CrazyChoir: A ROS 2 Toolbox for Simulations and Experiments on Swarms of Cooperating Crazyflies

Lorenzo Pichierri, Andrea Testa, Giuseppe Notarstefano

***Abstract*— This abstract presents CRAZYCHOIR [1] , a modular Python framework for distributed optimization and control of Crazyflie swarms. CRAZYCHOIR is based on the Robot Operating System (ROS) 2, allowing swarm algorithms testing over Crazyflie nano-quadrotors in a virtual and real experimental framework. Exploiting Python bindings of the Crazyflie firmware, the toolbox runs realistic simulations on Webots, a real-engine simulator. Moreover, CRAZYCHOIR provides a ROS 2 Python library to perform radio communication with Crazyflies. The modular structure of the package frees the user to design and implement novel control schemes and planning algorithms, either for centralized or decentralized strategies. Thus, the toolbox can be adopted to test and validate online distributed optimization algorithms over swarms of Crazyflies. CRAZYCHOIR is available at `https://github.com/OPT4SMART/crazychoir`.**

***Index Terms*— Distributed Robot Systems; Software Architecture for Robotics and Automation; Cooperating Robots; Optimization and Optimal Control**

I. INTRODUCTION

UAV swarm robotics has emerged as a promising field of research, drawing particular attention from both academic and industrial circles. This interest arises from their versatility and suitability for various applications, making them an attractive option for a range of scenarios. In the last years, the Crazyflie nano-quadrotor platform has gained a lot of attention among researchers. Recently, several efforts have been put into the development of simulation and control toolboxes based on the novel Robot Operating System (ROS) 2 [2]. However, the research community has provided few attention to the development of ROS 2 toolboxes for the Crazyflie. In this abstract, we present CRAZYCHOIR a ROS 2 package to run realistic simulations and experiments on a swarm of cooperating Crazyflie nano-quadrotors.

CRAZYCHOIR is tailored for swarms of cooperating Crazyflie nano-quadrotors (Figure 1) , allowing the user to perform simulations in a real-engine simulator (e.g., Webots [3]), and, therefore, experiments for fast prototyping. Thanks to its modular structure, CRAZYCHOIR handles each Crazyflie as a set of independent ROS 2 nodes, reducing the failure rate subjected to single processes errors.

This work was supported in part by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 638992 - OPT4SMART) and in part by the Italian Ministry of Foreign Affairs and International Cooperation (grant No BR22GR01).

Authors are with the Department of Electrical, Electronic and Information Engineering, University of Bologna, Bologna, Italy. {lorenzo.pichierri, a.testa, giuseppe.notarstefano}@unibo.it.

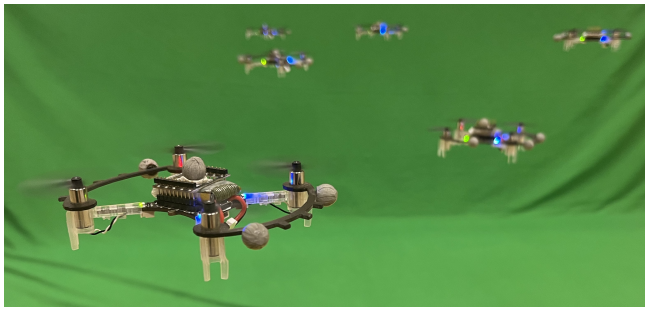

Fig. 1. Snapshot of a swarm of Crazyflie nano-quadrotors in our testbed.

The package is integrated with Bitcraze firmware bindings, which empower the realistic simulation directly implementing onboard control functions.

Finally, CRAZYCHOIR offers support for the DISROPT [4] and CHOIRBOT [5] toolboxes, enabling users to implement distributed optimization schemes and manage inter-robot communications through ROS 2.

II. CRAZYCHOIR ARCHITECTURE

Exploiting the modular paradigm of ROS 2, CRAZYCHOIR instantiates a dedicated process (also known as *node*) for each Crazyflie. Furthermore, all the programming layers (e.g., simulator, controller, planner, and radio handler) are organized in the same fashion. The package software architecture consists of:

i) control layer,
ii) swarm planning layer,
iii) cooperative decision-making layer,
iv) radio communication layer,
v) realistic simulation layer.

