3:0 HR

Walk the Line

Stay on track using the line follower array of sensors.

Introduction

A crucial element of robotics is ensuring machines do exactly what they are designed to do, no matter how many times they repeat a task. In the case of mobile robots like Alvik, this means staying on track as it moves around to complete jobs. If the robot suddenly lost its course, we would have a serious problem.

Trails of breadcrumbs simply won't work, but it is the right idea. Using **reference points** is key to knowing exactly where we are and where we are going. Humans use it all the time - we define where we are at the moment and where we want to go using information such as paths and pins on a map. Just imagine trying to get somewhere if you don't know your starting point or don't have a path to guide you!

In upcoming projects, we will experiment with the **Line Follower Array** to guide our robot along marked paths. This powerful array of sensors helps Alvik identify its location in space so that it can move with precision. We will warm up with a few practice exercises and finish off with a race against friends.

Buckle up and try to stay on track!

Learning Objectives

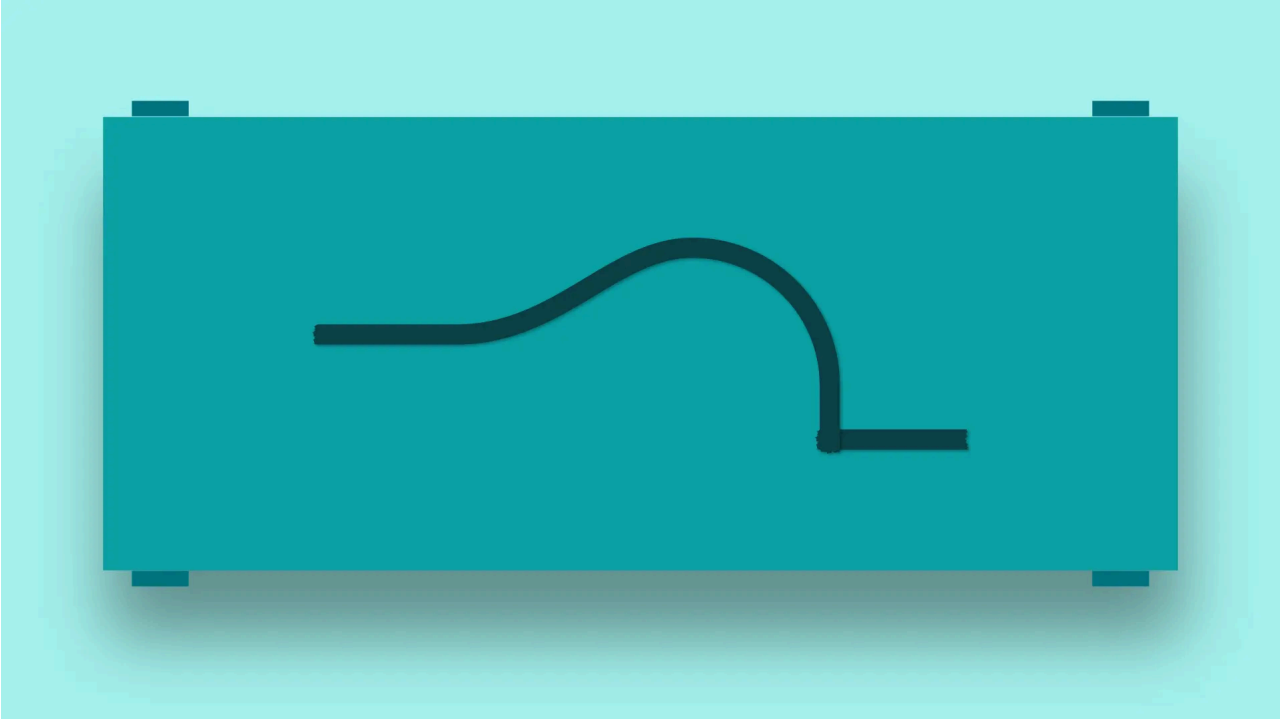
- Understand how the Line Follower Array of sensors work.
- Develop basic and advanced PID line following scripts.
- Program the robot to follow straight and curved paths.
- Program the robot to turn on changes in path angles.
- Calibrate key values to improve path following.

