colorspacious Documentation

Release 1.1.0

Nathaniel J. Smith, Richard Futrell

November 10, 2016

Contents

`Colorspacious` is a powerful, accurate, and easy-to-use Python library for performing colorspace conversions.

In addition to the most common standard colorspaces (sRGB, XYZ, xyY, CIELab, CIELCh), we also include: color vision deficiency ("color blindness") simulations using the approach of *[MOF09]*; a complete implementation of CIECAM02; and the perceptually uniform CAM02-UCS / CAM02-LCD / CAM02-SCD spaces proposed by *[LCL06]*.

Contents:

CHAPTER 1

Overview

`Colorspacious` is a powerful, accurate, and easy-to-use Python library for performing colorspace conversions.

Documentation: https://colorspacious.readthedocs.org

Installation: `pip install colorspacious`

Downloads: https://pypi.python.org/pypi/colorspacious/

Code and bug tracker: https://github.com/njsmith/colorspacious

Contact: Nathaniel J. Smith <njs@pobox.com>

Dependencies:

- Python 2.6+, or 3.3+
- NumPy

Developer dependencies (only needed for hacking on source):

- nose: needed to run tests

License: MIT, see LICENSE.txt for details.

Other Python packages with similar functionality that you might want to check out as well or instead:

- `colour`: http://colour-science.org/
- `colormath`: http://python-colormath.readthedocs.org/
- `ciecam02`: https://pypi.python.org/pypi/ciecam02/
- `ColorPy`: http://markkness.net/colorpy/ColorPy.html

CHAPTER 2

Tutorial

Colorspacious is a Python library that lets you easily convert between colorspaces like sRGB, XYZ, CIEL*a*b*, CIECAM02, CAM02-UCS, etc. If you have no idea what these are or what each is good for, and reading this list makes you feel like you're swimming in alphabet soup, then this video provides a basic orientation and some examples. (The overview of color theory starts at ~3:35.)

Now let's see some cut-and-pasteable examples of what `colorspacious` is good for. We'll start by loading up some utility modules for numerics and plotting that we'll use later:

```
In [1]: import numpy as np

In [2]: import matplotlib

In [3]: import matplotlib.pyplot as plt
```

Now we need to import `colorspacious`. The main function we'll use is *`cspace_convert()`*:

```
In [4]: from colorspacious import cspace_convert
```

This allows us to convert between many color spaces. For example, suppose we want to know how the color with coordinates (128, 128, 128) in sRGB space (represented with values between 0 and 255) maps to XYZ space (represented with values between 0 and 100):

```
In [5]: cspace_convert([128, 128, 128], "sRGB255", "XYZ100")
Out[5]: array([ 20.51692894,  21.58512253,  23.506738  ])
```

Colorspacious knows about a *wide variety of colorspaces*, and you can convert between any of them by naming them in a call to *`cspace_convert()`*.

We can also conveniently work on whole images. Let's load one up as an example.

```
# if you want this file, try:
#    hopper_sRGB = plt.imread(matplotlib.cbook.get_sample_data("grace_hopper.png"))
In [6]: hopper_sRGB = plt.imread("grace_hopper.png")
```

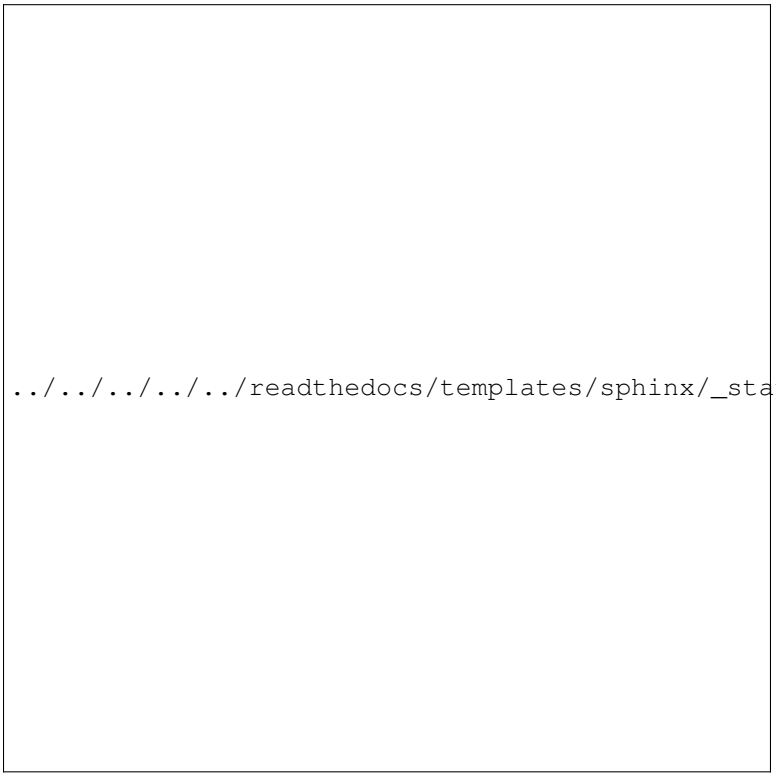
What have we got here?

```
In [7]: hopper_sRGB.shape
Out[7]: (600, 512, 3)

In [8]: hopper_sRGB[:2, :2, :]
Out[8]:
array([[[ 0.08235294,  0.09411765,  0.3019608 ],
        [ 0.10588235,  0.11764706,  0.33333334]],
```

```
       [[ 0.10196079,  0.11372549,  0.32156864],
        [ 0.09803922,  0.10980392,  0.32549021]]], dtype=float32)

In [9]: plt.imshow(hopper_sRGB)
Out[9]: <matplotlib.image.AxesImage at 0x7fa80fae0ed0>
```



It looks like this image has been loaded as a 3-dimensional NumPy array, where the last dimension contains the R, G, and B values (in that order).

