Cubehelix Explained

Sander Melnikov hey@sandydoo.me

Contents

Introduction

Cubehelix is a method of generating palettes of colours with a very important property — a monotonically, or constantly increasing perceived brightness. The main use-case for such palettes is in data visualisation, where it's common to represent ranges of numbers as ordered ranges of colours. But creating colourful palettes — not just greyscale — that actually look ordered turns out to be more difficult than simply increasing the overall lightness. I'll let Dave A. Green, the original author of the cubehelix scheme, explain the issue in detail:

> Images in astronomy often, but not always, represent the intensity of some source. However, the colour schemes used to display images are not perceived as increasing monotonically in brightness, which does not aid the interpretation of the images. The perceived brightness of red, green and blue are not the same, with green being seen as the brightest, then red, then blue. For example a bright yellow (i.e. full intensity red and green) is perceived as being very much brighter than a bright blue. So if a colour scheme has yellow for intermediate intensities, but blue or red for higher intensities, then the blue or red is perceived at lower brightness. This can be also seen when such colour images are printed in black and white, when increasing intensity in the image does not correspond to a greyscale with monotonically increasing brightness.

Unlike other methods of generating perceptually uniform colours, cubehelix isn't

a special, unique colour space. It's based on RGB: red, green, and blue colour components. To generate a palette, we plot a *helix*, a spiral, along and around the diagonal of the RGB *cube* (it's namesake). That diagonal is our overall lightness — a colourless scale of shades of grey, from black to white. The overall lightness increases, as we sample colours, from darkest to lightest, along our constructed spiral. And as we rotate in the RGB cube along the spiral, we rotate in a plane that's been adjusted for the perceptual brightness of the individual RGB components. So our spiral isn't symmetrical. It bows and rises depending on the hue.

There are a few things we can control about the palette. We can set the saturation of the colours by increasing the amplitude of the helix — its deviation from the diagonal of the cube, set the range of the palette by specifying how many times to rotate around the diagonal, and set the first colour of the palette with the initial directions of the helix.

Do we need the helix?

All right, so we can generate palettes of colours with monotonically increasing brightness. Our data visualisations now not only look good, but also accurately convey information. But can we do better? I mean, the helix is a fun concept and it's convenient for generating rainbow unicorn palettes, but it lacks precise control over the palette. Want a palette between two specific shades of blue and red? Well, good luck fiddling around with the parameters of the spiral until you, hopefully, get something "close enough".

Wouldn't it be nice to be able to interpolate between two specific colours, while maintaining the same perceived intensity? We already know how to adjust our red, green, and blue colour components to account for our perception of each component's intensity. But the original algorithm only allows us to convert locations on a helix to RGB components. Can we somehow convert RGB colours back into this perceptually uniform colour space and make use of it's beneficial properties?

Towards a cubehelix colour space

The first people to have had this idea and provide an implementation were Jason Davies and Mike Bostock, as part the d3 visualisation library back in 2015. While the code is open-source, it can be difficult to follow for the uninitiated, and — as far as I know — there is no write up of the mathematics employed in the solution.

```
function cubehelixConvert(o) {
  if (o instanceof Cubehelix) return new Cubehelix(o.h, o.s, o.l, o.opacity);
  if (!(o instanceof Rgb)) o = rgbConvert(o);
  var r = o.r / 255,
      g = o.g / 255,
```

```
      b = o.b / 255,
      l = (BC_DA * b + ED * r - EB * g) / (BC_DA + ED - EB),
      bl = b - l,
      k = (E * (g - l) - C * bl) / D,
      s = Math.sqrt(k * k + bl * bl) / (E * l * (1 - l)), // NaN if l=0 or l=1
      h = s ? Math.atan2(k, bl) * degrees - 120 : NaN;
  return new Cubehelix(h < 0 ? h + 360 : h, s, l, o.opacity);
}
```

To really understand what's going on, we're going to derive the solution from scratch. We'll need a tiny bit of linear algebra, and the rest will be some basic geometry and algebra.

Lightness

- Lightness — overall brightness of the colours.
- The RGB cube, the diagonal, colourless, overall brightness.
- R, G, and B are three orthogonal vectors. The normal vector to the three is the diagonal.

