UNIVERSITY OF WATERLOO
AI TRAINING

Project Module

UNIVERSITY OF WATERLOO | FACULTY OF ENGINEERING

ROBOT CAR PROJECT

8/20/2025

Salman Lari, PhD

UNIVERSITY OF WATERLOO | FACULTY OF ENGINEERING

Course Overview

- Two components: lectures and tutorials
  - During the lectures, you will hear about core technologies involved in robotics
  - During the tutorials, you will implement some of these technologies into a project

- The tutorials, and project generally, are semi-guided
  - We will provide a timeline you should probably follow to ensure you can get everything done on time, but there is flexibility in the control methods you choose to implement

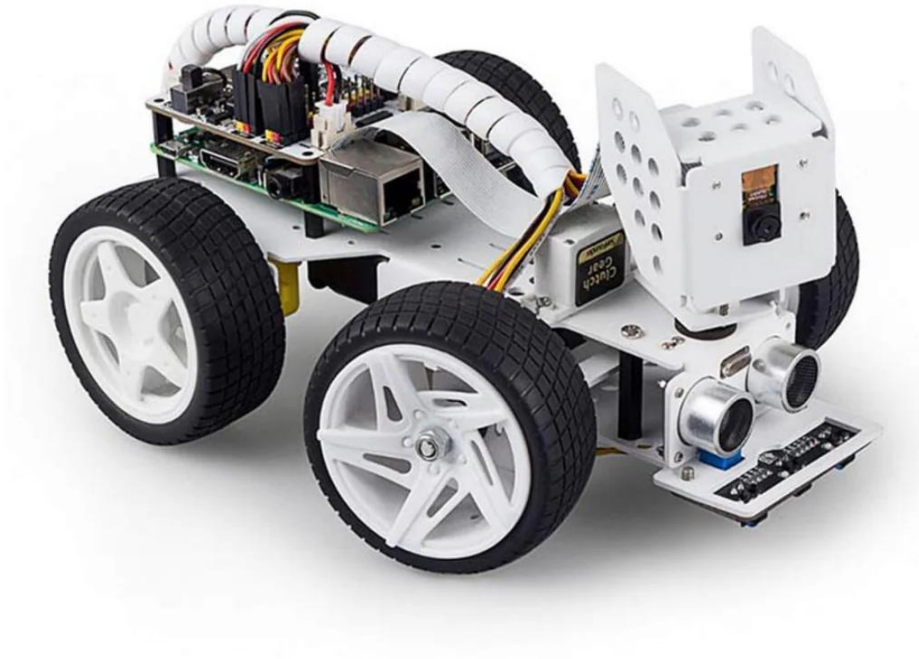
Course Contents + Project Schedule

Course Content	Tutorial Content	Dates
Sensor and Control Hardware	Assembly of Robot Kit	Monday August 18 Tuesday August 19
Robotics Control Software	Linux setup on Robot/Raspberry Pi	Wednesday August 20
Environmental Interaction + Machine Learning	Sign Recognition Setup	Thursday August 21
Planning of Trajectories	Navigation Routine Setup	Friday August 22
1-on-1 project help	Additional work day + tuning	Monday August 25
Demonstration Day	Demonstration of Project	Tuesday August 26

Project Overview and Key Learning Objectives

- Build a robot car that can navigate a traffic grid using sign recognition and avoid obstacles using proximity sensors

- Key learning objectives:
  - Construction of basic electromechanical systems
  - Configuration of Linux-based single board computers
  - Writing motor control code for robotics applications
  - Understanding of machine learning principles (classification)
  - Understanding of navigation techniques for autonomous mobile robots

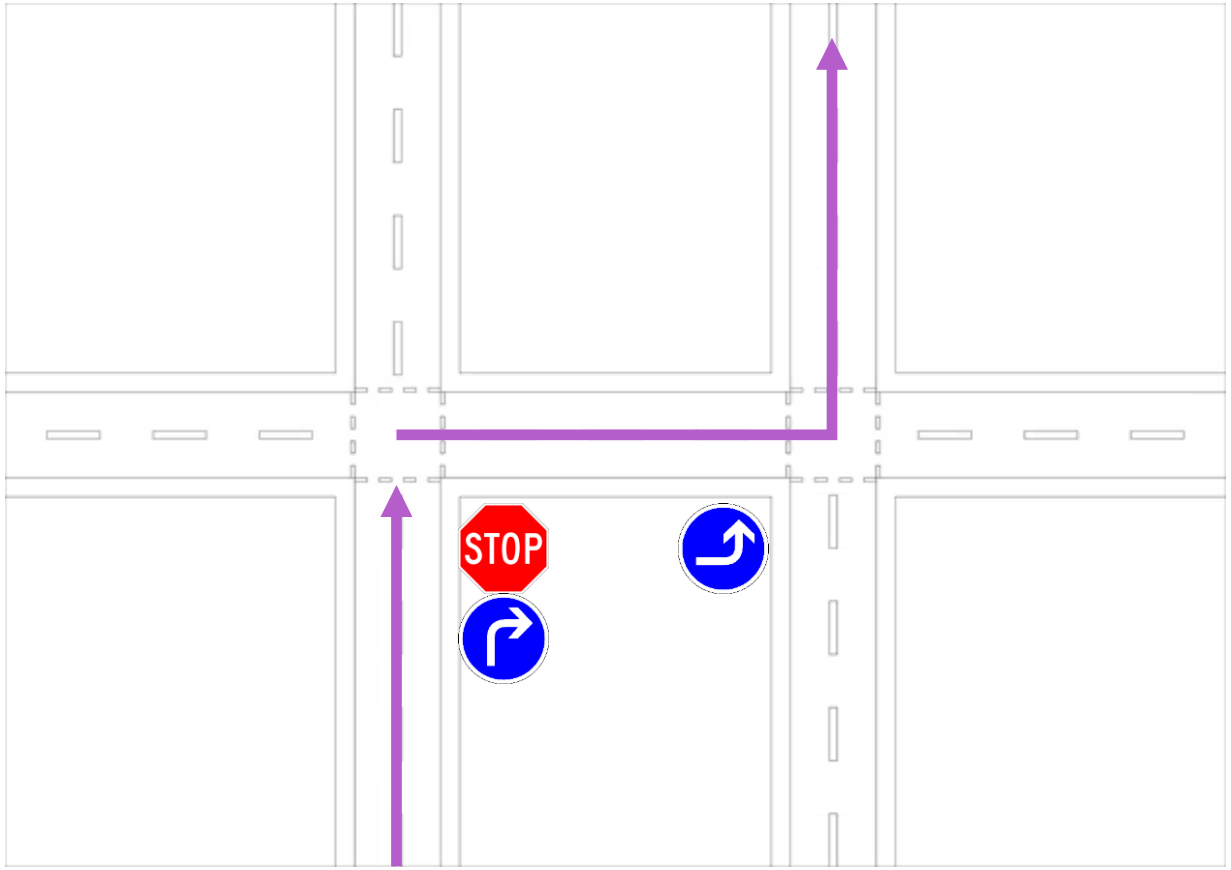


Project Group Division

- Divide yourselves into groups of 2-3
- Suggestions to work quickly on the project:
  - Half of the members work on assembly, the other half are working on the various software components
  - Start training the machine learning components early, in case of issues!

Project Guidelines

- You can choose the sensor technologies and control strategies you want to use in following the signs on the route
- The final tutorials will be dedicated to a competition between groups to see whose cars are most capable of completing the navigation
- Scoring points for following all signs, lack of manual intervention, and high speed



SENSOR AND CONTROL HARDWARE

Lesson - Section 1

Introduction to Robotics

https://standardbots.com/blog/articulated-vs-pick-and-place-robot-arm-whats-the-difference
https://www.mwes.com/types-of-industrial-robots/cartesian-gantry-rectilinear-robots/
https://www.therobotreport.com/clearpath-mobile-robots-adding-ros-on-windows-10/

- Variety of different types of robotic system design
  - **Articulated**: use multiple rotary joints to accomplish movement along a variety of axes (diverse range of movement)
  - **Cartesian**: also known as gantry robots, use x,y,z coordinate scheme to perform operations such as automated loading
  - **Mobile**: robot itself moves around the environment, rather than manipulating the environment from a fixed position.
    - **Mobile robots are the focus of this course**
- Different design techniques for each type

Key Facilities of Robots

- **Proprioception** - How to recognize relationship of self with environment?
- **Exteroception** – How to interpret stimuli of the surrounding environment (visual, acoustic, tactile)?

- Need to gather information -> sensors (covered now)
- Need to process & interpret collected information (covered later)

Sensor Types

- Sensors are the main way in which we provide the robot information about how it is to accomplish its tasks

- **Inertial information**
  - Accelerometer and motor encoder can be used to infer how position changes versus some starting point; gyroscope can tell us the “pose” of the robot (what orientation it is in)
- **Environmental information**
  - Global position can be determined with systems such as GPS, which rely on satellites
  - Vision sensors: RGB cameras, infrared depth-sensing, LiDAR scanning

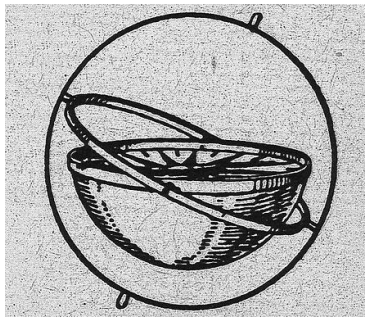
Inertial Tracking - History

https://upload.wikimedia.org/wikipedia/commons/thumb/f/f8/Kardanischer-Kompass.jpg/440px-Kardanischer-Kompass.jpg

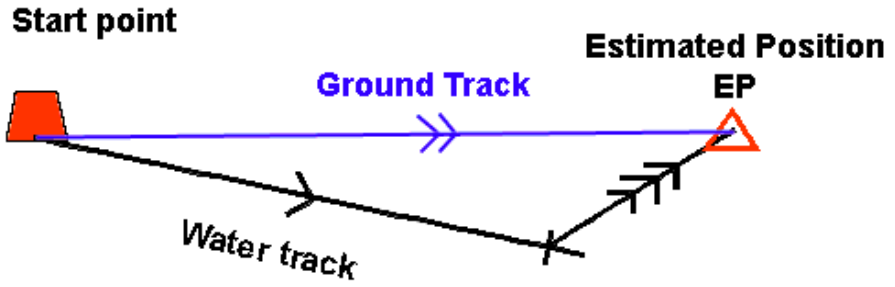
Inertial tracking is how people learned to navigate across oceans.

The compass was invented in the 11th or 12th century. Mariners would tie knots in an anchored rope at fixed intervals and throw it behind the ship as it sailed from shore. Given the length of rope pulled overboard and the compass bearing, the navigator could plot an estimated new position of the ship.

This proves infeasible beyond a certain distance due to the length of rope needed.



Early marine compass

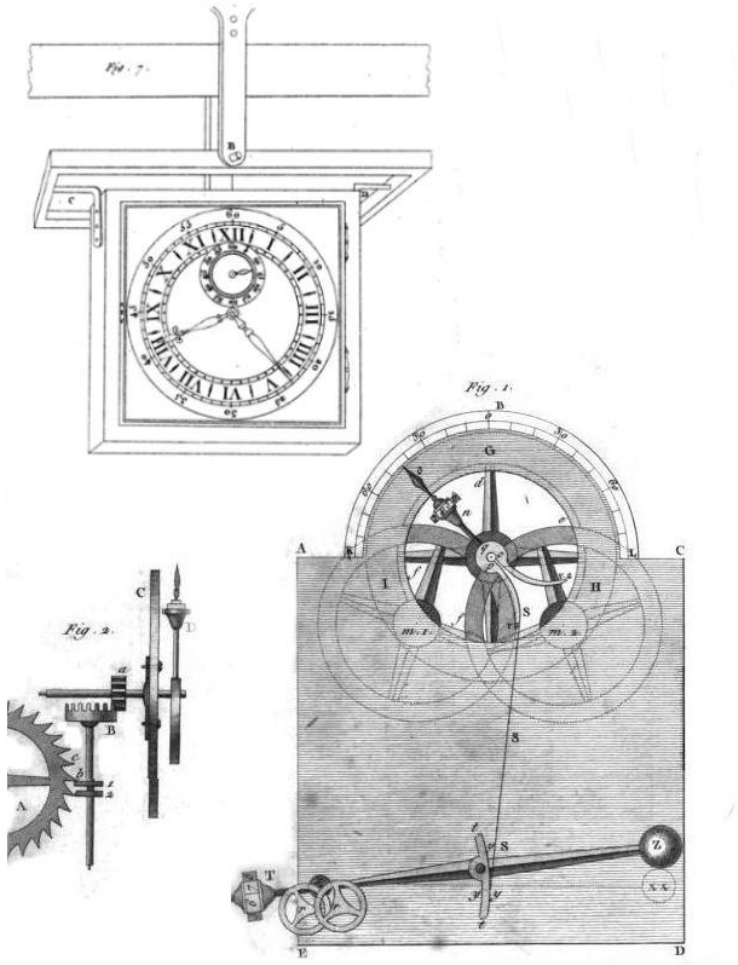


https://upload.wikimedia.org/wikipedia/commons/4/4b/Henry_Sully_clock_with_escapement_and_suspension_mechanism.jpg

Inertial Tracking - History

The invention of timekeeping (in particular, marine chronometers in the 1700's) made it possible to instead measure the *rate of change* of distance, rather than the absolute distance tracked, by calculating the rope pulled overboard per minute.

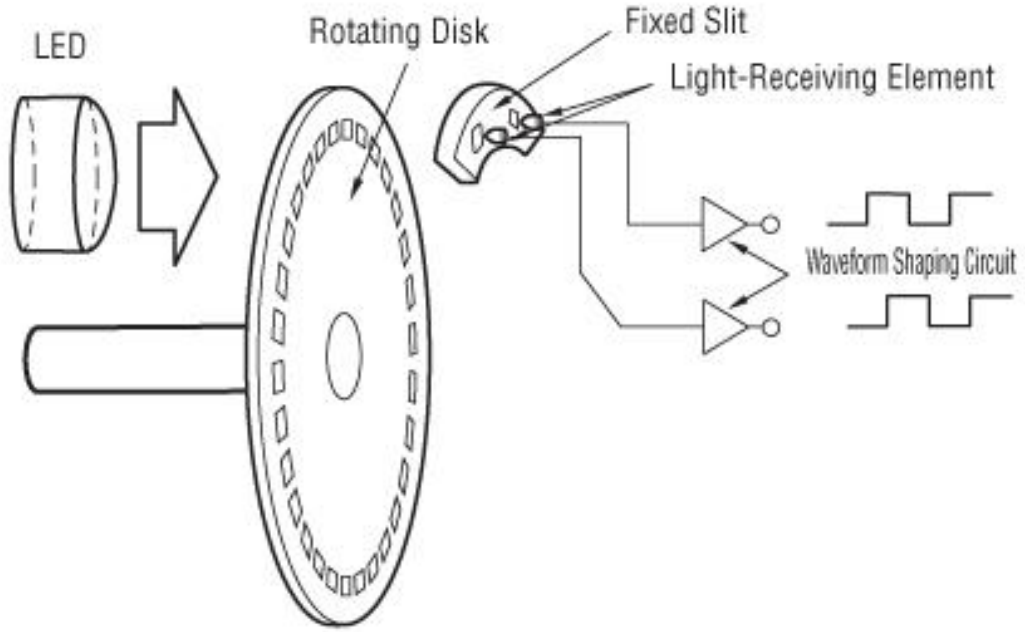
Hence, the trajectory of a ship could be inferred beyond the initial measurements, and detailed plans over long distances could be made.



1716 patent for Henry Sully's marine chronometer

Motor Encoders

- Motor encoders enable a reckoning-based approach to relative position tracking
- Based on the rotations of the wheel (of a known size), we can estimate the relative distance travelled of the mobile robot
- By measuring the time over which this rotation takes place, we can estimate speed, and thus plot trajectories



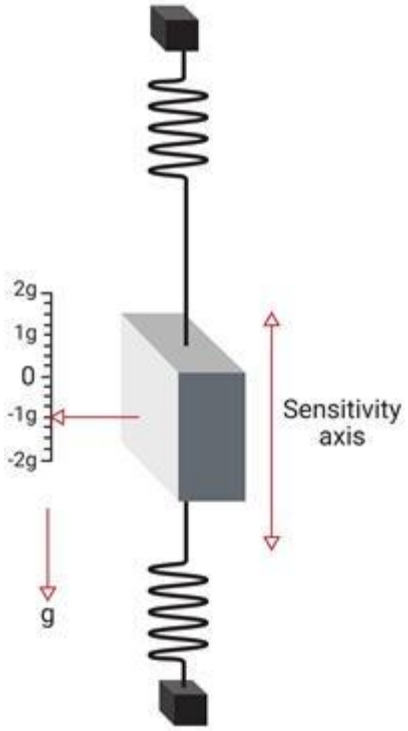
$$\Delta d = \theta r$$

Change in distance is proportional to angle of rotation times radius of wheel

https://www.orientalmotor.com/servo-motors/technology/servo-motor-glossary.html

Inertial Measurement Units (IMUs)

IMUs typically contain both accelerometers and gyroscopes.



