Project 5: Safe & Decentralized Multi-Agent Control[*]

Goals

In this lab, you'll implement control barrier functions over vision data to safely and collaboratively control a system of turtlebots. This task is made challenging by its *decentralized* nature - in the final stage of the project, each turtlebot interacts with the environment purely through computer vision! In this project, you will:

- Design a feedback linearizing tracking controller for a turtlebot using dynamic extension.
- Design a control barrier function $h(q)$ that encodes the safe set of a turtlebot interacting with multiple agents.
- Design and implement a deadlock-resolving control barrier function quadratic program (CBF-QP) controller based around a feedback linearizing turtlebot controller.
- Design and implement a vision-based CBF-QP controller based on your feedback linearizing turtlebot controller.
- Test your vision-based CBF-QP and tracking controller on turtlebot hardware.

The tasks in this project will have some open-ended components, and you'll have the chance to make your own design decisions to solve the problems we give you.

Contents

[*]Developed by Massimiliano de Sa and Han Nguyen, Spring 2023. Modified by Kirthi Kumar, Spring 2024.

1 Simulation

1.1 Feedback Linearizing Controller

Your first task in this project is to develop a feedback linearizing controller for the turtlebot that will allow it to track a trajectory $q_d(t)$. Recall that the dynamics of the turtlebot are described by:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\phi} \end{bmatrix} = \begin{bmatrix} \cos\phi & 0 \\ \sin\phi & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix} \tag{1}$$

Where the state vector of the system is $q = [x, y, \phi]^T$ and the input vector is $u = [v, \omega]^T$, where v is the forward velocity input of the turtlebot and ω is the angular rate of the turtlebot. To design a tracking controller for this turtlebot system, we can use the techniques of *feedback linearization*, which will cancel out the nonlinear dynamics of the turtlebot. To get started in designing a feedback linearizing controller, we notice that this system is control-affine, and may be written in the form:

$$\dot{q} = f(q) + g(q)u \tag{2}$$

Where q, u are the state and input vectors described above and $f(q) = 0$. Our goal in designing a feedback linearizing controller for this system is to find an input u that creates a linear relationship between the input vector, u, and the output of the system, which we define to be the vector:

$$y_{out} = \begin{bmatrix} x \\ y \end{bmatrix} \tag{3}$$

If we can accomplish this, we can easily control the x, y coordinates of the turtlebot.
If we were to try and directly design a feedback linearizing controller for this system, however, we would find that a matrix we would need to invert to cancel out the nonlinear terms in the dynamics would *not* be invertible! To get around this, we apply *dynamic extension*, and rewrite the system in the following equivalent form:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\phi} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} v\cos\phi \\ v\sin\phi \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = f(\tilde{q}) + g(\tilde{q})w \tag{4}$$

Where we *append* v to the state vector of the system to form the dynamically extended state vector $\tilde{q} = [x, y, \phi, v]^T$. In the extended system, instead of controlling the system with our original input, $u = [v, \omega]^T$, we use the new input vector:

$$w = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = \begin{bmatrix} \dot{v} \\ \omega \end{bmatrix} \tag{5}$$

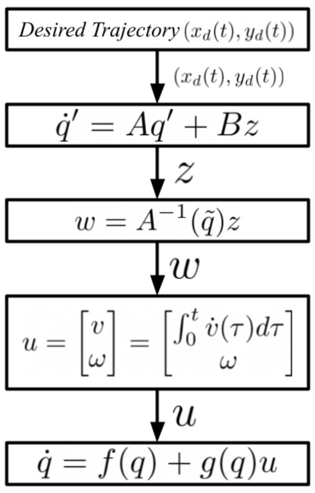
Where we control the *derivative* of input velocity instead of velocity itself. Once we have the system in this extended form, we can prove that the following relationship exists between input and output:

$$\begin{bmatrix} \ddot{x} \\ \ddot{y} \end{bmatrix} = \begin{bmatrix} \cos\phi & -v\sin\phi \\ \sin\phi & v\cos\phi \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = A(\tilde{q})w \tag{6}$$

Here, the matrix $A(\tilde{q})$ is always invertible if $v \neq 0$. By picking $w = A^{-1}(\tilde{q})z$, where $z \in \mathbb{R}^2$, we may derive the following linear input-output relationship.

