Part II: Closed-loop Control of Elevator [Timeline: 01-03 April]

For Part II, you will explore four different control algorithms: **logic control, proportional control, proportional integral control, and proportional-integral-derivative control**. Each control algorithm will specify a duty cycle to drive the motor up or down based on measurements of its position and the desired position specified by the user. Recall that a closed-loop system continuously monitors the "state" of the plant (in this case, the height of the elevator) and makes a decision (to go up or down) based on the error. This implies that in each cycle (iteration), your code needs to:

- Read the sensor to know where the elevator is at that precise instant (done in Part I);
- Convert the sensor reading to a height value in inches (done in Part I);
- Compute the error signal: Desired Height minus measured height;
- Compute the motor duty cycle according to one of the control laws (logic, proportional or proportional-integral);
- Send the commands to the motor by setting the PWM duty cycle;
- Transmit sensor data and time across the serial port back to Tera Term.
- Repeat indefinitely or as long as specified by user.

You may consult ICE 02 – Actuation for reference on the use of DC motors.

> **Note:** From now on, you will use the encoder as the position sensor and ignore the data from the infrared sensor. Although the encoder does not directly measure position and can accumulate what is called "dead-reckoning error", its sensitivity and accuracy do not depend on room illumination and barely present any signal noise like the IR sensor does.
>
> **Note:** Ideally, you should stick with the same Elevator station through the completion of this project. Different stations may have slightly different behaviors and, therefore, your sensor calibration may not be valid across all stations. Unfortunately, sticking to the same station might be unfeasible at all time (for example, if you decide to work during non-lab hours other groups from other sections might be using your station). Therefore, you have been provided with a mbed program that should guide you easily and quickly through the calibration process so that you don't have to perform all the steps from Part I. The program is called `Quick_Calibration_BIN_FILE.bin`. Instructions are given in Appendix A.

Task IV: Design and implement a Logic Controller

Write a new mbed program that asks the user for a desired height in inches and then proceeds to regulate the elevator at the desired height. Let the user specify the duration of the experiment (in seconds) and the control frequency (between 2 – 100 Hz). In order to regulate the position of the elevator, the program should continuously read the encoder, estimate the position, compute the error, and make a logic decision (move up or down) based on the error.

Procedure: A **logic-based controller** uses simple logical statements (if-else) to generate the control input. In this case, the control input sent to the actuator (DC motor) is the duty cycle of the PWM signal.

1. Return to the mbed compiler. Create a new program called Elevator-closed-loop-logic-control (or something similar) using the provided template `Mbed_Elevator_Control_template.txt`. Follow this guidelines along with the template.
2. Modify the program to allow the following functionality:
   (a) Add and import the `mbed.h`, `Motor.h`, and `QEI.h` libraries.
   (b) Declare the QEI object for the encoder and the serial class for the mbed serial connection.
   (c) Declare the motor object, e.g., `Motor mot(PwmOut,DigOut1,DigOut2)` in the preamble of your main program, where `PwmOut`, `DigOut1`, and `DigOut2` correspond to:

- `PwmOut` is a PwmOut-type mbed's pin connected to the H-bridge's ME1 input
- `DigOut1` is a DigitalOut-type mbed's pin connected to the H-bridge's IN1 input
- `DigOut2` is a DigitalOut-type mbed's pin connected to one of the H-bridge's IN2 inputs

Very likely, the correct sequence is: `Motor mot(p25,p27,p28)`; interchanging the last two pins only changes the polarization of your motor (direction of motion).

(d) Declare a "dummy" `PwmOut` object right after declaring the motor. Use a PwmOut pin in the mbed that is not being used (e.g., p23).

Note: The `Motor.h` library automatically sets the period of all PWM signals (including the one to the motor) to 0.001 s (or 1 KHz). This period (or frequency) resonates with the DC motor, making it vibrate and producing a high pitch sound. To avoid this behavior, one needs to increase the frequency (or, equivalently, decrease the period) of all PWM signals. We will use a "fake" pin to do so. It is important that you define the dummy `PwmOut` after you have defined the `Motor` object. We will later, within the main function, set the period to 0.00005 s (or 20 KHz).

