Intro to Robotics II Course Manual

Ross L. Hatton

December 31, 2025

Part I
Mobile systems

0 Build PiCar-X

Before starting, switch off the HAT board and plug in the battery to charge it as per the instructions at https://docs.sunfounder.com/projects/picar-x-v20/en/latest/python/python_start/power_supply.html#charge. The servos need power to be zeroed during assembly.

Follow the instructions in the PiCar-X pamphlet and the video at https://www.youtube.com/watch?v=i5FpY3FAcyA to build the robot car. A few notes during your build:

- Turn off the Robot HAT power switch before connecting the battery to avoid directly powering the Raspberry Pi and risking a short circuit.
- Make sure to use the smaller flat-ended servo horn screws, and only tighten until the screw is flush with the servo horn. The use of the longer sharp ones can jam the servo mechanisms, and overtightening can break your servo.
- Be very careful of the camera ribbon cable during assembly. They are fragile and will need to be bent at fairly sharp angles. When connecting the FFC cable, do not rely on the printed text on the cable, as it can appear on either side. Instead, ensure that the silver contacts on the cable align with the silver contacts on the camera port.
- Be careful when popping the plastic rivets into place. They break easily. It would be kind to save your spares to give to classmates in case someone breaks too many.
- During assembly, you will need to zero the PiCar's servos. Unlike the video instructions, you do not need to flash the Raspberry Pi OS at this stage. The Robot HAT board now includes a convenient button to zero the servos without requiring an OS. Flashing the Raspberry Pi OS will be completed in the next step.
- When turning off the Raspberry Pi, make sure that it is not actively processing anything before flipping the switch. At this point, look to make sure that light patterns are solid or at most steady blinking; in later stages we will set up the system for smooth shutdown commands.

1 Set up and secure your Raspberry Pi

Read through the instructions here all the way through before carrying out the steps in the linked documents – not all steps in the external documents will be followed, and some will be modified.

1. Insert the SD card into your SD card reader
2. Follow the Python-specific PiCar setup instructions at

   `https://docs.sunfounder.com/projects/picar-x-v20/en/latest/python/python_start/installing_the_os.html`

   Install the recommended OS version (64 bit Bookworm). If the Raspberry Pi Imager fails, format the SD card to be non-bootable drive using Rufus first.

   Although we will later create a more secure way to sign in to your Pi, please create your own username and password as a temporary measure for better security. It's also strongly suggested that you give your Pi a custom name to avoid accidentally connecting to the wrong Pi.

   The Raspberry Pi Imager allows you to configure only one WiFi network initially. You can add more WiFi networks later once you connect to the Raspberry Pi. To connect to the Robotics network on campus, use: `ssid=''robotics''`, no password is required. Ensure you check the `Hidden SSID` option, as this WiFi network is hidden.
3. After the SD card is back in the Pi, if you are on a personal network, use the command prompt to do

   ```
   ping raspberrypi.local
   ```

   to verify that your Pi is on the same network as your computer, replacing `raspberrypi` with your Pi's name if you changed that during setup. If communication is established, use

   ```
   ssh user@raspberrypi.local
   ```

   to connect to the Raspberry Pi, replacing `user` with the username you created in step 2, and using the accompanying password you also created to sign in.

   For those using the Raspberry Pi on campus, the hostname is tied to the ID number of your Raspberry Pi. Connect your laptop to `eduroam` or `OSU Secure`, use the following command to ssh into your Raspberry Pi, substituting your Raspberry Pi ID number for `#`:

   ```
   ping PiCar#.engr.oregonstate.edu
   ssh user@PiCar#.engr.oregonstate.edu
   ```

   Remember if swapping between home and school networks to always use the appropriate hostname for the network. If you are not able to make the connection, some trouble-shooting things to try are:

(a) Make sure that the Raspberry Pi has successfully connected to the network. Log into your network router and look to see if there is a computer named RASPBERRYPI on the network. If there isn't one, repeat the previous step, double-checking your network name and password.

(b) If you have an Ethernet cable available, connect your Raspberry Pi directly to the router. Then use SSH to connect to the Pi and use the `nmtui` tool (see below) to set up the WiFi.

(c) If you have an Ethernet cable but no access to a router, connect your laptop directly to the Raspberry Pi using the Ethernet cable. Ping `raspberrypi.local` to find the Pi's IP address (your laptop is not running a DHCP server, the Pi may only have an IPv6 address, but it can still be used for SSH). Then use SSH to connect to the Pi and set up the WiFi.

(d) If the SSH can connect, but the password is rejected, someone else on your network may have a Raspberry Pi with the same name as yours. In this case, log into the router and find the IP addresses of all RASPBERRYPI computers on the network, and then try the IP addresses directly. (In the setup process, we will change the name of the Raspberry Pi).

(e) If you have connected to a computer named RASPBERRYPI before, you may get a security warning and your computer may refuse to connect to your new computer. In this case, you can use

```
ssh-keyscan $target_host >> ~/.ssh/known_hosts
```

If you need to do this on a windows machine, there are additional instructions described in various places online.

(f) Change the name of your Raspberry Pi by using

```
sudo raspi-config
```

and then `System Options > Hostname`.

(g) Set the localization options on your Raspberry Pi to US. Run

```
sudo raspi-config
```

then scroll down to `en_US.UTF-8`. Enable it by tapping the space bar, then use "return" to move to the next screen.

4. Once connected to your Raspberry Pi, you can save additional WiFi networks using the `nmtui` tool. This tool provides a simple interface for managing network connections. Follow the detailed instructions at:

   `https://www.howtogeek.com/devops/how-to-manage-linux-wi-fi-networks-with-nmtui/`

   When adding WiFi configuration to all users, you may need `sudo` for superuser authorization.

