Andreas Döpkens
Brian Schüler

April 2018
[1.Rev. 2018-04-25]

An improved Wall Follower

At the latest after the implementation of the *Wall-Follower State Machine*, as we presented it in the paper "The Wall Follower" (Döpkens 2014), it becomes clear that the neat differentiation of States in terms of a redundancy-free, logical inclusion of the possible (qualitative) sensor boundaries brings significant improvements over a very simple Wall Follower, as we presented it in our paper "Get Started On acaBot" (p. 78). However, creating this improved code with finding suitable sensor values has been extremely cumbersome. Even with such elaborate attempts to realize a "quiet" robot ride both along the walls and in the curves, the result was not really satisfactory. The robot drove too strong in so-called serpentine lines.

Now, in the *Improved Wall Follower* presented here, we want to make changes that greatly or even completely reduce the use of the distance sensors in terms of traversing the inside and outside curves. Instead of using the distance sensors, the intensive use of the vOmega(v, ω)-Interface takes place. Driving straight ahead along a wall is still done using sensor values, but in addition a P-controller is implemented. The above-mentioned disadvantages are thereby eliminated, ie the bot will travel its way through the maze much more stable and quieter, and also the laborious finding of sensor values for a bot in the different states will be eliminated.

Implementation of a P-controller for straight-ahead driving along a wall

In a simple robotic travel along a wall controlled only by a distance sensor value, as we have known so far, the robot was steered away from the wall by a fixed change in the omega value when the sensor detected that the robot had come too close to the wall. The same was true in the opposite direction if the robot had moved too far away from the wall. The hysteresis game of "Bot too far away" and "Bot too close", which operated with fixed correction values, inevitably moved the robot more or less serpentine.

Now, by replacing Omega's *fixed* correction values with *dynamic* correction values that very accurately correct the robot's quantitative deviation from the

wall proportionally, the rob's driving along the wall can be significantly improved. By means of a so-called P-controller (or proportional controller), the omega is proportionally changed in one direction or the other when the robot has moved too far away from the wall or approached it, and so does omega very little change, if the deviation of the robot from the wall is only slight.

A P-controller has the following general appearance:

$$\text{error } (V_{diff}) = (\text{goal state } (V_{set}) - \text{measured state } (V_{act})) * k_p$$ [1]

In the present case of a P-control for the correction of the lateral robot distance from the wall, the equation takes the form:

$$omega_{new} = (distWall_{set} - distWall_{act}) * k_p$$

The value for $distWall_{set}$ is fixed in the program. For example, we want the robot to be 10 cm laterally from the wall while driving straight ahead. Through constant, cyclic evaluation of the sensor facing the side wall, we continuously receive the actual value for $distWall_{act}$. If the difference between the values $distWall_{set}$ and $distWall_{act}$ is zero, the robot is at the right distance from the wall, and omega does not need to be changed (omega is zero). On the other hand, if the robot is too close to the wall or away from it, omega assumes positive or negative values, which in the vOmega interface corresponds to a left or right cornering, meaning: The course of the robot will be corrected.

But hold on! We're not quite done here. In the above requirement, the bot should travel along the wall a certain predetermined distance, e.g. at a distance of 10 cm. "At a distance of" means that the robot should always be parallel to the wall while driving along it. How does the robot know if it keeps exactly the given distance to the wall on its journey?

[1] In control engineering V_{diff} is called **control deviation** (gr. Regelabweichung) or control error (error, gr. Regelfehler). The factor **kp** is called the **proportional gain or force factor**. This factor has to be determined empirically, ie you have to guess a bit, and for that you need experience.

So, again, we want the robot to travel at a certain predetermined distance to the wall (distWallSet). The only sensor we got to tell us what's going on to the right of the bot is sensor-R. Indeed, sensor-R shows us a distance, but it is not the predetermined distance we need. Yet, in order to sensorically tell, if the robot is at the correct distance to the wall, we can deploy sensor-R. But we have to calculate a bit beforehand and take the math. Our knowledge of trigonometry can help us.

