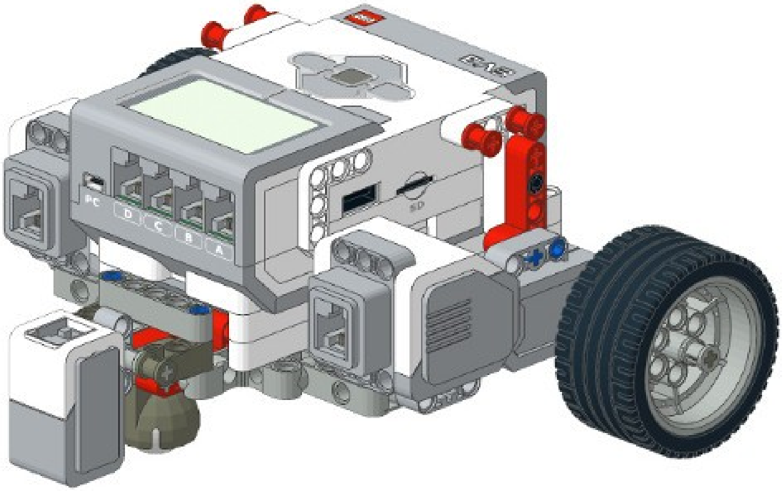
A POSTERIORI

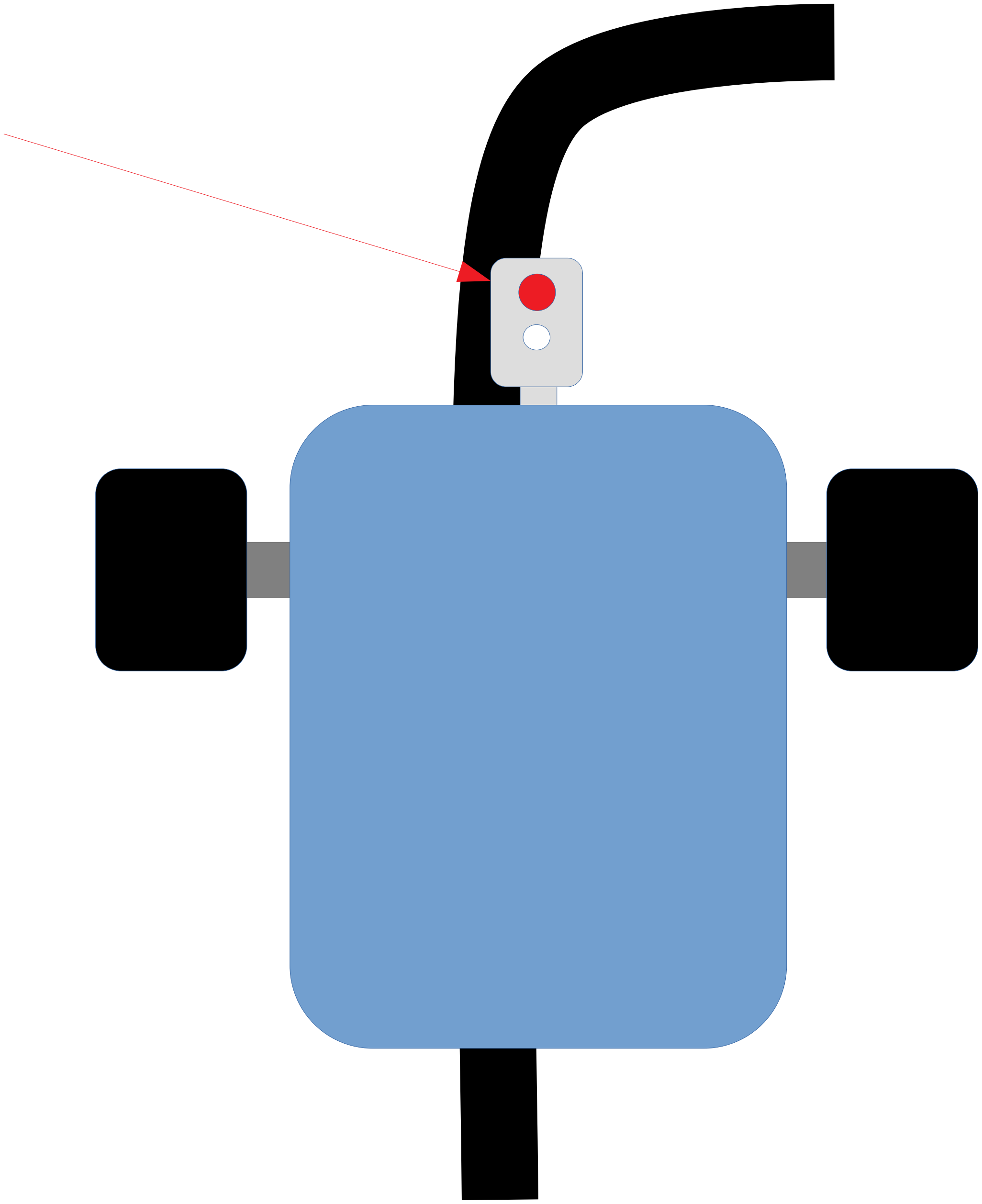
Play · Experience · Learn

SINGLE SENSOR LINE FOLLOWER



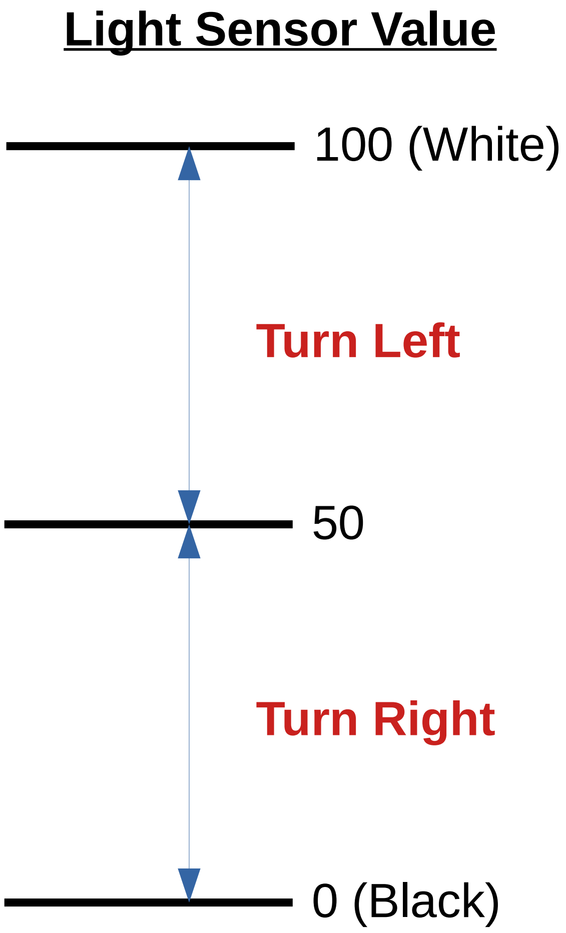
One Sensor Line Following

- Sensor on **edge** of line
- If sensor is reading…
  - White: Robot is too far right and needs to turn left
  - Black: Robot is too far left and needs to turn right



2 States Algorithm

- Loops forever
- Switch monitors reflected light
  - White (>50): Turn Left
  - Black (<50): Turn Right
- Robot “wiggles” left and right



Pseudo Code

```
IF Value > 50:
  Turn Left
ELSE:
  Turn Right
```

2 States Algorithm

```
def line_follow(speed):
  if color_value > 50:
    # Turn Left
    move_steering(-10, speed)
  else:
    # Turn Right
    move_steering(10, speed)
```

Pseudo Code
Don’t copy it blindly; it won’t work
Read it, understand it, write your own

IMPORTANT!
The function does not have a loop. You’ll need to either call the function in a loop, or add a loop into the function.

```
while True:
  line_follow(100)
```

Why 50?

