Simplified method for PID-like temperature control in digital environments

Marc Brevoort

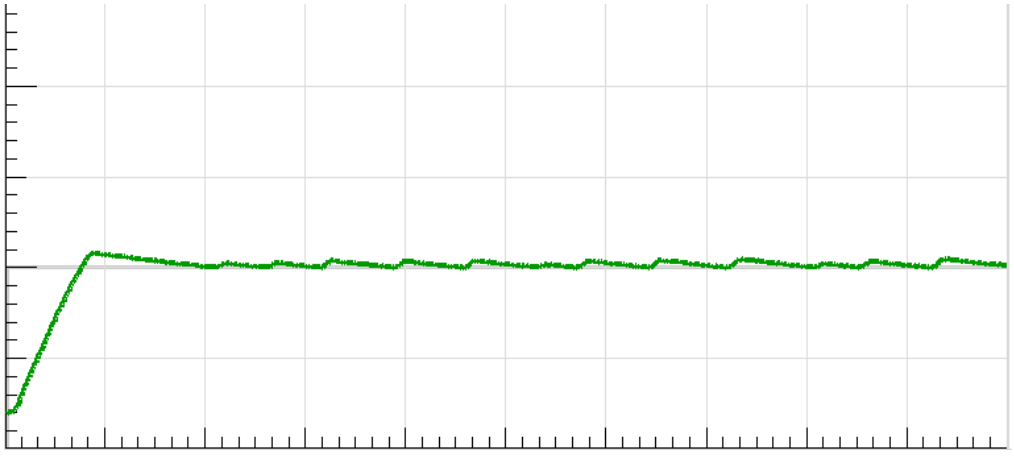
Abstract: Temperature control mechanisms based on classical PID control methods are based around analogue parameters, tend to require painstaking tuning and tend to target a single, static target temperature only. We propose a simplified method which reduces the effort required for tuning to a minimum, while also allowing use in environments where the target temperature varies over time, such as in coffee roasting appliances.

1. What's wrong with the naive approach?

The naive approach to temperature control would be to look at the current temperature and compare it against the desired current temperature. If the current temperature is higher than desired, we turn off the power:

```
def calculate_new_power_naive(self, curr_temp, desired_current_temp):
    if curr_temp > desired_current_temp:
        return 0.0
    return 1.0
```

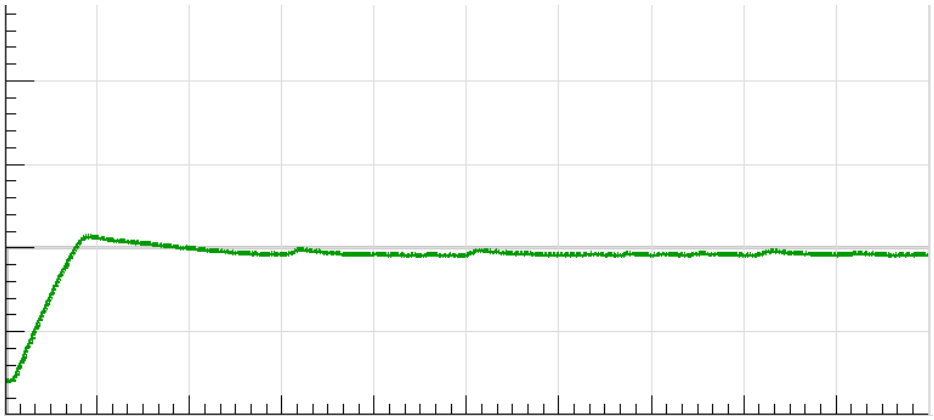
In practice, this approach will lead to overshoot, as there is a delay between applying the power and the desired temperature being reached. This also means that when we stop applying the power, the temperature will keep rising for some time to follow. The result is overshoot and oscillating around the target temperature, with a tendency to hover above the desired target temperature.



