Research Robotics

April 8, 2021

Part I

Mobile systems

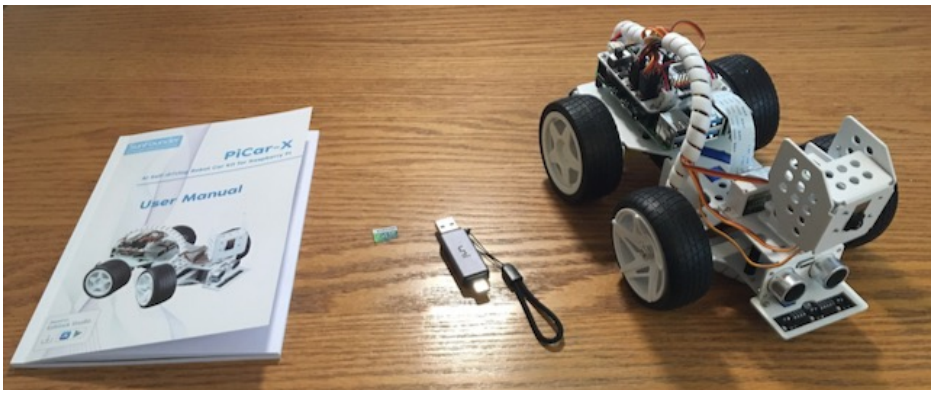
0 Build PiCar-X

Charge your batteries and download the updated image for your Raspberry Pi before starting. Both of these processes may take a while.

Follow the instructions in the PiCar-X booklet to build the robot car. Note that the first step in the instructions is to prepare the SD card with SunFounder's version of the Raspberry Pi operating system. You'll need to use the SD card about half-way through the mechanical build process (to make sure the steering and camera servos are centered before you attach them to the system).

Note that during the power-on process, the speaker on the Raspberry Pi will buzz for about a second. Do not be alarmed.

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. Extract the SD card from the Raspberry Pi and put it back in your SD card reader
2. Follow the instructions at

   `https://www.raspberrypi.org/documentation/configuration/wireless/headless.md`

   to enable the WiFi interface on the Raspberry Pi. Once you have created and populated the `wpa_supplicant.conf` file, put the SD card back into the Raspberry Pi.

   NON PICAR IMAGES MAY NEED SSH ENABLED
3. Use

   ```
   ssh pi@raspberrypi.local
   ```

   to connect to the Raspberry Pi, with password `raspberry`. 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 `wpa_supplicant.conf` creation process, double-checking your network name and password.

   (b) If fixing the `wpa_supplicant.conf` file doesn't get the Raspberry Pi onto the WiFi network and you have an ethernet cable available, you can plug the Raspberry Pi directly into the network, and then use the `raspi-config` tool (see below) to configure the wireless network.

   (c) If you can connect, but the password is rejected, someone else on your network may have a Raspberry Pi set up for which they changed the password but left the name at its default value. 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).

   (d) 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.

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

   ```
   sudo raspi-config
   ```

and then `System Options > Hostname`.

(f) 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. Carry out some basic setup operations for securing your Raspberry Pi. **Read through the notes listed below**, then follow the steps outlined at

   `https://www.raspberrypi.org/documentation/configuration/security.md`

   but **do not delete the user 'pi'**. 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)

   (a) Make sure to follow the instructions at

   `https://www.raspberrypi.org/documentation/raspbian/updating.md`

   (which are linked from the security-setup instructions above) for updating the system code. You are already starting with a "Buster" install, so you can ignore the section about upgrading from "Stretch". Don't worry about the "Third-party solutions" material for this class (though it is worth knowing about it if you start operating robots that you cannot physically access).

   (b) Make sure to set up your SSH installation to automatically update itself by following the instructions at

   `https://www.raspberrypi.org/documentation/linux/usage/cron.md`

   (also linked from the main security-setup instructions). You'll be rebooting this robot often, so it's probably easiest to put

   ```
   @reboot sudo apt install openssh-server
   ```

   as your line in the `crontab` file.

   (c) For the allow/deny options, put in an "Allow" for the new username you created, and a "Deny" for the 'pi' user.