Accelerometers measure specific force ("G-force") experienced, which can be used to determine the linear acceleration along each axis.

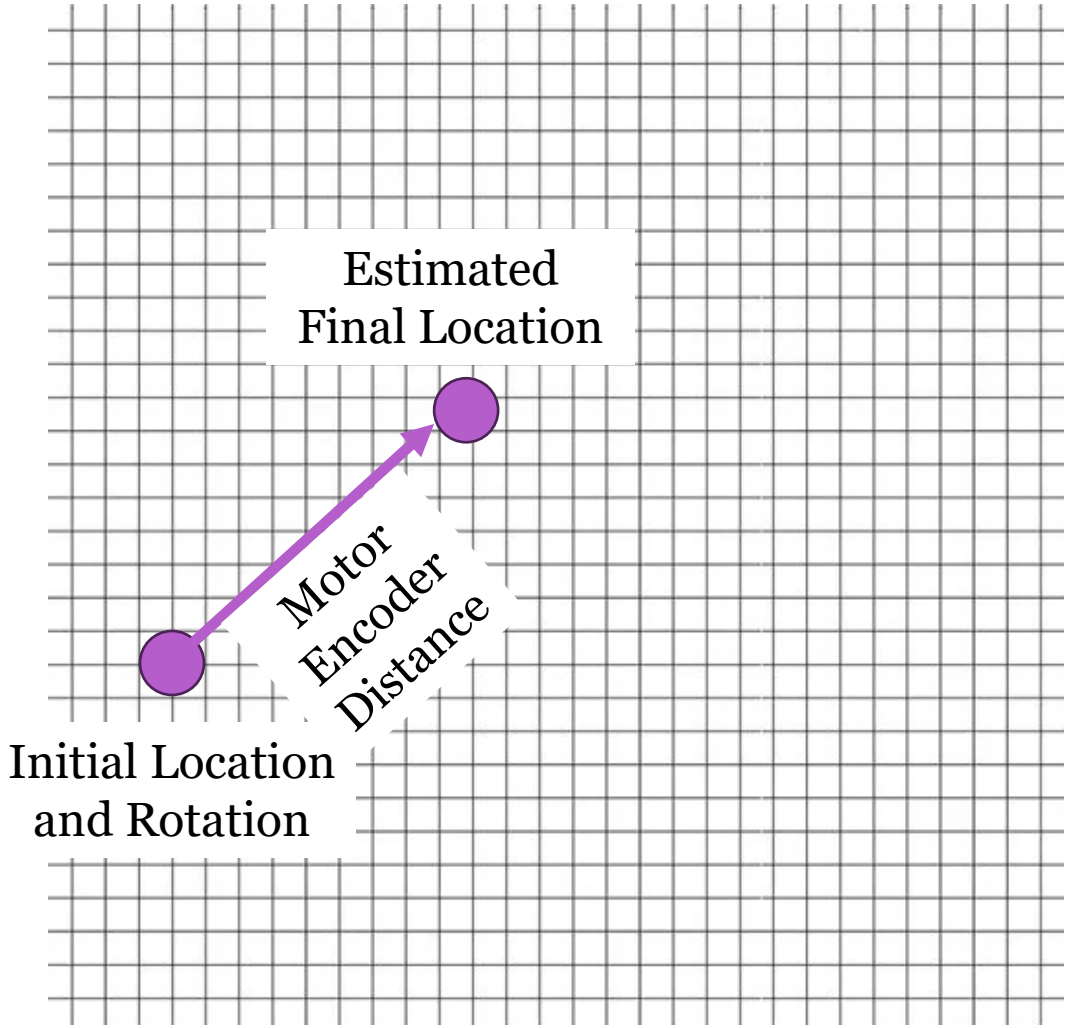
Gyroscopes measure the rotational pose of the robot. The spinning central weight will remain in the same orientation according to the conservation of angular momentum. Hence, changes of the robot's rotational orientation can be measured against this.

https://www.vectornav.com/resources/inertial-navigation-articles/what-is-an-inertial-measurement-unit-imu
https://en.wikipedia.org/wiki/Gyroscope#/media/File:Gyroscope_operation.gif

Sensor Fusion

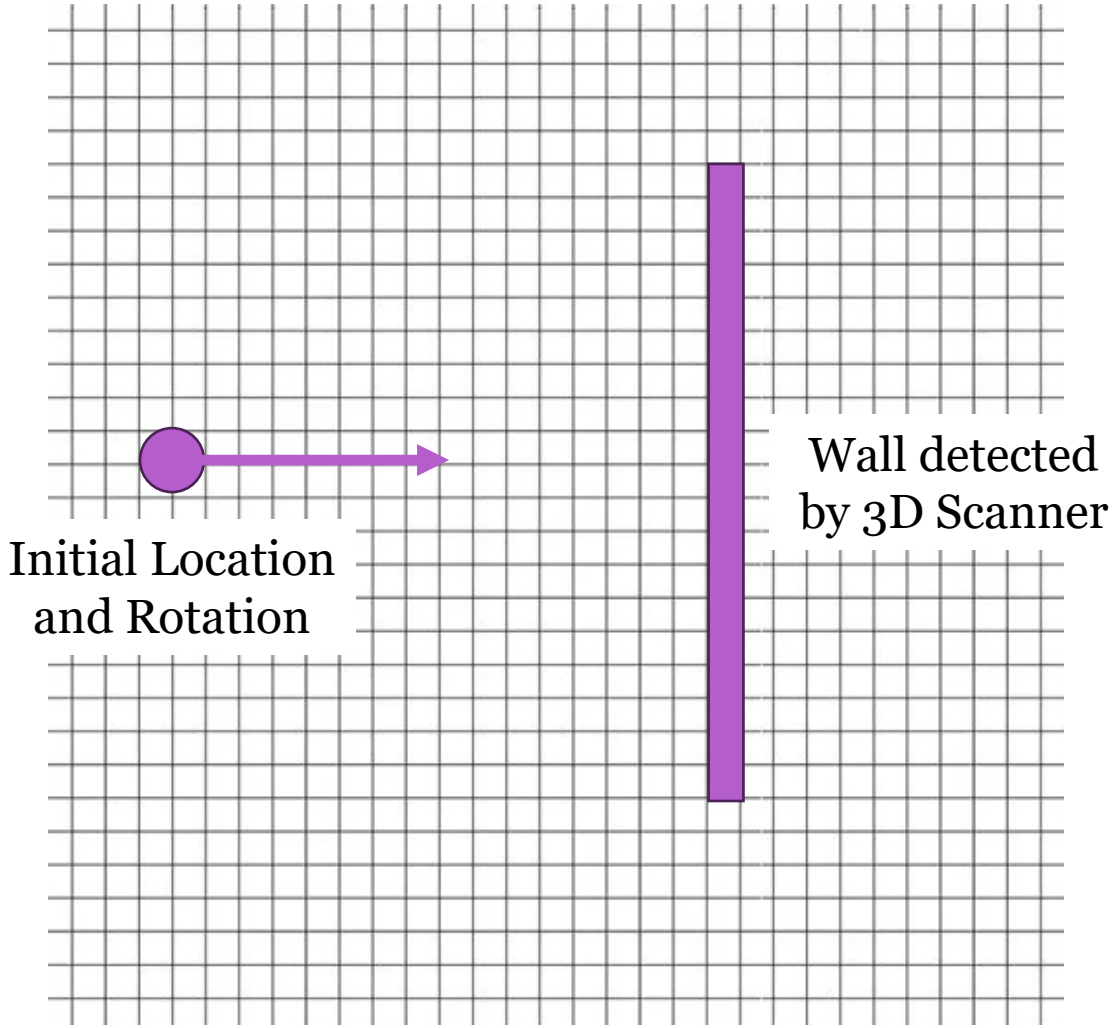
We can combine the encoder data, which measures linear change in position and speed, and the IMU data, which measures angle and acceleration, into a high-quality 2D representation of the robot's location relative to a starting point.

However, this does not tell us anything about the environment, and any obstacles that might exist in it.



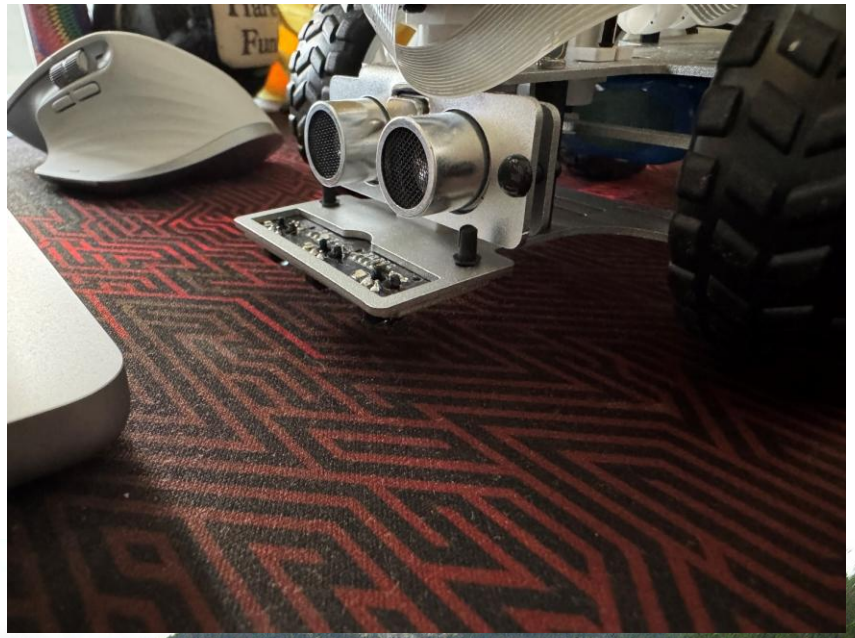
Exteroception Sensors

- Depth sensing cameras (LiDAR, infrared)
  - We can infer that a certain part of our ground grid is occupied
- Regular cameras (RGB)
  - Harder to infer obstacle locations, but useful for classifying the type of something we detect
  - More detail during ML section



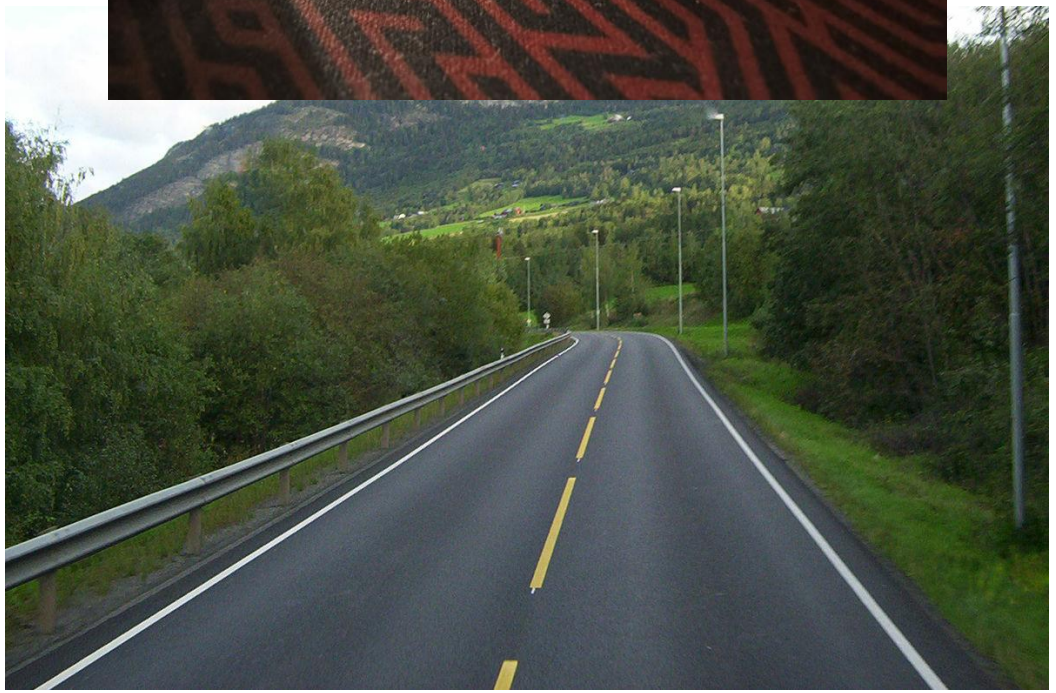
Grayscale Sensors

- Of particular interest to your project are grayscale sensors
- The robot car has a grayscale sensor facing downwards



- Consider that roads (and the project test track!) have lines on the edges and the centre of the route
- Grayscale sensor monitoring can provide a simple way to remain on the road!

https://en.wikipedia.org/wiki/Types_of_road#/media/File:Road_in_Norway.jpg

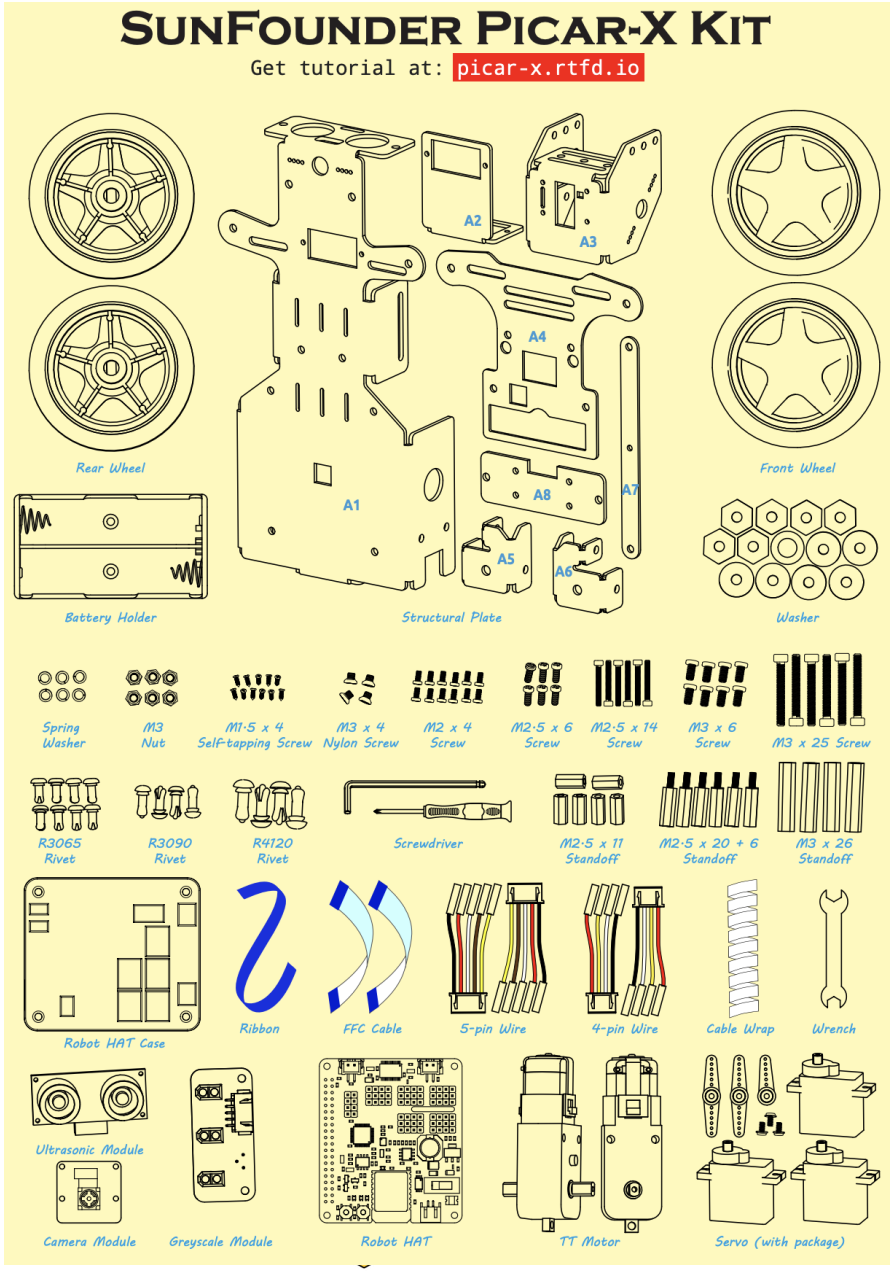


ASSEMBLY OF ROBOT KIT

Tutorial - Section 1

Hardware Overview

- Front-steering, rear-wheel-drive motor setup
- Robot HAT sits on top of the Raspberry Pi and provides motor controller interface
- Hardware provided:
  - Camera
  - Ultrasonic sensor for collision warning
  - IMU for rotation
  - Grayscale camera facing downwards
  - Variable speed motors



Tutorial Overview

- Assemble the cars together in your group
- We will be available for any questions you have during the assembly process
- Ideally you can finish a lot of the assembly today, to avoid having small pieces to carry around.

