2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
October 14-18, 2024. Abu Dhabi, UAE

CBFKIT: A Control Barrier Function Toolbox for Robotics Applications

Mitchell Black, Georgios Fainekos, Bardh Hoxha, Hideki Okamoto, Danil Prokhorov

Abstract— **This paper introduces CBFKIT, a Python/ROS toolbox for safe robotics planning and control under uncertainty. The toolbox provides a general framework for designing control barrier functions for mobility systems within both deterministic and stochastic environments. It can be connected to the ROS open-source robotics middleware, allowing for the setup of multi-robot applications, encoding of environments and maps, and integrations with predictive motion planning algorithms. Additionally, it offers multiple CBF variations and algorithms for robot control. The CBFKit is demonstrated on the Toyota Human Support Robot (HSR) in both simulation and in physical experiments.**

I. INTRODUCTION

Control Barrier Function (CBF) based control [1] is characterized by some distinct advantages that make it a good candidate for control within a safety-first development framework. First, CBFs provide a mechanism to guarantee safety similarly to how Lyapunov theory guarantees stability. That is, CBFs enforce a barrier-like inequality condition on the control input to the system and, in doing so, they may be used to render a set of states satisfying a given constraint forward invariant. In other words, if the system starts in a safe state, then it will always remain safe.

In addition, in many cases, CBFs can also be combined with control Lyapunov functions (CLFs) to both stabilize the system and guarantee safety. Combined CLF and CBF control can be achieved either through pairs of affine constraints in an optimization-based controller [2], [3] or via jointly synthesized Lyapunov-Barrier functions [4], [5]. The definition of the safe operating region under the influence of a CBF (forward invariant) is flexible and it provides an unambiguous way to document the operational domain for safety certification. Finally, many CBF-based control methods can be efficiently implemented online for control-affine system models, which are typically general enough to capture many systems of practical interest.

The strengths of CBF-based control have led to wide exploration in terms of both foundational research and practical applications. For instance, CBFs provide a versatile theoretical framework for control design using analytical formulations [4], [6] as well as optimization based control [2], [7]. It is also possible to investigate safe control synthesis for hybrid dynamical systems [8]. CBFs have also shown promise in randomized methods [9], [10], and in predictive control both as a stand-alone framework [11], [12] and as constraints in Model Predictive Control (MPC) [13]–[15]. In practice, CBFs have been applied to (semi-)automated driving [2], [16]–[18], arm manipulators [19]–[21], and multi-agent coordination [22]–[25] among other applications.

In this work, we build upon the momentum of CBF research to develop an open source publicly[1] available Python package CBFKIT for CBF-based control, which can be deployed within the Robot Operating System (ROS)[2]. Our objective is to build, maintain, and support a modular extendable toolbox that encompasses some of our recent research in risk-aware CBF control [10], [26]–[28] and beyond [7], [14], [29]–[31].

We take a functional programming approach where the user needs to instantiate functions representing a model/system and a controller. The controller itself can be model-based, where the model of the system is used for control, or model free. Following functional programming principles in Python offers two unique benefits for CBFKIT. First, automatic code optimization and execution becomes possible using JAX [32]. Second, it is easy to call CBFkit from a test generation / design optimization toolbox in Python such as Ψ-TALIRO [33]. Easy integration with a design optimization tool can assist the developer with tuning of control parameters for faster deployment. Likewise, simulation-based testing using formal requirements [34], [35] can be utilized to assess model safety and performance, and its relation to the real system safety and performance [36].

Moreover, ROS integration enables easy deployment on physical robot platforms as well as on digital twins. In addition, safety requirements can be monitored using online monitoring tools such as RTAMT [37] or STREM [38]. Finally, we demonstrate the CBFKIT framework integration with ROS by working with the Toyota Human Support Robot (HSR) [39] in both real and simulated experiments.

Contribution: We present the first publicly available Python toolbox, called CBFKIT, for CBF-based control in ROS. CBFKIT is modular and extendable with the ultimate goal of being the go-to-software library for all CBF needs. The toolbox not only contains control examples for Toyota HSR, but also tutorials for newcomers to the CBF-based control methods.

II. SUPPORTED MODELS AND CONTROL DESIGN PROBLEMS

Our goal for CBFKIT is to be an all encompassing tool for CBF-based feedback control design. As such, CBFKIT supports a number of different classes of control-affine models (model of system Σ):

The authors are listed in alphabetical order and are with Toyota Motor North America R&D, Ann Arbor, MI 48105, USA. email: first.last@toyota.com

[1] https://github.com/bardhh/cbfkit.git
[2] http://www.ros.org/

1) Deterministic, continuous-time Ordinary Differential Equations (ODE):

$$\dot{x} = f(x) + g(x)u, \tag{1}$$

where $x \in \mathcal{X} \subset \mathbb{R}^n$ is the system state, $u \in \mathcal{U} \subset \mathbb{R}^m$ is the control input, $f : \mathbb{R}^n \to \mathbb{R}^n$ and $g : \mathbb{R}^n \to \mathbb{R}^{n\times m}$ are locally Lipschitz functions, and $x(0) \in \mathcal{X}_0 \subseteq \mathcal{X}$ is the initial state of the system.

