Distributed Intelligence Systems
Homework 1

Louis Rosset, Léonard Pasi, Nicolaj Schmid

April 2022

1 Ant Colony Optimization

1.1

After running the traveling-salesman-problem (TSP) 10 times we get the following values:

- Mean: 468.8908
- Standard deviation (std): 16.1739

1.2

The results of the TSP when using alpha=5 and beta=10 are the following:

- Mean: 464.8712
- Std: 6.1368

Alpha is the weight of the pheromone and beta is the weight of the heuristic function. In question 1.2, the weight of beta was increased with respect to question 1.1. Therefore, the ants are more influenced by the heuristic function e.g. the ants are attracted more by nearby cities relative to cities that have a higher pheromone concentrations. The heuristic function is deterministic and always stays the same. On the other hand, the pheromone has a probabilistic element. Therefore, it makes sense that the std is reduced if beta is increased. The mean value over 10 runs is roughly the same, but individual runs have less variance.

1.3

The result of the TSP when changing alpha and beta linearly is the following:

- Mean: 462.2160
- Std: 8.1399

The values of alpha and beta depend on the iteration indicated by K as follows:

```
alpha = 2*(K/ntours)*A;
beta = 2*((ntours-K)/ntours)*B;
```

Where A and B are the values provided to the function *solvetsp_ant* (in this case $A = B = 5$). We introduced the factor of two (see equations above) such that the mean value of *alpha* and *beta* are equal to A and B respectively. Including this factor of two the performance is slightly increased compared to chapter 1.1. Without the factor of two the results are worse than befor:

- Mean: 494.6485
- Std: 11.2662

In chapter 1.2, we obtained similar results than when varying alpha and beta linearly. This shows how important the fine tuning of A and B is.

1.4

The result of the TSP is the following when pheromone is evaporating:

- Mean: 462.6435
- Std: 8.6726

The evaporation is done in the beginning of each iteration. The first iteration is skipped because there the values are just initialised. The variable *pheromonetrail* which contains the pheromone quantities for all cities is reduced by 10%. The evaporation of the pheromone prevents that certain paths are reinforced too strongly and it makes the system less rigid. Therefore, the ants are more encouraged to explore different paths. Testing the algorithm shows that the performance stays approximately the same (see chapter 1.2). Therefore, an evaporation rate of 10% has only a small influence on the algorithm.

1.5

The result of the TSP with local search is:

- Mean: 448.4010
- Std: 9.9134

The mean value is reduced by approximately 14 with respect to the results of chapter 1.4. This shows that local search increases the performance of the optimization algorithm quiet importantly and it is worth to do the additional computational.

2 Navigation and state estimation

2.1 Navigation

2.1.1

See line 243 in *controller.c*.

2.1.2 Question 2.1.2

The weights used for the Braitenberg controller are the following:

```
l_weight[NB_SENSORS] = {-0.035, -0.05, -0.03, 0.025,
                        0.025, 0.03, 0.05, 0.035};
r_weight[NB_SENSORS] = {0.035, 0.05, 0.03, 0.025,
                        0.025, -0.03, -0.05, -0.035};
```

The weights are chosen such that the robot moves to the right when an obstacle is detected on the left side and that it moves to the left when an obstacle is detected on the right side. A bias speed ($INITIAL\_SPEED = 3$) moves the robot straight if it is not close to any obstacle. The Braitenberg controller is only used when the IR sensor measurements exceed a certain threshold ($IR\_SENSOR\_BRAKE\_SYMMETRY = 90$). This is done because otherwise the robot would not move straight on a free area because of the randomness of the measurement values. An offset is add in situations where the measurement values are exactly symmetric (called *symmetry correction* in the upcoming chapters). This forces the robot to turn in one direction and prevents crashing. Please see chapter 2.1.3 for more explanations and chapter 2.1.4 for a comparison between the Braitenberg controller with and without symmetry correction.

2.1.3

In theory, there could be a situation where the robot approaches an obstacle perpendicular and all sensors measure the same value. This would lead to a crash because the measurements cancel each other out. Therefor, a symmetry correction is added. A symmetry is detected if both front sensors measure a high value at the same time as shown in the code below. In this case, the variable *left_correction* is decreased and *right_correction* is increased by $BRAKE\_SYMMETRY = 2$ respectively. In the case no symmetry is detected, *left_correction* and *right_correction* are divided by two such that it's influence vanishes if the robot moves away from the symmetric situation. Finally, *left_correction* and *right_correction* are added to the values calculated using the Braitenberg weights. We decided to give some memory to the symmetry correction (adding *left_correction* and *right_correction* to it's previous values) because otherwise the robot would oscillate between a correction state and a non-correction state when approaching a corner. This would lead to trajectories that are not smooth. The results when using the symmetry correction are shown in figure 1 for short and long simulations.

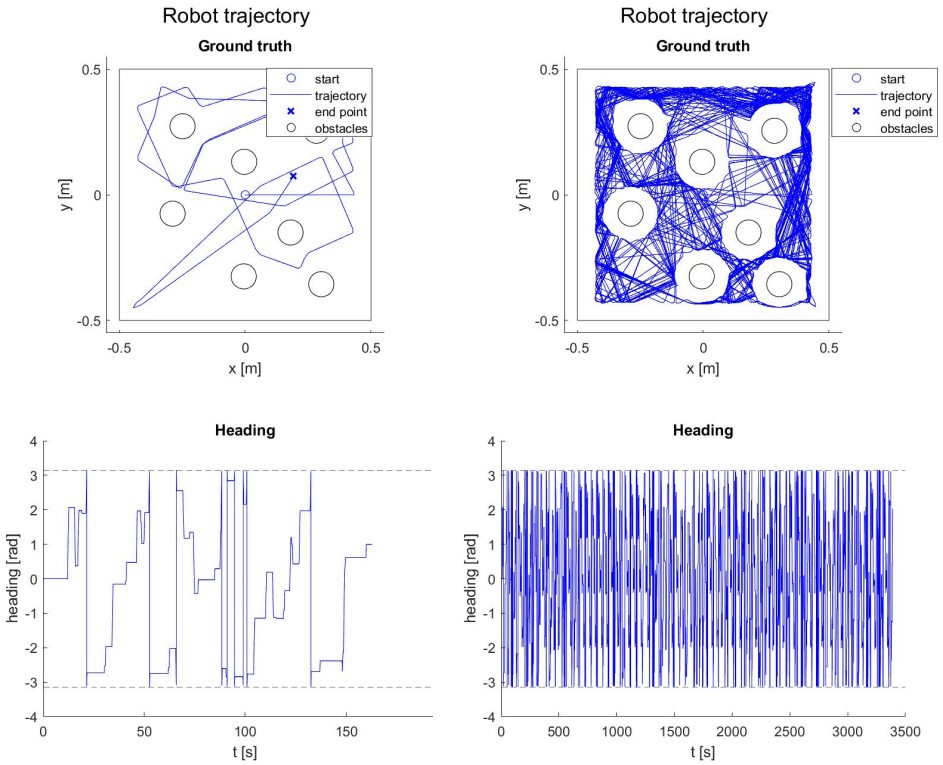


Figure 1: Braitenberg controller simulation: short (left) and long (right)

```
if(_meas.ds_values[0]>IR_SENSOR_BRAKE_SYMMETRY &&
    _meas.ds_values[7]>IR_SENSOR_BRAKE_SYMMETRY)
  {
    left_correction -= BRAKE_SYMMETRY;
    right_correction += BRAKE_SYMMETRY;
  }
  else
  {
    left_correction = left_correction/2;
    right_correction = right_correction/2;
  }

  msl_w =  left_speed + left_correction;
  msr_w =  right_speed + right_correction;
```

