Robotics Practical - Report

Topic 9 - Swarm

Group 29:

Simon GILGIEN 253797

Jean LESUR 284531

Filip SLEZAK 286557

Assistants:

Hala Khodr

Sina Shahmoradi

May 2, 2021

Contents

1 Introduction

This Practical Work aims at familiarizing us with the basics of swarming. We will implement simple swarming behaviors on cellulo robots. Those behaviors are "leader-follower", "aggregation" and "area coverage" and are implemented in C++ through ROS.

2 Interactive leader-follower

2.1 Leader selection

In this first step we implement leader selection by long touch on one of the sensors. The selected robot is called the leader and turns green whereas all the others become red. It is also an easy way to check that our publisher/subscriber mechanism through ROS meets expectations.

2.2 Leader following

In this second step we implement the leader-follower behavior that maintains constant distance between the leader and the follower in this two robot setup. We validate our code in Annex 6.1 by manually moving around the leader and observe the expected follower behavior. Further evaluation is provided in the following section.

2.3 Evaluation

We used the log files to plot the leader-follower positions over time in Figure 1. The leader was moved by hand so the speeds do not exactly match that of the follower which, combined with latency and reactivity of the follower, partially explains the errors visible on this graph. During the manipulation, the behavior of the robots was coherent. In the beginning the movements of the leader are neglected as we had to bring it to the neighborhood of the follower to get detected. The follower then does a decent job at following the leader up to the forth corner where the robot went too fast and went out of detection range for the follower. However, it corrects its position when the leader comes in range again. At the end of the trajectory, we captured the follower and we can see that it came back to its normal position. In Figure 2 we observe better that the x and y positions follow the same trend (we can ignore the theta orientation in the implementation of our algorithm).

At the beginning of the simulation, the follower takes time to start moving but it is able to roughly follow the leader's trajectory. The position difference between the two robots remains quite constant and this was expected. However, this leader-follower behavior is not perfect as they are some discrepancies in the way the straight lines in the motion of the leader are executed by the follower.

In addition, we observe in Figure 2 some latency between the motion of the follower. The red and green curves of the follower are delayed with regards to the ones of the leader. This is a common problem in swarm robotics where performant communication networks among agents are key to the swarms' operations.

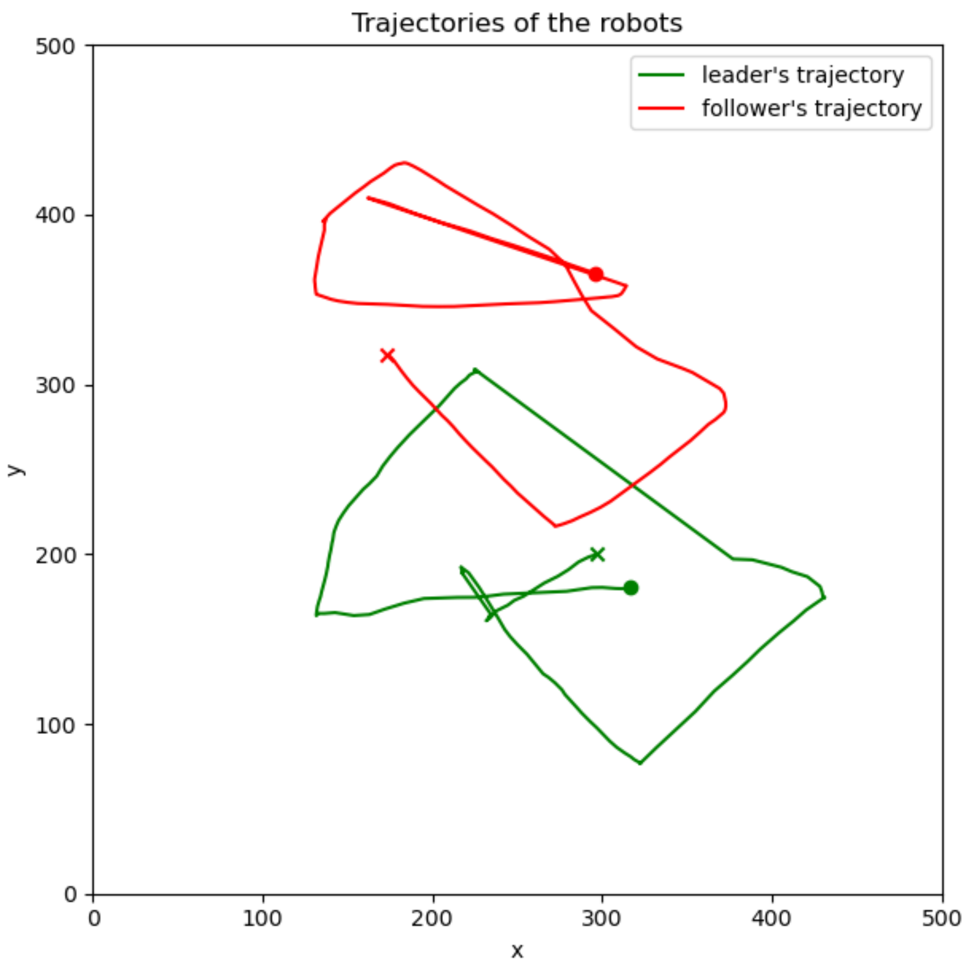

Figure 1: Trajectories of the leader and the follower robot on the map. **X**: start, **O**: end. The leader is moved by hand while the follower tries to stay within distance of its leader.

