Artificial Intelligence in Mobile Robots

Lab 3 - Navigation control

By *Tobias L* & *Özgun M*

Lab Assistant *Ali Abdul Khaliq*

Örebro University

September 23, 2016

Student information

Tobias Lindvall
870603-6657
tobiaslindwall@gmail.com

Özgun Mirtchev
920321-2379
ozgun.mirtchev@gmail.com

Report handed in: 2016-09-23

1 Overview

The objectives of this lab is to create a point-to-point navigation control for the ePuck. To do this, a feedback control is needed. A feedback control is summarzied to be a system which reads a current state, and act accordingly to specified rules. In example, if the temperature is too high, a thermostat would correct this by lowering the temperature until it's the correct value. Vice versa if it's too low. Same thing can be applied to the ePuck with its navigation control. A feedback control loop will be reading input from the ePuck to lead it to the specified position. This will be done by using an Achieve-and-exit loop.

This particular loop will:

1. { Measure system output }
2. { Calculate the error to the position }
3. { If there is an error, it will continue. Otherwise it will stop here. }
4. { Then compute a compensation, from the error values, for the ePuck.}
5. { The compensation values will be applied to the ePuck}
6. Repeat from 1.

The main tasks can be divided into first making the ePuck correct the heading, from its current heading to be able to face the target position and then correct its position, move towards the point. After the implementation is done, there will be some tuning to make sure the position of the ePuck becomes more precise.

2 Tasks

2.1 Task 1 - Implementing “GoTo” Divide et Impera (DEI)

To implement the Achieve-and-exit loop in a C-program, there are 2 options of GoTo() algorithms to use. One that is called “MIMO” and another called “Divide et Impera”. In this task, the chosen algorithm was “Divide et Impera”. The implementation was done by help from the provided algorithm description in the slides from lecture 3 (page 105).

Given a target position, the algorithm starts by getting the current position of the ePuck. This is done by calling the update_position function (implemented in lab 2) which stores the current position in global variables. Using the target position, the distance is calculated and converted into errors E_θ, E_{pos}. E_θ is the angle between the facing direction of the ePuck and the target direction. E_{pos} is the distance between the ePuck and the target position. If the angleE_θ is bigger than the margin of error, $delta_\theta$, which is a constant, a compensation movement is initiated which make the ePuck turn in the correct direction. Same procedure for the E_{pos}, which also intiates a compensation movement to make ePuck go forward. This will continue until the ePuck E_{pos} is smaller than another margin of error $delta_{pos}$.

Code for this function can be found in appendix: Divide et Impera.

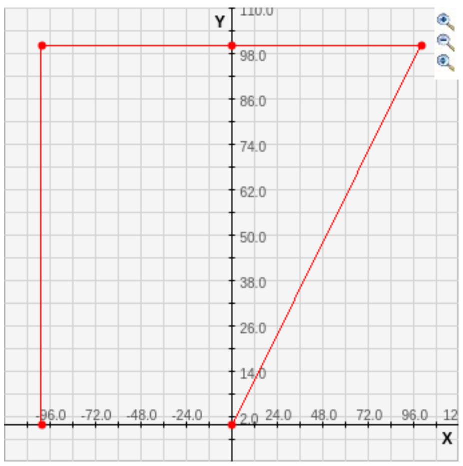
2.2 Task 2 - Tuning the GAIN parameters

To make the ePuck go smoother over a trajectory, tuning of the gain parameters was required. This was tested over simple trajectories after the implementation of the algorithm from the previous task. At one point it was noticed that the ePuck was oscillating too much to reach the target and made it jerk forward instead of going smoothly. It affected the end result and made the ePuck to not end up at quite the desired position. This was an indication that the gain values were too low. New values were tested until the desired behaviour of the ePuck was observed.

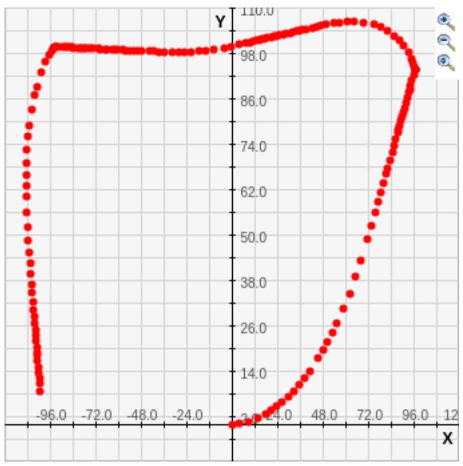
2.3 Task 3 - Implementing a Tracking-procedure

In this task a simple Tracking procedure is going to be implemented. It was done by making a function that takes two parameters. The first parameter is a list with coordinates and the second parameter is the size of the list, to make the function run that many times.

