Electronic Design®

HARRY HONG, Senior Embedded Software Engineer,
NeuronicWorks, www.neuronicworks.com

Positioning a Linear Servo Motor with a PID Controller

This article explains proportional integral derivative (PID) control, including the math behind it, with the use case of a linear servo motor.

Proportional integral derivative (PID) control is a common method used to regulate the dynamic behavior of a system. Examples are found in many industrial devices, where it's employed for control of temperature, pressure, flow, speed, or position, to name a few typical applications.

The theory and mathematics behind PID control have been the subject of much discussion. But how does one apply this math and theory to implement a real device? To demonstrate how that's done, this article will explore a thorough example.

The task of position control will be discussed for the case of a linear servo motor. To begin with, the mathematical function governing the PID controller's operation is presented. We'll show how the parts of the function fit together in a practical design. Specifically, we'll address considerations for interfacing elements in the electrical circuit to accomplish those parts of the PID function for position control, as well as what's involved in implementing the function in the firmware code of a microcontroller that will do the controlling.

PID Fundamentals

The universal mathematical function which is the basis of any PID control application can be stated as follow:

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

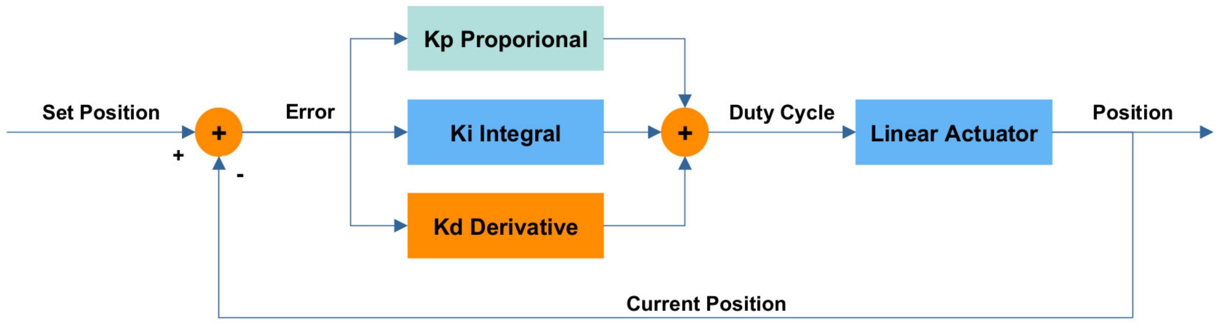
where *e(t)* is an error value:

$$e(t) = r(t) - y(t)$$

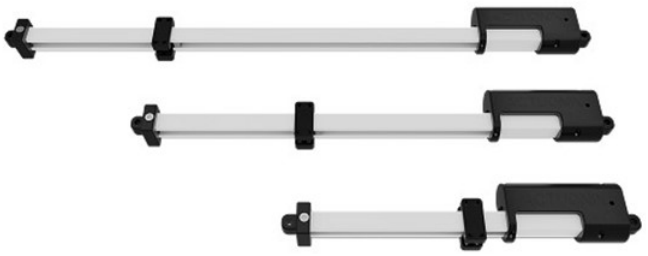
and for the case of position control, *r(t)* is a set position and *y(t)* is the current position.

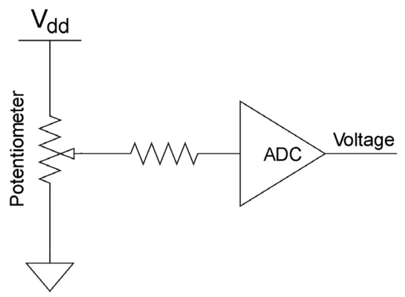
A diagram can provide us a much more intuitive understanding of how this math works when applied to a servo motor. *Figure 1* illustrates the block diagram of a linear servo PID control system.

Some key elements of the system depicted in *Figure 1* are a **Set Position** input (the setpoint, or our target position for the linear actuator), a pulse-width-modulated (PWM) sig-



1. Linear servo PID control system.


2. Shown are examples of linear actuators. (Source: actuonix.com)



3. Potentiometer ADC circuit.

nal of some **Duty Cycle** which drives the actuator, and the **Current Position** of the actuator. These correspond with the quantities *r(t), u(t),* and *y(t),* respectively, in the mathematical equations.

It's called closed-loop control because a feedback loop relays information about the current state back into the system, allowing it to obtain the difference between this current state and the desired setpoint, which must be corrected. Being specific for our case, **Current Position** is subtracted from **Set Position** to obtain the **Error** (or difference) signal, as shown above. This **Error** corresponds with the quantity *e(t)*.