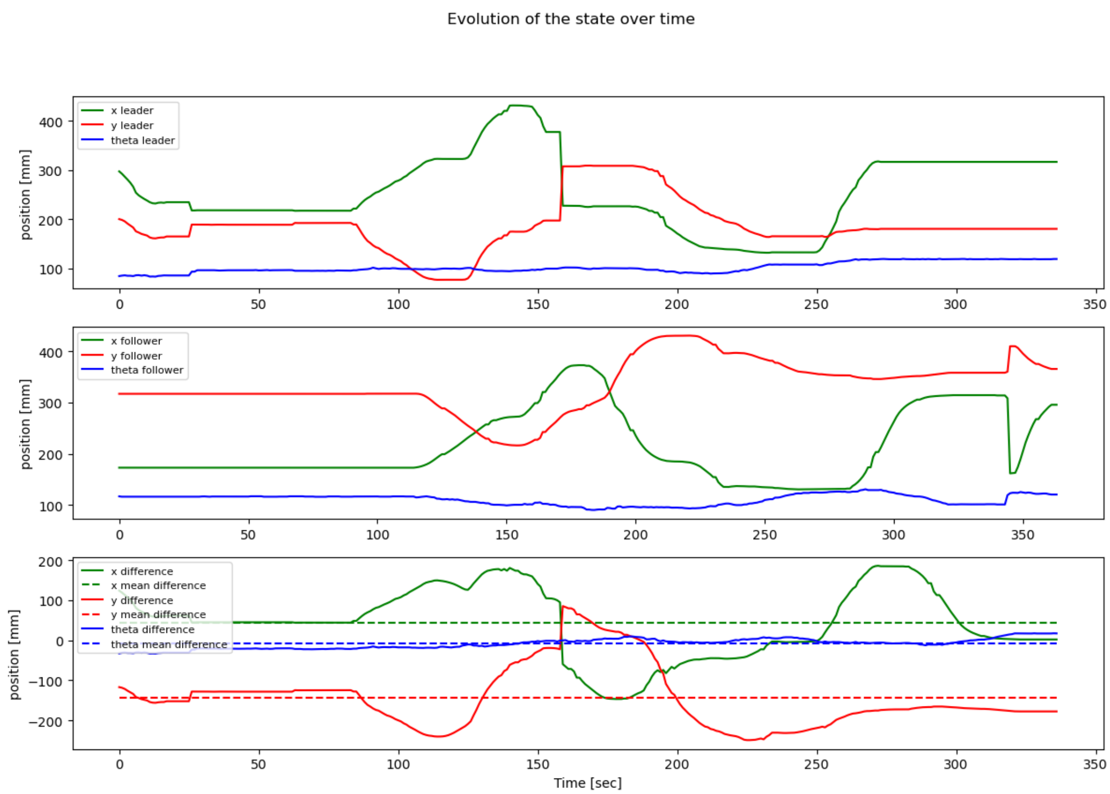

Figure 2: Evolution of the states of the robots during the manipulation in Figure 1.

3 Aggregation

3.1 Two states approach

The two states approach to the aggregation algorithm is for the robots to randomly walk in any direction on the map until they find a robot in their vicinity, at which point they stop. These two behaviors: 'random walk' and 'wait', are the two states implemented in our code in Annex 6.2 starting on line 90. The random walk is best explained by the code itself, in essence we try to avoid too frequent direction changes that would reflect a sort of white noise on the position resulting in no consequent movement that could serve in finding the other robots on the map, and thus aggregation itself, all the while avoiding leaving the map. The wait mechanism is fairly simple, it makes use of the proximity sensors and a threshold to toggle the state of the robot. After tuning our random walk algorithm using our visual observations, we obtain a decent aggregation time. A further evaluation using the proposed metrics is available in the following section.

3.2 Evaluation

To evaluate the efficiency of the aggregation, we computed two metrics : the cluster size and the total distance Z, as required by the practical's assignment. The cluster size is a little tricky to compute, the code in Annex 6.4 uses graph connectivity properties to distribute a value into the network starting from a node of interest. By iterative multiplication of the adjacency matrix with a one-hot vector (one at our node of interest, zero elsewhere) we distribute this one into all of its connected neighbours. Thus by counting the number of non-zero elements after N iterations (N assumed large enough to ensure convergence, must be at minimum equal to the number of robots in the network for the case where all robots are in a single cluster) we obtain the cluster sizes for each robot. It is a little computation heavy but it does the trick in the scope of this practical. The other parts of the code are relatively standard and mainly based on data manipulation.

We also compute the execution time, needed for convergence, defined as each robot being less than 2mm away, in $l1$ distance, from its final position. In Figure 3 we observe the aggregation results using different numbers of robots as a function of the proximity threshold. In all cases, as we increase the threshold the robots converge faster. For the four robots case, we always end up with two clusters of two robots. This is very likely influenced by the initial position that we did not change when generating this data. For cases with one, two or three robots, given our two state algorithm, the aggregation will always result in a single cluster containing all the robots.

