Homework 3: Computer Vision (Group) Due: Sept. 28, 11:59PM

Learning Objectives

- Use basic computer vision techniques, such as color spaces and filtering, to classify foliage in images and estimate plant height
- Perform color calibration of images and compare classification performance to uncorrected images
- Implement a behavior to acquire and process camera images, and test for robustness
- Characterize the behavior of the pump, weight sensors, and moisture sensors

This is a **group** assignment consisting of 6 parts (not equally weighted). You should decide with your teammates how to divide the work equitably. This assignment is a more open-ended than previous ones have been, so feel free to use techniques other than our "suggested approaches". In particular, the approach we suggest for classifying foliage (Part 1) has a lot of potential for improvement – see how well you can do!

In Parts 1-4, you will use `OpenCV`, which is an open-source computer vision package for Python that contains several thousand algorithms. Don't worry — you will need to use only some of the more basic functionality, but you should still get up-to-speed on what `OpenCV` has to offer by going through the tutorials: https://docs.opencv.org/4.x/d6/d00/tutorial_py_root.html.

`OpenCV` is already loaded onto your virtual machine if you are using it, but if not you should install `opencv-python` using `pip` (for M1 and M2 users, we will have an alternate guide for you). In addition, you are free to use the utilities (`cv_utils.py` and `filterColor.py`) that we developed for this class. The API for these two modules can be found in the September 14 lecture notes on Canvas in the **Computer Vision** module; the code itself is on Canvas and is also included in the assignment zip file.

For Parts 1-4, you can test your solution by running `autograder.py`. With no arguments, the autograder tests all four parts. You can test an individual part with the `-p[n]` flag, such as:

```
python autograder.py -p2
```

You can also test an individual image in the training set using the `-f [name]` flag, where `name` is the file name (without the `.jpg`) of one of the images in the `images` directory. For instance,

```
python autograder.py -p1 -f day08
```

Part 5 will be tested using the TerraBot simulator and will be evaluated using a tester file. Part 6 will be performed on your group's TerraBot itself.

Download and unzip the assignment code from Canvas. Place the folder in `~/Desktop/TerraBot/agents/` and copy (or link) the files `greenhouse_agents.py, layers.py,` and `ros_hardware.py` from your `ROS_HW` directory, and the files `greenhouse_behaviors.py`, and `ping_behavior.py` from your `FSW_HW` directory.

Part 1: Classifying Plant Foliage (30 points)

For this part, you will be editing the function `classifyFoliage` in `vision.py`.

While the sensors you have been using are able to give an indication of the environmental conditions in the greenhouse, they cannot give direct indications of the status of how well the plants are growing. The TerraBot has another sensor that you've not used yet — a color camera. In this part, you will detect plant foliage in camera images, which you could then use to determine such things as how tall the plants are or how many plants are growing.

The basic goal of this part is to write a program that creates a grayscale mask in which the pixel value is 255 if there is a plant pixel in the original image and 0 otherwise. The `images` directory contains images taken during 10 days of a run of the TerraBot. Modify `classifyFoliage` in `vision.py` to return a mask showing where foliage is located, given an input image (a `numpy` array of shape `[<num rows>, <num cols>, 3]`).

You should start by viewing the images to get a feel for the commonalities and differences between them. We also have a few more images that will be used in grading your submission, so you need to make sure that whatever approach you take can handle unseen images, as well.

