Robotics Report

Group Coursework

Riccardo Mangiapelo
Jack Moore
Prakash Waghela
Inthuch Therdchanakul
Jason Owusu-Afriyie
Matt Turner
Matthew Coates

16 December 2016

Part A

Reactive Braitenberg behaviours

Design

In this part of the coursework, we have attempted to recreate four Braitenberg behaviours: aggression, fear, love and curiosity. Such behaviours can be achieved by controlling the speed of the robot (i.e E-Puck) based on the sensor detection values. In fact, as explained by Braitenberg (1984), it is possible to simulate 'simple' behaviours by increasing/decreasing the speed of the motors for high value sensors. By taking into account a vehicle with two front sensors, one on each side, and two motors, right and left (Figure 1) it is possible to have two kind of vehicles, depending on whether the sensor to the motor is connected on the same side (a) or on the opposite side (b). Using such design, it is possible to make the vehicle respond in different ways in the presence of a stimulus. In fact, as Braitenberg (1984) illustrates, by having a positive influence sign, the vehicle will tend to accelerate when the sensors are excited. If the source is directly ahead, it may hit the object. However, if the source is on one side, one sensor is excited more than the other, resulting in different behaviours (observable in Figure 2). Vehicle (a) would accelerate the wheel parallel to the activated sensor, eventually running away from the detected object. This behaviour is named *Fear*. On the contrary, if the object is on one side of vehicle (b), this will turn toward the object and ultimately hit it. Such behaviour is called *Aggression*.

By switching influence sign from positive to negative, the vehicles respond differently to a stimulus, resulting in a new set of behaviours. By having a negative influence sign, the motor attached to the activated sensor will slow down in the presence of a stimulus. As it can be observed in Figure 3, when approaching the source, vehicle (a) will decrease the speed of the parallel sensor, eventually stopping close to it. Such behaviour is called *Love.* On the other hand, by having the sensor-

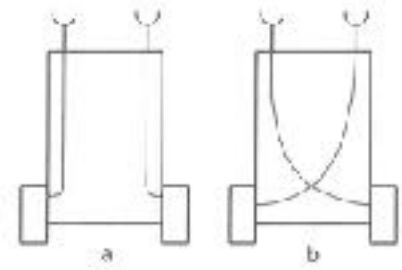


Figure 1: *Vehicle 2* (Braitenberg, 1984, p.7) with two motors and two sensors.

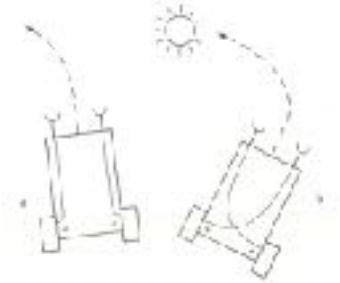


Figure 2: *Vehicle 2a and 2b* (Braitenberg, 1984, p.8) simulating Fear and Aggression.

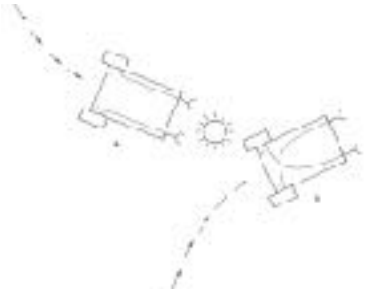


Figure 3: *Vehicle 3a and 3b* (Braitenberg, 1984, p.11)
simulating Love and Curiosity.

motor control crossed, the vehicle decreases its speed in the presence of the source while turning away. This will result in an increase of the speed as the vehicle gets away from the source (Braitenberg, 1984). This is referred to as *Explorer* (or *Curiosity*).