ROBOTICS CONTROL SOFTWARE

Lesson - Section 2

Introduction to Robotics Software

- Sensors provide us with low-level data
- However, we need to process this data to make it useful
  - **Interpretation**: we have some inertial and some environmental information. How can we combine the two together?
  - **Filtering**: data can be noisy, how do we extract high-quality estimates from it?

- There are significant constraints on robotics software
  - **Speed**: must make decisions in real-time
  - **Bandwidth**: sheer amount of data we are processing is high

Inputs and Outputs

- Inputs: sensor values, user commands
- Outputs: wheel motion

- Effectively, we are developing a function mapping input to output
- We can do this in a variety of ways:
  - **Deterministic programming** (course focus): develop set of rules the robot will follow to map from an input to an output. Can include ML as necessary
  - **Full ML control**: no human-defined rules on way in which robot will approach the problem

Programming Approach (Pseudocode)

- Detail steps the robot will follow in code, and gather information from our sensors (and provide outputs to our motors) as necessary to complete these defined steps

How do we determine these values?

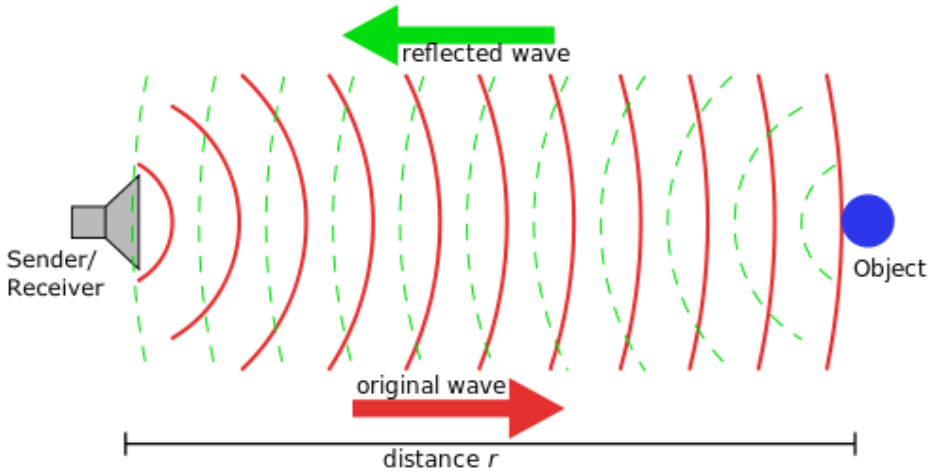
```
while True:
        if ultrasonic_distance >= 10cm & sign_detected == None:
                driveStraight()
        elseif sign_detected == “Right”:
                turnRight()
```

Filtering of Information

Sensor values can be unreliable. Consider the mechanism of operation for ultrasonic sensors, which utilize reflected sound waves to estimate the distance to an object.

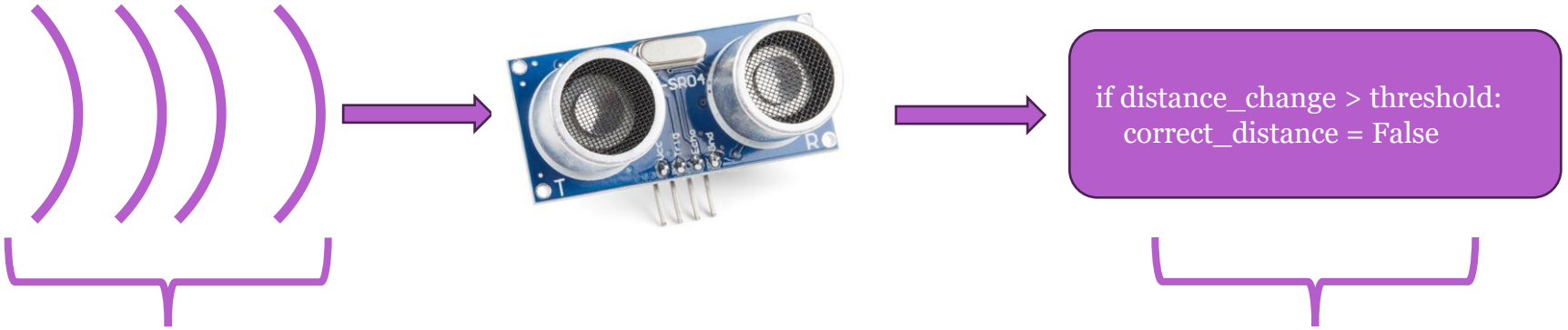
These reflected sound waves can be interrupted by other sound in the environment, or perturbed by a rough surface finish of the object they are bouncing back from.

We need some method to accurately determine the true sensor value given the noise that exists in the signal.



Filtering Techniques

- We can apply filtering techniques both on the initial ultrasonic frequencies we receive from the sensor, and the predicted distance the algorithm computes, to get an extremely high-quality estimate of the true distance value



Preprocessing:
Handling variation in measured ultrasonic waves

Postprocessing:
Handling variation in ultrasonic prediction

Signal Preprocessing

- **Signal averaging**: compute the average over a fixed time window, to smoothen out short-term fluctuations
- **Low-Pass and High-Pass filters**: don't include frequencies outside of the frequency band for the ultrasonic signal

Cytron Technologies

User's Manual

V1.0

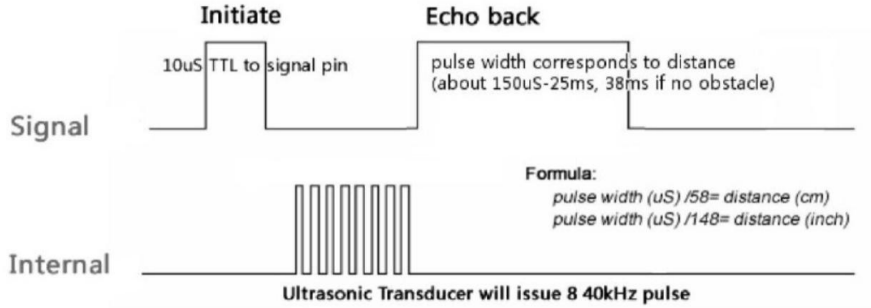
May 2013

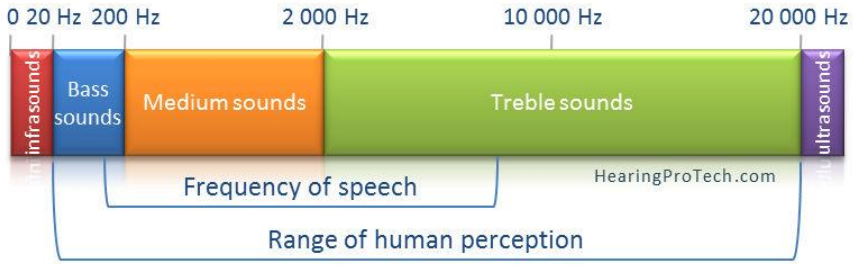
5.0 OPERATION

The timing diagram of HC-SR04 is shown. To start measurement, Trig of SR04 must receive a pulse of high (5V) for at least 10us, this will initiate the sensor will transmit out 8 cycle of ultrasonic burst at 40kHz and wait for the reflected ultrasonic burst. When the sensor detected ultrasonic from receiver, it will set the Echo pin to high (5V) and delay for a period (width) which proportion to distance. To obtain the distance, measure the width (Ton) of Echo pin.

Time = Width of Echo pulse, in uS (micro second)

- Distance in centimeters = Time / 58
- Distance in inches = Time / 148
- Or you can utilize the speed of sound, which is 340m/s



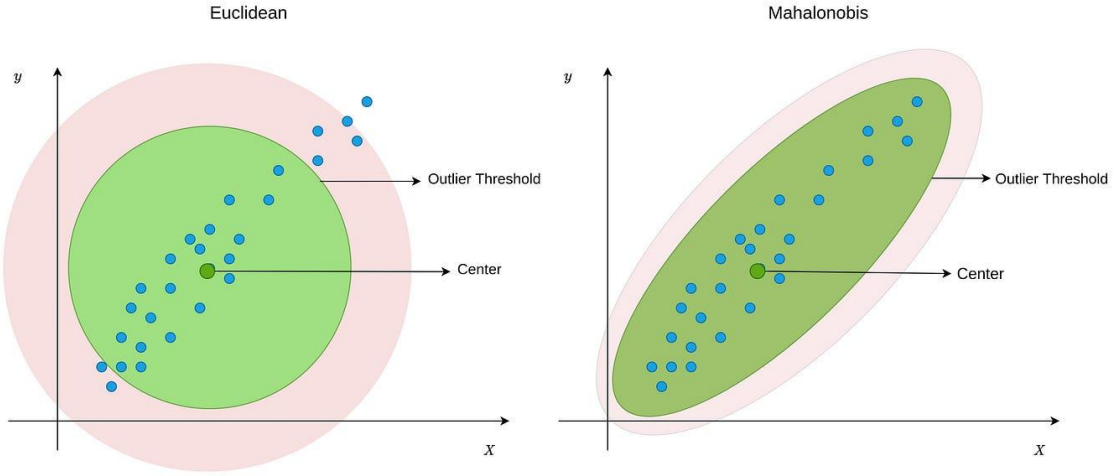


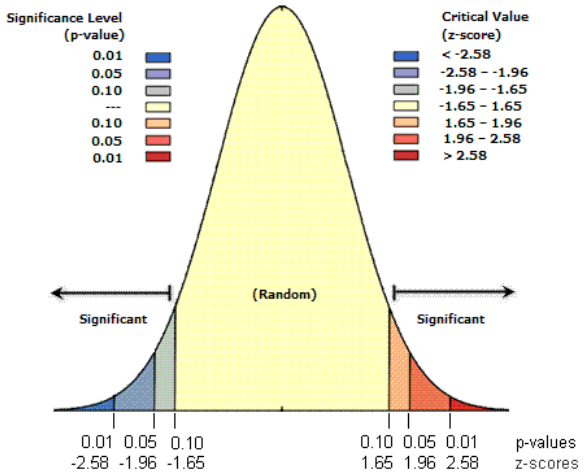
We can filter out all audible range sounds

https://docs.google.com/document/d/1Y-yZnNhMYy7rwhAgyL_pfa39RsB-x2qR4vP8saG73rE/edit?pli=1#bookmark=id.a3nei5z46z1p

Distance Estimate Postprocessing

- **Outlier detection**: a variety of methods exist including:
  - **Distance methods**: measure distance between a new sensor prediction and our recent list of measurements. Is this a reasonable value, or too unrealistic?
  - **Statistical methods**: compare new measurements to old distribution, how extreme is the new value?
- **Smoothing**: use a rolling average that allows driving logic to slowly adapt to significant sensor prediction changes





https://pro.arcgis.com/en/pro-app/latest/tool-reference/spatial-statistics/what-is-a-z-score-what-is-a-p-value.htm

Notes on Filtering and Processing

- This is a tuning exercise, and there are pros and cons of each decision
- Example: too much smoothing means suddenly detected obstacles won't cause a massive jump in predicted distance -> could result in collision!

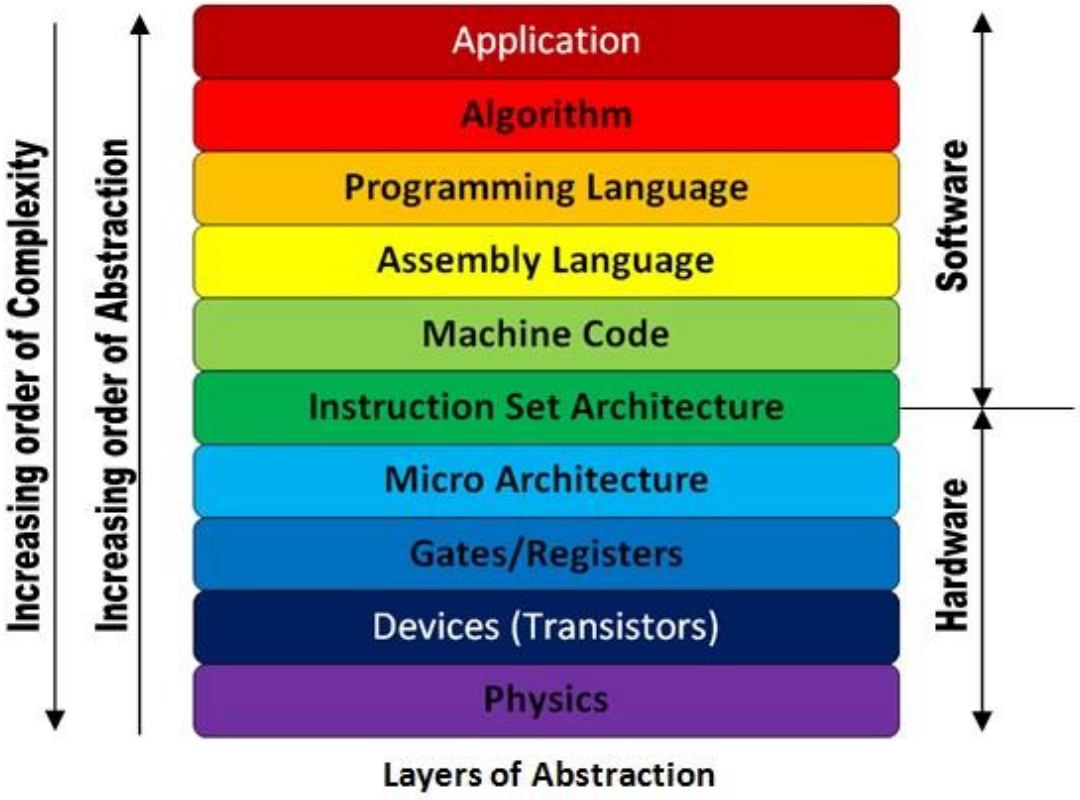
- Should run tests with a variety of scenarios you might encounter and decide the best way to handle them

LINUX SETUP ON ROBOT

Tutorial - Section 2

Software Architecture

- Although not a systems course, it is useful for us to understand how the different layers of abstraction in computer systems work together.
- We build on top of a lot of technologies to assemble robotics systems created by other people. It is not efficient for us to implement all levels of this ourself; we should use **software libraries** wherever possible to implement known algorithms.



https://electronics.stackexchange.com/questions/353915/what-is-the-role-of-isa-instruction-set-architecture-in-the-comp-arch-abstract

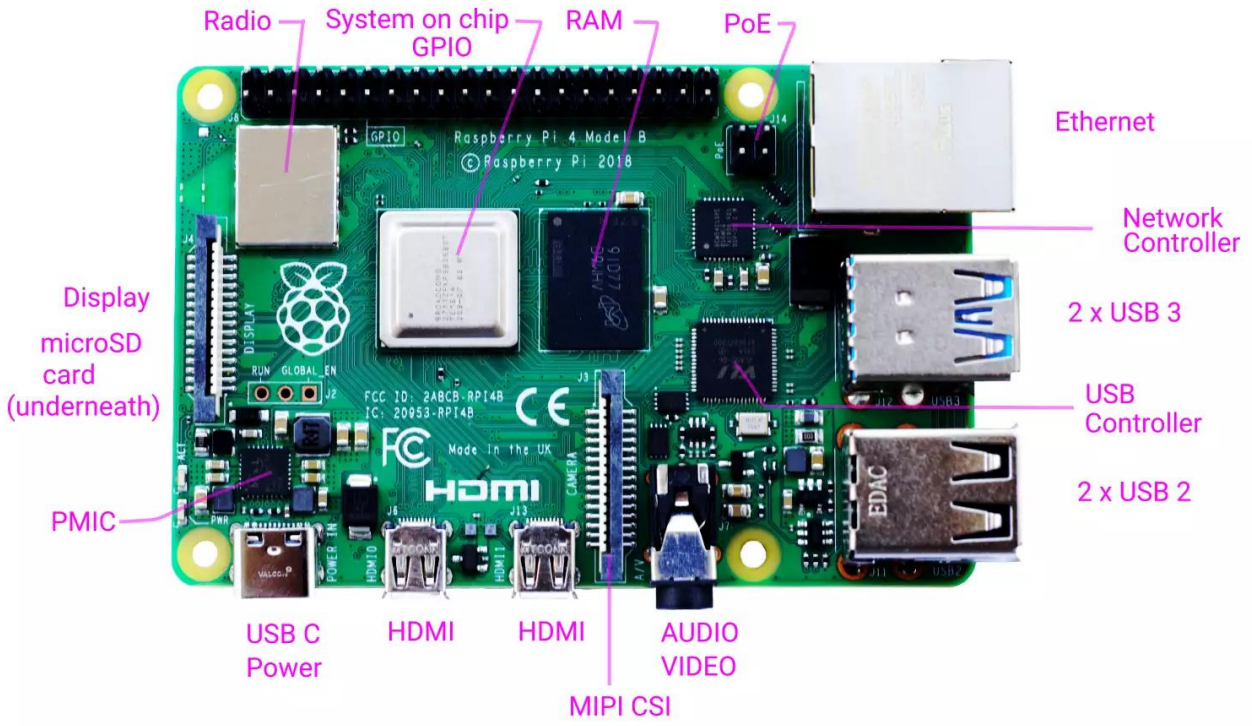
Software Architecture (Continued)

