CHILI Robots

Tangible Human-Swarm Interaction using ROS

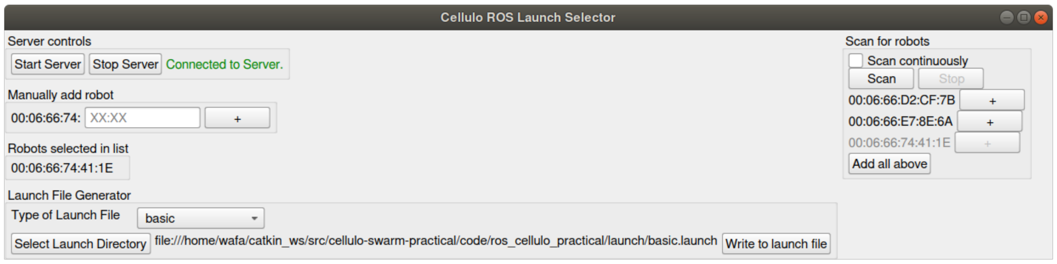
Part I

Understanding the Cellulo Sensor

Step 1 – Generating and understanding the launch files

In order to generate the launch file you should run the `cellulo-robot-pool-launch-selector.pro` project within QtCreator. To do that, open QtCreator and run the project called `cellulo-robot-pool-launch-selector.pro`.

As you run it you will see the Gui window as on Figure 1.


Cellulo ROS Launch Selector

Server controls
Start Server | Stop Server | Connected to Server.

Manually add robot
00:06:66:74: XX:XX +

Robots selected in list
00:06:66:74:41:1E

Launch File Generator
Type of Launch File: basic
Select Launch Directory: file:///home/wafa/catkin_ws/src/cellulo-swarm-practical/code/ros_cellulo_practical/launch/basic.launch | Write to launch file

Scan for robots
Scan continuously
Scan | Stop
00:06:66:D2:CF:7B +
00:06:66:E7:8E:6A +
00:06:66:74:41:1E +
Add all above

Figure 1: Screenshot of the *cellulo − robot − pool − launch − selector.pro* GUI

This tool will help you to generate the launch file for each of the practical task. You can use it to find and add the robots you want to work with.

For this first part, select the basic launch file.
Write your launch file in the `PATH-TO-YOUR-PROJECT/cellulo_swarm_practical_base/ros_cellulo_practical/launch/`

Below we explain what are the main parts of this `basic.launch` file:

```xml
<!-- Define the generic arguments: -->
<arg name="mac_adr0" default="00_06_66_XX_XX_XX" />
<arg name="scale" default="1" />
<arg name="paper_width" default="500" />
<arg name="paper_length" default="500" />
<arg name="threshold" default="200" /> <!-- Sensitivity of the localisation sensor -->
<!-- End of generic arguments: -->
```

Listing 1: Definition of generic parameters for the RosCelluloSensor

```xml
<node name="cellulo_node_$(arg mac_adr0)" pkg="ros_cellulo_swarm" type="ros_cellulo_reduced"
output="screen" args="$(arg mac_adr0)"> <param name="scale_coord" type="double"
value="$(arg scale)" />
    </node>
```

Listing 2: Adding the cellulo node to manage the connection and commands to the robot

```xml
<node name="sensor_node_$(arg mac_adr0)" pkg="ros_cellulo_swarm" type="ros_cellulo_sensor"
output="screen" args="$(arg mac_adr0) $(arg mac_adr1) ">
```

Listing 3: Adding the sensor node to calculate the distances to the neighbours

```xml
<node pkg="tf2_ros" type="static_transform_publisher" name="paper_world_broadcaster"
args="0 0 0 0 0 3.1415 base_footprint paper_world" />
```

Listing 4: Adding the paper frame in tf in the origin of the world

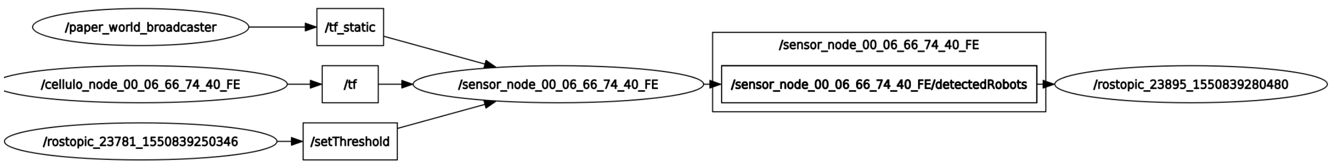

/paper_world_broadcaster
/tf_static
/cellulo_node_00_06_66_74_40_FE
/tf
/rostopic_23781_1550839250346
/setThreshold
/sensor_node_00_06_66_74_40_FE
/sensor_node_00_06_66_74_40_FE
/sensor_node_00_06_66_74_40_FE/detectedRobots
/rostopic_23895_1550839280480