All the four behaviours have been successfully implemented in the E-Puck robot. In order to make the robot exhibit the behaviours, we used four different selectors:

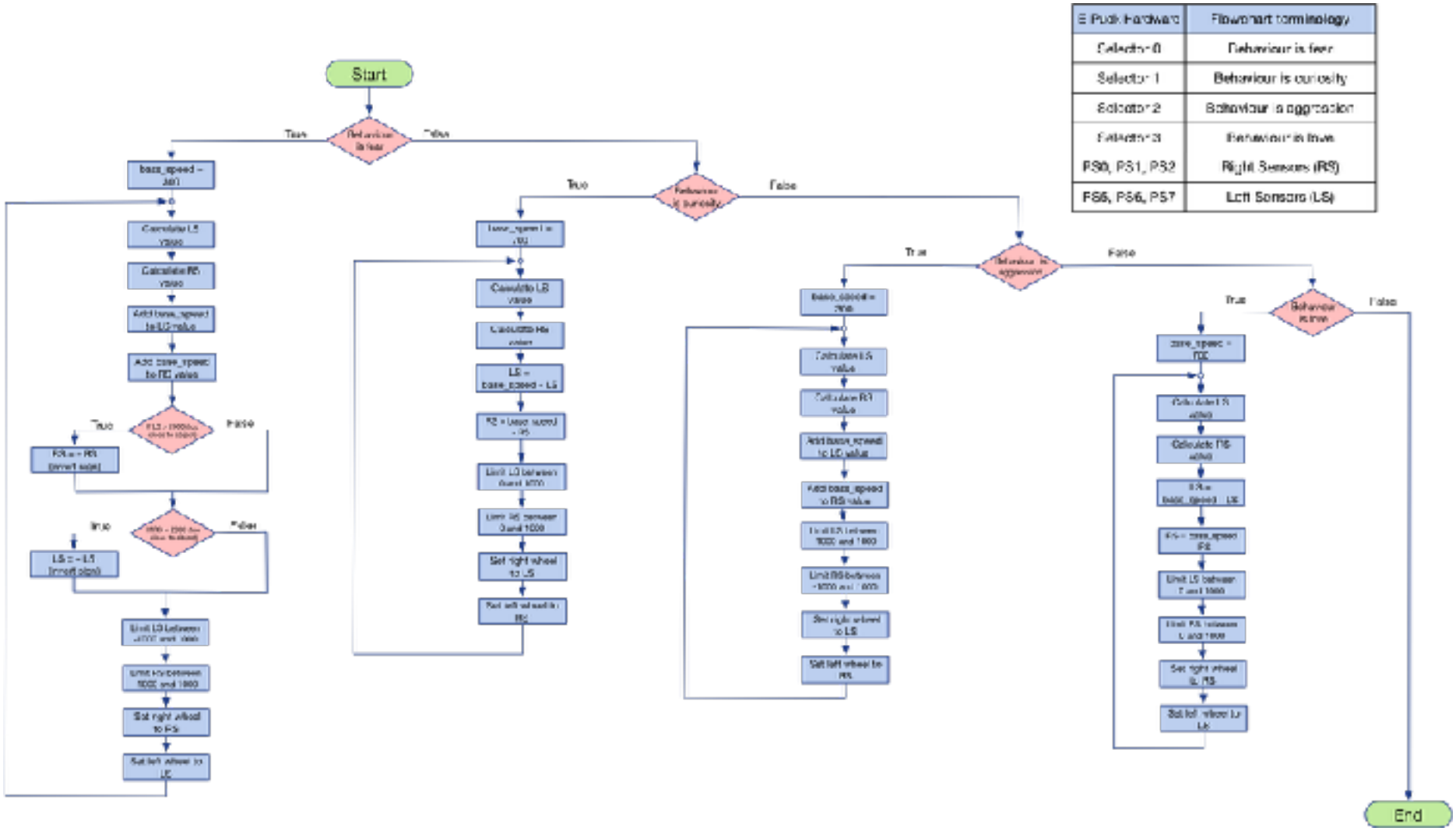
- selector 0: fear;
- selector 1: curiosity;
- selector 2: aggression;
- selector 3: love.

Consequently, the robot is only capable of representing one behaviour at the time. This results in the robot equally treating every detected object according to the selected behaviour.

