LED Tricks: Gamma Correction

Created by Phillip Burgess

https://learn.adafruit.com/led-tricks-gamma-correction

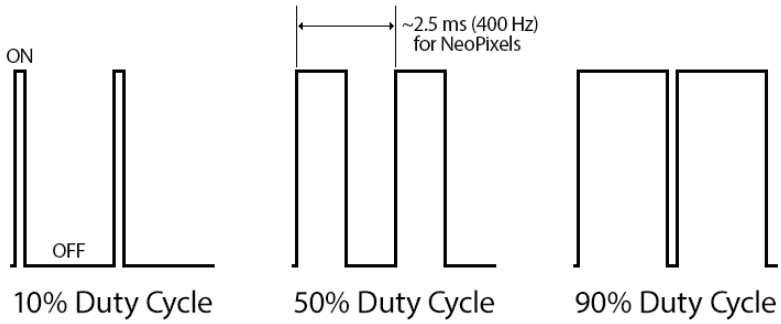
Last updated on 2021-11-15 06:17:29 PM EST

Table of Contents

The Issue

You're trying to program some cool LED effect but keep getting these weird not-quite-right colors. Maybe you're trying to mix orange (say 100% red, 50% green) but the LEDs show yellow instead. What gives?

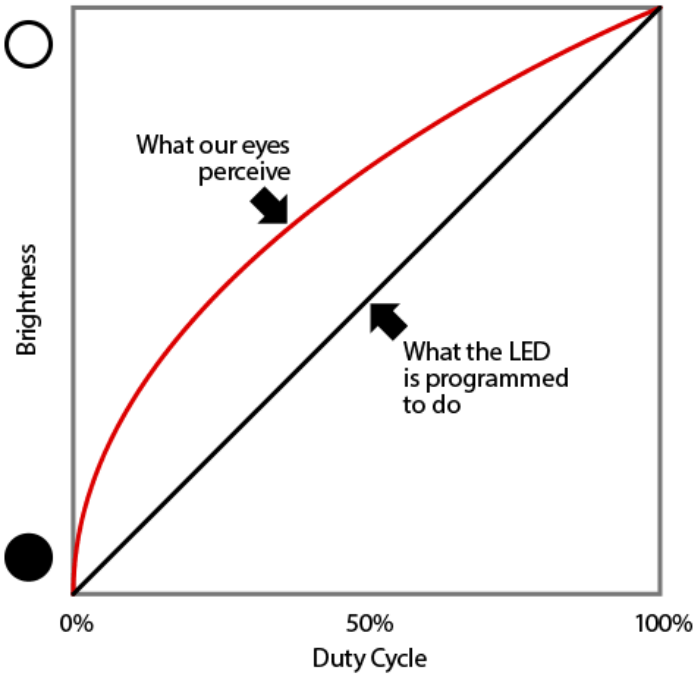
An LED driver IC — the "smarts" inside NeoPixels and other addressible LEDs — use p ulse-width modulation, switching the LED on and off very quickly (about 400 times per second in the case of NeoPixels), much faster than our eyes can perceive; we just see a uniform brightness. The "on" vs. "off" time determines the intensity.



When you program in a "halfway" level (like 127 out of the maximum 255), you are indeed getting something very close to a proper 50% duty cycle. The LEDs are doing the correct thing.

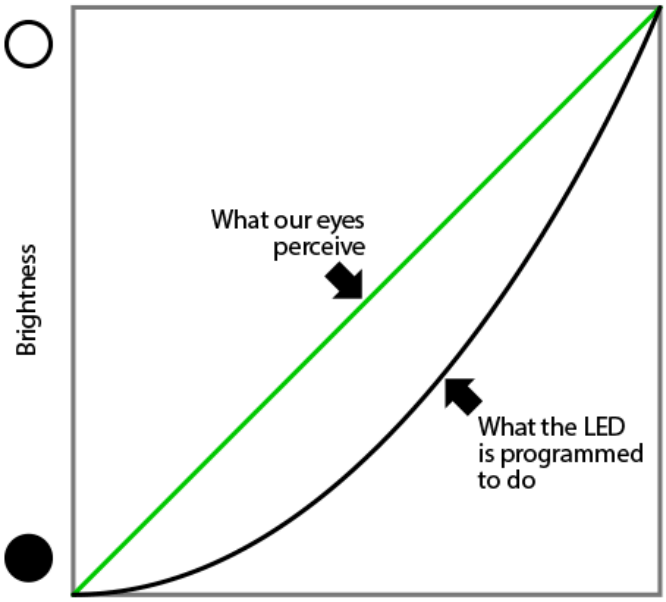
Why yellow then, instead of orange? It's nothing to do with the LEDs or your code, it's how our eyes work...