The following values was passed to the tracking function for the ePuck to test it:
$(x, y) = (100, 100), (0, 100), (-100, -100), (-100, 0)$



(a) The expected trajectory.



(b) The performed trajectory of the ePuck.

The position of the ePuck was retrieved by writing the values to a file which was then put into a plot drawer.

Code for this function can be found in appendix: Tracking.

2.4 Task 5 (Optional) - Implementing "GoTo" (MIMO-controller)

Another implementation of Achieve-and-exit loop was the GoTo-function, MIMO-controller. This algorithm retrieves its values slightly differently compared to its counter-part, "Divide et Impera". By using inverse kinematics, it can compute the values for the wheels by using kinematic equations on the margin of error of distance and angle.

Code for this function can be found in appendix: MIMO.

3 Conclusions

3.1 Implementing the algorithms

Divide et Impera

The reason we chose to implement Divide et Impera before MIMO was because we were unfamiliar with with the variables inverse kinematics used. We didn't continue with it as we weren't sure what the variables in the equations represented and how we were to implement them quite yet. The Divide et Impera did not use any kinematic equations and was a straight-forward implementation. We implemented it and by doing that we learned about what each variable represented and how we should tune the gain parameters, which enabled us to more easily continue to implement MIMO.

MIMO

Upon testing the MIMO-controller for the optional task 5, we noticed that the ePuck wasn't turning as expected. It was going opposite to what it should've been doing and we suspected it might be due to the provided equations that we used. We tried to change one of the kinematic equations.

The equations were originally like this:

$$\begin{cases} v_R = v_t + \frac{D}{2}w \\ v_L = v_t - \frac{D}{2}w \end{cases}$$

Since we noticed that the ePuck was going opposite to what it would've normally done, we tried to flip the equations so that the + and the - switched positions:

$$\begin{cases} v_R = v_t - \frac{D}{2}w \\ v_L = v_t + \frac{D}{2}w \end{cases}$$

This enabled the ePuck to follow its trajectory and go to the desired position, at least until we noticed as well that the robot could not move forward properly when it was turned around 180 degrees from it's original position. Example, we put the target position to (-150, 0), a position on the x-axis. The robot turned 180 degrees from its original orientation (0) but could not continue forward since it was turning 180 degrees back to normalize itself and its orientation and then turning back again to get to the target position. This caused it to loop around itself to the target position, while at the same time getting out of course. Not very ideal.

We tried to figure out what the problem could be. After some thinking and calculations we reduced the possible problems down to Eth in the goto-function, this included changing the previous equation back to how it was from the lecture (not flipped).

The equation we used to compute Eth used to be $Eth = Robpos.th - atan(dx, dy)$. We realised that we were subtracting it in the wrong order. The normalization of the angle (meaning that the angle always stays in the interval [-pi, pi]) in update_position caused the theta of the ePuck to change symbol when it was bigger than 180 or -180 degrees, causing it to go around itself as it was trying to reach the target. This resulted in changing the equation to $Eth = atan(dx, dy) - Robpos.th$. Doing this the normalization wouldn't cause the same problem anymore and the ePuck was able to move along the x-axis as it should've been able to from the start.

3.2 Tuning the algorithms

Once we had implemented the algorithms, we had the gains (kp_th, kp_pos) set to 1. From there we started to test different target positions for the ePuck to go and adjusted the values depending on the result. If the ePuck turned too slow to face the direction of the target position, the value of kp_th was too low. Likewise, if the forward motion of the ePuck was too slow, kp_pos was too low. Using these references, we increased/decreased the values after each testing. Since the two algorithms had some differences in how they computed the movement to the target position, the gain values were different for each algorithm.

For Divide et Impera, the movement of the ePuck would be to correct the heading, move forward a bit, and then correct the heading again. This is depending on how high the delta_th value is and how much the difference between it and Eth is. The turning becomes more abrupt than MIMO, since it applies a positive value on a wheel and a negative value on the other.

For MIMO, the movement of the ePuck would be to correct the heading as it was moving, meaning that it would lightly turn while moving forward towards the target position by applying less speed on a wheel than the other. This would result in the trajectory resembling an arc from its start position to the target position.

