
XRP

Release 2023.0

Worcester Polytechnic Institute

May 10, 2024

COURSE INFORMATION

1	What Is the XRP Platform?	1
2	Indices and tables	89

WHAT IS THE XRP PLATFORM?

The XRP (Experiential Robotics Platform) [beta], developed through a collaboration between WPI and DEKA Research & Development Corp., aims to level the STEM playing field globally and create a future generation of STEM innovators and technology leaders.

The robot kits you received are designed to operate autonomously and perform basic tasks. Its simple, tool-free assembly means robots can be built quickly, and replacement parts can be easily 3-D printed. As part of this platform, WPI will provide virtual support through online courses and will guide students and teachers through the new system, including the ability to scale up using the same hardware with free software updates.

The XRP platform is part of WPI's global STEM education initiative, which will bring inspiration and possibility to STEM education in ways that make it available to all.

1.1 Welcome to Introduction to Robotics

Welcome to the WPI Global STEM Education Initiative Introduction to Robotics course using the new XRP Robots. This course teaches the basics of robotics and programming using two programming languages: Blockly and Python. For new programmers, we recommend that you start with Blockly and switch to Python once you gain more familiarity with programming the robots.

The course has several modules you will work through, starting with an introduction to robotics and later covering driving, sensors, the manipulator (robot arm), and more. Each module contains many interactive challenges using the robot; we recommend that you attempt each one in order to get a better understanding of how to program your XRP. With each module, new programming techniques are introduced to solve concrete robot problems, not as abstract unattached learning. This way, students will see the relevance and need for each concept introduced. The course ends with a challenging final project that brings together everything you have learned in all the previous modules.

The course can be run with one robot per student or with teams of a few students working on each robot. Working in teams allows students to help each other as they work through the course. But one student must not be doing all the work, preventing the other team members from learning the material.

This is a brand-new course with new robots and software, so there may be bugs and unexpected problems. We will strive to be responsive to any questions you might have. If you have any questions or find anything not working as you expect it to, feel free to contact us.

For questions and technical support for the XRP, see our Discourse server. There are instructions on how to sign up in the *Getting Help* section of this course.

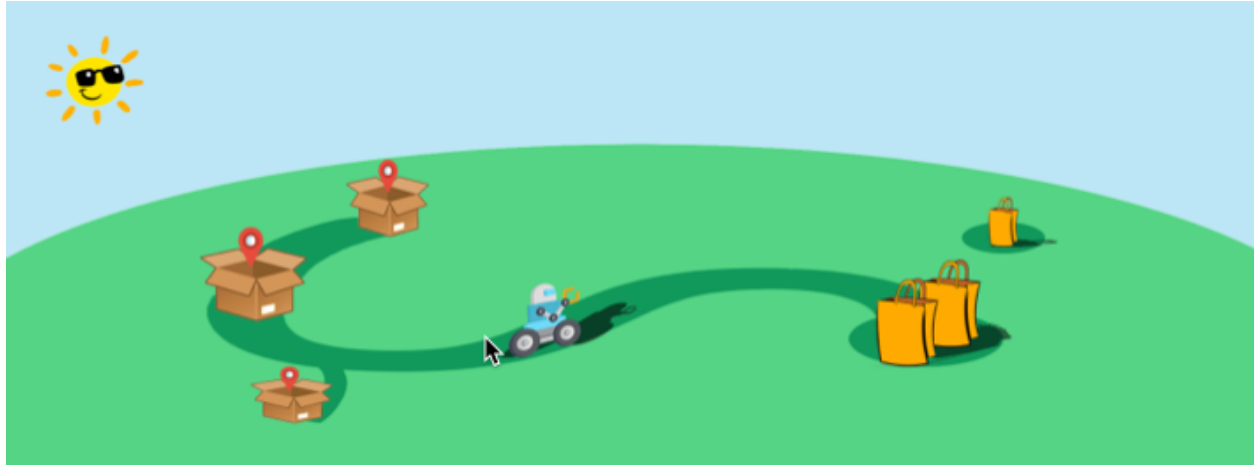


Fig. 1: Final project delivery robot challenge (see final project module for more information)

1.2 Getting Help

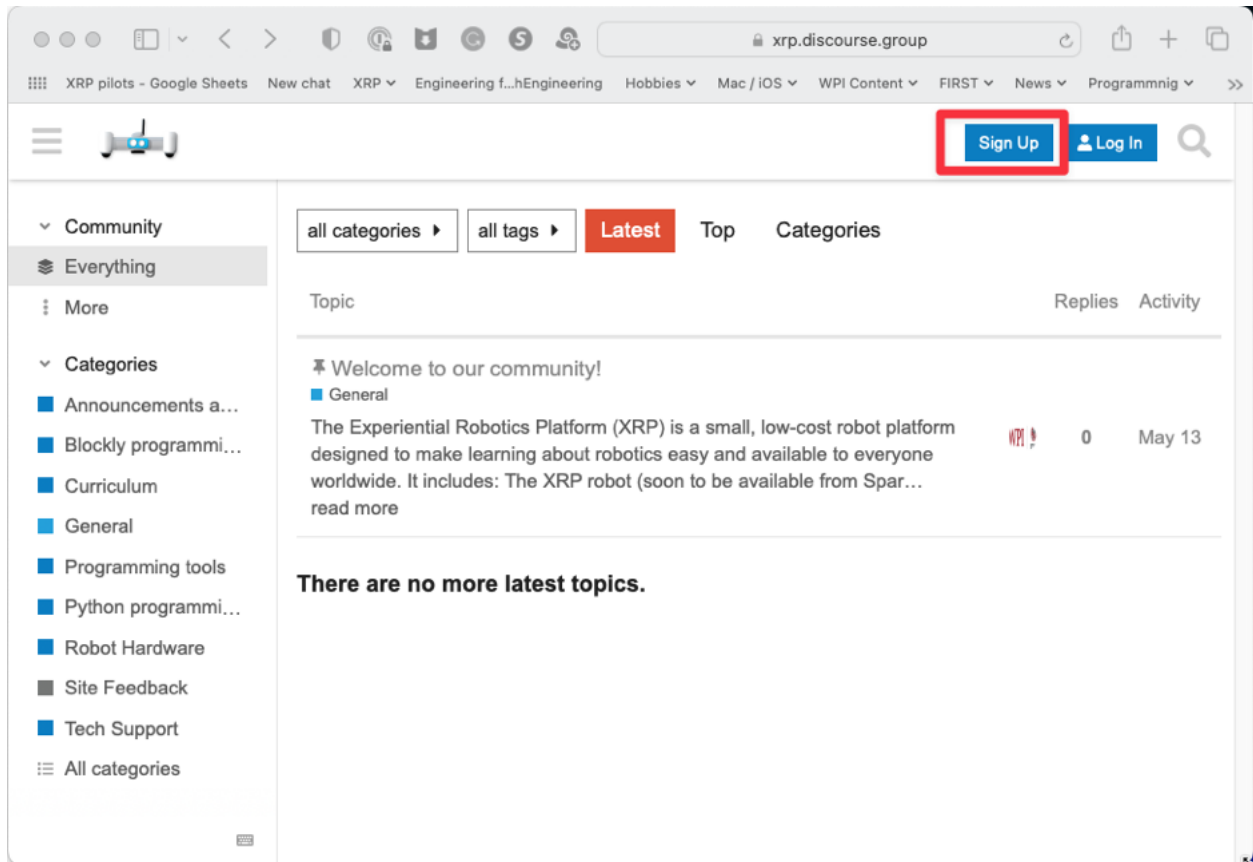
Alongside this ReadTheDocs course, you will be using several online resources for getting your questions answered. The support tools for any technical Q&A will be monitored by the development team and a growing community of users like you. We are using Discourse for online support during this course.

1.2.1 Discourse

The XRP Discourse server provides an easy way to communicate your questions, concerns, ideas, and conversations across the world (within this robotics program). Here you will find a place to post questions and get answers to issues that you might be having. The staff at WPI will do our best to provide timely answers to your questions, or you might find other more experienced community members jumping in and answering your questions. You will find important resources, links, event information, and office hours here.

Please follow these steps to join the Discourse server:

1. You will want to create your account, go to <https://xrp.discourse.group>
2. You can browse through all the resources on the Discourse server, if you want to post questions and participate, you need to sign up for an account. To do that, click on the Sign Up button in the top right corner of the window and enter your account information.



1. And you are done! You can now ask any questions you have about the course here or just chat with the other members of the course. We hope you will find this course fun and interesting!

If you want Discourse on your phone, for Android or iPhone there is a Discourse app in the App Store that you can download. When signing up remember to use `xrp.discourse.group` for the site's address when asked.

1.2.2 GitHub issues

Another way to contact us, particularly about issues or bugs with any of the software is to create issues in the GitHub repository where the software is developed.

To create a GitHub issue for the course that you are reading now, navigate to the [Issues page in the course GitHub repository](#)

To create a GitHub issue for the XRP software libraries, navigate to our [GitHub repository's issue page](#) and select "New issue".

Describe your problem in detail and click "Submit new issue" to post your issue in the repository. A member of the XRP team will reach out if we have further questions.

1.3 Introduction to the XRP: Module Overview

In this module you will:

- Learn about the field of robotics
- Assemble your robot
- Install the software required for programming the robot
- Write a short program to familiarize themselves with programming the XRP robot.

At the end of this module, you will be able to:

- Recognize the characteristics of robots
- Talk about the core disciplines that make up the field of robotics
- Understand the components of the robot and how it is assembled
- Build familiarity with the tools for programming the XRP

1.4 What is a robot

In this course we talk about robots as devices that can:

- Sense their environment
- Think and perceive what is happening around the robot
- Carry out actions using actuators (motors)

There are many ways to define a robot, and this is only one of them. Still, it is a good idea to have a definition of precisely what a robot is. Let's look at different devices and try to decide whether each of these are robots according to our definitions.

1.4.1 Examples

Radio Controlled Airplane



A radio-controlled airplane is operated by a person who holds a controller and uses the joysticks to control the airplane's flight path. It requires a remote pilot.

Actuators:

- propeller motor
- three control surface motors

Sensors: None

Summary: No sensors, no thinking, and it has to be completely controlled by a human.

Not a robot.

Drone



A quadrotor drone is capable of being teleoperated or using autonomous flight. The drone can fly a programmed course, avoid obstacles, return to the landing point, and automatically land.

Actuators:

- 4 propeller motors
- camera aiming motor

Sensors:

- GPS (Global Positioning System) receiver
- gyros and accelerometers
- heading sensor
- altitude sensor
- rangefinder

The drone senses the environment based on its sensed location and surroundings, and flies on its own from one place to another.

This is a robot.

Vacuum cleaner



A conventional vacuum cleaner is pushed by an operator around the area to clean the floor. Actuators: Motor to turn the fan that sucks up dirt.

Sensors:

- Fan speed sensor to ensure consistent performance.

The motor does have a sensor that keeps the motor running at a predetermined speed, but it does not sense the environment or have perception. The operator must supply all the “smarts”.

Not a robot.

Autonomous vacuum cleaner



An autonomous vacuum cleaner can — on its own — start up, vacuum one or more rooms, come back to clean its home base, then continue vacuuming until the whole job is done. It maps out each room in the house for more consistent operation.

Actuators:

- Drive motor for the wheels to get around.
- Motor for the vacuum for cleaning.
- Motor in the base (not shown) that will suck the dirt out of the vacuum cleaner so it can continue cleaning.

Sensors:

- Camera for visualizing the room.
- Switches on the bumper to allow it to turn around after hitting obstacles.
- Rangefinders on the side to measure distance from walls.
- Sensor to detect carpet vs floor to change the motor speed.

An autonomous vacuum is fairly smart. It can learn the map of a house after a few runs and efficiently clean rooms. It can avoid obstacles, clean rooms, stop to recharge, and continue where it left off.

This is a robot.

Self-driving car



A self-driving car can be driven conventionally by a human or driven autonomously on city streets and highways on its own.

Actuators:

- Wheels for driving.
- Motors for controlling turning.
- Actuators to allow the robot to break on its own.

Sensors:

- 8 cameras both outside and inside the car to view the environment and driver's attentiveness
- Rangefinders all around the car to measure the distance to adjacent vehicles
- GPS to determine car's location, and more.

The car is smart and represents state of the art robotics. It can sense the environment, understand where it will be over time, and drive to its destination while safely avoiding obstacles.

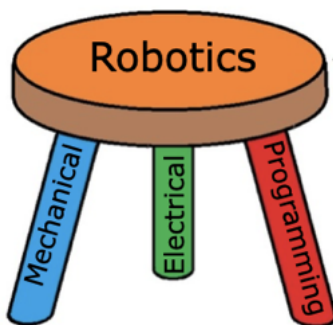
This is a robot.

1.4.2 What are the parts of robotics?

Robotics engineering is usually thought of as a combination of three disciplines. They are:

- Mechanical engineering - the design and analysis of mechanisms and other mechanical systems.
- Electrical engineering - the design of electronic circuits, especially all the sensors.
- Computer science - the development of advanced software (computer programs) to interpret all the sensor data, understand it, and drive the actuators.

Robotics can be thought of as the synergy of these three fields. Designing robots requires a “systems” approach to design. Having knowledge of all three subjects allows one to develop more complex and capable systems than one with only a unitary background.



Robotics is a 3-legged stool. Without any one of these subjects, it falls down.

1.5 Building the XRP Robot

Assembling the XRP robot is easy and a complete set of instructions and a video showing the process can be found in the [XRP User Guide](#).

1.6 Installing the programming tools

There are two programming languages used throughout this course, Blockly and Python. Both languages are programmed on the XRP robot using XRPCode, an online interactive development tool that can be used for developing programs in either language.

For more information about using XRPCode refer to the [XRP User Guide section on XRPCode](#).

1.6.1 Blockly

This is a drag-and-drop graphical programming system similar to Scratch. Blockly is a good choice for programming the robot, especially for learners with little to no programming experience.

1.6.2 Python

A more widely-used language that scales well for larger programs and works with professional programming tools such as Visual Studio Code and many others. The tool that will be used throughout this course for Python programming is XRPCode, although a more professional tool such as Visual Studio Code can also be used for writing CircuitPython code that is used by the XPR Robot.

Note: One strategy for getting students comfortable with programming more quickly is to start with Blockly getting through basic concepts such as functions, conditionals, loops, and basic robot programming. Once students are familiar with those concepts it is an easy move to Python to complete the course, especially for some more complex challenges, such as the final project. This will allow for quick onboarding without having to learn too many new concepts at the same time. But the decision will depend on the level and experience of the students taking the course.

1.7 Robot Driving: Module Overview

In this module students will:

- Learn how to make the robot drive at different speeds
- Understand the relationship between the wheel speeds and the robot's motion
- Be introduced to motor encoders and how to use them to measure the robot's movements
- Learn about the built in driving functions which help make complicated motions easy

At the end of this module, students will be able to...

- Make the robot drive in straight lines, turn corners, and turn in place
- Write and use basic functions in the Python programming language

- Use different types of loops in Python
- Break down a large, repetitive movement sequence into components and then convert into code

1.7.1 Covered Standards (NGSS and CSTA):

3A-AP-16 Design and iteratively develop computational artifacts for practical intent, personal expression, or to address a societal issue by using events to initiate instructions.

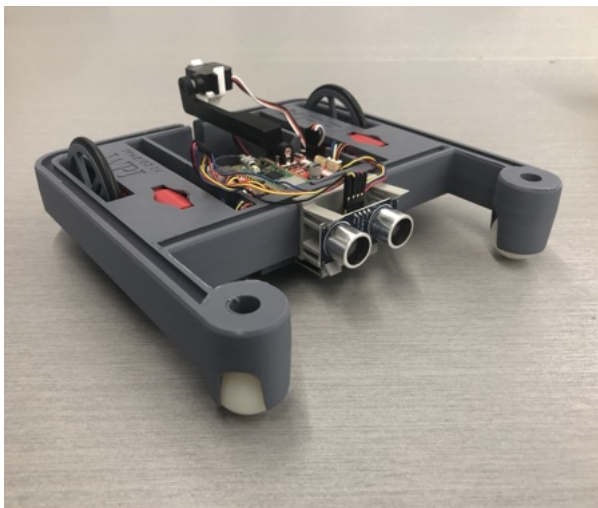
3A-AP-17 Decompose problems into smaller components through systematic analysis, using constructs such as procedures, modules, and/or objects.

3B-AP-16 Demonstrate code reuse by creating programming solutions using libraries and APIs.

1.8 Understanding Your Robot's Drivetrain

1.8.1 Introduction

The main body of your XRP is called the *drivetrain*. We call it this because it holds the two wheels which drive your robot around. Specifically, the XRP has a *differential drivetrain*. This is the same kind of drivetrain that you would see on a skid-steer loader.

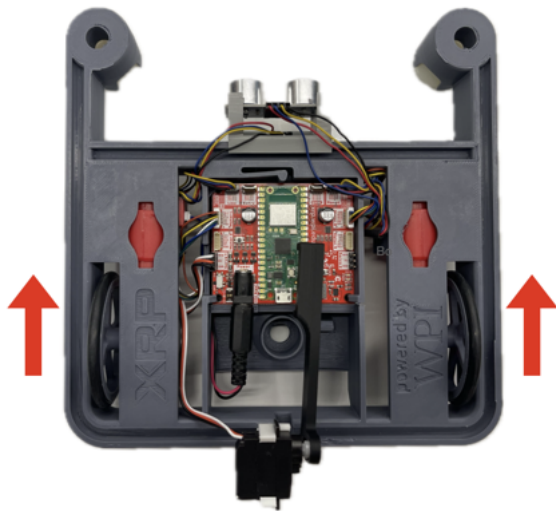


The XRP robot turns using skid-steering which means that while the robot is turning some of the wheels, sometimes the front casters, will be skidding. This requires more energy to complete the turns and depends on the friction of the front casters and the particular driving surface.

Each wheel of your XRP's drivetrain has a motor attached to it. We can use code to tell this motor what we'd like it to do. Getting the robot to move in a desired path is done by setting the speeds of the two drive motors. There are a lot of variations in how these motors can be set. Here are some basic examples:

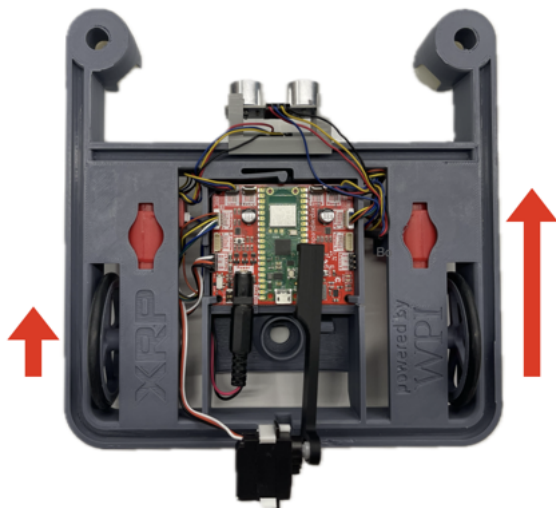
1.8.2 Driving straight

The most fundamental common driving that the robot will do is traveling straight. To do this, both wheels move forward at the same speed so that the robot moves straight.



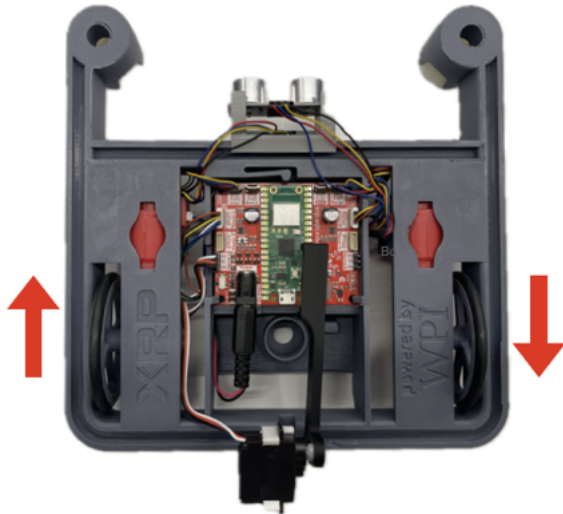
1.8.3 Turning in an arc

Robots can turn in an arc-shaped turn, where one wheel drives faster than the other, causing the robot to turn away from the faster wheel. As the difference in speed between the two wheels increases, the turn tightens. If the turn continues, the robot will drive in a complete circle. In this case, the radius of the circle decreases as the wheel speed differences increase.



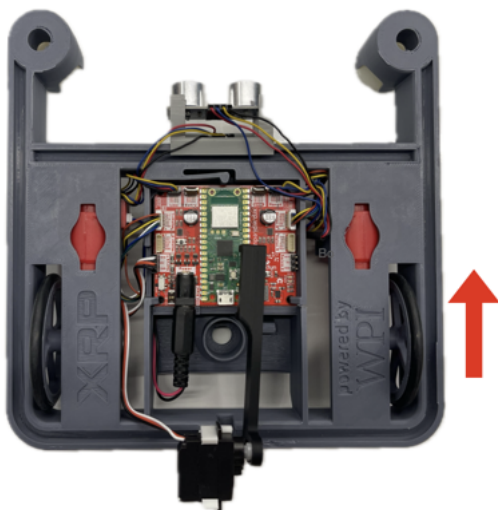
1.8.4 Turning in place

One advantage the XRP has over a car steering system is that it can turn in place where the robot turns on a point roughly between the two wheels. This allows the robot to easily get out of tight spaces without having to make a wide arc turn. The point turn is often used for navigating the robot between two places since it follows an easily predictable path.



1.8.5 Turning on one wheel

If one wheel drives forward or backward, and the other wheel remains stopped, the robot will turn in place, with the turning center being the stationary wheel. This is often called a swing turn because the robot swings around the non-moving wheel. With a swing turn, the diameter of the circle traced by the outside wheel is double the wheel track.



1.8.6 Effort

There are several ways we can tell the motors what to do. The most basic thing we can control is the *effort* the motor should be applying.

Imagine you are riding a bike on a flat surface, pedalling at a normal speed. Now imagine you encounter a hill. If you keep pedalling at the same speed, you won't slow down when you go up the hill. However, this is not easy! You'd need to pedal *harder* to go the same speed up the hill.

Now instead imagine that when you get to the hill, you keep pedalling as hard as you were on the flat section. You'll go up the hill slower, but you won't be as tired. This is what we mean by the *effort* of the motor. You're not telling the motor how fast it should move, but rather how hard it should work. If you tell your robot's motors to work at a constant effort, your robot's speed will change depending on whether it is driving on a flat surface or an inclined one.

<https://youtu.be/z6aIVpf3qN0>

https://youtu.be/Zcr83kcO_Pk

In both videos, the robot is using the same effort. In the first video, the robot is slowly moving uphill because gravity is fighting against its effort. In the second video, the robot is moving quickly downhill because gravity is working in the same direction as the effort. The force output from the motors is the same, but the speed will depend on resistance to the force.

Tip: Effort is also like the throttle in a car. If you're going up a hill, you need to push the throttle more to maintain the same speed on the hill. If you don't push the throttle more, you'll slow down.

1.8.7 First movements

Note: Elevate your XRP on top of a box or other object so that the wheels aren't touching anything and can spin freely.

Before driving the robot around, let's write some simple code to spin one of the wheels. This will help you get familiar with the XRP programming environment and check that your XRP itself is working properly.

Try it out

Create a new file in the IDE called `spin_wheels.py`. Add the following code to it:

```
from XRPLib.defaults import *  
  
left_motor.set_effort(0.5)
```

Run the code and see what happens.

Let's break down the code line by line:

`from XRPLib.defaults import *` tells your robot to load code from **XRPLib**. Don't worry too much about what all the commands in this line mean right now, just know that you'll put this line at the top of most of your XRP programs.

`left_motor.set_effort(0.5)` uses a *function* provided for you in **XRPLib** called `set_effort` that is applied to the left motor. The `0.5` is a *parameter* to this function which tells it that we'd like the motor to apply 50% effort. On the XRP, we write percentages as decimal numbers between 0 and 1, with 1 being 100%.

Now that we've tested the left motor, let's test the right one! How do you think you would modify the code to spin the right motor? Simply replace `left_motor` with `right_motor`.

Try it out

Modify your code and run it on the robot. Make sure the right wheel spins.

Push an object like a pencil against the wheel to add some resistance. Notice how the wheel slows down when you do this, since it would need more effort to keep going the same speed.

1.8.8 Going backwards

We've gotten the wheels spinning forwards, but what if we want to go backwards? To do this, we simply have to pass in a *negative* number for the effort parameter. This means that we can use any number between -1 and 1 for the effort value. -1 will be full effort backwards, 1 will be full effort forwards, and 0 will stop the motor.