2.1.4

Figure 2 shows that the controller can handle symmetric situations like approaching a corner: The left figure shows the implementation of the Braitenberg controller with symmetry correction and the right figure without symmetry correction (see chapter 2.1.3). Interestingly, even when using a pure Braitenberg controller without any additional symmetric correction (right figure 2), the robot is capable to escape the corner without crashing. However, the Braitenberg controller with additional symmetry correction (left figure 2) has a smoother trajectory and less oscillations in heading compared to the Braitenberg controller without symmetry correction. Therefore, it is worth it to add the symmetry

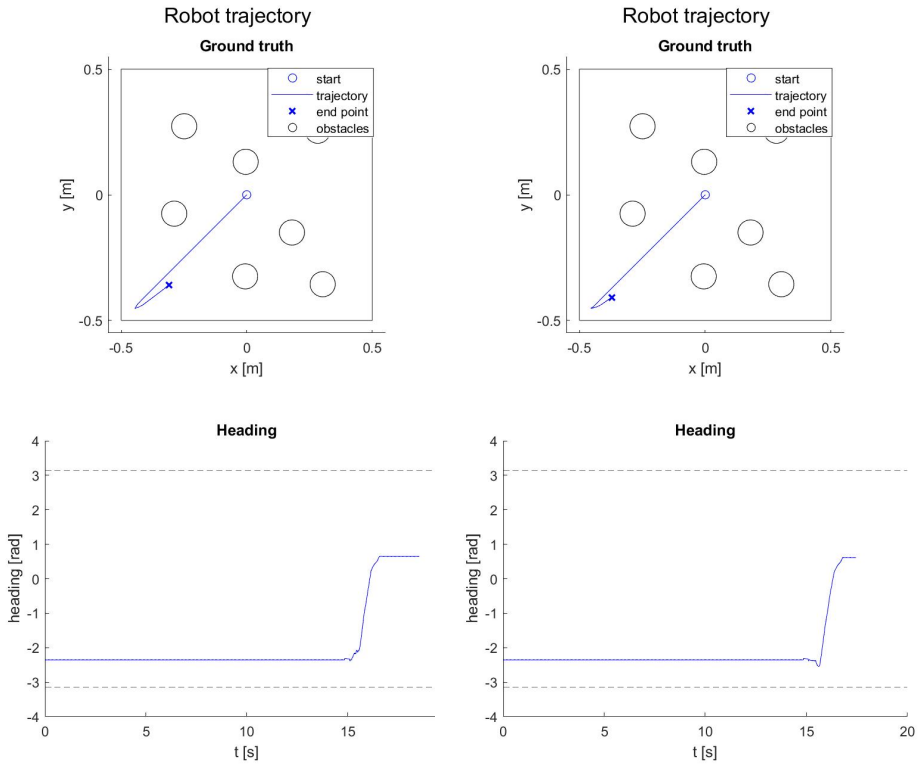


Figure 2: Braitenberg controller with (left) and without (right) symmetry correction

correction. However, if the computational power of the robot is very limited, it could be removed.

2.2 Odometry

2.2.1

The function *metric_scalar* calculates the error as difference between the measured value and the real value. In case of 2D measurements, the function *metric* uses the Euclidean distance. The two functions return also the mean error over the entire measurement series.

2.2.2

See lines 275-307 in *controller.c*.

2.2.3

The acceleration must be integrated once to get the velocity and twice to get the position: Equation 1 uses a second order Taylor expansion and updates the position (x^I_{i+1}) in function of the last position, the velocity and the acceleration (x^I_i, $\dot{x}^I_i$ and $\ddot{x}^I_i$). Alternatively, the system could be approximated by a first order Taylor expansion where the acceleration therm is omitted. The change from the robot reference frame to the global inertial frame is described by the

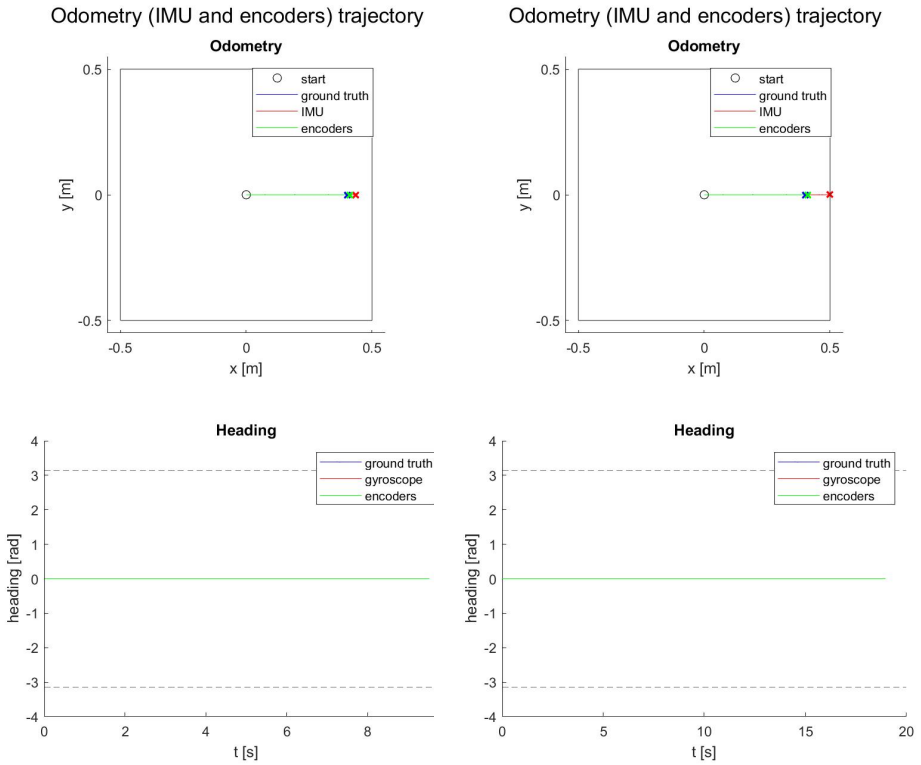


Figure 3: IMU based position estimation: simulation stopped (left) and continued (right) when robot finished its trajectory

equations 2 to 5:

$$\begin{pmatrix} x_{i+1}^I \\ y_{i+1}^I \\ \dot{x}_{i+1}^I \\ \dot{y}_{i+1}^I \end{pmatrix} = \begin{pmatrix} \Delta T & 0 & 0.5 * \Delta T^2 & 0 \\ 0 & \Delta T & 0 & 0.5 * \Delta T^2 \\ 1 & 0 & \Delta T & 0 \\ 0 & 1 & 0 & \Delta T \end{pmatrix} \begin{pmatrix} \dot{x}_i^I \\ \dot{y}_i^I \\ \ddot{x}_i^I \\ \ddot{y}_i^I \end{pmatrix} + \begin{pmatrix} x_i^I \\ y_i^I \\ 0 \\ 0 \end{pmatrix} \tag{1}$$

$$R_R^I = \begin{pmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{pmatrix} \tag{2}$$

$$\begin{pmatrix} \ddot{x}_i^I \\ \ddot{y}_i^I \end{pmatrix} = R_R^I \begin{pmatrix} \ddot{x}_i^R \\ \ddot{y}_i^R \end{pmatrix} \tag{3}$$

$$\begin{pmatrix} \dot{x}_i^I \\ \dot{y}_i^I \end{pmatrix} = R_R^I \begin{pmatrix} \dot{x}_i^R \\ \dot{y}_i^R \end{pmatrix} \tag{4}$$

$$\begin{pmatrix} x_i^I \\ y_i^I \end{pmatrix} = R_R^I \begin{pmatrix} x_i^R \\ y_i^R \end{pmatrix} \tag{5}$$

2.2.4

To estimate the position the IMU measurements have to be integrated twice. This means that any error on the initial measurement leads to a very large error on the estimated position. Left figure 3 shows surprisingly good results: The IMU position estimation corresponds more or less to the ground truth. However, if the simulation is not stopped after the robot finished it's trajectory, then the IMU estimation starts to diverge as seen in right figure 3.

