EPFL

Ecole Polytechnique
Fédérale de Lausanne

Distributed MPC for miniature hovercraft

Semester Project

Author

Andrea Grillo

371099

Professor

JONES Colin Neil

Supervisors

SCHWAN Roland, STOMBERG Gösta

Spring 2024

Automatic Control Laboratory

Contents

1 Abstract

This technical report presents a comprehensive overview of the work completed during the Semester Project at the *EPFL Automatic Control Lab - Predict section.*
The primary objective of this project was to integrate and enhance the work done by two researchers, Roland and Gosta, to achieve formation control of hovercraft. Roland's contributions included the construction of the hovercraft, development and identification of the model, implementation of an LQR controller, and establishment of a pose estimation system and necessary infrastructure. Gosta, on the other hand, developed advanced algorithms for formation control and has already tested them on other robotic architectures.
The ultimate goal was to implement formation control for the hovercraft fleet. Formation control involves coordinated control of multiple robots to follow a predefined trajectory while maintaining a desired spatial pattern. Achieving formation control using Distributed Model Predictive Control (DMPC) presents significant challenges, including the need for substantial computational power and low-latency communication capabilities.
Despite these challenges, successful implementation of well-controlled swarms of land-based and aerial mobile robots holds the potential to revolutionize various sectors, including logistics, transportation, and security.

2 Problem Statement

The objectives of the project included:

1. **Adapting the Software Framework**: Expanding the existing software framework from supporting a single hovercraft to control multiple hovercraft.
2. **Deploying DMPC Algorithms**: Implementing distributed model predictive control (DMPC) algorithms on workstations to control the hovercraft.
3. **Automating Experimental Setup**: Developing automation processes for the experimental setup, including software installation, configuration broadcasting, automatic initiation, and log recollection.
4. **Onboard Algorithm Execution**: Transitioning from workstation-based control to running the DMPC algorithms directly on the hovercraft.

In the following sections we will define the context of the project, describing the existing infrastructure and the new hovercraft developed.

2.1 Infrastructure

This section describes the existing infrastructure and highlights the challenges it poses for system development.
The hovercraft consist of a circular base made of insulation foam, supporting a circular platform with a diameter of 14 cm. This platform is equipped with six brushless motors, enabling movement in any direction on the plane. A Li-PO battery powers both the motors and the on-board electronics. In the existing iteration, the on-board electronics include an ESP32 microcontroller, which primarily communicates with workstations running the control algorithms and relays control signals to the flight controller board that houses the motor drivers (ESCs).

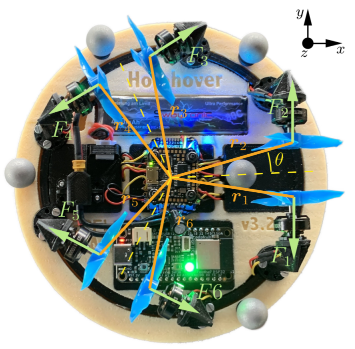


Figure 1: The existing hovercraft

The hovercraft are tracked using an OptiTrack system, which provides real-time position and orientation data. The communication is done on a local network, on which the workstations and the optitrack processing is connected via Ethernet cables, and the hovercraft are connected via Wi-Fi.
The hovercraft are operated on an air hockey table, offering a nearly frictionless surface that facilitates smooth movements.
The complete infrastructure is illustrated in the image below:

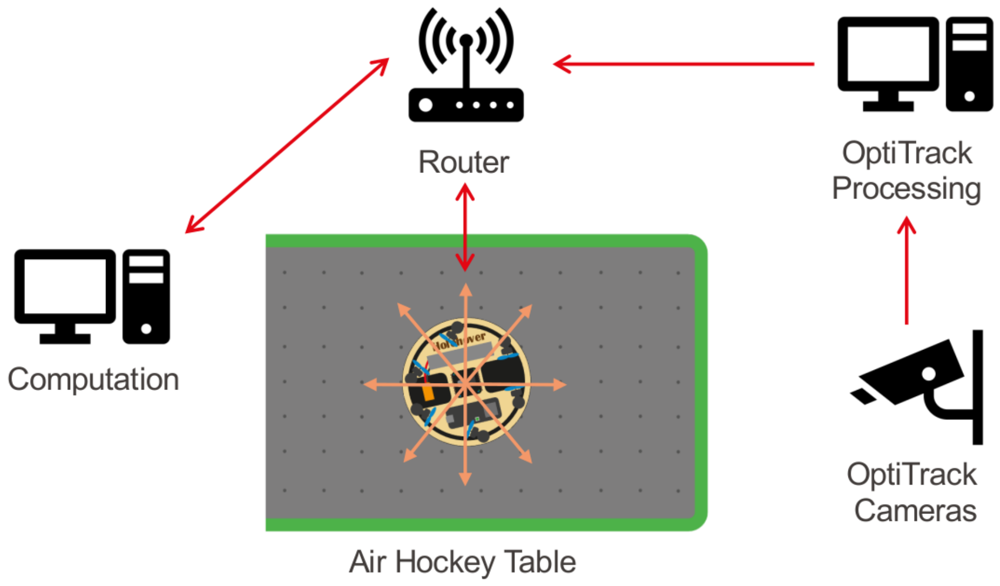


Figure 2: The hardware infrastructure

2.2 New hovercraft

To be able to run DMPC algorithms onboard the hovercraft, a new version of the hovercraft was developed by Roland.
The main idea of the new version was the use of a more powerful controller, switching from an ESP32 to a Radxa Zero 3W.
Switching from a microcontroller to a single-board computer (SBC) allows for switching from Micro-ROS to a full ROS2 stack, enabling the deployment more complex nodes on the hovercraft.
Moreover, a PCB has been developed by Roland, to make the design of the hovercraft tidier, improving the cabling management and the connection between the components and making the system less prone to failures due to electrical problems (e.g. shortcircuits).

3 Distributed MPC

The main featues of Distributed MPC are the following:

- **Decentralization**: The control problem is divided into smaller subproblems, each solved by a different agent. A common goal and objective function is shared between the agents.
- **Coordination and communication**: To solve cooperatively the control problem, the agents need to communicate within each other.

In our application DMPC is used for formation control of the hovercraft.
The formulation of the Optimal Control Problem (OCP) is provided below, starting from the centralized form and to its reformulation in terms of a Non-Linear Program (NLP), together with a brief overview of the distributed optimization algorithm used.

What follows is just an overview, which is not meant to be exhaustive. It has been adapted from the previous work of Gösta [1].

3.1 Centralized formulation of OCP

$$\begin{aligned}
&\min_{\boldsymbol{x},\boldsymbol{u}} \sum_{i\in\mathcal{S}} \sum_{k=0}^{N-1} \ell_i(x^k, u_i^k) + V_{\mathrm{f},i}(x^N) \\
&\text{subject to for all } i \in \mathcal{S} \\
&x_i^{k+1} = f_i^{\mathrm{d}}(x_i^k, u_i^k), && \forall k \in \mathbb{I}_{[0,N-1]}, \\
&(x_i^0, u_i^0) = (x_i(t), u_i(t)), \\
&x_i^k \in \mathbb{X}_i && \forall k \in \mathbb{I}_{[0,N]}, \\
&u_i^k \in \mathbb{U}_i && \forall k \in \mathbb{I}_{[0,N-1]} \\
&(x_i^k, x_j^k) \in \mathbb{X}_{ij}, \quad \forall j \in \mathcal{S}, && \forall k \in \mathbb{I}_{[0,N]}.
\end{aligned}$$

Where the decision variables $\boldsymbol{x}$ and $\boldsymbol{u}$ are the state and input trajectories predicted over the horizon N.
The stage and terminal cost have the following form:

$$\begin{aligned}
\ell_i(x, u_i) &\doteq \sum_{j\in\mathcal{S}} \frac{1}{2}(x_i - \bar{x}_i)^\top Q_{ij}(x_j - \bar{x}_j) + \frac{1}{2}(u_i - \bar{u}_i)^\top R_{ii}(u_i - \bar{u}_i), \\
V_{\mathrm{f},i}(x) &\doteq \frac{1}{2}\sum_{j\in\mathcal{S}}(x_i - \bar{x}_i)^\top P_{ij}(x_j - \bar{x}_j),
\end{aligned}$$

Where the weight matrices Q_{ij}, R_{ij}, P_{ij} are symmetric positive definite.
The sets $\mathbb{X}_i \subseteq \mathbb{R}^{n_{x,i}}$ and $\mathbb{U}_i \subseteq \mathbb{R}^{n_{u,i}}$ constrain the states and inputs of each subsystem.
The coupling is expressed both in the cost function and in the contraints. In our formulation we do not include coupling in the input, but only in the state.

3.2 Reformulation as NLP

$$\begin{aligned}
&\min_{z_i\in\mathbb{R}^{n_i}, i\in\mathcal{S}} \sum_{i\in\mathcal{S}} f_i(z_i) \\
&\text{subject to} \\
&g_i(z_i) = 0 \qquad \forall i \in \mathcal{S}, \\
&h_i(z_i) \le 0 \qquad \forall i \in \mathcal{S}, \\
&\sum_{i\in\mathcal{S}} E_i z_i = 0
\end{aligned}$$

Where the equality constraints include system dynamics and the initial condition. The last constraint expresses the coupling between the subsystems.

