RefineCBF Code Documentation

Table of Contents

Introduction

This document is to serve as a more detailed user guide for the intricacies of the refineCBF ROS2 package. Each node of the ROS framework, and instructions on how to set up your own experiment will be discussed.

General Preliminary Comments

The package is wrapped around modified ROBOTIS Turtlebot3 standard libraries and Dynamixel SDK libraries. As such, if other Turtlebot3 (TB3) packages are sourced at the same time in your environment, there may be erroneous behavior.

Nodes

Dynamic Programming Node

Description

Starting with a prior CBF, refines the CBF using Hamilton-Jacobi Reachability (which is based on dynamic programming). Upon finishing an iteration of refinement, the new CBF is saved locally to the file cbf.npy and the published topic /cbf_availabilty, will have its message changed to True once. Immediately after sending, the message value will be changed back to False.

Running the node independently:

```
cd ~/refinecbf_ws && ros2 run refine_cbf dynamic_programming
```

Topics Table

Published Topics	Published Topic Message Type	Subscribed Topics	Subscribed Topic Message Type
/cbf_availability	std_msgs/Bool	None	None

Nominal Policy Node

Description

Computes the nominal policy (i.e. the safety-agnostic) goal-seeking control action using the last known subscribed state from the Odometry topic. Uses a controller as defined in the high_level_controller.py file. The default controller for this node is a precomputed optimal control-based nominal policy lookup table. Once computed, the nominal policy is published over a Twist message.

Running the node independently:

```
cd ~/refinecbf_ws && ros2 run refine_cbf nominal_policy
```

Topics Table

Published Topics	Published Topic Message Type	Subscribed Topics	Subscribed Topic Message Type
/nom_policy	geometry_msgs/Twist	/<odom_topic>	nav_msgs/Odometry

Safety Filter Node

Description

Subscribes to the current state, nominal policy, and CBF availability and uses this information to compute and publish the optimal safety-preserving control policy over a Twist topic name that is recognized by the Turtlebot3. If the CBF availability is True, then the node updates the current CBF using a locally stored CBF value.

The current CBF is used to filter the safety-agnostic policy by solving a type of convex optimization problem called a CBF quadratic program (CBF-QP). The solver of choice is provided by the CVXPY library.

Additionally, the node publishes the current safety value (h(x)), which is not used by other nodes, but serves as feedback in rqt for the user.

Running the node independently:

```
cd ~/refinecbf_ws && ros2 run refine_cbf safety_filter
```

Published Topics	Published Topic Message Type	Subscribed Topics	Subscribed Topic Message Type
/cmd_vel	geometry_msgs/Twist	/<odom_topic>	nav_msgs/Odometry
/safety_value	std_msgs/Float32	/cbf_availability	std_msgs/Bool
		/nom_policy	geometry_msgs/Twist

RefineCBF Visualization Node

Description

This node handles most of the visualization done in RVIZ. It publishes topics which depict the obstacles, current safe set (0-levelset of current CBF), the initial safe set (0-levelset of initial CBF), and the goal set (the circle defined by the goal coordinate + padding). These topics are Path messages generated from the extracted vertices of the contours of these sets using Matplotlib.

Importantly, only 5 disjoint obstacles and safe sets contours can be depicted at once with the default node. For instance if there were 6 circular obstacles, only 5 would be depicted. If the environment configuration would result in 5 or more disjoint safe sets, this would also occur. In the thesis example provide, only 2 disjoint obstacles can be observed, and ~2 disjoint safe set contours will be depicted at all times. To exceed this limit the user will need to alter the refine_cbf_visualization.py ros node file by adding more publishers. Additionally, the RVIZ file will need to be altered to be able to hear these new topics.