   Note that the `nmtui` does not support configuring hidden SSID WiFi (e.g., the robotics network on campus) or adjusting auto-connect priority. To set up these features, you will

need to first use `nmtui` to create your WiFi profile, then update the settings using one of the following methods:

Use the command line tool `nmcli` by the commands, replacing "#Your-WiFi-Profile-Name#" with your profile name:

```
sudo nmcli connection modify #Your-WiFi-Profile-Name# connection.autoconnect-priority 10
sudo nmcli connection modify #Your-WiFi-Profile-Name# wifi.hidden true
```

or editing the network profile directly

```
sudo nano /etc/NetworkManager/system-connection/#Your-WiFi-Profile-Name#.nmconnection
```

adding `hidden=true` under `[wifi]` and `autoconnect-priority=10` under `[connection]`.

5. Carry out some basic setup operations for securing your Raspberry Pi. More information on many of these steps can be found in the official Raspberri Pi OS documentation at

   `https://www.raspberrypi.com/documentation/computers/os.html`

   (a) For your robots, I recommend having the `sudo` command require a password and then setting a long timeout on it, by using

   ```
   sudo visudo
   ```

   and then adding a line like

   ```
   Defaults:USER timestamp_timeout=60
   ```

   (where USER is your username, and 60 is the number of minutes I've specified for the timeout)

   (b) Make sure your Pi software is up-to-date by using

   ```
   sudo apt update
   sudo apt full-upgrade
   ```

   (c) To enable safer (possibly passwordless, if you didn't assign a passphrase to your SSH key) login to your Raspberry Pi, you need to generate a new SSH key or locate an existing one on your computer, then add the key to the SSH agent, by following the instructions at:

   `https://docs.github.com/en/authentication/connecting-to-github-with-ssh/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent`

   Then manually copy the public key to the `~/.ssh/authorized_keys` file on your Raspberry Pi following the guide at:

   `https://www.raspberrypi.com/documentation/computers/remote-access.html#manually-copy-a-public-key-to-your-raspberry-pi`

Note that the generated SSH key file might be named `id_ed25519.pub` instead of `id_rsa.pub`. Run the `scp` command from a terminal on your computer, not from a terminal SSHed into the Raspberry Pi. This is because the `scp` command copies the file from your computer to the Raspberry Pi over SSH.

If you want to add more known keys to your Raspberry Pi, do not overwrite the `authorized_keys` file. Instead, append the contents of your `id_ed25519.pub` file as a new line in the existing `authorized_keys` file.

After setting up the key, make sure you are actually able to log in without a password using the SSH key, then you can disable password authentication on your Raspberry Pi to enhance security by

```
sudo nano /etc/ssh/sshd_config
```

and replace `PasswordAuthentication yes` (if it's commented, uncomment it first) with `PasswordAuthentication no` and save.

From now on, you can log in by providing your hostname and private key location in PuTTY (or your preferred SSH client)

(d) Use `apt` to install `ufw` with

```
sudo apt install ufw
```

and then follow steps 2-4 of

`https://www.digitalocean.com/community/tutorials/how-to-set-up-a-firewall-with-ufw-on-ubuntu-20-04`

to turn on the `ufw` firewall. Then, set the options `sudo ufw allow ssh` and `sudo ufw limit ssh/tcp`.

(e) Install `fail2ban` as per the instructions at

`https://pimylifeup.com/raspberry-pi-fail2ban/`

to block brute force attempts at your passwords. For this class, you do not need to adjust settings, simply copy the `jail.conf` into `jail.local`, and modify `backend = systemd` under (the second) `[sshd]`. Then ensure that Fail2ban is installed correctly and the service is active by

```
sudo systemctl enable fail2ban
sudo systemctl start fail2ban
sudo systemctl status fail2ban
```

Deliverables

1. Reflection questions:
   (a) What did you learn while completing the tasks in this lesson?
   (b) If you see any shortfalls with the approach we have taken above, please suggest improvements.

To receive full credit for the weekly reflection assignments, your answers should meaningfully engage with the question – bullet-point lists naming the topics covered in a given week are not

sufficient responses. Tell me something about what your starting point for knowledge on the week's topics was, or how it relates to work you've done in the past. Then tell me something specific that you now know about that you didn't before you worked on this assignment (or if this is all stuff you have extensive knowledge of, turn the "starting point" into a story about using it.

2 Motor commands

Once your robot is assembled and you have its computer set up, the next step is to get a handle on the interfaces between the coding you can do in Python and the physical hardware. In this lesson, we will

1. Install SunFounder's Python code (and dependencies) for robot control
2. Create a copy of `picarx.py` with modifications to enable offline testing and improve the system performance
3. Create a "shadow" version of the `PiCar` package to enable offline testing
4. Write some basic maneuver functions to demonstrate control over the motors

2.1 Gather code pieces

1. On your Raspberry Pi, follow the instructions at these links to install the required libraries:

   `https://docs.sunfounder.com/projects/picar-x-v20/en/latest/python/python_start/install_all_modules.html`

   `https://docs.sunfounder.com/projects/picar-x-v20/en/latest/python/python_start/enable_i2c.html`

2. Note that the `install.py` script takes a while. It is recommended that you run this script on wired power, rather than using batteries.
3. After installing the libraries, explore the examples for the motors, servos, and sensors. These examples can help verify that your assembly and electrical connections are functioning correctly.

2.2 Make the robot move

Before really working with the robot, we should make some improvements to the stock code. I know you're here to play with robots, so let's see the car move before we go further. The PiCar-X manual describes the provided demo code; not all of it works as well as advertised. Try running the programs and see what happens. In order to properly calibrate the servo angles and motor directions, you can use the calibration helper by following the instructions at

`https://docs.sunfounder.com/projects/picar-x-v20/en/latest/python/python_calibrate.html`

2.3 Configure your Raspberry Pi to connect to a GitHub account (or other Git repository)

1. Create an account at GitHub.com. If you have a GitHub account already, you can use it. If you have a different Git setup that you prefer to use, go ahead and use it, and modify the other setup instructions as needed.

