MXET 300
Mechatronics I

Multidisciplinary
Engineering Technology
COLLEGE OF ENGINEERING

LABORATORY # 7

Closed Loop Control

Submission Date: 03/31/2025

Name: Kyle Rex and Rex Worley
Section: 501
UIN: 932008894 and 432006442

Introduction

This lab focused on assembling and configuring the SCUTTLE robot to implement closed-loop control using proportional-integral-derivative (PID) control strategies. The primary objectives included assembling the robot by securely mounting components and making the necessary electrical connections, ensuring proper motor functionality, and verifying encoder accuracy. After validating hardware operation, proportional control was first implemented to analyze the system's response to varying gain values. Subsequently, integral and derivative control elements were introduced to observe their effects on system stability and steady-state error. Through these exercises, key concepts of closed-loop control, including proportional, integral, and derivative tuning, were explored to enhance understanding of system dynamics and controller optimization.

Procedures and Lab Results

The lab begins with the assembly of the SCUTTLE robot, ensuring that all components, including the Raspberry Pi, battery pack, and motor driver bracket, are securely placed on the DIN rail, before wiring, following the same procedure for mounting these parts that is described in the Lab 3 manual. Placing the parts first, before wiring, prevents potential damage to connectors. Once all the parts are placed on the DIN rail the connections between the Raspberry Pi and Battery Pack can be completed following the same procedure for connecting these two parts described in the Lab 2 manual. Additional connections to the motors and L298N H-bridge are required on the SCUTTLE and can be completed following the procedure that is described in the Lab 4 manual. Two more connections are required between the absolute rotary encoders and the Raspberry Pi and can be done following the procedure described in the Lab 5 manual. These connection schemes can be seen in Figure 1, Figure 2, and Figure 3. The final setup for the SCUTTLE can be seen in Figure 4.

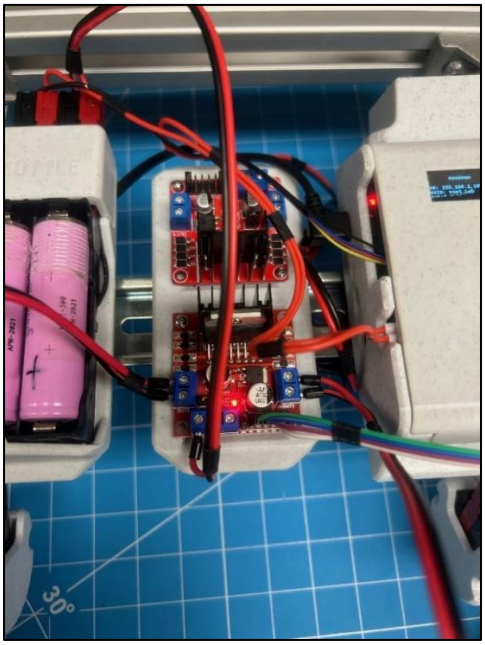


Figure 1: H-Bridge Connections

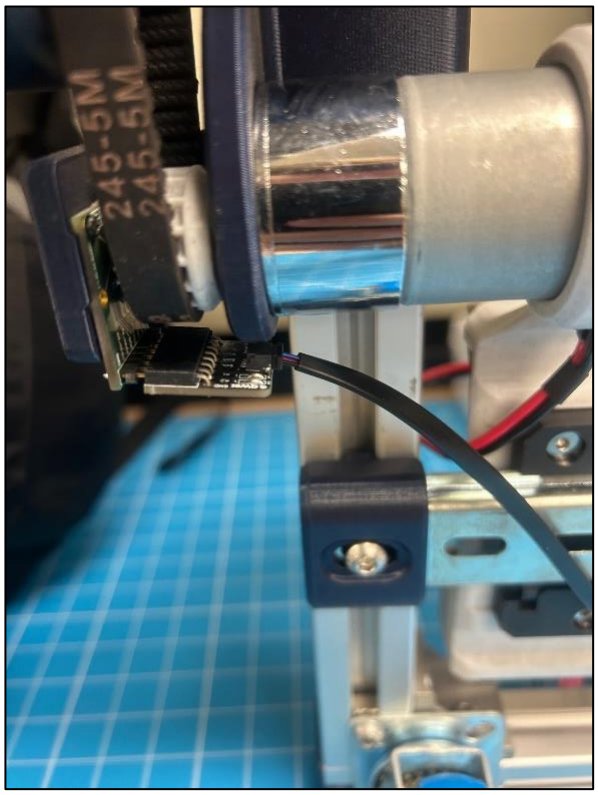


Figure 2: Encoder Connection (Same on Both Sides)

