3. PID Tuning

Pieter P

Use the Python script Python/Tuning.py to tune the controllers. Configure the Arduino code as described in the Python docstring, and make sure to select the correct serial port. To run the script, you'll have to install the `numpy`, `matplotlib` and `pyserial` packages:

```
$ python3 -m pip install numpy matplotlib pyserial
```

After installing the dependencies, verifying the port name, and specifying the desired tunings in the `tunings` list, run the script using:

```
$ python3 Python/Tuning.py
```

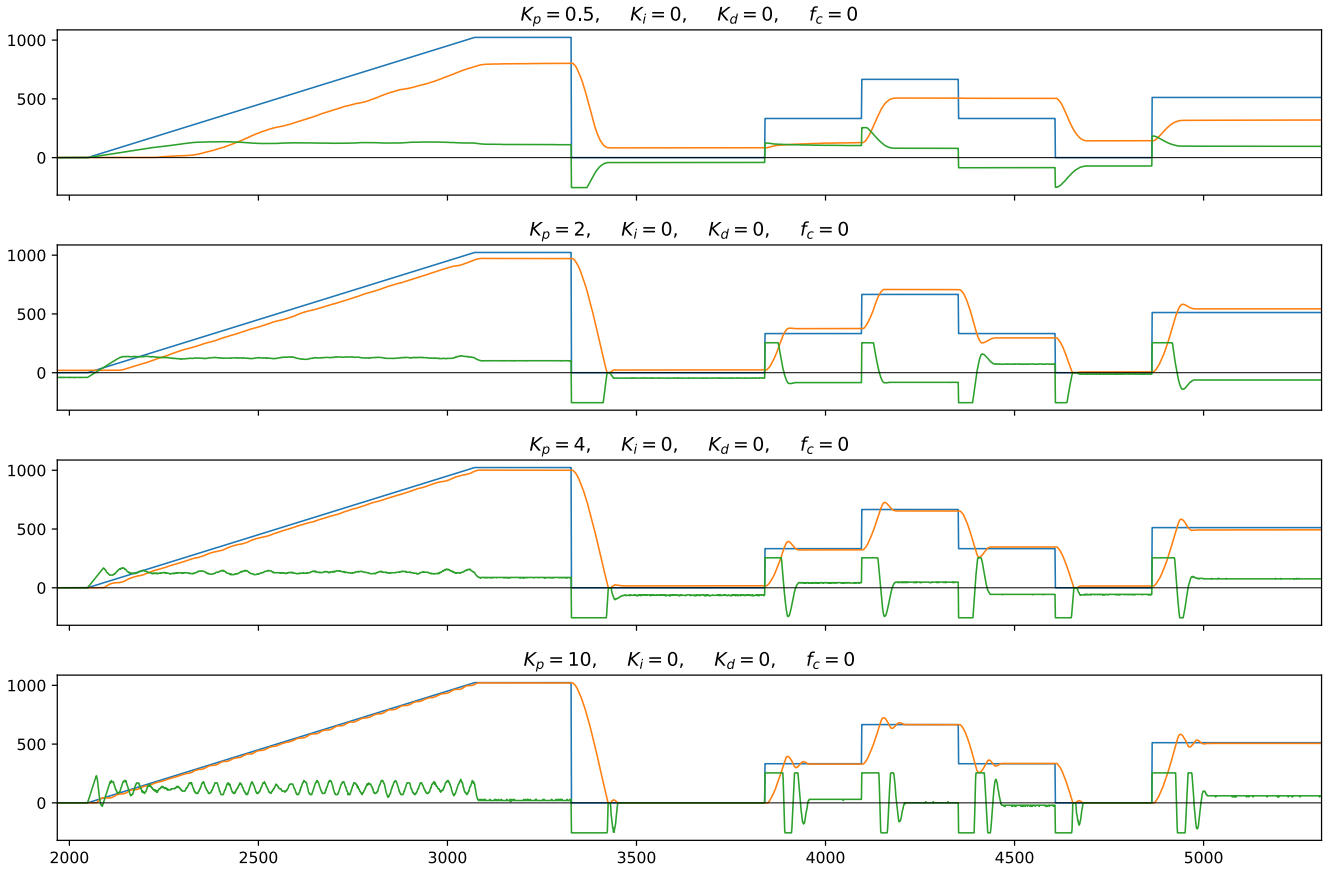
The script will then connect to the Arduino and perform an experiment with each of the tunings specified in the list. The results of these experiments are stored in a file, and then plotted in a graph as shown below.

Manual tuning

In this first section, we'll develop an intuition of what each of the four parameters of our PID controller does by tuning them manually. Later we'll use the heuristic Ziegler–Nichols method to get good starting values for the tuning parameters.

Proportional control

We start by setting K_i and K_d to zero, and by disabling the low-pass filter. We'll only vary the proportional gain factor K_p. This means that the control signal we send to the motor is proportional to the distance between the actual fader position and the desired fader position.
For example, if the target position is 512 and the actual position is 500, with a gain of $K_p = 10$, this would result in a PWM output of $10 \cdot (512 - 500) = 120$, or a duty cycle of around 47%.