2) Continuous-time ODE under bounded disturbances:

$$\dot{x} = f(x) + g(x)u + Mw, \tag{2}$$

where $w \in \mathcal{W}$ is the disturbance input, $\mathcal{W}$ is a hypercube in $\mathbb{R}^l$, and M is a $n \times l$ zero-one matrix with at most one non-zero element in each row.

3) Stochastic differential equations (SDE):

$$\mathrm{d}x = \big(f(x) + g(x)u\big)\mathrm{d}t + \sigma(x)\mathrm{d}w \tag{3}$$

where $\sigma : \mathbb{R}^n \to \mathbb{R}^{n\times q}$ is locally Lipschitz, and bounded on $\mathcal{X}$, and $w \in \mathbb{R}^q$ is a standard q-dimensional Wiener process (i.e., Brownian motion) defined over the complete probability space $(\Omega, \mathcal{F}, P)$ for sample space Ω, σ-algebra $\mathcal{F}$ over Ω, and probability measure $P : \mathcal{F} \to [0, 1]$.

We remark that currently only memoryless systems are supported. That is, the dynamics are independent of the time history of the system. Discrete-time variants can also be considered, e.g., for reinforcement learning methods or MPC. For example, in the literature [13]–[15], it is assumed that the ODE model in discrete time takes the form:

$$x_{i+1} = x_i + \int_{t_i}^{t_{i+1}} \big[f(x(t)) + g(x(t))u_i\big]dt$$

where the integral can be approximated either analytically or with numerical methods.

In certain practical applications, not all the states of the system may be observable. In such scenarios, we may assume that a state vector y is observable. For example, in the case of SDE, we may assume:

$$dy = Cx\mathrm{d}t + D\mathrm{d}v$$

where $C \in \mathbb{R}^{p\times n}$, $D \in \mathbb{R}^{p\times r}$, and $v \in \mathbb{R}^r$ is a standard Wiener process.

The framework of CBF-based control can provide safety guarantees by enforcing a forward invariant set. Namely, if the system starts in a safe state, then it should always stay in the safe set.

Definition *1:* [Set Invariance [40]] A set $S \subseteq \mathbb{R}^n$ is forward invariant w.r.t the system Σ iff for every $x(0) \in S$, its solution satisfies $x(t) \in S$ for all $t \geq 0$.

From a usability perspective, it may be easier to specify the unsafe set of states $\bar{S}$. The unsafe set of states $\bar{S}$ for a model Σ can be defined using a locally Lipschitz function $h : \mathbb{R}^n \to \mathbb{R}_+$ as

$$\bar{S} =: \{x \mid h(x) < 0\}. \tag{4}$$

It follows that the safe set of states is

$$S = \mathbb{R}^n \setminus \bar{S} = \{x \mid h(x) \geq 0\}. \tag{5}$$

In CBFKIT, the end-user is free within the framework provided to specify such a function (or functions) h, the CBF condition for which is then automatically evaluated for the provided model given the known states and inputs.

Given an unsafe set $\bar{S}$ and depending on the type of the model Σ under consideration, we have two different types problems that we consider:

1) In case of system models (1)- (2), the goal is to design a control $u(t)$ such that the state of the system $x(t)$ never enters the unsafe set $\bar{S}$ for any time $t \geq 0$.
2) In case of system models (3), the goal is to design a control feedback $u(t)$ such that the probability that the state of the system $x(t)$ enters the unsafe set $\bar{S}$ is upper bounded by a given probability $\bar{p}$ over a bounded time interval $[t, t + T]$ for some finite time T.

In CBFKIT, we provide solutions (feedback controllers) to the above two problems using Quadratic Program formulations as in, e.g., [2], [11] for model (1), [7], [41] for model (2), and [26]–[28] for model (3).

III. CBFKIT TOOLBOX

The Toolbox is developed in Python and is designed to connect to Robot Operating System (ROS).

A. Software Design

The toolbox is built on functional programming principles, emphasizing data immutability and programmatic determinism. This programming paradigm ensures that system states are not altered in-place; instead, functions return new states, thereby enhancing code reliability, maintainability, and facilitating debugging and testing.

The toolbox architecture is centered around a collection of pure functions that represent the core operations required for simulation and control of dynamical systems. These functions are organized into modules corresponding to specific functionalities, e.g., `dynamics`, `controller`, `estimator`, `integrator`, `perturbation`, and `sensor`. Each function is designed to take input parameters and return new outputs without side effects, adhering to the principles of functional programming.

The architecture replaces traditional object-oriented constructs, such as classes and inheritance, with function compositions and higher-order functions. For instance, system dynamics and controllers are defined through composable functions rather than through methods of a `System` class. This enables more flexible and reusable code, as functions can be combined and reused across the system without the tight coupling introduced by class inheritance.

For improved usability, we provide template functions for `dynamics` (i.e., given a state x compute the control-affine dynamics $f(x)$ and $g(x)$), and `controller` (i.e., given a time t and state x compute the control input $u(t, x)$).

