Hoplite: Coordinated LIMO Robot Swarm Control via Gamepad and OptiTrack MoCap with ROS Integration

Team member: Dakota Winslow, Kanghyun Lee

Github repo: [dakotawinslow/hoplite: System for moving AgileX Limo robots in formation]

Video: [EC535 Final project: Hoplite: Coordinated LIMOBot Swarm Control]

Abstract

We present a coordinated multi-robot system using three AgileX LIMO robots fitted with Mecanum wheels for omnidirectional movement and localized with an OptiTrack motion capture system. The robots act as individual agents loosely commanded by a centralized leader, itself directed by user input through a wireless gamepad. The end user directs the swarm through its center-of-mass, allowing the individual robots to direct themselves into position. The motion capture system provides real-time pose information to the entire swarm, enabling both precise movements and collision avoidance. We successfully demonstrated real-time direct control, verified accurate motion capture localization streaming, and created a simulation environment to test swarm behavior. This system lays the groundwork for a fully autonomous formation control framework for small mobile robot teams.

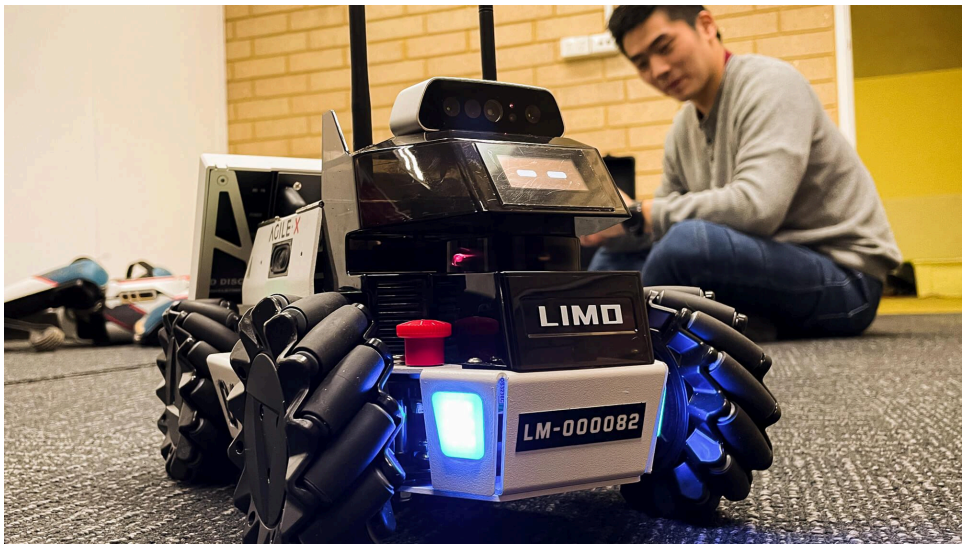

Figure 1: An AgileX LIMO robot with Mecanum wheels

Introduction

As multi-robot systems become increasingly common in autonomous logistics, warehouse automation, and swarm robotics, efficient coordination and communication among robots is essential. However, maintaining formation control across multiple robots in real-time poses significant challenges, especially when the localization system and control logic operate across different versions of the Robot Operating System (ROS).

This project explores the implementation of a small-scale mobile robot swarm using three AgileX LIMO robots. These robots rely on external localization via an OptiTrack motion capture system, which streams robot positions through the Motive software as ROS2 topics. Since the LIMOBots operate on ROS1, a ROS1–ROS2 bridge is required to translate positional data, allowing robots to use MoCAP-based feedback for control.

Our approach uses a leader–follower (master–slave) control architecture, where the leader LIMOBot receives DS5 joystick commands, determines the formation center, and computes target positions for follower LIMOBots. The system is designed to maintain fixed formation spacing and simulate coordinated motion, even in the presence of localization delays and communication overhead.

This project builds on concepts from embedded systems, real-time control, and distributed robotics, applying ROS frameworks and motion capture integration in a resource-constrained environment. Our implementation provides a foundation for future extensions such as dynamic formations, autonomous leader behavior, and sensor-based localization fallback systems.

System Architecture and Control Logic

1. High-Level System Overview

The system was designed such that the user would control the swarm through a wireless gamepad connected by USB to any one of the robots. Joystick events were communicated to a control node that integrated the joystick movements into a pose, which was processed by the hoplite leader node to create target poses for each bot in the swarm. Each individual robot then navigated itself to its target pose while receiving real-time feedback about every robot in the swarm from the motion capture system.