2. Create a new Git repository in your account named `RobotSystems`. Don't make a `README` file for it, as that file can cause headaches during setup. We'll make one later.

3. Set up SSH keys on your GitHub account.

   (a) Generate a new SSH key or locate an existing one on your computer by following the instructions at:

   `https://docs.github.com/en/authentication/connecting-to-github-with-ssh/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent`

   If you previously created an SSH key for passwordless SSH login to your Raspberry Pi, you can reuse that key.

   (b) SSH into your Raspberry Pi, and then follow the same instructions to create a SSH key **on your Raspberry Pi** and add it to the SSH agent. This will be similar to the process you used to create the SSH key that lets you log into your Raspberry Pi from your computer. On the "Generating a new SSH key and adding it to the ssh-agent" page, make sure you are reading the "Linux" instructions (because you are performing operations on the Raspberry Pi).

   (c) Set a non-empty passphrase for the SSH key on your Raspberry Pi. This means that your key is encrypted on your Raspberry Pi, and cannot trivially be used by others if your robot is lost or stolen. (This precaution is more important on something like a Raspberry Pi than on a computer with disk encryption enabled.)

   (d) Follow the instructions to add both the SSH key from your computer and the one from your Raspberry Pi to your GitHub account

   `https://docs.github.com/en/authentication/connecting-to-github-with-ssh/adding-a-new-ssh-key-to-your-github-account`

   (e) You will also need to set your username and email in Git:

   `https://docs.github.com/en/get-started/getting-started-with-git/setting-your-username-in-git`

   `https://docs.github.com/en/account-and-profile/setting-up-and-managing-your-personal-account-on-github/managing-email-preferences/setting-your-commit-email-address#setting-your-commit-email-address-in-git`

   (f) Now both your laptop and your Raspberry Pi should be able to pull and push code to and from your private repository on Github.

2.4 Set up coding environment

Make a copy of the existing `picar-x` directory and link it with your repo

1. Make a local `RobotSystems` repository onto your Pi

```
cd ~
mkdir RobotSystems
```

2. Copy all of the `picar-x` code into the `RobotSystems` repository

```
cp -a picar-x/. RobotSystems/
```

3. Copy all of the `robot-hat/robot_hat` code into the `RobotSystems` repository

```
cp -r robot-hat/robot_hat RobotSystems/sim_robot_hat
```

4. Copying all of the folder contents means that the new folder will inherit git settings for sunfounder's picar repo. Configure them for your personal repo and push to GitHub

```
git branch main
git checkout main
git remote set-url origin git@github.com:<YourGithubUsername>/RobotSystems.git
git add .
git commit -m ''Adding PiCar Files''
git push --set-upstream origin main
```

   Note that you only need to set the upstream once. Next time you want to do any updates, just do

```
git add .
git commit -m ''Some notes''
git push
```

   For more information on using git and Github, see

```
https://training.github.com/downloads/github-git-cheat-sheet/
```

5. Verify on GitHub that your repo now contains all the `picar-x` files and folders. Edit the `README` to reflect the personal nature of the repo

6. Use `sudo shutdown now` to turn off your car and save your batteries.

7. Clone your Git repository somewhere onto your desktop computer.

2.5 Create basic infrastructure

You will be setting up a remote testing environment, designed to let you test robot code on your desktop before pushing to the PiCar. These steps should be carried out on your desktop computer.

1. Copy `picarx.py` into a new file `picarx_improved.py`

2. Replace the opening lines of `from robot_hat import ...` in `picarx_improved.py` with

```
import os
try:
    from robot_hat import Pin, ADC, PWM, Servo, fileDB
    from robot_hat import Grayscale_Module, Ultrasonic, utils
except ImportError:
```

```
    import sys
    sys.path.append(os.path.abspath(os.path.join(
        os.path.dirname(__file__), ''..'')))

    from sim_robot_hat import Pin, ADC, PWM, Servo, fileDB
    from sim_robot_hat import Grayscale_Module, Ultrasonic, utils
import time
```

This lets you define `sim` functions, which shadow PiCar hardware calls when doing testing on your desktop.

3. Run the code via `python3 picarx_improved.py`. This should return an error.

   Use the exception traceback and the Python debugger to locate the parts of the `sim\_robot\_hat` module that interact with the hardware. You can use any editor or IDE, such as PyCharm or VSCode, to assist with debugging. Modify these sections to make the code runnable: Methods that don't return an output (e.g. methods that send messages over I2C to the Pi pins) can be implemented as a simple `pass`, other methods should return an output of the appropriate type.

   You can refer to the native Python debugger documentation at

   `https://docs.python.org/3/library/pdb.html`

   or the VSCode Python debugger documentation at

   `https://code.visualstudio.com/docs/python/debugging`

   Note that the SunFounder code calls bash commands to read the calibration file, which requires superuser authorization. To avoid running these commands on your laptop (for security reasons), implement an "on-the-robot" check. Below is a step-by-step guide on how to make these modifications:

   (a) Modify `picarx_improved.py`

   Define the `on_the_robot` variable based on the results of the `try-except` block for imports.

   Add an `if-else` statement to avoid passing `os.getlogin()` to the `fileDB` constructor when not running on the robot.

   (b) Modify `filedb.py` in `sim_robot_hat`

   Skip running `self.file_check_create()` since we don't need to read the config when not on the robot.

   Modify the `self.get()` and `self.set()` functions:

   For `get()`, directly return `default_value`.

   For `set()`, do nothing and directly pass the function.

   (c) Modify `motor.py` and `robot.py` in `sim_robot_hat`

   Comment out lines 9-12 and define `User`, `Userhome`, and `config_file` as `None` since we don't need to read and write configuration files when not running on the robot.

2.6 Logging

It is often helpful to have your code write out its progress to the command line, especially when debugging. The most basic way to do this is to insert `print` commands at various points in your code. This approach, however, tends to make your code hard to manage – the operational logic of what you are doing becomes interspersed with many `print` commands, and turning those commands off and on again requires manual commenting and uncommenting.