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \ddot{x} \\ \ddot{y} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} z \tag{7}$$

This is a linear system of the form $\dot{q}' = Aq' + Bz$, where $q' = [x, y, \dot{x}, \dot{y}]^T$. Note that A here is not the same as the $A(\tilde{q})$ matrix discussed above - here, A is the matrix in the linearized system dynamics, whereas $A(\tilde{q})$ is a mapping between w and $\ddot{q}$. We use A in both equations simply as a matter of convention.
Your job is to use this series of relationships to define a feedback linearizing trajectory tracking controller for the turtlebot system. The following flow-chart describes the process your controller design should take:


Desired Trajectory $(x_d(t), y_d(t))$
$(x_d(t), y_d(t))$
$\dot{q}' = Aq' + Bz$
z
$w = A^{-1}(\tilde{q})z$
w
$u = \begin{bmatrix} v \\ \omega \end{bmatrix} = \begin{bmatrix} \int_0^t \dot{v}(\tau)d\tau \\ \omega \end{bmatrix}$
u
$\dot{q} = f(q) + g(q)u$

First, we define the desired $(x_d(t), y_d(t))$ trajectory for our system. Both x_d, y_d are differentiable functions of time that describe where we'd like our turtlebot to be at all times. We send this desired trajectory to the feedback linearized system, $\dot{q} = Aq' + Bz$, which we defined above. We may then use linear control design to pick an input z that allows the system to track the desired trajectory. Note that we will also have access to $q_d(t), \dot{q}_d(t), \ddot{q}_d(t)$ - the desired trajectory and its first two time derivatives - when finding z.
Once we have this value of z, we convert it back into w using $w = A^{-1}(\tilde{q})z$, which came from the feedback linearizing relationship. Once we have w, we may integrate the first component of w, $w_1 = \dot{v}$, in time to find the value of velocity, v, we should send to our original system:

$$u = \begin{bmatrix} v \\ \omega \end{bmatrix} = \begin{bmatrix} \int_0^t \dot{v}(\tau)d\tau \\ \omega \end{bmatrix} = \begin{bmatrix} \int_0^t w_1(\tau)d\tau \\ w_2 \end{bmatrix} \tag{8}$$

Finally, once we have this value for the input, we send it to our original system:

$$\dot{q} = f(q) + g(q)u\dot{q} = \begin{bmatrix} \cos\phi & 0 \\ \sin\phi & 0 \\ 0 & 1 \end{bmatrix} u \tag{9}$$

This will enable our original nonlinear system to track the desired trajectory. Following this procedure, you will design a feedback linearizing tracking controller for the turtlebot.

1.1.1 Goals

Your goals for this section of the project are to implement the following:

1. Design a trajectory tracking controller for the linear system:

$$\dot{q}' = Aq' + Bz \tag{10}$$

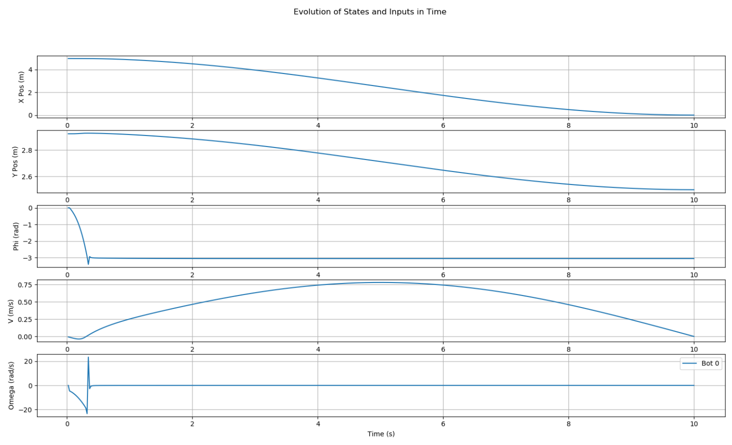
   That picks a value of z that allows the system to track a desired state vector $q_d(t) = [x_d(t), y_d(t), \phi_d(t)]^T$. Note that because the output of the system is $[x, y]^T$, we don't consider $\phi_d(t)$ tracking, and only care about following $[x_d(t), y_d(t)]^T$. The tracking controller design is entirely up to you! You can use simple technique such as hand-tuned state feedback, or something more advanced such as LQR or CLF tracking control, which might provide more optimal tracking performance. In addition to the desired trajectory q_d, you will have access to the first and second time derivatives, $\dot{q}_d, \ddot{q}_d$ of the desired trajectory.