Running the node independently:

```
cd ~/refinecbf_ws && ros2 run refine_cbf refine_cbf_visualization
```

Topics Table

Published Topics	Published Topic Message Type	Subscribed Topics	Subscribed Topic Message Type
/obstacle_1	nav_msgs/Path	/cbf_availability	std_msgs/Bool

Published Topics	Published Topic Message Type	Subscribed Topics	Subscribed Topic Message Type
/obstacle_2	nav_msgs/Path	/<odom_topic>	nav_msgs/Odometry
/obstacle_3	nav_msgs/Path		
/obstacle_4	nav_msgs/Path		
/obstacle_5	nav_msgs/Path		
/safe_set_1	nav_msgs/Path		
/safe_set_2	nav_msgs/Path		
/safe_set_3	nav_msgs/Path		
/safe_set_4	nav_msgs/Path		
/safe_set_5	nav_msgs/Path		
/initial_safe_set	nav_msgs/Path		
/goal_set	nav_msgs/Path		

Nominal Policy Node

Description

Computes the nominal policy (i.e. the safety-agnostic) goal-seeking control action using the last known subscribed state from the Odometry topic. Uses a controller as defined in the high_level_controller.py file. The default controller for this node is a precomputed optimal control-based nominal policy lookup table. Once computed, the nominal policy is published over a Twist message.

Running the node independently:

```
cd ~/refinecbf_ws && ros2 run refine_cbf nominal_policy
```

Topics Table

Published Topics	Published Topic Message Type	Subscribed Topics	Subscribed Topic Message Type
/nom_policy	geometry_msgs/Twist	/<odom_topic>	nav_msgs/Odometry

Transform Stamped to Odometry Conversion Node

This node translates state information from a TransformStamped message type to a Odometry message type. This won't run need to be run in most cases, but if using something like the UCSD aerodrome Vicon system - which outputs its state information in TransformStamped form - it will be necessary to run this node.

Running the node independently:

```
cd ~/refinecbf_ws && ros2 run refine_cbf refine_cbf_visualization
```

Topics Table

Published Topics	Published Topic Message Type	Subscribed Topics	Subscribed Topic Message Type
/vicon/odom	nav_msgs/Odometry	/<state_topic>	geometry_msgs/TransformStamped

Configuration File

To adjust parameters for the experiment, the user can alter some of the global variables found in config.py.

Global variables are denoted by being in all capital letters. For example, `TIME_STEP` is a global variable which is used in the Dynamic Programming node that defines the dynamic programming time step.

Comments within the file describe what each of the global variables define, but those of exceptional note will be covered here.

Initial State

`INITIAL_STATE`: 1x3 numpy array which describes the x, y, theta initial pose of the robot

State Feedback Topic

Because the state feedback can come from different topics, it will be necessary to redefine what topic the nodes are listening for:

`STATE_FEEDBACK_TOPIC`: string that is the topic name state feedback information is provided over

Grid

1. `GRID_RESOLUTION`: how discretized each of the state dimensions should be. A tuple of size 3 where it's (x, y, theta)
2. `GRID_LOWER`: a 1x3 numpy array that is the lower bound of the state domain [x, y, theta]
3. `GRID_UPPER`: a 1x3 numpy array that is the upper bound of the state domain [x, y, theta]

Control Bounds

1. `U_MIN`: a 1x2 numpy array which determines the lower bound of the linear and angular velocities
2. `U_MAX`: a 1x2 numpy array which determines the upper bound of the linear and angular velocities

Initial CBF

The initial default CBF takes the form of a circular paraboloid. The configuration file contains parameters which allow the user to alter the shape of this kind of function.

1. `RADIUS_CBF` : changes the radius of the 0-levelset
2. `CENTER_CBF`: a 1x2 numpy array that defines the x and y position of the center of the circle defined by the 0-levelset
3. `CBF_SCALAR`: a linear multiplier that adjust how fast h(x) increases or decreases. Shouldn't ever really need to be changed.