- The PiCar-X kit comes with a variety of software libraries developed for it, designed to be used with Python and Linux
- We will set up Linux on the device, and examine some of the libraries it has available for usage

Computer Overview

- Raspberry Pi 4 single-board computer
- Runs a full Linux-based operating system
- You will set the system up from scratch

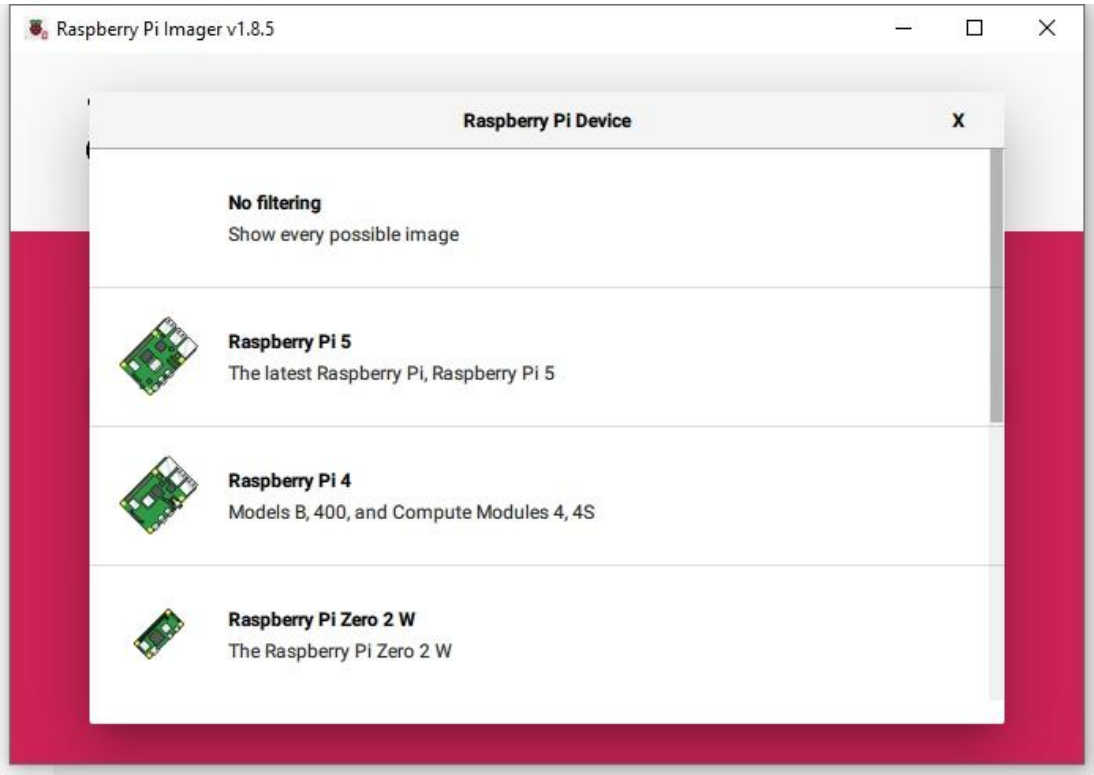
- Before you start:
  - Connect microSD to the computer you will be using for setup

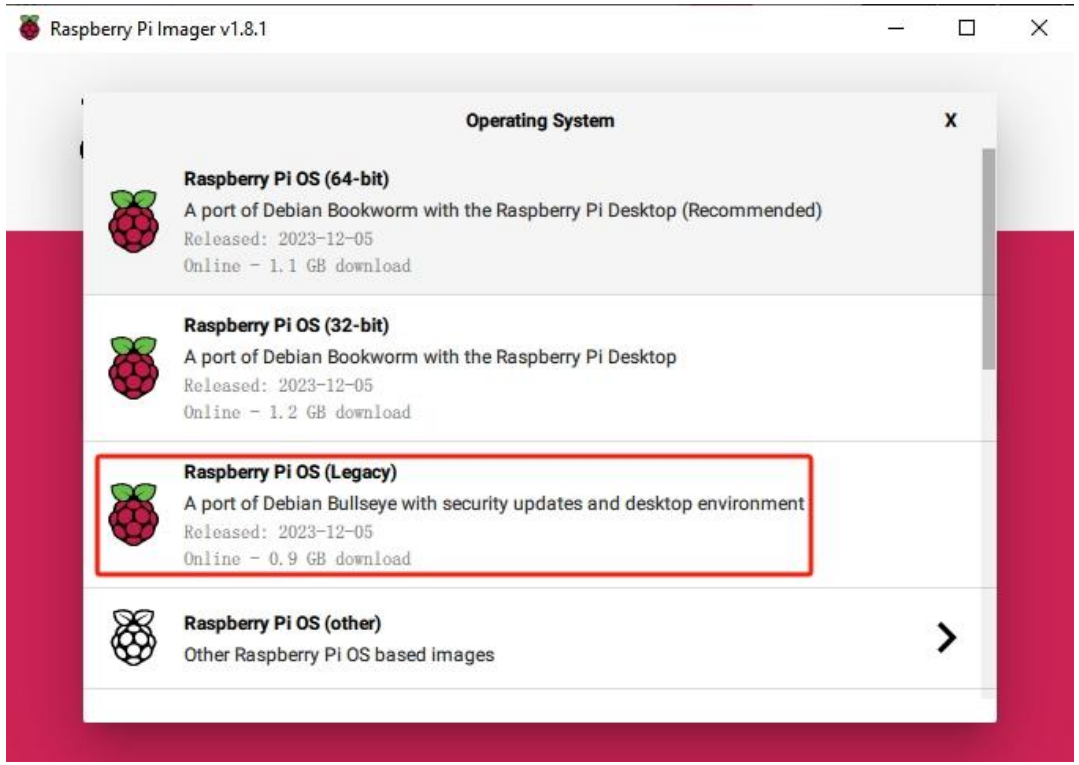


The Raspberry Pi 4 Model B

Software Download

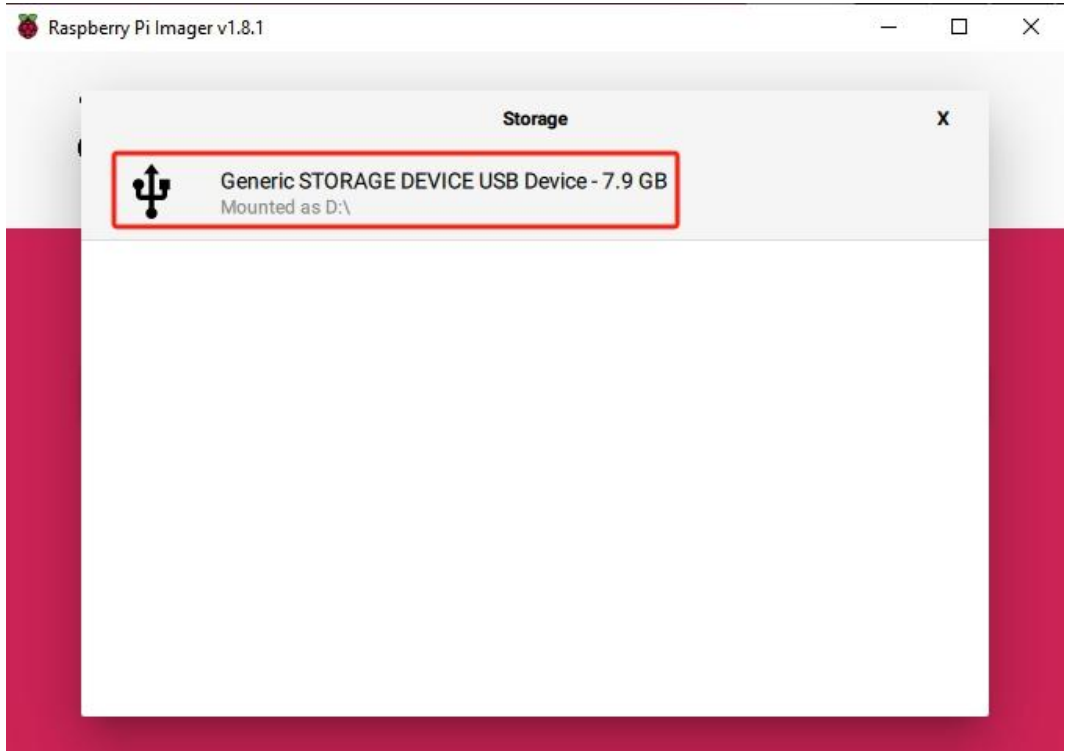
1. Download Raspberry Pi Imager: https://www.raspberrypi.com/software/
2. Select Raspberry Pi 4 from the device list
3. Use “Raspberry Pi OS (Legacy)” for the operating system

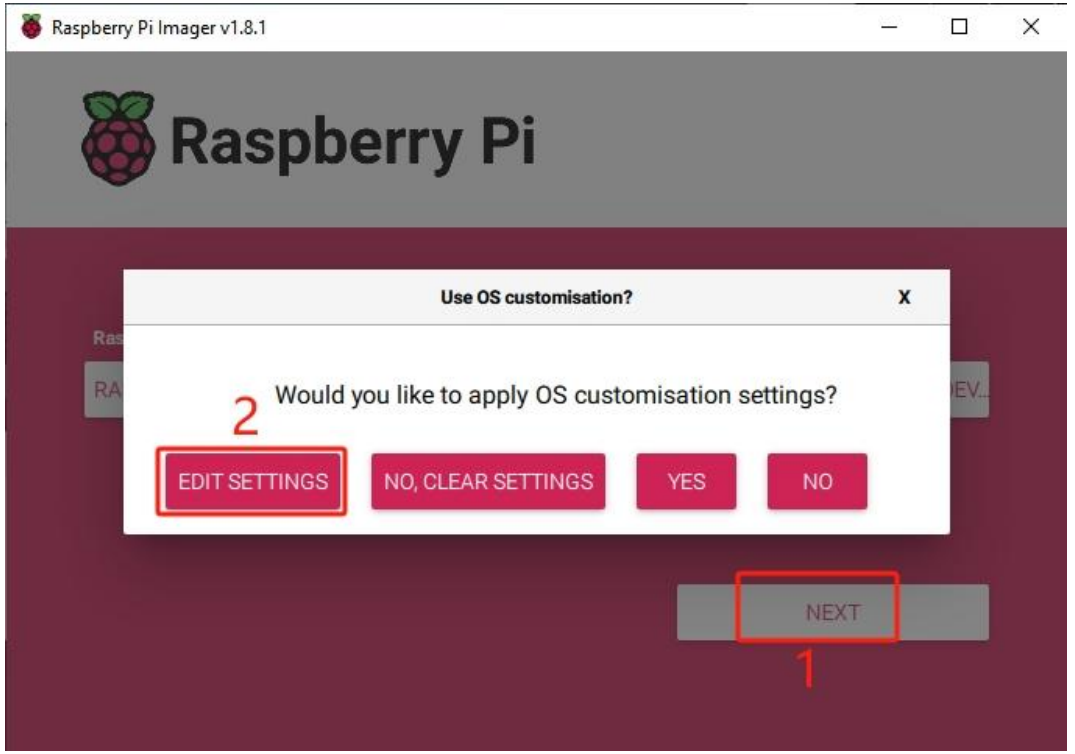




Software Download (Continued)

4. Select the microSD card to write the operating system to
5. Edit settings to choose a team name, username and password, enable SSH access, and configure the wireless network





WiFi Settings for Project

To aid in the project setup, we've created a local WiFi network you can use for managing the Raspberry Pi rather than needing to register it with the university network.

The WiFi network is as follows:

SSID: AITraining

Password: robotcar

You should fill this information in, as well as set a name for the robot car, and a username/password to log in.

After the Linux Installation

- You can put the microSD card in the Raspberry Pi, and power it on.
- Connect your computer to the same WiFi
- Open a terminal and use “SSH” to connect to the terminal of the Raspberry Pi.
- ssh username@your-pi-name.local

- Once you are logged in to the Pi, we can set up the software libraries

Installing PiCar-X Libraries

- The PiCar-X libraries are prewritten software components aimed at implementing basic functionality for you.
- Rather than needing to, for example, directly send electrical pulses to each wheel, you can instead call functions to do what you want to do (drive forward, etc), and the library will perform the translation to the low-level hardware control needed.

- The robot car's developer is a good place to find the documentation:

https://docs.sunfounder.com/projects/picar-x/en/latest/index.html

Basics of Library Installation

1. Make sure system is up-to-date
    1. sudo apt update
    2. sudo apt upgrade
2. Make sure required packages are installed
    1. sudo apt install git python3-pip python3-setuptools python3-smbus
3. Download libraries one-by-one from GitHub and perform the installation

GitHub Library Installation – robot-hat

1. Install robot-hat (library for the robot control board)
    1. cd ~/ ← Change directory to home folder
    2. git clone -b v2.0 https://github.com/sunfounder/robot-hat.git ← Download code from GitHub
    3. cd robot-hat ← Change directory to our downloaded code
    4. sudo python3 setup.py install ← Run install script as root

GitHub Library Installation – vilib

1. Install vilib (vision library for working with the camera)
    1. cd ~/
    2. git clone -b picamera2 https://github.com/sunfounder/vilib.git
    3. cd vilib
    4. sudo python3 install.py

GitHub Library Installation – picar-x

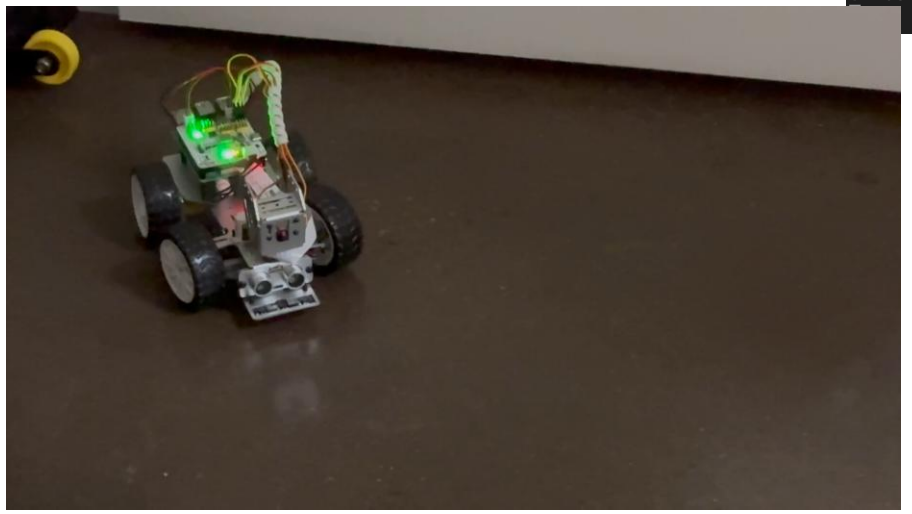
1. Install picar-x library (adds library interfaces specific to this car hardware config)
    1. cd ~/
    2. git clone -b v2.0 https://github.com/sunfounder/picar-x.git
    3. cd picar-x
    4. sudo python3 setup.py install

2. You can now navigate into the picar-x folder to find the calibration script

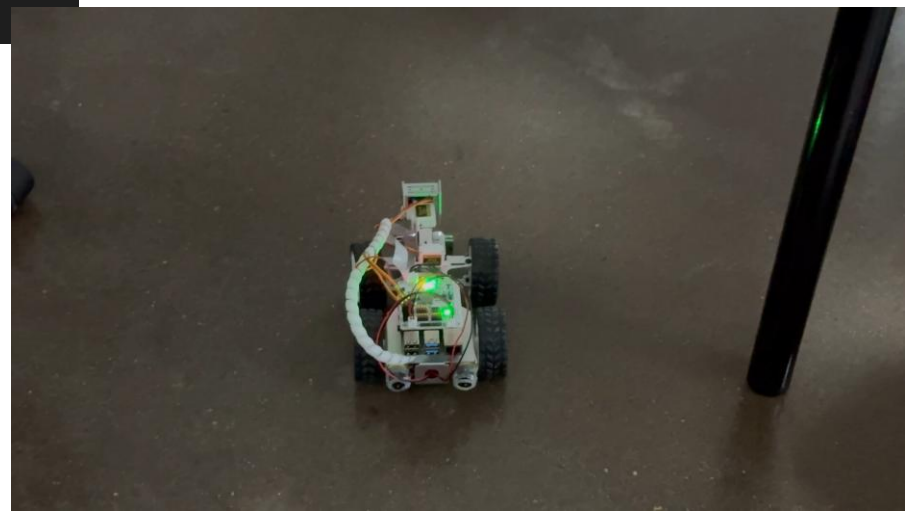
Calibrate the Robot

- Calibration script is needed to ensure the robot drives in the direction you ask

```
    [1]: direction servo            [W/D]: increase servo angle
    [2]: camera pan servo           [S/A]: decrease servo angle
    [3]: camera tilt servo          [R]: servos test

    [4]: left motor                 [Q]: change motor direction
    [5]: right motor                [E]: motors run/stop

    [SPACE]: confirm calibration                [Crtl+C]: quit


[ direction servo ] [ left motor ]
offset: [0.0, 0.0, 0.0], [1, 1]
```



