Lab 0x03 - Closed Loop Control

New Attempt

- Due Oct 30 by 12:10pm
- Points 20
- Submitting a file upload
- File Types pdf and py

Overview

The primary goal of this assignment is to develop a closed-loop velocity controller to operate the gearmotors on the Romi chassis; a secondary goal of this assignment is to implement the closed-loop controller in a parameterized manner so that it can be repurposed and reused in the future labs or in the term project with little or no modification. This assignment is to be completed in lab groups and is meant to take one week.

Background

In this lab we will focus exclusively on classical control theory, which deals primarily with single-input-single-output (SISO) systems. In lecture we may discuss modern control theory which is better suited for multi-input-multi-output (MIMO) systems. In either case, the essence of control theory is the *feedback law*, sometimes called the control law, which is what allows a closed-loop controller to achieve suitable tracking and disturbance-rejection performance.

The purpose of a control law is to take in a reference command and then modulate the effort supplied by an actuator such that the output of the system matches closely to the reference command. For example, a velocity controller will take in a reference command of a desired velocity and then modulate motor effort in such a manner that the motor operates at or near the desired reference velocity. This is achieved by using *negative feedback*.

With negative feedback, the measured output of the system is subtracted from the reference value to compute an "error" value which is then fed through a controller or "compensator" that amplifies the error to be used as an effort command to the actuator. In this way, once the system achieves the desired setpoint the error drops to zero. In real-world applications a control engineer should *not* expect zero error to be achievable. Instead, the engineer should assess the performance of the controller through various performance metrics.

PID Control

Typical control laws are of the form P, PI, PD, or PID, where "P" stands for proportional, "I" stands for integral, and "D" stands for derivative. A proportional-integral-derivative (PID) controller is shown below in a typical feedback loop. These three aspects each apply to how the error signal is converted into an actuation signal. The proportional component is most important and may be sufficient to achieve desired performance without the integral and derivative components, but typically maximum performance can be achieved by including integral and/or derivative action, and for some systems one or the other may be required for stability.

Consider the block diagram shown below; this diagram depicts a stand PID controller with no extra features included.

- The signal r is the setpoint or reference; often the setpoint is set to zero for "regulator" type controllers, but more commonly the setpoint is a nonzero value; if the setpoint changes over time the controller must be tuned to allow good tracking performance as controller performance will be different for regulation, constant setpoints, and moving setpoints if tuned the same.

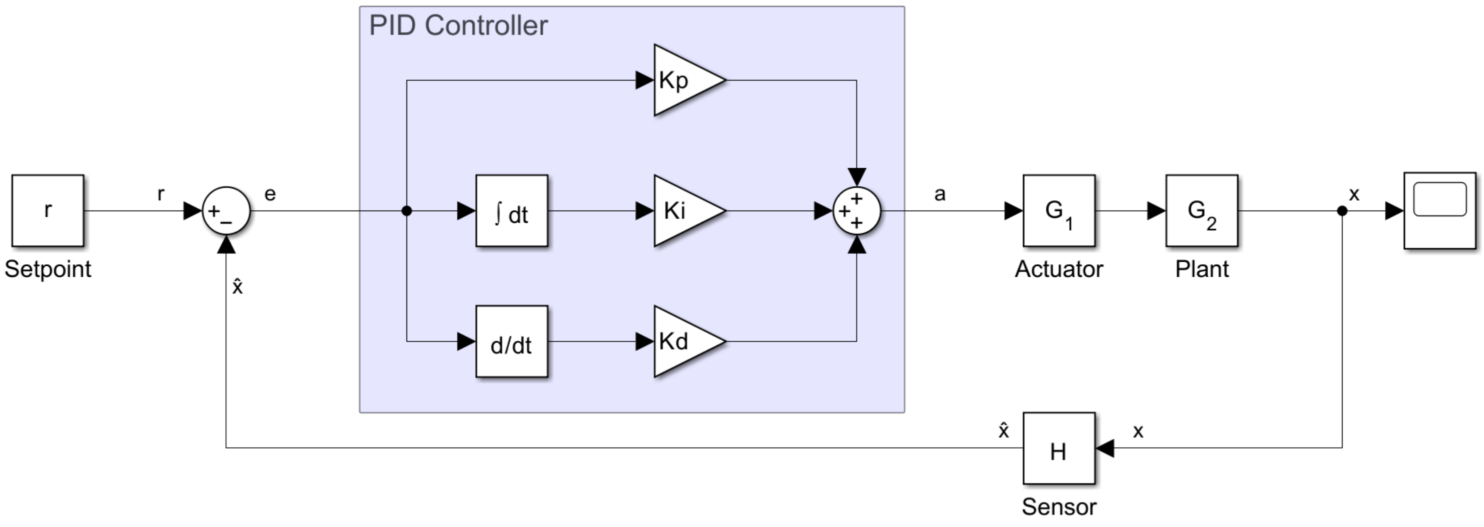
- The blocks labelled G_1 and G_2 represent the actuator and plant models, respectively; the actuator model and plant model are often lumped together and treated as a single plant $G = G_2\, G_1$, however it is important to understand that disturbances can enter the system in between the actuator dynamics and the plant dynamics due to unmodeled effects.