A graphical illustration of the software architecture is depicted in Figure 2. Specifically, the control layer is designed to implement off-board feedback-control schemes, and, according to their needs, users can extend the existent classes. Then, the swarm planning and the cooperative decision-making layer endow CRAZYCHOIR of distributed, online optimization and control algorithms. These structures compute the reference trajectories for each quadrotor of the swarm. Once the control input is generated, it can be sent via radio, or transmitted via ROS 2 topics to the simulators. In fact, users can decide to simulate experiments using either Webots or lightweight numerical integrations, visualized in RVIZ. Also, CRAZYCHOIR is integrated with motion capture systems, e.g., Vicon, which is exploited to retrieve the pose of Crazyflies during experiments. Furthermore, to collect also

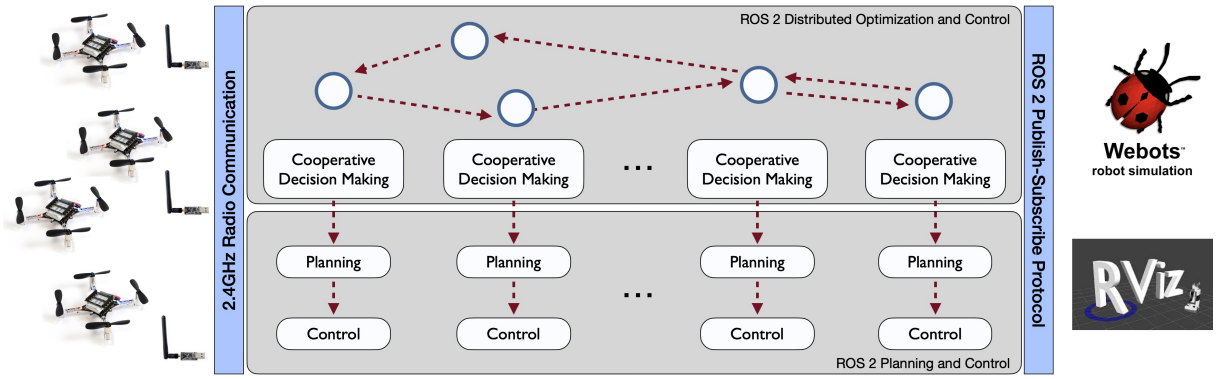

Fig. 2. CRAZYCHOIR architecture. Crazyflies can exchange information with neighbors to implement distributed feedback and optimization schemes. Local classes handle planning and low-level control. Control inputs can be sent to simulation environments (Webots or RVIZ) or to real robots via radio.

linear and angular velocities, usually not provided by motion capture systems, the package implements derivative and low-pass filters.

III. CRAZYCHOIR TOOLKIT

A. Control Library

The toolbox provides a template Python class named `CrazyflieController`, functional to implement the desired control schemes for the Crazyflie. Moreover, CRAZYCHOIR already provides classical flatness-based control laws, aiming at tracking a desired flat-output trajectory, e.g., position and yaw profiles with their derivatives. This control scheme is implemented in `HierarchicalController`, which extends the `CrazyflieController` template. The attributes of this class are two Strategy classes `PositionControl` and `AttitudeControl`.

B. Cooperative Decision-Making and Planning Strategies

The package provides a set of high-level functions to perform decision-planning task. Also, CRAZYCHOIR allows the implementation of cooperative decision-making and distributed feedback schemes on swarms of Crazyflies.

To facilitate the use of the package, CRAZYCHOIR provides a graphical user interface to hand-draw trajectories and directly send them to the Crazyflie(s). Also, classical operations needed during simulations/experiments are supported (e.g., hovering, landing, starting the experiment, etc.). A snapshot of the interface is reported in Figure 3, where we draw the name of our laboratory (i.e., *Casy*).