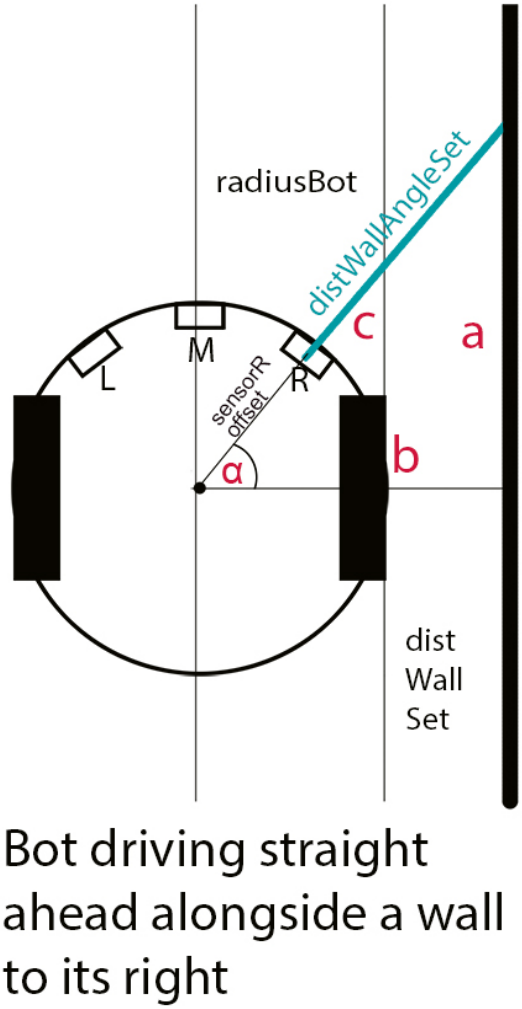

Fig. 1 Bot driving straight ahead

Given and searched values are the following ones:

$$b = radiusBot + distWallSet = Ankathete$$

$$c = sensorRoffset + distWallAngleSet = Hypothenuse$$

[Note: sensorRoffset = offset of sensor-R from the centre of the bot|

The value given is *distWallSet* (10 cm). The value searched is *distWallAngleSet*, it is corresponding to *distWallSet* over Trigonometry. (This is the value we are going to work with in our code.)

The trigonometric relationship between the two values is:

$$\cos \alpha = \frac{b}{c}$$

or

$$c = \frac{b}{\cos \alpha}$$

and then

$$distWallAngleSet = \frac{radiusBot + distWallSet}{\cos \alpha} - sensorRoffset$$

After these considerations, the above P-controller implemented in your code should be changed to this:

$omega_{new} = (distWallAngle_{set} - distWallAngle_{act}) * k_p$

Total waiver of sensor values for the cornering of the robot

In this improved Wall Follower, driving the corners, the robot will no longer be tedious to "touch down" the wall with its sensors, but it will be set on a fixed course using the vOmega interface. We will distinguish between two different turns, both of which have their own characteristics. These are the left and right bends (in a right-handed wall follower).

Driving a left turn

First, let's take a look at a left turn:

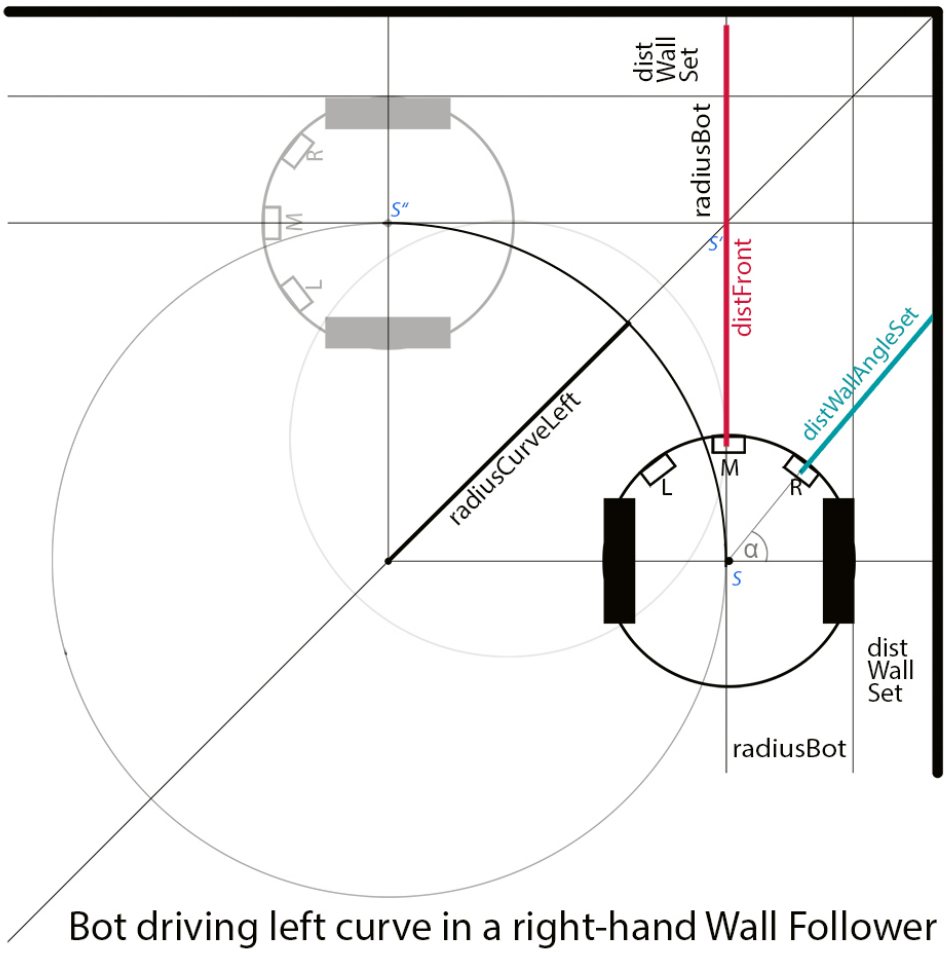

Fig. 2 Bot driving left curve

As the above figure clearly shows, there is a minimum value for the radius of curvature (radiusCurveLeft) that the bot should drive, and that is zero. In this case, the robot moves straight from its position S to S ', stops, turns 90 degrees to the left on the spot, and then continues straight ahead, beyond S ". We call this behavior **path control with the decoupling of the straight line (v) and angular velocity (ω)** (gr. Bahnsteuerung mit der Entkopplung von der Geraden- und Winkelgeschwindigkeit). If such an edged curve should be avoided, the radius for cornering must be increased. In the figure above the radius is twice as large as the radius of the bot. How big the radius should be chosen sensibly, cannot be said in general, because that depends on whether the robot should drive later also in small, narrow niches in the labyrinth or not. If the radius is too large, it is not possible for the bot to do so.

Depending on the selected radius *radiusCurveLeft*, the robot must turn left out of its straight-ahead position, when the front sensor M supplies the following value *distFront*:

$$distFront = radiusCurveLeft + radiusBot + distWallSet - sensorMoffset$$

sensorMoffset is the value that the sensor M is located away from the center of the robot.

Driving a right turn

When driving a right turn, the starting conditions and boundary conditions are slightly different for the robot than for the left turn (see above).

As shown in the figure below, the robot cannot immediately turn right when the R sensor signals that the wall end has been reached, but the robot must first move a little further straight ahead. In the figure, this route is marked with "freeDrive". (The meaning of *freeDrive* derives from the fact that no sensor sees an obstacle and therefore the bot "runs free".) Making use of trigonometry again, this value can be easily calculated, since *freeDrive* is the same as the counterkathete *a*.

$$freeDrive = a = b * \tan\alpha = (radiusBot + distWallSet) * \tan\alpha$$

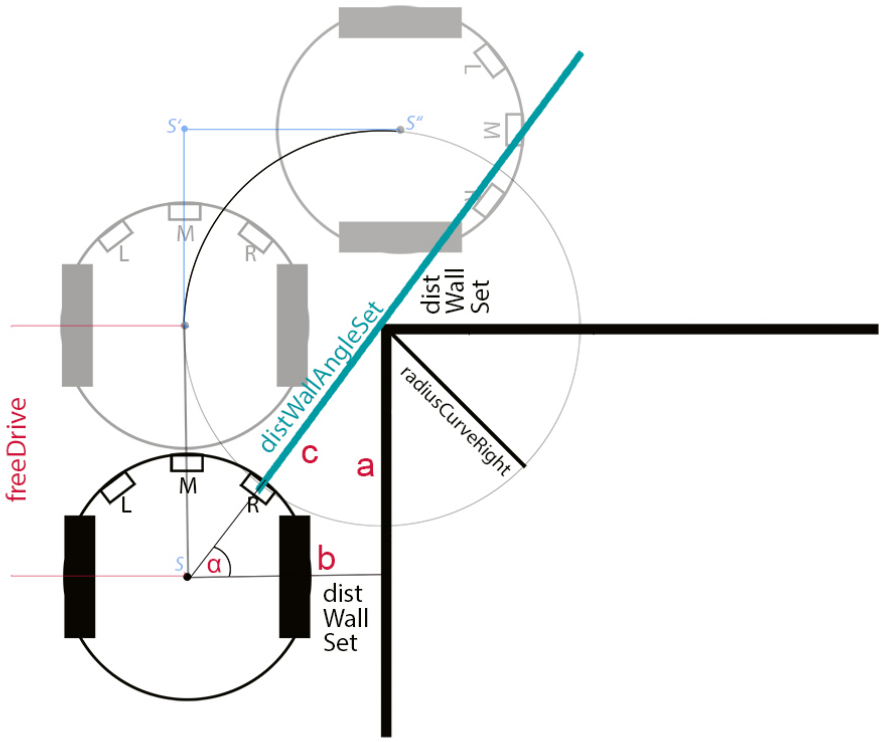

Fig. 3 Bot driving right curve

While the radius in the left curve (s.a.) could be at different values and - depending on the task - had to be chosen meaningful, it becomes clear from the above figure that this option does not exist for the radius of the right turn. In this case, there is an optimal radius *radiusCurveRight*, which is the sum of *radiusBot* and *distWallSet*.

$$radiusCurveRight = radiusBot + distWallSet$$