Lab 4: Coordinated Feedforward Control with Feedback Trim for Simple Trajectory

1 Introduction

This week we will learn how to execute simpler motions with even greater precision. Next week, we will extend that to any motion whatsoever.

2 Motivation

A first course in control always includes the PID feedback control loop and may not go far beyond that. In practice, however, feedforward control is often the secret to high performance. This lab will hopefully teach you a memorable lesson in the big picture of control.

3 Exercises

3.1 Warm Up Exercise 1: Basic Proportional Control

Implement a proportional feedback controller (position error and velocity commands) to drive the robot a fixed distance and come to a stop. You will use the encoders to decide when to stop, but unlike in the first lab, you will also make sure the robot is stopped at precisely the right place using feedback.

Implement a full PID servo. It is only a few lines more than a P servo. Recall that the formula is:

$$u_{pid}(t) = K_p * e + K_d * \dot{e} + K_i * \int e\, dt \qquad \textbf{Eq 1}$$

Initially, use $k_p = 10.0$, $k_d = 0.0$, and $k_i = 0.0$. These gains are defined for error and speed commands measured in arbitrary but consistent units of distance (e.g. cm and cm/s or m and m/s). Terminate the loop when the absolute value of the position error with respect to the goal position is less than 0.1 mm or when 6 seconds have elapsed – whichever happens first. Limit the output velocity magnitude to 0.3 m/sec. Set the goal distance to 0.1 meters.

3.1.1 Step Response in Simulation

With your PID controller running, command the simulated robot provided in the Raspbot class to move 0.1 meters forward **or backward** with respect to wherever it is when the command is started. Plot the **distance error** (in m, relative to the goal position) versus **time** in seconds (not the array index in the time array).

Go no further until this is working. Don't assume the wheel encoders starts at zero because they will not do so on the real robot. They may start close to zero - which is even worse because you otherwise may not notice the error. If the simulator simulates a time

delay for the wireless (if it does, you will see it in the feedback signal), you may have to reduce the proportional gain to avoid too much oscillation.

3.1.2 Step Response on Robot

a) Now command the real robot to do the same thing. You should see some oscillation or at least a single overshoot before the robot returns to the right position; if not, increase the proportional gain until you do. A little overshoot is ideal if you want to get there as fast as possible.

Please minimize the wear and tear on the robot. If the robot oscillates out of control, pick up the robot and reduce the gain for the next attempt. Running with the wheels off the ground will give totally different behavior, so you generally cannot tune the controller properly if you do it when the wheels are off the ground.

The command being used here is a classical "step" input and the graph you get may look like the "step response" of a second order system. Look it up online if you are not familiar with it.

Again plot the error (in m) versus time. It probably looks a lot different from the simulation.

b) Rerun the simulation with the gains used to induce oscillation on the real robot.

c) Change nothing except pick the robot up and turn it over. Run it again and plot the error. Think about what is going on.

3.2 Warm Up Exercise 2: PD Control

Put the robot back on its wheels. Leave the proportional gain alone while running the case that oscillates. Slowly increase the derivative gain until you get best performance (minimum overshoot and rapid convergence).

Plot the error versus time for your best result. One you have this most aggressive behavior, you may notice that it is pretty close to going unstable at times. In that case, reduce the gains a little for general use. The idea was just to make a point about derivative control.

3.3 Warm Up Exercise 3: Open Loop Control with Identified System

3.3.1 Code Feedforward Control

Implement a continuous time **velocity** reference trajectory composed of a short acceleration, a constant high speed section, a short deceleration, and a null velocity at the beginning and end:

$$u_{ref}(t) = \begin{cases} a_{max} * t\,, & t < t_{ramp} \\ a_{max} * (t_f - t)\,, & (t_f - t) < t_{ramp} \\ v_{max}\,, & t_{ramp} < t < t_f - t_{ramp} \\ 0\,, & otherwise \end{cases} \qquad \textbf{Eq 2}$$

Implement it as an actual MATLAB function because you will shortly need it twice.

```
function uref  = trapezoidalVelocityProfile( t , amax, vmax, dist, sgn)
%uref Return the velocity command of a trapezoidal profile.
%   Returns the velocity command of a trapezoidal profile of maximum
%   acceleration amax and maximum velocity vmax whose ramps are of
%   duration tf. Sgn is the sign of the desired velocities.
%   Returns 0 if t is negative.
```

Set $v_{max} = 0.25\ m/s$ and $a_{max} = 3 * 0.25\ m/s^2$. Then $t_{ramp} = v_{max}/a_{max}$[1]. The distance should be **1 meter.**

This speed command will be called the *feedforward* term. The PID controller you did in earlier exercises is the *feedback* term. Eventually, you will send the **sum** of both terms to the robot.