   (d) If you are on a Windows machine, use

   `https://www.ssh.com/ssh/putty/windows/puttygen`

   to generate your SSH keys, and then follow the instructions at

   `https://pimylifeup.com/raspberry-pi-ssh-keys/`

   (e) When disabling password-based authentication (after setting up SSH keys), note that `PasswordAuthentication` needs to be uncommented as well as changed from `yes` to `no`.

(f) Turn on the `ufw` firewall, and set the options `sudo ufw allow ssh` and `sudo ufw limit ssh/tcp`.

(g) Install `fail2ban` as per the instructions. Skip the part about modifying the `jail.local` file, which appears to be slightly out of date – the jail for sshd is already configured and active. If you expose your robot to more public parts of the internet, you may want to learn more about `fail2ban` settings.

(h) Remove the stored plaintext of your WiFi password by following the instructions at

`https://carmalou.com/how-to/2017/08/16/how-to-generate-passcode-for-raspberry-pi.html`

This step isn't critical. Some discussion on why it's worth doing is at

`https://superuser.com/questions/1411604/what-security-does-the-wpa-passphrase-tool-actually-add-to-a-wpa-supplicant-conf`

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

(a) 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.

(b) Create a new Git repository in your account named `RobotSystems`

(c) Set up SSH keys on your GitHub account.

i. Go to the Settings page for your account (Use the dropdown menu in the top right corner of the page; Settings is near the bottom of the menu.)

ii. Go to "SSH and GPG keys" in the list of settings panes at the left of the screen

iii. Open the "guide to generating SSH keys" link **in a new tab**.

iv. SSH into your Raspberry Pi, and then follow the instructions to create an SSH key **on your Raspberry Pi** and add that key to your GitHub account. 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).

v. Set a non-empty passphrase for your SSH key. 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.
If you follow the link for "Working with SSH key passphrases", you can follow the instructions under "Auto-launching ssh-agent on Git for Windows" to modify the `.profile` file on your Raspberry Pi so that you will be asked for this passphrase each time you boot the Raspberry Pi, and won't be asked for it again when you perform Git operations. If you are running `zsh`, there are various solutions online, e.g.,

`https://www.vinc17.net/unix/index.en.html#zsh-ssh-utils`

that offer more elegant handling of SSH keys, such as only prompting you for a passphrase when you actually use the keys instead of on startup.

vi. On the "Adding a new SSH key to your GitHub account" page, the fact that you are SSH'd into the computer on which you are generating keys makes Step 1 (copying the ssh key into your clipboard) a bit trickier than they otherwise would be. The easiest way to make this work will probably be to run

```
nano .ssh/id_ed25519.pub
```

and then copy the contents of the file out of the terminal.

vii. If you left the original Settings page open in another tab, you can then follow the remaining steps on the "Adding a new SSH key to your GitHub account" page.

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. For PiCar-X robots, this information is in two places:

- the `ezblock` Python package, which provides low-level control over the I/O pins
- `/opt/ezblock/picarx.py`, which provides commands for operations like setting the forward speed of the car and the angles of the steering and camera servos.

In this lesson, we will

1. Create a copy of `picarx.py` with modifications to enable offline testing and improve the system performance
2. Create a "shadow" version of the `ezblock` package to enable offline testing
3. Write some basic maneuver functions to demonstrate control over the motors

2.1 Gather code pieces

1. On your Raspberry Pi, follow the instructions from page 46 of the PiCar-X manual to download the software for the car. (Do this in your own account, not the 'pi' account).
2. Note that there is a typo in the last instruction on the page, and that you want to enter

   ```
   sudo service ezblock-reset stop
   ```

   to permanently turn off the ezblock service (typo is that the manual says "rest" instead of "reset").

2.2 Improve ezblock printing

(credit to Robert Brown for these suggestions)
The `ezblock` code does a few potentially-annoying things with printing. To fix them:

1. Lines 12-19 of

   ```
   /usr/local/lib/python3.7/dist-packages/ezblock-0.0.3-py3.7.egg/
      ezblock/utils.py
   ```

   the python print function is modified to include logging, which can significantly slow down Python's print function. Commenting out these lines will speed up performance. (Removing the function does not work because it brakes some dependencies). An alternative option is to keep this new print function as is, then use `__PRINT__()` to print without logging. See also the suggested logging framework described later in this document.
2. Line 110 of

   ```
   /usr/local/lib/python3.7/dist-packages/ezblock-0.0.3-py3.7.egg/
      ezblock/pwm.py
   ```

   prints out the PWM value each time it is set. This line can be safely commented out, with any logging/display commands that you actually want placed in your higher-level code.