Before Calibration



After Calibration

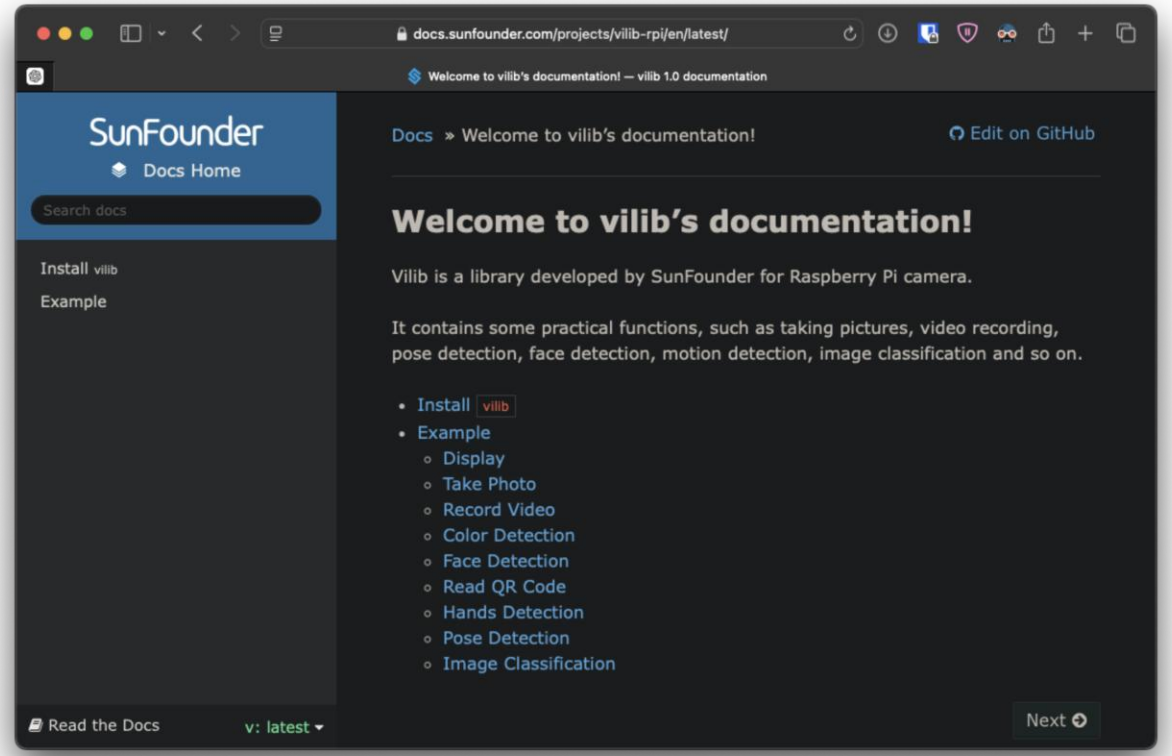
Calibrate the Robot

How to use the calibration script:

1. Run the robot forward with the “E” key
2. Use the “A” and “D” keys until the robot tracks straight
3. If you need to stop the robot during the calibration process due to lack of space, you can press “E” again to stop the robot

Capturing Camera Information

- Look through the example libraries included with the car
- vilib is Pi camera driver we have installed as part of initial setup
- Documentation will provide instructions on how to capture basic camera feed from the car
  - Don't worry – we also have example code in case you are stuck!



https://docs.sunfounder.com/projects/vilib-rpi/en/latest/

Capturing Camera Information – Example Code

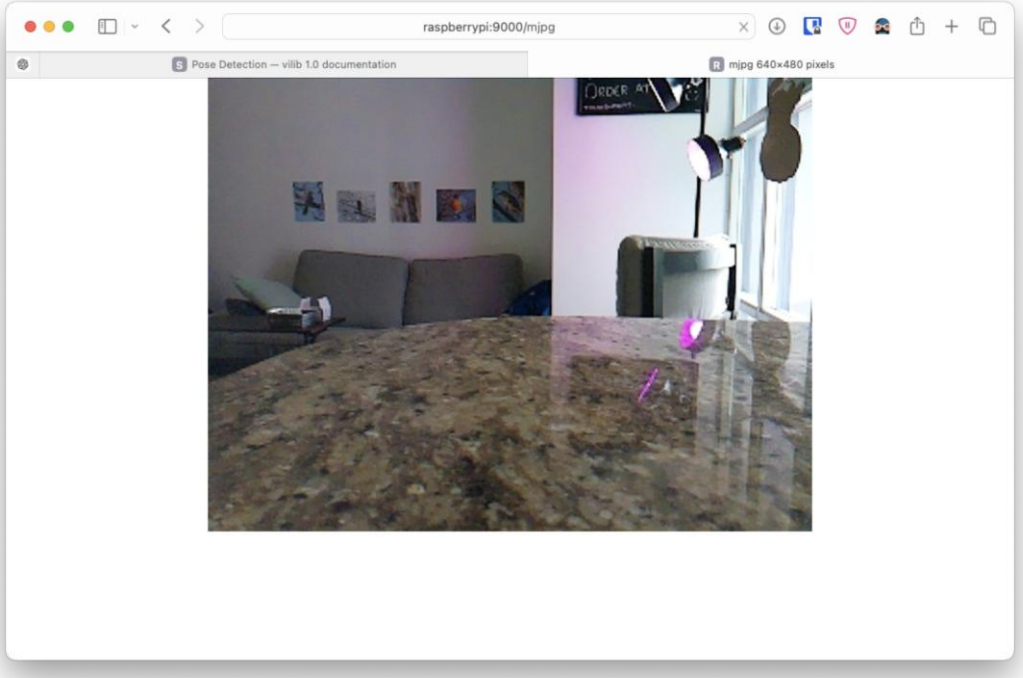
First, let's check the camera is working and we can see information from it.

We can use the example code on the right.

1. Start camera driver with no flipping
2. Start camera display on web server

Then, we can navigate to the Pi's IP at port 9000 to see the image.

```
from vilib import Vilib

def main():
    Vilib.camera_start(vflip=False,hflip=False)
    Vilib.display(local=False,web=True)

if __name__ == "__main__":
    main()
```



Storing Camera Information in Code

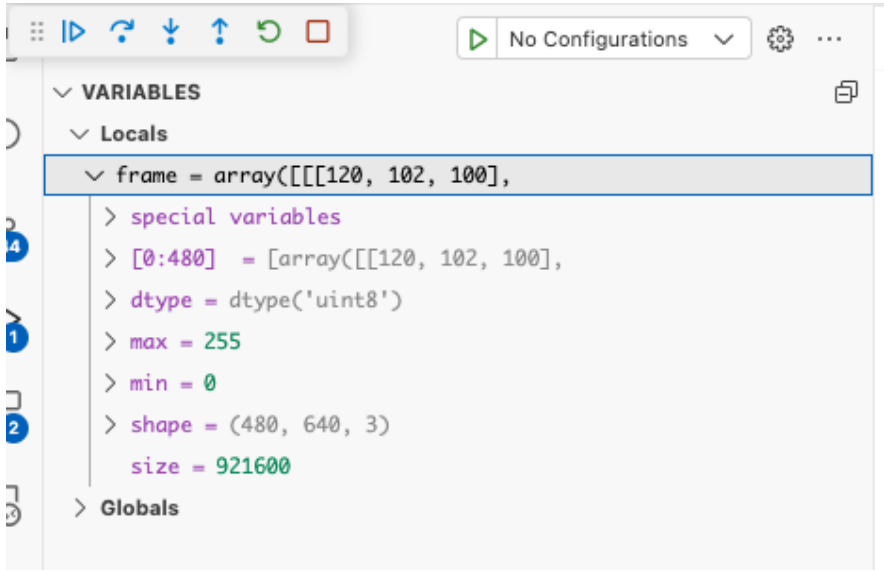
This example simply sends camera data over the network. We want to be able to use the camera data in our ML model.

The frames the camera captures therefore needs to be stored in a *variable* we can access.

Vilib allows us to get the most recent image frame as a NumPy array (perfect for feeding into our neural network)

If we use a debugger, *frame is* a 640 x 480 image of pixel values, with 3 channels (R, G, B)

```
from vilib import Vilib

def main():
    Vilib.camera_start(vflip=False,hflip=False)
    Vilib.display(local=False,web=True)

    while True:
        # get image frame
        frame = Vilib.img_array[0]

if __name__ == "__main__":
    main()
```



Other Library Utilities - Motors

- We highly recommend you look through the examples to learn how to interface with the robot
- How to move the car? Check example/11.video_car.py
  - Be careful, the script on the right will keep going forward! Make sure you either put a timeout or catch the car.

```
from robot_hat.utils import reset_mcu
from picarx import Picarx

reset_mcu()
sleep(0.2)

px = Picarx()

def move(operate:str, speed):
    if operate == 'stop':
        px.stop()
    else:
        if operate == 'forward':
            px.set_dir_servo_angle(0)
            px.forward(speed)
        elif operate == 'backward':
            px.set_dir_servo_angle(0)
            px.backward(speed)
        elif operate == 'turn left':
            px.set_dir_servo_angle(-30)
            px.forward(speed)
        elif operate == 'turn right':
            px.set_dir_servo_angle(30)
            px.forward(speed)

# now let's move the car forward at 50% speed
move('forward', 50)
```

Import libraries

Reset motor control unit (MCU) from any potential previous code

Spawn an instance of Picarx() object with name px

Steering servo angle
Drive wheel direction

Other Library Utilities – Greyscale Sensor

```
from picarx import Picarx
px = Picarx()
greyscale_value_list = px.get_grayscale_data()
```

- We can get the colour value of the downward-facing greyscale module using the px.get_grayscale_data() method
- Want to do line tracking with the greyscale sensor to follow the middle line?
  - Write a thresholding function that detects whether your car is tracking the line or not
  - Alternatively, look in the picar-x library code: there is already a set_line_reference() function, can you use it to follow the line?
  - Also, an example in the “5.minecart_plus.py” example file!

ENVIRONMENTAL INTERACTION

Section 3 - Lesson

Introduction

We now have a reliable interface to our sensors. We can make simple rules about how to navigate based on this, such as:

- Collision avoidance
- Line following

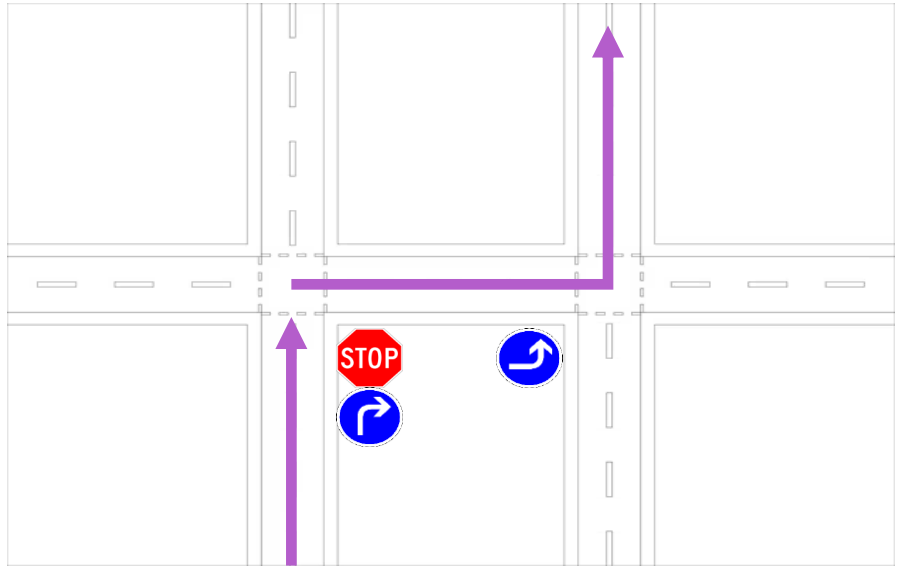
However, we often need to infer more complex data from the environment, such as where we should navigate based on cues such as signage. For this purpose, we need to do more advanced processing of information.

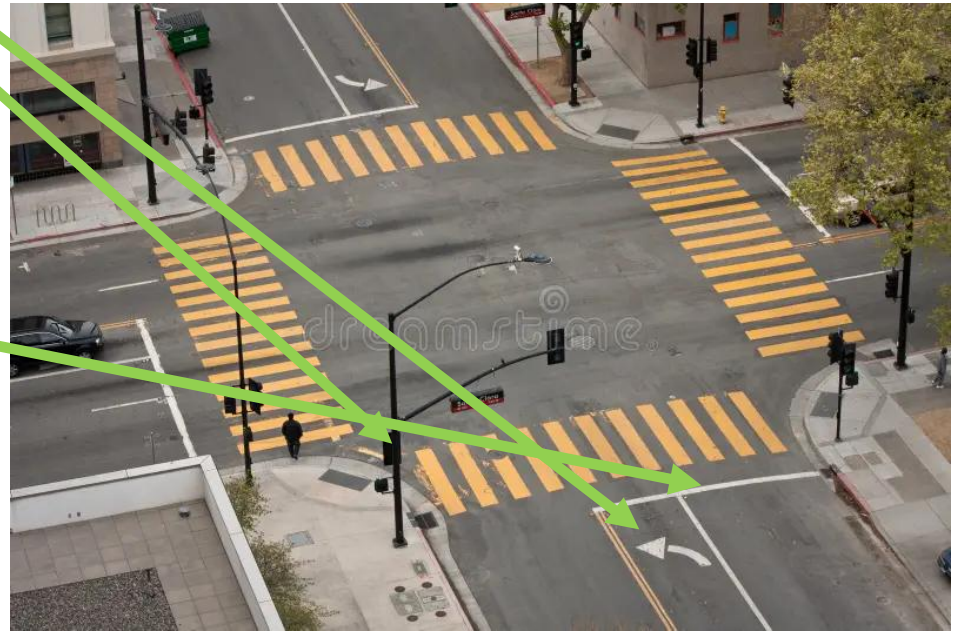
Building a Model of the Vehicle in Space

Consider the key points we might use to decide how to traverse a road. These include:

- Direction indicators (signs, turning lanes)
- Traffic control devices (stop signs, traffic lights)
- Stop lines

We use a combination of sensors to build a relationship between the robot's current position and these key points in the environment.





https://www.dreamstime.com/photos-images/traffic-light-empty-intersection.html

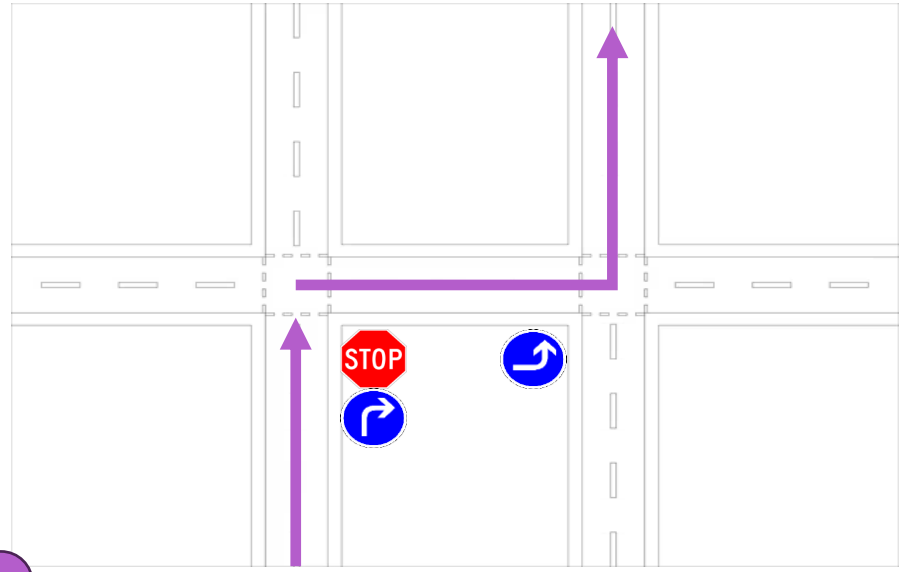
Navigating an Intersection (Pseudocode)

```
if stopLineDetected == True:
    forwardSpeed(0)
    if sign == “TurnRight”:
        turnRight()
    elseif sign == “TurnLeft”:
        turnLeft()
    else:
        forwardSpeed(10)
```