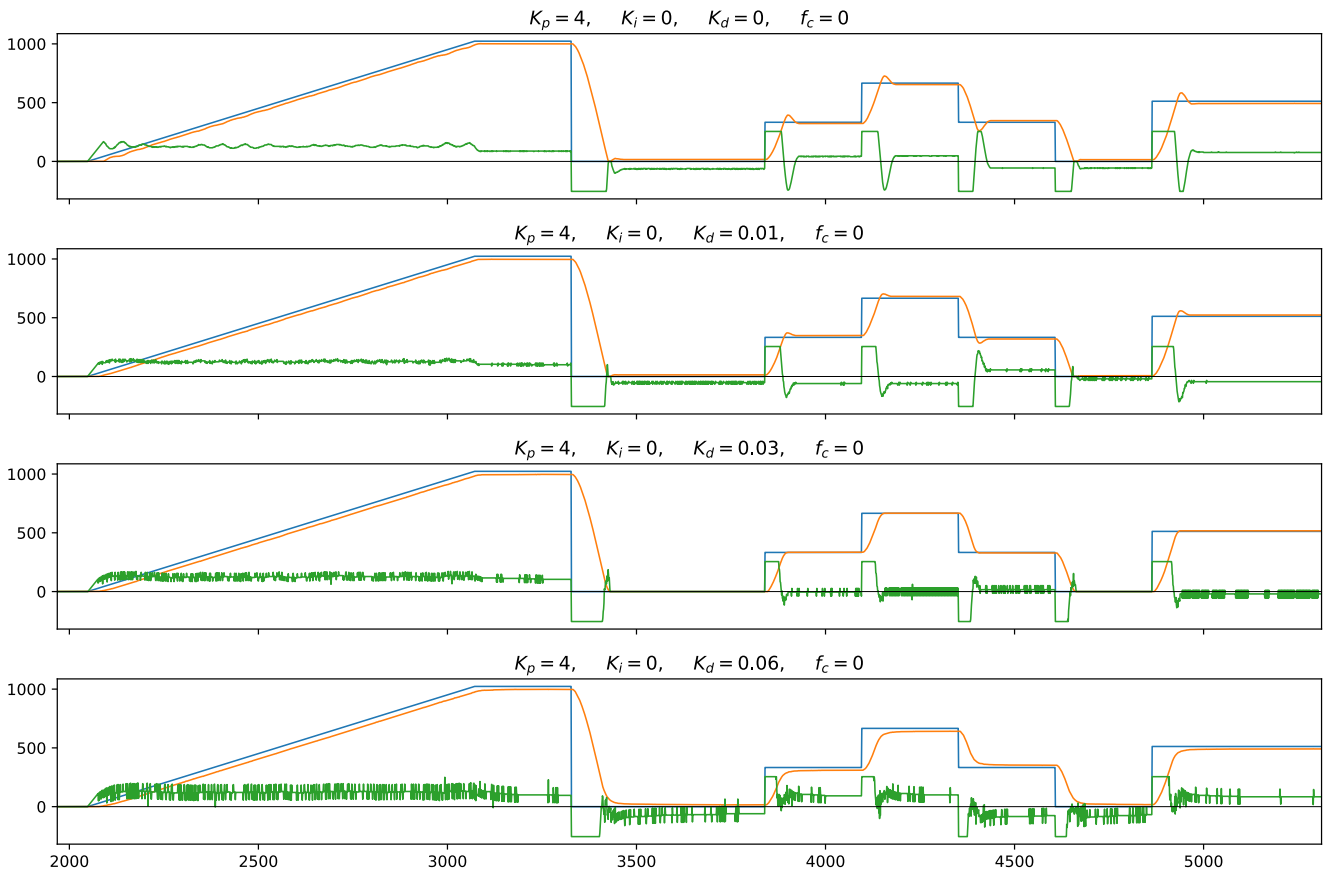
The figure above shows the results of four experiments with different values for the proportional gain K_p.
The blue line is the reference or target position, the orange line is the actual position of the fader, and the green line is the control signal sent to the motor.

We can make two main observations based on these experiments:

1. If the proportional gain is too low — as is the case in the top graph — the controller results in very poor tracking and in a large steady-state error. Even though the steady-state control signal is nonzero, it is not large enough to overcome the static friction of the fader.
2. If the proportional gain is too high — such as in the bottom two figures — the tracking and steady-state errors are smaller, but the system overshoots the setpoint significantly, with ringing after steep transients, and oscillations while tracking the ramp.