Point-to-point and smooth multi-spline trajectories are also included in CRAZYCHOIR. Planning strategies are managed by the `TrajectoryHandler` class. This class details the topic on which the desired trajectory will be published. Thereby, ROS 2 nodes can instantiate dynamically the appropriate publisher and subscribers. The class also defines the trajectory message type and the respective callback function.

In CRAZYCHOIR, distributed optimization and control schemes can be easily implemented by exploiting the compatibility with CHOIRBOT [5], a ROS 2 package for cooperative robotics. Specifically, in these cooperative applications, each nano-quadrotor exchanges data with few neighbors, possibly leveraging time-varying communication topologies. Thus, the swarm exploits local knowledge of each quadrotor, leveraging inter-robot communications to achieve a global mission. Either static or time-varying directed communication networks, are performed by publisher-subscriber protocol. The classes `CFGuidance` and `CFDistributedFeedback` are employed in CRAZYCHOIR to implement the aforementioned strategies.

Fig. 3. Snapshot of the proposed GUI.

C. Radio Communication

In this subsection, we present the physical communication functionalities that allow the user to send angular or angular-rate setpoints to the Crazyflie. Usually, Crazyflies communicate with a PC employing the Crazyradio PA, a 2.4 GHz USB radio dongle. To handle radio communication, CRAZYCHOIR provides a tailored `RadioHandler` class, which instantiates an independent ROS 2 node for each Crazyradio PA. Moreover, this method leverages the official `Crazyflie-lib-python` developed by Bitcraze to communicate with the swarm. Also, the use of official library is exploited to retrieve onboard logging data, and, hence, to provide logging features.

Following the modular paradigm, `RadioHandler` is extended by `RadioHandlerFPQR`, `RadioHandlerFRPY`, and `RadioHandlerXYZ`. In particular, the first one sends

to the nano-quadrotors thrust and angular-rate setpoints, leaving in charge the onboard controller to track this trajectory. The second subclass dispatches thrust and angular setpoints, and, finally, the third class assigns position setpoints to the `higher_level_commander` implemented on the Crazyflie firmware A noteworthy fact is that the communication of position setpoints can be performed with a low communication rate (e.g., 10 Hz), which allows for experiments with a significative number of Crazyflie controlled by an online planner.

D. Simulative tools

In this section, the simulative functionalities of CRAZYCHOIR are explained. Initially, we describe the integration of the package with Webots, a realistic simulator tailored for robot simulations. Then, we present an alternative to this method, a numerical simulation visualized in RVIZ.

The toolbox interacts with Webots through a specific set of functions. Specifically, these functions, designed as *plugins*, receive inputs, e.g., control inputs or waypoints coordinates, from the higher-level classes (e.g., controller, planner), mapping them into motor commands. The plugins, written as Python classes, evaluate these command inputs by leveraging Python bindings of the Crazyflie firmware functions. This is a remarkable result of CRAZYCHOIR, in fact, the designer can evaluate the realistic behavior of the quadrotor. Also, the designer could leverage this feature to test new firmware functions.

To implement simulations in the Webots environment, also called *World*, the designer is asked to specify (i) the geometry of the robot, (ii) external features (e.g. sensors and actuators), and (iii) plugins to control the actuators. CRAZYCHOIR is equipped with an exhaustive set of Webots plugins. The parent class is named `MotorCtrl` which makes available GPS measurements and IMU detections, as well as actuators and, possibly, camera images. Also, initializations of ROS 2 publishers and subscribers are provided, which specify the link between Webots and CRAZYCHOIR layers. The extended subclasses are `MotorCtrlFPQR` and `MotorCtrlXYZ`, which leverage functions of Python bindings of the Crazyflie firmware.

To provide the possibility of simulating in additional, external software, CRAZYCHOIR provides a numerical integrator. The Crazyflie dynamics is integrated through an Explicit Runge-Kutta method, performed at 100 Hz. To visualize the quadrotor motion, the toolbox is endowed with a visualization utility based on RVIZ.

