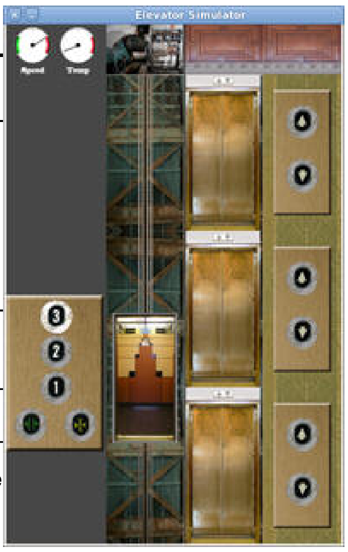
Project 2: Elevator Simulator Part A



Resources

Please refer to this section for resources to help complete the project.

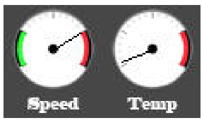
- Project Main Page
- The ElevatorLib Library Documentation

Project Prep

Add this new variable to main:

```
    double desiredSpeed = 1;
```

This is the speed we want the elevator to run at. In this case it is 1 meter per second. On the meter in the program it will look like this when it is working correctly:



Then change the code that reverses the elevator to this:

```
        if(GetFloor() > 3 && goingUp)
        {
            /* We are above the third floor. Reverse direction */
            goingUp = false;
            SetMotorPower(-1);
            ChangeLoading();
            desiredSpeed = -1;
        }
```

To get us started, we want the speed when going up to be 1 meter per second. Then we want the speed going down to be -1 meter per second.

When you add the PID controller in the next section, you will remove the SetMotorPower call from the code above.

Part A Tasks

Now we can indicate the tasks necessary for Project 2 Part A. The tasks are in sequential order. It is highly suggested that they be completed in sequential order. Do not proceed to the next task until the previous task is completed. This is for your benefit.

The PID Controller

Task: Implement a PID Controller to control the motor speed.

A PID Controller is a proportional-integral-derivative controller using in a closed loop feedback system. In our case we will use it to get the speed we want for our elevator. You can't just set the speed by choosing values for SetMotorPower, since the power needed varies with the loading of the elevator car. Instead, you need to change the power as necessary to get the speed you want. This is much like the cruise control in an automobile; when you go up hills it has to provide more power than when going downhill.

The formula for a PID controller is:

$$P(t) = K_p e(t) + K_i \int_0^t e(\tau)\,d\tau + K_d \frac{d}{dt} e(t)$$

This equation computes the power we will apply to our motor; the parameter to SetMotorPower.

At any point in time we have an observed speed: s(t) and a desired speed: d(t). s(t) is the value returned by GetVelocity(). d(t) is the value of the variable desiredSpeed. The error term e(t) is:

$$e(t) = d(t) - s(t)$$

A word about t: The value of t is the current time. Think of it as seconds since your system started running. So, when I say s(t), I mean the current velocity. You just get that using GetVelocity(). Think: s(t) = GetVelocity() (Don't put that in your program that way). There is no need for a variable in your program named "t". d(t) is the desired speed. Think: d(t) = desiredSpeed. You will set that and it will be just a simple variable.

The equation is a sum of three terms. These terms are called the proportional, integral, and differential terms. I am only going to require you to implement the proportional and integral terms, not the differential term, though you may do so if you want to.

Proportional Term

The proportional term of the equation is:

$$K_p e(t)$$

The value K_p is called the proportional gain. It is a tuning parameter. It is a number you will pick. If we are going too fast, e(t) will be negative because the actual speed is greater than the desired speed. So, this term will contribute values in the opposite direction. This term is trivial to implement. You will be required to select a K_p value in your program. Try different values to see what works. The Wiki page on PID controllers has a section on manual tuning. I found that I could increase it until the system becomes unstable (speed is erratic) and then drop well below that point and it works pretty well.

Hint: Implement just the proportional term first and get it working. You won't be able to get the speed exactly what you want, but you should be able to get it close. Add printf statements to output the speed continuously to see what you are getting.

