CSE 490R P2 - Local Control with Computer Vision
Due date: Sunday, Feb 11 - 11:59 PM

1 Introduction

Local control for robotics usually takes the form of a function that operates on observed sensor values to immediately change control outputs. This is in contrast to global methods that provide long term plans, perform more complex calculations, or are designed to accomplish secondary functions. While global planning/control techniques do provide long-term plans, they are usually too slow or computationally expensive to allow for fast reactions to sudden or unexpected events. Local control can react to and overcome these kinds of errors while driving the robot to a desired configuration. For example, your smart phone can generate driving directions (global), but someone needs to steer the car (local). In practice, robots may use both global planning techniques and local control methods - the global planner provides the long-term path which is executed by the local controller.

In this homework, we focus on designing a local controller that operates on visual observations (RGB/Depth images). Vision allows the robot to extract relevant features from the world which we can use to make control decisions to drive our racecar. In our setting, the relevant features are a path on the ground that your car will have to drive along (essentially, you are solving the line following problem using cameras to detect the line).

No skeleton code is provided for this lab, but you may refer to Lab 0's **BagFollower.py** and **ApplyFilter.py** for structure and/or inspiration. There are hints provided in this document that may or may not be relevant to your design decisions.

2 Feedback Control

A feedback controller is a function that provides a positive or negative gain on some measured term(s), using it to compute the desired control. The difference between the measured term and some desired value (for the corresponding term) is calculated, and "fed back" into a controller which calculates what the system will do next, repeating the process. This is also called "closed-loop control". First, however, you need to measure the error in your system.

2.0.1 ROI Vision Processing

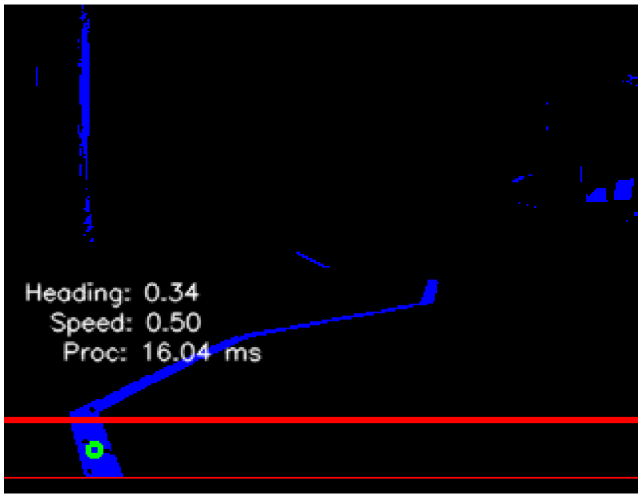
Your robot car is equipped with an RGBD camera. Color information in RGB space can change depending on lighting conditions (i.e. the RGB value for the path will be different in bright light or dim light), making it difficult to extract this information. A good local controller should be tuned to work robustly; we do that in this case by converting to a space that is more invariant to light - HSV space. Your image processing code should thus convert to HSV and, using the appropriate channels, create a pixel mask that isolates the colors you specifically care about. All paths that your robot will need to follow will either be completely blue or completely red. This mask is in essence a binary image: there should be a positive value at each pixel that corresponds to the color you wished to filter for, and a zero value pixel everywhere else. This mask should highlight all pixels in the image that are the desired color, which can include areas of the image that you do not care about. The next section discusses extracting information from just a region of interest.

While there is no requirement to use them, some useful commands may be:

`cv2.cvtColor()`	You can convert an RGB color as a three value array to a different color space. When you are searching for a desired color in HSV space, you will have to convert the range of RGB values into a range for HSV values, before using the next command to filter by range. RGB values can be queried from the raw images themselves.
`cv2.inRange()`	Given a lower and upper range for a color value, this function can filter your source image by those ranges.
`cv2.GaussianBlur()`	Using a Gaussian kernel, this function can blur your image making sharp features smoother.
`cv2.BilateralFilter()`	Similar to blurring, this function preserves edges at the cost of computation.
`cv2.imshow()`	Displays the image (not when code is running on the car) provided; useful for debugging if you are not publishing a manipulated image.

2.0.2 Region of Interest

After you have extracted the path from the original image, you will need to select a region of interest. This region represents the 'feature' the robot car seeks to perform local control on. The most naive method for doing this would be counting the non-zero pixels in the rows and columns of your HSV-derived mask. You can then find an approximate center of the non-zero pixels to act as the center of your region of interest. Another method could be to reduce the size of your mask image (i.e. non-zero pixels that are adjacent to zero pixels are set to equal zero) until just a few pixels remain, giving you the center of your ROI. You should do this for a sub-region of the full image: the car cannot drive up walls or fly, so computing some parts of the image may not be computationally worthwhile. Additionally, you should avoid unnecessary processing to save computation time.



```
Heading: 0.34
  Speed: 0.50
   Proc: 16.04 ms
```

Figure 1: Here we visualize our masked path (in blue) and the tracked region of interest (between the red lines). We also display relevant car info for debugging purposes. You can see in the screenshot the path moving up and to the right, but also there are spurious features elsewhere in the scene. The car needs to be robust to these extraneous features.