Thanks to the modular structure of the package, the user can extend or modify the proposed features to simulate more complex dynamics, such as the effect of rotor drag or aerodynamic disturbances.

IV. CRAZYCHOIR HANDS-ON EXAMPLE: BEARING-BASED FORMATION CONTROL

In this section, a bearing-based distributed formation control scheme (see [6]) is presented as a simulative and experimental example.

Let us consider a team of N Crazyflies, split in N_l leaders and $N - N_l$ followers. The swarm has to deploy a desired formation in the space, and, based on the leaders motion, the controller has to steer the followers, adapting their position. In this distributed framework, each quadrotor can exchange data with a set $\mathcal{N}_i$ of neighboring quadrotors. The desired formation is described by a set of *bearings* $g_{ij}^\star$ for all the couples (i, j) with $i \in \{1, \dots, N\}$ and $j \in \mathcal{N}_i$. According to [6], the control law is based on a double-integrator systems in the form $\ddot{\mathrm{p}}_i^{di} = u_i^{di}$. Specifically, leaders apply $u_i^{di} = 0$, while followers implement the control law

$$u_i^{di} = -\sum_{j \in \mathcal{N}_i} P_{g_{ij}^\star}[k_\mathrm{p}(\mathrm{p}_i - \mathrm{p}_j) + k_\mathrm{v}(\mathrm{v}_i - \mathrm{v}_j)], \qquad (1)$$

where $P_{g_{ij}^\star} = I_3 - g_{ij}^\star g_{ij}^{\star\top}$ and I_3 is the 3×3 identity matrix. Notice that the resulting input cannot be directly sent to Crazyflies, and, hence, we track this acceleration profile using a flatness-based controller (cf. Section III-A).

The software implementation of this example relies on three pillars: the Python class `BearingFormation` for the control law implementation, the extended class `HierarchicalController` to track the acceleration profile, and, then, the simulative or experimental module. The guidance node is implemented as follows:

```
guidance =
    BearingFormation(update_frequency=freq,
                     pose_handler='pubsub',
                     pose_topic='odom')
rclpy.spin(guidance)
```

While, the control strategy is written as:

```
position_ctrl = FlatnessAccelerationCtrl()
attitude_ctrl = GeometryAttitudeCtrl()
sender = FPQRSender()
desired_acceleration = AccelerationTraj()

controller = HierarchicalController(
       pos_strategy=position_ctrl,
       attitude_strategy=attitude_ctrl,
       command_sender=sender,
       traj_handler=desired_acceleration)
rclpy.spin(controller)
```

To this end, for the RVIZ simulation, the user employs the class `CrazyflieIntegrator`, to integrate the dynamics of the Crazyflie, and, hence, the `SimpleVisualizer` to visualize the quadrotor behavior in the RVIZ environment. Instead, to run the Webots simulation, it is solely necessary to substitute the integrator class with the Webots simulation architecture. For this purpose, the user can employ the Webots *world* described by `crazyflie_world.wbt` and the plugin `MotorCtrlFPQR` included in the `crazyflie.urdf` file. An illustrative example of the Webots framework, included in the ROS 2 launch file, is described by the following lines of code:

```
def generate_launch_description():
 # ... init settings

 # launch the Webots world
 webots = WebotsLauncher(world=
     'path_to/crazyflie_world.wbt')
 launch_description.append(webots)
```

```
for i in range(N): # for each quadrotor add
    needed nodes
  # ... set-up guidance node
  # ... set-up control node

  # set-up webots node
  crazyflie_driver = Node(
     package='webots_ros2_driver',
     executable='driver',
     namespace='cf_{}'.format(i),
     additional_env={'WEBOTS_ROBOT_NAME':
         'cf_{}'.format(i)},
     parameters=[{'robot_description':
         robot_description}]
  )
  launch_description.append(crazyflie_driver)

  # ... set-up robot state publisher node
return LaunchDescription(launch_description)
```

Moreover, to highlight the Webots functionalities, we increase the number of Crazyflie to 30, and, hence, we change the desired formation, modeling the bearings to draw a *grid*. Snapshots from the Webots simulation are depicted in Figure 4.

