Robot Projects for RobotBASIC
Volume I: The Fundamentals

Project 6: Introduction to the Line Sensors

Previous projects have provided a lot of basic information about controlling the robot. We have even explored the idea of letting the robot overrule us if we try to make it do something potentially dangerous like colliding with a wall. In this project, we will teach the robot how to follow a line on the floor. It is worth mentioning that the principles in this introductory project are designed to be easy to understand. Future exercises will explore this subject in much more detail and make the robot far more adept at following a line.

The simulated robot (as well as the RB-9 Robot) has three line sensors mounted near its front edge. In the case of the simulated robot, the sensors can respond to various colors. The real robot works best with a line that represents a high contrast with the floor. You can make a test environment by creating a line with black electrical tape on a white poster-board background. Take care not to overly stretch the tape as you apply it to the poster-board because doing so will tend to curl the poster-board.

The status of all three line sensors can be read with the function **`rSense()`**. Future projects will utilize the status of the *individual* sensors to create a highly capable line-following robot. As mentioned earlier though, this project will keep things simple. For now, we will not care which of the three sensors sees the line – the robot will respond based on only two conditions of **`rSense()`**. The first condition is zero, meaning none of the sensors are triggered. The second is represented by any value greater than zero, which indicates that one or more of the sensors is seeing the line.

Drawing a Line

The simulator will need a line to follow. We will draw the line with **`Line`** and **`LineTo`** statements as shown in Figure 6.1. Type in the program and verify that it draws a line that looks like Figure 6.2.

```
SetColor GREEN
LineWidth 10
Line 400,400,400,350
LineTo 430,300
LineTo 460,270
LineTo 470,250
LineTo 470,200
```

```
LineTo 450,150
LineTo 450,100
```

Figure 6.1: This program will draw a line for the robot to follow.

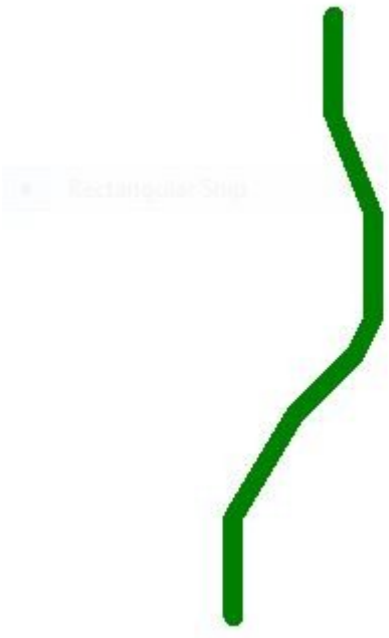

Figure 6.2: The simulated robot will follow this line.

Following the Line

The next question is how can we make the robot follow the line. Think about how you might follow a curvy wall if your eyes were closed or you were blindfolded. If the wall is on your left, you could reach out and touch it with your left hand and walk forward. Assume, for this example, that you must keep your arm straight at all times. If the wall happens to curve toward you, it will put a strain on your arm and scrape on your fingers, so ideally you will keep an arm's length from the wall – just barely touching it as you move along. One way to accomplish this is to constantly move away from the wall slightly whenever your fingers touch it. Of course, as soon as your fingers leave the wall, you will need to turn back to ensure that it is still at arm's length. If the wall turns sharply to the left, then, as you walk, you will have to turn back a little extra before you touch the wall again. If the wall curves toward you, will find yourself turning away from the wall to maintain the arm's length distance. The net effect of this plan is that you will constantly move away from, and back to the wall, allowing you to maintain a constant distance. Once you understand this principle, we can use it to make the robot follow the line.

The robot should constantly move forward, but after each movement it should check to see if it is on the line or not. If its sensors see the line the robot should turn to the right, trying to move away from the line. If the line is not seen, the robot should turn to the left, trying to find the line again. This should make the robot wiggle back and forth as it moves along the line. The program in Figure 6.3 shows how to create a program that performs these actions. Notice that the main portion of the program has been given its own title to make it easier to identify.

```
MainProgarm:
  gosub DrawLine
  gosub InitializeSimulator
  gosub FollowLine
end

DrawLine:
  SetColor GREEN
  LineWidth 10
  Line 400,400,400,350
  LineTo 430,300
  LineTo 460,270
  LineTo 470,250
  LineTo 470,200
  LineTo 450,150
  LineTo 450,100
return

InitializeSimulator:
  rLocate 400,400
  rInvisible GREEN
return

FollowLine:
  while true
    rForward 1
    if rSense()=0
      rTurn -1
    else
      rTurn 1
    endif
  wend
return
```

Figure 6.3: This program will draw a line and make the robot follow it.