You can use any approach to finding foliage that you'd like, but here is a suggested approach:

- Use the `filterColor.py` program to find ranges of values in some color space that is effective at finding plants in the training set of images. While there won't be one range of values that works for all the images, try to find a range that works reasonably well for them, on average.
- Modify `classifyFoliage` in `vision.py` to return a mask indicating foliage in the image passed as an argument. Create the mask based on the color space and the range of values you found that seem to indicate foliage. If your range of values includes the color calibration targets, don't forget to mask them out — you don't want to report the green target as plant life!
- Try creating masks using different color spaces — some may be more effective than others. Remember that if you use a color space other than BGR, you'll have to transform the image before creating the mask.
- Try performing gamma correction (see the Computer Vision I lecture slides) on the images before masking, to see how well that works. Remember that you'll probably have to find new filter values for the gamma-corrected images.
- Try blurring or sharpening the images before invoking `filterImage` (suggestions for how to do that can be found in the `OpenCV` tutorials or on the web). How does this affect your accuracy, if at all?
- Try eroding and then dilating after creating mask to remove small parts that are not really connected to plants (suggestions for how to do that can be found in the `OpenCV` tutorials or on the web). How does this affect your accuracy, if at all?
- For the radishes, try finding another mask that classifies their reddish stems, and combine the two masks to get a more robust classifier.

The approach above, using `filterColor` to find ranges of color space values, finds a 3D cube in color space. As discussed in class, there is often a relationship between the color space bases that cannot be captured by a cube. A better approach would be to find another shape, such as an ellipsoid, that better encapsulates the color values of the plants, while leaving out non-plant values. If you feel motivated, you can try this more complex approach.

If you'd like to try a more modern, machine-learning approach, the VM already has the `sklearn` module installed and the files in `autograder_files` contain masks that essentially label the images (**warning: the masks are not 100% accurate, but they are what we use to evaluate your classifier**). Developing a CNN that classifies foliage should be attainable with the images that we provide.

Run `python autograder.py -p1` to check how well you did. While the autograder for this part prints out several statistics (recall, precision, balanced accuracy, and F1 score), the grade is based only on balanced accuracy. In addition, we don't expect you to get 100% classification accuracy - the autograder curves the score (note: returning a blank mask – what you get without modifying `classifyFoliage` – gets you 50%). Since we hand-labeled the images to produce "ground truth" masks, even these are not 100% accurate. The curved accuracy, not the raw accuracy, corresponds to your grade on this part.

Note: your code should not include any display of images, nor any need for manual intervention. You can include either during development, but all such code should be commented out of the version you submit.

Write a paragraph in the `README` file describing your approach, including what color space you used and how you determined the subspace that corresponds to foliage, or the machine learning technique that you used.

Part 2: Measuring Plant Height (20 points)

For this part, you will be editing the file `vision.py`.

One thing you can do with the foliage mask is to determine how much foliage there is in the greenhouse, as a sign of growth by counting the non-zero pixels. Another thing is to calculate plant height, another sign of growth. While it is difficult to compute the height of all plants, since the 2D image does not readily allow us to calculate the distance of the plants from the camera, there is a simple measurement we can make. The greenhouse is equipped with a measuring stick that has black tick marks every centimeter (the lowest tick mark visible in the images is at 0 cm, even with the top surface of the "soil").

In this part, you are to determine the maximum height of the plants that are visible **in front of** the measuring stick (that is, plants that partially obscure the stick). Modify `measureHeight` in `vision.py` to return that height, given a `foliage_mask` of the type returned by `classifyImage`.

Suggested approach:

- Create a mask for the measuring stick; you may be able to use `filterImage` in some color space, but it may be easier to just determine the four corners of the stick by hand and assume that it will be mostly in the same place in each image. **Hint:** you can use `drawContours` to fill in a polygonal shape. A contour is simply a `numpy` array of a list of [row, column] points that define a polygon (in order). If `c` is such a contour, then `cv2.drawContour(image, [c], 0, 255, -1)` fills the interior of the polygon with the value 255.
- Find the black lines in the masked image (using one of the images before plants appear) and determine which row in the image corresponds to which centimeter (**note:** the lowest black line is level with the top of the "soil").
- In `measureHeight`, combine the `foliage_mask` with the measuring stick mask to see if there is any foliage that appears in front of the measuring stick. If not, return `None`. If there is, find the highest point and use the black line measurements to interpolate the plant height. Return that height.

The first two steps above should be done only once, offline, with the results saved to be used by measureHeight.

