

# Robotics 3



RLT Robotics Learning Track

Module for Level 2  
Version 4.3 - Sept 2012

# Colophon

This Robotics module is part of the RoboDidactics Robotics Learning Track (RLT). The material presented in this module is based on the Dutch Robotics material developed by the author for the SLO Certified Robotics Module.

All the original and associated material for the RoboDidactics Learning Track may be downloaded from the Phyrtual site or from the RoboPal for NXT site at [www.virtualbreadboard.com](http://www.virtualbreadboard.com). It can also be found in the English download section of [www.robocupjunior.nl](http://www.robocupjunior.nl). Teachers are permitted to modify this material for use in their own lessons, provided these changes are reported in the colophon of the modified material. The Phyrtual site can be reached through [www.phyrtual.org](http://www.phyrtual.org).

This module was developed and translated by the author (Peter van Lith) as part of a cooperation agreement with the Fondazione Mondo Digitale in Rome, Italy in 2010. The RoboPAL software used in this version has been developed with VirtualBreadBoard by James Caska.

This version is developed for use with the Lego MindStorms NXT and RoboPAL software. A more extensive version is available (currently only in Dutch). It is based on robots that can be programmed in Java, using the Java Simulator and Eclipse.

This NXT version is easy to use because it uses a graphical programming language.

The module consists of three parts. The first is the basic version needed for all further lessons. The second part deals with the RoboCup Junior Rescue challenge. The more demanding third part is optional and deals with a simple simulated organism, based on reactive behaviour.

Modified versions of this module may only be distributed if this colophon states that it is a modified version, including the name of the author of the modifications.

© 2010/12. Version 4.3

The copyright of this module rests with RoboCup Junior Netherlands that is the owner under the terms of the creative commons license as mentioned below.

The authors of this module have used material from third parties during its development and have received permission to use this material. During

research into the rights of text and illustrations, we have acted carefully. Should, however, any person or organization deem to have rights to parts of the text or illustrations, they are advised to contact RoboCup Junior Netherlands (info@robocupjunior.nl).

This module has been compiled with care and has been tested extensively by the authors and several test schools. The authors accept no responsibility for incorrect or incomplete parts of this module, nor do they accept any claims for damages as a result of using this module or its associated software.



This module is distributed under the Creative Commons License 3.0, Netherlands.

► <http://creativecommons.org/licenses/by-nc-sa/3.0/nl>

# Contents

- COLOPHON ..... 2**
- CONTENTS..... 2**
- INTRODUCTION..... 5**
- 8. ADAPTIVE BEHAVIOUR ..... 6**
  - THE SENSE-REASON-ACT LOOP ..... 7
  - SENSES VERSUS SENSORS ..... 7
  - DRIVING AROUND AT RANDOM..... 14
- 9. ADVANCED SENSORS ..... 18**
  - REACTING TO FAST AND SLOW MOVEMENTS ..... 19
  - DETECTING MOVEMENT ..... 19
  - USING THRESHOLD VALUES ..... 19
  - MAKING THE ROBOT REACT - CURIOUS..... 23
  - MAKING THE ROBOT REACT - SCARED ..... 24
  - SCARING THE ROBOT ..... 25
- 10. CONTROL SYSTEMS..... 28**

# Introduction

## Intro Parts 1 and 2

The RLT Robotics module consists of three parts that can be followed independently. In the first part, the emphasis is on learning about the robot, its development environment and how to program it with RoboPAL. The first part is intended for higher middle grade school and lower science education classes.

The second part focuses on the RoboCupJunior Rescue mission in which a dangerous container has to be removed from a swamp area in the Rescue Field. This part concentrates on simple programs and is useful for schools that want to participate in robotics competitions such as RoboCup Junior and the First Lego League.

## Intro Part 3

In the third part of the RLT Robotics module, we address what is known as *Adaptive Behavior*. This is the most difficult of the three parts. It includes a detailed explanation of the Sense-Reason-Act loop and information on the use of *State Machines*. We will copy the behavior of a (very) simple organism and program the robot to exhibit scared, curious and random behavior.

These three parts have been organised so that students may follow Parts 1 and 2 or Part 1 followed by Part 3. This final part is the most complicated and more suitable for a theoretical education. Part 2 (Chapters 5-7) is not superfluous and indeed provides a good basis for the material in Part 3. If you have sufficient time, of course, you could follow all three parts.

# 8. Adaptive Behaviour

We are going to make the robot more reactive to its environment. This is called *adaptive behaviour*. Instead of pre-programmed behaviour, the robot will change its behaviour based on the circumstances it encounters. We will also introduce a new concept: *States*.

States are used to determine the situation of the robot. This requires a structure that we will first present in a schematic manner as a state diagram. Subsequently, we will develop the state diagram into what is called “an architecture.” This is a structure into which the program parts are embedded step by step.

We will also see how the robot can use the information from its sensors to become scared or curious. In addition, we will also see how we can make a robot exhibit random behavior, so that it does not repeat the same movements cyclically, but just wanders about without any fixed pattern.

## 8.1 You will learn how to

- program adaptive behaviour
- use random numbers
- program a robot to become scared, curious or indifferent

## 8.2 You will need

- a NXT robot
- a computer with RoboPAL
- a Grid field
- the *CuriousBehavior* and *FleeBehavior* programs









## 8.3 You will experiment with

- copying small parts of the behavior of an ant
- modifying *FleeBehavior* and *CuriousBehavior*
- distinguishing between behaviors and states

## 8.4 After completing this chapter, you will be able

- to read and interpret a simple state diagram
- to use dummy routines
- to explain when random numbers are useful

Type	#	Assignment	Description
	<b>8A</b>	<b>AdaptiveBehavior</b>	<b>State Diagram</b>
	8A.1	Study	Studying the state diagram
	<b>8B</b>	<b>AdaptiveBehavior</b>	<b>Architecture</b>
	8B.1	Code modification	Setting up an architecture for adaptive behavior
	8B.2	Code modification	Entering the act function
	<b>8C</b>	<b>WanderBehavior</b>	<b>Using a dummy routine</b>
	8C.1	WanderBehavior	Testing the WanderBehavior dummy routine
	<b>8D</b>	<b>WanderBehavior</b>	<b>Random behavior</b>
	8D.1	Code modification	Driving around; random numbers.

## 8.5 Explanation

First, we will refresh our memory by reviewing part of the introduction to the Sense-Reason-Act loop that was presented in the chapters 5-7.

### The Sense-Reason-Act Loop

Three logical steps are used to control robots. You will find these same steps in the behavior of animals and possibly in humans, too. First, we observe things with our senses: the *Sense* step. Then, we decide what to do with this information: the *Reason* (or thinking) step. Finally, we take an action: the *Act* step.

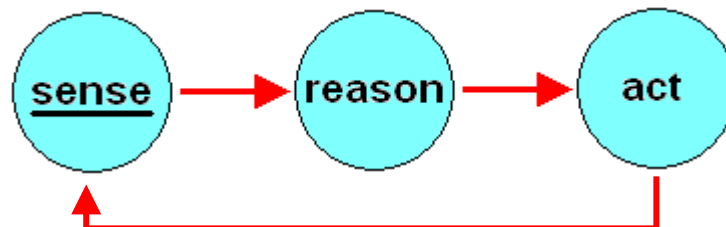


Fig 1: Sense-Reason-Act Loop

Robots use their sensors to collect information on their environment. In this chapter, you will see how a robot uses this information in the Sense step.

### Senses versus Sensors

By observing the environment with its sensors, a robot can adapt to the circumstances around it and operate more flexibly to reach its goal. Living organisms, whether they are single-celled or human beings observe their environment through their senses. Robots are equipped with sensors for the very same purpose. There are a large variety of sensors, but in general they perform the same function as the senses of a human being or animal.

Some sensors, like infrared or Röntgen rays, even detect information that we cannot perceive with our senses. Most sensors, however, are more

limited. Indeed, we sometimes need to use several different sensors just to match the operation of one of our senses.

We are so familiar with our senses that we often assume that sensors on a robot work in an identical manner, but this is often not true. The processing of raw data by a sound sensor, for instance, may lead to very different conclusions than those reached by people using their ears. In addition, our experiences are highly influenced by the associations our brains make with these observations (i.e., optical illusions).

Moreover, two sensors with the same function may provide different results due to small differences in their manufacturing process. These differences make it necessary to match sensors. We do this on the basis of a standard reference value. The process is called *tuning*.

When, for example, a sensor is tuned to represent raw data values on a scale from 0 to 100, we call this process *calibration*. The process of adapting the non-linear behavior of a sensor to linear behavior is also called calibration. Calibration makes it a lot simpler to compare the values of different sensors.