3.3 Distributed optimization algorithm

The key concept to implement DMPC lies in the solution of the centralized OCP using *decentralized* optimization methods. That is, algorithms in which each agent solves a subproblem of the OCP, and then communicates only with the neighbor in the communication graph to get to a global solution.

In this project we implemented and deployed two different variants of distributed optimization algorithms, both based on the Alternating Direction Method of Multipliers (ADMM).
The first variant is a purely distributed ADMM algorithm, while the second one is based on Distributed Sequential Quadratic Programming (dSQP). This algorithm has been proposed by Gösta during his previous work [3].
The main difference between the two algorithms is that the former is able to solve only Convex OCPs, while the latter can solve Non-Convex OCPs. This is useful in our application, as it allows to include non-linear constraints in the OCP, such as the minimum distance constraint, which makes the problem a Quadratically Constrained Quadratic Program (QCQP).

Method	**Centralized problem type**	**Problem solved by subsystem**
ADMM	convex QP	convex QP
dSQP	convex QP	convex QP
ADMM	non-convex NLP	non-convex NLP
dSQP	non-convex NLP	convex QP

Table 1: ADMM and dSQP properties.

Both algorithms are based on the same principle, but the dSQP method includes bi-level structure.
The base algorithm is the following:

Algorithm 1 Distributed optimization algorithm

```
1: Initialization: $l = 0$, $l_{\max}$
2: while $l < l_{\max}$ for all $i \in \mathcal{S}$ do
3:     Solve subsystem quadratic program (QP).
4:     Receive copies $\boldsymbol{w}_{ij}^{l+1}$ of trajectories from from all $j \in \mathcal{N}_i^{\text{out}}$.
5:     Average received copies with original trajectories:
           $$\bar{\boldsymbol{x}}_i^{l+1} = \frac{1}{|\mathcal{N}_i^{\text{out}}| + 1} \left( \boldsymbol{x}_i^{l+1} + \sum_{j \in \mathcal{N}_i^{\text{out}}} \boldsymbol{w}_{ij}^{l+1} \right)$$
6:     Send average $\bar{\boldsymbol{x}}_i^{l+1}$ to all $j \in \mathcal{N}_i^{\text{out}}$.
7:     Update dual variables.
8:     $l = l + 1$
9: end while
```

4 Framework

The software stack of the Holohover system is developed on top of ROS2 [4]. At the beginning of the project, a base structure was already developed, allowing for control of a single Hovercraft using a single LQR controller. The components of the original structure were the following:

- **LQR controller** - a controller that computes the control input for the hovercraft using a Linear Quadratic Regulator.
- **Navigation node** - a node for pose estimation using an Extended Kalman Filter.
- **Simulator** - a simple simulator that simulates the hovercraft dynamics using the model of the system.

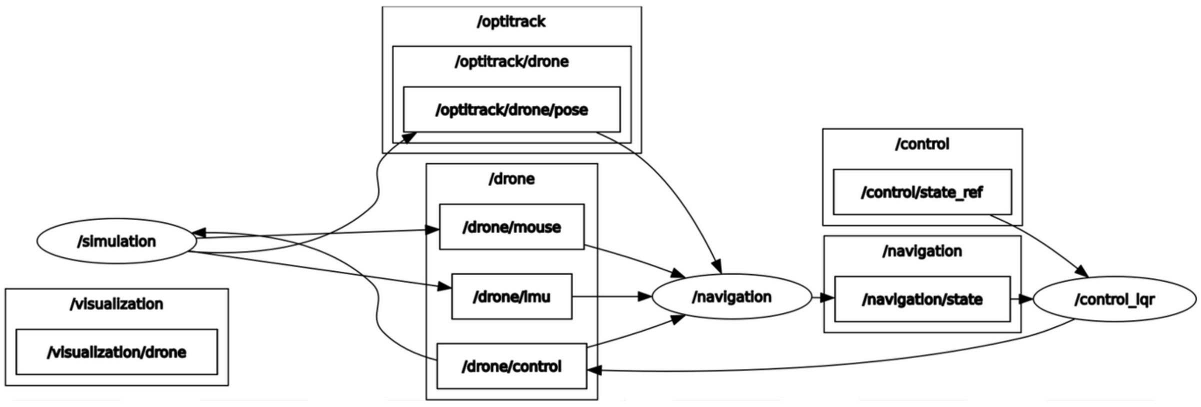


Figure 3: Original ROS2 framework structure

The first step of the project was to extend this structure to allow for control of multiple hovercraft. A new structure was conceived, based on the following:

- **Modularity** - the system should be modular, reusing existing components for each hovercraft, allowing scalability of the experiments with easy addition of new agents.
- **Configurability** - the system should be configurable, allowing for easy modification at runtime of the parameters of the system.

4.1 Simulator

To be able to test the control system before moving to the physical setup, a simulator was developed.
As multiple devices are used in the system, the already existing simulator was not sufficient, as it could not handle physical interaction between multiple hovercraft.

The new simulator has to be able to simulate the dynamics of multiple hovercraft, including impacts between each other and with the wall of the air hockey table.

The simulator was developed using the Box2D library [6].
Box2D is a 2D physics engine that is widely used in the game development industry.
The simulator was developed as a ROS2 node, taking as input the control signals of the controller nodes and giving as output the pose of each body at the next iteration in time.
The features of the simulator include the following:

- **Multiple hovercraft** - simulate the dynamics of multiple hovercraft. The number of hovercraft is configurable at runtime.
- **Physical interactions** - simulate the physical interactions between the hovercraft, including impacts and friction.
- **Table tilt** - simulate the tilt of the table, allowing for testing the robustness of the control system to external disturbances.

Algorithm 2 Simulator Algorithm

```
1: Initialize simulation world and bodies.
2: while true do
3:     Read control signals from controller nodes
4:     Integrate system dynamics and compute acceleration for hovercraft bodies.
5:     Box2D world step.
6:     Retrieve and publish the pose of each hovercraft.
7: end while
```

4.2 Configuration

The system to be developed carries complexities, and each section of it needs to be configurable.
ROS2 provides a parameter system, including a parameter server that allows for the loading of parameters that nodes access at runtime.
Multiple layers of configurations have been conceived:

- **Experiment configuration** - includes configuration that is specific to each experiment:
  - Which hovercraft is active and in which role (dmpc agent / obstacle)
  - Whether each hovercraft is simulated or physical
  - Model parameters to be used for each hovercraft
  - Optimization algorithm
  - OCP parameter file
  - Trajectory file
  - Hovercraft colors for RViz Visualization
  - On which machine runs the controller of each hovercraft
- **Common configuration** - includes configuration that are used either for common nodes or shared between the hovercraft:
  - Simulation parameters
  - LQR parameters
  - Flight controller parameters
  - Navigation node parameters
  - Hovercraft model
  - MOCAP Optitrack parameters
- **DMPC specific configuration** - includes configuration that is specific to the DMPC algorithm:
  - Prediction horizon
  - Sampling interval
  - ADMM ρ penalty parameter
  - Number of ADMM iterations
  - DMPC optimization parameters
  - Maximum waiting time for asynchronous ADMM
  - ...
- **Trajectories** - YAML files that describe trajectories for DMPC nodes and obstacles, read by the trajectory generator node.
  - Time step of the trajectory
  - Configuration of the graph
  - Trajectory points

4.3 Launch files

The system imagined is quite complex, as several different nodes need to be started at the same time, with different configuration and on different machines.
The high level structure of the system to be started is the following:

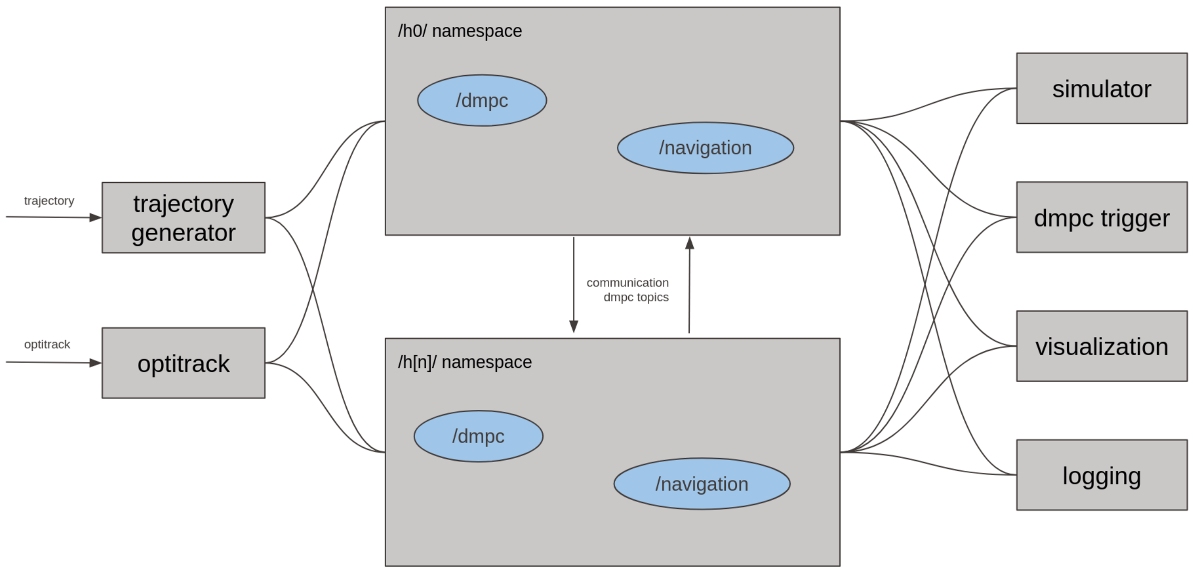


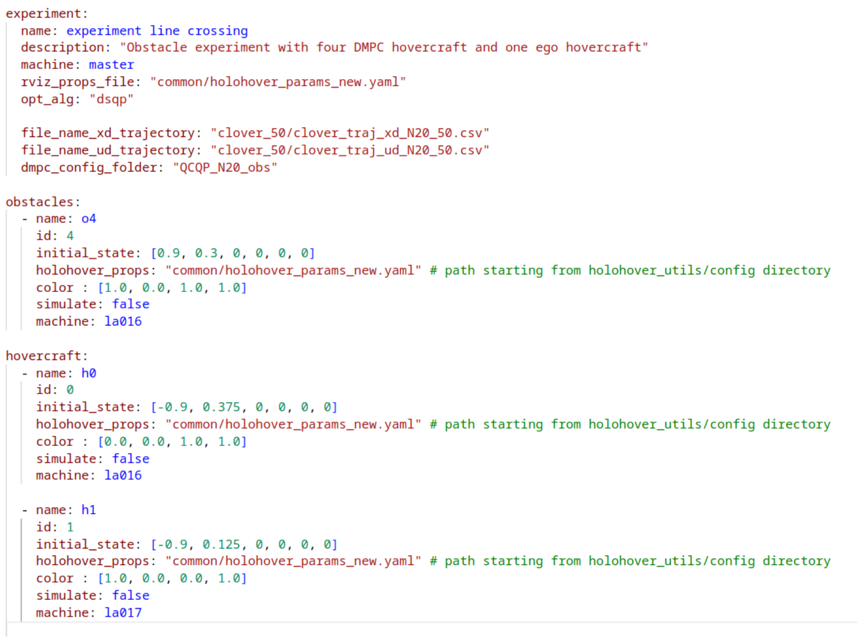
Figure 4: New ROS2 framework structure

4.3.1 Use of namespaces

Each node subscribes and publishes to topics. As we have multiple nodes of the same type but that have to communicate to different topics (e.g. navigation nodes), the launch file made use of namespaces, both for the addressing and mapping of the topic names and for making the graph structure tidier and cleaner.

4.3.2 Launch based on configuration

Starting from ROS2, launch files can be written in Python, which allows to write complex launch sequences. Moreover, a very complex and complete system for managing parameters is provided.
However, the parameters are meant to be read by the nodes and not by the launch files itselves, therefore it is not trivial to use ROS2 parameters to decide which nodes to start.
For this reason, the experiment configuration file has been parsed directly in the Python code, by means of the yaml Python module.



```yaml
experiment:
  name: experiment line crossing
  description: "Obstacle experiment with four DMPC hovercraft and one ego hovercraft"
  machine: master
  rviz_props_file: "common/holohover_params_new.yaml"
  opt_alg: "dsqp"

  file_name_xd_trajectory: "clover_50/clover_traj_xd_N20_50.csv"
  file_name_ud_trajectory: "clover_50/clover_traj_ud_N20_50.csv"
  dmpc_config_folder: "QCQP_N20_obs"

obstacles:
  - name: o4
    id: 4
    initial_state: [0.9, 0.3, 0, 0, 0, 0]
    holohover_props: "common/holohover_params_new.yaml" # path starting from holohover_utils/config directory
    color : [1.0, 0.0, 1.0, 1.0]
    simulate: false
    machine: la016

hovercraft:
  - name: h0
    id: 0
    initial_state: [-0.9, 0.375, 0, 0, 0, 0]
    holohover_props: "common/holohover_params_new.yaml" # path starting from holohover_utils/config directory
    color : [0.0, 0.0, 1.0, 1.0]
    simulate: false
    machine: la016

  - name: h1
    id: 1
    initial_state: [-0.9, 0.125, 0, 0, 0, 0]
    holohover_props: "common/holohover_params_new.yaml" # path starting from holohover_utils/config directory
    color : [1.0, 0.0, 0.0, 1.0]
    simulate: false
    machine: la017
```

Figure 5: Experiment Configuration File

4.3.3 Nodes running on different machines

While ROS1 allowed to set the machine on which a node should run, ROS2 does not provide this feature. Under the hood, the ROS1 *roslaunch* tool to connect via SSH to remote machines to start the nodes, but this feature was not ported to ROS2.
To overcome this limitation, another solution was developed. The idea was to add a parameter to the configuration file that specifies on which machine each node should run, with the following granularity:

- **Common nodes machine** - specifies on which machine the common nodes are run (simulator, Optitrack interface, RViz, RViz interface, trajectory generator).
- **Per hovercraft machine** - specifies on which machine the nodes relatives to each hovercraft are run (DMPC/LQR controller, navigation node).

The same launch file is then run on all the machines, with the same configuration. The launch file takes as a parameter the name of the machine on which it is running, and starts only the nodes that are meant to be run on that machine.

To be able to run experiments on a single machine without having to change the configuration file, an option was added to start all the nodes regardless of the configuration parameters.

4.4 Framework nodes

In order to make the whole system working, both in simulation and with physical hovercraft, several nodes were needed for the integration of the different subsystems and their communication.

4.4.1 Optitrack interface

The Optitrack system published the pose of each rigid body in a world frame that is not relative to our air hockey table, but to the whole lab environment. In the previous version of the system, the pose coming from the Optitrack was directly sent to the Navigation node. The navigation node assumed the first position received as the origin of the local frame and translated all the subsequent messages based on the first.
This was not possible anymore with the exisence of multiple hovercraft; a common frame of reference was needed.
Optitrack markers were added to the table, and an interface node was developed. This node subscribes to the Optitrack pose messages and computes the transformation between the Optitrack frame and the table frame. The transformation consists in both a translation and a rotation, to ensure that the origin of the reference frame is the center of the table, with the X axis aligned with the longer side of the table.
The output of the transformation is then published, and all the navigation nodes subscribe to it.

$$\begin{aligned} x_{\text{table\_frame}} &= x_{\text{optitrack}} - x_{\text{offset}} \\ y_{\text{table\_frame}} &= y_{\text{optitrack}} - y_{\text{offset}} \\ \theta_{\text{table\_frame}} &= \theta_{\text{optitrack}} - \theta_{\text{offset}} \end{aligned}$$

4.4.2 Visualization

To be able to both visualize what is happening during simulation and checking that the system is working as expected during the experiments with the physical hovercraft, the visualization was a necessary and valuable tool to be developed.
The RViz, included in the ROS suite of tools, offers the possibility to easily create visualizations based on markers that are published on specific topics.
The existing version of the visualization node was not able to handle multiple hovercraft. Initially, the first iteration involved just managing the namespace to have a visualization node for each of the hovercraft, that would subscribe to the state and publish the array of markers for that specific hovercraft.
However, the high number and rate of messages sent to the RViz program made it very slow and suffered of high rate of dropped messages, making the visualization of the system not fluid and with very low frame rate.
For this reason, a new version of the visualization node was implemented, in which the visualization node

receives as parameters all the hovercraft that are active in the experiment, and subscribes to the pose of each of them. The node then publishes the visualization markers for all the hovercraft in a single array of markers, in a single message.
The performances of the visualization after this second version and improvements dramatically increased. The visualization node subscribes to the pose of each hovercraft and obstacle and publishes the visualization markers for RViz.