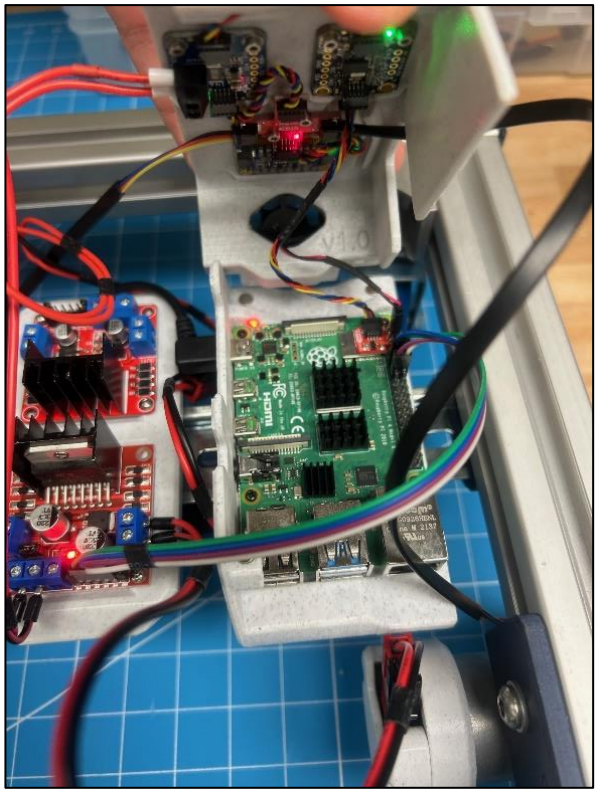


Figure 3: Raspberry Pi Enclosure Connections

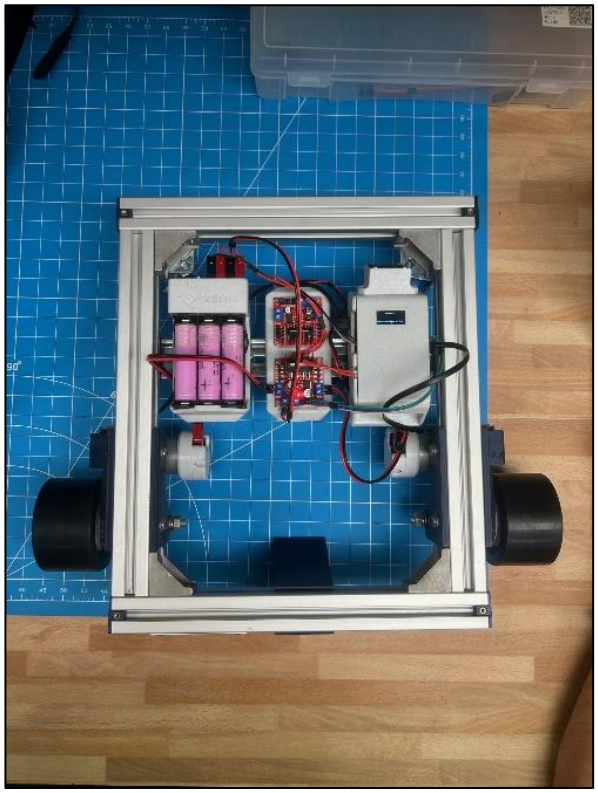


Figure 4: Complete SCUTTLE Physical Setup with All Connections

After checking the wiring and setup are correct the next step is to complete the booting and connecting procedure that are described in the Lab 1 manual. This will allow the Raspberry Pi to be accessed through Visual Code Studio's terminal over a wireless network. When the Raspberry pi powers on, connect to the network and open the VS code. In VS code, using previously downloaded code sequences, confirm that the SCUTTLE's motors and encoders are functioning correctly. This can be done by running L1_motor.py while ensuring each of the wheels move in the correct direction and running L1_encoder.py while ensuring the sign and magnitude of measurements are correct. The correct terminal outputs for each can be seen in Figure 5 and Figure 6 respectively.

```
SCUTTLE-07 basics git:(main) ➜ python3 L1_motor.py
motors.py: driving fwd
motors.py: driving reverse
stopping motors 4 seconds
^CTraceback (most recent call last):
  File "L1_motor.py", line 55, in <module>
    time.sleep(4)
KeyboardInterrupt
```

Figure 5: L1_motor.py Working Terminal Output

```
SCUTTLE-07 basics git:(main) ➜ python3 L1_encoder.py
Testing Encoders
Left:  92.5      Right:  47.1
Left:  92.6      Right:  47.1
Left:  92.6      Right:  47.1
Left:  92.6      Right:  47.0
Left:  92.5      Right:  47.1
Left:  92.5      Right:  47.1
Left:  92.6      Right:  47.1
```

