COS 120 Robot Module

Exercises

Spring 2017

Contents

Exercise 1: Building the robot

Overview

In this exercise, you are going to build your robot and verify that all of the connected components are working.

Goals

The goals of this exercise are to introduce you to robot hardware and for you to build a working robot. You will also understand how to operate the robot at a basic level and be familiar enough with it to troubleshoot basic problems.

Background and preparation

- Resource 1: The Tutorial for building the GoPiGo[1]
- Resource 2: GoPiGo Servo assembly instructions[2]

Materials needed

- Dexter Industries GoPiGo2 base kit
- Raspberry Pi 3 Model B
- Raspbian SD card
- GoPiGo servo assembly
- Phillips head screwdriver
- 8 Rechargeable (preferably) AA batteries

Requirements

- Build the GoPiGo base kit.
- Test and troubleshoot the GoPiGo.

[1] www.dexterindustries.com/GoPiGo/getting-started-with-your-gopigo-raspberry-pi-robot-kit-2/1-assemble-the-gopigo-2/assemble-gopigo-raspberry-pi-robot/1-assemble-the-gopigo2

[2] www.dexterindustries.com/GoPiGo/getting-started-with-your-gopigo-raspberry-pi-robot-kit-2/1-assemble-the-gopigo-2/assemble-servo-package-assemble-the-raspberry-pi-robot-servo-kit-with-the-gopigo

Building the GoPiGo base kit

1. Open the box and verify that you have all of the components. You should have:
    - 2 wheels
    - 2 wooden encoder wheels
    - 4 spacers
    - 4 t-shaped plastic pieces
    - two bags of hardware (bolts, screws, etc.) which include the items that are listed in the build tutorial
    - the robot base
    - the robot top
    - 2 motors
    - battery box
    - GoPiGo board
    - power cable
    - a Raspberry Pi 3 board
    - a camera and its ribbon cable
2. Either by following the instructions on the website, using the video, or both, assemble the GoPiGo Base Kit. **DO NOT** put batteries into the battery box in this step. If you have any questions or are not sure of an instruction, please ask the instructor or CLA.
3. Once you have completed assembly of your GoPiGo, ask the instructor or CLA to make sure everything looks correct. If everything is correct, they will provide you with the assembly for the GoPiGo servos. They should also screw a mount into the base board of your GoPiGo. Be careful, the mount is very delicate.
4. Attach the assembly to the mount and plug it into the GoPiGo board.
5. Plug the ultrasonic sensor into the A1 port on the GoPiGo board.
6. Mount the camera, if it is not already part of the servo assembly; ask your instructor or CLA for directions about how to mount it.
7. Plug the camera ribbon cable into the appropriate slot on the GoPiGo board.
8. Put the microSD card into the appropriate slot on the Raspberry Pi.
9. Put the batteries in the battery case and connect the battery cable and case to the GoPiGo.

Testing the robot

In this section, we will verify that the components of your GoPiGo are running correctly as well as set up the Raspberry Pi so that you know the password and hostname.

1. Connect to the Raspberry Pi:[3]
    - Ask the CLA for the hostname of the Raspberry Pi as well as the user name/password to use to start.
    - Follow the Connecting to the GoPiGo tutorial on the Dexter Industries website.[4]
    - Use the hostname provided instead of "dex.local" when connecting.
    - Use the user name and password provided to log in.
2. Use the `test_gogpigo` script to test the robot.

[3] **[This will be revised based on the available resources and configuration of the classroom.]**

[4] Go to www.dexterindustries.com/gopigo-tutorials-documentation, select "Getting Started", and then select the tutorial.

- Open a terminal on the Pi by clicking on this icon:

3. Type `test_gopigo` and press the enter key.
   - You should now see the GoPiGo perform a few actions, waiting for you to press enter between them:
     - Blink the left then right red forward-facing LED on the GoPiGo board.
     - Move forward two wheel rotations
     - Move backward two wheel rotations
     - Turn Right 90 degrees in place
     - Turn Left 180 degrees in place
     - Return to original position
     - Rotate Servo to the left 90 degrees and then pan all the way to the right printing numbers (distance to objects based on the ultrasonic sensor) as it goes.
     - Servo returns to original position.
   - If any of these do not work, notify the CLA or instructor for help.

Questions for thought

- How easy do you feel it was to build the robot?
- What was the hardest part?
- Based on the little bit you now know about the robot, how difficult do you think it would be to write a Python program to move the robot around while avoiding obstacles? What actions that you have seen the robot do would you expect to use?

Exercise 2: Using raspbian

Overview

In this exercise, you will begin to learn how to use Raspbian, the operating system on the Raspberry Pi processor of your GoPiGo. You will communicate with Raspbian either via a terminal window or a graphical user interface. You will also learn how to connect to your Raspberry Pi without a monitor in each of those ways.

Goals

The goal of this section is for you to learn to use your Raspberry Pi, which will be how you control your robot. It is recommended that you try all of the methods in the packet, although it is not required.

Resources

1. VNC viewer for the computer you want to control your Raspberry Pi.[5]
2. Useful tutorial on using the Linux command line.[6]
3. An Extremely Quick and Simple Introduction to the Vi Text Editor.[7]
4. List of vi commands.[8]
5. Beginner's Guide to Nano.[9]
6. Absolute Beginner's Guide to Emacs[10]
7. PyCharm tutorials and documentation[11]
8. Python Editor IDLE video[12], tutorials, and other information[13]

Materials needed

- Your GoPiGo.
- A computer with an Internet connection.
- The hostname of your GoPiGo. (Ask the CLA or instructor if you don't know this.)

[5] www.realvnc.com/download/viewer
[6] www.pcsteps.com/5010-basic-linux-commands-terminal
[7] heather.cs.ucdavis.edu/~matloff/UnixAndC/Editors/ViIntro.html
[8] www.cs.colostate.edu/helpdocs/vi.html
[9] www.howtogeek.com/howto/42980/the-beginners-guide-to-nano-the-linux-command-line-text-editor
[10] www.jesshamrick.com/2012/09/10/absolute-beginners-guide-to-emacs/
[11] Available at www.jetbrains.com/pycharm/documentation
[12] www.youtube.com/watch?v=lBkcDFRA958
[13] www.annedawson.net/Python\Editor\IDLE.htm

Connecting to the GoPiGo

There are two methods we will be using to connect to the Raspberry Pis in this class: VNC (Virtual Network Computing) which acts like remote desktop and can be used with either GUI or a terminal window; and SSH (secure shell) which can only be used as a terminal window.

Using VNC

1. Connect to the Raspberry Pi (same as in Exercise 1):
   - Ask the CLA for the hostname of the Raspberry Pi as well as the user name/password to use to start.
   - Follow the Connecting to the GoPiGo tutorial on the Dexter Industries website to connect to the Pi via VNC.[14]
   - Use the hostname provided instead of "dex.local" when connecting.
   - Use the user name and password provided to log in.
2. This should bring you to the user interface (the desktop) of the Pi. Have a look around to familiarize yourself with it.

Using SSH

SSH is a fast and easy way to communicate with another computer. You will need to open a terminal on your computer in order to run SSH; once you run it to log in to the Pi, it will be as if you were using a terminal on the Pi's own desktop.

1. Open a terminal:
   - If you are using a Macintosh or a Linux machine: Open a terminal in the usual way.
   - If you are using Windows **[to be written; may need additional software]**
2. At the command prompt, type:

   ```
   ssh USER@HOSTNAME
   ```

   where `HOSTNAME` (e.g., "foo.cs.umaine.edu") is the name of your Pi, and `USER` is the user name the CLA gave you.
3. SSH may ask you if you would like to connect to the unknown host for the first time; if so, type "yes" and press the enter (or return) key.
4. Next, you will be asked for a password; type in the one the CLA gave you and press enter.
5. You should be in! The line will look something like

   ```
   user@hostname:~$
   ```

   This is the pi's *command prompt*, telling you it is ready to accept commands.

[14] Go to www.dexterindustries.com/gopigo-tutorials-documentation, select "Getting Started", and then select the tutorial.

Using the Linux command line

- Once you are connected to the Pi, you are talking to its *operating system*; you are probably familiar with operating systems such as Windows, macOS, Linux, iOS, and/or Android.
- The Pi's operating system is a version of Linux called Raspbian. If you are familiar with Linux (or the macOS command line), Raspbian will seem very familiar.
- Raspbian organizes files into *directories*, which you may know as "folders" on other operating systems.
- Directories are in a hierarchy, starting from the *root directory*, which is just known as `/` (slash).
- At all times in a terminal, there is a *current working directory*. To see what this is, in a terminal window type:

  ```
  pwd
  ```

  which stands for "print working directory". (You'll note that Linux commands are very short.) You should see something at this point like: `/home/foo`, if your user name is "foo".
- To move between directories, use the `cd` (change directory) command. Try:

  ```
  cd Desktop
  pwd
  ```

  If you are doing this on the Pi and starting in your home directory, it should tell you that you are in the directory `/home/foo/Desktop` – that is, you are in the "Desktop" directory of the "foo" directory (your *home directory*) of the "home" directory, which is located in the root directory.
- Three special ways to refer to directories are worth knowing about:
  - "~" is your home directory; these two commands have the same result:

    ```
    cd ~
    cd /home/foo
    ```

    as do these:

    ```
    cd ~/Desktop
    cd /home/foo/Desktop
    ```
  - ".." is the parent directory of the current directory; so if you are in your Desktop directory, then these are the same:

    ```
    cd ..
    cd /home/foo
    ```

    as are these:

    ```
    cd ../..
    cd /home
    ```
  - "." is the current directory; you may have a use for this in the future.
- The command `ls` lists all files in the directory.
  - It accepts *wildcards*, which in Linux are represented as "*".
  - For example, to see all files with the characters "ware" in their name, do:

    ```
    ls *ware*
    ```
- See the tutorial on basic Linux commands[6] mentioned in the Resources section for more details.

Using Python on the Pi

Using a window-based integrated development environment (IDE):

- You can use the IDLE Python IDE (integrated development environment) or the PyCharm IDE (recommended) from the menu on the Pi's desktop (e.g., when using VNC). Click on the main menu, select programming, then click the menu item for the IDE of choice.
- To get started using PyCharm, see the documentation and tutorials available on the IDE's website.[11]
- To get started using IDLE, you can view an introductory movie[12]or read a one of the tutorials[13] from resources above.
- Note that we will be using Python 3 in this class.

Using Python from the command line

- To run Python from the command line, just type: `python3`
  - We will be using Python 3 in this class.
  - There are some minor differences (as far as we are concerned) between this and Python 2, the most noticeable of which is that the arguments to `print` have to be enclosed in parentheses in Python 3.
- To run a Python program from the command line, say `my_program.py`, then do:

```
python3 my_program.py
```

Editing files

There are several editors you can use to edit files on the Pi (and on Linux and Mac machines in general).

- nano: A very simple editor that is likely to be the best for your purposes, at least at first. See the Beginner's Guide to Nano[9] to get started.
- vi: One of the two most widely-used "programmer's editors". See An Extremely Quick and Simple Introduction to the Vi Text Editor[7] and a List of vi commands.[8]
- Emacs: The other one of the programmer's editors. Very much worth learning, since it can do just about everything (including simulate a terminal, run Python, etc.), but a steep learning curve. If you are interested, see the Absolute Beginner's Guide to Emacs[10]

Using SFTP to send files to your Raspberry Pi

You may want to create Python programs on your computer, then send them to the Pi, or you may create programs on the Pi and want to back them up to your own computer. In either case, you will need to be able to move files back and forth. For this, we will use either SFTP (SSH File Transfer Protocol) or SCP (secure copy), both of which are easy ways to transfer files between computers.

SFTP

1. From a new terminal window, type:

```
sftp user@hostname
```

where again `hostname` is the name of your Pi and `user` is the user name.

2. You are now in your user directory on the Pi (i.e., `/home/user`). We will call this the *remote* directory, as opposed to the directory you are in on your local machine, which is the *local* directory.
3. You can see what is the directory by typing `ls` (the list command).
4. You can move to another directory using the `cd` (change directory) command.
   - E.g., to go to the directory of your remote desktop and see what is there, type

     ```
     cd Desktop
     ls
     ```

   - To move up a directory, you can do: `cd ..`
   - To move to your home directory on the remote machine (the Pi), type: `cd /home/user`.
5. To see what directory you are in, type `pwd`.
6. There are local versions of all these commands: `lls`, `lcd`, and `lpwd`.
7. To get the file `filename` from the Pi:
   - `cd` to the correct directory on the PI and `lcd` to the correct directory on the local machine.
   - Type: `get filename`
8. To put a file `filename` onto the Pi:
   - `cd` to the correct directory on the PI and `lcd` to the correct directory on the local machine.
   - Type: `put filename`
9. To exit, type `exit` or `bye` and press enter.

SCP – SCP is somewhat more succinct, but relies on you knowing all the directories and filenames involved.

1. Suppose you have a file `filename` in the current directory and you want to copy it to the Pi on your desktop, but call it `newfilename`.
2. In a terminal window on the local computer in the local directory desired, type:

   ```
   scp filename user@hostname:Desktop/newfilename
   ```

   where `HOSTNAME` is as usual the name of your Pi. It will likely ask you for a password, then it will show the status of the copying as it completes your request.

Questions for thought

- Do you feel that you now understand Raspbian, VNC, and SSH enough to run programs on the Pi for the rest of this class?
- With the knowledge of this introduction, do you feel that you will be using command lines, or just the GUI (VNC) to navigate? Why?

Exercise 3: Exercising the robot

Overview

In this exercise, you will put your robot through its paces, learning how to control it as you go. You will also "trim", or adjust, the motors on your robot so that it be better at moving in a straight line.

Goals

The goal of this exercise is to familiarize yourself with the functions in the supplied `gopigo` module (`gopigo.py`) for commanding and getting information from the robot.

Resources

1. "Python Programming", from Dexter Industries' website.[15] We won't be doing part 5 ("Weaponizing" the GoPiGo), but watch the videos and read the tutorials on the page. This page also provides very valuable information about the GoPiGo's API (application program interface, the set of functions you use to interface with the robot).
2. "Add trim to the motors", from Dexter Industries' website.[16]

Materials needed

You will need:

- Your GoPiGo (with the sonar sensor attached) and some way (e.g., a computer) to communicate with it
- A smooth floor with enough space to move the robot around.