Note that the stick mask may not be perfect and the stick may move slightly from image to image (for instance, if a plant is growing against it). To handle this, you may want to erode the mask a bit -– but don't erode too much, or you will miss plants that just marginally overlap the stick.

Run `python autograder.py -p2` to check how well you did. The autograder will accept some range of values for the height, so you don't have to be exact.

Write a paragraph in the `README` file describing your approach, including how you created the measuring stick mask, how you processed the stick and foliage masks, and how you determined the plant height.

Part 3: Color Correction (17 points)

For this part, you will be editing the file `vision.py`.

As discussed in class, image classification can be very dependent on lighting conditions. While working in a color space that separates hue/chroma can help, it is not a perfect solution. That is where color correction comes in. In this part, you will implement the color correction algorithm described in class (and in the lecture notes). Modify `colorCorrect` in `vision.py` to return a color-corrected version of the image, given `blue_goal, green_goal` and `red_goal`, the "ground truth" values of the color calibration squares (each `_goal` is a tuple of blue, green, red — in that order).

Suggested approach:

- Create masks for each of the three color calibration squares; you may be able to use `filterImage` in some color space, but it may be easier to just determine the four corners of the squares by hand and assume that they will be mostly in the same place in each image. **Hint:** you can use `drawContours` to fill in a polygonal shape.
- Erode the masks a bit, so that border pixels are not included (since they may be mixtures of the color calibration square and the background).
- In `colorCorrect`, find the mean value of each color square in the image

- Set up the "A" matrix and "d" vector, based on the lecture notes
- `colorCorrect` already has the code to compute the "x" vector and the "T" matrix
- Transform the image using the "T" matrix
- Be sure to scale the transformed values from 0 to 255, since the transform may produce values that are below, or above, that range

The first two steps above should be done only once, offline, with the results saved to be used by colorCorrect.

Run `python autograder.py {p3` to check how well you did. The autograder will check to see how closely the color calibration squares in the transformed image correspond to ground truth or "goal" colors.

Part 4: Classifying Plant Foliage with Colored Corrected Images (5 points)

For this part, you will be editing the file `vision.py`.

In this part, you will combine Parts 1 and 3 — classifying images after first color correcting them. Modify `classifyFoliageCorrected` in `vision.py` to return a mask showing where foliage is located, given an input image.

We have provided most of the code for this part. The one issue you need to worry about is whether the color space filters you used in part 1 are still valid for part 4. You can handle this in one of two ways:

1. Find a set of goal colors that transform the images in such a way that the original color-space filter values still work well.
2. Find another set of filter values that work well for the color-corrected images and modify `classifyFoliage` so that it can use different filter values, depending on whether it is called directly by the autograder (for Part 1) or within `classifyFoliageCorrected` (part 4). You need to ensure that the autograder can be run on all parts without manual intervention while using the correct filter values for the two different parts.

Run `python autograder.py {p4` to check how well you did. As with Part 1, we don't expect you to get 100% classification accuracy - the autograder will curve the scores. However, we do expect the color-corrected results to be better, on average (it is perfectly acceptable for some images to have lower accuracy, as long as average accuracy is higher).

Part 5: Acquiring and Processing Images (18 points)

For this part, you will be editing the file `camera_behavior.py`.

As mentioned, the TerraBot has a color camera that enables your agent to acquire images and process them. In this part, you will write a behavior to acquire and process images.

To prepare for this part, you need to copy or link your versions of `behavior.py, greenhouse_behaviors.py, greenhouse_agents.py, hardware.py, layers.py, ping_behavior.py, ros_hardware.py`, and `schedule.py` from assignments 1 and 2 into `CV_HW`.

You then need to edit `greenhouse_agent.py` to import `camera_behavior` and add `camera_behavior.TakeImage()` to the list of behaviors in `LayeredGreenhouseAgent`. You may also need to edit `ros_hardware.py` to make sure that your `doActions` function handles the 'camera' action (and, while you are at it, make sure it handles the 'ping' action).