A better way of having your code display outputs is to use the Python `logging` package. For basic usage:

1. Place

   ```
   import logging
   ```

   at the top of your file.

2. After your package imports, put

   ```
   logging_format = ''%(asctime)s: %(message)s''
   logging.basicConfig(format=logging_format, level=logging.INFO,
                       datefmt=''%H:%M:%S'')
   ```

   to set up your basic logging format. You can then set the logging level to DEBUG with

   ```
   logging.getLogger().setLevel(logging.DEBUG)
   ```

   to have all logging commands at the "DEBUG" level print out to the command line. If you comment out this line, then all DEBUG-level logging messages will be suppressed.

3. At points in your code where you want a message printed to the command line, insert a line

   ```
   logging.debug(message)
   ```

   (where "`message`" is the text you want displayed).

The documentation for `logging` describes more advanced usages, such as setting different levels of logging that you can toggle individually, or setting up the `message` to include current values of system variables.

The logging package fixes one problem from naive `print`-logging, but still leaves the logging messages interspersed with your operational code. An even better approach is to use the `logdecorator` package:

1. Install the `logdecorator` Package The SunFounder installation settings may break the Python package management. You might need to use the `--break-system-packages` parameter with `pip` to install `logdecorator`:

   ```
   pip install --break-system-packages logdecorator
   ```

This will install `logdecorator` to the user directory.

If you need to call Python scripts with `sudo`, use the `-E` flag to preserve the environment settings so that the package installed in the user directory remains accessible:

```
sudo -E python your_script.py
```

2. Place

```
from logdecorator import log_on_start, log_on_end, log_on_error
```

at the top of your file after `import logging`.

3. Immediately before the `def` line for each function or method that you want to display logging information, insert lines like

```
@log_on_start(logging.DEBUG, ''Message when function starts'')
@log_on_error(logging.DEBUG, ''Message when function encounters an error before comple
@log_on_end(logging.DEBUG, ''Message when function ends successfully'')
```

These messages can be set to include any inputs provided to the function, using Python formatting notation. For example, if one of the inputs to the function is a string `name`, this string can be included in the logging message as in

```
@log_on_start(logging.DEBUG, ''{name:s}: Message when function starts'')
```

The `@log_on_end` decorator can also display the result returned by the function, e.g.,

```
@log_on_end(logging.DEBUG, ''Message when function ends successfully: {result!r}'')
```

Note that this logging only displays information as you go into and out of functions, and does not say anything about progress through a function. If you follow good code-abstraction processes and write small functions that carry out well-defined tasks, this will still give you a fine-grained understanding of how your program proceeds through its code.

2.7 Improve the PiCar-X code

1. As you may have noticed when running the demo code, the stock controller can leave the motors running if you terminate a program while they are on. Use the `atexit` Python module to make sure that the motors are set to zero speed when any program incorporating the `picarx_improved.py` code is terminated. A good way to do this is to register the `stop` function at the end of the class initialization.

   If you are using VSCode, be aware that killing the process in the debugger will not trigger the `atexit` functionality. To ensure that `atexit` is called, stop the process by pressing `Ctrl+C` in the terminal instead of terminating it through the debugger.

2. The stock code takes the commanded motor speeds and scales them (probably to prevent the programmer from commanding too slow a speed, which would not provide enough motor power to overcome friction in the system). Find the code that implements this scaling and remove it so that you get actual speed control.

3. The `forward` and `backward` functions run the motors at different speeds using a linear scaling on the steering angle. We can get turning motion that skids less by improving this speed scaling. Use an ackerman steering approximation to write a function that defines the wheel speeds as a sinusoid function of steering angle, and modify the `forward` and `backward` functions to implement this relationship.

   Remember to check that the steering angle is consistent with the direction the servo is rotating. If not, you should account for this in your code.

4. To make future troubleshooting simpler, add a `logging.debug(<message>)` line in functions where a motor setting is changed with a useful message letting you know what exactly is being done on the robot.

2.8 Maneuvering

1. Write a set of functions (either in `picarx_improved.py` or in a separate file that imports `picarx_improved.py`) that move the car via discrete actions:

   (a) Forward and backward in straight lines or with different steering angles
   (b) Parallel-parking left and right
   (c) Three-point turning (K-turning) with initial turn to the left or right

   (If we had encoders on the wheels, we could explicitly set the amount by which to turn the wheels. These robots don't, so you'll need to approximate the desired distance to travel via speed and length of time for which you turn on the motors. You might consider measuring your car's speed at 100% commanded power for use as a reference).

2. Write a script that runs a while loop in which each iteration

   (a) Asks the user for keyboard input via the `input` function
   (b) Maps the keyboard input to one of the maneuver functions
   (c) Executes the selected maneuver

   (For good style, set up the input parsing so that there is an input that cleanly breaks the while loop)

Deliverables

1. Reflection questions:

   (a) What did you learn while completing the tasks in this lesson?
   (b) What other opportunities for improving the `picarx.py` code did you observe?
   (c) If you see any shortfalls with the approach we have taken above, please suggest improvements.

3 Sensors and Control

Now that we've got basic motor control handled, we can start using sensors to provide control inputs

The code in `adc.py` provides code for reading the ADC (analog-to-digital converter) pins on the breakout board, and `grayscale_module.py` provides an example of collecting data from the ground-scanning photosensors attached to the ADC pins. The `example/5.minecart_plus.py` script uses these values to control the steering angle of the car, but it is not particularly robust to material and lighting conditions.

