Faculty of Natural and Mathematical Sciences
Department of Informatics

The Strand
Strand Campus
London WC2R 2LS
Telephone 020 7848 2145
Fax 020 7848 2851

7CCSMPRJ

Individual Project Submission 2014/15

Name: Claudio S. De Mutiis

Student Number: 1413244

Degree Programme: Master of Science (MSc) in Robotics

Project Title: Local Odometry Techniques for a Differential Wheeled Robot Using Bumper's Sensors and Wheels' Encoders

Supervisor: Dr. Daniele Magazzeni

Word Count: 13,408 (using the software Count Anything → http://ginstrom.com/CountAnything/)

RELEASE OF PROJECT

Following the submission of your project, the Department would like to make it publicly available via the library electronic resources. You will retain copyright of the project.

☒ I **agree** to the release of my project

☐ I **do not** agree to the release of my project

Signature: **Date: August 28, 2015**

Claudio S. De Mutiis

Local Odometry Techniques for a Differential Wheeled Robot Using Bumper's Sensors and Wheels' Encoders

MSc Individual Project

Author:
Claudio S. De Mutiis

Supervisor:
Dr. Daniele Magazzeni

Submitted in partial fulfilment of the requirements for the degree of
MSc in Robotics

Department of Informatics
School of Natural & Mathematical Sciences
King's College London

August 28, 2015 [1]

[1] Source for the template of this page: http://www.LaTeXTemplates.com. This template has been slightly modified by Claudio S. De Mutiis.

Local Odometry Techniques for a Differential Wheeled Robot Using Bumper's Sensors and Wheels' Encoders

Claudio S. De Mutiis
Dept. of Informatics
King's College London
London, United Kingdom

ABSTRACT

The main aim of this project is to provide the MIRTO robot, designed and built by a team led by Dr Franco Raimondi (F.Raimondi@mdx.ac.uk) at Middlesex University London, with autonomous navigation planning capabilities. This objective is achieved by using simple kinematic models for the pure rotations[2] and translations[3] of a differential wheeled robot in combination with a "greedy" version of the Bug2 algorithm. Moreover, the robot is required to operate by only being able to gather data from the encoders in the wheels and from the two bumper's sensors. The Java code developed for the high-level odometrical functionalities of MIRTO was implemented in the class *MobileRobot.java* and run on the Raspberry Pi installed on the robot's frame. Furthermore, I wrote a class named *RobotTesting.java* in order to test the basic rotational and translational capabilities of the robot and a class called *RobotMission.java* to plan a "mission" for the MIRTO robot. Most of the methods found in the class *MobileRobot.java* are used in the implementation of the Java method *public void goToGoal()*, which is meant to send the robot autonomously from a start to a goal point avoiding any obstacles on the way. In the *Results and Observations* section of this paper, I discuss the ten tests that I carried out on MIRTO in a scenario where the robot had to travel 150 cm and avoid an approximately rectangular obstacle located approximately 80 cm away from the start point. When testing my navigation algorithms on MIRTO, I observed that the differential wheeled robot, on average, ended up 12.8 cm away from the desired goal location with a standard deviation of 7.6 cm. I also observed an average percentage error and associated standard deviation of 8.5% and 5.1%, respectively, for the final distance from the robot to the goal. However, these values were calculated using the straight line distance between start and goal points, i.e. 150 centimeters.

[2] In this paper, pure rotation refers to a rotation about the central axis of the robot. Pure rotations are never accompanied by translations.

[3] A pure translation is a straight-line translation.

Acknowledgment

I would like to express my gratitude for my supervisor Dr. Daniele Magazzeni, whose support and vision for the project helped me keep on track and stay focused on very specific problems concerning local odometry. I appreciate the fact that Dr. Magazzeni always found time to talk to me whenever I needed help, recommendations and a direction for my project.

Special thanks go to my family and friends for their continued support during hard times and for their precious advices.

Furthermore, I would also like to thank my professors at King's College London for their brilliant teaching and personal feedback. These professors are Dr. Michael Spratling (Computer Vision, Pattern Recognition and personal tutor), Dr. Odinaldo Rodrigues (Artificial Intelligence), Dr. Elizabeth Black (Agents and Multi-Agent Systems), Dr. Hak Leung Lam (Biologically Inspired Methods and Pattern Recognition), Dr. Kaspar Althoefer (Robotics Systems), Dr. Thrishantha Nanayakkara (Real Time Systems & Control) and Dr. Jian S Dai (Sensors and Actuators).

Last but not least, I want to thank my soulmate and life partner, Lin Dan, for being there for me whenever I needed her, during my sad and happy moments. She always gave me the courage and strength to go on and never give up on my dreams. Thank you.

Contents

Nomenclature

θ_{cur} Current orientation of the mobile robot with respect to the positive x-axis in radians.

θ_{new} New orientation of the mobile robot with respect to the positive x-axis in radians.

c Translation constant to be determined empirically in the wall-following behavior

c_{full} Average of the encoders' counts for a 360 degrees rotation.

c_{rev} Encoders' counts per complete revolution

c_{rot} Average of the encoders' counts for the performed rotation.

c_{trans} Average of the encoders' counts for the performed translation.

$c_{wall-following}$ Number of "wall-following behavior" iterations before transitioning to "motion to goal behavior", assuming that the robot has not reached a "leave" point yet. The "greedy" version of the Bug2 algorithm becomes complete in the limit where $c_{wall-following} \to \infty$

$change$ Rotation in degrees the robot is required to perform.

d Distance measured by the sensor/s of a differential wheeled robot

D_0 Distance specified by the programmer in the wall-following behavior

$dist$ Distance in centimeters the robot is required to translate.

$dist_{actual}$ Actual distance in centimeters travelled by the robot after a translation.

$gain$ Rotation constant to be determined empirically in the wall-following behavior

l Distance between the two wheels of a differential wheeled robot

r Wheels' radius in centimeters.

rot_{actual} Actual rotation in degrees performed by the robot.

$Rotation$ Rotation of a differential wheeled robot in the wall-following behavior

$speed\_factor\_left$ Motor's input-speed conversion factor for the left wheel of the robot.

$t_{clockwise}$ Time in milliseconds required for a clockwise rotation of *change* degrees.

$t_{counterclockwise}$ Time in milliseconds required for a counterclockwise rotation of *change* degrees.

t_{trans} Time in milliseconds required to have the robot translate by $dist$ centimeters.

$Translation$ Translation of a differential wheeled robot in the wall-following behavior

v_l Linear speed of a differential wheeled robot

v_L Speed of the left wheel of a differential wheeled robot

v_R Speed of the right wheel of a differential wheeled robot

v_w Angular speed of a differential wheeled robot

x_{new} New x-coordinate of the mobile robot in centimeters.

x_{old} Old x-coordinate of the mobile robot in centimeters.

y_{new} New y-coordinate of the mobile robot in centimeters.

y_{old} Old y-coordinate of the mobile robot in centimeters.

List of Figures

List of Tables

List of Algorithms

1 Introduction

1.1 Project Aims and Objectives

One the main challenges in mobile robotics has always been to find computationally efficient and accurate ways to determine the position of a robot in an environment. It is often desirable to send an autonomous robot from a start point S to a goal point G in order to make it perform a particular task. However, several unpredictable things might happen on the way from point S to point G and the robot might lose knowledge of its location in the environment. For example, the actuators of a mobile robot might produce errors that might lead the robot to a location far away from the desired one. This particular effect gets more and more amplified as the path to be covered by the robot gets longer and longer. Also, the mobile robot might come in contact with obstacles and changes in the environment that are unpredictable. In this case, the robot needs to be able to find an alternative route to the goal point.

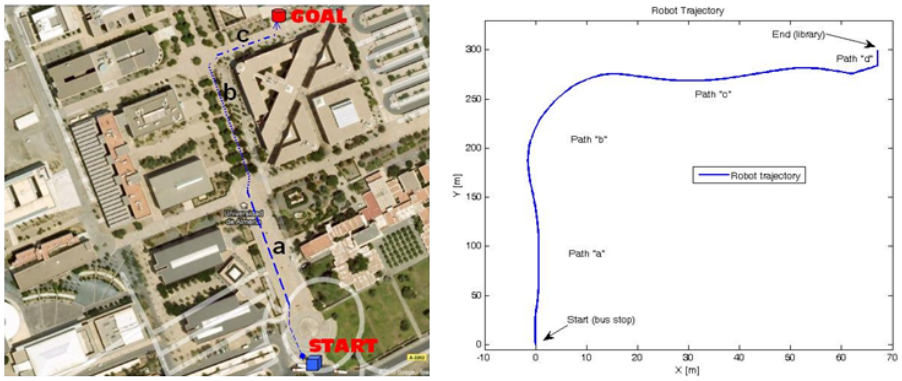


Figure 1: Finding a path from start to goal is one of the main challenges in mobile robotics [1].

The main aim of this project is to provide the MIRTO robot, designed and built by a team led by Dr Franco Raimondi (F.Raimondi@mdx.ac.uk) at Middlesex University London, with autonomous navigation planning capabilities. This objective is achieved by using simple kinematic models for the pure rotations and translations of a differential wheeled robot in combination with a “greedy” version of

the Bug2 algorithm. In particular, according to the modified Bug2 algorithm, the robot switches to motion to goal behavior whenever it has spent too long trying to overcome an obstacle using the standard wall-following behavior. As a result, this modified version of the Bug2 algorithm can allow the robot to reach the goal relatively fast. The "greedy" Bug2 tends to perform very well in scenarios where the obstacles are not U-shaped. However, it is important to mention that this Bug2's variant is not complete and the robot is not always guaranteed to find a solution if there is one.

As far as this project goes, the robot is required to operate by only being able to gather the environment's data from the encoders in the wheels and from the two bumper's sensors. The two encoders, embedded in the wheels, can provide the robot with precious information on its translations and rotations and are therefore essential for determining the current location of the robot and for adjusting its trajectory. On the other hand, the two bumper's sensors allow the robot to detect the presence of obstacles by bumping into them and progressively avoiding them.

The Java code developed for the high-level odometrical functionalities of MIRTO was implemented in the class *MobileRobot.java* and run on the Raspberry Pi installed on the robot's frame. Furthermore, I wrote a class named *RobotTesting.java* in order to test the basic rotational and translational capabilities of the robot and a class called *RobotMission.java* to plan a "mission" for the MIRTO robot. Most of the methods found in the class *MobileRobot.java* are used in the implementation of the Java method *public void goToGoal()*, which is meant to send the robot autonomously from a start to a goal point avoiding any obstacles on the way. In the Results and Observations section of this paper, I discuss the ten tests that I carried out on MIRTO in a scenario where the robot had to travel 150 cm and avoid an approximately rectangular obstacle located approximately 80 cm away from the start point. When testing my navigation algorithms on MIRTO, I observed that the differential wheeled robot, on average, ended up 12.8 cm away from the desired goal location with a standard deviation of 7.6 cm. I also observed an average percentage error and associated standard deviation of 8.5% and 5.1%, respectively, for the final distance from the robot to the goal. However, these values were calculated using the straight line distance between start and goal points, i.e. 150 centimeters. The distance that the robot usually travelled during each experiment was over two meters mainly due to the *wall following behavior* when going around the obstacle. So, the error on the distance travelled by the robot is notably smaller than the one mentioned above.

Finally, while testing MIRTO, I noticed several sources of random and systematic errors. In particular, besides translational and rotational errors, there were also human errors when taking all the required location measurements. Moreover, several other sources of errors included but were not limited to wheels' sliding, the battery

level affecting the operations of the robot, the wheels' slightly different power requirements, the robot not being perfectly balanced and completely stable and the floor not being very even and cleaned.

1.2 Description of the JMirto Robot

The MIddlesex Robotic plaTfOrm (MIRTO) is a differential drive mobile robot that was developed as part of the european project EU FP7 Project SQUIRREL by a team of researchers led by Dr. Franco Raimondi at Middlesex University [2]. Being an open-source and a low-cost robot, MIRTO is ideal for educational purposes and for experimenting with simple navigation algorithms and odometrical techniques[2].
The MIRTO robot, shown in Figure 2, is currently equipped with a Raspberry Pi and an Arduino Uno processor that controls the actuators in the wheels and gets readings from three infrared sensors and two bumper's sensors. The Raspberry Pi, connected to the Arduino through a serial connection, has an SD card to store the Java programs [2] and runs a Linux operating system which can be controlled from any laptop thanks to a WiFi USB dongle. Unfortunately, the WiFi connection tends to be unstable and frequent reconnections to the host are often necessary.

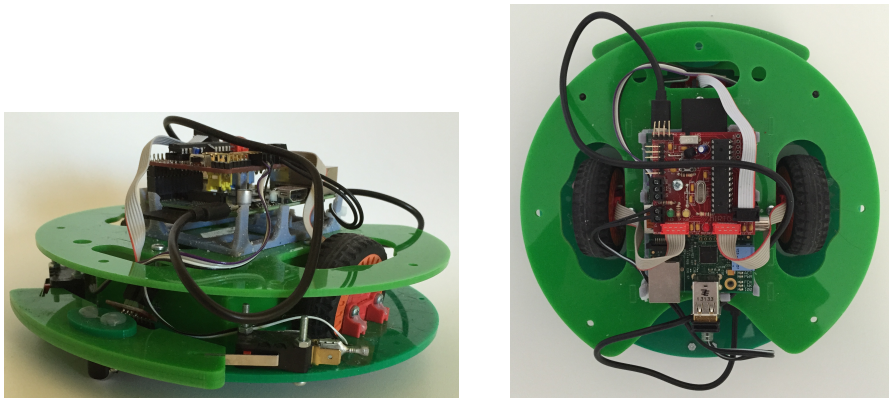


Figure 2: The MIRTO robot developed by a team led by Dr Franco Raimondi (F.Raimondi@mdx.ac.uk) at Middlesex University London [2]

The purpose of the three infrared sensors, located on the bottom surface of the robot and facing the floor, is to allow MIRTO to follow high-contrast lines on the floor. However, over the course of my project, I never made use of these infrared sensors because I wanted to give MIRTO more flexibility and not constrain it to follow lines on the ground.
The two bumper's sensors (see the left picture in Figure 2) are connected to the left and right part of the bumper installed on the frontal part of MIRTO. As soon as

the bumper hits an obstacle at least one of those two sensors "clicks" and sends a voltage to the Arduino Uno microcontroller, letting the robot know that an obstacle has been detected. Regrettably, the bumper does not cover the entire frontal section of the robot. As a result, there are cases, even though not very common, where MIRTO hits an obstacle that is not detected by the bumper's sensors. If ignored, this problem could lead to the robot being stuck or using the obstacle as a fulcrum to perform a rotation not detected by the encoders. I solved this problem by having the algorithm check on whether the encoders have substantially changed their count, i.e. the robot has moved, over two consecutive translation commands. In the case where MIRTO has not translated after each of two consecutive translation commands, my navigation algorithm makes the robot perform an evasion maneuver to get the obstacle out of the way and go back to motion to goal behavior (see the snippet of code shown below or the *public void translate(double dist)* method in the class *MobileRobot.java* which can be found in Appendix A).

```
... CODE ...
private int count_trans = 0;
... CODE ...
public void translate(double dist) {
... CODE ...
if (before_translation.distanceTo(after_translation) < 2) {
    count_trans++;
    if (count_trans == 2) {
        rotate(5);
        translate(-10);
        rotate(30);
        translate(30);
        count_trans = 0;
        if (obstacle_detected()) {
            wall_following();
        } else {
            motion_to_goal();
        }
    }
    if (count_trans > 2) {
        rotate(-5);
        translate(-10);
        rotate(-30);
        translate(30);
        count_trans = 0;
        if (obstacle_detected()) {
            wall_following();
        } else {
```

```
            motion_to_goal();
        }
    }
}else {
    count_trans = 0;
}
... CODE ...
```

The HUB-ee wheels[4] of the MIRTO robot are built with actuators and encoders inside of them. Moreover, they are completely independent from each other and can be controlled separately. Even though these wheels have the same technical specifications, they have slightly different performances and power requirements. For example, in order to make the mobile robot go straight, it is necessary to send different voltages to the wheels. In particular, I empirically found out that the left wheel needs approximately 93% and 95% of the power the right wheel needs in order to make MIRTO perform a pure translation and a pure rotation, respectively. Unfortunately, these values are mostly indicative and can vary even by a couple of units, depending on a lot of factors such us the battery level, the way the wheels are installed on the robot and the mass distribution of MIRTO. Moreover, the two HUB-ee wheels have been installed on the MIRTO robot so that, when given a voltage of the same sign in the method *public void setMotors(int s0, int s1)* [3], the wheels operate in opposite directions. For instance, in order to make the mobile robot perform a pure forward translation, the method *public void setMotors(93, -100)* [3] needs to be called. In particular, in order for the pure forward translation to occur, the ratio between the first and second argument in the method *public void setMotors(int s0, int s1)* [3] should be approximately equal to -0.93 with the first argument being positive and the second argument being negative. Table 1 shows the effects of all of the possible sign combinations in the method *public void setMotors(int s0, int s1)* [3].