Figure 2: Description of the nodes created and their connections

Step 2 – Launch and use Rviz

In order to test the basic behaviors of the robots you selected, open fist a terminal and run:
`- cd catkin_ws - catkin build`
`- source devel/setup.bash`
`roslaunch ros_cellulo_practical basic.launch`. Rviz should be launched. Under Marker/Marker Topic/, select `/cellulo_node_(mac_address)/visualization_marker_robotdisplay`. You should then see the robot you connected in the paper reference frame.

Step 3 – Send a first command

In order to test the commands that can be sent, we propose you to send a first command setting the velocity of a robot. In a new terminal type:
`rostopic pub /cellulo_node_(MacAddress)/setGoalVelocity geometry_msgs/Vector3 "x: 100.0 y: 50.0 z: 0.0"`.
Note: You can use the tab key for command line autocompletion.

Step 4 – Echo the velocity

Here is the command line to check the velocity of a robot:
`rostopic echo /cellulo_node_(MacAddress)/velocity`.

Step 5 – Echo the sensor value

The aim of this step is to understand and discover the features of the Cellulo sensor. As explained in the session 0, there is a threshold distance after which the robot will not be able to detect obstacles or other robots. The initial value of this threshold is set in the launch file by changing the default value of the ”threshold” argument. After running the launch file, this threshold can be changed by publishing of the corresponding topic as follows:
`rostopic pub /setThreshold std_msgs/Float64 "data:(your threshold value)"`.

Similarly, you can view the messages published on the detected robots/obstacles topic as follows:
`rostopic echo /sensor_node_(MacAddress)/detectedRobots`
and
`rostopic echo /sensor_node_(MacAddress)/detectedObstacles`.

Part II

Interactive Leader-Follower

In this part, you will implement a simple interactive leader follower behavior. In the first step, you will implement a node to select a leader by long pressing one of the Cellulo touch sensors. The LEDs on the leader robot should turn to red, whereas those of the follower should turn to green. In the second step, you will need to implement the control of the follower to maintain a constant bias from the leader equal to the same distance registered when the leader was selected. The final expected behavior that the follower Cellulo would change its position accordingly when the leader is moved by the user.

Step 1 – Leader Selection

Deliverable 1

Implement the function `topicCallback_getTouchKeys` in the file
`ros_cellulo_interaction/ros_cellulo_leader_node.cpp`

To test you code, run:
- `cd catkin_ws` - `catkin build`
- `source devel/setup.bash`
- `roslaunch ros_cellulo_practical (launch file)`. The launch file is the one you create using the same tool described in part I. You should use these steps everytime you do a change in your code.

Step 2 – Leader Following

Deliverable 2

Implement the function `followingLeader` in the file
`ros_cellulo_interaction/ros_cellulo_interaction_node.cpp`

Step 3 – Evaluation

The positions of the robots are saved in `catkin_ws/logs/latest/tf_echo_(mac_address)-stdout.log`.

Deliverable 3

Evaluate your implementation by plotting and comparing the trajectories of the leader and follower.

Part III

Aggregation

Aggregation is commonly observed in natural swarms, as agents can gather on a nutritive resource. In swarm robots aggregation, each of the robot positions itself close to each other in one spot. This is done by reducing the distance between them. The position of the aggregation can be specified or not. If it is not specified, the swarm self-organizes to find a consensus on the aggregation spot. Hence, aggregation at an unspecified spot has an inherent collective decision-making component. The aggregation spot can be specified, for example, by saying the swarm should gather at the brightest or warmest spot. Then each robot, possibly in collaboration with others, has to find that position and stop there.

Naive Approach to Self-aggregation

The behaviour of the swarm as a whole ultimately derives from the behaviour of the individual robot. Individual robot behaviour is organised into several behavioural states, each of which may be considered a distinct 'mode' of behaviour. Transitions between these states are triggered by certain events, just like in a state machine.