Setting Up

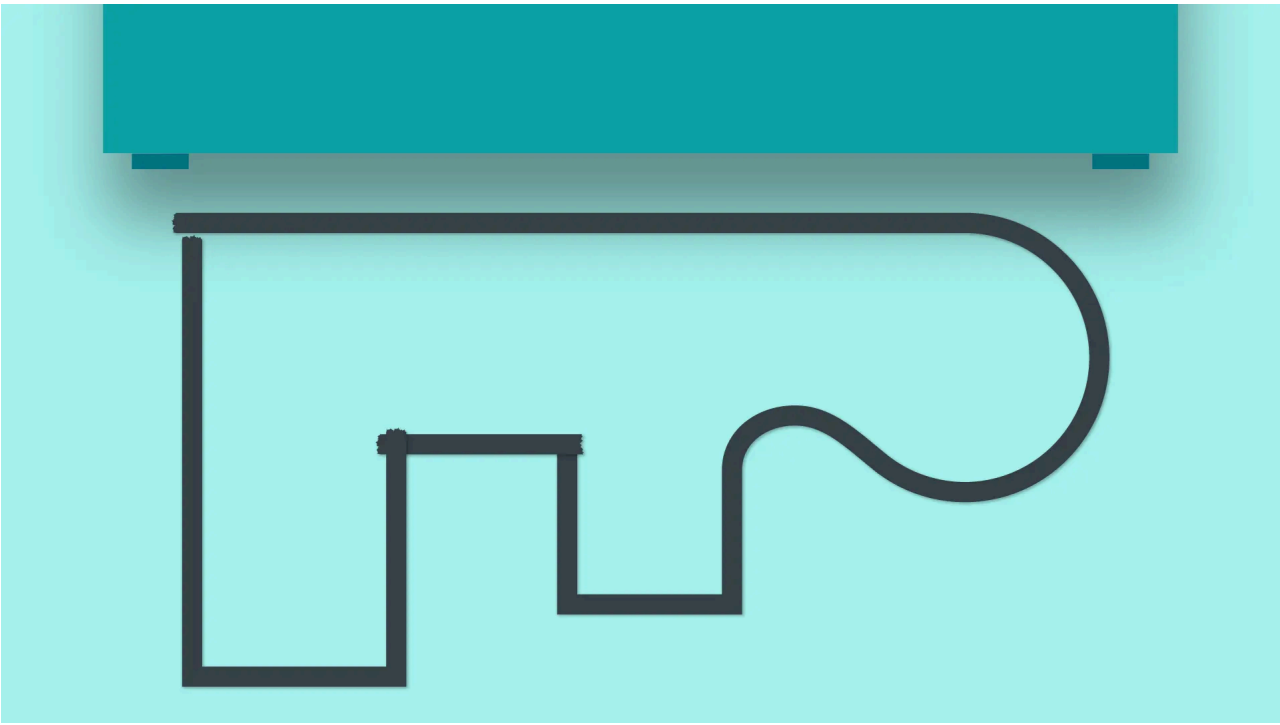
Make sure you have the following items ready for our project:

- **Black Electrical Tape**
- **Tape Measure or Ruler**
- **Notebook** (recommended)

We will use black electrical tape to mark the path for the robot to follow. The first example is for warm-up practice and the tape should ideally be placed on the tabletop of your computer workstation.