We can pass such an array directly to *cspace_convert()*. For example, we can convert the whole image to XYZ space. This time we'll specify that our input space is "sRGB1" instead of "sRGB255", because the values appear to be encoded on a scale ranging from 0-1:

```
In [10]: hopper_XYZ = cspace_convert(hopper_sRGB, "sRGB1", "XYZ100")

In [11]: hopper_XYZ.shape
Out[11]: (600, 512, 3)

In [12]: hopper_XYZ[:2, :2, :]
Out[12]:
array([[[ 1.97537605,  1.34848558,  7.17731319],
        [ 2.55586737,  1.81738616,  8.81036579]],

       [[ 2.38827051,  1.70749148,  8.18630399],
        [ 2.37740322,  1.66167069,  8.37900306]]])
```

2.1 Perceptual transformations

RGB space is a useful way to store and transmit images, but because the RGB values are basically just a raw record of what voltages should be applied to some phosphors in a monitor, it's often difficult to predict how a given change in RGB values will affect what an image looks like to a person.

Suppose we want to desaturate an image – that is, we want to replace each color by a new color that has the same lightness (so white stays white, black stays black, etc.), and the same hue (so each shade of blue stays the same shade of blue, rather than turning into purple or red), but the "chroma" is reduced (so colors are more muted). This is very difficult to do when working in RGB space. So let's take our colors and re-represent them in terms of lightness, chroma, and hue, using the state-of-the-art CIECAM02 model.

The three axes in this space are conventionally called "J" (for lightness), "C" (for chroma), and "h" (for hue). (The CIECAM-02 standard also defines a whole set of other axes with subtly different meanings – see Wikipedia for details – but for now we'll stick to these three.) To desaturate our image, we're going to switch from sRGB space to JCh space, reduce all the "C" values by a factor of 2, and then convert back to sRGB to look at the result. (Note that the CIECAM02 model in general requires the specification of a number of viewing condition parameters; here we accept the default, which happen to match the viewing conditions specified in the sRGB standard). All this takes more words to describe than it does to implement:

```
In [13]: hopper_desat_JCh = cspace_convert(hopper_sRGB, "sRGB1", "JCh")

# This is in "JCh" space, and we want to modify the "C" channel, so
# that's channel 1.
In [14]: hopper_desat_JCh[..., 1] /= 2

In [15]: hopper_desat_sRGB = cspace_convert(hopper_desat_JCh, "JCh", "sRGB1")
```

Let's see what this looks like. First we'll define a little utility function to plot several images together:

```
In [16]: def compare_hoppers(*new):
   ....:     image_width = 2.0  # inches
   ....:     total_width = (1 + len(new)) * image_width
   ....:     height = image_width / hopper_sRGB.shape[1] * hopper_sRGB.shape[0]
   ....:     fig = plt.figure(figsize=(total_width, height))
   ....:     ax = fig.add_axes((0, 0, 1, 1))
   ....:     ax.imshow(np.column_stack((hopper_sRGB,) + new))
   ....:
```

And now we'll use it to look at the desaturated image we computed above:

```
In [17]: compare_hoppers(hopper_desat_sRGB)
```

The original version is on the left, with our modified version on the right. Notice how in the version with reduced chroma, the colors are more muted, but not entirely gone.

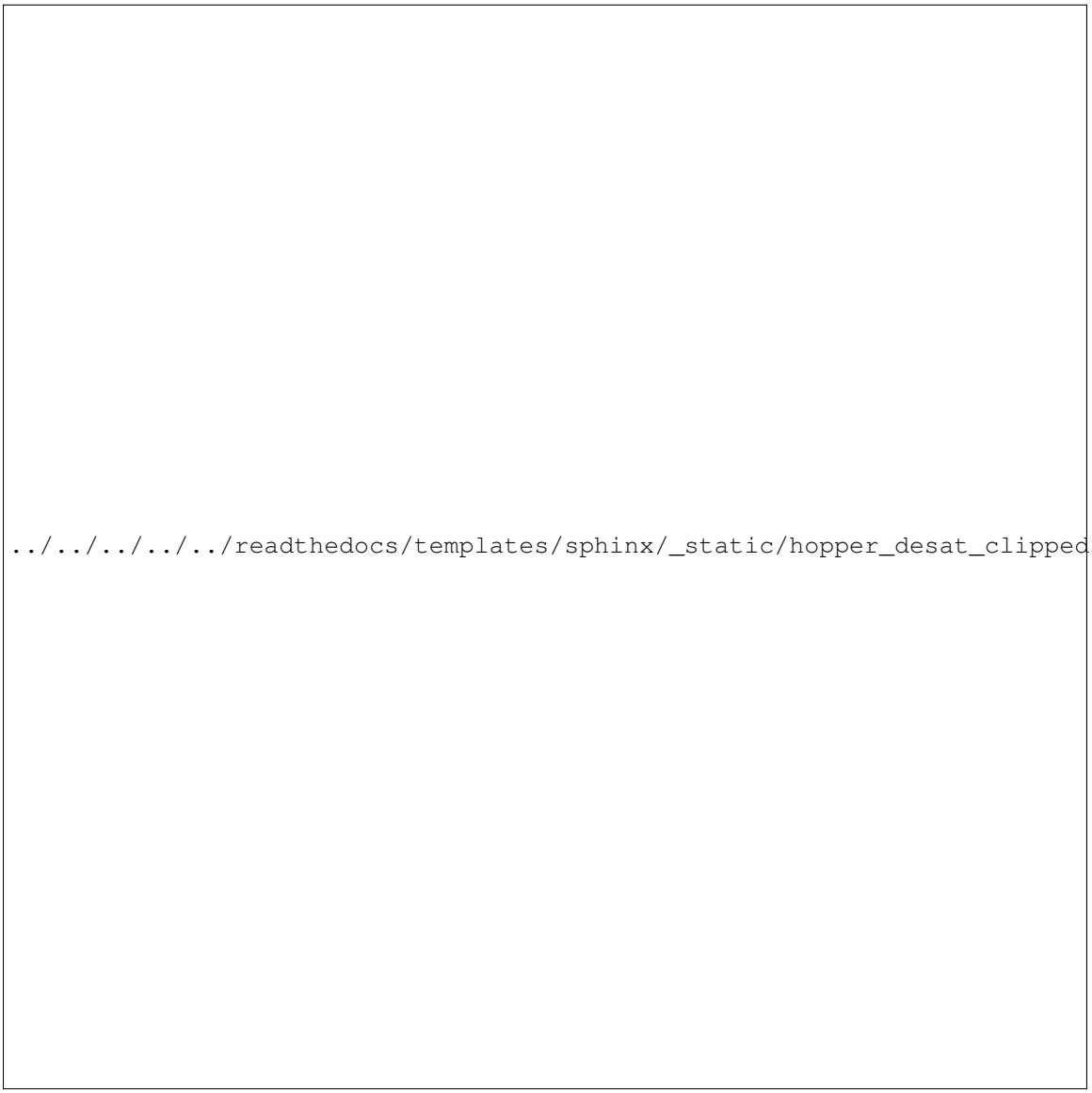
Except, there is one oddity – notice the small cyan patches on her collar and hat. This occurs due to floating point rounding error creating a few points with sRGB values that are greater than 1, which causes matplotlib to render the points in a strange way:

```
In [18]: hopper_desat_sRGB[np.any(hopper_desat_sRGB > 1, axis=-1), :]
Out[18]:
array([[ 1.00506547,  0.99532516,  0.96421717],
       [ 1.00104689,  0.98787282,  0.94567164],
       [ 1.00080521,  0.98563065,  0.96004546],
       ...,
       [ 1.00071535,  0.98363444,  0.97633881],
       [ 1.00445847,  0.99092599,  0.9908775 ],
       [ 1.00355835,  0.9900645 ,  0.97911882]])
```

