MMKN 30: Project #1

Kasper Ornstein Mecklenburg, Ludwig Jacobsson

February 2014

Objective and main challenges

The objective of the project was to design and program a *Lego Mindstorm* robot to follow a black line on a white board, further to be able to avoid a basic obstacle obtruding the path.
The main identified challenges were the following:

- Identify line-edges
- Regulating wheel-speed
- Identifying an obstacle
- Make basic diversion to avoid an obstacle

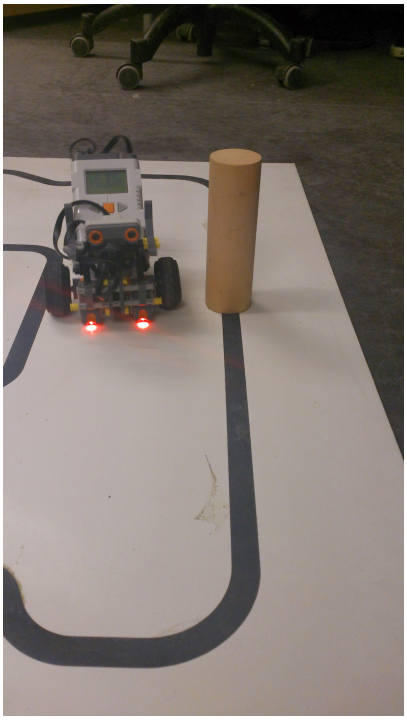

Figure 1: The robot successfully avoiding an object.

Concept and the proposed solution

Our initial solution was based on an "bang-bang" controller. Comparing the two sensor values and when a certain threshold was reached, set the speeds for the wheels. This concept turned out fine but we wanted to improve the result and decided to use a direct adjusting P-regulator. Our hope was that this would get us smoother control and a better time. This solution relies on the fact that the sensors gives a continuous value when passing over the border the line, which they in fact do.
For the obstacle avoidance we used a sonic-sensor to stop the motion when at a certain distance from the obstacle. Then make a pre-programmed diversion around it.

Mechanical design and building elements

We based our design our robot on the pre-designed model found in the *Mindstorms* manual. After some trial and error we made some changes optimizing the robot for our needs.

- **Rear-wheel**
  The rear-wheel's main task is to move freely and allow the robot to rotate on the spot. We never did any long lasting changes on the wheel as it met our needs.
- **Sensor-positioning**
  At first we had the sensors a distance from the wheel-base however we quickly decided to move them

closer. This fitted our steering algorithm better. When the sensors are further apart from each other the steering becomes smoother but it's easier to lose track of the line. The distance between the two sensors was a bit wider than the line's width. The ultra sound detector was placed on top of the other sensors; it's positioning didn't affect to outcome much.

- **Wheel-base**
  Having a broader wheel-base would worsen the ability to rotate and worsen steering. We experimented but chose to keep the original width which was 14 cm.
- **Wheel-size**
  By using larger diameter wheels the top speed increases but worsens acceleration. As full speed could not be obtained without loosing track of the line we kept the wheels with smaller diameter. We tried using a double set of wheels to increase the traction. It looked quite cool but it didn't improve the over-all performance.

(a) First version (b) Second version with sonic-sensor.

Figure 2: Images of nemo-robot

Description of the program and its function

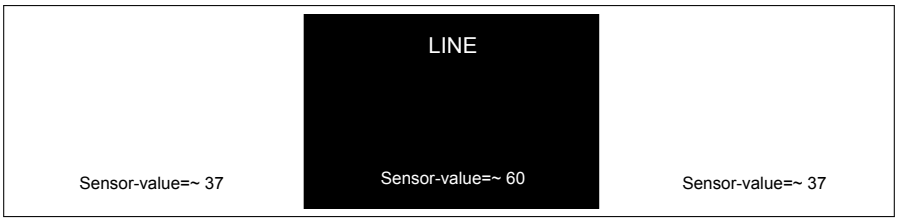

Figure 3: Sensor values on/off the line

”Bang-bang” regulation

Our initial attempt at regulating the wheels speed was using a very basic *”bang-bang” controller*. This type of controller changes the wheel speeds only when a certain threshold is reached. To make this work without specifying a number for the threshold we used comparisons between the left and right sensor combined with the logical less-than operator, see figure 4.

```
if (leftValue < rightValue){
}
    right.setSpeed(-100);
    left.setSpeed(400);
} else{
    left.setSpeed(-100);
    right.setSpeed(400);
}
```

Figure 4: Simple ”bang-bang” regulator

The result from this regulation was fairly successful and we reached lap-times as low as ~ 23 seconds. Still we wanted an improvement.