Input for *setMotors*	**Effect on the MIRTO Robot**
setMotors(93,-100)	Pure Forward Translation
setMotors(-93,100)	Pure Backwards Translation
setMotors(-95,-100)	Pure Counterclockwise Rotation
setMotors(95,100)	Pure Clockwise Rotation

Table 1: The effects of all of the possible sign combinations in the method *public void setMotors(int s0, int s1)*[3]

[4]http://www.creative-robotics.com/About-HUBee-Wheels

Finally, the Raspberry Pi and an Arduino Uno microcontrollers on the MIRTO robot are powered by a TECKNET rechargeable 9000 mAh power supply (see Figure 3), which is enough to operate the robot for several hours [2]. However, as I have previously mentioned, the battery level of the power supply can have a significant effect on the operations of the robot. The TECKNET battery pack can be easily recharged by connecting it to a laptop via USB cable.

Figure 3: The TECKNET rechargeable 9000 mAh power supply used by the MIRTO robot [2].

1.3 Background and Literature Survey

As I have already mentioned in the *Project Aims and Objectives* section of this report, one the main challenges in mobile robotics has always been to find computationally efficient and accurate ways to determine the position of a robot in an environment.
Cutting-edge research on Extended Kalman Filter, carried out by Muraca, P. et al., has put forward the idea that it is possible to localize a robot even if the measurements coming from its onboard sensors are intermittent [4]. Furthermore, Pinto, A.M.G. et al. suggested that by combining low-cost infra-red sensors, a map-matching method and EKF, it becomes possible to accurately localize small mobile robots [5]. In fact, their paper's abstract states that "a particle filter based on Particle Swarm Optimization (PSO) relocates the robot when the map-matching error is high" [5]. Several other researchers studied Extended Kalman Filters and PSO in order to propose better solutions to the mobile robots' localization problem.These localization methods can be computationally expensive depending on the processor a specific mobile robot is equipped with. Also, map-matching techniques can hardly work in a dynamic environment that changes with time. Therefore, some researchers started turning their attention to computer vision techniques and landmarks identification as an alternative approach to the localization problem in mobile robotics.
In 2014, J.L. Ortega-García et al. developed a new method for mobile robots to follow a path "conformed by straight lines, rotation angles and landmarks" [6] (see Figure 4 below).

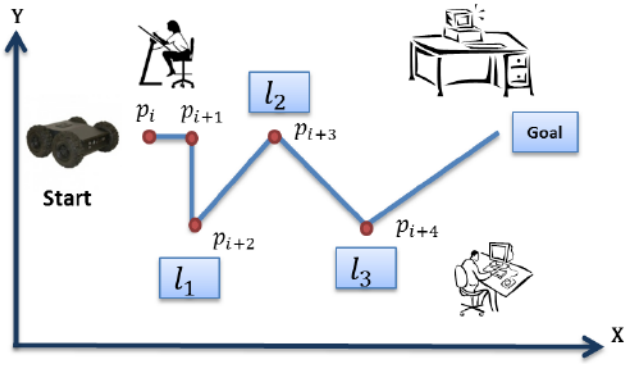


Figure 4: Trajectory execution as explained by Ortega-Garcia et al. in "A new method to follow a path on indoor environments applied for mobile robotics" [6].

The navigation algorithm researched by J.L. Ortega-García et al. [6] is remarkably simple and very successful under some very specific assumptions made on the environment. J.L. Ortega-García et al. worked with a differential drive robot that was supposed to reach a goal location autonomously. In particular, the mobile robot was constantly being driven by landmarks which were placed on its path so that they would always appear right in front of the robot's camera. In other words, every landmark was associated to an angle that would lead the mobile robot to the next landmark and so on. In a more realistic situation, a mobile robot might not always have landmarks on sight. Furthermore, when the robot approaches a landmark, it might not have a frontal view of the landmark object. Hence, a mobile robot should be able to deal with situations where the landmarks appear at different angles or do not appear at all. Moreover, in case a mobile robot gets lost, it should be able to determine its location with reasonable accuracy up until a new landmark is on sight and the robot can correct its position. Therefore, odometry techniques that use onboard sensors and do not rely on landmarks become very important.
In this paper, I argue that it is possible to localize a differential wheeled robot with reasonable accuracy relying solely on the bumper's sensors and encoders in the wheels. Naturally, longer distances to be travelled by the robot cause smaller accuracies and greater errors. In the case of a "cheap" robot like MIRTO, it is possible to localize the robot with a good accuracy for distances on the order of a few meters, including detours caused by potential obstacles on the way [5].

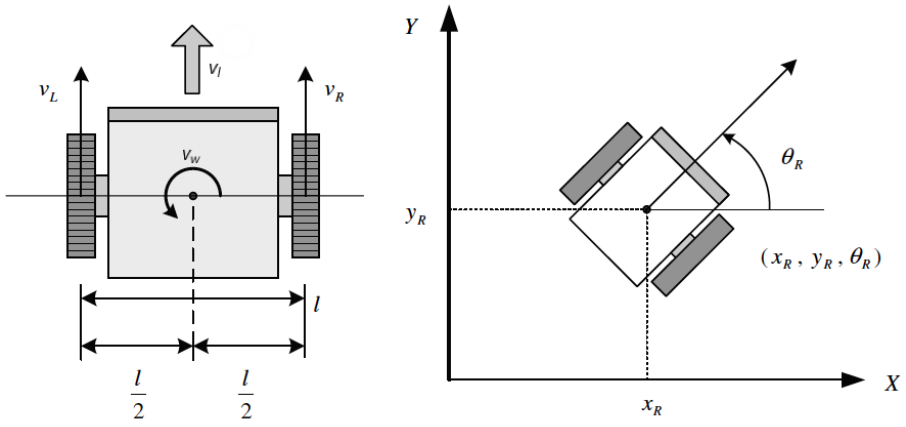
[5]MIRTO might be able to determine its position reasonably well even for distances greater than 10 meters. However, it has not been tested in such scenarios

2 Background Theories

The basic kinematics models for differential wheeled robots are nicely discussed by Han et al. in "A Precise Curved Motion Planning for a Differential Driving Mobile Robot" [7] (see Figure 5 below). If we call the velocities of the right and left wheel v_R and v_L, respectively, the linear velocity v_l for pure translations and the angular velocity v_w for pure rotations will be given by Equations 1 and 2, respectively [7].

$$\text{Linear Velocity: } v_l = \frac{v_R + v_L}{2}[7] \tag{1}$$

$$\text{Angular Velocity: } v_w = \frac{2 \cdot (v_R - v_L)}{l}^{6}[7] \tag{2}$$



(a) Velocity of a mobile robot. (b) Representation of robot position.

Figure 5: Kinematic models for the linear and angular speeds of a differential wheeled robot (a) [7]. Representation of the robot position on a Cartesian plane (b) [7]. This figure has been slightly modified and is different from the one provided by the source.

[6] l is the distance between the robot's wheels. In the case of Mirto, $l = 11.5$ cm.

We can observe that, if the velocities v_R and v_L are equal in absolute value and sign, we have $v_l \neq 0$ and $v_w = 0$. As a result, the robot will perform a pure translation, which will be forward if the signs of the wheels' velocities are positive and backwards otherwise. On the other hand, if the velocities v_R and v_L are equal in absolute value but have opposite signs, we are left with $v_l = 0$ and $v_w \neq 0$. In this case, the robot would perform a pure rotation, which will be counterclockwise if $v_R > 0$ and $v_L < 0$. Similarly, the pure rotation will be clockwise if $v_R < 0$ and $v_L > 0$, while still having $v_R = -v_L$. In the event that the two wheels' velocities are not equal in absolute value, the robot will perform a more complex curved trajectory that will be a mix of a rotation and a translation.
The next piece of the puzzle in order to allow a differential wheeled robot to explore and eventually get from point A to point B in an unknown environment is the navigation algorithm to be used. Plenty of research has been done on navigation algorithms using only sensors onboard mobile robots. However, I am going to focus my attention on a variant of the Bug2 algorithm, presented during a Stanford lecture [8] (see Algorithm 1, shown below).

Algorithm 1 Bug2 algorithm (variant) [8]

```
1: procedure BUG2(VARIANT)
2: Repeat:
3:     Head toward the goal along the goal-line
4:     if the goal is attained then
5:         stop
6:     if a hit point is reached then
7:         follow the obstacle's boundary (towards the left) until the goal-line
```

In this variant of the Bug2 algorithm, the robot starts heading for the goal location until it either reaches it and stops or hits an obstacle. As soon as the obstacle is detected, a hit point is recorded in the memory of the robot. The "hit"/"leave" points are the intersections between the start-goal straight line and the boundaries of all of the obstacles in between start and goal points. Hence, once the robot has recorded the "hit" point in its memory, it starts following the boundary of the obstacle until the goal-line [7] is crossed at a "leave" point that has not been visited yet [8]. Then, the robot starts moving towards the goal along the goal-line again and repeats the same process for any obstacles on its way until it reaches its destination. The Bug2 algorithm tends to work very well when applied on scenarios with relatively small and simple-shaped obstacles (see Figure 6 on the

[7] Straight line connecting start and goal points

next page) but performs quite poorly when used in maps with "maze-like" obstacles or similar.
There are a few techniques in order to allow a mobile robot to follow the boundary of an obstacle. All of the wall-following algorithms are strongly dependent on which onboard sensors are available and where exactly are they installed on the robot.

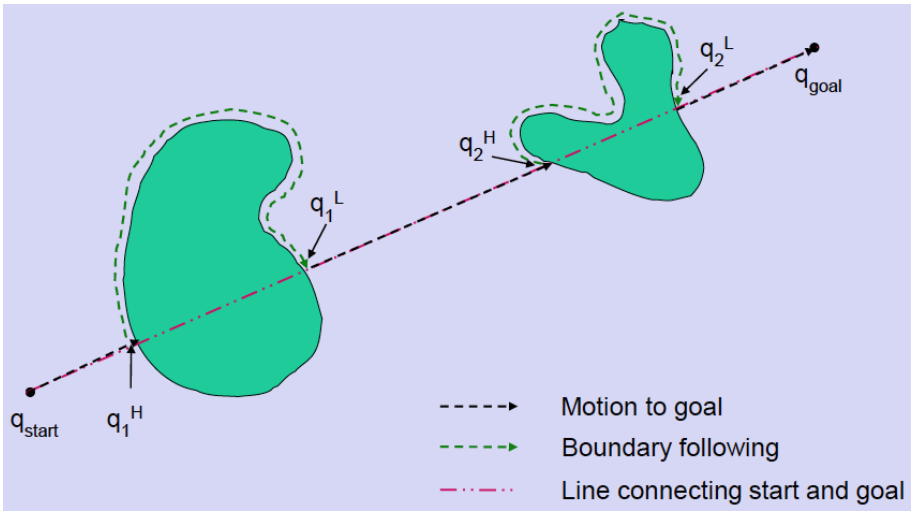


Figure 6: An example of the Bug2 algorithm [9]

Mobile robots using any kind of distance sensors to detect the obstacles can often[8] use a very simple feedback control system to follow the obstacle's wall and keep themselves at a specified distance. This "boundary following" behavior can be achieved using the following model[9]:

Behavior Range-Follow
– $Rotation = gain * (D_0 - d)$
– $Translation = c$
End Range-Follow [9]

where D_0 and d are the specified distance and the distance measured by the sensor respectively, while *gain* is a constant proportional to the rotation's magnitude which can be determined empirically. If we assume that when the robot faces an

[8] Assuming that at least one of the mobile robot's distance sensors can still detect the obstacle when the robot is parallel to it.

obstacle it always decides to go left, then $D_0 > d$ would cause a positive counter-clockwise rotation away from the obstacle. On the other hand, $D_0 < d$ would cause a negative clockwise rotation towards from the obstacle. Naturally, $D_0 = d$ would cause no rotation at all and a straight-line translation parallel to the obstacle. In conclusion, the Behavior Range-Follow alternates between rotations and translations to keep the robot at a specified distance D_0, while following the boundary of an obstacle.

One of the main problems that I faced in my project was to replicate the boundary-following behavior described above without having any sort of distance sensors installed on the robot. The solution that I propose in this paper still alternates between translations and rotations to keep track of where the obstacle is. However, the main difference between my idea and the one mentioned above is that my algorithm has the mobile robot regularly bumping into obstacles to gather information on where they are on the map.

3 Main Result

3.1 Theoretical Development

The navigation algorithm that I developed over the course of my MSc project makes use of pure rotations, pure translations and a "greedy" version of the Bug 2 algorithm. In the case of pure translations, I empirically found out motors' inputs-speed conversion factors by taking note of the distance travelled by MIRTO in a specific amount of time and by observing the motors' inputs associated with every experiment. In particular, when using the method call *setMotors(int s0, int s1)* with $(s0, s1) \in (93, -100), (-93, 100)$, I found out that MIRTO would travel at speeds of about 11/12 cm/s. Hence, I estimated the motors' inputs-speed conversion factors, as defined in Equation 3, to be the ones reported in Table 2. It is always important to remember that these factors are rough and can strongly depend on the type of ground the mobile robot operates on[9].

$$\text{Motor's Input-Speed Conversion Factor} = \text{Motor's Input/Robot's Speed} \tag{3}$$

Wheel & Direction	**Motor's Input-Speed Conversion Factor**
Left Wheel (Forward)	8.1
Right Wheel (Forward)	8.7
Left Wheel (Backwards)	8.3
Right Wheel (Backwards)	8.9

Table 2: The motors' inputs-speed conversion factors for the two MIRTO's wheels.

In order to allow MIRTO to translate by *dist* centimeters, we need to calculate the time t_{trans} in milliseconds during which the method *setMotors(*± 93*,* ∓ 100*)* should operate (see Equation 4).

[9] MIRTO was tested on laminate flooring.

$$t_{trans} = ROUND^{10}\left(\frac{1000 * |dist|)}{\frac{93}{speed\_factor\_left}}\right)^{11} \tag{4}$$

The actual distance, $dist_{actual}$, travelled by the robot after a translation can be calculated using the wheels' radius r, the average encoders' counts c_{trans} and the encoders' counts per complete revolution, c_{rev} (see Equation 5). In particular, for the MIRTO robot we have $r = 3$ cm and $c_{rev} = 64$.

$$dist_{actual} = 2 \cdot \pi \cdot r \cdot \frac{c_{trans}}{c_{rev}}{}^{12} \tag{5}$$

MIRTO uses the user's value *dist* to execute its translations. However, after each translation, the mobile robot updates its location using $dist_{actual}$ as calculated in Equation 5.
Pure counterclockwise and clockwise rotations of *change* degrees are performed using the method call *setMotors(*± 95, ± 100*)*[13] with times of execution given by Equations 6 and 7, respectively. In fact, I empirically found out that MIRTO took about 3810 and 3640 milliseconds to perform 360 degrees counterclockwise and clockwise rotations, respectively. It's worth mentioning that these two values are mostly indicative and can even vary as much as a few hundreds milliseconds, depending on several robotic and environmental factors. For example, I tested MIRTO on laminate flooring and observed variations of full-rotations times up to $100/200$ milliseconds.

$$t_{counterclockwise} = ROUND\left(3810 \cdot \frac{|change|}{360}\right)^{14} \tag{6}$$

[10] The method *Thread.sleep(int time)* takes an integer input.

