UNIVERSITÄT
BERN

Faculty of Business, Economics and Social Sciences

Department of Social Sciences

University of Bern Social Sciences Working Paper No. 42

ColrSpace: A Mata class for color management

Ben Jann

Current version: June 3, 2022
First version: April 10, 2022

http://ideas.repec.org/p/bss/wpaper/42.html
http://econpapers.repec.org/paper/bsswpaper/42.htm

University of Bern
Department of Social Sciences
Fabrikstrasse 8
CH-3012 Bern

Tel. +41 (0)31 684 48 11
Fax +41 (0)31 684 48 17
info@sowi.unibe.ch
www.sowi.unibe.ch

ColrSpace: A Mata class for color management

Ben Jann
Institute of Sociology
University of Bern
ben.jann@unibe.ch

Abstract. `ColrSpace` is a class-based color management system implemented in Mata. It supports a wide variety of color spaces and translations among them, provides color generators and a large collection of named palettes, and features functionality such as color interpolation, grayscale conversion, or color vision deficiency simulation. `ColrSpace` requires Stata 14.2 or newer.

Keywords: Stata, Mata, `Colrspace`, color, color palette, colormap, color generator, color interpolation, color mixing, color vision deficiency, color blindness, grayscale conversion, color difference, color contrast, color space, RGB, HSV, HSL, CMYK, CIE 1931 XYZ, CIELAB, CIELUV, HCL, CIECAM02, gamma compression, chromatic adaption, hue, brightness, luminance, lightness, chroma, chromaticity, colorfulness, saturation, opacity, perceptually uniform, colorblind-friendly, web colors

Contents

1 Introduction

The `ColrSpace` package provides a full-blown color management system written in Mata. It supports a wide variety of color spaces and translations among them, provides color generators and a large collection of colormaps, named palettes, and named colors, and features functionality such as color interpolation, grayscale conversion, or color vision deficiency simulation.

`ColrSpace` is primarily intended for use by programmers and the purpose of this document is to provide comprehensive documentation of the `ColrSpace` class system. Although interactive use of `ColrSpace` is possible, most applied users will find it easier to work with canned Stata commands such as `colorpalette` and `colorcheck`, which are built upon `ColrSpace` (for `colorpalette` see Jann 2018).

2 Installation

`ColrSpace` requires Stata 14.2 or newer. A compiled version of `ColrSpace` is available from the SSC Archive. To install `ColrSpace` type

```
. ssc install colrspace, replace
```

Some of the examples below make use of the `colorpalette` command, which is part of the `palettes` package (Jann 2018). To install the `palettes` package, type

```
. ssc install palettes, replace
```

`ColrSpace` and `palettes` are also available on GitHub; see github.com/benjann/colrspace and github.com/benjann/palettes. To install the packages from GitHub (as an alternative to installing from SSC), type

```
. net from https://raw.githubusercontent.com/benjann/colrspace/master/
. net install colrspace, replace
. net from https://raw.githubusercontent.com/benjann/palettes/master/
. net install palettes, replace
```

3 Overview of color spaces

A key feature of `ColrSpace` is that it can translate between many different color representations. Such translations are used, for example, when interpolating colors. The diagram in Figure 1 displays an overview of the different color spaces and coding schemes supported by `ColrSpace`. The shown acronyms are the names by which the color spaces are referred to in `ColrSpace`. The diagram also illustrates the path along which colors are transformed from one color space into another. The different color representations are as follows.

- `HEX` is a hex RGB value (hex triplet; see https://en.wikipedia.org/wiki/Web_colors). Examples are `"#ffffff"` for white or `"#1a476f"` for Stata's navy. `ColrSpace` will always return hex colors using their lowercase 6-digit codes. As input, however, uppercase spelling and 3-digit abbreviations are allowed. For example, white can be specified as are `"#ffffff"`, `"#FFFFFF"`, `"#fff"`, or `"#FFF"`.
- `RGB` is an RGB triplet (red, green, blue) in 0–255 scaling (see https://en.wikipedia.org/wiki/RGB_color_model). When returning `RGB` values, `ColrSpace` will round the values to integers and clip them at 0 and 255.
- `RGB1` is an RGB triplet in 0–1 scaling. `ColrSpace` does not clip or round the values and may thus return values larger than 1 or smaller than 0. Using unclipped values ensures consistency of translations among different color spaces. To retrieve a matrix of clipped values, you can type

  C `=` S`.clip(`S`.get("RGB1"), 0, 1)`

  `RGB1` is the native format in which `ColrSpace` stores colors internally. By default, `ColrSpace` assumes that the colors are in the standard RGB working space

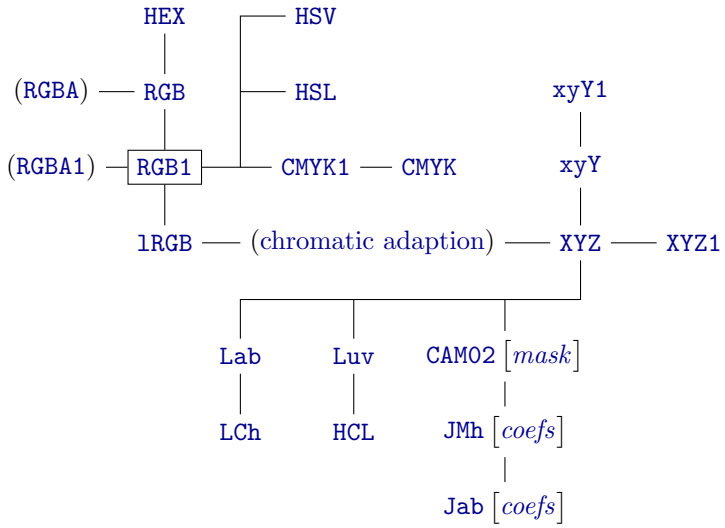


Figure 1: Color spaces and coding schemes supported by `ColrSpace`

(`"sRGB"`), but this can be changed; see function *S*`.rgbspace()`. Note that changing the RGB working space after colors have been added to a `ColrSpace` object will not change the stored values. To transform colors from one RGB working space to another RGB working space, export the colors to `XYZ` typing *C* `=` *S*`.get("XYZ")`, change the RGB working space using function *S*`.rgbspace()`, and then reimport the colors typing *S*`.set(`*C*`, "XYZ")`.