Get the sign right if the goal is behind the robot. Based on the desired distance to travel, one can derive (from the velocity and the distance) a general closed form solution for t_f as follows. Note that the average velocity in both ramps is just $v_{max}/2$ so the distance travelled for both ramps is just $v_{max}t_{ramp}$. The distance travelled during the constant velocity part is simply $v_{max}t_{const} = v_{max}(t_f - 2t_{ramp})$. So, the total distance is:

$$s_f = v_{max}t_{ramp} + v_{max}(t_f - 2t_{ramp}) = v_{max}(t_f - t_{ramp}).$$

But $t_{ramp} = v_{max}/a_{max}$ so $s_f = v_{max}(t_f - v_{max}/a_{max})$. This means that:

$$t_f = s_f/v_{max} + v_{max}/a_{max} = \left(s_f + \frac{v_{max}^2}{a_{max}}\right)/v_{max}.$$

Use positive signs for everything and switch the sign of u_{ref} if $s_f$was originally negative. Of course, the sign of t_f is always positive.

You will also need the integral of the **reference** velocity so integrate u_{ref} with respect to time **numerically** as you did in earlier labs, to produce $s_{ref}(t)$. You could do this integration in closed form here but you will need it to work later on the robot when the velocity is an arbitrary signal, so integrate it numerically instead. On the other hand, in this case, you could also figure out the right answer in closed form just to make sure that your numerical integration is working correctly.

Because there is value in having both running at the same time, it will be best to implement this reference trajectory **integrator** separately from any **simulator** you might have developed - although the code is almost identical. Verify that the integral achieves the right value of position at t_f. You will want to integrate the reference u_{ref} based on the **actual elapsed time** when you are controlling the robot so you may as well do so in simulation so that you can debug the real algorithm in simulation.

For this exercise, implement **only** the feedforward term in a controller based on true elapsed time (as you did in the last lab). That type of control is called *open loop* control. You will want to implement this by changing just a few lines of your PID controller exercise code. Leave the commented out PID control computation in place because you

[1] If there is not time to achieve maximum velocity, set t_{ramp} to $t_f/2$.

will need it shortly. Compute both the reference distance (integral of the reference velocity) and the actual distance...

3.3.2 Plot Reference Velocity and its Integral in Simulation

Run the feedforward control with the simulator and plot the reference distance and the simulated robot speed versus elapsed time on the same graph.

3.3.3 Plot Reference Distance and Actual Distance on Real Robot

When it is all working, run the feedforward control on the real robot and save the graph of both the reference **distance** and the actual **distance** versus elapsed time on the **same** graph from the real run.

You should see a delay and perhaps other things in the signals from the real robot. The robot should be able to execute the reference acceleration and velocity. If not, reduce them until it can.

Now "identify" the system by guesstimating the delay from the graphs. When this delay is correct the average error on the whole trajectory should be close to zero. Then alter your code to also compute a **delayed** reference (called $u_{delay}(t)$) and (numerically) its integral $s_{delay}(t)$. That is, set $u_{delay}(t) = u_{ref}(t - t_{delay})$. The delayed velocity will be zero until the initial delay is elapsed. You may need to make that happen if you did not code the formulas you were given.

3.3.4 Synchronize (Delayed) Reference Distance and Actual Distance on Real Robot

Redo the last experiment **using the delayed reference velocity and its integral** and plot the result as two graphs. The first graph is the delayed reference distance and the feedback distance on the same graph. The second graph is the difference between the two as a function of time. Be extra careful that this is done right because the challenge task depends on it. You will still send the **non-delayed** velocity **command** $u_{ref}(t)$ to the robot BUT you will integrate $u_{delay}(t)$ and plot $s_{delay}(t)$ and compare **them** to the **feedback**. The idea is to get the delayed reference distance and the real distance feedback to match as closely as possible. The better they match, the higher performance your controller will exhibit. Terminate your loop based on waiting for the ideal execution time, the delay, and an extra one second to allow things to settle. Don't forget to plot the result.

When all that is working go to the next step.

3.4 Challenge Task: Feedforward Control with Feedback Trim

Now, the point of all this work is that feedback alone knows nothing about the system it is controlling so it does not do very well. Conversely, feedforward knows the system generally but not what it is doing at this moment. A combination of both is a better approach. Enable both feedforward and feedback and send a command to the robot of the form:

$$u(t) = u_{ref}(t) + u_{pid}(t)$$

Where $u_{pid}(t)$ is the feedback term based on measuring the position error as it was defined in the last graph. In other words, it is critical that you change the error calculation of the PID controller as follows: now the error is defined as the difference between where the robot actually is and where it should be based on the **delayed reference trajectory** $s_{delay}(t)$. In principle, if there were no error (other than unavoidable delay), the feedback should be exactly $s_{delay}(t)$, and the feedback controller will have to do nothing at all to remove error – because there will be no error. You may find that it is possible to increase the gains in your PID controller in the first exercise in order to remove (the now smaller) errors even faster.