Figure 6: L1_encoder.py Working Terminal Output

With a working SCUTTLE the next step is to implement Proportional Control (k_p) as an exercise to better understand how to use closed loop feedback control with the SCUTTLE. This is the first type of controller that will be implemented in this lab because it is considered the simplest. The proportional control value is directly proportionate to the error between the target values and the measured values. The wheel speeds of the SCUTTLE (represented as phi dot) are measured to provide feedback for reaching targeted wheel speeds. To begin this exercise a file must be imported into the basics directory. To do this use the "cd" command to go into the basics directory. Once in this directory the "wget" command can be used, following the procedure in the Lab 2 manual, in the MXET300-SCUTTLE GitHub directory, to download and import Lab7Template.py, which uses closed loop control logic to drive the SCUTTLE forward at its maximum wheel speed of 9.7 rad/s while logging the speed of the SCUTTLE's left wheel and exporting it as a temporary file. The Lab7Template.py file actually calls the closed loop control from another file called L2_speed_control.py that has already been imported from a previous lab. The specific function it calls can be seen in Figure 7.

```python
def driveClosedLoop(pdt, pdc, de_dt):              # this function runs motors for closed loop PID control
    global u_integral
    e = np.array(pdt - pdc)                                 # compute error

    kp = pidGains[0]                               # gains are input as constants, above
    ki = pidGains[1]
    kd = pidGains[2]

    # GENERATE COMPONENTS OF THE CONTROL EFFORT, u
    u_proportional = (e * kp)                                       # proportional term
    try:
        u_integral += (e * ki)                                      # integral term
    except:
        u_integral = (e * ki)                                       # for first iteration, u_integral does not exist

    u_derivative = (de_dt * kd)                                     # derivative term takes de_dt as an argument

    # CONDITION THE SIGNAL BEFORE SENDING TO MOTORS
    u = np.round((u_proportional + u_integral + u_derivative), 2)   # must round to ensure driver handling
    #u = scaleMotorEffort(u)                                         # perform scaling - described above
    u[0] = sorted([-1, u[0], 1])[1]                                 # place bounds on the motor commands
    u[1] = sorted([-1, u[1], 1])[1]                                 # within [-1, 1]

    # SEND SIGNAL TO MOTORS
    m.sendLeft(round(u[0], 2))                                        # must round to ensure driver handling!
    m.sendRight(round(u[1], 2))                                       # must round to ensure driver handling!
    return
```

Figure 7: Closed-Loop Control Code in L2_speed_control.py

This exercise requires that the proportional constant value (k_p) be changed to 0.04 while the other constant values (k_i & k_d) be changed to 0 in the L2_speed_control.py file. The values that require these changes are the global variables in the first few lines of the file that can be seen below in Figure 8.

```
kp_left = 0.04
ki_left = 0.00
kd_left = 0.00

kp_right = 0.04
ki_right = 0.00
kd_right = 0.00
```

Figure 8: PID Control Values

With the values set, and the SCUTTLE on its stand, the Lab7Template.py file can be run for 10 seconds before being terminated with Ctril+C. After it has been terminated the temporary file created, with at least 60 data samples, can be found in the basics directory, exported, and converted to an excel file. An example of what one of these files will look like can be seen in Figure 9.

	A	B	C
1	1	0	9.7
2	2	2.368661	9.7
3	3	1.537552	9.7
4	4	1.786885	9.7
5	5	1.705663	9.7
6	6	1.824662	9.7
7	7	1.897097	9.7
8	8	1.745329	9.7
9	9	1.745329	9.7
10	10	1.705663	9.7
11	11	1.707387	9.7
12	12	1.707387	9.7
13	13	1.745329	9.7

Figure 9: Start of Temporary File for Left Wheel Speed Example

With this data a plot can be created to better interpret what the resulting values mean in comparison to the target values. This plot, properly formatted, can be seen in Figure 10.

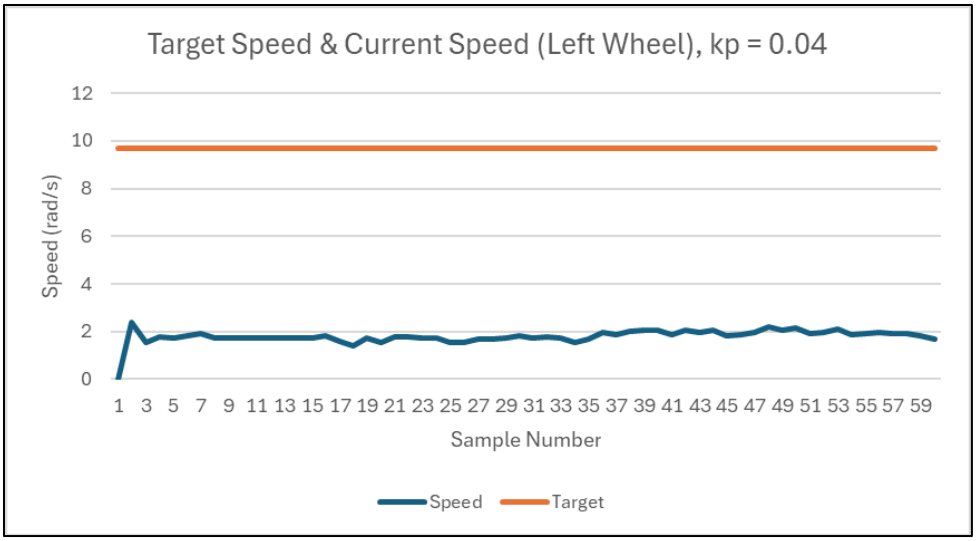


Figure 10: Target Speed & Current Speed (Left Wheel), kp = 0.04

The same steps that resulted in this plot for kp = 0.04 can be repeated for kp = 0.09 and kp = 0.5. As the value is increased the wheel should turn faster because they are getting closer to the actual target value. This is because the proportional control value and correlation between the target and measured values are proportionate. This correlation and the plots for each value can be seen in Figure 11 and Figure 12.