Proportional control

Our next attempt to control the wheel speeds was using a direct proportional controller. The sensor values from left and right would directly control the wheel speeds. Figure 3 shows the sensor values for on/off the line. At the boundary of the line the sensor-value changes continuously which gave us an opportunity to use this as input to our engines. In order for this to work the interval $37 - 60$ would have to be rescaled to match the input for the engines. Using the unregulated motor classes in the lejOS API the power to the motors could be set directly using function `setPower()`. This means we could set both negative power and positive power, i.e. control both forward and backward movement with one input, the sensor-value. By setting the center of the interval slightly positive the forward and backward movements where adaptive enough to steer effectively. The individual interval adjustment further allowed us to compensate for the slightly faster speed of the right wheel. This value was then adjusted with a coefficient to get a sufficient max speed. Using this regulation and a very short main-loop we got lap-times as low as ~ 16 seconds without obstacle.

```
left.setPower((left.readValue() - 46)*4.3f);
right.setPower((right.readValue() - 48)*4.3f);
```

Figure 5: P-regulator

Main-loop execution time

After some tweaking we realized that the biggest limitation to successfully following the line was making sure the main-loop was fast. The faster the loop, the more often our robot would check where it was in relation to the line. This meant we needed to optimize our code, remove unnecessary function calls and make sure the calls we did to the lejOS API was sufficiently fast. Using the unregulated motors proved to be a good choice since the original motors had a regulator running to make sure both wheels were rotating equally fast, which meant a heavier load for the computer. This was unnecessary since our wheel speeds were adjusted

very often when following the line.

Object avoidance

To detect objects we installed an ultra-sonic sensor on our robot. The sensor would check if there were any object within 15 cm in front of the robot. When at a certain distance: stop, turn and drive around the object. This part of the project demanded some further optimizations to keep the line-following regulation correct. This is since the call to check the ultra-sonic sensor was fairly heavy, especially when using a featureDetector object as recommended in the lejOS API tutorial. In the end we removed the featureDetector and made a call straight to the ultra-sonic sensor function `getDistance()`. To further reduce the loop-time we reduced the calls to `getDistance()` to every 10^{th} loop.
In the `avoidObstacle()` function the regulated motor had to be used in order to make sure the wheel angle rotations were correct and the inner-wheel and outer-wheel speeds had to be set more accurately. The two motor classes can be used in the same program, as soon as a call is made to the regulated motor the regulator kicks in. When the object avoidance subroutine finished the regulation was removed by calling `suspendRegulation()`.

Discussion

Our initial idea was to create a robot which would find its way back to the line if it got lost. This concluded two different ideas:

- having a third sensor in between the two other sensors. By having this setup the robot would be able find it's way back to the line. The middle sensor alerts when the line has been lost.
- having the sensors really close together. If both sensors are on the black line the robot would remember on which side it lost track of the line.

None of the two ideas worked out; we lacked a third sensor and we couldn't put the sensors close enough together. With other hardware these two methods would probably have been possible to implement, not saying anything about their performance.
As mentioned earlier in the text we went from a "bang-bang" regulator to a P-regulator. The speed improved from $\sim$23 s to $\sim$16 s, the regulation was a lot smoother and the amount of code decreased. A winning concept. When we added the ultrasonic-detector the loop speed decreased. This made the robot lose track of the line as the regulation became a lot slower. We researched a bit and after a while we found a faster method to get the distance as well as calling the method every 10^{th} time in the loop. This reduced the time required for the loop and the regulation was once again sufficiently fast.
After implementing the P-regulator with object avoidance we started thinking about implementing a PI-regulator with feedback. We had the idea that with integration and feedback the robot would have memory stored allowing it to find it's way back to the line if it got lost. Unfortunately we ran out of time before we had implemented anything that worked. We think it would work but it's hard to say how fast it would be as $\sim$16 s is already quite fast. We had some ideas of making the object avoidance possible going either way around the track and not so static. But once again we ran out of time.
Over all we are happy with our robot's performance and it was very satisfying when our attempt with the P-regulation worked better than anticipated!

Java code

nxt_line.java

```
import lejos.nxt.*;

/* Lego LineFollowers, we are the robots BY: Ludde & Casper
 */
public class nxt_line{

	public static void main(String[] args) throws Exception{
		LineCar car = new LineCar();
		Button.ENTER.waitForPress();
		car.whiletime = System.nanoTime();
		while(!Button.ESCAPE.isDown()) {
			// Print basic diag on disp
			car.basicDiag();

			// Get State
			int state = car.getState();

			// Set motor settings according to state
			car.motorSettings(state);

			/* SUPER FAST MOTORSPEEDZZ - Set motors to public in Car-class.
			int l = (car.left.readValue() - 45)*5;
			int r = (car.right.readValue() - 45)*5;
			car.m1.setPower(l);
			car.m2.setPower(r);
			*/

			// time the while-loop
			car.whiletime = System.nanoTime();
		}
	}
}
```