Plot, as you did in the last exercise, the delayed reference position and the feedback position on the same graph. Also plot the difference between the two ("error-from-the-reference") as a function of time. Hopefully, this is your best performing controller – particularly at the endpoint. That extra second will make all the difference at the endpoint.

4 Graded Demonstration and Report

Only this part is graded. Points are included in square braces [thus].

Demonstrate the challenge task on your robot for a travel distance of 1 meter. Do it first for feedforward only and a second time with both feedforward and feedback. Make sure you can switch between the two by changing one variable. Plot[2] the reference distance and the feedback distance versus time in one window (both cases simultaneously in one graph in the same window) and plot error versus time in a second window. This lab is testing control so the overshoot and error below are **as computed from the feedback**.

Make sure that your PID runs for 1 second longer than the period for which delayed reference trajectory is nonzero so that it can resolve any final errors.

[5] Switch between two cases with one variable change.

[5] Graphs are correct shape in both cases.

[5] Motion is mostly smooth and steady in feedback case with overshoot < 5 mm.

[25] Terminal error **one second after end** of delayed reference trajectory for fbk+ffwd case. 25 points if < 1 mm. 20 points if < 2 mm. 15 points if < 3 mm, etc.

[2] All graphs produced for labs in this course should have (1) a title and (2) axis labels that include (3) a description of the units. Plot different signals on the same graph using (4) different colors with (5) no point markers (triangles, squares, etc.). Use a (6) legend to describe which is which. Failure to comply with these rules will result in a 20% reduction in your grade. Whenever possible, produce graphs in real time as the robot moves, but its OK to plot results at the end of the test if real-time graphics would otherwise affect performance.

4.1 Report

~~Bring your report to the lab period and hand it in **before** the lab period starts. In it, include the content and answers to the questions provided below and any others in grey boxes above:~~

~~[2] How does the error derivative (the thing multiplied by k_d in the PID controller) relate to velocity error when the reference command is constant?~~

~~[2] What does the formula for u_{ref} do when you overshoot? Is it a good thing?~~

~~[2] What happens in execution to the integral part of feedback term if you exceed the true maximum robot velocity in the reference trajectory – meaning the robot cannot achieve the reference speed ?.~~

~~[2] What would happen at the slow down ramp in the above case of a too fast reference?~~

~~[**5 bonus**] It would be possible and just as good on these robots to drive to the end of the reference and then switch on aggressive high gain feedback term to zero in on the final position. However, this will not work well on a high curvature path. Why?~~

~~[**5 bonus**] The delay you see has two parts to it – uplink and downlink (to and from the robot). This means when the encoders say you are at the goal, you are likely already slightly past it. What can you do to mitigate each of these issues?~~

5 Notes and Hints

1. It's best to turn the robot over on your desk and test that the velocities look reasonable before doing a real test flight. Do this to avoid driving the robot with gross errors. Note the comments above on tuning though.
2. The robot jumps a lot when you give it a step command. The castors are supposed to prevent it doing itself any harm.
3. It helps to turn features on and off with switch variables and conditionals. For example, for feedforward and simulation, you might have:

```
if(doReal == true)
    sendVelocity(robot,control,control);
else
     left = left+control*delTime*1000.0;
     rght = rght+control*delTime*1000.0;
end
```

   If you get these working once, you will not have to remember what lines to comment in and out every time you want to switch back or forth and that matters because many lines I many places may depend on the switch variable.

4. Make the robot go forward and backward alternately to keep the robot near you while testing. One way is to enter mySign = 1 at the control window at first, and then insert the following at the top of your code:

```
goalState = L * mySign; % move distance L fwd/bwd
mySign = - mySign;

Better yet, use the following to detect the first
execution automatically:
if(~exist('mySign','var'))
    mySign = 1;
end
```

5. I find it convenient to allocate a few plotting arrays with names like PlotArray1, PlotArray2…and then change the one line of code that decides what to put in them. That way you do not have to change the names in two or more places every time you change the contents of the graphs.
6. Later on, you may want to use the trapezoidal ramp for small motions. In that case, check the second (ramp down) condition first so that the trajectory works on very short segments. In that case, you may also want to adjust the max velocity to be no larger than the speed for which the trajectory takes one second to execute. And make the acceleration equal to the max velocity change in one second. These measures avoid a high energy jolt that moves the robot too far before it even gets the motion under control.