An initial PID implementation based on just the error aims to adjust the power level based on the error.

```
def calculate_new_power_pid(self, curr_temp, want_temp_now):
    error = want_temp_now - curr_temp
```

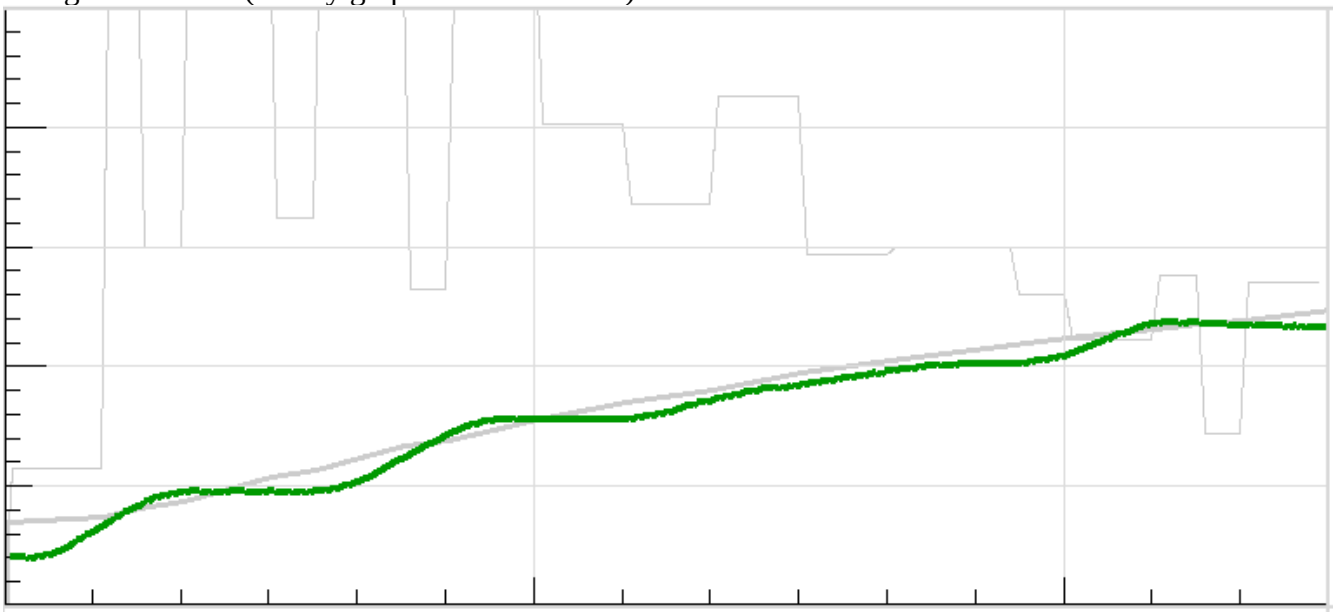
```
kp = 7
if curr_temp > want_temp_now:
    p = 0
else:
    p = error / 100
return p * kp
```



This allows shifting the result around the target temperature, but will require tuning. Observe also that oscillation around the target temperature still occurs. In addition, in digital environments, power levels other than “off” or “on” will require either hardware to permit analogue control, or some sort of dithering algorithm.