Finding the center of the region of interest applies structure to the pixels from the camera. Once you have the center of the ROI, the goal will be to direct the robot car's controls to drive the car in such a way to keep the center of the ROI in the center of the image. That is to say, if we are driving towards the center of your ROI, we are constantly following the goal, which in this case, is the line path.

While there is no requirement to use them, some useful commands may be:

`cv2.SimpleBlobDetector()`	Creates a detector that finds 'blobs' within your source image. These blobs can be useful features in your mask image.
`cv2.Canny()`	Runs the Canny edge detector on your source image. This can find the outline of your region of interest.
`numpy`	OpenCV images are represented in Python as nd numpy array. Any and all numpy operations are available for use, such as matrix indexing.
Drawing Functions	Useful for manipulating your processed images to draw additional features for debugging.

The full OpenCV documentation for the version on your car can be found here.

2.1 PID

The PID controller is a simple, but powerful, method for feedback control.

$$u_t = k_p * e_t + k_i * \int_0^t e_t \, dt + k_d * \frac{de_t}{dt} \tag{1}$$

The three gains k_p, k_i, k_d are used in conjunction with the error term to calculate our control. In the equation above, we can see the effects of each term on the right hand side. First, we have the proportional gain: the controls taken u_t should be proportional to the error measured from the system. Ideally, this control should make the error go down. However, when the gain is too large, we may overshoot our target, and start accumulating error in the other direction. The last term, the derivative term, combats this. By tracking the change in error over time, we attempt to reduce sudden changes in error, deliberately slowing the control response to avoid overshoot. Finally, the term in the middle, the integral term, accumulates the error over time. Imagine a PD controller that is never able to fully reduce the error to zero. If we accumulate this non-zero error, we can adjust our controls to negate this, thus bringing the error to zero. The I term is thus meant to combat *steady state* error, such as biases in the system or external effects such as wind.

In the prior section, you extracted the center of the path that is (presumably) in front of the car. If we want to follow the line that is the path, we should keep this line in the center of the car. You will specify and implement a controller of a PID design that keeps your car from leaving the line as it follows the loop. Your controller can output a constant linear velocity and adjust the steering angle as determined by the design of your PID. However, you do have the option to vary the linear velocity as well if you so choose. Either way, **please choose a velocity that minimizes the chance of breaking your car, and/or implement a safety controller.**

At a high-level, your feedback control module should do the following:

1. Subscribe to the appropriate camera topics and initialize your parameters
2. Convert to HSV space, and filter the image to isolate the line path as an image mask. You will have to design this process.
3. Calculate the center of your ROI, and compute the error. You will have to design this process.
4. Use the error in your PID controller to set the car controls.
5. Let your car drive at a constant speed to follow the path.

An example of line following in Sieg can be found here: Sieg 322 TA car. You may also search online for CMU mobot for other inspiration.

2.2 Submission

1. Provide an image capture of each stage of your visual processing pipeline. Starting from the raw camera image, render and annotate an image for each operation being performed on the image as you manipulate it to segment out just the relevant path. Operations can include removing noisy data, smoothing the scene, cropping the region of interest, etc.

2. Benchmark your image processing code. A local controller is expected to run at high frequency to compensate for perturbations and other errors, so your code will need to have low computational overhead. Time the inner loop (callback) of your image processing function, calculating the average processing time of each callback call and calculate (posthoc) the variance. Is it fast enough to process all available images provided by the camera (30 fps)? If not, think about how you would improve the performance of your code, describe it for submission here, and then implement it. Benchmark your image processing code after the improvements (mean and variance), and include them in your submission.
3. Describe the controller you implemented. Specify the gains you picked, and how you tuned them. If you did not use all the terms, justify why.

3 Feedforward Control

We can develop a model that directly gives us a pre-specified control based on some measured error. This is different from feedback control as our controllers outputs are somewhat fixed to specific errors, instead of purely proportional. This is also different from "open-loop" control where a robot runs some pre-specified controller regardless of whatever error is measured. In particular, when doing vision-based control, we can examine control schemes that consider the whole frame rather than just a region. You will achieve this through template matching.

3.1 Template Matching

Each template represents a generic path that the robot may encounter while going around a track. A template is an image that you will compare to the extracted color mask (from section 2) in order to find a match. Each template has a height equal to that of the mask, and a width that is a fraction of the mask's width. In addition, each template has an associated control that should be performed when that template is observed. Fig 2 illustrates a mask that the robot may find when driving. Given this mask, the robot must decide which of its pre-specified templates best matches the mask.