Try it out

Try to write code that makes both wheels spin backwards.

This table shows some different effort values and what the wheel would do:

Speed value	Wheel action
1	Wheel spins forwards at 100% effort
0.5	Wheel spins forwards at 50% effort
0	Wheel stops spinning
-0.5	Wheel spins backwards at 50% effort
-1	Wheel spins backwards at 100% effort

1.9 Getting the Robot Moving

1.9.1 Basic driving

In the last lesson we learned how to set the effort of each of your robot's motors individually. Since both of the motors make up the robot's drivetrain, there's an easier way to write code to move the robot.

Note: For this lesson, put your XRP on a flat surface like a table or the floor.

Getting your XRP robot to move is simple! Here is some code you can use to drive both the left and right motors at 50% effort:

Python

```
from XRPLib.defaults import *  
  
drivetrain.set_effort(0.5, 0.5)
```

Blockly



0.5 and 0.5 are the parameters of the function. The functions you used before only had one parameter, but functions can have as few or as many parameters as you want, or even none at all.

Hint: Parameters are inputs to a function that can dictate attributes like distance or angle to vary its behavior.

Try it out

Add the code to your program to see your robot drive.

Try using different values to make the robot move at different speeds. What happens if you use different values for the left and right wheels?

Afterward, place the robot on a ramp and run it again. Take notice of how the robot moves slower when on the ramp. Why does this happen?

You may notice that your XRP does not drive perfectly straight even though you used the same effort value for both motors. This is because the motors on the XRP aren't perfect. Every motor is a little bit different. Some of them have more friction inside them than others. In the next module we'll learn some ways to solve this problem so your robot goes straight every time.

1.10 Driving a Distance

1.10.1 Controlling your speed

In addition to setting the effort of the drivetrain's motors, we can also set their *speed*. Remember, effort is not the same as speed. We can also ask the XRP's motors to go a certain speed. When using this function, the XRP will actively measure the speed of the wheels using the motor's *encoder*. If the speed falls too low, the motor will automatically increase the effort it applies to speed back up.

Tip: Don't worry if you've never heard of an *encoder*. We'll talk more about them later in the lesson.

To set the speed of the drivetrain motors, we use a new function:

Python

```
from XRPLib.defaults import *
drivetrain.set_speed(5, 5)
```

Blockly



This tells the drivetrain to set the speed of each drivetrain wheel to travel at 5 centimeters per second. This means if you put the robot down and let both motors drive at this speed, the robot would move 5 centimeters forwards each second.

Try it out

Add the code to your program and run it. Try the same exercise of pushing something up against the wheels of your XRP. Notice how as you add resistance, the motor will increase its effort to keep the speed constant. When you remove the resistance, the effort will go back down.

Since both wheels are now going the same speed, your robot should now also drive straight, unlike when using the `set_effort` function.

Tip: If you want the robot to go backwards, use a negative speed value just like you did with the effort value.

1.10.2 Driving a distance

We know that we can ask the wheels to spin at a certain speed using a function, but what if we want to make the robot drive a certain distance?

We could ask the robot to move at some speed, and if we know how far it will move each second (for this example we are using a speed of 5 cm/s), we can calculate how many seconds we should drive for to reach that distance.

Let's use d to represent the distance we want to drive in cm. But, we want a number in seconds, so we need to convert by the means of *dimensional analysis*.

To do this, write an expression for the known value with units included:

$$(d \text{ cm})$$

Dimensional analysis involves multiplying this expression by special representations of “1” to convert units. In this case, our speed is 5 cm per second, so we can equate $5 \text{ cm} = 1 \text{ second}$. Rearranging, we have our special representation of 1:

$$\frac{1 \text{ second}}{5 \text{ cm}} = 1$$

We can now multiply our expression with this special representation of 1:

$$(d \text{ cm}) \cdot \frac{1 \text{ second}}{5 \text{ cm}}$$

Cancelling out units and simplifying, we obtain:

$$(d \text{ cm}) \cdot \frac{1 \text{ second}}{5 \text{ cm}} = \frac{d}{5} \text{ seconds}$$

This resultant expression makes sense! If we want to go 5 cm, we plug in $d = 5$, and $\frac{5}{5} = 1$, so we drive for one second. If we want to go 2.5 cm, we plug in $d = 2.5$, and $\frac{2.5}{5} = 0.5$, so we drive for half a second.

Keep in mind that this equation is only valid if the robot is moving at 5 cm per second. If you change that speed to be faster or slower, you'll need to change the denominator of the fraction to that speed to fix the equation.

Try it out

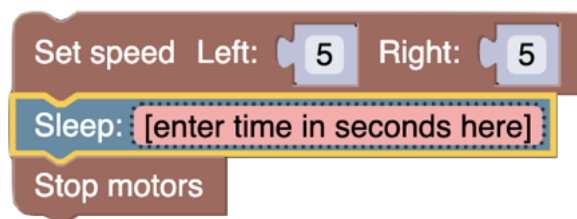
Calculate how many seconds you need to drive for to go one meter if your robot is moving at 5 cm per second. Remember, there are 100 cm in a meter.

To put the above theory into practice, we need to learn about a new function in Python: `sleep`, which makes the XRP wait for some number of seconds before continuing to the next instruction in the code.

Python

```
from XRPLib.defaults import *  
from time import sleep # We need to import the sleep function to use it.  
  
drivetrain.set_speed(5, 5)  
sleep(x) # replace x with the time you calculated to go one meter.  
drivetrain.stop() # This is another function which makes it easy to stop the robot
```

Blockly



Tip: The `#` symbol in Python creates a *comment*. If you add one to a line of code, anything that comes after it on that line will be ignored by the robot. You can use it to leave notes for yourself, or to quickly disable a line of code while debugging problems.

We use comments in our examples to give you hints about how to write your code. You don't need to copy our comments into your code, but you should write your own so that you can easily remember what your code does.

Try it out

Add the code to your program and try it out. Remember to replace `x` with the value you calculated. Try running your robot next to a meter stick to see how accurately your robot drives!

This code you wrote is pretty useful, but what if you wanted to drive other distances?

Let's say that we want to drive three distances in a row: 25, 50, and 75 cm. How could we program the robot to do this? The easy solution is to copy and paste the code you wrote before three times, and modify it each time:

```
from XRPLib.defaults import *
from time import sleep

# Drive 25 cm
drivetrain.set_speed(5, 5)
sleep(25 / 5) # Notice how we can write math directly in our program!
drivetrain.stop()

# Drive 50 cm
drivetrain.set_speed(5, 5)
sleep(50 / 5)
drivetrain.stop()

# Drive 75 cm
drivetrain.set_speed(5, 5)
sleep(75 / 5)
drivetrain.stop()
```

This looks pretty repetitive. Most of this code is exactly the same. In fact, the only change between each block is the parameter we are passing to the `sleep` function. This is a perfect example of why we have functions. Let's write our own function to drive the robot a certain distance.

Python

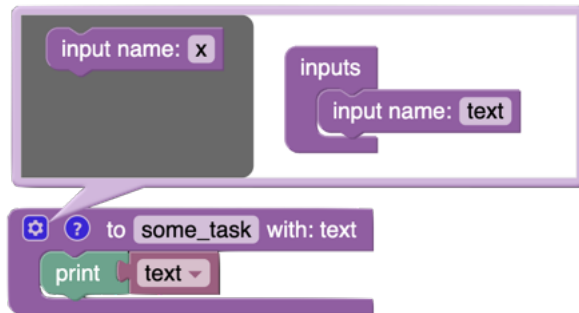
Python uses the keyword `def` to let you, the programmer, tell it that you would like to *define* a new function. A full function definition looks like this:

```
def function_name(parameter1, parameter2, parameter3):
    # put your code here
    # code in your function can use the parameters by name like this:
    print(parameter1 / 5)
```

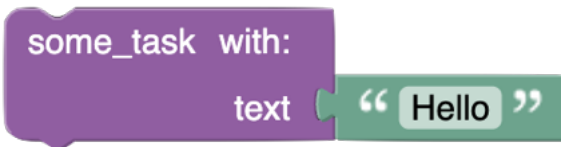
In this example function, there are three parameters. Functions can have as many or as few parameters as you want, or even have no parameters at all.

Blockly

In Blockly, you create functions by dragging a block that looks like the picture below. The interface allows you to specify the function name, and pass *parameters* to the function body. Here, we have a function called `some_task` (which you should rename based on what your function does) that takes in a parameter called `text`, and uses `prints` the `text` value. Functions can have as many or as few parameters as you want, or even have no parameters at all.



The below blocks *calls* the function we defined above to run it. The value “Hello” is passed to the `text` parameter, which results in “Hello” being printed to the console.



Try it out

Define a function called `drive_distance` that takes in one parameter: `distance_to_drive`. Use the parameter in your function as the numerator of your fraction.

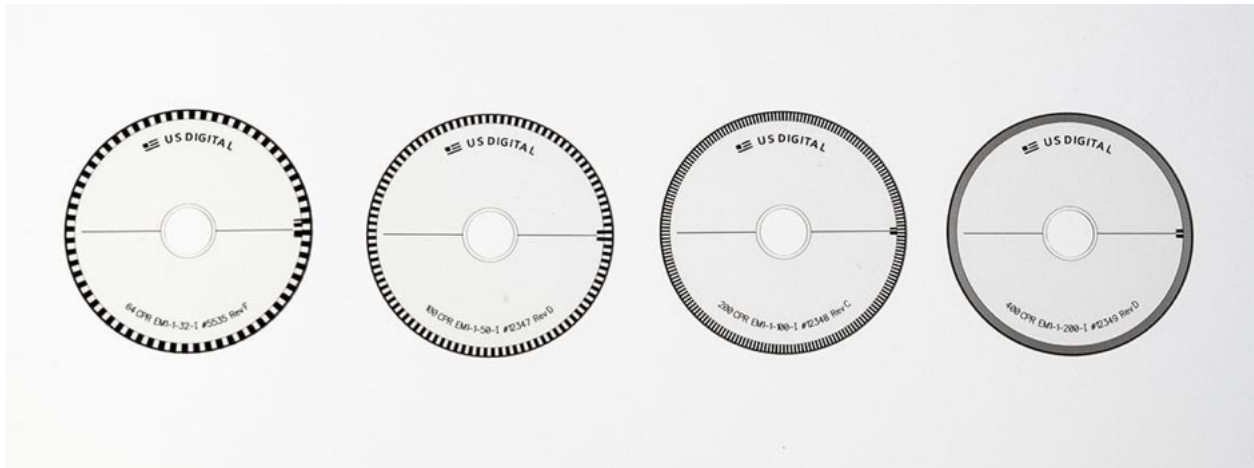
Use your function to make the robot drive 3 distances in a row.

Tip: Define your functions towards the top of your file, underneath the `import` statements. This way, code later in the file will be able to use them.

1.11 The Encoders

In the last lesson we mentioned *encoders*. What are they and what do they do?

Encoders are sensors which measure how far each motor (and thus each wheel) has rotated. We mentioned that our motors aren’t perfect, so when we tell them to go a certain effort we don’t know how fast it is actually rotating. Encoders measure exactly what the motor is doing and report this information back to the XRP.



An encoder uses a disk like the one above with alternating clear and black squares. The disk is attached to the output shaft of the motor. A small light is shined through the edge of the disk, and a sensor on the other side sees the light. When a black part of the disk is in front of the light, the sensor sees nothing. When a clear part of the disk is in front of the light, the sensor can see the light. As the disk rotates, the sensor constantly switches from seeing and not seeing the light. The XRP can automatically count how many times it has switched between seeing and not seeing the light, and can use this to calculate how far the wheel has moved.

The size of the black and clear squares determines how *precise* the encoder is. As the squares get smaller, the light switches more times for the same amount of rotations of the wheel, and thus we can more precisely measure the distance. The downside of having really tiny squares is that the XRP's processor needs to work harder to keep up with counting all the switches. In the image above you can see some disks with different levels of precision.

Tip: The disk and sensor in the XRP is hidden inside the motor's plastic case, so you won't be able to see them.

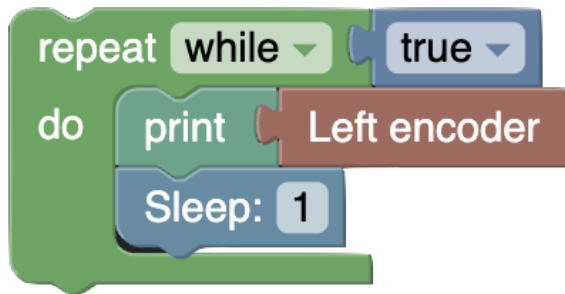
The XRP handles doing all the math to convert these switches into a real distance measurement for you, so you don't have to worry about it. We can use some new `drivetrain` functions to see what the encoders are measuring:

Python

```
from XRPLib.defaults import *
from time import sleep

while True:
    print(drivetrain.get_left_encoder_position())
    sleep(1)
```

Blockly



This code uses something new: a *while loop*. A *while* loop will run the code underneath it until a *condition* is no longer equal to `True`. In this case, the *condition* is the keyword `True`, which means that this code will run until `True` doesn't equal `True`. Since this will never happen, the code will run forever unless you stop it manually on your computer.

Tip: You will learn more about loops and conditions later in the course.

The code should run forever, display the drivetrain's left encoder position, and then wait for one second. Then, it repeats and does this forever. If we didn't wait for one second, the XRP would read and send the encoder position to your computer as fast as it could (very fast!) which would be too much data for your computer to display at once on the screen.

Try it out

Try running this code to see what happens. Spin the left wheel of the XRP by hand and notice how the number changes.

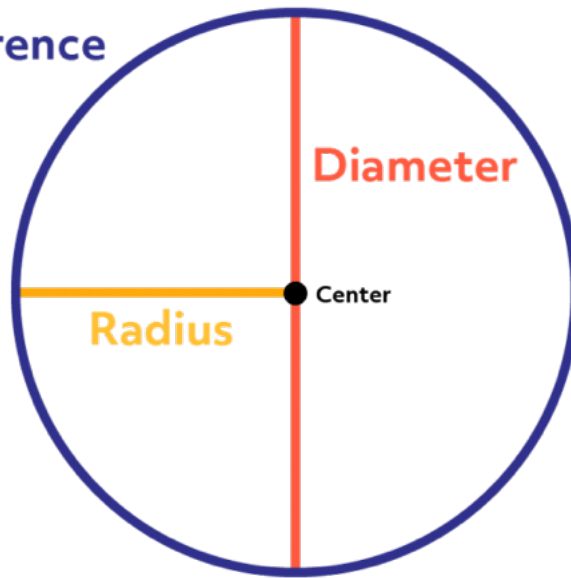
1.11.1 Calculating distance

Let's learn a bit about how the XRP uses the encoder to calculate how far the robot has moved.

The XRP knows the diameter of the robot's wheels; every XRP has the same wheels!

If a car's wheel rolls on the ground one full revolution, how far does the car move? The car moves by one *circumference* of the circle:

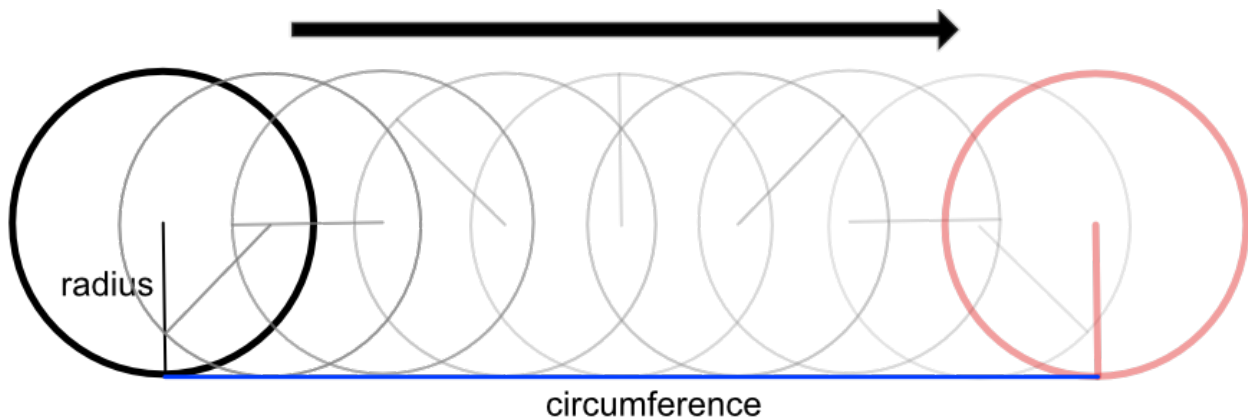
Circumference



Tip: Reminder that the circumference of a circle is the distance around it.

The circumference is calculated as $C = \pi \cdot d$, where d is the *diameter* of the circle.

If a car wheel (which is a circle with a circumference of 100 inches) rotates 5 times, how far does the car go? How would you find that?



You would rotate the wheel once, and find that it has traveled 100 inches, because the wheel would trace out its circumference on the ground. Then, you would rotate it a second time, and see it move another 100 inches. Then a third, a fourth and a fifth time, and see the wheel has traced out its circumference on the ground 5 times.

The amount it has traveled is 5 times the circumference. This can be used for any number of rotations. If you rotate the wheel 3 times, you would move forward 3 times the circumference (300 inches), if you rotated it 1 and a half times, you would move forward one and a half times the circumference (150 inches).

$$d \text{ cm} = (\text{number of rotations}) \cdot (\text{circumference})$$

The XRP uses this equation to automatically calculate how far the wheels have moved in centimeters using the encoders.

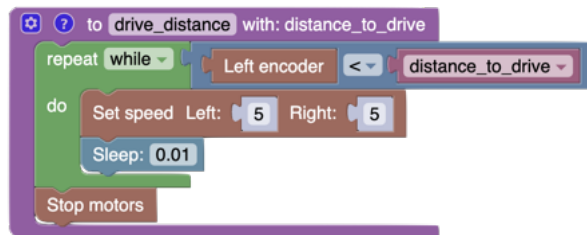
1.11.2 Driving a distance (again)

In the last lesson you used a constant speed and a time to drive a distance. Now that you know that the encoders actually measure the distance, it would be better to use them for your `drive_distance` function. We can modify your function to use a `While` loop:

Python

```
def drive_distance(distance_to_drive):
    while drivetrain.get_left_encoder_position() < distance_to_drive:
        drivetrain.set_speed(5, 5)
        time.sleep(0.01)
    drivetrain.stop()
```

Blockly



This code block uses a loop to constantly check if the left encoder position is less than the distance you want the robot to go. Once it is no longer less than this distance, the loop stops running and the code moves on to the next line. In this case, the next line tells the robot to stop.

Try it out

Replace your `drive_distance` function with this new one. Try it out next to a meter stick. Is it more or less accurate than before?

Challenge

This code only checks the left encoder. Since both wheels are moving the same speed, this *should* be fine, but as we said, the motors aren't perfect. Can you think of a way to combine both encoder values together?

To read the right encoder, you use `drivetrain.get_right_encoder_position()`

1.11.3 Turning to a heading

Once you know how to drive a certain distance with the XRP, it is easy to turn to a certain heading with it. First, you need to calculate the distance that a wheel must travel so that you are facing the correct heading, and then simply rotate the XRP until the encoders have traveled that distance.

Calculating the necessary distance is complicated, but we can break down this problem into steps.

first, lets make a fraction that represents from 0.0 to 1.0 how far around the robot's circumference the wheels need to travel. In this case 0.0 is 0 degrees and 1.0 is 360 degrees.

Now to get the distance the wheel travels, we need to multiply this fraction by the total distance the wheel travels to rotate 360 degrees, this number is the circumference of a circle with the diameter same diameter as the robot. We can calculate this by multiplying the wheel track distance by pi.

finally, to get the number of wheel rotations, we need to divide this distance by the circumference of the wheel, or pi times the wheel diameter. We can cancel pi from both sides of this division and that leaves us with.

$$\frac{\text{target degrees} \cdot \text{robot wheel track}}{360 \cdot \text{wheel diameter}}$$

Now that we have the number of wheel rotations, the rest of the program is easy. just turn the robot in the direction of the turn, and stop once the number of rotations has exceeded the calculated rotation goal.

Python

```
def turn(target):
    global rotations
    differentialDrive.reset_encoder_position()
    rotations = (target * 15.5) / (360 * 6)
    if target > 0:
        differentialDrive.set_effort((-0.3), 0.3)
    else:
        differentialDrive.set_effort(0.3, (-0.3))
    while not math.fabs(motor1.get_position()) >= math.fabs(rotations):
        differentialDrive.stop()
```

Blockly



1.12 Helpful Drivetrain Functions

Throughout this module, we've explored different ways to drive forwards and turn, through setting efforts, speeds, and reading the encoders. However, XRPLib also provides some handy functions to make it easy for the user. These functions use more complicated calculations to ensure that the XRP drives smoothly and exactly to the right distance every time.

Python

```
drivetrain.straight(20, max_effort = 0.5, timeout = None)
```

Blockly



This function will drive the robot straight forward for a distance in centimeters that you specify. The other two parameters are *optional*. You can provide a value for them, but if you don't, the default value will be used.

Calling the function like this: `drivetrain.straight(20)` will make the robot go straight 20 centimeters, and use the default values for everything else, meaning a maximum effort applied of 50% and no timeout.

You can also use a negative value for distance to drive backwards.

The `max_effort` parameter specifies how much effort the robot is allowed to apply while driving. By default it is 50%, which is a good effort for normal driving on a flat surface.

The `timeout` parameter specifies a time, in seconds, that the robot should try to drive before giving up. For example, what if your robot runs into something while driving, and the wheels get stuck? The robot will use the encoders to measure the wheels and notice that it never arrived at the distance you set, so it will try forever and none of your code will run afterwards. The timeout lets you set a maximum time that the XRP should try for before giving up. Usually, you won't need to use this, but it is there if you need it.

Python

```
drivetrain.turn(90, max_effort = 0.5, timeout = None)
```

Blockly



This function is similar to the `straight` function, except that it rotates the robot instead of driving it forwards.

The `turn_degrees` parameter lets you tell the robot how many degrees it should turn. Positive values will turn counterclockwise, and negative values turn clockwise.

Try it out

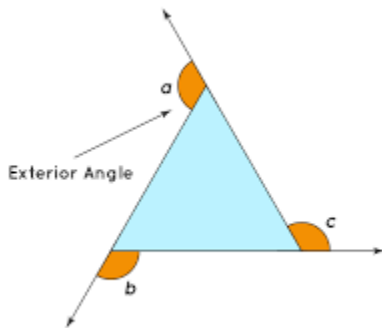
Write code to drive the robot straight for 20 centimeters and then turn 90 degrees clockwise. Don't forget to add the `from XRPLib.defaults import *` statement at the top of your program.

1.13 Driving With Geometry

The XRP's driving and turning functions can also be used to draw geometric patterns! By driving a set distance and turning by a certain angle several times, you can draw a shape with as many sides as you want.

Tip: To see the path your XRP has driven, you can place a dry-erase marker between the robot's wheels and trace your pattern on a whiteboard.

To draw any polygon, you just need to know the size of the shape's exterior angles, and you need to decide how long the sides should be. Then, you just have to drive straight with the distance of a side length and turn the number of degrees of the corresponding exterior angle. Repeat this until the shape is finished.



1.13.1 Triangle

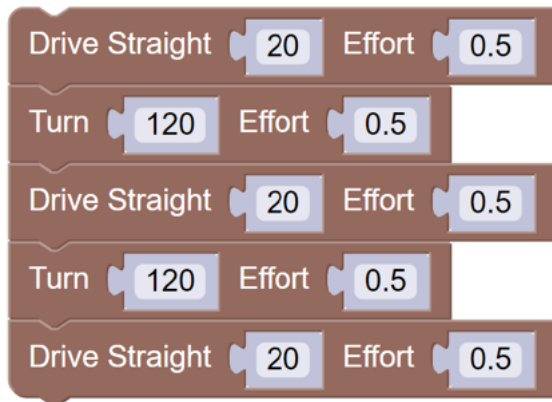
For a triangle, the interior angles measure 60 degrees and the exterior angles measure 120 degrees. We can trace a triangle with side lengths of 30 cm by having our robot drive straight for 30 cm three times and turn 120 degrees in between each drive straight.

Python

```
from XRPLib.defaults import *

differentialDrive.straight(30, 0.5)
differentialDrive.turn(120, 0.5)
differentialDrive.straight(30, 0.5)
differentialDrive.turn(120, 0.5)
differentialDrive.straight(30, 0.5)
```

Blockly



1.13.2 Square

The process for tracing a square is similar to tracing a triangle. The interior angles are 90 degrees so the exterior angles are also 90 degrees. Keeping a side length of 30 cm, we can trace a square by programming our robot to drive straight for 30 cm four times and turn 90 degrees between each drive straight.