Proportional-derivative control

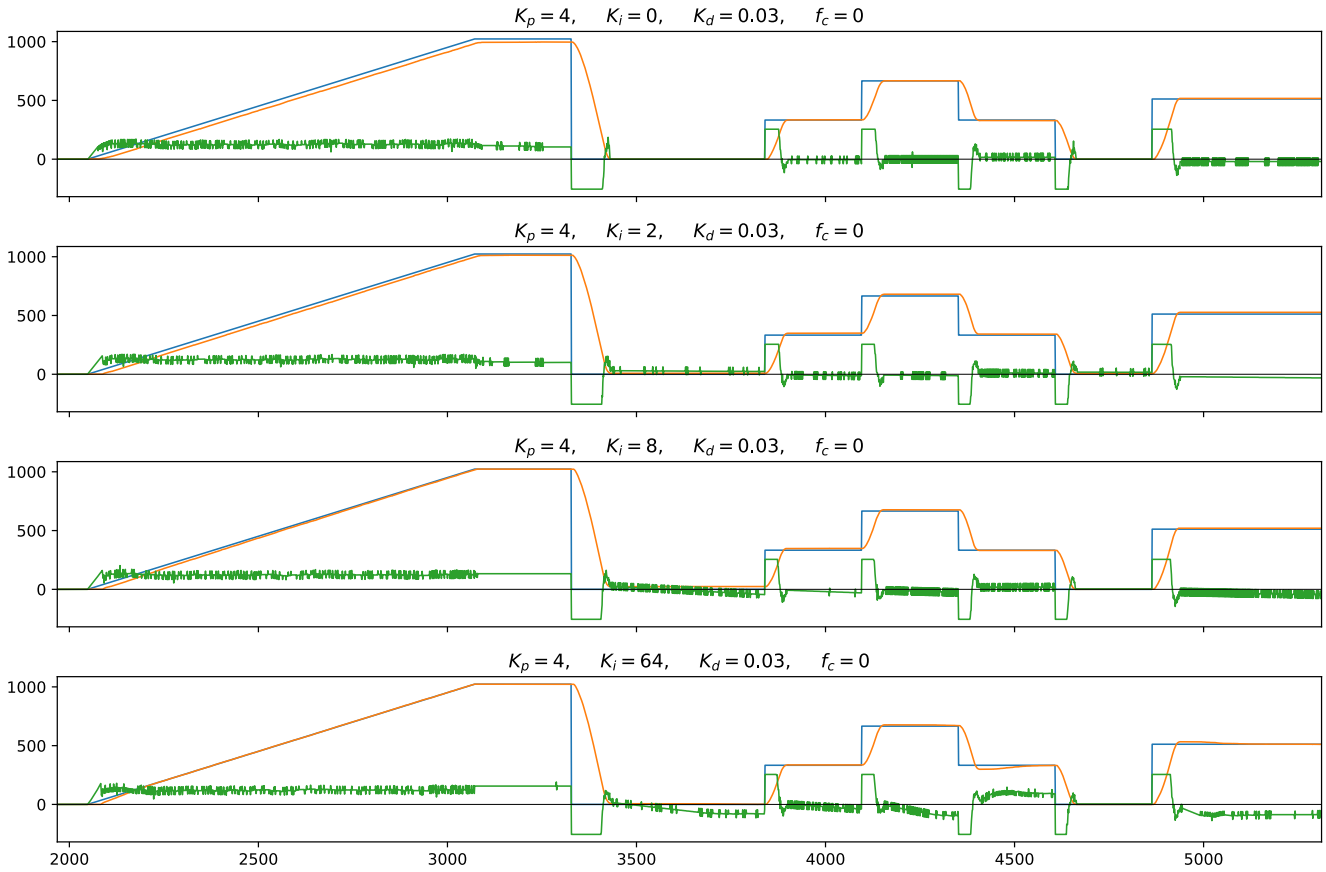
In order to minimize the overshoot, the derivative gain K_d is increased. This penalizes the velocity of the fader, so it won't approach the new target position too quickly, which would cause the fader to overshoot.



1. By increasing the derivative gain, the amount of overshoot decreases in the second graph, and is almost completely gone in the third graph.
2. Increasing the derivative gain even further in the fourth graph makes the controller more sluggish, it takes much more time to get close to the target position after a setpoint change.
3. The higher the derivative gain, the higher the noise in the control signal, which is noticeable as buzzing and slight rattling of the fader. This is because the derivative of the position is inherently noisy due to the discrete and quantized nature of the position measurement and the high-pass characteristics of the derivative operator.