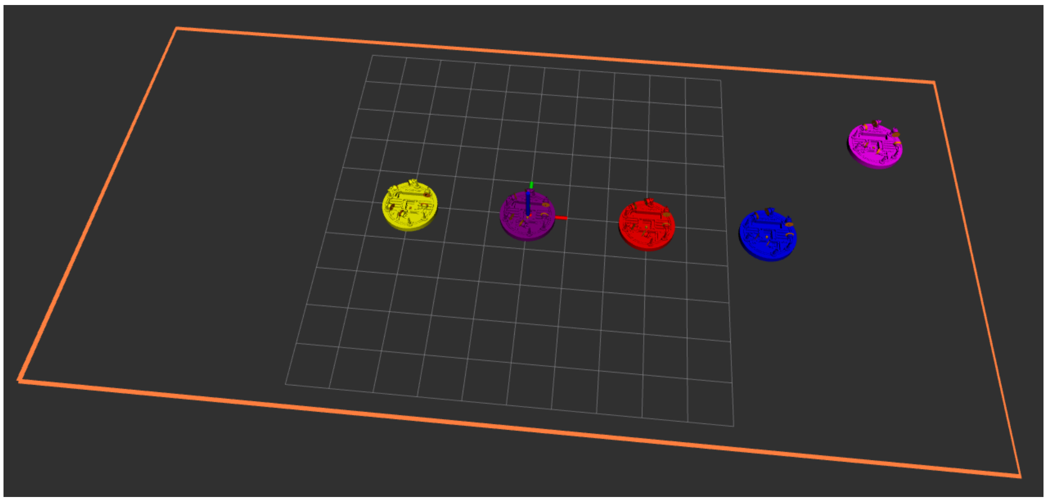


Figure 6: Example of visualization on RViz

4.4.3 Trajectory generation

To be able to send synchronized setpoints to all the agents, a trajectory generation node has been developed. It reads the trajectory files and publishes the trajectory points at the correct time and to the correct topics.
The trajectory generation node is able to read multiple trajectories, for both the DMPC controlled agents and the LQR obstacles.

In most of the trajectories that have been used for experimentation, the setpoint changed only for the first agent, which is the leader that the other agents have to follow. This means that the only setpoint to be changed is the one sent to the first agent. To optimize the number of messages sent over the network, the trajectory generator at each time step checks if the setpoint to be sent for each agent is the same as the one already sent. If this is the case, it does not send the message to that specific agent.
This optimization did not have a big impact for the type of trajectories we tested with this method (mostly point to point transitions), but could be useful in case of complex trajectories with high rate of changes in the setpoint.

In the first iteration of the trajectory generator node, the launch of the node was done manually by the user. After the first round of experiments, it was detected that starting the trajectory generator when the controllers were already started induced a spike in the computational delay and CPU usage of all the nodes, even on other machines. It was discovered that this is a known issue of the ROS2 framework [5], and this it is due to the distributed discovery system of ROS2.
Unlike ROS1 in which a server is used to keep track of all the active nodes and topics (*roscore*), ROS2 is a purely distributed system, so whenever a new node is started, it advertises itself and the published topics to the whole network using broadcast messages, and this whole process of advertising and discovery generated a overhead in the already existing nodes.

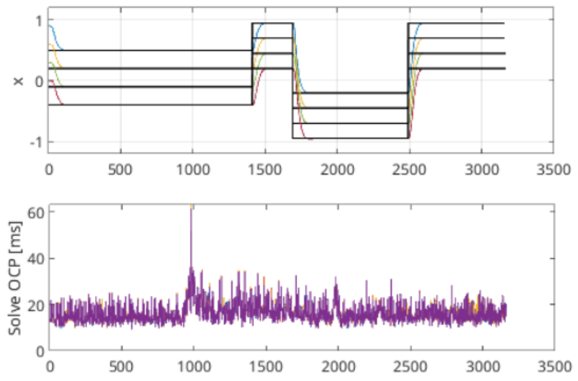
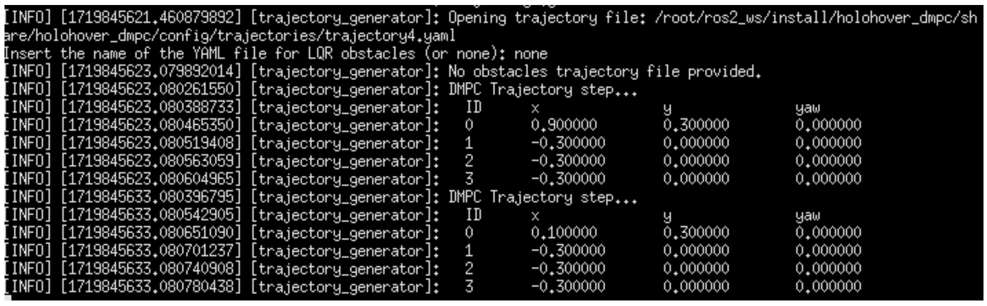


Figure 7: Spike visible in OCP solving time right before the trajectory start.

To overcome this problem, different solution have been tried:

- **Service ROS2 API -** by using the service/client API of ROS2, the trajectory generator node could be started at the beginning with the other nodes, and that would be the time at which the service would be advertised. Then a service call would be issued using the CLI ROS2 interface to start the generation of the trajectory. This solution was discarded as the call of the ROS2 CLI commands generated the same CPU usage spikes and delay as the original solution.
- **ROS agnostic solution -** after the first attempt of the service interface, it was decided to try a ROS agnostic solution, in order not to trigger any ROS action at the beginning of the trajectory generation. The trajectory generator node was started at the beginning of the experiment, and an XTerm window was launched, with a simple interactive interface in which the program asks the user to provide the name of the trajectory file to be launched. Both or just one between the filename of DMPC trajectory and the LQR obstacle can be provided, and then 2 different threads start generating both trajectories.



```
[INFO] [1719845621.460879892] [trajectory_generator]: Opening trajectory file: /root/ros2_ws/install/holohover_dmpc/sh
are/holohover_dmpc/config/trajectories/trajectory4.yaml
Insert the name of the YAML file for LQR obstacles (or none): none
[INFO] [1719845623.079892014] [trajectory_generator]: No obstacles trajectory file provided.
[INFO] [1719845623.080261550] [trajectory_generator]: DMPC Trajectory step...
[INFO] [1719845623.080388733] [trajectory_generator]:   ID      x               y               yaw
[INFO] [1719845623.080465350] [trajectory_generator]:   0       0.900000        0.300000        0.000000
[INFO] [1719845623.080519408] [trajectory_generator]:   1       -0.300000       0.000000        0.000000
[INFO] [1719845623.080563059] [trajectory_generator]:   2       -0.300000       0.000000        0.000000
[INFO] [1719845623.080604965] [trajectory_generator]:   3       -0.300000       0.000000        0.000000
[INFO] [1719845633.080396795] [trajectory_generator]: DMPC Trajectory step...
[INFO] [1719845633.080542905] [trajectory_generator]:   ID      x               y               yaw
[INFO] [1719845633.080651090] [trajectory_generator]:   0       0.100000        0.300000        0.000000
[INFO] [1719845633.080701237] [trajectory_generator]:   1       -0.300000       0.000000        0.000000
[INFO] [1719845633.080740908] [trajectory_generator]:   2       -0.300000       0.000000        0.000000
[INFO] [1719845633.080780438] [trajectory_generator]:   3       -0.300000       0.000000        0.000000
```

Figure 8: Trajectory Generator XTerm interface

4.5 Time synchronization

One of the main challenges of running a distributed algorithm in which the agents have to communicate between each other is time synchronization.
Each time step of the algorithm should ideally be executed at the same time by all the agents, and the communication between the agents should be done in a synchronized way.
As all the agents are running on different machines, said synchronization is not trivial.
During first experiments, we assumed negligible communication latency and no packet dropping. Each agent waited for an other to send messages before continuing with the next iteration.
This solution worked because the nodes were running on two very powerful machines interconnected by means of a wired ethernet connection. However, this cannot happen in a real-world scenario in which the communication latency is not negligible and packet dropping can happen.

4.5.1 Trigger

In order to synchronize the nodes, a trigger mechanism is needed. Ideally the nodes should be able to start the next iteration at the same time, and the communication should be done in a synchronized way.
In the first iteration, the trigger node publishes messages at a specific rate, equal to the DMPC sampling time, to a topic to which all the other nodes subscribe.
This first version presented the following limitations:

- **Need for a centralized coordinator**: the main goal of our project is to develop a distributed control system. The presence of a centralized coordinator, even if its only role would be the timing synchronization, would against this goal.
- **Communication jitter**: ROS2 does not guarantee that the messages are delivered at the exact same time to all the subscribers. There can be delays due to the network, to the processing of the message by the ROS2 stack, to the call to callback function. For this reason this mechanism is not reliable enough for our application.

To overcome these limitations, a new solution was developed.
The first step was ensuring that all the machines in the network were synchronized. This was done using an existing software, called Chrony [9]. Chrony is based on the NTP protocol, which is a widely used protocol for time synchronization in networks. It ensures synchronization of the clock up to a few microseconds.
Then a single trigger message is sent at the beginning of the experiment, to start the controllers. The controllers then save the time at which the trigger is received, and set their timer accordingly. As the time is synchronized, all the subsequent iterations will be started at the same time.

4.5.2 Waiting time and asynchronous ADMM

In the first version of the algorithm, each agent waited for the messages of the other neighbors to arrive before continuing with the algorithm. In our first experiments, this worked as tested the system on very fast machines interconnected via a wired connection, with little to no packet loss and delay. When switching to running the controllers onboard and communicating via wireless, this ideal situation was not the case anymore.
The latency of WiFi communication is not negligible, and so is the jitter, that is the variation over time of the latency.
If a single packet is lost or arrives too late, it is not possible to wait for it, as the agent would be stuck waiting for the message to arrive.
To overcome this problem, a maximum waiting time was set for the messages to arrive. If the messages did not arrive within this time, the agent would continue with the algorithm asynchronously, using the last received messages.