Part 1: Using the robot API

1. Start Python on the robot and do: `from gopigo import *`.
2. Try each of the motor control functions listed in Resource 1 (see above).
3. Try different settings of the motor speeds via the motor speed functions described in Resource 1.
4. Try setting encoder targets using `enc_tgt()` and using `fwd()` and `bwd()`.

[15] www.dexterindustries.com/GoPiGo/programming/python-programming-for-the-raspberry-pi-gopigo

[16] www.dexterindustries.com/GoPiGo/programming/python-programming-for-the-raspberry-pi-gopigo/add-trim-to-the-motors

5. Read the sonar (ultrasonic sensor) using `us_dist(15)`. (The "15" is the hardware address of the sonar.)
6. Try entering the following into Python:

   ```
   from time import sleep
   while True:
       print us_dist(15)
       sleep(0.5)
   ```

   Now move your hand in front of the sensor in various ways, watching what the loop prints in response.
7. Turn the LEDs on and off.
8. Move the servo back and forth.
9. Check the status of the robot, including the battery voltage.
10. Glance through the gopigo.py file; this should be in GoPiGo folder on the robot's desktop, under Software→Python. If there are any other functions you see there that look interesting, try them out and make a note of them for later.

Part 2: Trimming the motors

1. Follow the instructions on the Dexter Industries website[16] for adjusting the hardware settings on the robot to trim the motors:
2. Trim the motors so that the GoPiGo goes nearly in a straight line for a meter or more when you call `fwd()`.

Questions for thought

Was there anything unusual that happened during this exercise? Did you have any problems? If so, what could you have done differently, and/or what information do you wish we had given you beforehand?

Exercise 4: A movement experiment

Overview

In this assignment, you will consider real versus expected movement by your robot. Since experiments are important in computer science, just as in natural sciences, this assignment involves performing a formal experiment.

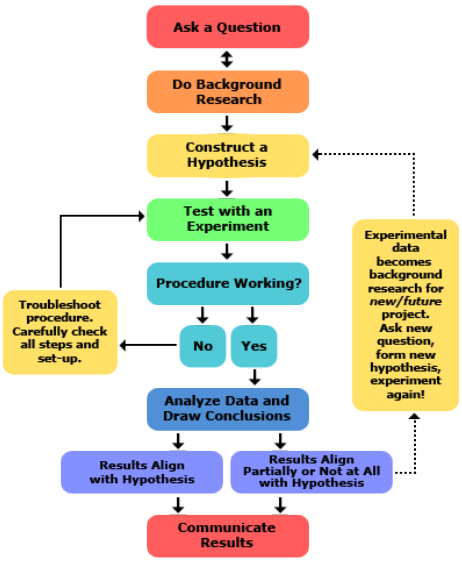
Goals

- Gain experience controlling the robot with Python.
- Refresh your memory of the scientific method.
- Gather data you will use in the next assignment.

Background

Scientific method

To refresh your memory about the scientific method, here is a good diagram of it from a site aimed at helping high schoolers select science fair projects:[17]



[17] www.sciencebuddies.org/science-fair-projects/project_scientific_method.shtml

The "question" you'll be asking in this assignment is: "Does the robot perform as expected when commanded to go forward, and if not, how does it deviate from expectations?" We'll skip the "background research" portion here. However, you do need to formulate a hypothesis to test. This can be very general (e.g., "The robot will not behave as expected.") or very specific (e.g., "The robot will deviate from expectations by over 10% to the right when moving at 100% of its maximum speed")—it's up to you.

You can gather evidence for a hypothesis, but you can't *prove* it outside of the pure realm of mathematics. It is also much easier in general to reject a hypothesis than to gather enough evidence to directly support it. The standard thing to do is to flip the hypothesis around into a *null hypothesis*, then try to *reject* that at some level of confidence; if that is possible, then the hypothesis itself is considered to be well-supported.

A null hypothesis is a statement about what you think would be true if your hypothesis is false. For example, if you were doing an experiment to see if fertilizer increased the growth rate of algae in water and your hypothesis was that it does, your null hypothesis would be "fertilizer has no effect on the growth rate of algae in water". In the case of this assignment, if your hypothesis is "the robot will not behave as expected", then the null hypothesis would be "the robot will behave as expected". You would then gather data to reject that hypothesis.

In most experiments, there is are one or more *independent variables* and one or more *dependent variables*. The independent variables are those that you can control in your experiment, while the dependent variables are those that really are of interest; they are the ones that change as a result of changes you make to the independent variables. Ideally, only a very small number of independent variables will be varied an experiment, and usually only one or a few dependent variables are measured.

In the algae experiment, there are many possible independent variables, including temperature, salinity, etc., but the one we're most interested in is the amount of fertilizer added to the water. All the rest we strive to keep constant. The dependent variable of interest is the growth rate of the algae (however it is measured).

For this assignment, there is more than one candidate for the independent variable: we could use speed, distance traveled as measured by the robot's wheel encoders, amount of time we let the robot run, and so on. The dependent variable is the position of the robot at the end of the run.

True formal experiments have a *control*, that is, something to compare the experimental *treatments* against—where a *treatment* is one setting of the independent variable(s). In the algae example, the control would be containers of water with no added fertilizer, while the treatments would be containers with differing amounts of fertilizer added. The purpose of a control is to factor out any variables that we may not have thought of; as much as possible, the controls are exposed to exactly the same conditions as the treatments, except for the actual settings of the independent variable(s).

For this assignment, we will use as our control the expected position of the robot after a run. In essence, we are ignoring the effects of *all* environmental variables that might affect the robot's final position. Another way of thinking of this is to flip things around a bit: the distance traveled (or duration, or speed) can be thought of as about the only thing that is constant between the (calculated) control and the other runs; the real independent variables are all the properties of the real world we can't account for; the dependent variable, from this view, would be the deviation from the expected position. Although this is arguably a better way to view the situation, we will continue assuming the independent variables are the ones having to do with our commands to the robot.

Since we can't control all variables in the environment, there will be *noise* in our data caused by random (or unknown) changes in the world. Thus, most experiments have multiple *replications*:

multiple controls, multiple treatment cases, etc. For the algae experiment, we might have 20 containers with no fertilizer (the controls) and 20 containers for each concentration of fertilizer applied. For this assignment, we will perform several runs for each setting of the the independent variable(s). If we were going to use statistics to reject the null hypothesis, then the number of replications would in part depend on the statistical method used (Student's t-test, ANOVA, etc.).

Once the data is gathered, it is time to compare the results of the experiments to the control. In the algae experiment, this would be the growth in the experimental cases versus the control. For us, this will be the real versus expected positions of the robot.

In real experiments, we would perform a statistical analysis of our data to see if we can reject the null hypothesis. We have to have some standard of how likely our rejection is to be valid, however, that is, how much confidence we have in our conclusion. The level of confidence we have in rejecting a null hypothesis is usually expressed as something called the *p-value*, and you'll see it written after a conclusion in a scientific paper as something like "($p = 0.013$)" or ($p < 0.05$)". This means that there is a probability of 0.013 (i.e., "a 1.3% chance") or less than 0.05, respectively, that the results were due to chance and not to a real difference from the null hypothesis. Statistical tests provide us with p-values for the effects noticed in our data.

The usual threshold for something being considered "statistically significant" in science is $p < 0.05$, i.e., a 0.95 probability that the effect observed was real. Anything less (i.e., any p-value ≥ 0.05) and we should be concerned that our results were just due to chance.

In this assignment, you will use a very simple statistical test, the Student's t-test, to see if your null hypothesis can be rejected or not.

Resources and tutorials

- Introduction to the Scientific Method[18]
- Wikipedia article on Scientific Method.
- Statistics for Dummies Cheat Sheet[19] – if you want to do some statistical testing of hypotheses, this may help a bit.
- Choosing which statistical test to use – statistics help[20] (video)

Materials needed

You will need:

- Your GoPiGo.
- A tape measure or some other way of measuring distance
- A smooth floor with sufficient space for the robot to run on (suggestion: minimum 1 m × 1 m)
- Masking tape or other easy-to-remove tape
- A means of doing a t-test; spreadsheet programs, for example, provide this, and there are also statistical packages available for Python.

[18] `teacher.nsrl.rochester.edu/phy_labs/appendixe/appendixe.html`
[19] www.dummies.com/education/math/statistics/statistics-for-dummies-cheat-sheet
[20] www.youtube.com/watch?v=rulIUAN0U3w

Requirements

- Based on what you know about how the robot moves when commanded to go forward (`fwd()`) (after trimming), formulate a hypothesis about how its motion will compare to that expected in an ideal case (i.e., going in a completely straight line).
- Formulate the null hypothesis.
- Design an experiment to test your hypothesis
  - Determine what the independent and dependent variables are.
    - Although we said there were several we could use, a good independent variable is the robot's speed.
    - Think about picking a set distance for the robot to travel in all treatments and varying the speed at which it covers the distance.
    - The motor speed for the GoPiGo is set using a scale of 0–255.
    - Distance can be measured by using the GoPiGo's wheel encoders, which generate a signal 18 times per revolution of a wheel.
  - Decide the number of treatments (i.e., settings of the independent variable, speed) you will have.
  - Decide how many replications you will do of each treatment. You may want to look at the suggested resources on the t-test to help you decide this.
- Perform the experiments
- Analyze the results
- Keep the data for use in the next assignment!

Programming notes

Overview: Top-down design

In this assignment, you are going to write a program to control the experiment you will be performing, including telling you what to do, moving the robot, and collecting data. This will be a fairly complex program.

When faced with a complex program to write, it helps to break it down into smaller pieces that make sense, even if you don't yet know how write *those* pieces. Then, once you know how the pieces fit together, you work on writing each of them in the same way: breaking them into smaller pieces, etc. This is called *top-down design*, *step-wise refinement*, or, sometimes *structural decomposition*. Often this is paired with *bottom-up implementation*: when you have broken up all the functions into other functions, starting at the smallest, implement them. The two are not really separate, however: often during top-down design, functions are essentially complete as soon as you understand how to break them into smaller pieces.

In this exercise, we'll develop the program in a top-down fashion. Let's first think about the design of the program needed to guide you as you conduct the experiment, to move the robot, and to collect the data, then think about how to design each of the pieces, and their pieces, etc. Because this will be a rather long program—maybe the longest you've done yet—we will provide you with a skeleton that includes the function headers and `import` statements.

Main program: `do_experiment()`

Let's call the main program `do_experiment()`, and let's assume it has the parameters:

- `num_trials`: number of trials to perform;

- `repetitions`: number of repetitions per trial;
- `distance`: the distance to move the robot each time; and
- `filename`: the name of the file in which to write the data.

With these parameters—and by not hard-coding their values into the program—we can use the function to carry out different versions of the experiment without making changes to it.

There are three tasks for `do_experiment`:

1. Set up the robot and initialize any variable needed.
2. Do the actual trials.
3. Write the data to a file.

The setup part of the program is simple enough that we can include that directly in `do_experiment()`. All it needs to do is to make sure the robot is ready to execute motion commands and return sensor values. In this case, this means just enabling the wheel encoders.

The other two pieces are more complex and should be written as separate functions, which we can design later. In the meantime, we can assume that we already have them written and write the main program using them.

Looking first at step 2, we can do this by calling a function, `do_trials()`. This function will need to know how many trials there are, how many repetitions to do for each, and the distance to move the robot, so these will be parameters; that is, we will call it as `do_trials(num_trials, repetitions, distance)`.

Step 3 will also be done by a function let's call `write_data()`. This will need to know the data to write and the file name to which to write it, so we can call it as `write_data(data,filename)`.

This raises the question: Where did `data` come from? The answer is, we want `do_trials()` to return it: it will be the data you measure for each repetition of each trial. But what should the format of the data be?

Oddly enough, we don't need to know this right now; the data is produced by `do_trials()` and given to `write_data`, so the main program, `do_experiment()`, never needs to know the format! This is one of the advantages of top-down design: we can delay committing to details (in this case, the data's format) until it makes sense to do so.

We can write *pseudocode* for the main program now. Pseudocode is a way to write the *algorithm* a program carries out in a way that is close enough to the way the program will be written to be easy to translate to code, yet is readable. Pseudocode for the main routine would look like:

- `do_trials(num_trials, repetitions, distance, filename)`:
    1. Set up the robot.
    2. Call `do_trials(num_trials, repetitions, distance)`, collecting data into `data` as a list.
    3. Call `write_data(data, filename)`

1. `do_trials(num_trials, repetitions, distance)`

    This function's overall purpose is to run each trial for the number of replications needed, collecting the data (e.g., into a list) to be written out later by `write_data()`. We can think of this function's structure as basically a loop, with one iteration for each trial to be performed. Since we are concentrating on the structure of this function here, we can let another function, `do_a_trial()`, be called in the loop to conduct a single trial.

    Before the loop, however, although we know the number of trials desired at this point (a parameter passed from the main program), we first need to determine what the trials should *be*. This can be done by another function, `create_trials()`, that generates a list of trials.

    The structure of the `do_trials()` function thus looks like:

- `do_trials(num_trials, repetitions, distance)`:
  (a) Let `trials` = `create_trials(num_trials)`.
  (b) Do for each `trial` in `trials`:
    – `do_a_trial(trial, repetitions, distance)`, collecting results into `data`.
  (c) Return `data`.

There are two variables introduced in this pseudocode, `trials` and `data`. This is a point where we need to think about the format of `trials`, since this function has to iterate over it. This means that it should be either a list or a tuple. Thinking ahead to `create_trials()`, which produces the value `trials` will contain, it is likely to be better to use a list, since (though we haven't really considered the details at this point) `create_trials()` will likely use the common coding pattern (sometimes referred to as an *idiom*) of accumulating the values for trials it creates one by one. Since a list can be modified (e.g., to add a new element) and a tuple cannot, then we'll assume it's a list and make a mental note that this is what `create_trials()` should return.

The other variable, `data`, contains the data collected from all the trials. We don't have to know what each piece of data looks like in order to write this function, but we *do* know that we'll be adding them to `data` as they are produced. This means that `data` should be a list, each element of which is a data point.