Proportional-integral-derivative control

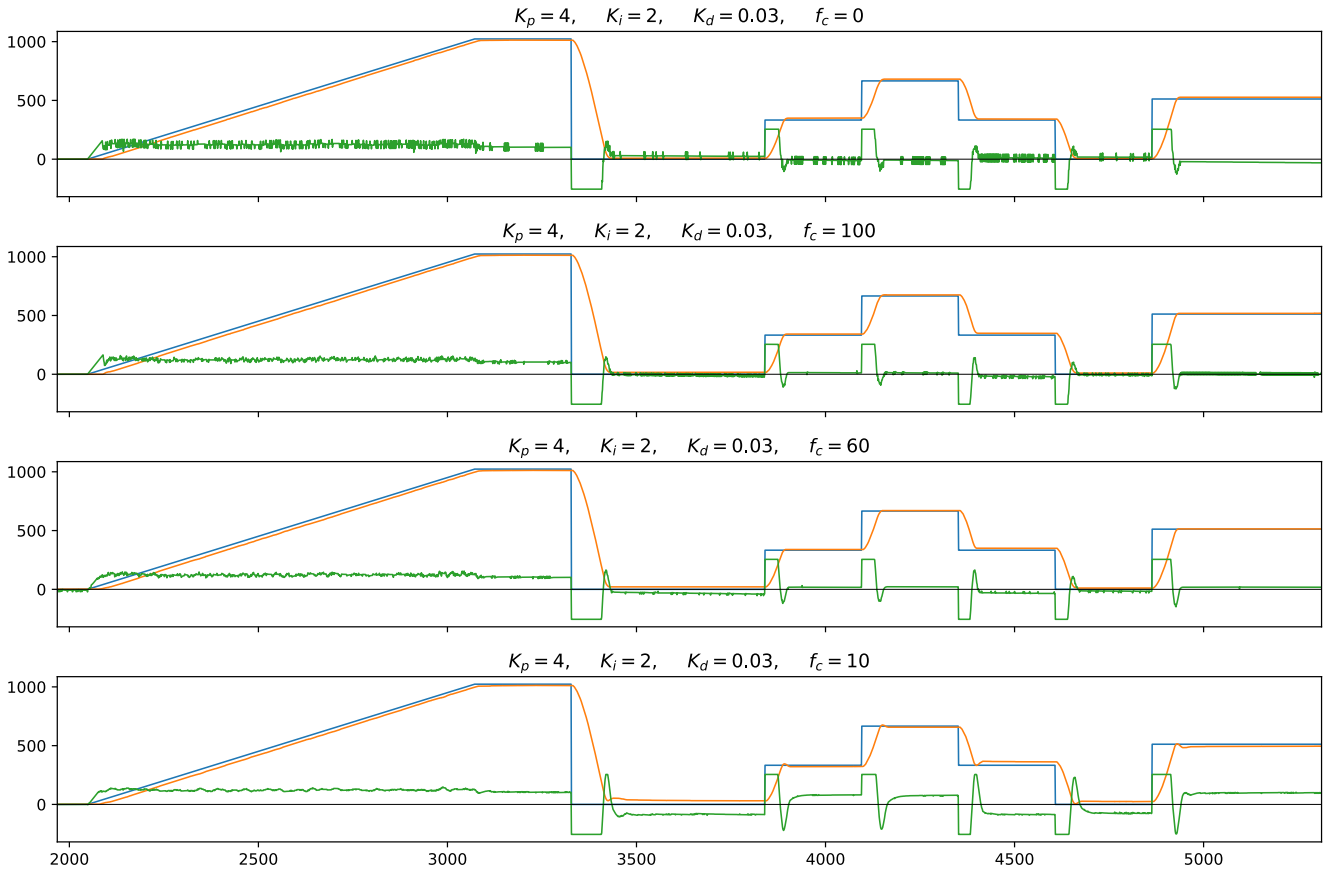
The addition of the derivative component in the previous section worked well to minimize the overshoot, but there is still some constant tracking error while following the ramp. This is addressed by the integral component: if the actual position is consistently below the target position, the integral will sum up those positive errors over time, which increases the control signal and reduces the tracking and steady-state errors.



1. The higher the integral gain, the smaller the tracking error for the ramp reference, and the faster the fader catches up with the reference.
2. If the integral gain is very large, the fader sometimes slightly overshoots the setpoint after steep transients — e.g. the rightmost step in the bottom graph. Despite this overshoot, the controller settles to zero steady-state error rather quickly.