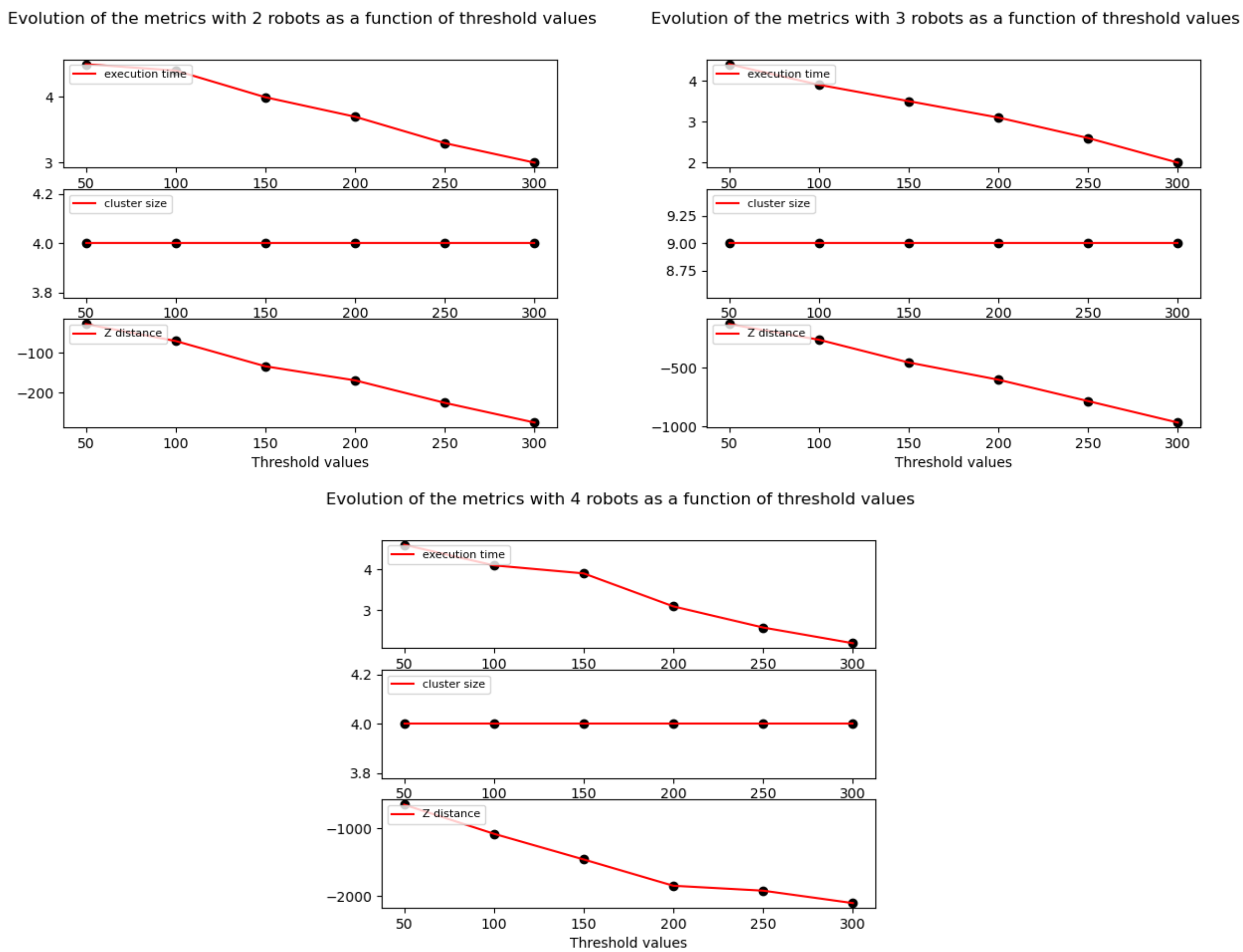

Figure 3: Evolution of the metrics for the aggregation algorithm as a function of the threshold value for 2,3 and 4 robots, respectively. The execution time is in seconds.

4 Deployment and area coverage

4.1 Potential field

As explained in the practical assignment, in this potential field approach, nodes and obstacles are treated as virtual particles that repel each other regardless of whether they are nodes or obstacles. As explained in the practical's assignment, 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). We can derive the force resulting from the potential field $\boldsymbol{F}_o$ as follows

$$\boldsymbol{F}_o = -\frac{dU_o}{d\boldsymbol{x}} = -\frac{d}{d\boldsymbol{x}}\left(k_o \sum_i \frac{1}{r_i}\right) = k_o \sum_i \frac{1}{r_i^2}\frac{dr_i}{d\boldsymbol{x}} = k_o \sum_i \frac{1}{r_i^2}\frac{d}{d\boldsymbol{x}}\left|\boldsymbol{x}_i - \boldsymbol{x}\right| = -k_o \sum_i \frac{\boldsymbol{x}_i - \boldsymbol{x}}{\left|\boldsymbol{x}_i - \boldsymbol{x}\right|^3}$$

As expected, this is similar to the gravitational or electrostatic force. The force above is expressed for obstacles, subscript o, the expression is identical for nodes, subscript n.

4.2 Equation of motion and control law

We use a first-order discrete-time integration to obtain the following control law. The code implementing this control law is available in Annex 6.3 starting on line 124.

$$\dot{\boldsymbol{x}}(t + \Delta t) = \dot{\boldsymbol{x}}(t) + \Delta t \cdot \ddot{\boldsymbol{x}}(t) = \dot{\boldsymbol{x}}(t) + \Delta t \cdot \frac{\boldsymbol{F}(t) - \mu\dot{\boldsymbol{x}}(t)}{m}$$

4.3 Influence of the control law parameters

The results of our analysis conducted by the code in Annex 6.4, very similar to the one used in Section 3.2, are presented in Figure 4. To analyze the effect of each parameter, we did four batches of size ten in which we only vary one parameter of interest at a time in a linear manner.

For varying parameter k_o, the greater the repelling constant k_o the greater the Z-distance between them as well as the cluster size. A big k_o make the robots group at the center of the map ; this is where the distance to the 4 obstacles (map edges) is minimal. Regarding the execution time, when k_o is big, the robots are moving faster away from the obstacles.

When increasing the parameter k_r, the cluster size as well as the total distance Z decrease. This makes sense as the robots will have an greater repelling effect on the others. Thus they will stay at a greater distance from each other. For a big k_r, the robots are moving faster away from each other but only up to a certain point, at which the obstacle avoidance behavior takes over.

For the mass, its variation does not influence the cluster size as we can see on the bottom left subplot of Figure 4. The Z distance increases but not significantly (going from -1162 to -1152). The execution time is not influenced by the parameter m.

The parameter μ is the viscosity of the simulation. It has an influence on the cluster size. The more viscous the simulation, the smaller the cluster size and the Z distance. The greater the μ parameter, the more the robot "brakes" and thus the more time is required to reach equilibrium.