Colorspacious doesn't do anything to clip such values, since they can sometimes be useful for further processing – e.g. when chaining multiple conversions together, you don't want to clip between intermediate steps, because this might

introduce errors. And potentially you might want to handle them in some clever way (there's a whole literature on how to solve such problems). But in this case, where the values are only just barely over 1, then simply clipping them to 1 is probably the best approach, and you can easily do this yourself. In fact, NumPy provides a standard function that we can use:

```
In [19]: compare_hoppers(np.clip(hopper_desat_sRGB, 0, 1))
```



No more cyan splotches!

Once we know how to represent an image in terms of lightness/chroma/hue, then there's all kinds of things we can do. Let's try reducing the chroma all the way to zero, for a highly accurate greyscale conversion:

```
In [20]: hopper_greyscale_JCh = cspace_convert(hopper_sRGB, "sRGB1", "JCh")

In [21]: hopper_greyscale_JCh[..., 1] = 0

In [22]: hopper_greyscale_sRGB = cspace_convert(hopper_greyscale_JCh, "JCh", "sRGB1")

In [23]: compare_hoppers(np.clip(hopper_greyscale_sRGB, 0, 1))
```

../../../../../readthedocs/templates/sphinx/_static/hopper_greyscale_unclipped.png

To explore, try applying other transformations. E.g., you could darken the image by rescaling the lightness channel "J" by a factor of 2 (`image_JCh[..., 0] /= 2`), or try replacing each hue by its complement (`image_JCh[..., 2] *= -1`).