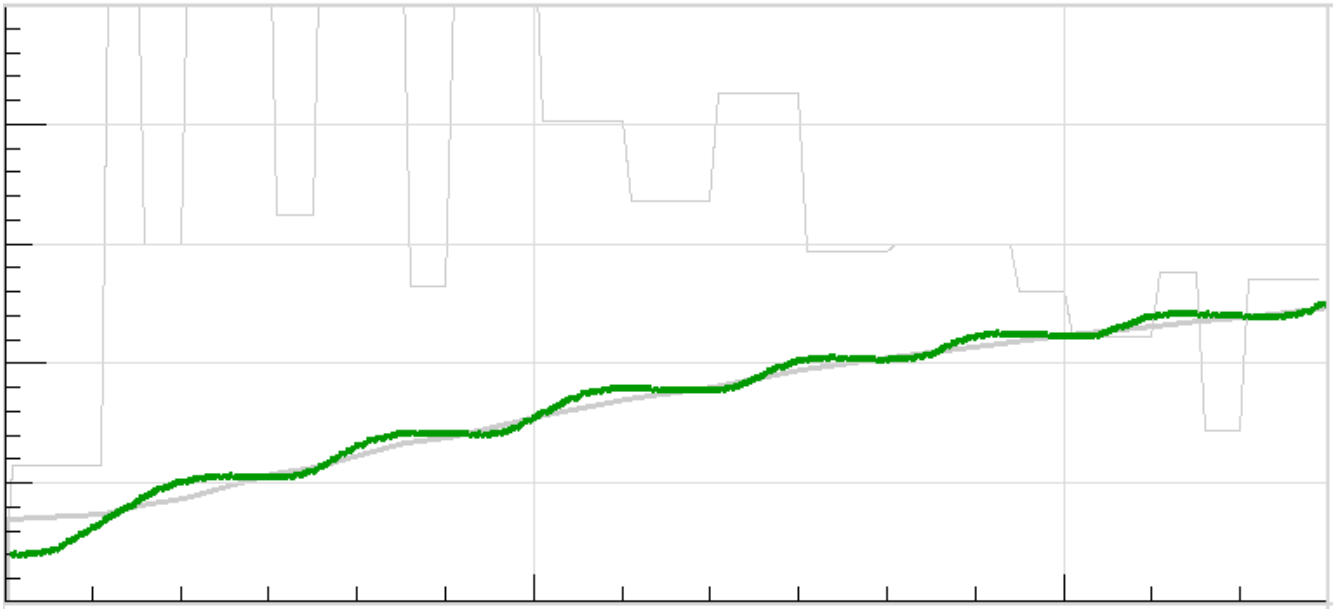
This brings us to an insight on digital temperature control; when briefly turning on the power of a heating element, it isn’t instantly “hot” or “cold”. A short “on” burst will warm up the heating element slightly, over time, and the laws of thermodynamics dictate that this heat will eventually be dissipated throughout the system in which the heating element operates.

Note also that the above performs particularly poorly in an environment where the target temperature changes over time (blocky graph is “rate of rise“):



```
Average absolute error: 4.741490271741292
Average error: 4.741490271741292
```

...and may in fact be outperformed by the “naive” implementation:



```
Average absolute error: 1.0595421893455086
Average error: 1.0595421893455086
```

Of course, we shouldn’t judge PID algos if we haven’t even implemented a complete PID implementation. When we do, note that we have several tunable parameters in the resulting code: **kp**, **ki** and **kd**, representing the parameters for **P**, **I** and **D** that PID algorithms are called after. While the following is a complete implementation, it was not (fully) tuned.

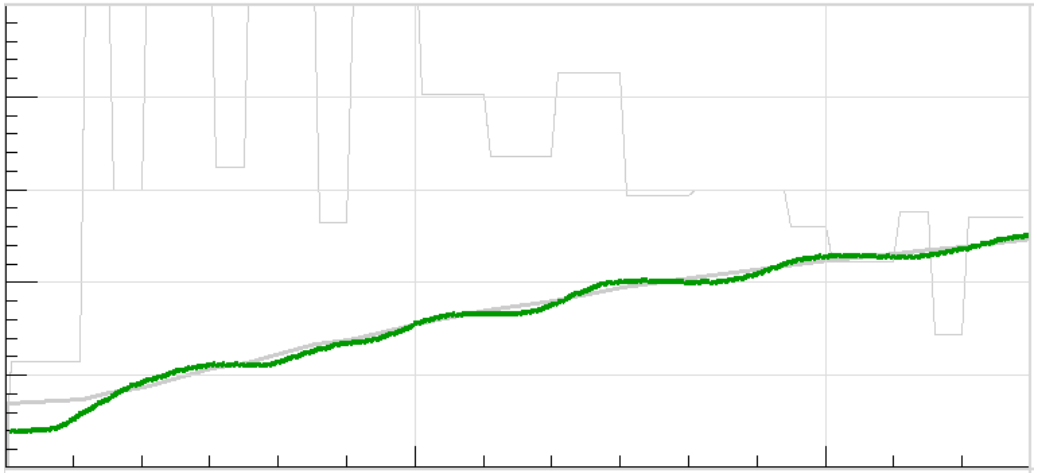
Having said that, tuning against our simulation would be no guarantee for accuracy in a physical device; as such, we may observe PID controllers “in the wild” that either perform poorly or leave tuning the parameters to the end user. A difference of 3 degrees C between target temperature and actual temperature is not unheard of.