- `lRGB` stands for linear RGB in 0–1 scaling, that is, `RGB1`, from which gamma compression has been removed.
- `HSV` is a color triplet in the HSV (hue, saturation, value) color space. Hue is in degrees of the color wheel (0–360), saturation and value are numbers in $[0, 1]$. `ColrSpace` uses the procedure described in https://en.wikipedia.org/wiki/HSL_and_HSV to translate between HSV and RGB.
- `HSL` is a color triplet in the HSL (hue, saturation, lightness) color space. Hue is in degrees of the color wheel (0–360), saturation and lightness are numbers in $[0, 1]$. `ColrSpace` uses the procedure described in https://en.wikipedia.org/wiki/HSL_and_HSV to translate between HSL and RGB.
- `CMYK` is a CMYK quadruplet (cyan, magenta, yellow, black) in 0–255 scaling. When returning CMYK values, `ColrSpace` will round the values to integers and clip them at 0 and 255. There is no unique standard method to translate between CMYK and RGB, as translation is device-specific. `ColrSpace` uses the same translation as is implemented in official Stata (for CMYK to RGB see program `setcmyk` in file `color.class`; for RGB to CMYK see program `rgb2cmyk` in file `palette.ado`).

- `CMYK1` is a CMYK quadruplet (cyan, magenta, yellow, black) in 0–1 scaling. `ColrSpace` does not clip or round the values and may thus return values larger than 1 or smaller than 0. To retrieve a matrix of clipped values, you can type

  C `=` S`.clip(`S`.get("CMYK1"), 0, 1)`

  See `CMYK` for additional information.

- `XYZ` is a CIE 1931 XYZ tristimulus value in $Y_{\text{white}} = 100$ scaling. See https://en.wikipedia.org/wiki/CIE_1931_color_space for background information. XYZ values are defined with respect to a reference white; see function S`.xyzwhite()`. The default illuminant used by `ColrSpace` to define the reference white is `"D65"` (noon daylight for a CIE 1931 2° standard observer). To transform RGB to CIE XYZ, `ColrSpace` first removes gamma compression to obtain linear RGB (`lRGB`) and then transforms linear RGB to XYZ using an appropriate transformation matrix (see, e.g., Pascale 2003 for detailed explanations of both steps), possibly applying chromatic adaption to take account of a change in the reference white between the RGB working space and the XYZ color space.

- `XYZ1` is a CIE XYZ tristimulus value in $Y_{\text{white}} = 1$ scaling. See `XYZ` for additional information.

- `xyY` is a CIE xyY triplet, where x (cyan to red for y around .2) and y (magenta to green for x around .2) are the chromaticity coordinates in $[0, 1]$, with $x + y \leq 1$, and Y is the luminance in $Y_{\text{white}} = 100$ scaling (Y in CIE xyY is the same as Y in CIE XYZ). `ColrSpace` uses the procedure described in https://en.wikipedia.org/wiki/CIE_1931_color_space to translate between XYZ and xyY.

- `xyY1` is a CIE xyY triplet, with Y in $Y_{\text{white}} = 1$ scaling. See `xyY` for additional information.

- `Lab` is a color triplet in the CIE $L^*a^*b^*$ color space. L^* in $[0, 100]$ is the lightness of the color, a^* is the green ($-$) to red ($+$) component, b^* is the blue ($-$) to yellow ($+$) component. The range of a^* and b^* is somewhere around ± 100 for typical colors. `ColrSpace` uses the procedure described in https://en.wikipedia.org/wiki/CIELAB_color_space to translate between XYZ and CIE $L^*a^*b^*$.

- `LCh` is a color triplet in the CIE LCh color space (cylindrical representation of CIE $L^*a^*b^*$). L (lightness) in $[0, 100]$ is the same as L^* in CIE $L^*a^*b^*$, C (chroma) is the relative colorfulness (with typical values in a range of 0–100, although higher values are possible), h (hue) is the angle on the color wheel in degrees (0–360). See https://en.wikipedia.org/wiki/CIELAB_color_space.

- `Luv` is a color triplet in the CIE $L^*u^*v^*$ color space. L^* in $[0, 100]$ is the lightness of the color, u^* is the green ($-$) to red ($+$) component, v^* is the blue ($-$) to yellow ($+$) component. The range of u^* and v^* is somewhere around ± 100 for typical colors. `ColrSpace` uses the procedure described in https://en.wikipedia.org/wiki/CIELUV to translate between XYZ and CIE $L^*u^*v^*$. L^* in CIE $L^*u^*v^*$ is the same as L^* in CIE $L^*a^*b^*$.

- `HCL` is a color triplet in the HCL color space (cylindrical representation of CIE $L^*u^*v^*$). H (hue) is the angle on the color wheel in degrees (0–360), C (chroma) is the relative colorfulness (with typical values in a range of 0–100, although higher values are possible), L (lightness) in $[0, 100]$ is the same as L^* in CIE $L^*u^*v^*$. See https://en.wikipedia.org/wiki/CIELUV.