3.3 Tracking trajectories

We used the MIMO-algorithm because it was more precise in our tests compared to the DEI-algorithm.

From the plot figure where the ePuck moved, the result was satisfying from our point of view because the ePuck didn't avert much from its trajectory and stopped at the right spot according to our delta_pos (10 mm). If we had a smaller delta_pos, the ePuck was stuttering around a lot to get to the exact target spot, which took extra time and in reality made it more inaccurate. To make it more accurate, more advanced rules could be implemented to help it maintain the right course in the trajectory and to stop closer to its target.

Divide et Impera

```
// Divide et Impera
void goto_dei(double xt, double yt){
    double dx, dy, Eth, Epos, D = CRD;
    int vR, vL;

   double Kp_pos = 3.0, Kp_th = 150.0, delta_pos = 10.0, delta_th =
      0.25;

    do {
        update_position(CRD);
        dx = xt - RobPos.x;
        dy = yt - RobPos.y;
        Epos = sqrt(pow(dx, 2) + pow(dy, 2));
        Eth = RobPos.th - atan2(dy, dx);
        printf("dx & dy: %lf %lf\n", dx, dy);
        if (sqrt(pow(Eth, 2)) >= delta_th){
            vR = -(Kp_th * Eth);
            vL = - vR;
            printf("Turned!\n\n\n");
        }
        else {
            vR = (Kp_pos * Epos);
            vL = vR;
            printf("No turn!\n\n\n");
        }
        while (abs(vL) > 1000 || abs(vR) > 1000) {
            vL /= 2;
            vR /= 2;
        }
        while (abs(vL) < 200 || abs(vR) < 200) {
            vL *= 1.2;
            vR *= 1.2;
        }
        printf("Epos: %lf\n", Epos);
        printf("Speed: %d %d\n", vL, vR);
        printf("Eth: %lf\n", Eth);
        SetSpeed(vL, vR);
        Sleep(100);
    } while (Epos > delta_pos);
    Stop();
}
```

Tracking-function

```
void track(Position positions[], int size) {
    int i;
    printf("size: %d\n", sizeof(positions) / sizeof(positions[0]));

    for (i = 0; i < size; ++i) {
        goto_mimo(positions[i].x, positions[i].y);
        Sleep(2000);
    }
}
```

MIMO-controller

```
// MIMO-controller
void goto_mimo(float xt, float yt){
    float dx, dy, Eth, Epos, vt, w, D = CRD;
   int vR, vL;

   float Kp_pos = 8, Kp_th = 40, delta_pos = 10.0;  // double instead
      of float?

    do {
        update_position(CRD);
        dx = xt - RobPos.x;
        dy = yt - RobPos.y;
        Epos = sqrt(pow(dx, 2) + pow(dy, 2));   // abs() does not work
           properly, so sqrt(pow()) is used instead
        Eth = atan2(dy, dx) - RobPos.th;
        if (Eth > PI)
            Eth -= 2 * PI;
        else if (Eth < -PI)
            Eth += 2 * PI;

        vt = Kp_pos * Epos;
        w = Kp_th * Eth;
        vR = vt + D/2 * w;
        vL = vt - D/2 * w;

        printf("Speed before normalization: %d %d\n", vL, vR);
        normalize_speeds(&vL, &vR);

        printf("Epos: %lf\n", Epos);
        printf("Speed: %d %d\n", vL, vR);
        printf("Eth: %lf\n", Eth);
        SetSpeed(vL, vR);
        Sleep(10);
    } while (Epos > delta_pos);
    Stop();
}
```

Normalize_speed-function for the algorithms

```
void normalize_speeds(int* vR, int* vL){
   const int minSpeed = 200;
    const int maxSpeed = 800;

    if (*vL > maxSpeed)
        *vL = maxSpeed;
    if (*vR > maxSpeed)
        *vR = maxSpeed;
    if (*vL < -maxSpeed)
        *vL = -maxSpeed;
    if (*vR < -maxSpeed)
        *vR = -maxSpeed;
    if ((*vL < minSpeed && *vL >= 0))
        *vL = minSpeed;
    if (*vL > -minSpeed && *vL <= 0)
        *vL = -minSpeed;
    if ((*vR < minSpeed && *vR >= 0))
        *vR = minSpeed;
    if (*vR > -minSpeed && *vR <= 0)
        *vR = -minSpeed;
}
```