2. Joystick Interface for Leader Control

The leader node was controlled by a generic USB gamepad, which could be connected to any USB- and ROS1-capable computer on the network (we connected it to a robot for simplicity). Joystick and button events were captured by the Linux xpad driver and exposed as a `/dev/input/js*` device. The device was read by ROS's off-the-shelf `joy` node, which emitted `/joy` messages detailing the status of each button and joystick every time the driver reported a change.

```
if __name__ == '__main__':
  try:
    controller = control_point()
    # rospy.spin()
    while not rospy.is_shutdown():
      controller.update_control_point()
```

These `/joy` messages were subscribed to by our `joystick_control` node, which contained an internal pose model. Each time a `/joy` message was received, the model's pose was updated and published as a `Marker()` topic, `/joystick_controller/marker`. We also published a `Twist()` message that was a direct pass-through of the joystick input for debugging (`/cmd_vel`).

While we read `/joy` messages as fast as they came in, we limited traffic on the network by only publishing markers every 0.1 seconds. This was plenty responsive enough for humans but quite infrequent for the network, leaving room for the more important real-time position updates.

3. Squad Leader Command

The squad leader node could run on any computer in the ROS network (for our demo, we used a VM running Ubuntu 18.04 and ROS Melodic). The node read in the markers from the joystick controller, extracted the pose from within, and calculated the target poses for each individual member of the swarm. Our only currently functional positioning mode, `regular_polygon_formation()`, defined a rigid spacing between robots and placed *n* of them around the central control point such that the spacing between neighbors was fixed, creating regular polygons. Each robot's facing was set to match the input pose.

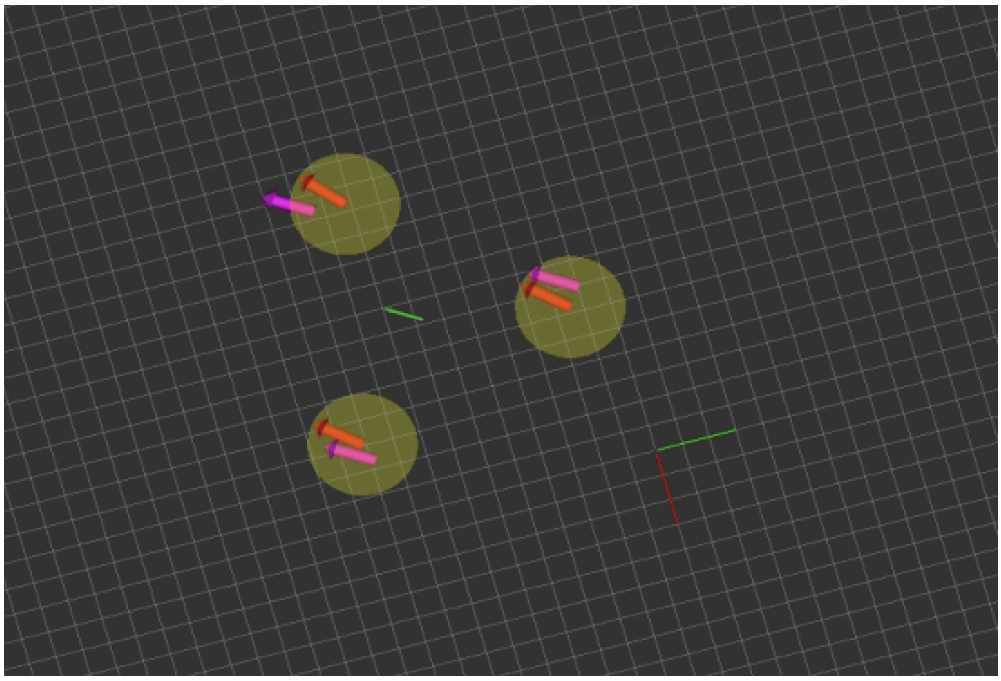

Figure 2: RViz visualization of the robots taking formation. The 'control point' is the small green arrow, the target poses for the robots are magenta.

From these calculations, the leader published the poses under the topics `/hoplite*/_pose` (one for each member of the swarm).