Eyes evolved to find food by daylight and evade predators by starlight. That's a huge dynamic range. A sort of non-linearity is built in so details can be seen at both extremes. It's extraordinarily sensitive at the low end...we perceive changes there as more pronounced than objective measurement (or LED duty cycle numbers) would suggest:



Going linearly by numbers, there's huge discontinuities at the left end of this gray ramp, while the right squares are nearly indistinguishable:

The trick then is to apply an inverse function — gamma correction — to compensate for this non-linearity in our perception:



Now each step appears more even:

To get something that appears 50% bright, we request a much dimmer value from the LED...instead of 127, it might be only 36 or so. The two extremes, 0 and 255, remain unchanged.

Your monitor, computer operating system and applications typically already have this correction built in. So when you pick orange in Photoshop, the R/G/B values shown are 255/127/0, as you'd intuitively expect. We can do something similar for LEDs...

The Quick Fix

For like 95% of most cases, copy the following table into your Arduino sketch. You'll see this same table a lot in our NeoPixel projects:

```
const uint8_t PROGMEM gamma8[] = {
    0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
    0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  1,  1,  1,  1,
    1,  1,  1,  1,  1,  1,  1,  1,  1,  2,  2,  2,  2,  2,  2,  2,
    2,  3,  3,  3,  3,  3,  3,  3,  4,  4,  4,  4,  4,  5,  5,  5,
    5,  6,  6,  6,  6,  7,  7,  7,  7,  8,  8,  8,  9,  9,  9, 10,
   10, 10, 11, 11, 11, 12, 12, 13, 13, 13, 14, 14, 15, 15, 16, 16,
   17, 17, 18, 18, 19, 19, 20, 20, 21, 21, 22, 22, 23, 24, 24, 25,
   25, 26, 27, 27, 28, 29, 29, 30, 31, 32, 32, 33, 34, 35, 35, 36,
   37, 38, 39, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 50,
   51, 52, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 66, 67, 68,
   69, 70, 72, 73, 74, 75, 77, 78, 79, 81, 82, 83, 85, 86, 87, 89,
   90, 92, 93, 95, 96, 98, 99,101,102,104,105,107,109,110,112,114,
  115,117,119,120,122,124,126,127,129,131,133,135,137,138,140,142,
  144,146,148,150,152,154,156,158,160,162,164,167,169,171,173,175,
  177,180,182,184,186,189,191,193,196,198,200,203,205,208,210,213,
  215,218,220,223,225,228,231,233,236,239,241,244,247,249,252,255 };
```

This table remaps linear input values (the numbers we'd like to use; e.g. 127 = half brightness) to nonlinear gamma-corrected output values (numbers producing the desired effect on the LED; e.g. 36 = half brightness).

Because the table is in program memory (PROGMEM in the declaration), it can't be accessed directly...elements must be read using the pgm_read_byte() function, like this:

```
result = pgm_read_byte(&amp;gamma8[input]);
```

Or, in the context of setting colors on an LED strip, it might resemble:

```
strip.setPixelColor(pixelNumber,
  pgm_read_byte(&amp;gamma8[red]),
  pgm_read_byte(&amp;gamma8[green]),
  pgm_read_byte(&amp;gamma8[blue]));
```

That might get tedious after a while...you can write a wrapper function around setPixelColor() to make it easier, do all your gamma table lookups in that single place.

PROGMEM is a fantastic RAM-saver for Arduino sketches...if you're not familiar, it's explained on the Arduino web site (https://adafru.it/aMw), and our Memories of an Arduino (https://adafru.it/q0e) guide also offers some insights.

Optional: you can move the gamma table to the bottom of your code (maybe you don't want to look at it as the first thing every time you open a sketch) by adding this line near the top:

```
extern const uint8_t gamma8[];
```

The table isn't really extern (this normally means some variable or data is located in another source file), but this lets us push it to the bottom while referring to it earlier in our code.

Gamma correction normally would use floating-point math, not something the Arduino excels at. This table lookup only takes about a microsecond, and despite its apparent size it's really much smaller than invoking the equivalent floating-point math function (256 bytes vs. ~2KB of flash space).

This doesn't give us ultimate control, but it's adequate for the vast majority of cases. Orange will now look orange!

