensuring that algorithms are robust to real world perturbations is so essential to the process.

2.7.8 How might shapes be projected onto the surface?

When run on the Robotarium, the simulator's figure window is scaled to the dimensions of the testbed. So anything that you plot within the boundaries shown in the Python simulator will be projected onto the testbed and scaled appropriately.

2.7.9 What are the operating hours of the Robotarium?

Currently, the Robotarium runs experiments continuously from 9am - 5pm (EST) Monday through Friday.

2.7.10 How can I know when my experiment will be executed?

The blue button shown in Fig. 23 indicates the real-time queue of experiments in the Robotarium. This provides an estimate of how long your experiment will take to execute, considering that each experiment may take up to 15 minutes (900 seconds).

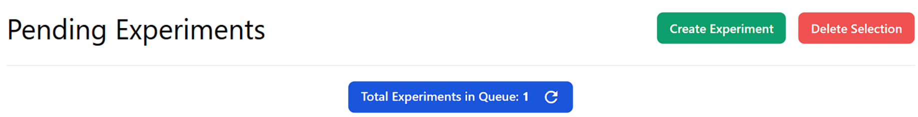


Figure 23: Total Experiments in Queue

The Robotarium operates on a first come first served basis. Depending on what time you submit and the current experiment queue, you can expect results as soon as 10 minutes after your submission (if the queue is empty) or usually within the day (if you submit during operational hours and there is not a large queue).

2.7.11 I still have questions and cannot find answers on the website or in the guide

If you cannot find the answer to your question, please contact Sean Wilson at Sean.Wilson@gtri.gatech.edu.

2.8 Communication Delays and Their Impact on Remote Robot Control

Imagine we are controlling a robot on the Moon from Earth. This robot is capable of perceiving its environment and sending the information back to Earth, where a group of scientists decides the next action based on the information received. There are two key pieces of information that we need to consider:

1. The average distance between Earth and the Moon (approximately 384,400 kilometers).[4]
2. The speed at which information travels (In a vacuum, radio waves travel at the speed of light, approximate $3 \cdot 10^8 \frac{m}{s}$.

With this information, we can calculate the time it would take for the robot to receive instructions from Earth and send information back.

$$\frac{\text{Distance between the planets}}{\text{Velocity of the information}} = \frac{384,400\ km \cdot (\frac{1,000m}{1km})}{3 \cdot 10^8 \frac{m}{s}} \approx 1.28\ s$$

It would take approximately 1.28 second (a little more than 1 second) for a one way trip of communication, and more than 2 seconds (2.56 second) for the round trip.

Let say the vehicle is continuously moving at $1 \frac{m}{s}$, the robot will be moving while waiting the next control command. Since the round trip is 2.56 seconds

$$1 \frac{m}{s} \cdot 2.56\ s \approx 2.56\ m$$

The robot would move 2.56 m between control command. And even more worrying, the robot is driving "blind" since is waiting instruction from earth.

If well this does not seems as a big deal, it important to consider the speed of the information as well as computation time. The Robotarium is a little bit closer to its robots than the Moon is to Earth, so we can obtain the position of the robot much faster. However, we also have to consider the time it takes for the central tracking server and robots on the Robotarium to process the information as well. The travel time of information and computation requirements for its baseline usage allows decisions to be made on the Robotarium approximately every 0.033 seconds. This is interpreted that each iteration in the Robotarium is equal to 0.033 s in the real world. For instance, if we have a simulation that has 1000 iteration, the time in real life would be approximately

$$0.033\ s \cdot 1000 \approx 33.00\ seconds$$

This 0.033 seconds is because of the time delay it takes on the real Robotarium to communicate and process data. It is important to note that if your submitted code takes longer than 0.033 seconds to execute a single loop this time delay will increase!

Figure 24: Vicon System in the Robotarium

[4]National Aeronautics and Space Administration. (n.d.). Facts about the Moon. NASA Science. Retrieved November 5, 2024, from https://science.nasa.gov/moon/facts/

3 Code Examples (`examples`)

In this section you will find different examples that demonstrate how to interact with the Python Simulator. These examples showcase different scenarios that you can apply when working with the robots. All of these examples are available for download with the Python Simulator Package.

Each example was designed to help you understand and implement the essential feature and capabilities of the Robotarium, from basic movement commands to more complex behaviors such as formation control. All videos of these examples and the codes can be downloaded whith the robotarium simulator, are available in our YouTube channel under the playlist called *Robotarium Examples*

3.1 `go_to_point`

3.1.1 Robots go to a point using Single Integrator

This code controls multiple robot that move towards a specified goal while avoiding collision with other robots or driving outside the testbed boundaries. The Single Integrator model was used for generating control inputs (`dxi`). The Unicycle Model was used for the actual control signal sent to the robots (`dxu`), this involved converting the single integrator control inputs to unicycle commands.

```
import rps.robotarium as robotarium
from rps.utilities.transformations import *
from rps.utilities.barrier_certificates import *
from rps.utilities.misc import *
from rps.utilities.controllers import *
import numpy as np
import time

N = 5 # Number of Robots

initial_conditions = np.array(np.mat('1 0.5 -0.5 0 0.28; 0.8 -0.3 -0.75 0.1 0.34; 0 0 0 0 0')) # Initial Positions

r = robotarium.Robotarium(number_of_robots=N, show_figure=True, initial_conditions=initial_conditions,
    sim_in_real_time=False) # Instantiate Robotarium

goal_points = generate_initial_conditions(N) # Goal Points

single_integrator_position_controller = create_si_position_controller() # Single integrator position controller

si_barrier_cert = create_single_integrator_barrier_certificate_with_boundary() # Barrier certificates to avoid collision

_, uni_to_si_states = create_si_to_uni_mapping() # Unicycle to Single Integrator States Mapping

si_to_uni_dyn = create_si_to_uni_dynamics_with_backwards_motion() # Single Integrator to Unicycle Velocity Commands

x = r.get_poses() # Get poses of the Robots

x_si = uni_to_si_states(x) # Transform poses to Single Integrator

r.step()  # Iterate the simulation

# While all Robots are not in the goal points...
while (np.size(at_pose(np.vstack((x_si,x[2,:])), goal_points, rotation_error=100)) != N):

    x = r.get_poses() # Get the poses of robots

    x_si = uni_to_si_states(x) # Transform poses to Single Integrator

    dxi = single_integrator_position_controller(x_si, goal_points[:2][:]) # Create single-integrator control inputs

    dxi_safe = si_barrier_cert(dxi, x_si) # Create safe control inputs (i.e., no collisions)

    dxu_safe = si_to_uni_dyn(dxi_safe, x) # Transform single integrator velocity commands to unicycle

    r.set_velocities(np.arange(N), dxu_safe) # Set the velocities of agents 1,...,N to dxu_safe

    r.step()  # Iterate the simulation

r.call_at_scripts_end() # Call at end of script to print debug information
```

Listing 3: `si_go_to_point.py` example

Algorithm 2 Pseudo Code `si_go_to_point.py`

```
Import important libraries ▷ [1-7]
Set the number of robots you would like to use ▷ [9]
An array with the initial positions of the robots. ▷ [11]
Initialize the Robotarium object. ▷ [13]
Generates a random set of points spaced within a bounded rectangle (the testbed) as goal points ▷ [15]
Create single integrator position controller ▷ [17]
Create barrier certificates to avoid collision ▷ [19]
Create mapping from single integrator states to unicycle states ▷ [21]
Create mapping from single integrator velocity commands to unicycle velocity commands ▷ [23]
Get initial poses of the robots ▷ [25]
Convert unicycle states to single integrator states ▷ [27]
Step the simulation to update visualization. Iterate the experiment ▷ [29]
while the number of robots not at the goal poses < N do
    Get current poses of the robots. ▷ [34]
    Convert unicycle states to single integrator states ▷ [36]
    Create single-integrator control inputs ▷ [38]
    Create safe control inputs in order to avoid collision. ▷ [40]
    Transform single integrator velocity commands to unicycle ▷ [42]
    Set the velocities by mapping the single-integrator inputs to unicycle inputs ▷ [44]
    Step the simulation to update visualization. Iterate the experiment ▷ [46]
end while
Call this script to debug at the end of the experiment. ▷ [48]
```

3.1.2 Robots go to a point Unicycle

This code controls multiple robots that move towards a specified goal while avoiding collision with other robots or driving outside the testbed boundaries. The Unicycle Model is used for generating control inputs and for the control command sent to the robots (`dxu`). Barrier certificates ensure safe control inputs to avoid collisions.

```
1  import rps.robotarium as robotarium
2  from rps.utilities.transformations import *
3  from rps.utilities.barrier_certificates import *
4  from rps.utilities.misc import *
5  from rps.utilities.controllers import *
6  import numpy as np
7  import time
8
9  N = 5 # Number of Robots
10
11 initial_conditions = np.array(np.mat('1 0.5 -0.5 0 0.28; 0.8 -0.3 -0.75 0.1 0.34; 0 0 0 0 0')) # Initial Poses
12
13 r = robotarium.Robotarium(number_of_robots=N, show_figure=True, initial_conditions=initial_conditions,sim_in_real_time=True) # Instantiate Robotarium object
14
15 goal_points = generate_initial_conditions(N) # Goal Points
16
17 unicycle_position_controller = create_clf_unicycle_position_controller() # Create unicycle position controller
18
19 uni_barrier_cert = create_unicycle_barrier_certificate() # Create barrier certificates to avoid collision
20
21 x = r.get_poses() # Get the poses of the Robots
22
23 r.step() # Iterate the simulation
24
25 # While all Robots are not in the goal points...
26 while (np.size(at_pose(x, goal_points, rotation_error=100)) != N):
27
28     x = r.get_poses() # Get the poses of Robots
29
30     dxu = unicycle_position_controller(x, goal_points[:2][:]) # Create single-integrator control inputs
31
32     dxu_safe = uni_barrier_cert(dxu, x) # Create safe control inputs (i.e., no collisions)
33
34     r.set_velocities(np.arange(N), dxu_safe) # Set the velocities of agents 1,...,N to dxu_safe
35
36     r.step() # Iterate the simulation
37
38 r.call_at_scripts_end() # Call at end of script to print debug information
```

Listing 4: `uni_go_to_point.py` example

Algorithm 3 Pseudo Code `uni_go_to_point.py`

```
Import important libraries                                                              ▷ [1-7]
Set the number of robots you would like to use                                          ▷ [9]
Define an array with the initial positions of the robots                                ▷ [11]
Initialize the Robotarium object                                                        ▷ [13]
Generate a random set of points spaced within a bounded rectangle (the testbed) as goal points ▷ [15]
Create unicycle position controller                                                     ▷ [17]
Create barrier certificates to avoid collision                                          ▷ [19]
Get initial poses of the robots                                                         ▷ [21]
Step the simulation to update visualization                                             ▷ [23]
while the number of robots not at the goal poses < N do
    Get current poses of the robots                                                     ▷ [28]
    Create unicycle control inputs                                                      ▷ [30]
    Create safe control inputs in order to avoid collision                              ▷ [32]
    Set the velocities by mapping the single-integrator inputs to unicycle inputs       ▷ [34]
    Step the simulation to update visualization                                         ▷ [36]
end while
Call this script to debug at the end of the experiment                                  ▷ [38]
```

3.2 go_to_pose

3.2.1 Robots go to a point using Unicycle Pose Controller

This code controls multiple robots that move towards a specified pose (i.e. a position and orientation) while avoiding collisions with other robots or driving outside the testbed boundaries. The Unicycle Pose Controller is used for generating control inputs that are sent directly to the robots (dxu). Barrier certificates ensure safe control inputs to avoid collisions.

```python
import rps.robotarium as robotarium
from rps.utilities.transformations import *
from rps.utilities.barrier_certificates import *
from rps.utilities.misc import *
from rps.utilities.controllers import *
import numpy as np
import time

N = 5 # Number of Robots

initial_conditions = np.array(np.mat('1 0.5 -0.5 0 0.28; 0.8 -0.3 -0.75 0.1 0.34; 0 0 0 0 0')) # Initial Positions

r = robotarium.Robotarium(number_of_robots=N, show_figure=True, initial_conditions=initial_conditions,
    sim_in_real_time=True) # Instantiate Robotarium object

goal_points = generate_initial_conditions(N) # Goal Points

unicycle_pose_controller = create_clf_unicycle_pose_controller() # Create unicycle pose controller

uni_barrier_cert = create_unicycle_barrier_certificate() # Create barrier certificates to avoid collision

x = r.get_poses() # Get the poses of Robots

r.step() # Iterate the simulation

# While the Robots are not in the goal points
while (np.size(at_pose(x, goal_points)) != N):

    x = r.get_poses() # Get the poses of Robots

    dxu = unicycle_pose_controller(x, goal_points) # Create unicycle control inputs

    dxu_safe = uni_barrier_cert(dxu, x) # Create safe control inputs (i.e., no collisions)

    r.set_velocities(np.arange(N), dxu_safe) # Set the velocities of agents 1,...,N to dxu_safe

    r.step() # Iterate the simulation

r.call_at_scripts_end() # Call at end of script to print debug information
```

Listing 5: uni_go_to_pose_clf.py example

Algorithm 4 Pseudo Code `uni_go_to_pose_clf.py`

```
Import important libraries                                                              ▷ [1-7]
Set the number of robots you would like to use                                          ▷ [9]
Define an array with the initial positions of the robots                                ▷ [11]
Initialize the Robotarium object                                                        ▷ [13]
Generate a random set of points spaced within a bounded rectangle (the testbed) as goal points ▷ [15]
Create unicycle pose controller                                                         ▷ [17]
Create barrier certificates to avoid collision                                          ▷ [19]
Get initial poses of the robots                                                         ▷ [21]
Step the simulation to update visualization                                             ▷ [23]
while the number of robots not at the goal poses < N do
    Get current poses of the robots                                                     ▷ [28]
    Create unicycle control inputs                                                      ▷ [30]
    Create safe control inputs in order to avoid collision                              ▷ [32]
    Set the velocities by mapping the single-integrator inputs to unicycle inputs       ▷ [34]
    Step the simulation to update visualization                                         ▷ [36]
end while
Call this script to debug at the end of the experiment                                  ▷ [38]
```

3.2.2 Robots go to a point using Unicycle Hybrid Pose Controller

This code controls multiple robots that move towards a specified goal while avoiding collisions with other robots or driving outside the testbed boundaries. The Unicycle Hybrid Pose Controller is used for generating control inputs that are directly sent to the robots (`dxu`). Barrier certificates ensure safe control inputs to avoid collisions.

```
1  import rps.robotarium as robotarium
2  from rps.utilities.transformations import *
3  from rps.utilities.barrier_certificates import *
4  from rps.utilities.misc import *
5  from rps.utilities.controllers import *
6  import numpy as np
7  import time
8
9  N = 5 # Number of Robots
10
11 initial_conditions = np.array(np.mat('1 0.5 -0.5 0 0.28; 0.8 -0.3 -0.75 0.1 0.34; 0 0 0 0 0')) #Goal Points
12
13 r = robotarium.Robotarium(number_of_robots=N, show_figure=True, initial_conditions=initial_conditions,
   ↪  sim_in_real_time=True) # Instantiate Robotarium object
14
15 goal_points = generate_initial_conditions(N) # Goal Points
16
17 unicycle_pose_controller = create_hybrid_unicycle_pose_controller() # Create unicycle hybrid pose controller
18
19 uni_barrier_cert = create_unicycle_barrier_certificate() # Create barrier certificates to avoid collision
20
21 x = r.get_poses() # Get the poses of Robots
22
23 r.step() # Iterate the simulation
24
25 # While the robots are not in the goal points
26 while (np.size(at_pose(x, goal_points)) != N):
27
28     x = r.get_poses() # Get poses of Robots
29
30     dxu = unicycle_pose_controller(x, goal_points) # Create unicycle control inputs
31
32     dxu_safe = uni_barrier_cert(dxu, x) # Create safe control inputs (i.e., no collisions)
33
34     r.set_velocities(np.arange(N), dxu_safe) # Set the velocities of agents 1,...,N to dxu_safe
35
36     r.step() # Iterate the simulation
37
38 r.call_at_scripts_end() # Call at end of script to print debug information
```