- `CAM02` is a color value in the CIECAM02 color space. See Luo and Li (2013) for details. In `ColrSpace`, CIECAM02 is specified as

  `"CAM02` $\left[mask\right]$`"`

  where optional *mask* selects the CIECAM02 attributes. The supported attributes are `Q` (brightness), `J` (lightness), `M` (colourfulness), `C` (chroma), `s` (saturation), `h` (hue angle), and `H` (hue composition). For example, you could type

  C `=` S`.get("CAM02 QJMCshH")`

  to obtain a $n \times 7$ matrix containing all available attributes for each color. When importing colors, e.g. using S`.colors()` or S`.set()`, *mask* must contain at least one of `Q` and `J`, at least one of `M`, `C`, and `s`, and at least one of `h` and `H`. If *mask* is omitted, `ColrSpace` assumes `"CAM02 JCh"`.

- `JMh` is a color triplet in the CIECAM02-based perceptually uniform $J'M'h$ color space. See Luo and Li (2013, chapter 2.6.1) and Luo et al. (2006) for details. In `ColrSpace`, $J'M'h$ is specified as

  `"JMh` $\left[coefs\right]$`"`

  where optional *coefs* selects the transformation coefficients. *coefs* can be

  `UCS`
  or `LCD`
  or `SCD`
  or K_L c_1 c_2

  (lowercase spelling and abbreviations allowed). `"JMh UCS"` is equivalent to `"JMh 1 .007 .0228"`, `"JMh LCD"` is equivalent to `"JMh .77 .007 .0053"`, `"JMh SCD"` is equivalent to `"JMh 1.24 .007 .0363"`. If *coefs* is omitted, the default coefficients as set by S`.ucscoefs()` will be used.

- `Jab` is a color triplet in the CIECAM02-based perceptually uniform $J'a'b'$ color space. See Luo and Li (2013, chapter 2.6.1) and Luo et al. (2006) for details. In `ColrSpace`, $J'a'b'$ is specified as

  `"Jab` $\left[coefs\right]$`"`

  where optional *coefs* is as described for `JMh`.

- `RGBA` is an opacity-extended RGB value (red, green, blue, alpha), where red, green, and blue are in 0–255 scaling and alpha is a number in $[0, 1]$ (0 = fully transparent,

1 = fully opaque). RGBA is not directly supported by *S*.convert(), but is allowed as input or output format in functions such as *S*.colors(), *S*.set(), or *S*.get(). Alternatively, in *S*.colors(), you can use non-extended RGB and specify opacity using Stata's [G] ***colorstyle*** syntax; for example "RGBA 26 71 111 0.7" is equivalent to "RGB 26 71 111%70" or "26 71 111%70" (see subsection 6.1). A further alternative is to manage opacity using *S*.opacity() or *S*.alpha().

- RGBA1 is an opacity-extended RGB value (red, green, blue, alpha), where red, green, and blue are in 0–1 scaling and alpha is a number in $[0, 1]$ (0 = fully transparent, 1 = fully opaque). See RGBA for additional information.

Internally, ColrSpace stores colors using their (unclipped) RGB1 values and additionally maintains an opacity value (alpha) in $[0, 1]$ and an intensity adjustment multiplier in $[0, 255]$ for each color.

4 Initialize a ColrSpace object

Initialize

To initialize a new ColrSpace object, type

```
S = ColrSpace()
```

where *S* is the name of the object. Alternatively, in non-interactive mode, declare a ColrSpace object as

```
class ColrSpace scalar S
```

Either way, after initialization, the object will be empty, that is, it will contain no colors. The object will be initialized, however, with the following default color space settings:

```
S.rgbspace("sRGB")
S.xyzwhite("D65")
S.viewcond(20, 64/(5*pi()), "average")
S.ucscoefs("UCS")
S.chadapt("Bfd")
```

Use *S*.settings() to display the current color space settings of object *S*. To restore the above defaults, you can type *S*.clearsettings().

Reinitialize

To reinitialize an existing ColrSpace object, type

S.clear()

This will remove all colors and meta data from S. Color space settings are not affected by *S*.clear(). Use *S*.clearsettings() if you want to reset the color space settings.

Clear internal look-up tables

Some of the functions below make use of look-up tables for palette names and named colors. ColrSpace stores these tables in S for reasons of efficiency. To remove these tables, type

S.clearindex()

This frees a little bit of memory, which may be relevant if you intend to create a lot of ColrSpace objects. The tables will be rebuilt automatically if a function is called that makes use of them.

5 Display contents and set meta data

Overview of contents

To display an overview of the contents of S, type

S.describe([*short*])

where $short \neq 0$ suppresses listing the individual colors.

Example

```
: S = ColrSpace()
: S.palette("HTML pink")
: S.describe()
  name()      = "HTML pink"
  pclass()    = "qualitative"
  note()      = "Pink HTML Colors from www.w3schools.com"
  source()    = "https://www.w3schools.com/colors/colors_groups.asp"
  isipolate() = 0
  N()         = 6
  N_added()   = 6
     Colors()               Names()                Info()
  ------------------------------------------------------------------
  1  255 192 203            Pink                   #FFC0CB
  2  255 182 193            LightPink              #FFB6C1
  3  255 105 180            HotPink                #FF69B4
  4  255 20 147             DeepPink               #FF1493
  5  219 112 147            PaleVioletRed          #DB7093
  6  199 21 133             MediumVioletRed        #C71585
: S.describe(1)
  name()      = "HTML pink"
```

```
pclass()    = "qualitative"
note()      = "Pink HTML Colors from www.w3schools.com"
source()    = "https://www.w3schools.com/colors/colors_groups.asp"
isipolate() = 0
N()         = 6
N_added()   = 6
```

Number of colors

To retrieve the number of colors defined in S, type

n = S.N[_added]()

S.N() returns the total number of colors; S.N_added() returns the number of colors added last.