Grayscale Sensor

Camera Sign Recognition

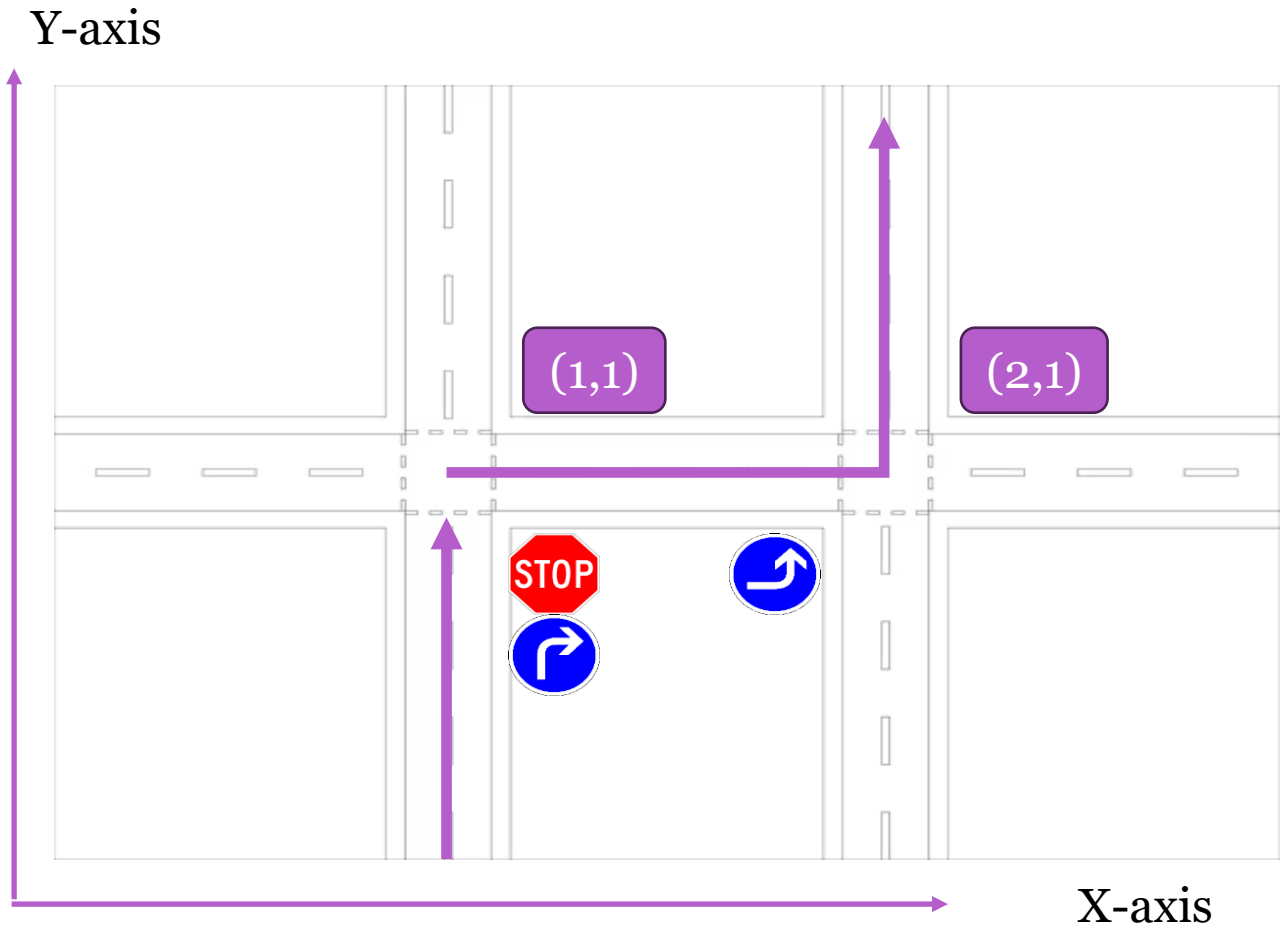
IMU or vision data to ensure proper turn



We use a combination of our sensors to properly identify the need to turn and make the maneuver properly

Robotic Pose

- The concept of **pose** describes the location and orientation of the robot in space.
- In a driving domain such as ours, this space can be simplified to 2D.
- By keeping track of the decisions we make along our route, we can develop some universal sense of our location, rather than solely responding to inputs as we go.



Utility of Maintaining Location Information

- Backtracking
  - Historical pose information is crucial to this; we can maintain a list of previous waypoints
- Relativistic Programming
  - As we get closer to some key point, we should slow down -> can bound our global distance to these waypoints with our location in the coordinate system
- Obstacle Avoidance
  - When combined with a solid understanding of the robot's dimensions in space, we can ensure the robot does not make navigation decisions that would result in a collision

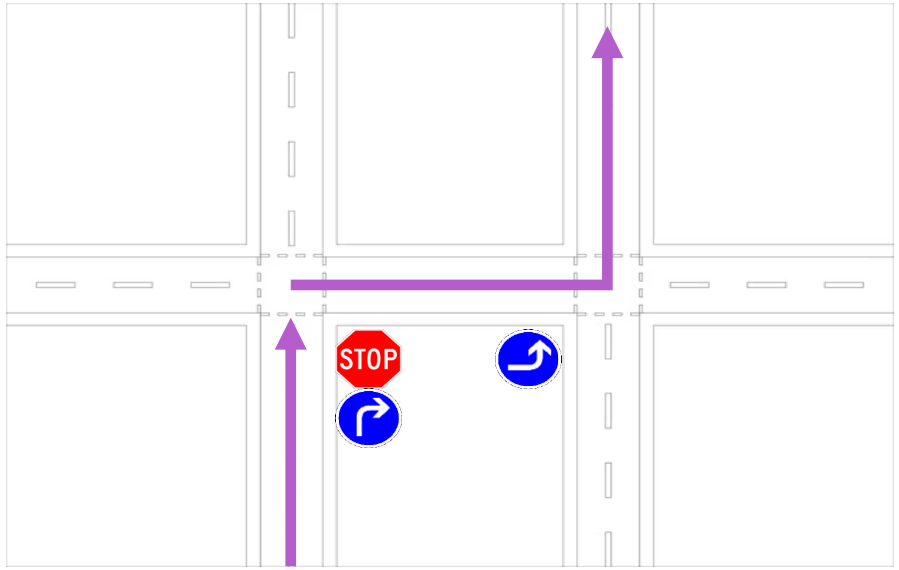
Adjusted Intersection (Pseudocode)

```
location = [0,0]
direction = 'north'
if stopLineDetected == True:
    location = [location[0], location[1] + 1]
    forwardSpeed(0)
    if sign == "TurnRight":
        turnRight()
        direction = 'east'
    elseif sign == "TurnLeft":
        turnLeft()
        direction = 'west'
    else:
        forwardSpeed(10)
```

Initial Location + Heading

New Location

New Heading



We are no longer simply responding to signals, but also maintaining a record of our current relationship with the environment.

More Complex Interpretation

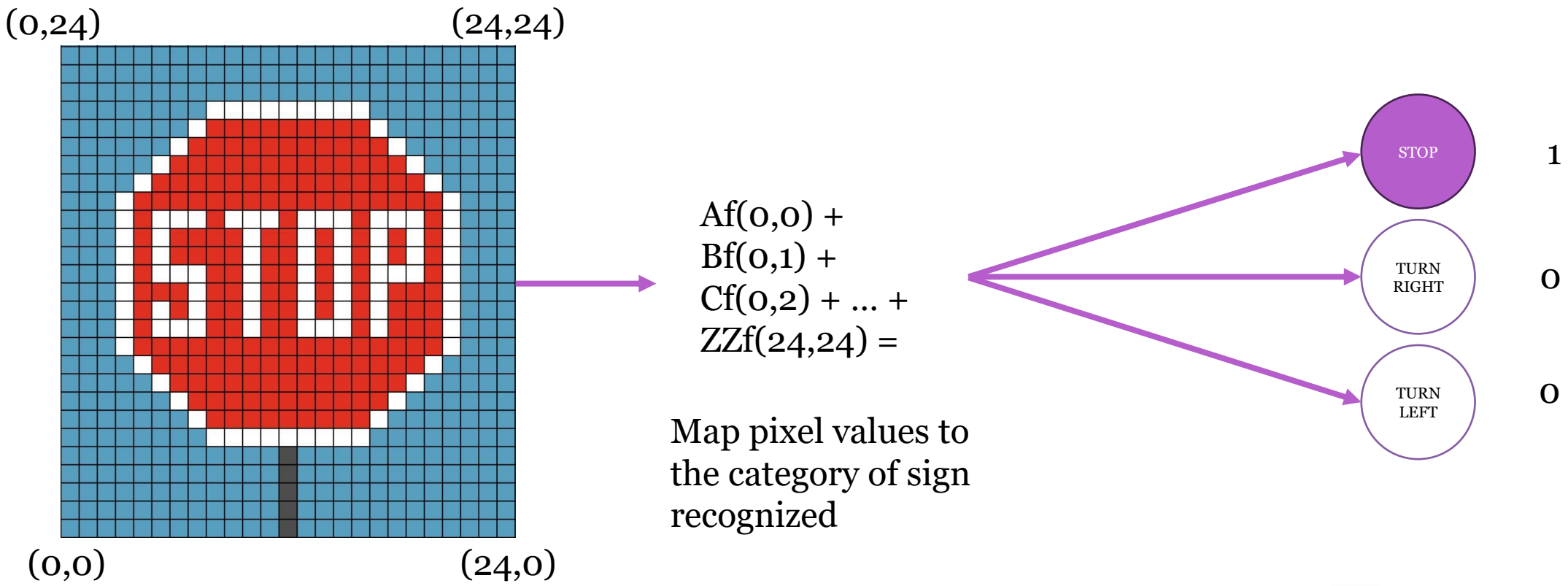
We now can determine some sort of spatial relationship between ourselves and components in the environment.

However, we still need to do some work to be able to understand more complex sensor data and properly understand what it is telling us about the environment.

Sign recognition -> using camera data, but how?

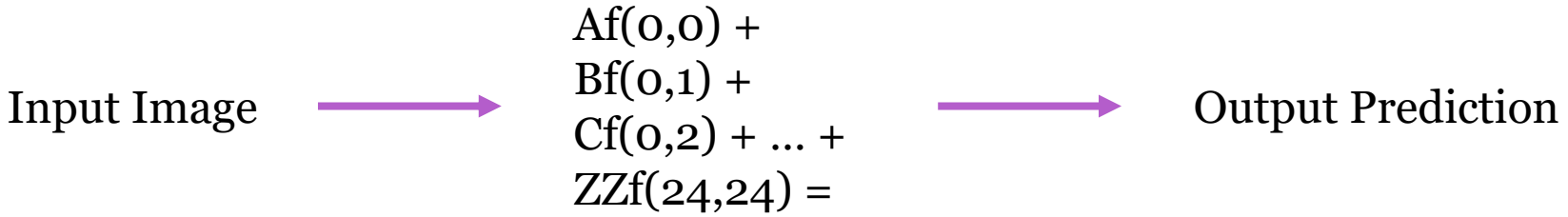
Introduction to Machine Learning

- It is hard for us to program explicit rules explaining how to recognize signs
- We can define our problem as one of mapping inputs to outputs



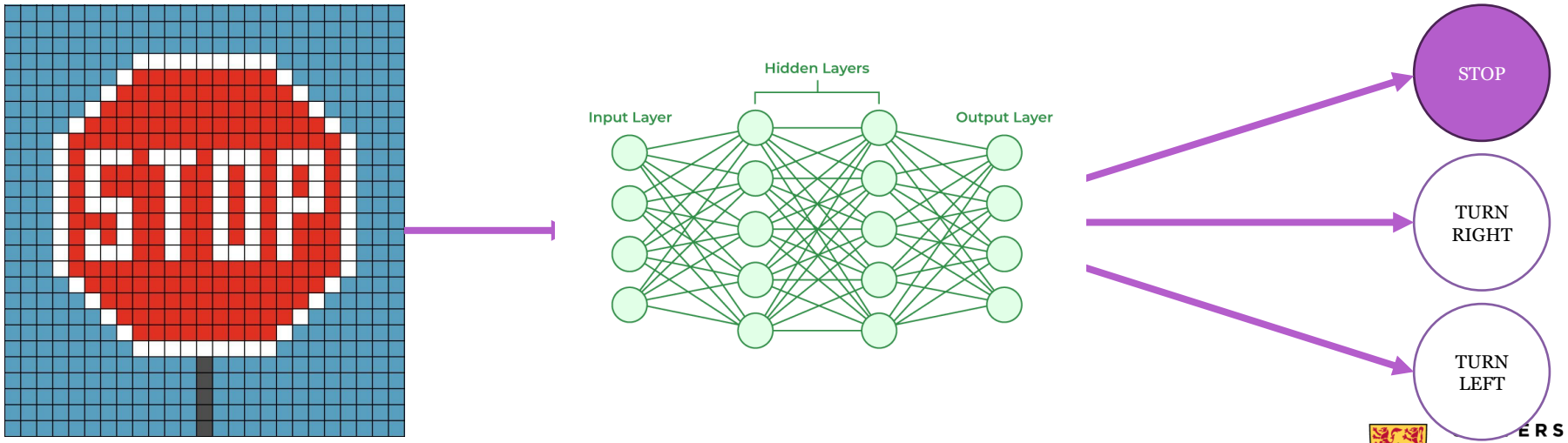
How to Define Function?

- The function representation is too complex to understand
- Instead, we assume there is some valid function, and use a *training process* to develop the function.
- We can provide a picture where we know what type of image it is (stop sign, etc), and compare the prediction of our current function with our known type
- Then, we can update the weights of our function (A, B, C, … ZZ) such that it produces the correct output for our input



The Big Idea

- Given we do this weight optimization process along enough known input+output pairs, in theory, our system should learn the generalized relationships between input images and output values to be able to accurately categorize novel images it hasn't seen before
- This is known as a "neural network", as it is analogous to a brain



More Complex Techniques

As people, when we recognize objects, we don't simply use single pixel values.

Instead, we rely on the *relationships* between pixels.

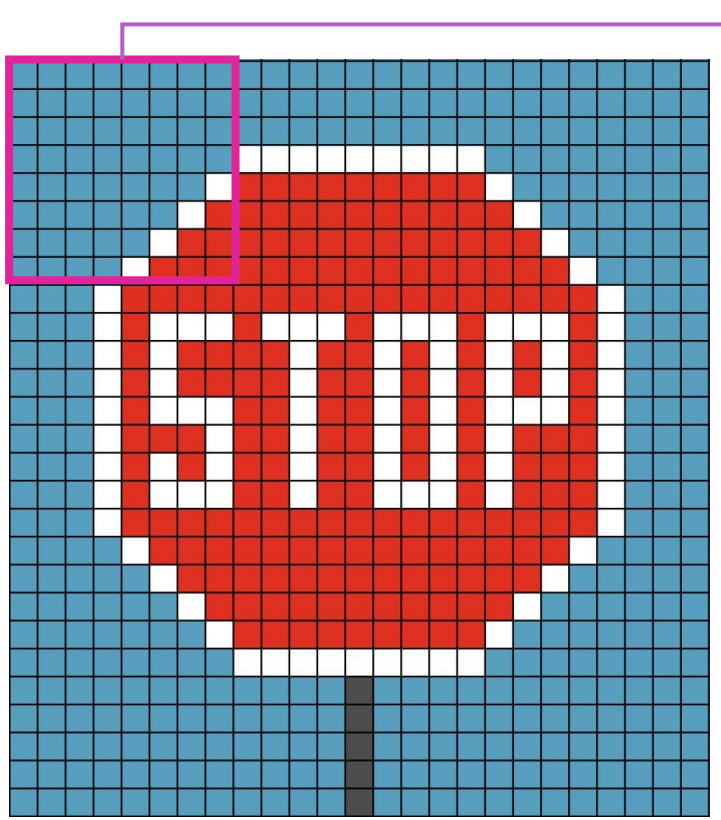
For example, in a stop sign, you have white text surrounded by the red of the sign; there is also a stark contrast between the sign and the surroundings.