And as mentioned at the beginning of the article, PID stands for Proportional, Integral, and Derivative. These refer to the three control signals generated for regulating a PID control system's operation.

As indicated in the math and the diagram, the three control signals are produced from the **Error** signal, are output from the **Proportional**, **Integral**, and **Derivative** blocks—also labeled with their respective gains K_p, K_i, and K_d—and are combined to produce the **Duty Cycle** of the PWM signal driving the actuator.

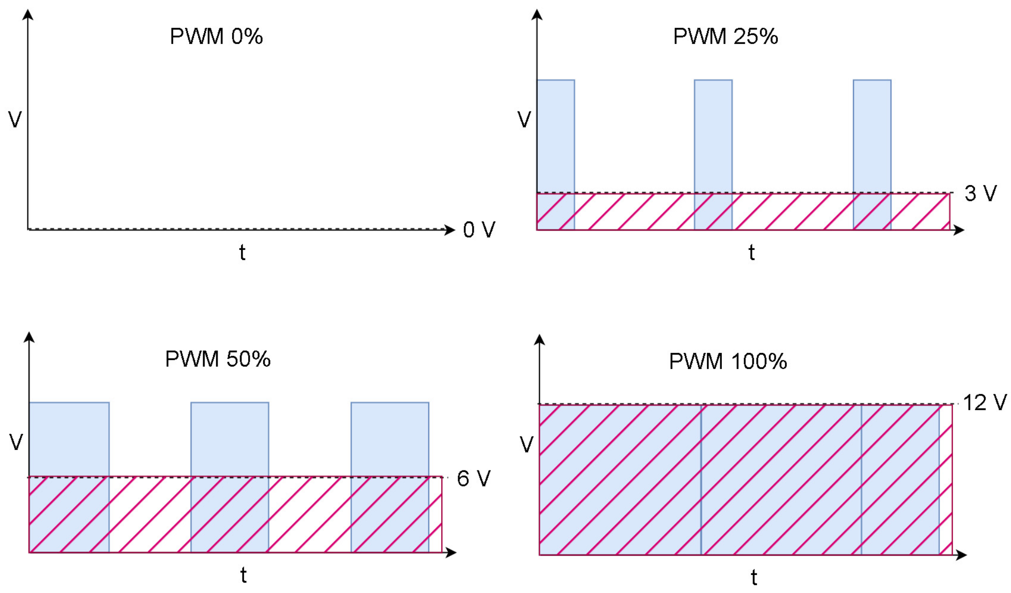
Now that we have described the structure of the system, we want to implement it in firmware. But to do this, we need to understand how to interface the linear actuator with the microcontroller. In particular, how do we obtain the **Duty Cycle** signal from the PID function to drive the actuator, and how can the actuator generate the **Current Position** signal to feed back to the PID function?

Subsequently, we can explain how to translate the PID function into firmware source code written in the C programming language. Some sample data demonstrating the working implementation will then be presented as the basis for understanding the roles of the three PID control signals and how to tune their performance.

Electrically Interfacing the Linear Actuator

A linear actuator is used to lift, tilt, pull, or push objects *(Fig. 2)*. The micro linear actuator we use here consists of a dc servo motor for the actuation part and a potentiometer for the position sensing part.

For this unit, the PID controller board needs to output a 12-V PWM signal to control motor speed, and use an analog-to-digital converter (ADC) channel to sense the position of the actuator. Accordingly, we should configure two



4. PWM signal graphs.

GPIO pins on the microcontroller, one for the PWM and the other for the ADC.

The position output of the linear actuator is a resistance value. If the potentiometer is connected between a power rail V_{dd} and ground *(Fig. 3)*, then the resistance at the wiper can be measured as the output of a simple voltage divider. The range and unit of position are changed from 0 ~ 10,000 Ω to 0 ~ V_{dd} V, and the ADC converts the voltage to a digital value that's our implementation's current position *y(t)*. If the ADC's resolution is 10 bits, this digital value is between 0 and 1023.

It's convenient for our controller output *u(t)* also to be a digital value representing a voltage. However, this controller output drives the linear actuator, which isn't expecting a varying voltage as input to control its speed, but rather a fixed-voltage PWM signal with varying duty cycle. Therefore, a conversion is needed.

The graphs in *Figure 4* show how voltages from 0 V up to 12 V translate to a 12-V PWM signal with variable-width pulses from 0% to 100% duty cycle. To be strictly correct, voltages above 12 V also must be accounted for, and must translate to a duty cycle of 100%, since the math in no way constrains the controller output to be below 12 V.