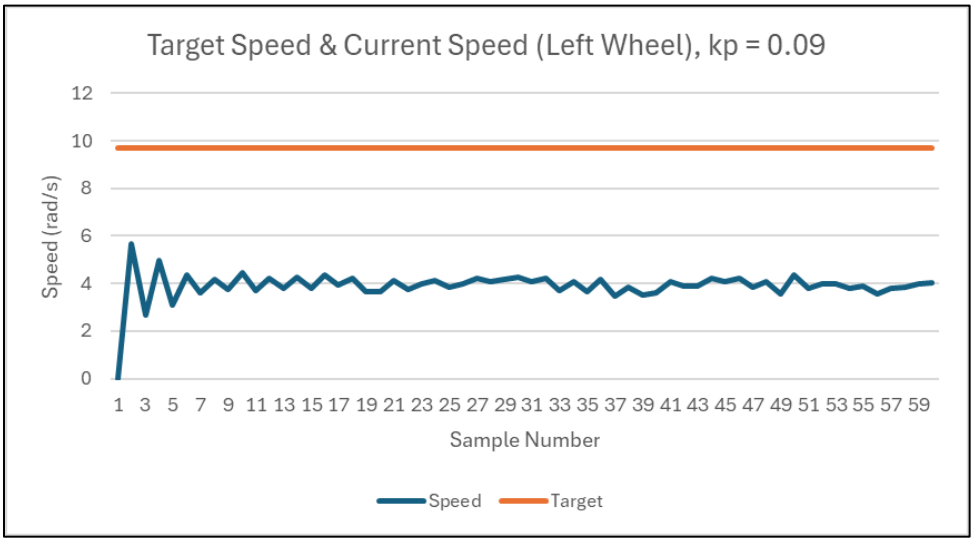


Figure 11: Target Speed & Current Speed (Left Wheel), kp = 0.09

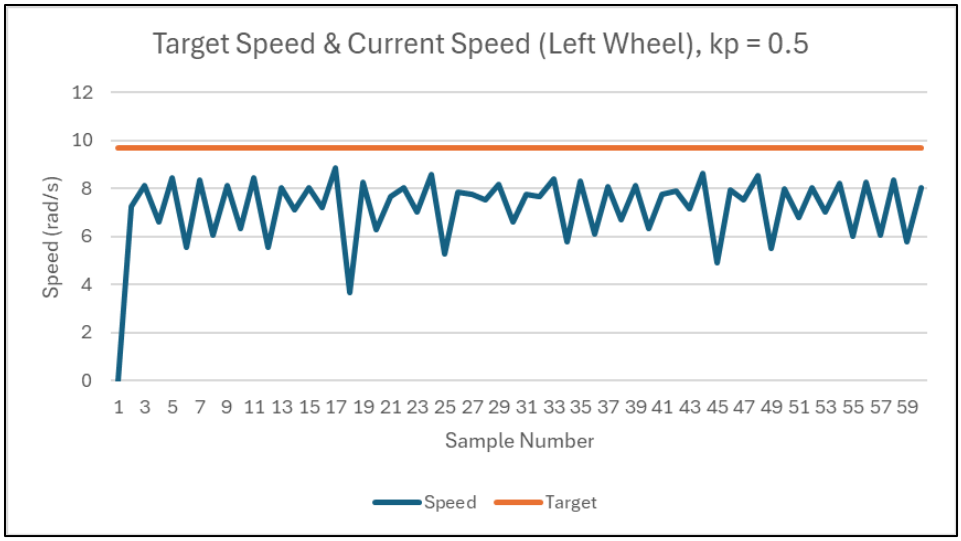


Figure 12: Target Speed & Current Speed (Left Wheel), kp = 0.5

The next exercise in this lab involves integrating both proportional and integral control (PI control) instead of just proportional control as before. The PI controller uses an accumulation of error over time to correct steady-state error. To better understand how integral control works, it must be observed on its own first. Integral control only corrects errors over time without proportional effort making it ideal for systems that need speed control, but don't care about how long it takes to reach that target. Using the same procedure as before, modify the L2_speed_control.py script to set the integral constant (k_i) to 0.01 and the other constant values (k_p & k_d) to 0. Running this script in the same manner as previously described will result in a plot that can be seen in Figure 13. Doing the same after changing the integral constant (k_i) values to 0.02 and 0.03 will result in the plots seen in Figure 14 and Figure 15 respectively. Lastly, setting both the proportional control and integral control values (k_p & k_i) to 0.04 will result in the plot seen in Figure 16.