If you run the program in Figure 6.3, the robot does follow the line, at least until it gets to the end of the line. Think about it. The robot has no easy way to know when the line ends. If it does not see the line, it just assumes it is right of the line and keeps turning to the left trying to find it again. After awhile, the robot will accidentally find the line at an angle that will allow it to start following again.

If you watch the robot very carefully, you will see that it does in fact wiggle as it moves along the line. If you change the parameter in the two **`rTurn`** statements to 20 and -20 instead of 1 and -1 the wiggle will be exaggerated making it easier to see. This wiggle can be a problem with the real robot, but we will examine that shortly.

As mentioned earlier, this is a very simplified way to follow a line. Because of the simplification, the robot is only capable of following lines that are relatively straight. You can see an example of this by adding the following line to the very end of the **`DrawLine`** subroutine.

```
LineTo 500,90
```

Figure 6.4 shows the new line, which now has a sharp bend at the top. The thin blue line shows the robot's path as it follows the green line. As you can see, the robot looses the line at the sharp bend. Since the line sensors do not see the line, the robot constantly turns to the left. Notice also, that the robot's movements now bring it higher in the window and eventually cause it to collide with the wall, causing an error. If you want your program to show the blue line, substitute the initialization subroutine in Figure 6.5 for the original one in Figure 6.3.

Notice that two colors are now declared invisible so that the robot does not see them as obstacles. When the pen is dropped, it always draws in the *first* color in the invisible list.

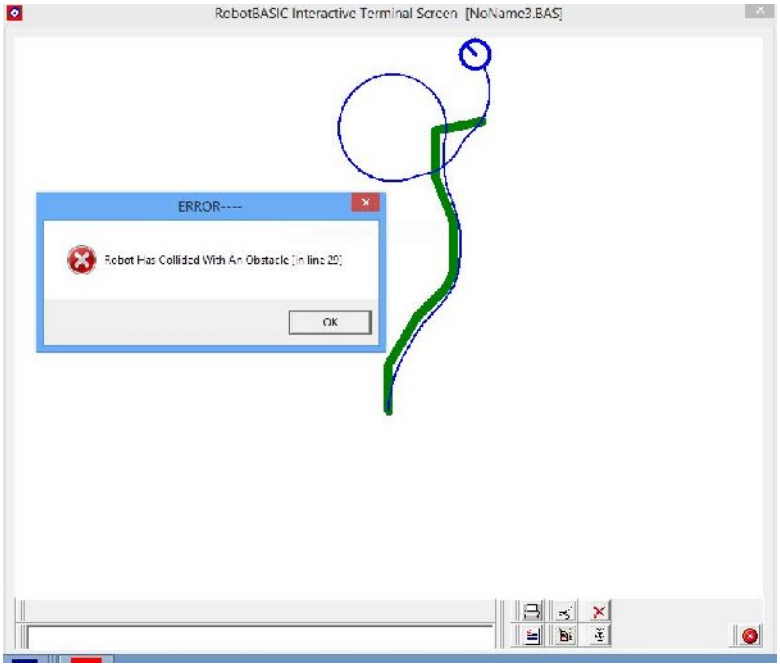

Figure 6.4: If the line is not fairly straight, the robot has trouble following it.

```
InitializeSimulator:
  rLocate 400,400
  rInvisible BLUE, GREEN
  rPen DOWN
  LineWidth 2
return
```

Figure 6.5: These modifications let the robot leave a blue trail as shown in Figure 6.4.

Suggestions for Further Study

Modify the program in Figure 6.3 so that it has the extra bend at the top. Verify that the robot will loose the line when it encounters the sharp turn. Verify also that the robot will cause the error shown in Figure 6.4. Modify the program using **`rFeel()`** so that the robot will turn away from walls or other objects it might encounter while it is following the line. Hint: Always check the **`rFeel`** sensors before moving forward. If an object is encountered, turn away some random amount. At this stage, your goal is to just avoid the error. Do not expect the robot to be intelligent. Later projects will determine which line sensors see the line and use that information to create a much better line-following robot.