  The input to the actuator represents the "requested" effort from the controller, and is labelled as a on the diagram. The actuator is what converts the numerical value of the requested effort, as produced by the control law, into a physical actuation parameter such as torque which then applies to the plant. The output of the plant is labelled as x on the diagram and should be understood as the true physical output of the system, not a numerical value used in the control law.

  For the Romi robot, the actuator model would include the H-bridge motor driver and gearmotor dynamics and the plant model would include the dynamics of the Romi chassis; in this initial lab these aspects of the system dynamics will be lumped together.

- The block labeled H is a model of the sensor used to measure the physical output of the system plant, x, and represent that measurement as a numerical value $\hat{x}$; the sensor model may or may not include dynamics; in many cases, unity feedback is assumed or a simple gain is used in the feedback loop to represent the sensor, but some types of sensors may have internal dynamics that needs to be accounted for.

- The error signal, e is produced by the difference between the system setpoint, r, and the system output, as measured by the sensor, $\hat{x}$; that is, $e = r - \hat{x}$. This signal is what goes into the PID controller to determine the actuation value, a, requested from the actuator. For a complete PID controller, the feedback (control) law is $a = K_p\, e + K_i \int e\, \mathrm{dt} + K_d \frac{\mathrm{d}}{\mathrm{dt}} e$. The control law is often represented as a transfer function as well; for a full PID the transfer function representation becomes $C = K_p + \frac{K_i}{s} + K_d s$ so that $a = C\,e$.



Intuitive understanding of PID controllers comes from experience and practice working with them. However, the table below is an attempt at summarizing the contribution of each of the three components of the PID.

Summary of PID Components

Component	**Gain**	**Purpose**	**Potential Drawbacks**
Proportional	K_p	The proportional action is the primary driving component for the system to approach its target.	Large proportional gains may be needed to achieve suitable performance and such gains can lead to oscillation or instability. P-only controllers also suffer with respect to disturbance rejection and dynamic tracking.
Integral	K_i	The integral action is the component that keeps the system at or near the target value and helps to reduce the system error. Depending on the system type and the reference profile, integral control can lead to very low steady-state error.	Integral control can also lead to instability issues for larger gains. Other issues such as integral windup and reset windup are very common drawbacks, but each can be handled easily with a few tweaks to the algorithm. Integral action can also lead to issues for systems that require a large amount of effort to reach the target refence in comparison to the effort required to overcome disturbances, as

			the same gain will not handle both cases equally.
Derivative	K_d	The derivative action is the component that allows the controller to aggressively command the system as it approaches the target value while mitigating large overshoot. In other words, as the system approaches the target value the derivative action will help slow down the approach. Controllers with derivative action can have higher proportional and integral gains for the same amount of overshoot. Derivative control also makes the system respond faster to changes in the reference setpoint.	Derivative control has the effect of amplifying noise greatly, as most noise is of relatively high frequency but low amplitude. For some systems with low performing sensors, the derivative control is made useless by the amount of noise produced by the sensors. Additionally, derivatives are challenging to compute accurately using numerical methods, such as the finite difference method, which may lead to further numerical instability.