Collection name

To assign a name or title to the collection of colors in S, type

S.name(*name*)

where *name* is a string scalar. To retrieve the name, type

name = S.name()

Class

To assign a class to the collection of colors in S, type

S.pclass(*class*)

where *class* is a string scalar such as "qualitative" (or "categorical"), "sequential", "diverging", or "circular" (or "cyclic"). To retrieve the class, type

class = S.pclass()

Description

To assign a description to the collection of colors in S, type

S.note(*note*)

where *note* is a string scalar. To retrieve the description, type

note = *S*.note()

Source

To assign information on the source of the colors in *S*, type

S.source(*source*)

where *source* is a string scalar. To retrieve the source, type

source = *S*.source()

Interpolation status

ColrSpace maintains a 0/1 flag of whether colors have been interpolated by *S*.ipolate(). To retrieve the status of the flag, type

flag = *S*.isipolate()

6 Define and transform colors

6.1 String input/output (Stata interface)

Color input

To import colors from a string scalar *colors* containing a list of color specifications, type

S.[add_]colors(*colors*[, *delimiter*])

or

rc = *S*._[add_]colors(*colors*[, *delimiter*])

where string scalar *delimiter* sets the character(s) delimiting the specifications; the default is to assume a space-separated list, i.e. *delimiter* = " ". To avoid breaking a specification that contains a delimiting character, enclose the specification in double quotes. *S*.colors() will replace preexisting colors in *S* by the new colors. Alternatively, use *S*.add_colors() to append the new colors to the existing colors. *S*._colors() and *S*._add_colors() perform the same action as *S*.colors() and *S*.add_colors(), but they return *rc* instead of aborting if *colors* contains invalid color specifications. *rc* will be set to the index of the first offending color specification, or to 0 if all specifications are valid. Also see function *S*.cvalid() for a way to check whether a color specification is valid.

To import colors from a string vector *Colors* (each element containing a single color

specification), type

S.[add_]Colors(*Colors*)

or

rc = *S*._[add_]Colors(*Colors*)

The syntax for a single color specification is

color[%#] [*#]

where %# sets the opacity (in percent; 0 = fully transparent, 100 = fully opaque), *# sets the intensity adjustment multiplier (values between 0 and 1 make the color lighter; values larger than one make the color darker), and *color* is one of the following:

name	a color name; this includes official Stata's color names as listed in [G] ***colorstyle***, possible user additions provided through style files, as well as a large collection of named colors provided by `ColrSpace`
`#`*rrggbb*	6-digit hex RGB value; white = `#FFFFFF` or `#ffffff`, navy = `#1A476F` or `#1a476f`
`#`*rgb*	3-digit abbreviated hex RGB value; white = `#FFF` or `#fff`
# # #	RGB value in 0–255 scaling; navy = `"26 71 111"`
# # # #	CMYK value in 0–255 or 0–1 scaling; navy = `"85 40 0 144"` or `".333 .157 0 .565"`
`RGB` # # #	RGB value in 0–255 scaling; navy = `"RGB 26 71 111"`
`RGB1` # # #	RGB value in 0–1 scaling; navy = `"RGB1 .102 .278 .435"`
`lRGB` # # #	linear RGB value in 0–1 scaling; navy = `"lRGB .0103 .063 .159"`
`CMYK` # # # #	CMYK value in 0–255 scaling; navy = `"CMYK 85 40 0 144"`
`CMYK1` # # # #	CMYK value in 0–1 scaling; navy = `"CMYK1 .333 .157 0 .565"`
`HSV` # # #	HSV value; navy = `"HSV 208 .766 .435"`
`HSL` # # #	HSL value; navy = `"HSL 208 .620 .269"`
`XYZ` # # #	CIE XYZ value in 0–100 scaling; navy = `"XYZ 5.55 5.87 15.9"`
`XYZ1` # # #	CIE XYZ value in 0–1 scaling; navy = `"XYZ1 .0555 .0587 .159"`
`xyY` # # #	CIE xyY value with Y in 0–100 scaling; navy = `"xyY .203 .215 5.87"`
`xyY1` # # #	CIE xyY value with Y in 0–1 scaling; navy = `"xyY1 .203 .215 .0587"`
`Lab` # # #	CIE $L^*a^*b^*$ value; navy = `"Lab 29 -.4 -27.5"`
`LCh` # # #	LCh value (polar CIE $L^*a^*b^*$; navy = `"LCh 29 27.5 269.2"`
`Luv` # # #	CIE $L^*u^*v^*$ value; navy = `"Luv 29 -15.4 -35.6"`
`HCL` # # #	HCL value (polar CIE $L^*u^*v^*$); navy = `"HCL 246.6 38.8 29"`
`CAM02` [*mask*] ...	CIECAM02 value according to *mask*; navy = `"CAM02 JCh 20.2 37 245"` or `"CAM02 QsH 55.7 69.5 303.5"`
`JMh` [*coefs*] # # #	CIECAM02 $J'M'h$ value; navy = `"JMh 30.1 21 245"`
`Jab` [*coefs*] # # #	CIECAM02 $J'a'b'$ value; navy = `"Jab 30.1 -8.9 -19"` or `"Jab LCD 39 -10.6 -23"`
`RGBA` # # # #	RGB 0–255 value where the last number specifies the opacity in $[0, 1]$
`RGBA1` # # # #	RGB 0–1 value where the last number specifies the opacity in $[0, 1]$

The color space identifiers (but not *mask*) can be typed in lowercase letters. The provided examples are for standard viewing conditions.