2. Implement the feedback linearizing structure defined above to transform your input z to the linear system $\dot{q}' = Aq' + Bz$ into an input u to the original turtlebot system. Note that to achieve this, you'll likely need to use a numerical integral such as an Euler integral to perform your calculations correctly! How you implement this is entirely your decision.

1.1.2 Getting Started

To implement the feedback linearizing controller in simulation, go to the file **controller.py** in the folder **proj5_sim**. In **controller.py**, you'll find a class called **TurtlebotFBLin**. You should implement your feedback linearizing controller

in this class using the provided functions.
Once you're ready to test your feedback linearizing controller, run the file **run_fb_lin.py**, which is also contained in the **proj5_sim** folder. This file will test your feedback linearizing controller on a single turtlebot with a simple point to point trajectory. If your controller is successful, the turtlebot should trace out the following trajectory, state, and input plots:



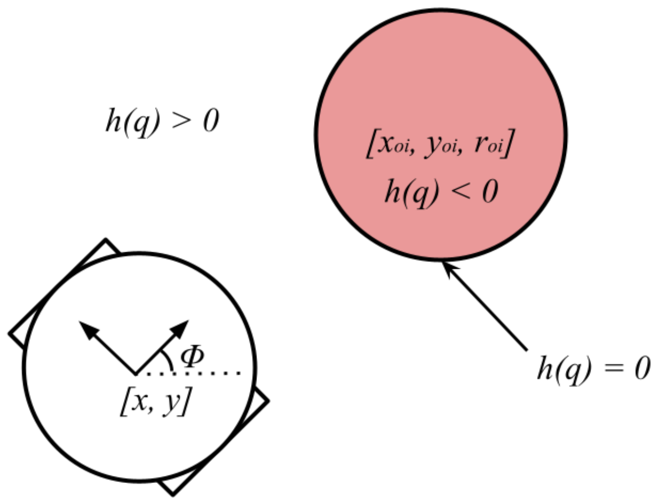
Above: Tracking plots for feedback linearization

1.2 Control Barrier Functions

Control barrier functions are a special class of function that encode the safety of a system. Typically denoted by $h(x)$, where x is the state vector of the system, control barrier functions have the following properties:

1. If the system is in a safe state x, the barrier function $h(x) > 0$.
2. If the system is on the boundary between the safe and unsafe sets, the barrier function $h(x) = 0$.
3. If the system is in an unsafe state x, the barrier function $h(x) < 0$.

Through this criteria, control barrier functions help us identify whether our robot is in a safe location or not. It's extremely important to note that in general, barrier functions should be continuous and differentiable functions of the state vector x - we cannot use a discontinuous "indicator" function as our barrier function, for instance.
In this project, we'll consider the safety of the following environment, where a set of turtlebots move around in space. From the perspective of each individual turtlebot, all of the other turtlebots in the environment may be modeled as *circular obstacles*. Each obstacle turtlebot O_i has a center described by coordinates (x_{oi}, y_{oi}) (which will change with respect to time) and a radius r_t.



Assuming that our turtlebot also has a radius r_t, we may find a barrier function $h(q)$ that encodes the safety of our turtlebot with respect to an arbitrary obstacle turtlebot O_i.
This barrier function should be defined such that $h(q) > 0$ when our turtlebot is not in collision with the obstacle turtlebot O_i, and is less than zero when our turtlebot *is* in collision with the obstacle turtlebot.
In this section, your job is to find the barrier function $h(q)$ that defines safety of our turtlebot with respect to a single obstacle turtlebot. If you wish, you can consider adding a buffer to your barrier function such that our turtlebot remains a certain threshold away from each obstacle turtlebot. Note that there are many right answers to this portion of the project! You're encouraged to keep your barrier function as simple as possible.

1.2.1 Goals

The goals for this section are as follows:

1. Come up with a barrier function $h(q)$ that encodes the safety of a turtlebot in the presence of obstacle turtlebots. Your barrier function should be a smooth function of the state vector q of the turtlebot and of any other environmental or physical parameters you deem necessary.

Note that there is nothing you have to implement in code for this section - simply design your barrier function $h(q)$.