Control Loop Modifications

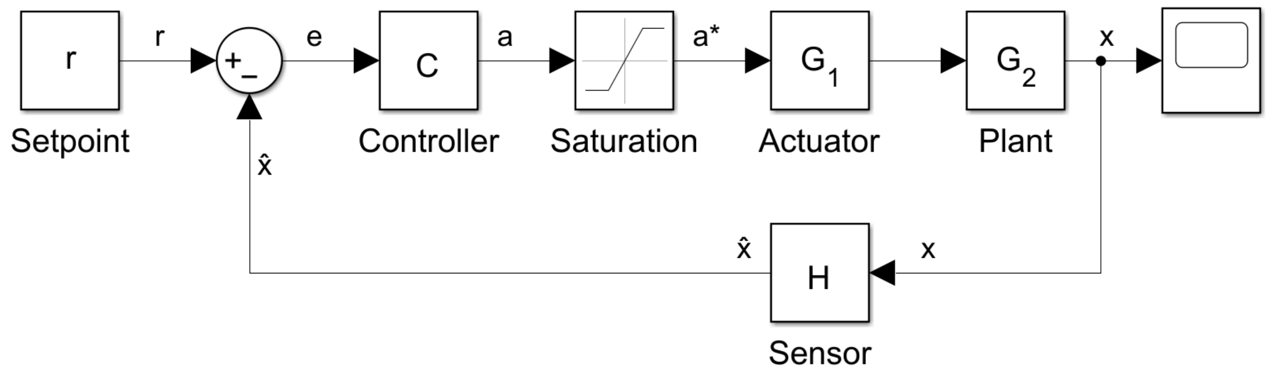
In many cases controls engineers will modify the control structure from the standard PID implementation in order to improve controller performance.

Actuator Saturation

One of the most important limitations of real world hardware is saturation. Actuators can only output so much power before they hit limits or fail. In some cases this has little effect on controller performance, but in general actuator saturation can have large ramifications.

Most importantly, the control law should never request more actuation effort than the actuator can safely supply. This is for two reasons: first, the control law should not request so much effort from the actuator to cause damage (due to overheating, torque limits, etc.) and second, the control law should be aware when actuator saturation occurs, even if no damage is expected from the actuator.

A modified control loop with actuator saturation is shown in the diagram below. The saturation block limits the true actuation value a^* to be between some fixed upper and lower limits even if the requested actuation value a exceeds those limits.



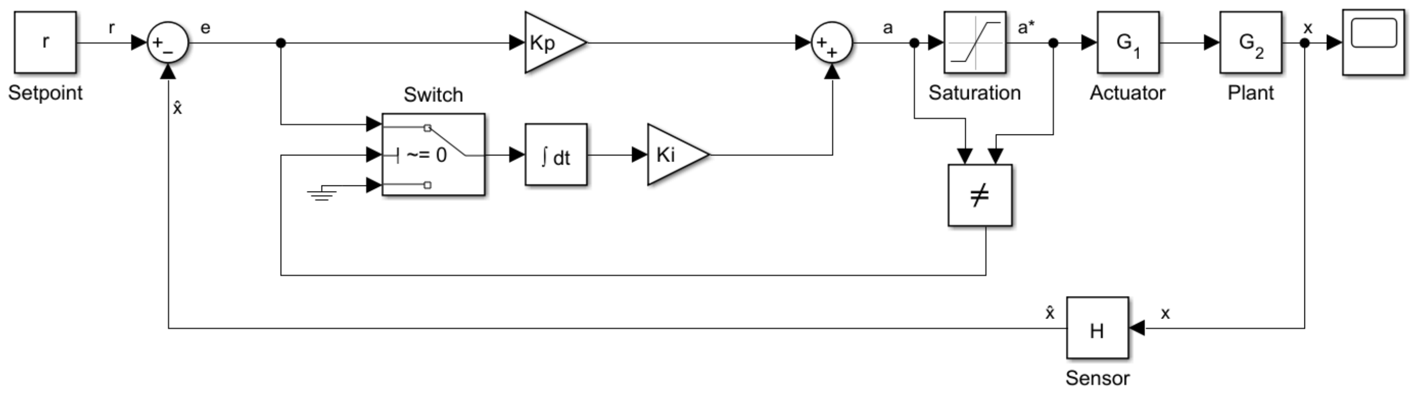
Anti-Windup

One of the unintended consequences of actuator saturation occurs in systems with integral control and is known as integrator windup, saturation windup, or reset windup.

