Massachusetts Institute of Technology

Robotics: Science and Systems I

Lab 2: Motor Characterization and Control

Distributed: Monday, 2/9/2009, 3pm
Due: Tuesday, 2/17/2009, 3pm

Objectives

In lecture, we learned about motor characterization and motor control: how to modulate the power supplied to one or more electric motors in order to achieve some desired behavior (typically, a specified velocity or velocity profile in time). Your specific objectives in this lab are to:

- Become familiar with reading and modifying Java source code;
- Become familiar with the software version control system used in RSS;
- Understand how to characterize and control an electric motor;
- Read a motor's performance specification, apply power to the motor, derive its rotational speed, and develop a PWM (pulse-width modulation) control algorithm so that the motor can be commanded to run at any desired speed.

This lab will introduce you to developing software under a version control system. You will craft open-loop, proportional velocity, and differential velocity control algorithms for PWM motor control. You will gain physical intuition about how motors behave under load. These skills will enable you to get your robot chassis moving around in a controlled way in the Chassis Lab and the Light Sensor Lab.

Physical Units

We will use MKS (meters, kilograms, seconds, radians, watts, etc.) units throughout the course and this lab. We insist that you do so as well. In particular, this means that *whenever you state a physical quantity, you must state its units*. Also show units in your intermediate calculations.

Time Accounting and Self-Assessment

Before starting the lab, devote a page to each team member on your wiki for "Time Accounting" and "Self Assessment". Neither of these sections will affect your grade. We will use them within RSS to adjust the lab time demands over this term and in future terms. We will use them outside RSS to justify the subject for ABET (national) accreditation.

Whenever you spend a substantial chunk of time thinking about or doing the lab, note the length of time you spent, and the section of the lab you were working on, on your Time page. A good way to do this is to use the time stamp command on the Wiki when you start working and when you finish.

Make a dated entry called "Start of Motor Control Lab" on your Assessment page. Before doing any of the lab parts below, answer the following questions:

Assign a number to describe your proficiency: 1=Not at all proficient; 2=slightly proficient; 3=reasonably proficient; 4=very proficient; 5=expert

- **Motors**: How proficient are you at Motor Characterization and Control?
- **Java**: How proficient are you at programming in Java?
- **Version Control**: How comfortable are you using a Version Control system?

Short statement

- **Other**: Beyond Motors, Java and Version Control, what else do you expect to learn in this Lab?

Materials

In addition to the Motor Control Lab Procedure that you are reading now (Handout 2-A), you should have at hand the following:
Paper handouts:

- Reference Guide to Subversion Source Control
- Motor Specification Sheet (available on course web site)
- Shaft Encoder Specification Sheet (only available on course web site)
- Battery Safety Tips

For Building Rig:

- 2 motors
- 2 wheels
- 2 motor screws
- 1 80/20 aluminum 15" beam
- 6 1/4-20 x 1/2" hex bolts
- 6 1/4-20 sliders
- 8 8-32 x 3/8" hex bolts
- 4 8-32 x 1" plastic hex standoff
- Your MASLab^{TM} ORCboard from the previous lab with connections to battery power and motors.
  Note: You should run the motors **only via the ORCboard with the battery**. *Never* use the recharger without the battery to provide power to ORCboard. It provides insufficient current; this causes incorrect information transmission between the ORCboard and software, and might physically damage the ORCboard.

1 Familiarize Yourself with Subversion Version Control

Working in a team to develop and test algorithms introduces another level of complexity into software engineering. Software version control tools have long been used in industry (and, more recently, open-source communities) to allow many developers to develop code cooperatively. Such software associates a version number and other metadata to every file, enabling code changes to be tracked, merged and reversed. Version control software generally prevents developers from stepping on each other's toes as they simultaneously access a common code base. Good version control tools (such as Subversion) also provide helpful means of to view the history of, and compare different versions of, source files. We have adopted the open-source "Subversion" (`svn`) tools as the RSS software version control system, and have created per-group repositories for team use.