I, the author, reserve the right not to cite any sources for this or any other claims made here; This is not a scientific paper, it merely explains how I’ve implemented my own temperature controller and share it here to use as you please, in the hope that it may be useful to you.

A more complete PID implementation will improve on this. Here is a full PID implementation taking in account error, integral, derivative and proportion.

```
    def calculate_new_power_pid4(self, curr_temp, want_temp_now):
        i=20
        curr_time = self.roast_time_in_seconds()
        (actual_temp_hist, power_hist, wanted_temp_hist) = self.data.history[-i]
        want_temp_future = self.temperature_offset() + target_temp(
            self.data.profile,
            (curr_time / self.data.time_stretch_factor)
            + i/self.data.history_samples_per_second
        )
        want_temp_spline = (wanted_temp_hist + want_temp_future) /2

        error = want_temp_spline - curr_temp

        kp = 20
```

```
        if curr_temp > want_temp_spline:
            p = 0
        else:
            p = error / 100

        histlen = self.data.history_len_in_seconds *
self.data.history_samples_per_second
        # histlen = 40
        ki = (1/histlen) /100
        total_error = 0
        for i in range(histlen):
            (actual_temp_hist, power_hist, wanted_temp_hist) = self.data.history[-
i]
            hist_error = wanted_temp_hist - actual_temp_hist
            total_error += hist_error

        (actual_temp_hist, power_hist, wanted_temp_hist) = self.data.history[-1]
        prev_error = wanted_temp_hist - actual_temp_hist
        d = (error - prev_error)
        kd = 2
        return p * kp + total_error * ki + d * kd
```



```
Average absolute error: 2.346105825277776
Average error: 1.6582687802777762
```

While this can undoubtedly be tuned to perform better, this is a cumbersome process, especially when we have to tune against a physical device rather than a simulation.

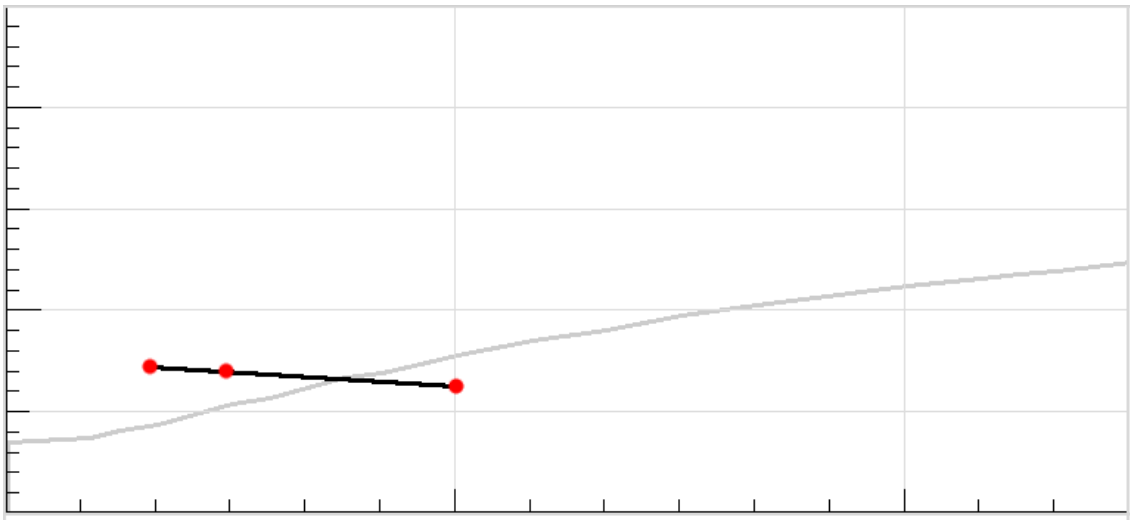
The proposed algorithm takes a slightly different approach.