Python

```
from XRPLib.defaults import *  
  
drivetrain.straight(30, 0.5)  
drivetrain.turn(90, 0.5)  
drivetrain.straight(30, 0.5)  
drivetrain.turn(90, 0.5)  
drivetrain.straight(30, 0.5)  
drivetrain.turn(90, 0.5)  
drivetrain.straight(30, 0.5)
```

Blockly



1.13.3 Polygons

We can generalize the procedure used to make a triangle and square to make any regular polygon; you just have to determine the exterior angle and choose a side length. However, for polygons with many sides, this process can get very tedious. Instead of repeating the same code multiple times, we can use a function to simplify the process.

First, let's determine what information the function needs. To trace a polygon, you need to determine the number of sides and the length of each side. We can create a function that takes these two values as an input. The function will drive the distance we give it, turn by the exterior angle, and then repeat that process as many times as there are sides in the shape. We can use a loop for this. The one problem is: how do we know the measure of the exterior angles? Fortunately, this can be easily calculated with this equation:

$$360/n$$

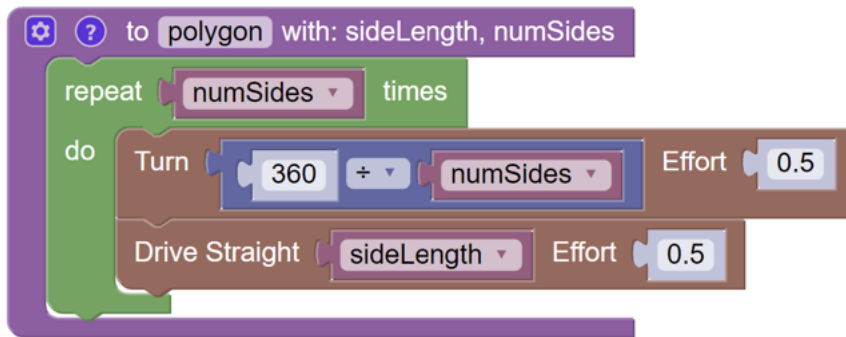
With the variable n representing the number of sides of your polygon, this equation determines the number of degrees of your polygon's exterior angles. With this information, you can now write a function to trace any regular polygon!

Python

```
from XRPLib.defaults import *

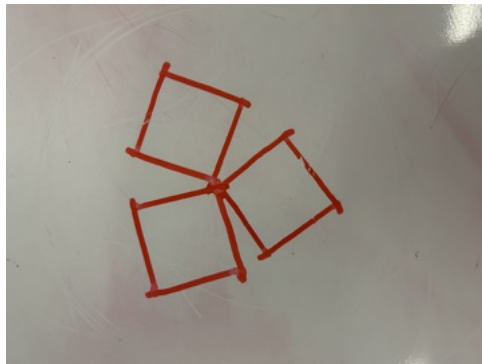
def polygon(sideLength, numSides):
    for i in range(int(numSides)):
        differentialDrive.turn((360 / numSides), 0.5)
        differentialDrive.straight(sideLength, 0.5)
```

Blockly



1.13.4 Pinwheel

Now we know how to easily draw any polygon, but we can take it one step further and draw a polygon pinwheel. This pattern consists of several polygons extending out from a center point. Your XRP can execute this by tracing several polygons consecutively and turning slightly between each new polygon. A pinwheel of 3 squares should look something like this:



Programming this may seem like a daunting task, but it is actually quite simple. Every time you want to trace a piece of the pinwheel, you just need to call your polygon function from before and then turn your robot slightly. We can calculate the measure of this turn by dividing 360 degrees by the number of polygons we are tracing in order to keep even spacing between each polygon. Repeat this process as many times as there are polygons in the pinwheel, and your pattern will be finished!

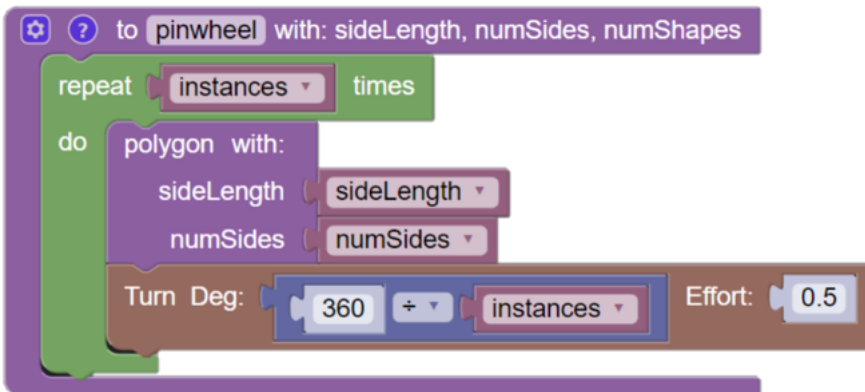
Python

```

from XRPLib.defaults import *

def pinwheel(sideLength, numSides, numShapes):
    for i in range(numShapes):
        polygon(sideLength, numSides)
        drivetrain.turn(360 / numShapes, 0.5)
  
```


Blockly



1.14 Waiting for Button Input

You may have noticed that your code runs immediately after uploading it. This is nice sometimes, but sometimes you aren't coding in the same place you will be running your code, and the robot suddenly driving itself off the table isn't an ideal result. In order to have the code run on command, we can use the on board buttons to tell the code when to run.

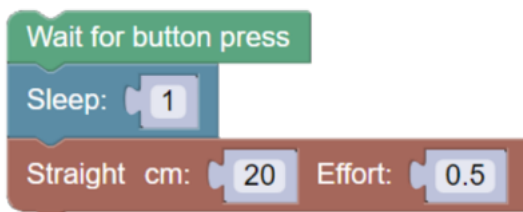
The XRP has a button which you can read from code. To make it easy, **XRPLib** has a built in function which will wait for the button to be pressed for you.

Python

```
from XRPLib.defaults import *
from time import sleep

board.wait_for_button()
sleep(1)
drivetrain.straight(20)
```

Blockly



This function is part of `board` since the button is on the XRP's main controller board.

This code will wait until the button is pressed, and then wait an additional 1 second (for you to get your finger out of the way) and then start driving.

There is also a function which lets you read the current state of the button without waiting for it:

```
board.is_button_pressed()
```

You could use this function as the *condition* for a *while* loop if you wanted to do something more complicated with the button than just waiting for it.

1.15 Parking Challenge

1.15.1 Challenge Description

This challenge features a sequence of turns that the robot must perform in order to get to the “end” of the Labyrinth. The robot must first begin at the starting point, and get to the goal area by completing turning and forward movement behaviors.

1.15.2 Materials needed

- White board or six 8.5”x11” sheets of paper (use scotch tape to tape the six sheets of paper together to form a 25.5”x22” rectangle. Turn it over and tape outside border to floor with blue masking tape.
- Black electrical tape (you can substitute blue masking tape for this exercise)
- Red electrical tape or red marker
- Scissors (or cutting tool)
- Ruler (or straightedge)

1.15.3 Project specifications

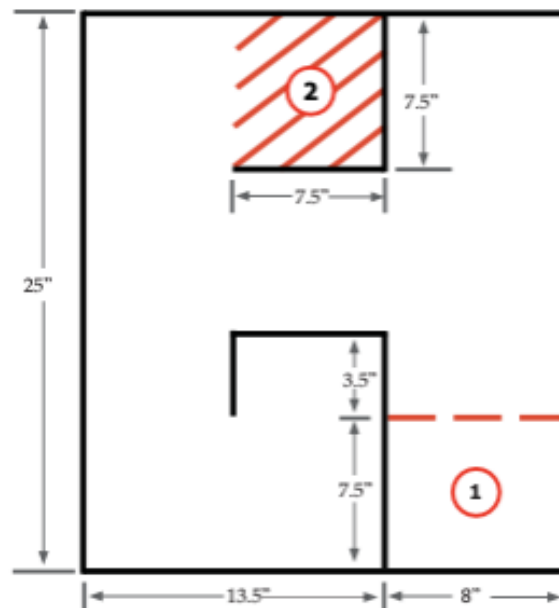


Fig. 2: Carnegie Mellon Robotics Academy, Labyrinth Challenge

The robot should do the following:

- Begin fully contained in position 1 and then maneuver to goal area.
- Reach and be fully contained within position 2 without crossing any black lines.

1.16 Advanced: Circles and Differential Steering

In this section, we derive the math needed to drive in circles of any radius. Fundamentally, this requires driving the two motors at different ratio, but it takes a little calculation to figure out this ratio for a given radius.

<https://youtu.be/kd2-mhI2CgE>

The robot goes in an arc. The wheels draw out 2 different circles with 2 different radii. The arc with the smaller radius (r_1 , the white line) has a smaller circumference. That means the left wheel has driven a smaller distance. It has gone slower.

1.16.1 How would you drive in an arc with a given radius?

We know the wheels trace out arcs when we make the wheels go at different speeds, but how do we decide what the wheel speeds should be?

1.16.2 What do we know?

Let us start with what we know about the two arcs, and the arc we want the center of the robot to go through.

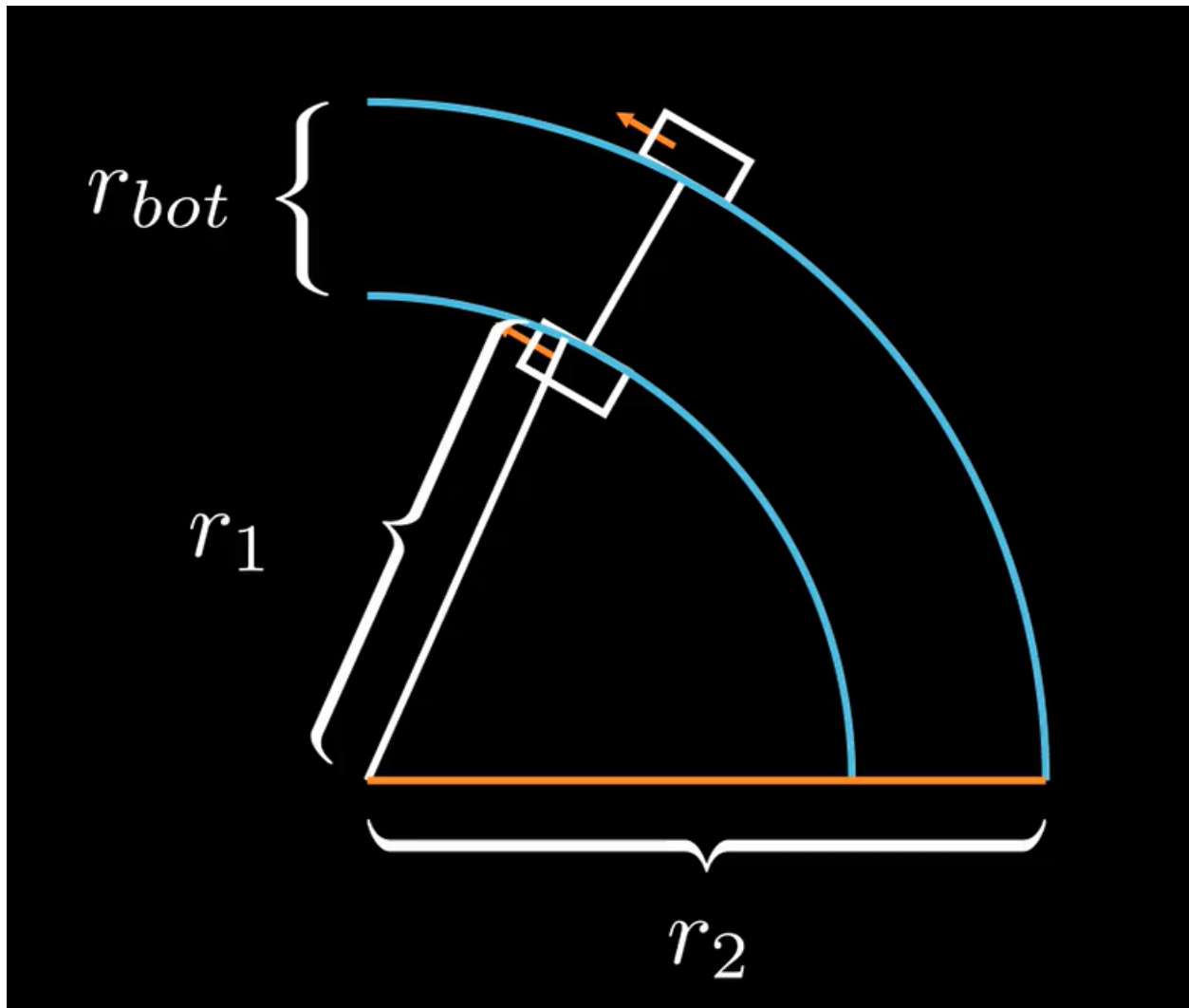
The first thing we can say about any circle is that the radius is proportional to the circumference. This means that the inner and outer arc lengths are proportional to the inner and outer radii, or

$$\frac{\text{left wheel distance}}{\text{right wheel distance}} = \frac{r_1}{r_2}$$

This means that the *ratio* between the wheel speeds is equal to $\frac{r_1}{r_2}$.

Hey! We found out what the ratio between the wheel speeds is supposed to be if we know the two radii. But how do we find what the two radii are? We only know what we want the radius of the orange circle to be.

But we know what the distance between the two wheels are.



If r_{bot} is the distance between the two wheels, then

$$r_1 = r_{desired} - \frac{r_{bot}}{2}$$

since r_1 is less than the desired radius, and

$$r_2 = r_{desired} + \frac{r_{bot}}{2}$$

We found out the radii of the circles we want the left and right wheels to trace, and we know how to find the ratio between the wheel speeds from that.

Remember:

$$\text{ratio between wheel speeds} = \frac{r_1}{r_2}$$

Putting them together, we get,

$$\text{ratio between wheel speeds} = \frac{r_{desired} - \frac{r_{bot}}{2}}{r_{desired} + \frac{r_{bot}}{2}}$$

Try it out

Write code to make the robot drive in a circle. Use the `set_speed` function, and use a speed of 5 cm per second for the left wheel speed. Use the ratio you calculated to find out what the right wheel speed should be (multiply 5 by the ratio)

Now try using 5 for the right wheel speed, and use the ratio for the left wheel. What happened?

1.16.3 Additional challenges

Try to make the robot do a point turn! What is the radius of the circle that you want it to drive in then?

Try and make the robot turn around one of the wheels. What is the new desired radius?

Try making the robot drive backwards in an arc

1.17 Measuring Distances: Module Overview

In this module students will:

- Understand the basics of measuring distance and its application
- Learn to use distance sensors to determine the robot's location with respect to the target.
- Use reading from the sensor to avoid obstacles or stop before the object
- Locate and align to objects using the distance sensor

1.17.1 Covered Standards (NGSS and CSTA):

HS-ETS1-2 Break a complex real-world problem into smaller, more manageable problems that each can be solved using scientific and engineering principles.

3A-AP-16 Design and iteratively develop computational artifacts for practical intent, personal expression, or to address a societal issue by using events to initiate instructions.

3A-AP-17 Decompose problems into smaller components through systematic analysis, using constructs such as procedures, modules, and/or objects.

3B-CS-02 Illustrate ways computing systems implement logic, input, and output through hardware components.

3B-AP-10 Use and adapt classic algorithms to solve computational problems.

3B-AP-11 Evaluate algorithms in terms of their efficiency, correctness, and clarity.

3B-AP-15 Analyze a large-scale computational problem and identify generalizable patterns that can be applied to a solution.

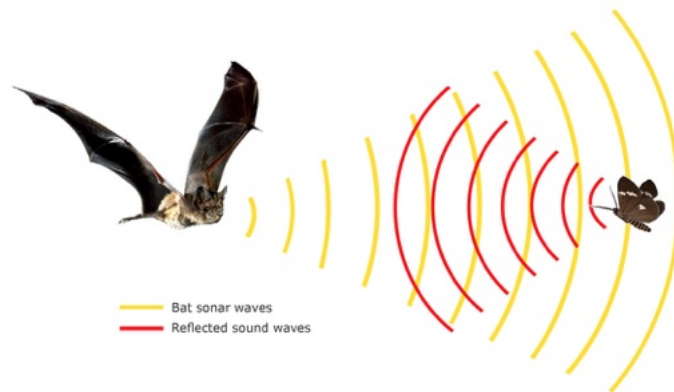
3B-AP-16 Demonstrate code reuse by creating programming solutions using libraries and APIs.

1.18 Measuring Distances

1.18.1 Animal Echolocation

Information about our environment helps our robots perform complex tasks such as obstacle avoidance, navigation, and path planning.

Similarly, animals like bats have evolved to use processes like echolocation which allow them to navigate through dark caves and find food where animals like bats emit high-frequency sound waves using their mouths. They listen to the echo of the sound waves bouncing back from the environment with their highly sensitive ears, allowing them to determine the size, shape, and texture of objects.



© Copyright, 2019, The University of Waikato - Te Whare Wānanga o Waikato. All rights reserved.
www.sciencelearn.org.nz

1.18.2 Robotic Echolocation

Just like bats, robots have evolved to have their own distance sensors which can give the robot more information about where it is relative to its environment. This allows the robot to make more informed decisions about how to navigate its environment.

So, what are some sensors that allow you to measure distances?

Reflective sensors

Lidar (Light Detection and Ranging), Sonar (Sound Navigation and Ranging), and Radar (Radio Detection and Ranging) all follow the same principle. These sensors all have one transmitter that “sends” a signal and a receiver that “listens” for the signal that was sent out.

One common example is an ultrasonic range finder, which emits sound waves and listens for the echo. The sensor then calculates the distance based on the time it takes for the sound wave to bounce back.

Tip: Since an ultrasonic range finder assumes that the sound waves will bounce back from a flat wall, pointing the range finder at an angled surface can lead to inaccurate readings. Try pointing the range finder at different surfaces and see how the readings change.

How would you use this sensor to detect the distance to a wall that’s angled? What information could you use about your environment to infer distances from angled/curved surfaces?

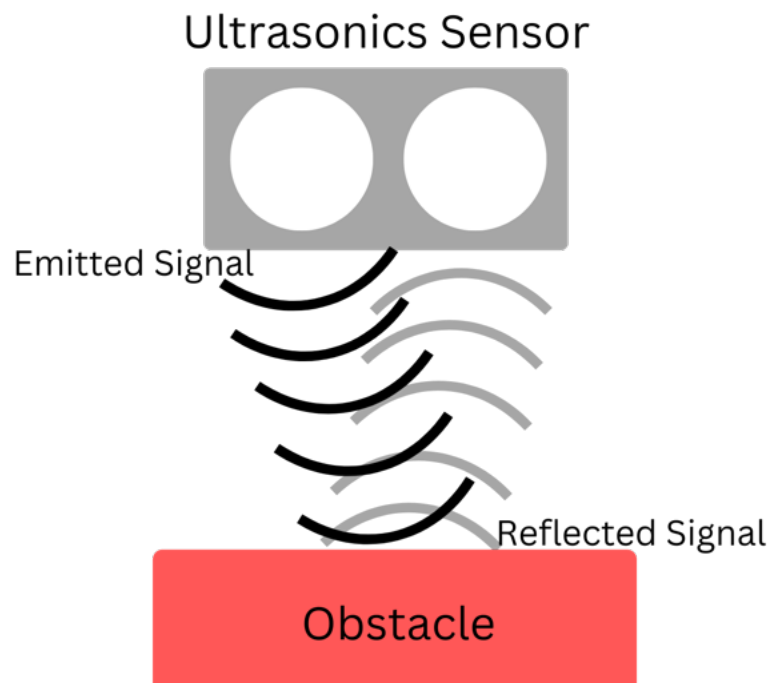
Mechanical Sensors

Another type of sensor that can directly inform the robot about distance are mechanical sensors such as limit switches. These sensors operate by sensing touching an object and can be used to detect when a robot has reached a certain point in its environment.

These sensors work by completing a circuit, either using magnets or physical force.

Completion of the circuit informs the robot that the actuator has reached a particular position. For example, a limit switch can be used to detect when a robot has reached the end of a track.

Using the Ultrasonic Sensor



While using the XRP, your distance sensor will be an ultrasonic range finder. Here is the method call to get the distance from the sensor:

Python

```
rangefinder.distance()
```

Blockly



This function returns the distance, in cm, from the sensor to the nearest object.

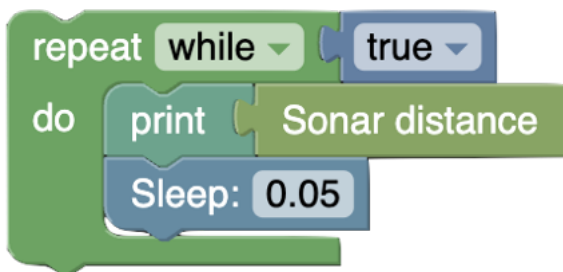
Note: Try it out! Try writing code that checks the distance every 50 ms (0.05 seconds) and prints the output.

Here's the answer:

Python

```
from XRPLib.defaults import *  
import time  
  
while True:  
    print(rangefinder.distance())  
    time.sleep(0.05)
```

Blockly



1.19 Obstacle Avoidance

One useful application of the ultrasonic sensor is obstacle avoidance.

In this tutorial, we will learn how to use the ultrasonic sensor to first stop at a certain distance from an object, and then to avoid the object by turning a random angle away from an object.

1.19.1 Step 1: Going forward a certain distance

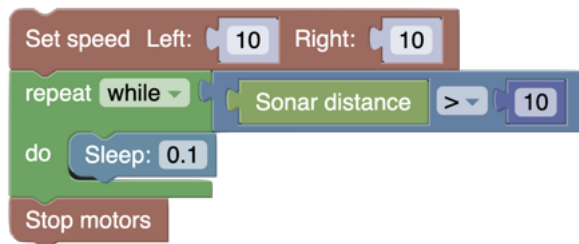
The first step in obstacle avoidance is stopping at a certain distance from an object. To do this, we want to continuously read the rangefinder distance and check whether it is less than, let's say, 10 cm. Once it crosses this threshold, we want to stop the robot.

To accomplish this, we can use a while loop with a condition that checks whether the rangefinder distance is less than 10 cm.

Python

```
drivetrain.set_speed(10, 10)
while rangefinder.distance() > 10:
    time.sleep(0.1)
drivetrain.stop()
```

Blockly



1.19.2 Step 2: Turing 180 degrees once an object is detected

Instead of simply stopping, we'd like to turn around 180 degrees, go forward, and repeat, turning 180 whenever we detect an object.

To turn 180 degrees, we'd want to replace `drivetrain.stop()` with `drivetrain.turn(180)`. After this, we'd want to go forward again. But instead of writing `drivetrain.set_speed(10, 10)` again, notice that we're just trying to run these two steps over and over:

1. Go forward until an object is detected
2. Turn 180 degrees

It looks like we can wrap these two steps in a while loop! Here's what the code looks like:

Python

```
# Repeat these two steps over and over again
while True:

    # Go forward until an object is detected
    drivetrain.set_speed(10, 10)
    while rangefinder.distance() > 10:
        time.sleep(0.1)

    # Turn 180 degrees
    drivetrain.turn(180)
```

Blockly



1.19.3 Step 3: Turing a random angle once an object is detected

Even though we're turning around after detecting an object, you should notice that your robot is getting stuck in a cycle. Because the robot is turning 180 degrees, it often turns back into the object it just detected. To fix this, many robots like iRobot's Roomba use a simple algorithm known as "bump and run." If you bump into an object, instead of turning 180 degrees, the robot should turn away from it at a random angle to increase the chance it'll explore a new area.

However, if the robot were to turn to a completely random angle, there would be a chance the robot barely turns at all if the random number is small. So, we'd want to give the robot a reasonable random range of angles to pick from.

Python

We can use `random.randint(135, 225)` to generate a random number between 135 and 225, which we can turn that many degrees. Though, note that we need to `import random` at the top of our program to import the library that contains this function.

```
# the library that contains random.randint
import random

# Repeat these two steps over and over again
while True:

    # Go forward until an object is detected
    drivetrain.set_speed(10, 10)
    while rangefinder.distance() > 10:
```

(continues on next page)

(continued from previous page)

```

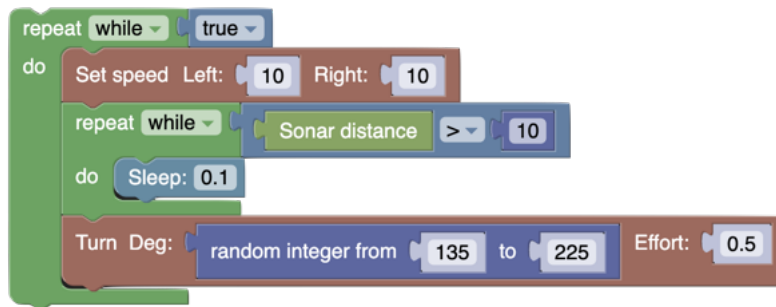
time.sleep(0.1)

# Turn random amount between 135 and 225 degrees
turnDegrees = random.randint(135, 225)
drivetrain.turn(turnDegrees)

```

Blockly

Blockly provides a handy block for generating a random number between lower and upper bounds, inclusive.