[11] I calculated two values of t for forward and backwards translation, respectively (i.e. using the two left wheel factors reported in Table 2).

[12] In the method *public double get_enc_distance()* (see Appendix A) there is a minus sign in the calculation of $dist_{actual}$. The reason for this is that, when MIRTO moves forward, the encoders' counts are both negative and I adopted a positive sign convention for forward translations.

[13] Throughout this paper I use the convention of positive counterclockwise rotations and negative clockwise rotations.

[14] In Java a conversion from Long to Integer is required (see the method *public void rotate(double change)* in Appendix A)

$$t_{clockwise} = ROUND\left(3640 \cdot \frac{|change|}{360}\right)^{15} \tag{7}$$

Similarly to the case of pure translations, the actual rotation, rot_{actual}, made by a differential wheeled robot can be calculated using the average of the encoders' counts for the performed rotation, c_{rot}, and the average of the encoders' counts for a 360 degrees rotation of MIRTO about its central axis, c_{full}.

$$rot_{actual} = \begin{cases} \frac{c_{rot}}{c_{full}} \cdot 360 & \text{if the encoder's count for the right wheel is negative} \\ -\frac{c_{rot}}{c_{full}} \cdot 360 & \text{otherwise} \end{cases}^{16} \tag{8}$$

MIRTO uses the user's value *change* to execute its rotations. However, after each rotation, the mobile robot updates its orientation using rot_{actual} as calculated in Equation 8.
Assuming that the initial position and orientation of MIRTO are known, the robot will enter the "motion to goal behavior" (see Algorithm 2 on the next page) of the "greedy" Bug 2 algorithm by first performing a pure rotation towards the goal and then executing a straight line translation[17] until it either hits an obstacle or the final destination has been reached. If an obstacle gets detected, an "hit" point is saved on a list and the mobile robot starts its "wall-following behavior" (see Algorithm 3 on page 22). As I have already explained in the "Background Theories" section, the "hit"/"leave" points are the intersections between the start-goal straight line and the boundaries of all of the obstacles in between start and goal points. Hence, once the robot has recorded the "hit" point in its memory, it will start following the boundary of the obstacle until the goal-line [18] is crossed at a "leave" point that has not been visited yet [8]. In particular, once in "wall-following behavior" mode, MIRTO will go backwards by 8 cm, turn 90 degrees counterclockwise and execute

[15] See footnote number 13.

[16] The plus or minus sign depends on whether the rotation is counterclockwise, i.e. positive, or clockwise, i.e. negative. In the case of MIRTO, negative encoders' values are caused by the wheels moving forward.

[17] The straight line translation will be executed by MIRTO in steps of 30 centimeteres and recalculating the current location-goal slope after each step.

[18] Straight line connecting start and goal points

a 12 cm forward translation. MIRTO will keep performing these three steps until no obstacle is detected after the last 12 cm forward translation. It will then perform a 90 degrees clockwise rotation, followed by a 12 centimeters translation, at which point it will start the above-mentioned three steps again.

Algorithm 2 A variant of the Motion to Goal Behavior that recalculates the slope between the current location and the goal point at the beginning of every iteration of the while loop. If an obstacle gets detected, an "hit" point is saved on a list and the mobile robot starts its Wall-Following Behavior (see Algorithm 3 on the next page). If the goal has been reached, nothing gets done by the algorithm.

```
1:  procedure MOTION TO GOAL BEHAVIOR
2:      if the goal has not been reached then
3:          while an obstacle is not detected and the goal has not been reached do
4:              get the slope s between the current location and the goal point
5:              new_orientation = atan(s)
6:              convert the new orientation from radians to degrees
7:              pick one of the two solutions given by the function atan, depending
                on where the goal point is (see Appendix A: MobileRobot.java).
                When using the Java method Math.atan, the two possible solutions
                are new_orientation and new_orientation + 180.
8:              Rotate to reach new_orientation
9:              Translate forward by 30 centimeters
10:         if the goal has not been reached then
11:             Add the new "hit" point to the list
12:             Call the Wall-Following Behavior
```

The "greedy" version of the "wall-following behavior" will switch to "motion to goal behavior" either if the method public boolean *goal_line_hit(double init_slope)* returns true (see Algorithm 4 on page 24) or if more than 5 iterations of the "wall-following behavior" have been executed. Similarly to the "motion to goal behavior", if the goal has been reached, nothing gets done by the algorithm.
The function *goal_line_hit(double init_slope)* returns true if the mobile robot hits the goal slope it stored at the beginning of the "wall-following behavior"[19] (see Algorithm 3 on the previous page), i.e. *init_slope*, at a "hit" point that has not been visited yet and the function *goal_line_hit(double init_slope)* has been called

[19]The function *goal_line_hit(double init_slope)* returns true if the mobile robot hits a slope within 5% of the goal slope it stored at the beginning of the "wall-following behavior"

at least 2 times[20]. It also returns true if *init_slope* is greater than $c_{wall-following} = 5$, if the absolute difference of the x-coordinates of the current location and goal point is less than 5 and if the function *goal_line_hit(double init_slope)* has been called at least 2 times.It returns false otherwise.

.

Algorithm 3 A "greedy" version of the Wall-Following Behavior that switches to Motion to Goal Behavior (see Algorithm 2 on the previous page) either if the method public boolean `goal_line_hit(double init_slope)` returns true (see Algorithm 4 on the next page) or if more than 5 iterations of the Wall-Following Behavior have been executed. If the goal has been reached, nothing gets done by the algorithm.

```
1:  procedure WALL-FOLLOWING BEHAVIOR
2:      if the goal has not been reached then
3:          set the iterations count for the Wall-Following Behavior, i.e.
            count_wall, to 0
4:          get the slope between the current location and the goal point, i.e.
            init_slope
5:          while !goal_line_hit(init_slope) and count_wall <= 5 and
                the goal has not ben reached do
6:              while an obstacle is detected by the robot and the goal has not been
                    reached do
7:                  translate backwards by 8 centimeters
8:                  Rotate 90 degrees counterclockwise
9:                  translate forward by 12 centimeters
10:             end while
11:             Rotate 90 degrees clockwise
12:             Translate forward by 12 centimeters
13:             count_wall = count_wall + 1
14:             if count_wall > 5 then
15:                 Translate backwards by 10 centimeters
16:                 if the goal has not been reached then
17:                     Call the Motion to Goal Behavior
18:         end while
19:         if the goal has not been reached then
20:             Call the Motion to Goal Behavior
```

[20]This is done in order to avoid false detections of new "hit"/"leave" points caused by the robot remaining in a similar location

Algorithm 4 Returns true if the mobile robot hits the goal slope it stored at the beginning of the Wall-Following Behavior (see Algorithm 3 on the previous page), i.e. `init_slope`, at a "hit" point that has not been visited yet and the function `goal_line_hit(double init_slope)` has been called at least 2 times. It also returns true if `init_slope` is greater than 5, if the absolute difference of the x-coordinates of the current location and goal point is less than 5 and if the function `goal_line_hit(double init_slope)` has been called at least 2 times.It returns false otherwise.

```
1:  procedure GOAL LINE "HIT"
2:      get the slope between the current location and the goal point, i.e. location_slope
3:      set a boolean flag to false
4:      if the function goal_line_hit(double init_slope) has been called at
            least 2 times, i.e. count_glh >= 2 then
5:          if 100*Math.abs(init_slope - location_slope )/init_slope < 5)
                and the "hit" point has not been visited yet, i.e. it is not in the list
                then
6:              flag = true
7:          end if
8:          if init_slope > 5 and the absolute difference between the x-
                coordinates of the current location and goal point is less than 5 then
9:              flag = true
10:         end if
11:     end if
12:     if flag == true then
13:         count_glh = 0
14:         return true
15:     else
16:         count_glh = count_glh +1
17:         return false
```

The "motion to goal" and "wall-following" behaviors will keep taking turns until there are no obstacles between the robot and the goal, at which point the mobile robot will mantain its "motion to goal behavior" until the destination has been reached. The main advantage of this "greedy" version of the Bug2 algorithm is that it allows a mobile robot equipped with encoders and bumper's sensors to reach the goal point very quickly in scenarios with simply-shaped and small obstacles. On the other hand, this version of the Bug2 algorithm is not complete and there is no guarantee it will find a solution if there exists one. In particular, the "greedy"

version of the Bug2 algorithm becomes complete in the limit where the mobile robot spends more and more time in the wall-following behavior or as much time as needed to reach a "leave" point. In other words, this "greedy" version of the Bug2 algorithm becomes complete in the limit where $c_{wall-following} \rightarrow \infty$.

3.2 Analysis and Design

The design of the *goToGoal()* function, which allows the robot to reach a goal point autonomously, is composed of four main blocks (see the flow diagram in Figure 7[21]): *translate(double dist)*, *rotate(double change)*, *motion_to_goal()* and *wall_following()*.

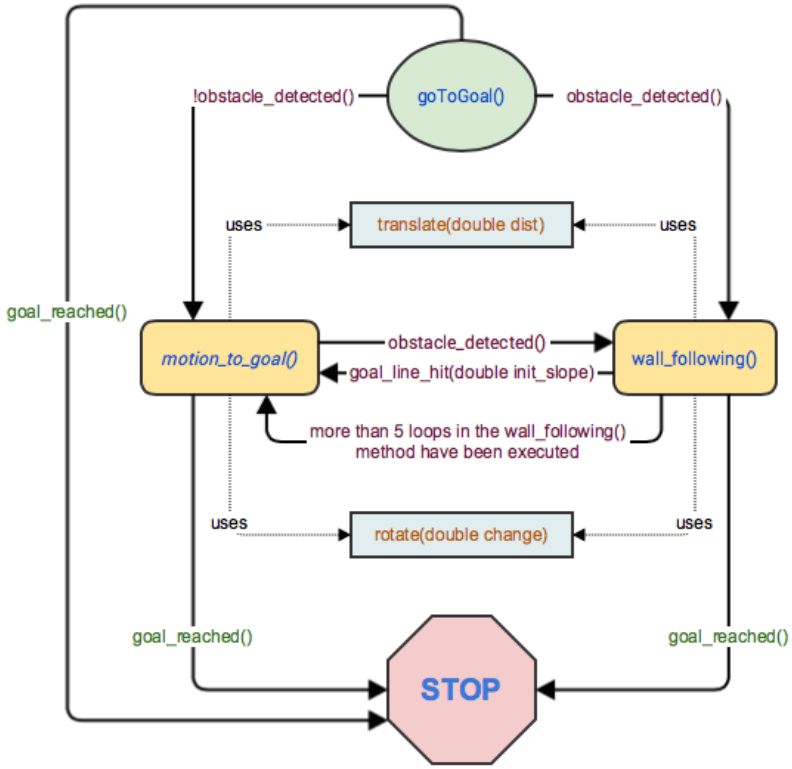


Figure 7: Flow diagram for the `public void goToGoal()` method, which allows the JMirto robot to reach a goal location autonomously. This algorithm is a "greedy" version of the Bug2 algorithm. As a result, it is not always guaranteed to find a solution if there is one. However, it allows the robot to reach a goal location relatively quickly in scenarios where the obstacles are small and not U-shaped. It is worth mentioning that before every transition to the `public void wall_following()` method a "hit" point gets stored in a list.

[21] The flow diagram in Figure 7 was made using *gliffy* (https://www.gliffy.com).

When the method *goToGoal()* two things can happen. If an obstacle is detected, the *wall-following behavior* gets called, ie. *wall_following()*. Otherwise, the robot enters its *motion to goal behavior*, i.e. *motion_to_goal()*. The function *obstacle_detected()* (see Appendix A) simply return true if one or both of the bumper's sensors have detected an obstacle. As I have already explained in the *Description of the JMirto Robot* section, MIRTO's bumper does not cover the entire front part of the robot. As a result, I wrote an evasive maneuver at the end of the *translate(double dist)* method to allow MIRTO to get the obstacle out of the way and start its *motion to goal behavior* again (see the *translate(double dist)* method in Appendix A).
The *motion to goal behavior* and *wall-following behavior* operate as described in the *Theoretical Development* section of this paper and as depicted in the flow diagram shown in Figure 7. Both of these two "higher-level" behaviors make use of the *translate(double dist)* and *rotate(double change)* methods.
The *translate(double dist)* function operates as described in *Theoretical Development* section and updates the location of the robot using Equations 9 and 10 (see below).

$$x_{new} = x_{old} + dist_{actual} \cdot cos(\theta_{cur}) \tag{9}$$

$$y_{new} = y_{old} + dist_{actual} \cdot sin(\theta_{cur}) \tag{10}$$

In Equations 9 and 10, (x_{old}, y_{old}) and (x_{new}, y_{new}) represent the old and new location of the mobile robot[22], respectively. Moreover, θ_{cur} represents the current orientation of the robot in radians. One interesting detail of the method *translate(double dist)* is that, when the robot is less than 50 centimeters away from the goal, the amounts of the forward translations commands get overridden in such a way that the robot starts making smaller and smaller translations as it gets closer and closer to its destination (see the *translate(double dist)* in Appendix A).
The *rotate(double change)* also behaves as described in *Theoretical Development* section and updates the orientation of the robot using Equation 11 (see below).

[22]The location of the robot is expressed in centimeters throughout the rest of my paper.

$$\theta_{new} = \begin{cases} (\theta_{cur} + rot_{actual})\%360 & \theta_{new} \geq 0 \ {}_{2324} \\ 360 + (\theta_{cur} + rot_{actual})\%360 & \theta_{new} < 0 \end{cases} \tag{11}$$

While testing the robot, I observed a systematic error in the rotations of about 1 degree counterclockwise. That is the reason why I subtract 1 degree from θ_{new} towards the end of the *rotate(double change)* function.
Finally, it is important to observe from the flow diagram portrayed in Figure 7, that whenever the goal is reached, i.e. *goal_reached()*, the mobile robot stops and the methods *goToGoal()*, *motion_to_goal()* and *wall_following()* do not do anything and eventually stop executing as the robot comes to a stop. In particular, even the *translate(double dist)* and *rotate(double change)* functions have been coded not to do anything in case the robot has reached its destination.
The method *goal_reached()* (see Appendix A) returns true if MIRTO is within 7 cm of the goal, in which case it prints the current location and orientation of the robot as well as the cumulative distance traveled[25]. It is definitely possible to allow MIRTO to get closer to the goal by lowering the *goal_reached()* threshold by 2/3 centimeters. In fact, the reason why I set the goal threshold to 7 cm is because I wanted MIRTO to terminate its navigation algorithm faster and not start wandering around the goal for a while before coming to a halt.

[23]The symbol % stands for the computer science "mod" operator.

[24]If $\theta_{new} < 0$ we need to add 360 to θ_{new} in order to keep the orientation of the robot positive. In fact, that is the convention that I use throughout this paper.

[25]In the calculation of the cumulative distance traveled by the robot, the backwards translations are considered as "positive" translations.