In this chapter, we will address the idea of adaptive behavior in greater detail. We will re-develop *FleeBehavior* and *CuriousBehavior* in a different and more extensive manner, so that your robot reacts to its environment. As a result, the robot will exhibit a more interesting behavior. This is a complex subject, so we will analyse it step by step. The best approach is to develop and test each piece of code separately on the simulator. If an error occurs, resolve the problem before moving on.

You will also have to make your own backup copies, preferably before every major step.

## State Diagram (Scared, Curious and Neutral)

### Adaptive Behavior

Earlier, we saw that adaptive behavior is a simple way of making an organism or a machine react to its environment.

If you tease an ant with a stick, it may decide to deviate from its path to avoid the obstacle. This is an example of adaptive behavior: the ant leaves its trail to move away from the stick.

The ant modifies its behavior according to the situation. You can also program a robot to behave like this. A robot may find itself in different *states*. In order to make a robot adapt to its environment, we first need to create a state diagram and then develop a program with a so-called *state-machine*.



## State Diagram

What kind of situation could scare a robot? A large object headed directly towards it or something coming at it at high speed, or both.

In such a situation, a robot may choose to flee. In other cases, it may choose to explore its environment or even to chase the object. If, on the other hand, nothing happens around it, the robot may choose to start driving around randomly.



### 8A.1 Assignment: Studying the State Diagram

Look at the following state diagram:

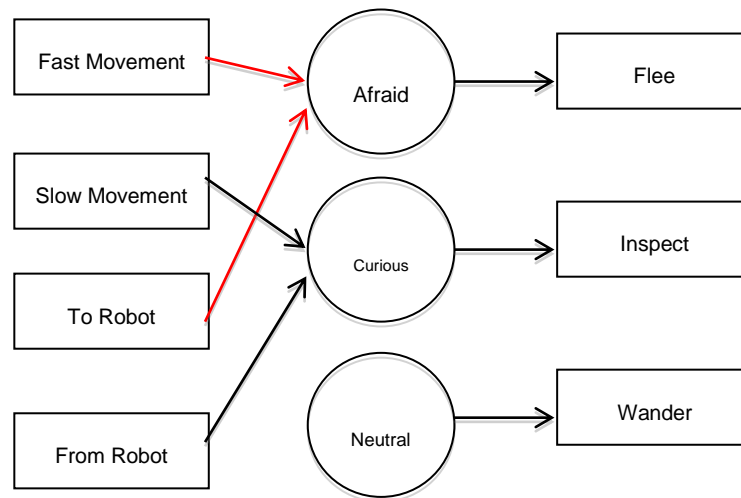


Fig 2: State Diagram

The behaviour delineated by the state diagram includes the following steps:

- As long as there are no stimuli, the robot remains neutral and drives around, waiting for something to happen.
- If something comes towards the robot, it gets scared and flees.
- If it detects a slow movement or if something moves away, it becomes curious and chases it.

Clearly, the behaviour of an ant is much more complex, because ants can detect odours, sounds and vibrations. Moreover, the transition from bright to dark lighting and the size of an object may also play a role.

In order to mimic this behaviour, you will have to modify the *CuriousBehavior* program so that the robot autonomously determines whether it should be scared or curious. According to the state diagram shown above, the robot should be able to determine if something is moving fast or slow and whether it is coming towards or moving away from it. Depending on the possible combinations, the *state* of the robot will be scared, curious or neutral. This, in turn, will determine whether the robot flees, moves towards the object or just moves about (seemingly uninterested or maybe quite alert).

These are different behaviours and situations that can be developed based on the following elements:

- is something moving fast or slow?
- in what direction?
- the state of the robot
- the robot must flee, become curious or neither
- the robot must drive around

## Architecture - Infrastructure

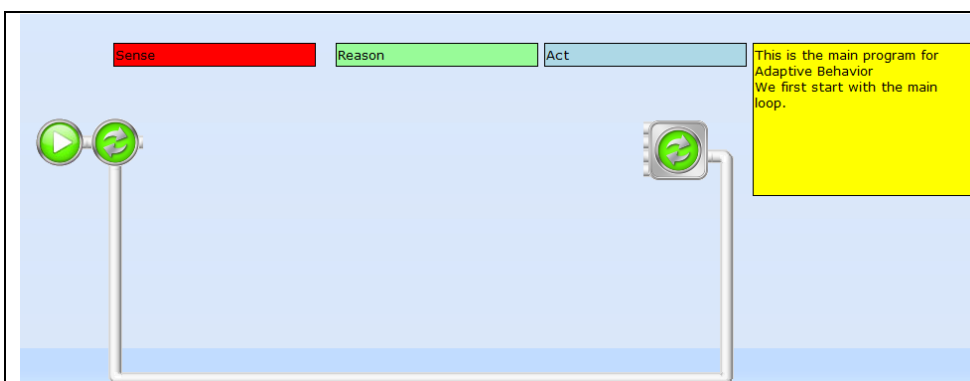
We are going to program several actions, so we will first need to build a structure (sort of like scaffolding) on which to place the various possible actions. This type of construction is called an *infrastructure* and its design is called its architecture. When developing computer software, the design of the architecture is always one of the most important steps.

As our architecture will be identical to that planned in the state diagram (fig. 3), we only need to concentrate on how to develop it.



### 8B.1 Assignment: Setting up an Architecture for Adaptive Behavior

- Load version 8B (which is a copy of *FleeBehavior*) and call it 8B.1 - Adaptive Behavior.
- Open the *Main* program and select all the icons. Then select Edit | Copy.
- Make three new FlowCodeSheets and call them *FleeBehavior*, *CuriousBehavior* and *WanderBehavior*. We will not add anything else for the moment. Make sure that the Start and Finish icons are connected; otherwise, the program will not work.
- Now, insert a copy of the three text boxes from FlowCode 1 in each of these new FlowCodeSheets by using the Edit | Paste command. You now have three identical subroutines that do not do anything useful, yet.
- Insert a loop in the *Main* routine as shown in FlowCode 1.



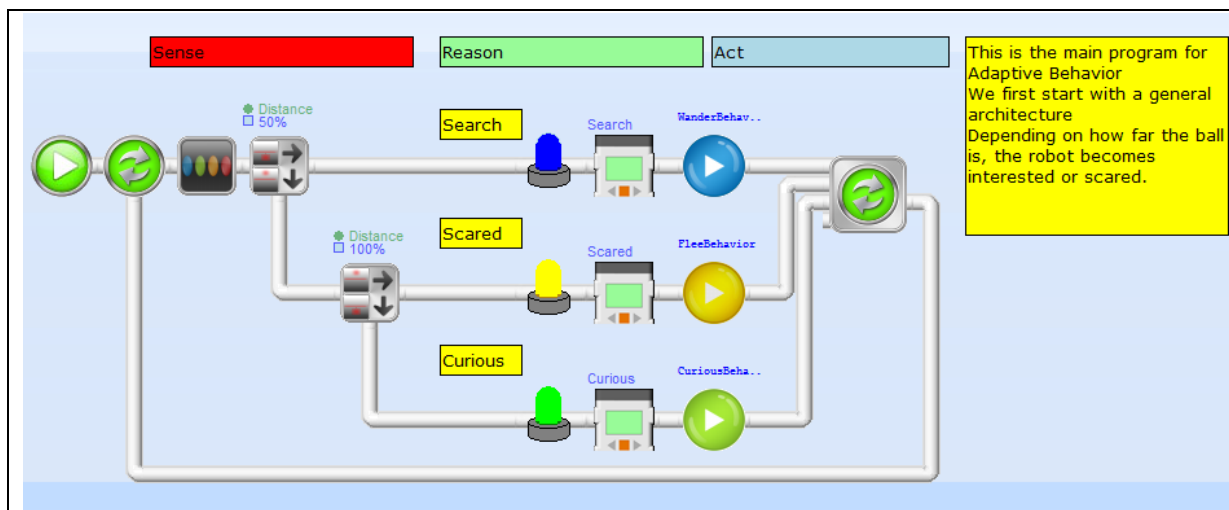
FlowCode 1: the Sense-Reason-Act Main Loop

This program does not do anything yet, but it forms the basic architecture onto which we will develop functionality, step by step. Use different colours to identify the Sense, Reason and Act steps. Also note that the Sense-Reason-Act Loop contains a four-way merge icon that we will later need to connect to various subroutines.