And voi la! We have successfully created a program where our robot can avoid objects forever!

1.20 Locating a Nearby Object

Another way to utilize the ultrasonic rangefinder is to use it to locate a nearby object.

Try it out

What's the most effective way to locate an object? Try it out!

1.20.1 Turn and Detect

To first detect an object, the robot can slowly spin in a circle while continuously polling the rangefinder. Then, when an object is detected, it can stop spinning, and head towards the object.

How can we tell when an object is detected? Imagine that the robot is in the center of an empty room, and a random object is placed somewhere near the robot. The rangefinder would be giving large distance readings, until it reaches the object, at which point the distance reading would drop. It's the *change* in distance readings that hints that an object has been detected.

How can we find the change in distance readings over each iteration of the loop? We can store the previous distance reading in a variable, and compare it to the current distance reading. If this change is greater than some threshold, then we can assume that an object has been detected.

To code this, we can start by setting the drive motor speeds to spin in opposite directions to start spinning the robot in place. Then, once the change in distance reading is greater than the threshold, the robot can stop spinning and head towards the object.

In the following example code, we use a change threshold of 30 cm.

Python

```

changeThreshold = 30 # distance change in cm needed to trigger detection

# store initial value for current distance
currentDistance = rangefinder.distance()

# start spinning in place until an object is detected
drivetrain.set_speed(5, -5)

while True: # doesn't actually repeat forever. loop will be broken if an object is
    ↪ detected

    # update previous and current distance
    previousDistance = currentDistance
    currentDistance = rangefinder.distance()

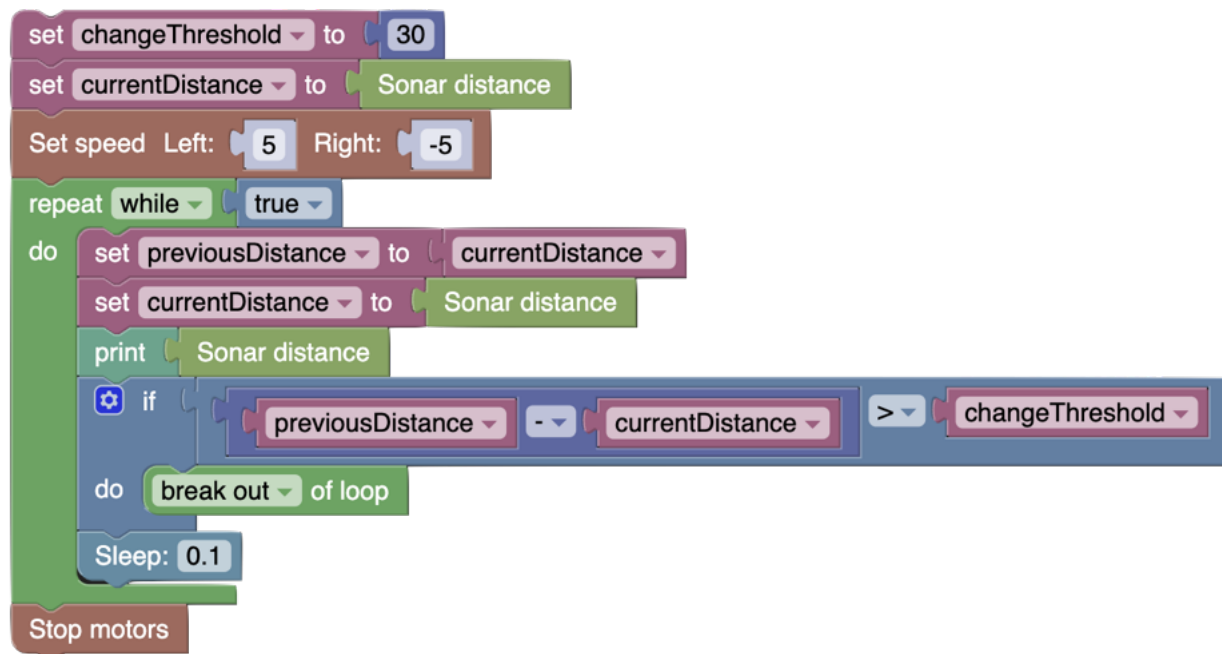
    # if sudden decrease in distance, then an object has been detected
    if previousDistance - currentDistance > changeThreshold:
        break # break out of the while loop

    time.sleep(0.1)

# stop spinning drive motors
drivetrain.stop()

```

Blockly



1.20.2 Improving Accuracy

Do you see any issues with this solution?

When the rangefinder distance dips below the threshold, the implication isn't that the robot has found an object, but rather that the robot has found its *edge*. So, the robot will aim for the edge of the object, rather than the center.

Instead, the robot should remember the heading it faces when detecting *both* edges. Then, the robot can aim for the center between those edges, and thus the center of the object. We can store each edge angle in variables, naming them `firstAngle` and `secondAngle`.

We're already quite familiar with turning until an edge is detected. Now, we'll need to detect *both* edges. However, it would be quite unwieldy and error-prone to just copy the edge detection code, so let's make a function to generalize this. Note that the existing code detects a sudden *decrease* in distance, but we want to handle sudden *increases* in distances too.

How can we support both behaviors in a single function? We can pass in a parameter to specify whether we want to detect an increase or decrease in distance! We can call this parameter `isIncrease` and pass in a boolean (true or false) value.

If `increase` is `True`, then we want to detect an increase in distance, which is when `currentDistance - previousDistance > changeThreshold`.

If `increase` is `False`, then we want to detect a decrease in distance, which is when `previousDistance - currentDistance > changeThreshold`.

For more flexibility, let's add a parameter for the change threshold.

Here's the function definition:

Python

```
def turnUntilEdge(isIncrease, changeThreshold):

    # store initial value for current distance
    currentDistance = rangefinder.distance()

    # start spinning in place until an object is detected
    drivetrain.set_speed(5, -5)

    while True: # doesn't actually repeat forever. loop will be broken if an object is_
        ↪ detected

        # update previous and current distance
        previousDistance = currentDistance
        currentDistance = rangefinder.distance()

        if isIncrease and currentDistance - previousDistance > changeThreshold:
            # if sudden increase in distance, then an object has been detected
            break
        elif not isIncrease and previousDistance - currentDistance > changeThreshold:
            # if sudden decrease in distance, then an object has been detected
            break

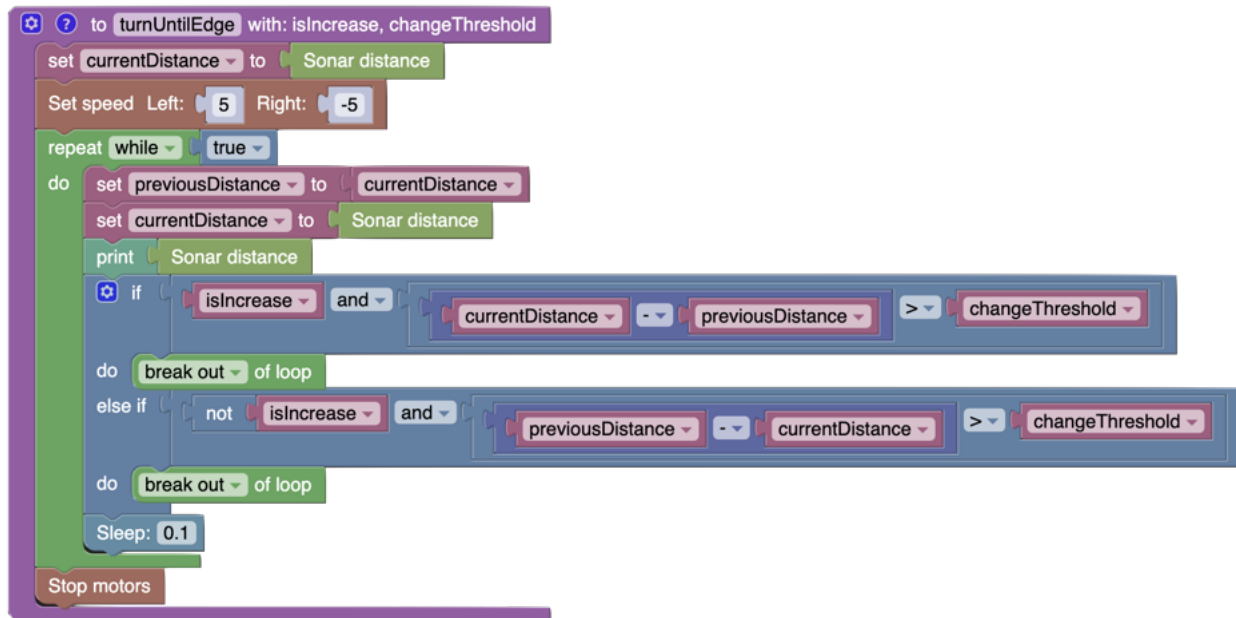
        time.sleep(0.1)
```

(continues on next page)

(continued from previous page)

```
# stop spinning drive motors
drivetrain.stop()
```

Blockly

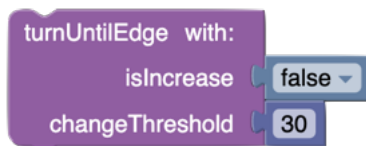


Here's the equivalent function call to the turn and detection code in the previous section:

Python

```
turnUntilEdge(False, 30)
```

Blockly



Now, it's time to write the full program to detect both edges and turn to the center.

1.20.3 Implementing Dual Edge Detection

Let's walk through each step of the process in code.

First, the robot should spin in place until it detects the first edge, then stop. This is simply the function call we saw earlier.

Python

```
turnUntilEdge(False, 30)
```

Blockly



Next, we want to record the robot's heading for this first edge, and store it to `firstAngle`.

Python

```
firstAngle = imu.get_yaw()
```

Blockly

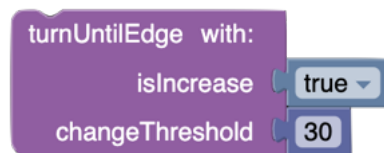


Then, the robot should spin in place again until it detects the second edge, which is when there is a sudden increase in distance.

Python

```
turnUntilEdge(True, 30)
```

Blockly



Once the robot has detected the second edge, it should record its heading and store it to `secondAngle`. Now, need to figure out how much the robot needs to backtrack to aim for the center of the object. We can do this by finding the

difference between the two angles, and dividing by two. This is half the angle between the two edges, and if the robot backtracks by this amount, it will be facing the center of the object. Let's store this in a variable called `angleToTurn`.

Python

```
secondAngle = imu.get_yaw()
angleToTurn = (firstAngle - secondAngle) / 2
```

Blockly



Finally, the robot can turn this much to face the center of the object, and head towards it.

Here's the full code. Note that half-second pauses are added to make the robot's actions more visible:

Python

```
# turn to first edge
turnUntilEdge(False, 30)

# store angle at first edge
firstAngle = imu.get_yaw()

time.sleep(0.5)

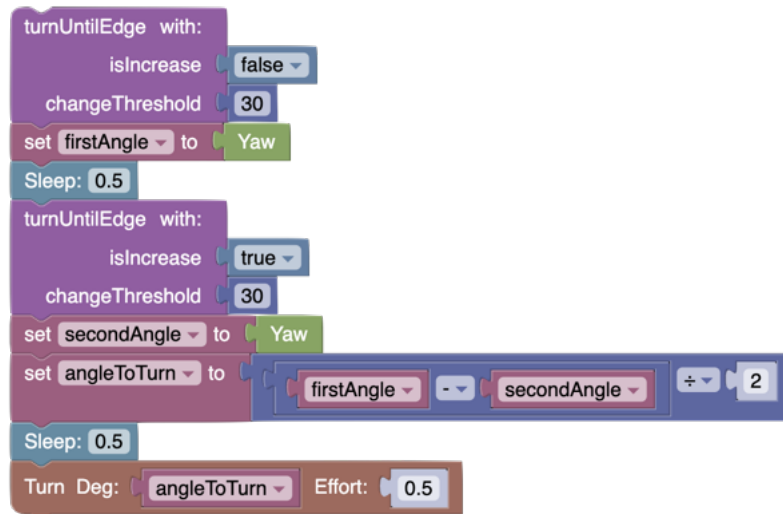
# turn to second edge
turnUntilEdge(True, 30)

# store angle at second edge and calculate angle to turn
secondAngle = imu.get_yaw()
angleToTurn = (firstAngle - secondAngle) / 2

time.sleep(0.5)

# turn to center of object
drivetrain.turn(angleToTurn)
```


Blockly



1.21 Robot Control: Module Overview

In this module students will:

- Understand the basics of robot control and its application

At the end of this module, students will be able to:

- Implement simple controllers such as on-off controllers
- Implement more complex controllers such as P controllers
- Explain the difference between open-loop and closed-loop control

1.21.1 Covered Standards (NGSS and CSTA):

HS-ETS1-2 Break a complex real-world problem into smaller, more manageable problems that each can be solved using scientific and engineering principles.

3A-AP-16 Design and iteratively develop computational artifacts for practical intent, personal expression, or to address a societal issue by using events to initiate instructions.

3A-AP-17 Decompose problems into smaller components through systematic analysis, using constructs such as procedures, modules, and/or objects.

3B-CS-02 Illustrate ways computing systems implement logic, input, and output through hardware components.

3B-AP-10 Use and adapt classic algorithms to solve computational problems.

3B-AP-11 Evaluate algorithms in terms of their efficiency, correctness, and clarity.

3B-AP-15 Analyze a large-scale computational problem and identify generalizable patterns that can be applied to a solution.

3B-AP-16 Demonstrate code reuse by creating programming solutions using libraries and APIs.

1.22 Controlling Behavior: Introduction

Now that robot has information about it's environment (through the use of sensors), let's go over how we can use this information to control our robot's behavior.

1.22.1 Open Loop Control

Before incorporating sensor information, let's go over a simple, open-loop controller.

All this means is that we are going to tell the robot to do something without checking to see if it actually did it.

For example, an oven is an open-loop controller. If you set the cook time to 30 minutes, the oven will cook for 30 minutes regardless of whether or not the food is actually done.

This is a useful controller for simpler processes like ovens that don't need to adapt to their environment. However, for more complex processes like autonomous cars, we would want to incorporate sensor information to make sure that the car is actually doing what we want it to do.

1.22.2 Closed Loop Control

Closed loop control is when we use sensor information to control our robot's behavior.

For example, a self-driving car is a closed-loop controller. It uses sensor information to determine if it is in the correct lane, if it is too close to other cars, etc. and adjusts its behavior accordingly.

In this lesson, we will be using closed-loop control to control our robot's behavior, specifically, we will cover discrete control.

1.22.3 Python Programming Note: Conditionals

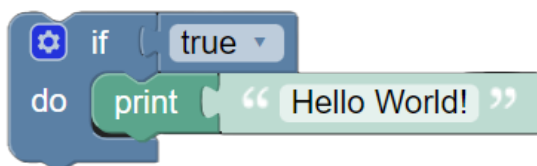
A conditional is a statement in code where the program will only execute a certain block of code if a certain condition is met.

In the context of on-off controllers, "if statements" are an important type of conditional and are shown in this example:

Python

```
if True:
    print("Hello World!")
```

Blockly

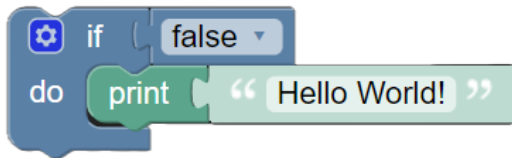


The if statement above will print "Hello World!" because its condition is true.

Python

```
if False:
    print("Hello World!")
```

Blockly

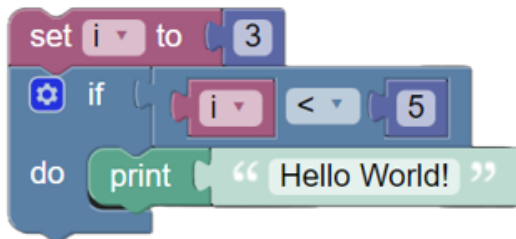


The if statement above will not print “Hello World!” because its condition is always false.

Python

```
int i = 3
if i < 5:
    print("Hello World!")
```

Blockly



The if statement above will print “Hello World!” because the variable “i” is less than 5, satisfying the condition.

In simpler controllers, if statements can be used to define our “control law”. A control law is a set of rules that determines how our robot should behave.

1.22.4 On-off Control

On-off control is a simple closed-loop controller that uses a binary signal (on or off) to control a process.

For example, a thermostat is an on-off controller. If the temperature is below the target temperature, the thermostat turns on the heater. If the temperature is above the target temperature, the thermostat turns off the heater.

In this lesson, we will be using on-off control to control our robot’s behavior. Specifically, we will use an on-off controller to “standoff” our robot from an object.

1.23 Distance Tracking

Now that we've covered on-off control, let's use that information to track an object from a certain distance.

1.23.1 The Process

Essentially, we want our robot to go towards the object if it's too far and away from the object if it's too close.

We can do this by using the distance sensor to determine how far away the object is and then using that information to determine how the direction in which the robot should be going.

For this activity, let's use a target distance of 30 cm.

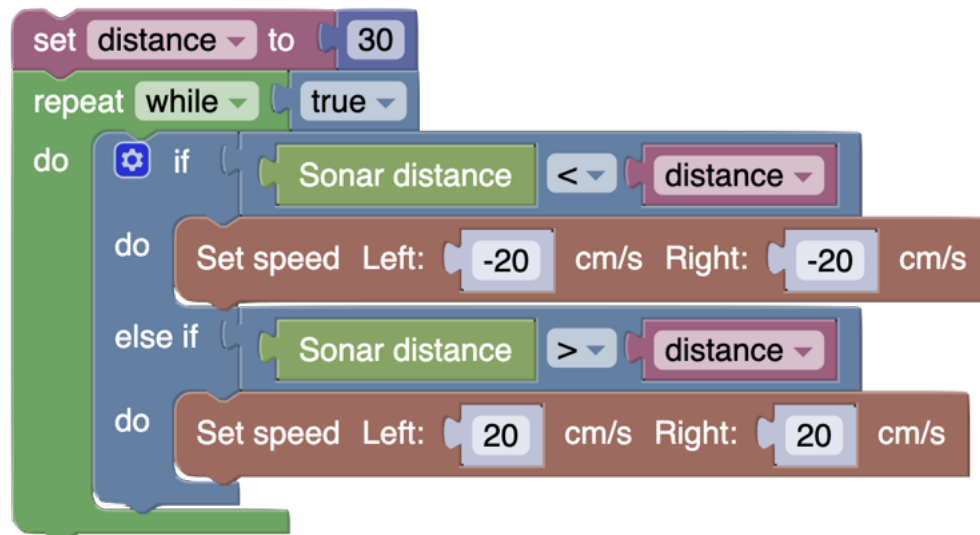
Python

```
from XRPLib.rangefinder import Rangefinder

rangefinder = Rangefinder.get_default_rangefinder()

distance = 30
while True:
    if rangefinder.distance() < distance:
        drivetrain.set_speed(-20, -20)
    elif rangefinder.distance() > distance:
        drivetrain.set_speed(20, 20)
```

Blockly



You'll notice that this code causes the robot to move back and forth, or oscillate, as the sonar distance continuously swaps between being greater than and less than 30 cm. So what if we add a third case that tells the robot's motors to stop when sonar distance equals 30 cm?

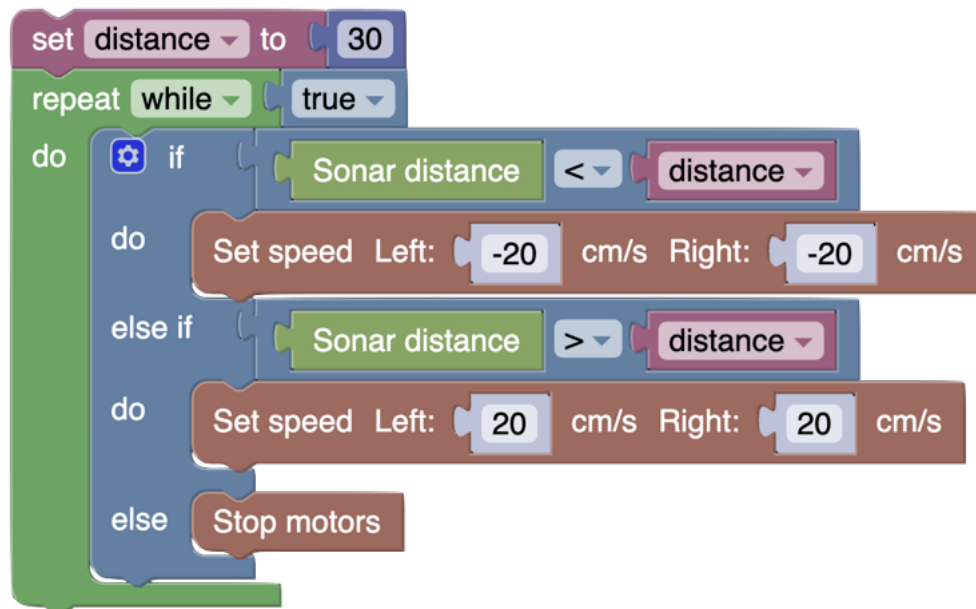
Python

```
from XRPLib.rangefinder import Rangefinder

rangefinder = Rangefinder.get_default_rangefinder()

distance = 30
while True:
    if rangefinder.distance() < distance:
        drivetrain.set_speed(-20, -20)
    elif rangefinder.distance() > distance:
        drivetrain.set_speed(20, 20)
    else:
        drivetrain.stop()
```

Blockly



Unfortunately, even with this code, our robot still doesn't stop! The issue is that the distance sensor is so precise that it never reads exactly 30 cm. We can combat this by making our robot stop when it's *close* to 30 cm instead of *exactly* 30 cm. We can do this by creating a range in which our robot stops called a "deadband." Using a range of ± 2.5 cm, our new code would look like this:

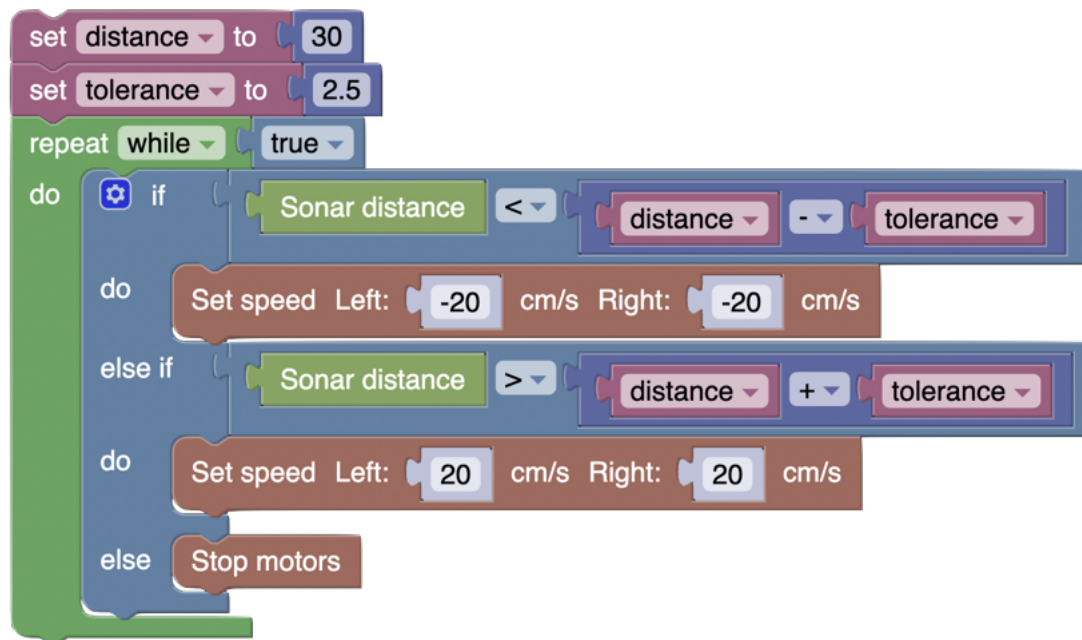
Python

```
from XRPLib.rangefinder import Rangefinder

rangefinder = Rangefinder.get_default_rangefinder()

distance = 30
tolerance = 2.5
while True:
    if rangefinder.distance() < distance - tolerance:
        drivetrain.set_speed(-20, -20)
    elif rangefinder.distance() > distance + tolerance:
        drivetrain.set_speed(20, 20)
    else:
        drivetrain.stop()
```

Blockly



Note: Notice how, instead of hardcoding numbers such as 27.5 and 32.5, we used variables. This gives us two benefits:

1. We can easily change the desired distance and tolerance without having to change the code itself.
 2. It's much easier to decipher what the code is doing, using "magic" numbers like 27.5 and 32.5 can be confusing to read because the user has to figure out what those numbers mean.
-

This code should allow the robot to stop when it senses a sonar distance of ~30 cm. Our issue now is that there is a potential error of 2.5 cm from our desired following distance. Luckily, in the next section, we'll learn about something called "proportional control"...