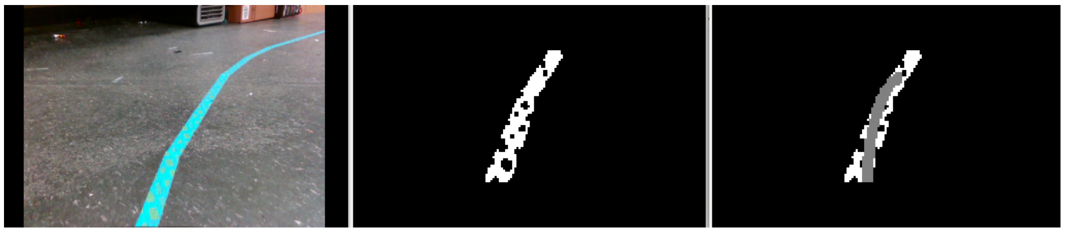


Figure 2: Left: A path that the robot is trying to follow. Middle: The mask of the path (note that the image has been cropped) Right: The mask of the path in white, with chosen template in gray overlaid.

After finding the best matching template, the robot executes the control that is associated with that template. Again, you can choose to have your controller output a constant velocity and only vary the steering angle. The robot may adjust this control depending on the part of the mask that the template best matched with. For example, if the template best matched with a region of the mask that is far to the left of the center of the mask, then the robot might choose to turn more to the left (in addition to whatever control is associated with the chosen template).

Your goal is to implement template matching such that your robot can follow a square shaped path. This will involve creating a set of appropriate templates, matching templates to the observed mask, and executing controls associated with the matched template. Hint: Use the provided *convolve* function to help find the best matching template.

As this is feed-forward, the controls you execute can be used for multiple frames (in feedback control it would constantly be adjusted). For example, if your car suddenly sees a sharp turn, you may want it to turn until it sees the path again. Similarly, if your car register's a U-Turn template, you may play an entire "3-point turn" control sequence. This is a design decision that you should consider.

At a high-level, your template matching module should do the following:

1. Initialize templates (once) - the number and content of the templates is for you to decide

2. Get the thresholded image (from section 2)
3. Compute the best matching template
4. Choose action that corresponds to chosen template and modify it according to the template's offset from the center of the image.

3.2 Submission

1. Implement template matching and visualize the mask and chosen template in a fashion similar to that of the right-most image in Fig. 2. Do this for at least three scenes/images that you find to be interesting.
2. Submit the templates that you generated. What was your thought process in choosing these templates?
3. How fast is your controller running? Your code should be efficient enough such that your robot can stay on the course.
4. Describe how you assigned controls to each template. Did your design execute a sequence of pre-specified controls for a template, or just one?

4 Extra

4.1 Extra Credit 1

Robustification. When testing your controllers, we will perturb the car, obscure the path, both, or other. Low level controllers can be improved in a variety of ways to be more resilient: they can slowly decay their last good controls to span gaps, plan for specific deviations (with a template) in the path, or other ideas you may come up with. You will be asked to describe how you included this in your design, and the TAs will test this at demo time.

4.2 Extra Credit 2

Higher performance. The ZR300 camera system contains a fisheye lens that can operate at 60fps, but is limited to greyscale (no color) images. Devise a way to use this camera to track the path with your feedback controller running at 60fps. You will be asked to describe your designed system and demonstrate it performing line following at 60Hz. You may need to dig into feature tracking systems, some of which is provided by OpenCV.

4.3 Extra Credit 3

The ROI extraction you developed here can be extended to find any region of interest, not just a path on the ground. Convert your code to have your robot follow you as you walk around.

You can extract specific colors that a group member is wearing, and use that information to calculate a heading and speed in order to maintain a specified distance from them. The depth camera can also be used for this. Finding an ROI is not limited to cameras, however: the LIDAR scanner can also be used to track legs (think about the shape a person's leg takes in the LIDAR scan).

To get this extra credit, your car will follow a group member, or a TA, as they move around Sieg, Partial credit will be given if your car can only follow by driving forward: your car should back up when necessary. As an additional option, you can select which color to track at run-time.

4.4 Extra Credit Submission

You will have to write up your design decisions to be submitted with the required work, as well as demo your robot to the TAs to earn the extra credit.

5 Demo

1. We will use your car's Feedback controller to follow the smooth path in Sieg 322.
2. We will use your car's Template Matching controller to follow the square path in Sieg 322.
3. You will be asked about your car's controller design and performance characteristics, and each team member's role in the implementation.
4. Describe how you assigned controls to each template. Did your design execute a sequence of pre-specified controls for a template, or just one?