### 8B.2 Assignment: Making the Act Function




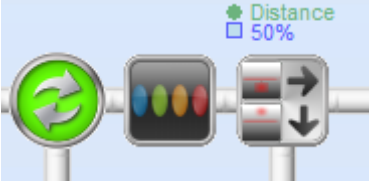
- Call your program 8B.2
- Insert the instructions shown below into the Reason and Act parts.
- Now, you can start testing your program. Follow the instructions provided below.



FlowCode 2: Basic Setup for Act

We will start with the construction of the Sense-Reason-Act functions. The first two icons test at what distance a ball can be detected. On the basis of this result, we will determine what the robot must do or - in other words - its State. The program then determines the state and calls the corresponding behaviour. We explain what is happening below.

8B.2 Entering the Act Function	
1	
	<p>Call your program 8B.2 and include two new icons. The first is a BranchIfLower icon that uses the Distance sensor. If its value is lower than 50%, it means that the ball is far away and the robot will continue to search for it. The second icon is a BranchIfHigher icon that checks to see if the ball is near. If it is, the robot must get scared. If both these conditions are false, then the robot sees the ball and must become curious. We must make sure the robot actually does something and reveals its State.</p>

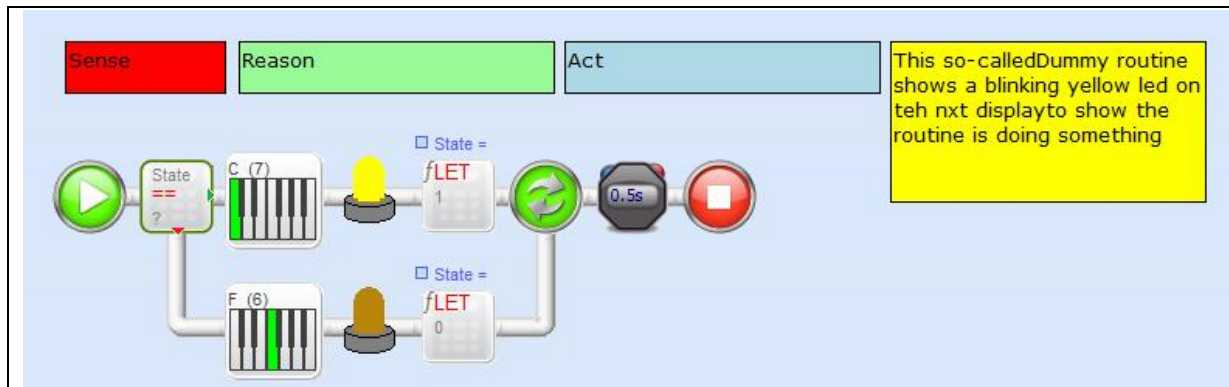
2		<p>Let's agree to use the following LED icons: Blue for Search, Yellow for Scared and Green for Curious.</p> <p>Insert an LCDDebugMessage icon after the LED icon and a third text variable for Search, Scared and Curious.</p>
3		<p>Depending on its State, the robot will need to call the corresponding subroutine. Insert Subroutine icons for WanderBehavior, FleeBehavior and CuriousBehavior into your program and give them the same colour as the corresponding LED.</p>
4		<p>Now, you can start testing your program. Start the program in the simulator and use the mouse to move the ball towards the robot. Look at the message that appears on the LCD screen.</p> <p>Two things will go wrong:</p> <ul style="list-style-type: none"> <li>• First, the LEDs turn on, but they do not turn off again; at any given moment, they will all be on.</li> <li>• Second, the robot does not react properly to the ball. This is because the second test has set the distance to 100% by default. Use the simulator to find out at what value you want the robot to become scared. Insert this value in to your program and save it.</li> </ul>
5		<p>Now you need to turn off the LEDs. Insert a LED Driver icon in the loop, so that every time the loop starts, the leds turn off.</p> <p>Now, test your program again. If it works, we are ready to make our program do some real work.</p>

## Using a Dummy Routine

We have three subroutines, but they do not work yet. In order to test them, we have to make a subroutine to show that a given routine is operative without actually implementing its functionality. This type of subroutine is called a Dummy Subroutine. This helps us set up and test the control structure of a program.

If we want to start *WanderBehavior*, the robot should start driving around at random. However, as we have not explained how to program a robot to exhibit random behavior, we will implement *WanderBehavior* as a Dummy subroutine that will show us what is happening by lighting a LED or sounding a beep (as we have seen before). Let's try to make the Dummy routine a little more interesting.

We are going to sound a short beep and turn on the yellow light and then turn it off again. Switching something on and off again is called a Toggle (like a toggle switch). An example of this dummy routine is shown in FlowCode 3.



FlowCode 3: Dummy Setup for WanderBehavior

As we are introducing new concepts in this code, we will first give you a step-by-step explanation.

8C.1 WanderBehavior Dummy Routine																
1		Call your program 8C.1. Start by making a local variable called <i>State</i> for the internal state of WanderBehavior. This variable will not be used as a parameter, so we set the Parameter property to False.														
2		As soon as WanderBehavior is called, the content of the State variable will be compared to the value zero via an IfThenBlock. If the value of State is zero, we know that this is the first time we have reached this point. A C note is sounded and the yellow light turns on. The comparison operator consists of two “equal” signs.														
3		We must now make sure that when we reach this point the next time, something different happens. So, we change the contents of the State variable. We do this by using a FunctionBlock from Variables.														
4	<table border="1"> <thead> <tr> <th colspan="2">Properties</th> </tr> </thead> <tbody> <tr> <td>Dest</td> <td>local_State</td> </tr> <tr> <td>Function</td> <td>LET(val)</td> </tr> <tr> <td>Param1</td> <td>1</td> </tr> <tr> <td>Param2</td> <td></td> </tr> <tr> <td>Type</td> <td>FunctionBlock</td> </tr> <tr> <td>Comment</td> <td></td> </tr> </tbody> </table>	Properties		Dest	local_State	Function	LET(val)	Param1	1	Param2		Type	FunctionBlock	Comment		<p>A FunctionBlock allows you to specify a number of elements.</p> <p>Insert the name of the variable that must be changed in Dest (Destination).</p> <p>In Function, use LET, which has one parameter: a value (val).</p> <p>Insert the value for State (in this case 1) in Param1.</p>
Properties																
Dest	local_State															
Function	LET(val)															
Param1	1															
Param2																
Type	FunctionBlock															
Comment																
5		The next time that the subroutine is run, we again check if State is zero. Now that this is no longer the case, the lower branch will be activated. We will hear an F note and the light will be turned off. Do this again with a LET function and set the value to zero this time.														
6		This program makes the lamp turn on and off and alternates a C and an F note. However, as this may happen to quickly, you could delay the function by adding a stopwatch.														



## 8C.1 Assignment: Testing the WanderBehavior Dummy Routine

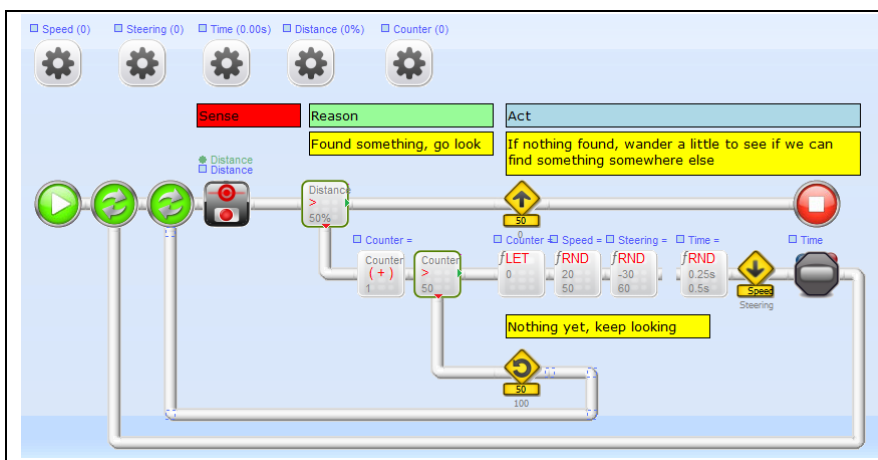
- Call your program 8C.1 - Adaptive Behavior.
- Test the program with the simulator. Make sure you execute the *Main* part.
- If you move the ball beyond the range of the sensor, the robot will enter the Search state and call *WanderBehavior*. This will turn the led on and off as long as the ball remains outside its range. If you bring the ball closer, this behavior will end.
- If this works OK, move on to the next step in which *WanderBehavior* really lets the robot wander about.
- Remember that if you are developing a complicated program, you can include a Dummy routine to test parts of a program that are not complete. The Dummy routine that you used here can be used in the following assignments as well.

