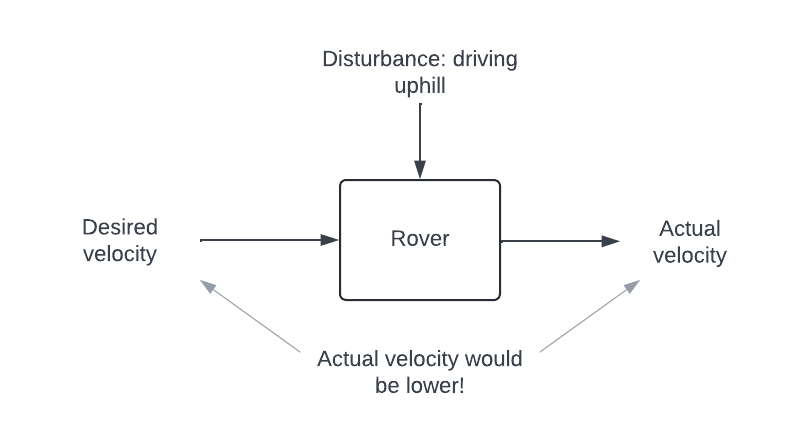
**Lesson 6 – Topics to Explore Next**

Congratulations on getting to this point! Many of the topics in these lessons are complicated and the work can be challenging. The point of this workshop is to help you build a foundational understanding of how robots work, which you can build on by exploring more complex topics. In this lesson we’ll summarize and share resources related to advanced robotics topics which may inspire advanced features for you to implement on future robotics projects.

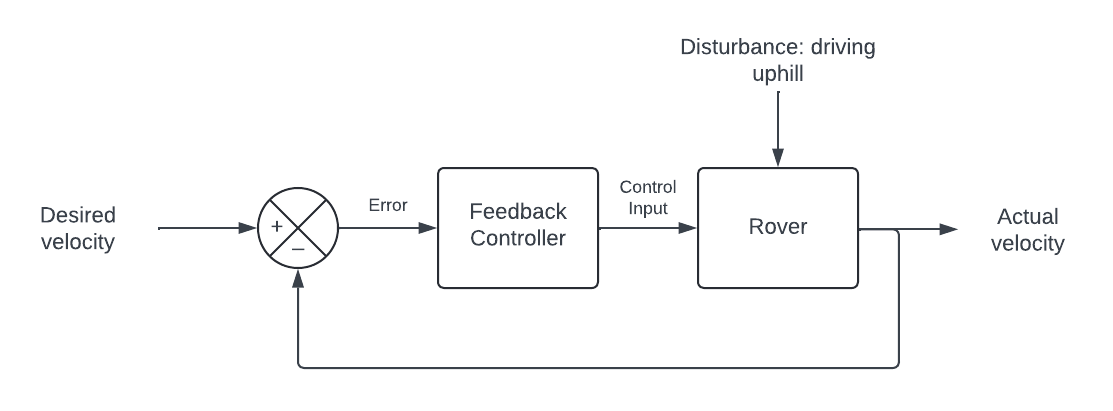
**Feedback Control Schemes**

In lesson 3 we assumed there was a linear relationship between the voltage we apply to our motors and the speed our motors run at. You may remember this was an example of *feed-forward control*, where we establish a relationship between our inputs and outputs and assume it always holds true. This works well enough for our rover when there isn’t any resistance applied to the motors, but what would happen if our rover was driving up a hill? In that case, there would be resistance on the motors our motor controller doesn’t account for, and the rover would drive slower than desired.



**Figure 6-1:** Feed-forward control may not adequately handle disturbances

Maybe it's important to you that the rover drives at a specific speed even when there are disturbances. In that case, you may want to try basing your control input not on your desired velocity, but on the difference between your desired velocity and your actual velocity, which you can measure with motor encoders.