- If the sensor is calibrated to “Black: 0”, “White: 100”, 50 is the mid point between them.
- Some robots / API do not have a way to calibrate the sensor, if so…
  - Black and White won’t be 0 and 100
  - Mid point will not be 50
  - You’ll need to measure black and white and determine the midpoint yourself

Looping

- If you tried the program now, it won’t work
- The “line_follow” function only checks the color sensor **ONE** time, then it’ll stop checking and continue moving in the same direction
- Need to use a loop to continuously check the color sensor

```
while True:
  line_follow(100)
```

Note

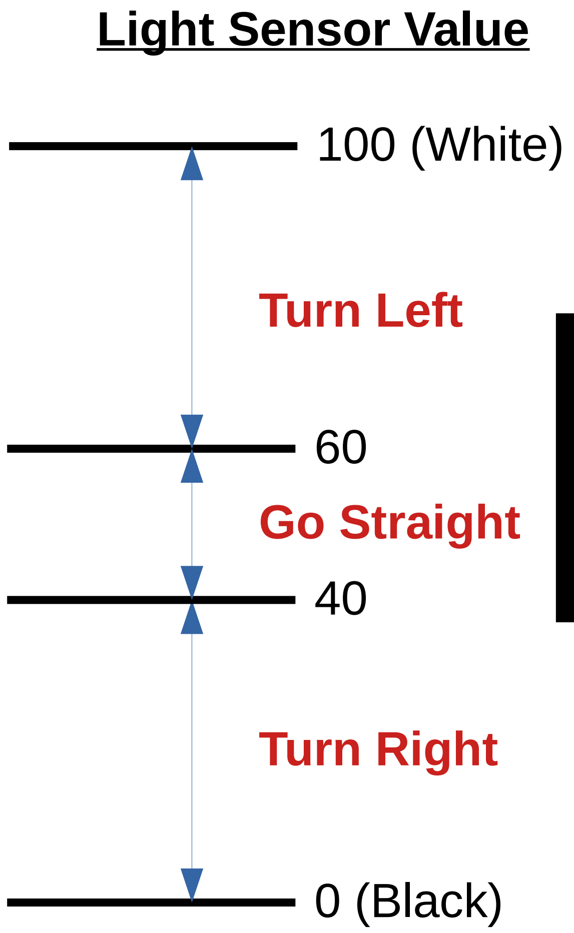
- A “while True” loop will never end, but it is useful for testing
- To make this useful, you’ll need someway to end the loop. Read the “Ending the loop” to learn how

Common Problems

- Problem:
  - Movement is slow and jerky
- Why?:
  - Robot ONLY move left and right. It never goes straight.

3 States Algorithm

- Check for **Black**, **White**, and **Grey**
  - White (>60): Turn Left
  - Black (<40): Turn Right
  - Grey (Between 40 to 60): Go Straight
- Robot runs smoother



Pseudo Code

```
IF Value > 60:
  Turn Left
ELSE IF Value > 40:
  Go Straight
ELSE
  Turn Right
```

3 States Algorithm

```
def line_follow(speed):
  if color_value > 60:
    # Turn Left
    move_steering(-40, speed)
  elif color_value > 40:
    # Go Straight
    move_steering(0, speed)
  else:
    # Turn Right
    move_steering(40, speed)
```

Pseudo Code
Don’t copy it blindly; it won’t work
Read it, understand it, write your own

Note

- The “40” and “60” are just examples, you’ll need to measure and decide on suitable values for yourself
- I like to perform my comparison from top down, starting from the highest value (>60), and moving down. It’s not the most efficient, but it’s neater and I’m less likely to make mistakes.