The course staff will present a brief introduction to version control at the start of the lab. See the Reference Guide to Subversion for a comprehensive primer to the most frequently used SVN commands. This reference guide also contains information on how to setup SVN on the lab laptops and your own computers. After hearing the staff presentation and skimming the Subversion guide, complete the following tasks:

1. Have each team member login to the Sun workstation assigned to your team, using their athena user name and default password. Reset the default password provided by the lab staff, by entering in the terminal

   ```
   yppasswd
   ```

2. As a group, when your *first* team member logs in, you will populate your repository with the MotorControl source as follows:

   (a) Open a terminal and make sure that you are in your home by typing in to a terminal.

   ```
   cd ~
   ```

   (b) Check out a working copy of your group's Subversion repository with the following command:

   ```
   svn checkout file:///home/group-(Your group number here)/SVNROOT/RSS-I-group
   ```

   (c) Copy the MotorControl source from `~/RSS-I-pub/labs/MotorControl/` into your working copy of the repository, `~/RSS-I-group`, using a recursive copy command:

   ```
   cp -r ~/RSS-I-pub/labs/MotorControl/ ~/RSS-I-group
   ```

   (type `man cp` for more information). Do not copy the files using the GUI filemanager. Copying from the GUI filemanager will include the .svn dotfiles which refer to our course staff subversion repository. This will confuse subversion and prevent you from adding the lab files to your own subversion repository.

   (d) Use the `svn add` command to add the MotorControl directory to your group repository:

   ```
   svn add MotorControl
   ```

   (d) Commit the changes in your local working copy of the repository to your group's repository on the svn server:

   ```
   svn commit -m "(Your commit comment here)"
   ```

   If you do not add a -m you will be brought to a screen where you can add a message to the first line by first typing `i`. You will then be able to insert text. To quit the program you need to press `ESC` and you will now be typing at the bottom of the window. Enter `:wq` meaning write and quit. The editor you are in is VI, you can learn more commands online.

3. Now that you have populated your group repository with the MotorControl source, each subsequent team member should log in and check out a working copy of your group's repository into their home directory as in 2(a and b) above. Notice that all members now have a working copy of the same revision (version number) of your group repository.

2 Start Your Lab Report

Starting with this Lab, each RSS Lab requires a team-authored Lab Report. Create your report as a new page within your group space on the course Wiki (`http://projects.csail.mit.edu/rss/wiki/`). Title the page with the Lab name and make use of the Wiki formatting syntax to present a clear and organized Lab Report.

As you work through the Lab, add text and upload plots to your Lab Report on the Wiki. You will probably want to save a copy of each image in your `MotorControl/plots/` folder before uploading to the Wiki.

3 Rig Setup

1. Following the example of the staff's example rig and referring to the Lab Rig Layout (Figure 1) build your own verion of the rig. You will need have the motors in place before you screw on the wheels.
2. Hook up one motor on MOT0 which is logically designated the left motor in the software. Hook up the other motor to MOT2. It is logically designated the right motor in the software.
3. Each motor has an internal encoder. Hook up the encoder for the left motor (in MOT0) to J8 of the QPDS inputs. Take care to orient this connector correctly. Black is negative (ground) and red is positive. Repeat for the right motor, placing its encoder into J9, again being careful about the polarity. Do not power on your board until you have visually verified the polarity of these connectors with the example setup.

