UNIVERSITÀ DI BOLOGNA

School of Engineering

Master Degree in Automation Engineering

Industrial Robotics

Laboratory Report

Professors: **Claudio Melchiorri**
Gianluca Palli

Group 16:
Donato Brusamento
Mattia Micozzi
Guido Carnevale
Lorenzo Draghetti

(paired with group 17)

Academic year 2018/2019

Contents

List of Figures

Chapter 1

Laboratory of Mobile Robotics

Arnold the robot

Arnold is the mobile robot that we have designed and built to perform the Industrial Robotics Laboratory Race. It is built with parts from a Lego Mindstorm NXT® kit, which includes electric motors used to move the robot as well as directing sensors, both of proprioceptive and exteroceptive kind. Control is performed by the NXT brick. The final race consists in two parts:

- ***Line Following:*** the goal is to follow a path, given by several strips of white tape on dark floor, starting from a square drawn on the pavement (Fig.1.1);
- ***Obstacle Avoidance:*** the goal is to move across an arena in which several obstacles (cardboard boxes) are present, avoiding collisions (Fig.1.2).

1.1 Structure and Hardware

1.1.1 Mechanical assembling

The mechanical assembly of the various parts and components of Arnold is shown from different perspectives in Fig. 1.3. It is a classic *differential drive* mobile robot whose configuration is represented by $[x, y, \theta] \in \mathbb{R}^3$.

This structure has been designed with a castor wheel to achieve mechanical stability and proper weight distribution, and allows plenty of space to place all necessary sensors and proper cable management.
Furthermore, it contains a sufficiently large gap below the brick to allow