1.3 CBF-QP Deadlock Controller

1.3.1 The Basic CBF-QP Controller

Let's apply the control barrier function $h(q)$ we found to the problem of collaborative and safe locomotion for a system of turtlebots. Suppose we have a system of n circular turtlebots, each of which have a desired trajectory $(x_{di}(t), y_{di}(t))$ that they would like to track.
How can we allow each turtlebot to track its trajectory while ensuring no turtlebot collides with any other? We may use a special type of feedback controller known as a control barrier function quadratic program (CBF-QP). This is a type of controller that provides guarantees on the safety of a system based on a control barrier function in a minimally invasive manner.
Let's begin by discussing the CBF-QP controller for a system with a single barrier function $h(q)$. Suppose that the system may use an input $k(q)$ to track its desired trajectory without considering safety. We may find a *safe* input to the system that's as close as possible to the trajectory tracking input by solving the optimization problem:

$$u^*_{safe} = \arg\min_{u \in U} ||u - k(q)||^2 \tag{11}$$

$$s.t.\ \dot{h}(q) \geq -\gamma h(q), \tag{12}$$

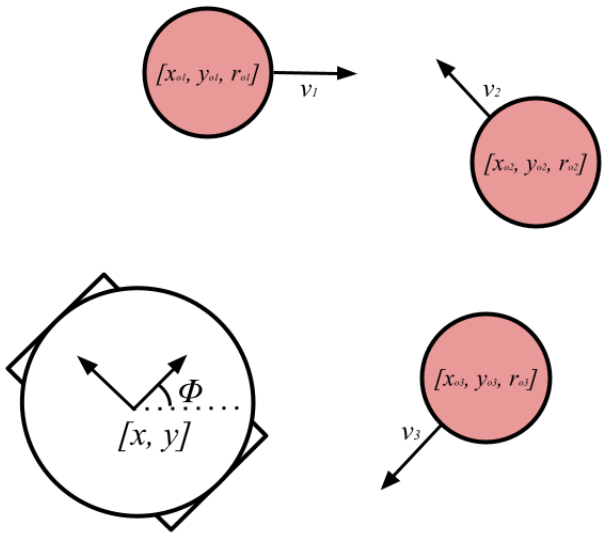
$$\dot{q} = f(q) + g(q)u, \tag{13}$$

Where $U \subseteq \mathbb{R}^m$ is the set of allowable inputs to the system, $\gamma > 0 \in \mathbb{R}$ is a positive constant, and $\dot{h}(q)$ is the derivative of h along the trajectories of the system.[1] If the optimization constraint $\dot{h}(q) \geq -\alpha h(q)$ is enforced at all instants of time, we can force the barrier function $h(q)$ to be greater than zero, which will therefore keep the system within its safe set.
By minimizing $||u - k(q)||^2$ while enforcing the optimization constraint that allows $h(q) > 0$, we find the input u that is as close as possible to our original input $k(q)$ that allows us to track our desired trajectory while ensuring the system stays within its safe set. For our complex system of n turtlebots, this simple formulation on the CBF-QP won't be sufficient. Let's discuss how we can modify this controller to suit our needs.

1.3.2 CBF-QP Deadlock Controller for n Turtlebots

There are a few major modifications we'll need to make to the simple CBF-QP defined above for use on our multi-agent systems. Let's focus in on a single turtlebot in this system, turtlebot 1, and think about how we can design a safe controller for this turtlebot. In this project, we'll refer to this turtlebot (the turtlebot we wish to control), as the *ego turtlebot*, and the others as the obstacle turtlebots.

[1] Note that this particular formulation of a CBF-QP is called an exponential CBF-QP.



Modification 1: $n - 1$ Barrier Functions:
In this section of the project, we'll assume that each turtlebot has access to the positions, velocities, and dimensions of all of the other turtlebots at every instant in time. If we know this information, we define a set of $n - 1$ barrier functions for the ego turtlebot that together tell us about the total safe set of the turtlebot.

$$\{h_2(q), ..., h_n(q)\} \tag{14}$$

Where h_i refers to the barrier function between turtlebot i and our ego turtlebot. This gives us our first modification of our CBF-QP. Instead of applying one barrier constraint, we will apply $n-1$ barrier constraints - one for each barrier function - to ensure the safety of the system.