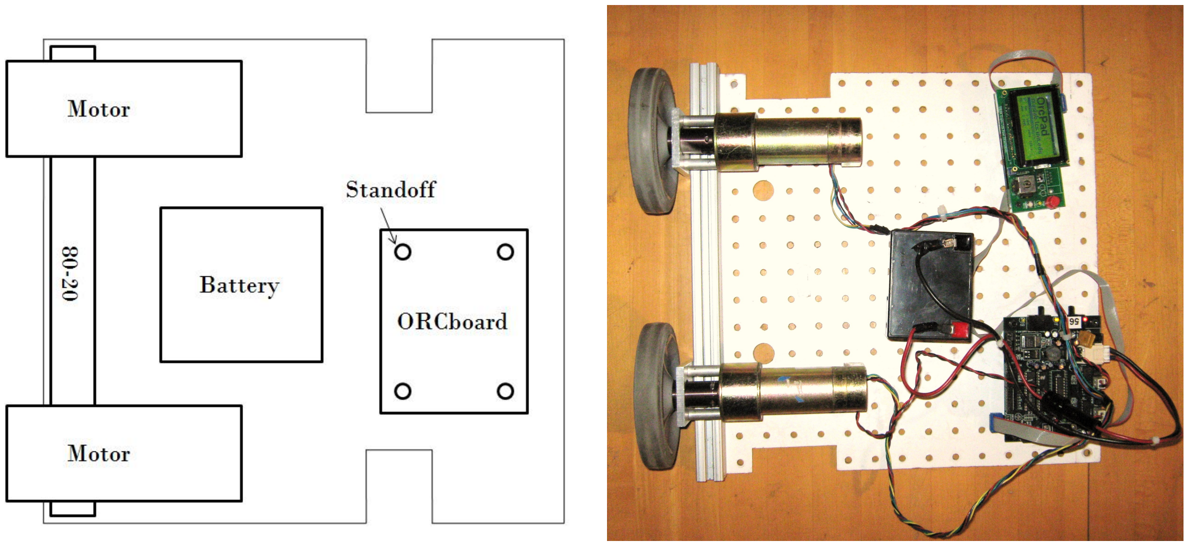


Figure 1: Lab Rig Layout

Deliverables: Put up a picture of your finished rig on your lab Wiki page.

4 Java Source and Plot Scripts

In this lab, you will be working with Java source code to create a motor velocity controller. It is essential that you become proficient with Java. There is an online Java tutorial that can help you do this if you feel you are not yet proficient. It is `http://java.sun.com/docs/books/tutorial/index.html`. The time it requires will vary depending on your existing Java knowledge. As part of this lab, you should complete the following sections of the online Java Tutorial outside of lab hours, or be convinced you have mastery of the principles in the sections:

1. Getting Started
2. Learning the Java Language
3. Essential Java Classes
4. Collections

Before continuing, familiarize yourself with the Java source in `MotorControl/`. Read and understand the classes along with their individual fields and methods. One of the staff will verbally provide a system overview of the RSS motor control software (which includes GUI and data logging classes). Refer to the motor control system overview diagram (Figure 2). IT IS HIGHLY RECOMMENDED THAT YOU DO THIS BEFORE STARTING THE REST OF