3.3 Implementation and Experimental Work

In order to implement the theory behind my ideas, I developed a Java class named *MobileRobot.java* meant to give high-level commands to MIRTO (see Appendix A). Moreover, I wrote a routine named *RobotTesting.java* containing a main unit to test the basic translational and rotational functionalities of the *MobileRobot* class (see Appendix C). In order to develop the class *MobileRobot.java* I used a few snippets of code and methods provided by the class, *JMirtoRobot.java*, developed by a team led by Dr Franco Raimondi (F.Raimondi@mdx.ac.uk) at Middlesex University London [3]. The class *JMirtoRobot* provides the user with lower-level commands to control the robot. For example, the function *public void setMotors(int s0, int s1)* sets the voltages of the two DC motors that are responsible for actuating the wheels. As a result, in the higher-level class *MobileRobot.java* I also take care of the conversion factor between DC motors' inputs and wheels' speeds in centimeters per second, as explained in the *Theoretical Development* section of my work. In order to send high-level commands to Mirto, the user will simply need to create a new *MobileRobot* object and take advantage of the functions implemented within the class. All of the methods and classes implemented in *MobileRobot.java* are shown in the comments below (see Appendix A for the implementation of each method or class).

```
/* File: MobileRobot.java
 * Date: June 2015
 * King's College London -- Dept. of Informatics -- MSc in Robotics
 * Author: Claudio S. De Mutiis (claudio.de_mutiis@kcl.ac.uk)
 * Purpose: Provide the MIRTO robot, developed by a team led by Dr
 * Franco Raimondi (F.Raimondi@mdx.ac.uk) at Middlesex University
 * London, with high-level odometrical functionalities.
 *
 * IMPORTANT NOTES:
 *
 * 1) WHERE FILES/FOLDERS SHOULD BE PLACED IN ORDER TO MAKE
 *    EVERYTHING COMPILE AND WORK CORRECTLY (on MIRTO's SD
 *    CARD):
 *    (i) MobileRobot.java in /csd2222
 *    (ii) RobotMission.java in /csd2222
 *    (iii) RobotTesting.java in /csd2222
 *    (iv) The following files and folders should also be placed
 *         in /csd2222: java-asip.jar, jssc, libs, META-INF and
 *         uk.
 *    (v) The modified version of the file JMirtoRobot.java (see
 *        below) should be placed in
```

```
*       /csd2222/uk/ac/mdx/cs/asip
*
*    Source: https://github.com/fraimondi/java-asip
*
* 2) The code in JMirtoRobot.java has been modified by Claudio S.
* De Mutiis (claudio.de_mutiis@kcl.ac.uk) in August 2015.
* Claudio S. De Mutiis added the method public void resetCount(),
* which resets both of the encoders' counts.
*
*  public void resetCount() {
*    e0.resetCount();
*    e1.resetCount();
*  }
*
* TRYING TO COMPILE THE CLASSES MobileRobot.java, RobotMission.java
* and RobotTesting.java WITH THE OLD VERSION OF JMirtoRobot.java
* WILL RESULT IN A COMPILATION ERROR !!!
*
* The modified version of JMirtoRobot.java is needed to make
* everything compile and run correctly !!
*
* ADDITIONAL CREDITS:
* - The class JMirtoRobot.java was developed by a team led by Dr
*   Franco Raimondi (F.Raimondi@mdx.ac.uk) at Middlesex University
*   London. When using the functionalities of the class
*   JMirtoRobot.java, in several occasions, I used or slightly
*   modified snippets of code taken from the file JMirtoRobot.java.
*   I copied and pasted the main method for testing the class
*   JMirtoRobot below in the comments (after the description for
*   the class Point).
* - In order to be able to work, the class MobileRobot.java makes
*   use of methods and classes developed by a team led by Dr
*   Franco Raimondi at Middlesex University (see the folders lib
*   and src and the files build.xml, README.md located in the
*   folder java-asip-master).
*   Source: https://github.com/fraimondi/java-asip
*/

/*  Useful Math Functions:
 * 1) Math.toDegrees(...) --> converts an angle from rad to deg
 * 2) Math.toRadians(...) --> converts an angle from deg to rad
 */

/************************** MobileRobot ***************************
 * public MobileRobot() --> Constructor with no arguments
```

```
* public MobileRobot(Point start_new) --> Constructor with the
*                 start point coordinates in cm
* public MobileRobot(Point start_new, double orientation_new) -->
*                 Constructor with start point coordinates in cm
*                 and orientation in degrees
* public MobileRobot(Point start_new, Point goal_new) -->
*                 Constructor with start and goal points
*                 coordinates in cm
* public MobileRobot(Point start_new, Point goal_new, double
*                 orientation_new) --> Constructor with start and
*                 goal coordinates in cm and orientation in degrees
* public Point getLocation() --> Get the current location of the
*                 mobile robot (x,y) -- cm
* public void setStart(Point start_new) --> Set the start point (x,y)
*                 -- cm
* public Point getStart() --> Get the start point (x,y) -- cm
* public void setGoal(Point goal_new) --> Set the goal point (x,y)
*                 -- cm
* public Point getGoal() --> Get the goal point (x,y) -- cm
* public void setOrientation(double orientation_new) --> Set the
*                 orientation of the robot -- degrees
* public double getOrientation() --> Get the current orientation of
*                 the robot with respect to the positive x-axis --
*                 degrees
* public double getStartOrientation() --> Get the initial orientation
*                 of the robot with respect to the positive x-axis
*                 -- degrees
* public void setSpeed(double speed_left, double speed_right) --> Set
*                 the speeds of the right and left wheel -- cm/s
* public Speed getSpeed() --> Get the speeds of the right and left
*                 wheel -- cm/s
* public double getLinSpeed() --> Get the linear speed of the mobile
*                 robot -- cm/s
* public double getAngSpeed() --> Get the angular speed of the mobile
*                 robot -- radians/sec
* public void translate(double dist) --> Make the mobile robot
*                 translate by "dist" cm
* public void rotate(double change) --> Make the mobile robot rotate
*                 by "change" degrees
* public void stop() --> Make the mobile robot stop
* public void reset(Point goal_new) --> Reset the goal point coordinates
*                 in cm
* public void reset(Point goal_new, double start_orientation_new) -->
*                 Reset the goal point in cm and orientation of the
*                 robot in degrees
```

```
* public void goToGoal() --> Make the mobile robot move to its goal point
* public void goToGoal(Point goal_new) --> Make the mobile robot move to
*                 the new Goal Point
* public void motion_to_goal() --> A variant of the Motion to Goal Behavior
*                 that recalculates the slope between the current location
*                 and the goal point at the beginning of every iteration
*                 of the while loop. If an obstacle gets detected, an "hit"
*                 point is saved on a list and the mobile robot starts its
*                 Wall-Following Behavior. If the goal has been reached,
*                 nothing gets done by the algorithm.
* public void wall_following() --> A "greedy" version of the Wall-Following
*                 Behavior that switches to Motion to Goal Behavior either
*                 if the method public boolean goal_line_hit(double
*                 init_slope) returns true or if more than 5 iterations of
*                 the Wall-Following Behavior have been executed.If the
*                 goal has been reached, nothing gets done by the
*                 algorithm.
* public boolean obstacle_detected() --> Returns true if an obstacle has
*                 been detected by the mobile robot's bumpers
* public void avoid_obstacle(char ch) --> Determines which bumper detected
*                 the obstacle and makes the robot act appropriately ('l'
*                 and 'r' for counterclockwise and clockwise rotation,
*                 respectively). This code has been written in a way to a
*                 allow further development for the programmer who wants
*                 the robot to perform different actions depending on which
*                 bumpers' sensors are activated.
* public boolean goal_line_hit(double init_slope) --> Returns true if the
*                 mobile robot hits the goal slope it stored at the
*                 beginning of the Wall-Following Behavior, i.e. init_slope,
*                 at a "hit" point that has not been visited yet and the
*                 function goal_line_hit(double init_slope) has been called
*                 at least 2 times. It also returns true if init_slope is
*                 greater than 5, if the absolute difference between the
*                 x-coordinates of the current location and goal point is
*                 less than 5 and if the function
*                 goal_line_hit(double init_slope) has been called at least
*                 2 times.It returns false otherwise.
* public boolean goal_reached() --> Returns true if the robot is within 7 cm of
*                 its current goal point
* public double getGoalSlope() --> Get the slope of the line connecting the
*                 start point and the goal point
* public double getLocationSlope() --> Get the slope of the line conecting the
*                 goal point and the current location of the robot
* public boolean hp_visited() --> Returns true if the mobile robot crosses one
*                 of the "hit" points previously visited
```

```
* public double get_enc_distance() --> Get the "encoder" distance traveled by
*                 the mobile robot
* public double get_cum_distance() --> Get the cumulative distance traveled by
*                 the mobile robot
* public void print_location(String current) --> Print the current location of
*                 the robot
* ********************************************************************************
*
* *************** Speed (part of the class MobileRobot.java) ********************
* public Speed() --> Constructor for zero Speed:(0,0)
* public Speed(double vr_new, double vl_new) --> Constructor for Speed:
*               (speed_right_wheel, speed_left_wheel)
* public void set_vr(double vr_new) --> Set the speed of the right wheel
* public double get_vr() --> Get the speed of the right wheel
* public void set_vl(double vl_new) --> Set the speed of the left wheel
* public double get_vl() --> Get the speed of the left wheel
* ********************************************************************************
*
* ********************************** Point **********************************
* public Point() --> Constructor for the origin point:(0,0)
* public Point(double x_new, double y_new) --> Constructor for a Point:(x,y)
* public void setX(double x_new) --> Set the x-coordinate of the point
* public double getX() --> Get the x-coordinate of the point
* public void setY(double y_new) --> Set the y-coordinate of the point
* public double getY() --> Get the y-coordinate of the point
* public double distanceTo(Point point) --> Get the distance of THIS point to
*                 another point
* ********************************************************************************
*
* The main method for testing the class JMirtoRobot.java, was written by a team led
* by Dr Franco Raimondi (F.Raimondi@mdx.ac.uk) at Middlesex University London
* (see Appendix A: MobileRobot.java)
*/
```

It is important to mention that I slightly modified the class *JMirtoRobot.java* by adding the method public *void resetCount()*, which resets the counts of both of the encoders embedded in MIRTO's wheels.
As we can observe from the comments of the *MobileRobot.java* class reported above, I created a *Point* class to handle the "blueprint" of a Point in the Cartesian plane. In particular, after creating a *Point*, the user can edit or retrieve the coordinates of the point by using methods implemented within the class. Moreover, the class *Point* provides the programmer with the very useful *public double distanceTo(Point point)* function, which calculates the distance between two points in

the Cartesian plane.
Similarly to the class *Point*, the class *Speed* provides the user functionalities to set and get the speeds of the MIRTO robot. At this point, I would like to highlight the fact that some parts of the class *MobileRobot* were designed in order to allow further developments of the code. For example, methods such as *public double get_vr()* and *public double get_vl()* to get the wheels' speeds or *public double getLinSpeed()* and *public double getAngSpeed()* to retrieve MIRTO's current linear and angular speeds would be a lot more useful if there was a user graphical interface and multithreading was used. Furthermore, I wrote a few methods in *MobileRobot* with the idea of allowing other programmers to further improve my navigation algorithm. For example, the function *public void avoid_obstacle(char ch)* can be easily modified to distinguish the cases of obstacle's avoidance depending on which bumper's sensors detected the obstacle, i.e. left sensor, right sensor or both of them. Also, MIRTO can be commanded to avoid the obstacle by going left, right or turn around by 180 degrees and go away from the obstacle. However, I decided not to use all of these functionalities of the *public void avoid_obstacle(char ch)* method in my project because I wanted to avoid unnecessary complications and keep my algorithms as simple as possible. Moreover, I believe that the way the bumper's sensors are installed on MIRTO is not good enough to allow the programmer to safely identify the orientation of the robot with respect to the obstacle. Before testing the basic rotational and translational functionalities of MIRTO with the program *RobotTesting.java* (see below or Appendix C), I carried out a long calibration process. In fact, I had to "tune" several parameters including motors' inputs-speed conversion factors (see the *Theoretical Development* section), the average of the encoder's counts (i.e. c_{full}) and the times taken for 360 degrees clockwise and counterclockwise rotations. I also had to determine the best input values in the method *setMotors(int s0, int s1)* for translations and rotations.

```
/* File: RobotTesting.java
 * Date: August 2015
 * King's College London -- Dept. of Informatics -- MSc in Robotics
 * Author: Claudio S. De Mutiis (claudio.de_mutiis@kcl.ac.uk)
 * Purpose: Test the basic rotational and translational capabilities
 *          of the MIRTO robot, developed by a team led by Dr Franco
 *          Raimondi (F.Raimondi@mdx.ac.uk) at Middlesex University
 *          London
 *
 * IMPORTANT NOTES:
 *
 * 1) WHERE FILES/FOLDERS SHOULD BE PLACED IN ORDER TO MAKE
 *    EVERYTHING COMPILE AND WORK CORRECTLY (on MIRTO's SD
```

```
*    CARD):
*    (i) MobileRobot.java in /csd2222
*    (ii) RobotMission.java in /csd2222
*    (iii) RobotTesting.java in /csd2222
*    (iv) The following files and folders should also be placed
*         in /csd2222: java-asip.jar, jssc, libs, META-INF and
*         uk.
*    (v) The modified version of the file JMirtoRobot.java (see
*        below) should be placed in
*        /csd2222/uk/ac/mdx/cs/asip
*
*    Source: https://github.com/fraimondi/java-asip
*
* 2) The code in JMirtoRobot.java has been modified by Claudio S.
* De Mutiis (claudio.de_mutiis@kcl.ac.uk) in August 2015.
* Claudio S. De Mutiis added the method public void resetCount(),
* which resets both of the encoders' counts.
*
*  public void resetCount() {
*    e0.resetCount();
*    e1.resetCount();
*  }
*
* TRYING TO COMPILE THE CLASSES MobileRobot.java, RobotMission.java
* and RobotTesting.java WITH THE OLD VERSION OF JMirtoRobot.java
* WILL RESULT IN A COMPILATION ERROR !!!
*
* The modified version of JMirtoRobot.java is needed to make
* everything compile and run correctly !!
*
* ADDITIONAL CREDITS:
* - The class JMirtoRobot.java was developed by a team led by Dr
*   Franco Raimondi (F.Raimondi@mdx.ac.uk) at Middlesex University
*   London.
* - In order to be able to work, the class RobotTesting.java makes
*   use of methods and classes developed by a team led by Dr
*   Franco Raimondi at Middlesex University (see the folders lib
*   and src and the files build.xml, README.md located in the
*   folder java-asip-master).
*   Source: https://github.com/fraimondi/java-asip
*/

/*  Useful Math Functions:
 * 1) Math.toDegrees(...) --> converts an angle from rad to deg
 * 2) Math.toRadians(...) --> converts an angle from deg to rad
```

```
 */
/**************************************************************************
/* VERY USEFUL FUNCTIONS OF MobileRobot.java (see the comments
 * of MobileRobot.java file to learn about other available functions)
 * ************************************************************************
 * public void translate(double dist) --> Make the mobile robot
 *                    translate by "dist" cm
 * public void rotate(double change) --> Make the mobile robot rotate
 *                    by "change" degrees
 * public void stop() --> Make the mobile robot stop
 * public void reset(Point goal_new) --> Reset the goal point in cm
 * public void reset(Point goal_new, double start_orientation_new)
 *                    --> Reset the goal point in cm and orientation
 *                    in degrees
 * public void goToGoal() --> Make the mobile robot move to its goal
 *                    point
 * public void goToGoal(Point goal_new) --> Make the mobile robot move
 *                    to the new goal point
 * ************************************************************************
 *
 */

public class RobotTesting {
    public static void main(String[] args) {
        // create a start point at the origin (0,0) --> current
        // location of the robot
        Point start = new Point();
        // create a goal point at (50, 50)
        Point goal = new Point(50,50);
        // specify the current orientation of the robot
        double orientation = 180;
        // create the robot object
        MobileRobot robot = new MobileRobot(start, goal, orientation);
        robot.translate(20);
        robot.translate(-20);
        robot.translate(20);
        robot.rotate(-180);
        robot.rotate(90);
    }
}
```

Once MIRTO had been calibrated and successfully tested using the *RobotTesting* routine, I could start "tuning" the parameters related to the *public void goToGoal()* method, which allows the robot to reach a destination on a Cartesian plane au-

tonomously. For instance, I had to fiddle with the translation steps' sizes in the *public void motion_to_goal()* and *public void wall_following()* methods. Moreover, I also took care of several "thresholds" such us the minimum distance from the goal in order for the goal to be "reached", the maximum error of the slopes matching in the *public boolean goal_line_hit(double init_slope)* function and the maximum "error distance" to compare and possibly match MIRTO's current location with one of the previously visited "hit" points in the list "visited_points" (see the method *public boolean hp_visited()* in Appendix A). I also had "tune" the maximum number of iterations before the motion following behavior would switch to *motion to goal behavior*, i.e. $c_{wall-following}$. It is very interesting to notice that the parameter $c_{wall-following}$ can be tuned by taking into account the environment the mobile robot is going to operate in. In particular, the sizes of the perimeters of the largest obstacles in the environment are the defining environmental characteristics that should be considered when deciding on the value of the $c_{wall-following}$ parameter.