The named colors provided by `ColrSpace` in addition to Stata's named colors are as follows (type `help colrspace_library_namedcolors` to view the source file containing all color definitions):

140 HTML colors

`AliceBlue`, `AntiqueWhite`, `Aqua`, `Aquamarine`, `Azure`, `Beige`, `Bisque`, `Black`, `BlanchedAlmond`, `Blue`, `BlueViolet`, `Brown`, `BurlyWood`, `CadetBlue`, `Chartreuse`, `Chocolate`, `Coral`, `CornflowerBlue`, `Cornsilk`, `Crimson`, `Cyan`, `DarkBlue`, `DarkCyan`, `DarkGoldenRod`, `DarkGray`, `DarkGrey`, `DarkGreen`, `DarkKhaki`, `DarkMagenta`, `DarkOliveGreen`, `DarkOrange`, `DarkOrchid`, `DarkRed`, `DarkSalmon`, `DarkSeaGreen`, `DarkSlateBlue`, `DarkSlateGray`, `DarkSlateGrey`, `DarkTurquoise`, `DarkViolet`, `DeepPink`, `DeepSkyBlue`, `DimGray`, `DimGrey`, `DodgerBlue`, `FireBrick`, `FloralWhite`, `ForestGreen`, `Fuchsia`, `Gainsboro`, `GhostWhite`, `Gold`, `GoldenRod`, `Gray`, `Grey`, `Green`, `GreenYellow`, `HoneyDew`, `HotPink`, `IndianRed`, `Indigo`, `Ivory`, `Khaki`, `Lavender`, `LavenderBlush`, `LawnGreen`, `LemonChiffon`, `LightBlue`, `LightCoral`, `LightCyan`, `LightGoldenRodYellow`, `LightGray`, `LightGrey`, `LightGreen`, `LightPink`, `LightSalmon`, `LightSeaGreen`, `LightSkyBlue`, `LightSlateGray`, `LightSlateGrey`, `LightSteelBlue`, `LightYellow`, `Lime`, `LimeGreen`, `Linen`, `Magenta`, `Maroon`, `MediumAquaMarine`, `MediumBlue`, `MediumOrchid`, `MediumPurple`, `MediumSeaGreen`, `MediumSlateBlue`, `MediumSpringGreen`, `MediumTurquoise`, `MediumVioletRed`, `MidnightBlue`, `MintCream`, `MistyRose`, `Moccasin`, `NavajoWhite`, `Navy`, `OldLace`, `Olive`, `OliveDrab`, `Orange`, `OrangeRed`, `Orchid`, `PaleGoldenRod`, `PaleGreen`, `PaleTurquoise`, `PaleVioletRed`, `PapayaWhip`, `PeachPuff`, `Peru`, `Pink`, `Plum`, `PowderBlue`, `Purple`, `RebeccaPurple`, `Red`, `RosyBrown`, `RoyalBlue`, `SaddleBrown`, `Salmon`, `SandyBrown`, `SeaGreen`, `SeaShell`, `Sienna`, `Silver`, `SkyBlue`, `SlateBlue`, `SlateGray`, `SlateGrey`, `Snow`, `SpringGreen`, `SteelBlue`, `Tan`, `Teal`, `Thistle`, `Tomato`, `Turquoise`, `Violet`, `Wheat`, `White`, `WhiteSmoke`, `Yellow`, `YellowGreen`

30 W3.CSS default colors

`w3-red`, `w3-pink`, `w3-purple`, `w3-deep-purple`, `w3-indigo`, `w3-blue`, `w3-light-blue`, `w3-cyan`, `w3-aqua`, `w3-teal`, `w3-green`, `w3-light-green`, `w3-lime`, `w3-sand`, `w3-khaki`, `w3-yellow`, `w3-amber`, `w3-orange`, `w3-deep-orange`, `w3-blue-grey`, `w3-brown`, `w3-light-grey`, `w3-grey`, `w3-dark-grey`, `w3-black`, `w3-white`, `w3-pale-red`, `w3-pale-yellow`, `w3-pale-green`, `w3-pale-blue`

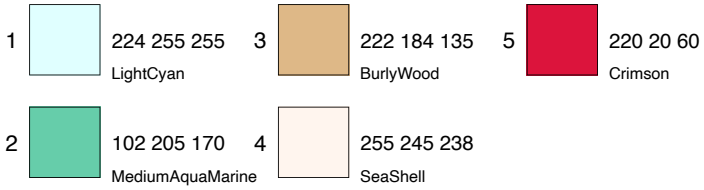
Further color collections from W3.CSS (using names as provided by W3.CSS, e.g. `w3-flat-turquoise`)
Flat UI Colors, Metro UI Colors, Windows 8 Colors, iOS Colors, US Highway Colors, US Safety Colors, European Signal Colors, Fashion Colors 2019, Fashion Colors 2018, Fashion Colors 2017, Vivid Colors, Food Colors, Camouflage Colors, ANA (Army Navy Aero) Colors, Traffic Colors

The color names can be abbreviated and typed in lowercase letters. If abbreviation is ambiguous, the first matching name in the alphabetically ordered list will be used. In case of name conflict with a Stata color, the color from `ColrSpace` will take precedence only if the specified name is an exact match including case. For example, `pink` will refer to official Stata's pink, whereas `Pink` will refer to HTML color pink.

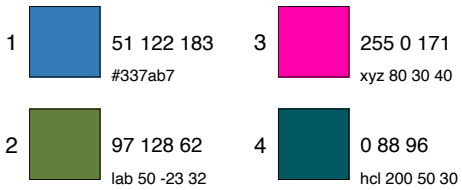
Examples

```
. mata: S = ColrSpace()
. mata: S.colors("LightCyan MediumAqua BurlyWood")
. colorpalette mata(S), rows(1)
```

```
. mata: S.add_colors("SeaShell Crimson")
. colorpalette mata(S), rows(2)
```




```
. mata: S.colors("#337ab7, lab 50 -23 32, xyz 80 30 40, hcl 200 50 30", ",")
. colorpalette mata(S), rows(2)
```




```
. mata: S.colors("navy*.5 orange%80 maroon*.7%60")
. colorpalette mata(S), rows(1)
```

Color output

To export colors into a string scalar containing a space-separated list of color specifications compatible with Stata graphics, type

colors = *S*.`colors`[`_added`]([*rgbforce*])

where *rgbforce* $\neq 0$ enforces exporting all colors using their in RGB values. Colors that have been defined in terms of their Stata color names are exported as is by default. Specify *rgbforce* $\neq 0$ to export these colors as RGB values. *S*.`colors()` exports all colors; use *S*.`colors_added()` to export only the colors that have been added last.