**Figure 6-2:** Feedback control bases the control input on the difference between a desired output and the actual output

We’ll call the difference between the desired velocity and the actual velocity *error*. We can then plug the error into the *feedback* *controller*, which comes up with an appropriate control input based on the error. This kind of system has a few advantages over feed-forward control. Think again about our little rover struggling to climb a hill. As it drives over flat ground, we take the measured velocity from the motor encoders and subtract it from the desired velocity. Because there aren’t any disturbances, the error is zero and the feedback controller sees no need to change anything. When our rover starts climbing the hill, we again measure the velocity and subtract it from our desired velocity. This time though, there is a disturbance (the hill) and so we have a non-zero error. The feedback controller sees this error and decides to increase the control input, which increases the rover’s measured velocity until it matches the desired velocity. We don’t have to assume the relationship between inputs and outputs always holds true; this feedback control system is tolerant of disturbances.

What is a feedback controller though? What's inside that black box that converts an error value into a useful control input? A feedback controller is just a math function that some engineer has tuned to deliver some sort of desired system behavior. This tuning may be achieved through careful mathematical analysis, trial and error, or both. Sometimes designing/tuning these math functions can be pretty involved, but a relatively approachable feedback control scheme to implement is [PID control](https://www.eecs.umich.edu/courses/eecs373.w05/lecture/control.html). There are [PID libraries](https://www.arduino.cc/reference/en/libraries/pid/) out there to simplify this work, but it's not too difficult to [write your own PID controller](http://brettbeauregard.com/blog/2011/04/improving-the-beginners-pid-introduction/) in Arduino.

Maybe you’re thinking this seems like a lot of effort, and even if we can get our rover to run at the speed we expect it to, so what? Will that really change how well our rover performs? If you feel that way, fair enough. You’re thinking like an engineer– trying to figure out where your effort is best spent to produce an acceptable result with finite time and resources. In this case using feedback control to make our rover’s motion more precise is more of an exercise than a necessity. But there are other cases where that isn’t true and feedback control systems are necessary for a functional product. A great hands-on way to experiment with feedback control systems is to build an [inverted-pendulum robot](https://www.shaysackett.com/inverted-pendulum-robot/) (IP robot). An inverted pendulum (like a segway or balancing a yardstick on your hand) is an unstable system. When left to its own devices it falls over. One incredible trait of feedback controllers, though, is that a well-designed feedback controller can stabilize certain unstable systems. With an IMU such as the ICM 20948 we used in lesson 5 and a properly tuned PID controller, it’s possible to build an inverted-pendulum robot that balances itself.