(e) Verify that the mbed code template has two timers to log data and regulate the control cycle. These variables should be already declared along with other variables needed to set the frequency of the control algorithm (similar to ICE 02 – Actuators).

(f) Closed-loop control systems typically require a fast sampling rate. To help execute your program faster, you will need to set a faster serial communication rate between the mbed and the PC (Tera Term). Set the baud rate of your PC serial object to **115200**. This should be done immediately after your main function but before the execution of your continuous loop. Keep in mind that you will need to use the same baud rate in the Tera Term window. You may use `pc.baud(115200)` assuming your serial object is called pc.

(g) Within your main function, you should have a while-loop that executes indefinitely like the one from Part 1. This while loop should ask the user for:

- The desired height of the elevator (in inches).
- The sampling frequency in Hz of your controller (a float value between 2 and 100). This represents how often you will update the control input for the motor. Save the frequency in the float variable `Fs`.
- The duration in seconds (as a float) for your experiment.

(h) At the beginning of each iteration of the indefinitely while loop, make sure you initialize the motor's speed to ZERO. You want to start and end your motor with a zero PWM value (steady rather than moving) after each run.

(i) Set the dummy variable's PWM period to 0.00005 s, e.g., `dummyPWM.period(0.00005)`. This will set the frequency of all PWM signals (including the one to the motor) to 20 KHz.

(j) After the user has entered the parameters asked, the program should start both timers (this should be already in place in the mbed code template).

(k) After the initialization of the timer and within the indefinite while loop, create a while loop that will execute the experiment as long as the user has specified. This while loop will implement the controller that will regulate the height of the elevator at the sampling frequency (period) also specified by the user. While inside this loop, you should:

- Read the encoder counts and cast them as a float
- Convert the encoder output to height using your calibration equation.
- Calculate the height error:

$$\text{height error} = \text{desired height} - \text{measured height}$$

Note: Do not use `error` as variable name, as the word "`error`" is already a function in the mbed library.

- Using the height error variable, create logic statements (if-else) that sets the duty cycle (dc) of the PWM signal to the motor according to the following the pseudo code:

```
If Error > 0          //The elevator is below desired height, GO UP
   dc = Positive Value (between 0 and 1)      //Start with 0.3
Else if Error < 0     //The elevator is above desired height, GO DOWN
   dc = Negative Value (between -1 and 0)     //Start with -0.3
Else                  //The elevator is at desired height, DO NOTHING
   dc = 0;
```

- Before you set this duty cycle signal to the motor, you need to consider the effect of gravity and friction in the elevator cage and motor. Similarly you need to consider the limits of the duty cycle and the polarization of your DC motor.
  - After your if-else logic statement, add the "dc hold value" approximation you found using the `Quick_Calibration_BIN_FILE.bin` file to the duty cycle (dc) from your logic statement. This value should be something between 0.3 and 0.6 and is meant to cancel out the effect of gravity and static friction. Since static friction is not uniform and the effect of gravity has not been measured, it is **better to underestimate** this value rather than to over-estimate it. Over estimating the "dc hold" value may lead the system to instability. Note that this dc hold value can highly differ among elevator stations. Do:

    dcNew = dc + dcHoldValue;
  - Using `if-else` statements, make sure that your new duty cycle never goes above 1.0 or below -1.0. The duty cycle should always be a number between -1 and 1.
  - Set the motor speed to the new duty cycle times the polarization of your motor. That's, multiply your total duty cycle by -1 if your motor has a reversed polarization. You should have found this using the `Quick_Calibration_BIN_FILE.bin` file.
- Print the current time (using `t.log`), the measured height in inches, the error in inches, and the duty cycle value to the serial PC port.

3. Run your code with a frequency of 100 Hz, a desired height of 16 inches and a duration time of 15 seconds. Your code should regulate the elevator at approximately 16 inches (bouncing is normal for a logic controller). If the elevator shoots to the top, it could mean several things (1) your polarization is wrong; (2) your dc hold value is too high; (3) you are not measuring and updating the position of the elevator with every cycle; or (4) you have a bug in your code. For debugging, the best practice is to look at your Tera Term data and to add printf statements at different stages as necessary.
4. Your elevator should not hit the top of the framework at any time. It cuts power to the elevator and affect the encoder counts. If it hits the top, you might need to lower the dc values in your logic statement.
5. Once you have a logic controller working, run your controller for the parameters in step 3. Log the data and import to MATLAB.