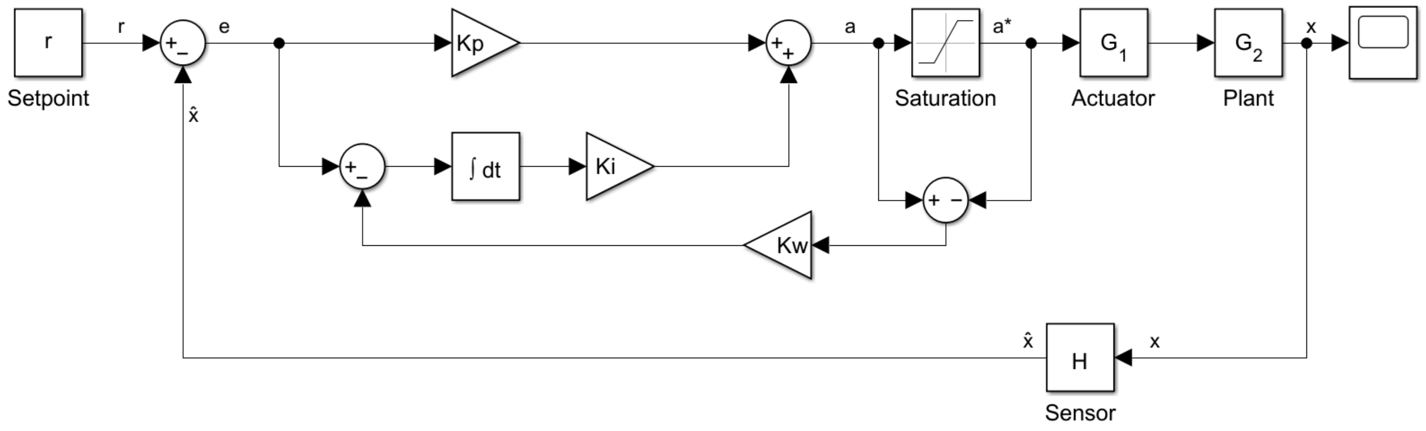
Saturation windup occurs when actuator saturation persists for a significant amount of time. During this period of time, the error continues to integrate even though the ever increasing integral value does not increase the actuator output. In other words, even though the controller is requesting more and more effort, the actuator clamps the effort at the saturation limit. The unbounded integration can cause bizarre controller output.

A common outcome of windup is prolonged overshoot. If the integrator has already grown large by the time the system reaches its setpoint then the actuation value will remain large even once overshoot occurs. Only after a sufficient amount of negative error during the overshoot period will the integrator value reduce enough to stop saturating the actuator output, eventually causing the system to reach a steady state.

The most common method of handling integrator windup is to "turn off" the integrator when the controller output is saturated. That is, as soon as the saturation takes place, the integrator should stop integrating the system error. This will keep the integrator value close to the threshold that just barely causes saturation.



Other anti-windup techniques use a feedback law to reduce the integrator value dynamically depending on the amount of saturation that is occurring.