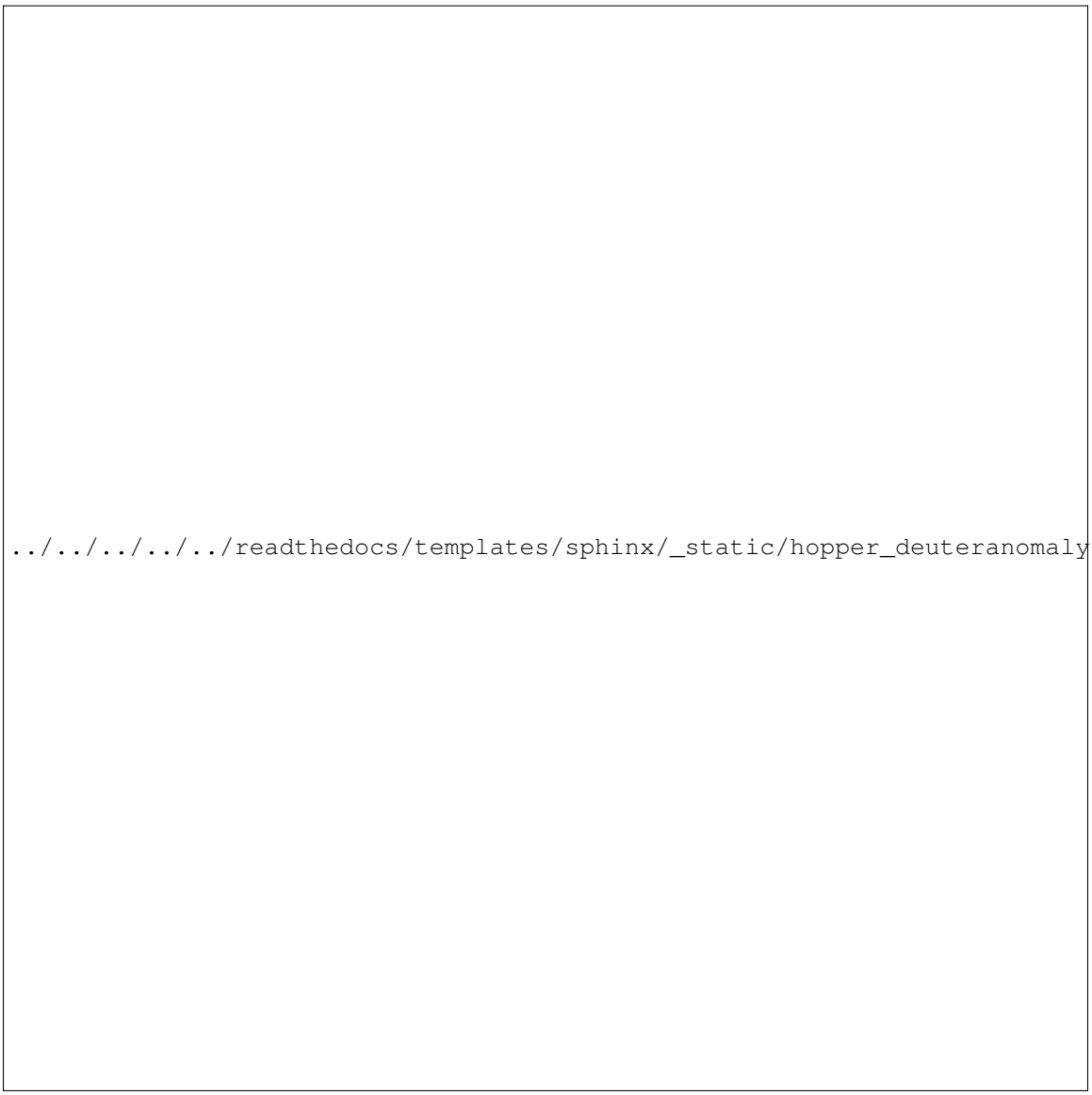
2.2 Simulating colorblindness

Another useful thing we can do by converting colorspaces is to simulate various sorts of color vision deficiency, a.k.a. "colorblindness". For example, deuteranomaly is the name for the most common form of red-green colorblindness, and affects ~5% of white men to varying amounts. Here's a simulation of what this image looks like to someone with a moderate degree of this condition. Notice the use of the extended syntax for describing color spaces that require extra parameters beyond just the name:

```
In [24]: cvd_space = {"name": "sRGB1+CVD",
   ....:              "cvd_type": "deuteranomaly",
   ....:              "severity": 50}
   ....:
```

```
In [25]: hopper_deuteranomaly_sRGB = cspace_convert(hopper_sRGB, cvd_space, "sRGB1")

In [26]: compare_hoppers(np.clip(hopper_deuteranomaly_sRGB, 0, 1))
```



Notice that contrary to what you might expect, we simulate CVD by asking *`cspace_convert()`* to convert *from* a special CVD space *to* the standard sRGB space. The way to think about this is that we have a set of RGB values that will be viewed under certain conditions, i.e. displayed on an sRGB monitor and viewed by someone with CVD. And we want to find a new set of RGB values that will look the same under a different set of viewing conditions, i.e., displayed on an sRGB monitor and viewed by someone with normal color vision. So we are starting in the `sRGB1+CVD` space, and converting to the normal `sRGB1` space.

This way of doing things is especially handy when you want to perform other operations. For example, we might want to use the JCh space described above to ask "what (approximate) lightness/chroma/hue would someone with this form of CVD perceive when looking at a monitor displaying a certain RGB value?". For example, taking a "pure red" color:

```
In [27]: cspace_convert([1, 0, 0], cvd_space, "JCh")
Out[27]: array([ 47.72696721,  62.75654782,  71.41502844])
```

If we compare this to someone with normal color vision, we see that the person with CVD will perceive about the same lightness, but desaturated and with a shifted hue:

```
In [28]: cspace_convert([1, 0, 0], "sRGB1", "JCh")
Out[28]: array([  46.9250674,  111.3069358,   32.1526953])
```

The model of CVD we use allows a "severity" scaling factor, specified as a number between 0 and 100. A severity of 100 corresponds to complete dichromacy:

```
In [29]: cvd_space = {"name": "sRGB1+CVD",
   ....:              "cvd_type": "deuteranomaly",
   ....:              "severity": 100}
   ....:

In [30]: hopper_deuteranopia_sRGB = cspace_convert(hopper_sRGB, cvd_space, "sRGB1")

In [31]: compare_hoppers(np.clip(hopper_deuteranomaly_sRGB, 0, 1),
   ....:                 np.clip(hopper_deuteranopia_sRGB, 0, 1))
   ....:
```

../../../../../readthedocs/templates/sphinx/_static/hopper_deuteranopia.png

Here the leftmost and center images are repeats of ones we've seen before: the leftmost image is the original, and the center image is the moderate deuteranomaly simulation that we computed above. The image on the right is the new image illustrating the more severe degree of red-green colorblindness – notice how the red in the flag and her medals is muted in the middle image, but in the image on the right it's disappeared completely.

You can also set the `"cvd_type"` to `"protanomaly"` to simulate the other common form of red-green color-

blindness, or to `"tritanomaly"` to simulate an extremely rare form of blue-yellow colorblindness. Here's what moderate and severe protanomaly look like when simulated by colorspacious:

```
In [32]: cvd_space = {"name": "sRGB1+CVD",
   ....:              "cvd_type": "protanomaly",
   ....:              "severity": 50}
   ....:

In [33]: hopper_protanomaly_sRGB = cspace_convert(hopper_sRGB, cvd_space, "sRGB1")

In [34]: cvd_space = {"name": "sRGB1+CVD",
   ....:              "cvd_type": "protanomaly",
   ....:              "severity": 100}
   ....:

In [35]: hopper_protanopia_sRGB = cspace_convert(hopper_sRGB, cvd_space, "sRGB1")

In [36]: compare_hoppers(np.clip(hopper_protanomaly_sRGB, 0, 1),
   ....:                 np.clip(hopper_protanopia_sRGB, 0, 1))
   ....:
```

../../../../../readthedocs/templates/sphinx/_static/hopper_protanopia.png

Because deuteranomaly and protanomaly are both types of red-green colorblindness, this is similar (but not quite identical) to the image we saw above.

2.3 Color similarity

Suppose we have two colors, and we want to know how different they will look to a person – often known as computing the "delta E" between them. One way to do this is to map both colors into a "perceptually uniform" colorspace, and then compute the Euclidean distance. Colorspacious provides a convenience function to do just this:

```
In [37]: from colorspacious import deltaE

In [38]: deltaE([1, 0.5, 0.5], [0.5, 1, 0.5])
Out[38]: 55.337158728500363

In [39]: deltaE([255, 127, 127], [127, 255, 127], input_space="sRGB255")
Out[39]: 55.490775265826485
```

By default, these computations are done using the CAM02-UCS perceptually uniform space (see *[LCL06]* for details), but if you want to use the (generally inferior) CIEL*a*b*, then just say the word:

```
In [40]: deltaE([1, 0.5, 0.5], [0.5, 1, 0.5], uniform_space="CIELab")
Out[40]: 114.05544189591937
```

CHAPTER 3

Reference

3.1 Conversion functions

colorspacious.**cspace_convert**(*arr*, *start*, *end*)

Converts the colors in `arr` from colorspace `start` to colorspace `end`.