To export colors into a string column vector (each row containing a single color specification), type

Colors = *S*.`Colors`[`_added`]([*rgbforce*])

Examples

```
: S = ColrSpace()
```

```
: S.colors("navy*.5 orange%80 maroon*.7%60")
: S.colors()
  navy*.5 orange%80 maroon%60*.7
: S.colors(1)
  "26 71 111*.5" "255 127 0%80" "144 53 59%60*.7"
: S.add_colors("SeaShell Crimson")
: S.colors_added()
  "255 245 238" "220 20 60"
: S.Colors()
                  1
  ┌──────────────────┐
1 │       navy*.5    │
2 │     orange%80    │
3 │   maroon%60*.7   │
4 │    255 245 238   │
5 │      220 20 60   │
  └──────────────────┘
```

Names input

To import information from a string scalar *names* containing a list of color names, type

S.`names`[`_added`](*names*[`,` *delimiter*])

where string scalar *delimiter* sets the character(s) delimiting the names; the default is to assume a space-separated list, i.e. *delimiter* = `" "`. To avoid breaking a name that contains a delimiting character, enclose the name in double quotes. *S*.`names()` affects all colors defined in *S*; use *S*.`names_added()` to affect only the colors that have been added last.

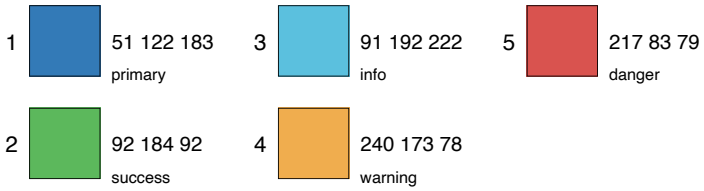
To import names from a string vector *Names* (each element containing a single name), type

S.`Names`[`_added`](*Names*)

Note that redefining the colors, e.g. by applying *S*.`colors()` or *S*.`set()`, will delete existing color names.

Example (using colors from getbootstrap.com/docs/3.3/)

```
. mata: S = ColrSpace()
. mata: S.colors("#337ab7 #5cb85c #5bc0de #f0ad4e #d9534f")
. mata: S.names("primary success info warning danger")
. colorpalette mata(S), rows(2)
```



Functions *S*.`colors()` and *S*.`palette()` fill in names automatically for colors that have a name.

Names output

To export color names into a string scalar containing a space-separated list of the descriptions, type

names = *S*.`names`[`_added`]`()`

S.`names()` exports names from all colors; use *S*.`names_added()` to export names only from the colors that have been added last.

Alternatively, to export the names into a string column vector (each row containing a single name) type

Names = *S*.`Names`[`_added`]`()`

Example

```
: S = ColrSpace()
: S.colors("SeaShell Crimson")
: S.colors()
  "255 245 238" "220 20 60"
: S.names()
  SeaShell Crimson
: S.Names()
              1
    ┌──────────┐
  1 │ SeaShell │
  2 │  Crimson │
    └──────────┘
```

Description input

To import information from a string scalar *info* containing a list of color descriptions (e.g. color names or other text describing a color), type

S.`info`[`_added`]`(`*info*[`,` *delimiter*]`)`

where string scalar *delimiter* sets the character(s) delimiting the descriptions; the default is to assume a space-separated list, i.e. *delimiter* = `" "`. To avoid breaking a description that contains a delimiting character, enclose the description in double quotes. *S*`.info()` affects all colors defined in *S*; use *S*`.info_added()` to affect only the colors that have been added last.

To import descriptions from a string vector *Info* (each element containing a single color description), type

S`.Info[_added](`*Info*`)`

Note that redefining the colors, e.g. by applying *S*`.colors()` or *S*`.set()`, will delete existing color descriptions.

Example (using colors from getbootstrap.com/docs/3.3/)

```
. mata: S = ColrSpace()
. mata: S.colors("#5cb85c #f0ad4e #d9534f")
. mata: S.info("color for success, color for warning, color for danger", ",")
. colorpalette mata(S), rows(1)
```

Functions *S*`.colors()` and *S*`.palette()` fill in descriptions automatically for colors that have been translated to RGB (the original color codes will be used as descriptions). Furthermore, modification functions such as *S*`.ipolate()` set the descriptions to the color codes in the color space in which the modification has been performed.

Description output

To export color descriptions into a string scalar containing a space-separated list of the descriptions, type

info `=` *S*`.info[_added]()`

S`.info()` exports descriptions from all colors; use *S*`.info_added()` to export descriptions only from the colors that have been added last.

Alternatively, to export the color descriptions into a string column vector (each row containing a single description) type

Info `=` *S*`.Info[_added]()`

Example

```
: S = ColrSpace()
```

```
: S.colors("SeaShell Crimson")
: S.colors()
  "255 245 238" "220 20 60"
: S.info()
  #FFF5EE #DC143C
: S.Info()
             1
  ┌──────────┐
1 │  #FFF5EE │
2 │  #DC143C │
  └──────────┘
```

6.2 Color palettes and color generators

`ColrSpace` features a large collection of named color palettes and color generators. The syntax to import colors from such a palette or color generator is

S.[`add_`]`palette(`[`"`*name*`",` *n*`,` *opt1*`,` *opt2*`,` *opt3*`,` *opt4*`]`)

where *name* selects the palette and *n* sets the desired number of colors. Arguments *opt1* to *opt4* depend the type of palette as explained below. *S*.`palette()` will replace preexisting colors in *S* by the new colors; *S*.`add_palette()` will append the new colors to the existing colors. The functions abort with error if *name* does not match an existing palette. See function *S*.`pexists()` for a way to check whether a palette exists or not. See function *S*.`palettes()` if you want to obtain a list available palettes.

