Robotics Club - Colony

Introductory Lab #1 - Line Following

Release Date: 10/16/13

Demo Date: 11/5/13

Goals of this Lab

- Learn PID controls
- Create a Fast & Reliable Line-Following Behavior

Environment Setup

- Downloading ScoutOS

Scoutsim Functionality

- Line Visualization in Scoutsim
- Ghost Scout

Writing your own behavior!

- Intro to PID
- Using the Linesensor
- Follow the Line!

Goals of this Lab

You can skip ahead to the Environment Setup section if you like.

Why Line Following?

Line following is an easy way for us to get our robots moving in an unknown terrain. The same program can be used in multiple settings without modification. Also, it is simple to write and we can use the lines to keep track of where we are.

The robots use line-following for a great number of higher level tasks so we can simplify driving around autonomously. This lab should teach you everything you need to know about writing line-following behaviors and how to test these behaviors in the simulator.

By the end of the lab, you will learn a control system called PID (for Proportional, Integral, Derivative control), as well as some additional features of the simulator that will make debugging useful. Then, you'll take it a step further by actually writing a line following behavior and using it to control your customized scout.

How much time should this take? A moderate amount. This lab is easy if you are familiar with line-following and programming. However, if you are not, it will take some time to get used to PID. Budget 30 minutes to get your environment set up. Then take 30 minutes to play with the new simulator features, and maybe another two or three hours to write your own behavior. We have given you some template code to get you started.

Environment Setup

If you already have the colony code, skip ahead to the Scoutsim Functionality section.

Step 1: Downloading ScoutOS Repository

To get the code, just download: http://roboticsclub.org/redmine/attachments/download/655/lab1.zip. Extract it and put it somewhere convenient in your Ubuntu workspace. For example, here's a setup we recommend, and the resulting file structure.

Workspace folder:
~/scoutos (meaning a scoutos folder, in the home directory. You can use terminal or the typical folder view).

You can create a new folder using the terminal with the command:
$ mkdir <folder name>

Extract the zip file to a scout/ folder within ~/scoutos. So now you have a path like this:
~/scoutos/scout

And within that, you have a bunch of folders:

analog	cliffsensor	encoders	linesensor	motors
sonar	bom	CMakeLists.txt	headlights	Makefile
power	stack.xml	buttons	Doxyfile	libscout
messages	scoutsim	usb_serial		

You are now set up!

Caution: ROS_PACKAGE_PATH
When ROS tries to find and run files, it expects them to be in certain locations. You can control where it looks by changing the ROS_PACKAGE_PATH variable, which is part of your **environment**. If ROS seems to have trouble finding your files, do this in a terminal:

$ echo $ROS_PACKAGE_PATH
(are the scout files within a folder somewhere on that path?)

$ export ROS_PACKAGE_PATH=$ROS_PACKAGE_PATH:/path/to/scout/files/goes/here
(adds the folder you specified - /path/to/scout/files/goes/here - to the path. Now ROS should find it.)

A second word of caution: You can't have duplicate package names on your ROS_PACKAGE_PATH. Move your Lab0 code elsewhere, and move your Lab1 code into the path (or change the path).

Scoutsim Functionality

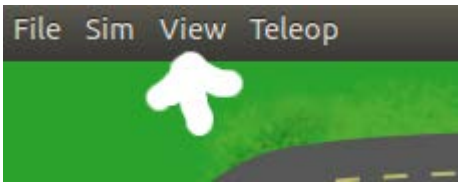
Oh you'd like to skip this section? Aren't you a smarty-pants.

Remember that you'll need to open up a separate terminal and run the following command before running any of the ScoutOS programs:

```
$ roscore
```

Step 1: Visualizing the Line in Scoutsim

At some point, you may wonder: "where is the line that the scout is supposed to be following?" In Scoutsim, there is a separate map that indicates where the lines are. These maps can be found in the `scout/scoutsim/maps` directory with filenames ending in '`_lines.bmp`'. Depending on where the scout is, scoutsim simulates the linesensor readings according to the line map. In order to visualize the line map, go to View>Lines in the Scoutsim menu. Note that this will automatically clear the trail indicating where the scout has traveled.



Step 2: Running Ghost Scout

When Scout travels, it has encoders on the motors that indicate how much each wheel has turned. Using some fancy math, we can use this information to estimate where the scout is. A fun thing to do is visualize 'ghost scout' -- the estimated position of the scout. This can be done by:

```
$ rosservice call /set_ghost <true/false> <scout_name>
```

This will set the ghost on if you used true, or off if you used false. Note that the ghost won't move unless you are running the odometry behavior that does all the fancy math for you.

Writing your own behavior!

Remember that nothing travels faster than the speed of light, with the possible exception of bad news, which obeys its own set of laws.

