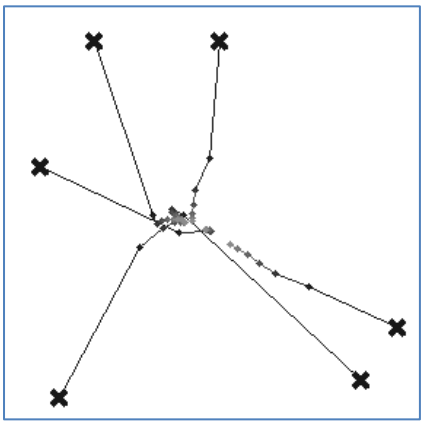
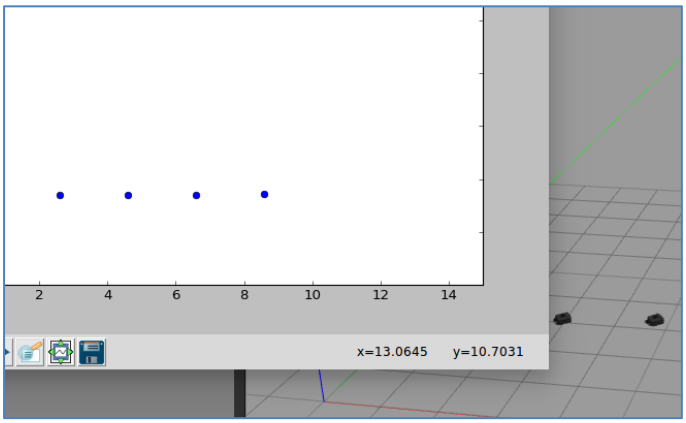
LAB: Multi-robot rendezvous and deployment

In this LAB session, we are implementing multi-robot rendezvous and deployment strategies for a multi-robot team. In the next figure, you can see an example of a 2D rendezvous (consensus on x- and y- variables, so that robots converge to a common point in space), and of a deployment strategy to place robots on a line (consensus on y- coordinate, and deployment on the x- coordinate, with robot-to-robot separation of 2 meters). In particular, these strategies will follow a gossip like communication scheme.





PREVIOUS WORK:

- Read carefully this document
- Choose the number of robots and the communication graph you will use
- Write down the gossip update rule for the rendezvous strategy on x- and y- coordinates
- Write down the gossip update rule for the deployment on a line strategy on x- and y- coordinates for a robot-to-robot separation of 1 meter.
- Make a decision on the setup you will use
- Make a decision on the data structures you will use for the message exchange and for the distributed implementation

SETUP AND ROBOT MODEL:

In this laboratory session, you have freedom to choose the programming and simulation setup. We suggest you to start with your own implementation in **python** from scratch (you can also use Colab if you want). Every robot will have its 2D position (x- and y- coordinates) and will be able to freely move in the environment without any restrictions. This means that, after an update of its position, we will assume that the robot can “jump” or “teleport” to this new position (a single-integrator robot model). Later in the course, you will consider other robot motions, such as, e.g., differential drive robots.

DISTRIBUTED IMPLEMENTATION:

In the previous COLAB exercise, we implemented a centralized global method for testing consensus algorithms. Here, instead, we will make a **distributed implementation**. This means that every robot will run a different instance of **the same code**, customized depending on the robot’s **local state and local data** (position, neighbors, robot identifier, etc.) You can

make the distributed implementation using separated concurrent processes, threads, ROS nodes, etc. However, a **classical** and simpler single program with a ***for loop*** that traverses the different robots can make the trick; in this case, just make sure that the robots do not use more information than the one that is available to it (neighborhood data, no use of global information).

Robots will **exchange data** with other robots by sending and receiving messages. Here, you will need to implement the mechanisms to simulate the exchange of data between robots. Probably one of the easiest ways to implement this is to have a global queue of messages in which robots post their messages. In this case, make sure a robot only reads the messages directed to itself (no use of global data or messages directed to other robots). Depending on the setup that you use, you may already have access to structures for sending/receiving messages and services between processes or nodes.

Recall that we are using a **symmetric (undirected communication)** gossip update scheme. This means that, if a robot i requests an update to robot j, both robots i and j should update their positions as a result.

Besides, you will need to implement additional processes or code in charge of compiling the robot states (positions, ids, etc.) and **plotting** them to see the results of the different simulations.

GENERAL DESCRIPTION: CONSENSUS-BASED MULTI-ROBOT RENDEZVOUS

This section gives you the general idea of what you should develop.

We are implementing a rendezvous system in 2D (consensus on x- and y- variables, gossip like). Robots will exchange data with a neighbor randomly selected, and will average their states. As the number of iterations increases, the robot positions will asymptotically converge towards a common value (i.e., they will achieve rendezvous).

We are creating our robot nodes (**rendezvous_robot**). Each robot will need:

- Its local **x,y** coordinates (stored and maintained by the robot)
- Its list of robot **neighbors** (according to the network topology)
- Store a local **Tlocal** (update period associated to each robot)
- Offer and request a service **gossip_update**. Robots will send / receive requests from other robots (their neighbors) to make a gossip update and get the new value for their x,y coordinates. Depending on your implementation, you may need a tiebreak mechanism to avoid deadlocks. If this is the case, you can for instance assume that a robot i can send only requests to robots j with j>i. Otherwise, it just waits for the other robot j to send its requests to i.
- Either after a state update or periodically, robots will publish their most recent x,y positions in a global data structure, e.g., the **queue_position_plot**. A separate process or program or node will print in a figure the evolution of these robot positions, for visualization purposes. You should include different graphical information to visualize the simulation and, additionally, to observe the evolution of the state of the mission along time.

Every rendezvous_robot will run the same code, which will consist of:

- The initialization: each node will be started with arguments to **customize** it (robot identifier, initial x and y coordinates, update period, list of neighbors..)
- A loop, executed periodically (according to the period Tlocal), in which the robot will randomly select a neighbor, will request a gossip update, and will update its state.

DEPLOYMENT ON A LINE:

This strategy is very similar as the previous one. The only difference here is that you need to modify the update rules so that you can establish the desired line formation.

You can define the formation as you want, but an easy approach is to make robots agree on the average plus a fixed value bx, by, that establishes the goal formation. For instance, you consider a line with robots sorted according to their identifiers, and separated by 1 unit in x and by 0 units in the y coordinate. Thus, robots can make decisions based on the identifier of their neighbors relative to their own id to compute bx_ij. For by_ij, it will be always zero (so it will be like in the rendezvous case).

LAB RESULTS (MANDATORY):

- Implement and submit your solutions to the multi-robot rendezvous and deployment strategy previously commented
- Perform different experiments (e.g., varying the number of robots, graph topologies, number of neighbors, etc.) and comment the results
- Include clear graphical information supporting your comments