1.24 Introduction to Proportional Control

1.24.1 What is Proportional Control?

Imagine you're driving a car and you want to keep at a steady speed. If you're going too slow, you press the accelerator a bit, and if you're going too fast, you ease off. But instead of just fully pressing or fully releasing the accelerator (like an on-off switch), you adjust how hard you press based on how far off you are from your desired speed. That's the basic idea behind proportional control. The further you are from your target, the harder you try to correct it. If you're a little off, you make a small adjustment. If you're way off, you make a big one.

Let's take this analogy further - you decide that the perfect cruising speed for your car ride is 70 mph. This speed represents your desired value or where you ideally want to be. In control theory, this is called the **setpoint**.

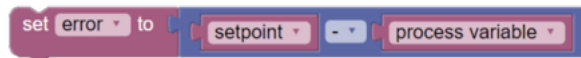
You get on the highway, and as you settle into your drive, you glance at your speedometer. It reads 65 mph, which is your current value. In control theory, this is called the **process variable**.

Naturally, you recognize there's a difference between where you want to be (70 mph) and where you currently are (65 mph). This difference is called the **error**, and in this case, it's 5 mph. It's easily calculated by the formula:

Python

```
error = setpoint - process_variable
```

Blockly



Knowing the error isn't enough. How should you, the driver, react to it? This is where the concept of Proportional control comes into play.

Think of P control as your driving instinct. Instead of abruptly flooring the accelerator or immediately slamming the brakes, you adjust your speed based on your error: how far you are from your desired speed.

A measure called **control output** tells you much to adjust. It's calculated as:

Python

```
control_output = Kp * error
```

Blockly



where K_p is a constant called the **proportional gain** and acts as a scaling factor for the error.

If K_p is high, it's like you have a heavy foot and you'll accelerate hard even for a small error. You'll get there faster, but less precisely and you're more likely to overshoot.

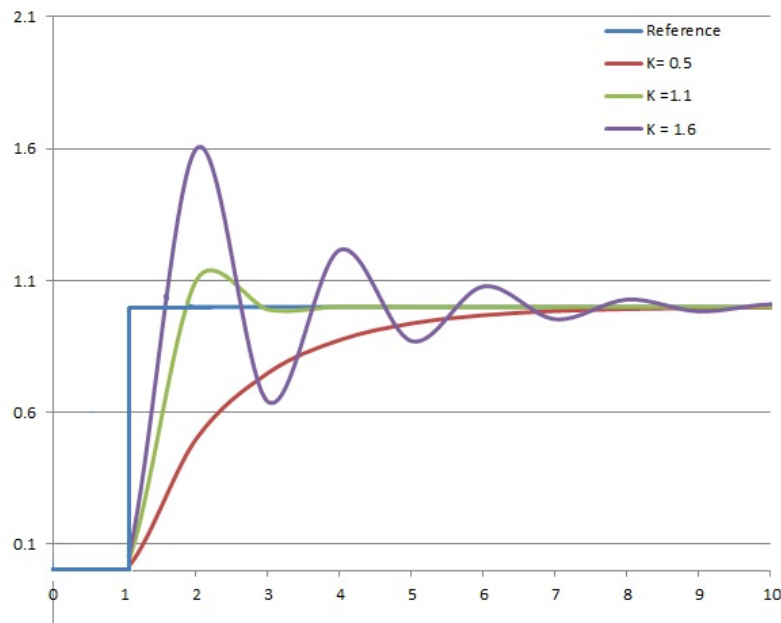
On the other hand, if K_p is low, you're more of a cautious driver, gently pressing the accelerator for the same error. You'll get there more slowly, but at a much smoother pace.

Having the constant K_p allows you to tune your control system to your liking.

Note that this analogy breaks down somewhat. Imagine your current speed is *greater* than your desired speed. In this case, the error is negative. If you plug this into the control output formula, you'll get a negative control output. This means that a proportional controller will actually slow you down if you're going too fast, proportional to how far you are from your desired speed. So, you can imagine that a proportional controller is like a driver who's always trying to get to the speed limit, utilizing both the accelerator and the brakes.

1.24.2 Tuning K_p

The following graph shows proportional control in action.



The blue line is the reference, and indicates the desired value. Red, green, and purple lines represent the current value over time when controlled by a proportional controller with different values of K_p .

With a low K_p , the red line is slow to react to the error, and the controller is sluggish. It takes a long time to reach the desired value, and it never quite gets there. This is called **underdamped** behavior.

With a high K_p , the purple line is quick to react to the error, and the controller is aggressive. It reaches the desired value quickly, but overshoots, causing the error to become negative. It then corrects itself, but overshoots again, and so on. This is called **overdamped** behavior, and results in oscillations around the desired value.

The green line is just right. It reaches the desired value quickly, and doesn't overshoot much. It's an important task to tune K_p so that the controller approaches the desired value as quickly and smoothly as possible.

Note: You'll find that, even with excellent tuning, a proportional controller often will either oscillate a little bit, or never quite reach the desired value. More advanced control systems like PID aim to minimize these issues, but they are out of the scope of this course.

1.25 Implementing a Proportional Controller

Now that you’ve learned about how proportional controllers, let’s implement one for our distance tracking activity, where we want to keep the robot some distance from the object in front of it using the rangefinder!

1.25.1 Defining Terminology

Let’s identify the terms we’ll need to use in our code:

Set Point or desired value: In this example, this is some set distance from the rangefinder we want the robot to be at. For this example, let’s say 20cm.

Process Variable or current value: We obtain our measured value from reading the rangefinder value. This is `rangefinder.distance()`.

Our goal is for our process variable (the rangefinder distance) to approach the set point (20 cm).

Thus, our **error** is calculated as `error = rangefinder.distance() - 20`. Note that `error = 20 - rangefinder.distance()` is also “correct”. The distinction in sign is simply whichever makes more sense for your application. Here, if we had an error of 30cm, we would want to drive forward 10cm, so we would want a positive error to make our motors spin forward.

Control Output: In this case, this is our motor effort. This is because we want to drive with a speed proportional to the distance error. As a reminder for P control, this will be calculated as `motor_effort = Kp * error`.

Kp: This is our proportional gain. Though we will need to tune this value, we can guess a somewhat reasonable value by considering the range of values our error can take, and the domain of our control output. In this case, if we’re 30cm away from the object, our error will be 10. We can guess that at this sufficient distance we will want to drive forward at a maximum effort of 1, as effort is restricted to the domain $[-1, 1]$. Thus, we can guess that $Kp = 1/10 = 0.1$. Of course, this isn’t likely the final value that works best for your robot, but it’s a good starting point.

1.25.2 Implementing the Controller

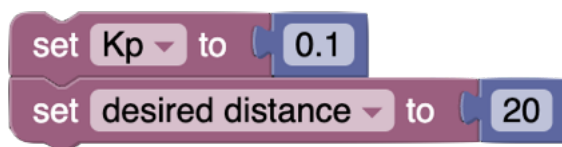
Now that we’ve defined our terms, let’s write the code!

Let’s start by defining our proportional gain and our set point:

Python

```
Kp = 0.1
desired_distance = 20
```

Blockly

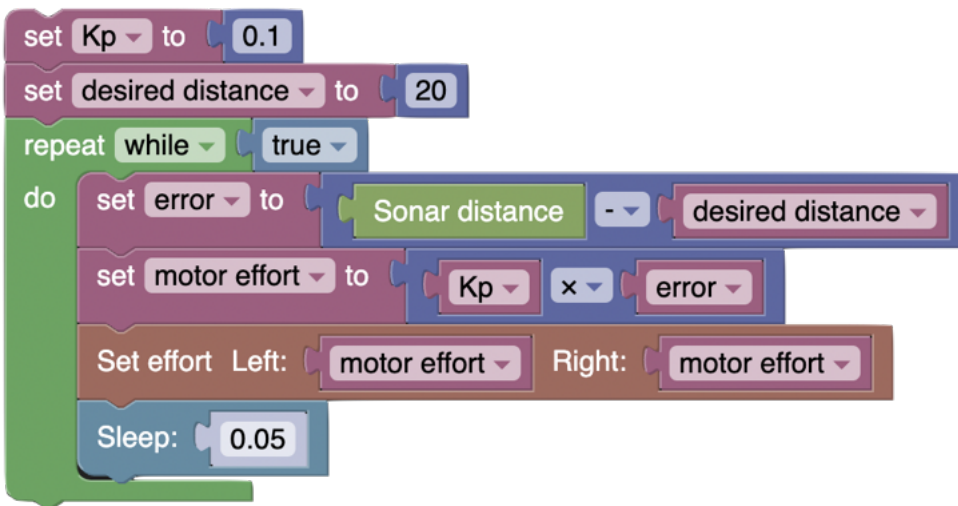


Next, we want to enter some sort of loop to continuously read our rangefinder value and update our motor effort from our controller output.

Python

```
Kp = 0.1
desired_distance = 20
while True:
    error = rangefinder.distance() - desired_distance
    motor_effort = Kp * error
    drivetrain.set_effort(motor_effort, motor_effort)
    time.sleep(0.05)
```

Blockly



Each iteration of the loop consists of the following steps:

1. Read the rangefinder value to get the current distance
2. Calculate the error
3. Calculate the control output through $K_p * \text{error}$
4. Set the drivetrain motor efforts to the control output
5. Wait for a short period of time

This code should give us a working solution to maintain a set distance from the object in front of the robot!

Try it out

Try moving the object in front of the robot and watch the robot attempt to maintain the set distance! What happens when you increase K_p ? Decrease it? What value of K_p works best for your robot?

1.26 Introduction to Wall Following

Now that you've developed a program to keep a certain distance from an object, let's implement this by having the XRP follow a wall while maintaining a target distance.

1.26.1 Building a control law

when you follow a wall, all you need to do is steer the robot so you can keep a certain distance from the wall. If you are following a wall on your right side, you will turn right if you are too far and left if you are too close. We can easily do this with a proportional control loop. The steering correction can be proportional to the error, in this case the difference between the distance to the wall and the target distance to the wall.

Tip: Remember that you will want to incorporate a “base effort” to ensure that the robot is moving forward at all times. Let's set this to **0.5**.

It is also important to note that the side you choose to plate your ultrasonic range finder will affect the implementation of the control law.

Now, implement a proportional controller given the steps that you have previously followed and the tips noted above.

Watch this video to see what a working wall-follower looks like.

1.26.2 Implementing Wall Following

Today, let's use the information we learned last time to actually implement a wall-follower.

Here is some code that would allow your XRP to track a wall on the right side of the robot.

Python

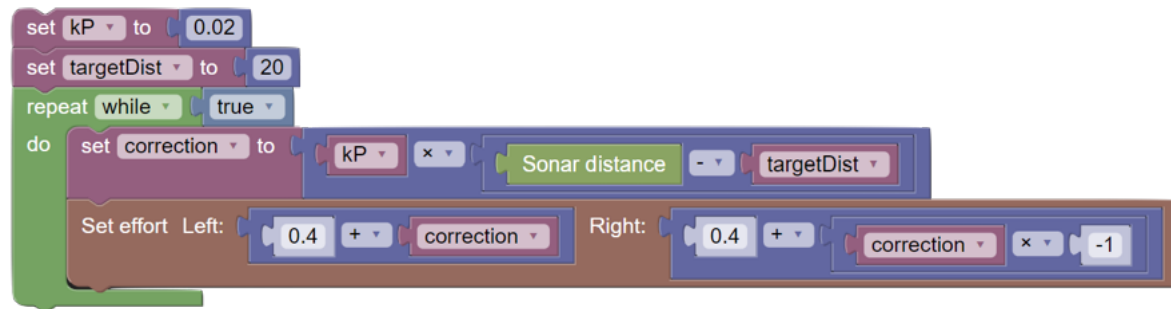
```
from XRPLib.differential_drive import DifferentialDrive
from XRPLib.rangefinder import Rangefinder

kP = None
targetDist = None

differentialDrive = DifferentialDrive.get_default_differential_drive()
rangefinder = Rangefinder.get_default_rangefinder()

kP = 0.02
targetDist = 20
while True:
    differentialDrive.set_effort((0.4 + kP * ((rangefinder.distance()) - targetDist)),
    ↪ (0.4 + (kP * ((rangefinder.distance()) - targetDist)) * -1))
```

Blockly



1.27 Sensing and Following Lines: Module Overview

In this module students will:

- Understand the task of following a line with the robot
- Learn to use the reflectance sensor to detect the line
- Write simple programs for detecting and responding to lines
- Write simple programs to follow lines
- Enhance the simple program with better sensor measurements and calculations to follow the line more smoothly.

At the end of this module, students will be able to:

- Discuss robot navigation concepts with their classmates
- Get their robot to follow a line
- Develop simple sensor logic control for robot decision making

1.27.1 Covered Standards (NGSS and CSTA):

HS-ETS1-2 Break a complex real-world problem into smaller, more manageable problems that each can be solved using scientific and engineering principles.

3A-AP-15 Justify the selection of specific control structures when tradeoffs involve implementation, readability, and program performance, and explain the benefits and drawbacks of choices made.

3A-AP-16 Design and iteratively develop computational artifacts for practical intent, personal expression, or to address a societal issue by using events to initiate instructions.

3A-AP-17 Decompose problems into smaller components through systematic analysis, using constructs such as procedures, modules, and/or objects.

3B-CS-02 Illustrate ways computing systems implement logic, input, and output through hardware components.

3B-AP-10 Use and adapt classic algorithms to solve computational problems.

3B-AP-11 Evaluate algorithms in terms of their efficiency, correctness, and clarity.

3B-AP-15 Analyze a large-scale computational problem and identify generalizable patterns that can be applied to a solution.

3B-AP-16 Demonstrate code reuse by creating programming solutions using libraries and APIs.

1.28 Understanding the line sensor

In the last module, you used the distance sensor to measure distances and make the XRP follow along a wall. The XRP has another sensor that allows it to navigate: the **line following sensor**.

The line following sensor consists of two “reflectance” sensors. Simply put, the reflectance sensor shines a light at the ground and measures how much of the light is reflected back. The darker an object is, the less light it reflects. The sensor uses infrared light, just like a TV remote, so the light is not visible to the human eye.

This sensor is perfect for sensing dark lines on a light background! If the sensor is on top of a dark line, less light will be reflected back, and if it is not on a line, more light will be reflected back. You can use this information in your code to let the robot decide what to do in these situations.

The XRP has *two* reflectance sensors, a left sensor and a right sensor. If you look at the bottom of your XRP on the sensor board, you will see the two sensors. **S1** is the left sensor and **S2** is the right sensor. Later in the module you will learn a way to use both sensors to follow lines very smoothly. For this lesson, we will only use the *right* sensor.

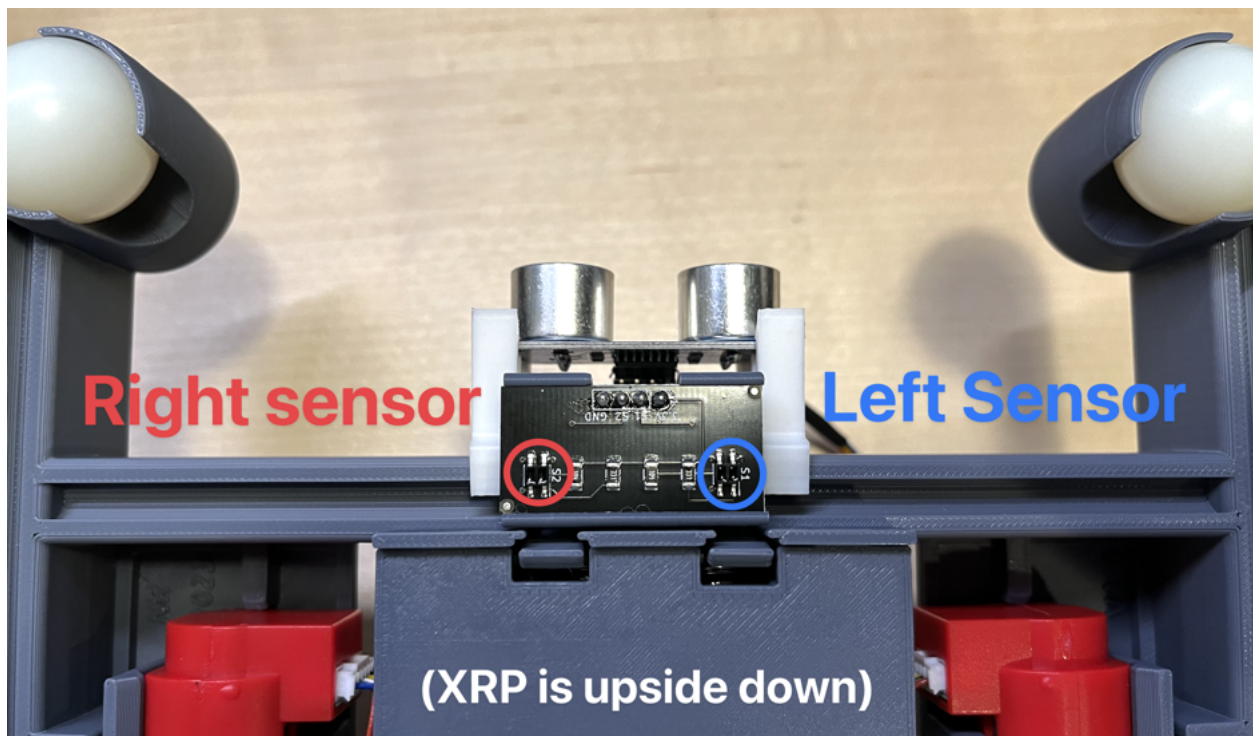


Fig. 3: The two reflectance sensors on the XRP.

XRPLib provides functions to read the values of the reflectance sensors:

```
from XRPLib.defaults import *

# Reads the left sensor and stores the value in the variable "left"
left = reflectance.get_left()

# Reads the right sensor and stores the value in the variable "right"
right = reflectance.get_right()
```

Before doing anything with a new sensor, you need to have a good understanding of the values it will give you in different conditions. For the reflectance sensor, it would be good to know what the sensor reports when it is completely

off of the line (seeing a white surface), completely on the line (seeing a black surface), and some “middle of the road” values, when the sensor is half on the line and half off the line.

Tip: Remember that for this exercise you should only be using the *right* line sensor. Make sure that you center the correct part of the sensor board over the line when taking your measurements.

Try it out

Write code to read the value of the right reflectance sensor and use the webserver to log and graph the values in an infinite loop. Move the XRP around on a white surface with a line, and take note of the values the sensor reads in the different conditions above.

What do you notice from the values you measured? The documentation for the `reflectance` module in **XRPLib** states that the `get_left()` and `get_right()` functions return a number between 0 and 1. Did your values ever reach exactly 0 or exactly 1? Can you tell which range of numbers corresponds to seeing white and which range of numbers corresponds to seeing black?

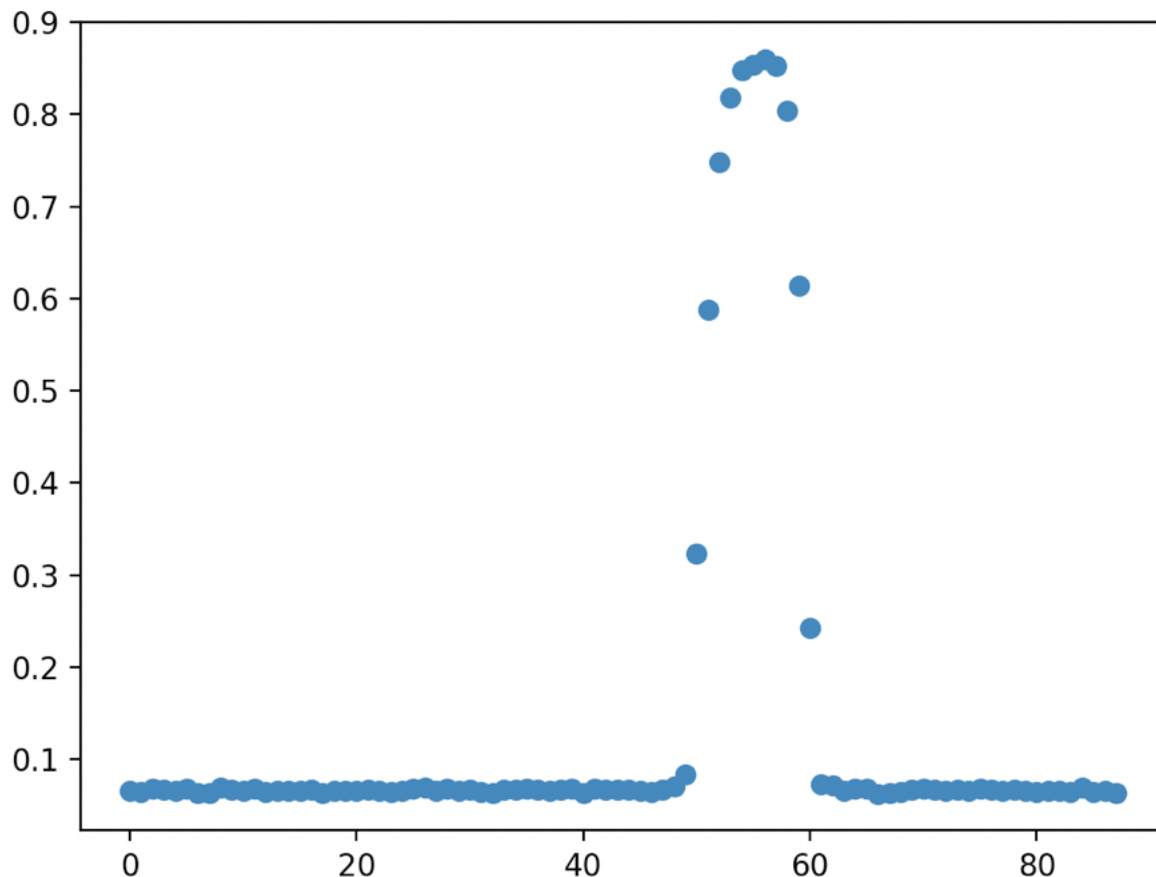


Fig. 4: Example graph of reflectance sensor data.

Above is an example graph of some data from the reflectance sensor. At around 50 units on the X axis, the reflectance sensor was moved over a line (this data was recorded while the robot was driving across a line) and at around 60 units, the reflectance sensor was moved back off of the line. Does your graph look similar to the one above?

Note: Your graph will not look exactly the same as ours. No two reflectance sensors are exactly the same, so it is important for you to take your own measurements with your own robot.

It's good to experiment with the reflectance sensor to see what it does, but you took this data for a reason. The line following sensor reports back a number, but what we'd really like it to tell us is whether it sees a line or not. To do this, you'll need to select a "threshold" value, where if the sensor reports a value greater than the threshold, we can confidently assume the sensor is seeing a line, and if the sensor reports a value below the threshold, we can assume it is not seeing a line.

Try it out

Look at your graph and select a threshold value that makes sense to you. A number around halfway between the minimum and the maximum value you measured is a good starting point.

Write a function called `is_over_line()` which reads the value of the right reflectance sensor and returns `True` if the sensor sees a line (value above the threshold) or `False` if it does not. Don't delete this function when you're done, because you'll use it for the rest of the module!

Use the webserver to log the result of calling your function in an infinite loop. Move your robot around a surface with lines on it to make sure it always returns the correct value based on what the sensor is seeing. If you are getting incorrect values, adjust your threshold value.

1.29 Stopping at a Line

In the last module, you wrote and tested a function which could accurately determine if the reflectance sensor was able to see a line. In this activity, you'll use that function to make the robot stop when it sees a line.

Fig. 5: The XRP stopping when it sees a line.