There are two ways in Python to add elements to the end of a list, as we need to do here, both of which are *methods* of the list data type, `extend()` and `append()` `append()` is the one we want, since `extend()` will splice two lists together if the data item is a list or a tuple, which it is likely to be. Append, on the other hand, does what we want:

```
>>> a = []
>>> a.append((1, 2))
>>> a
[(1, 2)]
>>> a.append((3,4))
>>> a
[(1, 2), (3, 4)]
```

whereas `extend()` does not:

```
>>> a.extend((5,6))
>>> a
[(1, 2), (3, 4), 5, 6]
```

Now we can look at the next-lower abstraction level: the two functions `create_trials()` and `do_a_trial()`, which this function needs.

2. `create_trials(num)`

Assuming that speed is the independent variable for our experiment, then `num_trials()` justs needs to return a list of speeds to use. It would be best to randomly generate the speed values, since this will help eliminate any possible bias on the part of the experimenter that might cause some representative speeds to be omitted. Thus this is a simple function:

- `create_trials(num)`:
  (a) Do `num_trials` times:

   i. Let `s` = a randomly-generated speed in the range 0–255.
   ii. Collect `s` into a list `trial_list` unless it is already in it.
   iii. If `s` *was* already in `trial_list`, then go to 2(a)i.
(b) Return `trial_list`.

Some tips for this function:

- Use a `for` the loop in step 2a.
- To randomly-generate a value, use Python's `randint()` function, which is in the `random` module.
  - At the top of your file, do:

    ```
    from random import randint
    ```
  - It takes two arguments; if it is called as `randint(a, b)`, then it will return a number n such that $a \leq n \leq b$.
- You will want to initialize `trial_list` to the empty list:

  ```
  trial_list = []
  ```
- You'll want to use the `append()` method of the list data type to add new values to this, e.g.:

  ```
  trial_list.append(x)
  ```

  to add `x` to the end of `trial_list`.
- To check to see if something is in a list, use `in`. For example,

  ```
  >>> a = [1, 2, 3, 4, 5]
  >>> 3 in a
  True
  >>> 6 in a
  False
  ```
- You'll want to keep generating values for `s` until it contains a value not already in the list. You can do this with another kind of loop, a `while` loop.
  - For example:

    ```
    >>> a = [1, 2, 3, 4, 5]
    >>> while int(input()) not in a:
    ...     print("nope!")
    ...
    14
    nope!
    23
    nope!
    3
    >>>
    ```
  - A word of explanation: `input()` gets a string from the user, and `int()` converts its argument into a number.
  - Don't forget to return `trial_list` at the end of the function!

`do_a_trial(trial, repetitions, distance)`

This function does the work of running some number of repetitions of a single trial (i.e., moving the robot at a given speed for `distance` cm). This is basically a loop that iterates `repetitions` times, each time conducting a single repetition of the trial:

- `do_a_trial(trial, repetitions, distance)`:
  1. Loop `repetitions` times:
     (a) Tell the user (i.e., you) to position the robot for a new repetition.
     (b) Wait for the user to indicate that the robot is in position.
     (c) Let `start_time` be the current time.
     (d) Move the robot `distance` cm at the speed specified in `trial`, keeping track of how long it takes.
     (e) Let `end_time` be the current time.
     (f) Tell the user to measure the robot's position and enter it (as an x and a y value).
     (g) Collect the data into a list `data`
  2. Return `data`.

You can just use a print statement to tell give the user instructions. You can wait for the user by using the `input()` function, e.g.,:

```
input('Press enter when the robot is in position: ')
```

To get the current time, we will use the `time()` function from the `time` module; put this at the top of your file:

```
from time import time, sleep
```

We suggest you also import `sleep()`, since that will be needed later.

It turns out that step 1d is more difficult than it seems, since we need to move the robot along the x-axis and wait for it to finish before doing the next step. We'll use a new function `move_x(speed,distance)` to do this step, described below.

Note that we *could* split this function up further if we wanted, for example, by creating a function that does all the work in the body of the loop. However, it is simple enough that we really don't need to.[21]

We now have to decide on the format of the data. We suggest that you create a tuple to hold the data for each replication of the form: `(start_time, end_time, speed, distance, x, y)`, where the values are can be found in variables in the function.

Again, use the `append()` method to add tuples to the end of `data`. Don't forget to return `data` from the function!

`move_x(speed, distance)`

This needs to move the robot `distance` cm along the x-axis at `speed`. It could look like:

- `move_x(speed, distance)`:
  1. Set the robot's speed to `speed`.
  2. Move forward `distance` cm, waiting until it is done.
  3. Return.

Some notes:

- To have the function wait until the robot is done moving, set a target, start the robot, then wait for it to reach the target.
- You will use the robot's *wheel encoders* to set a target distance for the robot.
  - The encoders send 18 pulses for each revolution of the wheels. (There is an encoder for each wheel, but we will treat them as a unit.)

[21] At least, at this level of thinking about it, it doesn't seem to need breaking up; if we discover as we are writing the code that the function is becoming complex and/or long, at that point it may make sense to *refactor* it into a parent function with one or more new functions.

  - There is a function, `enc_tgt()` that sets a target for the encoders that, once reached, stops the robot's motion; the target is specified as some number of encoder pulses.
  - Your program will need to convert `distance` into the corresponding number of pulses.
    * `WHEEL_CIRC` is a constant supplied in the `gopigo` module that contains the robot's wheels' circumference, in cm.
    * There are $r = \frac{\texttt{distance}}{\texttt{WHEEL\_CIRC}}$ wheel revolutions in `distance`.
    * `PPR` is a constant in the `gopigo` module that holds the number of encoder pulses per revolution (18).
    * The number of encoder pulses occurring in `distance` is then $r \times$ `PPR`.
  - The call to `enc_tgt()` takes three arguments:
    * The first two tell the robot which wheel encoder(s) to use.
    * The third is the number of pulses that is the target value.
    * If p is the number of pulses, call this as `enc_tgt(1, 1, p)`.
- To set the robot's motor speed, you use the function `set_speed()`.
  - Recall that motor speed must be in the range of 0–255, with 255 being full speed.
  - Your program should check to make sure that the specified speed is within this range.
  - If `speed` < 0, then set the motor speed to 0; if it is > 255, then set the motor speed to 255.
- Once the speed is set and the encoder target is set, your function can move the robot by calling `fwd()`.
- The function has to wait until the robot finishes movement before returning.
  - This can be done by using a loop that continuously checks the encoder status via the built-in function `read_enc_status()` and only exits when the status is 0 (signaling that the target has been reached).
  - You should consider using the `sleep()` function of the `time` module to wait a fraction of a second between checking the encoder status, since otherwise the loop will take up a great deal of the robot's processing capacity. An example of how this could be done is:

    ```
    from time import time, sleep  # see above
    while read_enc_status() is not 0:
       sleep(0.1)
    ```

    which would check to see if the motion has stopped every tenth of a second.[22]

`write_data(data, filename)`

This function handles writing the data from the experiment to the file named `filename`. We need to think about how we plan to use the data to figure out in what format we should write it. Since it is likely that we will want to manipulate the data—sorting it by speed or error, for example, or performing statistics on it—a good format is one that a spreadsheet program can read. Almost all such programs can read/write *comma-separated value* (CSV) files, so let's use that. A CSV file would have each data point on a line, with commas separating the individual pieces of the data.

The `write_data()` function will look like:

- `write_data(data, filename)`:
  1. Open the file `filename`.
  2. Loop for each data item (i.e., each tuple) in `data`:

[22] Note that if you later need to compute the robot's actual speed in cm/s, i.e., $d/\Delta t$, the accuracy will be inversely proportional to the wait time in this loop. Thus there is a trade-off between accuracy and wasting the computer's time and effort. This is not uncommon in computer science.

   (a) Write the tuple to the file, separating elements by commas.
3. Close the file.

Some information:

- Files have to be opened before they can be read, and they need to be closed after you are done with them.
- Use the `open()` function to open a file.
  - This takes two arguments, the name of the file to open (e.g., `filename` above) and the *mode*—read, write, etc.
  - To open the file named 'foo' for writing, do:

    ```
    f = open('foo', 'w')
    ```

  - The variable `f` above holds the *file object* returned by `open()`. You will use this object's methods to access the contents of the file.
- To write data to the file, use the `write()` method.
  - `f.write('hi there')` would write the string 'hi there' to the file.
  - You have to tell `write()` when you want to end the line. You do this via writing a *newline character* to the file, e.g., `write('\n')`.
  - To convert a number into a string for writing, use the `str()` function, e.g.,

    ```
    >>> str(2.3)
    '2.3'
    ```

- Since it's a little tricky to write commas between data items, but not after the last one, we'll give you this one:

```
for data_point in data:
    d = []
    for thing in data_point:
        d.append(str(thing))
    f.write(','.join(d) + '\n')
```

- To close the file, use the `close()` method, i.e., `f.close()`.

Testing & running the experiment

- Testing:
  - You should test your functions to make sure they do what you want before running your experiment.
  - Often it's easiest to test a program written top-down by starting at the smaller functions—i.e., testing it bottom-up—and making sure they behave correctly, then testing functions that use them, etc.
  - Test the entire program by following the steps for "Running the experiment" below, but with a replication of 1 or 2.
- Preparing to run the experiment:
  - With the tape, mark x- and y-axes on the floor, with each being at least as long the distance you intend the robot to travel in your experiments.
  - The robot will, if it performs correctly, travel straight down the x-axis.
  - The y-axis needs to be orthogonal to (90-degrees from) the x-axis, and pointing to the left as you face in the direction of the x-axis.

- Running the experiment:
  - Start Python.
  - Load (import) your Python file containing the functions you've written.
  - Call the `do_experiment()` function
  - Do what your program tells you.
  - When done, make sure that your CSV file looks correct.
- Analyze your data.
  - Import the CSV file into a spreadsheet program (e.g., Excel, Numbers, LibreOffice, OpenOffice, etc.) for easy viewing and manipulation.
  - Examine the data – what can you say about it in general?
  - What can you say about your hypothesis in light of the data? Can you back that up by using the spreadsheet's t-test or other statistical tests?

Questions for thought

- What was your hypothesis? Was it a good one? Would you create a different one if you were to do the experiment again?
- If there were errors in movement, can you conclude anything about them? Are they predictable, i.e., given a speed, can you predict the error? Do they conform to some equation?
- Is there any correlation between motor speed and actual speed of the robot? If so, is there an equation that can convert from one to the other?

Stretch goals

- It is possible that `trials()` could return two or more trials with the same motor speed. Modify it so that all of the trials are unique.
- Use a statistical method to provide support for your hypothesis based on your data.
- Write a function based on what you've learned that, given a speed and a distance, will return the estimated errors in x and y (as a list or *tuple*).
  - To do this, you will need to estimate the error rate, in meters of travel per second of travel, for the speed given, then determine how long it should take to go the required distance, and finally multiply that by the error rate (in x and y, again).
  - You may be able to derive a mathematical function for the error rates using the curve-fitting funcationality that is built into the spreadsheet you are using.
  - If there is no mathematical function you can find, or if none seem to fit well, then your function should use *interpolation* based on the speeds you *do* have data for.

Exercise 5: Compensating for servo errors

Overview

In this exercise, you will write a Python function to use to point the servo, instead of using the built-in `servo()` function. You will also write a function to replace the `us_dist()` built-in function to get sonar data.

The reason to do this is twofold. First, it is very likely that `servo()` may not exactly point the servo where you think it should; it is also likely that `us_dist()` may not return correct values for the range (distance to an object). Your functions, then, will compensate for this.

Second, this will introduce you to the idea of writing interface functions for hardware-specific functions. You would ideally want code you write to control the GoPiGo to work for other robots, so that you wouldn't have to change much to control another kind of robot. However, both `servo()` and `us_dist()` are functions for the GoPiGo and will likely not be supplied by other robots. By writing interface functions that your programs can call, which themselves call the robot-supplied functions, you can isolate your programs from the particular functions that are robot-specific; to change robots, the only thing you would have to do would be to use a different set of interface functions.

Goals

- Gain experience writing interface functions while building useful servo and sonar function.
- Begin to learn how to compensate for hardware imprecision.

Resources

1. "Python Programming", from Dexter Industries' website.[23] This page provides very valuable information about the GoPiGo's API (application program interface, the set of functions you use to interface with the robot).
2. Ultrasonic Sensing (Rockwell Automation)[24]

Materials needed

You will need:

[23] www.dexterindustries.com/GoPiGo/programming/python-programming-for-the-raspberry-pi-gopigo
[24] www.ab.com/en/epub/catalogs/12772/6543185/12041221/12041229/print.html

- Your GoPiGo (with the sonar sensor attached)
- A smooth floor or table

Requirements

- Write a function that points the servo by in turn calling `servo()`, mapping the angles -90 deg to 90 deg to the robot's native 0 deg $-$ $-$ 180 deg range.
- Write a function that calls the native sonar (ultrasound) range function, but corrects the returned value as needed to give accurate ranges.

Programming notes

1. Write a Python function `point_servo()` that moves the servo.
   - It should take one parameter, θ, that specifies the direction the servo should point (in degrees), $-90 \leq \theta \leq 90$, where:
     - $\theta = 0$ means that the servo points straight ahead of the robot;
     - $\theta < 0$ means the servo points to the right of the robot; and
     - $\theta > 0$ means that the servo points to the left of the robot.
   - We use these conventions to agree with ones you used in the previous exercise, where straight ahead is the robot's x-axis and 90° to the left is the y-axis.
   - You'll need to play around with the `servo()` function to see where the servo actually points when called with different values. You'll need to map from the parameter `a` of `point_servo()` to an appropriate parameter for `servo()` that will move the servo as desired.
2. Write a function `sonar_range()` that returns the range (distance) in cm from the robot to the nearest object in front of the sonar.
   - Experiment with the sonar to ensure that the values it returns when `us_dist()` is called are accurate.
   - If they are, then `sonar_range()` will only need to call `us_dist(15)`.
   - Otherwise, you should have `sonar_range()` try to compensate for any errors in the sonar system.
   - Note that you may not be able to find a consistent pattern of errors; in that case, you can't really do much to compensate.
   - However, if you *do* find a pattern, you should adjust what the raw sonar function returns to be more accurate.

Testing

- Make sure that the servo behaves as specified by trying it with several different angles.
- Make sure that what is returned from the sonar interface function is correct by positioning an object at various distances from the robot and checking what `sonar_range()` returns.

Questions for thought

- If there were errors in how `servo()` behaved, what might have caused them?
- How does the ultrasound sensor find the distance to an object? (See, e.g., Resource 2.)