2.3 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.

2.4 Copy necessary files into your coding environment

1. Make sure you have a directory on your robot that is linked to a Git repository
2. Copy the `/opt/ezblock/picarx.py` file into your Git-linked directory
3. Copy the `picar-x` directory into your Git-linked directory
4. Find the `ezblock` Python directory and copy it into your Git-linked directory. A good way to find its location is to start the Python interpreter and then run

   ```
   import ezblock
   print(ezblock.__file__)
   ```

5. Sync all of these files to your Git repository.
6. Use `sudo shutdown now` to turn off your car and save your batteries.
7. Clone your Git repository onto your desktop computer.

2.5 Create basic infrastructure

These steps can be carried out on your desktop computer

1. Copy `picarx.py` into a new file `picarx_improved.py`
2. Create an empty Python script named `sim_ezblock`
3. Replace the opening line `from ezblock import *` in `picarx_improved.py`

   ```
   import time
   try:
       from ezblock import *
       __reset_mcu__()
       time.sleep(0.01)
   except ImportError:
       print("This computer does not appear to be a PiCar-X system
          (/opt/ezblock is not present). Shadowing hardware calls
          with substitute functions")
       from sim_ezblock import *
   ```

4. Open a Python shell, and enter `import picarx_improved.py`. This should return an error along the lines of

```
name 'Servo' is not defined
```

5. Find the code in your copy of the `ezblock` package that implements the missing class, then define a class in your `sim_ezblock` file that has the same structure as the `ezblock` class. Methods that don't return an output can be implemented as a simple `pass`, other methods should return an output of the appropriate type.

6. Repeat the above steps until you have provisioned `sim_ezblock` with all of the classes that `picarx_improved.py` is expecting to see.

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. Place

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

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

2. 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 completing")
   @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.

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` function runs both motors at the same speed. If the front wheels are at an angle, the two rear wheels should spin at different speeds. Identify the (mathematical) function that defines the difference in speeds as a function of steering angle, and modify the `forward` function to implement this relationship.

4. Calibrate your steering: Write a function that imports `picarx_improved.py`, then commands the system to drive forward at zero steering angle for a short time. Based on how far left or right the car pulls over this motion, modify the steering calibration angle in `picarx_improved.py`.

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).

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)

2.9 Organize your code

The provided `picarx.py` defines the motor control and sensing operations as individual functions, with values such as the calibration angle passed into the functions by declaring them as global variables. It is good in general to avoid global variables. Start a new file that re-implements the motor commands as Python class methods.

1. The `__init__` method should set up the configuration constants and Servo/PWM/Pin constants as `self.` values.

2. Only implement the motor functions in this Python class. Future lessons will construct classes for the sensors and the control logic.

3. Instead of putting a call to `atexit` in the file, put a line

   ```
   atexit.register(self.cleanup)
   ```

   in the `__init__` method for the class, and then define a class method named `cleanup` that zeros out the motor speeds.

2.10 Deliverables

1. Code review from this week's downstream partner
2. Code review of this week's upstream partner's work
3. 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?

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 `picarx.py` provides code for reading the ADC (analog-to-digital converter) pins on the breakout board, and `3.gray_scale.py` provides an example of collecting data from the ground-scanning photosensors attached to the ADC pins. The `9.mine_cart.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 which 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 in the 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.

1. The example script `4.color_detection.py` has the commands needed for speaking with the camera. Comment out line 10 of this code, because the WiFi connection is already set up, and not trying to set it up again avoids needing `sudo` privileges. To connect to the video stream from a web browser, we need to open up port 9000 in the firewall, by entering

   ```
   sudo ufw allow 9000
   ```

   on the command line. You can connect to the stream either at the IP address of your Raspberry Pi (as described in the PiCar-X manual), or by replacing the IP address with `hostname.local` (where hostname is the name you gave your robot when you set it up).