   Logging Data into MATLAB: Before running your experiment, Go to File→Log in the top menu of Tera Term. Choose a descriptive name (e.g., logic_controller_trail_1.txt or something similar) and a good location in your C drive for the .txt file. Check only the "Plain Text" box and click OK. Start your experiment. When completed, go to File→Show Log Dialog and click Pause and Close. Then, go to MATLAB. Option 1: In the top menu of MATLAB, choose Import Data. In "Output Type" select "Column Vectors." In Range, select the range of values that only includes numbers. Click Import. Option 2: Import data using a MATLAB script and the command: `importdata('data_file_name.txt')`. Type `help importdata` in command window for detailed instructions. Either way, you should now have four vectors of equal length representing the time, the height, the error, and the motor duty cycle.

6. Plot the height column vector against the time vector. If your controller is working properly, more than likely you should get something similar to Figure 3 (left).
7. Tune your controller such that:
    - **Rise time between 0.2 and 0.5 seconds**
    - **Peak time less than 0.8 seconds**
    - **Settling time is less than 1.5 seconds**
    - **Overshoot is less than 10%**
    - **Steady state error is less than 0.5 inches (note for a logic controller, there is always some bouncing such that it does not completely settles).**

   If your system has a higher overshoot or smaller peak time, think on how can you slow down the system. You might need to adjust the duty cycle values when the system comes up and down and the duty cycle to keep it on hold or steady (modify "duty hold value"). Once you have a controller that fits the performance criteria, save the data, and generate/save a plot with proper labels of:
    - Heights vs Time (similar to Figure 3 (right))
    - Error vs Time
    - Duty Cycle vs Time

   <u>Note</u>: The elevator goes from approximately 6 inches to 16 inches. The measures of overshoot, rise time, and settling time should be computed according to this jump. For 10% of overshoot, it "***roughly means***" that the elevator should not exceed about 17 inches at any time assuming that it stabilizes at 16 inches. You will need to continuously import data into MATLAB and run it to verify that you are meeting your performance criteria.

   <u>Note</u>: You may not meet all the performance criteria using a logic controller. Try to find the best response that come close to meet them all and do not spend to much time trying to fine tune the system.

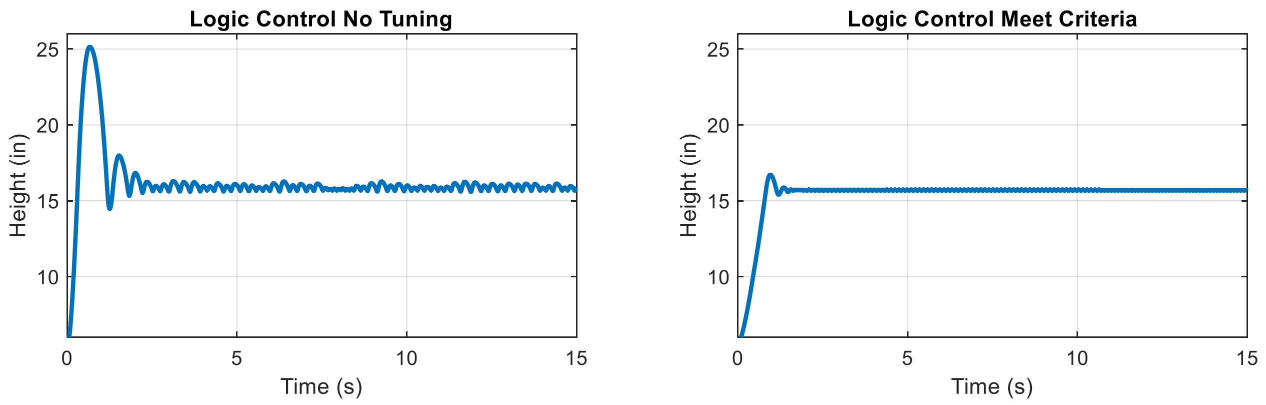

Figure 3. Performance of Logic Controller

8. Once you have a controller that performs as desired and have a plot to prove it, run the program again for a height of 16 inches, at a frequency of 100 Hz, but for a duration of 50 seconds. After 8 seconds, start adding weight inside the elevator. Observe what happens to the elevator. For your report, include a plot of (1) height vs time and (2) duty cycle vs time.
9. Once you have a controller that performs as desired, run your program for a desired height of 16 inches and a duration of 15 seconds. Run the program for a frequency of 40 Hz, 20 Hz, and 4 Hz. Describe and compare the performance of the system under lower frequencies. Generate and save a plot of <u>Height vs Time</u> when using a <u>frequency of 4 Hz</u>.