Stretch goals

- Experiment with objects of different shapes and materials; does the `range()` function return the same values for different objects when positioned at the same distance from the robot? If not, why not, do you think?
- Write a function `nearest_object()` that uses the servo and the sonar to scan the area in front of the robot (over the range -90° – 90°), point the servo back to 0°, and returns the servo to 0°.

Exercise 6: Compensating for motion errors

Overview

In this exercise, you will write Python interface functions that call the robot's own native movement functions to compensate for motion errors you discovered in your "real versus expected motion" experiment, done in an earlier exercise. Even if you did not discover any problems with the robot's movement in that exercise, you will still write these interface functions, since that will allow you to isolate the rest of your robot functions from any future changes needed to compensate for changes in how the robot moves.

Goals

- Gain more experience writing interface functions.
- Learn how to compensate for motion errors.
- Create useful interface functions.

Preparation and resources

- Make sure you have done Exercise 4 (Experiment: Real versus expected movement) prior to doing this exercise.
- Resource: "Python Programming", from Dexter Industries' website.[25] This page provides very valuable information about the GoPiGo's API (application program interface, the set of functions you use to interface with the robot).

Materials needed

You will need:

- Your GoPiGo.
- Data and analysis from Exercise 4 (Experiment: Real versus expected movement)
- A smooth floor or table
- A protractor.

[25] www.dexterindustries.com/GoPiGo/programming/python-programming-for-the-raspberry-pi-gopigo

Requirements

- Create the function `forward()` described below
- Create the functions `right()` and `left()` described below.

Programming notes

Write function `forward(distance,speed,wait)`

- This function will move the robot forward using the robot's native `fwd()` command, along with other functions as necessary.
- Parameters:
  - `distance` is the distance (in cm) to move, `speed` is the speed (1–255) at which to move, and `wait` is a *Boolean* (true/false) parameter telling the function whether (`True`) or not (`False`) to wait for motion to stop before returning.
  - Make `distance`, `speed`, and `wait` all optional parameters with sensible defaults.
  - If `distance` is not specified or if it is `None`, then move the robot forward without setting any distance at which to stop. (In this case, your function should probably behave as if `wait` = `False`.)
  - If `speed` is not set, then your function should not set the robot's speed; instead, leave it at whatever value other functions have set it.
  - If `wait` = `True`, then the function should not return until the robot has gone the requested distance.
- Overall style:
  - You will want to break the function up into manageable pieces, as you did in the motion experiment for the major function, then implement each piece as a separate function.
  - Don't forget to comment your code and to make variable and function names illustrative and helpful to the reader (who will likely be future you, when you are debugging the program!).

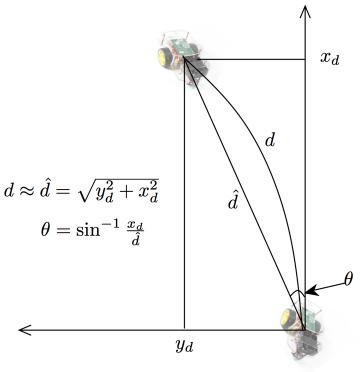
1. Managing distance:
   - You will use the `fwd()` GoPiGo function to move.
   - You will want to use the GoPiGo `enc_tgt()` function to set a target at which to stop – so don't forget to enable the encoders.
   - The target you set has to be specified in *encoder pulses*, of which there are 18 per wheel rotation. (But you should use the provided constant `PPR` from gopigo.py instead. Why do you think this is?)
   - The radius and circumference of the wheels are provided in gopigo.py as `WHEEL_RAD` and `WHEEL_CIRC`, respectively.
   - The first two parameters to `enc_tgt()` should both be 1; this will enable encoder targets for both wheels.
   - Setting the speed:
     - This can be done using both the `set_right_speed()` and `set_left_speed()` functions or just the `set_speed()` function.
     - If you found no errors in movement, then you will likely want to use the `set_speed()` function. Otherwise (see below) you will want to set the speeds of the two motors differently.
2. Waiting for the robot to reach its target:

- If the `wait` parameter is `False`, then immediately after initiating motion, your function will return; otherwise, it needs to wait until the robot has reached its target.
- Although the robot will stop at the target automatically, `fwd()` does not wait. This means that if your function needs to wait, it will have to periodically check if the robot has reached its target. This can be done in one of two ways:
  - Use the built-in function `read_enc_status()`, which returns 0 if the target has been reached, 1 otherwise.
  - Using the undocumented built-in function `enc_read(m)`, which returns the encoder value for motor `m` (0 for left, 1 for right). The function header in gopigo.py *claims* it returns the encoder distance in cm; to check this, call the function, move the robot a few cm, then call the function again. Do you think the value is in encoder pulses, cm, or something else?
  - You should have the function sleep a little bit between checking the encoders so as not to use too much of the CPU for no reason.
    * You can do this with the `sleep(s)` function (`from time import sleep`).
    * Try different values for `s` (in seconds) until you find one you feel doesn't wait too long (and cause the function to delay returning). Maybe start with a tenth of a second.

Compensating for motion errors:

- If you did not find any motor errors, then you do not need to do this—you lucked out!
- If you did find errors, then the problem is to get `forward()` to compensate for them so that the robot travels straight ahead.
- Let's define the error e_s for a given speed s to be the proportion of the desired distance traveled the robot deviated from the x-axis.
  - If d is the distance the robot traveled and y_d is the y-value when the robot was at its destination, then $e_s = y_d/d$.
  - Note that since both y_d and d are measured in centimeters, then e_s is dimensionless: it's just a ratio.
- If we know e_s for the speed desired, then we can predict where the robot would end up if we let it run for a given distance d': it's y-value would be $e_s \times d'$.
- We can compensate for this in one of two ways:
  1. Aim for a spot roughly $e_s \times d'$ away from the x-axis in the opposite direction of the expected error.
  2. Adjust the motor speeds to make one motor run Δs faster than s and one Δs slower so that the robot moves in a straight line.

1. Compensating by turning
   - If we aim away from the expected error, the question is, what angle do we turn to start?
   - Graphically, the situation looks like this:



- We will assume here that the error is small, so that $d \approx \hat{d}$. Thus, if we just turn θ degrees away from the expected error direction, the robot should end up about d along the x-axis.
- You will need the `asin()` (arc sine) function from the math module; so put this at the top of your file:

  ```
  from math import asin
  ```

- Keep in mind that θ is in radians, not degrees, and the turning functions use degrees, a conversion is in order.
- Since there are 2π radians in 360°, the angle in degrees $\theta_d = \frac{360}{2\pi}\theta$.

2. Compensating by adjusting wheel speeds.

   - If instead you want to compensate by changing the motor speeds, then it's a little more complicated—though also possibly a little more accurate.
   - We provide an appendix to this example that explains how we arrive at the formula to use to adjust the speeds, but for now, the formula we will to find the additional speed ω_{add} to add to the outer wheel is:

     $$\omega_{add} = \frac{2Wsy_d}{r_w(x_d^2 + y_d^2)}$$

     where:
     - s is the desired speed, in cm/s;
     - r_w is the radius of a wheel (3.25 cm, the `WHEEL_RAD` constant from the `gopigo` module);
     - W is the track (11.75 cm for the GoPiGo);
     - x_d is the distance the robot moved in the experiment along the x-axis for speed = ω; and
     - y_d is the distance the robot deviated from the x-axis (i.e., the y-value) in the experiment for speed ω.
   - Suppose that the call is `forward(distance, speed)`, w_i is the "inner" wheel, that is, the one on the side toward which the robot's direction deviated, and w_o is the "outer" wheel.
     - Set the speed for w_o to be \$$\omega$ = \$ `speed`. For the sake of argument, let's say that w_o is the right wheel; then:

       ```
       set_right_speed(speed)
       ```

     - Set the speed for w_i (the left wheel) to be `speed` + ω_{add}.
       * As you can see above, we need s, the speed in cm/s, that corresponds to `speed` (the motor speed), in order to calculate ω_{add}.
       * In the experiment, your program recorded the time t it took to go distance d. You can use this to find the speed, in cm/s: $s = d/t$.

    * Given that, you can do the following to set the left wheel's speed:
      ```
      W = 11.75
      rw = WHEEL_RAD
      s = d/t
      omega_add = (2 * W * y * s)/(rw * (x*x + y*y))
      set_left_speed(speed + omega_add)
      ```
      where `x` and `y` are x_d and y_d, respectively, described above, d is the distance from the trial, and t is the time it took to go d cm.

3. What if the motor speed wasn't one of the trials?
    - So far, we've figured out how to compensate for an error for a given speed.
    - If one of the experimental trials was for the desired speed given to `forward()`, then we're all set: find that speed in the data and compute e_s (for the first method) or x_d and y_d (for the second method) based on that data.
    - Most likely, however, the desired speed won't match one of the trials.
    - If you are lucky enough that the x- and y-values for the trials (averaged over the replications for each trial, say) can be predicted as a function of speed, you can use those values for the second method; you can determine e_s from those functions, too, for the first method.
    - If you aren't that lucky, then you will to *interpolate* to find the what you need (either x_d and y_d or e_s). If you don't remember how to interpolate from high school (assuming it was covered there), then search for "linear interpolation" on Wikipedia, for example, to see how to do this.

Write functions `left(angle)` and `right(angle)`

- In the best case, you can write these very easily: simply use the (undocumented) gopigo.py functions `turn_left(angle)` and `turn_right(angle)`.
- Try these functions for several different angles and measure the resulting angles turned using a protractor.
- If there were only minor errors, then you can just have your functions call the supplied ones.
- If there were errors, however, then see if you can figure out a pattern; if you can, compensate for the errors. The compensation doesn't need to be as extensive as for `forward()` – you can just try several angles, measure the results, and estimate the angle you would need to pass to `turn_left()` or `turn_right()` to get the correct turn.

Testing

Try your functions to make sure that the robot goes the distance expected, turns in the correct direction, etc.

Questions for thought

- There is a third option for compensating for wheel speed mismatches: just initially angle the robot away from the direction in which it curves. This will let it curve as before, but end up where it should be.
    - See if you can figure out, in general terms, how far to angle the robot.

  - Do you think this is a good way of doing the compensation? Why or why not? What if you are in a crowded environment? An empty environment?
- Can you see the utility of creating any other interface functions? If so, which ones, and why?

Stretch goals

- Instead of specifying the speed for `forward()` as being in the range of 0–255, allow it to be specified in cm/s.
  - This will require you to try running your robot at a variety of motor speed settings to determine the conversion from motor speed to actual speed, both going forward and backward.
  - It is likely that the battery status will impact this calculation. Can you figure out a way to compensate for this? Would it be worth it?
- Write a `backward()` function that does the same sort of thing as `forward()`.
- Provide optional `speed` and `wait` parameters to `right()` and `left()`. Speed should be specified in degrees/s (°/s), and `wait` will have the same meaning as for the `forward()` and `backward()` functions.

Appendix: Calculating wheel speeds

- In the following, we use ω to mean motor speed. ω usually means angular velocity (and this is what it means in the first formula), so for this to work, we have to assume that there is a linear relation between motor speed and angular velocity.
- The first formula that we need is modified from an answer to a question on math.stackexchange.com, a very useful site for technical questions, by user "DJohnM":[26]

$$t = \frac{W\theta}{r_w \omega_{add}}$$

  where:

  - t is the time the robot is moving;
  - W is the wheel track;
  - r_w is the wheel radius
  - θ is the angle the robot turns; and
  - ω_{add} is the angular velocity that needs to be added to turn the robot through θ.

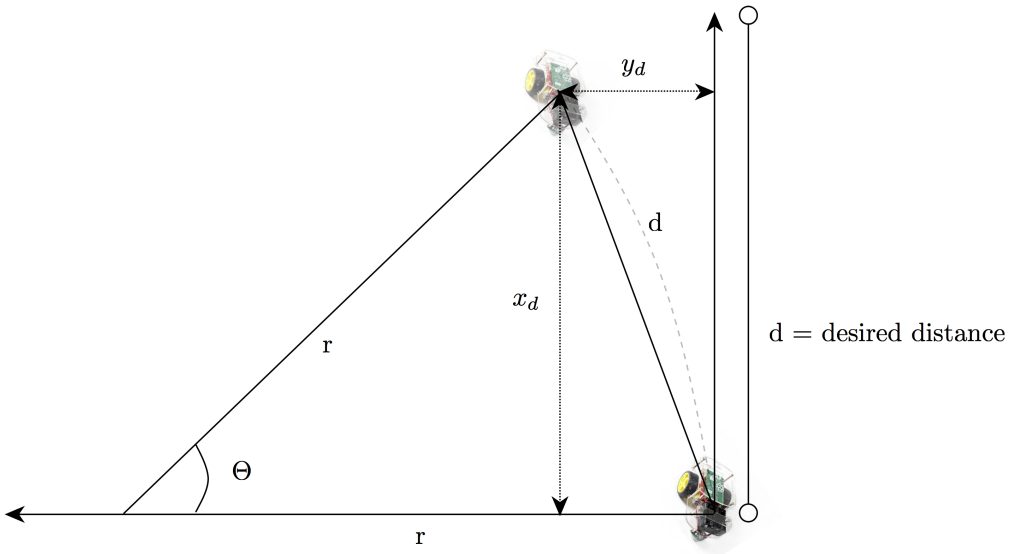
  This formula has more to do with how to turn a vehicle than how to correct one. However, if we add ω_{add} to the "slow" wheel, it should work for us as well.
- We can rearrange this equation to find ω_{add}:

$$\omega_{add} = \frac{W\theta}{t r_w}$$

- We know W and r_w; we "just" have to find θ and t.