2. A good tutorial on lane-following with OpenCV is at

   `https://towardsdatascience.com/deeppicar-part-4-lane-following-via-opencv-737dd9e47c96`

3.6 Deliverables

1. Code review from this week's downstream partner
2. Code review of this week's upstream 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?

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.

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.

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 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.

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 topics;
- Consumer-producers, reading information from some topics, processing that data, and writing the output to other topics; and

- Consumers, reading information from topics and acting on it without publishing data to topics.

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-producer should be defined as a function that takes instances of your topic class and a delay time as arguments
2. Each consumer-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 topic 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
   ```

2. 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,interpreter_delay)
   ```

   Note that to cancel execution of a program running under `concurrent.futures` you will need to hit `ctrl-C` multiple times. Additionally, any errors thrown by the functions will not be displayed in the terminal; this problem can be worked around by adding a line

   ```
   eSensor.result()
   ```

   outside of the `with` block.

3. Make sure that polling the sensors is not interrupted:

   (a) Put

```
from threading import Lock
```

in your header

(b) Put

```
lock = Lock()
```

inside your sensor consumer-producer before the while loop

(c) Use

```
with lock
        sensor_poll_1
        sensor_poll_2
        sensor_poll_3
```

to prevent the thread scheduler from switching tasks after polling sensor 1 and before polling sensor 3

4.4 Deliverables

1. Code review from this week's downstream partner
2. Code review of this week's upstream 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

The course documents include a Python file `RossROS.py` that contains topic and node 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 `nodeTimer` class from `RossROS.py` to control the run-time of the script

5.2 Concurrent control

The nodes-and-topics 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
2. Wrap the ultrasonic sensor, interpreter, and controller into `RossROS` nodes
3. Add the ultrasonic-based driving control to the `ThreadPoolExecutor` execution list

5.3 Deliverables

1. Code review from this week's downstream partner
2. Code review of this week's upstream 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

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. Arrange the arm, localization mat, and camera station.
2. On your computer, download and install VNC Viewer from

   `https://www.realvnc.com/en/connect/download/viewer/`
3. Connect your computer to the HiWonder wifi network provided by the arm. (During this step, you will be temporarily not able to connect to the internet).
4. Use VNC viewer to log into the arm, with hostname `raspberrypi.local`, username `pi` and password `raspberry`.
5. Remove the Chinese localization from the system:
   (a) Edit the locale file by entering

   ```
   sudo nano /etc/default/locale
   ```

   at a terminal command line and changing it to read

   ```
   LANG=en_US.UTF-8
   LC_ALL=en_US.UTF-8
   LANGUAGE=en_US.UTF-8
   ```

   (b) Fix the keyboard layout by modifying `/etc/default/keyboard` so that `XBLAYOUT="US"`
   (c) Right-click on the keyboard symbol in the top-right of the system menu bar, select "Configure", and then replace the Chinese input methods with an appropriate English keyboard.
   (d) Modify `/etc/wpa_supplicant/wpa_supplicant.conf` so that the country is "US"
6. Change the name of your Raspberry Pi by using

   ```
   sudo raspi-config
   ```

   and then `System Options > Hostname` (In some cases, this may be listed under "network options").
7. Carry out some basic setup operations for securing your Raspberry Pi. When copying passwords from your password manager into a terminal running inside VNC, you should be able to right-click in the terminal window to paste from your computer's buffer.
   (a) Start with the steps outlined at

   `https://www.raspberrypi.org/documentation/configuration/security.md`

   but **do not delete the user 'pi'**. 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) After creating your username and using `sudo su - username` to switch to your user, use `sudo raspi-config` and then use

```
Boot Options > Desktop / CLI > Desktop Autologin Desktop
   GUI, require user to log in
```

I recommend having your password generator produce a "memorable" password for your account rather than a "fully random" password, because it is not possible to paste a password into the login dialog. You could also choose to have the system automatically log you in under your username, but this reduces the security of your installation.

(c) Make sure to follow the instructions at

`https://www.raspberrypi.org/documentation/raspbian/updating.md`

(which are linked from the security-setup instructions above) for updating the system code. You are already starting with a "Buster" install, so you can ignore the section about upgrading from "Stretch". Don't worry about the "Third-party solutions" material for this class (though it is worth knowing about it if you start operating robots that you cannot physically access).