Modification 2: Nested Structure
We may prove that if we were to apply the input constraint $\dot{h}(q) \geq -\gamma h(q)$ to a turtlebot, the CBF-QP will only *slow the turtlebot down* when it encounters an unsafe region - the CBF-QP will *not* in this case allow us to steer around obstacles! If we didn't make this change, pairs turtlebots would get stuck in face-off positions called *deadlocks*, where they wouldn't be able to figure out how to move around each other in a safe manner.
To enable the turtlebot to slow down and *steer around* obstacles, we must change the structure of the controller. Instead of applying the CBF-QP around the input generated by the feedback linearizing tracking controller, we'll apply the CBF-QP around the *innermost* layer in feedback linearization, where our system is represented by the dynamics:

$$\dot{q}' = Aq' + Bz \tag{15}$$

However, if we take the derivative of the CBF along the trajectories of the linear system $\dot{q}' = Aq' + Bz$, we'll find that no input shows up! We have to take a *second* derivative of the CBF-QP to get an input term to appear. This then gives us the following second order CBF constraint around our linear system:

$$\ddot{h}_i(q') + k_1\dot{h}_i(q') + k_2 h_i(q') \geq 0 \tag{16}$$

Where $k_1, k_2 \in \mathbb{R}$ are constants that must be selected such that $h_i(q') > 0$ and $\ddot{h}_i$ is taken along the trajectories of $\dot{q}' = Aq' + Bz$. Remember that when taking the derivatives of your barrier function, you should also account for the velocity and acceleration of the obstacle turtlebots!
Once we solve the CBF-QP around this constraint, we will have a safe input z_{safe} to the linear system. We then convert this into w_{safe} and finally u_{safe} using the same process as feedback linearization. This gives us the process:

$$z_{track} \rightarrow z_{safe} \rightarrow w_{safe} \rightarrow u_{safe} \tag{17}$$

Where z_{track} is the trajectory tracking input sent to the linear system and u_{safe} is the safe input vector that gets sent to the turtlebot.

Modification 3: Weighted Cost Function
To encourage our turtlebot to steer around obstacles rather than slowing down and getting stuck in a "face-off"

deadlock situation, we'll find it helpful to change our cost function. Instead of using the standard CBF-QP cost function:

$$||z - k(z)||^2 \tag{18}$$

Where z is our input to the linear system and $k(z)$ is the nominal tracking z input, we'll use the cost function:

$$(z - k(z))^T Q(z - k(z)) \tag{19}$$

Where $Q \succeq 0$ is a positive semidefinite matrix. This is a *weighted* version of our original cost function. By changing the eigenvalues of Q, we can weight which terms in our input vector z have more cost, and therefore encourage our system to use more steering input instead of just slowing down.

Summary: CBF-QP Deadlock Controller
Let's summarize the overall control structure. Firstly, we calculate a safe input z_{safe} to the linear system $\dot{q}' = Aq' + Bz$ using the deadlock-resolving CBF-QP controller:

$$z_{safe} = \arg\min_{z \in \mathbb{R}^2} \; (z - k(z))^T Q(z - k(z)) \tag{20}$$

$$s.t. \; \ddot{h}_i(q') + k_1 \dot{h}_i(q') + k_2 h_i(q') \geq 0, \; i = 2, 3, ..., n, \tag{21}$$

$$\dot{q}' = Aq' + Bz, \tag{22}$$

Where k_1, k_2 *must* be tuned to ensure $h(t) > 0$ and $\ddot{h}$ is taken along the trajectories of the linear system $\dot{q}' = Aq' + Bz$. Note that $\dot{h}$ *does not* depend on z - this should *only* depend on the current velocities of the system! Once we have z_{safe}, we convert it to a w input using our feedback linearizing method:

$$z = \begin{bmatrix} \cos\phi & -v\sin\phi \\ \sin\phi & v\cos\phi \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = A(\tilde{q})w \tag{23}$$

Assuming $v \neq 0$, once we have w_{safe} from $w = A^{-1}(\tilde{q})z$, we convert w_{safe} to our actual input to the turtlebot, u_{safe}, by using the relationship:

$$u = \begin{bmatrix} v \\ \omega \end{bmatrix} = \begin{bmatrix} \int_0^t \dot{v}(\tau)d\tau \\ \omega \end{bmatrix} = \begin{bmatrix} \int_0^t w_1(\tau)d\tau \\ w_2 \end{bmatrix} \tag{24}$$

This will *finally* give us a safe input u_{safe} to our turtlebot that allows us to *steer around* obstacles.