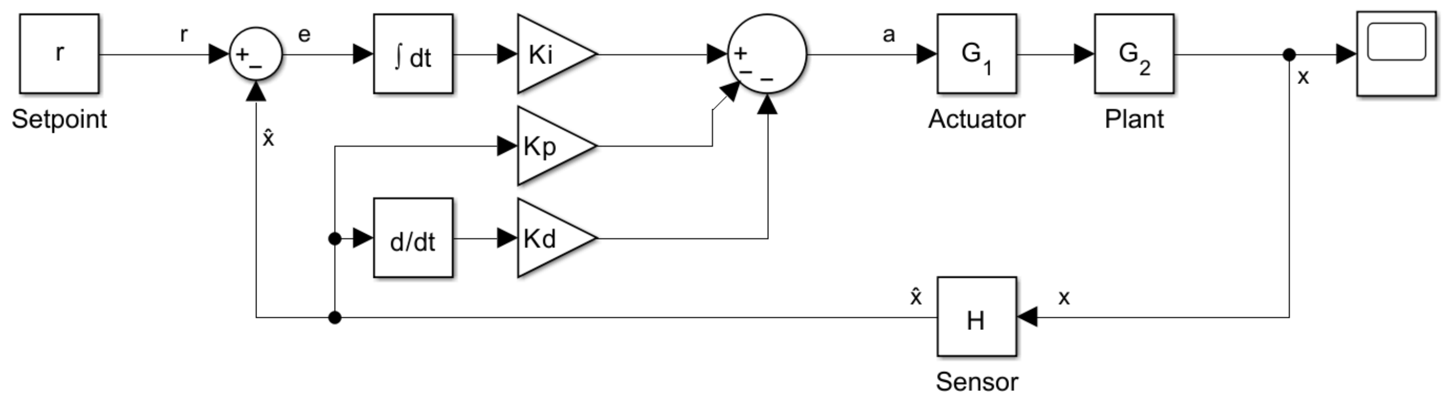


IP and IPD Controllers

Another common modification to the standard PID controller is meant to mitigate sensitivity to abrupt changes in setpoint. With a standard PID controller and a step input in setpoint, the system actuation will include a step change, from the proportional term, and an impulse, from the derivative term. These large spikes in actuation value can sometimes cause problems in implementation and in general they are unkind to the actuators in the system.

The difference in implementation is where the P and D terms are applied. In a standard PID controller, these terms apply to the error signal; with an IPD controller, only the integral term applies to the error signal with the proportional and derivative terms applying to the feedback (measurement) signal instead. That is, the modified control law is $a = -K_p\,\hat{x} + K_i \int e\,\mathrm{dt} - K_d \frac{\mathrm{d}}{\mathrm{dt}}\hat{x}$.

It should be noted that, partially through intentional design, the system will not respond as quickly if set up as an IPD controller instead of a PID controller.



Feedforward Control

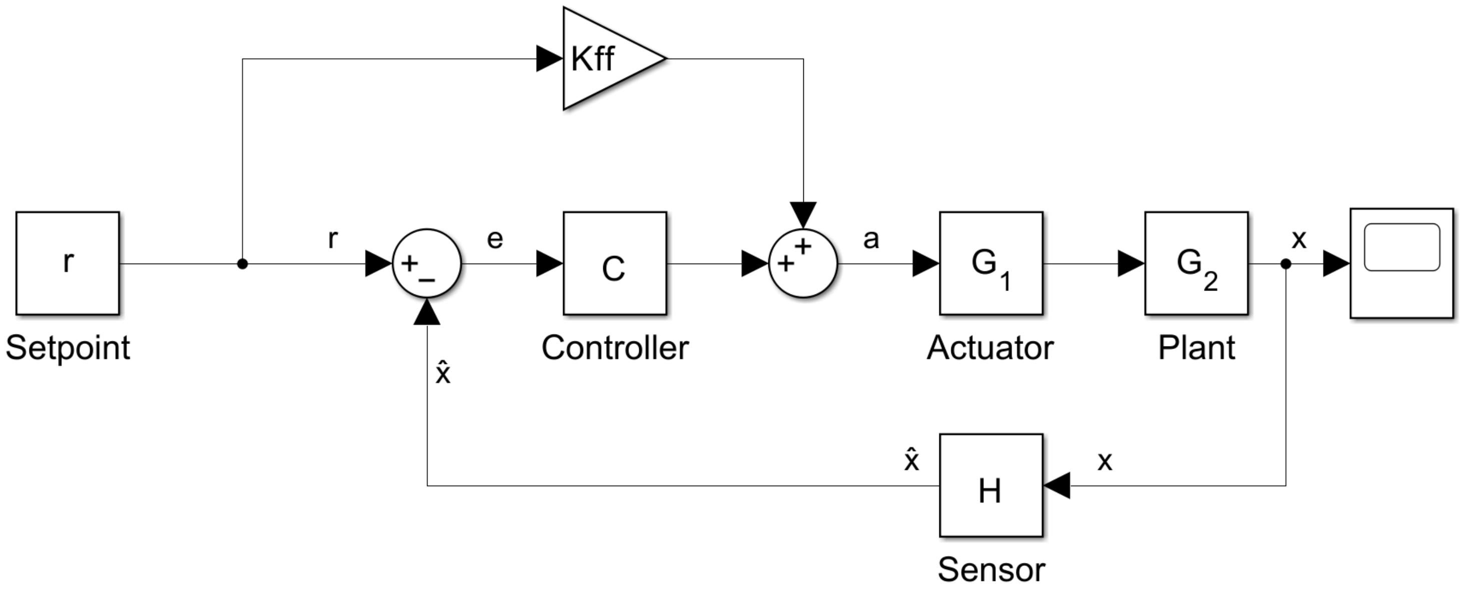
Another example modification that improves performance is the addition of a feedforward controller in parallel with the feedback controller. An example control loop is shown below as a block diagram.