4. Individual Hoplite Software

Each robot ran its own stack of control nodes, consisting of a `hoplite_soldier`, a `limo_driver`, and a `mocap_cleaner`.

hoplite_soldier

The `hoplite_soldier` node was responsible for reading target `_pose` messages from the leader and calculating a path to match that pose. It did so using a PD controller to set velocities for the `limo_driver` node, which controlled the underlying mecanum motion platform. A PD controller was used as the actual kinematic system on the LIMO platform was somewhat obscured, making model-based control difficult and unnecessary. No integral term was required, as the motion capture system provided incredibly stable positional feedback with no appreciable error or drift (<1 mm).

Within the `hoplite_soldier` node was a repulsion-force-based collision avoidance system. After the PD controller calculated a velocity toward the target (the “desired velocity”), the collision avoidance subroutine was called to take into account the positions of other robots. Any nearby robot applied a deflection force to the desired velocity, changing its direction. The deflection force decayed with distance, so nearer robots deflected by greater magnitudes. This resulted in a sort of virtual magnetism, where robots “bounced” off each other without touching.

limo_driver

The final corrected velocities were published as `Twist` messages under the `/hoplite*/cmd_vel` topic for the `limo_driver` to read. The `limo_driver` node served as a bridge between ROS and the LIMO’s motion platform, which was connected to the Linux system via a serial link. This node was provided as part of the LIMO’s default software package and only modified very slightly to read messages from our custom topic.

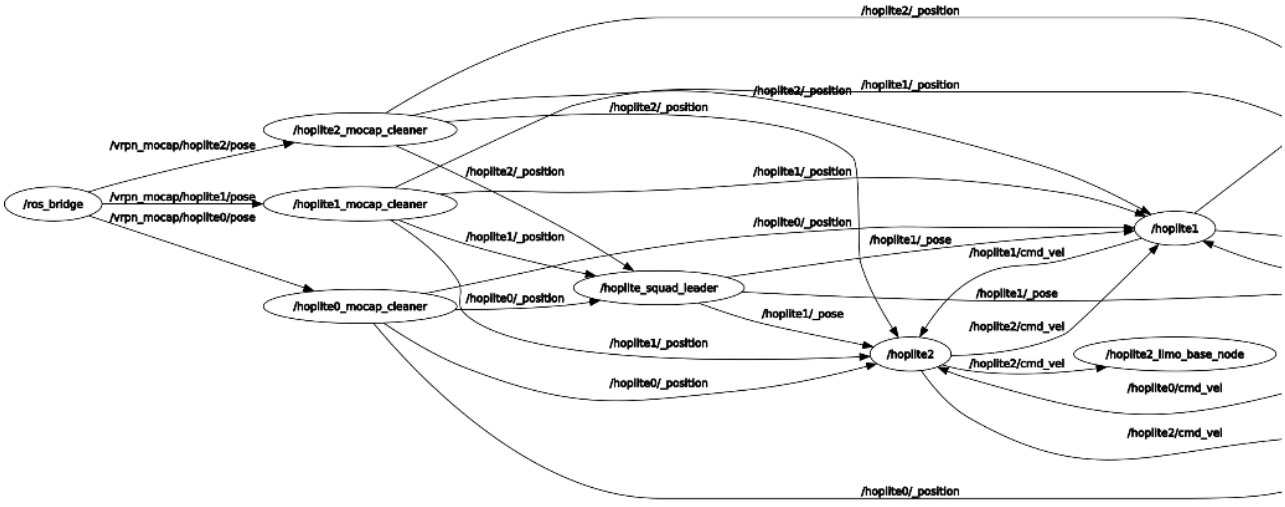

Figure 3: Incomplete graph of ROS nodes, showing data flow from mocap system all the way to driver (called *_limo_base_node here)

mocap_cleaner

The `mocap_cleaner` node acted as a simple translator to transform the XZY coordinate messages from the motion capture system into XY poses for the robots, which generally could only think in 2D. Messages from the motion capture system sometimes included movements or rotations in the “up” axis, but we discarded those since the robots could not actuate in those axes.

5. Localization via OptiTrack Motion Capture

We used an OptiTrack Motion Capture system to provide each robot with localization data about itself and its peers. OptiTrack used infrared cameras to track individual retroreflective markers. Markers in known configurations were tracked as rigid bodies. Poses for each of these detected rigid bodies were published to a VRPN server, which interfaced with a ROS2 node that converted the server updates into ROS2 pose messages. These messages were then bridged to our ROS1 network using a special bridge node that connected both networks. This bridge node (as well as the ROS2 VRPN node) ran on a Raspberry Pi located in RASTIC as part of the motion capture infrastructure there.