### Driving Around at Random

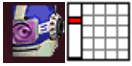
Now, we will begin adding parts step by step. We have a fair amount of code to change, so we will do it in small steps and test it after every change. If we do not do this gradually, it will become very difficult to find out where the mistakes are at a later stage.

We will start with *WanderBehavior* and then add other elements in the following chapters.

So far, we have used commands to keep a fixed speed for a fixed amount of time, but in order to drive around at random we have to continually change the speed, the duration and the direction of the robot's movement. We do this by using a *Random Number Generator (RNG)*. A random number is an arbitrary number. Every time the generator is called, it will return a different number. FlowCode 4 shows the final result of the steps listed below.

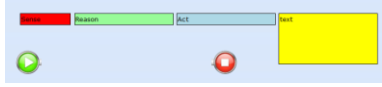
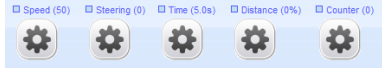

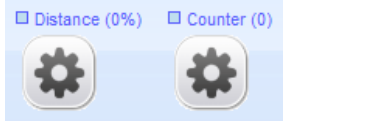
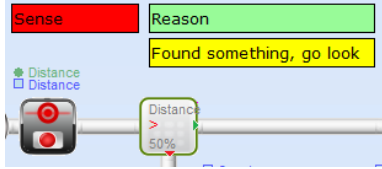
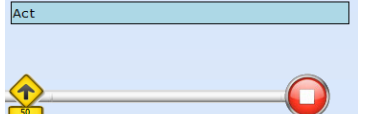
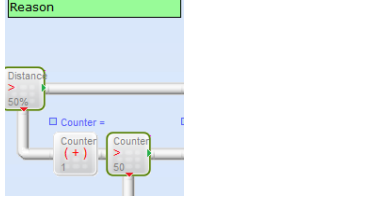
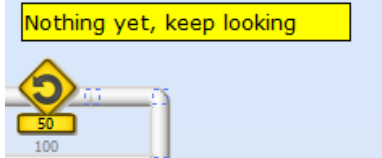


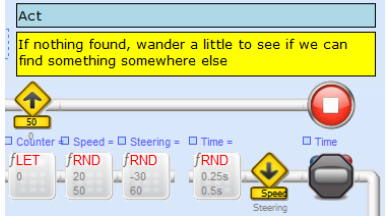
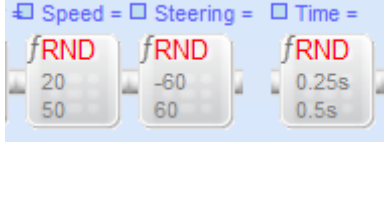
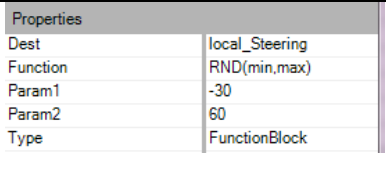

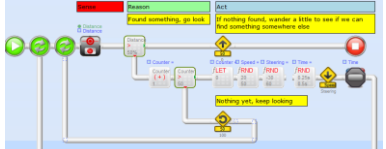
FlowCode 4: Setup for WanderBehavior



### 8D.1 Assignment: Driving Around; Random Numbers

- If you skipped step 8C start with 8B.2. Call your program 8D.1 - WanderBehavior.
- Follow the steps listed below to complete the program.
- Test the program in the simulator and watch the robot drive around. Change the random numbers so that it does not make completely “wild” movements in all directions and the behavior looks a little more natural.
- This completes the first part of Adaptive Behavior. In the next chapter, we will let the robot become scared or curious by using its sensors.

8D.1 Moving Around; Random Numbers		
1		Call your program 8D.1
2		First, we need a number of local variables, none of which will be used as parameters. These variables are used to calculate random values for speed, steering and duration. Moreover, we will also need temporary fields.
3		These three variables are given a default value of 0 (as shown on the left). The Time variable is of the Time type, while the other two are Integers (Int).
4		Next, we have to make two variables to use as temporary fields. The first one is Distance (Level type). The second is a Counter to keep track of how long WanderBehavior has been active (Int type).
5		First, we check to see if something interesting is happening around the robot. We read the distance sensor and store the value in the Distance variable. The content of this variable is used to check if an object is near. You can change the reference distance value and make it larger or smaller.
6		If something is detected, the robot will head toward it and the program will return to Main.
7		If nothing is found, the robot will start moving around, looking for an object. To do this, we need a counter that will execute the search behavior a maximum of 50 times. This is a practical way of setting a time limit for a behavior that consists of a number of steps. You could of course also use a timer loop, but we want to show you various alternatives too.
8		As long as the counter value is less than 50 and the robot has not found anything, it will continue to look around. We do this by making the robot turn around on its axis and return the program to its beginning point to check if an object has been found. If nothing is found, this action will be repeated fifty times. By changing the reference comparison value of the counter, you can make the time longer or shorter.

8		<p>If the robot has not found anything, it must move to another point on the field to continue its search. In order to make the robot move around, we need to generate random numbers (just like in the row of instructions that generate values for Speed, Steering and Time).</p> <p>The counter starts at zero and the robot moves around to different points on the field and looks around itself 50 times.</p>
9		<p>To generate an arbitrary (random) number, use the function RND (min, max). The two fields (min and max) in brackets are the parameters that indicate the range of values in which the random number must fall. So, if we want a speed between 20 and 50, Steering between 60 left and 60 right and a time of 0.25 to 0.50 seconds, we need to insert these values in the relative parameters.</p>
10		<p>Use the FunctionBlock icon and select the function RND (min, max). Insert the minimum and maximum desired values in Param1 and Param2.</p>
11		<p>These values are then used in the two following driver icons. The values for Speed, Steering and Time are stored in these two icons. This will make the robot start moving for a short while and then return to the beginning of its routine.</p>
12		<p>WanderBehavior continues to look at different places on the field, until it has found an object. Connect all the components in the program with wires and test it.</p>

## Staying within the Field Boundaries

When robot looks for an object, if it does not find one, it will start moving around at random. A good randomizer will always make sure that there is a 'normal' distribution of generated numbers. This also allows the robot to turn to the left more or less as much as it turns to the right. The practical result, however, is that the robot will probably drive off the field.

To prevent this from happening, you can program the robot to make larger left turns and smaller right turns. This will increase its chance of staying on the field. Change the randomizer to allow this; see what happens. In the next chapter, we will show you a better way to keep your robot on the field by using its field sensors.

## 8.6 In Practice

Most animals exhibit adaptive behavior: not only do they react to their environment, but they also learn from their mistakes. Robots cannot do this yet. By using the techniques described in this chapter, robots can react to their environment in a more sophisticated manner. Nonetheless, this is still rather basic compared to what even the simplest animals can do. Until not very long ago, industrial robots could not adapt to situations at all. Recently, however, a lot of work has been accomplished to create



robot arms that use a camera or touch sensors to determine if something goes wrong.



Fig 3: Industrial Robot Arm

Learning behavior is still beyond the reach of these applications, but research continues to explore this possibility. We have robotic arms that can remember the movements they need to make (as long as someone first shows them what needs to be done). However, this is not considered a learning behavior. If we want robots to perform domestic duties, they will not only have to exhibit adaptive behavior, but also be able to learn.

In laboratories all around the world, experimental robots are being developed that will soon be able to learn. In fact, the first prototypes are currently making their first appearance.

## 8.7 Test

1. What is the use of a state diagram?
2. How clever is a robot? What kind of things will it not notice?
3. What is the importance of an architecture?
4. What is a randomizer?
5. A randomizer generates the following sequence of numbers: 1, 3, 5, 7, 1, 1, 1. Is this an example of a good randomizer? Explain why.

# 9. Advanced Sensors

Sensors are fundamental to robots. They observe not only what happens around the robot, but also what happens inside it. The robot can do more with its sensors than just read sensor values. We will explain more about this in this chapter. We will see how to find out if something is coming towards the robot and how fast that object is moving. We will also see how to use sensors to detect obstacles and avoid them.