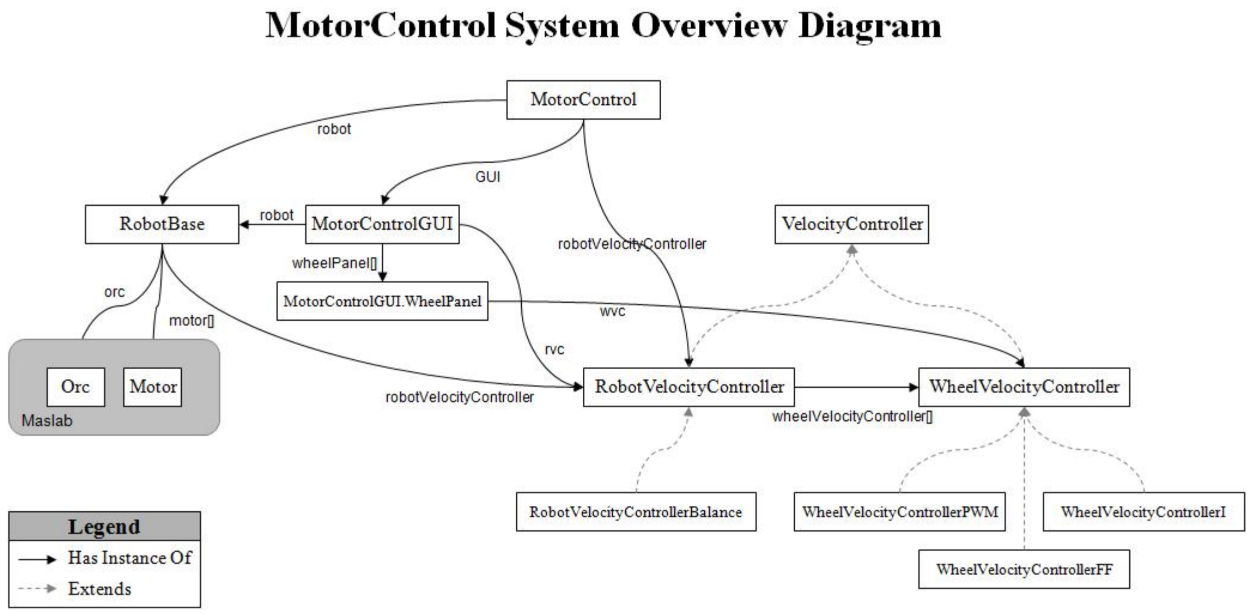


Figure 2: Shows the relationships of various motor controllers used in the lab.

THE LAB. If you understand the structure, your life will be much easier.

You will edit and add code to some of the class files in `MotorControl/`. We have written method templates within classes for you to use. We have placed `// StudentCode` comments throughout the code to mark locations in which we ask you (below) to complete the implementation of various supplied methods. This handout indicates explicitly which methods you will modify. The other classes are provided for comprehension and execution only.

We will use the `Apache Ant` build tool for compiling the Lab source files. In later parts of the lab, you will compile and run the code from `MotorControl/` with the following steps (until then treat these commands as an example):

1. Compile by typing

   ```
   ant
   ```

   in the `MotorControl/` directory. `ant` uses the configuration specified in `build.xml` for compilation.

2. Run by typing

   ```
   java MotorControl.MotorControl
   ```

   to the shell. This command can be used within `MotorControl/` or its parent directory, as both `.` and `..` are included for you in the classpath.

Note: that at this point the code will compile, but not run until you have started the `orcd` described in Part 5.

While you are required to be familiar with command-line Java utilities, you may also compile, run and debug your code within `Eclipse`, the IDE provided for RSS use. The staff will give a brief tutorial on `Eclipse` during the Lab.

For editing, use `Eclipse` or `emacs` or `vim`. (There are other useful applications on your Sun workstation, including a web browser (`mozilla`), `gnuplot`, a pdf viewer (`evince`), and a postscript viewer (`gv`), which can be used for viewing your plots.) The workstation should be used only for RSS-related tasks.

5 Characterize the encoder

1. Look through the ORC Manual in `GeneralInfo/`. Familiarize yourself with the OrcPad interface. Use the white menu button to bring up the Diagnostics screen. Use the black button (aka the stick) to select menu options. If you press the red button you will have to press the stick button to resume. Calibrate the joystick on the OrcPad by holding the menu button down for 5 seconds, which brings up a calibration screen. The battery should be supplying between 12 to 14 volts (check this on the OrcPad display).
2. Configure the OrcBoard. Set the `baud rate` to 115.2 Kbps. Set the `drive mode` in `config` menu to `Lr`.
3. Put the OrcPad in drive mode and test your motors manually while observing PWM, current and joystick position.
4. Plug in the serial cable between the OrcBoard and the **bottom serial port** of the Sun workstation.
5. From the Sun Workstation, start the `orcd` executable (We have added the location `/home/rss-staff/RSS-I-pub/scripts/` to your shell's `PATH` variable) by typing