Listing 6: `uni_go_to_pose_hybrid.py` example

Algorithm 5 Pseudo Code `uni_go_to_pose_hybrid.py`

1: Import important libraries ▷ [1-7]
2: Set the number of robots you would like to use ▷ [9]
3: Define an array with the initial positions of the robots ▷ [11]
4: Initialize the Robotarium object ▷ [13]
5: Generate a random set of points spaced within a bounded rectangle (the testbed) as goal points ▷ [15]
6: Create unicycle hybrid pose controller ▷ [17]
7: Create barrier certificates to avoid collision ▷ [19]
8: Get initial poses of the robots ▷ [21]
9: Step the simulation to update visualization ▷ [23]
10: **while** the number of robots not at the goal poses < N **do**
11:     Get current poses of the robots ▷ [28]
12:     Create unicycle control inputs ▷ [30]
13:     Create safe control inputs in order to avoid collision ▷ [32]
14:     Set the velocities by mapping the single-integrator inputs to unicycle inputs ▷ [34]
15:     Step the simulation to update visualization ▷ [36]
16: **end while**
17: Call this script to debug at the end of the experiment ▷ [38]

3.3 leader_follower_static

3.3.1 Leader-Follower Formation Control

This code controls multiple robots in a leader-follower formation. The leader moves towards specified waypoints, and the followers maintain a formation around the leader while avoiding collisions. The control inputs are generated using a Single Integrator model and are then converted to Unicycle commands that are sent to the robots.

```
import rps.robotarium as robotarium
from rps.utilities.transformations import *
from rps.utilities.graph import *
from rps.utilities.barrier_certificates import *
from rps.utilities.misc import *
from rps.utilities.controllers import *
import numpy as np

iterations = 5000 # Run the simulation/experiment for 5000 steps

N = 4 # Number of robots

waypoints = np.array([[-1, -1, 1, 1],[0.8, -0.8, -0.8, 0.8]]) # Waypoints the leader moves to.

close_enough = 0.03; # How close the leader must get to the waypoint to move to the next one.

# Create the desired Laplacian
followers = -completeGL(N-1)
L = np.zeros((N,N))
L[1:N,1:N] = followers
L[1,1] = L[1,1] + 1
L[1,0] = -1

# Find connections
[rows,cols] = np.where(L==1)

dxi = np.zeros((2,N)) # For computational/memory reasons, initialize the velocity vector

state = 0 # Initialize leader state

magnitude_limit = 0.15 # Limit maximum linear speed of any robot

formation_control_gain = 10 # Gains for the Formation Control

desired_distance = 0.3 # desired distance between the Robots

initial_conditions = np.array([[0, 0.5, 0.3, -0.1],[0.5, 0.5, 0.2, 0],[0, 0, 0, 0]]) # Initial Positions

r = robotarium.Robotarium(number_of_robots=N, show_figure=True, initial_conditions=initial_conditions,
↪  sim_in_real_time=True) # Instantiate the Robotarium

_, uni_to_si_states = create_si_to_uni_mapping() # Unicycle to Single Integrator States Mapping

si_to_uni_dyn = create_si_to_uni_dynamics() # Single Integrator to Unicycle Velocity Commands

si_barrier_cert = create_single_integrator_barrier_certificate_with_boundary() # Single-integrator barrier certificates

leader_controller = create_si_position_controller(velocity_magnitude_limit=0.1) # Single-integrator position controller

for t in range(iterations):

    x = r.get_poses() # Get the poses of robots

    xi = uni_to_si_states(x) # Transform poses to Single Integrator

    # Followers
    for i in range(1,N):
        dxi[:,[i]] = np.zeros((2,1)) # Zero velocities
        neighbors = topological_neighbors(L,i) # Get the topological neighbors of agent i

        for j in neighbors:
            dxi[:,[i]] += formation_control_gain*(np.power(np.linalg.norm(x[:2,[j]]-x[:2,[i]]),
            ↪  2)-np.power(desired_distance, 2))*(x[:2,[j]]-x[:2,[i]])

    # Leader
    waypoint = waypoints[:,state].reshape((2,1))

    dxi[:,[0]] = leader_controller(x[:2,[0]], waypoint)
    if np.linalg.norm(x[:2,[0]] - waypoint) < close_enough: # Leader moves to the waypoint and updates to the next one if
    ↪  close
```

```
68            state = (state + 1) % 4
69
70        # Keep single integrator control vectors under specified magnitude
71        norms = np.linalg.norm(dxi, 2, 0)  # Threshold control inputs
72        idxs_to_normalize = (norms > magnitude_limit)  # Threshold control inputs
73        dxi[:, idxs_to_normalize] *= magnitude_limit/norms[idxs_to_normalize]  # Threshold control inputs
74
75        dxi_safe = si_barrier_cert(dxi, x[:2,:]) # Use Barrier to avoid collision
76
77        dxu_safe = si_to_uni_dyn(dxi_safe, x) # Convert single-integrator to unicycle commands
78
79        r.set_velocities(np.arange(N), dxu_safe) # Set the velocities using unicycle commands
80
81        r.step() # Iterate the simulation
82
83    r.call_at_scripts_end() # Call at end of script to print debug information
```

Listing 7: `leader_follower.py` example

Algorithm 6 Pseudo Code `leader_follower.py`

```
Import important libraries                                                         ▷ [1-7]
Set the number of iterations and number of robots                                  ▷ [9,11]
Define waypoints and the proximity threshold for the leader                        ▷ [13,15]
Create the desired Laplacian matrix for leader-follower formation                  ▷ [17-25]
Initialize the velocity vector for computational/memory reasons                    ▷ [27]
Initialize the leader state                                                        ▷ [29]
Set the maximum linear speed limit                                                 ▷ [31]
Define gains and desired distance for formation control                            ▷ [33-35]
Set initial conditions to avoid barrier use at the beginning                       ▷ [37]
Instantiate the Robotarium object with the specified parameters                    ▷ [39]
Create mapping from single integrator states to unicycle states and vice versa.    ▷ [41]
Create mapping from single integrator velocity commands to unicycle velocity commands ▷ [43]
Create barrier certificates to avoid collision                                      ▷ [45]
Create single integrator position controller doe the leader                        ▷ [47]
for each iteration from 1 to iterations do
    Get the most recent pose information from the Robotarium                       ▷ [51]
    Convert unicycle states to single integrator states                            ▷ [53]
    Followers:
    for each follower robot i from 1 to N − 1 do
        Zero velocities and get the topological neighbors of robot i               ▷ [57-58]
        for each neighbor j of robot i do
            Update the velocity of robot i based on the distance to neighbor j     ▷ [61]
        end for
    end for
    Leader:
    Set the current waypoint for the leader                                        ▷ [64]
    Update the velocity of the leader towards the waypoint                         ▷ [66]
    if the leader is close enough to the waypoint then                             ▷ [67]
        Move to the next waypoint                                                  ▷ [68]
    end if
    Threshold the control inputs to maintain specified magnitude limit             ▷ [71-73]
    Use barrier certificates to ensure safe control inputs                         ▷ [75]
    Convert single-integrator inputs to unicycle commands                          ▷ [77]
    Set the velocities of the robots                                               ▷ [79]
    Iterate the simulation                                                         ▷ [81]
end for
Call this script to debug at the end of the experiment                             ▷ [83]
```

3.4 formation_control.py

3.4.1 Rectangle Formation Control

This code controls multiple robots to form a rectangular shape while avoiding collisions. The control inputs are generated using a Single Integrator model and are then converted to Unicycle commands that are sent to the robots. The inter-agent distances are controlled using a weight matrix.

```python
import rps.robotarium as robotarium
from rps.utilities.transformations import *
from rps.utilities.graph import *
from rps.utilities.barrier_certificates import *
from rps.utilities.misc import *
from rps.utilities.controllers import *

# Array representing the geometric distances betwen the agents.  In this case, the agents try to form a Rectangle
L = np.array([
    [3, -1, 0, -1, 0, -1],
    [-1, 3, -1, 0, -1, 0],
    [0, -1, 3, -1, 0, -1],
    [-1, 0 , -1, 3, -1, 0],
    [0, -1, 0, -1, 3, -1],
    [-1, 0, -1, 0, -1, 3]
])

d = 0.3 # Desired distance between agent
ddiag = np.sqrt(5)*d
formation_control_gain = 10 # Desired gain

# Weight matrix to control inter-agent distances
weights = np.array([
    [0, d, 0, d, 0, ddiag],
    [d, 0, d, 0, d, 0],
    [0, d, 0, ddiag, 0, d],
    [d, 0, ddiag, 0, d, 0],
    [0, d, 0, d, 0, d],
    [ddiag, 0, d, 0, d, 0]
])

iterations = 2000 # Run the simulation/experiment for 2000 steps

N = 6 # Number of Robots

magnitude_limit = 0.15 # Limit maximum linear speed of any robot

r = robotarium.Robotarium(number_of_robots=N, show_figure=True, sim_in_real_time=True) # Instantiate Robotarium object

si_barrier_cert = create_single_integrator_barrier_certificate_with_boundary()  # Barrier certificates to avoid collision

si_to_uni_dyn = create_si_to_uni_dynamics() # Single Integrator to Unicycle Velocity Commands

for k in range(iterations):

    x = r.get_poses() # Get the poses of the robots

    dxi = np.zeros((2, N)) # Initialize a velocity vector

    for i in range(N):
        for j in topological_neighbors(L, i):
            error = x[:2, j] - x[:2, i]  # Perform a weighted consensus to make the rectangular shape
            dxi[:, i] += formation_control_gain*(np.power(np.linalg.norm(error), 2)- np.power(weights[i, j], 2)) * error

    norms = np.linalg.norm(dxi, 2, 0)
    idxs_to_normalize = (norms > magnitude_limit) # Threshold control inputs
    dxi[:, idxs_to_normalize] *= magnitude_limit/norms[idxs_to_normalize] # Keep single integrator control vectors under
    ↪  specified magnitude

    dxi_safe = si_barrier_cert(dxi, x[:2, :]) # Make sure that the robots don't collide

    dxu_safe = si_to_uni_dyn(dxi_safe, x) # Transform the single-integrator dynamcis to unicycle dynamics

    r.set_velocities(np.arange(N), dxu_safe) # Set the velocities of the robots

    r.step() # Iterate the simulation

r.call_at_scripts_end() # Call at end of script to print debug information
```

Listing 8: formation_control.py example

Algorithm 7 Pseudo Code `formation_control.py`

```
Import important libraries ▷ [1-6]
Define the Laplacian matrix representing the geometric distances between the agents ▷ [9]
Define the gains for the experiment and the desired distances ▷ [18-21]
Define the weight matrix to control inter-agent distances ▷ [23]
Set the number of iterations and number of robots ▷ [32,34]
Set the maximum linear speed limit ▷ [36]
Instantiate the Robotarium object with the specified parameters ▷ [38]
Create barrier certificates to avoid collision ▷ [40]
Create mapping from single integrator velocity commands to unicycle velocity commands ▷ [42]
for each iteration from 1 to iterations do
    Get the poses of the robots ▷ [46]
    Initialize a velocity vector ▷ [49]
    for each robot i from 1 to N do
        for each neighbor j of robot i do ▷ [56]
            Perform a weighted consensus to make the rectangular shape ▷ [52-53]
        end for
    end for
    Threshold the control inputs to maintain specified magnitude limit ▷ [55-57]
    Use barrier certificates to ensure safe control inputs ▷ [59]
    Convert single-integrator inputs to unicycle commands ▷ [61]
    Set the velocities of the robots ▷ [63]
    Iterate the simulation ▷ [65]
end for
Call this script to debug at the end of the experiment ▷ [67]
```

3.5 barrier_certificate

3.5.1 Single Integrator Barrier Certificate

This code controls multiple robots to form a circle, switching their positions on the circle a specified number of times. The control inputs are generated using a Single Integrator model and are then converted to Unicycle commands that are sent to the actual robots. Barrier certificates applied to the single integrator control commands ensure safe control inputs to avoid collisions. Note that the safe single integrator control commands (dxi_safe) are transformed to unicycle control commands. Do not use single integrator barrier certificates with unicycle control commands.

```python
import rps.robotarium as robotarium
from rps.utilities.transformations import *
from rps.utilities.barrier_certificates import *
from rps.utilities.misc import *
from rps.utilities.controllers import *
import numpy as np

N = 10 # Number of Robots

r = robotarium.Robotarium(number_of_robots=N, show_figure=True, sim_in_real_time=True) # Instantiate the Robotarium object

num_cycles = 2 # Number times Robots form circle

count = -1 # Number times they have formed the circle (starts at -1 since initial formation will increment the count)

si_barrier_cert = create_single_integrator_barrier_certificate()  # Create barrier certificates to avoid collision

si_position_controller = create_si_position_controller() # Create single integrator position controller

si_to_uni_dyn, uni_to_si_states = create_si_to_uni_mapping() # Create SI to UNI dynamics transformation

# Generates points on a circle inscribed in a 6x6 square centered at the origin. Robots swap positions on the circle.
radius = 1
xybound = radius * np.array([-1, 1, -1, 1])
p_theta = 2 * np.pi * (np.arange(0, 2 * N, 2) / (2 * N))
p_circ = np.vstack([
    np.hstack([xybound[1] * np.cos(p_theta), xybound[1] * np.cos(p_theta + np.pi)]),
    np.hstack([xybound[3] * np.sin(p_theta), xybound[3] * np.sin(p_theta + np.pi)])
])

flag = 0 # These variables are so we can tell when the robots should switch positions on the circle.
x_goal = p_circ[:, :N]

while(1): # While True

    x = r.get_poses() # Get the poses of robots

    x_si = uni_to_si_states(x) # Transform unicycle state to Single Integrator state.

    si_velocities = np.zeros((2, N)) # Initialize a velocities variable

    if(np.linalg.norm(x_goal - x_si) < 0.05): # If all the agent are close enough to the goals
        flag = 1 - flag # Change
        count += 1 # Increase the counter
    if count == num_cycles: # if the robots did the circle twice
        break # End the Experiment

    if(flag == 0): # If 0, we change the goal to the other side of the circle
        x_goal = p_circ[:, :N]
    else:
        x_goal = p_circ[:, N:] # If 1, we change the goal to the other side of the circle

    dxi = si_position_controller(x_si, x_goal)  # Use a position controller to drive to the goal position

    dxi_safe = si_barrier_cert(dxi, x_si)  # Use the barrier certificates to make sure that the agents don't collide

    dxu_safe = si_to_uni_dyn(dxi_safe, x) # Use the second single-integrator-to-unicycle mapping to map to unicycle dynamics

    r.set_velocities(np.arange(N), dxu_safe) # Set the velocities of agents 1,...,N to dxu_safe

    r.step() # Iterate the simulation

r.call_at_scripts_end() # Call at end of script to print debug information
```

Listing 9: barrier_certficate.py example

Algorithm 8 Pseudo Code `barrier_certficate.py`

```
Import important libraries ▷ [1-6]
Set the number of robots for the simulation ▷ [8]
Instantiate the Robotarium object with the specified parameters ▷ [10]
Define the number of cycles for the circle formation and initialize the count ▷ [12,14]
Create single integrator barrier certificates ▷ [16]
Create single integrator position controller ▷ [18]
Create mapping from single integrator states to unicycle states and vice versa. ▷ [20]
Generate points on a circle inscribed in a 6x6 square centered on the origin ▷ [22-29]
Initialize variables to manage goal switching ▷ [31-33]
while True do
    Get the poses of the agents ▷ [37]
    Convert unicycle states to single integrator states ▷ [39]
    Initialize a velocities variable ▷ [41]
    if all agents are close enough to the goals then
        Switch the goal flag and increment the count ▷ [44-45]
    end if
    if count equals the number of cycles then
        Break the loop ▷ [48]
    end if
    Switch goals based on the flag state ▷ [50-53]
    Use the position controller to drive to the goal position ▷ [55]
    Use the barrier certificates to avoid collisions ▷ [57]
    Map single-integrator inputs to unicycle dynamics ▷ [59]
    Set the velocities of the robots ▷ [61]
    Iterate the simulation ▷ [63]
end while
Call this script to debug at the end of the experiment ▷ [65]
```