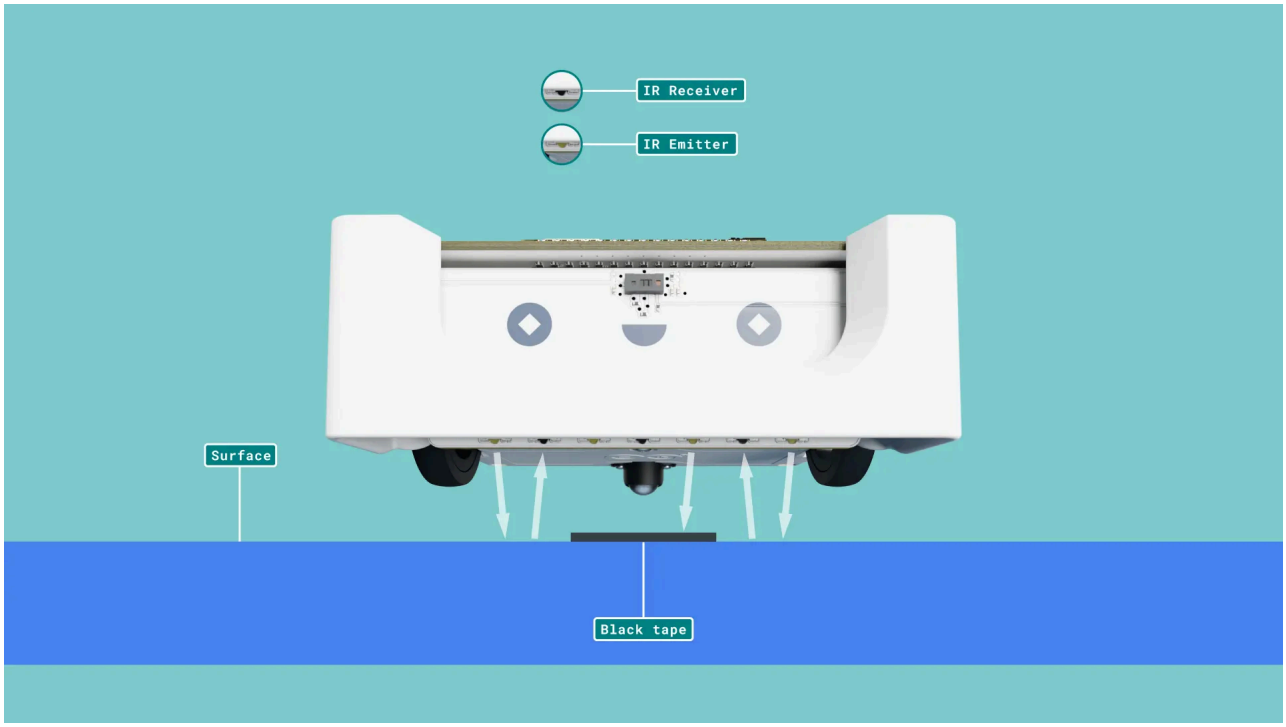
The second image provides an example design of the racetrack we will build for the final challenge, which due to its large size, should best be placed on the floor or a large tabletop.



Line Follower Array

A Line Follower Array is a set of infrared (IR) sensors that the robot uses to detect and follow lines on the ground. These sensors emit infrared light and then measure the amount of light that's reflected back. Lighter surface colors reflect more light, while dark colors (like a black line) absorb more, hence reflecting less light.

Look carefully at the front-underside of your robot where you will find a row of seven tiny photodiodes. There are a total of four transparent IR LEDs known as **IR Emitters**, which are responsible for emitting infrared light that is invisible to the human eye. The three black photodiodes measure the amount of emitted infrared light reflected back to them - these are called the **IR Receivers**.



Alvik uses the difference in reflected light to determine where the line is. For example, when the middle IR Receiver detects less reflected light, the robot knows it's centered over a black line. If Alvik starts to veer to the left or right, the IR Receiver on one side will detect less reflected light than the others. It can then adjust its course to re-center itself on the line.

Sensor Test

Make sure you are ready with a simple black-line path on your desk. We will use this line to test the signal strength of Alvik's three IR Receivers.

Create a new project file and copy the code below into your editor.

```
from arduino import *
from arduino_alvik import ArduinoAlvik

alvik = ArduinoAlvik()

def setup():
  alvik.begin()
  delay(1000)

def loop():
  ir_left, ir_center, ir_right = alvik.get_line_sensors()
  print(ir_left, ir_center, ir_right)
  delay(500)

def cleanup():
  alvik.stop()

start(setup, loop, cleanup)
```

With three IR receivers in a row, your robot can detect reflected light in the following zones - left, center, and right. For that reason, when unpacking the values from `alvik.get_line_sensors()`, we assign them to the variables `ir_left`, `ir_center`, and `ir_right` before printing to the terminal.

The goal during testing is to become familiar with sensor readings by placing the IR receivers over different parts of the black tape while observing the terminal. Here are a few testing examples to guide you:

1. Place each IR receiver directly over the black tape, one at a time.

2. Position the robot so that two or all three IR receivers are over the tape.

3. Align the tape between two IR receivers so they barely reach the tape's edge.

Signals will vary depending on external factors, but in any case, you should have noticed strong changes in readouts while testing. Assuming there is not much "noise" coming from other light sources and you are working on a light-colored

surface, you likely detected values around 50-60 where there is no tape and 700-750 while directly over the tape. The higher the value, the darker the surface.