Cross product of two vectors

$$\begin{aligned}\vec{X} &= Ar + Cg + Eb \\ \vec{Y} &= Br + Dg + Fb \\ l &= \vec{X} \times \vec{Y} \\ l &= \frac{(CF - DE)r + (EB - AF)g + (AD - BC)b}{CF - DE + EB - AF + AD - BC}\end{aligned}$$

Since, in this case, $F = 0$, we can simplify things further.

$$l = \frac{(AD - BC)b - DEr + EBg}{AD - BC - DE + EB}$$

You might have noticed that the code looks a bit different. Bostock and Davies are computing $\vec{Y} \times \vec{X}$ instead. Why? I'm not sure. But the cross product is anti-commutative, meaning that changing the order of the two vectors in the cross product doesn't change the result, apart from changing the sign. And since we're normalising the whole thing, the final lightness will always be positive. So, either way is fine.

Hue and Saturation

Here's where things seem a bit confusing at first. At first glance, `s` and `h` probably stand for "saturation" and "hue". But what are `bl` and `k`? How do they relate to saturation and hue?

We've got several clues. The saturation is computed from the square root of the sum of squares of `bl` and `k`. Hold on, that's the Pythagorean theorem! And the hue — that's the angle from the positive x axis. So `bl` and `k` are the x and y values in a Euclidean plane, respectively. What is this plane though?

Let's recall the original RGB transformation.

$$\begin{aligned} r &= l + \alpha\left(A\cos(h) + B\sin(h)\right) \\ g &= l + \alpha\left(C\cos(h) + D\sin(h)\right) \\ b &= l + \alpha\left(E\cos(h)\right) \end{aligned}$$

where $\alpha = s \cdot l \cdot (1 - l)$.

Remember what the definitions of $\cos(h)$ and $\sin(h)$ are? Our adjacent and opposite sides are x and y, respectively, and the hypotenuse is the saturation s.

$$\begin{aligned} \cos(h) &= \frac{x}{s} \\ \sin(h) &= \frac{y}{s} \end{aligned}$$

Let's plug these values in,

$$\begin{aligned} r &= l + (\cancel{s} \cdot l \cdot (1 - l)) \cdot \left(A\frac{x}{\cancel{s}} + B\frac{y}{\cancel{s}}\right) \\ g &= l + (\cancel{s} \cdot l \cdot (1 - l)) \cdot \left(C\frac{x}{\cancel{s}} + D\frac{y}{\cancel{s}}\right) \\ b &= l + (\cancel{s} \cdot l \cdot (1 - l)) \cdot \left(E\frac{x}{\cancel{s}}\right) \end{aligned}$$

We get quite lucky here. Not only do all the s cancel out, meaning we have one less unknown in our set of equations, but, since $F = 0$, we can immediately rearrange equation () to get x.

(1)
$$x = \frac{b - l}{E\tilde{\alpha}}$$

where $\tilde{\alpha} = l \cdot (1 - l)$.

Now, for the y, we replace x with this definition in equation (**??**).

$$
\begin{aligned}
g &= l + \tilde{\alpha} \cdot \left(\frac{C}{E\tilde{\alpha}}(b-l) + Dy \right) \\
&= l + \frac{C}{E}(b-l) + \tilde{\alpha}Dy
\end{aligned}
$$

$$
\begin{aligned}
y &= \frac{g - l - \frac{C}{E}(b-l)}{\tilde{\alpha}D} \\
&= \frac{\frac{1}{E}(E(g-l) - C(b-l))}{\tilde{\alpha}D} \\
&= \frac{E(g-l) - C(b-l)}{E\tilde{\alpha}D}
\end{aligned}
$$

Fantastic! We've got our x and y coordinates. There's one more clever thing we can do, though. Do you see how in the equations for both x (**??**) and y () we're dividing by $E\tilde{\alpha}$? We can delay that division and work with scaled x and y values, as long as we remember to adjust for it later. Division is an expensive operation for computers to perform, after all.

That way we define $\hat{x}$ and $\hat{y}$ as:

$$
\begin{aligned}
\hat{x} &= E\tilde{\alpha}x = b - l \\
\hat{y} &= E\tilde{\alpha}y = \frac{E(g-l) - C(b-l)}{D}
\end{aligned}
$$

Now our definition for $\hat{x}$ matches `bl` and $\hat{y}$ matches `k`.