   ```
   orcd
   ```

   into your shell.
   Only run one `orcd` at a time. Use `killall -9 orcd` to kill all instances of `orcd`. (To check to see if orcd is sucessfully communicating with the OrcBoard, look for the spinning progress indicator in the upper left corner of the OrcPad. **Note:** the OrcBoard cannot establish a serial connection if the OrcPad indicates "all stop.")
6. Make sure that your Orcpad is in the home menu. Run

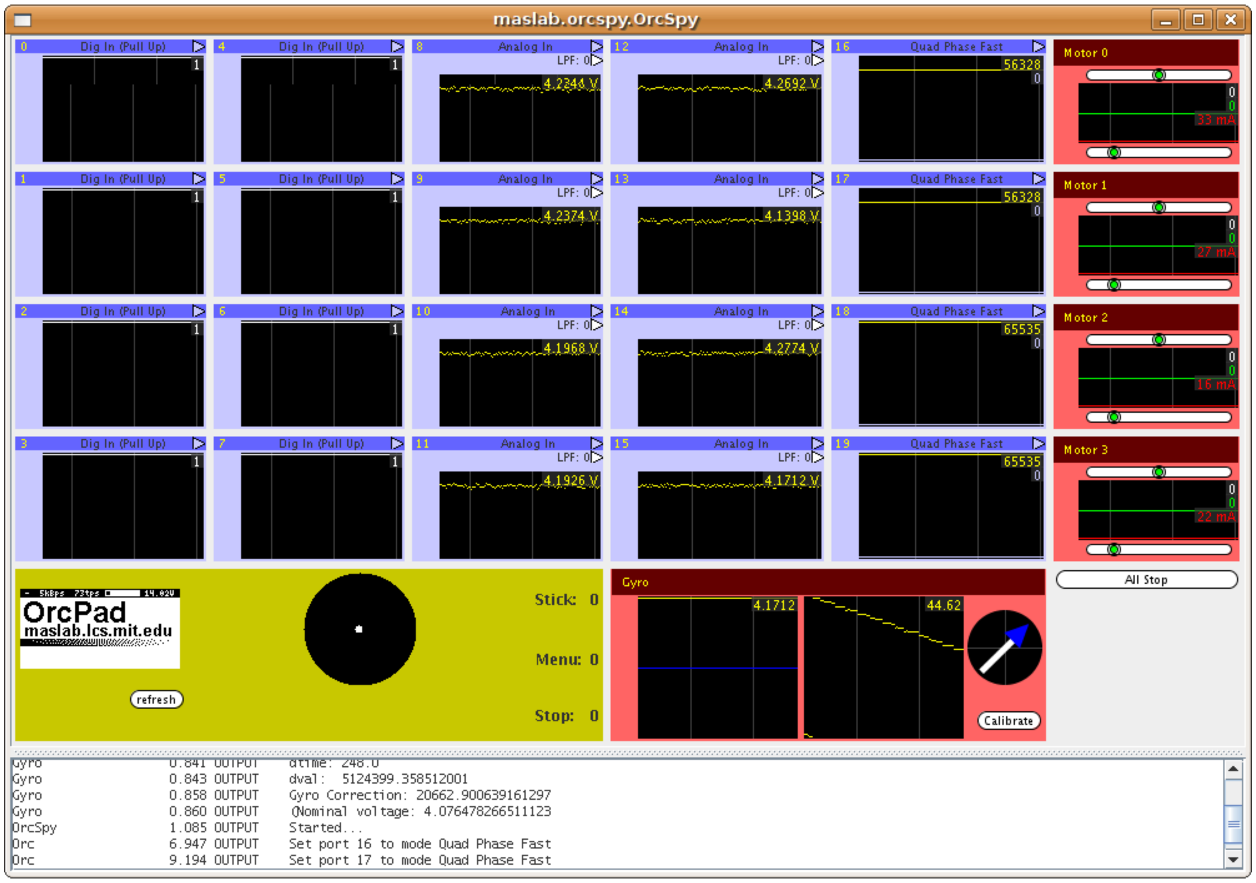
   ```
   orcspy
   ```

   Familiarize yourself with its graphical user interface (GUI). It should look like Figure 3. The `orcspy` executable is also available in your path.
7. Configure orcspy to read from the encoders (set pins 16-19 to Quad Phase Fast).
8. In the rightmost column of menus are the motor displays. They have a pink background. The top slider controls PWM. The plot shows current and PWM. The bottom slider controls slew. Spin the motor shaft back and forth manually by grasping the wheel, and watch the encoder value change. Make a thin pencil mark on the wheel, and turn it as nearly as possible through one full revolution. *How many ticks does the encoder report per shaft revolution?*
9. Calculate this value from the encoder and motor specification sheets. We are using the HEDS-9000 optical encoder module. It is incorporated into the motor package so you will not see it. Don't forget that the motor has a gearhead. *Show your calculations. How closely (to within what percentage) do they match?*
10. Notice that when you drive the motor, the encoder tick reading rolls over periodically. *Why is this?*
11. Measure the motor's maximum (unloaded) angular velocity by counting the number of rotations in a period of time. You can do this visually or by using the encoder output in orcspy. *What is that value?*
12. Add the constants `GEAR_RATIO`, `ENCODER_RESOLUTION`, and `MAX_ANGULAR_VELOCITY` in the appropriately labelled sections of WheelVelocityController.java to reflect the information obtained above.
13. *Derive the radians per tick from the ticks per revolution.*

    In WheelVelocityController.java, complete the implementations of:

    ```
    computeRadiansPerTick();
    computeEncoderInRadians();
    ```