3.5.2 Robots Moving Towards Unreachable Goals with Single-Integrator Model

This code controls multiple robots to move towards goal points that are outside the arena. The robots will never reach these goal points, and the simulation will run for a specified number of iterations. The control inputs are generated using a Single Integrator model and are then converted to Unicycle commands for actual movement. Barrier certificates applied to the single integrator control commands ensure safe control inputs to avoid collisions. Note that the safe single integrator control commands (dxi_safe) are transformed to unicycle control commands. Do not use single integrator barrier certificates with unicycle control commands.

```python
import rps.robotarium as robotarium
from rps.utilities.transformations import *
from rps.utilities.barrier_certificates import *
from rps.utilities.misc import *
from rps.utilities.controllers import *
import numpy as np

N = 5 # Number of Robots

initial_conditions = np.array(np.mat('1 0.5 -0.5 0 0.28; 0.8 -0.3 -0.75 0.1 0.34; 0 0 0 0 0')) # Initial positions

r = robotarium.Robotarium(number_of_robots=N, show_figure=True, initial_conditions=initial_conditions, sim_in_real_time=True) # Instantiate Robotarium object

iterations = 3000 # Run the simulation/experiment for 3000 steps

goal_points = np.array(np.mat('5 -5 5 -5 5; 5 5 -5 -5 5; 0 0 0 0 0')) # Goal points

si_position_controller = create_si_position_controller() # Create single integrator position controller

si_to_uni_dyn, uni_to_si_states = create_si_to_uni_mapping() # Create SI to UNI dynamics transformation

si_barrier_cert = create_single_integrator_barrier_certificate_with_boundary() # Create barrier certificates to avoid collision

x = r.get_poses() # Get the poses of robots

r.step() # Iterate the simulation

# While the Robots are not in the goal points
for i in range(iterations):

    x = r.get_poses() # Get the poses of robots

    xi = uni_to_si_states(x) # Transform unicycle states to single integrator

    dxi = si_position_controller(xi, goal_points[:2, :]) # Create single-integrator control inputs

    dxi_safe = si_barrier_cert(dxi, xi) # Create safe control inputs (i.e., no collisions)

    dxu_safe = si_to_uni_dyn(dxi_safe, x) # Map the single integrator back to unicycle dynamics

    r.set_velocities(np.arange(N), dxu_safe) # Set the velocities of agents 1,...,N to dxu_safe

    r.step() # Iterate the simulation

r.call_at_scripts_end() # Call at end of script to print debug information
```

Listing 10: `si_barriers_with_boundary.py` example

Algorithm 9 Pseudo Code `si_barriers_with_boundary.py`

```
Import important libraries                                      ▷ [1-6]
Set the number of robots for the simulation                     ▷ [8]
Define the initial positions of the robots                      ▷ [10]
Instantiate the Robotarium object with the specified parameters ▷ [12]
Set the number of iterations for the simulation                 ▷ [14]
Define goal points outside of the arena                         ▷ [16]
Create single integrator position controller                    ▷ [18]
Create SI to UNI dynamics transformation                        ▷ [20]
Create barrier certificates to avoid collision                  ▷ [22]
Get initial poses of the robots                                 ▷ [24]
Step the simulation to update visualization                     ▷ [26]
for each iteration from 1 to iterations do
    Get the current poses of the robots                         ▷ [31]
    Convert unicycle states to single integrator states         ▷ [33]
    Create single-integrator control inputs                     ▷ [35]
    Use barrier certificates to ensure safe control inputs      ▷ [37]
    Map single-integrator inputs to unicycle dynamics           ▷ [39]
    Set the velocities of the robots                            ▷ [41]
    Iterate the simulation                                      ▷ [43]
end for
Call this script to debug at the end of the experiment          ▷ [45]
```

3.5.3 Robots Moving Towards Unreachable Goals with Unicycle Model

This code controls multiple robots to move towards goal points that are outside the arena using a unicycle model. The robots will never reach these goal points, and the simulation will run for a specified number of iterations. The control inputs are generated using a Unicycle model. Barrier certificates ensure safe control inputs to avoid collisions. Since we did not specify the initial position, the robots start in random positions. Barrier certificates applied to the single integrator control commands ensure safe control inputs to avoid collisions. Do not use unicycle model barrier certificates with single integrator control commands.

```python
import rps.robotarium as robotarium
from rps.utilities.transformations import *
from rps.utilities.barrier_certificates import *
from rps.utilities.misc import *
from rps.utilities.controllers import *
import numpy as np

N = 5 # Number of Robots

r = robotarium.Robotarium(number_of_robots=N, show_figure=True, sim_in_real_time=True) # Instantiate Robotarium object

iterations = 3000 # Run the simulation/experiment for 3000 steps

goal_points = np.array(np.mat('5 5 5 5 5; 5 5 5 5 5; 0 0 0 0 0')) # Goal points

unicycle_position_controller = create_clf_unicycle_position_controller() # Create unicycle position controller

uni_barrier_cert = create_unicycle_barrier_certificate_with_boundary() # Create barrier certificates to avoid collision

x = r.get_poses() # Get the poses of robots

r.step() # Iterate the simulation

for i in range(iterations):

    x = r.get_poses()  # Get the poses of robots

    dxu = unicycle_position_controller(x, goal_points[:2][:]) # Create unicycle control inputs

    dxu_safe = uni_barrier_cert(dxu, x)  # Create safe control inputs (i.e., no collisions)

    r.set_velocities(np.arange(N), dxu_safe) # Set the velocities of agents 1,...,N to dxu_safe

    r.step() # Iterate the simulation

r.call_at_scripts_end() # Call at end of script to print debug information
```

Listing 11: `uni_barriers_with_boundary.py` example

Algorithm 10 Pseudo Code `uni_barriers_with_boundary.py`

```
Import important libraries                                                         ▷ [1-6]
Set the number of robots for the simulation                                        ▷ [8]
Instantiate the Robotarium object with the specified parameters                    ▷ [10]
Set the number of iterations for the simulation                                    ▷ [12]
Define goal points outside of the arena                                            ▷ [14]
Create unicycle position controller                                                ▷ [16]
Create barrier certificates to avoid collision                                     ▷ [18]
Get initial poses of the robots                                                    ▷ [20]
Step the simulation to update visualization                                        ▷ [22]
for each iteration from 1 to iterations do
    Get the current poses of the robots                                            ▷ [26]
    Create unicycle control inputs                                                 ▷ [28]
    Use barrier certificates to ensure safe control inputs                         ▷ [30]
    Set the velocities by mapping the single-integrator inputs to unicycle inputs  ▷ [32]
    Iterate the simulation                                                         ▷ [34]
end for
Call this script to debug at the end of the experiment                             ▷ [36]
```

3.6 data_saving

3.6.1 Leader-Follower Formation with Data Saving

This code controls multiple robots in a leader-follower formation. The leader moves towards specified waypoints, and the followers maintain a formation. The control inputs are generated using a Single Integrator model and are then converted to Unicycle commands that are sent to the robots. Barrier certificates ensure safe control inputs to avoid collisions. Additionally, the code saves data on the distance between connected robots and the distance between the leader and the goal location when the goal is reached. Two data sets will be saved, one saving the distance between connected robots through time, and another with the distance between the leader and goal location when the goal is "reached". They will each be saved as .npy files and human readable csv .txt files.

```python
import rps.robotarium as robotarium
from rps.utilities.transformations import *
from rps.utilities.graph import *
from rps.utilities.barrier_certificates import *
from rps.utilities.misc import *
from rps.utilities.controllers import *
import numpy as np
import time

iterations = 5000 # Run the simulation/experiment for 5000 steps

N = 4 # Number of robots

waypoints = np.array([[-1, -1, 1, 1],[0.8, -0.8, -0.8, 0.8]]) # Waypoints the leader moves to.

close_enough = 0.03; # How close the leader must get to the waypoint to move to the next one.

# Preallocate data saving
robot_distance = np.zeros((5,iterations)) # Saving 4 inter-robot distances and time
goal_distance = np.empty((0,2))
start_time = time.time()

# Create the desired Laplacian
followers = -completeGL(N-1)
L = np.zeros((N,N))
L[1:N,1:N] = followers
L[1,1] = L[1,1] + 1
L[1,0] = -1

# Find connections
[rows,cols] = np.where(L==1)

dxi = np.zeros((2,N)) # For computational/memory reasons, initialize the velocity vector

state = 0 # Initialize leader state

magnitude_limit = 0.15 # Limit maximum linear speed of any robot

formation_control_gain = 10 # Gains for the Formation Control

desired_distance = 0.3 # Desired distance between the Robots

initial_conditions = np.array([[0, 0.5, 0.3, -0.1],[0.5, 0.5, 0.2, 0],[0, 0, 0, 0]]) # Initial Positions

r = robotarium.Robotarium(number_of_robots=N, show_figure=True, initial_conditions=initial_conditions,
↪  sim_in_real_time=True) # Instantiate the Robotarium

_, uni_to_si_states = create_si_to_uni_mapping() # Unicycle to Single Integrator States Mapping

si_to_uni_dyn = create_si_to_uni_dynamics() # Single Integrator to Unicycle Velocity Commands

si_barrier_cert = create_single_integrator_barrier_certificate_with_boundary() # Single-integrator barrier certificates

leader_controller = create_si_position_controller(velocity_magnitude_limit=0.1) # Single-integrator position controller

for t in range(iterations):

    x = r.get_poses() # Get the pose of robots

    xi = uni_to_si_states(x) # Transform poses to Single Integrator

    # Followers
    for i in range(1, N):
```

```python
        dxi[:, [i]] = np.zeros((2, 1)) # Zero velocities
        neighbors = topological_neighbors(L, i) # Get the topological neighbors of agent i

        for j in neighbors:
            dxi[:, [i]] += formation_control_gain * (np.power(np.linalg.norm(x[:2, [j]] - x[:2, [i]]), 2) - np.power(desired_distance, 2)) * (x[:2, [j]] - x[:2, [i]])

    # Leader
    waypoint = waypoints[:, state].reshape((2, 1))

    dxi[:, [0]] = leader_controller(x[:2, [0]], waypoint)
    if np.linalg.norm(x[:2, [0]] - waypoint) < close_enough: # Leader moves to the waypoint and updates to the next one if close
        state = (state + 1) % 4
        goal_distance = np.append(goal_distance, np.array([[np.linalg.norm(xi[:, [0]] - waypoint)], [time.time() - start_time]]))

    # Keep single integrator control vectors under specified magnitude
    norms = np.linalg.norm(dxi, 2, 0)  # Threshold control inputs
    idxs_to_normalize = (norms > magnitude_limit)  # Threshold control inputs
    dxi[:, idxs_to_normalize] *= magnitude_limit / norms[idxs_to_normalize]  # Threshold control inputs

    dxi_safe = si_barrier_cert(dxi, x[:2, :]) # Use Barrier to avoid collision

    dxu_safe = si_to_uni_dyn(dxi_safe, x) # Convert single-integrator to unicycle commands

    r.set_velocities(np.arange(N), dxu_safe) # Set the velocities using unicycle commands

    # Compute data to be saved and stored in matrix (Distance between connected robots)
    robot_distance[0, t] = np.linalg.norm(xi[:, [0]] - xi[:, [1]])
    robot_distance[4, t] = time.time() - start_time
    for j in range(1, int(len(rows) / 2) + 1):
        robot_distance[j, t] = np.linalg.norm(xi[:, [rows[j]]] - xi[:, [cols[j]]])

    r.step() # Iterate the simulation

# Save Data Locally as Numpy
np.save('goal_distance_data', goal_distance)
np.save('inter_robot_distance_data', robot_distance)

# Save Data Locally as CSV Text File
np.savetxt('goal_distance_data.txt', goal_distance, delimiter=',')
np.savetxt('inter_robot_distance_data.txt', robot_distance.T, delimiter=',')

# Call at end of script to print debug information
r.call_at_scripts_end()
```

Listing 12: `leader_follower_data_saving.py` example

Algorithm 11 Pseudo Code `leader_follower_data_saving.py`

```
Import important libraries ▷ [1-8]
Set experiment constants, including the number of iterations and robots ▷ [10, 12]
Define waypoints for the leader and the distance threshold to switch waypoints ▷ [14,16]
Preallocate data arrays for saving distances ▷ [19-21]
Create the desired Laplacian for the formation ▷ [24-28]
Find connections between robots ▷ [31]
Initialize the velocity vector for computational efficiency ▷ [33]
Initialize leader state and set maximum linear speed ▷ [35,37]
Create gains for the formation control algorithm and set desired distances ▷ [39,41]
Define initial conditions to avoid barrier use at the beginning ▷ [43]
Instantiate the Robotarium object with the specified parameters ▷ [45]
Create mapping from single integrator states to unicycle states and vice versa. ▷ [47]
Single Integrator to Unicycle Velocity Commands ▷ [49]
Create barrier certificates to avoid collision ▷ [51]
Create single integrator position controller ▷ [53]
for each iteration from 1 to iterations do
    Get the most recent pose information from the Robotarium ▷ [57]
    Convert unicycle poses to single-integrator poses ▷ [59]
    for each follower robot i from 1 to N − 1 do
        Initialize the velocity vector for robot i ▷ [63]
        Get the topological neighbors of robot i ▷ [64]
        for each neighbor j of robot i do
            Perform formation control to maintain desired distances ▷ [67]
        end for
    end for
    Set the waypoint for the leader ▷ [70]
    Use the position controller to drive the leader to the waypoint ▷ [72]
    if leader is close enough to the waypoint then
        Update the state and save goal distance data ▷ [73-75]
    end if
    Threshold control inputs to maintain specified magnitude limit ▷ [78-80]
    Use barrier certificates to ensure safe control inputs ▷ [[82]
    Convert single-integrator inputs to unicycle commands ▷ [84]
    Set the velocities of the robots ▷ [86]
    Save inter-robot distance data ▷ [89-92]
    Iterate the simulation ▷ [94]
end for
Save data locally as Numpy and CSV text files ▷ [97-102]
Call this script to debug at the end of the experiment ▷ [105]
```

3.7 consensus

3.7.1 Consensus Algorithm for Multiple Robots

This code controls multiple robots to achieve consensus on their positions using a cycle graph Laplacian. The robots use Single Integrator dynamics for control inputs and Unicycle dynamics for actual movement. Barrier certificates ensure safe control inputs to avoid collisions and prevent robots from driving off the testbed.

```python
import rps.robotarium as robotarium
from rps.utilities.graph import *
from rps.utilities.transformations import *
from rps.utilities.barrier_certificates import *
from rps.utilities.misc import *
from rps.utilities.controllers import *
import numpy as np

N = 12 # Number of robots

r = robotarium.Robotarium(number_of_robots=N, show_figure=True, sim_in_real_time=True) # Instantiate Robotarium object

iterations = 1000 # Run the simulation/experiment for 1000 steps

si_barrier_cert = create_single_integrator_barrier_certificate_with_boundary() # Single-integrator barrier certificates

si_to_uni_dyn, uni_to_si_states = create_si_to_uni_mapping() # Create SI to UNI dynamics tranformation

L = cycle_GL(N) # Generated a connected graph Laplacian (for a cylce graph).

for k in range(iterations):

    x = r.get_poses()  # Get the poses of the robots

    x_si = uni_to_si_states(x) # Convert to single-integrator poses

    si_velocities = np.zeros((2, N)) # Initialize the single-integrator control inputs

    for i in range(N): # For each robot...

        j = topological_neighbors(L, i) # Get the neighbors of robot 'i' (encoded in the graph Laplacian)

        si_velocities[:, i] = np.sum(x_si[:, j] - x_si[:, i, None], 1) # Compute the consensus algorithm

    si_velocities_safe = si_barrier_cert(si_velocities, x_si) # Use the barrier certificate to avoid collisions

    dxu_safe = si_to_uni_dyn(si_velocities_safe, x) # Transform single integrator to unicycle

    r.set_velocities(np.arange(N), dxu_safe) # Set the velocities of agents 1,...,N to dxu_safe

    r.step() # Iterate the simulation

r.call_at_scripts_end() # Call at end of script to print debug information
```