Function currying is employed to refine functionality; for instance, a `cbfcontroller` function, a specialized

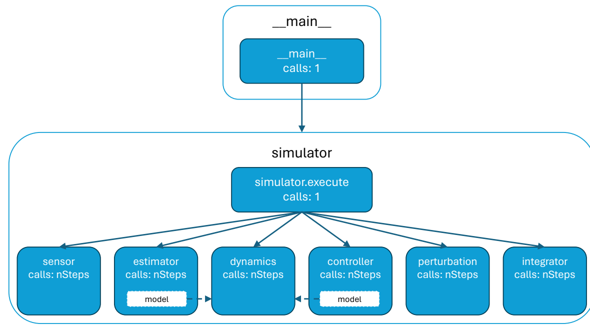

Fig. 1. Call graph depicting relations between function calls important for building an example simulation, i.e., `sensor`, `estimator`, `dynamics`, `controller`, `perturbation`, and `integrator`, using CBFKIT.

form of `controller`, initially requires multiple control parameters during setup. Post-initialization, the `controller` function consistently requires time t and state vector x as arguments. Similarly, while a `plant` may be initialized with numerous parameters, its `dynamics` function invariably accepts time t and state vector x as inputs.

The toolbox is managed using Poetry[3] and features both a Docker container and a VsCode devContainer. This setup expedites development and tutorial execution, ensuring a streamlined process and consistent user environment.

B. Args/Return Types for Important Template Functions

The functional design is reflected in the specification of input/output types for key functions in the toolbox, as outlined in the Table I. This explicit type declaration aids in understanding the flow of data through the system and ensures that functions can be composed safely and predictably.

The toolbox provides users with a library of example models, such as double integrator, unicycle, bicycle models and omnidirectional robots. It also includes example scripts that show how the toolbox can be used for various models and controllers. Also, some handy utility functions are included which can help store, load, and visualize system behaviors.

C. CBF Implementation

Implementation of a CBF-based controller uses the ego system dynamics, additional agents' dynamics, and the environment to synthesize a controller to enforce invariance of a region of the state space satisfying a set of safety constraints. There are two main steps in the control synthesis: 1) safety constraint generation and 2) controller design.

In order to generate the safety constraints, a barrier function as in (5) is related to the system inputs via mathematical derivation. A key feature of the toolkit is the use of the auto-differentiation capabilities of JAX [32] for computation of the derivative of the barrier function. For complex dynamics, this derivation can be computationally challenging using symbolic toolboxes such as SymPy [42]. In contrast, JAX computes derivatives of a function without manually or symbolically differentiating the function, which helps our tool support arbitrary systems and barrier functions, provided that the barrier functions used for control have relative-degree[4] one with respect to the system dynamics. If the barrier function of interest has a relative-degree greater than one with respect to the system dynamics, our `rectify-relative-degree` module may be used to derive a new barrier function whose zero super-level set is a subset of that associated with the original barrier function. The module works by iteratively differentiating the original barrier function with respect to the system dynamics until the control input appears explicitly (as determined by evaluating samples of the term $\frac{\partial h(x_s)}{\partial x} g(x_s) u$ for samples $x_s \in \mathcal{X}$), and applying either exponential CBF [43] or high-order CBF [44] principles to return a "rectified" barrier function.

The second step in the process is the controller design. One approach to CBF-based control is to pair it with a nominal or reference controller and solve an optimization problem to compute the control solution "nearest" to the reference input that satisfies the CBF safety constraint(s). For the class of systems we consider, the problem can be posed as a quadratic program (QP) and solved with jaxopt [45], a just-in-time (JIT) compilable Python package for optimization. In our code examples, the QP can be solved in a few milliseconds.

D. Code Generation

To expedite building new simulations and experiments, we provide a `codegen` module for creating arbitrary `dynamics`, `controller`, and `cbf` or `clf` Python functions. These templates provide the user with a direct interface to our `simulation` module, which supports both single example and multi-processor Monte Carlo trials. An example using `codegen` is highlighted in Section III-G.

E. Interaction with the Robot Operating System (ROS)

To develop robotics applications, we often use the Robot Operating System (ROS). ROS provides a set of libraries and tools for robotics applications. It provides a message passing infrastructure which enables a publish/subscribe messaging system to allow communication between various components or nodes of the system. For example, we may subscribe to an odometry topic using the `rospy.Subscriber` function, which triggers a callback function whenever a new message is published to the topic. The callback function can then be used to update the state of the robot. Similarly, we may publish a command to a topic such as angular velocity using the `rospy.Publisher` function to actuate the robot.

In our toolbox, we provide a `ros` module to connect `dynamics`, `controller`, `estimator`, etc. functions with ROS. The module essentially contains wrapper functions to link instantiated functions with ROS topics, which allows other nodes to interact with the system through ROS.

F. Tutorials

Finally, the toolbox includes a set of step-by-step tutorials for rapid prototyping and benchmark creation of CBF control

[3]https://python-poetry.org

[4]A function $p : \mathbb{R}_+ \times \mathbb{R}^n \to \mathbb{R}$ is said to be of relative-degree r with respect to the dynamics (1) if r is the number of times p must be differentiated before one of the control inputs u appears explicitly.