As a final remark regarding actuator interfacing, we should emphasize that it's only the nature of the speed-control and position-sensing features of our chosen device that have guided us to designate *u(t)* and *y(t)* both to be (conveniently) voltage quantities in our implementation. These quantities aren't otherwise related, and in another application might not even be in the same unit of measurement if the nature of the controlled device's interfaces were to dictate otherwise.

1. A fixed interval *T* is designated to be the time between iterations, i.e., their period.
2. Evaluation of the error value *e(t)* at the moment *t* in continuous time is replaced with evaluation at iteration *n* in discrete time, i.e., $e(n) = r(n) - y(n)$, where $n = 0, 1, 2, \ldots$
3. The continuous time integral of *e(t)* is replaced by a discrete time summation of $e(n)T$.
4. The continuous time derivative of *e(t)* is replaced by the linear slope of *e(n)* between the previous and the current iteration – i.e.:

$$\frac{e(n) - e(n-1)}{T}$$

And, therefore, the discrete time output signal *u(n)* evaluated at iteration *n* can be stated as follows:

$$u(n) = K_p \times e(n) + K_i \sum e(n)T + K_d \frac{e(n) - e(n-1)}{T}$$

Now we can implement the discrete time PID control function. In the example C code *(see codelist)*, variables and constants are given names that closely match the corresponding elements in the mathematical equations. This code can be executed in each iteration of the PID firmware, typically within a timer interrupt configured to trigger every *T* milliseconds.

[See codelist below]

What remains to be done is assign proper values to the PID gains K_p, K_i, and K_d so that the system performs correctly when asked to move to a chosen setpoint. We will manually select different values for these gains to investigate their effect on position control, and in so doing, demonstrate a common approach for tuning them. We also will

Writing the Firmware

For firmware to function as a PID controller, it must determine the error value *e(t)*, evaluate the PID function to adjust the signal *u(t)* that drives the device, and do this continuously over time. However, for firmware execution to perform a task in truly a continuous fashion isn't a feasible concept. The closest it can come is to repeat—or iterate—the task quickly at short time intervals.

If that task is the PID algorithm, then its continuous time math needs to be replaced with a discrete time version, with the following implications:

```
/* Current Error - Proportional term */
e = r - y;

/* Accumulated Error - Integral term */
totalError += e;

/* Difference of Error - Derivative term */
deltaError = e - previousError;

/* PID control */
u = Kp * e + Ki * (totalError * T) + Kd * (deltaError / T);

/* Also prepare for next iteration - set previous to Current Error */
previousError = e;
```

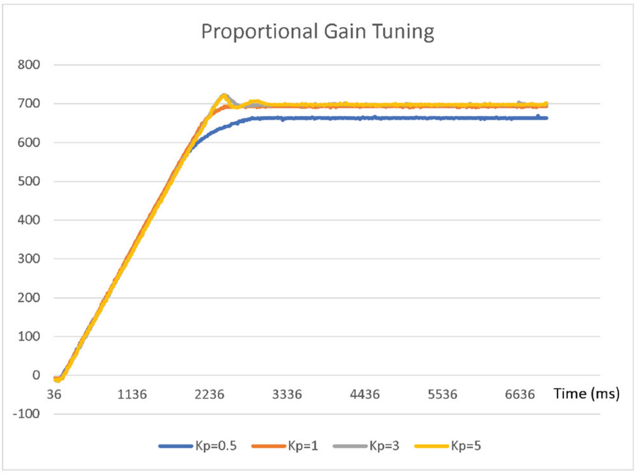
provide some insight into the purpose of each term in the control function.

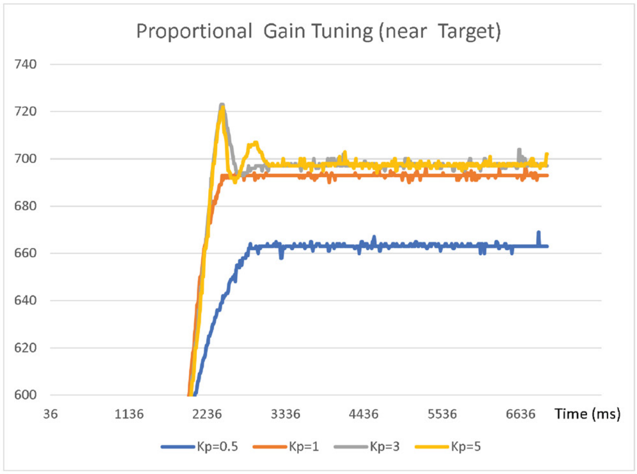
PID Gain Tuning

There are several criteria for evaluating the performance of the system, including *dead time*, *rise time*, *overshoot*, *settling time*, and *steady-state error*. While performance expectations should be defined according to these criteria before tuning the PID gains, such expectations depend on the application's requirements. So for the purpose of this article, it will be sufficient to provide some sense of when various criteria are affected by the adjustment of the different gains.