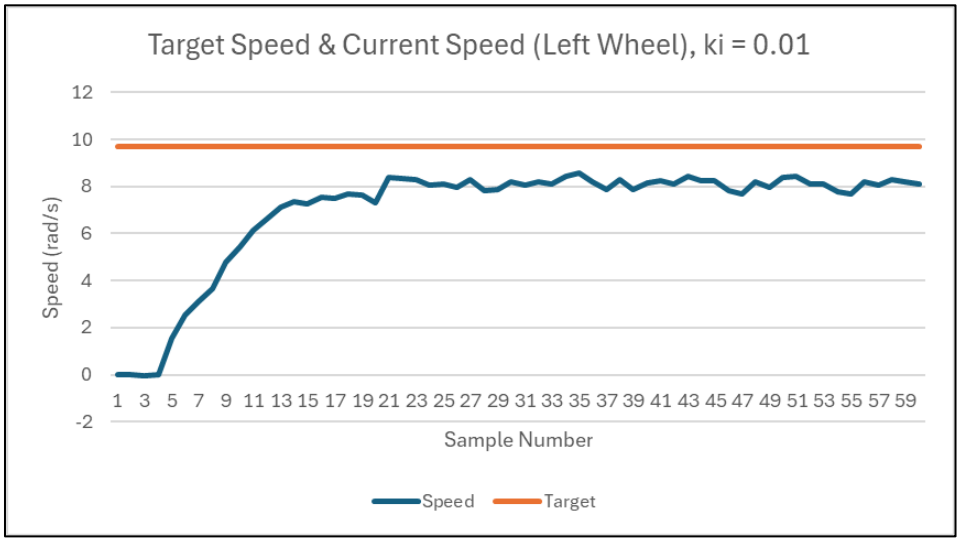


Figure 13: Target Speed & Current Speed (Left Wheel), ki = 0.01

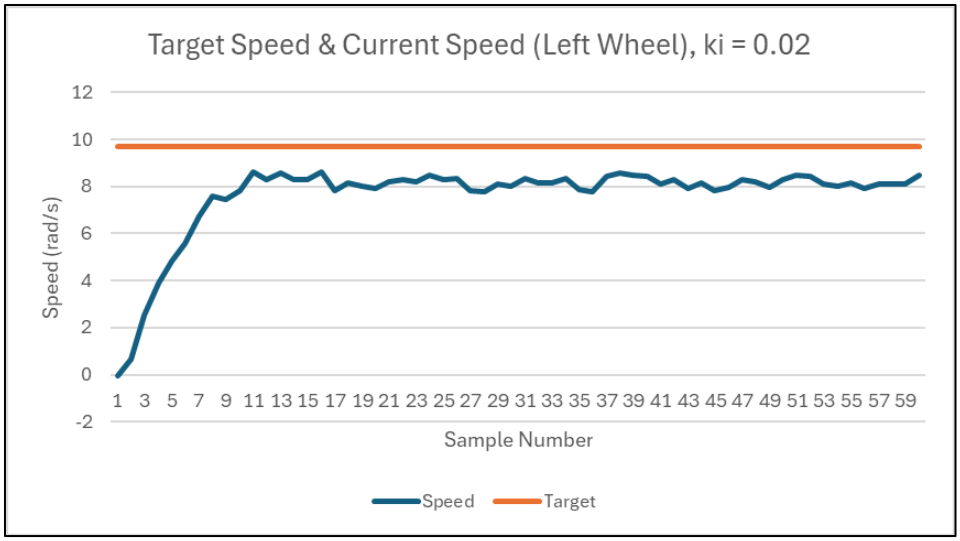


Figure 14: Target Speed & Current Speed (Left Wheel), ki = 0.02

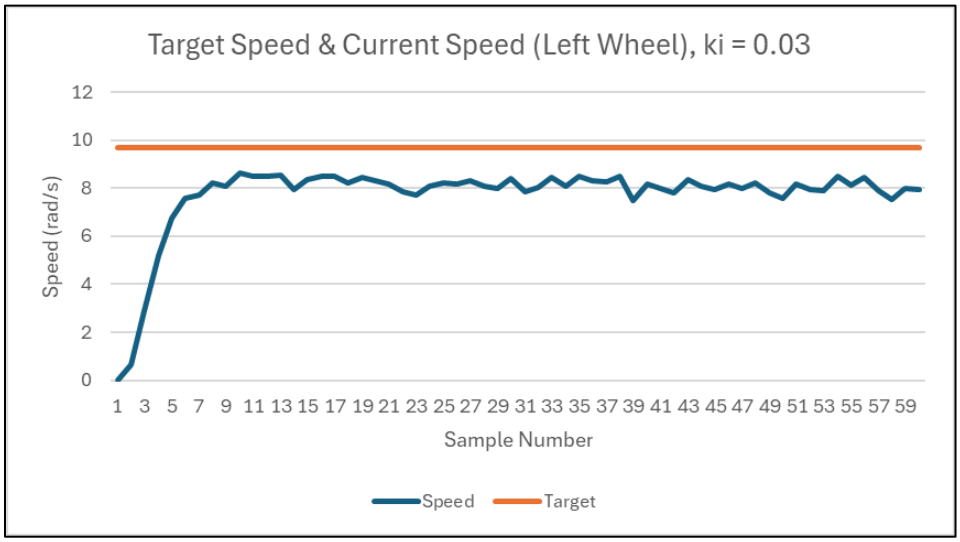


Figure 15: Target Speed & Current Speed (Left Wheel), ki = 0.03

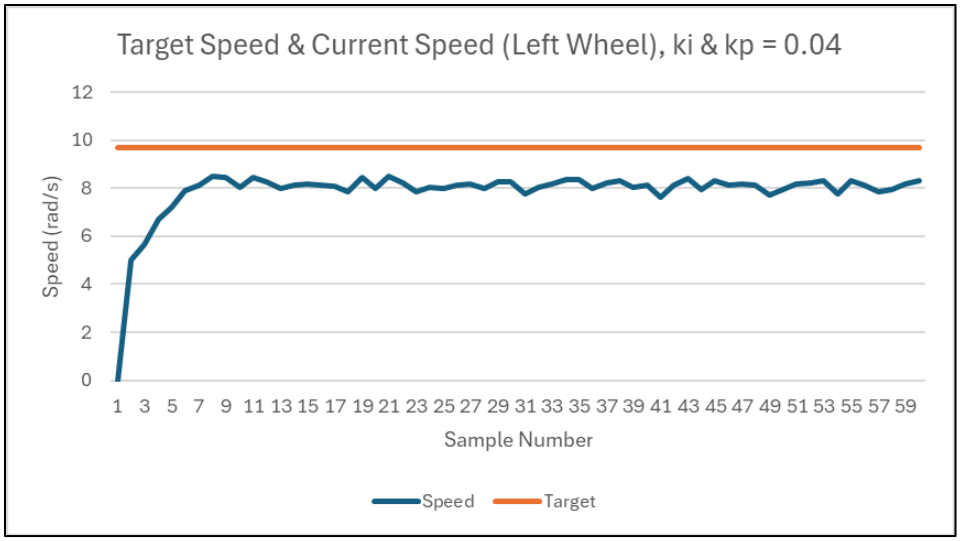


Figure 16: Target Speed & Current Speed (Left Wheel), ki & kp = 0.04

With all of these plots it can be inferred that as the integral constant is increased the time it takes for the wheel to get to its constant speed is significantly reduced. When combining the integral and proportional control it provides the best control. Not only does it reach its constant value the fastest, but it also reaches a value that is closest to the target value when compared to other plots. Now it's time to include the derivative control into this, therefore completing the PID controller, which is the last exercise. The derivative constant is meant to assist by anticipating future error by applying effort opposing sudden changes in error through reducing the influence of noisy measurements and impulses. Because the derivative constant can make a system unstable in certain conditions it is required to be determined experimentally using the maximum expected change in error.

To analyze the control effort, the process begins by identifying the two adjacent samples with the greatest change in error. The first ten samples are excluded to avoid the influence of initial large errors as the system stabilizes. The error for each sample is determined as the difference between the input reference and the measured output. The largest change in error between two consecutive samples is then identified by computing the difference between the current and previous error values. Since the measurements are taken with respect to discrete samples rather than time, the error gradient is approximated as the change in error per sample. The derivative control effort is then calculated by multiplying this gradient by a derivative gain of 0.04. Additionally, the proportional control effort is obtained by multiplying the current error by a proportional gain of 0.04. These two values represent the maximum derivative and proportional control efforts applied at that instance. These values were calculated as ud = 0.022765 and up = 0.0875.

Understanding how to solve for those values, the next step is to, using the same procedure as before, modify the L2_speed_control.py script to set the proportional, integral, and derivative constants (k_p & k_i & k_d) to 0.04, completing the PID loop, and then running the script to ultimately get the plot that can be seen in Figure 17.