TABLE I
ARGUMENTS/RETURN TYPES FOR IMPORTANT TEMPLATE FUNCTIONS

Function	Arguments	Types	Returns	Types
controller	t, x	float, Array[n]	u, data	Array[m], Dict[str, Any]
dynamics	t, x	float, Array[n]	f, g	Array[n], Array[n, m]
estimator	t, y, z, u, c	float, Array[p], Array[n], Array[m], Array[p, p]	z, c	Array[n], Array[p, p]
integrator	x, $\dot{x}$, dt	Array[n], Array[n], float	x	Array[n]
perturbation	x, u, f, g	Array[n], Array[m], Array[n], Array[n, m]	pert	Callable[[jax.random.PRNGKey], Array[n]]
sensor	t, x, σ, key	float, Array[n], Array[n, n], jax.random.PRNGKey	y	Array[p]

Note: arguments and return variable names are taken from the code, and Array[k] denotes a `jax.numpy.array` with data type `float` of length k.

laws for nonlinear dynamical systems. The tutorials are developed as Jupyter notebooks and walk users through the process step-by-step, from formulating the system using `codegen` to generating a reference trajectory, designing a barrier function, and relating it to the system dynamics via mathematical derivation. Once this is done, the task can be set as a Quadratic Program (QP) with the goal of achieving the desired outcome. Table II summarizes the available tutorials distributed with CBFKIT.

TABLE II
TUTORIALS ON CBF SYNTHESIS

System	Environment	CBF order
Van der Pol oscillator	Static	1
Nonlinear unicycle	Static	2
Nonlinear unicycle	Dynamic	2

G. Example

Consider a unicycle seeking to reach a goal set while avoiding a static, circular obstacle. The code in this subsection is taken directly from `tutorials/unicycle_reach_avoid.py`.

The `codegen` module is used to generate code for the unicycle equations of motion, the nominal control law, and the barrier functions as shown in Listing 1.

```
1  import jax.numpy as jnp
2  from cbfkit.codegen.create_new_system.generate_model import
       generate_model
3
4
5  def compute_theta_d(x, y, th):
6      thd = f"arctan2(yg - {y}, xg - {x})"
7      return f"{th} + arctan2(sin({thd} - {th}), cos({thd} - {th}))
       "
8
9
10 def norm(x, y):
11     z = f"jnp.linalg.norm(jnp.array([{x} - xg, {y} - yg]))"
12     return z
13
14
15 params = {}
16 x, y, v, th = "x[0]", "x[1]", "x[2]", "x[3]"
17 drift = [f"{v} * cos({th})", f"{v} * sin({th})", "0", "0"]
18 control_mat = ["[0, 0]", "[0, 0]", "[1, 0]", "[0, 1]"]
19 barriers = [f"({x} - xo)**2 + ({y} - yo)**2 - r**2", f"l**2 - {v
       }**2"]
20 params["cbf"] = [{"xo: float": 1.0, "yo: float": 1.0, "r: float":
       1.0}, {"l: float": 1.0}]
21 u_nom = f"kp * ({norm(x ,y)} - {v}), kp * ({compute_theta_d(x, y,
       th)} - {th})"
22 params["controller"] = {"kp: float": 1.0, "xg: float": 1.0, "yg:
       float": 1.0}
23
24 generate_model(
25     directory="./tutorials/models",
26     model_name="accel_unicycle",
27     drift_dynamics=drift,
28     control_matrix=control_mat,
29     barrier_funcs=barriers,
30     nominal_controller=u_nom,
31     params=params,
32 )
```

Listing 1. Codegen for dynamics, nominal control, and barrier functions.

The `codegen` module uses formatted strings extensively, which is why all the above formulae are defined as strings (or lists of strings). Since the state vector appears in functions as the variable `x` throughout the toolbox, it is important that the states are defined as "x[0]", "x[1]",...,"x[n-1]" (line 16). The `generate_model` function generates a new folder at `tutorials/models` called `accel_unicycle`, and populates it with files for the plant model (`plant.py`), the nominal control law (`controller_1.py` in `accel_unicycle/controllers`), and the barrier functions (`barrier_1.py`, `barrier_2.py` in `accel_unicycle/certificate_functions /barrier_functions`).

These modules are imported in Listing 2 and are used to define `dynamics` and `nominal_controller` functions.

```
1 import models.accel_unicycle as unicycle
2 from models.accel_unicycle.certificate_functions.
      barrier_functions.barrier_1 import cbf
3 from models.accel_unicycle.certificate_functions.
      barrier_functions.barrier_2 import cbf2_package
4
5 initial_state = jnp.array([2.0, 2.0, 0.0, -3 * jnp.pi / 4])
6 actuation_limits = jnp.array([1.0, jnp.pi])
7 dynamics = unicycle.plant()
8 nominal_controller = unicycle.controllers.controller_1(kp=1.0, xg
      =-2.0, yg=-2.0)
```

Listing 2. Instantiating dynamics and nominal controller functions.

In Listing 3, the CBF-based controller is built. The important components of this block are i) using the `rectify_relative_degree` function to extract a constraint function with relative-degree 1 w.r.t. the unicycle dynamics from the original obstacle avoidance constraint (line 12-14), ii) concatenating the cbf packages into a usable object using the `concatenate_certificates` function (lines 16-19), and iii) instantiating the CBF-based controller function `controller` (lines 20-26).

```
1 from cbfkit.controllers.model_based.cbf_clf_controllers import
      vanilla_cbf_clf_qp_controller
2 from cbfkit.controllers.model_based.cbf_clf_controllers.utils.
      barrier_conditions.zeroing_barriers import (
3     linear_class_k,
4 )
5 from cbfkit.controllers.model_based.cbf_clf_controllers.utils.
      certificate_packager import (
6     concatenate_certificates,
7 )
8 from cbfkit.controllers.model_based.cbf_clf_controllers.utils.
      rectify_relative_degree import (
```

```
9       rectify_relative_degree,
10  )
11
12  cbf1_package = rectify_relative_degree(
13      cbf(xo=0.9, yo=1.0, r=0.5), dynamics, len(initial_state),
          roots=-1.0 * jnp.ones((2,))
14  )
15
16  barriers = concatenate_certificates(
17      cbf1_package(certificate_conditions=linear_class_k(10.0)),
18      cbf2_package(certificate_conditions=linear_class_k(1.0), l
          =1.0),
19  )
20  controller = vanilla_cbf_clf_qp_controller(
21      actuation_limits,
22      nominal_controller,
23      dynamics,
24      barriers,
25      p_mat=jnp.diag(jnp.array([1.0, 0.1])),
26  )
```

Listing 3. Instantiating CBF-based controller function.