The PID controller codes goes in the control loop. It does not go in an if statement, though. Your control loop is going to start out looking something like this:

```
    while(IsElevatorRunning())
    {
        if(GetFloor() > 3 && goingUp)
        {
            /* We are above the third floor. Reverse direction */
            goingUp = false;
            /* SetMotorPower(-1); this should be removed */
            ChangeLoading();
            desiredSpeed = -1;
        }

        if(GetFloor() < 1 && !goingUp)
        {
            /* We are below the first floor. Reverse direction */
            goingUp = true;
            /* SetMotorPower(1); this should be removed */
            ChangeLoading();
            desiredSpeed = 1;
        }

        /* Compute new motor power here */
        power = I'll leave this up to you...
        SetMotorPower(power);
    }
```

See how, for ever iteration of the loop we are setting the motor power.

You might want to think later on if there should be an **if** statement here. Should we be running the PID controller if the brake is on?

Integral Term

The integral term of the equation is:

$$K_i \int_0^t e(\tau)\,d\tau$$

The main purpose for the integral term is to cancel out any droop you get. The proportional term will select a speed, but it will not be the one you want. There will always be a bias, some difference between the desired speed and where it settles. The integral term helps to cancel that bias.

I have added a help page for this section on how to store and sum the error history.

Of course, we can't implement this term as it is. Instead, we will approximate the term with this equation:

$$K_i \sum_{i=0}^{N} e(t-i)\Delta t$$

Again, K_i is a tuning parameter, a number you pick for your program. You will increase this to cancel the droop. If you increase it too much, you will increase overshoot or make the system unstable. All this really is is the sum of the last N errors multiplied by the timer period Dt. In this program, Dt = 0.001, because the main loop runs 1000 times per second. You will need an array to keep track of the last N errors and be sure to initialize it to all zeros. N is how much history our controller uses for the integral term. While the theoretical solution uses history that goes back until the system started, most real systems use a finite amount of history. You can try various values, but I have had good results with 200 (1/5 second). So, you need an array that keeps track of the last 200 error values and a way to add them up. If you need help on how to save the last 200 errors, see the Project 2 Error History Help page.

You are not required to implement the differential term:

$$K_d \frac{d}{dt} e(t)$$

If you want to try it, you can estimate the derivative by subtracting the previous loop iteration error from the current error and multiplying that by Dt.

Speed Control

Once you have a working PID controller, add code to decelerate the elevator car as it gets to the end of its travel. A floor in the program is 3.28 meters tall. Please don't use the number 3.28 in your program; instead use the constant FloorSpacing. The elevator should run at 1 meter per second until it gets to within 1.64 meters of the floor. At that point it should decelerate until reaching the floor. If we are going up and we are 25cm from the floor, the speed should be 25 / 164 = 0.15. Once we get 5 cm from the floor the speed should be 5 / 164 = 0.03. This should happen continuously as you approach the floor, not at points where the speed abruptly changes. For example, you will no longer be able to use the simple test anymore:

```
        if(GetFloor() < 1 && !goingUp)
        {
            /* We are below the first floor. Reverse direction */
            goingUp = true;
            /* SetMotorPower(1); this should be removed */
            ChangeLoading();
            desiredSpeed = 1;
        }
```

Instead, use something more like this:

```
        if(GetPosition() < FloorTolerance && !goingUp)
        {
            /* We are below the first floor. Reverse direction */
            goingUp = true;
            /* SetMotorPower(1); this should be removed */
            ChangeLoading();
            desiredSpeed = 1;
        }
```

You need to test that the elevator has reached the floor to within the floor tolerance.

Notice: If we see numbers like 3.28, 1.64, 0.16 in your program we will deduct points. It is very bad to embed these numbers in your program, especially in an embedded system. If the floor spacing number were changed, would you be sure to get it everywhere in your program? Use: FloorSpacing, FloorSpacing / 2, and FloorTolerance.

Braking and Buttons

I want the final operation of your program submitted for Part A to be as follows...

When the program starts it should up to the third floor, decelerating until it gets to the third floor. Then the program should return to the first floor, also decelerating properly. At that time, stop the motor and turn on the brake.