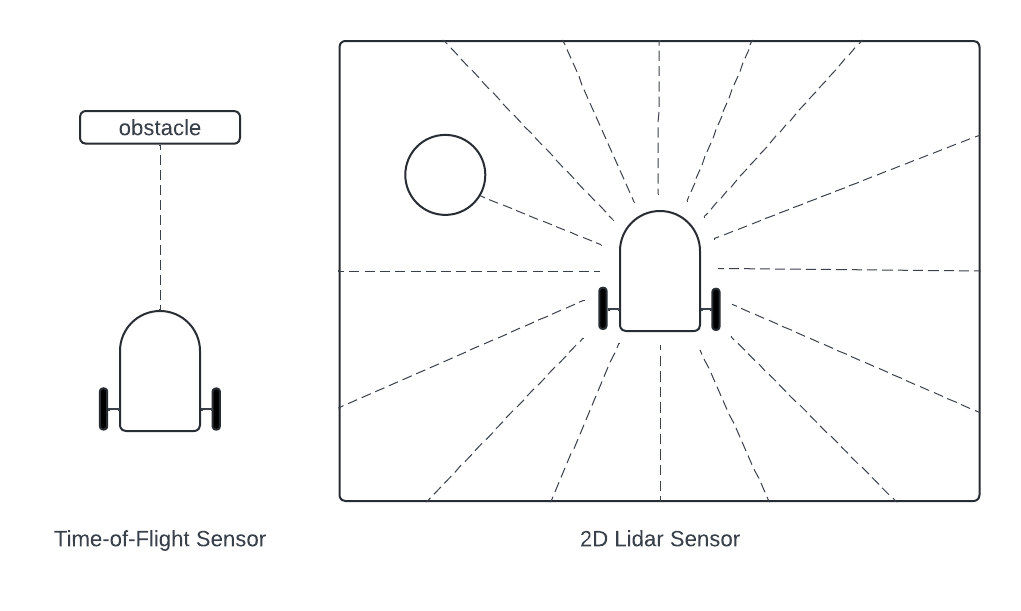


**Figure 6-3:** Example of an inverted-pendulum robot

Feedback controllers are used all over the place. You can find them in aircraft autopilot systems, in pointing systems for satellites, in control systems for robotic arms, or in cruise control in your car. Next time you build a robot, you almost certainly will be able to find a way to improve it with feedback control.

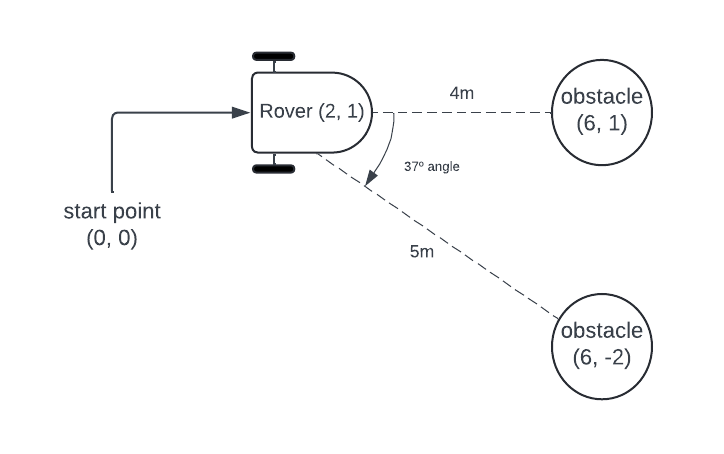
**Localization, Mapping, and Motion Planning**

In lesson 4 we used a time-of-flight sensor as a sort of digital tape measure. In this context, when used only to measure the distance to a single point, time-of-flight sensors have limited usefulness. Take the same time-of-flight sensor, though, attach it to a motor, and measure a whole bunch of points each time it turns, and suddenly you’re able to create a floor plan of the area around you. [Thats what a 2D lidar is](https://articulatedrobotics.xyz/tutorials/mobile-robot/hardware/lidar).



**Figure 6-4:** Time-of-flight sensor compared to a 2D lidar sensor

This enhanced sensing capability brings with it the opportunity to try some nifty tricks. We’ve used forward kinematics to track how far our rover drives, but with a bit more effort we can [improve our wheel odometry to track our robot’s position and orientation](https://medium.com/@nahmed3536/wheel-odometry-model-for-differential-drive-robotics-91b85a012299) relative to where it started. Once we can determine our robot’s position and orientation relative to its start point– which we’ll call its *pose*– then we can map the location of obstacles relative to the start point. For instance, let's say we put our rover down and turn it on. It drives up 1 meter and right 2 meters, then detects an obstacle straight in front at 4 meters and a second obstacle 37 degrees to the rover’s right at 5 meters.