Basic Line Follower

Problems have many solutions. Sometimes the answer is simple, while other times it is more complex. Our goal as problem-solvers is to find the best solution, which often means finding the right balance between simplicity and satisfying wants and needs.

In this lesson, we can state our problem as a question: **How do we make Alvik follow a line?** We can then redefine the question more specifically. For example, **how do we make Alvik follow a line smoothly?**

Now let's dig even deeper to improve our understanding of the problem by asking questions like:

- **How do we make Alvik follow a curved line smoothly?**
- **How does it know to turn if the path has a 90º angle?**
- **What if there is a break in the path - what do we do in that case?**

The ability to critically evaluate problems is what makes the best programmers. You should think carefully about the scenarios we will encounter ahead, but for the moment, let's focus on the problem of helping Alvik follow a straight or curved line. Once we have achieved a simple solution, we can begin thinking about improvements.

Create a new file called ***basic_line_follower.py*** and let's get started with an easy script.

```
from arduino import *
from arduino_alvik import ArduinoAlvik

alvik = ArduinoAlvik()

# Constants to set and adjust wheel speed changes
BASE_SPEED = 20
```

```
STRENGTH = 0.1  # Percentage of speed change, calculated in ADJUSTMENT
ADJUSTMENT = BASE_SPEED * STRENGTH

def setup():
  alvik.begin()
  delay(1000)

def loop():
  ir_left, ir_center, ir_right = alvik.get_line_sensors()
  print(ir_left, ir_center, ir_right)
  delay(10)
```

We begin by initializing `BASE_SPEED`, `STRENGTH`, and `ADJUSTMENT` - three constants used to calculate the right and left wheel RPMs of your robot. The goal is to keep both wheels moving in the same direction at an RPM of `BASE_SPEED` when Alvik is centered on the black line. While centered, Alvik will travel at exactly 20 RPM for both wheels.

However, if your robot is too far to the left or right, we need to apply an adjustment to the wheels to get it back on track. This is where `ADJUSTMENT` plays and important role, calculated by multiplying `BASE_SPEED` by `STRENGTH`, which is a percentage used to tweak the adjustment power. In the example above, the `ADJUSTMENT` formula is 20 * 10%, resulting in an adjustment of 2 RPM.

Moving on to the main loop, you already understand sensor reading with `alvik.get_line_sensors()`, so let's just focus on the conditional statements to learn how the basic line follower works.

```
if ir_center > 300:
  alvik.set_wheels_speed(BASE_SPEED, BASE_SPEED)
elif ir_left > 300:
  alvik.set_wheels_speed(BASE_SPEED - ADJUSTMENT, BASE_SPEED + ADJUSTMENT)
elif ir_right > 300:
  alvik.set_wheels_speed(BASE_SPEED + ADJUSTMENT, BASE_SPEED - ADJUSTMENT)
else:
  alvik.set_wheels_speed(0, 0)
```

Recall the values read while experimenting with the sensor - around 700 signifies a black line, while about 50 is registered from the table's surface. By choosing a number at the approximate halfway point of these two values (e.g. 300), we can conclude that any reading below the halfway point is a lighter surface while any reading above the halfway point is a darker surface. **In other words, if an IR sensor is reading above 300, it is detecting the black tape, and vice versa.**

With this in mind, we have three conditions to check the status of Alvik's three IR Receivers. If the first condition for `ir_center` is true, the tape is centered and your robot drives straight by maintaining the same base speed. If the condition checking `ir_left` is true, meaning the tape is too far to the left, we recalculate wheel RPMs using `ADJUSTMENT` to return your robot to a centered position by steering slightly to the left. For the condition of `ir_right`, we do a similar adjustment in the opposite direction.

At last, in the case that no line is detected, we simply stop Alvik from moving.

Take a moment to think about the adaptive changes we have programmed, and of course, put them to the test. How responsive is the robot at correcting its path when it veers too far to the left or right?

- **Challenge:** What happened when Alvik arrived at the 90º turn? Can you create a new condition to better navigate 90º turns?