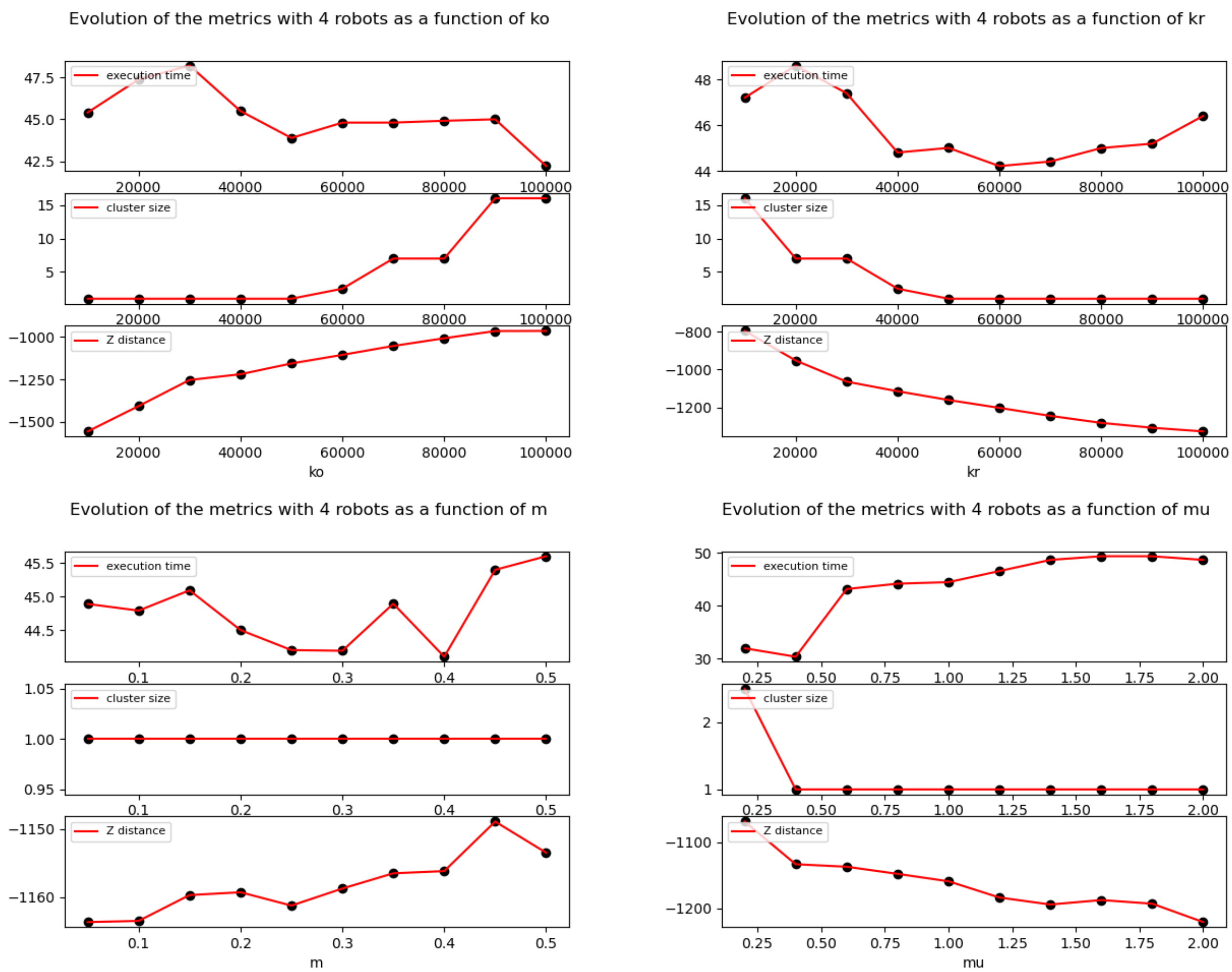

Figure 4: Evolution of the metrics for area coverage algorithm as a function of the control law parameters for a simulation with 4 robots.

4.4 Aggregation using Potential Field - OPTIONAL

To convert this algorithm into an aggregation algorithm all we have to do is change the sign of the force $\boldsymbol{F}_n$, thereby attracting the nodes/robots and repelling obstacles. This has been implemented in Annex 6.2 starting on line 158.

5 Conclusion

In conclusion, we implemented some basic swarming algorithms for the first time and got to use ROS which was new to one of us. We were surprised by the very precise localization method used by the cellulo robot. It seemed like an overkill at first but we were glad it was used for great precision, seamless interaction with the simulation and robustness against kidnapping situations. The inconvenience being one needs this specially printed map. We enjoyed this interactive practical in itself and received great support, even during the online session, but spent an awful lot of time on data generation, analysis and plotting the graphs. At least we now know how to better use these tools for future projects :)

6 Annex

6.1 Leader-Follower

```cpp
#include <ros/ros.h>
#include "ros_cellulo_swarm/cellulo_touch_key.h"
#include "ros_cellulo_swarm/cellulo_visual_effect.h"
#include "ros_cellulo_swarm/ros_cellulo_sensor.h"
#include "std_msgs/String.h"
#include "geometry_msgs/Pose2D.h"
#include "tf2_ros/transform_listener.h"
#include "std_msgs/Float64.h"


geometry_msgs::Vector3 operator-(geometry_msgs::Vector3 a, const
    geometry_msgs::Vector3 &b)
{
    a.x -= b.x;
    a.y -= b.y;
    a.z -= b.z;
    return a;
}

geometry_msgs::Vector3 operator*(double a, geometry_msgs::Vector3 b)
{
    b.x *= a;
    b.y *= a;
    b.z *= a;
    return b;
}

class FollowLeader
{
public:

    void followingMyLeader()
    {
        // Implement here your control.
        // 1- Calculate the required velocity
        // (Useful variables: Ku, distance_to_leader and
            reference_distance)
        // 2- Publish the velocity
        geometry_msgs::Vector3 velocity = Ku * (distance_to_leader -
            reference_distance);
        velocity.z = 0;
```

```
        VelocityPublisher.publish(velocity);
    }
};
```

```
#include <ros/ros.h>
#include "ros_cellulo_swarm/cellulo_touch_key.h"
#include "ros_cellulo_swarm/cellulo_visual_effect.h"
#include "std_msgs/String.h"

#include <string>
#include <iostream>

#include <ros/console.h>

class LeaderSelection
{
    // Call back function if a change on one of the long touch sensors
       on one of the robots is detected.
    // The functions should detects which robot was touched and publish
        its mac_adress on the LeaderPublisher
    void topicCallback_getTouchKeys(const ros_cellulo_swarm::
       cellulo_touch_key& message)
    {
        // 1- Evaluate if the call back is a touch or release
        // 2- If it is a touch:
        // a- detect which robot was selected.
        //     i- publish its mac address
        //    ii- turn its leds to red.
        // b- turn the leds of other robots to green

        std_msgs::String leader;
        ros_cellulo_swarm::cellulo_visual_effect effect;

        bool touch = false;
        for(int i = 0; i<6; i++)
        {
            if(message.keysTouched[i] != 0)
            {
                ROS_INFO("Key touched: %i", i);
                touch = true;
            }
        }

        if(touch)
```

```
        {
            ROS_INFO("Second loop");
            leader.data = message.header.frame_id;
            LeaderPublisher.publish(leader);

            ROS_INFO("published to /leader");

            for(int i = 0; i < nb_robots; i++)
            {
                effect.blue = 0;
                if(std::string(present_robots[i+1]) == message.header.frame_id)
                {
                    effect.red = 255;
                    effect.green = 0;
                }
                else
                {
                    effect.red = 0;
                    effect.green = 255;
                }
                VisualEffectPublisher[i].publish(effect);
            }
        }
    }
};
```