In this lesson, we create Python classes to handle three aspects of the sensor-to-control process

1. Sensing the darkness of the ground below the robot
2. Interpreting data from the photosensors into a description of the current state of the robot
3. Controlling the steering angle based on the interpretation of the photosensors

3.1 Sensing

The sensor class should incorporate the following features:

1. The `__init__` method should set up the ADC structures as attributes using the `self.` syntax
2. Your sensor-reading method should poll the three ADC structures and put their outputs into a list that it returns

3.2 Interpretation

The interpreter class should incorporate the following features:

1. The `__init__` method should take in arguments (with default values) for both the sensitivity (how different "dark" and "light" readings are expected to be) and polarity (is the line the system is following darker or lighter than the surrounding floor?)
2. The main processing method take an input argument of the same format as the output of the sensor method. It should then identify if there is a sharp change between two adjacent sensor values (indicative of an edge), and then using the edge location and sign to determine both whether the system is to the left or right of being centered, and whether it is very off-center or only slightly off-center. Make this function robust to different lighting conditions, and with an option to have the "target" darker or lighter than the surrounding floor.
3. The output method should return the position of the robot relative to the line as a value on the interval $[-1, 1]$, with positive values being to the left of the robot.

3.3 Controller

The controller class should incorporate the following features:

1. The `__init__` method should take in an argument (with a default value) for the scaling factor between the interpreted offset from the line and the angle by which to steer.

2. The main control method should call the steering-servo method from your car class so that it turns the car toward the line. It should also return the commanded steering angle.

3.4 Sensor-control integration

Write a function that combines the sensor, interpreter, and controller functions in a loop so that the wheels automatically steer left or right as you move the car right and left over a dark line in the floor. You may need to adjust the sensitivity and polarity values in your interpreter function. Once automatic steering is working, add a "move forward" command to your script to make the car drive along the line with automatic steering. You may need to adjust the magnitude of the steering angle and the delay in your loop to make this motion smooth and robust.

3.5 Camera-based driving

Sensing the line via the three photocells is essentially using a three-pixel camera to look for the line to follow. The robots have a camera with a much higher resolution. Write sensor-interpreter-controller functions that use the camera to identify the line location and drive along it. There are multiple way to stream the camera view from the raspberry pi back to your computer, we will introduce three ways here, you can pick whatever you preferred or works best for you.

1. Set up VNC:

   (a) Allow VNC connections through the UFW firewall:

   ```
   sudo ufw allow VNC
   sudo systemctl restart ufw
   ```

   (b) Follow sunfounder's instructions to set up VNC at

   `https://docs.sunfounder.com/projects/picar-x-v20/en/latest/python/python_start/remote_desktop.html`

   Because all users are permitted to log in, ensure that users have strong passwords. You should still be logging into the Raspberry Pi with your own user account.

2. Web-Based Streaming:

   Refer to the Example 7 in the PiCar-X library, which uses the `vilib` library from SunFounder, to start a Flask application for streaming the camera view.

   For a simpler reference on streaming camera frames in a web browser using Flask, see:

   `https://blog.miguelgrinberg.com/post/video-streaming-with-flask`

   Run the stream code through SSH tunnel using the following command:

   ```
   ssh -L remote_port:localhost:local_port username@rpi_ip
   ```

   where `remote_port` is the port used for streaming in Flask, `local_port` is the local port you wish to use.

   Once the SSH tunnel is established, access the camera stream by browsing to: `localhost:local_port`.

3. Forwarding X11:

   (Forwarding X11 is more for visualizing pictures than real-time camera views. While it is still a method you should know, there will be significant latency when using it to stream camera views)

   Follow the instructions at link below to forward X11 to your local machine

   ```
   https://www.raspberrypi.com/documentation/computers/remote-access.
   html#forward-x11-over-ssh
   ```

   If your system doesn't run an X server, you may need to download and run tools like `MobaXterm` (Windows) or `XQuartz` (macOS) If you encounter a "cannot find the display" error, manually set up the `DISPLAY` variable:

   ```
   export DISPLAY=localhost:10.0
   ```

   After this setup, you should be able to visualize the camera view directly using OpenCV, similar to how you would with your laptop's webcam.

At this point, you should be able to connect to the Raspberry Pi using VNC or stream the camera frames through the web. Here are some additional materials on camera-based line following:

1. Examples 8–12 in the PiCar-X library and vilib from SunFounder demonstrate how to implement camera-based driving.
2. The picamera2 documentation

   ```
   https://datasheets.raspberrypi.com/camera/picamera2-manual.pdf
   ```

   has the logic needed for speaking with the camera.
3. A good example on lane-following with OpenCV is at

   ```
   https://const-toporov.medium.com/line-following-robot-with-opencv-
   and-contour-based-approach-417b90f2c298
   ```
4. Note that OpenCV uses BGR order for color images by default. If your picamera2 reads it in RGB order, you can use `cv2.cvtColor()` to convert between them.

Deliverables

1. Code review from "Car week 2" partner
2. Code review of "Car week 2" partner's work
3. Reflection questions:
   (a) What did you learn while completing the tasks in this lesson?
   (b) If we spent more time "hardening" the control loop in this lesson, what feature or capability would you add?
   (c) Why does the use of VNC add a security vulnerability? How would you fix or improve the issue?

4 Simultaneity

It is often useful to decouple the various operations that your robot performs, so that sensing, interpretation, and control run independently and in parallel. This notion of *concurrency* can be implemented via

- Multitasking, in which a single processor rapidly switches between the different functions it is executing, and
- Multiprocessing, in which the functions are assigned to dedicated processors.

Multitasking can be further characterized as being *cooperative* or *pre-emptive*, respectively based on whether the time-sharing of the processor is determined by the functions themselves, or is allocated externally by the computer's operating system. Pre-emptive multitasking is also called *threading*. More information on these concepts can be found at

`https://realpython.com/python-concurrency/`

For pre-emptive multitasking and multiprocessing, there is a risk that multiple threads and processes will attempt to operate on the same data structures at the same time. At a low level, consequences can include a task attempting to read a data from a location in the middle of another task writing to the location, and getting a mix of the bits in old values and new values, or two tasks attempting to write data to the same location at the same time, and ending up storing a mix of bits corresponding to the two values.