Listing 13: `consensus.py` example

Algorithm 12 Pseudo Code `consensus.py`

```
Import important libraries ▷ [1-7]
Set the number of robots for the simulation ▷ [9]
Instantiate the Robotarium object with the specified parameters ▷ [11]
Set the number of iterations for the simulation ▷ [13]
Create barrier certificates to avoid collisions ▷ [15]
Create mapping from single integrator states to unicycle states and vice versa ▷ [17]
Generate a connected graph Laplacian for a cycle graph ▷ [19]
for each iteration from 1 to iterations do
    Get the current poses of the robots ▷ [23]
    Convert unicycle states to single integrator states ▷ [25]
    Initialize the single-integrator control inputs ▷ [27]
    for each robot i from 1 to N do
        Get the neighbors of robot i (encoded in the graph Laplacian) ▷ [31]
        Compute the consensus algorithm ▷ [33]
    end for
    Use the barrier certificate to avoid collisions ▷ [35]
    Transform single-integrator control inputs to unicycle commands ▷ [37]
    Set the velocities of the robots ▷ [39]
    Iterate the simulation ▷ [41]
end for
Call this script to print debug information and for your script to run on the Robotarium ▷ [43]
```

3.8 plotting

3.8.1 Robots Form a Circle and Switch Positions with Plotting

This code controls multiple robots to form a circle and switch positions while avoiding collisions using Single Integrator dynamics for control inputs and Unicycle dynamics for actual movement. Barrier certificates ensure safe control inputs, and plotting parameters are used for visualization.

```python
import rps.robotarium as robotarium
from rps.utilities.transformations import *
from rps.utilities.barrier_certificates import *
from rps.utilities.misc import *
from rps.utilities.controllers import *
import numpy as np
import time

N = 10 # Number of robots

r = robotarium.Robotarium(number_of_robots=N, show_figure=True, sim_in_real_time=True) # Instantiate the Robotarium object with these parameters

num_cycles=2 # How many times should the robots form the circle?

count = -1 # How many times have they formed the circle? (starts at -1 since initial formation will increment the count)

safety_radius = 0.17 # Default Barrier Parameters

# Plotting Parameters
CM = np.random.rand(N,3) # Random Colors array for plotting
safety_radius_marker_size = determine_marker_size(r,safety_radius) # Will scale the plotted markers to be the diameter of provided argument (in meters)
font_height_meters = 0.1
font_height_points = determine_font_size(r,font_height_meters) # Will scale the plotted font height to that of the provided argument (in meters)

x=r.get_poses() # Get the poses of robots

g = r.axes.scatter(x[0,:], x[1,:], s=np.pi/4*safety_radius_marker_size, marker='o', facecolors='none',edgecolors=CM,linewidth=7) # Initial plots

#g = r.axes.plot(x[0,:], x[1,:], markersize=safety_radius_marker_size, linestyle='none', marker='o', markerfacecolor='none', markeredgecolor=color[CM],linewidth=7)

r.step() # Iterate the Simulation

si_barrier_cert = create_single_integrator_barrier_certificate() #  Single-integrator barrier certificates

si_position_controller = create_si_position_controller() # Create single integrator position controller

si_to_uni_dyn, uni_to_si_states = create_si_to_uni_mapping() # Create SI to UNI dynamics tranformation

# This portion of the code generates points on a circle enscribed in a 6x6 square that's centered on the origin. The robots switch positions on the circle.
xybound = np.array([-1.2, 1.2, -1, 1])
p_theta = 2*np.pi*(np.arange(0, 2*N, 2)/(2*N))
p_circ = np.vstack([
            np.hstack([xybound[1]*np.cos(p_theta), xybound[1]*np.cos(p_theta+np.pi)]),
            np.hstack([xybound[3]*np.sin(p_theta), xybound[3]*np.sin(p_theta+np.pi)])
            ])

# These variables are so we can tell when the robots should switch positions on the circle.
flag = 0
x_goal = p_circ[:, :N]

while(1): # While True

    x = r.get_poses() # Get the poses of the robots

    # Update Plotted Visualization
    g.set_offsets(x[:2,:].T)
    # This updates the marker sizes if the figure window size is changed. This should be removed when submitting to the Robotarium.
    g.set_sizes([determine_marker_size(r,safety_radius)])

    x_si = uni_to_si_states(x) # Single-integrator to unicycle mapping

    si_velocities = np.zeros((2, N)) # Initialize a velocities variable

```

```
    if(np.linalg.norm(x_goal - x_si) < 0.05): # Check if all the agents are close enough to the goals
        flag = 1-flag
        count += 1

    if count == num_cycles: # If True, end simulation
        break

    # Switch goals depending on the state of flag (goals switch to opposite sides of the circle)
    if(flag == 0):
        x_goal = p_circ[:, :N]
    else:
        x_goal = p_circ[:, N:]

    dxi = si_position_controller(x_si,x_goal) # Use a position controller to drive to the goal position

    dxi_safe = si_barrier_cert(dxi, x_si) # Use the barrier certificates to make sure that the agents don't collide

    dxu_safe = si_to_uni_dyn(dxi_safe, x) # Single-integrator-to-unicycle mapping

    r.set_velocities(np.arange(N), dxu_safe) # Set the velocities of agents 1,...,N to dxu_safe

    r.step() # Iterate the simulation

# Caption before finishing the simulation
finished_caption = "All robots safely reached \n their destination"
finished_label = r.axes.text(0,0,finished_caption,fontsize=font_height_points,
↪  color='k',fontweight='bold',horizontalalignment='center',verticalalignment='center',zorder=20)
r.step()
time.sleep(5)

r.call_at_scripts_end() # Call at end of script to print debug information
```

Listing 14: barrier_certificates_with_plotting.py example

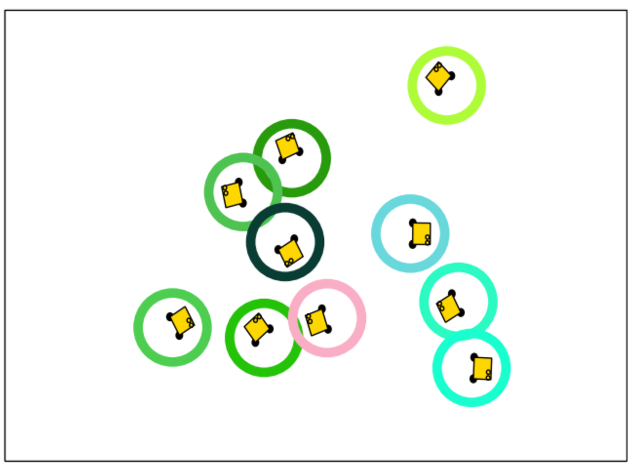


Figure 25: Barrier Certificates with Plotting

Algorithm 13 Pseudo Code `barrier_certificates_with_plotting.py`

```
Import important libraries ▷ [1-7]
Set the number of robots for the simulation ▷ [9]
Instantiate the Robotarium object with the specified parameters ▷ [11]
Set the number of cycles the robots should form the circle ▷ [13,15]
Define default barrier parameters ▷ [17]
Create array with different colors ▷ [20]
Create the appropriate Marker Size ▷ [21]
Create the appropriate Font Size ▷ [23]
Get the current pose of the robots ▷ [25]
Initialize plots for robot positions ▷ [27]
Iterate the simulation ▷ [31]
Create barrier certificates to avoid collisions ▷ [33]
Create single integrator position controller ▷ [35]
Create mapping from single integrator states to unicycle states and vice versa ▷ [37]
Generate points on a circle inscribed in a 6x6 square ▷ [40-45]
Set initial goals and flags for switching positions ▷ [48-49]
while True do
    Get the current poses of the robots ▷ [53]
    Update plotted visualization ▷ [56-58]
    Convert unicycle states to single integrator states ▷ [60]
    Initialize the single-integrator control inputs ▷ [62]
    if all agents are close enough to the goals then
        Switch the flag and increment the count ▷ [65-66]
    end if
    if count equals the number of cycles then
        Break the loop ▷ [69]
    end if
    if flag is 0 then
        Set goals to one side of the circle ▷ [72]
    else
        Set goals to the opposite side of the circle ▷ [75]
    end if
    Use the position controller to drive to the goal position ▷ [77]
    Use the barrier certificates to avoid collisions ▷ [79]
    Convert single-integrator control inputs to unicycle commands ▷ [81]
    Set the velocities of the robots ▷ [83]
    Iterate the simulation ▷ [85]
end while
Display a finished caption and wait for a few seconds ▷ [87-89]
Iterate the simulation ▷ [90]
Call this script to print debug information and for your script to run on the Robotarium ▷ [93]
```

3.8.2 Leader-Follower Formation Control with Plotting

This code controls a leader-follower formation of robots. The leader robot moves towards predefined waypoints, while follower robots maintain a desired distance from each other. The code uses Single Integrator dynamics for generating control inputs and Unicycle dynamics for actual robot movement. Barrier certificates ensure safe control inputs, and various plotting parameters are used for visualization.

```
import rps.robotarium as robotarium
from rps.utilities.transformations import *
from rps.utilities.graph import *
from rps.utilities.barrier_certificates import *
from rps.utilities.misc import *
from rps.utilities.controllers import *
import numpy as np

iterations = 5000 # Run the simulation/experiment for 5000 steps

N = 4 # Number of robots

waypoints = np.array([[-1, -1, 1, 1],[0.8, -0.8, -0.8, 0.8]]) # Waypoints the leader moves to.

close_enough = 0.03; # How close the leader must get to the waypoint to move to the next one.

# Create the desired Laplacian
followers = -completeGL(N-1)
L = np.zeros((N,N))
L[1:N,1:N] = followers
L[1,1] = L[1,1] + 1
L[1,0] = -1

# Find connections
[rows,cols] = np.where(L==1)

dxi = np.zeros((2,N)) # For computational/memory reasons, initialize the velocity vector

state = 0 # Initialize leader state

magnitude_limit = 0.15 # Limit maximum linear speed of any robot

formation_control_gain = 10 # Gains for the Formation Control

desired_distance = 0.3 # Desired distance between the Robots

initial_conditions = np.array([[0, 0.5, 0.3, -0.1],[0.5, 0.5, 0.2, 0],[0, 0, 0, 0]]) # Initial Positions

r = robotarium.Robotarium(number_of_robots=N, show_figure=True, initial_conditions=initial_conditions, sim_in_real_time=True) # Instantiate the Robotarium

_, uni_to_si_states = create_si_to_uni_mapping() # Unicycle to Single Integrator States Mapping

si_to_uni_dyn = create_si_to_uni_dynamics() # Single Integrator to Unicycle Velocity Commands

si_barrier_cert = create_single_integrator_barrier_certificate_with_boundary() # Single-integrator barrier certificates

leader_controller = create_si_position_controller(velocity_magnitude_limit=0.1) # Single-integrator position controller

# Plotting Parameters
CM = np.random.rand(N,3) # Random Colors array for plotting
marker_size_goal = determine_marker_size(r,0.2) # Will scale the plotted markers to be the diameter of provided argument (in meters)
font_size_m = 0.1
font_size = determine_font_size(r,font_size_m) # Will scale the plotted font height to that of the provided argument (in meters)
line_width = 5

# Create goal text and markers

# Text with goal identification
goal_caption = ['G{0}'.format(ii) for ii in range(waypoints.shape[1])]
# Plot text for caption
waypoint_text = [r.axes.text(waypoints[0,ii], waypoints[1,ii], goal_caption[ii], fontsize=font_size, color='k',fontweight='bold',horizontalalignment='center',verticalalignment='center',zorder=-2)
for ii in range(waypoints.shape[1])]
g = [r.axes.scatter(waypoints[0,ii], waypoints[1,ii], s=marker_size_goal, marker='s', facecolors='none',edgecolors=CM[ii,:],linewidth=line_width,zorder=-2)
for ii in range(waypoints.shape[1])]

# Plot Graph Connections
```

```
x = r.get_poses() # Get the pose of robots

linked_follower_index = np.empty((2,3))
follower_text = np.empty((3,0))
for jj in range(1,int(len(rows)/2)+1):
        linked_follower_index[:,[jj-1]] = np.array([[rows[jj]],[cols[jj]]])
        follower_text = np.append(follower_text,'{0}'.format(jj))

line_follower = [r.axes.plot([x[0,rows[kk]], x[0,cols[kk]]],[x[1,rows[kk]],
↪  x[1,cols[kk]]],linewidth=line_width,color='b',zorder=-1)
 for kk in range(1,N)]
line_leader = r.axes.plot([x[0,0],x[0,1]],[x[1,0],x[1,1]],linewidth=line_width,color='r',zorder = -1)
follower_labels = [r.axes.text(x[0,kk],x[1,kk]+0.15,follower_text[kk-1],fontsize=font_size,
↪  color='b',fontweight='bold',horizontalalignment='center',verticalalignment='center',zorder=0)
for kk in range(1,N)]
leader_label = r.axes.text(x[0,0],x[1,0]+0.15,"Leader",fontsize=font_size,
↪  color='r',fontweight='bold',horizontalalignment='center',verticalalignment='center',zorder=0)

r.step() # Iterate the simulation

for t in range(iterations):

    x = r.get_poses() # Get the pose of robots

    xi = uni_to_si_states(x) # Transform poses to Single Integrator

    for q in range(N-1):
        follower_labels[q].set_position([xi[0,q+1], xi[1,q+1]+0.15])
        follower_labels[q].set_fontsize(determine_font_size(r, font_size_m))
        line_follower[q][0].set_data([x[0, rows[q+1]], x[0, cols[q+1]]], [x[1, rows[q+1]], x[1, cols[q+1]]])

    leader_label.set_position([xi[0,0], xi[1,0]+0.15])
    leader_label.set_fontsize(determine_font_size(r, font_size_m))
    line_leader[0].set_data([x[0,0], x[0,1]], [x[1,0], x[1,1]])

    # This updates the marker sizes if the figure window size is changed. This should be removed when submitting to the
    ↪  Robotarium.
    for q in range(waypoints.shape[1]):
        waypoint_text[q].set_fontsize(determine_font_size(r, font_size_m))
        g[q].set_sizes([determine_marker_size(r, 0.2)])


    # Followers
    for i in range(1, N):
        dxi[:, [i]] = np.zeros((2, 1)) # Zero velocities
        neighbors = topological_neighbors(L, i) # Get the topological neighbors of agent i

        for j in neighbors:
            dxi[:, [i]] += formation_control_gain * (np.power(np.linalg.norm(x[:2, [j]] - x[:2, [i]]), 2) -
            ↪  np.power(desired_distance, 2)) * (x[:2, [j]] - x[:2, [i]])

    # Leader
    waypoint = waypoints[:, state].reshape((2, 1))

    dxi[:, [0]] = leader_controller(x[:2, [0]], waypoint)
    if np.linalg.norm(x[:2, [0]] - waypoint) < close_enough: # Leader moves to the waypoint and updates to the next one if
    ↪  close
        state = (state + 1) % 4

    # Keep single integrator control vectors under specified magnitude
    norms = np.linalg.norm(dxi, 2, 0)  # Threshold control inputs
    idxs_to_normalize = (norms > magnitude_limit)  # Threshold control inputs
    dxi[:, idxs_to_normalize] *= magnitude_limit / norms[idxs_to_normalize]  # Threshold control inputs

    dxi_safe = si_barrier_cert(dxi, x[:2, :]) # Use Barrier to avoid collision

    dxu_safe = si_to_uni_dyn(dxi_safe, x) # Convert single-integrator to unicycle commands

    r.set_velocities(np.arange(N), dxu_safe) # Set the velocities of agents 1,...,N to dxu_safe

    r.step() # Iterate the simulation

r.call_at_scripts_end() # Call at end of script to print debug information
```