Based upon Braitenberg's approach, we classified E-Puck's proximity sensors into two categories: "right sensors" and "left sensors". Under this scheme, we considered the top-right proximity sensors of the E-Puck (PS0, PS1, PS2) to be "right sensors" and the top-left ones (PS5, PS6, PS7) to be "left sensors". The reason for neglecting to use the rear sensors (PS3, PS4) is to prevent the robot from behaving eccentrically while moving away from the source (for instance, being attracted back to the object). Logically, this could be explained by thinking that the robot could only sense what is in front of it.

The left and right sensors are used to constantly monitor the adjacency of the robot's surroundings, returning high values when in proximity of an object. To such values is then added a basic speed, which is set differently for each behaviour. The result is then used to adjust the corresponding wheel's speed, according to the behaviour. In the case of *Fear*, these values are further used to examine whether the robot is excessively close to an object. In such cases, then the sign of the values are inverted, allowing the robot to move backwards slightly, exhibiting the behaviour correctly. A more detailed insight of the process can be observed in the flowchart overleaf.

Flowchart and function description



E Puck Hardware	Flowchart terminology
Selector 0	Behaviour is fear
Selector 1	Behaviour is curiosity
Selector 2	Behaviour is aggression
Selector 3	Behaviour is love
PS0, PS1, PS2	Right Sensors (RS)
PS5, PS6, PS7	Left Sensors (LS)

4

Description of the main program functions for Part A:

fear()

This function contains the *Fear* behaviour. The basic speed, the speed of the wheels when there is no sensory input, is set to 300. As a result, when the robot meets an object, the increase on the speed is more noticeable. The maximum speed of the wheel is set to 1000, while the minimum is set to -1000 (this represent the fastest speed the wheels can go in reverse). After having detected the proximity values for each sensor set (left and right), each side's sensor value is set to be the sensor value itself added to the basic speed. The right wheel speed is then set to the newly calculated right sensor value; similarly, the newly calculated left sensor value is assigned to the left wheel. However, if any of these two values are over a limit of 2000 (meaning that the object is extremely close to the robot) the sign of the value is inverted. This allows the robot to move backwards slightly, exhibiting the behaviour correctly. Hence, as the sensory value for a sensor on one side increases, the motor for the same side of the robot will increase in speed.

curiosity()

This function contains the *Curiosity* behaviour. The basic speed is set to 700, in order to make the robot relatively fast when it is not detecting any objects. By doing so, the reduction of the speed is more noticeable when the robot approaches an object. The maximum speed of the wheel is set to 1000, while the minimum is set to 0 (this represent the fastest speed the wheels can go in reverse). After having detected the proximity values for each sensor set (left and right), each side's sensor value is set to be the sensor value itself subtracted by the basic speed. The right wheel speed is then set to the newly calculated left sensor value; similarly, the newly calculated right sensor value is assigned to the left wheel. Hence, as the sensory values increase at the front, the robot will slow down. This will make the robot repel the object once close enough, speeding up as it moves away.

aggression()

This function contains the *Aggression* behaviour. As per the fear() function, the basic speed is set to 300, while the maximum and minimum speed of the wheel is set to 1000 and -1000 respectively. After having detected the proximity values for each sensor set (left and right), each side's sensor value is set to be the sensor value itself added to the basic speed. The right wheel speed is then set to the newly calculated left sensor value; similarly, the newly calculated right sensor value is assigned to the left wheel. As a result, as the sensory value for a sensor on one side increases, the motor for the opposite side of the robot will increase in speed.

love()

This function contains the *Love* behaviour. As per the curiosity() function, the basic speed is set to 700, while the maximum and minimum speed of the wheel is set to 1000 and 0 respectively. After having detected the proximity values for each sensor set (left and right), each side's sensor value is set to be the sensor value itself subtracted by the basic speed. The right wheel speed is then set to the newly calculated right sensor value; similarly, the newly calculated left sensor value is assigned to the left wheel. Therefore, as the sensory values increase, the robot is 'attracted' to the object, causing the increase. The robot also slows down as it gets closer. After getting close enough, the robot will stop and perform a signal of 'love' (lighting up the body lights).

