Massachusetts Institute of Technology

# Robotics: Science and Systems I

## Lab 3: Motor Characterization and Motion Control

**Distributed: Tuesday 16 February 2010 at 3pm**
**Due: Monday 22 February 2010 at 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:

- Understand how to characterize an electric motor;
- Read a motor's performance specification, apply power to the motor, use sensing to estimate its rotational speed, and develop a control algorithm allowing the motor to be commanded to run at any desired speed.
- Use your characterization to implement a two-wheel motor controller enabling you to command your robot to translate and rotate.
- Establish coordinate systems for the robot and the lab, measure "ground truth" (i.e., the robot's physical motions with respect to some world coordinate frame), and evaluate the accuracy of your robot's commanded motions.

In this lab you will craft feed-forward and proportional velocity (P) controllers for a single wheel 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.

### Physical Units

We will use MKS (meters, kilograms, seconds, radians, watts, etc.) units throughout the course. 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.

# Skill Point Allocation

Each team member should assess his or her skill levels **as of the start of the lab** in each of the following areas, on a scale of 1 to 5 (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?
- **Motion**: How proficient are you at crafting robot motion algorithms using odometry?

Each team member can add his or her self-assessment to the respective Wiki Avatars, in the provided Lab 3 section "Beginning of Lab."

## Materials

In addition to the Motor Control Lab Procedure that you are reading now, you should have on hand the following:
Paper handouts:

- Motor Specification Sheet (available on course web site)
- Shaft Encoder Specification Sheet (available on course web site)

In your bin:

- Tape measure
- Sharpie marker
- Masking tape for marking floor grid

# 1 Characterize the Wheel and Chassis Dimensions

*An attempt at visualizing the Fourth Dimension: take a point, stretch it into a line, curl it into a circle, twist it into a sphere, and punch through the sphere.*
— Albert Einstein

Measure and record your robot's left and right wheel circumferences and its wheelbase (distance between wheels), using the middle of each wheel tread as reference.

Connect the motors to the $\mu$Orc using the following steps:

1. Connect the left motor on MOT0, and the right motor on MOT1 (these are logically designated the left and right motor, respectively, in the supplied codebase).
2. Each motor has an internal encoder. Hook up the encoder for the left motor (in MOT0) to QuadPhase input 0. Take care to orient the connector correctly. Black is negative (ground) and red is positive. Repeat for the right motor, placing its encoder into QuadPhase input 1, again being careful about polarity. Do not power on your board until you have visually verified the polarity of these connectors using the exemplar robot as a reference.

*Deliverables: Create a new Wiki area for your lab materials, just as you did last week. Add a photo of your robot. Record your measurements of the wheel diameter, wheel circumference, and wheelbase.*

# 2 Motor Control JAVA source

On the Sun workstation, one person should first update the source directory `~/RSS-I-pub/` by running the following script:

```
update-RSS-pub.sh
```

You will be asked a password for the account *rss-student*.

Now the person should copy the MotorControl Lab java files (`*.java` and `build.xml`) from the `~/RSS-I-pub/labs/MotorControl` directory to your working copy of your group repository (`RSS-I-group`), in the usual manner:

```
cd RSS-I-group
svn export ~/RSS-I-pub/labs/MotorControl/
svn add MotorControl
svn commit MotorControl -m "added new source for MotorControl lab"
```

Before continuing, familiarize yourself with the Java source in `MotorControl/src/`. 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 1). IT IS HIGHLY RECOMMENDED THAT YOU DO THIS BEFORE STARTING THE REST OF THE LAB. If you understand the structure, your life will be much easier.