Your agent requests an image by calling `actuators.doActions` with the {`"camera": path_name`} tuple as its last argument. `path_name` is a string consisting of the directory and file name where the image should be stored.

The camera behavior to implement is as follows:

1. For consistency, adjust the light level to some reasonable value – full on is typically too bright for taking good pictures. A light level of around 400-600 is good for image collection (hint: look at how the `Light` behavior is implemented). Note that the TerraBot tries to adjust the shutter speed based on the light level. Since it takes a bit of time for the light level to be registered, wait a few seconds after finishing light adjustment before taking an image.
2. Once your agent requests an image, you need to wait until the file shows up. This is because it takes some time for the TerraBot to acquire the image and write it to disk. Wait for at least 10 seconds and then use `path.exists(pathname)` to check for whether the file has shown up. **Note that the image appears much faster in the simulator than on the TerraBot hardware. You should check to make sure that you are waiting long enough – if not, the image won't be read in properly.**
3. If the file does not show up, wait for 20 seconds and try again, but if you've tried unsuccessfully three times in a row, give up and print a warning message.
4. If the file does show up, read it in and call `processImage` (definition is at the end of `camera_behavior.py`), which does some processing (calling your `classifyFoliage` and `measureHeight` functions).
5. Acquire at most one image each time the behavior is invoked.

Start by drawing the FSM, labeling the states, conditions, and before/after functions. Then, use what you learned in Assignment 2 to create an FSM that implements the above behavior.

Some things to note:

- In the past, some teams have found that the adaptive shutter speed did not work for them when tested on the TerraBot hardware. If you want to use a fixed shutter speed, invoke `TerraBot.py` with the `-f` option.
- **Make sure that the directory you are writing to has been created already** – the command will fail if the directory does not exist!
- If the path name is relative, it is relative to where `TerraBot.py` is run from, **not** where your agent is. You are probably best off using an absolute path name.
- Make sure to name your image file names different from one another, to avoid overwriting them (a good strategy is to include the `unix_time` in the names).
- The `pathname` should not contain spaces or special characters, other than dashes or underlines.
- You can test using the simulator. While the color calibration squares don't match up and there is no measuring stick in the simulator, you want to be sure things are working correctly, rather than what the outcome is of processing the image. You won't be graded on the results of the image processing – in fact, the functions in the original `vision.py` that is included with the assignment will work just fine.
- One caveat is that the image size from the simulator does not match the image size of the real camera. Therefore, the masks that you developed in Parts 1 and 2 will not be the same sizes, either. To avoid that, you can add lines of the form:
  ```
  if (mask1.shape != mask2.shape):
      mask1 = cv2.resize(mask1, (mask2.shape[1], mask.shape[0]))
  ```
  to the appropriate places in `classifyFoliage` and `measureHeight` where you read in the predetermined masks.

This behavior is hard to test using tester files, since there are no sensor readings that get set. We have, however, provided you with `test_take_image.tst` that tests some of the parts of the behavior. In particular, it tests whether the light is within range (400-600), that the image file is successfully written to disk, and that exactly 3 images get taken each day. You can test whether it handles failure conditions correctly by requesting an image to be written to a non-existent directory. (Hint: use baseline files to set up the correct conditions for testing, without having to wait for the appropriate times to occur).

To test the behavior, run `python TerraBot.py -m sim -s <speedup> -t <dir>/test_take_image.tst` in one terminal and `python greenhouse_agent.py -L -m sim` in another terminal. Note that the camera behavior has a number of steps that happen every few seconds – although it will take longer to test, we don't recommend speedup of more than 250. However, if things fail unexpectedly, you can try using a slower speedup. You can also speed up testing by (temporarily) removing all lines in `greenhouse_schedule.txt`, except for the `LightBehavior` and `TakeImageBehavior` entries (since the simulator won't slow down if the fans or pump aren't turned on).