The feedforward controller can be thought of as an open-loop controller with the objective of choosing an actuation value that, through the dynamics of the actuator and plant, cause the output to match the setpoint. In theory, an inverse model of the actuator and plant, $\frac{1}{G_2\,G_1}$, would act as a perfect feedforward controller as it would cancel out all dynamics between the setpoint and the output.

In practice, inverse plant models don't work due to mathematical and practical restrictions. Primarily, it is impossible to implement inverse plant models that result in improper systems, as indicated by transfer functions with higher-order numerators than denominators. Commonly, feed forward controllers use a simple proportional gain applied to the reference signal instead of a full inverse plant model, but it is also typical to combine a filter with the feedforward gain so that quickly changing setpoints don't affect the system right away.

The feedback controller therefore only needs to work on the small error between the open-loop output and the setpoint. In this way a higher performing controller can be implemented because the feedback controller only needs to respond to the smaller fluctuations in the error and is not responsible for maintaining steady-state output.

For example, it should be reasonably straightforward to use the data collected in previous assignments to estimate the necessary voltage or duty-cycle required to maintain a certain velocity for the vehicle. This value can then be used as the feedforward gain; a model of the system along with accurate model parameters would also produce a suitable feedforward gain. A simple gain does ignore the dynamics associated with the motor and chassis, but nonetheless helps improve controller performance, especially in cases where the setpoint remains constant.



Battery Droop Compensation

As a battery discharges its voltage droops significantly. For example, a fully charged NiMH cell has a voltage of 1.4V, but after about 10% or 15% discharge, the voltage drops to the nominal battery voltage of about 1.2V. The nominal voltage is relatively stable until the battery is about 90% discharged when the voltage drops to about 1V. Batteries from reputable sources should have published discharge curves. For example, the Panasonic Eneloop batteries have plenty of published test data available on the internet. For those curious, consider looking at some of this data: **https://eneloop101.com/batteries/eneloop-test-results/** (https://eneloop101.com/batteries/eneloop-test-results/) .

If this battery droop is not compensated for, it may be challenging to tune the control loop as the sensitivity to each gain will effectively be reduced by the drooping battery voltage. Consider a model of the H-bridge and PWM that would be part of the "Actuator" block in the block diagrams above:

$$V_m = D\,V_{NOM}.$$

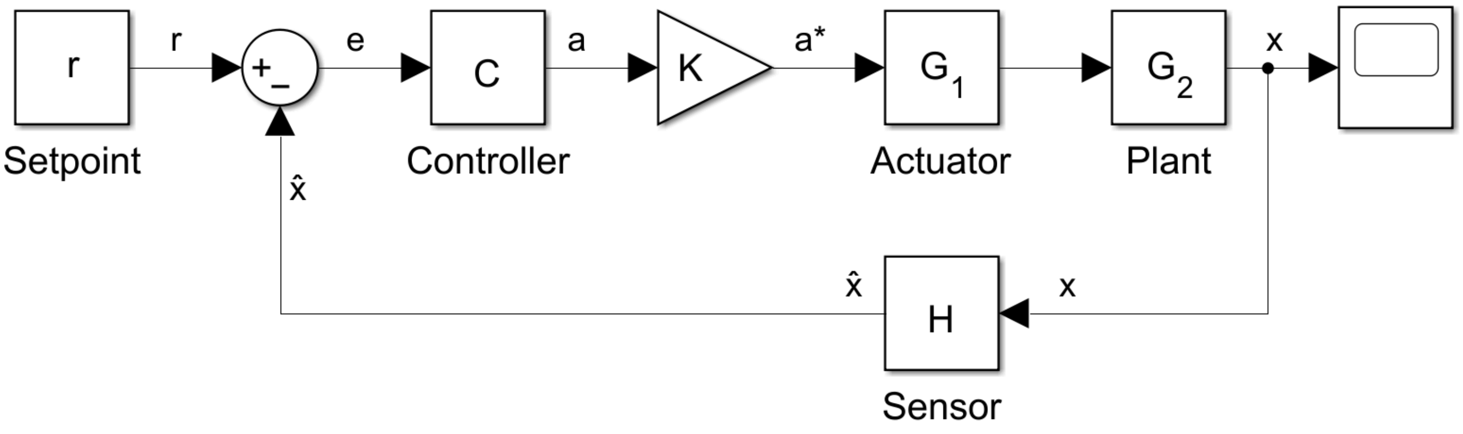
This model shows that the voltage going to the motor is equal to the nominal DC supply voltage, V_{NOM}, multiplied by the PWM duty cycle, D, represented as a signed number between -1.0 and 1.0. This works great for constant voltage supplies because the duty cycle is simply scaled by the constant voltage value (a gain) which can be handled easily as part of your tuning procedure. However, once the battery voltage, V_{BAT} starts drooping, the same duty cycle will not always represent the same output voltage. In this case, the equation becomes