    See the method `update` in the superclass, VelocityController, and notice how subclass WheelVelocityController overrides it.
14. Derive an expression that converts from encoder ticks per second to motor rotational velocity in radians/second. The field `sampleTime` is updated in `VelocityController.update()` and holds the time, in seconds, since the last call to the control loop. Use this information to implement the `computeAngularVelocity()` method.



```
Gyro          0.841 OUTPUT    dtime: 248.0
Gyro          0.843 OUTPUT    dval:  5124399.358512001
Gyro          0.858 OUTPUT    Gyro Correction: 20662.900639161297
Gyro          0.860 OUTPUT    (Nominal voltage: 4.076478266511123
OrcSpy        1.085 OUTPUT    Started...
Orc           6.947 OUTPUT    Set port 16 to mode Quad Phase Fast
Orc           9.194 OUTPUT    Set port 17 to mode Quad Phase Fast
```

Figure 3: OrcSpy GUI

You can test your methods either by inserting diagnostic print statements, or viewing program state within your IDE.

Deliverables: Write up your answers to the questions above (5.8, 5.9, 5.10, 5.11, 5.11) as part of your report on the wiki, and commit your completed methods along with the confirming print statements you added. If your methods are complicated, please leave comments in your code or write pseudocode on the wiki.

6 PWM Motor Control

While working through the following lab parts, you may wish to comment out the print statements you added in Part 5.

6.1 Direct PWM Control

1. You are now ready to actuate, i.e., control the motors via PWM, from the MotorControlGUI that is invoked by `java MotorControl.MotorControl`. This cannot run without `orcd` running in a seperate shell. Note that there is a button set labeled "Input Function" that, at present, does not work when it is set to `man(PWM)` and you move the "Desired" slider. Your goal is to make the "Desired" slider work properly. To do this, complete the implementation of `WheelVelocityControllerPWM.controlStep()`. This method should use the

desired PWM value set for each motor in the MotorControlGUI class and write it to the appropriate field (instance variable) of the class WheelVelocityController.

If you then compile your updated source files and run `MotorControl`, you should be able to control the motor's PWM from the "Desired" slider.