Parameters

- **arr** – An array-like of colors.
- **end** (*start,*) – Any supported colorspace specifiers. See *Specifying colorspaces* for details.

colorspacious.**cspace_converter**(*start*, *end*)

Returns a function for converting from colorspace `start` to colorspace `end`.

E.g., these are equivalent:

```
out = cspace_convert(arr, start, end)
```

```
start_to_end_fn = cspace_converter(start, end)
out = start_to_end_fn(arr)
```

If you are doing a large number of conversions between the same pair of spaces, then calling this function once and then using the returned function repeatedly will be slightly more efficient than calling *cspace_convert()* repeatedly. But I wouldn't bother unless you know that this is a bottleneck for you, or it simplifies your code.

3.2 Specifying colorspaces

Colorspacious knows about a wide variety of colorspaces, some of which take additional parameters, and it can convert freely between any of them. Here's an image showing all the known spaces, and the conversion paths used. (This graph is generated directly from the source code: when you request a conversion between two spaces, *cspace_convert()* automatically traverses this graph to find the best conversion path. This makes it very easy to add support for new colorspaces.)

The most general and primitive way to specify a colorspace is via a dict, e.g., all the following are valid arguments that can be passed to *cspace_convert()*:

```
{"name": "XYZ100"}
{"name": "CIELab", "XYZ100_w": "D65"}
{"name": "CIELab", "XYZ100_w": [95.047, 100, 108.883]}
```

These dictionaries always have a "name" key specifying the colorspace. Every bold-faced string in the above image is a recognized colorspace name. Some spaces take additional parameters beyond the name, such as the CIELab whitepoint above. These additional parameters are indicated by the italicized strings in the image above.

There are also several shorthands accepted, to let you avoid writing out long dicts in most cases. In particular:

- Any *CIECAM02Space* object myspace is expanded to:

```
{"name": "CIECAM02",
 "ciecam02_space": myspace}
```

- Any *LuoEtAl2006UniformSpace* object myspace is expanded to:

```
{"name": "J'a'b'",
 "ciecam02_space": CIECAM02.sRGB,
 "luoetal2006_space": myspace}
```

- The string "CIELab" expands to: {"name": "CIELab", "XYZ100_w": "D65"}
- The string "CIELCh" expands to: {"name": "CIELCh", "XYZ100_w": "D65"}
- the string "CIECAM02" expands to CIECAM02Space.sRGB, which in turn expands to {"name": "CIECAM02", "ciecam02_space": CIECAM02Space.sRGB}.
- The strings "CAM02-UCS", "CAM02-SCD", "CAM02-LCD" expand to the global instance objects CAM02UCS, CAM02SCD, CAM02LCD, which in turn expand to "J'a'b'" dicts as described above.
- Any string consisting only of characters from the set "JChQMsH" is expanded to:

```
{"name": "CIECAM02-subset",
 "axes": <the string provided>
 "ciecam02_space": CIECAM02.sRGB}
```

  This allows you to directly use common shorthands like "JCh" or "JMh" as first-class colorspaces.

Any other string "foo" expands to {"name": "foo"}. So for any space that doesn't take parameters, you can simply say "sRGB1" or "XYZ100" or whatever and ignore all these complications.

And, as one final trick, any alias can also be used as the "name" field in a colorspace dict, in which case its normal expansion is used to provide overrideable defaults for parameters. For example:

```
# You write:
{"name": "CAM02-UCS",
 "ciecam02_space": my_ciecam02_space}

# Colorspacious expands this to:
{"name": "J'a'b'",
 "ciecam02_space": my_ciecam02_space,
 "luoetal2006_space": CAM02UCS}
```

Or:

```
# You write:
{"name": "JCh",
 "ciecam02_space": my_ciecam02_space}

# Colorspacious expands this to:
{"name": "CIECAM02-subset",
 "axes": "JCh",
 "ciecam02_space": my_ciecam02_space}
```

3.2.1 Well-known colorspaces

sRGB1, **sRGB100**: The standard sRGB colorspace. If you have generic "RGB" values with no further information specified, then usually the right thing to do is to assume that they are in the sRGB space; the sRGB space was originally designed to match the behavior of common consumer monitors, and these days common consumer monitors are designed to match sRGB. Use `sRGB1` if you have or want values that are normalized to fall between 0 and 1, and use `sRGB255` if you have or want values that are normalized to fall between 0 and 255.

XYZ100, **XYZ1**: The standard CIE 1931 XYZ color space. Use `XYZ100` if you have or want values that are normalized to fall between 0 and 100 (roughly speaking – values greater than 100 are valid in certain cases). Use `XYZ1` if you have or want values that are normalized to fall between 0 and 1 (roughly). This is a space which is "linear-light", i.e. related by a linear transformation to the photon counts in a spectral power distribution. In particular, this means that linear interpolation in this space is a valid way to simulate physical mixing of lights.

sRGB1-linear: A linear-light version of **sRGB1**, i.e., it has had gamma correction applied, but is still represented in terms of the standard sRGB primaries.

xyY100, **xyY1**: The standard CIE 1931 xyY color space. *The x and y values are always normalized to fall between 0 and 1.* Use `xyY100` if you have or want a Y value that falls between 0 and 100, and use `xyY1` if you have or want a Y value that falls between 0 and 1.

CIELab: The standard CIE 1976 L*a*b* color space. L* is scaled to vary from 0 to 100; a* and b* are likewise scaled to roughly the range -50 to 50. This space takes a parameter, *XYZ100_w*, which sets the reference white point, and may be specified either directly as a tristimulus value or as a string naming one of the well-known standard illuminants like `"D65"`.