[26] math.stackexchange.com/questions/519523/calculating-individual-wheel-velocities-from-a-desired-angle-in-a-differential-w

- Here is a diagram of one trial for reference:



- We assume that the robot traces an arc of a circle of radius r. The length of the arc is d, the distance the robot traveled during the trial.
- We know from geometry that $d = r\theta$, so $\theta = d/r$ – now we have to find r.
- From the Pythagorean Theorem, we know that:

$$r^2 = x_d^2 + (r - y_d)^2$$

- We can solve this for r:

$$\begin{aligned} r^2 &= x_d^2 + (r - y_d)^2 \\ &= x_d^2 + r^2 - 2ry_d + y_d^2 \\ r^2 - r^2 + 2ry_d &= x_d^2 + y_d^2 \\ r &= \frac{x_d^2 + y_d^2}{2y_d} \end{aligned}$$

- Thus:

$$\begin{aligned} \theta &= d/r \\ &= \frac{2dy_d}{x_d^2 + y_d^2} \end{aligned}$$

- What about t, the time?
  - For now, let's just let the speed of the robot for motor speed ω be s cm/s.
  - $t = d/s$

– Thus:

$$\begin{aligned}\omega_{add} &= \frac{2W\theta}{tr_w} \\ &= \frac{2Wdy_d}{tr_w(x_d^2+y_d^2)} \\ &= \frac{2Wdsy_d}{dr_w(x_d^2+y_d^2)} \\ &= \frac{2Wsy_d}{r_w(x_d^2+y_d^2)}\end{aligned}$$

Exercise 7: An approach or avoid behavior

Overview

Behavior-based robotics is an approach to robot control that relies on combining self-contained programs called *behaviors* in such a manner that the desired overall behavior of the robot emerges from their interaction. It has some advantages over trying to write a single program to control all of the robot's behavior, including:

- each behavior can be relatively small, and hence, easy to write, easy to debug, and fast;
- splitting complex behaviors into simpler ones allows a "separation of concerns", which lets the programmer optimize the behavior for its task;
- since the behaviors are defined by what they do rather than how they are built, different kinds of programs that work better for different purposes can be used in the same robot controller;
- because the behaviors are defined by what they do and are separate from each other, it is easy to replace a behavior with a new version or a different version, either to improve the controller or to try out a new approach;
- it is easy to add different behaviors;
- it isolates concerns about interactions between behaviors to the behavior controller, keeping them out of the individual behaviors;
- faster behaviors can potentially respond more rapidly than slower ones, allowing both reflexive and more computationally-intensive responses to coexist; and
- it separates out some behaviors that may be useful in other robots.

We will be exploring this kind of robot control in the next few exercises as well as in the first two-week project. In this exercise, you will write a rather complex, standalone behavior that will allow the robot to approach an obstacle, stop before hitting it, and run away if the obstacle approaches *it*. In the next exercise, you will take this behavior apart for use in a simple behavior-based controller.

Goals

- The practical goal of this assignment is to create a behavior that will keep the robot safe and that will also do some interesting things when approaching an obstacle or when an object is approaching the robot.
- The pedagogical goals are:
  - to gain more experience programming the robot, especially experience using a real-world sensor (the sonar);

  - to start thinking about how you can make programs more general-purpose for use with other programs; and
  - to begin learning about behavior-based control.

Resources

1. Behavior-based robotics– Wikipedia[27]
2. Behavior-based robotics – Chapter 3 of Wahde, M., *An introduction to autonomous robots*, lecture notes, (MW).[28]
3. For an example of using the ultrasonic sensor to stop when an object is in front of the robot, see the `basic_obstacle_avoid.py` file in the GoPiGo directory on the robot's desktop; look in:

   `GoPiGo/Software/Python/Examples/Ultrasonic_Basic_Obstacle_Avoider`

4. A sample project to build a "guard robot" is included in the GoPiGo software on your robot; to see a description of this, take a look at this web page: www.dexterindustries.com/projects/agent-kk-2

Materials needed

- Your GoPiGo (with the sonar [ultrasound] sensor attached).
- A smooth floor with sufficient space for the robot to run.
- The Python functions you wrote in previous exercises.

Requirements

- If an object is detected within some distance, say, `APPROACH_RANGE`, in front of the robot, move toward the object.
- If the robot is moving forward and an object is detected within some distance, say, `CLOSE`, slow down, with the robot's speed being inversely proportional to the distance (i.e., slower the closer the robot gets to the object).
- If the robot is moving forward and an object is detected within some distance, say `STOPPING_RANGE`, then the robot should stop.
- If the robot is stopped and an obstacle in front of it moves closer to it than `STOPPING_RANGE`, then it should back away until it is farther away than `CLOSE`, then stop. (If the obstacle keeps chasing the robot, then it should continue backing away.)
- Note that `APPROACH_RANGE` $>$ `CLOSE` $>$ `STOPPING_RANGE`.
- If the robot is dealing with an obstacle and the obstacle moves away, then the robot should move forward; the speed with which it moves should be based on the above requirements. For example, r is the obstacle's range (with r being some large value if the obstacle disappears entirely from view) and s is its initial speed, then if:
  - $r >$ `APPROACH_RANGE` $\Rightarrow$ resume speed s going forward
  - `CLOSE` $< r <$ `APPROACH_RANGE` $\Rightarrow$ follow the object with speed s

[27] `https://en.wikipedia.org/wiki/Behavior-based_robotics`
[28] `http://www.am.chalmers.se/~wolff/AA/Chapter3.pdf`

  - STOPPING_RANGE $< r <$ CLOSE $\Rightarrow$ follow the object, setting the speed based on s as above
  - $r <$ STOPPING_RANGE $\Rightarrow$ continue moving back from the object, as above

Programming notes

Write a function `test_behavior(speed)` that meets the requirements above

- `speed` is the initial forward speed to use, in cm/s.
- The `test_behavior()` function should loop, repeatedly calling another function you will write (the *behavior*), `approach_or_avoid(speed,range)`
  - `speed` is the base speed for the requirements above; it is set by `test_behavior(speed)` to whatever was passed to it.
  - `sonar_range` is the distance to the nearest object in front of the robot, in cm. `test_behavior()` should use the `sonar_range()` function you wrote in a previous exercise ("Compensating for motion errors") to get this value.
  - `approach_or_avoid()` will return a single value which we can call `new_speed`.
- Use the `forward(distance,speed,wait)` function you defined in a previous exercise ("Compensating for motion errors") to actually move the robot; recall:
  - if `distance` = `False`, then the robot will not stop after any particular distance;
  - `speed` sets the motor speeds to achieve a particular forward speed, in cm/s; and
  - if `wait` = `False`, then `forward()` returns immediately to the caller.
- `test_behavior()` will adjust the speed of the robot using `new_speed`:
  - if `new_speed` > 0, then call `forward()` with the new speed
  - if `new_speed` < 0, then have the robot go backward (using either the built-in function `bwd()` along with `set_speed`, or using the `backward()` function you *may* have written in the previous exercise)
  - if `new_speed` $= 0$, then stop (i.e., call `stop()`)
- You may want to use `sleep()` to wait for a small amount of time each time through the loop; however, `range()` may also be waiting sufficiently long, depending on you implemented it, so you will have to play around with the timing.
- Note that you will want `test_behavior` to call `approach_or_avoid()` with the same speed each time, regardless of what it returns as `new_speed`. This is the only way for it to know what speed it should be basing `new_speed` on. (If the `approach_or_avoid(` behavior function were to be used with other behaviors that might also adjust the speed, how would you keep track of the speed?)

Write the function `approach_or_avoid(speed,range)`

- `speed`, `range` are as specified above.
- This returns the desired speed, given `speed` and `range`, as described in the requirements, based on the constants defined for distances.

Define the constants

- Define the constants `APPROACH_RANGE`, `CLOSE`, and `STOPPING_RANGE`, as described in the requirements.
- You can pick values for these that make sense to you.

Testing

1. Turning on the GoPiGo and placing it on the floor.
2. Start Python and load your file(s).
3. Put an object some distance in front of the robot on the floor.
4. Using VNC or ssh to connect to the robot and start Python.
5. Load the file containing your functions (e.g., `from xxxx import *`, if `xxxx.py` your file).
6. Run your test function: `test_behavior(s)`, where `s` is an initial speed in cm/s.
7. Now test your behavior by observing what the GoPiGo does. It should go forward until there is an obstacle, begin slowing down, and stop some distance away.
8. Now remove the obstacle. What does your robot do? It should start up again.
9. Put the obstacle in back in front of the robot. When the robot stops, slide the obstacle further from it and observe what happens.
10. Slowly move the obstacle toward the robot. It should back up in response, stopping when you stop moving the object toward it.
11. To stop the behavior, since `test_behavior()` is implemented as an infinite loop, you will have to interrupt Python (using control-C).

Questions for thought

In your blog, address these questions.

- Did the behavior work as you anticipated? If not, how did it differ from what you expected? Can you explain?
- Is this a good way to do obstacle avoidance? In particular, is symmetrically reducing motor speed, then stopping, the best way of obstacle avoidance?
  - What if we want to go from point A to point B and there is an object in the way—what will happen? Will we ever arrive at point B?
  - What would be some other ways that might work better avoiding an obstacle when there is some target in mind?
  - Do you think that modifying the behavior to do these better ways is what should be done? Or should it be split into *two* (or more) behaviors, one for when the robot is at rest or just looking for objects, and one when it is trying to get to some location?
- How hard would it be to modify your behavior so that it could work with other behaviors when there is *behavior controller* that loops forever, each time:
  - Reading the sensors.
  - Calling each defined behavior with a list of the sensor values (called the *percept string*).
  - Collecting the return values of each behavior in a list. Each return value would be some description of what action the behavior wants to do next; for example, the behavior in this exercise would want to "set motor speed to s".
  - Combine the various desired actions and issue GoPiGo commands to carry out the result. What would *you* need to do? I.e., is there anything about the way you wrote your behavior that would need to change? If so, what information would *we* (or whomever designs the controller) need to give to you in order for you to modify your behavior to work this way?

Stretch goals

- Modify the behavior to avoid obstacles in the way(s) you thought about above as better ways to do it. For example:
  - Maybe think about changing only *one* of the motor's speed.
  - Maybe think about stopping and turning using `right()` or `left()`.
- Write two versions of this behavior, one for when the robot is stationary and one for when it is moving, as discussed above. Write a simple control loop to choose one or the other to get control.

Exercise 8: Separating concerns

Overview

In this exercise, you will revise (*refactor*) the work you did in the previous one to split the `approach_or_avoid()` function into pieces, each of which deals with only one set of concerns: stopping, approaching, avoiding, etc.

Usually, it is a good idea when creating a program to identify related issues, called *concerns*, and address them together. For example, if you are creating a program to print financial reports, there are at least two sets of concerns: those having to do with the financial calculations, and those having to do with formatting and printing the results. (In fact, there may be more: inputting the data, formatting and printing as two separate sets of concerns, etc.)

By identifying and separating the concerns, you can isolate all the code having to do with the same data and I/O together, so that if the needs change, only those functions have to be changed. To extend the financial report-writer example, if the formatting and printing were scattered among the calculations, then if the format or the printer to use changed, the programmer would have to chase down all the places affected, rather than just changing one function.

Sometimes, however, you may not be able to know all the related concerns until after a first draft of the code has been written. In this case, a process of *refactoring* is done, which really is just shuffling parts of the code around and revising them to separate the newly-discovered concerns.

This is what you will be doing in this exercise: refactoring your `approach_or_avoid()` behavior into several behaviors based on separating related concerns.

Goals

The practical goal of this assignment is to end up with several behaviors that will keep the robot safe and that will also do some interesting things when approaching an obstacle or when an object is approaching the robot. The pedagogical goals are to introduce you to the idea of separating concerns, refactoring, give you more experience programming the robot, especially experience using a real-world sensor (the sonar), to get you started thinking about how you can make programs more general-purpose for use with other programs, and to begin learning about behavior-based control.

Resources

- Separation of concerns[29] – Wikipedia

[29] `https://en.wikipedia.org/wiki/Separation_of_concerns`

Materials needed

You will need:

- Your GoPiGo (with the sonar [ultrasound] sensor attached)
- A smooth floor with sufficient space for the robot to run.
- Your `test_behavior()`, `approach_or_avoid()`, and related functions from the previous exercise.

Requirements

Requirements from the previous exercise:

- If an object is detected within some distance, say, `APPROACH_RANGE`, in front of the robot, move toward the object.
- If the robot is moving forward and an object is detected within some distance, say, `CLOSE`, slow down, with the robot's speed being inversely proportional to the distance (i.e., slower the closer the robot gets to the object).
- If the robot is moving forward and an object is detected within some distance, say `STOPPING_RANGE`, then the robot should stop.
- If the robot is stopped and an obstacle in front of it moves closer to it than `STOPPING_RANGE`, then it should back away until it is farther away than `CLOSE`, then stop. (If the obstacle keeps chasing the robot, then it should continue backing away.)
- `APPROACH_RANGE` > `CLOSE` > `STOPPING_RANGE`
- If the robot is dealing with an obstacle and the obstacle moves away, then the robot should move forward; the speed with which it moves should be based on the above requirements. For example, r is the obstacle's range (with r being some large value if the obstacle disappears entirely from view) and s is its initial speed, then if:
  - $r >$ `APPROACH_RANGE` $\Rightarrow$ resume speed s going forward
  - `CLOSE` $< r <$ `APPROACH_RANGE` $\Rightarrow$ follow the object with speed s
  - `STOPPING_RANGE` $< r <$ `CLOSE` $\Rightarrow$ follow the object, setting the speed based on s as above
  - $r <$ `STOPPING_RANGE` $\Rightarrow$ continue moving back from the object, as above

Requirements new to this exercise:

- Identify and make a list of related concerns in the requirements above (and in your old `approach_or_avoid()` behavior).
- Refactor your `approach_or_avoid()` function to separate these concerns by creating new behaviors to address them.

Programming notes

Create multiple behaviors:

- Create multiple behaviors to replace `approach_or_avoid()`.
- Identify for each the parameters they should receive.
- Each should return a desired speed, in cm/s, for the robot.

Modify `test_behavior(speed)` to call the new behaviors

- Instead of calling a single behavior, now your function will call several.
- Each will return a desired speed or (new) `None`, which means the behavior does *not* want to change the speed.
- If all desired speeds are the same, there is no problem. However, this is unlikely.
- You must determine how best to deal with the different desired speeds. There are many possible ways you could think about to do this, including:
  - use the maximum of all the ones returned
  - use the minimum
  - use the average
  - use only one, selected randomly
  - use only one, but selected in some other way: the first one that doesn't return None, etc.

Testing

1. Turn on the GoPiGo and place it on the floor.
2. Load Python and your file(s).
3. Put an object some distance in front of the robot on the floor.
4. Use VNC or ssh to connect to the robot and start Python.
5. Load the file containing your functions (e.g., `from xxxx import *`, if `xxxx.py` your file).
6. Run your test function: `test_behavior(s)`, where `s` is an initial speed in cm/s.
7. Now test your behavior by observing what the GoPiGo does. It should go forward until there is an obstacle, begin slowing down, and stop some distance away.
8. Now remove the obstacle. What does your robot do? It should start up again.
9. Put the obstacle in back in front of the robot. When the robot stops, slide the obstacle further from it and observe what happens.
10. Slowly move the obstacle toward the robot. It should back up in response, stopping when you stop moving the object toward it.
11. To stop the behavior, since `test_behavior()` is implemented as an infinite loop, you will have to interrupt Python (using control-C).