Finally, the simulation is executed in Listing 4 from `initial_state` and lasts 10 sec at a time-step of 0.01 sec. Additional imports are required, including the `simulator` module itself and a `sensor`, `estimator`, and `integrator` (lines 1-4). When the simulation is executed by calling `simulator.execute`, the results for the state, control, state estimate, and state covariance matrix are stored in variables `x, u, z` and `p` as `numpy.ndarray` objects and are immediately available for plotting/analysis.

```
1   from cbfkit.simulation import simulator
2   from cbfkit.sensors import perfect
3   from cbfkit.estimators import naive
4   from cbfkit.integration import forward_euler
5
6   x, u, z, p, dkeys, dvals = simulator.execute(
7       x0=initial_state,
8       dt=1e-2,
9       num_steps=1000,
10      dynamics=dynamics,
11      integrator=forward_euler,
12      controller=controller,
13      sensor=perfect,
14      estimator=naive,
15  )
```

Listing 4. Executing the unicycle simulation.

IV. Experiments

A. Human Support Robot (HSR)

The HSR is a robot developed by Toyota Motor Corporation to operate in diverse environments while interacting with humans (Figure 2). It has a differential wheeled base and five degrees-of-freedom arm. With its highly maneuverable, compact, and lightweight cylindrical body and folding arm, the HSR can pick objects up off the floor, retrieve objects from shelves, and perform a variety of other tasks.

TABLE III
Dimensions and Specifications of the Human Support Robot

Specification	Value
Body diameter	430 mm
Body height	1,005 mm - 1,350 mm
Weight	Approx. 37 kg
Arm length	Approx. 600 mm
Shoulder height	340 mm - 1,030 mm
Objects that can be held	1.2 kg or less, 130 mm wide or less
Maximum speed	0.8 km/h

To demonstrate the use of the CBFkit to control a physical system, we conducted an experiment in a laboratory

Fig. 2. Human Support Robot (HSR). Cr: TOYOTA

environment in which an HSR was tasked with reaching a goal location while avoiding collisions with a careless human agent and remaining within a specified corridor. In software, the HSR was taken to be the ego agent and assumed to evolve according to the following dynamic unicycle model:

$$\dot{x}_e = v_e \cos\theta_e, \tag{6a}$$
$$\dot{y}_e = v_e \sin\theta_e, \tag{6b}$$
$$\dot{v}_e = a_e, \tag{6c}$$
$$\dot{\theta}_e = \omega_e, \tag{6d}$$

where x_e and y_e are the lateral and longitudinal coordinates (in m) with respect to the origin $\boldsymbol{s}_0$ of an inertial frame $\mathcal{F}$, v_e is the speed (in m/s), θ_e is the heading angle (in rad), a_e is the acceleration (in m/s^2) in the heading direction, and ω_e is the heading angular velocity (in rad/s), such that the state is $\boldsymbol{x}_e = [x_e \; y_e \; v_e \; \theta_e]^\top$ and the control is $\boldsymbol{u}_e = [a_e \; \omega_e]^\top$. The human agent was modeled as a 2D single-integrator, i.e.,

$$\dot{x}_h = v_{x,h}, \tag{7a}$$
$$\dot{y}_h = v_{y,h}, \tag{7b}$$

where the state is $\boldsymbol{x}_h = [x_h \; y_h]^\top$ and the control input is $\boldsymbol{u}_h = [v_{x,h} \; v_{y,h}]^\top$, where $v_{x,h}$ and $v_{y,h}$ are the velocities of the human with respect to the x- and y-directions. To ensure collision avoidance, we used a form of the future-focused CBFs introduced for autonomous vehicle control in [11], in this case defined as

$$h_{ff}(\boldsymbol{x}_e, \boldsymbol{x}_h) = \min_{\tau \in \mathcal{T}} \hat{D}^2(\boldsymbol{x}_e, \boldsymbol{x}_h, \tau) - (2R)^2,$$

where $\mathcal{T} = [t, t+T]$ for some time headway $0 < T < \infty$, $\hat{D} : \mathbb{R}^4 \times \mathbb{R}^4 \times \mathcal{T} \to \mathbb{R}_+$ is the predicted distance between agents under zero-control policies, i.e., $a_i(\tau) = \omega_i(\tau) = 0$, $\forall \tau \in \mathcal{T}$, $i \in \{e, h\}$, and $R > 0$ is a safe distance. To enforce that the HSR remained inside the specified corridor, we defined an admissible rectangular region with edges $x_{min}, x_{max}, y_{min}, y_{max} \in \mathbb{R}$ and used the following CBFs:

$$h_{r1}(\boldsymbol{x}_e) = v_e \cos\theta_e + x_e - x_{min},$$
$$h_{r2}(\boldsymbol{x}_e) = x_{max} - x_e - v_e \cos\theta_e,$$
$$h_{r3}(\boldsymbol{x}_e) = v_e \sin\theta_e + y_e - y_{min},$$
$$h_{r4}(\boldsymbol{x}_e) = y_{max} - y_e - v_e \sin\theta_e.$$