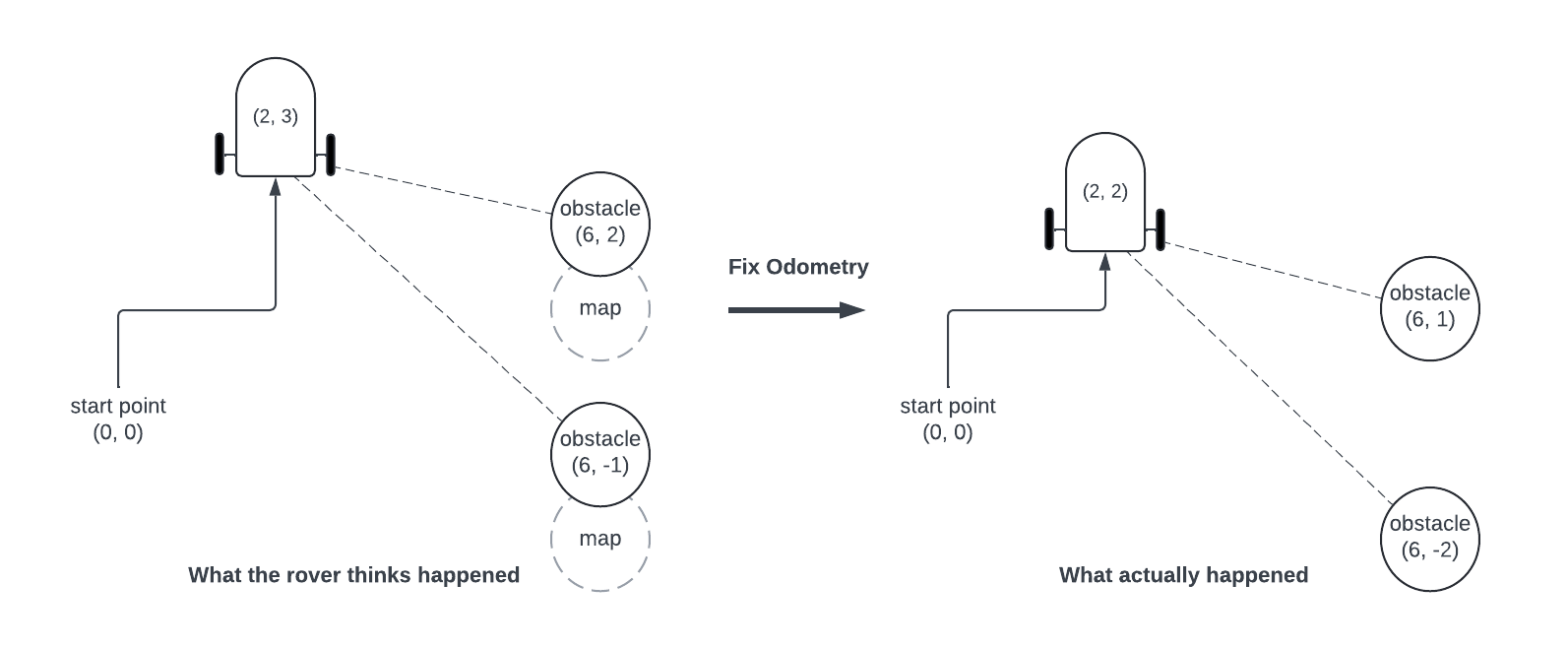


**Figure 6-5:** The rover starts, drives to position (2, 1), and sees two obstacles

Knowing the robot’s pose and the angle and distance to the obstacles gives us all the information we need to use trigonometry to place those obstacles relative to the start point. And with that, we’ve generated a map.