Questions for thought

1. What did you have to do to refactor your behavior?
2. What control mechanisms did you try and why? What worked? What didn't work? Do you have any other ideas for control mechanisms?
3. How did the refactored behaviors compare to the original in terms of performance?

Stretch goal

- Implement multiple control mechanisms and compare them under different conditions.

o#+LATEX:

Exercise 9: An object-oriented interface to the robot

Overview

In this exercise, you will learn to use an object-oriented interface to the GoPiGo that you will use in the next exercise and in Project 1.

Goals

- Begin learning about *object-oriented programming* (OOP).
- Learn a little about software interfaces to hardware, and how OOP can help by allowing new interfaces to *inherit* functionality from existing ones.

Background and resources

- An *object* is a collection of variables and functions (called *methods*) associated with those variables that represent some object in the world of the program.
- For example, a `Student` class might be implemented as:

```
class CSStudent(Student):
    major = "COS"
    advisor = None

    def __init__(self,name):
        self.name = name

    def greet(self):
        print("Hello!  I'm", self.name, "and I say COS is fun!")
```

- A *class* defines a class of objects. Objects are created from the class by *instantiating* them, that is, making an *instance* of the class, e.g.:

```
me = Student("Joe", "COS")
```

  The variable `me` now contains a `Student` object.
- The variables of a class/object are called *instance variables*, and can be accessed by prepending the object or class to the variable name. For example:

```
print(me.name)
```

would print "Joe". We see other examples of that in the above definition, in that case using the parameter `self`.

- Methods are called the same way. Thus, to have the student greet us, we would do:

```
me.greet()
```

which would print "Hello! My name is Joe and my major is COS ."

- Note that `greet()` takes no arguments. This is because the object contained in `me` is automatically given to the method as the first parameter, `self`.
- We can create classes that represent particular kinds of students by defining *subclasses* of `Student`, for example:

```
class CSStudent(Student):
  major = "COS"
  advisor = None

  def greet(self):
     print("Hello!  I'm", self.name, "and I say COS is fun!")
```

- Here, the subclass' `major` instance variable value overrides that in the parent `Student` class, as does its `greet()` method. Here is an example:

```
a = CSStudent("Sally")
a.greet()
```

would print "Hello! I'm Sally and I say COS is fun!" The Sally object's `major` would be "COS".

Resources:

1. Object-oriented programming Wikipedia article[30]
2. Python Object Oriented[31] – tutorial

Materials needed

You will need:

- Your GoPiGo (with the sonar [ultrasound] sensor attached) and some way (e.g., a computer) to communicate with it
- A smooth floor with sufficient space for the robot to run.
- Past programs you have written for this module.
- Modules provided by the instructor/CLA:
  1. `utilities` – contains some useful utility functions
  2. `robot` – a generic OO robot interface
  3. `gopigo_interface` – an OO interface to the GoPiGo

[30] `http://www.wikiwand.com/en/Object-oriented_programming`
[31] `https://www.tutorialspoint.com/python/python_classes_objects.htm`

The `robot` and `gopigo_interface` modules

- The `robot` module contains the `Robot` class, a class for generic robots.
- The `gopigo_interface` module contains the `GoPiGo` class, which is specific for your robot.
- The `Robot` class contains instance variables and methods that should be useful for all robots, while the `GoPiGo` class contains those specific to the GoPiGo. Look through the file to get a sense of what each class contains.
- You will create an instance of the `GoPiGo` class to control your robot.
  - This class has functions (methods) that create a *percept* dictionary that contains all sensory data for use by you or your controlling program.
  - The class also has methods that accept either individual commands (e.g., to go forward) or a command dictionary that contains a set of commands to carry out simultaneously.

Percepts

- So far, we have dealt with sensed values, also called *percepts* (e.g., sonar range, wheel encoder values, etc.), on a piecemeal basis as needed by your functions.
- Now, however, we are moving toward more generic functions—your behaviors—that we would like to all take the same input and provide output in a common form.
- This way, new behaviors can be added at any time, and the controller will not need to know anything about the internals of your behaviors, including what sensor data they need.
- Consequently, we need to develop a standard way to represent all sensor data that all of the behaviors can understand.
- A *percept* is the collection of all perceptual data provided by the robot's sensors, plus some data that may be calculated from it (e.g., velocity).
- In the case of this exercise, the percept will be a Python dictionary whose keys are the names (as strings) of the sensory data and the values are the actual data values.
- For example, the distance to the nearest object in front of the ultrasound sensor might have the key '`sonar-range`' and a value of 20 (cm).
- The percept, let's call it `percept`, will contain both things that directly correspond to the robot's sensors as well as other things that are obtained elsewhere or calculated, for example, the time the percept was created, the robot's velocity, and others—these latter kinds of information are often thought of as the output of *virtual sensors*.
- The `GoPiGo` class returns a percept when its `percept()` method is called, e.g.:

```
robot = GoPiGo()
percept = robot.percept()
```

- The percept has at least the following keys and their values:
  - `current_time` – the time the percept was created
  - `encoders` – a tuple containing the wheel encoder values (left, right)
  - `motor_speeds` – a tuple containing the motor speeds (left, right)
  - `sonar_range` – the range to the nearest object in front of the sonar (in cm)
  - `servo_angle` – servo angle, from -90 (all the way to the right) to +90 (all the way to the right; 0 = straight ahead
  - `odometer` – the distance, in cm, the robot has traveled since it was turned on (calculated from the averaged encoder values)

  - `cumulative_turn` – like an odometer for heading; how far, in degrees, the robot has turned since it was turned on (*not* reduced to 0–360; calculated from turn commands to the robot)
  - `speed` – the speed in cm/s (calculated from odometer value change since the last percept divided by the time elapsed)
  - `acceleration` – the acceleration (foward/backward) of the robot in cm/s^2 (calculated by the velocity change divided by the elapsed time since the last percept)
  - `voltage` – the voltage of the battery pack, a (very) rough indicator of charge remaining
  - `target_reached` – a Boolean (True/False) value, true if the robot has reached a target encoder value that was set
- To get a value from the percept, e.g., for `speed`:

```
speed = percept['speed']
```

Commands

- `GoPiGo` has methods to carry out commands.
  - Commands methods can be called directly, for example:

    ```
    robot.set_speed(5)
    ```

    to set the speed of the robot to 5 cm/s.
  - Commands can also be given to the robot by calling the `command(name, value)` method, e.g.:

    ```
    robot.command('set_speed', 5)
    ```
  - Finally, multiple commands can be given to the robot simultaneously by calling the `commands(cmd)` method, where `cmd` is a dictionary of command name/value pairs, e.g.:

    ```
    commands_to_do = {'set_speed': 5, 'forward': 20}
    robot.commands(commands_to_do)
    ```

    This would tell the robot to set the speed to 5 cm/s and to go forward 20 cm.[32]
- The commands available are:
  - `set_motor_speeds` – value is a tuple (left, right)
  - `set_speed` – value is either a number or a tuple.
    * If the value is a number, then this is the the speed both motors will be set. The speed is in the range of 0–255, with no correspondence attempted to cm/s or any other speed measurement.
    * If the value is a tuple, then the first value is the speed (0–255), and the second is the compensation factor.
    * The compensation factor can be calculated from what you have done previously; indeed, this is basically a refactoring of your `forward()` interface function written previously, but instead of setting the speeds and going forward, you are now converting that into a differential between the wheel speeds.
    * For example, if for a raw speed of 200 you would set the left motor speed to 210 and the right to 180, then the compensation factor would be:

$$(210 - 180)/200 = 30/200 = 0.15$$

[32]Note that the order in which GoPiGo carries out the commands is not defined; the idea is that they will be done simultaneously, not one after the other.

    - Had the left needed to be slower than the right and the numbers reversed, then the compensation factor would be -0.15.
    - This will be applied to the base speed to adjust the two motors. If the base speed is b and the factor is f, the left motor will be $(1 + f)b$ and the right will be $(1 - f)b$.
  - `servo` – value is an angle (-90 – 90) to which to turn the servo
  - `change_servo` – move the servo left (positive) or right (negative) *value* degrees
  - `forward` – value is how far (in cm) to go, if value is `None`, then just go forward forever
  - `backward` – ditto, but for reverse
  - `turn_left` – value is how many degrees to turn left
  - `turn` – value is the degrees to turn, with 0 being straight ahead and positive angles meaning "turn left"
  - `turn_rigth` – value is how many degrees to turn right
  - `left_led` – value is `True` (on) or `False` (off)
  - `right_led` – value is `True` (on) or `False` (off)
  - `set_target` – set a target distance at which to stop; the first target is an amount, the second (optional) argument is `True` for centimeters, `False` for encoder pulses
  - `deactivate_target` – turns off distance target, if one is set; no parameters
  - `stop` – stop immediately; no parameters
- If the names of these commands are used to create a dictionary to use as a parameter to the `commands` method, then if any of the keys has the value `None`, nothing will be done for that command. (This will be useful in the next exercise.)

Requirements

- Instantiate the `GoPiGo` class and get several percepts from the robot as it is placed at different locations.
  - Make sure you understand how the `percept` method works and what it returns.
  - Make sure you can access individual values from the `percept`
- Use the class' methods to control the robot.
  - Try out all the commands.
  - Try some directly (e.g., `robot.set_speed(5)`).
  - Try some using the `command` method.
  - Try doing several simultaneously using the `commands` method and a command dictionary.
- Create your own class that is a subclass of `GoPiGo`:
  - This class' methods should implement your interface functions from prior exercises, e.g., `forward()`.
  - You will also want to override some other methods, too; for example, change `set_speed` to accept cm/s rather than the motors' raw speed (0–255).
  - Test your new methods, for example:

    ```
    robot = MyGoPiGo()
    robot.forward([10,200,False])
    ```

    where your `forward(distance,speed,wait)` function from the motion compensation exercise has been reimplemented as the method `forward(parameters)` of class `MyGoPiGo`. (You don't have to call your class that, by the way.)
    - Note that the method can only take one parameter, since we allow calling it as part of a command dictionary.
    - Thus, the parameters that your old function had are now combined into a single list parameter.

Programming notes

- Create a subclass of `GoPiGo` in a file, say `my_robot.py`.
  - Don't forget to include the line: `from gopigo_interface import *`
  - To create the new subclass, use the `class` Python keyword.
    - In the Python line:

      `class Foo(Bar):`

      a new class, `Foo` is created that is a subclass of the existing class `Bar`. Note that the convention is to capitalize class names.
    - Choose a name for your class, and have it inherit (be a subclass of) `GoPiGo`.
- Define any instance variables you your new class needs.
  - If your old interface functions used global variables, then you should make these instance variables.
  - For example, if your function needed a `default_speed` variable, then part of your class definition, assuming your class name is "MyGoPiGo", would look something like:

    ```
    class MyGoPiGo(GoPiGo):
       default_speed = 200
    ```

  - Note that in methods of your class, you would refer to this as `self.default_speed`.
- Convert your interface functions to methods:
  - If your function has more than one parameter—e.g., `forward(distance, speed, wait)`—then replace them with a list—e.g., `forward(parameters)`, where `parameters` would be given a list such as `[distance, speed, wait]`.
  - This means that anywhere you access, say, `distance`, in your function, your method will need to access `self.parameters[0]`.
- Your methods should not access the raw robot functions (from `gopigo.py`) unless there is no corresponding `Robot` or `GoPiGo` method.
  - So for example, to move the robot backward (if you for some reason needed to) in a method that was *not* `backward()`, you would call `GoPiGo`'s `backward()` method, e.g,:

    `self.backward(5)`

  - However, inside `forward()` you would call the raw `fwd()` method, since you are defining the robot interfaces corresponding function:

    `fwd()`
- Don't forget to add override `GoPiGo`'s `sonar_range` method with one of your own that compensates for the sonar errors, i.e., the function you created in a previous example.

Testing

1. Turn on the GoPiGo.
2. Start Python on the GoPiGo.
3. Import your robot interface: `from my_robot import *`
4. Do the testing specified in the Requirements section.

Questions for thought

1. How hard was it to refactor your existing interface functions to fit the object-oriented methodology of the `GoPiGo` interface?
2. Was the effort worth it? What benefits do you see from using an OO interface? What drawbacks?

Stretch goals

- Implement additional "virtual sensors" or "virtual actuator commands". For example, you could write a "follow an arc" command, that will have the robot follow a curving trajectory, or a "picture" sensor that would return an image from the camera (using, e.g., the `picamera` Python module) as part of the percept.

Exercise 10: Using the camera

Overview

In this exercise, you will learn to use the GoPiGo's camera to take pictures.

Goals

- Learn to use the camera using the `picamera` module.
- Practice using the servo and ultrasound.
- Practice Python programming.

Background and preparation

- The `picamera` package documentation, available at picamera.readthedocs.io.
- The `PIL` (Python Imaging Library) documentation, available at effbot.org/imagingbook/image.htm (for the Imaging module of PIL, which is what we'll be using).

Materials needed

You will need:

- Your GoPiGo with the camera attached.
- Something interesting to take a picture of.
- A behavior-based controller from Project 1. (Since your group will be newly-created, you will likely have two different controllers, one from each prior group; use either or both.)

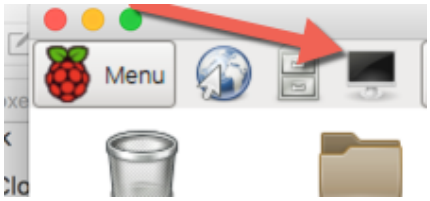
Requirements

- Install the Imagemagick bundle of Linux programs, if it is not already installed.
- Take a picture with the camera using picamera and display it on the screen using PIL's Image module.
- Write a `photograph_object` behavior and add it to your behavior-based controller; this behavior should:
  - do nothing (i.e., return no commands) if the robot is greater than some distance away from any object, say `PHOTO_DISTANCE`. (You can set this to whatever you like.)
  - take a picture when there is an object within `PHOTO_DISTANCE` of the robot
  - display the image using the Image module
- Test your behaviors and controller.