This asynchronous version of the algorithm posed a challenge to the convergence of the control to the optimal solution and the desided position, as we observed that a steady-state error was present whenever this asynchronicity happened. This challenge has been addressed by Gösta in the optimization algorithm, and will not be discussed in depth in this report.

4.6 Docker environment

ROS is quite a complex system to install and maintain. It has strict requirements on the operating system and the libraries that have to be installed.
For this project, ROS was needed on multiple machines. To ease the installation and configuration of the system, a Docker image was developed.
This ensures that the exact same environment is shared across the machines. The system can be modified and updated easily, and then the image shared across the machines.
Online, there are images available with ROS2 Humble installed, that have been used as a base for the development of the image.
All our requirements are included, such as the PIQP optimizer, Casadi and the Box2D library.

4.6.1 Image upload to the remote machines

At the beginning, only the Dockerfile was shared between the machines, and the image was built locally on each machine. This operation is not optimal, as if any change is made to the image, it has to be rebuilt on all the machines and it is difficult to ensure that the image version is consistent on all the machines.
A first iteration of the image transferring was done by building the image on one machine and then exporting it as a tar file. The tar file was then transferred to the other machines and imported.
This solution was not optimal, as for each built the whole image is transferred. Docker images are organized in multiple layers, that can be cached, and it is possible to upload only the modified layers.
Then the tool `Docker Push SSH` [10] was used, that allows to push the image to a remote machine using SSH.
The tools create a temporary registry for each upload and then deletes it at the end of the transfer.
The final solution included the use of a local registry on the LA013 machines, that is kept up and running and is used to store all the built images.
Each machine can connect to the registry and download the image.
The registry on LA013 is running in a Docker container.

4.7 Automatized Launch Scripts

In the initial stages of our project, we faced the challenge of manually starting each process on each machine. This involved individually configuring each machine, launching processes, and collecting logs, which was both time-consuming and error-prone. To streamline and automate these tasks, I developed a series of bash scripts that significantly improved our workflow.

The scripts I developed serve multiple purposes. They connect to each machine via SSH, copy all necessary source code and configuration files to the target machines, start the experiment by launching or connecting to Docker containers, initiate the necessary processes, capture console logs in real-time, and automatically collect and consolidate logs from all machines onto the main machine at the end of the experiment.

To enhance usability and monitoring, during the experiment a dashboard is started, using `tmux`. This dashboard includes console views for each node displayed in separate `tmux` tabs, allowing for real-time monitoring. Additionally, there is a dedicated tab for the master node and another tab to stop the experiment, ensuring a controlled and orderly shutdown of processes.

One of the significant challenges was deviating from the standard practice of running a single application per Docker container. Our requirement was to start different processes at different times within the same container (e.g. flight controller node started before the experiment). This approach was necessary to maintain ROS2 processes within the same container, enabling ROS2 direct communication and avoiding network overhead.

The implementation of these automated scripts brought about several improvements. The scripts ensured that experiments started exactly as intended without manual intervention, significantly reducing the time and effort required by automating the configuration, process initiation, and log collection. Moreover, all logs were systematically organized on one machine, simplifying the analysis process. These sets of scripts were crucial in enhancing our operational efficiency and reliability, allowing us to focus more on the experimental results rather than the setup process.

5 Embedded DMPC

The ultimate goal of the project was to be able to run the DMPC algorithms onboard, without any external workstation doing the computation.
The main idea that allowed the possibility of going onboard was the use of a SBC (single board computer) instead of a MCU.
The existent version of the hovercraft had an ESP32 MCU onboard, and porting the software to the ESP32 would have been a big challenge of optimization. The ROS2 stack cannot run on a microcontroller, so a complete refactor of the code was needed. The DMPC algorithms are also quite computationally intensive, and most likely the MCU would not have been enough for the task.
The hovercraft architecture pose a challenge in terms of weight and size of the components, as it has to

be as light possible to be able to float on the air hockey air cushion.

This goal has been achieved in several steps:

5.1 Redesign of the hovercraft

To accomodate the new board, the entire hovercraft had to be redesigned. The first SBC taken into consideration is the Raspberry Pi model Zero 2W, that has a form factor and weight compatible with the hovercraft constraints. Then the final choice was for the Radxa Zero 3W, that offers more computational power at the same form factor. The only downside of using the Radxa is the lack of documentation and software support.
For the new hovercraft, Roland has designed a PCB that serves both as a structural skeleton of the hovercraft and to integrate all the components and the electrical connections.

5.2 Implementation of the drivers

The Radxa has to communicate with the Betaflight firmware on flight controller to control the motors and to collect datas from the IMU and the battery state. Moreover, it has to communicate with the mouse sensors. Roland has developed two nodes that address those needs.

One feature that was not present in the previous version of the hovercraft is the monitoring of the battery level.
To ensure that the batteries are not ruined by overdischarge, a check on the voltage is executed periodically. If the voltage drops under a 7.5 volts, a warning message is printed to the user. If the voltage drops under the critical voltage of 7.4 volts (nominal voltage of a 2S Li-Po battery), the motors are shutdown automatically.

5.3 Adaptation of the Docker image

The Radxa boards have a different CPU architecture (ARM64) with respect to the workstations (X86-AMD64). Moreover, the compilation of the needed packages cannot be done onboard, as the RAM memory is not sufficient (1GB). Another solution was needed for the compilation.
Docker provides the possibility of cross-compiling for other architectures, by adding a level of virtualization between the kernel and the container.
A new Docker image has been developed, using as a base smaller image which does not include the graphical ROS2 tools such as RViz and RQT.
During the building of the image, the ROS2 workspace is built, in order to have the compiled binaries ready together with the image produced.
The downside of this approach, is that at each build the whole workspace is compiled from scratch. As the compilation is not native but it is a cross-compilation, it is slow and it takes long. To optimize this, the building of the image has been split in two different steps:

1. **Base image**: a base image is built compiling the whole workspace as previously stated.
2. **Final image**: the final image starts from the base image and compiles only selected packages, that are the ones under developments.

This optimization drastically reduced the downtime between experiments, bringing the building time from around 20 minutes to around 4 for the compilation of the `holohover_dmpc` package.

5.4 Optimization of the DMPC controllers

communication optimization latency analysis concurrency optimization

5.5 Analysis of the network usage

With the objective of optimizing all the network traffic over WiFi, the Wireshark network sniffer has been used to check all the packets that are sent between the agents. This thorough analysis has brought to attention different aspects that were not optimal:

- **Rosbag**: having a rosbag recorder active recording all the topics on the main machine means that all the messages of every topic has to be sent to the main machine. This adds a conspicuous network traffic.
  To overcome this, multiple rosbag recorders are started, one on each machine, recording just the topics generated by the nodes on that machine. The rosbag files are then merged at the end of the experiment.
- **Rosout topic**: by default, all ROS2 nodes publish their logs on the `/rosout` topic. This topic is then sent to the main machine, generating a high network traffic. There is an option to disable the logging to said topic.
- **ROS2 discovery messages**: ROS2 uses a discovery mechanism to find all the nodes and topics in the network. This mechanism generates a high network traffic, as all the nodes have to advertise themselves and the topics they publish. To overcome this, a *discovery server* can be started, that keeps track of all the nodes and topics in the network. The nodes then query the server to get the list of the nodes and topics.
- **Visualization**: to have the visualization system working, the `rviz_interface_node` has to subscribe to the pose and the control of each hovercraft, thus generating network traffic. Once the visualization is was not needed anymore because we ensured that everything was working correctly, we stopped the node, reducing the network traffic.

6 Experiments

Thanks to the simulator that has been developed at the early stage of the project, we were able to test and validate the functionality of the system from the beginning.
However, the simulator is based on the hovercraft model, which is not perfect and does not perfectly adhere to the phyisical device. This also means that in the simulations we had no model-mismatch.

The first round of experiments on physical hardware have been conducted with the old hovercraft, running the controllers offboard on the workstations.
The goal of those experiments was validating that the system works and is able to effectively control the hovercraft, therefore the experimental scenarios were easy and with the only objective of testing the system.
After the first round of experiments, the last iteration of the hovercraft has been designed and produced, which allowed the deployment of the controllers onboard.
This new setting needed a first phase of optimization and tuning, as described in the previous chapter using WiFi communication on less powerful hardware posed a few challenges.
Once the setting was tested and validated, the experiments were repeated with the new hovercraft.
The purposes of these new experiments were the following:

- Test more challenging scenarios.
- Compare onboard and offboard control.
- Test interaction with LQR controlled obstacles.

Below a summary of the experiments conducted is provided.