Test procedure and result analysis

In order to test the correctness of each behaviour, no special environment's set up was required. It has been sufficient to place the robot on a straight surface (e.g. a table) long enough for the robot to safely navigate on it. A handful of objects were then positioned on the robot's path in order to observe the response of the vehicle to the object.

The results have fulfil expectations. Each behaviour is exhibited correctly from the robot, as per behaviour explanation in section Design.

Conclusion

The experiment in this Task simulated four Braitenberg's behaviour (fear, curiosity, aggression, love) on an E-Puck robot. By adjusting the wheels speed according to the sensor values and by changing the influence sign from positive to negative, we have been able to create the set of behaviours required. Upon the correct choice of the selector, the E-Puck is now capable of reacting differently when approaching obstacles.

Part B

Goal seeking and obstacle avoidance

Design

Our task for this section of the coursework was to build a robot that would find its way to a specified goal whilst avoiding obstacles. For this task we decided to use the Bug-2 algorithm as explained in the lectures.

The idea behind the basic version of the Bug-2 algorithm is to plot a line from the start point to the goal (i.e. the m-line), moving along that line towards the goal. If the robot encounters an object, it would follow its perimeter until it reaches a closer point of the m-line than it was before. When this happens, the robot will stop following the perimeter of the object and continue navigating along the m-line towards the goal (as shown on Figure 4).

The Bug-2 algorithm is a greedy algorithm and generally outperforms the Bug-1 algorithm except for some cases where there are many objects intersecting the m-line.

The method we used to set the start and end point were to implement a co-ordinates system, where we specified the start point as (0, 0) in the X and Y directions respectively, and the goal point as any arbitrary (X, Y). This meant that to keep track of the robot's current position in the environment, we had to store the robot's current rotation in respect to its starting rotation, and use that combined with how far the robot has moved since we last calculated its position. In order to prevent the

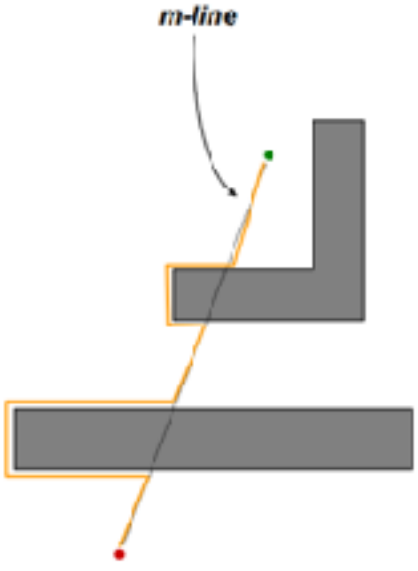


Figure 4: *Bug-2 Algorithm.*

calculation from becoming too complex, we allow the robot to only rotate when it is stationary.

To detect an obstacle while travelling along the m-line, we average the value of the front sensors. If such value is above a set threshold, an object has been detected. Hence, the robot begins rotating left, since we opted for a left-leaning variant of the algorithm. The robot will rotate until it is roughly perpendicular to the object, at which point it will continue moving forwards until the average of the right hand side sensors no longer detect the object. The robot would then rotate right and move forwards, keeping roughly perpendicular with the object. This process is repeated until either the object is completely circled or the m-line is detected. If the latter happens and the robot's position is closer to the goal than when it started navigating the object, then it moves on towards the goal.

Flowchart and function description

distToGoal()

We use this function to calculate the distance from where the robot currently is to the goal. To do this we calculate the distance away from the goal in the x-direction and y-direction, and then use Pythagoras's theorem to get the direct straight line distance.

stepsToRotate()

This is a utility function where we give the robot the angle in degrees we want to rotate clockwise, and it returns the number of steps the wheels have to turn to rotate that far. To do this we know that a full rotation of the wheel is 1000 steps, and we calculate how far 1000 steps will rotate us by using the diameter of the wheels and the diameter of the robot.

angleFromGoal()