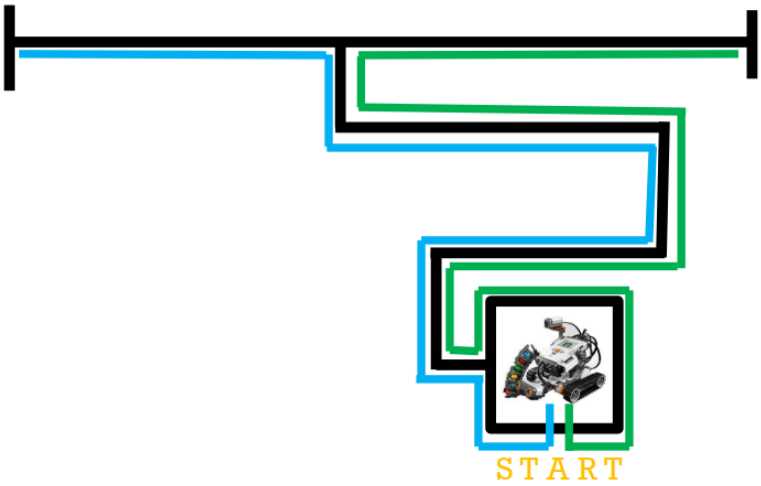


Figure 1.1: Line following section of the race

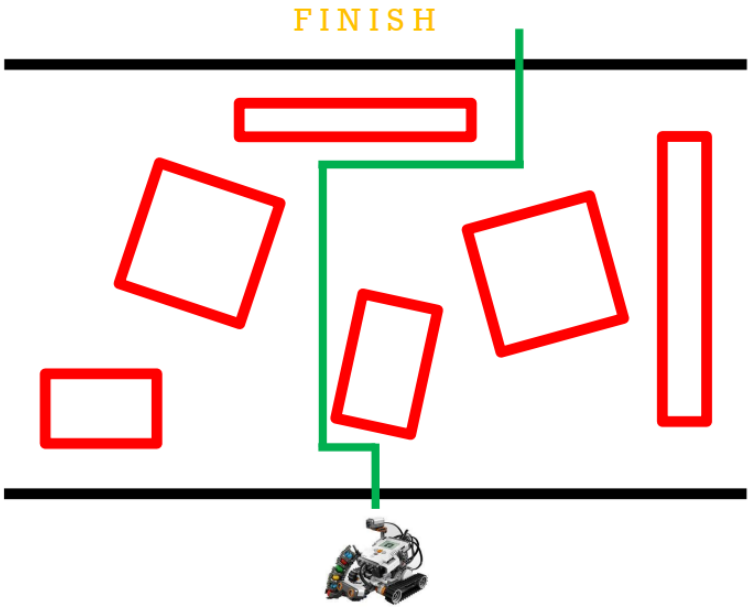


Figure 1.2: Obstacle avoidance section of the race

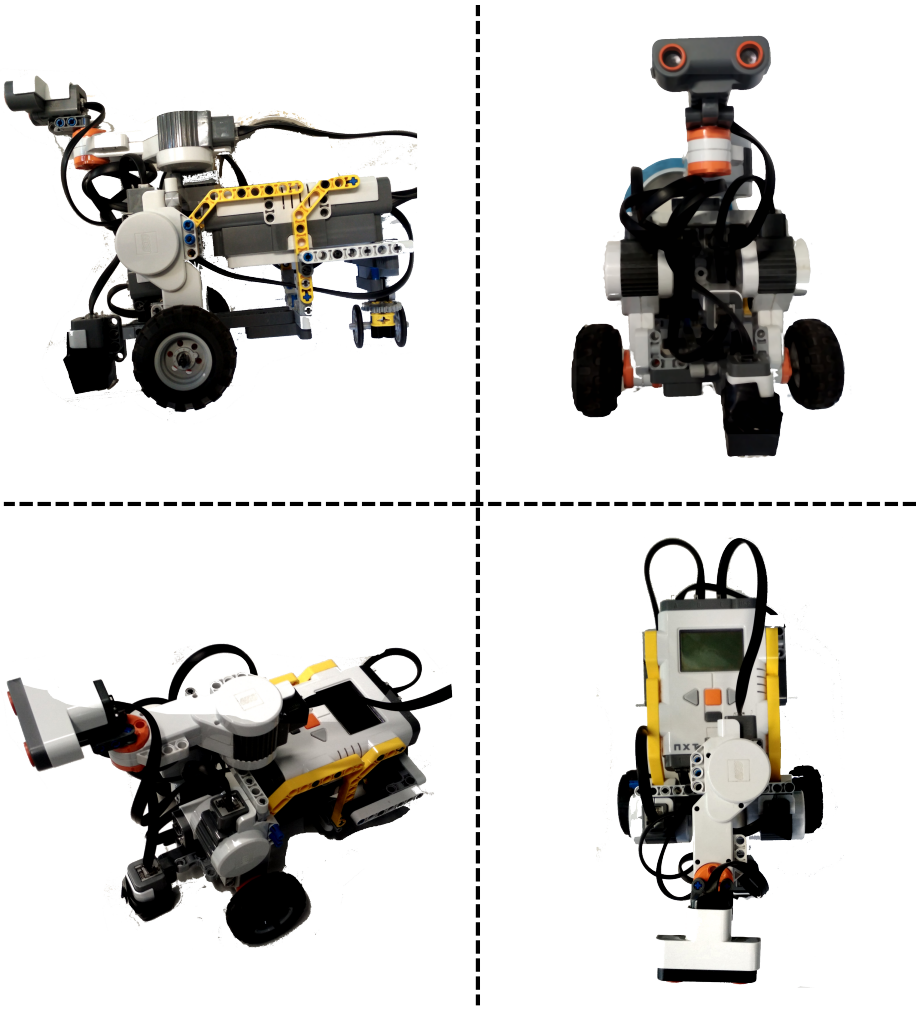


Figure 1.3: Different views of the robot.

further additions (as in the case of the gyroscopic sensor) and easy battery replacement.

1.1.2 Sensors

Arnold uses the following sensors:

- ***HiTechnic Gyroscope:*** measures the *yaw* angular velocity as shown in Fig.1.4, so that it can be used to determine the robot orientation θ in the $\mathbb{R}^3$ configuration space, with a proper placement on the structure;

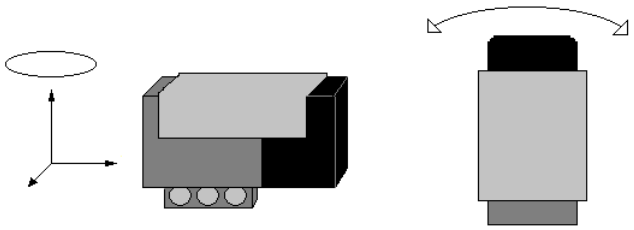


Figure 1.4: HiTechnic gyroscope.