Listing 15: `leader_follower_with_plotting.py` example

Algorithm 14 Pseudo Code `leader_follower_with_plotting.py`

```
Import important libraries ▷ [1-7]
Set number of iterations and number of robots ▷ [9,11]
Define waypoints for the leader and the distance threshold to switch waypoints ▷ [13,15]
Create the desired Laplacian and find connections ▷ [18-25]
Initialize the velocity vector ▷ [27]
Initialize leader state and set magnitude limit ▷ [29,31]
Create gains for the formation control algorithm and desired distance ▷ [33,35]
Set initial conditions to avoid barrier use at the beginning ▷ [37]
Instantiate the Robotarium object with specified parameters ▷ [39]
Create mapping from single integrator states to unicycle states and vice versa. ▷ [41]
Create Single Integrator to unicycle dynamic mapping ▷ [43]
Create barrier certificates to avoid collision ▷ [45]
Create single integrator position controller for leader ▷ [47]
Create array with different colors ▷ [50]
Create the appropriate Marker Size ▷ [51]
font size ▷ [52]
Create the appropriate Font Size ▷ [53]
Desired line width ▷ [54]
Create goal text and markers ▷ [59-64]
Plot graph connections ▷ [68-81]
Step the simulation to update visualization ▷ [83]
for each iteration (1 to iterations) do
    Get the most recent pose information from the Robotarium ▷ [87]
    Convert unicycle poses to single-integrator poses ▷ [89]
    Update plot handles for followers and leader ▷ [91-98]
    Update marker sizes if the figure window size changes ▷ [101-103]
    Followers:
    for each follower robot (1 to N-1) do
        Zero velocities and get topological neighbors ▷ [108-109]
        for each neighbor do
            Compute formation control inputs ▷ [112]
        end for
    end for
    Leader:
    Set goal waypoint and compute control inputs ▷ [115, 117]
    if leader is close enough to the waypoint then
        Update the waypoint state ▷ [119]
    end if
    Threshold control inputs to keep them under the magnitude limit ▷ [122-124]
    Use barriers certification ▷ [126]
    Convert single-integrator to unicycle commands ▷ [128]
    Set the velocities of the robots ▷ [130]
    Iterate the simulation ▷ [132]
end for
Call this script to print debug information and for your script to run on the Robotarium ▷ [134]
```

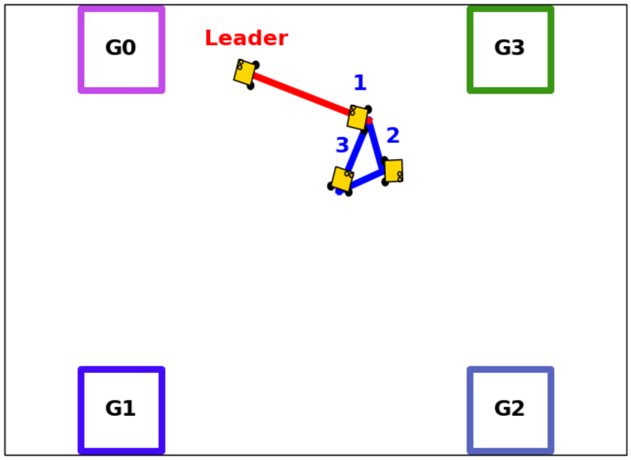


Figure 26: Leader Follower with Plotting

3.8.3 Robots go to a point using Single Integrator and Unicycle Dynamics with background

This code controls multiple robots that move towards specified goal points while avoiding collisions with other robots and boundaries. The Single Integrator model is used for generating control inputs (`dxi`), and the Unicycle model is used for actual robot movement (`dxu`). The control inputs are converted from Single Integrator to Unicycle commands using a specified transformation. Additionally, the background of the testbed can be changed.

```python
import rps.robotarium as robotarium
from rps.utilities.transformations import *
from rps.utilities.barrier_certificates import *
from rps.utilities.misc import *
from rps.utilities.controllers import *
import numpy as np
import time

N = 4 # Number of Robots

initial_conditions = np.array(np.mat('1 0.5 -0.5 0; 0.8 -0.3 -0.75 0.1; 0 0 0 0'))

r = robotarium.Robotarium(number_of_robots=N, show_figure=True, initial_conditions=initial_conditions, sim_in_real_time=False) # Instantiate Robotarium

goal_points = 0.9*np.array([[-1, 1, -1, 1],[-1, -1, 1, 1],[0,0,0,0]]) # Goal Points

single_integrator_position_controller = create_si_position_controller() # Single integrator position controller

si_barrier_cert = create_single_integrator_barrier_certificate_with_boundary() # Barrier certificates to avoid collision

_, uni_to_si_states = create_si_to_uni_mapping() # Unicycle to Single Integrator States Mapping

si_to_uni_dyn = create_si_to_uni_dynamics_with_backwards_motion() # Single Integrator to Unicycle Velocity Commands

# Read in and scale image
gt_img = plt.imread('GTLogo.png')
x_img = np.linspace(-1.0, 1.0, gt_img.shape[1])
y_img = np.linspace(-1.0, 1.0, gt_img.shape[0])

gt_img_handle = r.axes.imshow(gt_img, extent=(-1, 1, -1, 1))

x = r.get_poses() # Get poses of the Robots

x_si = uni_to_si_states(x) # Transform poses to Single Integrator

r.step()  # Iterate the simulation

# While all Robots are not in the goal points...
while (np.size(at_pose(np.vstack((x_si,x[2,:])), goal_points, rotation_error=100)) != N):

    x = r.get_poses() # Get the poses of robots

    x_si = uni_to_si_states(x) # Transform poses to Single Integrator

    dxi = single_integrator_position_controller(x_si, goal_points[:2][:]) # Create single-integrator control inputs

    dxi_safe = si_barrier_cert(dxi, x_si) # Create safe control inputs (i.e., no collisions)

    dxu_safe = si_to_uni_dyn(dxi_safe, x) # Transform single integrator velocity commands to unicycle

    r.set_velocities(np.arange(N), dxu_safe) # Set the velocities  using the Unicycle Commands

    r.step()  # Iterate the simulation

r.call_at_scripts_end() # Call at end of script to print debug information
```

Listing 16: `si_go_to_point_gt.py` example

Algorithm 15 Pseudo Code `si_go_to_point_gt.py`

1: Import important libraries ▷ [1-7]
2: Set the number of robots and initial conditions ▷ [9-11]
3: Initialize the Robotarium object ▷ [13]
4: Define goal points for the robots ▷ [15]
5: Create single integrator position controller ▷ [17]
6: Create barrier certificates to avoid collisions ▷ [19]
7: Create mapping from unicycle states to single integrator states ▷ [21]
8: Create mapping from single integrator velocity commands to unicycle velocity commands ▷ [23]
9: Read in and scale image for visualization ▷ [26-30]
10: Define initial robot poses ▷ [32]
11: Convert unicycle states to single integrator states ▷ [34]
12: Iterate the simulation ▷ [36]
13: **while** number of robots at the required poses is less than N **do**
14: Get current poses of the robots ▷ [41]
15: Convert unicycle states to single integrator states ▷ [43]
16: Create single-integrator control inputs ▷ [45]
17: Create safe control inputs to avoid collisions ▷ [47]
18: Transform single integrator velocity commands to unicycle ▷ [49]
19: Set the velocities of the robots ▷ [51]
20: Iterate the simulation ▷ [53]
21: **end while**
22: Call this script to print debug information and for your script to run on the Robotarium ▷ [55]

Figure 27: Si go to point with GT logo at the Background

3.8.4 Robots go to a point using Single Integrator with Plotting

This code controls multiple robots that move towards specified goal points while avoiding collisions with other robots and boundaries. The Single Integrator model is used for generating control inputs (`dxi`), and the Unicycle model is used for actual robot movement (`dxu`). The control inputs are converted from Single Integrator to Unicycle commands using a specified transformation. Additionally, the safety radius of robots and goal points are visualized in real-time.

```python
import rps.robotarium as robotarium
from rps.utilities.transformations import *
from rps.utilities.barrier_certificates import *
from rps.utilities.misc import *
from rps.utilities.controllers import *
import numpy as np
import time

N = 5 # Number of Robots

initial_conditions = np.array(np.mat('1 0.5 -0.5 0 0.28; 0.8 -0.3 -0.75 0.1 0.34; 0 0 0 0 0')) # Initial Positions

r = robotarium.Robotarium(number_of_robots=N, show_figure=True, initial_conditions=initial_conditions, sim_in_real_time=False) # Instantiate Robotarium

goal_points = generate_initial_conditions(N, width=r.boundaries[2]-2*r.robot_diameter, height = r.boundaries[3]-2*r.robot_diameter, spacing=0.5) # Define goal points by removing orientation from poses

single_integrator_position_controller = create_si_position_controller() # Single integrator position controller

si_barrier_cert = create_single_integrator_barrier_certificate_with_boundary() # Barrier certificates to avoid collision

_, uni_to_si_states = create_si_to_uni_mapping() # Unicycle to Single Integrator States Mapping

si_to_uni_dyn = create_si_to_uni_dynamics_with_backwards_motion() # Single Integrator to Unicycle Velocity Commands

x = r.get_poses() # Get poses of the Robots

x_si = uni_to_si_states(x) # Transform poses to Single Integrator

# Plotting Parameters
CM = np.random.rand(N,3) # Random Colors array for plotting
robot_marker_size_m = 0.15
marker_size_goal = determine_marker_size(r,goal_marker_size_m) # Will scale the plotted markers to be the diameter of provided argument (in meters)
marker_size_robot = determine_marker_size(r, robot_marker_size_m) # Will scale the plotted markers to be the diameter of provided argument (in meters)
font_size = determine_font_size(r,0.1) # Will scale the plotted font height to that of the provided argument (in meters)
line_width = 5

# Create Goal Point Markers
# Text with goal identification
goal_caption = ['G{0}'.format(ii) for ii in range(goal_points.shape[1])]
#Plot text for caption
goal_points_text = [r.axes.text(goal_points[0,ii], goal_points[1,ii], goal_caption[ii], fontsize=font_size, color='k',fontweight='bold',horizontalalignment='center',verticalalignment='center',zorder=-2)
for ii in range(goal_points.shape[1])]
goal_markers = [r.axes.scatter(goal_points[0,ii], goal_points[1,ii], s=marker_size_goal, marker='s', facecolors='none',edgecolors=CM[ii,:],linewidth=line_width,zorder=-2)
for ii in range(goal_points.shape[1])]
robot_markers = [r.axes.scatter(x[0,ii], x[1,ii], s=marker_size_robot, marker='o', facecolors='none',edgecolors=CM[ii,:],linewidth=line_width)
for ii in range(goal_points.shape[1])]

r.step()  # Iterate the simulation

# While all Robots are not in the goal points...
while (np.size(at_pose(np.vstack((x_si,x[2,:])), goal_points, rotation_error=100)) != N):

    x = r.get_poses() # Get poses of Robots

    x_si = uni_to_si_states(x) # Transform poses to Single Integrator

    # Update Plot
    # Update Robot Marker Plotted Visualization
    for i in range(x.shape[1]):
        robot_markers[i].set_offsets(x[:2,i].T)
        # This updates the marker sizes if the figure window size is changed.
        # This should be removed when submitting to the Robotarium.
        robot_markers[i].set_sizes([determine_marker_size(r, robot_marker_size_m)])
```

```
64
65      for j in range(goal_points.shape[1]):
66          goal_markers[j].set_sizes([determine_marker_size(r, goal_marker_size_m)])
67
68      dxi = single_integrator_position_controller(x_si, goal_points[:2][:]) # Create single-integrator control inputs
69
70      dxi_safe = si_barrier_cert(dxi, x_si) # Create safe control inputs (i.e., no collisions)
71
72      dxu_safe = si_to_uni_dyn(dxi_safe, x) # Transform single integrator velocity commands to unicycle
73
74      r.set_velocities(np.arange(N), dxu_safe) # Set the velocities  using the Unicycle Commands
75
76      r.step()  # Iterate the simulation
77
78  r.call_at_scripts_end() # Call at end of script to print debug information
```

Listing 17: `si_go_to_point_with_plotting.py` example

Algorithm 16 Pseudo Code `si_go_to_point_with_plotting.py`

1: Import important libraries ▷ [1-7]
2: Set the number of robots and initial conditions ▷ [9-11]
3: Initialize the Robotarium object ▷ [13]
4: Define goal points for the robots ▷ [15]
5: Create single integrator position controller ▷ [17]
6: Create barrier certificates to avoid collisions ▷ [19]
7: Create mapping from unicycle states to single integrator states ▷ [21]
8: Create mapping from single integrator velocity commands to unicycle velocity commands ▷ [23]
9: Define initial robot poses ▷ [25]
10: Convert unicycle states to single integrator states ▷ [27]
11: Create array with different colors ▷ [30]
12: Goal Marker Size ▷ [31]
13: Appropriate Marker Size Goal ▷ [33]
14: Robot Marker Size ▷ [32]
15: Appropriate Marker Size Robot ▷ [33]
16: Create the appropriate Font Size ▷ [34]
17: Line width ▷ [35]
18: Create the text with Goal Identification ▷ [39]
19: Create Plot for Caption ▷ [41-46]
20: Iterate the simulation ▷ [48]
21: **while** number of robots at the required poses is less than N **do**
22:     Get current poses of the robots ▷ [53]
23:     Convert unicycle states to single integrator states ▷ [55]
24:     Update robot marker plotted visualization ▷ [59-66]
25:     Create single-integrator control inputs ▷ [68]
26:     Create safe control inputs to avoid collisions ▷ [70]
27:     Transform single integrator velocity commands to unicycle ▷ [72]
28:     Set the velocities of the robots ▷ [74]
29:     Iterate the simulation ▷ [76]
30: **end while**
31: Call this script to print debug information and for your script to run on the Robotarium ▷ [78]

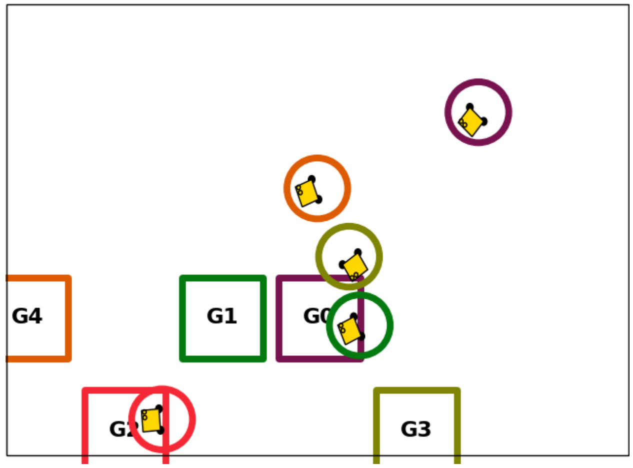


Figure 28: si go to point with plotting

3.8.5 Robots go to a point using Unicycle Pose Controller with Plotting

This code controls multiple robots that move towards specified goal points while avoiding collisions with other robots and boundaries. The Unicycle model is used for generating control inputs (`dxu`). Additionally, the positions and orientations of robots and goal points are visualized in real-time.

```python
import rps.robotarium as robotarium
from rps.utilities.transformations import *
from rps.utilities.barrier_certificates import *
from rps.utilities.misc import *
from rps.utilities.controllers import *
import numpy as np
import time

N = 5 # Number of Robots

initial_conditions = np.array(np.mat('1 0.5 -0.5 0 0.28; 0.8 -0.3 -0.75 0.1 0.34; 0 0 0 0 0')) # Initial position

r = robotarium.Robotarium(number_of_robots=N, show_figure=True, initial_conditions=initial_conditions,sim_in_real_time=True) # Instantiate Robotarium object

goal_points = generate_initial_conditions(N, width=r.boundaries[2]-2*r.robot_diameter, height = r.boundaries[3]-2*r.robot_diameter, spacing=0.5) # Define goal points by removing orientation from poses

unicycle_pose_controller = create_hybrid_unicycle_pose_controller() # Create unicycle hybrid pose controller

uni_barrier_cert = create_unicycle_barrier_certificate() # Create barrier certificates to avoid collision

x = r.get_poses() # Get the poses of the robots

# Plotting Parameters
CM = np.random.rand(N,3) # Random Colors array for plotting
goal_marker_size_m = 0.2
robot_marker_size_m = 0.15
marker_size_goal = determine_marker_size(r,goal_marker_size_m) # Will scale the plotted markers to be the diameter of provided argument (in meters)
marker_size_robot = determine_marker_size(r, robot_marker_size_m) # Will scale the plotted markers to be the diameter of provided argument (in meters)
font_size = determine_font_size(r,0.1) # Will scale the plotted font height to that of the provided argument (in meters)
line_width = 5

# Create Goal Point Markers
# Text with goal identification
goal_caption = ['G{0}'.format(ii) for ii in range(goal_points.shape[1])]
# Arrow for desired orientation
goal_orientation_arrows = [r.axes.arrow(goal_points[0,ii], goal_points[1,ii], goal_marker_size_m*np.cos(goal_points[2,ii]), goal_marker_size_m*np.sin(goal_points[2,ii]), width = 0.02, length_includes_head=True, color = CM[ii,:], zorder=-2)
for ii in range(goal_points.shape[1])]
# Plot text for caption
goal_points_text = [r.axes.text(goal_points[0,ii], goal_points[1,ii], goal_caption[ii], fontsize=font_size, color='k',fontweight='bold',horizontalalignment='center',verticalalignment='center',zorder=-3)
for ii in range(goal_points.shape[1])]
goal_markers = [r.axes.scatter(goal_points[0,ii], goal_points[1,ii], s=marker_size_goal, marker='s', facecolors='none',edgecolors=CM[ii,:],linewidth=line_width,zorder=-3)
for ii in range(goal_points.shape[1])]
robot_markers = [r.axes.scatter(x[0,ii], x[1,ii], s=marker_size_robot, marker='o', facecolors='none',edgecolors=CM[ii,:],linewidth=line_width)
for ii in range(goal_points.shape[1])]

r.step() # Iterate the simulation

# While all Robots are not in the goal points...
while (np.size(at_pose(x, goal_points)) != N):

    x = r.get_poses() # Get poses of agents

    # Update Plot
    # Update Robot Marker Plotted Visualization
    for i in range(x.shape[1]):
        robot_markers[i].set_offsets(x[:2,i].T)
        # This updates the marker sizes if the figure window size is changed.
        # This should be removed when submitting to the Robotarium.
        robot_markers[i].set_sizes([determine_marker_size(r, robot_marker_size_m)])

    for j in range(goal_points.shape[1]):
        goal_markers[j].set_sizes([determine_marker_size(r, goal_marker_size_m)])

    dxu = unicycle_pose_controller(x, goal_points) # Create unicycle control inputs
```

```
65
66      dxu_safe = uni_barrier_cert(dxu, x) # Create safe control inputs (i.e., no collisions)
67
68      r.set_velocities(np.arange(N), dxu_safe) # Set the velocities of agents 1,...,N to dxu_safe
69
70      r.step()  # Iterate the simulation
71
72  r.call_at_scripts_end() # Call at end of script to print debug information
```