CIELCh: Cylindrical version of **CIELab**. Accepts the same parameters. h* is in degrees.

3.2.2 Simulation of color vision deficiency

We provide simulation of common (and not so common) forms of color vision deficiency (also known as "colorblindness"), using the model described by *[MOF09]*.

This is generally done by specifying a colorspace like:

```
{"name": "sRGB1+CVD",
 "cvd_type": <type>,
 "severity": <severity>}
```

where `<type>` is one of the following strings:

- `"protanomaly"`: A common form of red-green colorblindness; affects ~2% of white men to some degree (less common among other ethnicities, much less common among women, see Tables 1.5 and 1.6 in *[SSJN00]*).
- `"deuteranomaly"`: The most common form of red-green colorblindness; affects ~6% of white men to some degree (less common among other ethnicities, much less common among women, see Tables 1.5 and 1.6 in *[SSJN00]*).
- `"tritanomaly"`: A very rare form of colorblindness affecting blue/yellow discrimination – so rare that its detailed effects and even rate of occurrence are not well understood. Affects <0.1% of people, possibly much less (*[SSJN00]*, page 47). Also, the name we use here is somewhat misleading because only full trit**anopia** has been documented, and partial trit**anomaly** likely does not exist (*[SSJN00]*, page 45). What this means is that while Colorspacious will happily allow any severity value to be passed, probably only severity = 100 corresponds to any real people.

And `<severity>` is any number between 0 (indicating regular vision) and 100 (indicating complete dichromacy).

Warning: If you have an image, e.g. a photo, and you want to "convert it to simulate colorblindness", then this is done with an incantation like:

```
cspace_convert(img, some_cvd_space, "sRGB1")
```

Notice that these arguments are given in the *opposite order* from what you might naively expect. See *Simulating colorblindness* for explanation and worked examples.

3.2.3 CIECAM02

CIECAM02 is a standardized, rather complex, state-of-the-art color appearance model, i.e., it's not useful for describing the voltage that should be applied to a phosphorescent element in your monitor (like RGB was originally designed to do), and it's not useful for modelling physical properties of light (like XYZ), but it is very useful to tell you what a color will look like subjectively to a human observer, under a certain set of viewing conditions. Unfortunately this makes it rather complicated, because human vision is rather complicated.

If you just want a better replacement for traditional ad hoc spaces like "Hue/Saturation/Value", then use the string `"JCh"` for your colorspace (see *Perceptual transformations* for a tutorial) and be happy.

If you want the full power of CIECAM02, or just to understand what *exactly* is happening when you type `"JCh"`, then read on.

First, you need to specify your viewing conditions. For many purposes, you can use the default *`CIECAM02Space.sRGB`* object. Or if you want to specify different viewing conditions, you can instantiate your own *`CIECAM02Space`* object:

class `colorspacious.`**`CIECAM02Space`**(*XYZ100_w*, *Y_b*, *L_A*, *surround=CIECAM02Surround(F=1.0, c=0.69, N_c=1.0)*)

An object representing a particular set of CIECAM02 viewing conditions.

Parameters

- **`XYZ100_w`** – The whitepoint. Either a string naming one of the known standard whitepoints like `"D65"`, or else a point in XYZ100 space.
- **`Y_b`** – Background luminance.
- **`L_A`** – Luminance of the adapting field (in cd/m^2).
- **`surround`** – A *`CIECAM02Surround`* object.

`sRGB`

A class-level constant representing the viewing conditions specified in the sRGB standard. (The sRGB standard defines two things: how a standard monitor should respond to different RGB values, and a standard set of viewing conditions in which you are supposed to look at such a monitor, and that attempt to approximate the average conditions in which people actually do look at such monitors. This object encodes the latter.)

The CIECAM02Space object has some low-level methods you can use directly if you want, though usually it'll be easier to just use *`cspace_convert()`*:

`XYZ100_to_CIECAM02`(*XYZ100*, *on_negative_A='raise'*)

Computes CIECAM02 appearance correlates for the given tristimulus value(s) XYZ (normalized to be on the 0-100 scale).

Example: `vc.XYZ100_to_CIECAM02([30.0, 45.5, 21.0])`

Parameters

- **XYZ100** – An array-like of tristimulus values. These should be given on the 0-100 scale, not the 0-1 scale. The array-like should have shape `(..., 3)`; e.g., you can use a simple 3-item list (shape = `(3,)`), or to efficiently perform multiple computations at once, you could pass a higher-dimensional array, e.g. an image.
- **on_negative_A** – A known infelicity of the CIECAM02 model is that for some inputs, the achromatic signal A can be negative, which makes it impossible to compute J, C, Q, M, or s – only h: and H are spared. (See, e.g., section 2.6.4.1 of *[LLI3]* for discussion.) This argument allows you to specify a strategy for handling such points. Options are:
  - `"raise"`: throws a *NegativeAError* (a subclass of `ValueError`)
  - `"nan"`: return not-a-number values for the affected elements. (This may be particularly useful if converting a large number of points at once.)

Returns A named tuple of type *JChQMsH*, with attributes `J`, `C`, `h`, `Q`, `M`, `s`, and `H` containing the CIECAM02 appearance correlates.

CIECAM02_to_XYZ100(*J=None*, *C=None*, *h=None*, *Q=None*, *M=None*, *s=None*, *H=None*)

Return the unique tristimulus values that have the given CIECAM02 appearance correlates under these viewing conditions.

You must specify 3 arguments:

•Exactly one of `J` and `Q`

•Exactly one of `C`, `M`, and `s`

•Exactly one of `h` and `H`.

Arguments can be vectors, in which case they will be broadcast against each other.

Returned tristimulus values will be on the 0-100 scale, not the 0-1 scale.

class colorspacious.**CIECAM02Surround**(*F*, *c*, *N_c*)

A namedtuple holding the CIECAM02 surround parameters, F, c, and N_c.