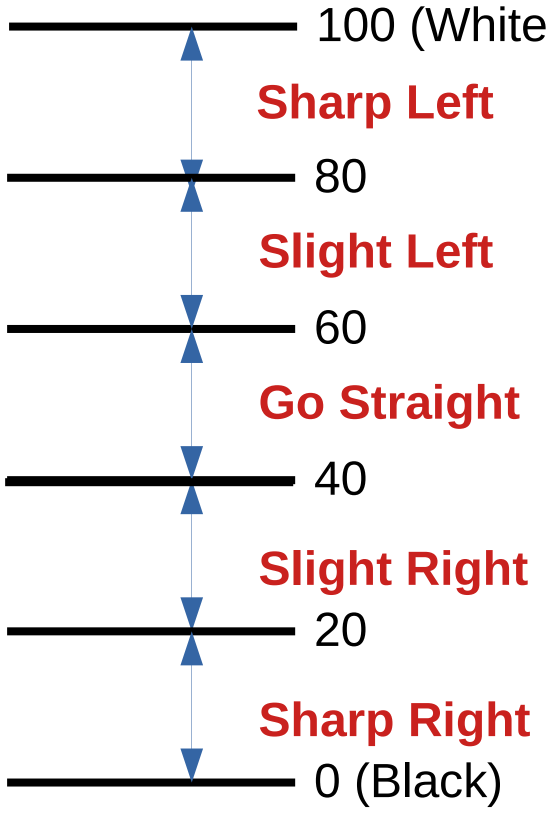
Common Problems

- Problem:
  - Better than 2 states, but still a little jerky
  - May be good enough
- Can we do better?

5 States Algorithm

- Take it a step further by checking for 5 levels of light sensor value:
- Robot runs even smoother than 3 states

Light Sensor Value



Pseudo Code

```
IF Value > 80:
  Turn Sharp Left
ELSE IF Value > 60:
  Turn Slight Left
ELSE IF Value > 40:
  Go Straight
ELSE IF Value > 20:
  Turn Slight Right
ELSE
  Turn Right
```

5 States Algorithm

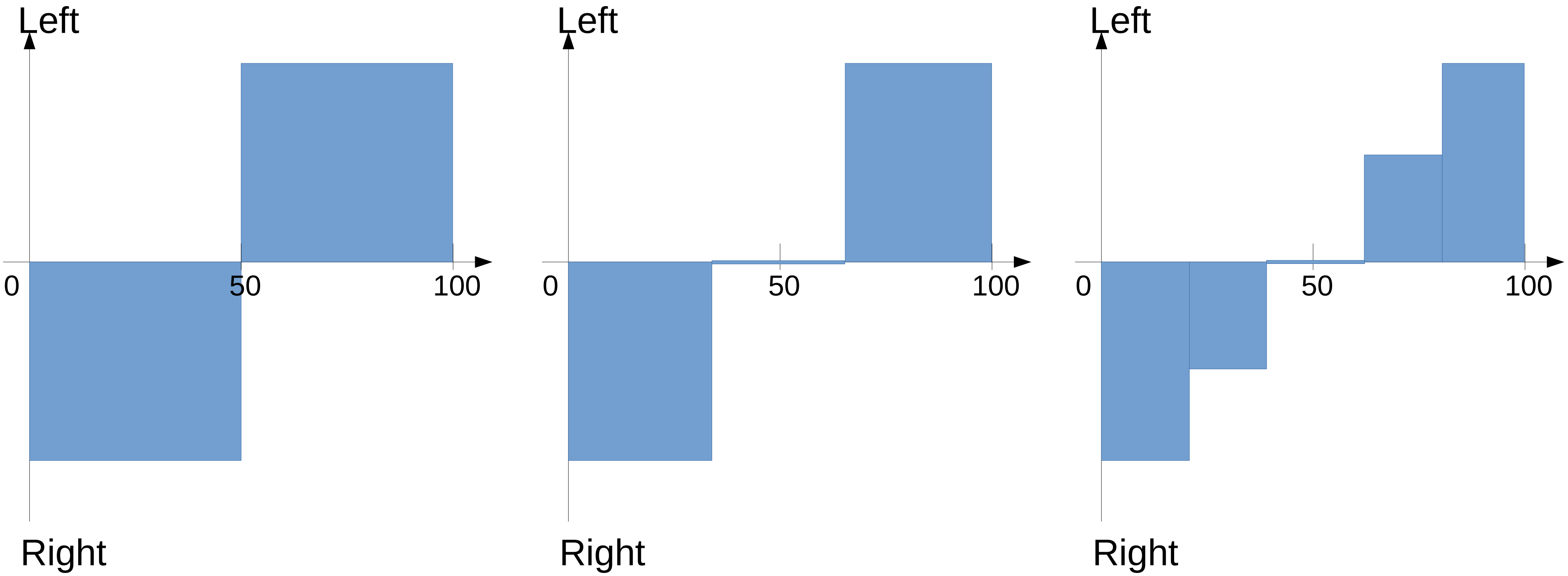
```
def line_follow(speed):
  if color_value > 80:
    # Turn Sharp Left
    move_steering(-80, speed)
  elif color_value > 60:
    # Turn Slight Left
    move_steering(-40, speed)
  elif color_value > 40:
    # Go Straight
    move_steering(0, speed)
  elif color_value > 20:
    # Go Slight Right
    move_steering(40, speed)
  else:
    # Turn Sharp Right
    move_steering(80, speed)
```