Step 1 – Two states approach

In this naive self-aggregation approach, we consider the two behaviours for the robots : 1) wait and 2) random walk. In the random walk, each robot is asked to move in a random direction. If it sees another robot close by a certain distance `th`, it waits. Obviously, the decision to move again depends on the distance and number of the neighbouring robots.

Similarly to the previous parts, you will start be generating the launch file for the aggregation task.

You can edit the launch file to add some useful parameters for your functions.

```
<!-- Define the aggregation arguments: -->
<arg name="[yourvarname]" default="1" />
<!-- End of aggregation arguments: -->
```

Feel free to add arguments and methods to access them in real-time (i.e. timer for the wait state ...).

Deliverable 4

Implement the two basic behaviours `wait` and `random_walk`
Based on these two behaviors, implement the function
`RosCelluloAggregation::naive_calculate_new_velocities()` in the file
`src/ros_cellulo_aggregation/RosCelluloAggregation.cpp`

Step 2 – Evaluation

There are several methods to evaluate the efficiency of the aggregation.

First let's consider the cluster size C_s. For this, we use a threshold $t_robotClose$ to determine robots in the same cluster. Considering $dist(R_i, R_j)$ as the distance between the i^th and j^th robots. The neighboring relationship is defined by the fact :
- $Neigh(R_i, R_j) = 0|iffdist(R_i, R_j) > t_robotClose$ - $Neigh(R_i, R_j) = 1|iffdist(R_i, R_j) <= t_robotClose$ The cluster size for a robot, $size(R_i)$, is the number of robots in the cluster that robot belongs to. This metric calculates the average of cluster sizes for each robot in the swarm.

$$C_s = \frac{1}{n} \sum_{i=1}^{n} size(R_i)^2 \tag{1}$$

This metric ignores spatial distribution of clusters, but gives a measure for size of cluster each robot belongs to. This approach is useful for applications where robots must maintain local links with other robots in a cluster.

Another metric that can be used is the total distance Z. The total distance uses the distance for each pair of robots:

$$Z = -\sum_{i=1}^{n} \sum_{j=i+1}^{n} dist(R_i, R_j) \tag{2}$$

Deliverable 5

Test your implementation with and increasing number of robots.
What parameters could influence this type of aggregation method?
How does the time of the task completion change?
Try to vary some parameters (sensor threshold, other you might implement) and plot the evaluation metrics against them.

Step 3 – Four states approach - Optional

You can try to enhance your implementation by having 4 state behaviors: search, wait, leave a group and change direction. In the search, the robot is looking for other robots and aggregates with them; then there is a transition to the waiting state in such a way that the robot tries to keep a fixed distance from each of its neighbours. As in the standard version of the probabilistic algorithm described earlier, in order to avoid situations where little aggregations inhibit the formation of large aggregates, robots can leave their group at any time with a predetermined probability.

Part IV

Deployment and Area Coverage

Swarm deployment, or dispersion, is a swarm behavior typically used for area coverage. This can be useful for example in a scenario involving a hazardous materials leak in a damaged structure: a swarm of mobile robots equipped with chemical sensors are deployed in the environment and return real-time data indicating the location and concentration of hazards. Other examples include multi-robot exploration and mapping problem. From an educational perspective, this behavior can be used in an activity aiming to illustrate for students some physical phenomenon, for instance, the distribution of charges on a conductor.

Potential fields

One deployment approach is potential-field-based, in which the nodes are treated as virtual particles, subject to virtual forces. To cover the area, these forces should repel the nodes from each other as well as from the obstacles. The initial compact configuration of nodes will therefore spread out to cover the desired area. In addition to these repulsive forces, nodes are subject to a viscous friction force used to ensure that the network will eventually reach a state of static equilibrium (nodes coming to a complete stop).