Advanced Line Follower

You probably noticed that the last solution wasn't perfect. It works for the most part, but there may have been a late response in path correction, which could become a serious problem on more complex routes. Also, the robot might have started oscillating, meaning it was bouncing left and right as it drove along the line. **Oscillation often begins when the robot tries to center itself on the line but instead overshoots its target and travels too far to one side.** Attempting to correct the error, the wheels strongly adapt their speeds and the robot overshoots the line again, traveling too far to the opposite side. The result is a "bouncing" loop as the robot tries to correct errors caused by its own correctional efforts.

A better solution is inspired by a branch of engineering called **control theory**, which is all about using clever feedback loops to develop smart controls. Over a century ago, a very famous control system evolved that is widely used today - the **PID (Proportional-Integral-Derivative) Controller.** With the help of PID, we will

develop a new program to help Alvik follow lines smoothly and turn corners at ease.

Understanding PID

It is incredible to think how much the design of intelligent machines has been inspired by the human system. Some of the best solutions, including PID, exist by attempting to simulate our behaviors and ways of thinking. Before we dive into PID, it is best to understand our own thinking with a relatable example.

Our brains are hardwired to seek out solutions, constantly gathering information and evaluating experiences to make better decisions. You are so good at problem solving, in fact, that you do it hundreds or thousands of times per day and may not even realize it!

As a simple example, imagine the water is too cold in the shower and you want to adjust the temperature. Your immediate reaction is to adjust the water knob to reach your preferred temperature. The colder the water, the more you turn the knob. Thinking as engineers, let's call the difference between the current temperature and your preferred temperature the "error".

Chances are that you didn't reach exactly the perfect temperature - maybe the water is still just a little too cold. You then make a very small adjustment to bring the temperature closer to your target. But wait, you sense the temperature rising too fast and anticipate that it may become too hot! Forecasting what will happen, you quickly dial back the water knob just a bit.

This analogy might seem a little abstract at first, but this is exactly how a PID controller works. Using the shower example, we can best understand PID controls as follows:

- **Proportional (P):** Adjustment made based on current error (first immediate turn of the water knob).
- **Integral (I):** Corrective adjustment based on past error (moving the knob a little bit more).
- **Derivative (D):** Additional adjustment based on anticipated future error (slight turn to fix forecasted overshoot).

PID in Practice

With help of the shower analogy, you should now be warmed up to PID... he he, get it?

Alright, dad jokes aside, let's start with a full script example of the PID Controller in action. First take a few minutes to familiarize yourself with the code. As you're reading, keep in mind that the overall goal is simple - find the best adjustment speed to keep Alvik centered on the line.

```
from arduino import *
from arduino_alvik import ArduinoAlvik

alvik = ArduinoAlvik()

# PID and Speed constants
KP = 40
KI = 1
KD = 0.05
BASE_SPEED = 20

# Function to determine positional error from the line
def get_position_error():
  ir_left, ir_center, ir_right = alvik.get_line_sensors()
  total = ir_left + ir_center + ir_right
  error = ((ir_left * 1 + ir_center * 2 + ir_right * 3) / total) - 2
  return error

def setup():
  alvik.begin()
```

We begin by defining the PID constants used to calculate automatic speed adjustments. These are `KP`, `KI`, and `KP`.

The PID values are not "one size fits all", so you may want to experiment with them while testing Alvik until you find the best balance that works for you. The constant `BASE_SPEED` is the usual RPM speed without any adjustment applied. Note that if you change the base speed, you'll need to re-calibrate the PID values.

We then define a clever function called `get_position_error` to determine Alvik's position on the line, which uses a weighted average to calculate how far the center of your robot is from the line. Remember while testing the sensor, we found values around 50 for lighter surfaces and 700 on dark surfaces like black tape. By applying numbered weights (1, 2, 3) to the `ir_left`, `ir_center`, and `ir_right` readout values, we determine position by understanding how "heavy"

each variable is. The best way to explain this is with examples as shown in the scenarios below.

- **Scenario 1: The robot is centered** on the line, reading values of (50, 700, 50). The `error` formula becomes (50*1 + 700*2 + 50*3)/800 - 2, resulting in **error = 0**. Note that the subtraction of 2 from the weighted average is done to "normalize" values, which just means making data easier to interpret or use. The number 0 for "no error" definitely makes more sense than the number 2!
- **Scenario 2: The robot is too far to the left** of the line, reading values of (50, 50, 700). The `error` formula becomes (50*1 + 50*2 + 700*3)/800 - 2, resulting in **error = 0.81**.
- **Scenario 3: The robot is too far to the right** of the line, reading values of (700, 50, 50). The `error` formula becomes (700*1 + 50*2 + 50*3)/800 - 2, resulting in **error = -0.81**.
- **Scenario 4: The robot is slightly to the right** of the line, reading values of (100, 100, 50) because the left and center sensors are barely touching the tape's edge. The `error` formula becomes (100*1 + 100*2 + 50*3)/250 - 2, resulting in **error = -0.2**.