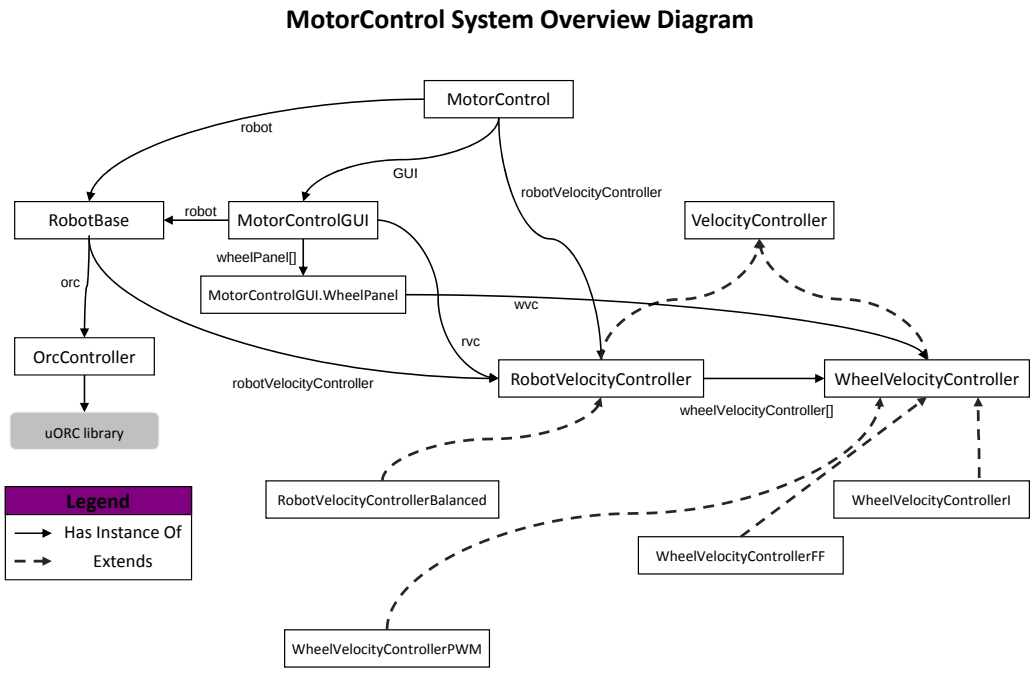


Figure 1: Solid lines and labels show dependency relationships between the classes used in this lab. For example, the main MotorControl class has a field `robot` of class `RobotBase`. Dashed lines show class inheritance relations (`is-subclass-of`) between various motor controllers used in the lab.

# 3 Characterize the encoder

1. Plug in the ethernet cable between the $\mu$Orc and the ethernet port on your laptop. Note: working on the laptop requires you to update your team's svn repository there.
2. Next, we need to initiate the visualizer just as we did in the previous lab. Run

   ```
   orcspy
   ```

   Familiarize yourself with its graphical user interface (GUI). It should look like Figure 2. The `orcspy` executable is also available in your path.