The algorithm, briefly explained

1. We measure the historic average overshoot. This is our error. We will compensate for this while extrapolating.
2. We keep track of historic temperature (a “short” time ago, for example 1-2 seconds) and a desired future temperature (“long” into the future, for example 10-20 seconds). Our “current” temperature will act as a fulcrum between recent historic temperature and the desired temperature far in the future. It is important that the portion on the left of our “fulcrum” be smaller than that on the right.
3. We draw a line from “historic temperature” through “current temperature” to “x seconds into the future” to extrapolate what our future temperature will be. By having the left of the fulcrum be smaller than the right, we amplify our error by (future length/history length).
4. If our extrapolated temperature at point X in the future is higher or equals to the desired temperature at X seconds in the future, we switch off the power. Otherwise we switch on the power.

What we achieve with the above is that in essence our temperature follows a spline, all but eliminating overshoot and undershoot. The decision at point (4) will result in “emergent” dithering of the output, so we no longer need a dithering algorithm nor PCM hardware. Since the power is only switched on or off, parameters for proportion/integral/derivative are all but rendered irrelevant. Furthermore, this method works well with target temperatures that change over time and will mostly self-tune.

An example is in order to visualise how this works.



In the image above, historic temperature measured (point on the left) was too high.
Current temperature measured is *still* too high, but not by as much. Current temperature acts as a “fulcrum” for extrapolating at where the temperature will be in the future (also taking in consideration the historic error which we seek to eliminate over time – if the average historic error shows that we *structurally* aim too high, we will adjust our aim to minimise the error. At present, we see that based on the desired target temperature in the future, we are likely to undershoot at that point.
Decision: despite our current temperature being too high, **power is switched on** as in the future we will need a higher temperature than were we’ll end up if the current trend continues.

The implementation

```
    def calculate_new_power_naive2(self, curr_temp, want_temp_now):
        curr_time = self.roast_time_in_seconds()
        # calculate average error (called "overshoot" to indicate its polarity)
        history_in_seconds = 1
        rangelen = history_in_seconds * self.data.history_samples_per_second
        overshoot = 0
        for i in range(1, rangelen+1):
            (actual_temp_hist, power_hist, wanted_temp_hist) = self.data.history[-
i]
            overshoot += actual_temp_hist - wanted_temp_hist
        avg_overshoot = (overshoot / rangelen) - self.temperature_offset()

        # Then look at the temperature rise of the past second or so.
        (actual_temp_hist, power_hist, wanted_temp_hist) = self.data.history[-
rangelen]
        short_delta = curr_temp - actual_temp_hist

        # and extrapolate that to the expected future temp, taking the above error
in account.
        # we are looking WAAAAAAAAY into the future to magnify the error and smooth
        # out the temperature curve.
        future_in_seconds = 10
        future = curr_time + future_in_seconds

        # ... also compensate for the "overshoot" (or undershoot). We don't want
it.
        # keeping in mind history delta is small, and future delta
        # must be much bigger to see any compensation
        #
        want_temp_future = (
            self.temperature_offset() +
            target_temp(self.data.profile, future)
            - avg_overshoot
        )

        # in turn this allows us to take corrective action before we deviate much
        # from the target.
        #
        # Look mom, no tuning!
        expected_future_temp = curr_temp + (short_delta * future_in_seconds)
        return 0 if expected_future_temp > want_temp_future else 1
```