Listing 18: `uni_go_to_pose_hybrid_with_plotting.py` example

Algorithm 17 Pseudo Code `uni_go_to_pose_hybrid_with_plotting.py`

```
Import important libraries                                   ▷ [1-7]
Set the number of robots and initial conditions              ▷ [9-11]
Initialize the Robotarium object                             ▷ [13]
Define goal points for the robots                            ▷ [15]
Create unicycle pose controller                              ▷ [17]
Create barrier certificates to avoid collisions              ▷ [19]
Define initial robot poses                                   ▷ [21]
Create array with different colors                           ▷ [24]
Goal Marker Size                                             ▷ [25]
Robot Marker Size                                            ▷ [36]
Appropriate Marker Size Goal                                 ▷ [27]
Appropriate Marker Size Goal                                 ▷ [28]
Create the appropriate Font Size                             ▷ [29]
Line width                                                   ▷ [30]
Create the text with Goal Identification                     ▷ [34]
Create arrow for the desired orientation                     ▷ [36]
Create Plot for Caption                                      ▷ [38-44]
Iteration the simulation                                     ▷ [46]
while number of robots at the required poses is less than N do
    Get current poses of the robots                          ▷ [51]
    Update robot marker plotted visualization                ▷ [53-62]
    Create unicycle control inputs                           ▷ [64]
    Create safe control inputs to avoid collisions           ▷ [66]
    Set the velocities of the robots                         ▷ [68]
    Iterate the simulation                                   ▷ [70]
end while
Call this script to print debug information and for your script to run on the Robotarium   ▷ [72]
```

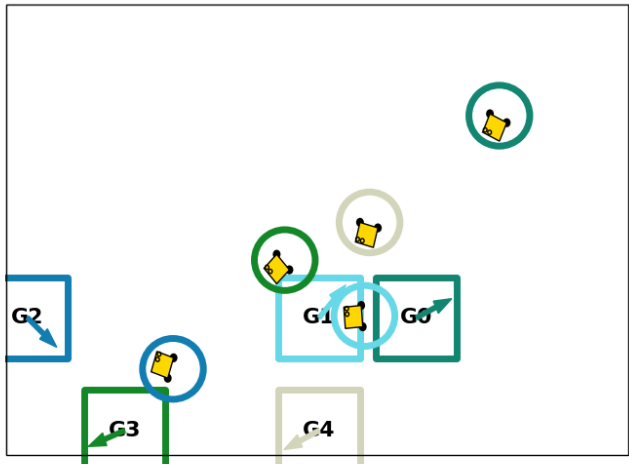


Figure 29: `uni_go_to_pose_hybrid` with plotting

4 Functions

This section describes a variety of functions that are essential for controlling robots in the Robotarium. This functions facilitate different aspects of robot control, such as state transformations, collision avoidance, and movement controllers. Below is a brief introduction to the key functions available in the `rps.utilities` module.

Units in the Robotarium

The Robotarium uses the International System of Units (SI). The following quantities are used in the functions and scripts:

- Time: seconds [s]
- Length: meters [m]
- Angles: radians [rad]
- Linear Velocity: $\frac{meters}{seconds}$ $[\frac{m}{s}]$
- Angular Velocity: $\frac{radians}{seconds}$ $[\frac{rad}{s}]$

4.1 Libraries

In order to have all the utilities that the Robotarium offers you need to import the functions. Do not forget to include this line at the beginning of your code. You can check the other libraries available in Appendix A

```python
import rps.robotarium as robotarium
from rps.utilities.transformations import *
from rps.utilities.graph import *
from rps.utilities.barrier_certificates import *
from rps.utilities.misc import *
from rps.utilities.controllers import *
import numpy as np
import time
```

Listing 19: Robotarium Libraries

In case you do not want to use everything, you can always just import the function that you know you will use.

4.2 Class Robotarium

This section provides an overview of the key functions in the Robotarium API. These functions allow you to control the robots, retrieve their states, and manage the simulation environment.

4.2.1 Initialization

```python
r = robotarium.Robotarium(number_of_robots=N, show_figure=True, initial_conditions=
initial_conditions, sim_in_real_time=False)
```

Listing 20: Initialization

This function initializes the Robotarium class, setting up parameters such as the number of robots, initial conditions, and visualization settings. It takes for inputs:

- **Number of Robots (N)**: The number of Robots. This parameter is used for almost all the functions. Currently the Robotarium accepts up to 20 robots for the experiments.
- **Show Figure**: If True, you can see the simulation. Else, you cannot.
- **Initial Conditions**: Initial conditions are the initial positions that we want for our robots. The initial positions are given in a $3 \times N$ numpy array that specify the pose and orientation of each robot
- **Simulation in Real Time**: If true, the simulation will take exactly 0.033 s. If false, the simulation will run as fast at it can depending the power of your computer.

This function will be always in your code.

4.2.2 Set velocity of Robots

```
r.set_velocities(ids, velocities)
```

Listing 21: Set velocity of Robot function

The set_velocities function allows you to set the linear and angular velocities of the robots. It is important to highlight that to set the velocity of the robots, we always use **unicycle control inputs**. It takes for input:

- **ids**: The ID's of each robot, usually we use a $1 \times N$ numpy array
- **dxu**: A $2 \times N$ numpy array with the unicycle control inputs.

4.2.3 Retrieving Poses

```
r.get_poses()
```

Listing 22: Get poses function

This function returns a $3 \times N$ numpy array with the current states (positions and orientations) of the robots. Every angle is interpreted as a radian on the Robotarium.

4.2.4 Step in the Simulation

```
r.step()
```

Listing 23: Iteration in the Robotarium

The step function advances the simulation by one time step, updating the dynamics of the robots and visualization (if enabled). A time step in the Robotarium is 0.033 seconds. This is important to consider since there is a maximum time per experiment in the Robotarium, so be sure that your experiment can run within the limit.

4.2.5 End of Experiment

```
r.call_at_scripts_end()
```

Listing 24: Function to end experiments

Every-time we finish our simulation we need to include this function in order to print out any errors that might cause our experiment to fail or to be rejected by the Robotarium

4.3 Transformations

Transformations between the unicycle model and the single integrator model are essential for simplifying control design, decoupling control inputs, easing the implementation of algorithms, and facilitating path planning and tracking.

4.3.1 Single Integrator to Unicycle Dynamics

This function returns a function that maps from single-integrator to unicycle dynamics with angular velocity magnitude restrictions.

```
def create_si_to_uni_dynamics(linear_velocity_gain=1, angular_velocity_limit=np.pi):
```

Listing 25: Single Integrator to Unicycle Dynamics Main Function

This function has two parameters that can be adjusted based on the user's requirements

- **Linear Velocity Gain** (v_g): Gain for the unicycle's linear velocity. This parameter must be $v_g > 0$. Different gain values alter the robots' responses.
- **Angular Velocity Limit** (w): Limit for the angular velocity. This parameter must be $|\omega| \leq \omega_{max}$.

The function returns a mapping function from single-integrator to unicycle dynamics with angular velocity magnitude restrictions.

```
def si_to_uni_dyn(dxi, poses):
```

Listing 26: Single Integrator to Unicycle Dynamics Returned Function

This function maps single-integrator dynamics to unicycle dynamics. It takes the following inputs:

- **dxi**: A $2 \times N$ numpy array with single-integrator control inputs.
- **poses**: A $2 \times N$ numpy array with single-integrator poses.

It returns a $2 \times N$ numpy array of unicycle control inputs (linear velocity v and angular velocity w).

4.3.2 Single-Integrator to Unicycle Dynamics with Backwards Motion

```
def create_si_to_uni_dynamics_with_backwards_motion(linear_velocity_gain=1,
angular_velocity_limit=np.pi):
```

Listing 27: Function to Create a Mapping from Single-Integrator to Unicycle Dynamics

This function has two parameters that can be adjusted based on the user's requirements:

- **Linear Velocity Gain** (v_g): Gain for the unicycle's linear velocity. This parameter must be $v_g > 0$. Different gain values alter the robots' responses.
- **Angular Velocity Limit** (w): Limit for the angular velocity. This parameter must be $|\omega| \leq \omega_{max}$.

The function returns a mapping function from single-integrator to unicycle dynamics with angular velocity magnitude restrictions.

```python
def si_to_uni_dyn(dxi, poses):
```

Listing 28: Single Integrator to Unicycle Dynamics Returned Function

This function maps single-integrator dynamics to unicycle dynamics. It takes the following inputs:

- **dxi**: A $2 \times N$ numpy array with single-integrator control inputs.
- **poses**: A $2 \times N$ numpy array with single-integrator poses.

It returns a $2 \times N$ numpy array of unicycle control inputs (linear velocity v and angular velocity w).

The key difference with the previous function is that this function allows the robots to drive backwards if that direction of linear velocity requires less rotation. In simple terms, this allows the robots to drive in reverse. This can be useful in situations where turning around to face forward would require a large angular change, but simply moving backward would be more efficient and quicker. Imagine the following scenario: a robot needs to move to a target behind it. There are two options:

- The robot can turn around (rotate 180 degrees) and then move forward.
- The robot can move backward directly, requiring minimal or no rotation.

In many cases, moving backward directly can be more efficient because it avoids the need for large rotational movements.

4.3.3 Mapping from Single Integrator to Unicycle Dynamics

```python
def create_si_to_uni_mapping(projection_distance=0.05, angular_velocity_limit=np.pi):
```

Listing 29: Function to Create Single Integrator to Unicycle Dynamics Mapping

This function creates two functions: one for mapping single-integrator dynamics to unicycle dynamics (`si_to_uni_dyn`) and one for mapping unicycle states to single-integrator states (`uni_to_si_states`). This mapping is done by placing a virtual control "point" in front of the unicycle, which helps in simplifying the control of unicycle robots.
This function has two parameters that can be adjusted based on the user's requirements:

- **Projection Distance**: How far ahead to place the virtual control point. This parameter must be positive ($ProjectionDistance > 0$).
- **Angular Velocity Limit**: The maximum angular velocity that can be provided. This parameter must be $|\omega| \leq \omega_{max}$. (Between lines 117 and 121, there is an error; it should be angular_velocity_limit).

The first function it returns is the same as before:

```python
def si_to_uni_dyn(dxi, poses):
```

Listing 30: Single Integrator to Unicycle Dynamics Returned Function

This function maps single-integrator dynamics to unicycle dynamics. It takes the following inputs:

- **dxi**: A $2 \times N$ numpy array with single-integrator control inputs.
- **poses**: A $2 \times N$ numpy array with single-integrator poses.

It returns a $2 \times N$ numpy array of unicycle control inputs (linear velocity v and angular velocity w).

The second function is:

```
def uni_to_si_states(poses):
```

Listing 31: Unicycle to Single Integrator States Mapping Function

This function maps unicycle states to single-integrator states by placing a virtual control point ahead of the unicycle. It takes the following input:

- **poses**: A $3 \times N$ numpy array of unicycle states, where each column represents the x position, y position, and orientation θ.

It returns a $2 \times N$ numpy array of single-integrator states, where each column represents the x and y positions of the virtual control point.

4.3.4 Mapping from Unicycle Dynamics to Single Integrator Using Projection Distance

```
def create_uni_to_si_dynamics(projection_distance=0.05):
```

Listing 32: Function for Mapping from Unicycle Dynamics to Single Integrator Using Projection Distance

This function creates a mapping function from unicycle dynamics to single-integrator dynamics.

```
def uni_to_si_dyn(dxu, poses):
```

Listing 33: Unicycle Dynamics to Single Integrator Returned Function

This function maps unicycle dynamics to single-integrator dynamics. It takes the following inputs:

- **dxu**: A $2 \times N$ numpy array of unicycle control inputs.
- **poses**: A $3 \times N$ numpy array of unicycle poses.
- **Projection Distance**: How far ahead of the unicycle model to place the control point.

It returns a $2 \times N$ numpy array of single-integrator control inputs.

4.4 Controllers

Controllers are necessary to ensure that robots and control systems behave as desired, maintain stability, and handle disturbances and uncertainties.

4.4.1 Position Controller for Single Integrator

```
def create_si_position_controller(x_velocity_gain=1, y_velocity_gain=1,
velocity_magnitude_limit=0.15):
```

Listing 34: Position Controller for Single Integrator Function

This function creates a position controller for a single integrator. It drives a single integrator to a user-defined point using a proportional controller. The inputs of this function are:

- **X Velocity Gain** (v_{x_g}): Gain for the x-direction velocity. This parameter must be $v_{x_g} > 0$. Different gain values alter the robots' responses.
- **Y Velocity Gain** (v_{y_g}): Gain for the y-direction velocity. This parameter must be $v_{y_g} > 0$. Different gain values alter the robots' responses.
- **Velocity Magnitude Limit** ($|v|$): The maximum magnitude of the velocity vector (should be less than the max linear speed of the platform). This parameter must be $|v| \leq V_{max}(0.2\frac{m}{s})$.

The function returns another function that creates a position controller for a single integrator.

```
def si_position_controller(xi, positions):
```

Listing 35: Position Controller for Single Integrator Function

This returned function takes the following inputs:

- x_i: A $2 \times N$ numpy array of the single-integrator states of the robots.
- **positions**: A $2 \times N$ numpy array with the desired points each robot should achieve.

It returns a $2 \times N$ numpy array of single-integrator control inputs.

4.4.2 Unicycle Model Pose Controller Based on Control Lyapunov Function

```
def create_clf_unicycle_position_controller(linear_velocity_gain=0.8, angular_velocity_gain=3):
```

Listing 36: Unicycle Model Pose Controller Function

This function returns a unicycle model pose controller function that allows robots to drive the unicycle model to a given position and orientation. It takes the following parameters:

- **Linear Velocity Gain** (v_g): The gain impacting the produced unicycle linear velocity. This parameter must be $v_g \geq 0$. Different gain values alter the robots' responses.
- **Angular Velocity Gain** (ω_g): The gain impacting the produced unicycle angular velocity. This parameter must be $\omega_g \geq 0$.