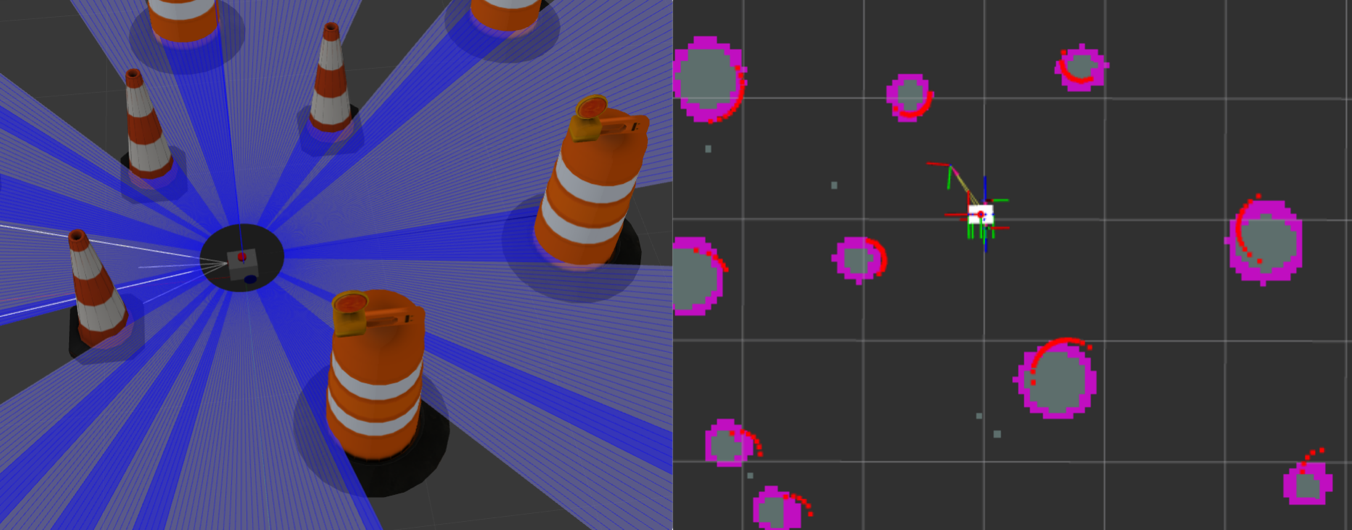
There is a problem here though. Back in lesson 4 you may have wondered if odometry would even work on surfaces such as sand or dirt where the wheels might slip. If so, you realized something important: wheel odometry can be unreliable. Even on grippy, level ground, wheels always slip a slight bit, and so as time goes on and the rover drives further and further, the rover’s assessment of where it is becomes less and less accurate. Slippery surfaces only make this issue worse. We need some way of frequently correcting the robot’s odometry to make sure the error never gets too large.

Conveniently, we’ve already got a map (or at least part of one). Much like a human would, our robot can use the obstacles around it as landmarks to double check its position and correct if its odometry becomes inaccurate. Say our rover from figure 6-5 now turns left 90 degrees and starts driving forward. Based on its wheel encoders, it thinks it’s driven 2 meters forward, but those obstacles it just mapped now appear like they’re at positions 6, -1 and 6, 2. What happened? Well, those obstacles didn’t move, so the rover must not have driven as far as it thought it had. Maybe it got bogged down in some sand, or maybe it started slipping on some ice. Regardless, the rover can change its guess of where it is to bring its observations back in alignment with the map.



**Figure 6-6:** The rover fixes its odometry to align with the map

This is an iterative process. The rover can’t build a map without reliable odometry, and it can’t maintain reliable odometry without a map. On their own odometry and mapping are at best unreliable, but used together, the rover can build an accurate picture of its environment and where it is within that environment. This process is known as Simultaneous Localization and Mapping (SLAM).



**Figure 6-7:** Simulation of SLAM on differential drive rover

In lesson 5 we had our rover navigate around obstacles, but when we have a map and can place ourselves in it, we can do more than just blindly drive forward and respond to nearby obstacles. We can pick a goal pose and plan routes there that minimize the distance we must drive and the odds of colliding with an obstacle.

There are a lot of ways of picking a path through a map. One approach is to break the map down into discrete cells the rover can occupy, called *nodes*. We can then assign a “cost” to entering nodes that are close by obstacles, which is how we create a *costmap*.



**Figure 6-8:** An example costmap generated from a SLAM simulation; colors nearby obstacles represent the cost of entering those nodes

Any given path through the map will have a cost associated with it. The cost is based on the length of the path (shorter paths being better) and how close that path gets to obstacles. The goal of the rover is to find the path to the goal pose with the least possible cost. [One method to do this](https://en.wikipedia.org/wiki/A*_search_algorithm) builds the path incrementally starting from the rover’s initial position. The rover has a few different neighboring nodes it could add to the path, so it needs a way of determining which node to choose. To do this, the rover chooses the node that minimizes this function:

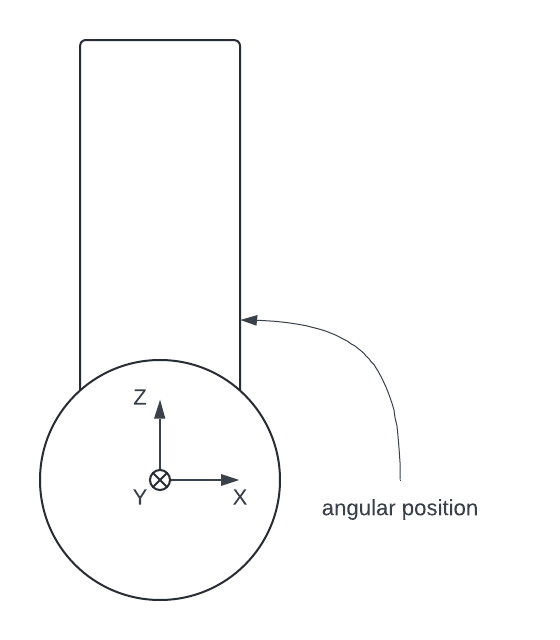
In this function, the cost of the path to node n is g(n) and h(n) is an estimate of the cost of the path from node n to the goal, which is usually based on the distance to the goal. Adding the cost of the path so far and the estimate of the cost of the rest of the path gets us an estimate of the total cost of a path through node n, which we call f(n). The node we’ll add to the path is the neighboring node with the smallest f(n).

The rover will repeat this process over and over, expanding a path into a new node each time, until one path finally reaches the goal. [The result of this](https://upload.wikimedia.org/wikipedia/commons/5/5d/Astar_progress_animation.gif) is a tree of possible paths that branches out from the start point before converging towards the goal as the optimal path becomes more apparent.

**State Estimation**

In lesson 5 we explored a few functions of an Inertial Measurement Unit. IMUs use accelerometers, gyroscopes, and magnetometers to track motion, but each of those sensors on their own are limited in their usefulness. In particular, gyroscopes are very precise but [gyroscopic “drift”](https://www.phidgets.com/docs/Gyroscope_Guide) causes them to become highly inaccurate over time, while accelerometers are accurate but noisy and imprecise. All sensors are like this; each has strengths and weaknesses that limit their usefulness.

Luckily, [*state estimation*](https://en.wikipedia.org/wiki/State_observer) and [*sensor fusion*](https://en.wikipedia.org/wiki/Sensor_fusion) techniques allow us to combine measurements from multiple sensors to take advantage of the best traits of each. One technique is known as the [Kalman filter](https://en.wikipedia.org/wiki/Kalman_filter). Just like with feedback control, the inverted pendulum robot is a great project to explore how a Kalman filter works. Let's consider an inverted pendulum robot trying to determine its state with an accelerometer and a gyroscope.



**Figure 6-9:** Side profile of inverted pendulum robot with IMU axes labeled

The IMU gyroscope can measure the rate of change of the robot’s angular position about the Y-axis (the angular velocity with which it tips over). We can integrate this measurement to predict the robot’s angular position, but due to gyroscopic drift this prediction will always be a bit inaccurate. We can measure the angular position with the accelerometer and atan2 just like we did in lesson 5, but this measurement will be noisy and imprecise. The 2-step process of Kalman filtering will help us estimate the angular position better than either of these sensors could alone.

The first step in Kalman filtering is the prediction step. Like we just discussed, we can integrate the rate measurement from the gyroscope to predict the angular position. Then we estimate the *uncertainty* of that prediction based on the elapsed time since the last estimate. Next, in the update step, we measure the angular position with the accelerometer and find the difference between the measured angular position and the predicted angular position. When the uncertainty is small and the difference is small, we are more confident in our prediction, but when the uncertainty is large and the difference is large, we’ll be less confident in our prediction. In theory, as the prediction from the gyroscope drifts away from reality, the measurements from the accelerometer should pull it back towards the real position. The result is an estimate of the angular position that should be better than either the measurement or the prediction alone.