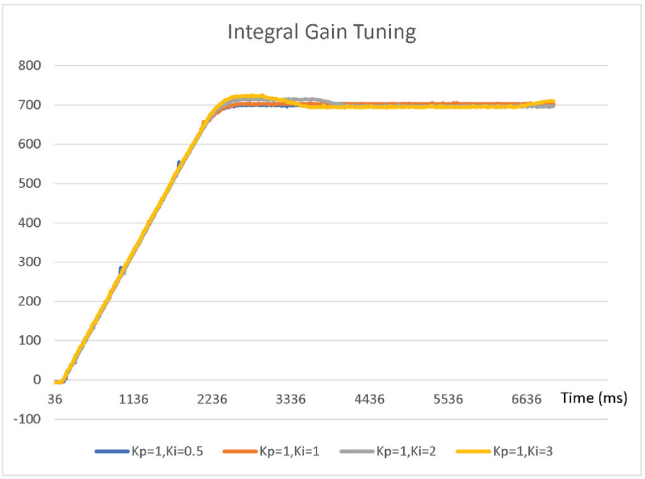
Each of the K_p, K_i, and K_d gains will be tuned separately, and in that order, given a selected setpoint. More specifically, the code will be executed with one of the gains set to a different value for each execution, and the value of r set to 700.

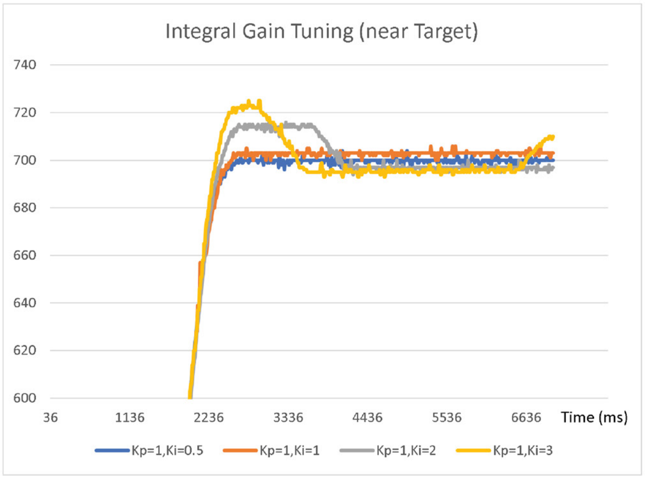
As to the relevance of this 700 value, the reader should recall that **Current Position** is a digital value representing





5. Proportional gain tuning.





6. Integral gain tuning.

the voltage obtained from the actuator's potentiometer, and **Current Position** is now represented by the variable y in our code. The **Set Position**—represented by the variable r in our code—is a digital value in the same range, which as mentioned previously is between 0 and 1023 if the ADC has 10-bit resolution. A setpoint value of 700 is therefore reasonable, although arbitrary.

Tuning K_p to get close to the target position

K_p is the proportional gain. The proportional term of the control function compensates for the current error by moving the linear actuator with a signal proportional to this current error. It makes sense that the proportional term is used to get the current position close to the target, since this error is the difference between the actuator's set position and its current position. The proportional term makes the control function seek to reduce it to zero.

In this first step of tuning, we set the integral and derivative gains K_i and K_d to zero and increase the proportional gain K_p until the actuator settles near the target position (700). A proportional gain that's too high will cause oscillation.

The graphs in *Figure 5* show how the actuator's current position changes over time for different values of K_p. We will select $K_p = 1$, observing that it causes the current position to settle near the target and with the fastest settling time.

The reader will notice that there's a residual *steady-state error*, where the final current position is offset from the target setpoint position. This offset is common in the case of a purely proportional controller and will be eliminated when the integral gain is tuned in the next step.

Tuning K_i to eliminate steady-state error

K_i is the integral gain. The integral term of the control function compensates for the past error by moving the linear actuator with a signal proportional to the amount of this past error, which has accumulated over time. It makes sense that the integral term is used to eliminate *steady-state error*, since this error is a constant offset that grows the integral over time, thus making the control function seek to reduce it to zero.

In this second step of tuning, we keep the proportional gain $K_p = 1$ selected in the first step, set the derivative gain K_d to zero, and increase the integral gain K_i until the actuator settles much nearer the target position (700)—i.e., until the *stead-state error* is close to zero.