Derivative filtering

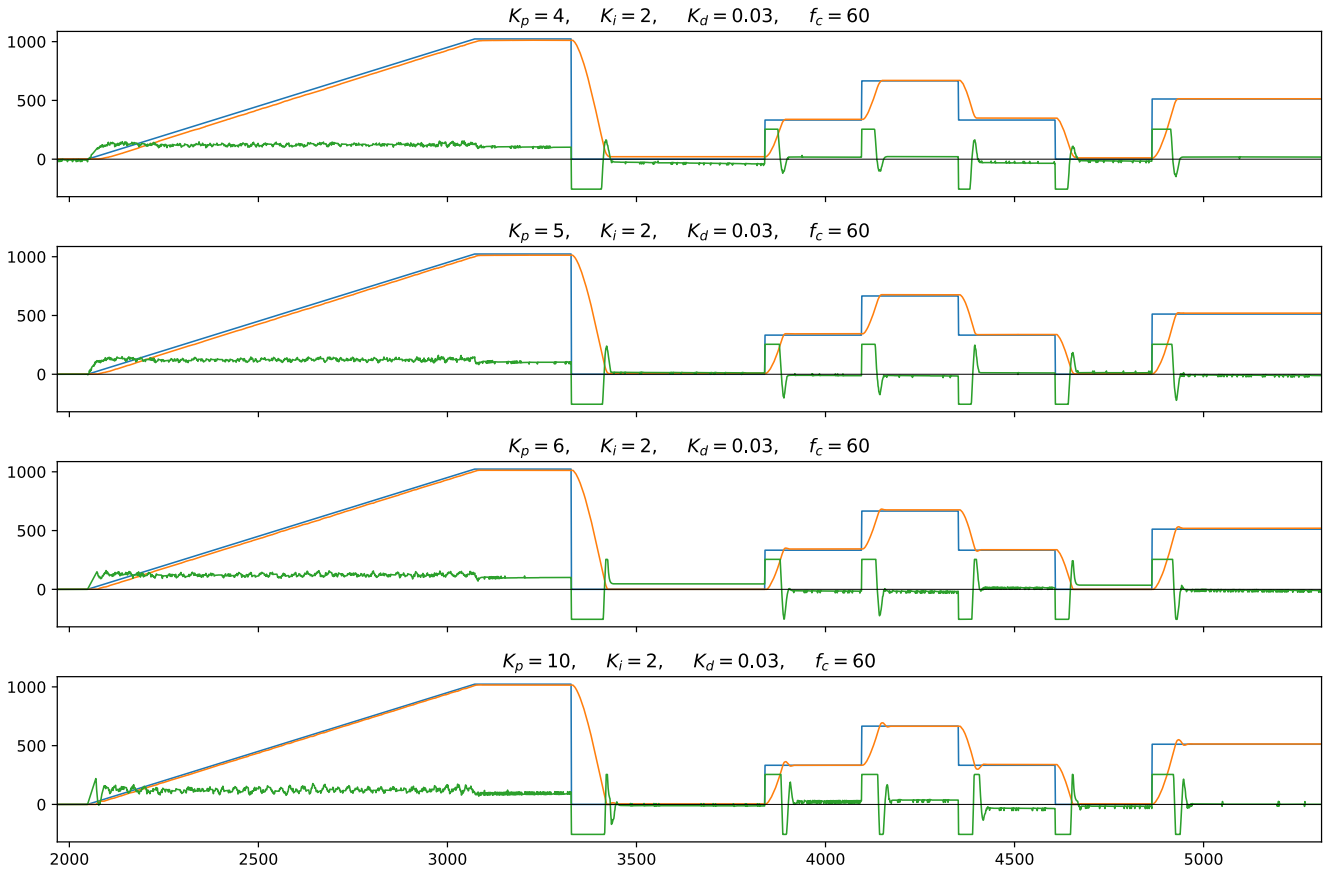
We'll now address the noisy derivative component by enabling the low-pass filter, tuning the cut-off frequency f_c (the -3dB point). If $f_c = 0$, the filter is disabled. If the filter is enabled, higher values of the cut-off frequency have little effect, because the filter allows frequencies below f_c to pass through mostly unhindered. Lowering the value of f_c, the filter becomes more and more restrictive, and you can more clearly see the effects.



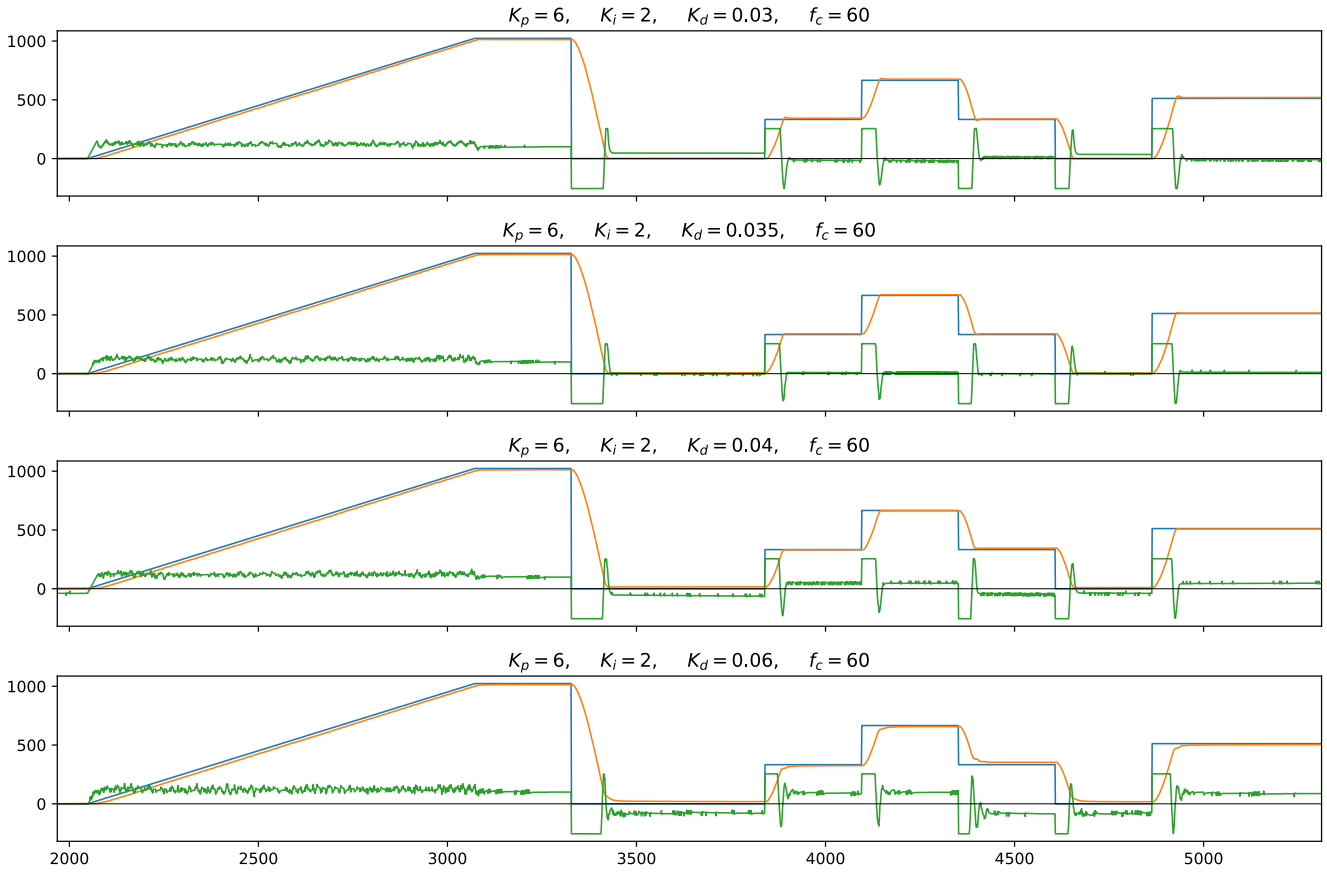
1. A lower cut-off frequency results in less noise in the control signal, which reduces the buzzing and rattling of the fader.
2. If the cut-off frequency is too low — as in the bottom figure — the filter introduces too much delay into the loop, which causes the derivative component to lag behind the actual velocity of the fader, reducing the effect of K_d, and resulting in higher overshoot.