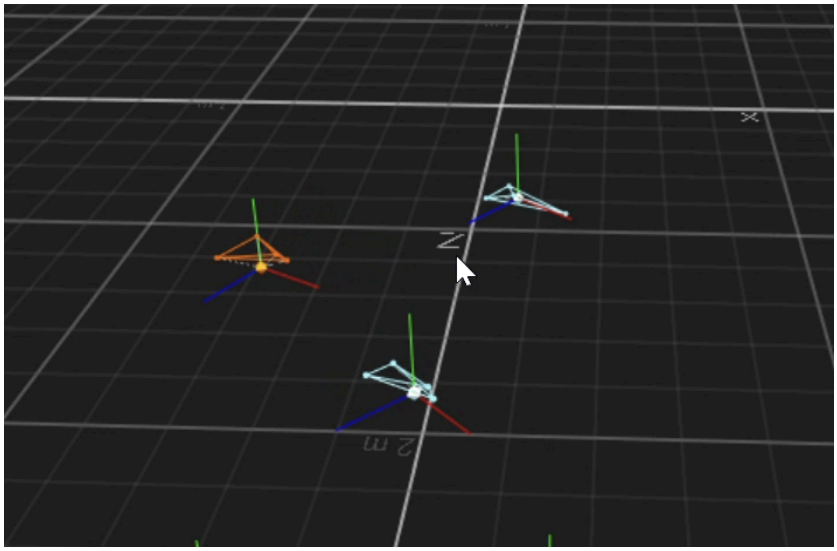

Figure 4: All 3 robots as seen by the motion capture system. Each is a collection of points in 3D space with known spacing between them, called a rigid body in Motive.

The motion capture system captured and reported position data at 120 Hz with positional precision under 1 mm, making it more than capable for our system. While we explored data cleaning via Kalman filter or rolling average, neither was found to be necessary, and we safely used the motion capture localization as ground truth for all positional calculations.

Results

Our process was highly iterative, with each incremental success supporting and informing the next. As a result, the project grew very organically, rather than progressing along a

pre-established path of goals. At the point of evaluation, we were able to place three robots in arbitrary positions within the RASTIC MoCap arena. When the system was started, these three robots determined their own locations in 2D space and moved themselves to orient with the start position, defined as a triangle around the central origin of the room pointing East. The robots then moved in response to inputs from the gamepad, maintaining their formation and returning to it if perturbed. The movements of the robots were smooth and free of resonance or oscillation, thanks to the PD controller.

Limitations and Future Work

The system, while functional, is far from perfect. The collision avoidance system tends to cause the robots to move very slowly any time they might collide. The function scheduling system for the joystick_control python node prioritizes joystick input response over publishing marker outputs, meaning that as long as a controller stick is held down, the robots do not receive updated commands. However, despite these shortcomings, the system is generally functional, smooth, and presents a solid base upon which to develop additional functionality.

We see a future for the system as a functional semi-autonomous lab assistant, capable of moving in concert with systems being tested. With the on-board cameras, a hoplite squad could capture 360-degree video of a robot moving for further analysis. With arms attached to their chassis, the robots could serve as a moving support platform for unsteady walking robots or hold targets for other robots to attempt to contact. Additionally, other formations could be explored, creating a system of robots capable of moving tools and equipment around during lab restructuring or moving in concert to support large loads.

Bibliography

https://github.com/agilexrobotics/limo-doc/blob/master/Limo%20user%20manual(EN).md

The official manual for AgileX LIMO Robots. This github repository also contains links to source code for the robot itself.

https://limo-agx.github.io/index.html

This unofficial guide has additional helpful information about the LIMOs, including how to return them to their factory default settings (very helpful for RASTIC bots which are in an unknown state).

https://www.baillieul.org/Robotics/agitr-letter.pdf

Excellent primer to the world of ROS.

https://gamma.cs.unc.edu/ORCA/

https://www.sciencedirect.com/science/article/pii/S1474667015404616

https://link.springer.com/article/10.1007/s11721-013-0089-4

3 Papers on the topic of swarm navigation and collision avoidance. Unfortunately we were unable to make use of the techniques described in these papers due to time constraints, but these would be the starting point for a continued project.