The theory behind Kalman filtering can be intimidating ([though a good guide helps a lot](https://www.alanzucconi.com/2022/07/24/kalman-gain/)), but if you’re prepared to learn you can [implement your own Kalman filter in Arduino](https://github.com/Whitw-pers/Inverted_Pendulum_Bot/blob/main/IMUtestCode/IMUtestCode.ino). If you want to try using a Kalman filter but don’t yet feel ready try building one from scratch, there are [Arduino libraries for Kalman Filtering IMU data](https://www.arduino.cc/reference/en/libraries/kalman-filter-library/). We’ve only discussed the Kalman filter in the context of IP robots and IMUs, but this technique is applicable to so many different systems and sensors. The first implementation of a Kalman filter was famously [used for the Apollo navigation computer](https://cybernetist.com/2019/01/13/apollo-kalman-filter-and-go/)!

**Computer Vision**

If you, dear reader, are in fact human, you probably realize how important our sense of sight is for understanding and moving around our environment. Though the sensors we’ve used on our rover so far can be quite powerful (especially with the techniques we’ve discussed in this lesson!), our rover’s sensing capabilities are nothing close to the marvel of the human visual system.

Until relatively recently, the idea of a [machine that sees](https://en.wikipedia.org/wiki/Computer_vision) was stuck firmly in the world of science fiction. For the last half century, however, clever artificial intelligence researchers first crawled, then walked, and with the advancement of [deep learning](https://en.wikipedia.org/wiki/Deep_learning) are now running into a world where computers can understand their environment through image data. Simply put, computers have recently gotten a lot better at [mimicking human vision](https://www.ibm.com/topics/computer-vision).

A key concept necessary to wrap your head around computer vision is a mathematical operation called the [*convolution*](https://en.wikipedia.org/wiki/Convolution#Applications). Convolutions are often [used in image processing](https://medium.com/@timothy_terati/image-convolution-filtering-a54dce7c786b) to compress, filter, and/or extract information from images. When we’re trying to feed an image into a [*neural network*](https://en.wikipedia.org/wiki/Neural_network_(machine_learning)) (the technology behind deep learning), we use the convolution to extract features (what we call the inputs to a neural network) from an image. "Extending” a neural network by adding a convolution step in this way gets you a [Convolutional Neural Network](https://www.geeksforgeeks.org/introduction-convolution-neural-network/) (CNN), which is the underlying technology behind many computer vision systems.

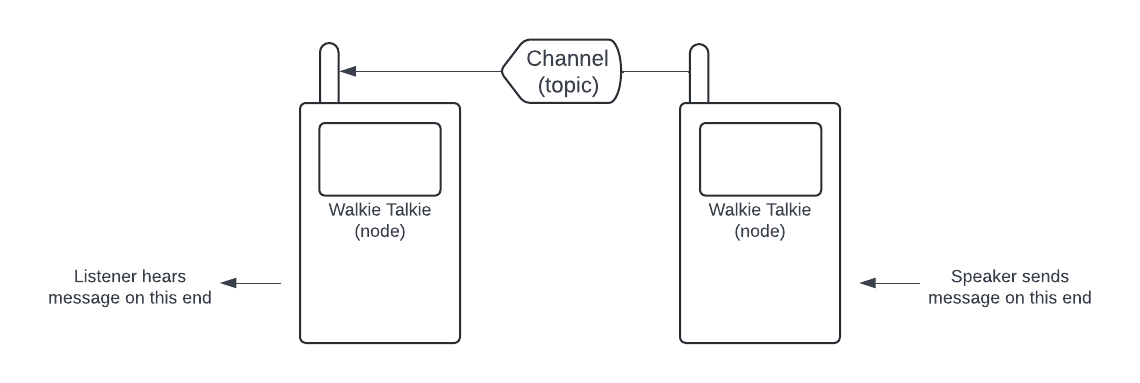
Thankfully, to play around with computer vision you don’t need to build your own CNN. [OpenCV](https://opencv.org) and the [PixyCam](https://pixycam.com) are two tools available to us that make computer vision techniques more accessible. The PixyCam in particular is well-adapted for use with Arduino (and other) microcontrollers. It handles the image processing on board and sends the data it extracts to your Arduino over your choice of common communication protocol. It’s a great way to quickly [get up and running with object identification and tracking](https://www.instructables.com/Auto-Turret-With-Pixy-and-Nerf-Gun/).