- **Point to point transition**: by giving step setpoints to the hovercraft, the ability of the system to follow the trajectory was tested. More challenging tasks included an abrupt change of setpoint while the hovercraft where moving, and another scenario in which the hovecraft needed to change their ordering, to test the obstacle avoidance constraint.
- **Line crossing**: Scenario in which 4 DMPC controlled hovercraft move from one side of the air hockey table to the other, avoiding a LQR controlled obstacle. The experiment was tested with a static obstacle and with the obstacle moving in circle, with 2 different radii.
- **Trajectory following**: A trajectory has been designed and the hovercraft were tested to follow it. The trajectory was a three-leaved clover. This trajectory has been tested with and without the presence of an obstacle. The obstacle entered the field of the trajectory during the experiment, to test the obstacle avoidance capabilities.

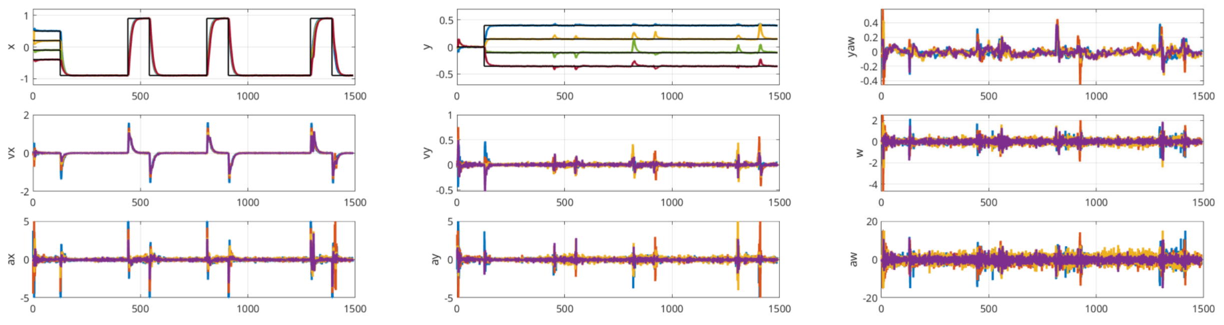
All the aforementioned scenarios have been tested with the following 3 configurations:

- **Onboard control** with 6 ADMM iterations for each MPC step, $0.15s$ sampling interval, horizon length of 7 steps.
- **Offboard control** with the same configuration as the onboard controller.
- **Offboard control** with 30 ADMM iterations for each MPC step and a sampling time of $0.05s$, horizon length of 20 steps.

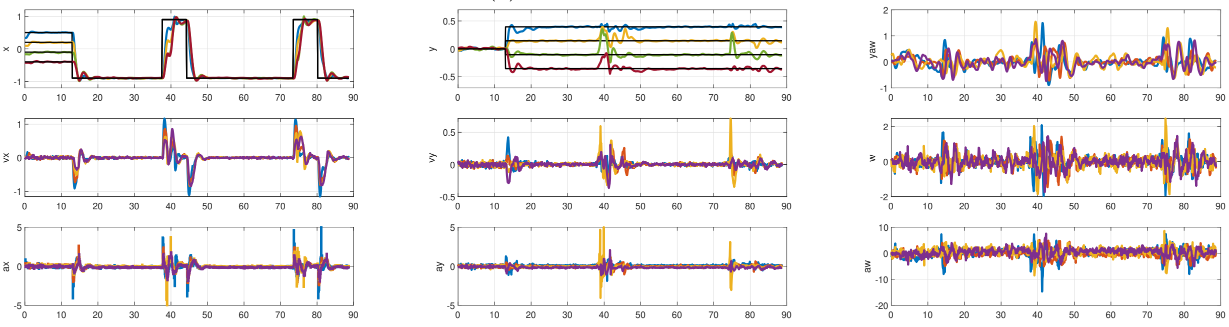
During all the experiments, we recorded the following data:

- Position of all the hovercraft at each time step.
- Position of the obstacle at each time step.
- MPC predicted trajectories.
- Time measurements of the various sections of the algorithms (QP solving, communication, etc.).

This data allows to compare the performances due to different configuration of the algorithms (n of ADMM iterations, horizon lenght, sampling time), as well as the difference of running the same controller onboard and offboard.
The whole dataset that has been collected will be analyzed and used to writing of a paper to be presented to the ICRA Conference.
Nonetheless, a preliminary comparison is presented in the following graphs.



(a) Offboard controllers



(b) Onboard controllers

Figure 9: Line crossing experiment

In the pictures the closed loop trajectories of the hovercraft are shown for the line crossing experiment, for both onboard and offboard computation.
The difference in performance is evident, as the offboard controllers are able to follow the trajectory with less oscillations.
This is due to both the sampling time and the number of ADMM iteration, the hovercraft is a system with fast dynamics, so having a sampling time of $0.05s$ allows for a better control of the system.

7 Results

The primary outcome of this project is the successful implementation of a Distributed Model Predictive Control (DMPC) system for controlling hovercrafts.
Moreover, the transitioning to onboard controllers has been successfully completed, and represents a big advancement.

This system has been tested and validated across different scenarios, including point-to-point transitions, line crossing, and trajectory following, demonstrating its robustness and versatility.

The experiments generated a significant amount of data on hovercraft positions, trajectories, and algorithm performance metrics. This data provides a valuable resource for further analysis and future research, enabling continued refinement and enhancement of the control system.

8 Future work

A survey on swarm robotics [8] (for micro aerial vehicles, but the outcomes are applicable to our platform), showed that the main challenges in the scalability of the number of agents is the control and the positioning.

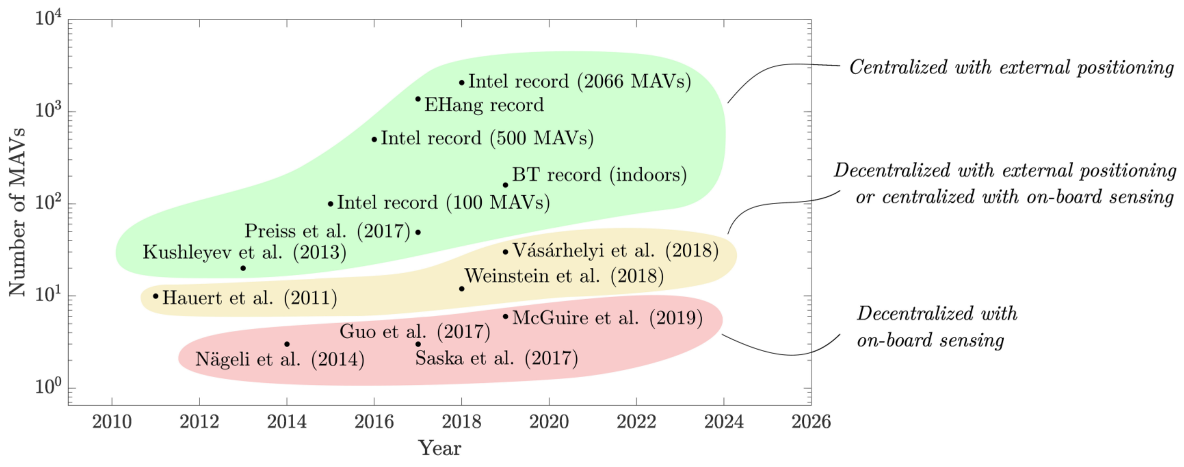


Figure 10: Number of MAVs with respect to control and localization system [8].

The present system lies in the yellow region, as the control is decentralized and the positioning is external.

Our control solution allows for scalability of the decentralized control, as the size of the problem to be solved does not depend on the size of the whole swarm, but only on the size of the neighborhood of each agent.

However, the bottleneck of our current system is the communication. Even though the messages are directly exchanged between the agents, currently a standard WiFi network is used, in which all the agents are connected to the same access point.

Moreover, the positioning is done by the Optitrack system, which is an external system that provides the position of the agents. This system is not scalable, as the number of agents that can be tracked is limited.

Future work will focus on improving the current system and addressing these scalability limitations, particularly by enhancing communication infrastructure and developing more scalable positioning solutions.

8.1 Communication infrastructure

Investigation has to be done to improve our actual communication infrastructure, that is the main bottleneck of our system.

The two crucial requirements of our communication are:

- Low latency communication. At present, most of the CPU time of the agent is spent for the communication - mostly waiting for messages to be delivered.
- Decentralized network. If detached from the necessity of a central device (access point), the scalability of the system would be greatly improved. As there would be many different peer to peer direct connections, one main challenge of this system is to avoid interference between the different connections.

A more thorough analysis of the state of the art in this direction would be necessary, as there seem to exist some solutions [7] that could be adapted to our system.

8.2 Control algorithms

8.2.1 Obstacle modeling

In the current implementation of the OCPs, the obstacles are modeled as a static constraints. The agent is aware of the position of the obstacle at the sampling time, and it assumes it will remain in the same position for the whole prediction horizon.
This assumption is not realistic: in simulations with dynamic obstacles, especially when using a bigger sampling interval and low number of optimization iterations, the performances of the collision avoidance are greatly reduced, leading to impacts with the obstacle.