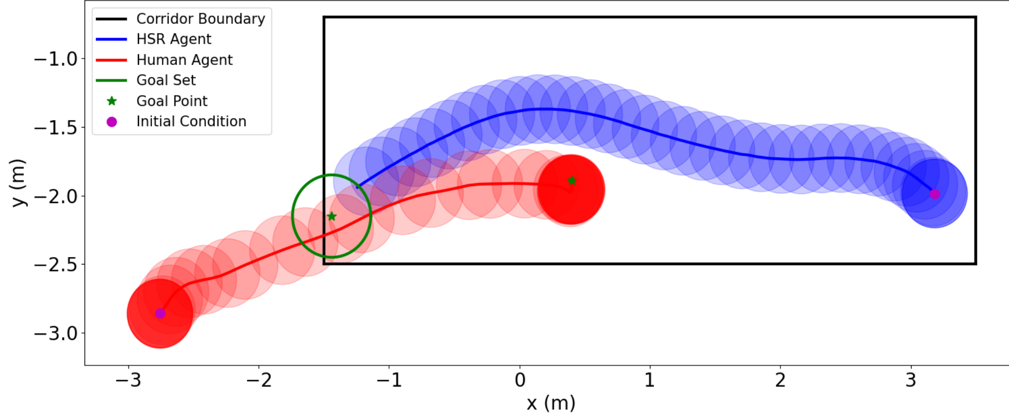

Fig. 3. xy paths of the HSR and human agents taken in the goal-reaching experiment. The overlaid circles represent the specified sizes of the ego and human agents and represent temporal evolution at 0.25 sec increments. An animation of the system behavior can be found at: `https://youtu.be/MXQAK2jwLLE`.

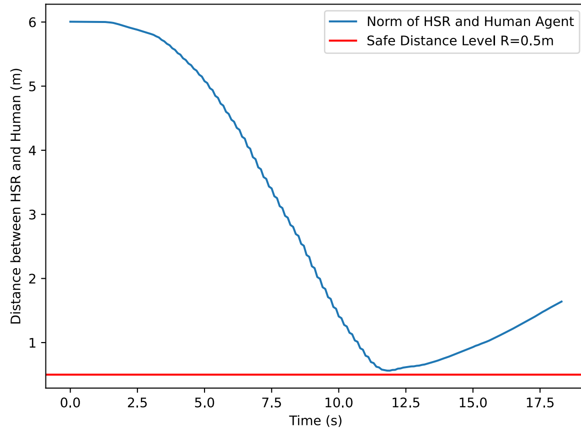

Fig. 4. Distance between HSR and Human Agents.

We then used a form of the well-known CBF quadratic program (CBF-QP) control law for the HSR, as follows:

$$\boldsymbol{u}_e^* = \underset{\boldsymbol{u}_e \in \mathcal{U}_e}{\arg\min} \frac{1}{2}\|\boldsymbol{u}_e - \boldsymbol{u}_e^0\|^2 \tag{8a}$$

$$\text{s.t.} \quad \forall k \in \{ff, r1, r2, r3, r4\}$$

$$L_f h_k + L_g h_k \boldsymbol{u}_e \geq -\alpha(h_k), \tag{8b}$$

where (8a) seeks to minimize the deviation of the solution $\boldsymbol{u}_e^*$ from the reference input $\boldsymbol{u}_e^0$, and (8b) enforces the CBF condition for each of the specified constraint functions. In this experiment, we used a reference control $\boldsymbol{u}_e^0$ based on the LQR control law found in [11, App. I], and chose $\alpha(y) = y$.

The paths for the HSR and human agents are shown in Figure 3. Evidently, the CBF-QP controller (8) steers the HSR out of the path of the human agent in advance of any danger despite the HSR's nominal LQR controller driving it directly through the human agent en route to the goal. As shown in Figure 4, the HSR obeys the required safety margin with respect to the human at all times.

V. Conclusion

In this paper, we presented the first open source, publicly available Python toolbox for CBF-based control for generic systems both in simulation and ROS-enabled hardware experimentation. The toolbox, called CBFkit, is developed using a functional programming architecture so it promotes immutability and reliability. The toolbox already contains a number of CBF methods from the literature, and it has been successfully applied for motion planning on the Toyota HSR in a laboratory environment.

In the near future, we plan to extend CBFkit to include a variety of methods and algorithms from the literature. Ongoing work has added higher-level planners to the toolbox [46]. We also envision the toolbox to provide a number of benchmarks that will enable the research community to compare different algorithms and approaches.

Acknowledgments

The authors would like to thank Hardik Parwana for his continued support in improving CBFkit since its release. Hardik's improvements have been documented in [46].

References

[1] A. D. Ames, S. Coogan, M. Egerstedt, G. Notomista, K. Sreenath, and P. Tabuada, "Control barrier functions: Theory and applications," in *European Control Conference*, 2019.