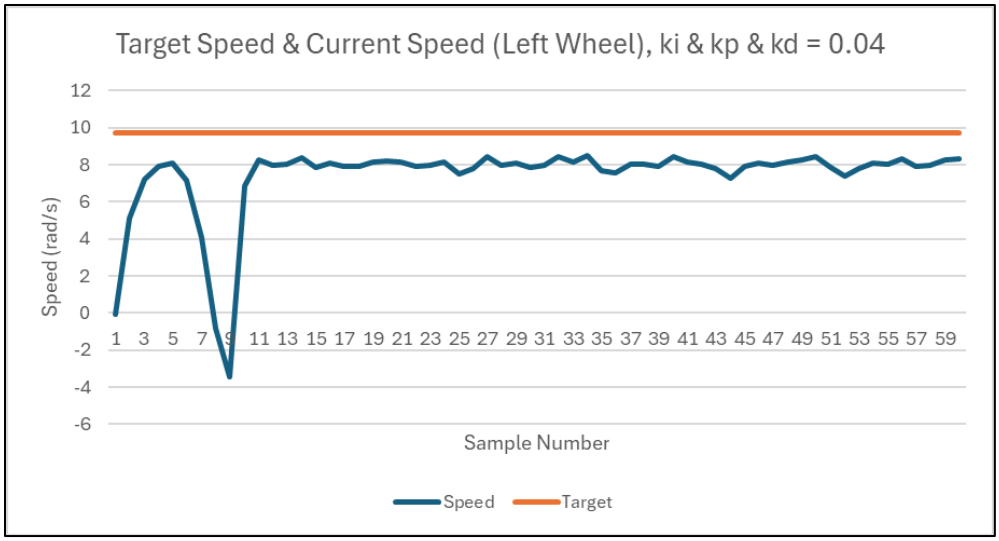


Figure 17: Target Speed & Current Speed (Left Wheel), ki & kp & kd = 0.04

The plots offer a distinct visual representation of how adjusting the PID constants influences the performance of the closed-loop control system. In the P control charts (featuring different kp values), it is observable that a low kp like 0.04 yields a slow response accompanied by a prominent steady-state error. As kp increases to 0.09, the system responds more swiftly to deviations, shortening the time to reach the setpoint; however, this enhancement results in overshoot and heightened oscillations. When kp is adjusted to a very high value, like 0.5, the proportional response turns excessively aggressive, resulting in considerable overshoot and erratic oscillatory behavior. This pattern corresponds with the theory outlined in the manual, which cautions that while higher proportional gain may decrease rise time, it could also create instability if not adequately fine-tuned.

Examining the charts from the I controller experiments, it becomes clear that raising ki values (from 0.01 to 0.03) gradually diminishes the steady-state error. Nevertheless, since the I action exclusively integrates past error, the response is characteristically slower and more susceptible to overshoot when ki is excessively high. This compromise illustrates why a purely I controller is typically less advantageous in scenarios requiring prompt transient performance. When merging the P and I components into a PI controller (with kp and ki established at 0.04), the resulting graph demonstrates a balanced behavior: the proportional component offers a quick reaction to error, while the integral component aims to eradicate any lingering offset. The combination of these effects results in a more rapid and precise convergence to the setpoint compared to utilizing either action in isolation.

Ultimately, the PID controller graph, which integrates a derivative term (kd = 0.04) along with the previously calibrated P and I values, showcases the advantages of incorporating derivative action. The derivative term anticipates future error by considering the rate of change, which significantly lessens overshoot and mitigates oscillations. Consequently, the PID response appears smoother and more critically damped in comparison to the PI controller. The capability of the derivative action to temper rapid fluctuations reinforces the necessity of meticulous tuning to avert aggressive control measures that might ultimately lead to instability or adversely affect the system or user.

Conclusion

The lab successfully demonstrated the impact of different PID control components on the SCUTTLE robot's motion and stability. The proportional control experiments revealed that increasing the proportional gain reduced steady-state error but also introduced oscillations when set too high. The integral control helped eliminate steady-state error but required careful tuning to avoid excessive overshoot, while the derivative control mitigated oscillations and improved system stability. Implementing the complete PID controller provided the most balanced performance, achieving precise and stable wheel speed regulation. Challenges encountered during the lab included ensuring accurate wiring connections, debugging script execution errors, and fine-tuning control parameters for optimal performance. Overcoming these issues reinforced the importance of systematic troubleshooting and iterative tuning in control system design.

Post-Lab Questions

1. Explain the difference between a P and PI controller.

A P controller (P for proportional) solely utilizes the control effect directly proportional to the current error. The difference between the predetermined point and measured output at that point in time. The PI controller (Proportional Integral) combines the proportional and integral terms to aid in eliminating steady-state errors. Compared to the P controller, the response of the PI controller is slower.

2. What does an increase in the proportional gain (kp) do to the control system response?

Increasing the proportional gain provides a reduced time to reach the desired output, increased risk of very high values causing persisting and erratic oscillations, faster response from a given error, and may overshoot within the oscillations.

3. There are three different and well-known step response waveform types for control systems: over-damped, under-damped, and critically-damped. Briefly explain each.

Over-damped responses do not over shoot and are relatively slow as the system gradually settles at equilibrium without oscillations. Under-damped responses are typically faster and overshoot the goal before reaching a stable point. These systems display oscillations around the setpoint and settles at equilibrium. While being more responsive, the stability and precision can become inaccurate if overshoot is excessive. Critically-damped responses are ideal for most applications as they return to equilibrium as quickly as possible without overshooting the predetermined point. This has a balance of stability and performance.