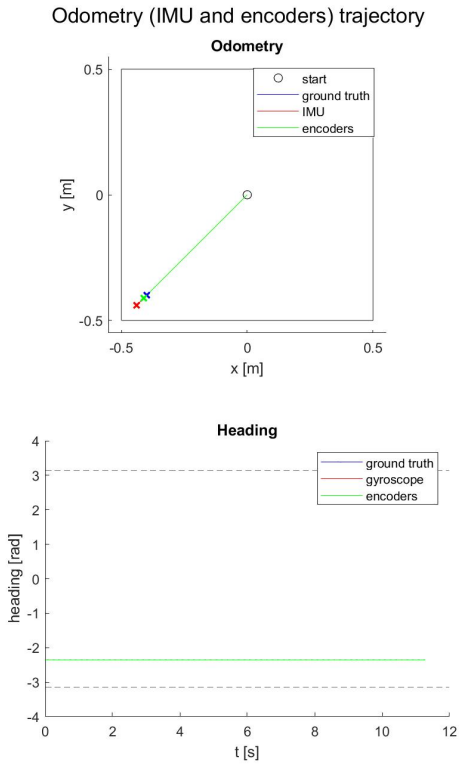


Figure 4: IMU based position estimation when moving towards corner

2.2.5

The result is very similar to the one obtained in question 2.2.4 (see figure 4). Again, there is an error observable because any deviation of the acceleration measurement from the real value is double integrated.

2.2.6

The yaw (angular velocity around z axis: ω) must be integrated once to get the change in orientation (see equation 6). The change of reference frame from the robot to the global inertia frame does not affect the change in orientation. Finally, the orientation is the sum of the previous orientation and the change in orientation (see equation 7):

$$\Delta\theta^I = \Delta\theta^R = \omega * \Delta T \tag{6}$$

$$\theta_{i+1} = \theta_i + \Delta\theta^I \tag{7}$$

2.2.7

As seen in figure 5, the performance of the IMU position estimation is really poor. Also, figure 6 shows that the error between the estimation and the ground truth is very large: Total position error: 7.502212m and Total heading error 0.001106rad. This makes sense because the IMU estimation has an important drift for long simulation and especially during the periods where the robot is moving slowly a large error is accumulated. It seems like the error in heading is much smaller than the one from the position estimation. This is due to the fact that any error in heading leads to even larger errors in positioning, but

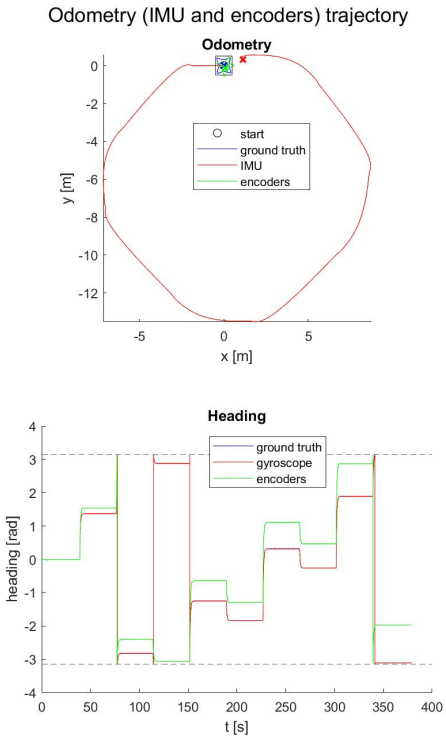


Figure 5: IMU based position estimation - longer trajectory

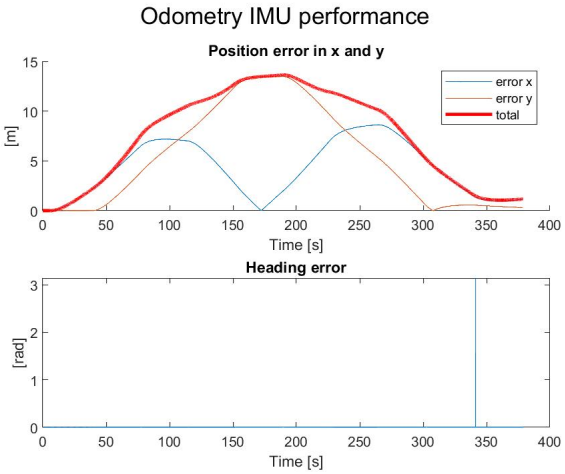


Figure 6: IMU based pose error - longer trajectory

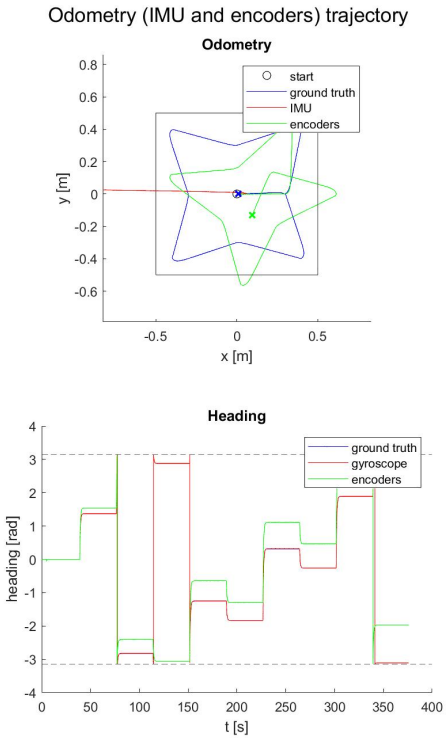


Figure 7: Encoder based position estimation - longer trajectory

also because the yaw is only integrated once to obtain the heading where the acceleration is integrated twice to obtain the position.

2.2.8

See lines 254-272 in *controller.c*.

2.2.9

With help of the encoder measurements (rotation of the wheels in rad) the translation of each wheel can be determined (see equations 8 and 9). Then, the displacement is calculated like in the course (see equation 10 taken from lecture 4 slide 29):

$$\Delta s_r = \alpha_r * R \tag{8}$$

$$\Delta s_l = \alpha_l * R \tag{9}$$

$$d_{i+1} = \begin{pmatrix} x_{i+1} \\ y_{i+1} \\ \theta_{i+1} \end{pmatrix} = d_i + \begin{pmatrix} \frac{\Delta s_r + \Delta s_l}{2} * cos(\theta_i + \frac{\Delta s_r - \Delta s_l}{2*b}) \\ \frac{\Delta s_r + \Delta s_l}{2} * sin(\theta_i + \frac{\Delta s_r - \Delta s_l}{2*b}) \\ \frac{\Delta s_r - \Delta s_l}{b} \end{pmatrix} \tag{10}$$

where $\alpha_{r/l}$ is the wheel rotation measured by the encoder, $\Delta s_{r/l}$ is the displacement of the wheels, R is the wheel radius and b is the inter-wheel distance.

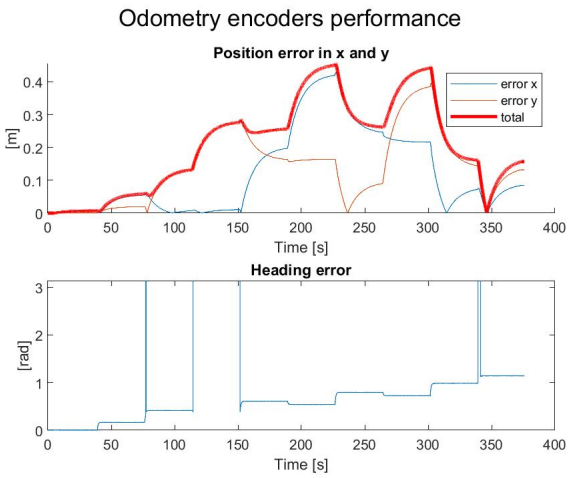


Figure 8: Encoder based pose error - longer trajectory

2.2.10

Compared to the position estimation when using the IMU (see figure 5) the result is much better (see figure 7). Also, the position error as shown in figure 8 is much lower: $totalpositionerror : 0.206m$. This makes sense because encoders do not require any integration and are much better than the IMU for estimating displacement. On the other hand, the error in heading is larger than when using the IMU: $totalheadingerror 1.145rad$. Encoder error and wheel slipping makes localization using uniquely encoders difficult. One way to increase the accuracy would be to use IMU data for rotation or even merge the two using a Kalman filter as explained in chapter 2.3.