$$V_m = D\,V_{BAT}.$$

So, with a drooping battery voltage, the effect cannot be handled by a static gain, but instead requires a dynamic gain. To fix this, you can insert an additional gain that applies a correction to the requested duty cycle in between the controller output and the actuator that scales the requested actuation value by a measurement of the true battery voltage. The gain can be handled in a variety of ways, depending on the units your group selects for each signal. In any case, the gain needs to be inversely proportional to the battery voltage so that the dynamic gains cancel out. For example, suppose an extra gain $K = \frac{V_{NOM}}{V_{BAT}}$ is added, making the new equation

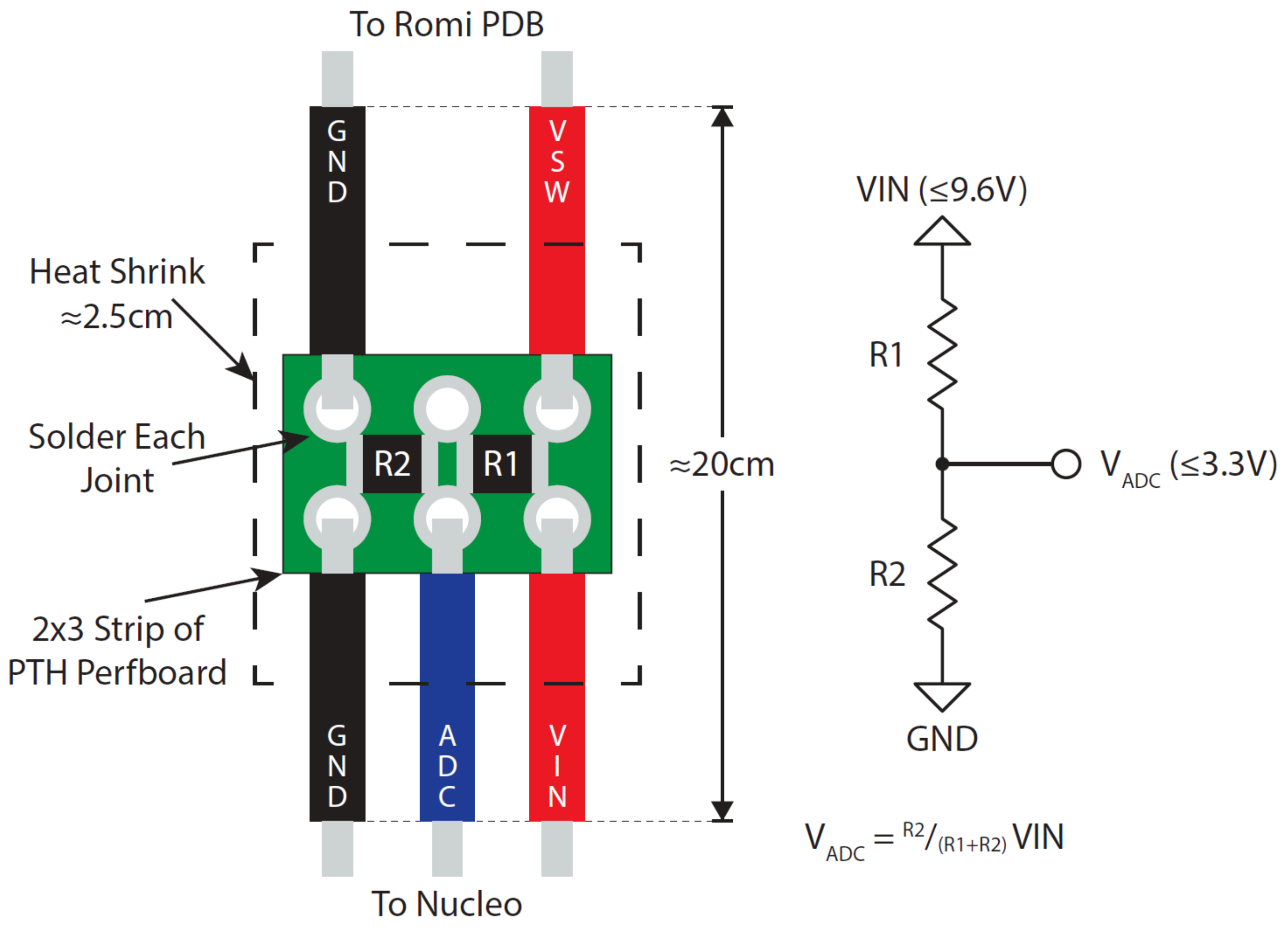
$$V_m = D\,\frac{V_{NOM}}{V_{BAT}}\,V_{BAT}.$$