[2] A. D. Ames, J. W. Grizzle, and P. Tabuada, "Control barrier function based quadratic programs with application to adaptive cruise control," in *53rd IEEE Conference on Decision and Control (CDC)*, 2014, pp. 6271–6278.

[3] K. Garg and D. Panagou, "Control-lyapunov and control-barrier functions based quadratic program for spatio-temporal specifications," in *2019 IEEE 58th Conference on Decision and Control (CDC)*. IEEE, 2019, pp. 1422–1429.

[4] P. Braun, C. M. Kellett, and L. Zaccarian, "Complete control lyapunov functions: Stability under state constraints," in *11th IFAC Symposium on Nonlinear Control Systems*, 2019.

[5] Z. Wu, F. Albalawi, Z. Zhang, J. Zhang, H. Durand, and P. D. Christofides, "Control lyapunov-barrier function-based model predictive control of nonlinear systems," *Automatica*, vol. 109, p. 108508, 2019.

[6] P. Wieland and F. Allgower, "Constructive safety using control barrier functions," *IFAC Proceedings Volumes*, vol. 40, no. 12, pp. 462–467, 2007, 7th IFAC Symposium on Nonlinear Control Systems.

[7] S. Yaghoubi, G. Fainekos, and S. Sankaranarayanan, "Training neural network controllers from control barrier functions in the presence of disturbances," in *IEEE Intelligent Transportation Systems Conference (ITSC)*, 2020.

[8] Y. Meng and J. Liu, "Lyapunov-barrier characterization of robust reach–avoid–stay specifications for hybrid systems," *Nonlinear Analysis: Hybrid Systems*, vol. 49, p. 101340, 2023.

[9] A. Ahmad, C. Belta, and R. Tron, "Adaptive sampling-based motion planning with control barrier functions," in *IEEE 61st Conference on Decision and Control (CDC)*, 2022, pp. 4513–4518.

[10] K. Majd, S. Yaghoubi, T. Yamaguchi, B. Hoxha, D. Prokhorov, and G. Fainekos, "Safe navigation in human occupied environments using sampling and control barrier functions," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2021.

[11] M. Black, M. Jankovic, A. Sharma, and D. Panagou, "Future-focused control barrier functions for autonomous vehicle control," in *2023 American Control Conference (ACC)*. IEEE, 2023, pp. 3324–3331.

[12] J. Breeden and D. Panagou, "Predictive control barrier functions for online safety critical control," in *2022 IEEE 61st Conference on Decision and Control (CDC)*. IEEE, 2022, pp. 924–931.

[13] J. Zeng, B. Zhang, and K. Sreenath, "Safety-critical model predictive control with discrete-time control barrier function," in *American Control Conference (ACC)*, 2021, pp. 3882–3889.

[14] A. S. Lafmejani, S. Berman, and G. Fainekos, "Nmpc-lbf: Nonlinear mpc with learned barrier function for decentralized safe navigation of multiple robots in unknown environments," in *IEEE Conference on Robotics and Automation*, 2022.

[15] R. Grandia, A. J. Taylor, A. Singletary, M. Hutter, and A. D. Ames, "Nonlinear model predictive control of roboticsystems with control lyapunov functions," in *Robotics: Science and Systems*, 2020.

[16] W. Xiao, N. Mehdipour, A. Collin, A. Y. Bin-Nun, E. Frazzoli, R. D. Tebbens, and C. Belta, "Rule-based optimal control for autonomous driving," in *12th ACM/IEEE International Conference on Cyber-Physical Systems*, p. 143–154.

[17] S. He, J. Zeng, B. Zhang, and K. Sreenath, "Rule-based safety-critical control design using control barrier functions with application to autonomous lane change," in *American Control Conference (ACC)*, 2021, pp. 178–185.

[18] T. G. Molnar, A. Alan, A. K. Kiss, A. D. Ames, and G. Orosz, "Input-to-state safety with input delay in longitudinal vehicle control," *IFAC-PapersOnLine*, vol. 55, no. 36, pp. 312–317, 2022, 17th IFAC Workshop on Time Delay Systems.

[19] W. Shaw-Cortez, C. K. Verginis, and D. V. Dimarogonas, "Safe, passive control for mechanical systems with application to physical human-robot interactions," in *IEEE International Conference on Robotics and Automation (ICRA)*, pp. 3836–3842.

[20] A. Singletary, W. Guffey, T. G. Molnar, R. Sinnet, and A. D. Ames, "Safety-critical manipulation for collision-free food preparation," vol. 7, no. 4, 2022.

[21] F. Ferraguti, C. T. Landi, S. Costi, M. Bonfe, S. Farsoni, C. Secchi, and C. Fantuzzi, "Safety barrier functions and multi-camera tracking for human–robot shared environment," *Robotics and Autonomous Systems*, vol. 124, 2020.

[22] H. Parwana, A. Mustafa, and D. Panagou, "Trust-based rate-tunable control barrier functions for non-cooperative multi-agent systems," in *IEEE 61st Conference on Decision and Control (CDC)*, pp. 2222–2229.

[23] L. Lindemann and D. V. Dimarogonas, "Control barrier functions for multi-agent systems under conflicting local signal temporal logic tasks," *IEEE Control Systems Letters*, vol. 3, no. 3, pp. 757–762, 2019.

[24] U. Mehmood, S. Roy, A. Damare, R. Grosu, S. A. Smolka, and S. D. Stoller, "A distributed simplex architecture for multi-agent systems," *Journal of Systems Architecture*, vol. 134, 2023.