As you have learned from the scenarios above, the robot's position will always be somewhere within the range of -0.81 to 0.81, where the middle value of zero signals that the robot is perfectly centered. As you will soon find out, these values are important in deciding how strongly we need to adjust motor speeds to re-center on the line.

Let's move on to `def loop():` to learn how this all comes together, starting with the key global variables `error`, `integral`, and `derivative`.

```
# Initialize key global variables
global error
global integral
global last_error

error = get_position_error()
integral += error
derivative = error - last_error

# PID calculation to determine speed adjustment
adjustment = KP * error + KI * integral + KD * derivative
```

The value returned by the function `get_position_error()` is saved to `error` as our **Proportional**, which we use to update the `integral` and `derivative` values of PID. Considering that **Integral** is based on the cumulative past errors, we simply use the `+=` assignment operator to add the new error to whatever is currently saved to `integral` (integral = integral + error). The **Derivative** attempts to predict upcoming errors by comparing the current error to the the previous error, calculated by subtracting `last_error` from `error`.

At this point, we have three error variables ready for PID:

1. Proportional: `error` is the actual positional **error right now**.

2. Integral: `integral` represents the **cumulative past errors**.

3. Derivative: `derivative` anticipates **future errors**.

The speed `adjustment` is calculated by multiplying each of these PID-related variables by corresponding `KP`, `KI`, and `KD` constants that we initialized earlier in our script. **KP, KI, and KD** are known as **PID Gains** which control how strongly `error`, `integral`, and `derivative` affect the adjustment calculation. You will need to play with the PID Gain constants during testing to find which values provide the best outcomes.

```
# Adjust speeds
left_speed = BASE_SPEED + adjustment
right_speed = BASE_SPEED - adjustment

# Ensure speeds stay in range 0-70 RPM
left_speed = max(0, min(70, left_speed))
right_speed = max(0, min(70, right_speed))

alvik.set_wheels_speed(left_speed, right_speed)
last_error = error
delay(100)
```

With `adjustment` calculated, we can correct Alvik's position by adapting the `BASE_SPEED` for each wheel. Check out these scenarios to get a general idea of directional changes based on returned `error` values.

- `error = 0`: Alvik is centered. The adjustment is a very low value or even zero, resulting in little or no change to the base speed.

- `error = -0.81` : Alvik is too far to the right and the adjustment calculation returns a negative value. This means `left_speed` will decrease by the negative adjustment, while `right_speed` increases because subtracting a negative value results in a positive value added. The final speed changes produce a corrective slight turn to the left.
- `error = 0.81` : Avlik is too far to the left and the adjustment calculation returns a positive value. This is the exact opposite of the previous scenario. `left_speed` will increase and `right_speed` will decrease, causing Alvik to slightly turn to the right.

Before setting the new wheel speeds, its important to add a control that ensures our calculations don't generate adjusted speeds that fall outside the realistic range of your robot (approximately 0 to 70 RPM). We do this using the `min()` and `max()` functions, which respectively return the lowest or highest value from the numbers passed to its parameters. For example, `max(8, 2)` would return the number 8, while `min(5, 1, 9)` would return the number 1.

Focusing on `min(70, left_speed)` , the function safely assures the RPM is never over 70. This may be counterintuitive as first glance, but it makes sense if you think about the possible values of `left_speed` . If `left_speed` is lower than 70, the `min()` function returns the value of `left_speed` , but if `left_speed` is ever greater than 70, the function returns 70 as the lower value.

With this in mind, let's use pseudo-language to simplify the code inside our `max()` function as **max(0, 0-70)**. You'll notice the `min()` function was converted to 0-70, because we know it can only produce values within that range. In this clever way, we can safely confirm that `left_speed` and `right_speed` will always remain within the range 0-70 RPM as returned by the `max()` function.