With this new gain included, the dynamic values of V_{BAT} cancel with each other, effectively removing the effects of the battery droop. The block diagram below generalizes this compensation concept. In the diagram each signal or gain is written using more general symbolic values; that is, in the diagram, $a = D$, $K = \frac{V_{NOM}}{V_{BAT}}$, $a^* = D\,K$, and the V_{BAT} gain is hidden within the actuator block G_1.



Measuring V_{BAT}

To measure the battery voltage safely, a voltage divider is necessary. The maximum raw battery voltage (as high as 9.6V for AA Alkaline batteries) is much too high to read with the STM32's ADC which has an active range of 0V to 3.3V. To reduce the battery voltage into a safe range a divider with a ratio of approximately $\frac{1}{3}$ is required. A compact divider circuit can be built by integrating the voltage divider into the power cable that connects between the Romi PDB and the Nucleo board. A small 2x3 piece of perfboard along with two SMD resistors will work well for a small footprint. See the image below for specific details. To achieve such a divider ratio two resistors are needed, one that is equal to or slightly higher than double the value of the other resistor. Additionally, the sum of the two resistor values should be large enough to mitigate significant current flow or risk draining the battery prematurely. Values of 10k and 4.7k will work nicely; the divider ratio of $\frac{4.7\,k\Omega}{14.7\,k\Omega} \approx 0.32$ will work nicely, and the steady-state current draw will only be $\frac{9.6\,V}{14.7\,k\Omega} \approx 650\mu A$ which is small compared to the draw from the rest of the system.



Assignment

Modify your Lab 0x02 to allow closed-loop control of Romi's gearmotors and automated data collection.

1. Start by adding to or reworking your task diagram and state transition diagrams from Lab 0x02. If needed, add new tasks to the design, but at the least modify the set of shares and queues to facilitate the new lab objectives on the task diagram and update each finite state machine as needed to perform closed loop control.

   For example, the user interface task should be augmented to allow the user to select an input for a step response test. You will want to be able to test Romi's speed control driving in a straight line, driving in an arc, and also pivoting in place, so a manner of selecting velocity setpoints for each wheel will be necessary.

   You will also need to design a new class to implement the closed loop controller; although, if any teams decide it is most appropriate, the closed loop controller code can also be implemented as part of a given task instead of as a stand alone driver. Consider with your team what will be more useful in future labs.

2. Make adjustments to the firmware as needed to implement the changes to your software design.

3. Modify your Python script that runs on your laptop to interface with the MCU firmware and automate data collection from step responses. Ideally this script would allow you to perform a set of tests with different setpoints and gains, each time automatically retrieving data from Romi and generating plots. The script should then store the collected data and step response plots in time-stamped and clearly named files so that you can review your testing results easily and without confusion.

4. Once your automated testing system is operational, use the system to tune Romi's performance as best as reasonably possible. Note that the goal of this lab is specifically not to get the best possible performance from Romi. Instead, the goal is to establish means of effective tuning that will scale well into the remainder of the quarter.

Deliverables

Submit a PDF memo along with Python attachments to Canvas. In your memo present a "story" showing your tuning results. That is, show a variety of plots trending from poorly tuned to reasonably well tuned for a variety of setpoints. Make sure to annotate the memo clearly with the motivation for each test, parameters for each test, and the results as shown in step response plots. If your team has decided on a specific performance metric, please indicate so in the memo along with what values were achieved for each metric.

As usual, submit your Python source code along with your memo.