3.4 Results and Observations

In order to observe the performance of the *public void goToGoal()* method, meant to send the robot autonomously from a start to a goal point avoiding any obstacles on the way, I planned a simple "mission" for the MIRTO robot (see the *RobotMission.java* class below or in Appendix B).

```
/* File: RobotMission.java
 * Date: June 2015
 * King's College London -- Dept. of Informatics -- MSc in Robotics
 * Author: Claudio S. De Mutiis (claudio.de_mutiis@kcl.ac.uk)
 * Purpose: Plan a mission for the MIRTO robot, developed by a team led
 * by Dr Franco Raimondi (F.Raimondi@mdx.ac.uk) at Middlesex University
 * London.
 *
 * IMPORTANT NOTES:
 *
 * 1) WHERE FILES/FOLDERS SHOULD BE PLACED IN ORDER TO MAKE
 *    EVERYTHING COMPILE AND WORK CORRECTLY (on MIRTO's SD
 *    CARD):
 *    (i) MobileRobot.java in /csd2222
 *    (ii) RobotMission.java in /csd2222
 *    (iii) RobotTesting.java in /csd2222
 *    (iv) The following files and folders should also be placed
 *         in /csd2222: java-asip.jar, jssc, libs, META-INF and
 *         uk.
 *    (v) The modified version of the file JMirtoRobot.java (see
 *        below) should be placed in
 *        /csd2222/uk/ac/mdx/cs/asip
 *
 *    Source: https://github.com/fraimondi/java-asip
 *
 * 2) The code in JMirtoRobot.java has been modified by Claudio S.
 * De Mutiis (claudio.de_mutiis@kcl.ac.uk) in August 2015.
 * Claudio S. De Mutiis added the method public void resetCount(),
 * which resets both of the encoders' counts.
 *
 *  public void resetCount() {
 *    e0.resetCount();
 *    e1.resetCount();
 *  }
 *
 * TRYING TO COMPILE THE CLASSES MobileRobot.java, RobotMission.java
```

```
 * and RobotTesting.java WITH THE OLD VERSION OF JMirtoRobot.java
 * WILL RESULT IN A COMPILATION ERROR !!!
 *
 * The modified version of JMirtoRobot.java is needed to make
 * everything compile and run correctly !!
 *
 * ADDITIONAL CREDITS:
 * - The class JMirtoRobot.java was developed by a team led by Dr
 *   Franco Raimondi (F.Raimondi@mdx.ac.uk) at Middlesex University
 *   London.
 * - In order to be able to work, the class RobotMission.java makes
 *   use of methods and classes developed by a team led by Dr
 *   Franco Raimondi at Middlesex University (see the folders lib
 *   and src and the files build.xml, README.md located in the
 *   folder java-asip-master).
 *   Source: https://github.com/fraimondi/java-asip
 */

/*  Useful Math Functions:
 * 1) Math.toDegrees(...) --> converts an angle from rad to deg
 * 2) Math.toRadians(...) --> converts an angle from deg to rad
 */
/*************************************************************************
/* VERY USEFUL FUNCTIONS OF MobileRobot.java (see the comments
 * of MobileRobot.java file to learn about other available functions)
 * ***********************************************************************
 * public void translate(double dist) --> Make the mobile robot
 *                   translate by "dist" cm
 * public void rotate(double change) --> Make the mobile robot rotate
 *                   by "change" degrees
 * public void stop() --> Make the mobile robot stop
 * public void reset(Point goal_new) --> Reset the goal point in cm
 * public void reset(Point goal_new, double start_orientation_new)
 *                   --> Reset the goal point in cm and orientation
 *                   in degrees
 * public void goToGoal() --> Make the mobile robot move to its goal
 *                   point
 * public void goToGoal(Point goal_new) --> Make the mobile robot move
 *                   to the new goal point
 * ***********************************************************************
 *
 */

public class RobotMission {
    public static void main(String[] args) {
```

```
        // create a start point at the origin (0,0) --> current
        // location of the robot
        Point start = new Point();
        // create a goal point at (0, 150)
        Point goal = new Point(0,150);
        // specify the current orientation of the robot
        double orientation = 180;
        // create the robot object
        MobileRobot robot = new MobileRobot(start, goal, orientation);
        // send the robot to its goal location
        robot.goToGoal();
    }
}
```

I carried out ten tests in a scenario where MIRTO had to travel 150 cm and avoid a roughly rectangular obstacle located approximately 80 cm away from the start point. In particular, I set the start and goal points at the origin (i.e. $(0,0)$) and at $(0,150)$, respectively. In order to make things "harder", I gave MIRTO an initial orientation of 180 degrees. As a result, during his *motion to goal behavior*, the robot had to rotate 90 degrees clockwise before heading north towards his destination (see Figure 8 on the next page).

When testing my navigation algorithms on MIRTO, I observed that the differential wheeled robot, on average, ended up 12.8 cm away from the desired goal location with a standard deviation of 7.6 cm (see Table 3 on page 42). I also observed an average percentage error and associated standard deviation of 8.5% and 5.1%, respectively, for the final distance from the robot to the goal. However, these values were calculated using the straight line distance between start and goal points, i.e. 150 centimeters. The distance that the robot usually travelled during each experiment was over two meters mainly due to the *wall following behavior* when going around the obstacle. So, the error on the distance travelled by the robot is notably smaller than the one reported in the Table 3. Moreover, the reason why the final y-coordinate in the robot's memory always came short of the goal's coordinate is that the goal threshold was set to 7 cm and MIRTO always approached its destination from the south.

Generally speaking, at the time of this paper, it is hard to get a very accurate and constant performance for the high-level odometrical function *goToGoal()*. However, good results can be achieved if the "calibration" of MIRTO is done correctly. I would like to highlight the fact that translations are much easier to calibrate than rotations are. In fact, individual rotations can quite easily be off by $1-2$ degrees and this type of error is quite random and unpredictable. Furthermore, even small rotation errors can strongly influence the performance of the *public void goTo-*

Goal() method, especially when the MIRTO robot is still close to its start point. Also, even though, over the course of MIRTO's journey to the goal, the rotations errors could balance themselves out, it is usually the case that a systematic error in one direction of rotation prevails. This could be due to the fact that, according to the design of the *public void wall_following()* method, the MIRTO robot always chooses to avoid an obstacle by going left. In other words, an "asymmetry" has been introduced into the navigation algorithm.

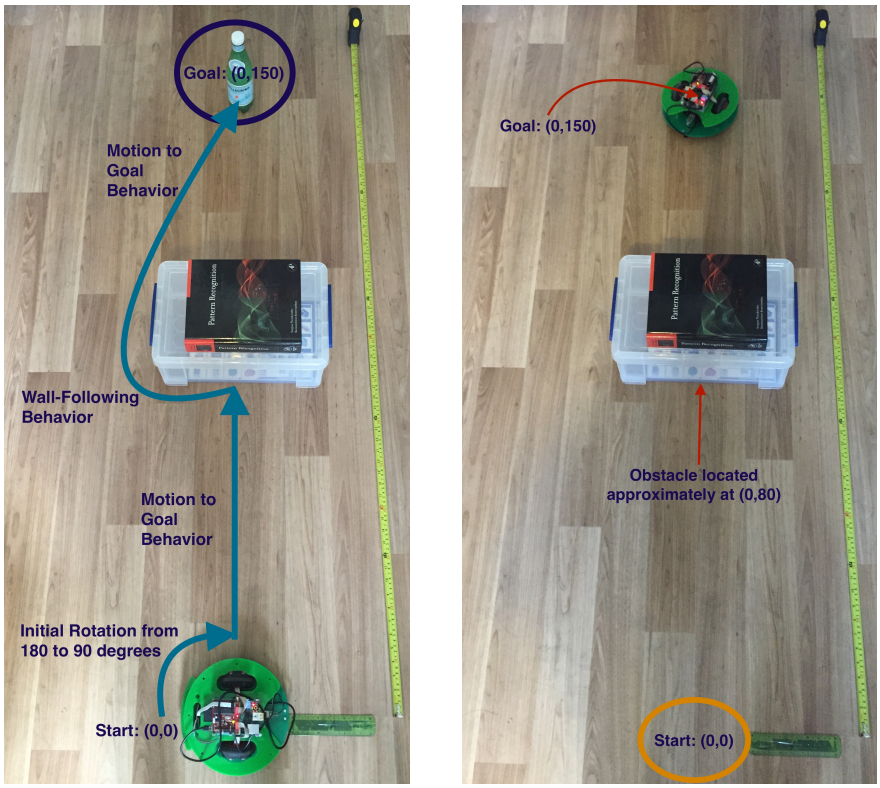


Figure 8: The MIRTO robot can reach the goal by alternating between motion to goal and wall-following behaviors.

While testing the MIRTO robot, I noticed several sources of random and systematic errors. In particular, besides translational and rotational errors, there were also human errors when taking all the required location measurements (see Figure 9 on the next page). Moreover, several other sources of errors included but were not limited to wheels' sliding, the battery level affecting the operations of the robot,

Start: (0,0) – cm	**Real**		**Robot's Memory**		**DTG (cm)**	**DTG Error**
Goal: (0,150) – cm	**x (cm)**	**y (cm)**	**x (cm)**	**y (cm)**		
Test 1	7.5	145.0	-1.8	143.5	9.0	6.0 %
Test 2	-11.5	142.0	-2.8	144.7	14.0	9.3 %
Test 3	0.0	151.5	-3.2	144.3	1.5	1.0 %
Test 4	-2.0	144.0	-3.1	144.6	6.3	4.2 %
Test 5	-16.0	141.5	-1.2	144.8	18.1	12.1 %
Test 6	7.5	139.5	-5.7	146.6	12.9	8.6 %
Test 7	6.5	143.5	-1.2	143.9	9.2	6.1 %
Test 8	9.0	140.5	-2.8	144.8	13.1	8.7 %
Test 9	-30.0	149.5	-2.3	143.9	30.0	20.0 %
Test 10	12.5	145.0	-2.6	144.5	13.5	9.0 %
Average	-1.7	144.2	-2.7	144.6	12.8	8.5 %
Std. Deviation	13.6	3.8	1.3	0.8	7.6	5.1 %

Table 3: Results of ten tests of the high-level odometrical function *goToGoal()* using the origin (0,0) and (0,150) as the start and goal points. "DTG" stands for the actual distance between the robot and the goal once the navigation algorithm has finished executing. In the ten tests reported in the table above, the average error and associated standard deviation for the final distance between the robot and the goal turned out to be 8.5% and 5.1%, respectively. However, these values were calculated using the straight line distance between start and goal points, i.e. 150 centimeters. The distance that the robot usually travelled during each experiment was over two meters mainly due to the *wall following behavior* when going around the obstacle. So, the error on the distance travelled by the robot is notably smaller than the one reported in the Table 3. The final average absolute distance from the goal and associated standard deviation were found to be 12.8 cm and 7.6 cm. However, these values are strongly influenced by one or two outliers among the data points.

the wheels' slightly different power requirements, the robot not being perfectly balanced and completely stable and the floor not being very even and cleaned.

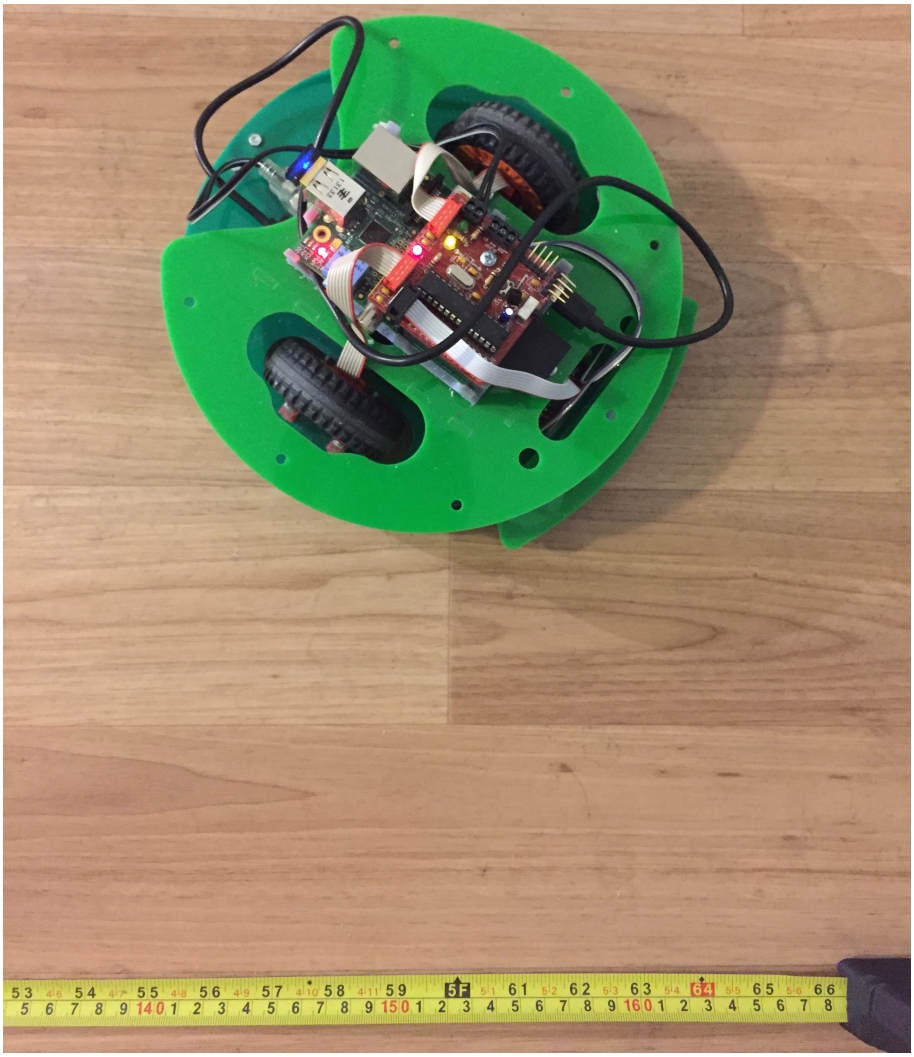


Figure 9: Before measuring the final location of the robot I would take a reference point to act as the center of MIRTO and then gently remove the robot while putting a five pence coin on the ground. Then, I would quite easily measure the location of the coin. I believe the error of this measuring process to be at most 1-2 centimeters.

4 Conclusion and Future Developments

In spite of the satisfactory results reported in the previous section of this report (see Table 3), there is plenty of room for improvements and further developments of my work. In particular, besides making MIRTO more balanced and stable, potential improvements include the use of wheels with more similar power requirements[26] and better performance. Moreover, in order to make any navigation algorithm on MIRTO more accurate and consistent, future work should also focus on trying to develop a system to remove part of the random error generated by individual rotations. In fact, in most cases, that is the main cause behind MIRTO heading off course and "believing" to be in a different position on the Cartesian plane in its memory. One method to regularly correct the robot's position in the environment is definitely the use of landmarks like the ones shown in Figure 10 below [6]. In particular, it would be very interesting to integrate the research carried out by Ortega-Garcia et al. [6] with the work discussed in this paper. In fact, the robot's memory could store a map of the landmarks present in the environment so that MIRTO could correct its own position every time one of the known landmarks was on sight. Naturally, in order to be able to do all of this, MIRTO would need to be equipped with a camera and a computer vision algorithm to identify the landmarks.

Figure 10: Examples of landmarks used by Ortega-Garcia et al. [6].