Programming notes

Install ImageMagick if not installed

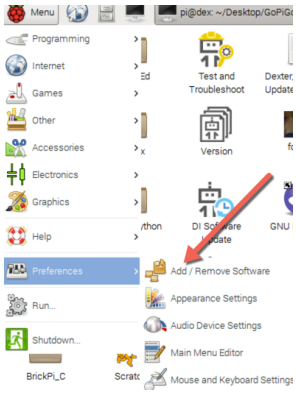
1. Check to see if ImageMagick is installed on your GoPigo
   - Open a terminal window using either the menu on Raspbian or the terminal icon, shown below with an arrow pointing toward it:



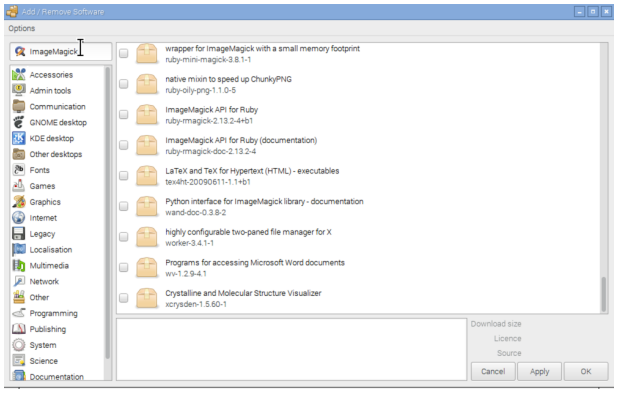
   - Type the following in this window, then hit return:

     ```
     display Desktop/GoPiGo/GoPiGo_Chassis-300.jpg
     ```

   - If a picture of a GoPiGo robot appears, ImageMagick is installed and you can skip step 2.
2. If an error message was displayed (e.g., `display: command not found`) when you typed the line in the terminal window, then you have to install ImageMagick.
   - From the menu, select "Preferences", then "Add/Remove Software":



   - A window will appear that looks like:



   - In the search box in the upper left of that window, type `ImageMagick` and press return.
   - It will search for the ImageMagick bundle of programs on the web, then display a lot of choices. The correct ones *should* be checked already.
   - Click the `Apply` button.
   - It will likely ask you for your password; enter it.

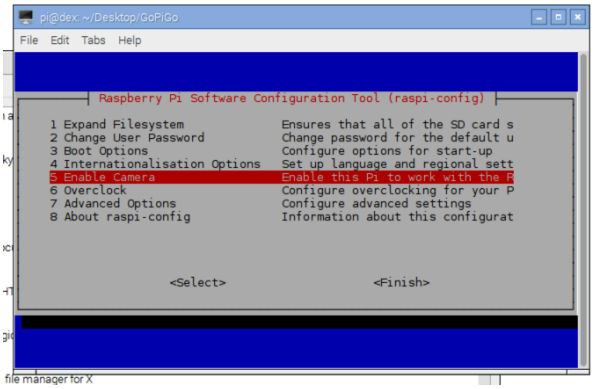
- It will then download and install the desired program.
- When it is done, repeat step 1. above to see if you can now display the file.
  - If so: you're done!
  - If not: ask the CLA or instructor for help.

Taking a picture

- Turn on the robot.
- Enable the camera:
  - Open a terminal window (see above).
  - In the terminal, type:

    ```
    sudo raspi-config
    ```

    This will bring up a menu to allow you to configure the Raspberry Pi and its associated hardware. The `sudo` part is a command that is used to run commands as the "superuser", and is not something you need be concerned about here.
  - Using the arrow keys, select the line to enable the camera:



    - Press return; it will display another screen, asking if you really want to enable the camera; press return again.
    - At this point, you can use the mouse and click on the little "x" in the upper lefthand corner of the window to exit the program – the camera is enabled.
- Take a picture:
  - For this, we'll use a "basic recipe" from the picamera documentation (referenced above).
  - Start Python.
  - Type in the following (without the comments) or type into a file and then import the file:

    ```
    from time import sleep
    from picamera import PiCamera

    file = "/home/pi/Desktop/picture.jpg"

    camera = PiCamera()                          # the 'camera' object
                                                 # now represents the robot's camera
    camera.resolution = (1024,768)
    camera.start_preview()                       # starts the camera warming up
    ```

```
# wait for it to warm up:
sleep(2)

# actually take the picture
camera.capture(filename)
camera.close()                                  # to turn off the camera
```

  - There should now be a picture on the desktop called "picture.jpg".
    * Double-click on it to view it, or...
    * ...from within Python, do:

      ```
      import os
      os.system('display /home/pi/Desktop/picture.jpg')
      ```

      If you are curious, this tells the operating system (i.e., Raspbian) to display the file using one of the ImageMagick commands you may have installed above.
    * ...or:

      ```
      from PIL import Image
      image = Image.open('/home/pi/Desktop/picture.jpg')
      image.show()
      ```

      This uses the Python Image Library to open and display (using one of ImageMagick's programs) the picture.

The `photograph` behavior

- Define a distance at which you will start taking pictures of an object; call it, for example, `PHOTO_DISTANCE`.
- Using your past behaviors as a template, create a new behavior called `photograph(percept)` that has one parameter, a percept dictionary (as per a previous exercise) and returns a command dictionary (ditto).
- The only thing this will need from `percept` is the range to the nearest object:

  ```
  range = percept['sonar_range']
  ```

- If the range is $>$ `PHOTO_DISTANCE`, then just return an empty dictionary or one with all commands set to `None`, e.g.:

  ```
  return {}
  ```

- If range is $\leq$ `PHOTO_DISTANCE`, then take a picture and display it on the desktop.
  - Use the Python code you used previously to take a picture as a starting point.
  - You may want to move the `camera = PiCamera()` and `camera.start_preview()` lines, as well as the call to `sleep()` out of the behavior and into the controller itself.
    * To do this, modify the `__init__()` method of your robot class to add these lines.
    * This way, when you start the robot, the camera will always be available and warmed up.
  - You can use either way we discussed above to display the image from within Python.
- If this is all you did, then you would notice that there would be, over time, more and more windows opening with images in them.
  - This would use up the Raspberry Pi's resources pretty rapidly, so we need to do something to avoid that.

- The thing we are going to do is something known as a *kludge*: a workaround or inelegant fix for a problem.
- Make sure that your program does: `import os`. This loads the operating system interface package.
- We are going to use the operating system to kill any open image windows when the behavior is about to take another picture.
- When distance $\leq$ `PHOTO_DISTANCE`, do the following before taking a picture:

  ```
  os.system('killall display')
  ```

  This tells the operating system to kill any processes that are running that have "display" as part of their name; this will include any images your behavior has displayed.

Testing

1. Turn on the GoPiGo and place it on the floor.
2. Start Python and load your file(s).
3. Put some objects on the floor.
4. Run the simple controller with all your old behaviors + `photograph`.
5. Let the robot run to see if it takes pictures when it gets close to objects.

Questions for thought

1. If the robot was truly autonomous, would you want it to display pictures?
2. Can you think of any other ways to get a picture from the GoPiGo, for example, if you didn't have VNC open?
3. What improvements would you suggest for the `photograph()` behavior?

Stretch goals

- Find another way to delete the open windows.
- Find another way to display the pictures.
- Look through the picamera documentation to see if you could somehow send the photographs over the network to a web client (e.g., Firefox, Chrome, etc.).
- Take a video of your robot's travels using the camera.

Exercise 11: Following an object

Overview

In this exercise, you will write a behavior that will look for a red object using the camera.

Goals

- Learn a little bit about dealing with images in Python.
- Continue learning about behavior-based robotics.
- Build a useful behavior.
- Practice Python programming.

Background and resources

- In this exercise, you will use the camera to find a "blob" of a particular color (red, in this case) and to follow it.
- The `blob_finder.py` package (from the module website) will be used to find the color.
- The function you will use from this package is `find_blob()`.
- This uses a very simple method, modified from that found on the "Physical computing with Raspberry Pi" website,[33] to find the center of a blob of red in an image captured by the GoPiGo's camera:
  - Using the Python Image Library (PIL), the image is opened and assigned to a variable.
  - Each pixel in the image is examined in turn to see if it is red.
    - PIL represents the image as a rectangular grid, or array, of pixels.
    - Each pixel is a tuple of (r,g,b), where the values are intensities of red, green, and blue (respectively), each in the range 0–255.
    - If the red value is sufficiently higher than the blue and green values, then the function considers the pixel to be red.
    - The threshold for "redness" in the function is 100, i.e., if it is 100/255 redder than it is green or blue.
  - If a pixel is red, then its x and y values are added to two variables that are initially both 0. These "accumulate" the x and y values of all red pixels.
  - When all pixels have been examined, then each of the variables is divided by the number of red pixels found to give a mean x and y value for the center of the red area(s).
- Note that this method is simplistic! For example, if there are two red objects of equal size in the picture, then it will "find" a red object midway between them.

[33] www.cl.cam.ac.uk/projects/raspberrypi/tutorials/robot/image$_{\text{processing}}$

Resources

1. The picamera package documentation, available at picamera.readthedocs.io.
2. The PIL (Python Imaging Library) documentation, available at effbot.org/imagingbook/image.htm (for the Imaging module of PIL, which is what we'll be using).
3. Information about basic image processing from the "Physical computing with Raspberry Pi" website[33]

Materials needed

You will need:

- Your GoPiGo (with the camera and sonar attached).
- A bright red object.
- Your behavior-based controller and behaviors from Project 1. (Since your group will be newly-created, you will likely have two different controllers, one from each prior group; use either or both.)
- The `blob_finder.py` file (from the module website).

Requirements

- A behavior that will use the servo to scan for a red object, then orient to (turn toward) that object.
- This behavior should subsume `wander()`, but not the avoidance behaviors.
- Test your behaviors and controller.

Programming notes

- Use the `blob_finder.py` package:

  ```
  from blob_finder import find_blob
  ```

  - The `find_blob()` function takes one positional argument and several others that are optional:
    * The positional argument is the name of the image file you got from the camera:

      `find_blob("foo.jpg")`

      would find a red blob in the file `foo.jpg`.
    * The second argument is the threshold for redness. It defaults to 100; you can set it to something else if you like either by doing (e.g.):

      `find_blob("foo.jpg",200)`

      or

      `find_blob("foo.jpg",red_threshold=200)`
    * The third argument tells the function whether or not to display the image (default is `True`). If you do not want to display the image, you can give this the value `False`, e.g.:

      `find_blob("foo.jpg", show=False)`

which would not show the image (and notice that we left the `red_threshold` alone here by using the keyword form of argument for `show`).
    - The fourth argument tells the function how long you would like the image displayed, if `show` is `True`; the default is 2 seconds, but you could set it to a different value, e.g.,

      ```
      find_blob("foo.jpg", show_for=5)
      ```
    - A final argument, `red_pixels`, is another threshold, this one for how many red pixels there must be in the entire image for a blob to be considered found. This is initially set to 200, and can be set by you:

      ```
      find_blob("foo.jpg", red_pixels=1000)
      ```
  - If a blob is found, `find_blob` returns a tuple, `(x,y)`, that gives the center of the red blob. (It also draws a small red cross at the center, turns the red pixels white, and dims the rest of the image, if the image is displayed.)
  - If no blob is found, then `None` is returned.
- Build the behavior, `find_object()`
  - Use the behaviors you built previously as templates.
  - Use what you learned in the previous exercise about using the camera to take a picture each time the behavior is called by the controller, or rely on the `photograph()` behavior to take the picture for you, whichever works for you.
  - Use `find_blob()` as described above to look for a red blob in the image.
  - If none is found, then this behavior should not suggest any actions.
  - If a blob is found, then this behavior should turn the robot to point to it.
- To point toward the blob, you can either:
  - Determine mathematically how far to turn and use `robot.py`'s `turn_left()`, `turn_right()`, or `turn()` commands; or
  - Turn a little bit toward the center of the blob; each time the behavior is called, it will turn a little more toward it.
    - You can use the blob's center to tell which way to turn—if it's to the left of the image's center, then (assuming the servo is pointing to 0 degrees!) you need to turn left, otherwise right.
    - Experiment with how much to turn each time.
    - You could use a fixed angle, say 10°.
    - You could also turn more the further the blob is from the center of the picture.
    - You won't need to worry about the y-value for this, of course.

Testing

1. Turn on the GoPiGo and place it on the floor.
2. Start Python and load your file(s).
3. Let the robot wander for a bit.
4. Place the red object somewhere on the floor, and see if the robot will find it and turn toward it.
5. Try this both with the simple controller and the blended controller from your prior work.

Questions for thought

1. How well did the new behavior integrate with others? Did it significantly slow down your robot's behavior?

2. Would you recommend running with this behavior enabled? Can you think of a way to speed things up, say by only running the behavior occasionally—and, if so, in general how would you do this?
3. Were there any differences between the two controllers (simple and blended) when this behavior is added to the robot?
4. Did you notice any problems with how the robot found or didn't find objects? Was it accurate?
5. What improvements would you suggest?

Stretch goals

- Copy `blob_finder.py` and modify it to look for green objects.
- Modify it so that `find_blob()` takes a parameter (or parameters) to allow the caller to select the color it looks for.
- Have it look for purple objects.
- Implement a better way to find blobs – for example, one that would allow finding only a blob of at least some size, or that would not find the "center of mass of redness", as this one does, but rather would give back the center of the larger of two blobs rather than a point between them.

Exercise 12: A behavior that remembers

Overview

In this exercise, you will write a behavior that will build a map as the robot moves about its environment.

Goals

- Continue learning about behavior-based robotics.
- Learn about using an object remember information about past states.
- Build a useful behavior.
- Practice Python programming.

Background and resources

Behavior-based robotics, as it was originally conceived, is *stateless*: each behavior responds to the current precepts with commands it wants to perform, regardless of the history of the robot. Such a system is sometimes called a *Markov process.*[34] However, some useful tasks require a *memory*, that is, that a robot maintain some notion of the past and present states of the world. Creating a plan to achieve a goal and carrying out that plan, for example, requires that the agent (person or robot, e.g.) remember where it is in the plan, the outcome of past actions, what its future actions will be, etc.