Pseudo Code
Don’t copy it blindly; it won’t work
Read it, understand it, write your own

Note

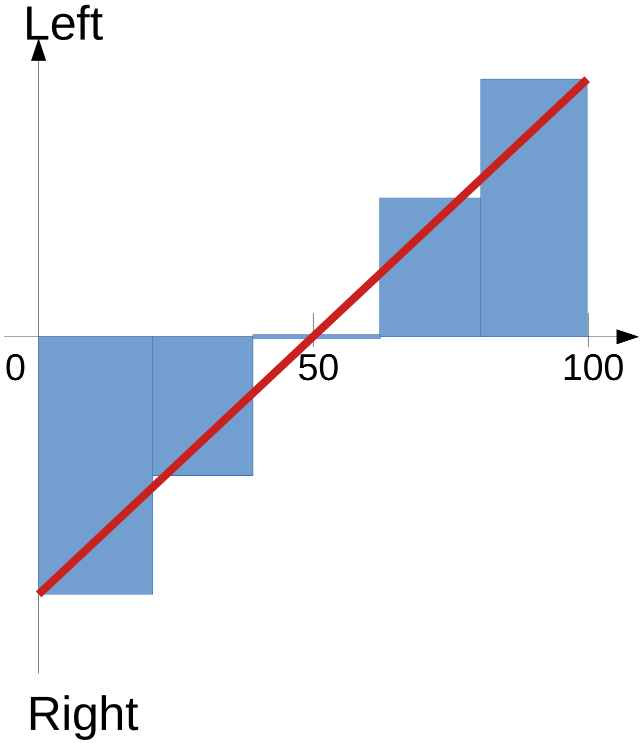
- As before, the numbers used are just examples, you’ll need to measure and decide on suitable values for yourself

Comparison of 2, 3, 5 states



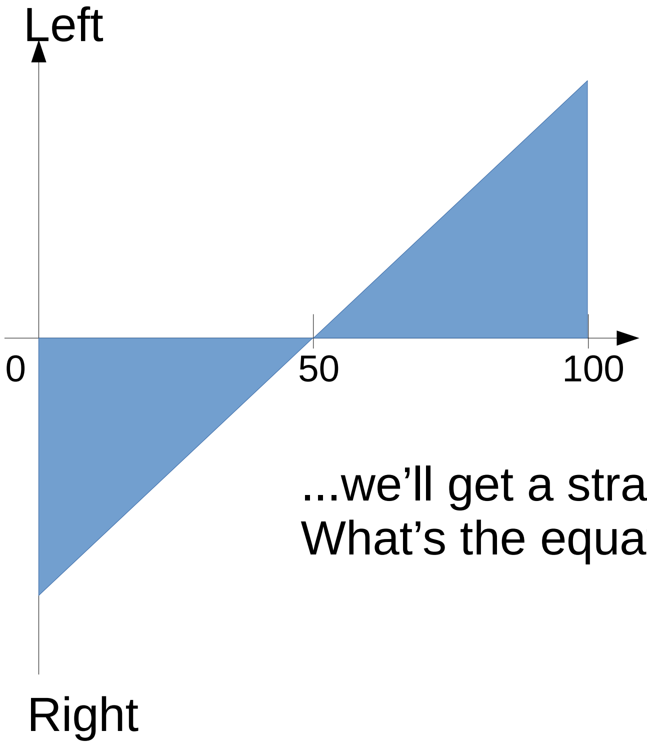
What happens if I increase the number of states?
(eg. 7 states, 9 states, 11 states)

Increasing number of states



As we increase the number of states, the diagram starts to look more like a straight line.

What if we have an infinite number of states?



...we’ll get a straight line!
What’s the equation of the line?