2. Test your routine with the **right** motor pane's "Start Log" and "Stop Log" buttons, while moving the "Desired" slider back and forth. Each Start/Stop pair creates a sequentially numbered file `dataPlotN.txt` in the current directory. An `ls -t` will display files sorted by creation time; use your editor or `more foo` to inspect the contents of the file "foo". Verify that the log files contain data in the correct format by viewing the responsible print statements in RobotBase.java.

   Now note the subdirectory `MotorControl/plots/`. Here you will find a helpful file README.txt. It tells you how to invoke the plotting script *makeplot* which will in turn invoke *gnuplot* to plot data you log, and save the plot to a postscript file (file suffix .ps).

   You must copy or move your data files into `plots` (using `cp` or `mv` respectively) before invoking makeplot. Take care not to overwrite similarly-named files.

   In `MotorControl/plots/`, use gnuplot with preset parameters on your PWM data:by

   `./makeplot Vel_PWM.gp` (Your PWM data)`.txt` (Your output file)`.ps`

   The first parameter, `Vel_PWM.gp`, tells gnuplot the parameters and functions to use when formatting the .txt data.

   To view the resulting plot, use the command `gv foo.ps`, where foo.ps is the filename you provided to makeplot. Verify that the angular velocity tracks the PWM value, and that the velocity values seem reasonable given your observations of the motor's physical behavior. After you get each plot working, use `svn` to add the plot data file and the generated plot.

6.2 Open-Loop Velocity Control

It is better to control the motor in terms of angular velocity rather than PWM. Thus, we would like to code a linear open-loop (no encoder sensing) angular velocity controller. To do this:

1. Compute the average angular velocity when PWM is at 100%. This provides a scaling factor to convert from desired angular velocity to commanded motor PWM. Through the GUI, experiment with different values of the scaling factor by changing the FF-gain.
2. Complete the `controlStep()` method, and the constructor for the `WheelVelocityControllerFF()` class, to implement a open-loop velocity controller making use of the `gain` variable inherited from the `WheelVelocityController` class. You will have to scale the desired velocity with the scaling factor you determined above to convert it to PWM. Test your controller manually, through the GUI, by clicking on the "man(Vel)" button.
3. Select the "step(vel)" button. The "Desired" slider will disappear. Use the "Start Log" and "Stop" buttons to drive the **right** motor through a pre-programmed velocity range and log the results. Plot the test using `FFVel_Unloaded.gp`. Save the log file and plot to your `plots/` directory. *Include the plot, with a caption in your report on the Wiki.*
4. Log another test, this time loading the wheel with your hand. Plot this data with `FFVel_loaded.gp`. *Save and upload the plot as before.*

   Compare the motor's unloaded and loaded behaviors. Comment on the ability of the open-loop controller to track the desired angular velocity. On a mobile robot, when would this type of controller fail? When is a open-loop controller useful?

Deliverables: Write up your answers to the questions above and post plots as pictures on your wiki(you can take a printscreen shot of plot by pressing ALT+PrtSc). Commit your modified source files with the descriptive comment "Motor Control Lab, Part 6."

7 Integral Velocity Motor Control

1. Complete the implementation of an integral gain feedback controller in method `WheelVelocityControllerI.controlStep()`. This method should scale the angular velocity error by the I-gain acquired from the Motor-Control GUI. Run the controller, using the "Desired" slider to command the setpoint. Observe the motor velocities that result. *What happens with a gain that is too low? Too high?* Plot and save your experiments with the **right** motor to show that you effectively varied motor velocity, controller gain, and load. Use *makeplot* with the gnuplot script: `PVel_VariedGain.gp`. *Include the resulting plot(s) in on the wiki.*
2. Tune your controller so that it will be stable, and respond quickly enough for real-world operation. Make it as responsive as you can. Choose the value you think is optimal for I-gain. Use the `Step` function in the GUI to characterize the controller's performance and *describe your controller's performance. If it performs poorly, why do you think this is so?*

Study the step-response of your controller, i.e. bring your motor up to maximum velocity, let it settle there, then command it to zero velocity with your controller. Depending on the I-gain value, there are six distinct convergence behaviors that your system will produce. Vary the wheel gain slider to find them. They are:

1. Overdamped: The system converges, but does so slowly.
2. Critically Damped: The gain is just right; the system converges as fast as possible without going past the desired value (i.e., without overshoot).
3. Underdamped with Overshoot: The system overshoots, but corrects quickly.
4. Underdamped with Oscillation: The system overshoots, but corrects slowly. (This behavior is called "ringing.")
5. Oscillation - The system oscillates around the desired solution forever.
6. Instability - The system diverges from the desired solution.