If the user wishes to alter the initial CBF to a different type of function beyond the circular paraboloid, it is necessary to redefine the `DiffDriveCBF` class within utils.py. The global parameters defined above will no longer apply to the new defined CBF and as such will need to be altered.

```
# Define a class called DiffDriveCBF.
class DiffDriveCBF(ControlAffineCBF):
    """
    Class representing the control barrier function for the differential drive robot.
    Inherits from the ControlAffineCBF class.
    """

    def __init__(self, dynamics: DiffDriveDynamics, params: dict = dict(), **kwargs) -> None:
        """
        Constructor method.

        Args:
            dynamics (DiffDriveDynamics): Dynamics of the differential drive robot.
            params (dict, optional): Dictionary containing parameters. Defaults to an empty dictionary.
            **kwargs: Variable number of keyword arguments.
        """
        self.center = params["center"]  # Center of the circle defined by 0-superlevel set of h(x)
        self.r = params["r"]            # Radius of the circle defined by 0-superlevel set of h(x)
        self.scalar = params["scalar"]  # Scalar multiplier of h(x)

        super().__init__(dynamics, params, **kwargs)

    def vf(self, state, time=0.0):
        """
        Value function (h(x)) method.

        Args:
            state (numpy.array): Array representing the state.
            time (float, optional): Time value. Defaults to 0.0.

        Returns:
            jnp.array: JAX NumPy array representing the value function.
        """
        return self.scalar * (self.r ** 2 - (state[..., 0] - self.center[0]) ** 2 - (state[..., 1] - self.center[1]) ** 2)

    def _grad_vf(self, state, time=0.0):
        """
        Gradient of the value function (del_h(x)) method.

        Args:
            state (numpy.array): Array representing the state.
            time (float, optional): Time value. Defaults to 0.0.

        Returns:
            jnp.array: JAX NumPy array representing the gradient of the value function.
        """
        dvf_dx = np.zeros_like(state)
        dvf_dx[..., 0] = -2 * (state[..., 0] - self.center[0])
        dvf_dx[..., 1] = -2 * (state[..., 1] - self.center[1])
        return self.scalar * dvf_dx
```

Gamma

`GAMMA` controls the rate at which the current safety value (h(x)) decreases. In other words, how fast the robot is allowed to approach obstacles. A higher gamma, will increase the rate at which the robot decreases in safety while a lower rate will decrease this rate. A gamma of 0 will prevent the current safety value from ever decreasing, thus being extremely conservative.

Generating A New Nominal Policy Table (Optional)

If the user wishes to use the same type of nominal policy as the example (reachability-based optimal controller) for a different environment, the policy_generator.py script can be used. There are a few variables that can be changed for a new experimental setup:

1. Goal Coordinates
2. Goal Padding (i.e. radius and allowed orientation)

3. Grid Lower Bound
4. Grid Upper Bound
5. Grid Resolution
6. Minimum linear velocity & angular velocity
7. Maximum linear velocity & angular velocity

There are also some more advanced parameters that can be changed. These shouldn't need to be changed unless the new grid size is very large:

1. Time horizon
2. Value Function time step
3. Value Function time index

The grid size and resolution **must** be the same as those defined in config.py.

Making Your Own Experiment Environment in Gazebo

To create your own Gazebo environment, the user can alter files within the turtlebot3_simulations sub-package.

Making or Modifying a World

If making a new world, the launch file must be altered (or a new one made) to point to the world file. This is found here:

src/turtlebot3_simulations/turtlebot3_gazebo/worlds/refine_cbf_experiment/grid_size_2x2/

The orignal, world file is burger.model.

If a new world file is made, the user will need to point to it in the launch file, found at:

src/turtlebot3_simulations/turtlebot3_gazebo/launch/refine_cbf_experiment.launch.py

Alternatively, a different launch file can be created.

Making or Modifying Models

To add or alter models, the user can refer to the Gazebo models folder for the package

src/turtlebot3_simulations/turtlebot3_gazebo/models/

The current provided models are:

1. A Turtlebot3 Burger of dimensions equivalent to the real-life equivalent.
2. Small cylinder of radius 0.1m and height of 1m, with collision
3. Large Cylinder of radius 1m and height 1m, with collision
4. Goal transluscent cylinder, of radius 0.1 and length 1m, no collision.