Equation of line

- Standard form

  $y = mx + c$

- Crosses x axis at x = 50, y = 0

  $0 = m(50) + c$

  $m = -c / 50$

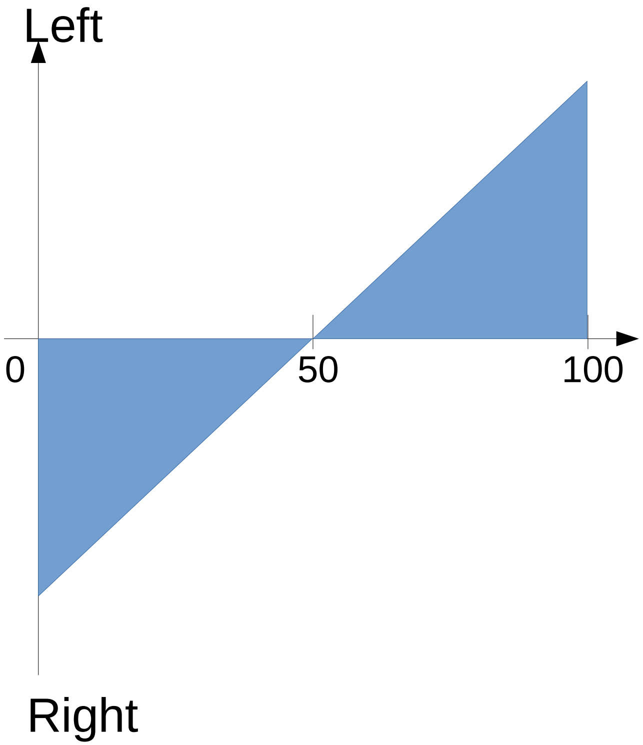
- Substitute and rearrange

  $y = (-c / 50)x + c$

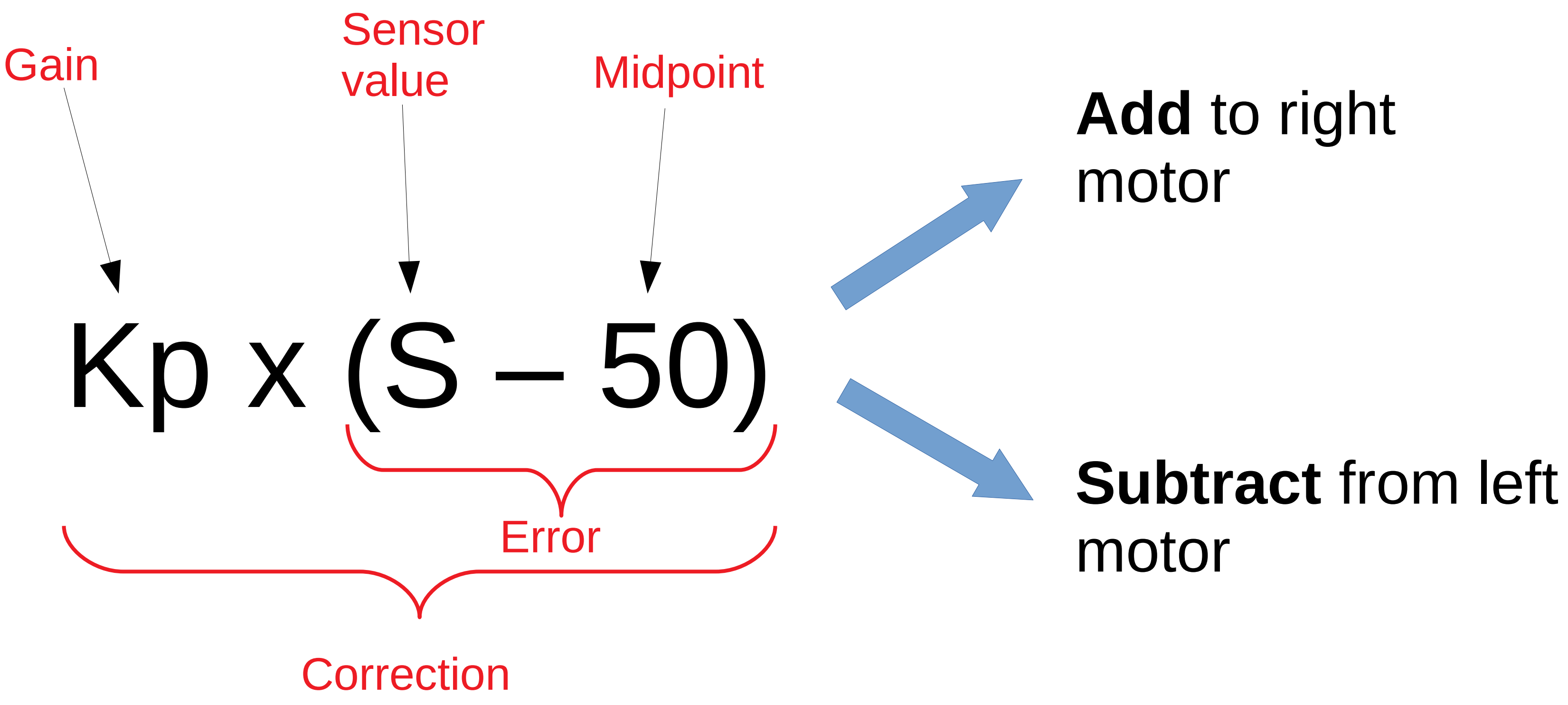
  $y = -c (x / 50 - 1)$

  $y = -c / 50 (x - 50)$

  $y = k (x - 50)$, where $k = -c / 50$



Equation of line (Engineering Style)



Gain

Sensor value

Midpoint

Kp x (S – 50)

Error

Correction

Add to right motor

Subtract from left motor

These are standard engineering terminology. Professional engineers uses these terms to make themselves sound smarter. You should do the same!

* The “p” in “Kp” stands for proportional. In a full PID (Proportional, Integral, Derivative) control, you will also have an “Ki” and “Kd”.

Proportional Control

```
def line_follow(speed):
  GAIN = 2
  error = color_value - 50
  correction = GAIN * error

  move_steering(correction, speed)
```

Pseudo Code
Don’t copy it blindly; it won’t work
Read it, understand it, write your own

Note

- The value of “GAIN” doesn’t change when the program is running. Such values are called **constants**, and by convention, we use all CAPS to name them.
- As before, you’ll need to determine a suitable mid point
- GAIN will need to be tuned for your robot

Proportional Control

- Changing Gain:
  - Increase: Turns more sharply, may wobble
  - Decrease: Tuns more smoothly, may fail at sharp turns
- Is proportional control the best solution?
  - Depends. Proportional controls have a **straight line** response, and you **can only tune the Gain** (gradient of the line)