Future improvements of my code include but are not limited to the creation of a graphical user interface and the use of multithreading in order to greatly enhance the user's experience and scientists's efficiency in conducting their research. Furthermore, a rough map of the "visited" obstacles in the environment could be created by using all of the Cartesian points where MIRTO detected an obstacle by bumping against it. In particular, an interpolation algorithm should be used in

[26] In the case of pure translations and rotations, the absolute value of the ratio of the two arguments in the function *setMotors(int s0, int s1)* should be as close to 1 as possible in order for MIRTO to improve its overall performance.

order to better define the continuous boundaries of the obstacles. Knowing the location of the "visited" obstacles in the environment would allow MIRTO to reach a solution faster (at least in some cases) and would also greatly increase the robot's chances of finding a solution, if such solution exists.
As mentioned in the "Implementation and Experimental Work" section of this paper, the method *public void avoid_obstacle(char ch)* could be easily modified to distinguish the cases of obstacle's avoidance depending on which bumper's sensors detected the obstacle, i.e. left sensor, right sensor or both of them. Moreover, I wrote my code so that MIRTO can be commanded to avoid the obstacle by going left, right or turn around by 180 degrees and go away from the obstacle. Hence, especially once better bumper's sensors have been installed on MIRTO[27], there is plenty of potential to improve the *wall_following behavior* of the robot.
Finally, even though there is still much to do in local navigation and odometry, the current research carried out in these fields looks very promising. MIRTO is a quite "cheap" robotic platform which can not only be used for educational purposes [2] but also for future research in mobile robotics.

[27] As already mentioned in the *Implementation and Experimental Work* section of my work, I believe that the way the bumper's sensors are currently installed on MIRTO is not good enough to allow the programmer to safely identify the orientation of the robot with respect to the obstacle.

5 References

[1] Guirado, Rafael, Ramón González, Fernando Bienvenido, and Francisco Rodríguez. "Knowledge Modelling in Two-Level Decision Making for Robot Navigation." *Advances in Robot Navigation*. N.p.: INTECH Open Science | Open Minds, 2011. 218. Web. 23 Aug. 2015. Figure 7

[2] Huyck, Christian, Giuseppe Primiero, and Franco Raimondi. "Programming the MIRTO Robot with Neurons." *Procedia Computer Science* 41 (2014): 75-82. ScienceDirect. Web. 21 Aug. 2015.

[3] Fraimondi, Franco. "Fraimondi/java-asip." *GitHub*. N.p., Oct. 2014. Web. 22 Aug. 2015. <https://github.com/fraimondi/java-asip>.

[4] Muraca, Pietro, Paolo Pugliese, and Giuseppe Rocca. "An Extended Kalman Filter for the State Estimation of a Mobile Robot from Intermittent Measurements." *2008 16th Mediterranean Conference on Control and Automation* (2008): 1850-855. *IEEE Xplore*. Web. 23 Aug. 2015.

[5] Pinto, Andry Maykol G., A. Paulo Moreira, and Paulo G. Costa. "Robot@factory: Localization Method Based on Map-matching and Particle Swarm Optimization." *2013 13th International Conference on Autonomous Robot Systems* (2013): 1-6. *IEEE Xplore*. Web. 23 Aug. 2015.

[6] Ortega-Garcia, J.l., J.l. Gordillo, and R. Soto. "A New Method to Follow a Path on Indoor Environments Applied for Mobile Robotics." *11th IEEE International Conference on Control & Automation (ICCA)* (2014): 631-36. *IEEE Xplore*. Web. 23 Aug. 2015.

[7] Han, Soonshin, Byoungsuk Choi, and Jangmyung Lee. "A Precise Curved Motion Planning for a Differential Driving Mobile Robot." *Mechatronics* 18.9 (2008): 486-94. *ScienceDirect*. Web. 23 Aug. 2015.

[8] Latombe, Jean-Claude. "Motion Planning for a Point Robot (1/2)." CS26N: Motion Planning for Robots, Digital Actors, and Other Moving Objects. Stanford University, Stanford, California, US. *CS26N: Motion Planning for Robots, Digital Actors, and Other Moving Objects Winter 2012*. Web. 1 Aug. 2015. <http://ai.stanford.edu/ latombe/cs26n/2012/home.htm>.

[9] Althoefer, Kaspar. "Local Navigation". Robotics Systems Lecture. King's College London (Strand Campus), London, UK. 2015. Lecture.