6.2 Aggregation

```
#include "ros_cellulo_aggregation/RosCelluloAggregation.hpp"
#include "tf2_ros/transform_listener.h"

#include <cstdlib>

#define VEL_INCREMENT 10
#define MAX_VEL       200


geometry_msgs::Vector3 operator*(geometry_msgs::Vector3 b, double a)
{
    b.x *= a;
    b.y *= a;
    b.z *= a;
    return b;
```

```
}

RosCelluloAggregation::RosCelluloAggregation(ros::NodeHandle&
    nodeHandle) : nodeHandle_(nodeHandle)
{
        if (!readParameters()) {

                ROS_ERROR("Could not read parameters :(.");
                ros::requestShutdown();
        }

        //subscribers
              // TODO add subscribers to parameters you want to test in
                 live

        nb_of_robots_detected=-1;
        nb_of_obstacles_detected=-1;
        velocity_updated=false;

}


void RosCelluloAggregation::naive_calculate_new_velocities()
{
    if(nb_of_robots_detected!=-1 && nb_of_obstacles_detected!=-1 &&
       velocity_updated)
    {
        geometry_msgs::Vector3 vel;
        //****************
        // FILL HERE - Part III Step 1
        //****************
        if(nb_of_robots_detected > 0)
        {
            vel.x = 0;
            vel.y = 0;
            vel.z = 0;
        }
        else
        {
            if(nb_of_obstacles_detected > 0)
            {
                bool velocity_reset = false;
                for(int i = 0; i < nb_of_obstacles_detected; i++)
                {
```

```
                    if(fabs(distances_to_obstacles[i].x) < 150 || fabs(distances_to_obstacles[i].y) < 150)
                    {
                        if(!velocity_reset)
                        {
                            velocity.x = 0;
                            velocity.y = 0;
                            velocity_reset = true;
                        }
                        if(fabs(distances_to_obstacles[i].x) > 5)
                            velocity.x -= MAX_VEL/4*fabs(distances_to_obstacles[i].x)/distances_to_obstacles[i].x;
                        if(fabs(distances_to_obstacles[i].y) > 5)
                            velocity.y -= MAX_VEL/4*fabs(distances_to_obstacles[i].y)/distances_to_obstacles[i].y;
                    }
                }
            }

            vel.x = velocity.x;
            vel.y = velocity.y;
            if(fabs(velocity.x) < 10 && fabs(velocity.y) < 10)
            {
                vel.x = MAX_VEL * ((double) rand())/RAND_MAX - MAX_VEL/2;
                vel.y = MAX_VEL * ((double) rand())/RAND_MAX - MAX_VEL/2;
            }


            //vel.x = fmax(fmin(velocity.x + VEL_INCREMENT*((double) rand()) / RAND_MAX - VEL_INCREMENT/2, MAX_VEL), -MAX_VEL);
            //vel.y = fmax(fmin(velocity.y + VEL_INCREMENT*((double) rand()) / RAND_MAX - VEL_INCREMENT/2, MAX_VEL), -MAX_VEL);

            vel.z = 0;
        }


        ROS_INFO("velocity calcul %lf %lf",vel.x,vel.y);
```

```cpp
        RosCelluloAggregation::publisher_Velocity.publish(vel);

        nb_of_robots_detected=-1;
        nb_of_obstacles_detected=-1;
        velocity_updated=false;

    }
    else
    {
        return;
    }
}


void RosCelluloAggregation::field_calculate_new_velocities()
{

    if(nb_of_robots_detected!=-1 && nb_of_obstacles_detected!=-1 && velocity_updated)
    {
        //****************
        // FILL HERE - Part IV
        //****************
        double ko, m, mu, kr;

        nodeHandle_.getParam("ko",ko);
        nodeHandle_.getParam("m", m);
        nodeHandle_.getParam("mu", mu);
        nodeHandle_.getParam("kr", kr);
        double Fx = 0;
        double Fy = 0;
        for(int i = 0; i<nb_of_robots_detected; i++)
        {
            Fx -= -kr * distances_to_robots[i].x / pow(distances_to_robots[i].z,3);
            Fy -= -kr * distances_to_robots[i].y / pow(distances_to_robots[i].z,3);
        }
        for(int i = 0; i<nb_of_obstacles_detected; i++)
        {
            Fx += -ko * distances_to_obstacles[i].x / pow(distances_to_obstacles[i].z,3);
            Fy += -ko * distances_to_obstacles[i].y / pow(distances_to_obstacles[i].z,3);
```

```
        }

        //2- compute V
        // NB: you can use timestep=dt=1/rate.
        geometry_msgs::Vector3 vel;
        vel.x = velocity.x + (Fx - mu*velocity.x)/(m*rate);
        vel.y = velocity.y + (Fy - mu*velocity.y)/(m*rate);
        vel.z = 0;
        //3- Limit speed (in below function limit_velocity)
        velocity = limit_velocity(vel, 300);
        //4- Publish on the corresponding topic
        ROS_INFO("velocity calcul %lf %lf",vel.x,vel.y);
        RosCelluloAggregation::publisher_Velocity.publish(vel);

        nb_of_robots_detected=-1;
        nb_of_obstacles_detected=-1;
        velocity_updated=false;

    }
    else
    {
        return;
    }
}

geometry_msgs::Vector3 RosCelluloAggregation::limit_velocity(geometry_msgs::Vector3 v,double limit)
{
    // TO DO //
    //Limit speed to a maximum (300 mm/s). NB: direction should not change, only the norm.
    //You can also put a lower bound for the speed to avoid small in-place oscillations
    geometry_msgs::Vector3 newV;
    double vNorm = norm(v.x, v.y);
    if(vNorm > limit)
    {
        newV = v * (limit/vNorm);
    }
    else
    {
        newV = v;
    }