Each robot is subject to a force $\mathbf{F}$ that is the gradient of a scalar potential field U: $\mathbf{F} = -\nabla U$. The potential field can be divided into two components: the field U_o due to obstacles, and the field U_n due to other robots; these fields give rise to repulsive forces $\mathbf{F}_o$ and $\mathbf{F}_n$ respectively. Thus $U = U_o + U_n$ and $\mathbf{F} = \mathbf{F}_o + \mathbf{F}_n$. Consider the potential field due to obstacles. If we imagine that each node and each obstacle carries an electric charge, we can write doen an expression for the resultant 'electrostatic' potential:

$$U_o = k_o \sum_i \frac{1}{r_i}.$$

The summation is over all obstacles that can be seen by the robot, k_o is a constant describing the strength of the field, and r_i is the Euclidean distance between the robot and obstacle i. Let $\mathbf{p}$ denote the position of the robot and let p_i denote the position of obstacle i. The distance r_i is then given by $r_i = |\mathbf{r_i}| = |\mathbf{p_i} - \mathbf{p}|$. Using these definitions, the force F_o can be computed as:

$$\mathbf{F}_o = -\frac{dU_o}{d\mathbf{p}}$$

Deliverable 6

Derive the force equation F_o in function of k_o and $\mathbf{r_i}$.

By analogy with the obstacle field, we can derive expressions for the potential U_n and force F_n by replacing a summation over visible obstacles with a summation over visible nodes.

Equation of motion and control law

The trajectory of a node subject to force F can be computed using an equation of motion of the following form:

$$\ddot{\mathbf{x}} = (\mathbf{F} - \mu\dot{\mathbf{x}})/m$$

where μ is the viscosity coefficient, m is the mass of the robot.

Deliverable 7

Derive the control law: what should be the new commanded velocity vector given the total force acting on the robot and its current velocity? Hint: Convert to discrete form.

Deliverable 8

Implement the function `RosCelluloCoverage::calculate_new_velocities` in the file `src/ros_cellulo_coverage/RosCelluloCoverage.cpp`.

Effect of changing the control law parameters

There are four internal factors that will effect the output of the experiments: the weights k_o and k_n, the robot mass m and the viscosity coefficient μ.

```
<!-- Define the coverage arguments: -->
<arg name="ko" default="50000" />
<arg name="kr" default="50000" />
<arg name="m" default="0.2" />
<arg name="mu" default="1" />
<!-- End of coverage arguments: -->
```

The initial value of these parameters can be set in the launch file by changing the default value of the corresponding argument. After running the launch file, these values can be changed dynamically by publishing of the corresponding topic using the command line as follows:
rostopic pub /setFieldStrengthObstacles std_msgs/Float64 "data:(your threshold value)".
rostopic pub /setFieldStrengthRobots std_msgs/Float64 "data:(your threshold value)".
rostopic pub /setMass std_msgs/Float64 "data:(your threshold value)".
rostopic pub /setViscocity std_msgs/Float64 "data:(your threshold value)".

➠ **Deliverable 9**

Study the effect of varying these factors on the quality of coverage measured using the metrics described below.

Coverage performance metrics
We will consider two metrics:

1. Deployment entropy described as
$$H = -\sum_{i=1}^{n} p_i \cdot ln(p_i)$$
where $p_i = \frac{ratio_i}{\sum_{k=1}^{n} ratio_k}$ and $ratio_i = \frac{N_i}{S_i}$
Assuming that the map is divided into n (take n=16) uniform sub-regions, N_i is the number of robots in the ith sub-region and S_i is its area.
2. Spacing between the nodes: the average distances between one robot and its neighbouring robots.
3. **(Bonus:)** Coverage Percentage defined as the ratio of the area covered by the robots to the total area. Assuming that the area covered by one robot is defined as the closed disk centered at the robot with a radius of 200 mm. The area covered by the robots is therefore the surface of the union of the disks defined by all the robots.

Aggregation using Potential Field

Now that you have implemented coverage using potential field, you can use a similar method to implement the self-aggregation task in a smarter way.

➠ **Deliverable 10**

Implement the function `RosCelluloAggregation::field_calculate_new_velocities()` in the file `src/ros_cellulo_aggregation/RosCelluloAggregagtion.cpp`.

➠ **Deliverable 11**

Study the effect of varying the physics factors on the aggregation performance metrics.

Bonus - Spot Aggregation

Another type of aggregation consists of having the swarm aggregating in a given spot (a resources sport for instance).
Modify your potential field aggregation to be able to set a spot on which your robots get attracted.
Start by setting an attractive spot in the environment (sx,sy,i), where sx and sy are the coordinate of the spot and i its intensity.

➠ **Deliverable 12**

Implement a new function for aggregation in `RosCelluloAggregation`

➠ **Deliverable 13**

Study the effect of varying the physics factors and of the intensity of the spot on the aggregation performance metrics.