At this time, pressing the up button on the first floor should start the elevator going up again and repeat the process. When the elevator starts, turn the light on the up button back off. You can determine if the up button has been pressed on the first floor using:

```
                if(GetCallLight(1, true))
                {
                    /* Button has been pressed */


                }
```

To turn off the light, use the SetCallLight function:

```
        /* Turn off the fisrt floor up call light */
        SetCallLight(1, true, false);
```

See the ElevatorLib documentation for details on the parameters to these functions.

When done, turn in only the file elevator.c via http://www.cse.msu.edu/handin

Grading Criteria

The following criteria are used to grade the project:

- 30 Points - program compiles and does something (even catching on fire is "something" :)
- 35 Points - Correct PID controller.
  - -35 for grievously incorrect PID controller operation
  - -25 for significant misunderstanding of PID controller operation
  - -15 for a PID controller with one significant oversight
  - -5 for PID controller that uses a loop for integral summation
  - -2 for not initializing a variable
- 35 Points - Correct elevator operation
  - 10 Points - Elevator does not catch fire
  - 15 Points - Elevator slows appropriately
  - 10 Points - Elevator responds to the "up" button

Hints and Suggestions

If you turn on the brake with a speed faster than 0.33 meters per second, the brake will fail and will no longer work in your program. Your passengers will either be trapped in the elevator or fall to their deaths.

A common issue with PID Controllers is overshoot, the tendency of the system to accelerate to a speed momentarily faster than desired, then return. This is common in many control systems. We will accept some overshoot.

The default behavior for the buttons is to turn on the light. So, you only need to check to see if the light is on to know if the button has been pressed.

We will keep track of who comes closest to 1.00 meters per second up and down, while not having excessive overshoot. The two closest students will automatically get full credit for Part A. The formula we will use for the determination is:

$$\max\left(\left|s_{up}-1\right|\left|s_{down}-1\right|\right)\max\left(s_{max}-s_{up},s_{max}-s_{down},0.01\right)$$

In this equation, s_{up} is the average speed in the up direction and s_{down} is the average speed in the down direction, both determined when the elevator is going full speed. s_{max} is the maximum speed ever observed in our tests.

If you want to see how your program stacks up, this is the code that I add in order to calculate the above values. Add this line to the top of your file:

```
#include <math.h>
```

Add this line to the top of the main method (with your other variable declarations)

```
double Sup=0, Sdown=0, Smax=0;
double numSup=0, numSdown=0;
```

Add these lines to the bottom of the while(IsElevatorRunning()) loop.

```
        /* Averaging - get the average speed before the elevator starts slowing down
         */
        if (goingUp && GetPosition() < 1.5*FloorSpacing && GetPosition() > .5*FloorSpacing)
        {
            Sup += GetVelocity();
            numSup += 1;
        }
        else if (!goingUp && GetPosition() > .5*FloorSpacing && GetPosition() < 1.5*FloorSpacing)
        {
            Sdown += GetVelocity();
            numSdown += 1;
        }
        if (fabs(GetVelocity()) > fabs(Smax))
            Smax = GetVelocity();
} // End of the while(IsElevatorRunning()) loop
```

Finally, add this code to the very end of the program, after the end of the while(IsElevatorRunning()) loop.

```
    Sup /= numSup;
    Sdown /= numSdown;
    printf("Average speed up: %lf\n", Sup);
    printf("Average speed down: %lf\n", Sdown);
    printf("Maximum speed: %lf\n", Smax);

    double maxAvgError = fmax(fabs(Sup-1), fabs(Sdown+1));
    double maxMaxError = fmax(fabs(Smax) - fabs(Sup), fabs(Smax) - fabs(Sdown));
    printf("Result: %lf\n", maxAvgError * maxMaxError);
```

If you want to see how you stack up, here are the results from the two winners last semester (the smallest "Result" number is the winner):

```
Average speed up: 0.998002
Average speed down: -1.001429
Maximum speed: -1.043050
Result: 0.000090

Average speed up: 0.998940
Average speed down: -0.998319
Maximum speed: -1.072676
Result: 0.000125
```

You may now proceed to Project 2 Part B.

CSE 251