2.3 Sensor Fusion

2.3.1

The main function of the controller.c file consists in an infinite perception to action loop. The first step is to acquire data through sensing. The global variable `_meas` is thus updated, with readings from the distance sensors, the wheel encoders, the accelerometers, the gyroscope and the GPS. Additionally, the global variable `_ground_truth` is also updated with the true pose of the robot (position and heading), using the supervisor. The second step is to estimate the pose of the robot with the measurements stored in `_meas`. Several estimations are performed, and they are stored in the global variables `_odo_imu`, `_odo_enc` and `_kalman`. These estimations are performed using odometry on the encoders, odometry on the IMU, and the Kalman prediction with the accelerometers, respectively. Depending on the values of the flags `FUSE_GYRO`, `FUSE_ENC` and `FUSE_GPS`, the Kalman prediction stored in `_kalman` is then updated with the measurements from the gyroscope, the encoders and the GPS. Finally, the action step is performed. Note that only the ground truth is used to set the wheels' speed. Also note that the pose estimation and action steps are skipped initially, in order to calibrate the accelerometers (i.e. compute the constant bias of the accelerometers).

2.3.2

The prediction step of the Kalman filter is given in Eq.11. As specified in the worksheet, the state μ contains both the pose and the velocities of the robot (in the world frame), while the input u contains the measurements from the accelerometers in x and y, projected into the world frame. We thus define the state matrix A and the input matrix B as in eq.12-14.

$$\overline{\mu}_{t+1} = A\mu_t + Bu_t \tag{11}$$

$$A = \begin{pmatrix} 1 & 0 & 0 & \Delta T & 0 & 0 \\ 0 & 1 & 0 & 0 & \Delta T & 0 \\ 0 & 0 & 1 & 0 & 0 & \Delta T \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{12}$$

$$B = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ \Delta T & 0 \\ 0 & \Delta T \\ 0 & 0 \end{pmatrix} \tag{13}$$

While the position and the linear velocities are modified in the prediction step, the heading and the angular velocity are not. Also, note that while the given definition of B is in line with what was done in Lab 3, another possible definition (using a second order Taylor expansion instead of first order) is the following:

$$B = \begin{pmatrix} 0.5 * \Delta T^2 & 0 \\ 0 & 0.5 * \Delta T^2 \\ 0 & 0 \\ \Delta T & 0 \\ 0 & \Delta T \\ 0 & 0 \end{pmatrix} \tag{14}$$

2.3.3

The state should be initialized with the initial state. This is a zero vector if the robot starts at the origin, is not moving initially and is oriented along the x-axis of the inertial frame. If the initial state is known with absolute certainty, then the covariance matrix Σ should also be initialized with zeros.

2.3.4

The process noise matrix R should account both for the integration error and the inherent noise of the accelerometers. As indicated in Lab 3, the integration error is proportional to the time step ΔT. Additionally, the fact that the acceleration

needs to be integrated twice to obtain the new position of the robot justifies the larger process noise in position than in velocity. We arbitrarily chose a factor of 5, as was done in lab 3. Although we are given the standard deviation of the accelerometer measurements, it is not exactly clear how to use this information in combination with the integration error. One possible way is presented in Eq.15. Note that the heading-related variances are high: in fact, since we can't estimate the heading just with the accelerometers, all initial knowledge on the heading should be lost after some time.

$$R = \Delta T \begin{pmatrix} 5\sigma^2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 5\sigma^2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \sigma^2 & 0 & 0 \\ 0 & 0 & 0 & 0 & \sigma^2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{15}$$

with $\sigma = 0.003\, m/s^2$. Errors are assumed to be independent.

2.3.5

See the function `kal_reset` in `kalman.cpp`.

2.3.6

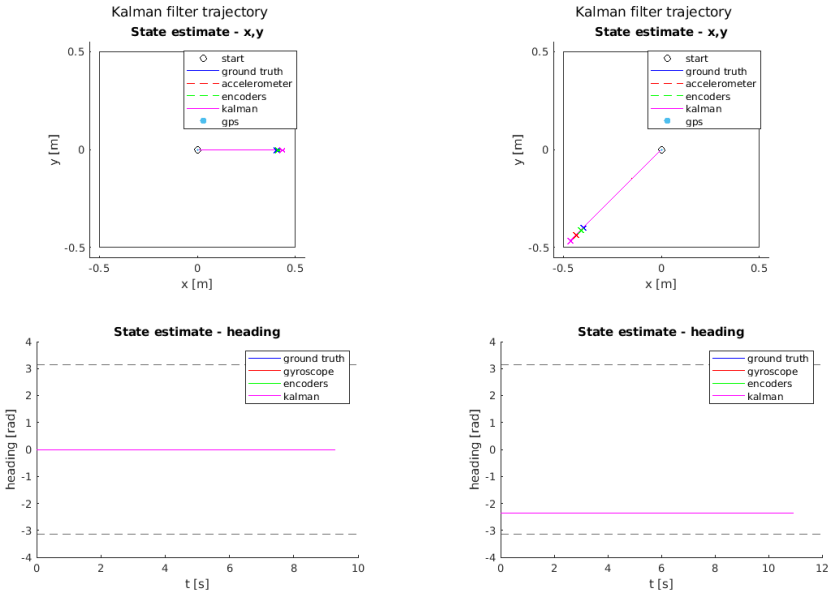
See the function `kal_predict` in `kalman.cpp`.

2.3.7

As expected, the IMU based odometry and the prediction step of the Kalman filter have identical performance in Fig.9(a) (the magenta Kalman curve is superimposed with the red accelerometer curve, which makes the latter difficult to see). In (b), some difference can be seen between the two. This is only because the accelerometers' measurement were debiased for the Kalman prediction, and they weren't for the IMU odometry. The estimated trajectories would otherwise be identical. Interestingly, in this case, debiasing leads to a worse performance, which is all the more evident if the simulation is paused later (i.e. if the robot spends more time in its end position).

2.3.8

The prediction steps relies on two entities: First, it estimates how the current state evolves with time (see equation 12). And second, it adds the influence of any control input to the system (see equation 14). In our case, the motors of the robot are controlled and they change the robots state. This influence is described by the acceleration of the robot. Therefore, the acceleration measurement is included in the prediction step. The other measurements (gyroscope, wheel encoders and GPS) are part of the update step because they measure the



(a) Heading = 0°

(b) Heading = 0°

Figure 9: Pose estimation with Kalman prediction

absolute state (position by the GPS, angular velocity by the gyroscope) or the change of the state (displacement by the encoders).

2.3.9

The computed standard deviation is $6.77e-4\ rad/s$. While computing the heading rate ground truth, special care was taken to deal with the fact that the heading is constrained in the $[-\pi, \pi]$ range.

2.3.10

For the gyroscope update, C, Q and z are defined as follows:

$$C = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix},\ z = \dot{\theta}_{gyro},\ Q = \sigma^2 \tag{16}$$

where σ is the standard deviation of the gyroscope measurements (see previous section), and $\dot{\theta}_{gyro}$ is the heading rate measurement from the gyroscope.