Summary for Part II – Logic Controller

Deliverables:

1. Check with your Instructor after the completion of this Task.
2. An mbed that implements the Logic Controller that meets the performance criteria.
3. Logic statement along duty cycle values used to meet performance criteria.
4. Keep record of data files (experiments).
5. Three Plots from step 7, two plots from step 8, and one plot from step 9.

Task V: Design and Implement a Proportional (P) Controller

Write a new mbed program that asks the user to choose between a proportional controller or a logic controller and then proceeds to implement the controller.

Procedure: A **proportional (P) controller** is one that sends to the actuator a signal that is proportional to the error. In the case of the elevator, your mbed should send a duty cycle signal to the motor that is proportional to height error.

1. Make a copy (clone) of your program from Task IV. Name it something similar to Elevator-Closed-Loop-Control.
2. Among the instructions and questions given to the user (such as desired height and control frequency) ask the user to pick a control type. You should give the user two options: (a) to implement the logic controller from Task IV or (b) to implement the Proportional controller to be designed in this task.
3. Rather than using a logic if-else statement and about 5 lines of code like the logic controller, a proportional controller only requires one line to set the duty cycle of the motor proportional to the error:

   dc = kp * Height Error

   In your program, modify (using either switch-cases or if-else statements) the code such that it either runs the logic if-else controller from Task IV or the proportional controller defined above, according to the use choice. Use a float for the value of kp (the proportional gain). A good start point for the kp is 0.04.

   Note: When switching among controllers, the switching should go right after you compute the height error and before you truncate the dc value to -1 or 1. You may use a different "dc hold value" for each controller. This will allow you more freedom when tuning each controller.

4. Run your code with a frequency of 100 Hz, a desired height of 16 inches and a duration time of 15 seconds. Your code should regulate the elevator at approximately 16. If the elevator shoots to the ceiling, it could mean several things (1) your polarization is wrong; (2) your kp gain is too high; (3) you are not measuring and updating the position of the elevator with every cycle; or (4) you have a bug in your code. For debugging, the best practice is to look at your Tera Term data and to add printf statements at different stages.

   Note: The elevator should not hit the ceiling of the framework at any time. It cuts power to the elevator and affects the encoder counts. If it hits the top, you might need to lower kp.

5. Once you have a proportional controller working, run your controller for the parameters in step 4. Log the data and import to MATLAB.
6. Plot the height column vector against the time vector and tune your controller such that you get a:
   - **Rise time between 0.2 and 0.5 seconds**
   - **Peak time less than 0.8 seconds**
   - **Settling time is less than 1.5 seconds**
   - **Overshoot is less than 10%**

- **Steady state error is less than 0.5 inches (note for a logic controller, there is always some bouncing such that it does not completely settles).**

Use your knowledge on proportional controller to tune the kp gain based on the performance. Once you have a controller that fits the performance criteria, save the data, and generate/save a plot with proper labels of:

- Heights vs Time
- Error vs Time
- Duty Cycle vs Time

<ins>**Note**</ins>: You may not meet all the performance criteria using a proportional controller. Try to find the best response that come close to meet them all.

7. Once you have a controller that performs as desired and have a plot to prove it, run the program again for a height of 16 inches, at a frequency of 100 Hz, but for a duration of 50 seconds. After 8 seconds, start adding weight inside the elevator. Observe what happens to the elevator. For your report, include a plot of (1) height vs time and (2) duty cycle vs time.
8. Once you have a controller that performs as desired, run your program for a desired height of 16 inches and a duration of 15 seconds. Run the program for a frequency of 40 Hz, 20 Hz, and 4 Hz. Describe and compare the performance of the system under lower frequencies. Compare response of proportional controller with that of a logic controller. Generate and save a plot of <ins>Height vs Time</ins> when using a <ins>frequency of 4 Hz</ins>.