At a higher level, interleaving the execution of tasks can cause other problems even if bit-mixing is avoided. For example, the operation `x=x+1` in many languages does not mean "increment the value in `x` by `1`" but instead means "read the value in `x`, add `1` to it, then write the resulting value into `x`". If two threads or processes attempt to execute this operation at the same time, the sub-operations may become interleaved such that the second thread reads the value in `x` before the first has incremented it. In this case both threads will write out a value to `x` that is `1` more than its *original* value, and an increment will be lost.

These difficulties with concurrency can be addressed by

1. Designing the system to minimize the opportunity for conflicts (e.g., by avoiding situations in which two operations write to the same location) and to be robust in the case that one such error does occur. This tends to be easier when the information being passed through the system is an approximation of an analog signal rather than an encoded text. (See also "Gray code" for an example of handling concurrency problems at a hardware level).

2. Ensuring that operations are *atomic* (not interruptible). Different programming languages handle atomicity at different levels. Some languages guarantee that reading and writing simple data types such as integers and floats is atomic, so that bit-mixing is not a worry, but do not guarantee that writing values to an array will not be interrupted partway through by a read operation. Python goes further, making read and write operations to many array types (including lists) atomic.

   Atomicity for higher-level operations generally needs to be specified by the programmer (to ensure that it is applied to operations for which interruption would be problematic are made atomic, while allowing other operations to be interruptible to keep the program flowing).

Setting up the *locks* that ensure atomicity can quickly become complicated. If this were a CS course, we'd stop here and spend a good chunk of time going over various approaches to locking. Many of these approaches, however, are related to passing along discrete messages. By focusing our attention on tasks that broadcast their most-recent estimates of system states on dedicated channels, we can sidestep most of this extra complexity and leave it on a need-to-know basis.

A second effect to be careful of in pre-emptive multi-tasking is when the task scheduler interrupts a set of nominally simultaneous interactions with an external system. For example, if a program polls several sensors in succession to get a snapshot of the external world at a given time, the sensor measurements will be slightly offset in time from each other, which can cause "rolling shutter" distortion. This distortion will be amplified if the task scheduler switches to a different task in the middle of cycling through the sensors, increasing the delay between sensor readings. *Cooperative multitasking* can help to mitigate these effects, by having the programmer specify places where it is safe for threads to switch out (at the cost of requiring the programmer to think about and specify these points. Cooperative multitasking is not directly covered in this lesson (but example code of how it could have been used here will be made available in the following week).

For this lesson, you will assign the sensor, interpretation, and steering-control functions to independent threads of the program, so that they can run at independent rates. In doing so, we will implement *consumer-producer* functions passing *messages* via *busses*.

4.1 Busses

When we assign functions to individual threads, we generally need to provide a means for them to pass information between threads, e.g., so that the interpreter can see the most recently reported reading from the sensors, and the controller can see the most recent interpretation of the system state.

A basic means of passing messages between threads or processes is to create "message busses" that processes can read from or write to. The messages on these busses can be "state updates" that act as "one-way broadcasts", in that reading the message does not remove it from the bus, or "queued", in that messages are "cleared" as they are read.

For this class, we will use "broadcast" messages. Conceptually, this is similar to how global variables can be used to pass information between functions, but with a bit more structure

---

Define a simple Python class to serve as your bus structure. It should have:

1. A "message" attribute to store values
2. A "write" method that sets the message
3. A "read" method that returns the message

4.2 Consumer-producers

The core elements of a robot control program are operations that carry out tasks such as polling sensors, interpreting data, and controlling motors. When these processes exchange data via busses, we can think of them as

- Producers, writing newly-created data to busses;
- Consumer-producers, reading information from some busses, processing that data, and writing the output to other busses; and
- Consumers, reading information from busses and acting on it without publishing data to busses.

These categories closely correspond to our previous notions of sensor, interpretation, and control functions.

Define producer, consumer-producer, or consumer functions for the sensor, interpretation and control processes in the previous lesson.

1. Each consumer/consumer-producer/producer should be defined as a function that takes instances of your bus class and a delay time as arguments
2. Each consumer/consumer-producer/producer function should contain a while loop
3. The sensor, interpretation, or control function should be executed inside the loop function, with data read from or written to bus classes as appropriate
4. The loop should `sleep` by the delay amount once each cycle

4.3 Concurrent execution

Once the busses and consumer-producers have been created, we can have the consumer-producer functions execute concurrently. Some basic steps to get concurrent execution up and running are:

1. Import the `concurrent.futures` module with:

   ```
   import concurrent.futures
   ```

   and tell the system to run your system components concurrently. For example, to run your `sensor_function` with inputs of the `sensor_values_bus` to write data to and the `sensor_delay` to set the polling rate together with your `interpreter_function`, you can run

   ```
   with concurrent.futures.ThreadPoolExecutor(max_workers=2) as executor:
    eSensor = executor.submit(sensor_function, sensor_values_bus, sensor_delay)
    eInterpreter = executor.submit(interpreter_function, sensor_values_bus, interpreter_bus,i
   ```

   You will place a similar block of code setting up the three sensor, interpreter, and control threads where you originally ran the full line-following functionality. Make sure you allocate enough workers.