1.3.3 Goals

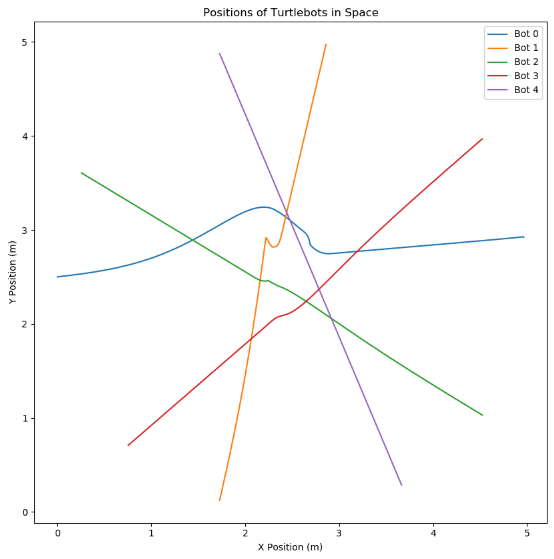
The goals for this section of the project are as follows:

1. Implement the CBF-QP deadlock controller defined above in simulation. You should implement your CBF-QP controller using the CasADi optimization library. If you'd like a refresher on CasADi, check out the tutorial notebook here. This library will allow you to solve the optimization problem in the CBF-QP.

2. Tune the values of k_1, k_2 such that your system is able to reliably avoid obstacles and face-off deadlock scenarios where pairs of turtlebots get stuck head to head. To tune k_1, k_2 such that $\ddot{h} + k_1\dot{h} + k_2 h \geq 0$ guarantees that $h(t) > 0$, you'll have to figure out what conditions on k_1 and k_2 give $h(t) > 0$. *Hint: Try solving the differential equation* $\ddot{h} + k_1\dot{h} + k_2 h = 0$. *For what* k_1, k_2 *is the solution to this ODE always* > 0*?*

1.3.4 Getting Started

To implement this portion of the project, go to the file **lyapunov_barrier.py**, located in the **proj5_sim** folder. In this file, you should implement the class **TurtlebotBarrierDeadlock** with your barrier function and derivatives. This class defines a barrier function object for an ego turtlebot with respect to a single obstacle turtlebot.
Once you have implemented this class, go to the file **controller.py** and implement the class **TurtlebotCBFQPDeadlock**. In this class, you'll implement the CBF-QP controller discussed above. Notice that this class stores an instance of your nominal feedback linearizing controller as a parameter.
Once these two classes have been implemented, you may test your controller by running the file **run_cbf_qp_dlock.py**, which will test your controller on a system of five turtlebots that experience safety conflicts. If your controller is successful, each turtlebot should be able to avoid all other turtlebots while reaching its goal state. Your solution should produce a plot similar to the following:

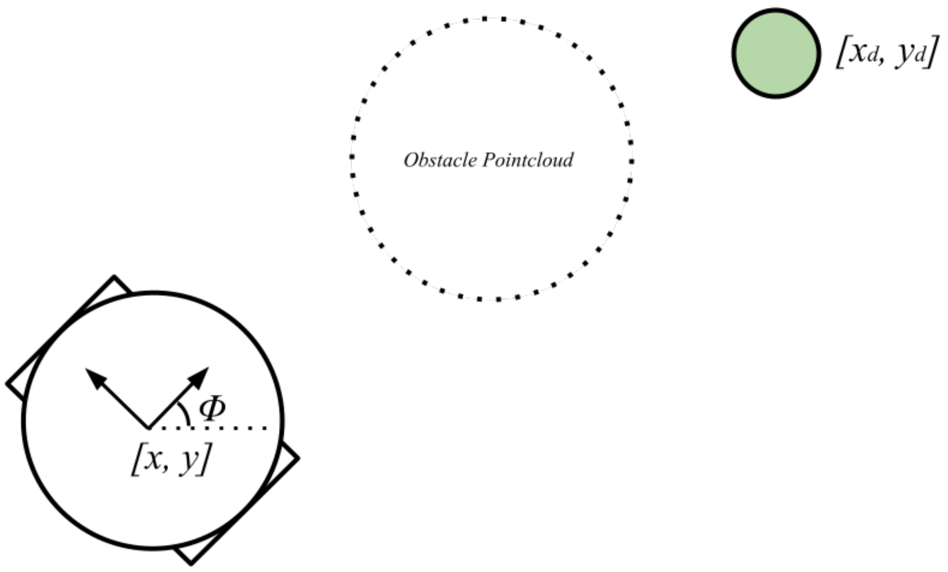


Above: A multi-agent simulation with deadlock resolution.

In the animation, you should see the turtlebots steer around one another while reaching their goal states. Each goal state is located directly across the circle from where the turtlebot started.