We can model these relationships in two ways:

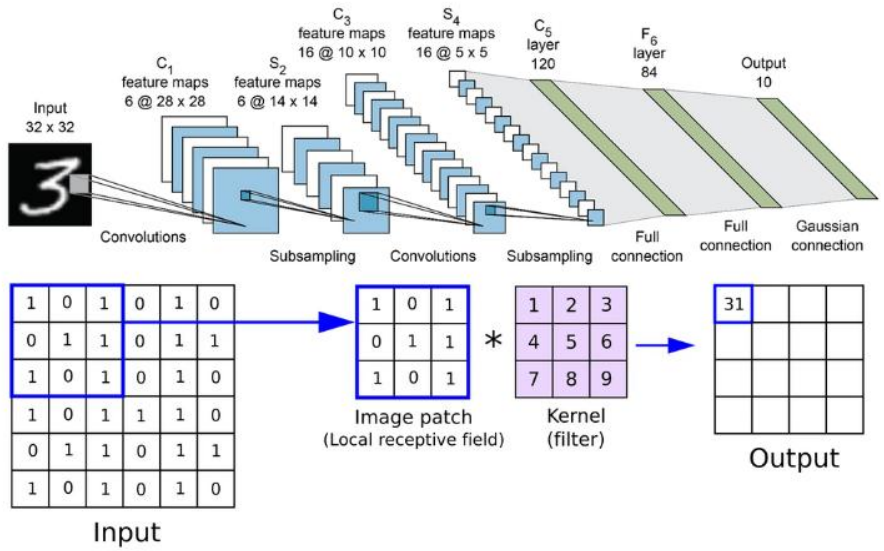
- Convolutional windows
- Interconnected network layers

Convolutional Networks

- Over our grid of an image, we can apply a window, in which we do some summarizing operation defining the key features we have detected in that window



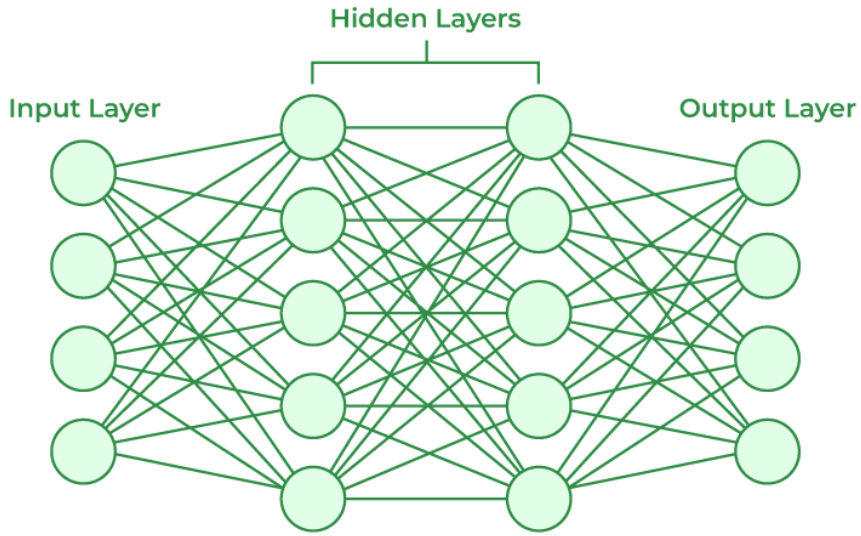
By sliding the window over all of the image, we can define a representation for the features that reside in each part of the image.



https://www.superannotate.com/blog/guide-to-convolutional-neural-networks

Interconnected Network Layers

- Using multiple hidden layers in the network that link to each other, we can recognize the relationship between features in each window that make up a cohesive image of a sign
- The hidden layers are unique in that they form part of the internal layers of the network, and their value is never directly observed, unlike the inputs or outputs of the network.
- The combination of hidden layers and convolution allows us to recognize both components of the sign (edges, etc), and how these group together to define a sign of a particular type



Distance Estimation

- If we want to put our ML prediction at a physical location in space, we'll have to use some sort of distance estimation
- Given the signs are of a predictable size, we can use the width as an estimation of where it is in space
- Alternatively, we can use simplified approaches, such as remembering the most recently viewed sign, and then inferring that it corresponds to this stop line when the grayscale sensor detects one

SIGN RECOGNITION SETUP

Tutorial - Section 3

Notes about Machine Learning

The “black box” nature of machine learning algorithms makes them notoriously inefficient.

Unlike deterministic programming, where we can easily notice areas for potential efficiency improvement, neural networks are a balance of attempting to simply the neural grid as much as possible, whilst leaving enough complexity in the network to properly recognize objects in the variety of circumstances the network may encounter. To improve peak recognition performance, we can do the following:

- Run the network on a remote, more powerful computer
- Use a high-quality network, but don’t attempt to scan all images in real-time

Libraries for ML Programming

- The optimization code for ML can be written by hand, but it is more efficient to use a library that handles this for you
- Most popular are:
  - PyTorch
  - TensorFlow
- The general structure of the training approach is seen on the right

```
# Train the model
for epoch in range(num_epochs):
    start_time = time.time()
    lstm.train()
    for i, (images, labels, timestamps) in enumerate(train_loader):
        # todo: add timestamps to model for time series analysis
        # Forward pass
        with autocast():
            predicted_control = lstm(images)

            # get last label to compare to output
            actual_control = labels[:, -1, :]

            # calculate loss
            loss = wrapper_loss(predicted_control, actual_control)

        # at what time was this batch relative to start?
        time_elapsed = time.time() - start_time
        # use minutes basis
        time_elapsed = time_elapsed / 60

        # # Backward and optimize
        optimizer.zero_grad()
        scaler.scale(loss).backward()
        scaler.step(optimizer)
        scaler.update()
```

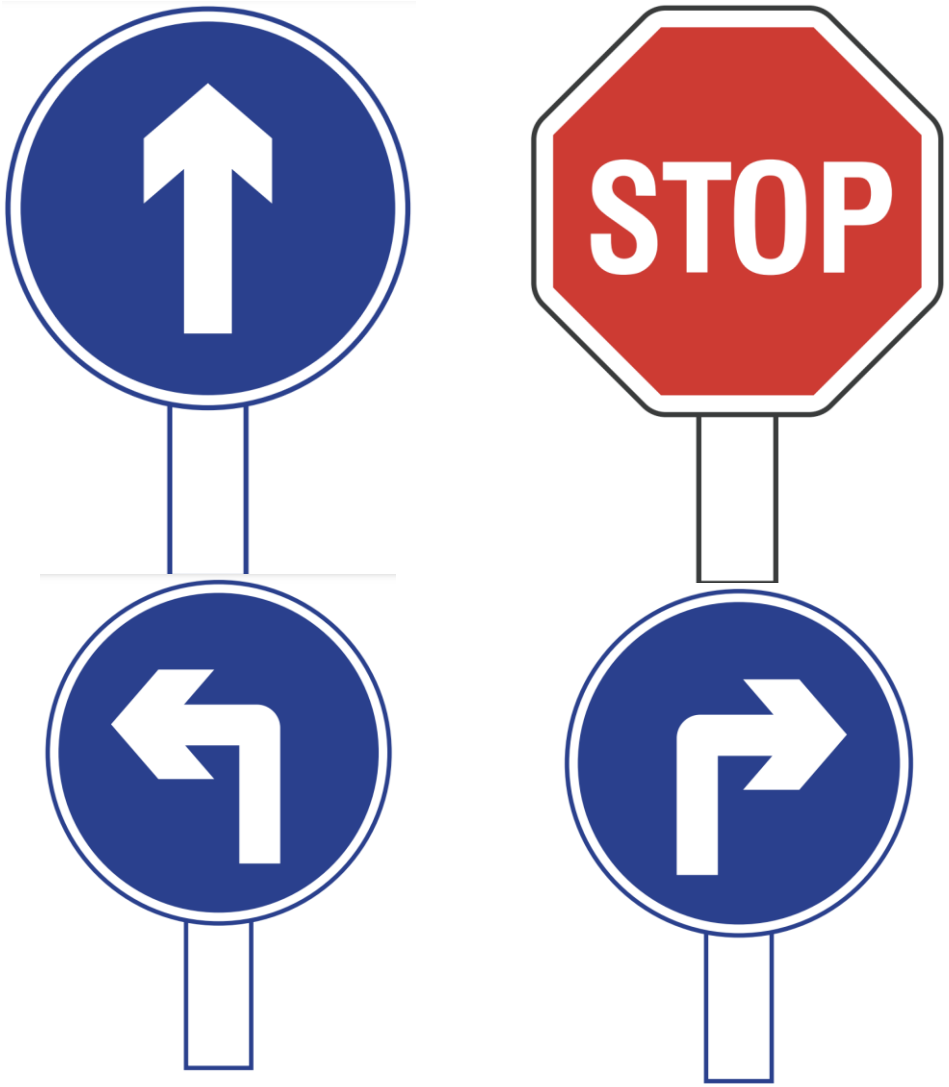
1. Put model in training mode
2. For each image on the training dataset:
   1. run it through the model
   2. Calculate difference between prediction and actual (“loss”)
   3. Use loss to optimize network weights

Traffic Sign Setup

We will be using traffic signs to inform the robot what direction it should be driving at any given point.

These signs will be placed at specific points on the road grid, and the car will have to follow the route as marked.

Your goal is to train the computer to recognize what signs it is seeing.



Training Dataset

- As mentioned in the lesson, we need to map *inputs* to *outputs* by weighting and biasing a neural grid -> we need a large collection of input/output pairs
- This is referred to as the **training dataset**
- There are open-source training datasets
- We are using a ViCOS traffic sign dataset, which has pairs of traffic signs combined with labels describing their type, to train the robot to drive
- You can always augment the model with your own data to potentially improve prediction quality in novel environments

https://www.vicos.si/resources/dfg/

What's in the Dataset?

- 200 different types of signs
- ~7000 real images, ~8700 augmented images (signs synthetically placed in new images to create a bigger dataset)
- Co-ordinates describing the location of the sign in the image

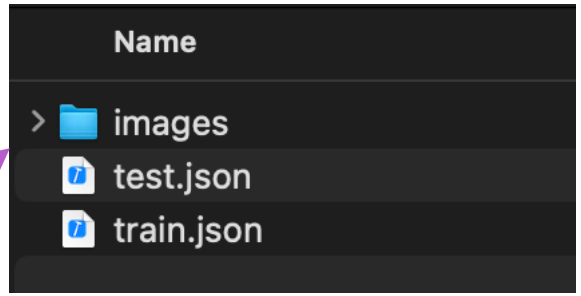
Setting up the Training Process

- The Raspberry Pi is *not very powerful*.
- The training process requires us to perform learning across a large dataset
  - Dataset is 7GB even compressed!

- Rather than perform training on the Raspberry Pi, better to use a more powerful system and deploy the finished model to Raspberry Pi
- Two options:
  - Your own computer
  - Google Colab (online cloud service)

Loading the Data

- The annotations are saved into a JSON file in COCO format, which pairs image IDs with the associated classes
- Let's set up our folder structure so all the data is accessible, and start loading it
- We can then install "pycocotools" which enables us to easily import the JSON format and start loading data as needed
- Finally, create a basic script which loads the annotations into memory




```
pip3 install pycocotools
```

```
from pycocotools.coco import COCO

# path to training annotation file
train_json_path = './train.json'

# Load the annotation file
coco = COCO(train_json_path)
```

Verifying the Data Looks Correct

- We can expand the code to display samples from the data to properly understand the structure of the annotations.
- Image info contains the file name and dimensions of the image
- Annotation info contains the categories of each sign in the image and a segmentation polygon around it

```python
from pycocotools.coco import COCO

# path to training annotation file
train_json_path = './train.json'

# Load the annotation file
coco = COCO(train_json_path)

# Load all image IDs and annotations
img_ids = coco.getImgIds()
ann_ids = coco.getAnnIds()

# Load the first image info and its annotations
img = coco.loadImgs(img_ids[0])[0]
anns = coco.loadAnns(coco.getAnnIds(imgIds=img['id']))

print("Image info: ", img)
print("Annotation info: ", anns)
```

Image info: {'id': 0, 'height': 1080, 'width': 1920, 'file_name': '0000001.jpg'}
Annotation info: [{'id': 0, 'area': 14560, 'bbox': [312, 369, 104, 140], 'category_id': 78, 'segmentation': [[357, 369, 371, 370, 383, 375, 394, 384, 402, 393, 407, 403, 411, 414, 414, 424, 416, 434, 416, 444, 416, 454, 414, 465, 410, 476, 404, 487, 396, 497, 385, 505, 372, 509, 358, 508, 345, 503, 335, 494, 327, 485, 321, 475, 317, 465, 315, 454, 313, 444, 312, 434, 313, 424, 315, 413, 319, 402, 324, 391, 333, 381, 344, 373, 357, 369]], 'image_id': 0, 'ignore': False, 'iscrowd': 0}, {'id': 1, 'area': 7904, 'bbox': [304, 295, 104, 76], 'category_id': 158, 'segmentation': [[307, 371, 304, 299, 405, 295, 408, 366, 307, 371]], 'image_id': 0, 'ignore': False, 'iscrowd': 0}, {'id': 2, 'area': 3400, 'bbox': [30, 274, 50, 68], 'category_id': 136, 'segmentation': [[77, 274, 80, 339, 33, 342, 30, 281, 77, 274]], 'image_id': 0, 'ignore': False, 'iscrowd': 0}, {'id': 3, 'area': 6666, 'bbox': [303, 231, 101, 66], 'category_id': 179, 'segmentation': [[402, 231, 404, 292, 304, 297, 303, 240, 402, 231]], 'image_id': 0, 'ignore': False, 'iscrowd': 0}, {'id': 4, 'area': 14544, 'bbox': [300, 90, 101, 144], 'category_id': 59, 'segmentation': [[354, 90, 368, 94, 380, 102, 388, 112, 394, 123, 398, 134, 400, 145, 401, 155, 401, 165, 400, 175, 398, 185, 395, 195, 390, 206, 383, 216, 373, 226, 361, 232, 347, 234, 333, 231, 322, 223, 313, 213, 307, 201, 303, 190, 301, 180, 300, 170, 299.9391229381186, 160.667506138333, 301, 150, 303, 140, 307, 129, 312, 119, 319, 108, 328, 99, 341, 92, 354, 90]], 'image_id': 0, 'ignore': False, 'iscrowd': 0}]

Filtering the Data to Relevant Classes

- For our purposes, we care about a few signs:
  - Turn Left (Class 70)
  - Turn Right (Class 69)
  - Go Straight (Class 68)
  - Stop (Class 43)
- We can filter the data to only include these classes, which is much quicker than 200 types of signs
- Notice we have less images than annotations as each image may contain multiple signs

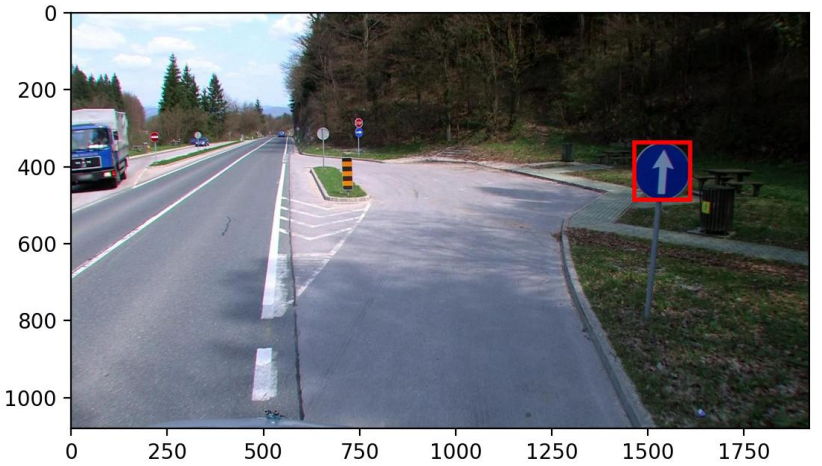
```python
from pycocotools.coco import COCO

# path to training annotation file
train_json_path = './train.json'

# Load the annotation file
coco = COCO(train_json_path)

# Get annotation IDs where the category ID is 70, 69, 68, 43
ann_ids = coco.getAnnIds(catIds=[70, 69, 68, 43])

# Load the images and annotations matching the IDs
anns = coco.loadAnns(ann_ids)
img_ids = [ann['image_id'] for ann in anns]
img_ids = list(set(img_ids))
imgs = coco.loadImgs(img_ids)

print("Number of images: ", len(imgs))
print("Number of annotations: ", len(anns))
```

Number of images: 774
Number of annotations: 799

Preparing the Data for Training

- To load the data into the model, we need to create a *dataloader*
- This is responsible for fetching the image and all annotations associated with the image (so that it can learn to detect multiple objects within an image at the same time)
- Now that we have both the data and the annotations loaded in memory, we can use matplotlib to draw a box around the image, to further check we are loading things properly

```python
class COCODataset(Dataset):
    def __init__(self, image_dir, json_path, category_ids):
        # perform the initial filtering of the dataset to only the images we want
        coco = COCO(json_path)
        ann_ids = coco.getAnnIds(catIds=category_ids)
        anns = coco.loadAnns(ann_ids)
        img_ids = [ann['image_id'] for ann in anns]
        img_ids = list(set(img_ids))

        # save the relevant info
        self.coco = coco
        self.category_ids = category_ids
        self.image_dir = image_dir
        self.img_ids = img_ids

    def __len__(self):
        return len(self.img_ids)

    def __getitem__(self, idx):
        img_id = self.img_ids[idx]
        ann_ids = self.coco.getAnnIds(imgIds=img_id, catIds=self.category_ids)
        anns = self.coco.loadAnns(ann_ids)
        img = self.coco.loadImgs(img_id)[0]

        img = Image.open(f'{self.image_dir}/{img["file_name"]}')
        return img, anns

# Create a COCODataset object
training_dataset = COCODataset(image_dir='./images', json_path='./train.json', category_ids=[70, 69, 68, 43])

image, annotations = training_dataset[100]
plot_image_with_annotations(image, annotations)
```



Resizing the Data

MobileNet is a 320x320 model. Hence, it expects images to have this dimension.