## 9.1 You will learn









- to use sensors to determine the direction and speed of an object
- to use the distance sensor in a different way
- something about sensor input ports

## 9.2 You will need

- a NXT robot
- a PC with *RoboPAL*
- a Grid field

## 9.3 You will experiment with

- making the robot react to movements
- detecting and avoiding an obstacle

Type	#	Assignment	Description
	<b>9A</b>	<b>Movement Detection</b>	<b>Reacting to movements</b>
	9A.1	Movement Detection	Detecting movements
	<b>9B</b>	<b>Movement Detection</b>	<b>Using the debugger</b>
	9B.1	Movement Detections	Testing with the debugger
	<b>9C</b>	<b>Flee and Curious</b>	<b>Becoming scared or curious</b>
	9C.1	CuriousBehavior	Acting curious
	9C.2	FleeBehavior	Flee behavior
	9C.3	Testing	Becoming scared
	<b>9D</b>	<b>Movement Detection</b>	<b>Scaring the robot</b>
	9D.1	Testing	Scaring the robot
	<b>9E</b>	<b>Test</b>	<b>Written test on chapters 8 + 9</b>

## 9.4 After completing this chapter, you will be able

- to explain the difference between intrinsic and extrinsic sensors and their functions
- to use of various sensor properties
- to make a robot modify its own behavior

## 9.5 Explanation

### Reacting to Fast and Slow Movements

How is a robot capable of determining whether an object detected by its sensors is moving or standing still? To do this, a robot needs to take at least two distance measurements in succession and then compare the values. If there is a large difference between the readings, then the object is moving quickly (or the robot is). If the difference is small then the object (or the robot) is moving slowly. As soon as the robot notices something, it should stop wandering about and determine what action needs to be taken.

### Detecting Movement

The sensor value is stored in a variable called *Previous*. In order to determine the distance that an object has moved, a second measurement must be taken (within a given time period) and stored as the variable *Distance*. These two values are then compared.

If you use a ball as an object, it will often seem to suddenly appear “out of nowhere.” This will result in a large difference with the previous reading, which makes the robot behave as if something were moving towards it very quickly.

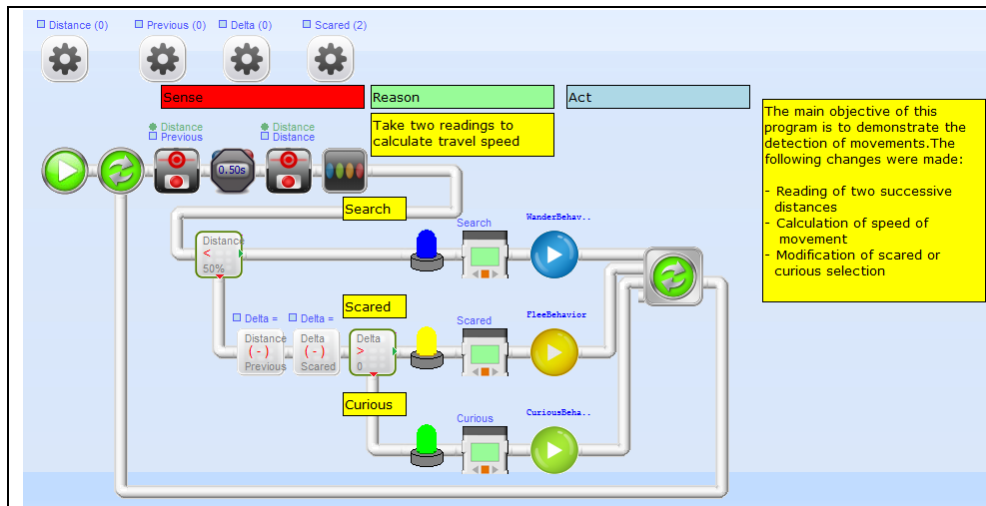
### Using Threshold Values

We will start by measuring movement, but what is actually moving: the robot, the ball or both? We obtain the speed of the moving object by subtracting the robot’s own speed from that of the object. In order to do this, we need to use the Threshold variable in our program. The robot should not react to very low sensor values (very slow movements). So, we use a **threshold value** in our code. This helps to determine if a sensor value is relevant. We can also use a variable called *Scared* to store the value that makes a robot frightened. The larger you make this value, the harder it will be for the robot to become scared.



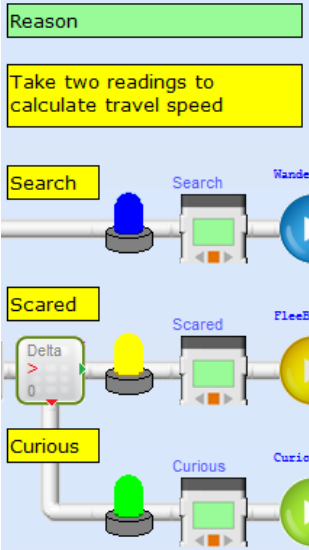
#### 9A.1 Assignment: Detecting Movements

- Load program 8D.1 and call it 9A.1 - Movement Detection.
- We will develop some code to detect movement.
- In particular, we will check if the movement is slow or fast and if it is towards the robot or away from it. We will integrate this routine into the *Main* program and allow the robot react to this information.
- The code looks like that in FlowCode 5. You will find the explanation below.



FlowCode 5: the Modified Main Routine

9A.1 Detecting Movements		
1		<p>We will need the following local variables:</p> <p><i>Distance</i> – to measure the distance to an object;</p> <p><i>Previous</i> – to store the previous distance measurement;</p> <p><i>Delta</i> – to calculate the difference between the two measurements. This value determines the direction of the movement. A negative value is away from the robot; a positive value is toward the robot.</p> <p><i>Scared</i> – is the threshold value that determines when the robot should become scared. This value has two functions: it compensates for the robot's own speed and determines how easily the robot becomes scared.</p>
2		<p>We must make two distance measurements to calculate the speed. The first one is stored in Previous, the second in Distance. Set a fixed 0.5 seconds time-lapse between the two measurements. You can, however, make this time shorter or longer. The longer the time, the larger the distance that can be travelled, but nothing else can be done while the program is waiting and this could become a problem if the period is too long.</p>
3		<p>First, check if an object is in sight, as we did in the previous version. If nothing is happening, we just leave the subroutine.</p> <p>In the previous version, we then checked if the distance to an object was short, which made the robot scared. We now want the robot to get scared when something drives towards it and not when something is only nearby. So, the movement of the object is important.</p>
4		<p>We first calculate the difference. Then we subtract the threshold value. This difference, which is positive if something is moving towards the robot, is used to determine if the robot calls FleeBehavior.</p>

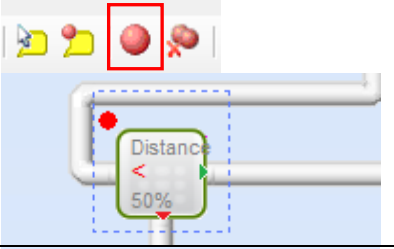
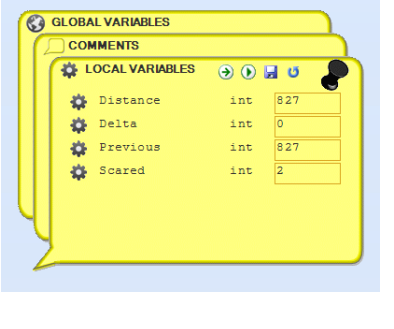
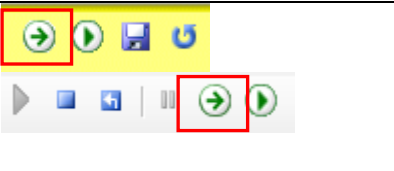
5		<p>If nothing is moving or if it is moving slower than the threshold value, the robot becomes curious and CuriousBehavior is called. Complete the program and test it.</p> <p>It is hard to frighten the robot. You have to move the ball towards the robot and then check the LCD screen to see if the robot has become scared. This is more difficult than it seems. One way of testing it is to use the debugger and go through the program step-by-step, manually changing the values that are read by the sensors and forcing the robot to take the desired course of action.</p> <p>If your program works fine, you can move on to the next part.</p>
---	---	---

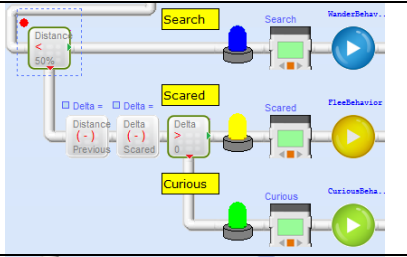
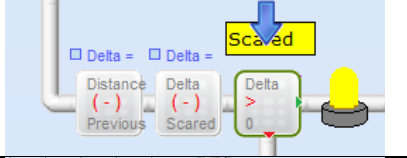
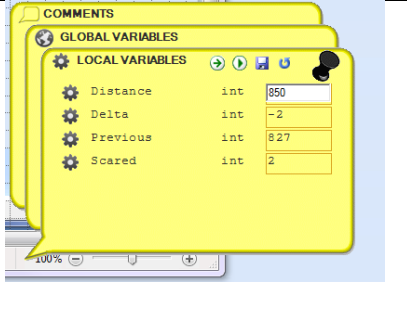

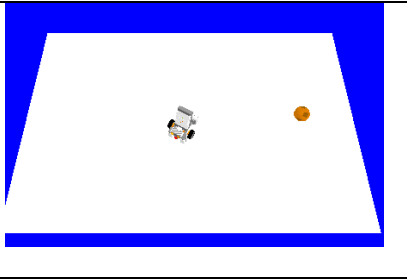
It is now very important to make sure that the program works correctly. During the following steps, many things happen at the same time, so testing will become even more difficult. Start using the debugger. You may skip the next assignment, but it is very useful to learn how to use the debugger.