The CIECAM02 standard surrounds are available as constants defined on this class; for most purposes you'll just want to use one of them:

•`CIECAM02Surround.AVERAGE`

•`CIECAM02Surround.DIM`

•`CIECAM02Surround.DARK`

class colorspacious.**NegativeAError**

A `ValueError` that can be raised when converting to CIECAM02.

See *CIECAM02Space.XYZ100_to_CIECAM02()* for details.

Now that you have a *CIECAM02Space* object, what can you do with it?

First, you can pass it directly to *cspace_convert()* as an input or output space (which is a shorthand for using a space like `{"name": "CIECAM02", "ciecam02_space": <whatever>}`).

The plain vanilla `"CIECAM02"` space is weird and special: unlike all the other spaces supported by colorspacious, it does not represent values with ordinary NumPy arrays. This is because there are just too many perceptual correlates, and trying to keep track of whether M is at index 4 or 5 would be way too obnoxious. Instead, it returns an object of class *JChQMsH*:

class colorspacious.**JChQMsH**(*J*, *C*, *h*, *Q*, *M*, *s*, *H*)

A namedtuple with a mnemonic name: it has attributes `J`, `C`, `h`, `Q`, `M`, `s`, and `H`, each of which holds a scalar or NumPy array representing lightness, chroma, hue angle, brightness, colorfulness, saturation, and hue composition, respectively.

Alternatively, because you usually only want a subset of these, you can take advantage of the `"CIECAM02-subset"` space, which takes the perceptual correlates you want as a parameter. So for example if you just want JCh, you can write:

```
{"name": "CIECAM02-subset",
 "axes": "JCh",
 "ciecam02_space": CIECAM02.sRGB}
```

When using `"CIECAM02-subset"`, you don't have to worry about *JChQMsH* – it just takes and returns regular NumPy arrays, like all the other colorspaces.

And as a convenience, all strings composed of the character JChQMsH are automatically treated as specifying CIECAM02-subset spaces, so you can write:

```
"JCh"
```

and it expands to:

```
{"name": "CIECAM02-subset",
 "axes": "JCh",
 "ciecam02_space": CIECAM02.sRGB}
```

or you can write:

```
{"name": "JCh",
 "ciecam02_space": my_space}
```

and it expands to:

```
{"name": "CIECAM02-subset",
 "axes": "JCh",
 "ciecam02_space": my_space}
```

3.2.4 Perceptually uniform colorspaces based on CIECAM02

The $J'a'b'$ spaces proposed by *[LCL06]* are high-quality, approximately perceptually uniform spaces based on CIECAM02. They propose three variants: CAM02-LCD optimized for "large color differences" (e.g., estimating the similarity between blue and green), CAM02-SCD optimized for "small color differences" (e.g., estimating the similarity between light blue with a faint greenish cast and light blue with a faint purpleish cast), and CAM02-UCS which attempts to provide a single "uniform color space" that is less optimized for either case but provides acceptable performance in general.

Colorspacious represents these spaces as instances of *LuoEtAl2006UniformSpace*:

class `colorspacious.`**`LuoEtAl2006UniformSpace`**(*KL*, *c1*, *c2*)

A uniform space based on CIECAM02.

See *[LCL06]* for details of the parametrization.

For most purposes you should just use one of the predefined instances of this class that are exported as module-level constants:

- `colorspacious.CAM02UCS`
- `colorspacious.CAM02LCD`
- `colorspacious.CAM02SCD`

Because these spaces are defined as transformations from CIECAM02, to have a fully specified color space you must also provide some particular CIECAM02 viewing conditions, e.g.:

```
{"name": "J'a'b'",
 "ciecam02_space": CIECAM02.sRGB,
 "luoetal2006_space": CAM02UCS}
```

As usual, you can also pass any instance of *LuoEtAl2006UniformSpace* and it will be expanded into a dict like the above, or for the three common variants you can pass the strings `"CAM02-UCS"`, `"CAM02-LCD"`, or `"CAM02-SCD"`.

Changed in version 1.1.0: In v1.0.0 and earlier, colorspacious's definitions of the `CAM02-LCD` and `CAM02-SCD` spaces were swapped compared to what they should have been based on the *[LCL06]* – i.e., if you asked for LCD, you got SCD, and vice-versa. (`CAM02-UCS` was correct, though). Starting in 1.1.0, all three spaces are now correct.

3.3 Color difference computation

`colorspacious.`**`deltaE`**(*color1*, *color2*, *input_space='sRGB1'*, *uniform_space='CAM02-UCS'*)

Computes the ΔE distance between pairs of colors.

Parameters

- **`input_space`** – The space the colors start out in. Can be anything recognized by *cspace_convert()*. Default: "sRGB1"
- **`uniform_space`** – Which space to perform the distance measurement in. This should be a uniform space like CAM02-UCS where Euclidean distance approximates similarity judgements, because otherwise the results of this function won't be very meaningful, but in fact any color space known to *cspace_convert()* will be accepted.

By default, computes the euclidean distance in CAM02-UCS $J'a'b'$ space (thus giving $\Delta E'$); for details, see *[LCL06]*. If you want the classic ΔE^*_{ab} defined by CIE 1976, use `uniform_space="CIELab"`. Other good choices include `"CAM02-LCD"` and `"CAM02-SCD"`.

This function has no ability to perform ΔE calculations like CIEDE2000 that are not based on euclidean distances.

This function is vectorized, i.e., color1, color2 may be arrays with shape (..., 3), in which case we compute the distance between corresponding pairs of colors.

For examples, see *Color similarity* in the tutorial.

3.4 Utilities

You probably won't need these, but just in case they're useful:

`colorspacious.`**`standard_illuminant_XYZ100`**(*name*, *observer='CIE 1931 2 deg'*)

Takes a string naming a standard illuminant, and returns its XYZ coordinates (normalized to Y = 100).

We currently have the following standard illuminants in our database:

- `"A"`
- `"C"`
- `"D50"`
- `"D55"`
- `"D65"`
- `"D75"`

If you need another that isn't on this list, then feel free to send a pull request.

When in doubt, use D65: it's the whitepoint used by the sRGB standard (61966-2-1:1999) and ISO 10526:1999 says "D65 should be used in all colorimetric calculations requiring representative daylight, unless there are specific reasons for using a different illuminant".