Aspect ratio is an important element for recognizing objects. We can't just squish the elements, we should resize them to keep the ratio between width and height the same.

In doing this, we need to also pad the images with empty data, so that they still fill the 320x320 input space (as the matrix multiplication will fail if they don't).

We've also adjusted the annotation scale to match the new dimensions. Now our plot shows the input image is 320x320, and the bounding box still looks correct.


```python
def __getitem__(self, idx):
    img_id = self.img_ids[idx]
    ann_ids = self.coco.getAnnIds(imgIds=img_id, catIds=self.category_ids).copy()
    anns = self.coco.loadAnns(ann_ids).copy()
    img = self.coco.loadImgs(img_id)[0].copy()

    img = Image.open(f'{self.image_dir}/{img["file_name"]}')
    saved_img = img.copy()
    # let's get the dimensions of the image, it will be needed to scale the annotations
    width, height = img.size
    # whichever is bigger, let's make it 320
    if width > height:
        img = img.resize((320, int(320 * height / width)))
    else:
        img = img.resize((int(320 * width / height), 320))
    # make image a tensor and pad it (centered)
    img = torchvision.transforms.ToTensor()(img)
    # make annotations into expected format
    # first, let's scale the annotations [x, y, width, height]
    for ann in anns:
        ann['bbox'][0] = ann['bbox'][0] * img.size(2) / width
        ann['bbox'][1] = ann['bbox'][1] * img.size(1) / height
        ann['bbox'][2] = ann['bbox'][2] * img.size(2) / width
        ann['bbox'][3] = ann['bbox'][3] * img.size(1) / height

    # now, convert from [x, y, width, height] to [x1, y1, x2, y2]
    for ann in anns:
        ann['bbox'][2] += ann['bbox'][0]
        ann['bbox'][3] += ann['bbox'][1]

    # normalize the image
    # img = torchvision.transforms.functional.normalize(img, mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

    # grayscale the image
    # img = torchvision.transforms.Grayscale()(img)
    # img.repeat(3, 1, 1)

    # pad
    img = torch.nn.functional.pad(img, (0, 320 - img.shape[2], 0, 320 - img.shape[1]), value=1)

    # category shift
    for ann in anns:
        ann['category_id'] = self.category_ids.index(ann['category_id']) + 1

    # let's visualize our squished image and annotations to verify
    converted_image = torchvision.transforms.ToPILImage()(img)
    plot_image_with_annotations(converted_image, anns)

    # finally, convert to the record format expected by the model
    anns = {
        'boxes': torch.tensor([ann['bbox'] for ann in anns], dtype=torch.float).to(self.device),
        'labels': torch.tensor([ann['category_id'] for ann in anns], dtype=torch.int64).to(self.device),
        'image_id': torch.tensor([img_id], dtype=torch.int64).to(self.device),
    }
    # move image to device
    img = img.to(self.device)
    return img, anns
```

Starting with a Pre-Trained Baseline Model

- In machine learning, training from scratch can be extremely time-intensive
- Given the restrictive timeline of the project, better to use an existing neural network that can recognize other objects (for example, keyboard, mouse, pencil) and cross-train it on the new dataset
- Hypothesis: the earlier training on other objects will train the model *how* to recognize unique objects, and this process will simply give it the ability to understand the novel objects in our dataset.
- MobileNet-v2 architecture is a good fit for Raspberry Pi due to its efficiency in low-power systems

Downloading the Pre-Trained Model

PyTorch/Torchvision contains the pre-trained model weights.

We initialize a model object of the SSDLite320_MobileNet_v3_Large type.

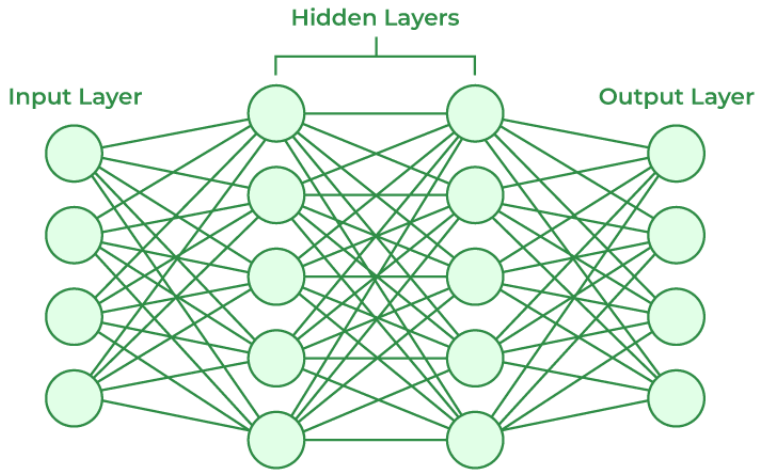
Then, we specify it has the IMAGENET1K_V2 pretrained weights as its backbone.

Finally, we replace the classification head with one the same length as our sign categories (4), plus 1 (in MobileNet, the background of the image is a category too).

```
import torchvision
import torch
import torch.nn as nn
from torchvision.ops import *
from torchvision.models.detection import SSDLite320_MobileNet_V3_Large_Weights
from torchvision.models.detection.ssdlite import SSDLiteClassificationHead
from torchvision.models.mobilenetv3 import MobileNet_V3_Large_Weights

model = torchvision.models.detection.ssdlite320_mobilenet_v3_large(weights_backbone=MobileNet_V3_Large_Weights.IMAGENET1K_V2, num_classes=len(categories) + 1,
trainable_backbone_layers=6)
```

What's a Backbone?

- The backbone refers to the hidden layers in our network
- The input layer changes with each image
- Output layer we are removing and resizing to fit our 4-category problem (Stop/Left/Right/Straight)
- Hence, we want to "import" pretrained layers in the middle, as these are the key components responsible for recognizing objects, and it should be less work to retrain them onto the new problem set.



With our code implemented to fetch the images and associated annotation tags, we're able to pipe it into the model, and calculate the "loss" (difference between our input image and output prediction of the detected sign).

We run this on our GPU, with a dataloader that fetches multiple image/target pairs in one "batch".

Our loss value informs the training process on how updates to the weights and biases in side the network improves its ability to predict outputs; it moves in the direction of decreasing loss.

Finally, we save the network weights and biases with each training cycle, to use later on our test data.

```
# move to device
model.to(device)

batch_size=60

for epoch in range(num_epochs):
    model.train()
    training_dataset = COCODataset(image_dir='./images', json_path='./train.json', category_ids=categories, device=device)
    training_dataloader = torch.utils.data.DataLoader(training_dataset, batch_size=batch_size, shuffle=True, collate_fn=collate_fn)
    i = 0
    total_loss = 0
    for images, targets in training_dataloader:
        i += 1
        # move the images and targets to the device
        loss_dict = model(images, targets)
        losses = sum(loss for loss in loss_dict.values())
        total_loss = total_loss + (losses/batch_size)

        optimizer.zero_grad()
        losses.backward()
        optimizer.step()

    # print average loss
    print(f"Epoch: {epoch}, Loss: {total_loss/i}")

    # update the learning rate
    lr_scheduler.step()

    # evaluate on the test dataset
    # metric = testing_dataset.coco_evaluate(model, testing_dataloader, device)
    # print(f"Epoch: {epoch}, mAP: {metric}")

    # save the model with epoch number
    torch.save(model.state_dict(), f"model_epoch_{epoch}.pth")
```

Inference with the Built-In Camera

Remember the code we wrote earlier storing camera data as a NumPy array?

We can now feed this code through our neural network on the robot to get it to predict signs in the image!

However, the Pi is not very powerful. We can either:

1. Simplify the model to move it to the Pi
2. Send the most recent Pi camera data to another device over ROS

```
from vilib import Vilib

def main():
    Vilib.camera_start(vflip=False,hflip=False)
    Vilib.display(local=False,web=True)

    while True:
        # get image frame
        frame = Vilib.img_array[0]
```

Input to our Neural Network

```
if __name__ == "__main__":
    main()
```

Robot Operating System (ROS)

- ROS is a method by which robot components can communicate with each other
- It uses a “publisher-subscriber” model
  - Certain subscribers may depend on new data being published of a certain type, i.e. camera data
  - When this data is received, the subscriber will perform some action (i.e. predict what is in the camera image), and publish some form of response for other nodes in the network

Basic ROS Implementation to Run Inference over Network

- Image capture code:

```
from vilib import Vilib

def main():
    Vilib.camera_start(vflip=False,hflip=False)
    Vilib.display(local=False,web=True)

    while True:
        # get image frame
        frame = Vilib.img_array[0]
```

Input to our ROS Function

```
if __name__ == "__main__":
    main()
```

- ROS publish code

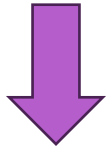
```
import rospy

pub = rospy.Publisher('/camera/array', np.array)
```

```
pub.publish(frame)
```

Listener Side

- On the side running the ML code:

```
def listener():
    rospy.init_node('listener', anonymous=True)

    rospy.Subscriber("/camera/array", np.array, callback, queue_size=1)

    # spin() simply keeps python from exiting until this node is stopped
    rospy.spin()

def callback(data):
    # ML pipeline here
```



Feeds into machine learning model

Listens to new messages on /camera/array and executes callback function

Pre-Trained Model and Detailed Code

- In the event that you can't get a network running properly, there is a pre-trained model you can use which can do the sign recognition for you.
- You can also use this pre-trained model as a checkpoint from which you do additional training, to expose the model to new datapoints and expand its quality.
- If you're having issues with any area of implementing it yourself, it might be a good idea to look at the detailed code provided with our pre-trained model
- You can also ask us for more help understanding the code

TRAJECTORY PLANNING

Section 4 - Lesson

What We've Learned

So far, we've learned a few key components of robotic programming:

1. How to read data from our sensors
2. How to use filtering and other statistical approaches to turn our sensor data into reliable information
3. How to build a cohesive model of the environment we're operating in from this information

The final step is to combine all of our components together to perform actions and move through the world.

Controlling Driving

Variety of approaches can be used depending on the type of maneuver performed.

Driving straight:

- Grayscale sensor tracking of centerline
- Camera recognition of edges of road

Cornering:

- Use IMU gyroscopic data to stop turn when we've moved by the correct angle
- Use camera data to estimate when the car is correctly oriented after the turn
- Calibration by hand: perform lots of turns by manually changing values, record average motor outputs necessary to perform that type of maneuver
- Combination of all, depending on *confidence*? (Multimodal sensor fusion)

Confidence Intervals

- *Confidence* is some metric by which we rate the quality of an algorithm's output
- In ML, we often have floating point classification outputs, not binary 1 for correct class and 0 for incorrect class
  - Take class with highest prediction score as correct
  - Can also choose to disregard even highest prediction score if it's below some threshold

- We can apply same concept to other computer control problems
  - How valid is IMU data if we turn too quickly? Too slowly?
  - If we're using multimodal approaches, which of the sensors should we rely on more?

Filtering our ML Outputs

- Machine learning produces an *inference*, or *prediction*, of the sign in an image
- Like any prediction, this can be entirely incorrect.
- There are plenty of biases in the pre-trained model I've created, and in the model you have trained too

- We need to figure out what sorts of biases we might see to address these

Poor Results ML Models Might Output

- Confusion if multiple signs are visible but close together
- Falsely recognizing similarly coloured items as a sign

Combining Understanding: Filtering ML Output

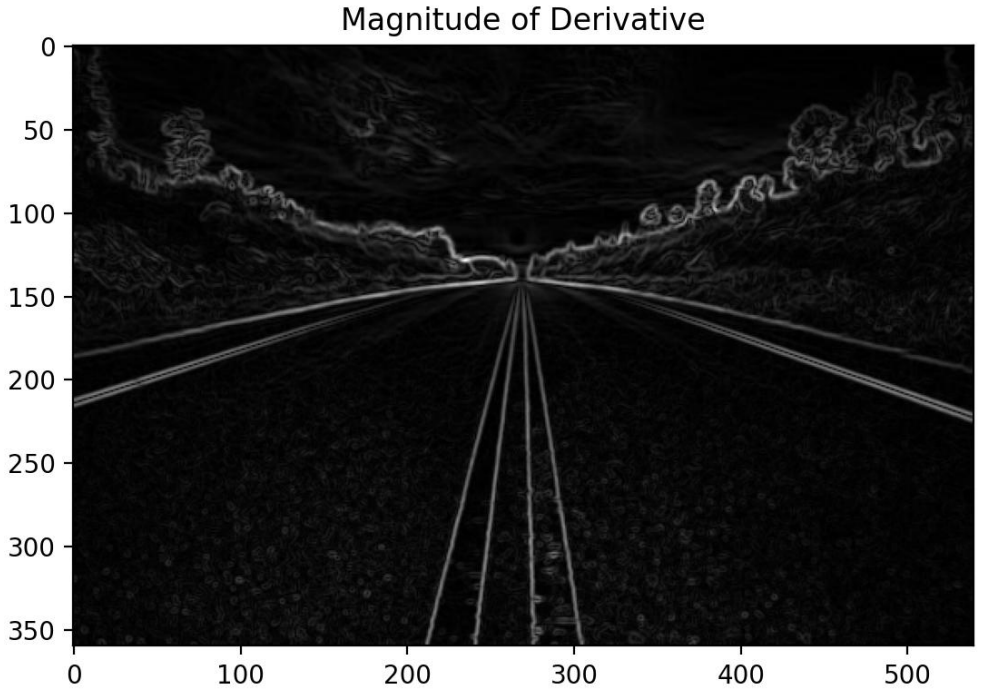
- Remember filtering in Section 2?
- To deal with unreliable models, we can take a sample of the predictions of the model over a certain period of time, and use the most likely class over, say, the past 3 seconds to decide what sign is visible.
- We can also apply weighting
  - As we get closer to the sign, might be more likely we predict it correctly since it's bigger in the frame?
  - Apply greater weight to new data samples

Proper Cornering

- When we navigate around corners, we can apply classical computer vision techniques to try and aid in ensuring the corner has been executed properly
- Take a picture of a road. How can we determine if we are properly aligned with the road?
  - Centrepoint alignment w/ grayscale sensor
  - Edge alignment w/ image filtering

Image Filtering

- Much like we can filter sensor data, we can filter image data
- Image is represented by (R, G, B) array
- At the edge of an object in an image, colour changes significantly over only a few pixels in distance
- We can represent this with the *derivative*, or *rate of change* of pixel intensity!
- We can combine rate of change in x and y directions to cleanly extract road edges



Edge Detection Code

- You can use a function like that implemented in edge_detection() on the right to implement this in your car, except feeding in the most recent frame from the camera rather than a JPG
- How do we use this logic to do positioning?
  - Use some thresholding to ensure the edges are the same distance apart on either side
  - Can make sure you are tracking the centerline by looking at the intensity values directly in the middle of the image

```
# implement derivative based edge detection
import numpy as np
import cv2
import matplotlib.pyplot as plt

def edge_detection(img):
    # convert to gray scale
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    # apply sobel filter
    sobelx = cv2.Sobel(gray, cv2.CV_64F, 1, 0, ksize=5)
    sobely = cv2.Sobel(gray, cv2.CV_64F, 0, 1, ksize=5)
    # calculate magnitude and direction
    magnitude = np.sqrt(sobelx**2 + sobely**2)
    direction = np.arctan2(sobely, sobelx)
    return magnitude, direction

def main():
    img = cv2.imread('edge.jpg')
    magnitude, direction = edge_detection(img)
    plt.figure()
    plt.imshow(magnitude, cmap='gray')
    plt.title('Magnitude of Derivative')
    plt.show()

if __name__ == '__main__':
    main()
```

INTEGRATION OF ALL COMPONENTS

Tutorial - Section 4

Tutorial Time

- With the variety of techniques explained, you should be able to implement a self-driving car that can travel through the grid system and follow signs as needed
- You have lots of freedom as to how you will implement each function: ensuring centering, sign recognition, etc

- As you go through this process, I will be available to help with any particular implementation questions you may have
- The exact approach is up to you- try different things and tune your algorithms as you go!


UNIVERSITY OF
WATERLOO
FACULTY OF ENGINEERING