The returned function utilizes a Control Lyapunov Function (CLF) to drive a unicycle system to a desired position. This function operates on unicycle states and desired positions to return a unicycle velocity command vector.

```
def position_uni_clf_controller(states, positions):
```

Listing 37: Unicycle Model Pose Controller Returned Function

The inputs required by this function are:

- **states**: A $3 \times N$ numpy array of unicycle states (x, y, θ).
- **positions**: A $3 \times N$ numpy array of the desired positions (x_{goal}, y_{goal}).

It returns a $2 \times N$ numpy array of unicycle control inputs (linear velocity v and angular velocity ω).

4.4.3 Unicycle Model Position Controller Based on Control Lyapunov Function

```
def create_clf_unicycle_pose_controller(approach_angle_gain=1, desired_angle_gain=2.7,
rotation_error_gain=1):
```

Listing 38: Unicycle Model Position Controller

This function returns a controller that will drive a unicycle-modeled agent to a desired pose (position and orientation). The inputs of this function are:

- **Approach Angle Gain**: Affects how the unicycle approaches the desired position.
- **Desired Angle Gain**: Affects how the unicycle approaches the desired angle.
- **Rotation Error Gain**: Affects how quickly the unicycle corrects rotation errors.

The returned function is:

```
def pose_uni_clf_controller(states, poses):
```

Listing 39: Unicycle Model Position Controller Returned Function

The inputs required by this function are:

- **states**: A $3 \times N$ numpy array of unicycle states (x, y, θ).
- **poses**: A $3 \times N$ numpy array of the desired positions $(x_{goal}, y_{goal}, \theta_{goal})$.

It returns a $2 \times N$ numpy array of unicycle control inputs (linear velocity v and angular velocity ω).

4.4.4 Unicycle Model Position Controller Based on Hybrid Controller

```
def create_hybrid_unicycle_pose_controller(linear_velocity_gain=1, angular_velocity_gain=2,
velocity_magnitude_limit=0.15, angular_velocity_limit=np.pi, position_error=0.05,
position_epsilon=0.03, rotation_error=0.05):
```

Listing 40: Unicycle Model Position Controller Based on Hybrid Controller Function

This function returns a controller that drives a unicycle-modeled agent to a pose (x, y, θ). This controller is based on a hybrid approach that drives the robot in a straight line to the desired position and then rotates to the desired orientation. The inputs for this function are:

- **Linear Velocity Gain**: Affects how much the linear velocity is scaled based on the position error.
- **Angular Velocity Gain**: Affects how much the angular velocity is scaled based on the heading error.
- **Velocity Magnitude Limit**: Threshold for the maximum linear velocity that the robot can achieve. $|v| \leq V_{max}(0.2\frac{m}{s})$
- **Angular Velocity Limit**: Threshold for the maximum rotational velocity that the robot can achieve. $|\omega| \leq \omega_{max}$
- **Position Error**: The error tolerance for the final position of the robot.
- **Position Epsilon**: The amount of translational distance allowed during rotation before correcting the position again.
- **Rotation Error**: The error tolerance for the final orientation of the robot.

The returned function is:

```
def pose_uni_hybrid_controller(states, poses, input_approach_state=np.empty([0, 0])):
```

Listing 41: Unicycle Model Position Controller Based on Hybrid Controller Returned Function

The inputs required by this function are:

- **states**: A $3 \times N$ numpy array of unicycle states (x, y, θ).
- **poses**: A $3 \times N$ numpy array of the desired positions $(x_{goal}, y_{goal}, \theta_{goal})$.
- **input_approach_state**: Optional input representing the approach state; it can be an empty array if not used.

It returns a $2 \times N$ numpy array of unicycle control inputs (linear velocity v and angular velocity ω).

4.5 Barrier Certificates

Barrier certificates are used in the Robotarium to ensure that robots avoid collisions while executing user-defined control inputs. Essentially, these certificates enforce a "do not collide" constraint, which is expressed as a differential constraint in terms of the control signal.

Here's how they work:

1. User-Defined Control Input: At any given moment, a user generates a control input that defines how they want the robot to move. This input is typically based on the robot's current task or objective, such as following a path or reaching a target point.
2. Collision Avoidance Constraint: The barrier certificate imposes a constraint that prevents collisions. This constraint is mathematically expressed in a way that considers the control signal and ensures that the robot's trajectory stays clear of obstacles and other robots.
3. Minimally Invasive Adjustments: The key feature of barrier certificates is that they are minimally invasive. This means that they make the smallest possible changes to the user-defined control input to satisfy the "do not collide" constraint. In other words, the barrier certificate adjusts the robot's movement just enough to avoid a collision, while preserving as much of the original control intent as possible.

Barrier certificates are functions that guarantee collision free behavior for robots once provided the control input (velocities) and the states of all the robots considered. Unless you have implemented your own obstacle avoidance program to ensure collisions are avoided, we strongly recommend implementing our provided barrier certificates to ensure your programs are accepted!

4.5.1 Create a Barrier Certificate for a Single-Integrator System

```python
def create_single_integrator_barrier_certificate(barrier_gain=100, safety_radius=0.17,
magnitude_limit=0.2):
```

Listing 42: Barrier Certificate for a Single-Integrator System

This function creates a barrier certificate for a single-integrator system. It returns another function for optimization reasons. The inputs for this function are:

- **Barrier Gain (B_g)**: Controls how quickly agents can approach each other. A lower value implies a slower approach. This parameter is strictly positive ($B_g > 0$).
- **Safety Radius (S_r)**: Determines how far apart the robots will stay. This parameter must be greater than or equal to 0.12 m ($S_r \geq 0.12$).
- **Magnitude Limit ($|v|$)**: Limits the linear speed of the robot. This parameter must be $|v| \leq V_{max}(0.2\frac{m}{s})$.

The returned function is:

```python
def f(dxi, x):
```

Listing 43: Barrier Certificate Returned Function

The inputs required by this function are:

- **dxi**: A $2 \times N$ numpy array of the single-integrator robot velocity commands.
- **x**: A $2 \times N$ numpy array of the robot states.

It returns a $2 \times N$ numpy array of modified velocity commands that ensure safety.

4.5.2 Create a Barrier Certificate for a Single Integrator with Boundary

```
def create_single_integrator_barrier_certificate_with_boundary(barrier_gain=100,
safety_radius=0.17, magnitude_limit=0.2, boundary_points=np.array([-1.6, 1.6, -1.0, 1.0])):
```

Listing 44: Barrier Certificate for a Single-Integrator System with Boundary

This function creates a barrier certificate for a single-integrator system with a rectangular boundary included, ensuring that the robots do not leave the arena. It returns another function for optimization reasons. The inputs for this function are:

- **Barrier Gain (B_g)**: Controls how quickly agents can approach each other. A lower value implies a slower approach. This parameter is strictly positive ($B_g > 0$).
- **Safety Radius (S_r)**: Determines how far apart the robots will stay. This parameter must be greater than or equal to 0.12 m ($S_r \geq 0.12$).
- **Magnitude Limit ($|v|$)**: Limits the linear speed of the robot. This parameter must be $|v| \leq V_{max}(0.2\frac{m}{s})$.
- **Boundary Points**: This array corresponds to the arena's dimensions. It should not be edited.

The returned function is:

```
def f(dxi, x):
```

Listing 45: Barrier Certificate Returned Function

The inputs required by this function are:

- **dxi**: A $2 \times N$ numpy array of the single-integrator robot velocity commands.
- **x**: A $2 \times N$ numpy array of the robot states.

It returns a $2 \times N$ numpy array of modified velocity commands that ensure safety and boundary compliance.

4.5.3 Create a Barrier Certificate for a Single Integrator with Dynamic Gains

```
def create_single_integrator_barrier_certificate2(barrier_gain=100, unsafe_barrier_gain=1e6,
safety_radius=0.17, magnitude_limit=0.2):
```

Listing 46: Barrier Certificate for a Single Integrator with Dynamic Gains Function

This function creates a barrier certificate for a single-integrator system. It returns another function for optimization reasons. This function is different from `create_single_integrator_barrier_certificate()` as it dynamically changes the barrier gain to a large number if the single integrator point $(\dot{x}, \dot{y})$ enters an unsafe region. The inputs for this function are:

- **Barrier Gain (B_g)**: Controls how quickly agents can approach each other. A lower value implies a slower approach. This parameter is strictly positive ($B_g > 0$).
- **Unsafe Barrier Gain ($B_{unsafeg}$)**: Controls how quickly the Barrier Gain changes when entering an unsafe region. This parameter must be positive ($Bunsafeg > 0$).

- **Safety Radius (Sr)**: Determines how far apart the robots will stay. This parameter must be greater than or equal to 0.12 m ($S_r \geq 0.12$).
- **Magnitude Limit ($|v|$)**: Limits the linear speed of the robot. This parameter must be $|v| \leq V_{max}(0.2\frac{m}{s})$.

The returned function is:

```python
def f(dxi, x):
```

Listing 47: Barrier Certificate for a Single Integrator with Dynamic Gains Returned Function

The inputs required by this function are:

- **dxi**: A $2 \times N$ numpy array of the single-integrator robot velocity commands.
- **x**: A $2 \times N$ numpy array of the robot states.

It returns a $2 \times N$ numpy array of modified velocity commands that ensure safety.

4.5.4 Create a Barrier Certificate for a Unicycle Model

```python
def create_unicycle_barrier_certificate(barrier_gain=100, safety_radius=0.12,
projection_distance=0.05, magnitude_limit=0.2):
```

Listing 48: Barrier Certificate for a Unicycle Model Function

This function creates a barrier certificate for a unicycle model to avoid collisions. It uses diffeomorphism mapping and single integrator implementation. For optimization purposes, this function returns a unicycle barrier certificate function. The inputs for this function are:

- **Barrier Gain (B_g)**: Controls how quickly agents can approach each other. A lower value implies a slower approach. This parameter is strictly positive ($B_g > 0$).
- **Safety Radius (S_r)**: Determines how far apart the robots will stay. This parameter must be greater than or equal to 0.12 m ($S_r \geq 0.12$).
- **Projection Distance**: Determines how far ahead of the unicycle model to place the safety "bubble".
- **Magnitude Limit ($|v|$)**: Limits the linear speed of the robot. This parameter must be $|v| \leq V_{max}(0.2\frac{m}{s})$.

The returned function is:

```python
def f(dxu, x):
```

Listing 49: Barrier Certificate for a Unicycle Model Returned Function

The inputs required by this function are:

- **dxu**: A $2 \times N$ numpy array of the unicycle robot velocity commands.
- **x**: A $2 \times N$ numpy array of the robot states.

It returns a $2 \times N$ numpy array of modified velocity commands that ensure safety.

4.5.5 Create a Barrier Certificate for a Unicycle Model with Boundary

```python
def create_unicycle_barrier_certificate_with_boundary(barrier_gain=100, safety_radius=0.12,
projection_distance=0.05, magnitude_limit=0.2, boundary_points=np.array([-1.6, 1.6, -1.0, 1.0])):
```

Listing 50: Barrier Certificate for a Unicycle Model with Boundary Function

This function creates a barrier certificate for a unicycle system with a rectangular boundary included, ensuring that the robots do not leave the arena. It returns another function for optimization reasons. The inputs for this function are:

- **Barrier Gain (B_g)**: Controls how quickly agents can approach each other. A lower value implies a slower approach. This parameter is strictly positive ($B_g > 0$).
- **Safety Radius (S_r)**: Determines how far apart the robots will stay. This parameter must be greater than or equal to 0.12 m ($S_r \geq 0.12$).
- **Projection Distance**: Determines how far ahead of the unicycle model to place the safety "bubble".

- **Magnitude Limit** ($|v|$): Limits the linear speed of the robot. This parameter must be $|v| \leq V_{max}(0.2\frac{m}{s})$.
- **Boundary Points**: This array corresponds to the arena's dimensions and should not be edited.

The returned function is:

```
def f(dxu, x):
```

Listing 51: Barrier Certificate for a Unicycle Model with Boundary Returned Function

The inputs required by this function are:

- **dxu**: A $2 \times N$ numpy array of the unicycle robot velocity commands.
- **x**: A $2 \times N$ numpy array of the robot states.

It returns a $2 \times N$ numpy array of modified velocity commands that ensure safety and boundary compliance.

4.5.6 Create a Barrier Certificate for a Unicycle Model with Dynamic Gains

```
def create_unicycle_barrier_certificate2(barrier_gain=500, unsafe_barrier_gain=1e6,
safety_radius=0.12, projection_distance=0.05, magnitude_limit=0.2):
```

Listing 52: Barrier Certificate for a Unicycle Model with Dynamic Gains Function

This function creates a barrier certificate for a unicycle system. It returns another function for optimization reasons. This function is different from `create_unicycle_barrier_certificate` as it dynamically changes the barrier gain to a large number if the vehicle enters an unsafe region. The inputs for this function are:

- **Barrier Gain** (B_g): Controls how quickly agents can approach each other. A lower value implies a slower approach. This parameter is strictly positive ($B_g > 0$).
- **Unsafe Barrier Gain** (B_{unsafe_g}): Controls how quickly the barrier gain changes when entering an unsafe region. This parameter must be positive ($B_{unsafe_g} > 0$).
- **Safety Radius** (Sr): Determines how far apart the robots will stay. This parameter must be greater than or equal to 0.12 m ($S_r \geq 0.12$).
- **Projection Distance**: Determines how far ahead of the unicycle model to place the safety "bubble".
- **Magnitude Limit** ($|v|$): Limits the linear speed of the robot. This parameter must be $|v| \leq V_{max}(0.2\frac{m}{s})$

The returned function is:

```
def f(dxu, x):
```

Listing 53: Barrier Certificate for a Unicycle Model with Dynamic Gains Returned Function

The inputs required by this function are:

- **dxu**: A $2 \times N$ numpy array of the unicycle robot velocity commands.
- **x**: A $2 \times N$ numpy array of the robot states.

It returns a $2 \times N$ numpy array of modified velocity commands that ensure safety by dynamically adjusting the barrier gain if needed.

4.5.7 Create Unicycle Differential Drive Barrier Certificate

```
def create_unicycle_differential_drive_barrier_certificate(max_num_obstacle_points=100,
max_num_robots=30, disturbance=5, wheel_vel_limit=12.5, base_length=0.105,
wheel_radius=0.016, projection_distance=0.05, barrier_gain=150, safety_radius=0.17):
```

Listing 54: Unicycle Differential Drive Barrier Certificate Function

This function creates a barrier certificate for a unicycle differential drive system that considers errors in the motor dynamics. More information about this barrier certificate function can be found in Emam et al. 2022. It ensures that robots avoid collisions and operate within safe boundaries. The inputs for this function are:

- **Max Number of Obstacle Points**: Maximum number of obstacle points considered in the environment.
- **Max Number of Robots**: Maximum number of robots in the environment.
- **Disturbance**: A disturbance factor affecting the system.
- **Wheel Velocity Limit**: Maximum wheel velocity limit for the differential drive.
- **Base Length**: Distance between the wheels of the differential drive robot.
- **Wheel Radius**: Radius of the wheels.
- **Projection Distance**: Determines how far ahead of the unicycle model to place the safety "bubble".
- **Barrier Gain (B_g)**: Controls how quickly agents can approach each other. A lower value implies a slower approach. This parameter is strictly positive ($B_g > 0$).
- **Safety Radius (S_r)**: Determines how far apart the robots will stay. This parameter must be greater than or equal to 0.17 m ($S_r \geq 0.17$).

The returned function is:

```
def f(dxu, x):
```

Listing 55: Barrier Certificate for a Unicycle Differential Drive Returned Function

The inputs required by this function are:

- **dxu**: A $2 \times N$ numpy array of the unicycle robot velocity commands.
- **x**: A $2 \times N$ numpy array of the robot states.

It returns a $2 \times N$ numpy array of modified velocity commands that ensure safety.