The graphs in *Figure 6* show how the actuator's current position changes over time for different values of K_i with $K_p = 1$. The results for $K_i = 0.5$ may be quite satisfactory for a given set of requirements, and we may elect not to involve a derivative term, in which case the solution would be a proportional integral (PI) controller.

Alternatively, though, we may prefer to select $K_i = 2$, perhaps due to the improved *rise time* shown in its results. The reader will notice that the better *rise time* in this case comes at the expense of an *overshoot*. This *overshoot* will be eliminated when the derivative gain is tuned in the next step.

Tuning K_d to eliminate overshoot

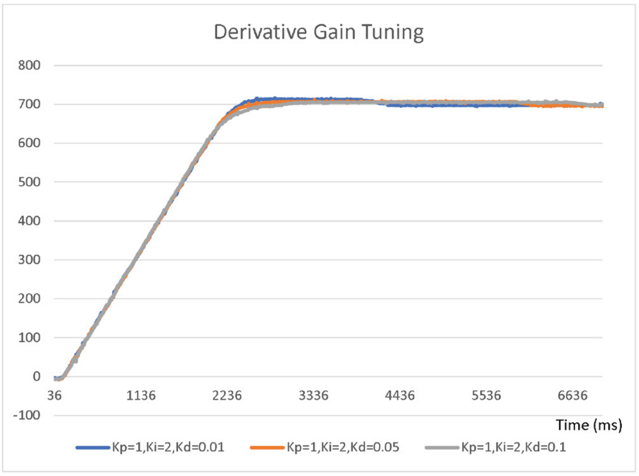
K_d is the derivative gain. The derivative term of the control function compensates for the future (estimated) error by moving the linear actuator with a signal proportional to the amount of this future error as estimated based on the time derivative of the error, i.e., its rate of change.

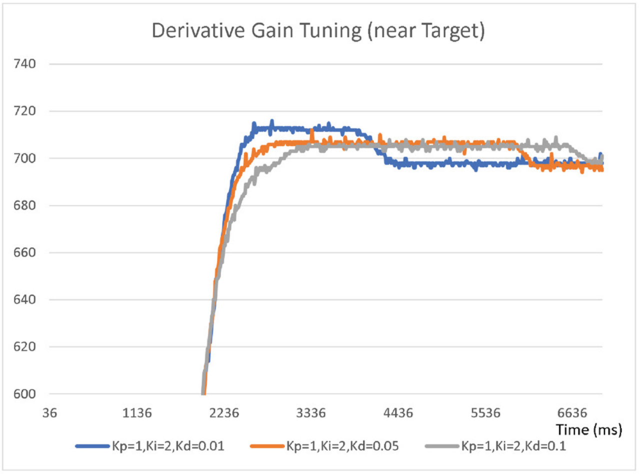
It makes sense that the derivative term is used to eliminate transient effects such as *overshoot*, which are naturally reflected in the time derivative, thus making the control function seek to reduce them to zero. Improved stability in the presence of disturbances and better *settling time* are additional related benefits. Note, however, that the derivative term can make a control system unstable if the error signal is very noisy.

In this third step of tuning, we keep the proportional and integral gains $K_p = 1$ and $K_i = 2$ selected in the first two steps and increase the derivative gain K_d until the *overshoot* is eliminated. A derivative gain that's too high will cause oscillation.

The graphs in *Figure 7* show how the actuator's current position changes over time for different values of K_d with $K_p = 1$ and $K_i = 2$. We'll select $K_d = 0.05$, observing that it effectively reduces *overshoot* while maintaining improved *rise time*.

Then, in the version of our controller, which has all three signals Proportional, Integral and Derivative enabled, we've successfully tuned their gains for proper controller behavior. We've also determined that the gain values should be $K_p = 1$, $K_i = 2$, and $K_d = 0.05$.





7. Derivative gain tuning

Conclusion

This article about PID control explained the mathematics at the heart of a PID controller and provided a practical example of how to implement this math to run on a microcontroller. Practical considerations were discussed about the nature of signals between the microcontroller and a dc servo motor for the purpose of position control.

Finally, some data was presented to demonstrate how the Proportional, Integral, and Derivative terms of the control function can be manually tuned for proper performance. It also gives the reader an idea of the purpose served by each of them in the PID algorithm.

Instead of designing PID control into a custom embedded device, generic off-the-shelf PID controllers are available alternatives in the industrial market, some of which, for example, are based on programmable logic controllers (PLCs). These may satisfy the needs of many users.

However, it may not be satisfactory for your application if it requires non-standard functions related to your plant processes. Or if it has special data communication needs, or if the generic controller has unneeded features you wish to avoid for a cost-sensitive application. A custom PID controller design is an option in such cases.