Digging Deeper

You might want a slightly different gamma table. Here's the code that generated our sample. This is not an Arduino sketch...it's written in Processing (free download (https://adafru.it/aPt))...chosen because it installs easily on Windows, Mac or Linux. Run this program, then copy-and-paste the output into an Arduino sketch, replacing the gamma[] table.

```
// Generate an LED gamma-correction table for Arduino sketches.
// Written in Processing (www.processing.org), NOT for Arduino!
// Copy-and-paste the program's output into an Arduino sketch.

float gamma   = 2.8; // Correction factor
int   max_in  = 255, // Top end of INPUT range
```

```
      max_out = 255; // Top end of OUTPUT range

void setup() {
  print("const uint8_t PROGMEM gamma[] = {");
  for(int i=0; i&lt;=max_in; i++) {
    if(i &gt; 0) print(',');
    if((i &amp; 15) == 0) print("\n  ");
    System.out.format("%3d",
      (int)(pow((float)i / (float)max_in, gamma) * max_out + 0.5));
  }
  println(" };");
  exit();
}
```

This first line sets the exponent for the correction curve:

```
float gamma   = 2.8; // Correction factor
```

Higher values here will result in dimmer midrange colors, lower values will be brighter. 1.0 = no correction. The default of 2.8 isn't super-scientific, just tested a few numbers and this seemed to produce a sufficiently uniform brightness ramp along an LED strip; maybe you'll refine this further.

max_in and max_out set the input and output ranges of the table. The defaults here are for the NeoPixel brightness range of 0–255...if you're working with LPD8806 strips (which have a 7-bit brightness), max_out can be changed to 127 (might want to leave max_in at 255, since a lot of existing code assumes 8-bit colors).

If you're really persnickety, you can make separate tables for red, green and blue to achieve a more neutral white balance, adjusting max_out for each.

Perks and Caveats

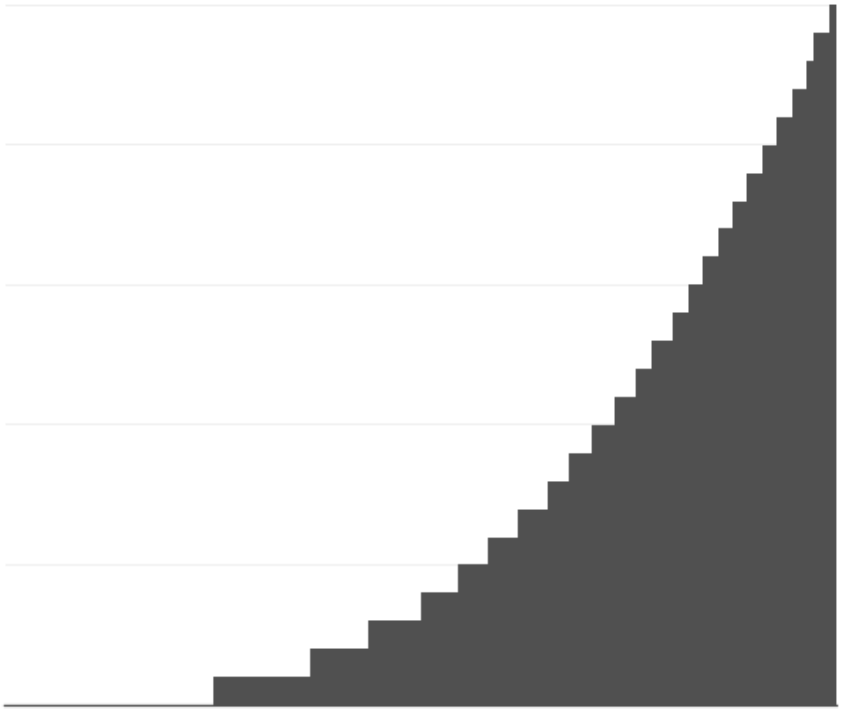
Aside from aesthetics, an unexpected benefit of gamma correction is that battery-operated projects tend to run longer, because of the lower intermediate brightnesses.

On the downside, look at the first few lines of the gamma-correction table:

```
const uint8_t PROGMEM gamma[] = {
    0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
    0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  1,  1,  1,  1,
    1,  1,  1,  1,  1,  1,  1,  1,  1,  2,  2,  2,  2,  2,  2,  2,
    2,  3,  3,  3,  3,  3,  3,  3,  4,  4,  4,  4,  4,  5,  5,  5,
...
```

Notice the first 28 elements are all 0, the next 12 are 1, next 7 are 2, and so forth. Input values from 1–27 all result in an "off" LED. This is the unfortunate reality of quan

tization. The LED driver only handles 256 distinct PWM settings, period. When we move values up or down, they still must fall in one of those same 256 fixed bins...we don't get new ones...the result being much fewer distinct output values (163 in this case). It's most pronounced at the low end, progressively less toward the top.



"Luxury" LED drivers such as the PCA9685 (https://adafru.it/dUG) and TLC5947 (http://adafru.it/1429) use 12-bit PWM (4096 output levels) to minimize the effects of quantization. More advanced drivers like FadeCandy use dithering to 'fake' a wider dynamic range