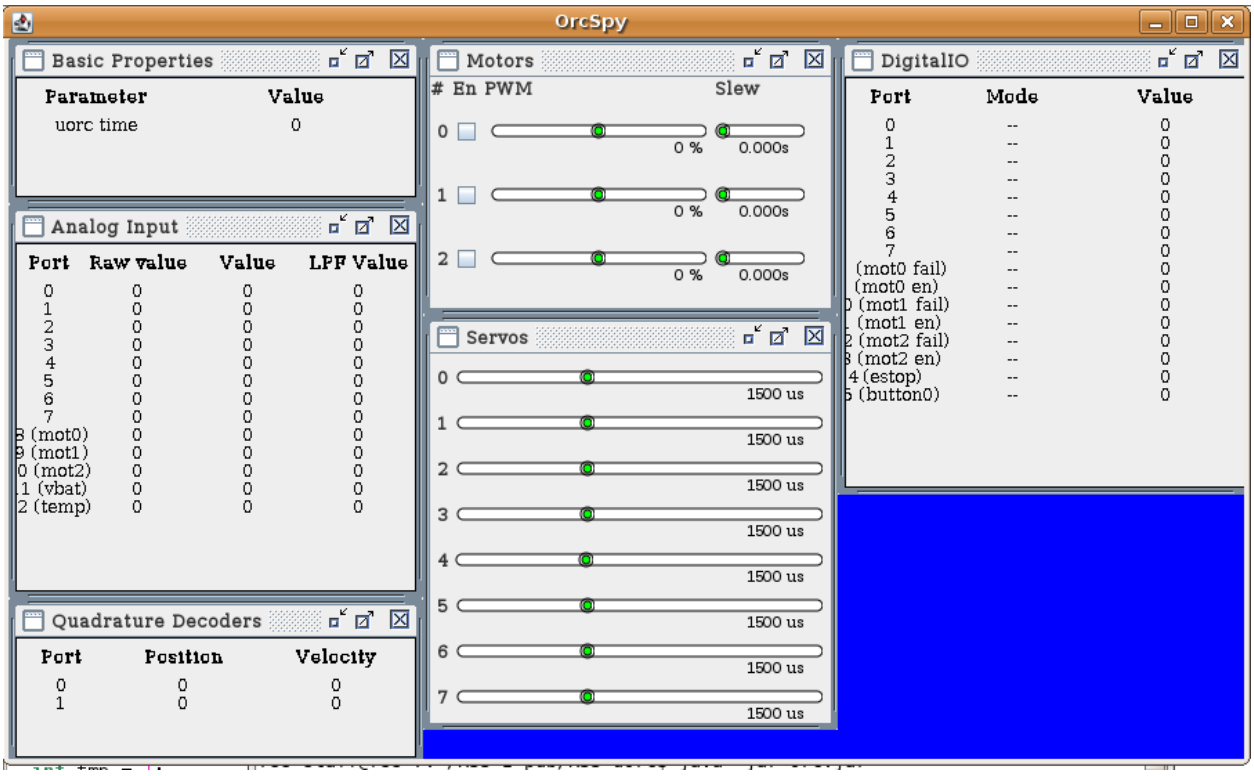


Figure 2: OrcSpy GUI

3. Orcspy should already be displaying its interpretation of the encoders from the quad phase ports on the $\mu$Orc; make sure that you have both motor encoders plugged into the $\mu$Orc correctly.
4. As in the previous lab, the center channels are used for directly modulating the PWM of the motor ports. The window for the two encoder ports is in the lower left of the screen. 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 (calculate via experimentation). *How many ticks does the encoder report per shaft revolution?*
5. Next, calculate the expected ticks per revolution from the encoder and motor specification sheets (calculate via documentation). We are using the HEDS-9000 optical encoder module. It is incorporated into the motor package so you cannot see it directly. Don't forget that the motor has a gearhead. *Show your calculations. How closely (to within what percentage) do they match?*
6. 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?*
7. 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.
8. *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.

9. 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.

Check that you can still compile and run your code at this point. 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 (3.4, 3.5, 3.6, 3.8) as part of your wiki materials. Comment your code, and commit your completed methods along with the confirming print statements you added.*

# 4 Motor Control

*If everything seems under control, you're just not going fast enough.*
— Mario Andretti

For each of the following sections, you will need to update the `MotorControl.java` file to call the appropriate type of Velocity Controller. While working through the following lab parts, you may wish to comment out the print statements you added in Part 3.

## 4.1 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`. 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 utility script `makeplot` and a file `README.txt` explaining how to use it. Try generating a small log file from your java code, then run one of the `makeplot` examples in `README.txt` and view the resulting postscript plot.

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

   In `MotorControl/plots/`, use gnuplot to plot your logged data with staff-supplied plot parameters from file `Vel_PWM.gp`:

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

   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 to the repository.

### 4.2 Feed-Forward Velocity Control

Direct specification of PWM is not a very useful abstraction for controlling a single motor. It would be preferable to specify desired angular velocity. We will start by coding a linear feed-forward (i.e., no encoder sensing) angular velocity controller. (Such controllers are called "feed-forward" since they produce a behavior responsive to a control signal but with no dependence on how the system reacts to load.) 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 for the `WheelVelocityControllerFF()` class, to implement an FF velocity controller using 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 this plot, with a caption, in your wiki materials.*
4. Log another test, this time subjecting the wheel to variable loading with your hand. Plot this data with parameter file `FFVel_loaded.gp`. *Save and upload the plot as before. Compare the motor's unloaded and loaded behaviors. Comment on the ability of the FF controller to track the desired angular velocity. On a mobile robot, When would an FF controller be useful? When would this type of controller fail?*

*Deliverables: Write up brief answers to the questions above and post plots to your wiki (you can use* `ps2pdf` *to convert your postscript plots to PDF, or screen-capture any display by pressing ALT+PrtSc). Add a comment line reading "Motion Control Lab, Part 4" to the top of any code block that you modified.*

## 5 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) on the wiki.*
2. Tune your controller to be stable and sufficiently responsive for real-world use. 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. 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.