- ***NXT Color Sensor:*** can be used to both measure RGB values of received light (unused in the project), or just the light intensity;
- ***NXT Ultrasonic Sensor:*** determines the distance of an object in front of itself by emitting sound in the ultrasonic bandwidth and measuring the time between departure and reception.

1.2 Algorithm

1.2.1 Threading

The overall algorithm is multithreaded via the Java *Thread* class, though the microcontroller used in the NXT brick (**Atmel AT91SAM7S256**) is single core, so all threads compete for the execution.

Different threads are used for different sections of the race. Grouping them according to their purpose, they are:

- **Line following**
  - LineFollower: implements the PID control by measuring the light intensity;
  - SonarBumpThread: checks for the wall at the end of the line with the *ultrasonic* sensor.
- **Arenas travelling**
  - GyroThread: samples the *gyroscopic* sensor and perform orientation odometry, which consists in a simple integration (discrete sum) as the reconstruction of the robot angle θ from the *yaw* speed is exact;
  - ArenaTraveler: goes from line following arena to obstacle avoidance, by implementing a proportional control with respect to the angle measured by the gyroscope, while checking the starting line.

- **Obstacle Avoidance**
  - GyroThread: same as previous one;
  - SonarThread: constantly rotates the ultrasonic sensor between fixed angles with an auxiliary motor while calculating distance of objects (if any) at those angles;
  - Roamer: controls the driving motors accordingly to the orientation (used for the *attractive potential*) and the distances to objects at different angles (used for the *repulsive potential*).

For each change between sections, the relevant threads are properly exited and collected by the Java garbage collector, with the exception of the gyroscope thread between sections II and III.

While this has allowed an easier management of the code and a more performant algorithm, it did pose some issues, mainly related to the unclear functions for thread management implemented in *LeJos*: methods like

```
Thread.sleep(int ms);
```

often don't work as expected, which results in two issues:

- trying to sleep inside the gyroscope thread for 1 ms (which is the minimum allowed) often results in sleeping for more;
- the *Delay* class methods, which are based on *Thread.sleep*, don't interrupt the calling thread as they should:
  - in higher priority threads this won't result in preemption and thus lower priority ones will never execute
  - while reading distances with

    ```
    UltrasonicSensor.getDistance();
    ```

    all execution is blocked, as the thread isn't properly preempted, which gives significant drift in orientation odometry and delays in the PID controller for line following, of which the initial (very performing) speed had to be reduced.

1.2.2 Line Following

The Line Following part is performed by a PID controller on the light level and a intermittent monitoring of distance from the wall. It can be broken down into 4 parts:

1. calibration of maximum and minimum light values;

2. escape from the square: this is simply done by commanding the robot to go forward until a value of light corresponding to the tape is seen, and to the darker floor subsequently. Once this is verified, start the line following algorithm;

3. follow the line;

4. stop at the wall: the ultrasonic sensor thread is activated every 350ms and checks for the wall ahead (when the distance returned is below 35cm). The length of this period is not ininfluent and a too large value may cause the robot to miss the detection and subsequent collision; however, if this period is too little, because of the reasons stated above about threading in *LeJos* the PID controller would significantly lose performance because of all the non preemptable interruptions.

1.2.3 PID control

The light value is read by the respective sensor and can be a value between 0 and 1023. The system is initialized by taking the average (over 20 samples) of both values returned when over the dark area and the light tape, which respectively compose the calibration values *minLight* and *maxLight*.
Then, an error range is defined (in our case, 150); the error takes any value between $-range/2$ and $range/2$, in a linear fashion with respect to the actual light value, i.e.

$$e = m \cdot light + c$$

where

$$m = range/(maxLight - minLight)$$

and

$$c = range/2 - m \cdot maxlight$$

In this way, the error is equal to 0 approx. where the boundary between tape and floor lies, and any controller trying to push it to 0 will force the robot to stay on that boundary (Fig.1.5).