There are several ways to incorporate memory into a behavior-based controller, including:

1. The memory can be external to the controller or behaviors. In this approach, the behaviors are still without memory; they just treat the external remembered information as part of the current state. For example, this is done in a simple form via the `gopigo_interface` module, which keeps track of the distance moved and the degrees turned so far. A behavior could just take action when these "percepts" have a particular value, such as indicating that the robot has traveled over 10 m, etc.
2. The controller can incorporate memory, for example, recording the behaviors' actions and their effect on the world or recording past percepts. Behaviors in this approach would work the same as in the previous one.
3. Behaviors themselves can incorporate memory.

The last approach is the one we will use here, and it is the one used most often, for example, in artificial intelligence systems that control robots. The question is, how can we keep this information around in the Python function that implements such a behavior from one call to the next?

[34] See, e.g., the Wikipedia entry for "Markov Property" if you are curious.

As you know, in Python, unless told otherwise, variables that first appear on the left-hand side of an assignment statement in a function create *local variables*. While these can be given values within the function, they cease to exist as soon as the function returns. Thus, they cannot be used to hold our behavior's memory.

We *can* access global variables from within a function by using:

```
global varname
```

Thus, we could use a global variable to hold the remembered information. Unfortunately, this is not particularly good programming practice, since: it makes debugging difficult, since the reader will need to refer back to global variables when looking at a function; it requires every behavior function that needs to remember something to use a unique name for its global variable; and it violates the idea of separation of concerns, since now the main program's file must include global variables that behaviors depend on.

Some programming languages have a mechanism to declare that a local variable is *static*, that is, that it doesn't disappear when the function returns, but maintains its value until the next time the function is called. For example, in this C language function:

```
int count() {
  static int counter = 0;
  return ++counter;
}
```

the local variable `counter` keeps its value from call to call. (The `++` in C before a variable means increment it, then return its value.) Thus the first time we call `count()`, it would return 1, the next time, 2, etc.

Unfortunately, Python does not have such a mechanism to keep variables static. There are two mechanisms it *does* have, however, that will work: defining a class to contain the remembered information; and using a closure.

Remembering information using a class or object

If we define a class that includes a variable definition, then that variable is considered a *class variable* and is static. For example:

```
class Counter:
   count = 0
   def increment():
       Counter.count = Counter.count + 1
```

This defines `count` as a class variable. It can be used like this:

```
>>> Counter.count
0
>>> Counter.increment()
>>> Counter.count
1
```

We can write a behavior function, then, that uses a class and its class variable to hold its memory.

One problem with this that each time the behavior controller is restarted, it will need to reset the class variables of all behavior memory classes to the default state. This can be done by defining

a common name for class methods to reset the state in all classes (e.g., `reset_memory()`), keeping a list of all memory classes, then calling the method on each class.

A second problem is that if two behaviors want to use the same kind of memory, two classes would have to be defined, one for the memory for each behavior, which would unnecessarily duplicate code. This can lead not only to wasted disk and memory space, but also to inconsistencies if one class is modified and the other is not (perhaps inadvertently).

A final problem is that this violates the separation of concerns ideal, since once again the controller has to be aware of things (the classes and how to reset them) that concern only the behaviors.

We could solve at least one of these problems by instead using objects (i.e., instances of classes) rather than classes to hold the memories. For example:

```
class Counter:
   count = 0
   def increment(self):
       self.count = self.count + 1
```

Note that here, we no longer use the class' name in `increment()`, but now use the special name `self` to refer to `count`. `self` refers to the *object* whose method is being called, which is given automatically to the method as its first argument when the method is called. When this class is *instantiated* to create a new object, the object is given its own variable `count` that is initialized to 0. For example:

```
>>> c = Counter()  # c contains a new instance of Counter
>>> c.count
0
>>> c.increment()
>>> c.count
1
>>>
```

Now, different behaviors can use the same class, but different instances of it, to hold their memories and there will be no problem. The question is, how do we separate concerns so that the controller doesn't have to know about the behaviors' needs, but still get behaviors to create instances of their memories once, and not each time they are called?

One way we can do this is to make the behaviors themselves methods of classes that do one of the following:

1. Use the memory class. In this case, the class implementing the behavior method would have an `__init__()` method that would create a variable holding an instance of the memory class; when the class is instantiated, the behavior object would have the behavior method and its own copy of the memory object.
2. Inherit from the memory class. Each behavior class would be defined to be a subclass of the memory class; e.g., `class Behavior(Memory):`. The variables holding the memory would be inherited by the behavior and would be accessible in methods using the `self` variable described above.
3. Implement their own memory variables and do not use a separate class.

The concerns of the controller and the behaviors can be separated by changing the controller slightly. Thinking back to Project 1, you defined a list `behaviors` to contain the behaviors that the controller would call. It contained a list of functions. Now, we would require `behaviors` to initially

contain a list of behavior *classes*, not functions, that the controller would instantiate when it starts, replacing `behavior`'s contents, and the controller would then just call a standard method of each instance (e.g., `behavior`) each time through its loop. This would, however, mean refactoring all of your existing behaviors to be objects and methods.

We can avoid changing your existing behaviors with a slight change to the controller. When the controller is first started, you would add code to check each element of `behaviors` to see if it is a class and, if it is, instantiate it and replace it in the list with the method to call to get the behavior.

This function should do this when called by the controller, assuming that `behaviors` is a global variable:

```
def initialize_behaviors():
    global behaviors
    for i in range(len(behaviors)):
        if type(thing) is type:              # then it's an object
            behaviors[i] = behaviors[i]().behavior
```

What this does is if element `i` is a class, then it instantiates it via `behaviors[i]()` and gets the `behavior` method associated with that instance (`.behavior`). This implies that each behavior you create as an object must have a method named `behavior` defined that carries out the behavior.

Remembering information using a closure

A different way of having information stay around from function call to function call is to use a closure. A *closure* is a function that has additional variables associated with it due to the way it was defined; it has an *environment* that contains variables that behave like, but aren't, global variables. To create a closure, we define a function inside a function. Here's an example for a counter:

```
>>> def createIt(initval):
...     count = initval
...     def counter():
...         nonlocal count
...         count = count + 1
...         return count
...     return counter
...
>>> a = createIt(0)
>>> a()
1
>>> a()
2
>>> a()
3
```

Here, `createIt()` defines a local variable `count`, then defines a function `counter()`. `count` is in the *scope* of `counter`: that is, it is *visible* from inside the function. The `nonlocal` statement tells Python that `count` is not a local variable of `counter`, nor is it (necessarily) a global variable; instead, it tells Python to look in the function's scope for the variable.

When `createIt()` is called, it returns the new function created; but since `count` was visible to `counter()`, it has to return that, too as part of the environment. This is a closure, which can be called just like a regular function.

For our purposes here, you could define behaviors needing memory as closures and put the closure in `behaviors`. The controller would not have to be changed, except to make sure that the closure is put on the list to begin with, and that each time the controller is restarted, a *new* version of the closure is created and placed on the list.

A downside of closures is that it is very difficult for anything outside of the closed function to access the function's nonlocal variables. This can make debugging difficult.

Resources

1. Article "Classes and Objects" at learnpython.org.[35]
2. Article "Closures" at learnpython.org.[36]

Materials needed

You will need:

- Your GoPiGo (with the camera and sonar attached).
- Your behavior-based controller and behaviors from Project 1 and the previous exercise (if desired).

Requirements

- Create a behavior, "map", that keeps track of objects seen while moving around the world.
  - You can be as fancy as you like with the map.
  - At minimum, you should record in the memory any sonar contact's position in the world if the sonar is within some predefined range of the robot, say `OBJECT_RANGE = 30`.
- After the controller is done, you should be able to print a list of things seen by the robot during its "mission".

Programming notes

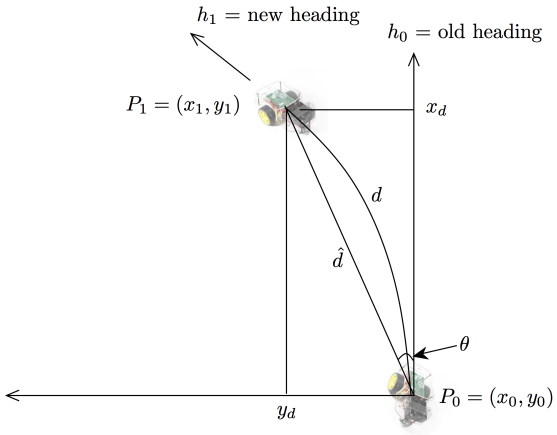
Choosing behavior implementation

- You should decide on one of the ways to implement your behavior's memory described in the Background section.
- Given the need to print the list of things seen, you should probably consider using a class or object for the memory.
- If you *do* want to use a closure, then you will need to have some way to tell the closure that it is to print the map rather than keep building it.
  - One way would be to give the closed function an optional variable, say `print_map`, that defaults to `False`; if this is `True`, it should print the map.
  - Assuming the closure is in the i^{th} element of `behaviors`, then after the controller is stopped, you would call `behaviors[i](True)` to print the map.

[35] `http://www.learnpython.org/en/Classes_and_Objects`
[36] `http://www.learnpython.org/en/Closures`

Estimating the robot's position

- You will need to estimate the robot's position in the world, which we haven't done up until now, in order to determine the location of sonar contacts.
- Assume that where the robot is placed when it is turned on is the origin of an xy-plane, and that the robot is pointing down the x-axis.
- We will call this location, then, (0, 0), and will refer to the direction the robot is pointing as its *heading*, which initially is 0°.
- The `gopigo_interface` you were given in a previous exercise includes two "sensors" that will be of use. These are given as part of the precept object to each behavior:
  - `odometer` contains the total distance traveled since the robot started; and
  - `cumulative_turn` contains the total heading change since the robot started.
- `map()` will have to update the robot's estimated location and heading each time it is called.
  - It should keep track of the prior `odometer` and `cumulative_heading` values as part of its memory.
  - It can estimate the current position based on these and the new values as shown here:



Assuming Δt is short, $\hat{d} \approx d$ and $\theta \approx h_1$.

$y_d \approx= d \sin h_1 = d \sin(h_0 + \Delta h)$

$x_d \approx= d \cos h_1 = d \cos(h_0 + \Delta h)$

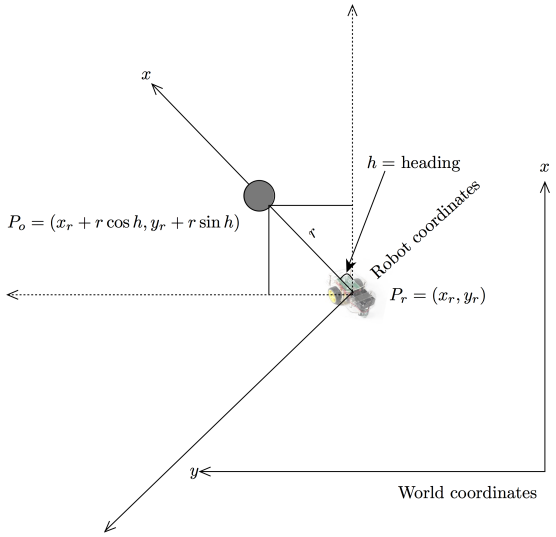
$P_1 = (x_1, y_1) \approx (x_0 + x_d, y_0 + y_d)$

  - Suppose that `map()` thinks the robot is currently at $P_0 = (x_0, y_0)$.
  - Δh, the change in heading, is just old heading, h_0 + the change in heading from the last time, which can be computed from the current value of `cumulative_heading` minus the old value.
  - d can be computed by subtracting the old value of `odometer` from the current value.
  - Thus, the new position could be computed by:

```
from math import sin
from math import cos
delta_h = percept['cumulative_turn'] - prev_cumulative_turn
d = percept['odometer'] - prev_odometer
heading = heading + delta_h
delta_y = d * sin(heading)
delta_x = d * cos(heading)
location = (location[0] + delta_x, location[1] + delta_y)
```

Estimating an object's position

- When an object is within `OBJECT_RANGE` of the robot, `map()` will record its position in the map.
- We will assume that the object is directly in front of the robot.
- The object's location can be estimated as shown here:



i.e., the location is $(x_r + r\cos h, y_r + r\sin h)$, where h is the robot's current heading, the robot's current location is (x_r, y_r), and r is the sonar range to the object.

Maintaining the map

- In the simplest case, you can just store the map as a list of tuples (x,y), one for each object found.
- For example, if the map is:

```
[(1,2), (100, 2), (105, 50)]
```

this would mean that three objects were found, one at each of the positions.

- You should make sure that you don't store duplicate objects.
  - You can do this by checking to see if a new object's location is already in this list.
  - Note that if you store locations as floating point numbers (i.e., numbers with decimal points), it is likely that you won't match two objects even if they are the same, if the robot moved slightly between sonar samples.
  - To solve this, you can either assume that an object is the same as another if it is within a certain radius of it or you can store all locations as integers (by calling, e.g., `int()` on x and y as you store them).
  - You may want to do the former, since this would help cut down on duplicate objects. For example, if you assume that all objects are at least 5 cm in diameter, then you could ignore any object that is within 2.5 cm of another one.

Testing

1. Turn on the GoPiGo and place it on the floor.
2. Start Python and load your file(s).
3. Let the robot run for a while. Perhaps you will need to move an object or place to have the robot continue moving, if it stops facing an object.
4. After a while, see what the map looks like.
5. Try this both with the simple controller and the blended controller from your prior work.

Questions for thought

1. How well did the mapping work? Were there many duplicate objects found?
2. How good were the estimates of where objects were? What do you think would contribute to any errors observed?
3. Did the errors get worse the longer the robot ran? If so, why, do you think?
4. Did you have to intervene a lot to keep the robot moving? Why? How could this be solved, do you think?
5. Were there any differences between the two controllers (simple and blended) when this behavior is added to the robot?

Stretch goals

- Change the avoidance behaviors so that the robot never stops permanently. One way to do this might be to use random numbers to have the robot, with some small probability, turn or move away from an object. For example, if you wanted the robot to usually stop when there is an object in front of it, but 20% of the time turn left, you could use `randint()` to get a random integer in the range 0–99 and turn if this integer is ≥ 80.
- Improve the way the robot keeps track of its location. For example, you could attempt to make a better estimate of where the new location is after moving and turning, instead of using the approximations we used above.
- Draw the map on the screen rather than printing out just the locations.
- Keep track of objects detections that could be part of the same object and merge them. Now your object representation might be (x, y, diameter), where x and y are the coordinates of the estimated *center* of an object with that diameter.