(d) Turn on SSH for your system in `raspi-config > Interface Options`.

(e) Make sure to set up your SSH installation to automatically update itself by following the instructions at

`https://www.raspberrypi.org/documentation/linux/usage/cron.md`

(also linked from the main security-setup instructions). You'll be rebooting this robot often, so it's probably easiest to put

```
reboot sudo apt install openssh-server
```

as your line in the `crontab` file.

(f) For the allow/deny options, put in an "Allow" for the new username you created, and a "Deny" for the 'pi' user.

(g) If you are on a Windows machine, use

`https://www.ssh.com/ssh/putty/windows/puttygen`

to generate your SSH keys.

(h) When disabling password-based authentication (after setting up SSH keys), note that `PasswordAuthentication` needs to be uncommented as well as changed from `yes` to `no`.

(i) Turn on the `ufw` firewall, and set the options `sudo ufw allow ssh`, `sudo ufw limit ssh/tcp`, and `sudo ufw allow 5900` (which enables VNC access).

(j) Install `fail2ban` as per the instructions. Skip the part about modifying the `jail.local` file, which appears to be slightly out of date – the jail for sshd is already configured and active. If you expose your robot to more public parts of the internet, you may want to learn more about `fail2ban` settings.

(k) Remove the stored plaintext of your WiFi password by following the instructions at

`https://carmalou.com/how-to/2017/08/16/how-to-generate-passcode-for-raspberry-pi.html`

, and then co This step isn't critical. Some discussion on why it's worth doing is at

`https://superuser.com/questions/1411604/what-security-does-the-wpa-passphrase-tool-actually-add-to-a-wpa-supplicant-conf`

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

(a) 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.

(b) Create a new Git repository in your account named `RobotSystems`

(c) Set up SSH keys on your GitHub account.

i. Go to the Settings page for your account (Use the dropdown menu in the top right corner of the page; Settings is near the bottom of the menu.)

ii. Go to "SSH and GPG keys" in the list of settings panes at the left of the screen

iii. Open the "guide to generating SSH keys" link **in a new tab**.

iv. SSH into your Raspberry Pi, and then follow the instructions to create an SSH key **on your Raspberry Pi** and add that key to your GitHub account. 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).

v. Set a non-empty passphrase for your SSH key. 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.
If you follow the link for "Working with SSH key passphrases", you can follow the instructions under "Auto-launching ssh-agent on Git for Windows" to modify the `.profile` file on your Raspberry Pi so that you will be asked for this passphrase each time you boot the Raspberry Pi, and won't be asked for it again when you perform Git operations. If you are running `zsh`, there are various solutions online, e.g.,

`https://www.vinc17.net/unix/index.en.html#zsh-ssh-utils`

that offer more elegant handling of SSH keys, such as only prompting you for a passphrase when you actually use the keys instead of on startup.

vi. On the "Adding a new SSH key to your GitHub account" page, the fact that you are SSH'd into the computer on which you are generating keys makes Step 1 (copying

the ssh key into your clipboard) a bit trickier than they otherwise would be. The easiest way to make this work will probably be to run

```
nano .ssh/id_ed25519.pub
```

and then copy the contents of the file out of the terminal.

vii. If you left the original Settings page open in another tab, you can then follow the remaining steps on the "Adding a new SSH key to your GitHub account" page.

1 Basic arm operations

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. This code is, however, written in large monolithic form. Refactor this code into smaller blocks.

1. Run through all of the example programs. You may need to edit the position value at which the gripper servo is considered "closed".
2. Break down pick-and-place operations into a hierarchy of functions. At each level of abstraction, the purpose of each function should capture a concrete idea, and the number of subfunctions directly called should be small.

1.1 Deliverables

1. Code review from this week's downstream partner
2. Code review of this week's upstream partner's work
3. Reflection questions:
   (a) What did you learn while completing the tasks in this lesson?

2 Path following with servos

sudo -E

3 Arm Jacobians

The inverse-kinematics algorithm used in the stock arm code is written in an explicit closed form. Often, it is more useful to approach arm control problems via a "resolved-rate" inverse kinematics approach, in which a the motion of the end effector is specified and