LineCar.java

```
import lejos.nxt.*;
import lejos.robotics.objectdetection.*;

/*
 * TODO:
 * - Implement NXTMotor to use non regulated speed for a smoother ride - DONE
 * - Set up a good display diagnosis class    NOGOOD, DONE
```

```
 * - Set both motors method     (l_speed, r_speed) DONE, NOT USED
 * - Optimze getValues, only in diag? - DONE,
 * - Regulate power-constant by checking main-loop exec time
 */

public class LineCar {
    LightSensor left;
    LightSensor right;
    NXTMotor m1;
    NXTMotor m2;
    NXTRegulatedMotor m1_reg;
    NXTRegulatedMotor m2_reg;
    UltrasonicSensor us;
    int currentState;
    int lPower;
    int rPower;
    int lValue;
    int rValue;
    float range;
    long whiletime;
    int count;

    public LineCar() throws Exception {
        LCD.drawString("LineCar started", 3, 2);
        Thread.sleep(1000);
        LCD.clear();

        // Using the non-regulated engines to speed up main-loop.

        this.m1 = new NXTMotor(MotorPort.A);
        this.m2 = new NXTMotor(MotorPort.B);

        this.m1_reg = new NXTRegulatedMotor(MotorPort.A);
        this.m2_reg = new NXTRegulatedMotor(MotorPort.B);
        this.m1_reg.suspendRegulation();
        this.m2_reg.suspendRegulation();


        us = new UltrasonicSensor(SensorPort.S3);

        this.right = new LightSensor(SensorPort.S1, true);
        this.left = new LightSensor(SensorPort.S2, true);
    }

    public int getState() {
        // Two states
        // I. Normal run
        // II. Obstacle detected

        this.count++;
        // Using getDistance on the Ultrasonic sensor instead of
        // a featureDetector speed up main-loop.
        // To further speed up "getDistance" is only called every 10th loop.
```

```
        if (this.count%10 == 0 && this.us.getDistance() < 15) {
            this.count = 0;
            return 2;
        }
        return 1;
    }

    public void motorSettings(int state) throws Exception {
        this.currentState = state;
        switch (state) {
            case 1: // Run as normal
                    // Adaped scale with a proportional control system
                    // Scale allows wheels run both forward and backward.
                    //
                    // TODO: Set the power constant ("4.3f") by checking how
                    // long time the main-loop takes to complete.
                    float l = (left.readValue() - 46)*4.3f;
                    float r = (right.readValue() - 48)*4.3f;
                    m1.setPower(Math.round(l));
                    m2.setPower(Math.round(r));
                    break;
            case 2: // Obstacle
                    this.avoidObstacle();
                    break;
        }
    }

    private void avoidObstacle() throws Exception {
        // Stop unregulated motors
        Motor.A.stop();
        Motor.B.stop();
        // Reset tachocount to make sure the turns are correct.
        Motor.A.resetTachoCount();
        Motor.B.resetTachoCount();
        // Set speed and rotate
        Motor.A.setSpeed(200);
        Motor.B.setSpeed(200);
        Motor.A.rotate(200, true); // 2pi/4
        Motor.B.rotate(-150, false); // 2pi/4

        while((left.readValue()-right.readValue())<3){
            // Set speeds for inner/outer wheels.
            // Inner ~= outer/2
            Motor.A.setSpeed(260);
            Motor.B.setSpeed(600);
            Motor.A.forward();
            Motor.B.forward();

            basicDiag();
            LCD.drawString("In obstacle-while", 0, 5);
        }
        LCD.clear();
        // Rotate to "follow-line"-position
```

```
        Motor.A.rotate(90, true);
        Motor.B.rotate(-90, false);
        Motor.A.suspendRegulation();
        Motor.B.suspendRegulation();
    }

    /*
    * Basic on-screen diagnostics.
    * Omit from main loop for increased main-loop performance.
    */
    public void basicDiag() {
        this.lValue = left.readValue();
        this.rValue = right.readValue();
        this.lPower = m1.getPower();
        this.rPower = m2.getPower();

        LCD.drawString("M1:" + Integer.toString(this.lPower), 0, 0);
        LCD.drawString("M2:" + Integer.toString(this.rPower), 8, 0);

        LCD.drawString("S1:" + Integer.toString(this.rValue),0, 1);
        LCD.drawString("S2:" + Integer.toString(this.lValue),8, 1);
        LCD.drawString("RG:" + Float.toString(this.range), 0, 2);
        //LCD.drawString("AG:" + Float.toString(this.angle), 8, 2);
        LCD.drawString("ST:" + this.currentState, 0, 3);
        LCD.drawString("WT:" + Long.toString(System.nanoTime()-this.whiletime), 8, 3);
        LCD.drawString("TL:" + m1_reg.getTachoCount(), 0, 4);
        LCD.drawString("TL:" + m1.getTachoCount(), 8, 4);

    }

}
```