This function gives us the angle we need to rotate clockwise by to face the goal, taking into account the angle we are currently rotated at. We work out the distance from the goal in the x direction and the y direction and then use trigonometry to get the angle we need to rotate by from 0. We then take away the angle we are currently rotated at to get the angle we need to rotate by in total. We can then pass this into our stepsToRotate() function to tell the wheels how far to turn.

distToMline()

We use this function to tell us if we are on the m-line from the start to the goal or not, if the distance away from any point on the line is less than a certain threshold then we say we are on the m-line.

setGoal()

This allows us to specify the goal position in x and y coordinates.

moveToGoal()

Called at the start of the program and again each time we have finished navigating around an object and have encountered the m-line again. Uses angleFromGoal() to find out how far it has to rotate to point towards the goal, taking into account if rotating clockwise or anticlockwise would be faster based on current orientation, and then moves forwards towards the goal until it reaches it, or encounters an obstacle.

updateProgress()

This function is called after every time we have finished moving somewhere, such as when we encounter an object. It takes the distance we have travelled in steps since we last finished moving, and the angle we are currently rotated by, and uses trigonometry to calculate how far we have moved in the x and y directions, and adds that to our current position.

avoidBoundary()

This function runs as soon as the robot detects an object, and if the front sensor values are above a certain threshold, then we rotate the robot left until the front sensors no longer detect the object. The robot will continue forwards until the right sensors no longer detect the robot, in which case it will rotate right and move forwards until it does again. It will continue moving around the object until it reaches the m-line, or it has reached its start location again. If while moving around the obstacle it detects another object in front of it that is in the way, that could mean there is another object to navigate around, or the current object is not rectangular. In this case it will call the avoidBoundary() function again, recursively.

pathfinder()

Contains the main loop for the bug-2 algorithm. It will keep moving forwards towards the goal using the moveToGoal() function until it either reaches the goal or the front sensors detect an obstacle. If the robot reaches the goal then it stops and

lights up all of the LEDs to let you know. If it reaches an object it uses our avoidBoundary() function to navigate around it. Once the robot has navigated around the object and has detected that it is on the m-line, then it will call the moveToGoal() function again to move towards the goal. If the robot does not detect a point on the m-line around the object that is closer to the goal than when it started, then it will stop as it cannot reach the goal.

Test procedure and result analysis

In order to test the correct behaviour of the robot, we tested each subpart of the program individually. This allowed us to partition the testing into subtests of each functionality. One of these included placing the robot on a flat surface, setting the goal to 1000 units away directly in front of the starting point (i.e. 1000 units in the y direction). By doing this, we observe the robot moving forwards towards the goal, stopping once at the goal position. This showed the correct behaviour of the functionality.

A further test we conducted was to evaluate the ability of the robot to move to a specified location while considering its current orientation. In order to do this, we tested the capability of the robot to move 1000 units up, right, down and left, ending in its initial location.

Further to the above, we tested a simple object avoidance behaviour by placing a single object in front of the robot. We observed that the robot would stop upon the detection (by the front sensors) of the obstacle. Next, the robot would turn and navigate around to the other side of the object.

Once the robot could reach a goal and successfully navigate around an object, we implemented the Bug-2 algorithm, combining these behaviours. We tested the Bug-2 algorithm using different sized obstacles in different orientations, and with the goal in different places. This showed the robot successfully navigating through the m-line, avoiding object(s) and finally stopping at the goal.

Conclusion

Overall, the Bug-2 algorithm works well and is efficient in most cases where there aren't too many obstacles intersecting with the m-line. By storing the robots current rotation and tracking how far it moved we were able to calculate its current position relative to the world, and to allow it to move towards a specified goal whilst avoiding obstacles using the IR sensors.

Part C

High level behaviour

Design

Design explanation.

Flowchart and function description

The flowchart of the system + Description of the classes or program functions

Test procedure and result analysis

The program test procedure, results and analysis

Conclusion

Conclusion for this part.

Bibliography

Braitenberg, V. (1984) V*ehicles: Experiments in Synthetic Psychology*. Massachusetts: The MIT Press.