[25] Y. Emam, P. Glotfelter, S. Wilson, G. Notomista, and M. Egerstedt, "Data-driven robust barrier functions for safe, long-term operation," *IEEE Transactions on Robotics*, vol. 38, no. 3, pp. 1671–1685, 2022.

[26] S. Yaghoubi, K. Majd, G. Fainekos, T. Yamaguchi, D. Prokhorov, and B. Hoxha, "Risk-bounded control using stochastic barrier functions," *IEEE Control Systems Letters*, vol. 5, 2021.

[27] S. Yaghoubi, G. Fainekos, T. Yamaguchi, D. Prokhorov, and B. Hoxha, "Risk-bounded control with kalman filtering and stochastic barrier functions," in *IEEE Conference on Decision and Control*, 2021.

[28] M. Black, G. Fainekos, B. Hoxha, D. Prokhorov, and D. Panagou, "Safety under uncertainty: Tight bounds with risk-aware control barrier functions," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2023.

[29] D. R. Agrawal and D. Panagou, "Safe and robust observer-controller synthesis using control barrier functions," *IEEE Control Systems Letters*, vol. 7, pp. 127–132, 2022.

[30] J. Breeden, K. Garg, and D. Panagou, "Control barrier functions in sampled-data systems," *IEEE Control Systems Letters*, vol. 6, pp. 367–372, 2021.

[31] M. Black and D. Panagou, "Safe control design for unknown nonlinear systems with koopman-based fixed-time identification," *IFAC-PapersOnLine*, vol. 56, no. 2, pp. 11 369–11 376, 2023.

[32] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, "JAX: composable transformations of Python+NumPy programs," 2018. [Online]. Available: http://github.com/google/jax

[33] Q. Thibeault, J. Anderson, A. Chandratre, G. Pedrielli, and G. Fainekos, "PSY-TaLiRo: A python toolbox for search-based test generation for cyber-physical systems," in *Formal Methods for Industrial Critical Systems (FMICS)*, 2021.

[34] C. E. Tuncali, G. Fainekos, D. Prokhorov, H. Ito, and J. Kapinski, "Requirements-driven test generation for autonomous vehicles with machine learning components," *IEEE Transactions on Intelligent Vehicles*, vol. 5, pp. 265–280, 2020.

[35] M. Hekmatnejad, B. Hoxha, and G. Fainekos, "Search-based test-case generation by monitoring responsibility safety rules," in *IEEE Intelligent Transportation Systems Conference (ITSC)*, 2020.

[36] D. J. Fremont, E. Kim, Y. V. Pant, S. A. Seshia, A. Acharya, X. Bruso, P. Wells, S. Lemke, Q. Lu, and S. Mehta, "Formal scenario-based testing of autonomous vehicles: From simulation to the real world," in *23rd IEEE International Conference on Intelligent Transportation Systems (ITSC)*, pp. 1–8.

[37] D. Nickovic and T. Yamaguchi, "RTAMT: Online robustness monitors from STL," in *Automated Technology for Verification and Analysis (ATVA)*, ser. LNCS, vol. 12302. Springer, 2020, pp. 564–571.

[38] J. Anderson, G. Fainekos, B. Hoxha, H. Okamoto, and D. Prokhorov, "Pattern matching for perception streams," in *23rd International Conference on Runtime Verification (RV)*, 2023.

[39] T. Yamamoto, K. Terada, A. Ochiai, F. Saito, Y. Asahara, and K. Murase, "Development of human support robot as the research platform of a domestic mobile manipulator," *ROBOMECH Journal*, vol. 6, no. 1, 2019.

[40] F. Blanchini, "Set invariance in control," *Automatica*, vol. 35, no. 11, pp. 1747–1767, 1999.

[41] M. Jankovic, "Robust control barrier functions for constrained stabilization of nonlinear systems," *Automatica*, vol. 96, pp. 359–367, 2018.

[42] A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, T. Rathnayake, S. Vig, B. E. Granger, R. P. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, M. J. Curry, A. R. Terrel, v. Roučka, A. Saboo, I. Fernando, S. Kulal, R. Cimrman, and A. Scopatz, "Sympy: symbolic computing in python," *PeerJ Computer Science*, vol. 3, p. e103, Jan. 2017. [Online]. Available: https://doi.org/10.7717/peerj-cs.103

[43] Q. Nguyen and K. Sreenath, "Exponential control barrier functions for enforcing high relative-degree safety-critical constraints," in *2016 American Control Conference (ACC)*. IEEE, 2016, pp. 322–328.

[44] W. Xiao and C. Belta, "Control barrier functions for systems with high relative degree," in *2019 IEEE 58th Conference on decision and control (CDC)*. IEEE, 2019, pp. 474–479.

[45] M. Blondel, Q. Berthet, M. Cuturi, R. Frostig, S. Hoyer, F. Llinares-López, F. Pedregosa, and J.-P. Vert, "Efficient and modular implicit differentiation," *Advances in neural information processing systems*, vol. 35, pp. 5230–5242, 2022.

[46] H. Parwana, M. Black, G. Fainekos, B. Hoxha, H. Okamoto, and D. Prokhorov, "Model predictive path integral methods with reach-avoid tasks and control barrier functions," arXiv:2407.13693, Tech. Rep., 2024.