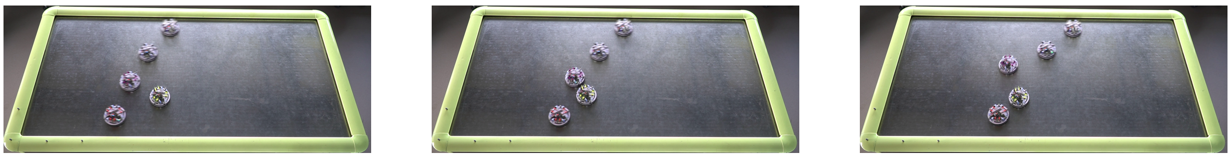


Figure 11: An impact during obstacle avoidance.

In the first picture we can see the for hovercraft in formation moving from left to right. In the second picture the obstacle is being hit by the third hovercraft (from top). In the last picture, we can see that the trajectory has been severely affected by the impact.

By modeling the dynamics of the obstacle with a simple model, such as a zero acceleration or zero jerk model, the agent could have an uncertain prediction of the future position of the obstacle in the horizon and set the constraint accordingly.

8.2.2 Graph topology

In the DMPC algorithms, the structure of the swarm can be described as a graph. The agents are the nodes of the graph, and the edges are the connections between the agents. The neighborhood of an agent is the set of agents that are connected to it. In the present system, we have a very simple line-type graph, in which each agent is connected to the previous and next agent.

Different types of graph topology could be explored. Each of the topologies would have different impacts on the control performances, that could be analyzed and experimented.

8.2.3 Flexibility

At present, the OCPs are designed for very specific and well defined scenarios.
The number of agents and the graphs structure is fixed, as well as the cost function weights and the constraints.
One major improvement of this would be able to adapt the control problem at runtime to different scenarios without having to redesign a new OCP. Being able to handle a different number of agents, as well as having configurable weights and constraints would make the system much more flexible and adaptable to different scenarios.

For instance, the minimum distance constraint could even be modified during execution, to be able to handle challenging situation such as a narrow passage in which the maximum speed would be reduced.

8.3 Real-time constraints

In the current setup we cannot enforce real-time constraints. Even by setting the maximum waiting time for the messages, the system is not real-time, as the operating system we are using (*Ubuntu 22.04*) is using the default Linux scheduler, which is not real-time.
Further investigations would involve the use of the modified Linux kernel with the PREEMPT-RT patch. This is not trivial, as it is necessary to ensure that all the drivers and the software used are compatible with the real-time kernel.
The main challenge would be making sure that the driver for the WiFi chip is capable of enforcing real-time constraints, as the WiFi communication is the main time bottleneck of our system.

8.4 Hardware revision

During this semester a PCB has been developed, to make the design of the hovercraft tidier, improving the cabling management and the connection between the components.
A new PCB could leverage those improvements and add new features to the hovercraft, such as the onboard charging of the batteries and a simpler interface to the hovercraft.
A draft of the specifications for the new PBC could be as follow:

- USB-C PD voltage negotiation (possible chip - Infineon CYPD3177)
- Onboard LiPo battery charger (possible chip - Microchip BQ25703A)
- LED indicators for battery level and charging status.
- LEDs connected to Radxa's GPIO for showing the state of the system.
- Buttons connected to Radxa's GPIO for user interaction (e.g. graceful shutdown of the Radxa, start of the flight controller).
- Battery temperature sensor for safety of the battery charging.
- Motor kill-switch for safety purposes.

8.5 Improvement of onboard sensing

To enhance the system's adaptability to real-world applications, the pose estimation that is now solely based on data coming from the Optitrack Motion Capture system could integrate the IMU and mouse sensor data. This data would then be fused by means of the Extended Kalman Filter.
As both sensors are based do not give absolute positioning, the Optitrack system would still be necessary to correct the drift introduced by integration. However, the rate at which the Optitrack data is used could be significantly reduced, reducing the load of the WiFi network.
Moreover, by reducing dependence on the OptiTrack system and emphasizing onboard sensors, the system would be more similar to a real-world scenario (e.g. low rate GPS RTK positioning).

8.6 Native Build on ARM64 architecture

At present, the build process for the onboard software is carried out using cross-compilation on a standard x86 PC for the Radxa's AMD64 architecture. This method introduces a level of hardware emulation, which slows down the build process.
To address this, the build process could be significantly accelerated by performing a native build directly on an ARM PC. ARM machines are becoming increasingly prevalent in the market, though they are typically low-power devices. However, an exception to this trend is the Apple Silicon chips, which offer high performance and efficiency.
By leveraging these high-performance ARM chips, we could eliminate the need for cross-compilation and hardware emulation, thus streamlining the build process.

9 Conclusion

The presented achievements made this semester project really successful. It has been a very nice opportunity to learn a lot and to build a system that is really interesting. Running onboard distributed DMPC on such dynamic systems as hovercrafts is a very challenging task, that was never achieved before. The system has a very high potential and there is a lot of room for improvement and future experimentation. I would be eager to continue working on this project.

10 Acknowledgements

I would like to express my gratitude to the supervisors who played a key role to the success of this project.

I extend my deepest thanks to Roland for his invaluable guidance and his superior knowledge of C++. His willingness to answer questions at any hour, day or night, has been immensely helpful. Remember, no raw pointers!

I am also grateful to Gösta for providing his algorithms for this project. To the endless hours staring at the plots, that we will print and hang on our walls, and to the "magic script"!

Finally, I would like to thank Professor Colin Jones for giving me this incredible opportunity. I look forward to continuing our collaboration in the future.

11 References

References

[1] Stomberg, G., Ebel, H., Timm Faulwasser and Eberhard, P. (2023). Cooperative distributed MPC via decentralized real-time optimization: Implementation results for robot formations. *Control engineering practice, 138, pp.105579–105579.* doi: 10.1016/j.conengprac.2023.105579

[2] Schwan, R., Schmid, N., Chassaing, E., Samaha, K. and Jones, C.N. (2024). On identifying the non-linear dynamics of a hovercraft using an end-to-end deep learning approach. doi: 10.48550/arXiv.2405.09405

[3] Stomberg, G., Engelmann, A. and Timm Faulwasser (2022). Decentralized non-convex optimization via bi-level SQP and ADMM. *2022 IEEE 61st Conference on Decision and Control (CDC)* doi: 10.1109/CDC51059.2022.9992379

[4] ROS2 - Robot Operating System `https://index.ros.org/doc/ros2/`

[5] CPU spikes of existing nodes when starting new node - ROS2 Github Issue #741 `https://github.com/ros2/rmw_fastrtps/issues/741`

[6] Box2D - A 2D Physics Engine for Games `https://box2d.org/`

[7] Dmitry Bankov, Khorov, E.M., Lyakhov, A.I. and Sandal, M. (2019). Enabling real-time applications in Wi-Fi networks. *International Journal of Distributed Sensor Networks, 15(5), p.155014771984531-155014771984531* doi: 10.1177/1550147719845312

[8] Coppola, M., McGuire, K.N., De Wagter, C. and de Croon, G.C.H.E. (2020). A Survey on Swarming With Micro Air Vehicles: Fundamental Challenges and Constraints. *Frontiers in Robotics and AI, 7* doi: 10.3389/frobt.2020.00018

[9] Chrony: an implementation of the NTP protocol. `https://chrony-project.org/`

[10] Docker Push SSH: a command line utility to push docker images to remote machine via SSH `https://github.com/brthor/docker-push-ssh`

Note: All the links included in the Reference have been accessed on July 4, 2024.

12 Appendix

12.1 Abbreviations

• **ADMM**	Alternating Direction Method of Multipliers
• **API**	Application Programming Interface
• **CLI**	Command Line Interface
• **CPU**	Central Processing Unit
• **DMPC**	Distributed Model Predictive Control
• **EKF**	Extended Kalman Filter
• **ESC**	Electronic Speed Controller
• **GPS**	Global Positioning System
• **IMU**	Inertial Measurement Unit
• **LAN**	Local Area Network
• **Li-Po**	Lithium Polymer Battery
• **LQR**	Linear Quadratic Regulator
• **MOCAP**	Motion Capture
• **MPC**	Model Predictive Control
• **OCP**	Optimal Control Problem
• **PCB**	Printed Circuit Board
• **QCQP**	Quadratically Constrained Quadratic Programming
• **QP**	Quadratic Programming
• **RAM**	Random Access Memory
• **ROS**	Robot Operating System
• **RTK**	Real Time Kinematic
• **SBC**	Single Board Computer
• **SQP**	Sequential Quadratic Programming

12.2 IP and login configuration of the machines

- LA013
  - **ip:** 192.168.0.70
  - **user:** andrea
  - **password:** grillo;2024
- LA016
  - **ip:** 192.168.0.71
  - **user:** ubuntu
  - **password:** ubuntu;2024
- LA017
  - **ip:** 192.168.0.72
  - **user:** ubuntu
  - **password:** ubuntu;2024
- RADXA-H0 (White)
  - **ip:** 192.168.0.131
  - **user:** ubuntu
  - **password:** ubuntu;2024
- RADXA-H1 (Black)
  - **ip:** 192.168.0.122
  - **user:** ubuntu
  - **password:** ubuntu;2024
- RADXA-H2 (Purple)
  - **ip:** 192.168.0.136
  - **user:** ubuntu
  - **password:** ubuntu;2024
- RADXA-H3 (Red)
  - **ip:** 192.168.0.107
  - **user:** ubuntu
  - **password:** ubuntu;2024
- RADXA-H4 (Yellow)
  - **ip:** 192.168.0.108
  - **user:** ubuntu
  - **password:** ubuntu;2024