2.3.11

See the function `kal_update_gyro` in `kalman.cpp`.

2.3.12

See the function `kal_update` in `kalman.cpp`.

2.3.13

From Fig.10, it can be seen that the Kalman filter that incorporates gyroscope measurements is better at predicting the heading than the odometry with wheel encoders. In fact, in the bottom part of the figure, the magenta curve of the Kalman filter and the blue curve of the ground truth are superimposed (thus difficult to discern). However, the x and y estimation is worse: the wheel encoders result in better estimation of the position than the Kalman filter, despite the worse heading estimation, because of the inferiority of the accelerometers.

2.3.14

The encoder velocity standard deviations in x, y and θ are: 6.1 mm/s, 7.1 mm/s and $9.386e-2$ rad/s, respectively. The approach is essentially the same as the one in section 2.3.9. For each couple of consecutive estimations (x_{t+1},x_t), a velocity estimation is computed with the equation $\dot{x}_t = (x_{t+1} - x_t)/\Delta t$. This is repeated for y and θ, both for ground truth values and values estimated through odometry. Six velocity time series are thus obtained, resulting in one error time series for each pose component. The standard deviation of each of these error time series is computed. As in section 2.3.9, the computation of the heading rate time series are a bit more delicate, as one needs to take care of those time steps where the heading goes from a neighborhood of $-\pi$ to a neighborhood of $+\pi$ (and vice-versa).

2.3.15

For the encoders update, C, Q and z are defined as follows:

$$C = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}, \; z = \begin{pmatrix} \dot{x}_{enc} \\ \dot{y}_{enc} \\ \dot{\theta}_{enc} \end{pmatrix}, \; Q = \begin{pmatrix} \sigma_{\dot{x}}^2 & 0 & 0 \\ 0 & \sigma_{\dot{y}}^2 & 0 \\ 0 & 0 & \sigma_{\dot{\theta}}^2 \end{pmatrix} \tag{17}$$

where $\sigma_{\dot{x},\dot{y},\dot{\theta}}$ are computed in the previous section, and $\dot{x}_{enc}$, $\dot{y}_{enc}$ and $\dot{\theta}_{enc}$ are the velocity estimates from the encoders (projected in the inertial frame).

2.3.16

See the function `kal_update_enc` in `kalman.cpp`.

2.3.17

The results of the simulation with the encoders measurements incorporated in the Kalman filter are displayed in Fig.11-12. The performance of the Kalman filter is by far superior to both IMU odometry and encoders odometry. A close

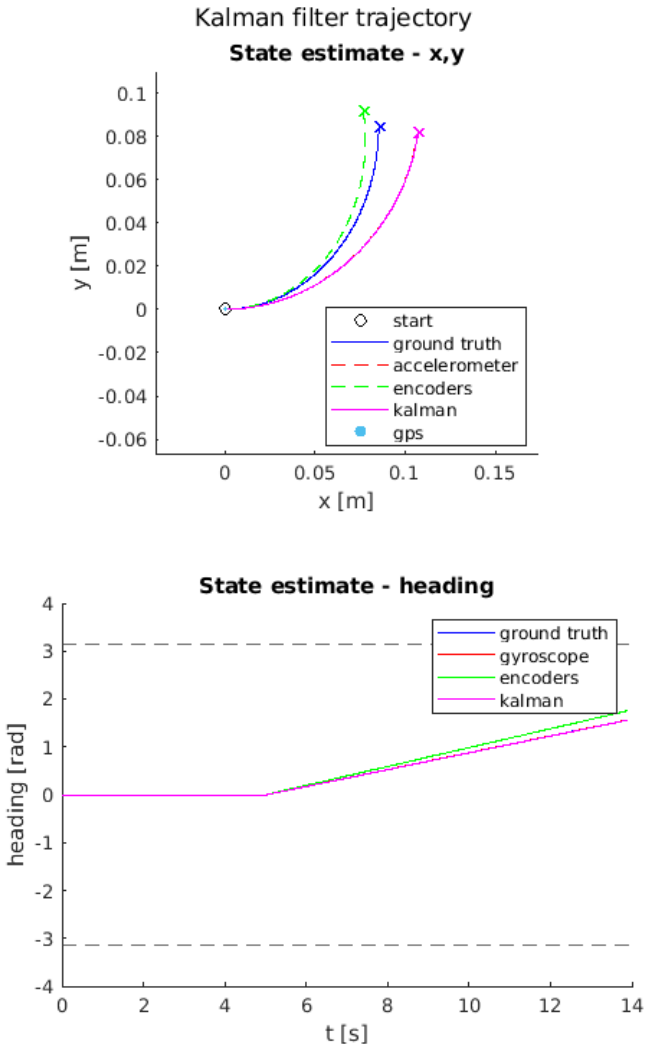


Figure 10: Pose estimation with gyroscope measurement fused to Kalman prediction

inspection of the heading estimate graph reveals however that the gyroscope odometry might be (slightly) better than the Kalman filter at estimating the heading. This can be explained by the fact that the Kalman filter incorporates encoders measurements in the heading estimates, but as shown in section 2.3.13, those were shown to do a worse job than the gyroscope based estimate. Finally,

note that the three large spikes in the heading error that can be seen in Fig.12 are actually just artifacts, due to the constraint on the heading to be in the range $[-\pi, +\pi]$.

As computed in `compute_metrics.m`, part C, the total position error is 0.0077 m and the total heading error is 0.0024 rad. Again, this shows that the Kernel filter is much better than IMU odometry and encoders odometry in position estimation (see sections 2.2.7 and 2.2.10). For heading estimation, while the Kalman filter is superior to encoder odometry, it is slightly worse than IMU odometry.

2.3.18

See the function `controller_get_gps` in `controller.cpp`.

2.3.19

For the GPS update, C, Q and z are defined as follows:

$$C = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix},\ z = \begin{pmatrix} x_{gps} \\ y_{gps} \end{pmatrix},\ Q = \begin{pmatrix} \sigma_x^2 & 0 \\ 0 & \sigma_y^2 \end{pmatrix} \tag{18}$$

where $\sigma_{x,y} = 0.01m$, and (x_{gps}, y_{gps}) is the position GPS measurement.

2.3.20

See the function `kal_update_gps` in `kalman.cpp`.

2.3.21

The results of the simulation with the GPS measurements incorporated in the Kalman filter are displayed in Fig.13-14. As computed in `compute_metrics.m`, part C, the total position error is 0.0077 m and the total heading error is 0.0024 rad. This is equivalent to the performance of the filter without GPS measurements. Perhaps, a significant difference in performance can only be observed for longer simulation times.

2.3.22

The variance on the linear velocities and the heading rate are constant. This is expected, since the gyroscope measures directly the heading rate, while the wheel encoders relate directly to the linear velocities (and the heading rate). In other words, uncertainties do not accumulate thanks to the measurements from the encoders and the gyroscope. However, strangely enough, the variance on the heading rate appears to be zero, which we can't really explain. The variance on the position raises sharply (which is to be expected because of the integration) and drops at each available GPS reading: thus, it has an oscillatory behavior. The variance on the heading increases sharply (linearly) as well, and