### 9B.1 Assignment: Testing with the Debugger

- Start your program on the simulator
- Make sure that everything is ready so you can follow what is happening with your program step-by-step.

9B.1 Testing with the Debugger		
1		<p>Start program 9A.1. Go to the Main subroutine and select the icon where the program checks if an object is nearby. Select the red ball from the Debugger icons. A red dot will appear in the upper left-hand corner of the selected icon. This indicates that a breakpoint has been inserted. Start the simulator. Press the start button on the NXT of the simulator.</p>
2		<p>As soon as the program reaches the breakpoint, it will stop and the pop-up break window will appear, on the right, above the icon with the breakpoint.</p> <p>Place your mouse pointer on the pin and drag the pop-up window to a position on your screen where it does not block your view of anything else. Select the Local Variables and observe their measured values.</p>
3		<p>Press the Step button in the debug window or in the RoboPAL menu (upper left). You will see the blue arrow (the highlight) move to the next icon. Press the Step button again and you will see what the program does. As the Delta has now become negative, the robot will</p>

		<p>assume a Curious state. If you press the Run button next to the Step button, the program will continue until it reaches the next breakpoint.</p>
4		<p>Start the program with the Run button in the Debugger window until it stops at the next breakpoint.</p>
5		<p>Increase the value in the Distance variable to something like 850. Step through the program. You will see how the calculations are executed and how the program assumes the Scared state and then also calls the FleeBehavior subroutine. As both FleeBehavior and CuriousBehavior have not been created yet, nothing will really happen.</p>
6		<p>Select the icon with the breakpoint and press the red ball in the icon bar again. This removes the breakpoint and when you select Run, the program continues without stopping.</p>
7		<p>If during its explorations the robot (or the ball) moves outside of the field, you will not be able to see it or pick it up. The simulator will put the robot or the ball back in the middle of the field after a short time.</p> <p>If you put the ball in front of the robot, the robot will return to its initial state and become Curious, but not do anything else (yet).</p>

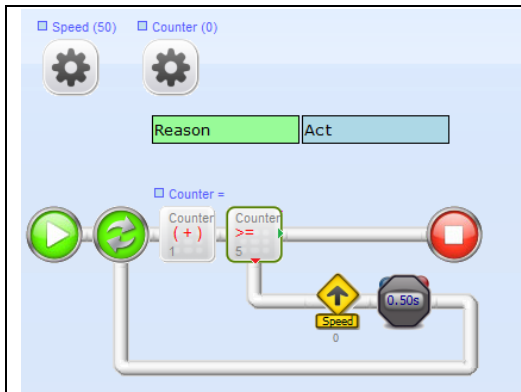
The idea is that we can test and observe the steps of our program as we develop it. Make sure that you test every branch of your program at least once. You must be sure that its logic is correct. In order to this, change the position of the robot and the ball on the simulator and make sure you have created the correct situation. Practice until you understand how to use the debugger.

## Using the State

We are currently using the state set by the Reason step and observing whether the robot becomes scared or curious. Although the correct routine is called, neither is functional yet. We have already tested *WanderBehavior*. Now, we can also start developing and testing *CuriousBehavior* and *FleeBehavior*.

## Making the Robot React - Curious

We have already seen Curious behaviour in Part One. It is rather simple. As the sensor values are read by the Main loop, we will have no Sense step in this subroutine. The time spent moving towards an object is controlled by a counter, just as in WanderBehavior. Although we could do without it in this routine, we will include it in all three behaviour subroutines. The code is shown in FlowCode 6.



FlowCode 6: Curious Behavior



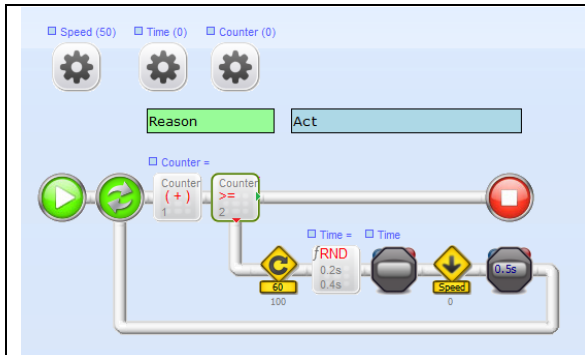
### 9C.1 Assignment: Acting Curious

- Name your program 9C.1 - Curious Behavior
- Open the subroutine *CuriousBehavior*.
- Develop the program using the instructions provided below.
- Test the program in the simulator. First, check what the robot is doing. Then, use the debugger and observe how the robot reacts, step by step.

9C.1 Acting Curious		
1		We need the following local variables: <i>Speed</i> – the speed at which the robot starts moving; <i>Counter</i> – how long the robot has been moving;
2		Increase the counter and check if it has reached the maximum value. Although we could do this without a counter, it helps to maintain a uniformity of behavior and also serves as a preparation for the changes that we will make in the next chapter.
3		If the counter has not reached the maximum value yet, the robot will drive towards the object for a fixed amount of time.
4		Start the simulator and observe how the robot reacts. You may also test the program on the NXT with a ball or your hand as an object.

## Making the Robot React - Scared

The robot can determine its states, but it also needs to act accordingly. It must move away from an object until it reaches a safe distance. So, it will make a slight turn away from the object and then drive backward. It may repeat this action one or more times. FlowCode 7 shows how this is done.



FlowCode 7: Flee Behavior



### 9C.2 Assignment: Getting Scared

- Name your program 9C.2 - Flee and Curious Behavior.
- Develop your program like the FlowCode above. The explanation is provided below.
- Use Breakpoints and the Step facility to check exactly what your program is doing.

9C.2 Flee Behavior	
1	<p>We will need the following local variables:  <i>Speed</i> – the speed at which the robot moves away from an object;  <i>Time</i> – how long the robot moves away from an object (using a randomizer);  <i>Counter</i> – how long the robot stays scared.</p>
2	<p>There is no Sense step in this subroutine as the Main routine determines whether the robot should become scared.</p> <p>We need to determine how long the frightened reaction will last. This is done with a counter that is set to 2 by default, but can be changed.</p>
3	<p>Every time the robot executes a scared reaction, it will make a slight turn to change direction and move away. Subsequently, a randomizer determines for how long the robot moves away and at what speed to drive backward.</p>
4	<p>Complete the program and test it in the simulator. To scare the robot you will have to move the ball towards it. Alternatively, you may use the debugger to change the sensor values manually.</p>



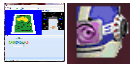


### 9C.3 Assignment: Becoming Scared

- Name your program 9C.3 - Flee and Curious Behavior.
- Upload your program to the NXT and test it on the robot.
- You will probably have to make some changes to your program as the NXT will react differently from the simulator.
- Scaring your robot is easy: just hold your hand or a ball right in front of it.

### Scaring the Robot

During testing, you may notice that it is not easy to scare the robot, but this will happen only if it drives quite near to the ball. The main reason for this is that there are no objects coming towards the robot. You can drag the ball towards the robot, but it is not easy to do this at the right time. What you need is an object that is actually moving towards the robot.



### 9D.1 Assignment: Scaring the Robot

- Save your program as 9D.1 - Flee and Curious Behavior.
- We already have an object that can move towards the robot: another robot. If we create a second robot and let it move towards the first robot, we will have our moving object.
- Remove the ball from the playing field, select a second robot (RobotNXT - Rescue) and tell the robot (in the properties of the second robot) that it should start with the Main program, too.
- Use the Breakpoint and the Step facilities of the simulator to check what the program is doing.
- You now have two moving objects and you can make the robots much more sensitive and let them react better to one another.
- Take into account that using two robots in the simulator will use up much more processor time and consequently slow it down. This may also make it behave erratically. This is especially true for slower computers with a minimum of main memory.
- You now also have two control panels. As soon as you start one of the two robots, the other will start automatically.