6 Appendices

6.1 Appendix A: MobileRobot.java

```
/* File: MobileRobot.java
 * Date: June 2015
 * King's College London -- Dept. of Informatics -- MSc in Robotics
 * Author: Claudio S. De Mutiis (claudio.de_mutiis@kcl.ac.uk)
 * Purpose: Provide the MIRTO robot, developed by a team led by Dr
 * Franco Raimondi (F.Raimondi@mdx.ac.uk) at Middlesex University
 * London, with high-level odometrical functionalities .
 *
 * IMPORTANT NOTES:
 *
 * 1) WHERE FILES/FOLDERS SHOULD BE PLACED IN ORDER TO MAKE
 *    EVERYTHING COMPILE AND WORK CORRECTLY (on MIRTO's SD
 *    CARD):
 *    (i) MobileRobot.java in /csd2222
 *    (ii) RobotMission.java in /csd2222
 *    (iii) RobotTesting.java in /csd2222
 *    (iv) The following files and folders should also be placed
 *         in /csd2222: java-asip.jar, jssc, libs, META-INF and
 *         uk.
 *    (v) The modified version of the file JMirtoRobot.java (see
 *        below) should be placed in
 *        /csd2222/uk/ac/mdx/cs/asip
 *
 *    Source: https :// github.com/fraimondi/java-asip
 *
 * 2) The code in JMirtoRobot.java has been modified by Claudio S.
 * De Mutiis (claudio.de_mutiis@kcl.ac.uk) in August 2015.
 * Claudio S. De Mutiis added the method public void resetCount(),
 * which resets both of the encoders' counts.
 *
 *  public void resetCount() {
 *    e0.resetCount();
 *    e1.resetCount();
 *  }
 *
 * TRYING TO COMPILE THE CLASSES MobileRobot.java, RobotMission.java
 * and RobotTesting.java WITH THE OLD VERSION OF JMirtoRobot.java
 * WILL RESULT IN A COMPILATION ERROR !!!
 *
 * The modified version of JMirtoRobot.java is needed to make
 * everything compile and run correctly !!
 *
```

```
 * ADDITIONAL CREDITS:
 * - The class JMirtoRobot.java was developed by a team led by Dr
 *   Franco Raimondi (F.Raimondi@mdx.ac.uk) at Middlesex University
 *   London. When using the functionalities of the class
 *   JMirtoRobot.java, in several occasions, I used or slightly
 *   modified snippets of code taken from the file JMirtoRobot.java.
 *   I copied and pasted the main method for testing the class
 *   JMirtoRobot below in the comments (after the description for
 *   the class Point).
 * - In order to be able to work, the class MobileRobot.java makes
 *   use of methods and classes developed by a team led by Dr
 *   Franco Raimondi at Middlesex University (see the folders lib
 *   and src and the files build.xml, README.md located in the
 *   folder java-asip-master).
 *   Source: https :// github.com/fraimondi/java-asip
 */

/*  Useful Math Functions:
 * 1) Math.toDegrees (...) --> converts an angle from rad to deg
 * 2) Math.toRadians (...) --> converts an angle from deg to rad
 */

/************************** MobileRobot **************************
 * public MobileRobot() --> Constructor with no arguments
 * public MobileRobot(Point start_new) --> Constructor with the
 *                      start point coordinates in cm
 * public MobileRobot(Point start_new, double orientation_new) -->
 *                      Constructor with start point coordinates in cm
 *                      and orientation in degrees
 * public MobileRobot(Point start_new, Point goal_new) -->
 *                      Constructor with start and goal points
 *                      coordinates in cm
 * public MobileRobot(Point start_new, Point goal_new, double
 *                      orientation_new) --> Constructor with start and
 *                      goal coordinates in cm and orientation in degrees
 * public Point getLocation() --> Get the current location of the
 *                      mobile robot (x,y) -- cm
 * public void setStart(Point start_new) --> Set the start point (x,y)
 *                      -- cm
 * public Point getStart() --> Get the start point (x,y) -- cm
 * public void setGoal(Point goal_new) --> Set the goal point (x,y)
 *                      -- cm
 * public Point getGoal() --> Get the goal point (x,y) -- cm
 * public void setOrientation(double orientation_new) --> Set the
 *                      orientation of the robot -- degrees
 * public double getOrientation() --> Get the current orientation of
 *                      the robot with respect to the positive x-axis --
 *                      degrees
 * public double getStartOrientation() --> Get the initial orientation
```

```
*                    of the robot with respect to the positive x-axis
*                    -- degrees
* public void setSpeed(double speed_left, double speed_right) --> Set
*                    the speeds of the right and left wheel -- cm/s
* public Speed getSpeed() --> Get the speeds of the right and left
*                    wheel -- cm/s
* public double getLinSpeed() --> Get the linear speed of the mobile
*                    robot -- cm/s
* public double getAngSpeed() --> Get the angular speed of the mobile
*                    robot -- radians/sec
* public void translate (double dist) --> Make the mobile robot
*                     translate by "dist" cm
* public void rotate (double change) --> Make the mobile robot rotate
*                    by "change" degrees
* public void stop () --> Make the mobile robot stop
* public void reset (Point goal_new) --> Reset the goal point coordinates
*                    in cm
* public void reset (Point goal_new, double start_orientation_new ) -->
*                    Reset the goal point in cm and orientation of the
*                    robot in degrees
* public void goToGoal() --> Make the mobile robot move to its goal point
* public void goToGoal(Point goal_new) --> Make the mobile robot move to
*                    the new Goal Point
* public void motion_to_goal() --> A variant of the Motion to Goal Behavior
*                    that recalculates the slope between the current location
*                    and the goal point at the beginning of every iteration
*                    of the while loop. If an obstacle gets detected, an "hit"
*                    point is saved on a list and the mobile robot starts its
*                    Wall-Following Behavior. If the goal has been reached,
*                    nothing gets done by the algorithm.
* public void wall_following () --> A "greedy" version of the Wall-Following
*                    Behavior that switches to Motion to Goal Behavior either
*                    if the method public boolean goal_line_hit (double
*                    init_slope ) returns true or if more than 5 iterations of
*                    the Wall-Following Behavior have been executed. If the
*                    goal has been reached, nothing gets done by the
*                    algorithm.
* public boolean obstacle_detected () --> Returns true if an obstacle has
*                    been detected by the mobile robot's bumpers
* public void avoid_obstacle (char ch) --> Determines which bumper detected
*                    the obstacle and makes the robot act appropriately ('l'
*                    and 'r' for counterclockwise and clockwise rotation,
*                    respectively ). This code has been written in a way to a
*                    allow further development for the programmer who wants
*                    the robot to perform different actions depending on which
*                    bumpers' sensors are activated.
* public boolean goal_line_hit (double init_slope ) --> Returns true if the
*                    mobile robot hits the goal slope it stored at the
*                    beginning of the Wall-Following Behavior, i.e. init_slope,
```

```
*                       at a "hit" point that has not been visited yet and the
*                       function goal_line_hit (double init_slope ) has been called
*                       at least 2 times. It also returns true if init_slope is
*                       greater than 5, if the absolute difference between the
*                       x-coordinates of the current location and goal point is
*                       less than 5 and if the function
*                       goal_line_hit (double init_slope ) has been called at least
*                       2 times. It returns false otherwise.
* public boolean goal_reached() -->Returns true if the robot is within 7 cm of
*                       its current goal point
* public double getGoalSlope() -->Get the slope of the line connecting the
*                       start point and the goal point
* public double getLocationSlope() -->Get the slope of the line conecting the
*                       goal point and the current location of the robot
* public boolean hp_visited () -->Returns true if the mobile robot crosses one
*                       of the "hit" points previously visited
* public double get_enc_distance () -->Get the "encoder" distance traveled by
*                       the mobile robot
* public double get_cum_distance() -->Get the cumulative distance traveled by
*                       the mobile robot
* public void print_location ( String current ) -->Print the current location of
*                       the robot
* ************************************************************************
*
* ************** Speed (part of the class MobileRobot.java) ******************
* public Speed() -->Constructor for zero Speed :(0,0)
* public Speed(double vr_new, double vl_new) -->Constructor for Speed:
*                       (speed_right_wheel, speed_left_wheel)
* public void set_vr(double vr_new) -->Set the speed of the right wheel
* public double get_vr() -->Get the speed of the right wheel
* public void set_vl(double vl_new) -->Set the speed of the left wheel
* public double get_vl() -->Get the speed of the left wheel
* ************************************************************************
*
* ********************************** Point ********************************
* public Point() -->Constructor for the origin point :(0,0)
* public Point(double x_new, double y_new) -->Constructor for a Point :(x,y)
* public void setX(double x_new) -->Set the x-coordinate of the point
* public double getX() -->Get the x-coordinate of the point
* public void setY(double y_new) -->Set the y-coordinate of the point
* public double getY() -->Get the y-coordinate of the point
* public double distanceTo(Point point) -->Get the distance of THIS point to
*                       another point
* ************************************************************************
*
* The main method for testing the class JMirtoRobot.java, was written by a team led
* by Dr Franco Raimondi (F.Raimondi@mdx.ac.uk) at Middlesex University London
*
*
```

```
*   // A main method for testing
*   public static void main(String[] args) {
*
*   // JMirtoRobot robot = new JMirtoRobot("/dev/tty.usbserial-A903VH1D");
*   JMirtoRobot robot = new JMirtoRobot("/dev/ttyAMA0");
*
*
*   try {
*   Thread.sleep(500);
*   robot.setup();
*   Thread.sleep(500);
*   while (true) {
*   System.out.println("IR: "+robot.getIR(0) + ","+robot.getIR(1)+",
*                       "+robot.getIR(2));
*   System.out.println("Encoders: "+robot.getCount(0) + ","+robot.getCount(1));
*   System.out.println("Bumpers: "+robot.isPressed(0) + ","+robot.isPressed(1));
*   System.out.println("Setting motors to 50,50");
*   robot.setMotors(100, 0);
*   Thread.sleep(1500);
*   System.out.println("Stopping motors");
*   robot.stopMotors();
*   Thread.sleep(500);
*   System.out.println("Setting motors to 100,100");
*   robot.setMotors(0,-250);
*   Thread.sleep(1500);
*   System.out.println("Stopping motors");
*   robot.stopMotors();
*   Thread.sleep(500);
*   }
*   /*    System.out.println("Setting motors to 50,50");
*         robot.setMotors(50, 50);
*         Thread.sleep(3000);
*         System.out.println("Stopping motors");
*         robot.stopMotors();
*         Thread.sleep(500);
*         System.out.println("Setting motors to 80,-80");
*         robot.setMotors(80, -80);
*         Thread.sleep(3000);
*         System.out.println("Stopping motors");
*         robot.stopMotors();
*         Thread.sleep(3000);
*         System.out.println("Setting motors to -100,100");
*         robot.setMotors(-100, 100);
*         Thread.sleep(3000);
*         System.out.println("Stopping motors");
*         robot.stopMotors();
*         System.out.println("All done, see you soon!");
*   */
/*
```

```
 * } catch ( InterruptedException e) {
 * e. printStackTrace ();
 * }
 *
 * }
*
*/

import uk.ac.mdx.cs.asip.JMirtoRobot;
import uk.ac.mdx.cs.asip.services.BumpService;
import uk.ac.mdx.cs.asip.services.EncoderService;
import uk.ac.mdx.cs.asip.services.IRService;
import uk.ac.mdx.cs.asip.services.MotorService;
import java.util.*;

public class MobileRobot {
    // Distance between the two robot's wheels in cm
    private static final double l = 11.5;
    // Forward Motor-Speed Conversion Factor for laminate flooring
    // (left wheel)
    private static final double speed_factor_left_f = 8.1;
    // Forward Motor-Speed Conversion Factor for laminate flooring
    // (right wheel)
    private static final double speed_factor_right_f = 8.7;
    // Backward Motor-Speed Conversion Factor for laminate flooring
    // (left wheel)
    private static final double speed_factor_left_b = 8.3;
    // Backward Motor-Speed Conversion Factor for laminate flooring
    // (right wheel)
    private static final double speed_factor_right_b = 8.9;
    // Encoder count for a full 360 degrees rotation
    private static final double enc_full = 125;
    // Start Point (x,y) -- cm
    private Point start = new Point();
    // Goal Point (x,y) -- cm
    private Point goal = new Point();
    // Current location of the mobile robot (x,y) -- cm
    private Point location = new Point();
    // Speed (right wheel, left wheel) -- cm/s
    private Speed speed = new Speed();
    // Initial orientation of the robot with respect to the x-axis
    // -- degrees
    private double start_orientation ;
    // Current orientation of the robot with respect to the x-axis
    // -- degrees
    private double orientation ;
    // Linear Speed of the robot -- cm/s
    private double linear_speed ;
    // Angular Speed of the robot -- radians/sec
```

```
private double angSpeed;
// Robot (control motors and sensors)
public JMirtoRobot robot = new JMirtoRobot("/dev/ttyAMA0");
// Set of the 2D Cartesian "hit" points visited by the mobile
// robot
private Set<Point> visited_points = new HashSet<Point>();
// Number of iterations of the wall-following behaviour
private int count_wall;
// Cumulative distance traveled by the mobile robot
private double cum_distance;
// Slope from start to goal point
private double goal_slope;
// Number of times the function goal_line_hit () gets called
private int count_glh = 0;
// Counter for the number of translations when the robot
// gets stuck on an obstacle
private int count_trans = 0;
// Counter for the number of times the function
// goal_reached() gets called and returns true
private int count_goal_reached = 0;


// Constructor with no arguments
public MobileRobot() {
    try {
     location.setX(0);
     location.setY(0);
    goal.setX(0);
    goal.setY(150);
    robot.setup();
    robot.resetCount();
    Thread.sleep(500);
    stop();
     start.setX(0);
     start.setY(0);
      start_orientation = 90;
     orientation = 90;
     linear_speed = 0;
    angSpeed = 0;
    cum_distance = 0;
     visited_points = new HashSet<Point>();
     print_location("initial");
    System.out.println();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

// Constructor with the start point coordinates in cm
```

```
public MobileRobot(Point start_new) {
    try {
     location.setX(start_new.getX());
     location.setY(start_new.getY());
    goal.setX(0);
    goal.setY(150);
    robot.setup();
    robot.resetCount();
    Thread.sleep(500);
    stop();
     start.setX(start_new.getX());
     start.setY(start_new.getY());
      start_orientation = 90;
     orientation = 90;
     linear_speed = 0;
    angSpeed = 0;
    cum_distance = 0;
     visited_points = new HashSet<Point>();
     print_location("initial");
    System.out.println();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}


// Constructor with start point coordinates in cm and orientation
// in degrees
public MobileRobot(Point start_new, double start_orientation_new) {
    try {
     location.setX(start_new.getX());
     location.setY(start_new.getY());
    goal.setX(0);
    goal.setY(150);
    robot.setup();
    robot.resetCount();
    Thread.sleep(500);
    stop();
     start.setX(start_new.getX());
     start.setY(start_new.getY());
      start_orientation = start_orientation_new;
     orientation = start_orientation_new;
     linear_speed = 0;
    angSpeed = 0;
    cum_distance = 0;
     visited_points = new HashSet<Point>();
     print_location("initial");
    System.out.println();
    } catch (InterruptedException e) {
```

```
            e.printStackTrace();
        }
    }

    // Constructor with start and goal coordinates in cm
    public MobileRobot(Point start_new, Point goal_new) {
        try {
        location.setX(start_new.getX());
        location.setY(start_new.getY());
        goal.setX(goal_new.getX());
        goal.setY(goal_new.getY());
        robot.setup();
        robot.resetCount();
        Thread.sleep(500);
        stop();
        start.setX(start_new.getX());
        start.setY(start_new.getY());
         start_orientation = 90;
        orientation = 90;
        linear_speed = 0;
        angSpeed = 0;
        cum_distance = 0;
        visited_points = new HashSet<Point>();
        print_location("initial");
        System.out.println();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    // Constructor with start and goal coordinates in cm
    // and orientation in degrees
    public MobileRobot(Point start_new, Point goal_new, double start_orientation_new) {
        try {
        location.setX(start_new.getX());
        location.setY(start_new.getY());
        goal.setX(goal_new.getX());
        goal.setY(goal_new.getY());
        robot.setup();
        robot.resetCount();
        Thread.sleep(500);
        stop();
        start.setX(start_new.getX());
        start.setY(start_new.getY());
         start_orientation = start_orientation_new;
        orientation = start_orientation_new;
        linear_speed = 0;
        angSpeed = 0;
        cum_distance = 0;
```

```
        visited_points  = new HashSet<Point>();
        print_location (" initial ");
       System.out. println ();
       } catch ( InterruptedException  e) {
           e. printStackTrace ();
       }
}

// Get the current location of the mobile robot (x,y) -- cm
public Point getLocation () {
     return location ;
}

// Set the start point (x,y) -- cm
public void setStart (Point start_new ) {
      start .setX( start_new .getX ());
      start .setY( start_new .getY ());
      location .setX( start_new .getX ());
      location .setY( start_new .getY ());
}

// Get the start point (x,y) -- cm
public Point getStart () {
     return start ;
}

// Set the goal point (x,y) -- cm
public void setGoal(Point goal_new) {
     goal.setX(goal_new.getX ());
     goal.setY(goal_new.getY ());
}

// Get the goal point (x,y) -- cm
public Point getGoal() {
     return goal;
}

// Set the orientation of the robot -- degrees
public void setOrientation (double orientation_new ) {
     double rotation ;
     if (Math.abs( orientation_new - orientation ) > 180) {
          if ( orientation_new > orientation ) {
               rotation = -1*(360 - orientation_new + orientation );
          }else {
               rotation = 360 - orientation + orientation_new ;
          }
     }else {
          rotation = orientation_new - orientation ;
     }
```

```
        rotate ( rotation );
    }

    // Get the current orientation of the robot with respect to the
    // positive x-axis -- degrees
    public double getOrientation () {
        return orientation ;
    }

    // Get the initial orientation of the robot with respect to the
    // positive x-axis -- degrees
    public double getStartOrientation () {
        return start_orientation ;
    }

    // Set the speeds of the right and left wheel -- cm/s
    public void setSpeed(double speed_left , double speed_right ) {
        if (! goal_reached ()) {
            System.out. println ("Setting motors to (" + speed_left + "," + speed_right + ")");
            System.out. println ();
        }
        speed. set_vr ( speed_right );
        speed. set_vl ( speed_left );
        Long speed_l = Math.round( speed_left );
        int speed_left_new = Integer .valueOf(speed_l. intValue ());
        Long speed_r = Math.round(speed_right );
        int speed_right_new = Integer .valueOf(speed_r. intValue ());
        if (!(( Math.abs( speed_left )) > 50 || (Math.abs( speed_right ) > 50))) {
            if ( speed_left >= 0) {
                speed_l = Math.round( speed_left * speed_factor_left_f );
            }else {
                speed_l = Math.round( speed_left * speed_factor_left_b );
            }
            speed_left_new = Integer .valueOf(speed_l. intValue ());
            if ( speed_right < 0) {
                speed_r = Math.round(speed_right * speed_factor_right_f );
            }else {
                speed_r = Math.round(speed_right * speed_factor_right_b );
            }
            speed_right_new = Integer .valueOf(speed_r. intValue ());
        }
        robot .setMotors(speed_left_new , speed_right_new);
    }

    // Get the speeds of the right and left wheel -- cm/s
    public Speed getSpeed() {
        return speed;
    }
```

```
// Get the linear speed of the mobile robot -- cm/s
public double getLinSpeed() {
    return (speed.get_vr() + speed.get_vl())/2;
}

// Get the angular speed of the mobile robot -- radians/sec
public double getAngSpeed() {
    return 2*(speed.get_vr() - speed.get_vl())/l;
}

// Make the mobile robot translate by "dist" cm. If the robot has
// hit an obstacle not detected by the bumpers' sensors, perform an
// evasion manoeuvre.
public void translate(double dist) {
    try {
        if (!goal_reached()) {
            if (getLocation().distanceTo(getGoal()) < 50 && dist > 0) {
                dist = 20;
                if (getLocation().distanceTo(getGoal()) < 30) {
                    dist = 10;
                    if (getLocation().distanceTo(getGoal()) < 20) {
                        dist = 5;
                        if (getLocation().distanceTo(getGoal()) < 10) {
                            dist = 2;
                            if (getLocation().distanceTo(getGoal()) < 6) {
                                dist = 1;
                            }
                        }
                    }
                }
            }
            Point before_translation = new Point(location.getX(), location.getY());
            int time = 0;
            if (dist >= 0) {
                Long time_l = Math.round(1000*Math.abs(dist)/(93/speed_factor_left_f));
                time = Integer.valueOf(time_l.intValue());
                setSpeed(93, -100);
            }else {
                Long time_l = Math.round(1000*Math.abs(dist)/(93/speed_factor_left_b));
                time = Integer.valueOf(time_l.intValue());
                setSpeed(-93, 100);
            }
            Thread.sleep(time);
            stop();
            dist = get_enc_distance();
            cum_distance = cum_distance + Math.abs(dist); // Use wheels' encoders
            System.out.print("Translation of " + dist + " cm executed! ");
            System.out.println("(" + time + " ms)");
            location.setX(location.getX() + dist*Math.cos(Math.toRadians(orientation)));
```

```
                location.setY(location.getY() + dist*Math.sin(Math.toRadians(orientation)));
                print_location("new");
                System.out.print("The current location-goal slope is ");
                System.out.println(getLocationSlope());
                System.out.println();
                Point after_translation = new Point(location.getX(), location.getY());
                if (before_translation.distanceTo(after_translation) < 2) {
                    count_trans++;
                    if (count_trans == 2) {
                        rotate(5);
                        translate(-10);
                        rotate(30);
                        translate(30);
                        count_trans = 0;
                        if (obstacle_detected()) {
                            wall_following();
                        } else {
                            motion_to_goal();
                        }
                    }
                    if (count_trans > 2) {
                        rotate(-5);
                        translate(-10);
                        rotate(-30);
                        translate(30);
                        count_trans = 0;
                        if (obstacle_detected()) {
                            wall_following();
                        } else {
                            motion_to_goal();
                        }
                    }
                }else {
                    count_trans = 0;
                }
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    // Make the mobile robot rotate by "change" degrees. Try to correct
    // a systematic error by subtracting 1 degree to the final
    // orientation of the robot.
    public void rotate(double change) {
        try {
            if (!goal_reached()) {
                Long time_l;
                int time;
```

```
                if (change >= 0) {
                    time_l = Math.round(3810*(Math.abs(change)/360));
                    time = Integer.valueOf(time_l.intValue());
                    setSpeed(-95, -100);
                }else {
                    time_l = Math.round(3640*(Math.abs(change)/360));
                    time = Integer.valueOf(time_l.intValue());
                    setSpeed(95, 100);
                }
                Thread.sleep(time);
                stop();
                change = get_enc_rotation();
                System.out.print("Rotation of " + change);
                System.out.println(" degrees executed! (" + time + " ms)");
                orientation = (orientation + change)%360;
                if (orientation < 0) {
                    orientation = orientation + 360;
                }
                orientation = orientation - 1;
                print_location("new");
                System.out.print("The current location-goal slope is ");
                System.out.println(getLocationSlope());
                System.out.println();
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    // Make the mobile robot stop
    public void stop() {
        try {
            if (!goal_reached()) {
                System.out.println("Stopping motors");
                System.out.println();
            }
            robot.stopMotors();
            Thread.sleep(500);
            speed.set_vr(0);
            speed.set_vl(0);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    // Reset the goal point coordinates in cm
    public void reset(Point goal_new) {
        robot.resetCount();
        stop();
```

```
        start .setX( location .getX ());
        start .setY( location .getY ());
        goal .setX(goal_new.getX ());
        goal .setY(goal_new.getY ());
        cum_distance = 0;
        print_location (" initial ");
        System.out. println ();
        visited_points . clear ();
    }

    // Reset the goal point coordinates in cm and orientation in
    // degrees
    public void reset (Point goal_new, double start_orientation_new ) {
        robot . resetCount ();
        stop ();
        start .setX( location .getX ());
        start .setY( location .getY ());
        goal .setX(goal_new.getX ());
        goal .setY(goal_new.getY ());
        start_orientation = start_orientation_new ;
        orientation = start_orientation_new ;
        cum_distance = 0;
        print_location (" initial ");
        System.out. println ();
        visited_points . clear ();
    }

    // Make the mobile robot move to its Goal Point
    public void goToGoal() {
        try {
            Thread. sleep (700);
            if (! goal_reached ()) {
                goal_slope = getLocationSlope ();
                System.out. println ("The current location -goal slope is " + goal_slope );
                if (! obstacle_detected ()) {
                    motion_to_goal ();
                } else {
                    Point temp_location = getLocation ();
                    visited_points .add(temp_location );
                    wall_following ();
                }
            }
        } catch ( InterruptedException e) {
            e. printStackTrace ();
        }
    }

    // Make the mobile robot move to the new Goal Point
    public void goToGoal(Point goal_new) {
```

```
    try {
        Thread.sleep(700);
        goal.setX(goal_new.getX());
        goal.setY(goal_new.getY());
        Thread.sleep(700);
        if (!goal_reached()) {
            goal_slope = getLocationSlope();
            System.out.println("The current location-goal slope is " + goal_slope);
            if (!obstacle_detected()) {
                motion_to_goal();
            }else {
                Point temp_location = getLocation();
                visited_points.add(temp_location);
                wall_following();
            }
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

// A variant of the Motion to Goal Behavior that recalculates the slope between
// the current location and the goal point at the beginning of every iteration
// of the while loop. If an obstacle gets detected, an "hit" point is saved on a
// list and the mobile robot starts its Wall-Following Behavior. If the goal has
// been reached, nothing gets done by the algorithm.
public void motion_to_goal() {
    if (!goal_reached()) {
        System.out.println();
        System.out.println();
        System.out.println("********** Motion to Goal Behavior ... **********");
        System.out.println("********** Motion to Goal Behavior ... **********");
        System.out.println("********** Motion to Goal Behavior ... **********");
        System.out.println();
        System.out.println();
        while (!obstacle_detected() && !goal_reached()) {
            goal_slope = getLocationSlope();
            double orientation_new = Math.atan(goal_slope);
            orientation_new = Math.toDegrees(orientation_new);
            double orientation_1 = orientation_new;
            double orientation_2 = orientation_new + 180;
            if (orientation_1 > 0) {
                if (goal.getY() > location.getY()) {
                    orientation_new = orientation_1;
                }else {
                    orientation_new = orientation_2;
                }
            } else if (orientation_1 < 0) {
                if (goal.getY() > location.getY()) {
```

```
                    orientation_new = orientation_2;
                } else {
                    orientation_new = orientation_1;
                }
            } else {
                if (goal.getX() > location.getX()) {
                    orientation_new = orientation_1;
                } else {
                    orientation_new = orientation_2;
                }
            }
            if (orientation_new < 0) {
                orientation_new = 360 + orientation_new;
            }
            setOrientation(orientation_new);
            translate(30);
        }
        if (!goal_reached()) {
            Point temp_location = getLocation();
            visited_points.add(temp_location);
            wall_following();
        }
    }
}

// A "greedy" version of the Wall-Following Behavior that switches to Motion to
// Goal Behavior either if the method public boolean goal_line_hit(double init_slope)
// returns true or if more than 5 iterations of the Wall-Following Behavior have
// been executed. If the goal has been reached, nothing gets done by the algorithm.
public void wall_following() {
    if (!goal_reached()) {
        System.out.println();
        System.out.println();
        System.out.println("********** Wall-Following Behavior ... **********");
        System.out.println("********** Wall-Following Behavior ... **********");
        System.out.println("********** Wall-Following Behavior ... **********");
        System.out.println();
        System.out.println();
        count_wall = 0;
        double init_slope = getLocationSlope();
        while ((!goal_line_hit(init_slope) && count_wall <= 5) && !goal_reached()) {
            while (obstacle_detected() && !goal_reached()) {
                // make the robot be approximately parallel to the
                // obstacle
                avoid_obstacle('l');
                translate(12);
            }
            // rotate 90 degrees clockwise
            if (orientation - 90 < 0) {
```

```
                setOrientation ( orientation  - 90 + 360);
            }else {
                setOrientation ( orientation  - 90);
            }
             translate (12);
            count_wall++;
            if (count_wall > 5) {
                translate (-10);
                if (!goal_reached ()) {
                    System.out. println ();
                    System.out. print ("****** Leaving Wall-Following Behavior ... ");
                    System.out. println ("trying to get to the goal ! ******");
                    System.out. println ();
                    motion_to_goal ();
                }
            }
        }
        if (!goal_reached ()) {
            System.out. println ();
            System.out. print ("****** Leaving Wall-Following Behavior ... ");
            System.out. println ("trying to get to the goal ! ******");
            System.out. println ();
            motion_to_goal ();
        }
    }
}

// Returns true if an obstacle has been detected by the mobile
// robot's sensors
public boolean obstacle_detected () {
    if (robot. isPressed (0) || robot. isPressed (1)) {
        if (!goal_reached ()) {
            System.out. println ();
            System.out. println ("****** An obstacle has been hit !!! ******");
            System.out. println ();
        }
        return true;
    }else {
        return false;
    }
}

// Determines which bumper detected the obstacle and makes the robot
// act appropriately ('l' and 'r' for counterclockwise and clockwise
// rotation, respectively). This code has been written in a way to a
// allow further development for the programmer who wants the robot
// to perform different actions depending on which bumpers' sensors
// are activated.
public void avoid_obstacle (char ch) {
```

```
    double angle;
    double trans = -8;
    if (ch == 'l') {
        angle = 90;
    }else if (ch == 'r') {
        angle = -90;
    }else {
        angle = 180;
    }
    if ( obstacle_detected ()) {
        if (robot.isPressed(0) && robot.isPressed(1)) {
            translate (trans);
            if ( orientation + angle < 0) {
                setOrientation ( orientation + angle + 360);
            }else {
                setOrientation ((orientation + angle)%360);
            }
        }else if (robot.isPressed(0) && (!robot.isPressed(1))) {
            translate (trans);
            if ( orientation - angle < 0) {
                setOrientation ( orientation + angle + 360);
            }else {
                setOrientation ((orientation + angle)%360);
            }
        }else {
            translate (trans);
            if ( orientation - angle < 0) {
                setOrientation ( orientation + angle + 360);
            }else {
                setOrientation ((orientation + angle)%360);
            }
        }
    }
}

// Returns true if the mobile robot hits the goal slope it stored at
// the beginning of the Wall-Following Behavior, i.e. init_slope, at
// a "hit" point that has not been visited yet and the function
// goal_line_hit(double init_slope) has been called at least 2 times.
// It also returns true if init_slope is greater than 5, if the
// absolute difference between the x-coordinates of the current
// location and goal point is less than 5 and if the function
// goal_line_hit(double init_slope) has been called at least 2 times.
// It returns false otherwise.
public boolean goal_line_hit(double init_slope) {
    double location_slope = getLocationSlope();
    boolean flag = false;
    if (count_glh >= 2) {
        if ((100*Math.abs(init_slope - location_slope)/init_slope < 5) && !hp_visited()) {
```

```
                flag = true;
            }
            if ((init_slope > 5) && (Math.abs(location.getX() - goal.getX()) < 5)) {
                flag = true;
            }
        }

        if (flag == true) {
            if (!goal_reached()) {
                System.out.println();
                System.out.println("****** The start-goal line has been hit !! ******");
                System.out.println();
            }
            count_glh = 0;
            return true;
        }else {
            count_glh++;
            return false;
        }

    }

    // Returns true if the robot is within 7 cm of its current goal
    // point
    public boolean goal_reached() {
        if (getLocation().distanceTo(getGoal()) < 7) {
            if (count_goal_reached == 0) {
                print_location("new");
                System.out.println(" ****** The robot has reached its goal !!! ******");
                System.out.println("The total distance traveled is " + cum_distance);
                System.out.println();
            }
            count_goal_reached++;
            return true;
        } else {
            return false;
        }
    }

    // Get the slope of the line connecting the start point and the goal
    // point
    public double getGoalSlope() {
        double slope;
        if (goal.getX() == start.getX()) {
            slope = 999999999;
        }else {
            slope = (goal.getY() - start.getY())/(goal.getX() - start.getX());
        }
        return slope;
```

```
}

// Get the slope of the line conecting the goal point and the
// current location of the robot
 public double getLocationSlope() {
    double slope_location;
    if (location.getX() == goal.getX()) {
        slope_location = 999999999;
    }else {
        slope_location = (location.getY() - goal.getY())/(location.getX() - goal.getX());
    }
    return slope_location;
}

// Returns true if the mobile robot crosses one of the "hit" points
// previously visited
public boolean hp_visited() {
    Point temp_location = getLocation();
    boolean result = false;
    for(Point p : visited_points){
        if (temp_location.distanceTo(p) < 5) {
            result = true;
        }
    }
    return result;
}

// Get the "encoder" distance traveled by the mobile robot
public double get_enc_distance() {
    int d1 = robot.getCount(0);
    int d2 = robot.getCount(1);
    double enc_dist = (d1 + d2)/2;
    enc_dist = -6*Math.PI*enc_dist/64;
    robot.resetCount();
    return enc_dist;
}

// Get the "encoder" rotation made by the robot
 public double get_enc_rotation() {
    int d1 = robot.getCount(0);
    int d2 = robot.getCount(1);
    double enc_rot = (Math.abs(d1) + Math.abs(d2))/2;
    enc_rot = (enc_rot/enc_full)*360;
    if (d2 < 0) {
        enc_rot = -enc_rot;
    }
    robot.resetCount();
    return enc_rot;
}
```

```
// Get the cumulative distance traveled by the mobile robot
public double get_cum_distance() {
    return cum_distance;
}

// Print the current location of the robot
public void print_location (String current) {
    if (current.equals("initial")) {
        System.out.print("The initial ");
    }else if (current.equals("current")) {
        System.out.print("The current ");
    }else {
        System.out.print("The new ");
    }
    System.out.println("location and orientation of the robot are");
    System.out.print("(" + location.getX() + " cm," + location.getY() + " cm) ");
    System.out.println("and " + orientation + " degrees, respectively.");
}

// This class manages the speeds of the right and left wheels of
// the robot
private class Speed {
    private double vr; // speed of the right wheel
    private double vl; // speed of the left wheel

    // Constructor for zero Speed:(0,0)
    public Speed() {
        vr = 0;
        vl = 0;
    }

    // Constructor for Speed:(speed_right_wheel, speed_left_wheel)
    public Speed(double vr_new, double vl_new) {
        vr = vr_new;
        vl = vl_new;
    }

    // Set the speed of the right wheel
    public void set_vr(double vr_new) {
        vr = vr_new;
    }

    // Get the speed of the right wheel
    public double get_vr() {
        return vr;
    }

    // Set the speed of the left wheel
```

```
        public void set_vl(double vl_new) {
            vl = vl_new;
        }

        // Get the speed of the left wheel
        public double get_vl() {
            return vl;
        }
    }

}

// A point in the cartesian plane P:(x,y)
class Point {
    private double x;
    private double y;

    // Constructor for the origin point :(0,0)
    public Point() {
        x = 0;
        y = 0;
    }

    // Constructor for a Point :(x,y)
    public Point(double x_new, double y_new) {
        x = x_new;
        y = y_new;
    }

    // Set the x-coordinate of the point
    public void setX(double x_new) {
        x = x_new;
    }

    // Get the x-coordinate of the point
    public double getX() {
        return x;
    }

    // Set the y-coordinate of the point
    public void setY(double y_new) {
        y = y_new;
    }

    // Get the y-coordinate of the point
    public double getY() {
        return y;
    }
```

```
    // Get the distance of THIS point to another point
    public double distanceTo(Point point) {
        double delta_x = point.getX() - x;
        double delta_y = point.getY() - y;
        double distance = Math.sqrt(Math.pow(delta_x,2) + Math.pow(delta_y,2));
        return distance;
    }
}
```