Let's consider a previous exercise - using a while loop to drive a certain distance:

Python

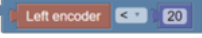
```
from XRPLib.defaults import *

while drivetrain.get_left_encoder_position() < 20:
    drivetrain.set_speed(5, 5)
drivetrain.stop()
```

In this code, the condition being checked is `drivetrain.get_left_encoder_position() < 20` meaning that the robot will drive forward at 5 cm/s until the left encoder reads a distance of 20 cm. This code can be easily modified to replace the current condition with a condition that uses the function you wrote.

Blockly



In this code, the condition being checked is  meaning that the robot will drive forward at 5 cm/s until the left encoder reads a distance of 20 cm. This code can be easily modified to replace the current condition with a condition that uses the function you wrote.

Try it out

Modify the example code to use your function (`is_over_line()`) as the condition for the loop.

If you need to “invert” the value of your function (convert `False` to `True` and `True` to `False`), you can use the *not operator* before calling your function like this: `not is_over_line()`. This code does exactly what it sounds like: returns `True` when the robot is *not* over the line.

Once you’ve tested your code and proved it to meet the challenge, make a new function called `drive_until_line()` and put your code in it. Don’t delete this function, as you’ll need it later!

1.29.1 Challenge activity

For an added challenge, try to write code which makes the robot capable of driving over and stopping at several lines. The robot should drive over a line, stop for some amount of time, say two seconds, and then start driving again until it sees another line. This cycle should repeat forever.

Tip: You’ll need to write some logic which handles the robot driving *off* of the line too! Your code from the main activity might not be enough to handle this! Think about what your code would do if it started out *already on* a line.

1.30 Staying in the Circle

Your robot is now capable of stopping when it sees a line. You can use this functionality to keep your robot trapped inside a circle! You’ll find out why you’d want to do this in the next module, which is a challenge activity!

Let’s break this problem down into a series of steps:

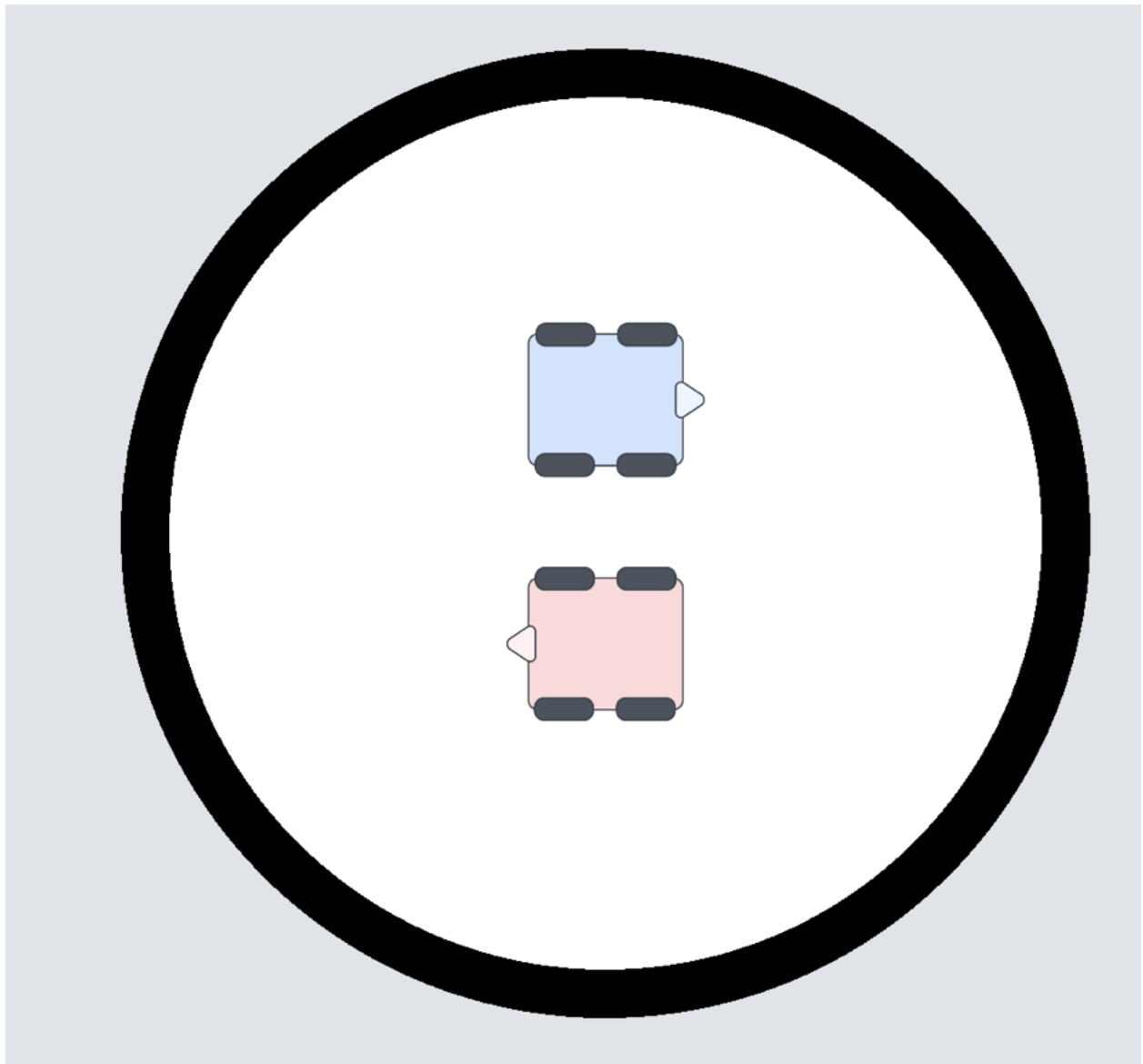
1. Drive forward until a line is seen (the edge of the circle)
2. Stop driving so that the robot doesn’t leave the circle
3. Turn around
4. Repeat

You already have code which does steps 1 and 2 (`drive_until_line()`), and you learned back in the robot driving module how to do step 3 (`drivetrain.turn()`, see [/course/driving/calling_drive_functions](#) for a refresher)

Try it out

Write an infinite loop which keeps the robot in a circle. Try out different angles when turning around. You may want to try not turning a full 180 degrees.

1.31 Challenge: Sumo-Bots!



It's time for SUMO bots! Two XRP bots battle it out in the ring in a completely autonomous match to push the other robot outside of the ring.

Robots start facing away from each other in the orientation above, and have one minute to knock the other robot outside. They may utilize distance sensors to detect the presence and location of the other robot, and use the reflectance sensors to keep themselves inside the ring.

Hint: A basic SUMO-bots program may consist of a robot continuously point turning until an enemy robot is found with the distance sensor, and then charging at the robot until the black line is detected, so that the robot stays inside the ring. However, worthy extensions include: aligning the robot to be perpendicular from the black line so that the robot is not misaligned, and devising an algorithm to attack the opponent robot from the side to avoid a head-on collision and gain more leverage.

1.32 Following the Line: On/Off Control

Now, let's turn our attention towards one of the core challenges in the final project - following a line. In the project, the robot will need to drive to multiple different locations, but doing this just based on distance can result in the robot not getting to exactly the right place. What if the wheels slip while driving? What if the robot needs to drive along a complex curve? It's easier to follow a line than it is to exactly measure out the course the robot needs to follow and program it.

1.32.1 How do we follow a line?

Consider using one of the reflectance sensors. As a refresher, gives a reading from 0 (black) to 1 (white). Assuming that the reflectance sensor is approximately at the center of the robot, it will at least partially reading the black line when the robot is centered on the line. What type of logic would we need if we wanted to follow the center of the line?

Well, if the reflectance sensor reads black, it means the robot is perfectly on the line, and we'd want to go straight, setting the motors at the same speed. But if the reflectance sensor reads grey or white, it would mean that the robot is partially or completely off the line. We'd want to correct this by steering it back to the center, but does it turn left or right?

Unfortunately, there's no way to tell. The robot has no way of knowing which direction it is drifting off the line. Instead, try following an edge of the line. If we try to follow the left edge, then there's two possible states in which the robot reacts.

- If the sensor reads closer to white, that means we're too far to the left, so we need to turn slightly to the right.
- If the sensor reads closer to black, that means we're too far to the right, so we need to turn slightly to the left.

And that's it! We want to keep polling (getting the value of) the reflectance sensor quickly, and at each time determine whether it's closer to white (with a value less than 0.5) or closer to black (with a value greater than 0.5), and depending on the result, either set the motor to turn right (set left motor speed to be faster than right) or turn left (set right motor speed to be faster than left).

This seems like a solution involving an if-else statement. Our condition would be related to whether the value is greater or less than 0.5.

An `if / else` statement allows you to run different blocks of code based on a *condition* (the same kind of *condition* you used in a `while` loop)

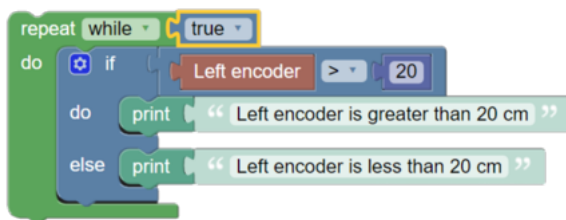
Consider the following example code:

Python

```
from XRPLib.defaults import *

while True:
    if drivetrain.get_left_encoder_position() > 20:
        print("Left encoder is greater than 20 cm")
    else:
        print("Left encoder is less than 20 cm")
```

Blockly



In this example code we just show different messages on the computer based on the value of the left encoder, but you could put whatever code you want in the blocks instead. For example, you could have the robot turn clockwise or counterclockwise depending on a condition using an if / else statement.

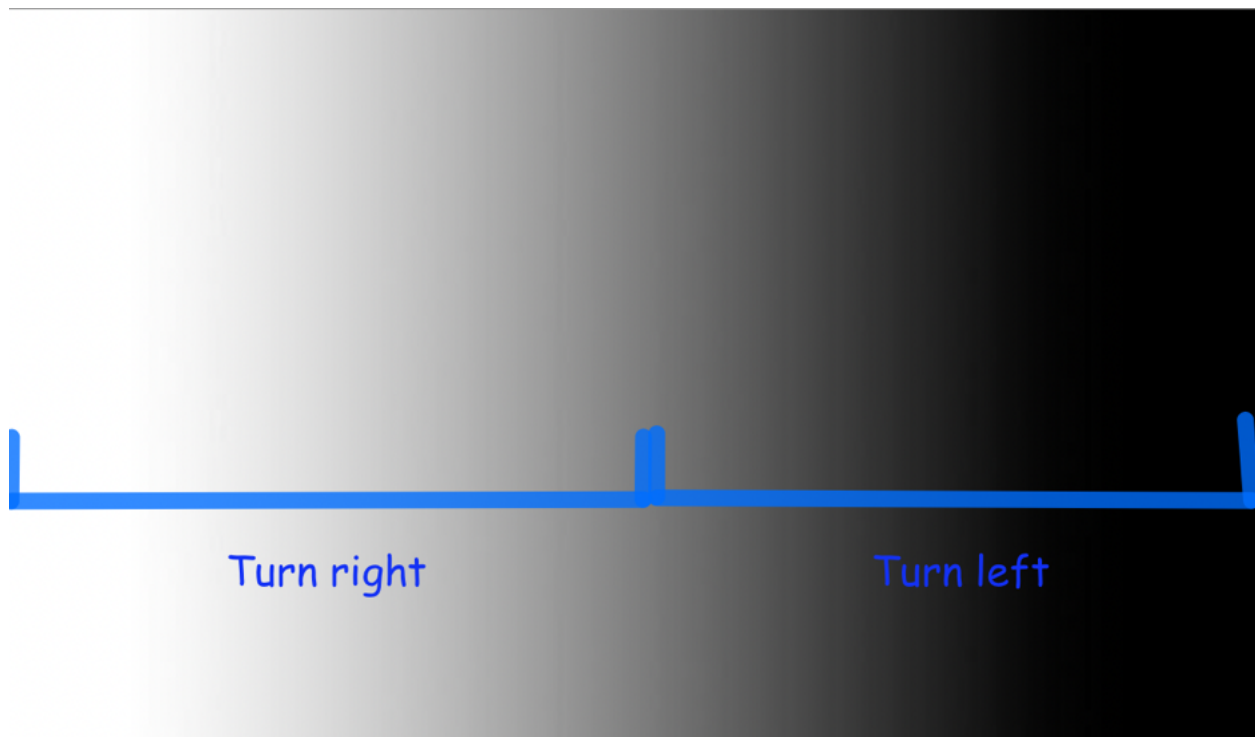


Fig. 6: Actions your robot should take based on what the sensor sees.

Above is an illustration of how we'd want the robot to act based on the reading of the sensor.

Try it out

Write some code which uses an `if / else` statement to turn the robot one direction or another based on the reflectance sensor. To do this, you'll use the `drivetrain.set_speed` function using different speeds for the left and right wheels. Remember, using a higher speed on the left wheel will make the robot turn right, and vice versa. You can use your `is_over_line()` function to check if the sensor sees a line.

You will need to experiment with different speed values for each wheel; too high and your robot will drive off the line before it gets a chance to correct for it, too low and your robot will not correct in time and will spin in circles. Try to get your robot to follow the line as fast as you can!

1.33 Following the Line: Proportional Control

In the measuring distances module, you used proportional control to make the robot drive straight along a wall. We can apply proportional control to the line following problem too!

1.33.1 The perks of proportional control

Let's circle back to the previous exercise - following the line by either turning left or right depending on whether the robot is situated to the left or the right of the line.

What is immediately striking about following the line in this way? Well, the robot must constantly oscillate in order to stay on the edge of the line, because even the smallest deviation from the edge of the line results in the robot wildly turning to compensate. In addition, it does not react any more forcefully to bigger deviations like when the line starts curving, and as soon as it loses sight of the line, it has no way of recovering.

Instead of only having two cases, it seems like we'd want a whole bunch of cases, for anywhere from a sharp left turn to going perfectly straight to a sharp right turn, and everything in between, based on whether the reflectance sensor is completely on white, grey, black, or somewhere in between.

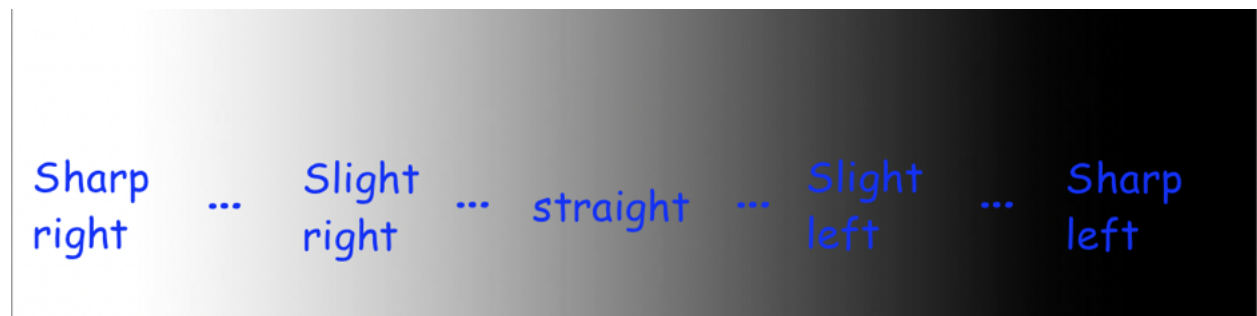


Fig. 7: Desired steering actions based on what the sensor sees.

Having a long chain of if-else statements doesn't sound fun. Perhaps we can look at this with a completely fresh approach?

From the previous module, we looked at proportional control to smoothly control the robot's distance to the wall using the distance sensor. Can we use the same concept here?

1.33.2 Calculating error

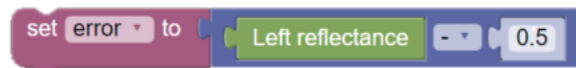
With proportional control, we have an error value we desire for it to tend to zero, and some motor output is controlled proportional to the error to minimize that error - in the case of maintaining a certain distance to the wall, the error was the difference between the target and actual distance, and the output was the speed of both drive motors. In the case of line following, the error is the difference from 0.5 - since ideally, the robot follows the grey edge of the line and goes straight - and the motor output is how the robot should turn.

So, we can obtain error value with the following code:

Python

```
error = reflectance.get_left() - 0.5
```

Blockly



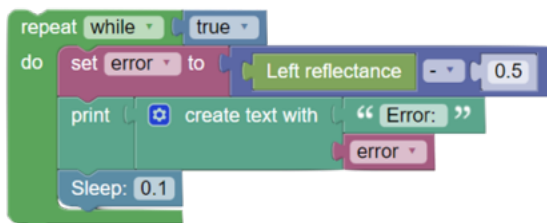
Above, we subtract 0.5 to *normalize* the reflectance value: the error is negative when the robot is too far left and needs to turn right, and positive when the robot is too far right and needs to turn left. Let's put that code into the test. We can put it in the loop, print out the error at each iteration, and move the robot around the line to see how the error changes. The code is as follows:

Python

```
from XRPLib.defaults import *
from time import sleep

while True:
    error = reflectance.get_left() - 0.5
    print("Error: ", error)
    sleep(0.1) # This sleep makes the loop run 10 times every second
```

Blockly



1.33.3 Implementing proportional control

Based on the computed error, we want that to determine how much the robot turns.

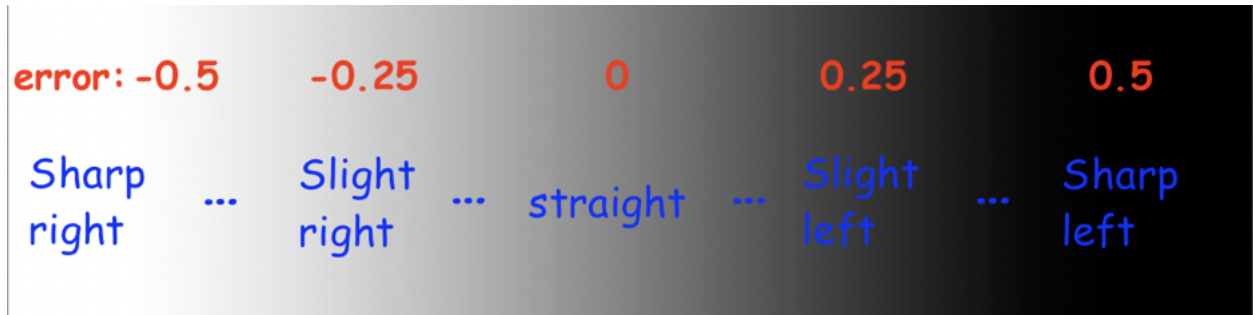


Fig. 8: Desired steering actions and proportional control output based on what the sensor sees.

This image illustrates how the error impacts how much we want to turn. Remember: making the robot turn is simply setting the left and right motors to different efforts. So, the solution is to set a base effort, say, 50% effort, that both motors move at when the error is at 0. Then, have the calculated error influence the difference in efforts between the two motors. As explained through code:

Python

```
drivetrain.set_effort(base_effort - KP * error, base_effort + KP * error)
```

Blockly



This would be run inside the loop. The `base_effort` represents the average effort of the motors, no matter how much the robot turns. `KP` scales how much the robot should turn based on the error - a higher `KP` means the robot will react more violently to small deviations in error.

Let's do a quick check to make sure the code makes sense. We assume `base_effort = 0.5` and `KP = 1`. If the reflectance reads whitish-grey and yields a value of around 0.25, the error would be -0.25, meaning that the left motor's effort is:

$$\begin{aligned}
 &= 0.5 - 1 \cdot -0.25 \\
 &= 0.5 + 0.25 \\
 &= 0.75
 \end{aligned}
 \tag{1.1}$$

and the right motor's speed is:

$$\begin{aligned}
 &= 0.5 + 1 \cdot -0.25 \\
 &= 0.5 - 0.25 \\
 &= 0.25
 \end{aligned}
 \tag{1.3}$$

Motor efforts of 0.75 and 0.25 would indicate a turn to the right, and the code does as desired.

This is a video illustrating line following with one-sensor control. Notice the smoother tracking compared to on/off control, yet the robot is still unable to recover from the last bend, because even a small amount of strafing from the line

Fig. 9: XRP following a line with proportional control. The robot would not be able to follow a curved line this quickly using on-off control!

results in the robot completely losing where it is. Also, the KP value was not equal to 1 here; it's up to you to figure out the best KP value for your bot.

Try it out

Write code for the robot to follow the line with proportional control, as shown in the video above. Note: this isn't much more than calculating error as shown in the previous section then integrating the above line of code in a loop.

Play around with the value of KP. How does a higher or lower KP affect the amount of oscillation when following the line, and how responsive the robot is to curved lines? What is the optimal value of KP?

1.34 Following the Line: Proportional Control with Both Sensors

Line following with proportional control using one sensor is quite an improvement to on/off control. Yet, it's not perfect - if the reflectance sensor crosses over the center of the line, it's game over.

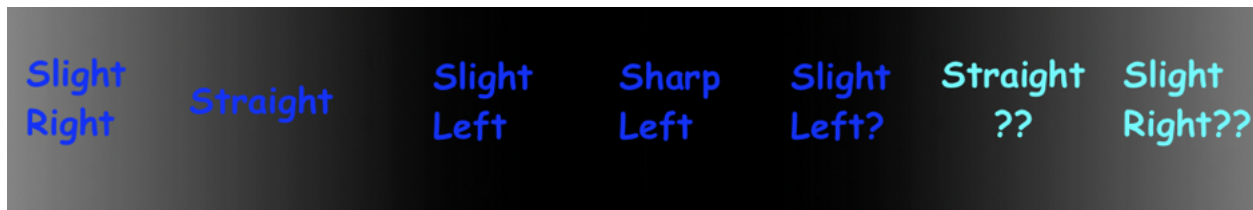


Fig. 10: Desired steering actions based on what the sensor sees.

The issue is - the robot has no way of knowing which side of the line it is following at all! If it sees the right edge of the line, it will assume it still is detecting the left edge, and thus keep turning right past the point of no return!

It would be neat if we could actually follow the center of the line, and can recognize both cases in which the robot drifts to the left or the right of the center of the line, and makes a correction. This would largely increase the "controllable" range of what the reflectance sensor sees and can correctly react to. Conveniently, it seems like we haven't yet made use of the second reflectance sensor on the right...

If we recorded the reflectance of both sensors as we moved the robot around the line, there are a few major "categories" of behaviors the robot would perform. For a minute, assume the rectangle is a black line and the two red squares are the location of the reflectance sensors.

Sensors read ~0 and ~0. Robot does not know if it is on either the left or right side of the line.

Sensors read ~0 and ~1. Robot knows it is on the left side and turns right.

Sensors read ~0.5 and ~0.5. Robot knows it is on the center and goes straight.

The other two major categories you can extrapolate for yourself.

So, how can we effectively combine the readings of the left and right reflectance sensors using proportional control to have the robot follow the line? There's quite an elegant solution that I encourage for you to try to figure out yourselves before the answer is revealed.

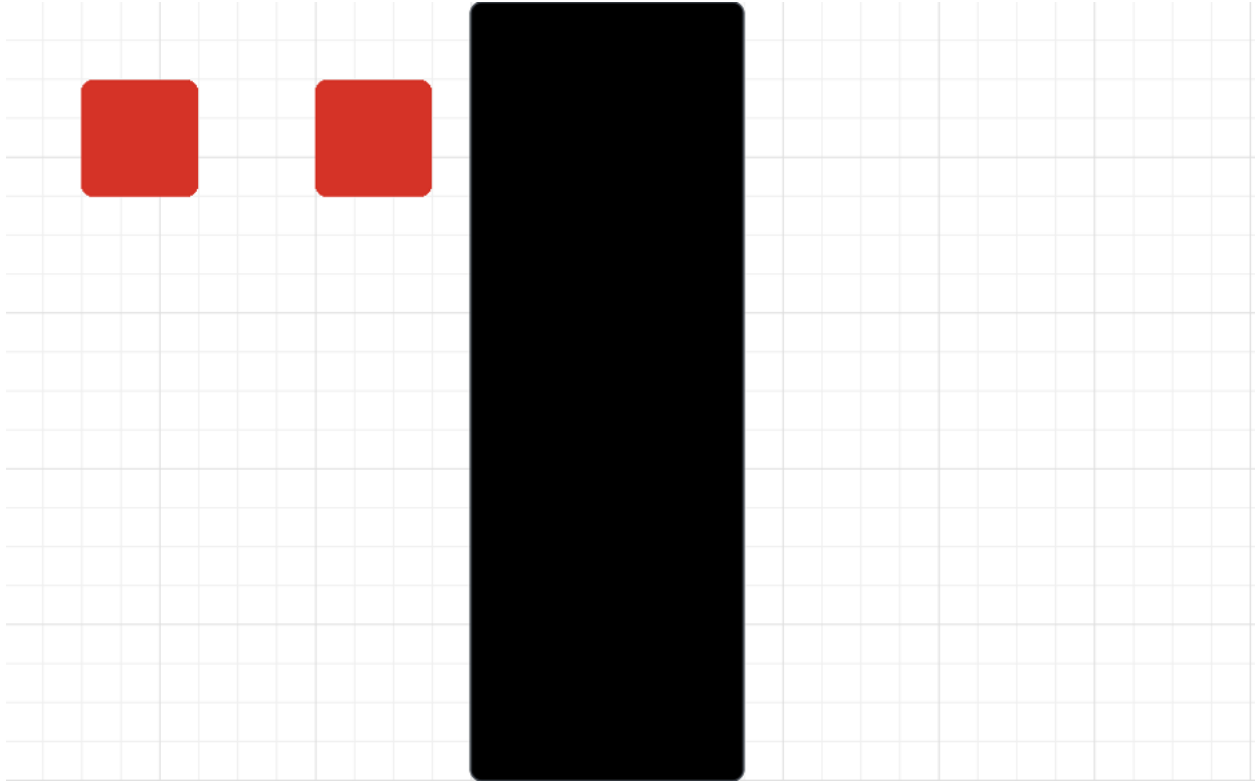


Fig. 11: Simplified view of both reflectance sensors completely off of the line.