Proportional fine-tuning

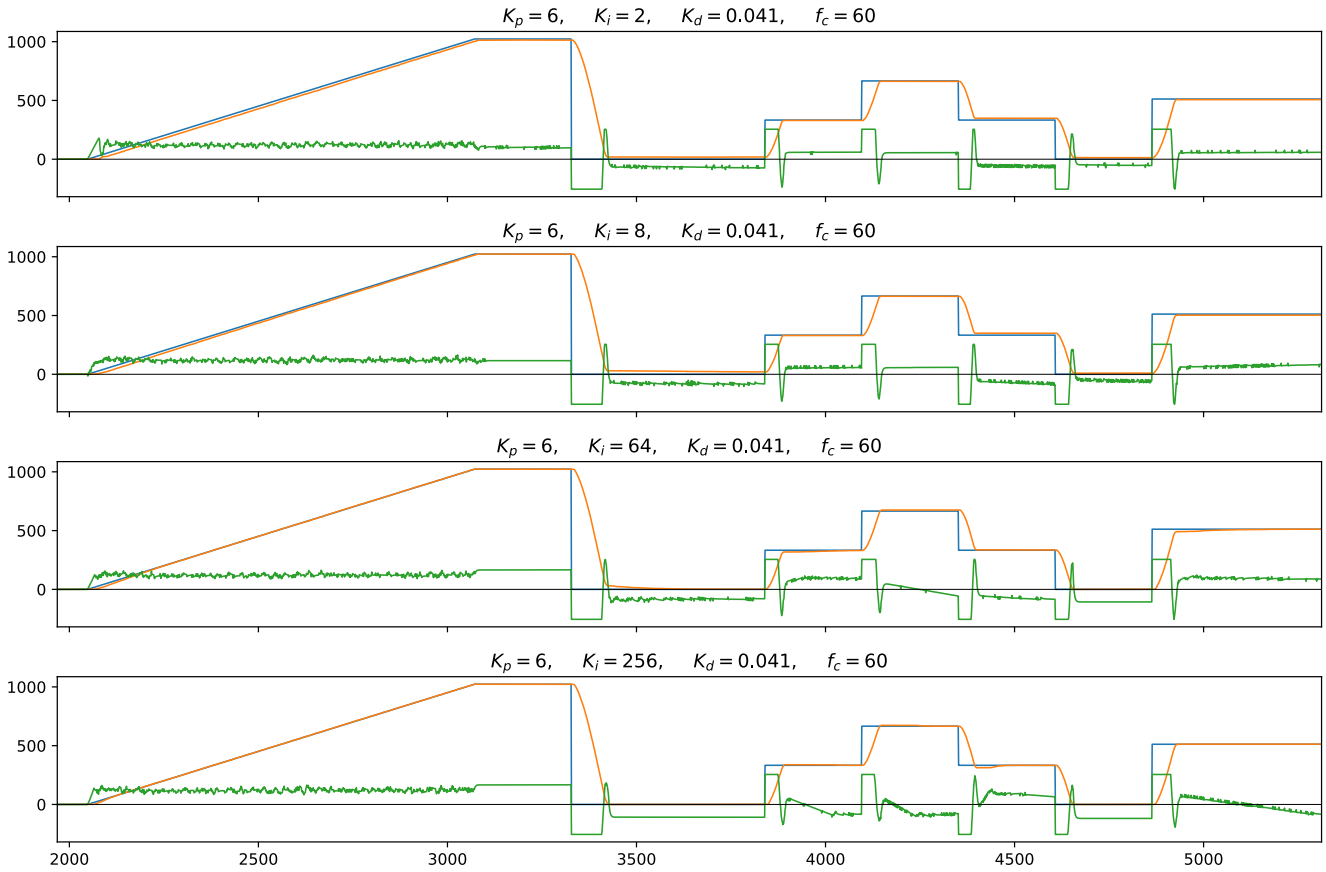
All parameters influence each other, so manual tuning is an iterative process. Now that we have a good baseline controller, we can iteratively fine-tune the parameters until we're satisfied with the results.