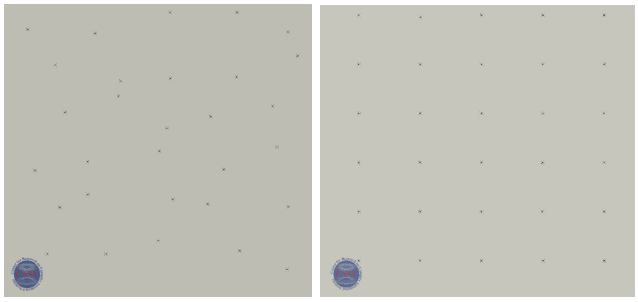

Fig. 4. Begin of the simulation (left) and end of the simulation (right) of the bearing formation control problem in Webots.

A. Running the Experiment

A notable contribution of CRAZYCHOIR is the possibility to execute experiments simply by changing only a few lines of code. Specifically, the user needs to include the radio nodes in the launch file, pointing out the Crazyflie URIs. Then, the `ros2-vicon-receiver` package [1] has to be launched, which allows the communication with the Vicon, and results essential to retrieve the pose of the quadrotors. Snapshots from an experiment [2] with 4 Crazyflies are provided in Figure 5.

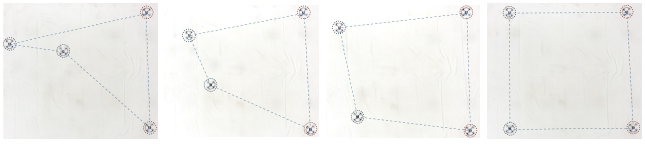

Fig. 5. Snapshots from an experiment at different time instants. Leaders are circled in red, followers in blue.

[1] `https://github.com/OPT4SMART/ros2-vicon-receiver`
[2] The video is also available at `https://youtu.be/mJ1HOquR-vE`

It is worth noticing that, thanks to the modular structure supported by ROS 2, if we lose control of one or more Crazyflies the experiment continues to run. This result enhances the potential of the decentralized framework, increasing the swarm robustness and lowering the risk of failure.

V. CONCLUSIONS

In this abstract, we presented CRAZYCHOIR, a ROS 2 package tailored for swarms of Crazyflie. The toolbox allows the user to perform Webots simulations and experiments of Crazyflie swarms, leveraging firmware bindings. Also, the package provides a set of libraries to communicate via radio dongles with nano-quadrotors. CRAZYCHOIR includes several template classes to perform cooperative decision-making, planning, and control Finally, users can conveniently extend the toolbox by implementing their own algorithms. To assess the potential of CRAZYCHOIR, descriptive simulations and experiments have been provided.

REFERENCES

[1] L. Pichierri, A. Testa, and G. Notarstefano, "Crazychoir: Flying swarms of crazyflie quadrotors in ros 2," *arXiv preprint:2302.00716*, 2023.

[2] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, "Robot operating system 2: Design, architecture, and uses in the wild," *Science Robotics*, vol. 7, no. 66, p. eabm6074, 2022.

[3] O. Michel, "Cyberbotics ltd. webots™: professional mobile robot simulation," *International Journal of Advanced Robotic Systems*, vol. 1, no. 1, p. 5, 2004.

[4] F. Farina, A. Camisa, A. Testa, I. Notarnicola, and G. Notarstefano, "Disropt: a python framework for distributed optimization," *IFAC-PapersOnLine*, vol. 53, no. 2, pp. 2666–2671, 2020.

[5] A. Testa, A. Camisa, and G. Notarstefano, "ChoiRbot: A ROS 2 toolbox for cooperative robotics," *IEEE Robotics and Automation Letters*, vol. 6, no. 2, pp. 2714–2720, 2021.

[6] S. Zhao and D. Zelazo, "Translational and scaling formation maneuver control via a bearing-based approach," *IEEE Transactions on Control of Network Systems*, vol. 4, no. 3, pp. 429–438, 2017.