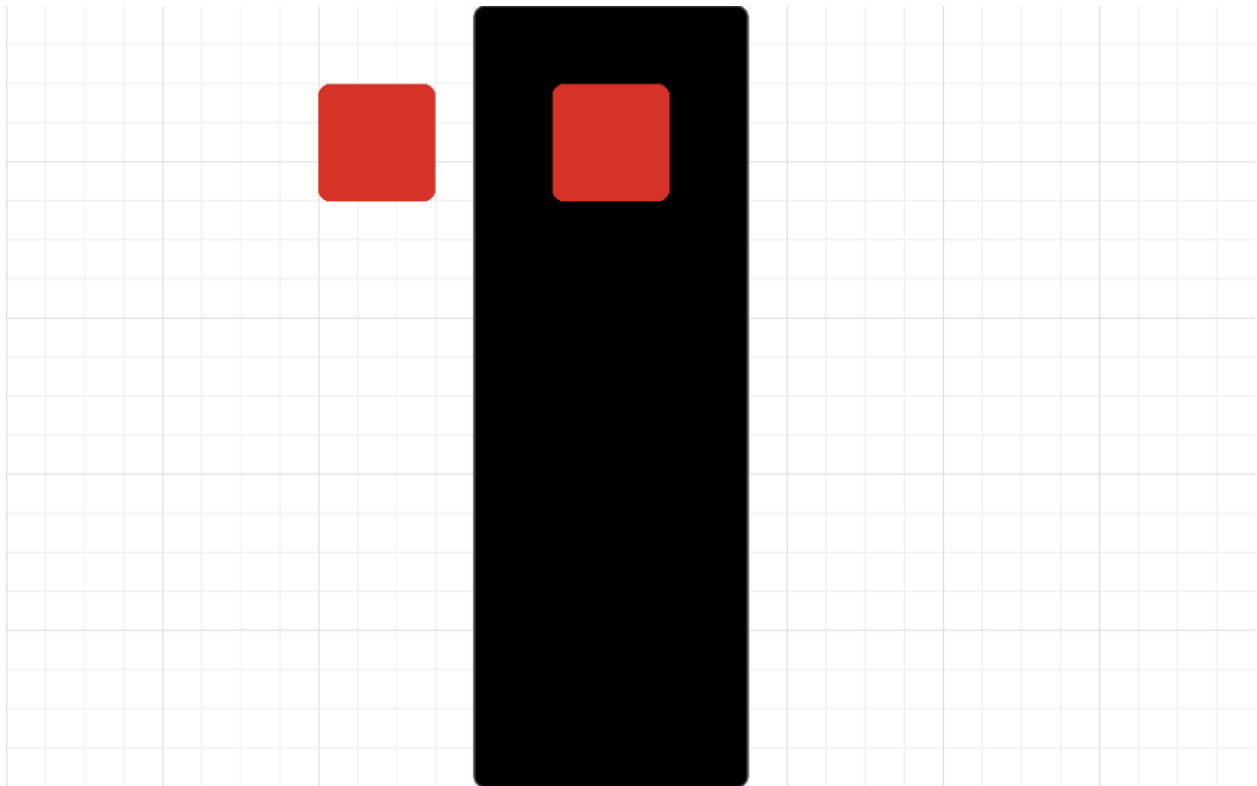


Fig. 12: Simplified view of both reflectance sensors half on the line.

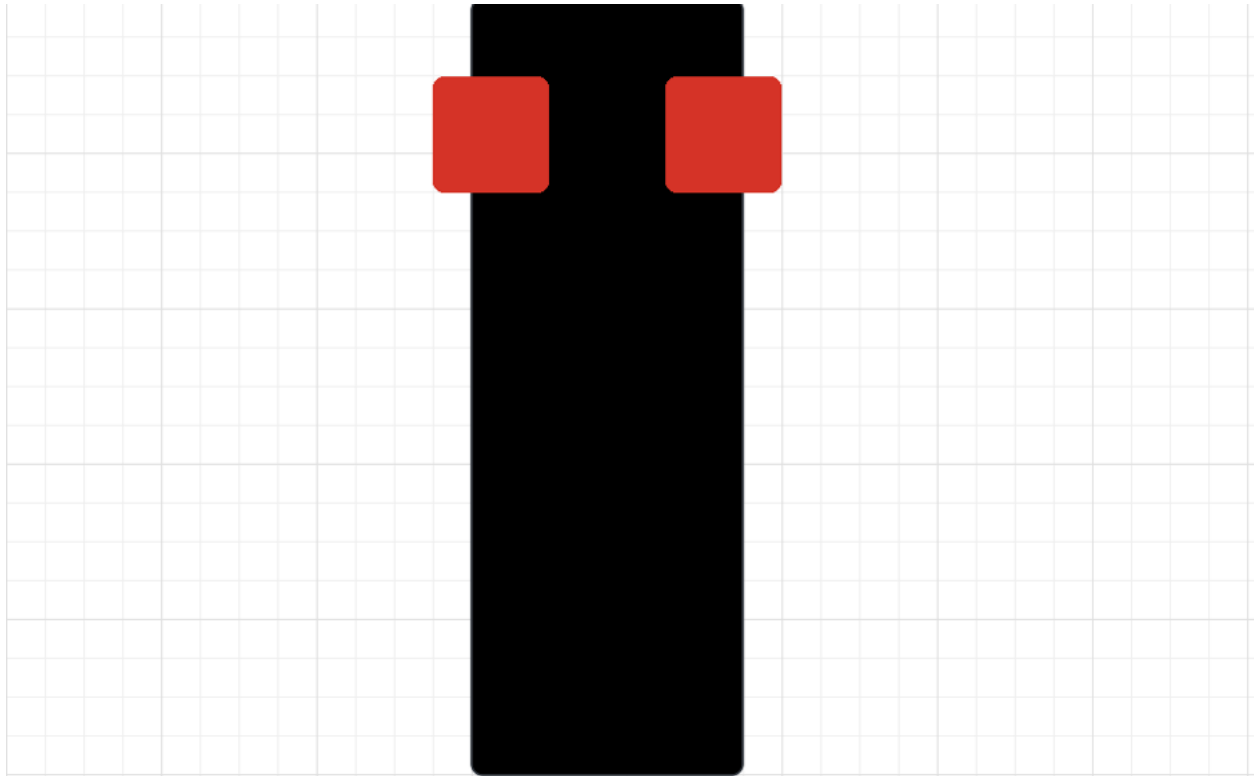


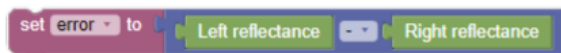
Fig. 13: Simplified view of both reflectance sensors centered on the line.

1.34.1 Implementation

Python

```
error = reflectance.get_left() - reflectance.get_right()
```

Blockly



At the beginning this line of code may not make a lot of sense - but let's dissect it. Remember our previous convention of positive error meaning the robot is too far left and needs to turn right, and vice versa.

In this case, if the robot is following the left edge of the line, then the left sensor detects close to white while the right sensor detects close to black, and so:

$$\begin{aligned} \text{error} &= 0 - 1 \\ &= -1 \end{aligned} \tag{1.5}$$

Which causes the robot to turn right. On the other hand, if the robot is following the right edge of the line:

$$\begin{aligned} \text{error} &= 1 - 0 \\ &= 1 \end{aligned} \tag{1.7}$$

Which causes the robot to turn left. When the robot is right at the center, both sensor values are the same and so the error is 0, and as the robot starts drifting towards either direction, the magnitude of the error increases and thus the robot compensates accordingly.

The most interesting case is when the robot is completely off the line - in this case, both sensors read white, leaving an error of 0, and so the robot just goes straight. Given that the robot wouldn't know which direction to compensate if it was completely off the line, this seems like a reasonable result.

And so, our final code is as follows:

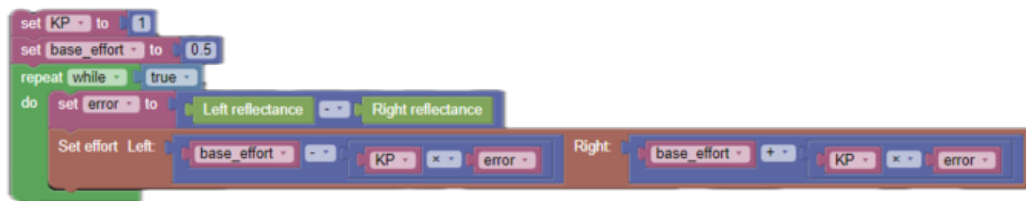
Python

```
from XRPLib.defaults import *

# Try different values for KP and base_effort to get things working smoothly
KP = 1
base_effort = 0.5

while True:
    error = reflectance.get_left() - reflectance.get_right()
    drivetrain.set_effort(base_effort - KP * error, base_effort + KP * error)
```

Blockly



Here's what that looks like. Note that KP used in this video was not equal to 1:

Error: TODO add video

Try it out

- Combine what you've learned with encoders to create a function that follows the line using two sensors for some given distance, and, then stop the motors.
 - What KP value is best?
 - Compare one sensor to two sensor line following. What bends in the black line is two sensor line following able to handle that one sensor line following cannot?
-

1.35 Stopping at an Intersection

With what you've learned in this module, you can now use proportional control to drive your robot quickly and accurately along a line using both reflectance sensors. However, smooth line following is not the only reason the XRP has two sensors. Having two sensors also allows us to detect intersections between two lines.

Consider this diagram from the last module:

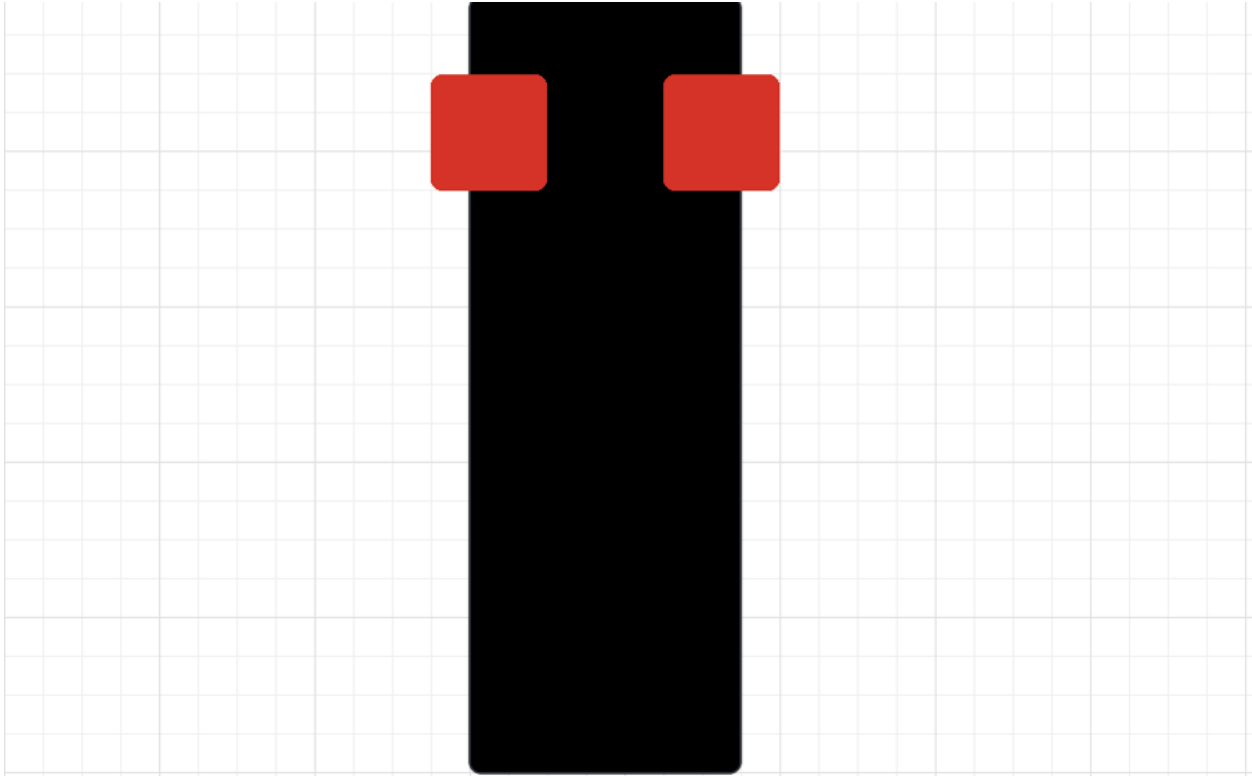


Fig. 14: Simplified diagram of the reflectance sensors centered on a line.

Notice how both reflectance sensors are mostly on the line, but not fully. Just like at the start of the module, you'll need to do some measurements of what the line sensors read. This time, you'll need to see what they read when the robot is centered on the line. This is how the robot would be positioned on the line if it was following it accurately.

Try it out

Write an infinite loop to log the values from both line sensors using the webserver. Place your robot centered on the line and see what the values are.

Use the values you measure to determine a new threshold value. Note that this threshold will probably not be close to halfway between the white and black values, as your robot's sensors may not actually be centered perfectly on the edges of your line, as this depends on the width of the line you are using. A good value for this threshold would be about halfway between the value you just measured and the maximum value you measured back in the first module.

Tip: If you don't remember your sensor's maximum value, just place the robot sideways on the line so that both line sensors are over the line. Use the same code you just used to read the values again.

Consider what the line sensor would see when it crosses an intersection:

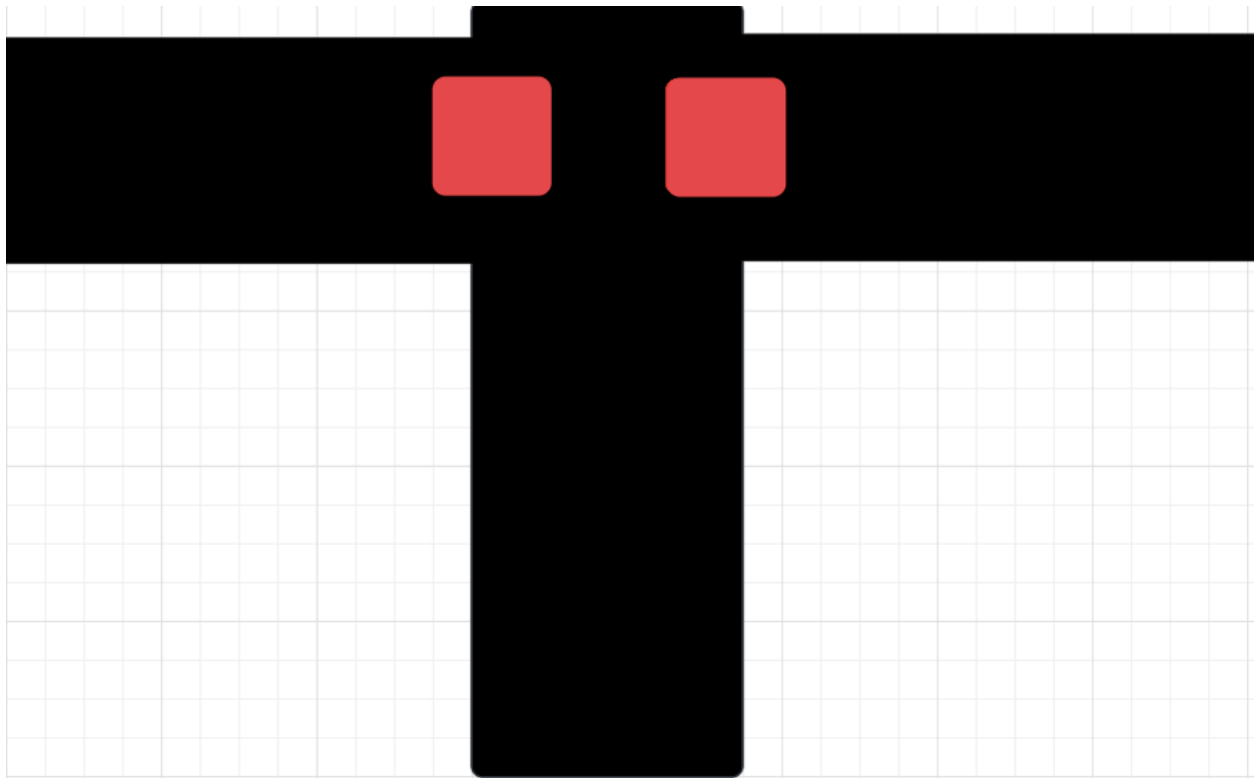


Fig. 15: Simplified diagram of the reflectance sensors over an intersection.

When crossing an intersection, the sensor sees nothing but black. This means that when **both sensors** see **completely black**, we can assume we are at an intersection.

Try it out

Write a function `is_at_intersection()` which reads the values of both line sensors and returns `True` if the robot is at an intersection.

Now that you have a way to detect an intersection, it's time to tie everything together.

Try it out

Write a function `follow_line_until_intersection()` which runs your line following algorithm until an intersection is detected, and then stops the robot.

1.35.1 Challenge

Try writing code to turn at an intersection. Pick a direction to turn, and then have your robot turn that direction until it is over the other line of the intersection, and then start following that line

Tip: You will need to have your robot's wheels centered over the intersection for this to work. Use one of the `drivetrain` functions to do this after you arrive at the intersection.

1.36 Parking Garage Challenge! (Bonus Activity)

Now that we've learned how to follow lines using a proportional controller, let's try to apply this knowledge to a more complicated scenario: a parking garage!

1.36.1 The Goal

The primary goal of our robot is to successfully find and park in an empty space.

To accomplish this goal, we can break the problem down into 2 smaller steps:

1. Find an empty parking spot
2. Properly park in the empty parking spot

1.36.2 Finding an Empty Parking Spot

The main process we will employ here is simple:

1. Go forward the length of a parking spot
2. Turn 90 degrees right and check to see if the parking spot is open
3. If the parking spot is not open, turn 180 degrees left and check to see if the parking spot on the left is open
4. If both parking spots are not open, turn back to straight using the line following `understanding_the_sensor`

In order to see if a parking spot is open, we can use our ultrasonic range finder and see if there is any object in a parking spot.

1.36.3 Leveraging the Line Following Sensors

In this activity, we can use our line following sensors for three main purposes:

1. To follow the line on the ground to the next parking spot
2. To detect the "end" of the parking spot
3. To turn back to straight after checking the left parking spot

In terms of following the line, we can use the same proportional controller that we used in the previous activities.

As for detecting the end of the parking spot, we can use the same logic that we used when detecting an intersection.

Finally, to turn back to straight, we can turn the XRP clockwise until the left line following sensor detects the line.

By breaking this complicated problem down into a series of smaller steps, we can easily program our XRP to park itself!

1.37 Manipulation: Module Overview

In this module students will:

- Understand the basics of manipulation and its application
- Learn how to control their XRP's robot arm

At the end of this module, students will be able to:

- Write a quick program for controlling the robot arm
- Implement custom functions for controlling the robot arm
- Manipulate their environment using the robot arm

1.37.1 Covered Standards (NGSS and CSTA):

HS-ETS1-2 Break a complex real-world problem into smaller, more manageable problems that each can be solved using scientific and engineering principles.

3A-AP-16 Design and iteratively develop computational artifacts for practical intent, personal expression, or to address a societal issue by using events to initiate instructions.

3A-AP-17 Decompose problems into smaller components through systematic analysis, using constructs such as procedures, modules, and/or objects.

3B-CS-02 Illustrate ways computing systems implement logic, input, and output through hardware components.

3B-AP-10 Use and adapt classic algorithms to solve computational problems.

3B-AP-11 Evaluate algorithms in terms of their efficiency, correctness, and clarity.

3B-AP-15 Analyze a large-scale computational problem and identify generalizable patterns that can be applied to a solution.

3B-AP-16 Demonstrate code reuse by creating programming solutions using libraries and APIs.

1.38 Introduction to Manipulation

Now that we've covered different ways of controlling your robot's movement, let's cover how we can control the robot's arm to manipulate our environment.

1.38.1 The XRP Arm

Every robot needs to be able to interact with its environment.

Your XRP does this with a simple 1 DOF arm.

In this case, "DOF" stands for "Degree of Freedom" and refers to the number of ways the arm can move.

1.38.2 Accurate Control of the XRP arm

To move your XRP's arm, use this function:

```
servo.set_angle(angle)
```

In this case, the function takes in a single argument, which is the angle in degrees that you want the arm to move to. The angle has a range of [0, 135] degrees.

Try it out

Try writing code that moves the arm to an angle of 90 degrees, sleeps for 1 second, and then moves the arm back to an angle of 0 degrees.

1.38.3 Other Robotics Manipulators

There are many types of robotic manipulation. [Here](#) is a very complicated example, where Boston Dynamics' Spot robot opens a door and lets itself in.

The XRP arm is a very simple manipulator.

Most robots have more complex manipulators, with more degrees of freedom.

For example, Boston Dynamic's Spot robot has a 5 DOF arm that can be used to open doors, turn valves, and even pick up objects.

For more complicated manipulators like the one that Spot has, roboticists often have to create control laws specific to those manipulators.

In the case of Spot, the arm is equipped with a camera that help Spot better understand it's environment and avoid obstacles while trying to use it's arm.

1.39 Picking up a Basket

Now that we've covered how to move the XRP arm to specific angles, we can start to think about how to use the arm to pick up objects. In this section, we'll cover how to use the arm to pick up a basket which is one of the challenges for your final project described in detail in the next module.

1.39.1 The Process

Let's first think about what the process of picking up a basket would look like. Imagine the a small paper cup with a bail attached over the top to that you can hook with the servo arm. An example of a basket is shown in the video below.

An example of the steps required to aquire the basket are:

1. Lower the arm to a height where it will be inside the bail.
2. Back up the robot so that the arm goes inside the bail.
3. Raise the arm to lift the bucket off the ground.
4. Drive away carying the bucket.

Now the robot can drive away while carrying the basket.

Try placing your robot in front of a basket and then writing a program to pick it up as shown in the following video.

1.39.2 Integrating Locating and Pickup

Now that we've covered how to pick up a basket, we can start to think about how to integrate this with code that we have previously written to locate a nearby object.

To do this, you will re-use the code that you have written to how execute the pickup process when the robot is in front of the basket.

1.40 Intersection into Drop Off

Now that we've learned how to detect an intersection and develop custom functions, let's try to put it all together to create a program that can drop a basket off at a designated location.

1.40.1 The Process:

Let's first think about the process:

We want to first detect an intersection and know that we can accomplish this by waiting for both line sensors to detect a line.

Once we detect an intersection, we want to turn 180 degrees to ensure that we are facing the correct direction to drop off the basket.

Then, we want to travel a certain distance to ensure that we are at the correct location to drop off the basket.

Once we are at the correct location, we want to drop off the basket.

Then, we want to drive back to the intersection. Since we are already facing "backwards", this means that we don't have to correct our orientation before returning to the intersection.

Finally, we want to put the arm back into the starting position and start line following again.