Derivative fine-tuning



Integral fine-tuning

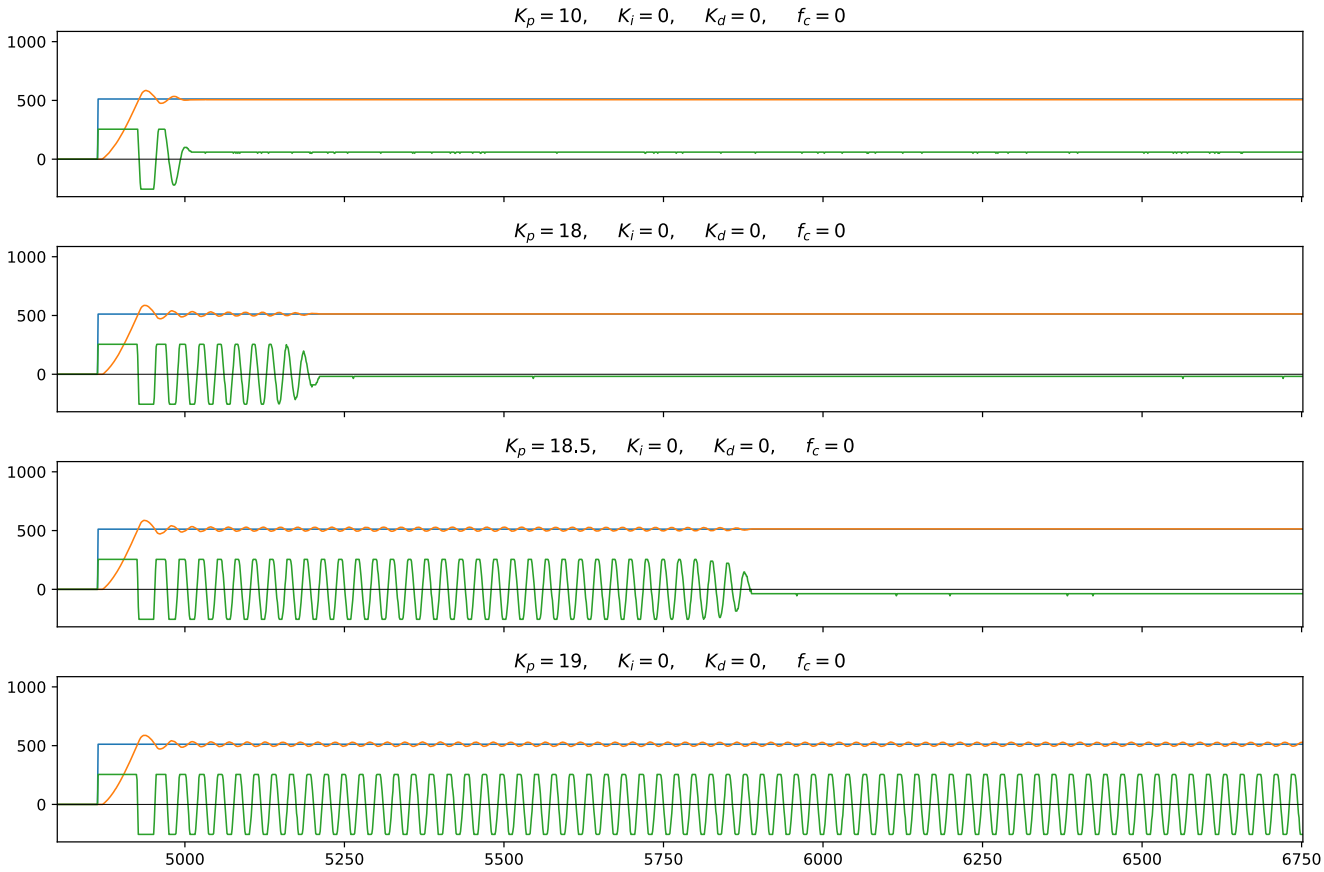


Ziegler—Nichols

Manual tuning requires trial-and-error and some knowledge of how the system is affected by each parameter. An alternative is to use a heuristic formula to choose the gains, such as the Ziegler—Nichols method.

Ultimate gain

First, the integral and derivative gains are set to zero and the proportional gain is increased until the system starts oscillating with a constant amplitude, recording the value of K_p and the period of the oscillations when this happens. This gain is called the "ultimate gain" K_u, and the period of the oscillation period is denoted by T_u.



In this experiment, the ultimate gain was determined to be $K_u \approx 19$, and the period of oscillation was around 29 cycles, or $T_u \approx 29\,T_s$, with $T_s = 960\mu s$.