1.4 Vision-Based CBF-QP Controller

Let's now remove the assumption that we know the states of other robots in the system. Now, each turtlebot will only be able to interact with its environment through its LIDAR sensor, which provides a pointcloud of the turtlebot's environment with respect to the turtlebot frame, which has been draw in the image below.



Now, instead of being able to define the safe set of the ego turtlebot perfectly using the states of each obstacle turtlebot, we must use vision data to *approximate* the safe set of the turtlebot.
Your first step in this project is to come up with a *single* barrier function $h(q)$ that encodes the safety of the ego turtlebot based on the pointcloud of discrete (x, y, z) points in the environment. Note that you may assume that the z coordinates of the ego and obstacle turtlebots are all zero.
Try thinking about a few of the different ways of approaching this problem, and find a method that you believe will give a strong balance between performance efficiency and accuracy of describing the safe set. Since you're working with incomplete sensor data, it's alright to make assumptions and approximations as long as they're justifiable!
The following two ideas are two possibilities for identifying a single barrier function over the pointcloud data.

1. Define a barrier function based on the distance to the closest point within the pointcloud.

2. Applying clustering methods to identify turtlebots in the pointcloud. Apply a barrier function over the closest cluster.

These are by no means the only method of performing this task, but might be good starting points! If one method isn't working out, don't be afraid to experiment with something new or ask the staff for suggestions! Remember - the method you devise will eventually have to run in real time on hardware, so ensure it isn't too computationally intensive!
Once you have defined your barrier function or set of barrier functions, you will adapt your CBF-QP controller from your CBF-QP deadlock controller to find the best input to the system based on pure vision data.

1.4.1 Goals

The goals for this section of the project are as follows:

1. Develop a barrier function or a set of barrier functions that encode the safe set of the system based on the vision data of each turtlebot.

2. Write a CBF-QP controller that allows the system to safely and collaboratively operate using just vision data. Note that in this section of the project, it's alright if some of the turtlebots get stuck in face-off deadlock situations while trying to reach their goal - you'll be working with imperfect vision data, so this is a possibility.

1.4.2 Getting Started

First, go to the file **lyapunov_barrier.py**, and implement the **TurtlebotBarrierVision** class using your pointcloud-based barrier function. Objects of this class implement the logic of vision-based barrier functions. Pointcloud data will be passed into this class for you. Remember that when pointlcoud data is passed in, it is in the *ego turtlebot frame*, not the spatial frame! Before using the pointcloud, you'll have to transform it to the spatial frame. It might be helpful to have some helper functions to compute the rotation matrix of the turtlebot with respect to the spatial frame as a function of ϕ and to transform the pointcloud.
After implementing this class, go to **controller.py** and implement the class **TurtlebotCBFQPVision**. Your CBF-QP should have an identical structure to the CBF-QP in the previous section - you might have to re-tune some constants. This class is automatically passed TurtlebotBarrierVision objects - the vision to controller pipeline is implemented for you.
Once you've implemented your controller, you may test it by running the file **run_cbf_qp_vision.py**. Tune your controller until there are no collisions remaining in your system. For this portion of the project, it is fine if some of your turtlebots experience deadlock.

2 Hardware

Once you've finished the simulation portion of this project, you may move onto the hardware portion, where you'll implement a simple version of the CBF-QP on a real turtlebot. Due to sensing limitations of the turtlebot, we won't implement the full steering-based version of the vision-based CBF-QP, but will rather implement a braking CBF-QP based on vision data.

2.1 Braking CBF-QP

We'll implement the simple relative degree one barrier function directly over the feedback linearizing input. To find the safe input to the system, we'll solve the optimization problem:

$$u^*_{safe} = \arg\min_{u \in U} ||u - k(q)||^2 \tag{25}$$

$$s.t.\ \dot{h}(q) \geq -\gamma h(q), \tag{26}$$

$$\dot{q} = f(q) + g(q)u, \tag{27}$$

Where $h(q)$ is your vision-based barrier function you defined in the previous section, $k(q)$ is the input to the turtlebot from the feedback linearizing controller, and u_{safe} is the safe input we send to the turtlebot. Note that in this case, we *do not* apply the CBF-QP over the z input to the system, rather just to the final input, u.