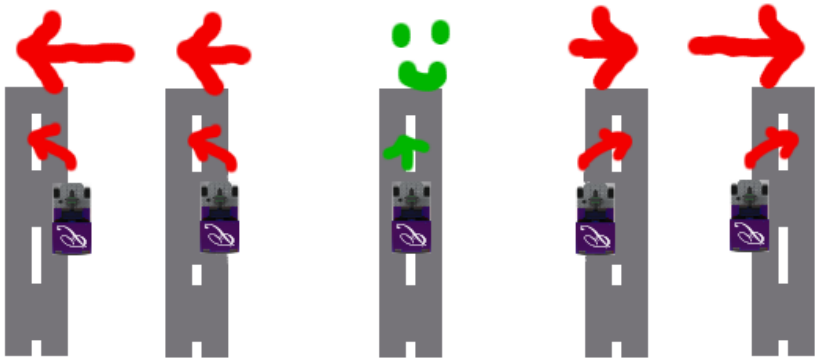
Step 1: Introduction to PID

PID stands for proportional, integral, and derivative controls. Each of these three terms in the control algorithm specifies a way the Scout can respond to an error between its state and its goal (whatever those are). For line-following, you will probably only need P and D controls.

A quick note! The line sensor is on the **back** of the robot. This means that you should drive backwards! One thing this means is that the right and left sides of the robot are switched when it comes to setting motor speeds. Try to keep this in mind as you read our code below.

P Control:

When driving on the racetrack, if your Scout is to the left of the line, you want to turn right; if your Scout is to the right of the line, you want to turn left. Proportional control means that you turn proportionally to how far away you are from the line. When you're really far away, you make a hard turn, when you are just a little off, you turn less drastically.



This is generally done by having some sort of error measure. For example, error == 0 means you're on the line; positive error means you're to the right; negative error means you're to the left. You can use the error measure to change your base left and right wheel speeds. Depending on the magnitude of the error and base speed, you might need a scale term to magnify/shrink the error accordingly.

TLDR Formula:
```
right wheel = base_speed - scale * error;
left wheel  = base_speed + scale * error;
```

Derivative Control:

Once you have implemented P control, you might notice some overshooting. That is, the Scout turns correctly and approaches the line, only to overshoot where error is 0, and now must turn the other way. This overshooting can be a problem on tight turns if the Scout completely loses the line. Derivative control helps smooth out the Scout's motion.

Derivative control works by calculating the change in error, and changing the speed according to that difference. Again, depending on the magnitude of the error and base speed, you might need a scale term to magnify/shrink the error accordingly. **Note that this scale term might need to be positive or negative according to your program.**

TLDR Formula:

```
right wheel = base_speed - scale * (old_error-error);
left wheel  = base_speed + scale * (old_error-error);
```

Integral control:

Sometimes, if there are outside forces, P and D control are not sufficient to reach a desired position. For example, consider a pendulum attached to a motor. With only P and D control, the force that the motor exerts follows this equation:

```
Force = P_scale * error + D_scale * (old_error - new_error)
```

However, because of gravity, there is another force at play on the pendulum besides that from the motor. When the force that the motor exerts is equal to gravity, the pendulum will stop moving. In order to overcome the force exerted by gravity in this situation, P and D control are not sufficient.

Thus we have integral control, which slowly accumulates all the error over time and changes the speeds based on that sum of error. In the pendulum case, eventually, the integral term will build up since the pendulum is not reaching its desired position and cause the motor to exert more force that will allow it to overcome gravity. Again we need a scale term that can be positive or negative to account for magnitude differences between error measurement and base speed.

TLDR Formula:

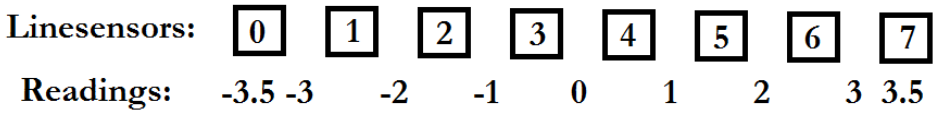
```
right wheel = base_speed - scale * sum_of_errors;
left wheel  = base_speed + scale * sum_of_errors;
```

Step 2: Using the Line Sensor

The line sensor library provides a built in function that tells you 'where' the line is. You can get the line position using the following command:

```
double line_position = linesensor->readline();
```

The resulting line position variable contains a number between -3.5 and 3.5 that corresponds to where the line is in relation to the line sensors. Whole numbers correspond to line positions between line sensors. You can use the following image to get an idea of what line sensor readings mean.



Note: If the line sensor fails to read a line, it returns the last value that was seen.

Step 3: Follow the Line!

The final goal! Write a behavior that will use the motors and line sensors to drive your scout around a racetrack. Try to make the fastest scout. Everyone's code will be demoed on a custom never-before-seen map.

We will demo the line-following behaviors on **11/5/13**. As always with Colony introductory labs, there will be a prize for the fastest code!