Explanation of some identifiers in the code

rangelen – the total amount of samples kept.

temperature_offset() - For calibration, a function that returns the amount of degrees that readings need to be shifted to show the true temperature. Assuming the device requires no calibration to output the correct, true temperature, this function will return the value **0**.

target_temp – an array containing the desired target temperature for any given point in the future.
data.history – a log of previously collected data as tuples of

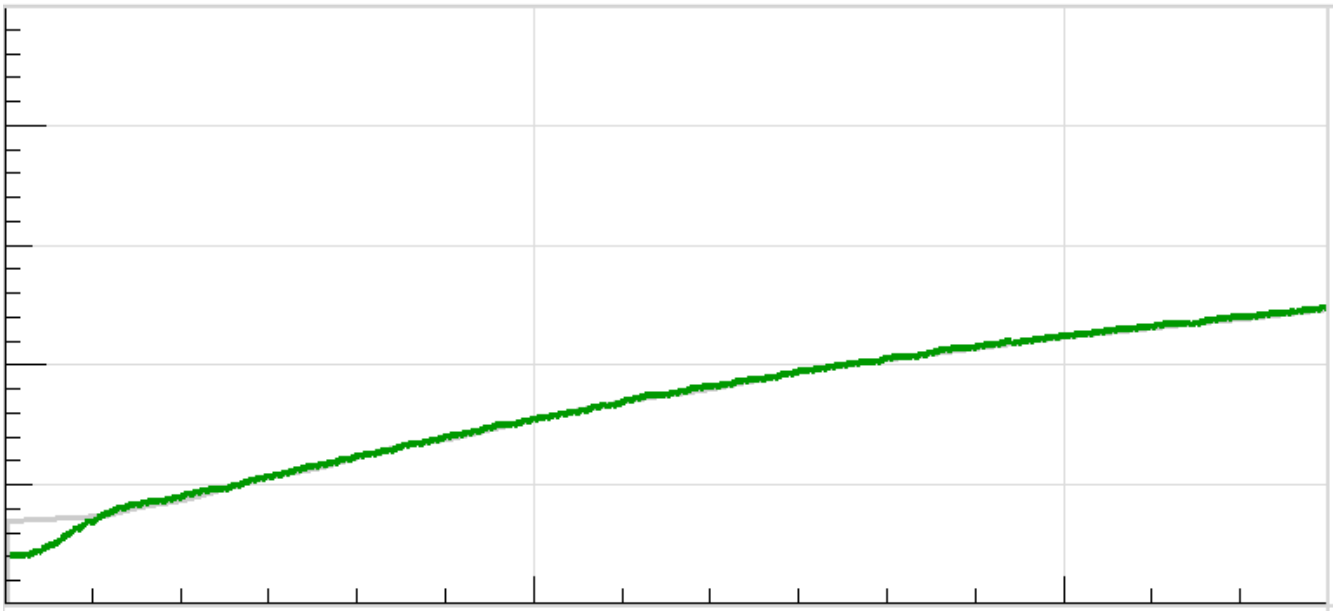
- measured historic temperature at a given time
- desired temperature at that point in time
- amount of power we were applying at that point in time

With the historic data, we can calculate average error over time, amount of power applied, and temperature delta between a given time in history and the current time.

It is assumed that the algorithm that calls our "**calculate_new_power**" function collects current and historic temperature data, and that the algorithm has access to the desired temperature profile through **target_temp**.