   Note that cancelling execution of a program running under `concurrent.futures` by `ctrl-C` is tricky and the `atexit()` function may not be called. Additionally, any exceptions that occur in the child thread may not be reported back to the main thread and thus will not be printed in the terminal. Here is an example of how to terminate a multithreaded script gracefully and report any exceptions:

```
from time import sleep
from concurrent.futures import ThreadPoolExecutor
from threading import Event

# Define shutdown event
shutdown_event = Event()

# Exception handle function
def handle_exception(future):
    exception = future.exception()
    if exception:
        print(f'Exception in worker thread: {exception}')

# Define robot task
def robot_task(i):
    print('Starting robot task', i)
    while not shutdown_event.is_set():
        # Run some robot task...
        print('Running robot task', i)
        sleep(1)
    # Print shut down message
    print('Shutting down robot task', i)
    # Test exception
    if i == 1:
        raise Exception('Robot task 1 raised an exception')

if __name__ == '__main__':
    futures = []
    with ThreadPoolExecutor(max_workers=3) as executor:
        for i in range(3):
            # Spawn task threads
            future = executor.submit(robot_task, i)
            # Add exception call back
            future.add_done_callback(handle_exception)
            futures.append(future)

        try:
            # Keep the main thread running to response for the kill signal
            while not shutdown_event.is_set():
                sleep(1)
        except KeyboardInterrupt:
            # Trigger the shutdown event when receive the kill signal
            print('Shutting down')
            shutdown_event.set()
        finally:
```

```
# Ensures all threads finish
executor.shutdown()
```

Set up concurrent futures for each of your consumer-producer functions.

2. We want to ensure that writing and reading bus messages is properly locked. The suggested locking code here is an "writer-priority read-write lock", which means that:

   (a) Only one thread is allowed to write the bus message at any given time, and may not start writing while any other thread is reading the message

   (b) Any number of threads may read the bus message at any given time, but may not start reading while a thread is writing the message

   (c) Write operations are given priority over read operations

   A physical analogy of this situation is a whiteboard at the front of a room, with instructors taking turns writing information on the board, and students able to read the information on the board whenever an instructor is not blocking their view. Writer-priority means that the students may have to wait for information, but it will always be the most up-to-date information (see `https://en.wikipedia.org/wiki/Readers-writer_lock` for more details).

Set up locking for your busses.

(a) Import and implement concurrent futures

(b) Install the `readerwriterlock` Package The SunFounder installation settings may break the Python package management. You might need to use the `--break-system-packages` parameter with `pip` to install `readerwriterlock`:

```
pip install --break-system-packages readerwriterlock
```

This will install `readerwriterlock` to the user directory.

If you need to call Python scripts with `sudo`, use the `-E` flag to preserve the environment settings so that the package installed in the user directory remains accessible:

```
sudo -E python your_script.py
```

(c) Put

```
from readerwriterlock import rwlock
```

in your header

(d) Put

```
self.lock = rwlock.RWLockWriteD()
```

inside your `__init__` method for the `bus` class

(e) Use

```
with self.lock.gen_wlock():
    self.message = message
```

inside your "write" method, and

```
with self.lock.gen_rlock():
    message = self.message
```

inside your "read" method.

3. Note that if your child thread runs at a very high frequency to read the sensors and control the robot, there is a chance that you will get some garbage data (such as 0 or very large numbers from the grayscale sensor). This is because both sensor reading and servo/motor control go through the same I2C bus, which runs in half-duplex mode. An easy way to solve this problem is to limit the frequency of the threads. Alternatively, adding read and write locks for reading the sensors and writing the steering angle and forward speed from the picarx instance can also solve this problem.

Deliverables

1. Code review from "Car week 3" partner
2. Code review of "Car week 3" partner's work
3. Reflection questions:
   (a) What did you learn while completing the tasks in this lesson?
   (b) With the architecture that you've created, what is the next feature that you would add to the robot control stack?

5 Multimodal control

The threaded architecture in the previous lesson lets us execute multiple processes concurrently. The assignment, however, only really had a single task flow, and as such didn't really need to be threaded. In this lesson, we will set up a system that uses threads to separately poll two different sensors and combine their information for the control system.

5.1 RossROS

Fetch RossROS from

`https://www.rossros.org/`

`RossROS.py` contains bus and consumer-producer classes (reference implementations of the structures discussed in the last lesson).

1. Reimplement your concurrent line-follower script using the `RossROS.py` classes
2. As you do so, use the `Timer` class from `RossROS.py` to control the run-time of the script. `Timer` instances are Producers that change their output bus value to "true" after a set amount of time; if you set the output bus to the termination bus that it and the other Consumer-Producers are reading, all of the threads will shut down when the `Timer` reaches the set time.

5.2 Concurrent control

The consumer-producer/busses framework allows us to run multiple control loops at the same time. For example, we can have the car use the photosensors to follow a line, while having the ultrasonic sensors stop the robot if it comes too close to an obstacle on the line.

1. Create sensor and interpreter classes for the ultrasonic sensors, along with a controller that moves the car forward if the way forward is clear, and stops it if there is an obstacle immediately in front of it

   Please note that the ultrasonic sensor returns the distance in centimeters. It may also return -1 or -2 as an error code. You can refer to `robot_hat/modules.Ultrasonic._read()` for more details.
2. Wrap the ultrasonic sensor, interpreter, and controller into `RossROS` consumer-producers
3. Add the ultrasonic-based driving control to the `ThreadPoolExecutor` execution list

Deliverables

1. Code review from "Car week 4" partner
2. Code review of "Car week 4" partner's work
3. Reflection questions:
   (a) What did you learn while completing the tasks in this lesson?
   (b) With the architecture that you've created, what is the next feature that you would add to the robot control stack?

Part II
Fixed-base manipulators

0 Set up and secure your Raspberry Pi

Much of the arm setup will be identical to setting up the Picar. We'll link back to those steps for setting up the arm, and point out where things are slightly different.

The SD card that came with your arm should be pre-imaged with the appropriate software. You will need to modify `wpa_supplicant.conf` to add your home WiFi credentials. You can do this either by supplying a new supplicant file to the SD card (allowing you to initialize on your home network), or you can SSH in using the robotics network WiFi and modify the supplicant file in place.