Summary for Part II – Proportional Controller

Deliverables:

1. Check with your Instructor after the completion of this Task.
2. Record the value for kp and "dc hold".
3. Keep record of data files (experiments).
4. Three Plots from step 6, two plots from step 7, and one plot from step 8.

Task VI: Design and Implement a Proportional-Integral (PI) Controller

Write a new mbed program that asks the user to choose between a logic controller, a proportional controller, or a proportional-integral controller and then proceeds to implement it.

<ins>**Procedure**</ins>**:** A **proportional-integral (PI) controller** is one that sends to the actuator a signal that is proportional to the error and the integral of the error. In the case of the elevator, your mbed should send a duty cycle signal to the motor that is proportional to height error and the integral of the height error.

1. Modify your program from Task V such that it now allows for a third option: A PI controller.
2. To implement a PI controller, you will need to compute the integral of the error as a function of time (or iterations). In each iteration of your mbed program, you will need to update the integral value. To do so, you will use Euler's method. Euler's method is an iterative implementation of Riemann's left sum used to compute integrals.
   - Define the integral of the error as a float and initialize it as zero before you enter the control while loop. Within your infinite while-loop, right after you start your timers, define and declare a new variable for the integral, e.g., `float IntError = 0.0`. It is important that every time you start a new run, the integral is reset to zero.
   - Within your control loop, add the new choice of PI control (along with logic and p-controllers). The PI control statements should take error (e.g., `Error`), update the integratal of the error (`IntError`), and compute the dc signal (`dc`) to be sent to the motor. To update the integral of the error, you can do:

```
IntError = intError + Error*(1/Fs);
```

This is in fact Euler's method or Riemann's left sum. Then, compute the equation of the PI controller and add the "dc hold value" for your motor:

```
dc = Kp*Error + Ki*IntError;
dc = dc + dcHoldValue;
```

Kp and Ki values should be defined as a float. For the PI controller, start with a Kp value equal to 75% of the Kp value you found for your P-controller. For Ki, start with a low value of Ki = 0.005.

- Make sure that your are not passing a dc value to the motor greater than 1.0 or smaller than -1.0.

3. Run your code with a frequency of 100 Hz, a desired height of 16 inches and a duration time of 15 seconds. Your code should regulate the elevator at approximately 16 inches. Debug if necessary. Once you have a PI controller working, log the data and import to MATLAB.
4. Plot the height column vector against the time vector and tune your controller such that you get a:
   - **Rise time between 0.2 and 0.5 seconds**
   - **Peak time less than 0.8 seconds**
   - **Settling time is less than 1.5 seconds**
   - **Overshoot is less than 10%**
   - **Steady state error is less than 0.5 inches (note for a logic controller, there is always some bouncing such that it does not completely settles).**

   Use your knowledge on proportional-integral controller to tune the gain based on the performance. Once you have a controller that fits the performance criteria, save the data, and generate/save a plot with proper labels of:
   - Heights vs Time
   - Error vs Time
   - Duty Cycle vs Time
5. Once you have a controller that performs as desired and have a plot to prove it, run the program again for a height of 16 inches, at a frequency of 100 Hz, but for a duration of 50 seconds. After 8 seconds, start adding weight inside the elevator. Observe what happens to the elevator. For your report, include a plot of (1) height vs time and (2) duty cycle vs time.

Summary for Part II – Proportional-Integral Controller

Deliverables:

1. Check with your Instructor after the completion of this Task.
2. Record the value for kp, ki, and "dc hold".
3. Keep record of data files (experiments).
4. Three Plots from step 4 and two plots from step 5.

Task VII: Design and Implement a Proportional-Integral-Derivative (PID) Controller

Write a new mbed program that asks the user to choose between a logic controller, a proportional controller, a proportional-integral controller, and a proportional-integral-derivative controller, and then proceeds to implement it.

Procedure: A **proportional-integral-derivative (PID) controller** is one that sends to the actuator a signal that is proportional to the error, the integral of the error, and the derivative of the error. In the case of the elevator, your mbed should send a duty cycle signal to the motor that is proportional to height error, the integral of the height error, and the derivative of the height error.