```

```
    return newV;
}



double RosCelluloAggregation::norm(double x,double y)
{
        return sqrt(pow(x,2)+pow(y,2));
}
```

6.3 Area Coverage

```
#include "ros_cellulo_coverage/RosCelluloCoverage.hpp"
#include "tf2_ros/transform_listener.h"

#include <cmath>

geometry_msgs::Vector3 operator*(double a, geometry_msgs::Vector3 b)
{
    b.x *= a;
    b.y *= a;
    b.z *= a;
    return b;
}

geometry_msgs::Vector3 operator*(geometry_msgs::Vector3 b, double a)
{
    b.x *= a;
    b.y *= a;
    b.z *= a;
    return b;
}

void RosCelluloCoverage::calculate_new_velocities()
{
    //// TO IMPLEMENT ////
    if(nb_of_robots_detected!=-1 && nb_of_obstacles_detected!=-1 && velocity_updated)
    {
        //1- compute F
        double Fx = 0;
        double Fy = 0;
        for(int i = 0; i<nb_of_robots_detected; i++)
        {
```

```
            Fx += -kr * distances_to_robots[i].x / pow(distances_to_robots[i].z,3);
            Fy += -kr * distances_to_robots[i].y / pow(distances_to_robots[i].z,3);
        }
        for(int i = 0; i<nb_of_obstacles_detected; i++)
        {
            Fx += -ko * distances_to_obstacles[i].x / pow(distances_to_obstacles[i].z,3);
            Fy += -ko * distances_to_obstacles[i].y / pow(distances_to_obstacles[i].z,3);
        }

        //2- compute V
        // NB: you can use timestep=dt=1/rate.
        geometry_msgs::Vector3 vel;
        vel.x = velocity.x + (Fx - mu*velocity.x)/(m*rate);
        vel.y = velocity.y + (Fy - mu*velocity.y)/(m*rate);
        vel.z = 0;
        //3- Limit speed (in below function limit_velocity)
        velocity = limit_velocity(vel, 300);
        //4- Publish on the corresponding topic
        publisher_Velocity.publish(velocity);
        //reset
        nb_of_robots_detected=-1;
        nb_of_obstacles_detected=-1;
        velocity_updated=false;
    }

}


geometry_msgs::Vector3 RosCelluloCoverage::limit_velocity(geometry_msgs::Vector3 v,double limit)
{
    // TO DO //
    //Limit speed to a maximum (300 mm/s). NB: direction should not change, only the norm.
    //You can also put a lower bound for the speed to avoid small in-place oscillations
    geometry_msgs::Vector3 newV;
    double vNorm = norm(v.x, v.y);
    if(vNorm > limit)
    {
```

```
            newV = v * (limit/vNorm);
        }
        else
        {
            newV = v;
        }

        return newV;
}

double RosCelluloCoverage::norm(double x,double y)
{
            return sqrt(pow(x,2)+pow(y,2));
}
```

6.4 Analysis Code

In this section, you will find the three python scripts that we've developed for the the analysis conducted in Part 1, Part 2 and Part 3, respectively. They are not optimal with regards to memory footprint and speed but they work well for our purpose.

6.4.1 Part 1 Analysis

```
# script for first part of the PW (leader-follower)
import numpy as np
import matplotlib.pyplot as plt

files= ['tf_echo_00_06_66_74_41_4C-8-stdout.csv',
         'tf_echo_00_06_66_74_3E_82-9-stdout.csv',
         'tf_echo_00_06_66_74_41_4C-12-stdout.csv',
         'tf_echo_00_06_66_74_3E_82-11-stdout.csv']

X = 1
Y = 2
THETA = 4

colors = ['g', 'r', 'b', 'k']

def plotTrajectories(leader, follower):
    fig, ax = plt.subplots()
    ax.plot(leader[0],leader[1],colors[0])
    ax.scatter(leader[0][0], leader[1][0], marker='x', c=colors[0])
    ax.scatter(leader[0][-1], leader[1][-1], marker='o', c=colors[0])

    ax.plot(follower[0],follower[1],colors[1])
```

```python
    ax.scatter(follower[0][0], follower[1][0], marker='x', c=colors[1])
    ax.scatter(follower[0][-1], follower[1][-1], marker='o', c=colors[1])

    plt.title('Trajectories of the robots')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.legend(('leader\'s trajectory','follower\'s trajectory'))
    plt.xticks([])
    plt.yticks([])
    plt.axis('equal')
    plt.show()

def plotState(leader, follower):
    fig, ax = plt.subplots(3)

    timeline = np.arange(len(leader[0]))
    ax[0].plot(timeline, leader[0], colors[0])
    ax[0].plot(timeline, leader[1], colors[1])
    ax[0].plot(timeline, leader[2], colors[2])
    ax[0].legend(('x leader','y leader','theta leader'), loc='upper left', fontsize=8)

    timeline = np.arange(len(follower[0]))
    ax[1].plot(timeline, follower[0], colors[0])
    ax[1].plot(timeline, follower[1], colors[1])
    ax[1].plot(timeline, follower[2], colors[2])
    ax[1].legend(('x follower','y follower','theta follower'), loc='upper left', fontsize=8)

    nb = min(len(leader[0]), len(follower[0]))
    timeline = np.arange(nb)
    ax[2].plot(timeline, [x - y for x,y in zip(leader[0][:nb], follower[0][:nb])], colors[0])
    ax[2].plot(timeline, np.ones(nb)*np.mean([x - y for x,y in zip(leader[0][:nb], follower[0][:nb])]), colors[0]+'--')
    ax[2].plot(timeline, [x - y for x,y in zip(leader[1][:nb], follower[1][:nb])], colors[1])
    ax[2].plot(timeline, np.ones(nb)*np.mean([x - y for x,y in zip(leader[1][:nb], follower[1][:nb])]), colors[1]+'--')
    ax[2].plot(timeline, [x - y for x,y in zip(leader[2][:nb], follower[2][:nb])], colors[2])
    ax[2].plot(timeline, np.ones(nb)*np.mean([x - y for x,y in zip(leader[2][:nb], follower[2][:nb])]), colors[2]+'--')
```

```

    ax[2].legend(('x error','x mean error', 'y error', 'y mean error','theta error', 'theta mean error'), loc='upper left', fontsize=8)

    plt.suptitle('Evolution of the state over time')
    plt.show()



if __name__ == '__main__':

    leader_traj = []
    follower_traj = []

    for idx, item in zip([0,1],files[:2]):
        xs = []
        ys = []
        thetas = []

        f = open(item, 'r')
        lines = f.readlines()
        print('lines # : ', len(lines))
        f.close()

        for l in lines[1::100]: #discard the first row (title for the columns)
            tab = l.split(',')
            xs.append(float(tab[X]))
            ys.append(float(tab[Y]))
            thetas.append(float(tab[THETA].replace('\n', '')))

        if idx == 0:
            leader_traj= [xs,ys,thetas]
        else:
            follower_traj = [xs,ys,thetas]

    plotTrajectories(leader_traj, follower_traj)
    plotState(leader_traj, follower_traj)

#EOF
```