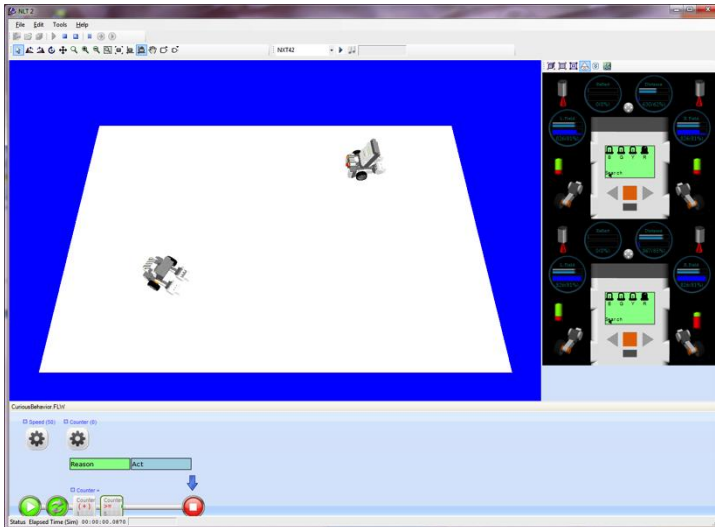


Fig 4: Simulation with two Robots

## 9.6 In Practice

Naturally, we have no real use for robots that get scared or curious, but the ability to detect objects in a given area or objects that are moving towards a robot is very useful. Various car manufacturers are working on a new generations of automobiles that will be able detect pedestrians in front of the car or about to cross a street.

The intelligent computers in cars use a camera to detect pedestrians and then quickly calculate if the car, based on its current speed, will be able to brake in time.

If this is the case, the car will take over if the driver does not react in time. If the braking distance is too short and the pedestrian risks being run over by the car, the bonnet will tilt at an upward angle, so the pedestrian will slide off the car if he should be hit by the car.



Fig 5: Automatic Container Transporter, Rotterdam Harbour

Another example is an autonomous vehicle such as the unmanned trains that are used in harbours. It is important that these vehicles are capable of detecting obstacles or living creatures in front of them. If they do, they emit a warning and brake to avoid a collision.

These vehicles are often guided by wires embedded in the road surface and use their sensors to detect pedestrians and other obstacles.

## 9.7 Test

1. How reliable are the sensors on your robot?
2. What factors may disturb sensors?
3. How can you eliminate these disturbing factors?
4. What does a robot need to measure the speed and direction of a moving object?
5. Provide an example of a useful intrinsic (internal) sensor for a robot



### **Exam 9E: Written Test on Chapters 8 & 9**

These chapters are concluded by a written examination.

# 10. Control Systems

We already have seen several ways in which to control a robot. The Sense-Reason-Act loop is the most important one, but there are several other algorithms that can be used, too. The most important principle is feedback. A feedback loop ensures that a desired situation is maintained through the use of a so-called *controller* and guarantees a stable situation. The controller is one of the most important control systems and can be found both in industrial and domestic systems.

## 10.1 You will learn







- several aspects of control algorithms and real-time control
- the basic principles of feedback loops in control systems

## 10.2 You will need

- a NXT robot and a ball
- a computer with RoboPAL
- a Grid field

## 10.3 You will experiment with

- making the robot avoid obstacles
- developing a program to follow a ball

Type	#	Assignment	Description
	<b>10A</b>	<b>Adaptive Behavior</b>	<b>Avoiding Obstacles</b>
	10A.1	StayInField	Staying within the field
	10A.2	StayInField	Calling StayInField
	<b>10B</b>	<b>Adaptive Behavior</b>	<b>Proportional controller</b>
	10B.1	CuriousBehavior	Making Curious behavior proportional
	10B.2	Obstacle	Varying the speed
	10B.3	Testing	Testing with an obstacle
	<b>10C</b>	<b>Object Tracking</b>	<b>Following and avoiding objects</b>
	10C.1	Curious Behavior	Follow and avoid an object

## 10.4 After completing this chapter, you will be able

- to explain the principle of a feedback loop with your own example
- to indicate what happens when a robot controller only works with 'on' and 'off' and the motors run at full speed

## 10.5 Explanation

This chapter deals with control technology and, in particular, control algorithms. An important part of this - *real-time control* - has already been discussed. The timing aspect is essential here. The Line-Follower is an example of real-time control. If the robot does not react in time, it may overshoot the line and completely lose the track.

We are going to look at the most important control principles and work out a simple example by tackling a control problem. You will have to make sure that the robot no longer drives off the field. We will do this by continually making sure the robot detects its blue surroundings.

The second adaptation concerns the detection of obstacles. We will change *CuriousBehavior* so that the speed of the robot is related to its distance from an object. The closer it gets, the slower it will move. This will ensure that the robot no longer bumps into the ball (or any other object) and consequently the ball will stay closer to the robot.

### Feedback

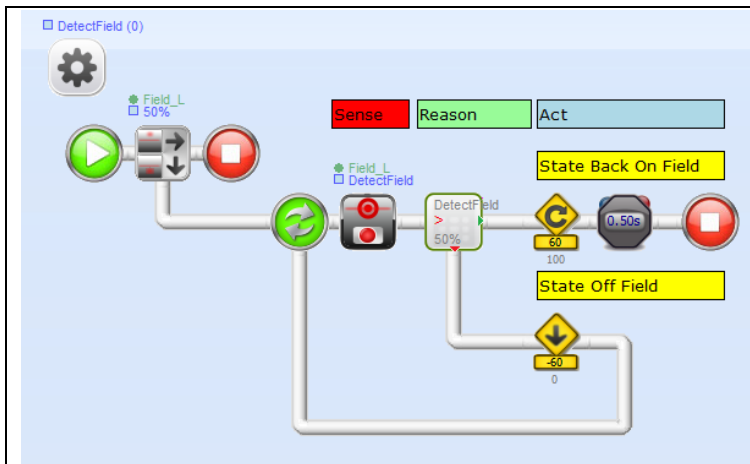
*Feedback* is one of the most important principles in control theory. If you want a robot to reach a given point, you have to control its behavior and make sure that it continually checks that it is on the right track.

With the line-follower, we used the sensors to keep the robot on track. This is an example of a feedback mechanism. The sensor detects the line. If the robot moves away from the line, the sensors no longer detects it and turn the robot back towards the line.

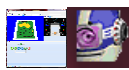
### Staying on the Field

We have already seen that the robot has a tendency to drive off the field. After a while, the simulator puts it back at the centre of the field. By making the randomizer use more right than left turns, we can increase the chance that the robot will stay within the field. However, there is a better way of doing this. We can use the field sensors to check if it is about to drive off the field.

In this optional assignment, you will develop a new subroutine to detect if the robot has left the field and make it return to it. In order to accomplish this, one of the field sensors needs to check for the colour blue.



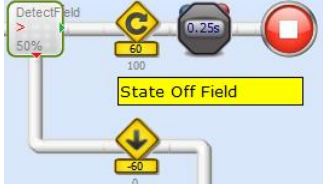
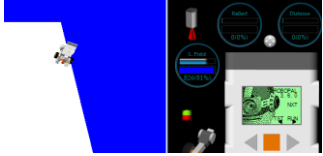
FlowCode 8: StayInField Subroutine

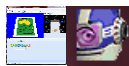


### 10A.1 Assignment: Stay on the Field

- Change the program to make sure that the robot does not drive off the field. Call your program 10A.1 - Adaptive Behavior.
- If the robot drives off the field, its field sensors will detect a different colour. The idea is that the robot should then take action to return to the field.
- Develop the subroutine as explained below and try it out by having the robot directly call the subroutine. You must move the robot manually to make it turn back to the field. In order for the program to work correctly you first have to place the robot outside the field manually.

10A.1 Staying on the Field		
1		Make a local variable with the name DetectField that will not be used as a parameter.
2		First, we need to check that the robot is still on the field. We do this by using the left field sensor. As the surface is white, we need to look for a high value. If this is correct, we're doing OK.
3		If the robot has left the field, a loop starts to check that we return to it. As long as it does not detect the field, the robot needs to do something to return to it.
4		If the field is detected again, the robot must turn for about a quarter of a second (in the same loop) and then return to the original subroutine.

5		<p>If the field is not detected, the robot must move backward and restart the loop to check once again if it is back on the field.</p>
6		<p>Start the simulator and indicate (in the properties) that the robot needs to execute StayInField directly. Drag the robot out of the field and press the RUN button in the control panel of the NXT. Make sure that you press the RUN button every time, because the program stops as soon as the robot is back on the field.</p>



### 10A.2 Assignment: Calling StayInField

- Name your program 10A.2
- Make sure that StayInField is called from your program. Which is the best place to do this?
- Find out which subroutine is the most logical to call StayInField.
- Hint: it does not make sense to call StayInField from a subroutine that will not cause the robot to drive off the field.
- Insert a call to StayInField in the selected subroutine.
- Insert (in its properties) that the robot must start with the Main program.
- Test your program on the simulator.