*Vary the wheel gain slider to find the approximate gain regimes in which each behavior occurs. Sketch each behavior on sheet of paper. Photograph your sketches and post them 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 on the wiki; it will come in handy when you design more complex controllers later in the class.*

*Deliverables: Add brief answers to these questions to the wiki. Commit your modified source files with descriptive comment lines reading "Motor Control Lab, Part 5" as above.*

# 6 Controlling Two Wheels

Consider how to coordinate both motors so that the robot drives in a straight line, so that the velocities at the rim of each wheel are equal. Under our earlier-stated assumption that the wheels have equal radii, this requires making both wheel motors turn at the same angular velocity. Moreover, our controller should do the right thing even when the motors are subjected to varying loads. Think about the two things your controller is trying to effect simultaneously: (1) each motor runs its wheel at (or near) the commanded speed; (2) both wheels propel the robot along at the speed. *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?*

Your two-wheel controller would consist at the lowest level of two integral angular velocity controllers you developed in Section 5 (scaled to wheel rim velocity), with one controller designated for the left wheel and the other controller for the right wheel. 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 wheel speeds. The higher-level controller should use *proportional control* with a gain term multiplying the summed difference in wheel speeds.

Next, consider how to make the robot drive in an arc, rotate in place, or translate perpendicular to its wheel axis. Turning requires the desired speed of each wheel to differ. For example, if you command more speed from the right motor, you would expect the robot to turn to the left. Two-wheel speed control with an offset effectively allows the robot to do 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`.

Briefly describe your efforts on the wiki. Here are some steps to follow:

Generate a plot of the speeds of both motors, and the differential error, while heavily loading first one wheel, then the other. This plot uses parameter file `DError_Loaded.gp`. What are the corner cases? Does your code handle them elegantly?

*Checkoff: Demonstrate your working feedback controllers to the LA's before continuing.*

# 7 Establish Body and World Coordinates

You will now start tracking the robot's motion. To do so, you will first need to establish a 2D "body coordinate frame" for your robot. This frame should have its origin $O = (0, 0)$ at the ground point directly below the midpoint of the robot's drive wheel axis. Your frame should be right-handed, with unit vectors $\hat{x}$ pointing toward the front of the robot and $\hat{y}$ pointing to the left as the robot is viewed from behind. Carefully mark $O$ and (the directions of) $\hat{x}$ and $\hat{y}$ on your robot chassis with masking tape and a sharpie.

Measure and record the positions, in body coordinates, of the two drive-wheel ground contact points, the two caster wheel contact points, and the four corners of the robot's chassis. Make a sketch the robot frame, wheels, motors and casters with each of these points $(x, y)_b$ labeled in the body frame. You will use this information later in the lab to recover the robot's actual displacement and orientation, or "pose" $(x, y, \theta)_w$ in world coordinates.

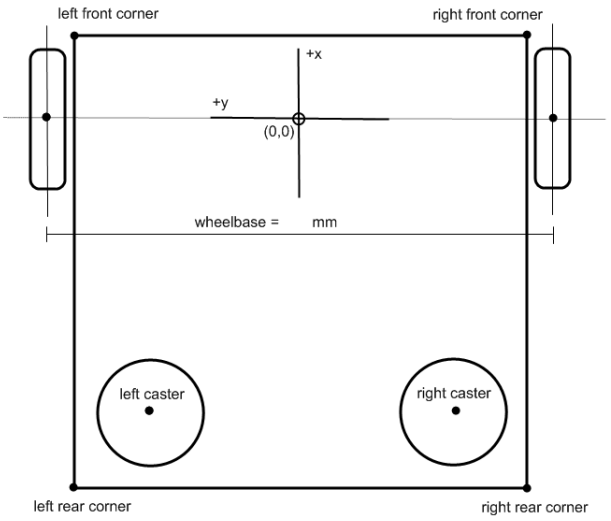


Figure 3: Robot body coordinates.

The staff have already marked at least one two-meter square on the floor, in masking tape. This will be the robot's "world coordinates". You may use this square if it is not already taken (please coordinate with other teams to share access). If there are not enough squares, make your own with masking tape and a tape measure. Take care to make your right angles true, by checking that the square's diagonals have equal length. Subdivide the square into four one-meter squares.