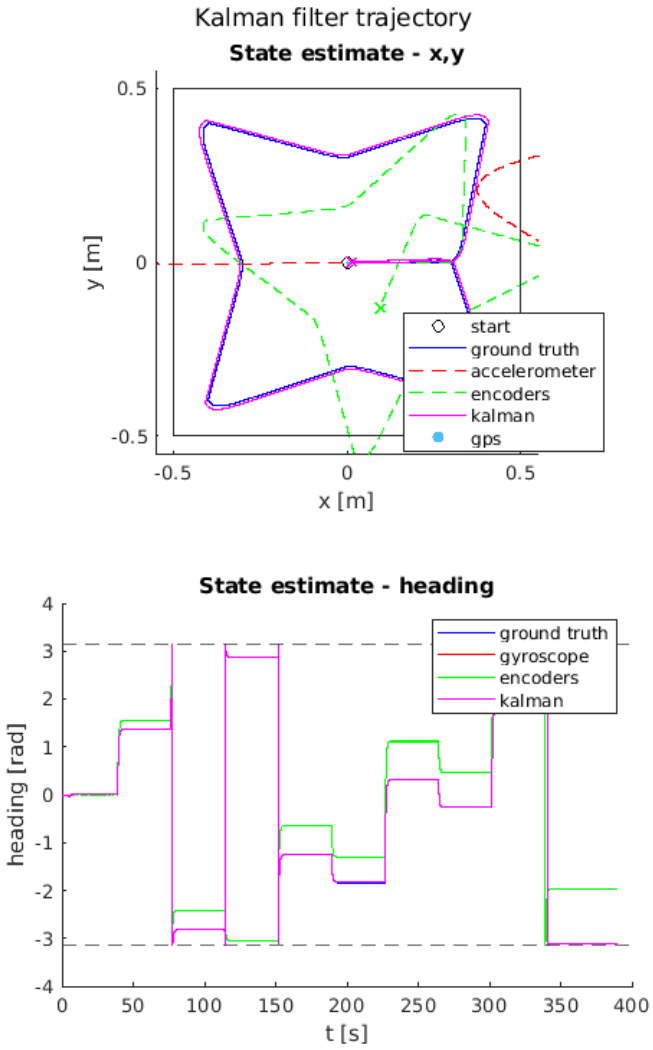


Figure 11: Pose estimation with gyroscope measurement fused to Kalman prediction

there is nothing to stop that. The rate of the increase is however not necessarily very reasonable: after 40 seconds of motion, the standard deviation is already around 2π, i.e. the heading is completely uncertain. Refer to Fig.15 to visualize the evolution of the variances.

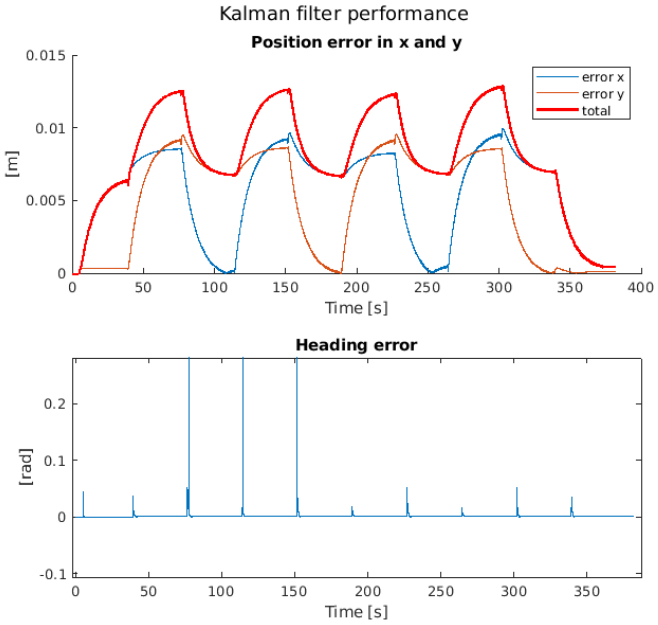


Figure 12: Estimation error with gyroscope measurement fused to Kalman prediction

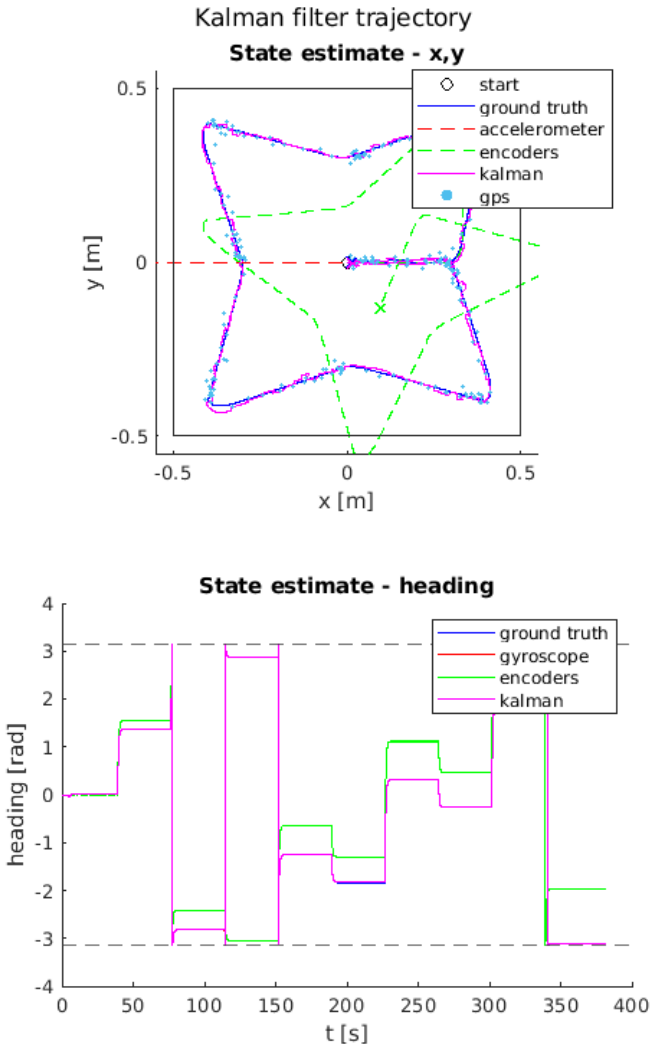


Figure 13: Pose estimation with GPS measurement fused to Kalman prediction

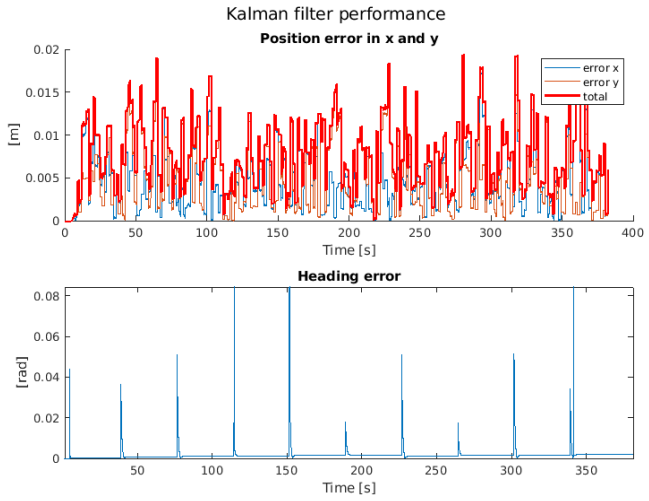


Figure 14: Estimation error with GPS measurement fused to Kalman prediction

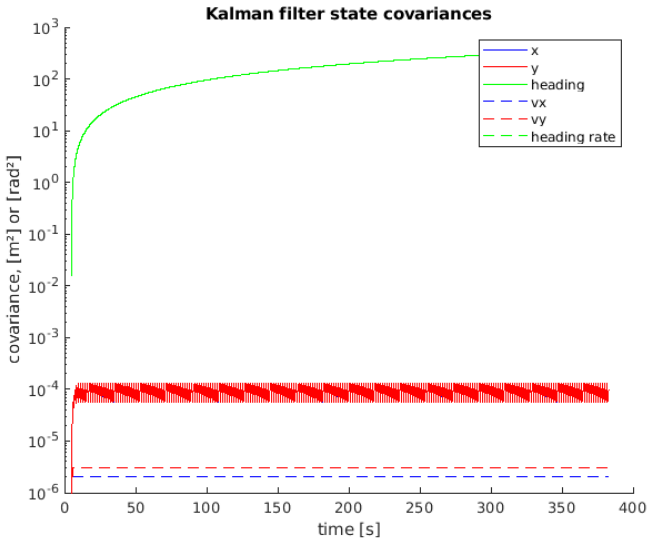


Figure 15: Evolution of the state variances using Kalman filter

3 Collective Movements

3.1 Flocking with Reynold Rules

3.1.1

The variable *robot_id_u* receives and stores the robots identities (ID) from the robots names set in the E-Puck's parameters. The *robot_id* variable saves the robots IDs in a normalised format, to get ID numbers between 0 and FLOCK_SIZE-1. The variables *loc, prev_loc* are arrays of dimensions [FLOCK_SIZE]x[3] to save the positions *X, Y* and the orientation *Theta* of all robots. *loc* represents the current (at t time) position of every robot and *prev_loc*, the previous position (at t-1 time). The variable *speed* is an array of dimensions [FLOCK_SIZE]x[2] to save the speed, in axis X and Y, of all robots computed either with Reynolds or Laplacian methods. Finally, in order to implement a migratory urge towards a certain position, the latter is stored in the vector *migr* of 2 dimensions, with the *X and Y* position.