1. Modify your program from Task V such that it now allows for a fourth option: A PID controller.
2. The PID controller is the same as the PI controller you designed in Task VI but with the addition of a derivative term. So now, in each iteration of your mbed program, you will need to compute the derivative of the error. Note that derivative of the error can be approximated by:

$$\frac{d(error)}{dt} = \frac{d(desired\ height)}{dt} - \frac{d(measured\ height)}{dt}$$
$$\frac{d(error)}{dt} = 0 - \frac{d(measured\ height)}{dt} \approx -\frac{\Delta(measured\ height)}{\Delta t}$$

Where $\Delta(measured\ height)$ and Δt are the change in height and the change in time, respectively, between two samples. To compute the change in height you will need to keep track of the last height sample. The change in time Δt is the period of your controller, which is also the inverse of the sampling frequency. To implement this estimation in your mbed, do the following.

- Define a new float variable that you will use to keep track of the last height sample and initialize it to 6 inches (this should be close enough to the elevator resting position) before you enter the control while loop. For instance, use `float lastHeight = 6.0`.
- Within your control loop, add the new choice of PID control (along with logic, P-controller, and PI controllers). The PID control statements should take error (e.g., `Error`), update the integral of the error (`IntError`), compute the derivative of the error, and compute the dc signal (`dc`) to be sent to the motor. To compute the derivative of the error, you can do:
```
float derError = -(measuredHeight-lastHeight)*(Fs);
lastHeight = measuredHeight;
```
  The first line computes the derivative of the error using the inverse of the control frequency (entered by the user) as Δt. The second line saves the last position sample to be used in the next iteration of your controller.
- Compute the equation of the PID controller and add the "dc hold value" for your motor:
```
dc = Kp*Error + Ki*IntError + Kd*derError;
dc = dc + dcHoldValue;
```
  Kp, Ki, and Kd values should all be float. For the PID controller, start with a Kp value equal to 150% of the Kp value you found for your PI-controller. Start with the same Ki value you use for the PI controller. For Kd, start with Kd = 0.005. Make sure that your are not passing a dc value to the motor greater than 1.0 or smaller than -1.0.

3. Run your code with a frequency of <u>100 Hz</u>, a desired height of <u>16 inches</u> and a duration time of <u>15 seconds</u>. Your code should regulate the elevator at approximately 16 inches. Debug if necessary. Once you have a PI controller working, log the data and import to MATLAB.
4. Plot the height column vector against the time vector and tune your controller such that you get a:
   - **Rise time between 0.2 and 0.5 seconds**
   - **Peak time less than 0.8 seconds**
   - **Settling time is less than 1.5 seconds**
   - **Overshoot is less than 10%**
   - **Steady state error is less than 0.5 inches (note for a logic controller, there is always some bouncing such that it does not completely settles).**

   Use your knowledge on proportional-integral controller to tune the gain based on the performance. Once you have a controller that fits the performance criteria, save the data, and generate/save a plot with proper labels of:
   - Heights vs Time
   - Error vs Time
   - Duty Cycle vs Time
5. Once you have a controller that performs as desired and have a plot to prove it, run the program again for a height of 16 inches, at a frequency of 100 Hz, but for a duration of 50 seconds. After 8 seconds, start adding weight inside the elevator. Observe what happens to the elevator. For your report, include a plot of (1) height vs time and (2) duty cycle vs time.

Summary for Part II – Proportional-Integral Controller

Deliverables:

1. Check with your Instructor after the completion of this Task.
2. Record the value for kp, ki, kd, and “dc hold”.
3. Keep record of data files (experiments).
4. Three Plots from step 4 and two plots from step 5.

Task VIII: Compare the performance of all four controller.

Using all plots and observations, draw conclusions about their performance. You can do so by measuring time response parameters such as %OS, steady-state error, peak time, rise time, and settling time. For most systems, it is ideal to have ZERO overshoot, ZERO steady state error, and quick (short) peak, rise, and settling time. Note that when measuring these parameters, you have to take into account “the jump” from where the elevator started to its final value.

In addition, create a plot of Height vs Time where you overlay the response of all four controllers in a single plot (for a desired height of 16 inches, 100 Hz sampling frequency, and a duration of 15 seconds, with no disturbances). Include the plot in your report. This plot will help you to draw comparisons.