## Proportional Behaviour

*Feedback* is an important principle, but if it is organized with an all-or-nothing approach, the reaction may be too large or too small. This type of feedback loop is called *discrete* feedback. It is best to regulate a deviation based on its magnitude. We call this *proportional* feedback.

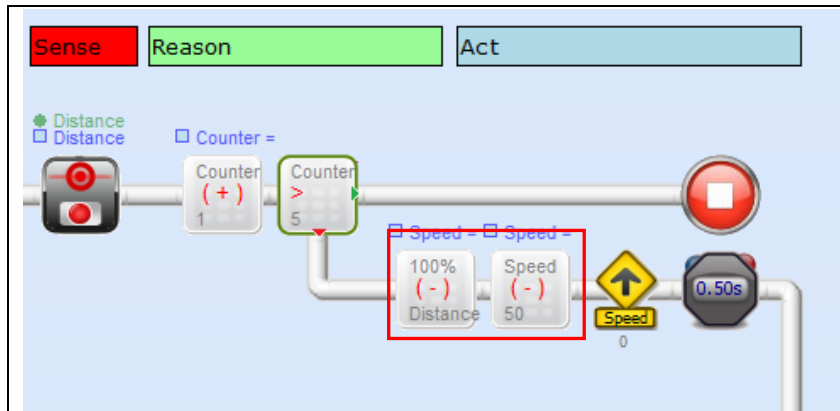
This allows us to relate a scaled reaction to the speed at which an object is coming towards the robot. If the object is moving fast, the robot will move backward quickly; if the object is moving slowly, the reaction can also be slow. The principle of proportional feedback is found in many control systems and controllers.

Proportional feedback control systems are more complicated and therefore more expensive than discrete controllers.

The thermostat of the central heating in your home, for example, is a discrete controller. If it is too cold, the heater turns on. If it is warm enough, the heater turns off. However, the heater does not know how large the difference between the actual temperature and the desired temperature actually is. So, when the thermostat has reached the desired temperature, the heat in the radiators continues to heat the room and the temperature will rise beyond that set by the thermostat.

This phenomenon is called *overshoot*. Thermostats that make a heater work proportionally to the difference in temperature do not overshoot. Most domestic heating systems, however, do not have such a feature. So there is no sense in setting the thermostat extra high when it is cold, it will not heat any more, just longer.

We will now modify our program to make the reaction to an approaching object proportional.



FlowCode 9: Making Curious Behavior Proportional




### 10B.1 Assignment: Make CuriousBehavior Proportional

- Name your program 10B.1
- In this assignment, we will make the robot react proportionally.
- If the robot becomes curious, it will drive towards the ball. If it were to drive toward the ball at a continuous speed, it would bump into it and cause it to roll away.
- We want the robot to drive slower as it gets closer to the ball.
- The necessary modifications are explained below.

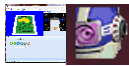
10B.1 Make CuriousBehavior Proportional		
1		In order to react proportionally, the robot needs to know the distance to the ball. This value is not used in CuriousBehavior, so we must read the sensor value. We need to add a Sense step to read this distance. Do not forget to include a Distance variable to store the value.
2		Include two icons to make a calculation in the routine in which the robot drives towards the ball. The first one stores the distance that was measured in the Sense step. If you subtract this value from 100% you get the distance that must be covered, which continues getting smaller as the robot moves forward.
3		Continue this calculation by dividing the result by 4. This will give you a number that is the speed at which the robot needs to move.



4		<p>As the speed is corrected in each cycle of the loop counter, the robot will drive slower and slower. Change your program, save it and test it on the simulator. Watch how the robot is much more “careful” with the ball.</p>
---	---	--



FlowCode: 10 FleeBehavior Subroutine



### 10B.2 Assignment: Varying the Speed

- *FleeBehavior* moves the robot backward a little and changes the Steering at random.
- In order to obtain a proportional reaction, you need to change the Speed by assigning the value of Delta to Speed (instead of the fixed value that is currently being used).
- Notice that this time you are passing a parameter to *FleeBehavior*.
- Make all necessary changes and then test this version on the simulator.



### 10B.3 Assignment: Testing with an Obstacle

- Apply all changes and test the program.
- You will have to tweak the used values to make the reactions more natural.
- Check that your program works well with the NXT and a real obstacle.

## More Proportional Control

Now that you know what proportional control is, you might want to test it in different circumstances. Remember that you used a line follower in Part 2 and that the robot reacted to detecting the black line. This is a discrete feedback loop.

In order to detect the black line, the sensors check a range of values. But if the sensor is halfway the green field and the black line, it will see a value that is a little higher than that of the pure colour black.

By making the feedback loop proportional, you can make the robot react to the black line more accurately. What can be done here is to first calculate the

difference between the sensor reading and the value for black. The closer the value is to true black, the larger the correction should be. That way the line follower will be capable of following the line much better and you may achieve higher speeds.

## Follow and Avoid

You will complete this final assignment on your own. Use a ball as the object and change *CuriousBehavior* so that the robot moves towards the ball and touches it lightly. The ball will then roll away. This will make the robot either scared or curious. You will also need to develop a new behaviour - *AngryBehavior* - to make the robot move faster and faster. You are using two robots. The result of your changes will be that the robots seem to play with the ball, but are afraid of fast movements.



### 10C.1 Assignment: Follow and Avoid an Object

- Make a new program called 10C.1
- Add a second robot and give it a different behavior.
- The new robot gets angry when it sees the ball and drives faster and faster as it gets closer to the ball.
- As soon as an object moves towards the first robot, it must get scared and react proportionally to the speed at which the object is moving.
- When the object has moved away far enough, both robots start to search for it again.
- As the ball is usually not moving towards the robot, the first robot will get scared by the other robot and not by the ball and will preferably chase the ball, but move away from the other robot. Change your program so this happens.
- The idea is for you find out how to make a program exhibit interesting behavior in which the robot plays with the ball.
- This is your final assignment. Your grade will depend on how well you develop your program.
- The more interesting the behavior is, the higher your grade will be. Also, show your teacher how you changed the program.
- Have fun making and testing your program.
- You may also select to make a proportional line follower; provided your teacher lets you do that and you have a rescue field at hand.

## 10.6 In Practice

Feedback is a very important and frequently used control structure. We find examples of it in almost all devices. We have already seen how a thermostat works and determined that it is a discrete controller. On the other hand, the temperature control of an automatic shower faucet has a mechanical feedback system that makes sure the water will be mixed to the desired temperature.

Servomotors have a proportional controller that ensures that the desired position is reached and maintained. One example of a servomotor is the tail rotor of a helicopter. It makes sure the helicopter does not start spinning around on its axis. A gyroscope continually measures the position of the tail and modifies the tail rotor speed so that a helicopter can stay in the desired position. Most airplanes are also equipped with this kind of feedback system. Even model racecars have this kind of feature to control the wheels.

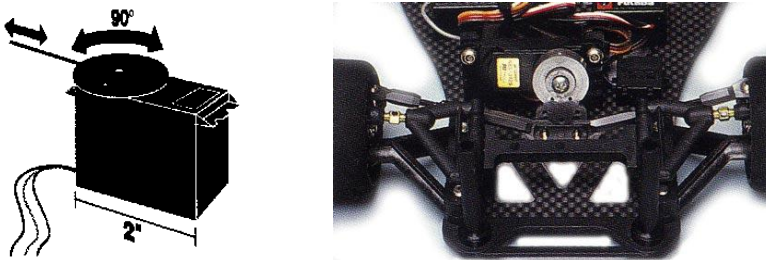


Fig 7: Servomotor and Steering of Model Car

Feedback systems do not always need to be electronic. The control valve of an old-fashioned steam engine is a good example of a mechanical construction that makes sure the pressure in a steam chamber does not get too high, but at the same time makes sure the steam does not escape when the pressure is too low.

Yet another example is the controller of your Wii or the newer Kinect sensor of the Xbox 360. They both have all kinds of feedback systems that make sure that your computer can detect the speed and direction of your movements. This is accomplished with some very sophisticated sensors, including a tiny camera and in the Wii controller an acceleration sensor. The acceleration sensor was first developed for use with airbags to instantly detect if a car is about to crash. These same sensors are now also used in WII's.

## 10.7 Test

1. Why do we use feedback systems?
2. What is the difference between a discrete and a proportional feedback system?
3. Provide an example of the use of a feedback loop in your home and explain what kind of feedback it is.
4. The final part is based on your program: the better it allows the robot to chase the ball, the higher your grade will be.