6.4.2 Part 2 Analysis

```
# script for second part of the PW (aggregation)
import os
```

```python
import numpy as np
import matplotlib.pyplot as plt

part = '/part2/'
curr_dir = os.getcwd()
folders = os.listdir(curr_dir+part)

TIMESTAMP = 0
NB_ROBOTS = 2
THRESH = 4
X = 1
Y = 2

colors = ['r', 'g', 'b', 'k']

def plotData(list_thresh, data1, data2, data3, title1, title2, title3, nb_rbt):
    fig, ax = plt.subplots(3)

    ax[0].plot(list_thresh, data1, colors[0])
    ax[0].scatter(list_thresh, data1, c='k', marker='o')
    ax[0].legend((title1,), loc='upper left', fontsize=8)

    ax[1].plot(list_thresh, data2, colors[0])
    ax[1].scatter(list_thresh, data2, c='k', marker='o')
    ax[1].legend((title2,), loc='upper left', fontsize=8)

    ax[2].plot(list_thresh, data3, colors[0])
    ax[2].scatter(list_thresh, data3, c='k', marker='o')
    ax[2].legend((title3,), loc='upper left', fontsize=8)
    ax[2].set_xlabel('Threshold values')

    plt.suptitle('Evolution of the metrics with ' + nb_rbt + ' robots as a function of threshold values')
    plt.show()

def getSize(rbt, adjacency):
    nx = adjacency.shape[0]
    b = np.zeros((nx,1))
    b[rbt,0] = 1
    output = np.linalg.matrix_power(adjacency, 10) @ b
    return np.count_nonzero(output)

if __name__ == '__main__':
```

```python

    list_thresh = []
    list_cs = []
    list_z = []
    list_execution_time = []

    for f in sorted(folders): #for each folder
        if f[0] != '.':
            tab = f.split('_')
            nb_rbt = int(tab[NB_ROBOTS]) #find infos of current log
            thresh = int(tab[THRESH])

            dir = curr_dir+part+f+'/'
            if dir[0] != '.':
                execution_time = 0.

                rbt_list = []
                for file in sorted(os.listdir(dir)): #for each file in folder
                    if '.csv' in file[-4:]:
                        f = open(dir+file, 'r')
                        lines = f.readlines()
                        f.close()
                        tab = lines[-1].split(',') #find position at convergence = the end
                        rbt = np.array([float(tab[X]),float(tab[Y])])
                        rbt_list.append(rbt)

                        for l in lines[1:]: #discard the title row
                            tab = l.split(',') #find position at convergence = the end
                            temp = np.array([float(tab[X]),float(tab[Y])])
                            if (np.abs(temp - rbt) < 2).all():
                                execution_time = max(float(tab[TIMESTAMP]), execution_time)
                                break

                adjacency   = np.eye(nb_rbt) #np.zeros((nb_rbt, nb_rbt))
                dist_matrix = np.zeros((nb_rbt, nb_rbt))

                for i in range(len(rbt_list)):
                    for j in range(i+1, len(rbt_list)):
```

```
                        dist = np.linalg.norm(rbt_list[i] - rbt_list[j])
                        if dist <= thresh:
                            adjacency[i,j] = 1
                            adjacency[j,i] = 1
                        dist_matrix[i,j] = dist
                        dist_matrix[j,i] = dist

                cs = 0
                for i in range(nb_rbt):
                    size = getSize(i, adjacency)
                    cs += size**2
                cs /= nb_rbt

                Z = 0
                for i in range(len(rbt_list)):
                    for j in range(i+1, len(rbt_list)):
                        Z += dist_matrix[i,j]
                Z *= -1

                print('=== === ===')
                print('NB RBT : ', nb_rbt)
                print('THRESH :', thresh)
                print("CS : ", cs)
                print("Z : ", Z)
                print("EXEC TIME :", execution_time)

                list_thresh.append(thresh)
                list_cs.append(cs)
                list_z.append(Z)
                list_execution_time.append(execution_time)

    plotData(list_thresh[:6], list_execution_time[:6], list_cs[:6], list_z[:6], 'execution time', 'cluster size', 'Z distance', '2')
    plotData(list_thresh[6:12], list_execution_time[6:12], list_cs[6:12], list_z[6:12], 'execution time', 'cluster size', 'Z distance', '3')
    plotData(list_thresh[12:18], list_execution_time[12:18], list_cs[12:18], list_z[12:18], 'execution time', 'cluster size', 'Z distance', '4')

# EOF
```