OpenCV on the other hand is a bit more involved, but with that added difficulty comes a more powerful toolset. OpenCV is open-source software you can run on some single-board computers (SBCs) such as a [Raspberry Pi](https://www.raspberrypi.com/products/raspberry-pi-5/) or [Nvidia Jetson](https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/). Be careful though, computer vision is computationally expensive and not all SBCs will be able to run OpenCV at any useful speed. If you have the right hardware, though, you can hook it up to a USB webcam and get started with some [face](https://core-electronics.com.au/guides/face-identify-raspberry-pi/) or [gesture recognition](https://core-electronics.com.au/guides/hand-identification-raspberry-pi/).

**The Robot Operating System**

There's a saying that “a good programmer never writes the same code twice”. In lessons 3, 4, and 5, we did our best to adhere to this guideline by organizing our code into modular functions we can copy paste into future projects. There are other techniques we could employ (such as object-oriented programming) that would make our previously written code even more plug-and-play. At the extreme end of this modularity spectrum, though, sits a tool called the [Robot Operating System](https://www.ros.org) (ROS).

“Operating system” is a bit of a misleading way to describe ROS; it's nothing like Windows or MacOS. ROS is really a vast collection of open-source libraries and software packages that make using advanced robotics tools and techniques simple (compared to building them from scratch). The way ROS achieves this (relative) simplicity is through a system of *nodes*, *messages*, and *topics*. Nodes are like walkie talkies and topics are the radio channels that connect them. Messages are the information the walkie talkies (nodes) transmit on the channels (topics).



**Figure 6-10:** Visual representation of nodes, topics, and messages

By combining nodes and connecting them to each other in intelligent ways with topics, the network we create can perform complex autonomous tasks such as navigating spaces, tracking objects, and more.



**Figure 6-11:** A *heavily* simplified example of a ROS network for a diff drive rover that performs autonomous navigation and chases a ball; the boxes are nodes, arrows with /[topic\_name] are topics

ROS has become a bit of a ubiquitous tool across the world of robotics: it’s used in the upper echelons of robotics research ([a](https://leggedrobotics.github.io/viplanner.github.io/) – [few](https://arxiv.org/pdf/2402.19341) – [examples](https://arpg.github.io/papers/biggie_field_robotics_2023.pdf)), and it’s used in certain beginner-oriented [projects designed to introduce people to advanced robotics](https://www.turtlebot.com) topics. ROS can feel overwhelming to a beginner, but a good guide helps. One of the best ways to introduce yourself to ROS is to follow Josh Newans excellent [Building a Mobile Robot tutorials](https://articulatedrobotics.xyz/tutorials/) (and the [accompanying Youtube videos](https://www.youtube.com/playlist?list=PLunhqkrRNRhYAffV8JDiFOatQXuU-NnxT)). Once you feel relatively comfortable with [*Unified Robot Description Format*](https://articulatedrobotics.xyz/tutorials/ready-for-ros/urdf) (URDF) files, [ROS’s transform system](https://articulatedrobotics.xyz/tutorials/ready-for-ros/tf), [ROS2 Control](https://articulatedrobotics.xyz/tutorials/mobile-robot/applications/ros2_control-concepts), and Gazebo ([Gazebo install](https://classic.gazebosim.org/tutorials?tut=install_ubuntu) can be tricky!), you may want to jump into [F1tenth](https://f1tenth.org) to stretch your robotics muscles even further.

Hopefully this lesson has given you some ideas for how to improve your robotics projects in the future, and maybe it's also opened your eyes to how vast the robotics community is. Creating anything worthwhile requires the help of others. If you’re struggling with a problem, reach out to the larger community for help (for example, the [discussion page for the robotics workshop](https://github.com/Whitw-pers/my_rover/discussions)). If you figure out something cool, share it so others can learn from you. Exciting engineering isn’t something you do alone.