Sketch each picture on sheet of paper to be handed in and briefly discuss each behavior on the wiki. Record the i-gain value or approximate range of values that produces each behavior. Across what time scale does the behavior occur in each case? Record this information in one the wiki; it will come in handy as you design more complex controllers in RSS I and RSS II.

Deliverables: Write up your answers to the questions on the wiki. Commit your modified source files with the descriptive comment "Motor Control Lab, Part 7."

8 Controlling Two Motors

Consider how to coordinate both motors so that the robot can drive in a straight line. The desired velocity of each motor is the same. The controller should work even if one motor carries more load than the other. Think about the two things your controller is trying to effect simultaneously: (1) that each motor runs at (or near) the commanded velocity; and (2) that both motors run at the same velocity. *Can both of these conditions always be met at the same time? In particular, think about how a large load on either or both motors must be handled by your controller.* There are four cases: (A) both motors unloaded; (B) left loaded, right unloaded; (C) left unloaded, right loaded; and (D) both loaded. *Briefly describe what your controller must do in each case. Is such a controller linear? Why or why not?*

This type of controller would consist at the lowest level of two open-loop wheel velocity controllers you developed in Section 5.2, with one controller designated for the left motor and the other controller for the right motor. It would use the output of these controllers as feedback to a higher level controller that deals with steady state error in terms of the difference in motor outputs. The higher level controller should use *integral control* whose gain acts on the summed difference in motor outputs over time.

Next, consider how to make the robot drive in an arc or rotate in place as well as translate per above. The desired velocity of each motor will be different to enact a turn. For example, if you prescribe more velocity to the right motor, you would expect the robot to turn to the left. Velocity control with a curve offset effectively allows the robot to this. The lab staff can help you develop a block diagram of the desired controller and you have been given well-documented software which indicates where you need to add code to implement one.

Implement this feedback controller by completing the `controlStep()` method in `RobotVelocityControllerBalanced.java`.

Include this in your write up on the wiki. Here are some steps to follow:

Generate a plot of both motor velocities and the differential error while heavily loading first one wheel, then the other. This plot uses `DError_Loaded.gp`.

Deliverables: Write up your answers to the questions above on the wiki, and `svn commit` *your modified source files with the descriptive comment "Motor Control Lab, Part 8."*

Time Accounting and Self-Assessment: After preparing all the hand-in materials, return to the Time and Assessment pages of your wiki. Each of you should tally your total effort there, including time spent writing up your report. Answer the "Self Assessment" proficiency questions again and answer: Beyond Motors, Java, and Version Control, what else did you learn in this Lab? Add the date and time of your post-lab responses.

Presentation

You may put whatever you feel is relevant in your presentation; these are merely suggestions.

- A brief description of what you did, as individuals and as a team
- Demo video of your differential controller working
- Plots from all the experiments
- Any issues and how you dealt with them, problem areas for improvement next year
- Time spent individually and as a team

Upload your presentation onto the wiki.

Report

Suggested content:

- Introduce lab and give context
- Include a discussion of your procedure
- Include plots and explanation for errors
- Explain your code where large parts were written, or include pseudocode

This concludes the Motor Control Lab. The completed code and committed reports (with linked plots) are due by **3pm on Tuesday, February 17th**. To submit your software for grading, commit your code into your group's repository with `svn` as described above. The RSS staff has read permission to all repositories.