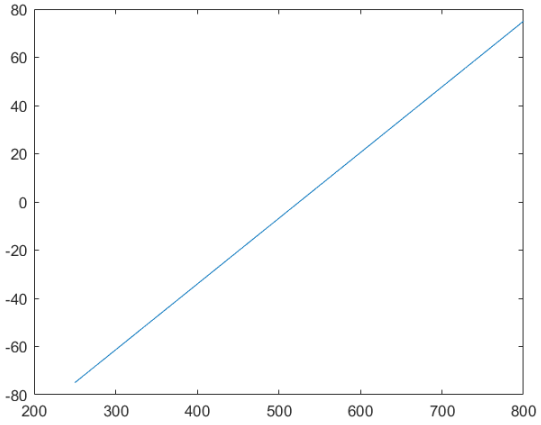


Figure 1.5: Light values from a minimum of 250 and a maximum of 800, with an error range of 150.

The controller is a classical PID one, acting directly on the power of the wheels, with a feedforward action on them to make the robot move forward (Fig. 1.6).

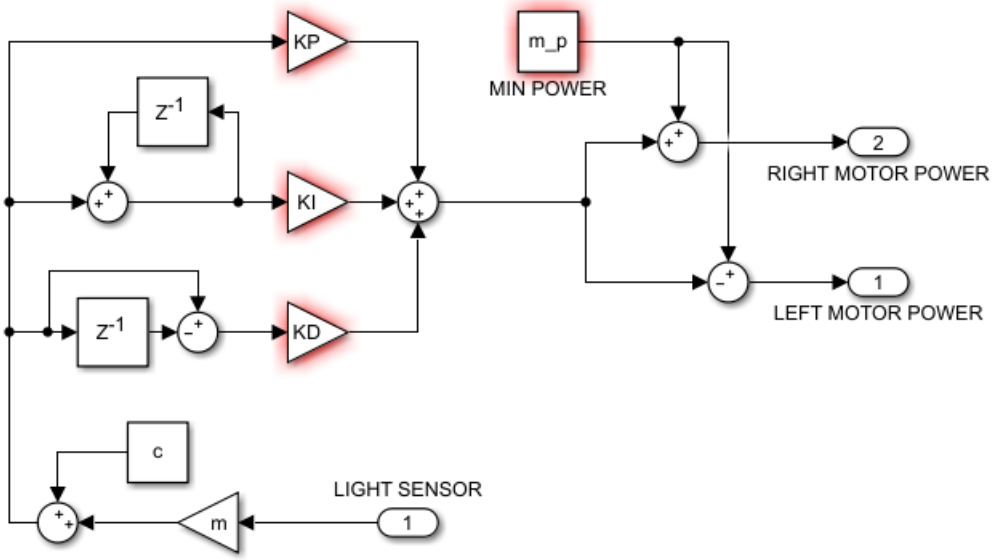


Figure 1.6: Simulink scheme of the robot controller.

The derivative action is simply done by multiplying the differential gain KD by the difference between the previous error and the current one; like-

wise, the integral gain KI acts on the continuous sum of the errors.

For the PID calibration we followed a simple but efficient procedure that exploits the *Ziegler-Nichols method*. At the beginning, we fixed only the proportional gain to a reasonable value that let the robot to lead the error to zero, but with considerable oscillations (we will call this value Kc). Starting from this point, we used the *Ziegler-Nichols* table to tune also the integral and the derivative gains, exploiting the period of oscillation (referred to as Tp) we had under proportional control only and the *time per loop* (dT) needed by the control action.
The values were:

- $KP = 0.6 \cdot Kc$
- $KI = 2 \cdot KP \cdot dT / Tp$
- $KD = KP \cdot Tp / (8 \cdot dT)$

Once we obtained these values, we adapted them to the optimal behaviour we wanted to adopt during the task, tweaking the gains in order to get the best result.