*Deliverables: Post a diagram of your robot to the Wiki. The diagram should include positions, in body coordinates, of all the components mentioned above.*

# 8 Translate a Specified Distance

Fill in the `translate` method in `RobotPositionController.java` to command your robot to move forward or backward at a specified constant velocity for a specified distance in meters, relative to its current pose. *Document your code with comments as you work.* Remember that this velocity is a *linear* measure with units in `m/s` (versus rotational velocity with units of `radians/s`). You may choose to implement either *feed-forward* or *feedback* control. In any case, your implementation of `translate` should *block*, i.e. it should not return until the translation is considered complete or an un-recoverable error has occurred. The robot should not be moving after `translate` returns.

1. The velocity controller developed in the previous lab expects to see angular velocity of the wheel. Define a constant that converts linear translational velocity of the wheel on the ground to the equivalent (wheel) angular velocity. It should use the wheel circumference, which you measured above.
2. The (signed) number of encoder ticks which have occurred since your program started running will be available in instance variables (read the code to find them). Define another constant that converts ticks to linear translation in meters.
3. It should be possible to implement feed-forward control entirely within the `translate` method. To implement closed-loop control, you will probably want to write code both in the `translate` and `controlStep` methods. Note that `controlStep` and `update` are both called from the control loop thread for your robot (defined in `RobotBase.java`), and that this is a different thread than the one that calls `translate`. Since states, like encoder tick variables, can be accessed from both threads, you will likely need to use the `synchronized` keyword to reliably read and write states. The synchronized keyword locks a method, allowing only one thread at

a time to access it.

4. It is up to you to define what it means for the motion to have "completed" or "errored". One way to do this is to define an error tolerance (in the appropriate units) for the robot position, and to periodically check that the robot is within tolerance of where you expect it to be. When you expect it to be in the goal position (i.e. the position it should have at the end of the translation) and your code concludes that it is indeed within the defined tolerance of that position, you can consider the motion complete.
5. Wheel slip (e.g. skidding) introduces odometry errors (why?). If the wheels slip, the robot's motions will be less accurate, and pose error will accumulate. To reduce wheel slip, you can limit the robot's acceleration. One way to do this is to drive very slowly; a preferable way is to explicitly ramp up the commanded speed, over a specified distance, at the start of each motion, and then ramp down the commanded speed at the end of the motion.
6. Having the right parameter values (e.g. control gains) makes the difference between a solution that works great and solution that doesn't work at all. Finding the right parameter values can be hard work. You can make this job much easier by building a user interface for adjusting parameters and viewing the internal state of your solution. You can see how to do this by examining the source code for the MotorControlGUI and OrcSpy applications.
7. To measure the actual pose, consider the body frame you established earlier, and work out how a triplet of numbers $(x, y, \theta)_w$ describes the robot's pose (position and orientation) in world coordinates. Given two measured points $(x_L, y_L)_w$ and $(x_R, y_R)_w$ describing the robot's left-wheel and right-wheel ground contacts, respectively, write expressions for the three components of the robot's absolute pose in world coordinates, i.e.:

$$\begin{aligned} x_w &= x(x_L, y_L, x_R, y_R) \\ y_w &= y(x_L, y_L, x_R, y_R) \\ \theta_w &= \theta(x_L, y_L, x_R, y_R) \end{aligned}$$

8. Place your robot so that its initial pose $(x, y, \theta)_w$ is $(0, 0, 0)$ with respect to marked lab coordinates. Using calls to `translate` at the indicated place in `Chassis.java`, issue a series of translation commands varying in sign, duration and speed. (You may want to adjust the controller gains, which are set by code in that file). *Measure the robot's actual final pose $x, y, \theta$ (translation and orientation) in world coordinates. Plot the your relative (%) error for 3 trials using* `gnuplot` *. Explain any significant errors.*

**Hint** Log your hand-measured $(x_L, y_L, x_R, y_R)$ values to an ascii file. You may use whatever software you like to produce three error vs. trial number plots. If using gnuplot,write gnuplot expressions to compute $x_w$, $y_w$, and $\theta_w$ from each data entry. The differential error gnuplot file from the Motor Control Lab shows how to express algebraic expressions in gnuplot; the terms \$5 and \$4 in the "`plot`" command refer to the fifth and fourth column of `data.txt`, respectively. Add the line `set angles radians` somewhere above your `plot` command, and use the built-in gnuplot function `atan2` (google gnuplot atan2 for more info). Note that the backslashes in the `plot` command are required when splitting plot descriptors across multiple lines.