6.4.3 Part 3 Analysis

```
# script for third part of the PW (area coverage)
import os
import numpy as np
import matplotlib.pyplot as plt

part = '/part3/'

curr_dir = os.getcwd()
folders = os.listdir(curr_dir+part)

TIMESTAMP = 0
PARAM_ID = 1
PARAM_VAL = 2
X = 1
Y = 2
DEFAULT_KO = 50000
DEFAULT_KR = 50000
DEFAULT_M = 0.2
DEFAULT_MU = 1
KO = 2
KR = 4
MU = 6
M  = 8

colors = ['r', 'g', 'b', 'k']

def plotData(list_thresh, data1, data2, data3, title1, title2, title3,
    param):
    fig, ax = plt.subplots(3)

    ax[0].plot(list_thresh, data1, colors[0])
    ax[0].scatter(list_thresh, data1, c='k', marker='o')
    ax[0].legend((title1,), loc='upper left', fontsize=8)

    ax[1].plot(list_thresh, data2, colors[0])
    ax[1].scatter(list_thresh, data2, c='k', marker='o')
    ax[1].legend((title2,), loc='upper left', fontsize=8)

    ax[2].plot(list_thresh, data3, colors[0])
    ax[2].scatter(list_thresh, data3, c='k', marker='o')
    ax[2].legend((title3,), loc='upper left', fontsize=8)
    ax[2].set_xlabel('{}'.format(param))

    plt.suptitle('Evolution of the metrics with 4 robots as a function
```

```python
        of {}'.format(param))
    plt.show()

def getSize(rbt, adjacency):
    nx = adjacency.shape[0]
    b = np.zeros((nx,1))
    b[rbt,0] = 1
    output = np.linalg.matrix_power(adjacency, 10) @ b
    return np.count_nonzero(output)

def renameThem():
    old_names = []
    new_names = []

    default_name = os.getcwd() + part
    for f in sorted(folders): #for each folder
        name = default_name + 'logs_'
        if f[0] != '.':
            tab = f.split('_')
            ko = float(tab[KO]) #find infos of current log
            kr = float(tab[KR])
            mu = float(tab[MU])
            m  = float(tab[M])

            if kr == DEFAULT_KR and mu == DEFAULT_MU and m == DEFAULT_M
                and ko == DEFAULT_KO:
                name = default_name + f
            elif kr == DEFAULT_KR and mu == DEFAULT_MU and m ==
              DEFAULT_M:
                name += 'ko_' + str(ko)
            elif ko == DEFAULT_KO and mu == DEFAULT_MU and m ==
              DEFAULT_M:
                name += 'kr_' + str(kr)
            elif kr == DEFAULT_KR and ko == DEFAULT_KO and m ==
              DEFAULT_M:
                name += 'mu_' + str(mu)
            elif kr == DEFAULT_KR and mu == DEFAULT_MU and ko ==
              DEFAULT_KO:
                name += 'm_' + str(m)

            old_names.append(default_name+f)
            new_names.append(name)

    for old, new in zip(old_names, new_names):
```

```
        os.rename(old, new+'/')

if __name__ == '__main__':
    list_params = []
    list_cs = []
    list_z = []
    list_execution_time = []

    thresh_val_default = 150
    nb_rbt = 4

    param_names = []

    for f in sorted(folders): #for each folder
        if f[0] != '.':
            tab = f.split('_')
            param_id = tab[PARAM_ID] #find infos of current log
            param_val = float(tab[PARAM_VAL])
            if param_id not in param_names:
                param_names.append(param_id)

            dir = curr_dir+part+f+'/'
            if dir[0] != '.':
                execution_time = 0.

                rbt_list = []
                for file in sorted(os.listdir(dir)): #for each file in folder
                    if '.csv' in file[-4:]:
                        f = open(dir+file, 'r')
                        lines = f.readlines()
                        f.close()
                        tab = lines[-1].split(',') #find position at convergence = the end
                        rbt = np.array([float(tab[X]),float(tab[Y])])
                        rbt_list.append(rbt)

                        for l in lines[1:]: #discard the title row
                            tab = l.split(',') #find position at convergence = the end
                            temp = np.array([float(tab[X]),float(tab[Y])])
                            if (np.abs(temp - rbt) < 2).all():
                                execution_time = max(float(tab[
```

```
                                TIMESTAMP]), execution_time)
                            break

            adjacency   = np.eye(nb_rbt) #np.zeros((nb_rbt, nb_rbt))
            dist_matrix = np.zeros((nb_rbt, nb_rbt))

            for i in range(len(rbt_list)):
                for j in range(i+1, len(rbt_list)):
                    dist = np.linalg.norm(rbt_list[i] - rbt_list[j])
                    if dist <= thresh_val_default:
                        adjacency[i,j] = 1
                        adjacency[j,i] = 1
                    dist_matrix[i,j] = dist
                    dist_matrix[j,i] = dist

            cs = 0
            for i in range(nb_rbt):
                size = getSize(i, adjacency)
                cs += size**2
            cs /= nb_rbt

            Z = 0
            for i in range(len(rbt_list)):
                for j in range(i+1, len(rbt_list)):
                    Z += dist_matrix[i,j]
            Z *= -1

            print('=== === ===')
            print('NB RBT : ', nb_rbt)
            print("CS : ", cs)
            print("Z : ", Z)
            print("EXEC TIME :", execution_time)

            list_params.append(param_val)
            list_cs.append(cs)
            list_z.append(Z)
            list_execution_time.append(execution_time)

print(param_names)
plotData(list_params[:10], list_execution_time[:10], list_cs[:10], list_z[:10], 'execution time', 'cluster size', 'Z distance', 'ko')
```

```
    plotData(list_params[10:20], list_execution_time[10:20], list_cs[10:20], list_z[10:20], 'execution time', 'cluster size', 'Z distance', 'kr')
    plotData(list_params[20:30], list_execution_time[20:30], list_cs[20:30], list_z[20:30], 'execution time', 'cluster size', 'Z distance', 'm')
    plotData(list_params[30:], list_execution_time[30:], list_cs[30:], list_z[30:], 'execution time', 'cluster size', 'Z distance', 'mu')

# EOF
```