Remarks

We tried to build a simpler controller, such as a PD, PI or even only a proportional one, but we experienced the best solution was to implement a full PID controller. The reason was that each kind of control action gives a useful contribution, i.e.:

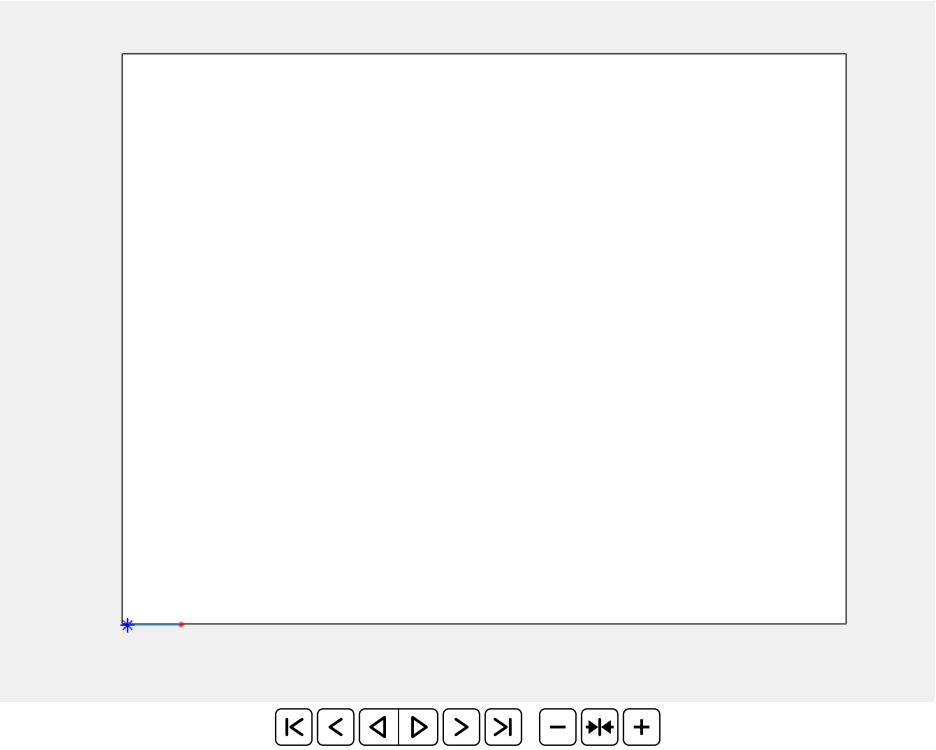
- the proportional gain is the strongest that keeps the error near to zero, holding the robot following the line robustly, but with noticeable oscillations;
- the integral control is of fundamental value on straight lines, since it mitigates oscillations;
- the derivative gain helps the robot in approaching corners fast.

As said, the most important part in parameters calibration was done on field, adjusting the values to improve performances.

1.2.4 Obstacle Avoidance

The obstacle avoidance algorithm is an attempt to port the ***Potential Fields*** approach to a very constrained scenario.
The basic idea is to define an attractive potential represented by the difference between the current measured θ of the robot and the target angle being $\pi/2$, that is the direction to go in order to traverse the arena, if considering $\theta = 0$ when we first stop the PID in front of the wall. The angle is measured by the already mentioned gyroscopic sensor.
Additionally, a repulsive potential is defined by taking into considerations possible obstacle on the course ahead, their distance and angle with respect to the robot.

However, the original formulation of this approach involve the computation of the exact *gradient* at each instant, of the quantities represented by the surrounding environment. While the drifting problem of orientation odometry is not strictly related to this issue, the calculation of the obstacle distances and position is: in fact, not only the ultrasonic sensor is quite slow, but also it can just get the distance of object ahead of itself, and not around.
For this reason we mounted it on a third motor, which switches between different angles, thus trying to mimic a continuous monitoring of the environment, as shown in the animation below.



For each sector we apply a repulsive action inversely proportional to the obstacles distance measured in that sector, while the attractive function tries to maintain the robot along the direction parallel to the walls ($\pi/2$). By neglecting all the distances greater than a certain threshold, along relatively free space sections the robot should proceed along a straight line. Ideally, the control should be:

$$u_{left} = minPower + \nabla U$$
$$u_{right} = minPower + \nabla U$$

with:

$$U = U_{att} + U_{rep}$$

However, due to the issue mentioned previously, we can just approximate U_{rep},

1.3 Performances

1.3.1 Line Following

The results noticed in the line following task are really good, symptom of an accurate and relevant choice of the parameters.

Chapter 2

Matlab Simulations

2.1 Section title

2.1.1 Subsection title

Bibliography