Saturation in our HSL space is the distance from $(0, 0)$ to (x, y). Using Pythagoras's theorem,

$$
\begin{aligned}
s &= \sqrt{x^2 + y^2} \\
&= \sqrt{\left(\frac{\hat{x}}{E\tilde{\alpha}}\right)^2 + \left(\frac{\hat{y}}{E\tilde{\alpha}}\right)^2} \\
&= \frac{\sqrt{\hat{x}^2 + \hat{y}^2}}{E\tilde{\alpha}}
\end{aligned}
$$

Lastly, we can compute the hue using the two-argument inverse tangent function, remembering to convert from radians to degrees,

$$
h = arctan2\left(\hat{y}, \hat{x}\right) \cdot \frac{180^\circ}{\pi}
$$

Spinning the hue

Once last thing! Remember how our x value (**??**) was calculated solely from the blue component of our colour? Well, that means that we've rotated our coordinate space. Typical hue values are set to 0° at red, 120° at green, and 240° at blue. At 0°, our hue is actually blue. So we've rotated everything by 120° counter-clockwise, adding 120° to our hue value.

Luckily, there's a simple fix! We'll just subtract 120° from our final hue, and then, when converting back to RGB, make sure to add it back.

$$h = arctan2\left(\hat{y}, \hat{x}\right) \cdot \frac{180}{\pi} - 120°$$

Should I use it?

At this point, there's nothing "cube" or "helix" about this colour space; it's a cylindrical HSL colour space that can be converted to "adjusted" RGB values. People have created many such "adjusted" colour spaces over the years, some focused on how humans perceive colours, others correcting for the peculiarities of various display technologies. Each has its own set of pros and cons. This colour space tries to adjust the RGB components to create a uniform, even perception of colour intensity — either always increasing, always decreasing, or staying the same across all hues. That's the pro. The con is that you might create impossible or unrepresentable colours: colours with a saturation or lightness outside of the range that these values can realistically take. In that case, the RGB colour components will be clipped — adjusted to the closest maximum value —, limiting the range of colours you can use while still maintaining perceptual uniformity.

A general solution

Our final colour space conversion took advantage of the fact that, for the cubehelix transformation, the constant F is equal to 0. That simplified things for us, but we can also derive a more general solution, for any F. Perhaps, someone may find this useful.

Let's start with our set of red, green, and blue colour components. We'll solve this system of equations for x and y using elimination.

$$\begin{aligned} r &= l + \tilde{\alpha}\left(Ax + By\right) \\ g &= l + \tilde{\alpha}\left(Cx + Dy\right) \\ b &= l + \tilde{\alpha}\left(Ex + Fy\right) \end{aligned}$$

$$\begin{aligned} Dr &= Dl + \tilde{\alpha}\left(ADx + BDy\right) \\ Bg &= Bl + \tilde{\alpha}\left(BCx + BDy\right) \end{aligned}$$

$$Dr - Bg = Dl - Bl + \tilde{\alpha}ADx - \tilde{\alpha}BCx$$

$$x = \frac{D\left(r - l\right) - B\left(g - l\right)}{\tilde{\alpha}\left(AD - BC\right)}$$

$$\begin{aligned} Eg &= El + \tilde{\alpha}\left(CEx + DEy\right) \\ Cb &= Cl + \tilde{\alpha}\left(CEx + CFy\right) \end{aligned}$$

$$Eg - Cb = El - Cl + \tilde{\alpha}DEy - \tilde{\alpha}CFy$$

$$y = \frac{E\left(g - l\right) - C\left(b - l\right)}{\tilde{\alpha}\left(DE - CF\right)}$$

Finally, we have,

$$\begin{aligned} x &= \frac{D\left(r - l\right) - B\left(g - l\right)}{\tilde{\alpha}\left(AD - BC\right)} \\ y &= \frac{E\left(g - l\right) - C\left(b - l\right)}{\tilde{\alpha}\left(DE - CF\right)} \\ h &= arctan2\left(y, x\right) \cdot \frac{180^{\circ}}{\pi} \\ s &= \sqrt{x^2 + y^2} \\ l &= \frac{\left(CF - DE\right)r + \left(EB - AF\right)g + \left(AD - BC\right)b}{CF - DE + EB - AF + AD - BC} \end{aligned}$$