  - High gain may wobble too much, low gain may fail at sharp turns. **Depending on the map and robot, there may not exist a Gain value that is both smooth and can handle sharp turns.**

Proportional Control

- Test to find the best gain!
  - Suggest testing within the range of 0.1 to 4
- Possibilities to explore:
  - Gain as a parameter to the line follower function
    - Allow you to use the best gain for each situation
  - Non-proportional control (ie. not a straight line eqn).
    - Will a quadratic eqn work? (spoiler: No it won’t, but why not?)
    - What about a cubic eqn?
  - Add in Integral and Derivative terms to make it a PID controller

Ending the Loop

- A “while True” loop will never end; your robot will line follow forever and won’t do anything else
- Need to stop the line following at some point
- Most common is by wheel rotations

```
while True:
  line_follow(100)
```

```
def line_follow_distance(cm, speed):
  target_degrees = cm / circumference * 360
  left_wheel_reset_degrees()
  while left_wheel_degrees < target_degrees:
    line_follow(speed)
```

Note

- (Slightly) Better to use the average of the left and right wheel
- Reset the wheel rotation to zero before starting the loop
- If the wheel is going backwards, the degrees will **decrease** and become **negative**. Adjust the code accordingly.

Ending the Loop

- Other options for ending the loop…
  - By ultrasonic sensor distance
  - Until left / right color sensor sees black
  - Until left / right color sensor sees white
- The robot will not stop automatically when the loop ends, you’ll need to give it a stop command
- Same technique applies to gyro follower

Copyright

A POSTERIORI

Play · Experience · Learn