4.5.8 Create Unicycle Differential Drive Barrier Certificate with Boundary

```
def create_unicycle_differential_drive_barrier_certificate_with_boundary(
max_num_obstacle_points=100, max_num_robots=30, disturbance=5, wheel_vel_limit=12.5,
base_length=0.105, wheel_radius=0.016, projection_distance=0.05, barrier_gain=150,
safety_radius=0.17, boundary_points=np.array([-1.6, 1.6, -1.0, 1.0])):
```

Listing 56: Unicycle Differential Drive Barrier Certificate with Boundary Function

This function creates a barrier certificate for a unicycle differential drive system with a rectangular boundary included, ensuring that robots avoid collisions and stay within the defined operational area. More information about this barrier certificate function can be found in Emam et al. 2022. The inputs for this function are:

- **Max Number of Obstacle Points**: Maximum number of obstacle points considered in the environment.
- **Max Number of Robots**: Maximum number of robots in the environment.
- **Disturbance**: A disturbance factor affecting the system.
- **Wheel Velocity Limit**: Maximum wheel velocity limit for the differential drive.
- **Base Length**: Distance between the wheels of the differential drive robot.
- **Wheel Radius**: Radius of the wheels.
- **Projection Distance**: Determines how far ahead of the unicycle model to place the safety "bubble".
- **Barrier Gain (B_g)**: Controls how quickly agents can approach each other. A lower value implies a slower approach. This parameter is strictly positive ($B_g > 0$).
- **Safety Radius (S_r)**: Determines how far apart the robots will stay. This parameter must be greater than or equal to 0.17 m ($S_r \geq 0.17$).
- **Boundary Points**: This array corresponds to the arena's dimensions and should not be edited.

The returned function is:

```
def f(dxu, x):
```

Listing 57: Barrier Certificate for a Unicycle Differential Drive with Boundary Returned Function

The inputs required by this function are:

- **dxu**: A $2 \times N$ numpy array of the unicycle robot velocity commands.
- **x**: A $2 \times N$ numpy array of the robot states.

It returns a $2 \times N$ numpy array of modified velocity commands that ensure safety and boundary compliance.

4.6 Barrier Certificates 2

4.6.1 Create Robust Barriers for Unicycle Differential Drive with Dynamic Gains

```
def create_robust_barriers(max_num_obstacles=100, max_num_robots=30, d=5, wheel_vel_limit=12.5,
base_length=0.105, wheel_radius=0.016, projection_distance=0.05, gamma=150,
safety_radius=0.12):
```

Listing 58: Robust Barriers for Unicycle Differential Drive with Dynamic Gains Function

This function creates robust barriers for a unicycle differential drive system, ensuring that robots avoid collisions and operate within safe boundaries. It dynamically adjusts barrier gains if the vehicle enters an unsafe region. The inputs for this function are:

- **Max Number of Obstacles**: Maximum number of obstacle points considered in the environment.
- **Max Number of Robots**: Maximum number of robots in the environment.
- **Disturbance (d)**: A disturbance factor affecting the system.
- **Wheel Velocity Limit**: Maximum wheel velocity limit for the differential drive.
- **Base Length**: Distance between the wheels of the differential drive robot.
- **Wheel Radius**: Radius of the wheels.
- **Projection Distance**: Determines how far ahead of the unicycle model to place the safety "bubble".
- **Barrier Gain (γ)**: Controls how quickly agents can approach each other. A lower value implies a slower approach. This parameter is strictly positive ($\gamma > 0$).
- **Safety Radius (S_r)**: Determines how far apart the robots will stay. This parameter must be greater than or equal to 0.12 m ($S_r \geq 0.12$).

The returned function is:

```
def robust_barriers(dxu, x, obstacles):
```

Listing 59: Robust Barriers for Unicycle Differential Drive with Dynamic Gains Returned Function

The inputs required by this function are:

- **dxu**: A $2 \times N$ numpy array of the unicycle robot velocity commands.
- **x**: A $2 \times N$ numpy array of the robot states.
- **obstacles**: A $2 \times M$ numpy array of the positions of obstacles.

It returns a $2 \times N$ numpy array of modified velocity commands that ensure safety by dynamically adjusting the barrier gain if needed.

4.7 Miscellaneous

This section provides a set of utility functions designed to support the initialization and validation of robot positions and orientations.

4.7.1 Generate Random Initial Positions for the Robots

```
def generate_initial_conditions(N, spacing=0.3, width=3, height=1.8):
```

Listing 60: Initial Position Function

This function generates random initial conditions in an area of the specified width and height at the required spacing. The inputs for this function are:

- **N**: Number of robots used in the experiment. This parameter must be positive.Specifically, $N = 1, ..., 20$
- **Spacing**: How far apart positions can be. This parameter must be positive.
- **Width**: Width of the area. This parameter must be positive.
- **Height**: Height of the area. This parameter must be positive.

It returns a $3 \times N$ numpy array of the robot states.

```
def at_pose(states, poses, position_error=0.05, rotation_error=0.2):
```

Listing 61: Evaluate Whether Robots are "Close Enough" to Poses Function

In some cases, it is not necessary for robots to reach the exact position commanded. A small margin of error is acceptable, such as 1%. This function checks whether robots are "close enough" to the desired positions and orientations. The inputs for this function are:

- **States**: A $3 \times N$ numpy array of the unicycle states.
- **Poses**: A $3 \times N$ numpy array of the desired states.
- **Position Error**: The tolerable error in the position.
- **Rotation Error**: The tolerable error in the heading.

It returns a $1 \times N$ index array of the agents that are close enough.

4.7.2 Evaluate Whether Robots are "Close Enough" to Desired Position

```
def at_position(states, points, position_error=0.02):
```

Listing 62: Evaluate Whether Robots are "Close Enough" to the Desired Position Function

As before, in some cases, it is not necessary for robots to reach the exact desired position. A small margin of error is acceptable. This is useful when robots are moving through waypoints; if they are close enough, they can proceed to the next waypoint. The inputs for this function are:

- **States**: A $3 \times N$ numpy array of the unicycle states.
- **Points**: A $2 \times N$ numpy array of the desired positions.
- **Position Error**: The tolerable error in the position.

It returns a $1 \times N$ index array of the agents that are close enough.

4.7.3 Marker Size

```
def determine_marker_size(robotarium_instance, marker_size_meters):
```

Listing 63: Marker Size Function

This function determines the appropriate marker size in points so it fits the Robotarium listing window. It calculates the ratio of the robot size to the x-axis and adjusts the marker size accordingly. The inputs for this function are:

- **Robotarium Instance**: The Robotarium object.
- **Marker Size Meters**: The desired size in meters.

It returns the appropriate marker size in points.

4.7.4 Determine Font Size

```
def determine_font_size(robotarium_instance, font_height_meters):
```

Listing 64: Determine Font Size Function

This function determines the appropriate font size in points so that it fits the Robotarium figure window. It calculates the ratio of the desired font height to the y-axis of the figure window and adjusts the font size accordingly. The inputs for this function are:

- **Robotarium Instance**: The Robotarium object.
- **Font Height Meters**: The desired font height in meters.

It returns the appropriate font size in points.

4.8 Graph

This section provides a set of utility functions for generating graph Laplacians for different types of graphs and determining neighbors based on the graph Laplacian or distance.

4.8.1 Generate a Graph Laplacian for a Cycle Graph

```
def cycle_GL(N):
```

Listing 65: Generate a Graph Laplacian for a Cycle Graph

This function generates a graph Laplacian for a cycle graph. A cycle graph is a graph that consists of a single cycle, meaning each node is connected in a closed loop. The function ensures the resulting Laplacian matrix correctly represents the cycle graph structure. The inputs for this function are:

- **N**: Integer, the number of agents in the cycle graph. Must be positive.

It returns an $N \times N$ numpy array representing the graph Laplacian.

4.8.2 Generate a Graph Laplacian for a Line Graph

```
def lineGL(N):
```

Listing 66: Generate a Graph Laplacian for a Line Graph

This function generates a graph Laplacian for a line graph. A line graph is a graph where each node is connected to its predecessor and successor nodes, forming a straight line. The inputs for this function are:

- **N**: Integer, the number of agents in the line graph. Must be positive.

It returns an $N \times N$ numpy array representing the graph Laplacian.

4.8.3 Generate a Graph Laplacian for a Complete Graph

```
def completeGL(N):
```

Listing 67: Generate a Graph Laplacian for a Complete Graph

This function generates a graph Laplacian for a complete graph. A complete graph is a graph where each node is connected to every other node. The inputs for this function are:

- **N**: Integer, the number of agents in the complete graph. Must be positive.

It returns an $N \times N$ numpy array representing the graph Laplacian.

4.8.4 Generate a Laplacian for a Random, Connected Graph

```
def random_connectedGL(v, e):
```

Listing 68: Generate a Laplacian for a Random, Connected Graph

This function generates a Laplacian for a random, connected graph with a specified number of vertices and additional edges. It ensures the graph remains connected.The inputs for this function are:

- **v**: Integer, the number of vertices (nodes) in the graph. Must be positive.
- **e**: Integer, the number of additional edges to add to the graph. Must be non-negative.

It returns a $v \times v$ numpy array representing the graph Laplacian.

4.8.5 Generate a Laplacian for a Random Graph

```
def randomGL(v, e):
```

Listing 69: Generate a Laplacian for a Random Graph

This function generates a Laplacian for a random graph with a specified number of vertices and edges. The inputs for this function are:

- **v**: Integer, the number of vertices (nodes) in the graph. Must be positive.
- **e**: Integer, the number of edges to add to the graph. Must be non-negative.

It returns a $v \times v$ numpy array representing the graph Laplacian.

4.8.6 Determine Topological Neighbors

```
def topological_neighbors(L, agent):
```

Listing 70: Determine Topological Neighbors

This function returns the neighbors of a particular agent using the graph Laplacian. Neighbors are defined as nodes directly connected to the agent. The inputs for this function are:

- **L**: An $N \times N$ numpy array representing the graph Laplacian.
- **agent**: Integer, the agent number (0 to N-1).

It returns a 1xM numpy array with M neighbors.

4.8.7 Determine Delta-Disk Neighbors

```
def delta_disk_neighbors(poses, agent, delta):
```

Listing 71: Determine Delta-Disk Neighbors

This function returns the agents within a specified distance (delta) from a given agent, excluding the agent itself. It uses the 2-norm to determine the distance. The inputs for this function are:

- **poses**: A $3 \times N$ numpy array representing the unicycle states of the robots.
- **agent**: Integer, the agent whose neighbors within a radius will be returned.
- **delta**: Float, the radius of the delta disk considered.

It returns a 1xM numpy array with M neighbors.

5 Acknowledgments

This guide would not have been possible without the contributions of the following individuals[5]

- Abhinandan Krishnan
- Amogh Chinnakonda
- Atharv Marathe
- Avaye Dawadi
- Cason Couch
- Christopher Banks
- Daniel Pickem
- Gennaro Notomista
- Kyle Slovak
- Li Wang
- Magnus Egerstedt
- Mark Mote
- Paul Glotfelter
- Paul Wilson III
- Prajwal Bharadwaj
- Renato Maynard Etchepare
- Sean Wilson
- Siddharth Mayya
- Soobum Kim
- Xiaoyi (Jeremy) Cai
- Yash Srivastava
- Yousef Emam

Thank you to everyone who contributed their time and expertise to make this guide as comprehensive and useful as possible.

6 Conclusion

Congrats! You have successfully completed this guide and should now be closer to running your own full scripts/algorithms on the Robotarium. We are always improving the simulator and Robotarium so feel free to check this guide or the website for any updates! If you have any questions please contact Sean Wilson (Sean.Wilson@gtri.gatech.edu)

[5]The names above are listed in alphabetical order and not according to the level of contribution.

References

Emam, Yousef et al. (2022). “Data-Driven Robust Barrier Functions for Safe, Long-Term Operation”. In: *IEEE Transactions on Robotics* 38.3, pp. 1671–1685. DOI: 10.1109/TRO.2021.3118965.

Kim, Soobum et al. (Sept. 2024a). *GTernal*. URL: https://github.com/robotarium/GTernal.

— (2024b). “GTernal: A robot design for the autonomous operation of a multi-robot research testbed”. In: *International Symposium on Distributed Autonomous Robotic Systems*.

Pickem, Daniel et al. (2017). “The Robotarium: A remotely accessible swarm robotics research testbed”. In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1699–1706. DOI: 10.1109/ICRA.2017.7989200.

Wilson, Sean and Magnus Egerstedt (2023). “The Robotarium: A Remotely-Accessible, Multi-Robot Testbed for Control Research and Education”. In: *IEEE Open Journal of Control Systems* 2, pp. 12–23. DOI: 10.1109/OJCSYS.2022.3231523.

Wilson, Sean, Paul Glotfelter, Siddharth Mayya, et al. (2021). “The Robotarium: Automation of a Remotely Accessible, Multi-Robot Testbed”. In: *IEEE Robotics and Automation Letters* 6.2, pp. 2922–2929. DOI: 10.1109/LRA.2021.3062796.

Wilson, Sean, Paul Glotfelter, Li Wang, et al. (2020). “The Robotarium: Globally Impactful Opportunities, Challenges, and Lessons Learned in Remote-Access, Distributed Control of Multirobot Systems”. In: *IEEE Control Systems Magazine* 40.1, pp. 26–44. DOI: 10.1109/MCS.2019.2949973.

A Libraries Available in the Robotarium

Below is a list of the current libraries that are available in the Robotarium environment:

1. apturl==0.5.2
2. attrs==19.3.0
3. blinker==1.4
4. Brlapi==0.7.0
5. cached-property==1.5.1
6. certifi==2019.11.28
7. chardet==3.0.4
8. Click==7.0
9. colorama==0.4.3
10. command-not-found==0.3
11. contourpy==1.1.0
12. cryptography==2.8
13. cupshelpers==1.0
14. cycler==0.11.0
15. dbus-python==1.2.16
16. defer==1.0.6
17. distro==1.4.0
18. distro-info==0.23+ubuntu1.1
19. dnspython==2.6.1
20. docker==4.1.0
21. docker-compose==1.25.0
22. dockerpty==0.4.1
23. docopt==0.6.2
24. entrypoints==0.3
25. fonttools==4.42.0
26. graphviz==0.20.1
27. httplib2==0.14.0
28. idna==2.8
29. importlib-metadata==1.5.0
30. importlib-resources==6.0.1
31. jsonschema==3.2.0
32. keyring==18.0.1
33. kiwisolver==1.4.4
34. language-selector==0.1
35. launchpadlib==1.10.13
36. lazr.restfulclient==0.14.2
37. lazr.uri==1.0.3
38. louis==3.12.0
39. macaroonbakery==1.3.1
40. matplotlib==3.7.2
41. more-itertools==4.2.0
42. netifaces==0.10.4
43. numpy==1.23.5
44. oauthlib==3.1.0
45. olefile==0.46
46. packaging==23.1
47. paho-mqtt==1.6.1
48. pandas==2.0.3
49. pexpect==4.6.0
50. Pillow==7.0.0
51. protobuf==3.6.1
52. pycairo==1.16.2
53. pycups==1.9.73
54. PyGObject==3.36.0
55. PyJWT==1.7.1
56. pymacaroons==0.13.0
57. pymongo==3.13.0
58. PyNaCl==1.3.0
59. pyocclient==0.6
60. pyparsing==3.0.9
61. pyRFC3339==1.1
62. pyrsistent==0.15.5
63. python-apt==2.0.1+ubuntu0.20.4.1
64. python-dateutil==2.8.2
65. python-debian==0.1.36+ubuntu1.1
66. pytz==2023.3
67. pyxdg==0.26
68. PyYAML==5.3.1
69. reportlab==3.5.34
70. requests==2.22.0
71. requests-unixsocket==0.2.0
72. scipy==1.10.1
73. screen-resolution-extra==0.0.0
74. SecretStorage==2.3.1
75. simplejson==3.16.0
76. six==1.14.0
77. ssh-import-id==5.10
78. systemd-python==234
79. texttable==1.6.2
80. tzdata==2023.3
81. ubuntu-drivers-common==0.0.0
82. ubuntu-pro-client==8001
83. ufw==0.36
84. unattended-upgrades==0.1
85. urllib3==1.25.8
86. vizier==0.0.0
87. wadllib==1.3.3
88. websocket-client==0.53.0
89. xkit==0.0.0
90. zipp==3.16.2