There are three types of palettes, discussed under the following headings (also see repec.sowi.unibe.ch/stata/palettes/colors.html for an overview of all palettes):

Standard palettes

Colormaps

Color generators

Standard palettes

The syntax for standard palettes is

S.[`add_`]`palette(`[`"`*name*`",` *n*`,` *noexpand*`]`)

where

name selects the palette; see below for available names. Default is `s2`; this default can also be selected by typing `""`.

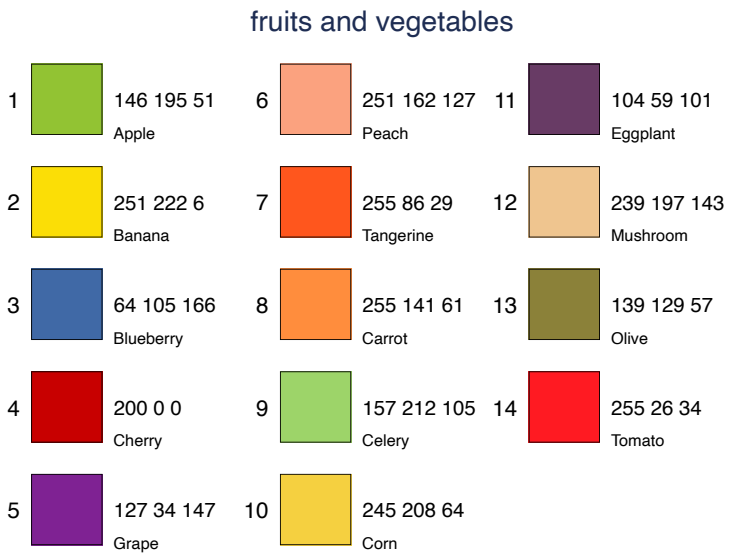
n is the number of colors to be retrieved from the palette. Many palettes, such as, e.g., the sequential and diverging ColorBrewer palettes, are adaptive to *n* in the sense that they return different colors depending on *n*. Other palettes, such as

s2, contain a fixed set of colors. In any case, if n is different from the (maximum or minimum) number of colors defined by a palette, the colors are either recycled (qualitative palettes; i.e. if S.pclass() is "qualitative" or "categorical") or interpolated (all other palettes) such that the number of retrieved colors is equal to n. Interpolation will be performed in the "Jab" space; if you want to interpolate in another space, specify *noexpand* $\neq 0$ and then apply S.ipolate().

noexpand $\neq 0$ prevents recycling or interpolating colors if n, the number of requested colors, is larger (smaller) than the maximum (minimum) number of colors defined by a palette. That is, if *noexpand* $\neq 0$ is specified, the resulting number of colors in S may be different from the requested number of colors. Exception: *noexpand* $\neq 0$ does not suppress "recycling" qualitative palettes if n is smaller than the (minimum) number of colors defined by the palette. In this case, the first n colors of the palette are retrieved irrespective of whether *noexpand* $\neq 0$ has been specified or not.

Example

```
. mata: S = ColrSpace()
. mata: S.palette("lin fruits")
. mata: S.add_palette("lin veg")
. mata: S.name("fruits and vegetables")
. colorpalette mata(S)
```



Currently available standard palettes are as follows:

s2	15 qualitative colors as in Stata's s2color scheme ([G] **Scheme s2**)
s1	15 qualitative colors as in Stata's s1color scheme ([G] **Scheme s1**)
s1r	15 qualitative colors as in Stata's s1rcolor scheme ([G] **Scheme s1**)