This means that when we take $\dot{h}(q)$ along the trajectories of the system, we should *not* compute $\dot{h}(q)$ along the trajectories of $\dot{q}' = Aq' + Bz$, as we did before, but rather along the trajectories of the actual turtlebot dynamics:

$$\dot{q} = \begin{bmatrix} \cos\phi & 0 \\ \sin\phi & 0 \\ 0 & 1 \end{bmatrix} u \tag{28}$$

Where $u = [v, \omega]^T$, our input vector to the turtlebot. This form of the CBF-QP will apply a *braking constraint* to the turtlebot based on the vision-based barrier function $h(q)$. This means that as opposed to the previous CBFs, it will *not* steer the system around obstacles, but rather slow the system down around obstacles.

2.1.1 Goals

Your goals for this portion of the project are the following:

1. Implement the feedback linearizing controller on turtlebot hardware.
2. Test your feedback linearizing controller by running a trajectory.
3. Implement the simple braking CBF-QP on turtlebot hardware.
4. Test your CBF-QP by running a trajectory and blocking the path of the robot with an obstacle.

You should record videos of your tracking controller and CBF-QP working as desired (specified below).

2.1.2 Getting Started

To implement this portion of the project, go to the file **lyapunov_barrier.py** within the **src** folder in **proj5_pkg**. Using your vision-based barrier function $h(q)$ from above, implement the **TurtlebotBarrierVision** class. Recall that now, you only need to return $h, \dot{h}$, and that $\dot{h}$ should be taken along the trajectories of the nonlinear system.
Once you've implemented this class, go to **controller.py** within the **src** folder in **proj5_pkg** and implement the **TurtlebotFBLin** class - this should be almost identical to your class from the simulation. After you've implemented TurtlebotFBLin, implement the class **TurtlebotCBFQP** using the simple braking CBF-QP discussed above.
To run your code, go to the file **main.py** within the **src** folder in **proj5_pkg**. Near line 57, you'll find the option to select your controller. First, select the feedback linearization controller, and run the code on the turtlebot to ensure that the turtlebot correctly tracks the provided trajectory.
Once the tracking controller has been verified, select the CBF-QP controller in **main.py** and run the code on the turtlebot. If you stand in the path of the turtlebot, the turtlebot should brake automatically using the CBF-QP. Further, if you move slowly towards the turtlebot, the turtlebot should move backwards to avoid you.

3 Deliverables

3.1 Tasks

In this project, you will implement each of the sections described above in code. You should begin by working on the controllers in simulation. First, develop and test your feedback linearizing controller. Once you've verified that this controller works, you can move on to developing the deadlock CBF-QP, and from there a the vision-based CBF-QP. The simulation deliverables are fully optional – but a majority of the code is well-outlined for you in simulation. Your main deliverable is a video demonstrated your code implemented on the turtlebot hardware. The specific deliverables for each section are:

1. *Feedback Linearizing Controller*: Video demonstrating tracking control on hardware.
2. *Braking CBF-QP*: Video demonstrating CBF-QP braking on hardware (obstruct the path of the turtlebot and show that it stops).

It may not work perfectly, but that's okay!

4 Getting Started

4.1 Github

To find the starter code for this project, go to the GitHub classrooms invite page. You should now be asked to create a team or join from a list of existing teams. If one of your teammates has already created a team, you should join that team instead of creating a new one. After you have joined a team, you will not be able to switch teams by yourself. If you make a mistake or something else comes up that requires you to switch teams, let us know.

4.2 Submission

The only thing required for this project is a video submission. In the video, please film all group members saying their names first, and then the functionality of the robot. It would be nice if someone in the group narrates what's being run (as if you were explaining it to a TA). You should first run the feedback linearizing controller, and then the braking CBF-QP. The video can be uploaded to Youtube or Google Drive, just make sure that we have permissions to view it. The video should be less than 5 minutes long.

References

[1] Aaron D Ames, Samuel Coogan, Magnus Egerstedt, Gennaro Notomista, Koushil Sreenath, and Paulo Tabuada. Control barrier functions: Theory and applications. pages 3420–3431, 2019.

[2] Richard M Murray, Zexiang Li, and S Shankar Sastry. *A mathematical introduction to robotic manipulation.* CRC press, 1994.

[3] Shankar Sastry. *Nonlinear systems: analysis, stability, and control*, volume 10. Springer Science & Business Media, 2013.

[4] Li Wang, Aaron D. Ames, and Magnus Egerstedt. Safety barrier certificates for collisions-free multirobot systems. *IEEE Transactions on Robotics*, 33(3):661–674, 2017.

[5] Guofan Wu and Koushil Sreenath. Safety-Critical Control of a 3D Quadrotor With Range-Limited Sensing. 10 2016. V001T05A006.