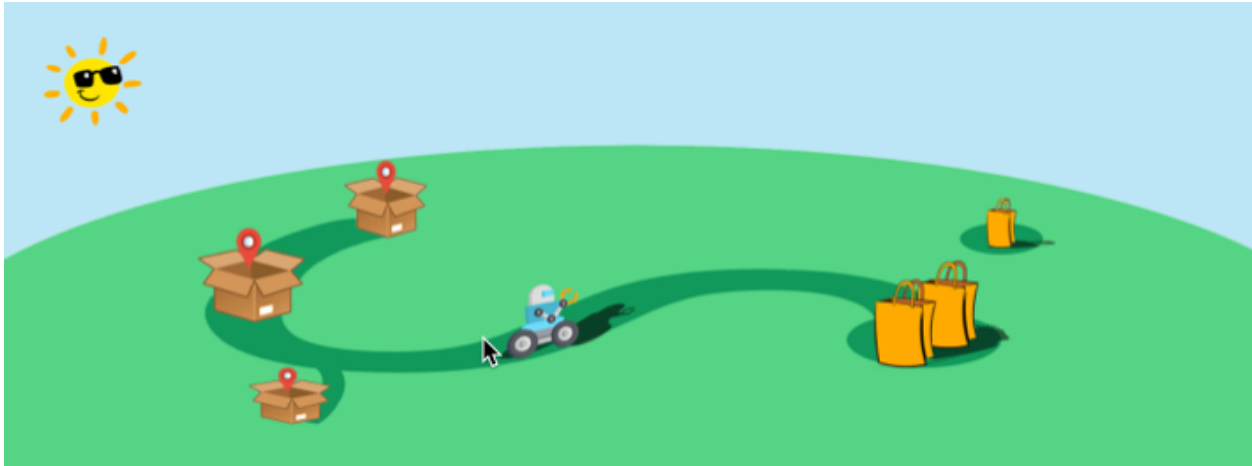
Put together, this process can be broken into these steps:

1. Detect an intersection
2. Turn 180 degrees
3. Travel our desired distance
4. Drop off the basket
5. Drive back to the intersection
6. Put the arm back into the starting position
7. Start line following again

We can use the functions we've already created to do this:

Error: TODO add code and a video

1.41 Delivery Challenge: Module Overview



The delivery robot has to bring packages from one location to another. Introduction To help people in need in the current pandemic, you and your colleagues have decided to build an autonomous delivery robot.

Your robot will pick up food and other supplies and deliver them to residences with minimal contact, all to help fight the spread of the coronavirus.

Of course, a full-sized version will have to wait until you get millions of dollars in investments for your robotics company, but that doesn't mean you can't have fun dreaming with a scale model here.

Your robot will have an arm for lifting and carrying "bags" of goods and sensors to help it navigate a simple network of "streets." Complicating things a little, not everything to be delivered is always put in the right spot for easy collection and sometimes road construction might block your path. But you'll still need to make it work!

The world is counting on you! Can you make it happen?

1.41.1 Objectives

The final project is a chance for you and your teammates to demonstrate that you can apply concepts and strategies from the course to a specific challenge. You will apply theoretical knowledge to the design of your system and use focused testing to improve the performance. The project will culminate in a demonstration where you will prove your robot's performance. To help us understand more about your system and your process, you will also produce a report describing the system development and an assessment of how well it met your goals.

The successful team will design, build, and demonstrate a robot that can accomplish a prescribed set of tasks. To be successful, you will need to:

Identify key performance criteria and develop a strategy for meeting your team's objectives, Identify key factors that affect performance and use analysis and testing to specify them, Develop and apply a testing strategy to ensure performance, Evaluate the system performance, and Describe the system and your design process.

1.41.2 Challenge

Your challenge is to program your robot to pick up bags of supplies from known and unknown locations and deliver them to specified delivery points. The challenge is constructed such that the tasks have a range of difficulty. For example, locating the free range bag and scoring them will earn more points, as will being able to navigate around the construction sign. Since you will have to perform multiple runs, reliability will be essential.

1.41.3 Course

The course will consist of a strip of tape (to simulate roads) with a designated place to pick up bags and three specified drop zones. Most bags will be placed at the end of the main road, though some will be placed in a “free range” zone. Figure 1 shows a typical arena, though we realize that there will be some variation in each course.

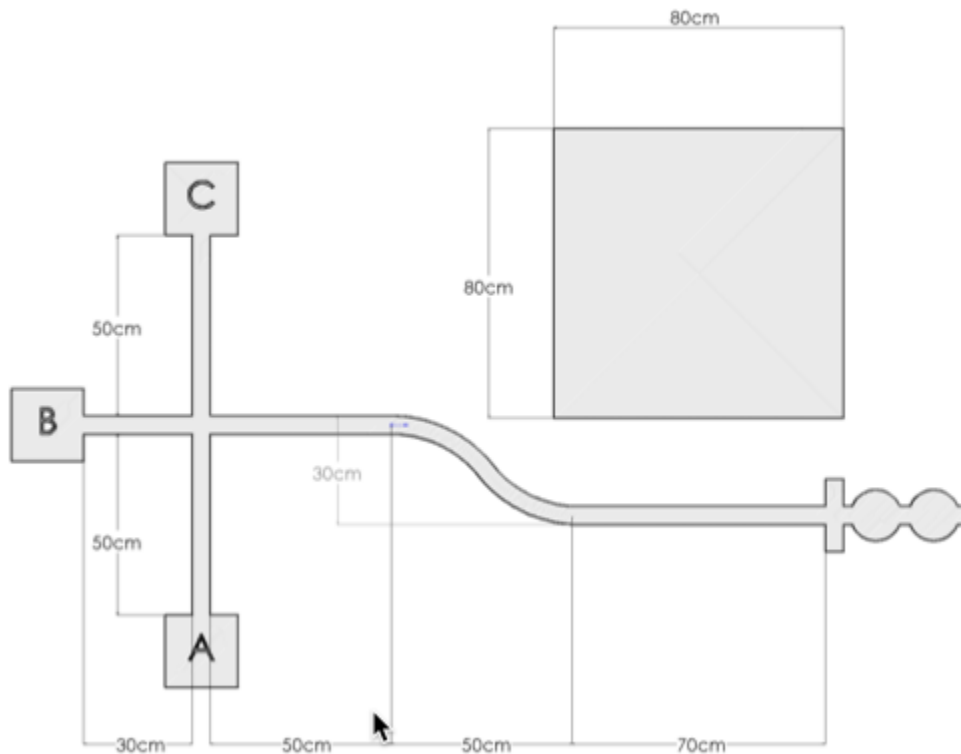


Figure 1: Diagram of the arena. Individual arenas will vary.

1.41.4 Bags

You will be expected to build your own delivery bags, for example from paper or card stock and paper clips. You should have at least two bags ready for the demo. You may “recycle” them as the demo progresses. Here is an example of an easy to make bag that can be constructed from a drink disposable cup you might find in a restaurant.



Figure 2: Example of a bag for the delivery challenge

1.41.5 Collection

Most of the bags will be placed on the line at the end of the main road. You may place a piece of tape near the pickup zone to indicate where it starts, but the bags will be placed at different distances from the tape.

To earn points for collecting the free-range bag, you must demonstrate that its position can be arbitrary within the free-range zone (with the exception that you may face the bail in whatever position you deem most favorable).

1.41.6 Delivery zones

Each delivery zone or platform may be no larger than 10 cm in any horizontal dimension. The platforms for the delivery zones will be marked on the ground. You can make them out of cardboard or any other material. To score points, each container must be placed in a delivery zone and left there (upright) long enough to prove that it is stable.

1.41.7 Operation

You will start with your robot on the main road and a bag in the pickup zone. On command (a press of either button on the robot), your robot will drive to the pickup zone, pick up a bag, and deliver it to the specified address, which will be determined by the button press (e.g., 'GP20' indicates a delivery to address A, etc.). Your robot will then return to the starting point, stop, and wait for the next command.

After each delivery, you will place another bag in the pickup zone and repeat the process, pressing the button to indicate another delivery. You may recycle the bags as much as you wish.

At two random points in the demo, your instructor will place a road construction sign on the main road for 30 seconds, and you will not receive credit for any delivery during which your vehicle hits the sign.

The challenge will last 5 minutes. You may use any of the sensors that you've explored in this class to accomplish the challenge. Line following will be an important behavior, but collecting the free-range bags autonomously will require some creativity on your part.

[Video of a sample run coming soon!]

1.41.8 Scoring

[Tune point values over time]

In your run, your team should deliver as much weight as possible, plus and including the “free range” bags. Points will be allocated as follows:

You will receive 5 point for every package you deliver to addresses A or C. However, you may only get 50 points max (corresponding to 10 packages) for each delivery address; i.e., you must deliver to all three addresses to receive the maximum points. You will receive 5 additional points for each free-range bag (20 points max) scored at address B. Your total score will be multiplied by the number of unique addresses you delivered bags to. That is to say, if you scored 1 bag to Address A, and 1 bag to Address B, your final score would be $2 * (5+5) = 20$ points. If you scored a free range bag on top of that, your score would be $3 * (5+5+5) = 45$ points. No points will be received for a delivery where the robot hits the road construction sign. You will lose 2 points for each time you have to touch your robot (e.g., to put it back on the line), other than to specify the delivery zone at the start of each delivery.

1.41.9 Covered Standards (NGSS and CSTA):

HS-ETS1-2 Break a complex real-world problem into smaller, more manageable problems that each can be solved using scientific and engineering principles.

HS-ETS1-3 Evaluate a solution to a complex real-world problem based on prioritized criteria and tradeoffs that account for a range of constraints, including cost, safety, reliability, aesthetics, and maintenance, as well as social, cultural, and environmental impacts.

3A-AP-16 Design and iteratively develop computational artifacts for practical intent, personal expression, or to address a societal issue by using events to initiate instructions.

3A-AP-17 Decompose problems into smaller components through systematic analysis, using constructs such as procedures, modules, and/or objects.

3B-CS-02 Illustrate ways computing systems implement logic, input, and output through hardware components.

3B-AP-10 Use and adapt classic algorithms to solve computational problems.

3B-AP-11 Evaluate algorithms in terms of their efficiency, correctness, and clarity.

3B-AP-15 Analyze a large-scale computational problem and identify generalizable patterns that can be applied to a solution.

3B-AP-16 Demonstrate code reuse by creating programming solutions using libraries and APIs.

1.42 Using the XRP Webserver

1.42.1 Introduction:

In this section, you will learn about web servers and how to use them to wirelessly communicate with the XRP. A Webserver is a computer program that takes instructions from another computer over the internet. They are used to display web pages, run video games, send messages, and do almost anything else you can do on the internet. Using the XRP's built-in Webserver, you can wirelessly display values on your computer and even send instructions to the XRP with just the press of a button. With this functionality you can easily debug programs and even use your phone or computer to remotely control your XRP.

Writing programs that use the webserver is simple. If you are using python, the webserver is included in your `from XRPLib.defaults import *` call. If you are using blockly, this will be done for you as well.

Our webserver supports a bunch of useful logging and debugging tools you can use. This includes adding buttons to call functions that you write directly from the webserver, which allows for remote control of your robot, or logging variables to the webserver as your program runs, allowing you to see your values change in real time.

For now, we will focus on the basics of using the webserver and getting it to start, and in the next section we will go over how to use its full functionality.

1.42.2 Starting the Webserver:

There are two main modes that the webserver can run in. The first is called “bridge mode”, and the second is called “access point mode”. In bridge mode, the XRP will connect to your home wifi network and use that to communicate with your computer or mobile device. In access point mode, the XRP will create its own wifi network that you can connect to with your computer or mobile device.

Note: The XRP can only connect to 2.4GHz wifi networks. If you are using a 5GHz network, you will need to switch to a 2.4GHz network.

To start the webserver in bridge mode, you will need to know the name and password of your home wifi network. In the root directory of your robot, you will find a file called `secrets.json`. This is where you can safely store your wifi network name and password to be used by the webserver. An example of this file can be seen here:

```
{
  "wifi_ssid": "YOUR_WIFI_SSID",
  "wifi_password": "YOUR_WIFI_PASSWORD",
  "robot_id": 1,
  "ap_ssid": "XRP {robot_id}",
  "ap_password": "remote.xrp"
}
```

In this file, you will create a field called `wifi_ssid` and `wifi_password`. If you are using bridge mode, this is where you will put your wifi network name and password. If you are using access point mode, you will need to use the `ap_ssid` and `ap_password` fields instead. The `robot_id` field can be used to give your robot a unique name when it is in access point mode, and will replace the `"{robot_id}"` in the `ap_ssid` field. Using this json file is not necessary, and these fields can be later specified in either python or blockly.

Starting the webserver in access point mode is simple. All you have to do is call the `start_network` function, followed by the `start_server` function. If you didn't configure the `secrets.json` file, you will need to specify the access point name and password in the `start_network` function. Otherwise, you can leave these fields blank and the webserver will use the values from the `secrets.json` file.

In bridge mode, it's almost exactly the same, except you will call the `connect_to_network` function instead of the `start_network` function. Similarly, if you didn't configure the `secrets.json` file, you will need to specify the network name and password here.

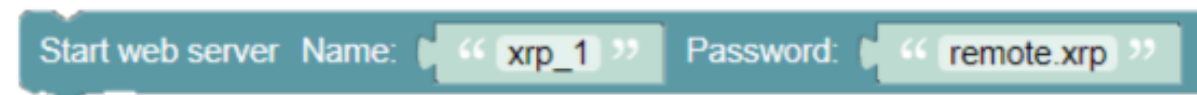
Python

```
webserver.start_network(ssid="XRP_{robot_id}", robot_id=15, password="remote.xrp")
webserver.start_server()
```

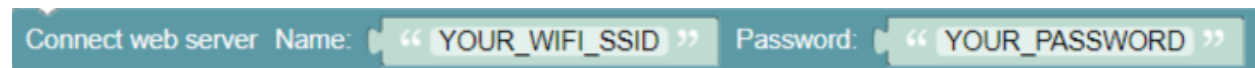
OR

```
webserver.connect_to_network(ssid="YOUR_NETWORK", password="YOUR_PASSWORD")
webserver.start_server()
```

Blockly

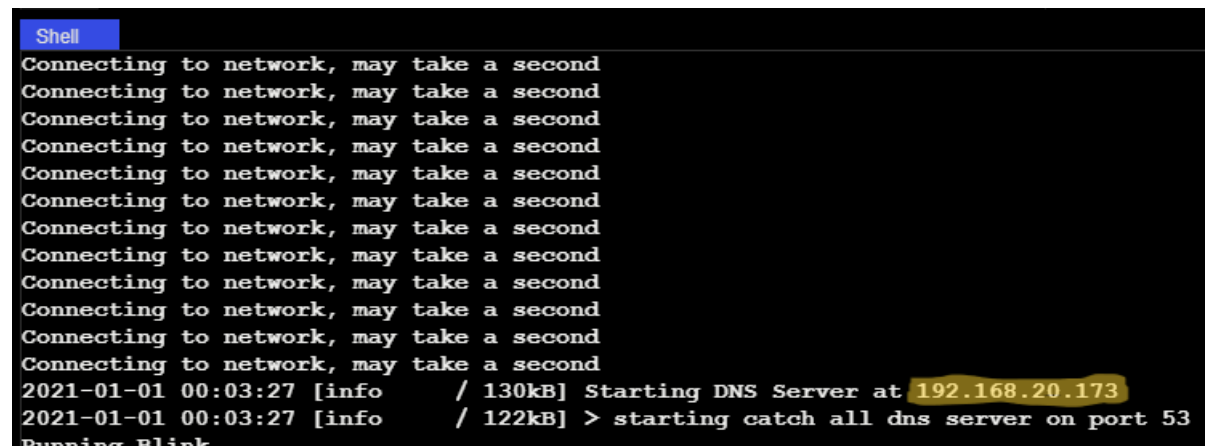


OR



If you are using access point mode, search for wifi networks on your phone or computer, and join the wifi network with the name and password you chose.

Finally, open a new page on your browser. In bridge mode you will have to navigate to the IP given when the webserver starts up (highlighted below). You should see your new custom dashboard! For now, it will look pretty empty, but we will add more to it later.



Note: Once you start the server, the XRP program takes control of program execution, and will not return until the webserver is stopped. Make sure that anything you want to do is done before you start the webserver. This may change in a future update.

Now you know how to start your web server. Next, you will learn how to use its full functionality.

1.43 Using the Web Server as a Dashboard

The most useful application of the webserver is as a dashboard. The webserver can display live-updated data from the robot, which can be a great debugging tool. Printing values that your robot calculates or senses can help you understand why a bug is occurring because these values might not be what you expect, and the only way to determine them is to read them as the program runs.

You can send a value to the web server with just one line. The `webserver.log_data(label, data)` function will create a label on the webserver with the corresponding data. If you call this multiple times, it will update the label on the webserver with the most recent value, which is how it is live-updated.

Python

```
from XRPLib.defaults import *

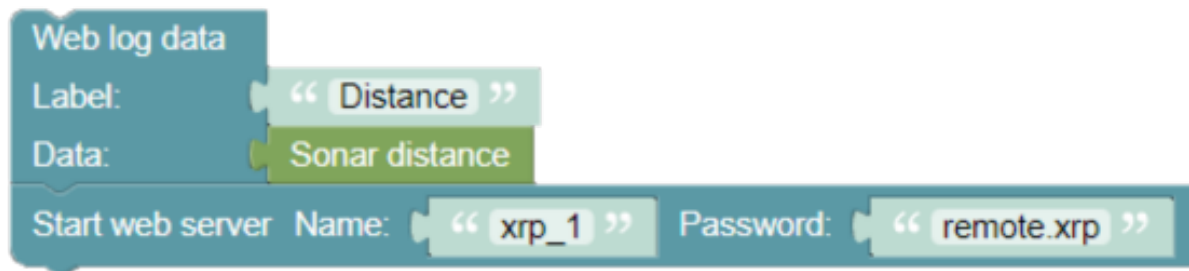
webserver = Webserver.get_default_webserver()

rangefinder = Rangefinder.get_default_rangefinder()

webserver.log_data("Distance", rangefinder.distance())

# Uses defaults from secrets.json
webserver.start_network()
webserver.start_server()
```

Blockly



Now, after running this code, you may have noticed that it isn't live-updating. This is because the `log_data` function is only being called once. Your first instinct may be to put this in a loop, but that won't work because the webserver takes control of the main event loop. To get your data updating automatically, we can use the MicroPython `Timer` class.

Python

```
from XRPLib.defaults import *
from machine import Timer

webserver = Webserver.get_default_webserver()

rangefinder = Rangefinder.get_default_rangefinder()
```

(continues on next page)

(continued from previous page)

```
def update_distance(timer):
    # We move the log_data call into this function so that it is called every time the
    ↪ timer is triggered
    webserver.log_data("Distance", rangefinder.distance())

# Timer(-1) creates a timer that is not attached to any hardware, so it is purely a
    ↪ software timer
timer = Timer(-1)
# The timer is set to trigger the update_distance function every 1000 milliseconds (1
    ↪ second)
timer.init(period=1000, mode=Timer.PERIODIC, callback=update_distance)

# Uses defaults from secrets.json
webserver.start_network()
webserver.start_server()
```

Blockly

This is not yet supported in Blockly.

This will now update the distance every second. You can change the period to whatever you want. One thing to take note of here is that the function that is called by the timer must take one parameter, which is the timer itself. We don't use this parameter in this example, but it is required to be there.

Now you can use this to debug your robot. You can add as many labels as you want, and in the future there will be more logging options available.

1.44 Remotely Controlling your XRP

With the webserver, you can also control your XRP remotely. The webserver class has a few methods that allow you to register functions to be called when an arrow button is pressed, which we can use to control the XRP.

Below is an example of how to register some basic drive functions to be called when the arrow buttons are pressed:

Python

```
from XRPLib.defaults import *

def forward():
    differentialDrive.set_effort(0.5, 0.5)

webserver.registerForwardButton(forward)

def back():
    differentialDrive.set_effort(-0.5, -0.5)

webserver.registerBackwardButton(back)

def left():
    differentialDrive.set_effort(-0.5, 0.5)
```

(continues on next page)

(continued from previous page)

```
webserver.registerLeftButton(left)

def right():
    differentialDrive.set_effort(0.5, -0.5)

webserver.registerRightButton(right)

def stop():
    differentialDrive.stop()

webserver.registerStopButton(stop)
```

Blockly

These functions are then registered to the webserver, and when the arrow buttons are pressed, the corresponding function is called. The arrows will appear if any of these functions are registered, and will be disabled if they are not.

Note: You can also use lambda functions to register functions to the webserver, which can be useful for simple functions, but are out of scope for this lesson.

1.45 Creating Custom Webserver Buttons

Last lesson, you learned how to bind simple drive commands to the arrow buttons. But, what if you want to perform more complicated code tasks? or what if you have more than 5 commands you want to run from the webserver?

Well, XRPLib provides the tools to do that! With the `webserver.add_button(label, func)` method, you can add any function you want to the webserver, along with a helpful label that describes what your function does!

Python

```
from XRPLib.defaults import *

def raiseArm():
    servo_one.set_angle(100)

def lowerArm():
    servo_one.set_angle(0)

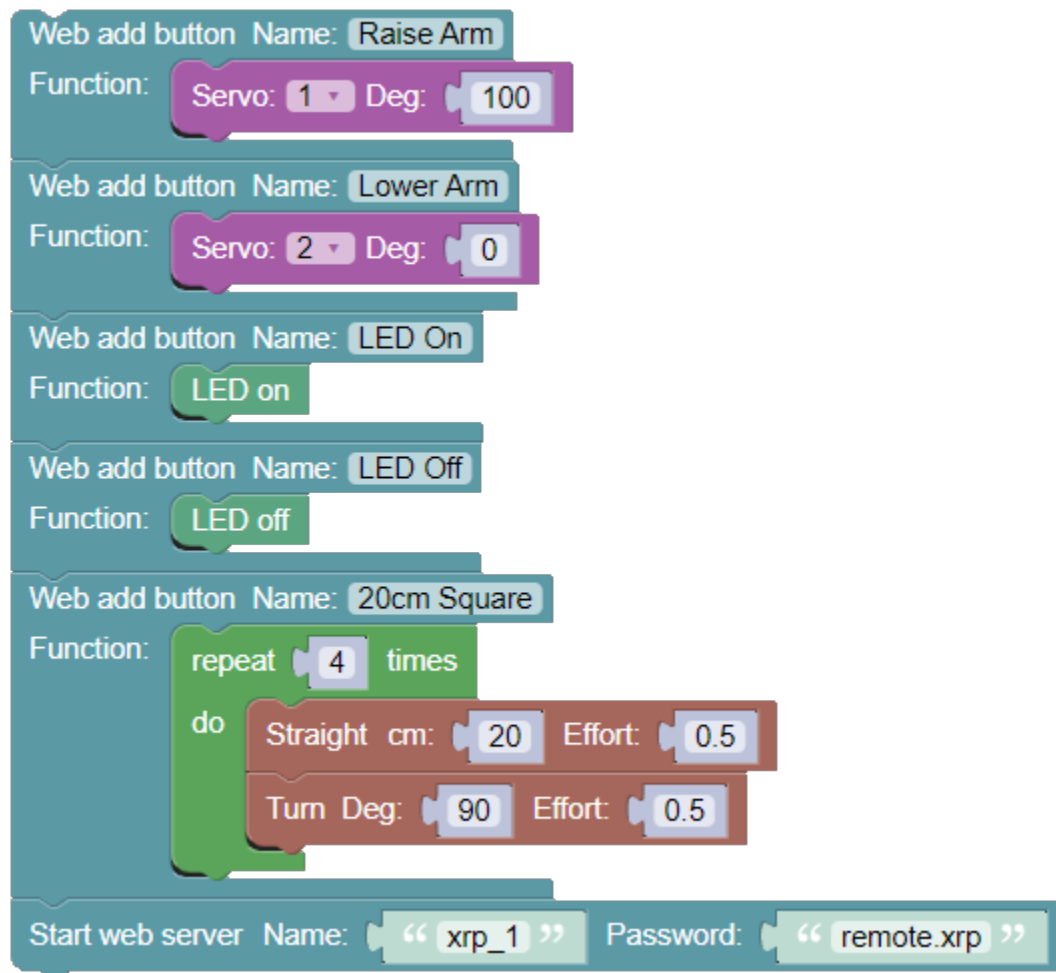
def led_blink():
    board.led_blink(5)

def led_stop():
    board.led_off()

def square():
    # Drives in a square with 20cm sides
    for side in range(4):
        drivetrain.straight(20)
        drivetrain.turn(90)

webserver.add_button("Raise Arm", raiseArm)
webserver.add_button("Lower Arm", lowerArm)
webserver.add_button("Blink LED", led_blink)
webserver.add_button("Stop LED", led_stop)
webserver.add_button("20cm Square", square)

webserver.start_network()
webserver.start_server()
```

Blockly

After running this code, you should see your new button(s) appear under the Custom Buttons section of the web page!
Note: We recommend leaving your robot plugged in the first time you run a new function on the webserver, just to make sure that it doesn't have any errors.

And there you go! That's all the basics of how to use the webserver!

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`