Make sure you run for several simulated days – a common bug is to forget to reset the image counter and after 3 images the first day it never takes another image.

Part 6. Characterizing the Watering Process (10 points)

Watering is probably the most difficult behavior to perform autonomously on the real hardware. This is mainly because the sensors that are available related to watering (soil moisture, water level, and weight) are very noisy and/or not well modeled. In this part, you will characterize how things work on the actual TerraBot hardware.

Refer to the notes "Using the TerraBots" in the "Grow Periods" module on the Assignments page on Canvas for instructions on how to access the TerraBots.

ssh into your TerraBot and create two windows (or use tmux to use just one window, as described in "Using the TerraBots"). In one, cd to `Desktop/TerraBot` and run `./TerraBot.py`. In the other, cd to `Desktop/TerraBot/agents` and run `./interactive_agent.py`. Entering "h" at the interactive agent will give you the list of what it can do. Try turning the lights and fans on and off, and look at the various sensor values (and how they change as you turn the actuators on and off). **Don't turn the pump on, at this time!**

Once you are comfortable with running the TerraBot and the interactive agent, you can prepare for this part of the assignment. The idea is to record the sensor readings you get from turning the pump on and off, to see how they react to the addition of water. You can do all of this part manually, using just the interactive agent, or you can write code to automate some, or all, of the process.

Note that you really have just one chance to do this, because the "soil" will get wet and it takes at least a week to completely dry out. So, it is recommended to try everything first with the simulator, so you know you have the process correct.

Here is the procedure:

- Record the initial readings for the two moisture sensors, the water level sensor, and the weight sensor. Look at the water reservoir and estimate the amount of water from the markings on the side of the reservoir.
- Turn the pump on for 20 seconds, then turn it off. Try to be fairly accurate in your timing, but it is not necessary to be exact.
- Record all the readings immediately after turning off the pump, 1 minute later, and then 4 minutes after that. As "ground truth", for the water level sensor, measure from the bottom of the reservoir to the top of the water level. Note that the water level sensor is quite noisy – you should take a series of readings and compute the mean and standard deviation at each of the three time intervals.
- Continue the above two steps until you have done this process for at least 10 times. It should take about an hour to do so — one reason why you might want to write an FSM behavior to automate this part of the assignment.
- Wait 30 minutes and take one final set of sensor readings (to see how the moisture sensor changes as the water gets soaked up).

If you write code to automate the procedure, you can use Python's logging package to record the data – look at the coffee maker code from Assignment 2 as an example of what can be done.

Once you have the data, graph the results where the x-axis is the amount of water that has been pumped out (estimated from the measured water level in the reservoir, and knowing that 100 ml of water equates to about 4.5 mm in the reservoir), and the y-axes are the various sensor readings at the various times for each water level. In particular, you will have three readings for each water level (immediately, 10 seconds later, 3 minutes later), plus a fourth reading at the very end representing the readings after 30 minutes.

Create a document with the graphs and an analysis of the results. The analysis should address:

- How much noise there is in each sensor
- The relationship between the water level sensor and the actual water level (ideally it should be linear)
- The relationship between the moisture sensor(s) and the actual water level
- The relationship between the weight sensor (average of the two weight readings) and the actual water level – note that water weighs about 1 gram/ml.
- How the moisture level changes over time (if it does) even after the pump is off
- Any aspects that you found surprising
- What changes you would make to the watering procedure that is currently implemented in `greenhouse_behaviors.py`, based on your analyses. In particular, how might you incorporate the weight sensor, given that the weight is based both on the water in the "soil" and the amount of plant growth.

Submission

Submit your `vision.py`, and `camera_behavior.py` files to Canvas, along with the `README` containing descriptions for Parts 1-2 and the document containing your graphs and analyses for Part 6. When put in a clean directory with relevant files from Assignments 1, 2 and 3, they should run without error.

Important: Since you need to specify in your code where the camera images are being saved, be sure to indicate in your submission which directory should be created when grading on our machines.