3.1.2

See the function `compute_wheel_speeds` in `collective.c`. This function is copied from a function implemented in the reynolds.c file from Lab 4 seen during the semester.

3.1.3

For the Braitenberg controller, we took the same weights as in question 2.1.2. The additional symmetry correction described in chapter 2.1.3 is not implemented here. The robots only use the Braitenberg controller if the four IR sensors in the front exceed a certain threshold. To prevent the system from oscillating between a state where the robot uses the Braitenberg controller and a state where the robot uses the normal controller (*reynolds_rules* or *laplacian_formation*), there is a certain margin where it does not change the state. This rule is defined like the following:

```
// enable obstacle avoidance
if((ds_value[0]>85) || (ds_value[7]>85) || (ds_value[1]>90)
                    || (ds_value[6]>90)) {
     *avoid = true;
}
// disable obstacle avoidance
else if((ds_value[0]<75) && (ds_value[7]<75)  && (ds_value[1]<80)
                         && (ds_value[6]<80)) {
    *avoid = false;
}
```

3.1.4

See the function `reynold_rules` in `collective.c`. This function is copied from the function reynolds_rules() implemented in the reynolds.c file from Lab 4 seen during the semester.

3.1.5

See the function `reynold_rules` in `collective.c`. This function is also copied from the function reynolds_rules() implemented in the reynolds.c file from Lab 4 seen during the semester. The weights have been calibrated in order that the flock behaves appropriately.

3.1.6

See the function `reynold_rules` in `collective.c`. This function is copied from the function reynolds_rules() implemented in the reynolds.c file from Lab 4 seen during the semester.

3.1.7

See the function `compute_flocking_fitness` in `flocking_super.c`. The following lines of code have been implemented following the mathematical equations given in the worksheet.

Orientation performance (o):

```
// Computing orientation performance
        for (i=0;i<FLOCK_SIZE;i++) {
                // Angle measure for each robot
                sum_cos += cos(loc[i][2]);
                sum_sin += sin(loc[i][2]);
        }
        o = sqrtf(powf(sum_cos,2)+powf(sum_sin,2))/FLOCK_SIZE;
```

Distance performance (d):

```
// Computing distance performance
         for (i=0;i<FLOCK_SIZE;i++) {
                 d += fabs(sqrtf(powf(loc[i][0]-flock_center[0],2)+powf(loc[i][1]-
         }

        d = powf((1 + d/FLOCK_SIZE),-1);
```

Velocity performance (v):

```
// Computing velocity performance (if migratory urge enabled)}
float angle_flock_v =
atan2f(fabs(flock_center[1]-prev_flock_center[0]),fabs(flock_center[0
]-prev_flock_center[0]));
```

```
float angle_flock_migr =
atan2f(fabs(migry-flock_center[1]),fabs(migrx-flock_center[0]));

float angle_diff = fabs(angle_flock_v-angle_flock_migr);

v = sqrtf(powf(flock_center[0]-prev_flock_center[0],2)
+powf(flock_center[1]-prev_flock_center[1],2));
v = (1/MAX_SPEED_WEB)*fmax((v/(TIME_STEP/1000.))*cos(angle_diff),0);
```

3.1.8

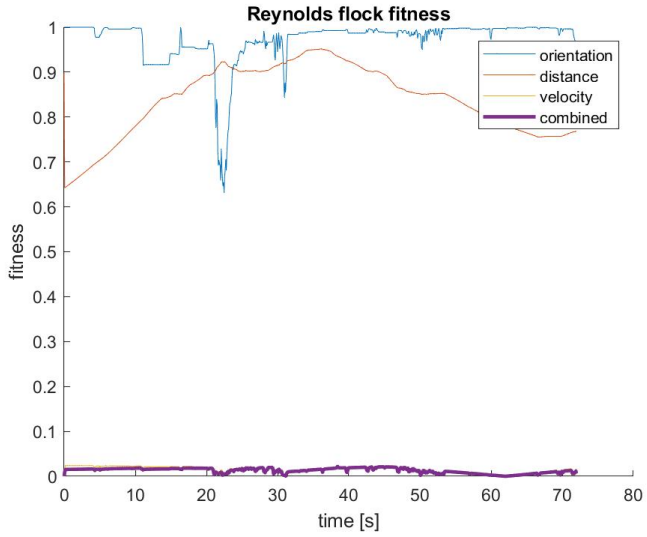


Figure 16: Fitness of Reynold controller with 5 robots

The time needed for the robots to move to all way-points is 1min 14s. This seems to be a reasonable time for the robots to move this distance. The simulation shows that the robots are moving normally as a group in direction of the way-points. Therefore, the Reynold controller is working. Also, the Braitenberg controller prevents the robots from crashing into obstacles. Nevertheless, in some situation a robot is blocked by an obstacle and falls behind the robot group. Then, it takes a moment until the robot could catch up to the group.
The obtained fitness is represented in figure 16 and seems to be really low. In the beginning, the fitness is relatively constant when the robots are moving straight to the first way-point. After around 20s, the fitness drops. This is caused by one of the robots that is blocked by an obstacle and falls behind the group as explained above. Afterwards, the robot catches up with the group and the fitness increases again. After, around 50s the group reaches the second way-point. This is visible in figure 16: First, the fitness drops and then it increases again,

when the robots start moving to the next way-point. The low fitness at around 62s is strange because then the robots seem to move in the right direction and in a well defined group. In general, the fitness could be improved by fine tuning the parameters of the *reynolds_rules controller*.

3.1.9

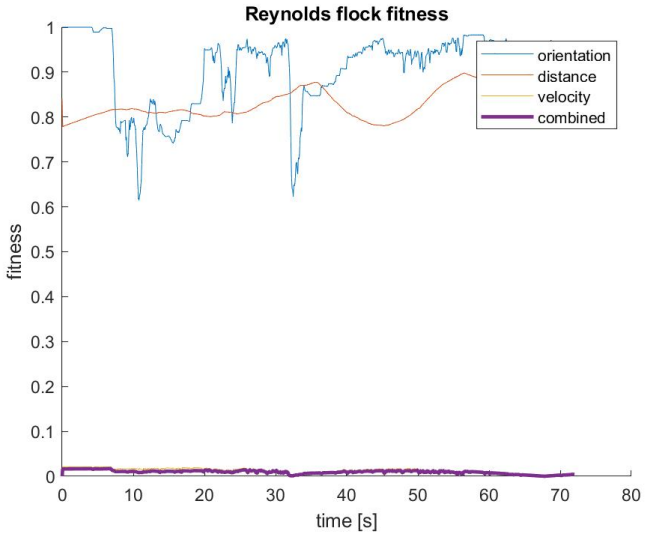


Figure 17: Fitness of Reynold controller with 10 robots

The time needed for the robots to move to all way-points is 1min 19s. It is very similar (5s longer) to the result of question 3.1.8. The fitness is represented in figure 17 and is in general even lower than in question 3.1.8. After around 7s, two robots are blocked behind a large obstacles. This initiates a drop in fitness. After 30s there is again a drop in fitness visible (see figure 17). This is the moment when the robots reach the first way-point. The cause is that the group reaches its first goal and must reorient itself. After, 60s the fitness decreases significantly. This decrease has no obvious reason because the movement of the group looks good and certainly not worse than before.