*Deliverables: Post any plots and explanations on the wiki, with a brief discussion of the procedure you followed. Document and commit your java code and any* `gnuplot` *scripts you wrote or used. Be prepared to demonstrate your robot's motion in lab.*

# 9 Rotating in Place

1. Fill in the `rotate` method in `RobotPositionController.java` to to rotate your robot about its origin at a specified speed, relative to its current pose. Think about how to implement counter-clockwise and clockwise rotation commands, using your knowledge of the wheelbase and wheel radii. As with `translate`, `rotate` should block until the motion has completed or errored, and the robot should not be moving after it returns. Again, you may implement either feed-forward or feedback control.

**Hint** Convert the desired robot rotation speed to a desired linear or angular speed for each wheel, and the desired robot rotation angle to a linear or angular distance for each wheel.

2. By analogy to Part 8, measure the robot's actual final pose in world coordinates for a series of three commanded rotations differing in sign, duration and speed. (Include at least one turn of $\pi$ or more radians.) *Plot the your relative (%) error for 3 trials using* `gnuplot`. *Report any undesired translations and/or rotations and try to explain them.*

*Deliverables: Include a brief discussion of your procedure on the wiki, along with any plots and error explanations. Document and commit your java code and any* `gnuplot` *scripts you wrote or used. Be prepared to demonstrate your robot's motion in lab.*

# 10 Driving Out and Back: Error Upon Return

*Few can forsee whither their road will lead them, until they come to its end.*
— J.R.R. Tolkien, *The Lord of the Rings*

1. Write code in `Chassis.java` to command your robot to drive "out and back:" roll straight one meter ahead, rotate $\pi$ radians, roll straight one meter ahead, and rotate $-\pi$ radians. In other words, the robot should take a two-meter excursion, returning to its starting pose. You can specify the speed at which, or time duration over which, your robot should execute this excursion. (Try a few different values.)
2. Before running your method, use your results from Part 8 and Part 9 above to predict what your sensed and measured final poses will be after execution. *Post your predictions on the wiki.* Now execute your program and measure the resulting pose. *How close are your predictions to reality?*

*Deliverables: Include your answers to questions above, explanations, and a brief discussion of your procedure in your lab report. Be prepared to demonstrate your work in lab on the due date marked at the top of this document.*

# 11 Driving in a Square: Error Upon Return

1. Write code in `Chassis.java` to command your robot to drive around a square with one-meter side-length. Experiment with a variety of commanded translation and rotation speeds.
2. *Predict and measure the robot's final pose as above.* Record your results. *How reliably does your robot return to its starting position and orientation?*
3. Use the RSS video camera (or your own) to capture a short video of your robot's excursion (max size 10MB or as directed by lab staff).

*Deliverables: Discuss your robot's performance. Post your video. Be prepared to demonstrate your work in lab on the due date marked at the top of this document. Also, be sure to* `svn commit` *your modified source files into your group's repository with the descriptive comment "Motion Control Lab, Part 11."*

## Wiki Materials

Check that you have posted to the wiki:

- Any notable procedures you employed;
- A brief description of any major code edits;
- Plots for §4 through 6 and explanation for errors.
- Discussions of pose errors for §8, 9, 10, and 11.

## Briefings

You may brief whatever you feel is relevant; these are merely suggestions.

- A brief description of what you did, as individuals and as a team;
- Brief video of your differential controller working;
- Brief video of your robot driving in a square;
- Plots from a salient subset of your experiments;
- Any issues and how you dealt with them;
- Time spent individually and as a team; and
- Any suggestions for improving the lab next year.

*Post your briefing on the wiki.*

Your committed code and Wiki materials (with your brief responses to the lab questions, as well as your images, plots, and videos) are due on the date/time marked at the top of this handout. To submit your software for grading, commit all files into your group's repository under a top-level directory named "MotionControl". Make sure that a fresh checkout, which the staff will perform, reproduces your code in its entirety.

**Wiki Avatar Updates:** After preparing all the hand-in materials, return to your team's Wiki Avatars. Each of you should tally your total effort there, including time spent writing up your report, under your Saved Progress section. Additionally, be sure to answer the proficiency questions again **as of the end of the lab**, and place your self-assessments into the "End of Lab" portion of the lab 3 section.

## This concludes the Motor Control Lab.