PID gains

These values of K_u and T_u are then plugged into the heuristic formula

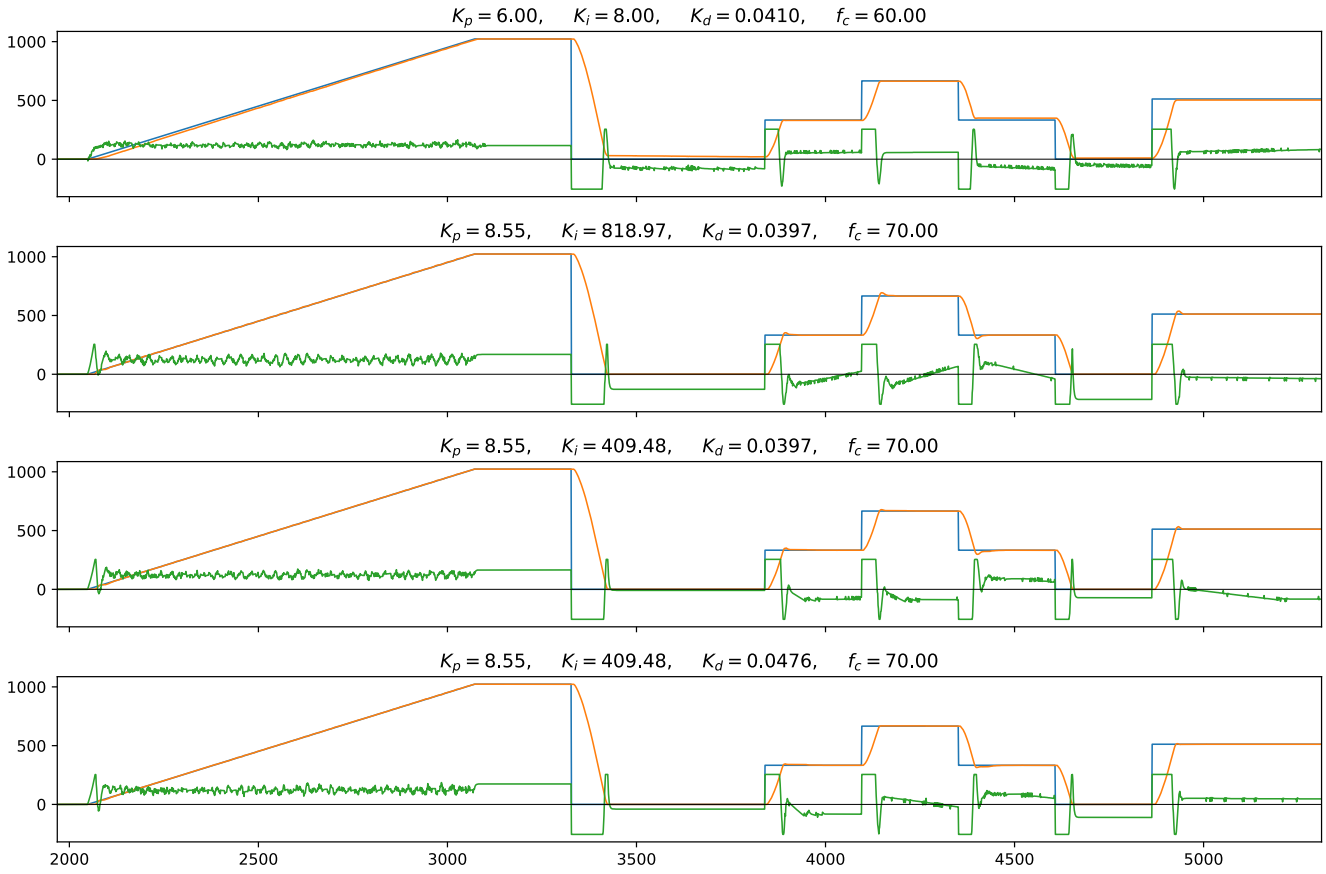
$$K_p = 0.6\,K_u, \qquad K_i = 1.2\,K_u/T_u, \qquad K_d = 0.075\,K_u T_u.$$

The results for these parameters are shown in the second graph in the figure below. The top graph shows the best manually tuned controller.

The resulting controller was relatively aggressive and had quite a bit of overshoot, so K_i was halved (third graph) and K_d was increased slightly (fourth graph),

$$K_p = 0.6\,K_u, \qquad K_i = 0.6\,K_u/T_u, \qquad K_d = 0.09\,K_u T_u.$$

The Wikipedia page linked to above also contains some alternative rules that promise less or no overshoot, it might be worth trying these out as well.


$K_p = 6.00,$ $K_i = 8.00,$ $K_d = 0.0410,$ $f_c = 60.00$
$K_p = 8.55,$ $K_i = 818.97,$ $K_d = 0.0397,$ $f_c = 70.00$
$K_p = 8.55,$ $K_i = 409.48,$ $K_d = 0.0397,$ $f_c = 70.00$
$K_p = 8.55,$ $K_i = 409.48,$ $K_d = 0.0476,$ $f_c = 70.00$
1000
500
0
2000
2500
3000
3500
4000
4500
5000