Appendix

Lab 7 Commands Used:

- python3 L1_motors.py
- python3 L1_encoder.py
- wget Lab7Template.py
- python3 L2_speed_control.py
- sudo mv /tmp/excel_data.csv

Lab 6 Commands Used:

- wget L1_lidar.py
- sudo python3 L1_lidar.py
- wget L2_vector.py
- sudo python3 L2_vector.py
- touch L3_logs
- log.tmpFile(voltage, "voltage_log.txt")
- log.tmpFile(B[0], "L2_dist.txt")
- log.tmpFile(B[1], "L2_angle.txt")
- log.tmpFile(C[0], "L2_x.txt")
- log.tmpFile(C[1], "L2_y.txt")
- git add .
- git commit -m “lab6”
- git push

Lab 5 Commands Used:

- sudo watch -n0.2 i2cdetect -y -r 1
- wget
- python3 L1_encoder.py
- wget
- python3 L2_kinematics.py
- python3 L1_motor.py
- ctrl + c
- log.tmpFile(voltage, "voltage_log.txt")
- log.tmpFile(C[0], "xdot.txt")
- log.tmpFile(C[1], "thetadot.txt")
- log.tmpFile(B[0], "pdl.txt")
- log.tmpFile(B[1], "pdr.txt")
- python3 L3_log_speeds.py
- ctrl + c
- python3 L3_log_speeds.py
- python3 L3_path_template.py

Lab 4 Commands Used:

- pwd
- python3 L3_path_template.py
- python3 L3_noderedControl.py
- git add .
- git commit -m “Lab 4”
- git push

Lab 3 Commands Used:

- cd basics/
- wget
- python3 L2_compass_heading.py

- python3 L3_Compass.py
- tmpFile
- stringTmpFile
- git add .
- git commit -m “Lab 3”
- git push

Lab 2 Commands Used:

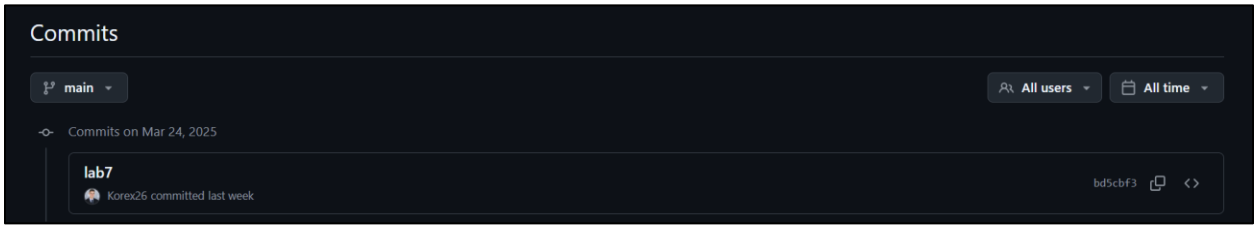
- cd
- mkdir basics
- pwd
- wget
- python3 L1_ina.py
- sudo systemctl stop oled.service
- sudo mv oled.py/usr/share/pyshared/oled.py
- git status
- git add .
- git commit -m
- git push

GitHub Submission

All required files were pushed to the team’s repository for review and backup as seen in the GitHub Submission Figure 1 and GitHub Submission Figure 2. The repository can be found using the following link: https://github.com/Korex26/mxet300_lab.

```
SCUTTLE-07 basics git:(main) ➜ git push
Enumerating objects: 13, done.
Counting objects: 100% (13/13), done.
Delta compression using up to 4 threads
Compressing objects: 100% (8/8), done.
Writing objects: 100% (8/8), 3.06 KiB | 448.00 KiB/s, done.
Total 8 (delta 5), reused 0 (delta 0)
remote: Resolving deltas: 100% (5/5), completed with 5 local objects.
To https://github.com/Korex26/mxet300_lab
   80ab635..bd5cbf3  main -> main
```

GitHub Submission Figure 1: Commit and Push to GitHub on Visual Studio Code



GitHub Submission Figure 2: Commit and Push to GitHub on GitHub

References

K. Rex, R. Worley, *MXET 300 Lab Report 6*, 2025.

K. Rex, R. Worley, *MXET 300 Lab Report 5*, 2025.

K. Rex, R. Worley, *MXET 300 Lab Report 4*, 2025.

K. Rex, R. Worley, *MXET 300 Lab Report 3*, 2025.

K. Rex, R. Worley, *MXET 300 Lab Report 2*, 2025.

X. Song, *Lab 6 Manual - Lab 7 Manual - Closed-Loop Control*, 2025.

X. Song, *Lab 6 Manual - LIDAR and Obstacle Detection*, 2025.

X. Song, *Lab 5 Manual - Encoders _ Forward Kinematics*, 2025.

X. Song, *Lab 4 Manual - Motor Drivers _ Inverse Kinematics*, 2025.

X. Song, *Lab 3 - Compass Calibration*, 2025.

X. Song, *Lab 2 Manual - Displays and GUI*, 2025.

X. Song, *Lab 1 Manual - Pi Setup*, 2025.