Even though our line following program has its limitations, it can be improved considerably with very little effort. The first step in trying to improve the robot's ability to follow the line is to determine why it fails when it encounters a sharp turn. The reason

is not obvious, but it is not complicated once you understand. When the robot encounters the sharp turn, it tries to turn to the right because it sees the line. The line at that point, though, appears to be very wide, so even though the robot is trying to turn away from the line, it is not able to do so before it moves forward past the line. Once the robot gets on the left side of the line, it is always turning left, which now takes it away from the line instead of back to it.

The robot will work much better with this line if you make it turn 2° or even 3° instead of the 1° turns in the module **`FollowLine`**. Making the robot turn more will cause it to wiggle more, though, as we saw earlier.

It is important to realize that the improvements we are making to this program only work with this line. If the **`rTurn`** statements, for example, turn 3°, the program works better with the original line. Try reducing the line's width to 2 though, and see if the 3° turns are still effective.

Don't be discouraged by these complications. As you increase your knowledge of programming, we will be able to create far better programs that can cope with a much wider variety of problems.

Using the Real Robot

The next step is to modify the original program (Figure 6.3) so that it can control the real robot. Try to make the modifications on your own first, then check your work against the program in Figure 6.6. If you run that program, the RobotBASIC Robot should be able to follow a line. In order to test the program, you will need to create a line on white poster board with black electrical tape. At least for now, the real line should be two inches or so wide (use multiple layers of tape). It should also not have any overly sharp turns. Note**:** Figure 6.6 only shows the subroutines that had to be modified.

The real robot will have a wiggle just like the simulated robot, but the RB-9's movement will be more annoying. The reason is based on the fact that both the simulation and the real robot normally rotates around its center when it turns. This means that one wheel actually runs backwards when making a turn. The jerky wiggle is caused by a wheel moving forward, then backward, then forward, over and over again.

Fixing this problem is actually easier than you might imagine. We just have to tell the real robot to modify the way it makes turns. A better way to turn, for this situation, would be for one wheel to just stop instead of turning backwards. This allows the robot to turn, but the wheels never have to reverse directions.

RobotBASIC provides numerous **`rCommands()`** that let you modify how the real robot responds to various commands. Notice that an **`rCommand`** has been added to the main

program in Figure 6.6. It tells the real robot to utilize a special turning style where the slower wheel stops (speed 0) instead of reversing. If 60 is used, for example, instead of 0, the slower wheel will only slow to 60% of the speed of the other wheel. This makes the robot turn very slowly. This eliminates the jerkiness completely, but it can prevent the robot from following lines with a lot of sharp turns, especially using the simplified methods discussed in this introductory project.

Later projects will show you how the robot can modify its own turning style depending on the curviness of the line it is following. For now though, experiment with the **`rCommand`** and find a value that works well for your line.

```
#include “RobotBASICrobot”
gosub InitRROScommands

Real = TRUE // set to FALSE to use simulator
PortNum = 5 // set this variable to your Bluetooth Port
Number

MainProgarm:
  if Real
    gosub InitializeRealRobot
    rCommand(SetTurnStyle,0)
  else
    gosub DrawLine
    gosub InitializeSimulator
  endif
  gosub FollowLine
end
```

Figure 6.6: These modifications allow the program in Figure 6.3 to control the real robot.

Limitations

It is important to remember that this is a very basic way to follow a line. Because of that, especially with the real robot, the line must be relatively straight or the robot will lose it. Even with this limitation though, this project furthers the idea that a robot can utilize sensory information to react appropriately with its environment. As we proceed through future projects we will improve the robots ability to handle more complex situations.

Suggestions for Study

After working through the examples in this project, experiment with the real robot to determine its limitations and to see if you can improve its ability to handle situations

where the robot fails. Try lines with various curves and lines that are thinner or thicker to see how that affects the robot's performance.

Place an object on the line (draw an object for the simulator or place a real object for the RB-9 robot). Program the robot to stop and make a noise if it encounters an object while following the line. The **beep** command can be used to make the PC create a sound when using the simulation. You can make the RB-9 emit a low tone with the following command.

```
rCommand(PlaySound,LowTone)
```

This option is only one of many sounds that the RROS can produce for you. The table in Figure 6.7 list some additional choices.

Parameter	**Description**
`Blip1`	drip/blip sound
`Blip2`	another dip/blip
`InitTone`	startup RROS tone
`LowTone`	low tone
`BeepBeep`	two quick beeps
`BeepBeepBeep`	three quick beeps
`Phasor`	a phasor sound
`Siren1`	a type of siren
`Siren2`	another siren
`Siren3`	still another siren

Figure 6.7: These are the standard sounds for the RROS.