Wheel speeds are now updated and checked within range, ready to be put to work with `alvik.set_wheels_speed(left_speed, right_speed)` . Here is also the moment we record the `last_error` for calculations coming up in the next loop iteration.

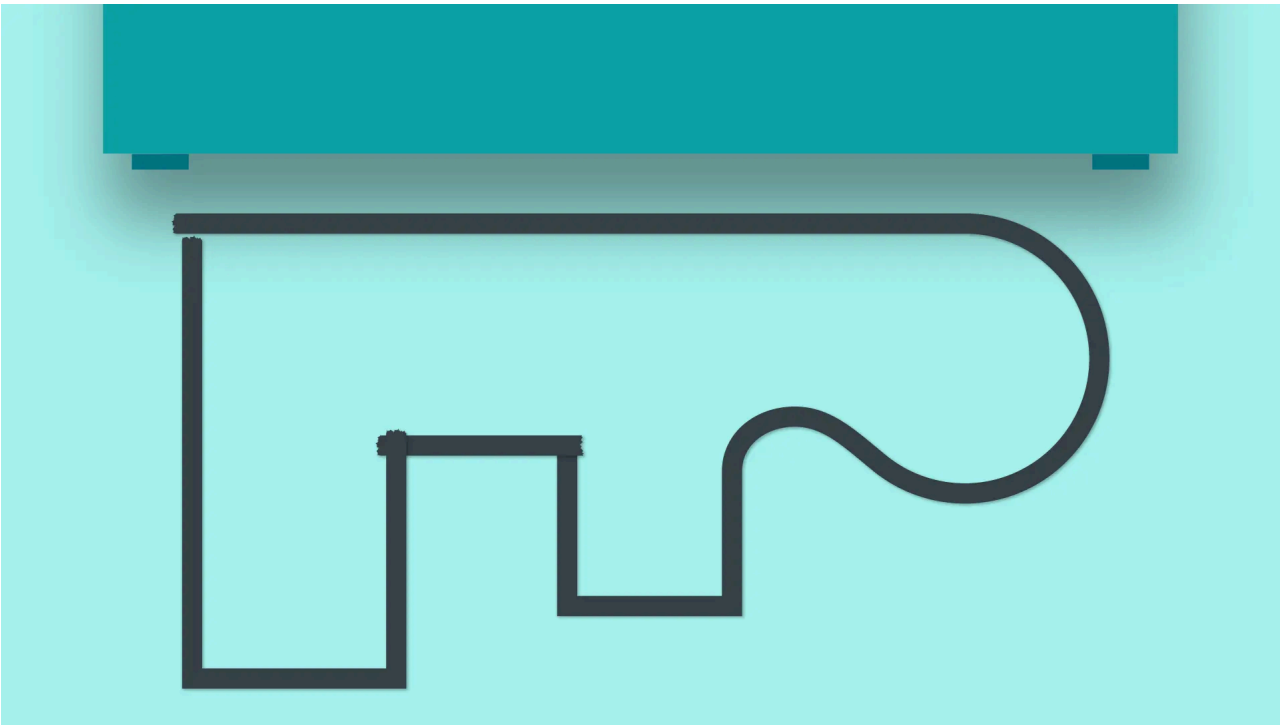
Like a Boss

Congratulations that you have come this far! Across all lessons, this is arguably the most complex script we have developed, so you should be proud of your efforts. If

you don't fully understand how to program with PID, that is completely natural and you should remember that understanding comes with time. You may need to repeat this section, possibly at another moment after giving your brain a small break. Soon enough, the principles of PID will become clear.

In all this coding and mathematic theory, make sure you have taken sufficient time to test the script and have fun! Remember that you need to play with PID Gains (KP, KI, KD), which will help you better understand how they effect Alvik's movements. This is especially important if you want to program a faster line following robot by increasing `BASE_SPEED`, which will require totally new calibration of PID Gains to achieve smooth travels.

Speed Run



Get your stopwatches ready, we are about to participate in an exciting robot race against friends!

You should have practiced calibrating PID Gains on Alvik, so let's test your engineering skill with a bit of fun competition. Create a racetrack on the floor using the image example above, or create your own track of any size. The goal is to calibrate Alvik so that it completes the course in the shortest time possible. Use

a stopwatch to record lap times, and if you are the winner, please share your secrets with the rest of the team so they can learn from your skills!