12.3 Procedures

12.3.1 Experiment procedure

Connecting to the Main Machine (LA013)

1. Make sure you are in the EPFL network
2. SSH into the machine using

   ```
   ssh andrea@128.178.5.134
   ```

   where the IP is the public address of the router of the lab, on which there is set the port forwarding to reach LA013.
   Use the `-X` (`-Y` on MacOS) to enable X forwarding for GUI applications.
3. You can then use this machine to SSH into other machines on the local network

Running Experiments

1. Turn on the hovercraft.
2. Make sure that in the `~/holohover-docker/remote-launch/config.sh` the right configuration (onboard/offboard).
3. Build the image. In `~/holohover-docker` directory
   - PC: `./build.sh pc`
   - RADXAS: `./build.sh pi` and then `./update.sh`. If you just modified the `dmpc` package you can just run the update script. Make sure to run the update script after each complete build. The packages that are built by the update script are configured in a variable at the beginning of the script.
4. Run the `upload-images.sh` script - make sure that the `config.sh` is right.
5. To start the FC nodes on the hovercraft, run the `launchFC.sh` script. It will give warnings when the battery goes under 7.5V, and shut off the motors at 7.4V (thresholds configurable in the driver's config file).
6. Run the container on LA013 (`./run.sh`) and run the command `ros2 launch holohover_utils hw_env.launch.py experiment:=XXX`
7. Run the experiment: `./experiment.sh XXXXX.yaml` where XXXXX is the name of the experiment config file. This script will do the following:
   (a) Copy the configuration files (`holohover_utils/config` and `holohover_dmpc/config`) from LA013 to all the remote machines
   (b) Start the docker containers locally and on the remote machines
   (c) Start all the processes in the docker container
   (d) Start a `tmux` environment in which in the first panel the console of the local Docker container is shown, in the other 4 it automatically connects to the remote machines and shows the `stdout` of the nodes. In the lower right panel, the command to stop the experiment is already written. To move between `tmux` panels, use `ctrl-b + arrow`
8. To stop the experiment, inside the `tmux` environment run the `stop_experiment.sh` script (lower right panel). This script will do the following:
   (a) Stop all the remote Docker containers
   (b) Do the collection of the logs of the main and remote machines inside `~/holohover-docker/log` directory
   (c) Stop the `tmux` session
9. Keep checking the `launchFC.sh` script, it will give warnings for the battery levels. To shut the hovercraft off, run the `shutdownAll.sh` command.

Compile New `locFuns.so` Files

- The `locFuns.so` file has to be built for both the PC architecture and for the RADXA architecture. To build for the PC, just `cd` to `~/holohover-docker/ws/src/holohover/holohover_dmpc/ocp_specs` and run `./compileAll.sh x86` script.
- To build for the RADXA, go to `~/holohover-docker` and run `./run.sh holohover-light-aa` to run the virtualized image of the RADXA.
  `cd` to `/root/ros2_ws/src/holohover/holohover_dmpc/ocp_specs` and run `./compileAll.sh arm64`. You can then exit from the container.

Syncing the Files
At the beginning of each experiment, all the source code is synced to all the remote machines. This means that any change in the config files is copied to the remote machines. However, if a new config file is added, it won't be seen by `ros` as the new file does not have a symlink in the share directory. If new files are added, the new image has to be built. To run the synchronization without actually running an experiment, you can run the `rsync.sh` script in the `~/holohover-docker/remote-launch` directory.

12.3.2 Radxa Installation procedure

SD card flashing and first boot
Download the ubuntu-22.04-preinstalled-server-arm64-radxa-zero3.img.xz image and flash it onto a SD card using something like balenaEtcher or the dd linux utility.
Mount the just flashed SD card and configure your wifi (`network-config`) accordingly along the lines of

```
network:
  version: 2
  ethernets:
    zz-all-en:
      match:
        name: "en*"
      dhcp4: true
      optional: true
    zz-all-eth:
      match:
        name: "eth*"
      dhcp4: true
      optional: true

  wifis:
    wlan0:
      dhcp4: true
      optional: true
      access-points:
        "TP-Link_08B0_5G":
          password: "78122987"
```

Insert the SD card into the Zero 3 and let it boot. After some time (give it a couple minutes) you should be able to see a new device on the network. Connect to it using

```
ssh ubuntu@192.168.0.xxx
```

with password `ubuntu`. You will be prompted to change it, so do so.

Docker installation
Run the following commands to install some additional libraries and Docker:

```
sudo apt update
sudo apt install gpiod ca-certificates curl
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg \
          -o /etc/apt/keyrings/docker.asc
sudo chmod a+r /etc/apt/keyrings/docker.asc

echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc] \
  https://download.docker.com/linux/ubuntu \
  $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update

sudo apt-get install docker-ce docker-ce-cli containerd.io docker-compose-plugin

sudo groupadd docker
sudo usermod -aG docker $USER
```

Add the following to `/etc/docker/daemon.json`

```
{
    "insecure-registries" : [ "192.168.0.70:5000" ]
}
```

Chrony configuration
To configure chrony for time synchronization:

```
sudo apt install chrony
```

Then edit the configuration file `/etc/chrony/chrony.conf`. Comment out the following lines:

```
pool ntp.ubuntu.com        iburst maxsources 4
pool 0.ubuntu.pool.ntp.org iburst maxsources 1
pool 1.ubuntu.pool.ntp.org iburst maxsources 1
pool 2.ubuntu.pool.ntp.org iburst maxsources 2
```

And add the following line:

```
server 192.168.0.70 iburst
```

Restart chrony with `sudo systemctl restart chronyd`
Then you can check if the source is right (should be 192.168.0.70) with `chronyc sources`, and if the tracking is working with `chronyc tracking`.

Docker image and Holohover source code
Now we will configure the `holohover-docker` repository and the `holohover` source code. Clone the `holohover-docker` repository:

```
git clone https://github.com/grilloandrea6/holohover-docker.git
```

Add the new machine to the config file in la013 (`/holohover-docker/remote-launch/config.sh`). Run the `upload-image.sh` script to transfer the image to the Radxa and the `rsync.sh` script to transfer the holohover source code.

SPI and UART configuration
We are going to use a custom spi driver.
To build and install the driver `cd` into the
`holohover/holohover_drivers/kernel_modules`
folder and run the following commands:

```
make
sudo cp spideve.ko /lib/modules/$(uname -r)/kernel/drivers/spi/spideve.ko
sudo depmod
```

Edit `/etc/modules` and add a new line with `spideve`.
To get the connected uart and spi peripherals working we need to modify the device tree. Go into the folder `/usr/lib/firmware/$(uname -r)/device-tree/rockchip/overlay` and create a new file `radxa-zero3-uart2-m0-spi3-`
with following content:

```
/dts-v1/;
/plugin/;

/ {
metadata {
title = "Enable UART2 M0 and SPI3 M1 CS0";
compatible = "radxa,zero3";
category = "misc";
description = "Enable UART2 M0 and SPI3 M1 CS0 on Zero 3W.";
};

fragment@0 {
target = <&uart2>;

__overlay__ {
status = "okay";
pinctrl-0 = <&uart2m0_xfer>;
};
};

    fragment@1 {
target = <&fiq_debugger>;

__overlay__ {
status = "disabled";
};
};

    fragment@2 {
target = <&spi3>;

__overlay__ {
status = "okay";
#address-cells = <1>;
#size-cells = <0>;
num-cs = <1>; pinctrl-names = "default", "high_speed";
pinctrl-0 = <&spi3m1_cs0 &spi3m1_pins>;
pinctrl-1 = <&spi3m1_cs0 &spi3m1_pins_hs>;
max-freq = <2000000>;

spideve@0 {
compatible = "rockchip,spideve";
status = "okay";
reg = <0>;
};
};
};
};
```

Run the following command to compile the `dts`

```
sudo dtc -@ -O dtb -o radxa-zero3-uart2-m0-spi3-m1-cs0.dtbo radxa-zero3-uart2-m0-spi3-m1-cs0.dts
```

Edit the file `/etc/default/u-boot` and modify the variable `U_BOOT_FDT_OVERLAYS` as

```
U_BOOT_FDT_OVERLAYS="device-tree/rockchip/overlay/radxa-zero3-uart2-m0-spi3-m1-cs0.dtbo"
```

To not have have the system console printed out on the uart2 port, modify the content of `/etc/kernel/cmdline` to the following:

```
rootwait rw console=tty1 cgroup_enable=cpuset cgroup_memory=1 cgroup_enable=memory
```

After saving the config files, use the `u-boot-update` utility to apply it:

```
sudo u-boot-update
```

Reboot the system with

```
sudo reboot
```

You should now see two new devices: `/dev/ttyS2` and `/dev/spideve3.0`