Performance

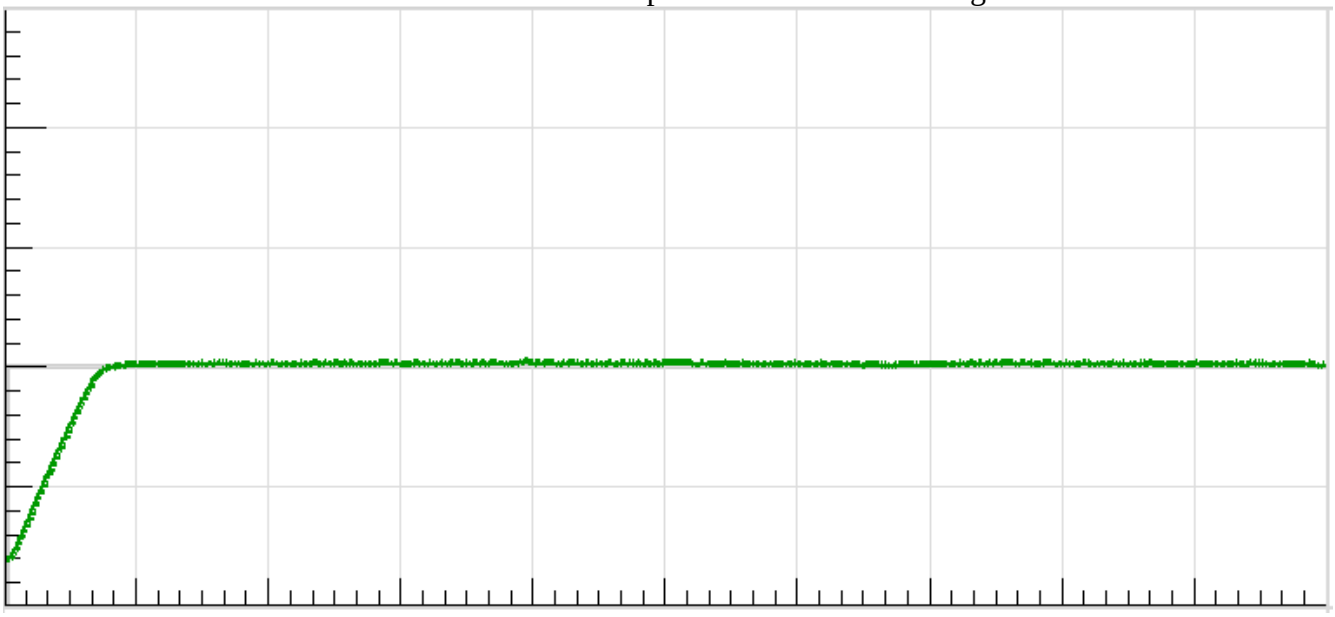
When matching a dynamic temperature profile



```
Average absolute error: 0.285721675596201
Average error: 0.006875055460771211
```

When matching a static temperature profile

Note the absence of overshoot. This is due to the spline-like nature of the algorithm.



```
Average absolute error: 1.0606342186819375
Average error: -1.0606342186819375
```

Safety considerations

The above performs well, however we can further tweak this by forcing a power off if we are currently already more than a certain amount above the maximum allowed target temperature.

```
    def calculate_new_power_naive2(self, curr_temp, want_temp_now):
        if abs(curr_temp - want_temp_now) > 4:
            if curr_temp > want_temp_now:
                # More than 4 degrees above target
                # Assume the device is on fire, power off
                return 0.0
        . . .
```

Likewise, CE regulations may require that at any given time, the rolling average of power consumed by the device never exceeds a certain maximum. We can guarantee this by looking at historic power use and never switching on the power if the maximum amount of power allowed for that time period has already been consumed.

Conclusion

With that, we have arrived at a temperature control solution which:

- Accurately controls temperature with very small error;
- Can be used with either static temperatures or changing temperature over time;
- Requires little to no tuning, making it port well between simulated and physical environments;
- All but eliminates overshoot and undershoot;
- Works through strictly binary (on/off) switching;
- Uses “emerging” dithering, without the need for further dithering algorithms
- Is more easily understood compared to many other temperature control solutions

References

https://en.wikipedia.org/wiki/Proportional%E2%80%93integral%E2%80%93derivative_controller