6.2 Appendix B: RobotMission.java

```
/* File: RobotMission.java
 * Date: June 2015
 * King's College London -- Dept. of Informatics -- MSc in Robotics
 * Author: Claudio S. De Mutiis (claudio.de_mutiis@kcl.ac.uk)
 * Purpose: Plan a mission for the MIRTO robot, developed by a team led
 * by Dr Franco Raimondi (F.Raimondi@mdx.ac.uk) at Middlesex University
 * London.
 *
 * IMPORTANT NOTES:
 *
 * 1) WHERE FILES/FOLDERS SHOULD BE PLACED IN ORDER TO MAKE
 *    EVERYTHING COMPILE AND WORK CORRECTLY (on MIRTO's SD
 *    CARD):
 *    (i) MobileRobot.java in /csd2222
 *    (ii) RobotMission.java in /csd2222
 *    (iii) RobotTesting.java in /csd2222
 *    (iv) The following files and folders should also be placed
 *         in /csd2222: java-asip.jar, jssc, libs, META-INF and
 *         uk.
 *    (v) The modified version of the file JMirtoRobot.java (see
 *        below) should be placed in
 *        /csd2222/uk/ac/mdx/cs/asip
 *
 *    Source: https :// github.com/fraimondi/java-asip
 *
 * 2) The code in JMirtoRobot.java has been modified by Claudio S.
 * De Mutiis (claudio.de_mutiis@kcl.ac.uk) in August 2015.
 * Claudio S. De Mutiis added the method public void resetCount(),
 * which resets both of the encoders' counts.
 *
 *  public void resetCount() {
 *    e0.resetCount();
 *    e1.resetCount();
 *  }
 *
 * TRYING TO COMPILE THE CLASSES MobileRobot.java, RobotMission.java
 * and RobotTesting.java WITH THE OLD VERSION OF JMirtoRobot.java
 * WILL RESULT IN A COMPILATION ERROR !!!
 *
 * The modified version of JMirtoRobot.java is needed to make
 * everything compile and run correctly !!
 *
 * ADDITIONAL CREDITS:
 * - The class JMirtoRobot.java was developed by a team led by Dr
 *   Franco Raimondi (F.Raimondi@mdx.ac.uk) at Middlesex University
```

```
 *    London.
 * - In order to be able to work, the class RobotMission.java makes
 *    use of methods and classes developed by a team led by Dr
 *    Franco Raimondi at Middlesex University (see the folders lib
 *    and src and the files build.xml, README.md located in the
 *    folder java-asip-master).
 *    Source: https :// github.com/fraimondi/java-asip
 */

/*  Useful Math Functions:
 * 1) Math.toDegrees (...)  --> converts an angle from rad to deg
 * 2) Math.toRadians (...)  --> converts an angle from deg to rad
 */
/*********************************************************************
/* VERY USEFUL FUNCTIONS OF MobileRobot.java (see the comments
 * of MobileRobot.java file to learn about other available functions)
 * *******************************************************************
 * public void translate (double dist) --> Make the mobile robot
 *                          translate by "dist" cm
 * public void rotate (double change) --> Make the mobile robot rotate
 *                          by "change" degrees
 * public void stop () --> Make the mobile robot stop
 * public void reset (Point goal_new) --> Reset the goal point in cm
 * public void reset (Point goal_new, double start_orientation_new)
 *                          --> Reset the goal point in cm and orientation
 *                          in degrees
 * public void goToGoal() --> Make the mobile robot move to its goal
 *                          point
 * public void goToGoal(Point goal_new) --> Make the mobile robot move
 *                          to the new goal point
 * *******************************************************************
 *
 */

public class RobotMission {
    public static void main(String[] args) {
        // create a start point at the origin (0,0) --> current
        // location of the robot
        Point start = new Point();
        // create a goal point at (0, 150)
        Point goal = new Point(0,150);
        // specify the current orientation of the robot
        double orientation = 180;
        // create the robot object
        MobileRobot robot = new MobileRobot(start, goal, orientation);
        // send the robot to its goal location
        robot.goToGoal();
    }
}
```

6.3 Appendix C: RobotTesting.java

```
/* File: RobotTesting.java
 * Date: August 2015
 * King's College London -- Dept. of Informatics -- MSc in Robotics
 * Author: Claudio S. De Mutiis (claudio.de_mutiis@kcl.ac.uk)
 * Purpose: Test the basic rotational and translational capabilities
 *          of the MIRTO robot, developed by a team led by Dr Franco
 *          Raimondi (F.Raimondi@mdx.ac.uk) at Middlesex University
 *          London
 *
 * IMPORTANT NOTES:
 *
 * 1) WHERE FILES/FOLDERS SHOULD BE PLACED IN ORDER TO MAKE
 *    EVERYTHING COMPILE AND WORK CORRECTLY (on MIRTO's SD
 *    CARD):
 *    (i) MobileRobot.java in /csd2222
 *    (ii) RobotMission.java in /csd2222
 *    (iii) RobotTesting.java in /csd2222
 *    (iv) The following files and folders should also be placed
 *         in /csd2222: java-asip.jar, jssc, libs, META-INF and
 *         uk.
 *    (v) The modified version of the file JMirtoRobot.java (see
 *        below) should be placed in
 *        /csd2222/uk/ac/mdx/cs/asip
 *
 *    Source: https :// github.com/fraimondi/java-asip
 *
 * 2) The code in JMirtoRobot.java has been modified by Claudio S.
 * De Mutiis (claudio.de_mutiis@kcl.ac.uk) in August 2015.
 * Claudio S. De Mutiis added the method public void resetCount(),
 * which resets both of the encoders' counts.
 *
 *  public void resetCount() {
 *    e0.resetCount();
 *    e1.resetCount();
 *  }
 *
 * TRYING TO COMPILE THE CLASSES MobileRobot.java, RobotMission.java
 * and RobotTesting.java WITH THE OLD VERSION OF JMirtoRobot.java
 * WILL RESULT IN A COMPILATION ERROR !!!
 *
 * The modified version of JMirtoRobot.java is needed to make
 * everything compile and run correctly !!
 *
 * ADDITIONAL CREDITS:
 * - The class JMirtoRobot.java was developed by a team led by Dr
```

```
 *    Franco Raimondi (F.Raimondi@mdx.ac.uk) at Middlesex University
 *    London.
 * - In order to be able to work, the class RobotTesting.java makes
 *    use of methods and classes developed by a team led by Dr
 *    Franco Raimondi at Middlesex University (see the folders lib
 *    and src and the files build.xml, README.md located in the
 *    folder java-asip-master).
 *    Source: https :// github.com/fraimondi/java-asip
 */

/*  Useful Math Functions:
 * 1) Math.toDegrees (...)  --> converts an angle from rad to deg
 * 2) Math.toRadians (...)  --> converts an angle from deg to rad
 */
/*********************************************************************
/* VERY USEFUL FUNCTIONS OF MobileRobot.java (see the comments
 * of MobileRobot.java file to learn about other available functions)
 * *******************************************************************
 * public void translate (double dist) --> Make the mobile robot
 *                          translate by "dist" cm
 * public void rotate (double change) --> Make the mobile robot rotate
 *                          by "change" degrees
 * public void stop () --> Make the mobile robot stop
 * public void reset (Point goal_new) --> Reset the goal point in cm
 * public void reset (Point goal_new, double start_orientation_new)
 *                          --> Reset the goal point in cm and orientation
 *                          in degrees
 * public void goToGoal() --> Make the mobile robot move to its goal
 *                          point
 * public void goToGoal(Point goal_new) --> Make the mobile robot move
 *                          to the new goal point
 * *******************************************************************
 *
 */

public class RobotTesting {
    public static void main(String[] args) {
        // create a start point at the origin (0,0) --> current
        // location of the robot
        Point start = new Point();
        // create a goal point at (50, 50)
        Point goal = new Point(50,50);
        // specify the current orientation of the robot
        double orientation = 180;
        // create the robot object
        MobileRobot robot = new MobileRobot(start, goal, orientation);
        robot.translate(20);
        robot.translate(-20);
        robot.translate(20);
```

```
            robot . rotate (-180);
            robot . rotate (90);
        }
}
```