Note that the Arm runs an older version of Debian than the Picar, so the process for adding wifi is different, you can refer to the following link for the process:

`https://www.raspberrypi-spy.co.uk/2017/04/manually-setting-up-pi-wifi-using-wpa_supplicant-conf/`

0.1 If you have a camera mounted on a separate support

1. Install the SD card into the robot.
2. Arrange the arm, localization mat, and camera support, and provide power on the arm.
3. SSH into your arm using

   ```
   ssh pi@<yourMusicianName>.engr.oregonstate.edu
   password: mytaisrad
   ```

   Once you're SSH'd in to the Pi, you will see RobertJohnson as the localhost, regardless of the name you used to SSH in. This doesn't matter, but you can change the internal localhost through raspi-config if it bothers you.
4. Modify `/etc/wpa_supplicant/wpa_supplicant.conf` to add your home WiFi network credentials.
5. Using the documentation at

   `https://www.raspberrypi.com/documentation/computers/configuration.html#change-user-password`

   to change the password for your Pi.
6. Follow the picar setup instructions starting at Step 5 to set up the arm. When setting up the firewall, turn on the `ufw` and set the options `sudo ufw allow ssh`, `sudo ufw limit ssh/tcp`, and `sudo ufw allow 5900` (which enables VNC access).
7. Configure a place for your arm code on GitHub. You can use the same RobotSystems repo and two new folders to separate arm code from car code, or you can make a new repo just for your arm. Generate an SSH keypair used to connect your arm to your github as you did in Picar setup step 3.

1 Basic arm operations

Here are some materials from the manufacturer to help you get started with your arm:

> `https://drive.google.com/drive/folders/1VFbx3n0GjP46kT-yj92-lW7LcfXnrcEs?usp=sharing`

You can start from `ArmPi/2.Quick User Experience/Lesson 1 Position Calibration` for calibrating the camera. Because the SD card and SSH are already set up, you can then start from `ArmPi/3.AI Vision Games Lesson` for running the code. For most of the code that comes with Arm you will need to use VNC to run it.

The arm comes with some basic programs for identifying colored blocks, picking them up and placing them in designated locations, and stacking them on top of each other. Run these programs to get familiar with their general operation. You may need to edit the position value at which the gripper servo is considered "closed".

The sample code with the arms is written as a large monolithic loop with no abstraction. Assignments for the remainder of the term will be about refactoring the pick-and-place code into a better format.

Deliverables

1. Code review from "Car week 5" partner
2. Code review of "Car week 5" partner's work
3. Reflection questions:
   (a) What are your initial thoughts on the provided code?

2 Perception

The pick-and-place code has two key components: A perception component that identifies and locates blocks in the target area, and a motion component that sends commands to the servos.

This week, your mission is to refactor the perception code to make it readable with useful abstractions. As a first step:

1. Identify where in `ColorTracking.py` the perception code is located
2. Make a copy of the code and add comments to it. (Google Translate may be helpful here)
3. Make a flow chart of high-level tasks that the perception code accomplishes
4. Write a Python class with methods corresponding to the high-level tasks you identified.
5. Set up a simple program that uses this class to identify the location of a block in the pickup area and labels it on the video display from the camera.

Once you have finished the above tasks:

1. Identify any additional functionality that appears in the perception code from `ColorSorting.py` and `ColorPalletizing.py`. You may find a code comparison tool (such as the difference viewer in PyCharm) to be useful here.
2. Add any methods necessary to implement this functionality into your perception class.
3. Set up a simple program that demonstrates this added functionality.

Deliverables

1. Flow chart showing steps in perception process (including the extra steps in the `ColorSorting.py` code)
2. Reflection questions:
   (a) What did you learn while completing the tasks in this lesson?

3 Motion

The pick-and-place code has two key components: A perception component that identifies and locates blocks in the target area, and a motion component that sends commands to the servos.

This week, your mission is to refactor the motion code to make it readable with useful abstractions. As a first step:

1. Identify where in `ColorTracking.py` the motion code is located
2. Make a copy of the code and add comments to it. (Google Translate may be helpful here)
3. Make a flow chart of high-level tasks that the motion code accomplishes
4. Write a Python class with methods corresponding to the high-level tasks you identified.
5. Combine this motion class with your perception class to perform a basic pick-and-place operation.

Once you have finished the above tasks:

1. Identify any additional functionality that appears in the motion code from `ColorSorting.py` and `ColorPalletizing.py`. As in the perception assignment, you may find a code comparison tool (such as the difference viewer in PyCharm) to be useful here.
2. Add any methods necessary to implement this functionality into your motion class.
3. Use your motion class with your perception class to implement block sorting and block stacking.

Deliverables

1. Code review from "Arm week 2" partner
2. Code review of "Arm week 2" partner's work
3. Flow chart showing steps in motion process (including the extra steps in the `ColorSorting.py` code)
4. Reflection questions:
   (a) What did you learn while completing the tasks in this lesson?
   (b) The final two weeks of the course will be a project with the arm. Propose an idea for something "interesting" to do with the arm.

4 Final Project

For the two weeks, do something "interesting" with the arm that goes beyond the previous lessons. "Interesting" here is open to wide interpretation, and could range from getting interesting behavior out of the arm (e.g., writing with a pen or chalk, or attaching the camera to the arm and implementing visual servoing) to taking a deeper dive into the lower-level arm code (e.g., reworking the arm kinematics implementations). You may form groups of two or three students for the projects.

Deliverables, Week 9

1. Code review from "Arm week 3" partner
2. Code review of "Arm week 3" partner's work
3. Progress update for your project:
   (a) Names of group members
   (b) One paragraph description of project concept
   (c) Flow chart or other high-level description of how your project code is expected to work
   (d) Description of where you are in the design/implementation process
4. Reflection questions:
   (a) What did you learn this week?

Deliverables, Week 10

1. Project report:
   (a) Description of your project
   (b) Flow chart or other high-level description of how your project code works
   (c) Video of your robot operating with your code. (If you do a lower-level code dive, you may want to include something else here to show off what you did)
2. Reflection questions:
   (a) What did you learn this week?

Deliverables, Finals Week

1. Short (< 5 minute) video presentation about your project. I'll make the videos viewable in a shared folder, and we'll have a watch session with time for questions during the "Final exam" time for the class.