By default, we return points in the XYZ space defined by the CIE 1931 2 degree standard observer. By specifying `observer="CIE 1964 10 deg"`, you can instead get the whitepoint coordinates in XYZ space defined by the CIE 1964 10 degree observer. This is probably only useful if you have XYZ points you want to do calculations on that were somehow measured using the CIE 1964 color matching functions, perhaps via a spectrophotometer; consumer equipment (monitors, cameras, etc.) assumes the use of the CIE 1931 standard observer in all cases I know of.

`colorspacious.`**`as_XYZ100_w`**(*whitepoint*)

A convenience function for getting whitepoints.

`whitepoint` can be either a string naming a standard illuminant (see *`standard_illuminant_XYZ100()`*), or else a whitepoint given explicitly as an array-like of XYZ values.

We internally call this function anywhere you have to specify a whitepoint (e.g. for CIECAM02 or CIELAB conversions).

Always uses the "standard" 2 degree observer.

`colorspacious.`**`machado_et_al_2009_matrix`**(*cvd_type*, *severity*)

Retrieve a matrix for simulating anomalous color vision.

Parameters

- **`cvd_type`** – One of "protanomaly", "deuteranomaly", or "tritanomaly".
- **`severity`** – A value between 0 and 100.

Returns A 3x3 CVD simulation matrix as computed by Machado et al (2009).

These matrices were downloaded from:

http://www.inf.ufrgs.br/~oliveira/pubs_files/CVD_Simulation/CVD_Simulation.html

which is supplementary data from *[MOF09]*.

If severity is a multiple of 10, then simply returns the matrix from that webpage. For other severities, performs linear interpolation.

CHAPTER 4

Changes

4.1 v1.1.0

- **BUG AFFECTING CALCULATIONS:** In previous versions, it turns out that the CAM02-LCD and CAM02-SCD spaces were accidentally swapped – so if you asked for CAM02-LCD you got SCD, and vice-versa. This has now been corrected. (Thanks to Github user TFiFiE for catching this!)
- Fixed setup.py to be compatible with both python 2 and python 3.
- Miscellaneous documentation improvements.

4.2 v1.0.0

Notable changes since v0.1.0 include:

- **BUG AFFECTING CALCULATIONS:** the sRGB viewing conditions (`colorspacious.CIECAM02Space.sRGB`), which are used by default in all calculations involving CIECAM02 or CAM02-UCS, were previously incorrect – the L_A parameter was supposed to be $(64/\pi)/5$, but instead was incorrectly calculated as $(64/\pi) * 5$. The effect of this was to assume much brighter ambient lighting than actually specified by the sRGB standard (i.e., the sRGB standard assumes that you are looking at your monitor in a dim environment, like a movie theatre; we were calculating as if you were looking at your monitor in an environment that was 125 times lighter – something like, outside on an overcast day). This bug is corrected in this release.

  Fortunately this turns out to have had a negligible effect on viridis and the other matplotlib colormaps that were computed using the buggy code. Once the bug is corrected, the old colormaps' perceptual uniformity is no long analytically exactly perfect, but the deviations are numerically negligible, so there's no need to regenerate the colormaps. (Indeed, the buggy viewing conditions, while different from those specified in IEC 61966-2-1:1999, are probably still within the range of realistic viewing conditions where these colormaps will be used.)

  If it is necessary to reproduce results using the old code, then this can be accomplished by instantiating a custom `CIECAM02Space` object:

```
from colorspacious import CIECAM02Space
# almost, but not quite, the sRGB viewing conditions:
buggy_space = CIECAM02Space(
    XYZ100_w="D65",
    Y_b=20,
    # bug: should be (64 / np.pi) / 5
    L_A=(64 / np.pi) * 5)
```

This can be used directly, or to create custom colorspace specifications to use with `cspace_convert()`. E.g., to convert from sRGB1 to JCh using the buggy viewing conditions:

```
cspace_convert(..., "sRGB1",
               {"name": "JCh", "ciecam02_space": buggy_space})
```

Or to convert from XYZ100 to CAM02-UCS using the buggy viewing conditions:

```
cspace_convert(..., "XYZ100",
               {"name": "CAM02-UCS", "ciecam02_space": buggy_space})
```

Similar code has been added to viscm to allow reproduction and editing of viridis and related colormaps that were designed using the old code.

- *`colorspacious.deltaE()`* is now available as a convenience function for computing the perceptual distance between colors.
- Substantially improved docs (i.e. there is now actually a comprehensive manual).
- Better test coverage (currently at 100% statement and branch coverage).
- Miscellaneous bug fixes.

4.3 v0.1.0

Initial release.

CHAPTER 5

Bibliography

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Bibliography

[LCL06] M. Ronnier Luo, Guihua Cui, and Changjun Li. Uniform colour spaces based on CIECAM02 colour appearance model. *Color Research & Application*, pages 320–330, 2006. doi:10.1002/col.20227.

[LL13] Ming Ronnier Luo and Changjun Li. CIECAM02 and its recent developments. In Christine Fernandez-Maloigne, editor, *Advanced color image processing and analysis*, pages 19–58. Springer New York, New York, 2013. doi:10.1007/978-1-4419-6190-7_2.

[MOF09] Gustavo M. Machado, Manuel M. Oliveira, and Leandro A. F. Fernandes. A physiologically-based model for simulation of color vision deficiency. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1291–1298, 2009. URL: http://www.inf.ufrgs.br/~oliveira/pubs_files/CVD_Simulation/CVD_Simulation.html, doi:10.1109/TVCG.2009.113.

[SSJN00] Lindsay T. Sharpe, Andrew Stockman, Herbert Jägle, and Jeremy Nathans. Opsin genes, cone photopigments and color vision. In Karl R. Gegenfurtner and Lindsay T. Sharpe, editors, *Color vision: From genes to perception*, pages 3–51. Cambridge University Press, Cambridge, 2000.

A

C

D

J

L

M

N

S

X