3.1.10

For this part, we were experiencing problems with deadlocks. Some robots would appear to be stuck by the obstacles, oscillating between the Braitenberg behavior and the Reynold rules. We thus extended the idea explained in section 3.1.3 to include the lateral distance sensors, namely sensor 2 and 5. We were thus able to avoid deadlocks, but the overall behavior is far from perfect (occasionally, one robot would remain stuck).

Radius [m]	Time [s]
0.3	82
0.4	79
0.5	75
0.6	75
10	99

3.1.11

In question 3.1.8, the robots are moving in a group of 5 robots through all way-points. In question 3.1.9, the group size is increased to 10 robots. The total time need is 1min14s and 1min19s for group size of 5 and 10 robots respectively. It makes sense that the smaller group is faster because it is more agile and encounters fewer obstacles. However, we expected the difference to be larger than only 5s. In question 3.1.10, the Braitenberg controller was slightly modified to avoid deadlocks. In the last simulation reported in the table, the neighborhood threshold is set very high, such that essentially the neighborhood concept doesn't apply, as in question 3.1.9. However, the simulation time is different from 3.1.9, which shows that the modified Braitenberg controller has an influence. For smaller (more reasonable) neighborhood thresholds, it seems that slightly increasing the threshold decreases the simulation time. In general, the speed could be improved by adjusting the parameters of the *reynolds_rules controller* and *laplacian_formation controller* and increasing the running speed.

3.2 Laplacian formation

3.2.1

The Laplacian method gives the direction vector at each point in time. From that, a Laplacian matrix can be calculated as follows: $L = D - A$ where A (adjacency matrix) $A_{i,j} = 1$ if i and j are connected. And D (degree matrix) $D_{i,j} = 0 \; \forall i \neq j$ and $D_{i,i} =$ number of edges attached to each robot.

Assuming all weights equal to one, we have the following Laplacian matrix:

$$L = \begin{vmatrix} 4 & -1 & -1 & -1 & -1 \\ -1 & 1 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 0 & 1 \end{vmatrix} \tag{19}$$

And the bias matrix sets the target position of each robot in the formation. According to the reference point showed in blue in the Fig.18, the following matrix has been computed:

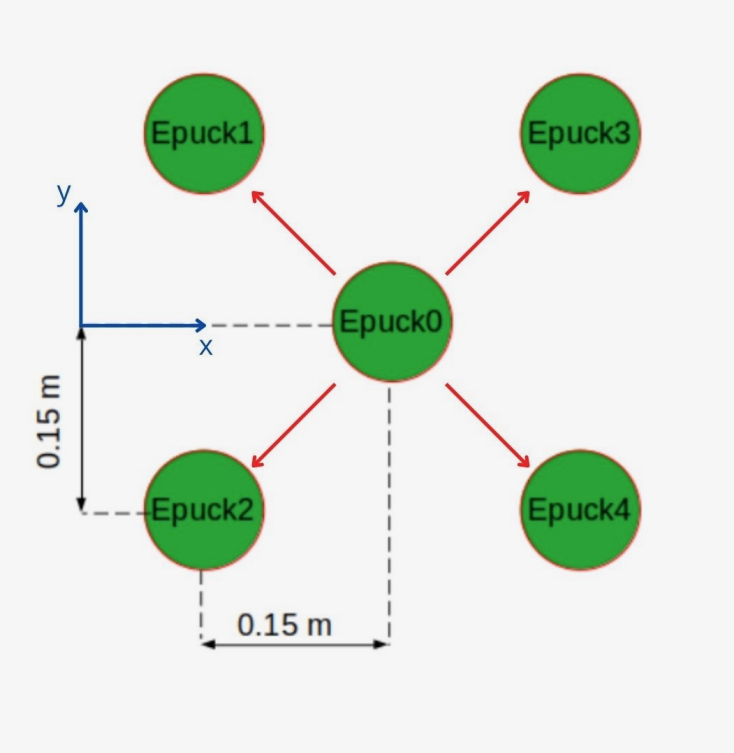


Figure 18: Schema on the formation of 5 robots under Laplacian controller.

$$bias = \begin{vmatrix} 0.15 & 0 \\ 0 & 0.15 \\ 0 & -0.15 \\ 0.3 & 0.15 \\ 0.3 & -0.15 \end{vmatrix} \tag{20}$$

3.2.2

The laplacian_formation function has been implemented in the file collective.c. This function is inspired by the Matlab function `laplacian_formation.m` file from Lab 4, seen during the semester.
Through the `laplacian_formation` function, the robots' speed has been implemented by solving the following equation:

$$\dot{x}(t) = -L(t) * (x(t) - bias). \tag{21}$$

Translating to c language, the value of x(t) is given by x_next and y_next:

```
for (i=0;i<FLOCK_SIZE;i++){
```

```
        x_next += -L[robot_id][i]*(loc[i][0]-bias[i][0]);
        y_next += -L[robot_id][i]*(loc[i][1]-bias[i][1]);
    }
```

3.2.3

The time it took for the flock to go through all waypoints is **84.960 seconds**.

3.2.4

The compute_formation_fitness function has been implemented in the file flocking_super.c. The following lines of code have been implemented following the mathematical equations given in the worksheet.

Distance metric (d):

```
// Computing distance metric
for (i=0;i<FLOCK_SIZE;i++) {
        fit_distance += fabs(sqrtf(powf(loc[i][0]-(loc[0][0]+bias[i][0]),2)
            +powf(loc[i][1]-(loc[0][1]+bias[i][1]),2)));
}

fit_distance = powf((1 + fit_distance/FLOCK_SIZE),-1);
```

Velocity metric (v):

```
// Computing velocity metric
fit_velocity = sqrtf(powf(flock_center[0]-prev_flock_center[0],2)
                +powf(flock_center[0]-prev_flock_center[0],2));
fit_velocity /= D_MAX;
```

With D_MAX, a global constant defined as (TIME_STEP*MAX_SPEED_WEB/1000) or the maximal distance possible per time step.

3.2.5

As displayed in Fig.19, while the distance metric seems quite satisfactory (i.e. the robots are close enough from their target position in the formation), the velocity metric scores very low, which would suggest that the formation goes much slower than it actually could.

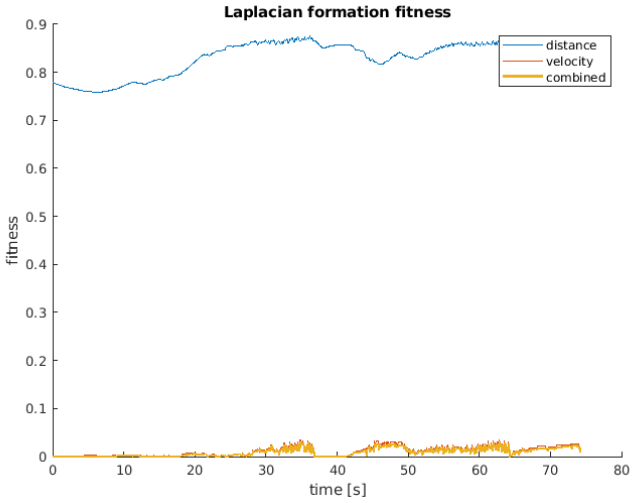


Figure 19: Fitness evolution over one simulation