`economist`	15 qualitative colors as in Stata's `economist` scheme ([G] **Scheme economist**)
`mono`	15 gray scales (qualitative) as in Stata's monochrome schemes
`okabe`	8 CVD-friendly qualitative colors by Okabe and Ito (2002)
`cblind`	like `okabe`, but including gray on second position
`plottig`	15 qualitative colors as in `plottig` by Bischof (2017b)
`538`	13 qualitative colors as in `538` by Bischof (2017a)
`mrc`	8 qualitative colors as in `tfl` by Morris (2015)
`tfl`	7 qualitative colors as in `mrc` by Morris (2013)
`burd`	13 qualitative colors as in `burd` by Briatte (2013)
`lean`	15 gray scales (qualitative) as in `lean` by Juul (2003)
`tableau`	20 qualitative colors from Lin et al. (2013)
pals	qualitative palettes from the `pals` package in R (github.com/kwstat/pals), where *pals* is `alphabet` (26), `alphabet2` (26), `cols25` (25), `glasbey` (32), `kelly` (22; default), `polychrome` (36), or `watlington` (16)
`d3` [*scheme*]	qualitative palettes from D3.js, where *scheme* is `10` (default), `20`, `20b`, or `20c` (aliases: `tab10`, `tab20`, `tab20b`, and `tab20c`)
`sb` [*scheme*]	qualitative palettes from seaborn.pydata.org, where *scheme* is one of the following 10-color variants: `deep` (default), `muted`, `pastel`, `bright`, `dark`, `colorblind` 6-color variants: `deep6` (alias: `sb6`), `muted6`, `pastel6`, `bright6`, `dark6`, `colorblind6`
`tab` [*scheme*]	color schemes from Tableau 10 (source), where *scheme* is one of the following qualitative: `10` (default), `20`, `Color Blind` (10), `Seattle Grays` (5), `Traffic` (9), `Miller Stone` (11), `Superfishel Stone` (10), `Nuriel Stone` (9), `Jewel Bright` (9), `Summer` (8), `Winter` (10), `Green-Orange-Teal` (12), `Red-Blue-Brown` (12), `Purple-Pink-Gray` (12), `Hue Circle` (19) sequential: `Blue-Green` (7), `Blue Light` (7), `Orange Light` (7), `Blue` (20), `Orange` (20), `Green` (20), `Red` (20), `Purple` (20), `Brown` (20), `Gray` (20), `Gray Warm` (20), `Blue-Teal` (20), `Orange-Gold` (20), `Green-Gold` (20), `Red-Gold` (21) diverging (7): `Orange-Blue`, `Red-Green`, `Green-Blue`, `Red-Blue`, `Red-Black`, `Gold-Purple`, `Red-Green-Gold`, `Sunset-Sunrise`, `Orange-Blue-White`, `Red-Green-White`, `Green-Blue-White`, `Red-Blue-White`, `Red-Black-White`, `Orange-Blue Light`, `Temperature`
`tol` [*scheme*]	color schemes by Paul Tol (personal.sron.nl/~pault; using definitions from source file tol_colors.py, which may deviate from personal.sron.nl/~pault), where *scheme* is one of the following qualitative: `bright` (8), `high-contrast` (4), `vibrant` (8), `muted` (11; default), `medium-contrast` (7), `light` (10) sequential: `YlOrBr` (9), `iridescent` (23) rainbow: `rainbow` (1-23), `PuRd` (22), `PuBr` (26), `WhRd` (30), `WhBr` (34) diverging: `sunset` (11), `BuRd` (9), `PRGn` (9)
colorbrewer	color schemes from ColorBrewer (Brewer 2016; Brewer et al. 2003), where *colorbrewer* is one of the following qualitative: `Accent` (8), `Dark2` (8), `Paired` (12), `Pastel1` (9), `Pastel2` (8), `Set1` (9), `Set2` (8), `Set3` (12) sequential (3–9): `Blues`, `BuGn`, `BuPu`, `GnBu`, `Greens`, `Greys`, `OrRd`, `Oranges`, `PuBu`, `PuBuGn`, `PuRd`, `Purples`, `RdPu`, `Reds`, `YlGn`, `YlGnBu`, `YlOrBr`, `YlOrRd` diverging (3–11): `BrBG`, `PRGn`, `PiYG`, `PuOr`, `RdBu`, `RdGy`, `RdYlBu`, `RdYlGn`, `Spectral` CMYK variants: add keyword `cmyk` (e.g., type `Accent cmyk`)
`carto` [*scheme*]	color schemes from carto.com/carto-colors, where *scheme* is one of the following qualitative (3–12): `Antique`, `Bold` (default), `Pastel`, `Prism`, `Safe` (CVD-friendly), `Vivid` sequential (2–7): `Burg`, `BurgYl`, `RedOr`, `OrYel`, `Peach`, `PinkYl`, `Mint`, `BluGrn`,

DarkMint, Emrld, ag_GrnYl, BluYl, Teal, TealGrn, Purp, PurpOr, Sunset, Magenta, SunsetDark, ag_Sunset, BrwnYl
diverging (2–7): ArmyRose, Fall, Geyser, Temps, TealRose, Tropic, Earth

`ptol` [*scheme*] — color schemes from Tol (2012), where *scheme* is `qualitative` (1–12; default), `rainbow` (4–12), or `diverging` (3–11)

`lin` [*scheme*] — semantic colors from Lin et al. (2013), where *scheme* is `carcolor` (6; default), `food` (7), `features` (5), `activities` (5), `fruits` (7), `vegetables` (7), `drinks` (7), or `brands` (7) (algorithm-selected variants: add keyword `algorithm`; e.g., type `lin carcolor algorithm`)

`spmap` [*scheme*] — color schemes from `spmap` by Pisati (2007), where *scheme* is `blues` (2–99; default), `greens` (2–99), `greys` (2–99), `reds` (2–99), `rainbow` (2–99), `heat` (2–16), `terrain` (2–16), or `topological` (2–16)

`sfso` [*scheme*] — color schemes by the Bundesamt für Statistik (2017), where *scheme* is
qualitative: `parties` (11), `languages` (5)
sequential (7): `blue` (default)
sequential (6): `brown`, `orange`, `red`, `pink`, `purple`, `violet`, `ltblue`, `turquoise`, `green`, `olive`, `black`
diverging (10): `votes`
CMYK variants: add keyword `cmyk` (e.g., type `sfso blue cmyk`)

`HTML` [*scheme*] — HTML colors in groups from www.w3schools.com, where *scheme* is `pink` (6), `purple` (19), `red` (9), `orange` (5), `yellow` (11), `green` (22), `cyan` (8), `blue` (16), `brown` (18), `white` (17), or `gray` (10; alias: `grey`); all 140 HTML colors (alphabetically sorted) will be returned if *scheme* is omitted (alias: `webcolors`)

`w3` [*scheme*] — W3.CSS color schemes from www.w3schools.com, where *scheme* is as follows
qualitative collections: `default` (30 Default Colors), `flat` (20 Flat UI Colors), `metro` (17 Metro UI Colors), `win8` (22 Windows 8 Colors), `ios` (12 iOS Colors), `highway` (7 US Highway Colors), `safety` (6 US Safety Colors), `signal` (10 European Signal Colors), `2019` (32 Fashion Colors 2019), `2018` (30 Fashion Colors 2018), `2017` (20 Fashion Colors 2017), `vivid` (21 Vivid Colors), `food` (40 Food Colors), `camo` (15 Camouflage Colors), `ana` (44 Army Navy Aero Colors), `traffic` (9 Traffic Colors)
sequential themes (11): `black`, `blue`, `blue-grey`, `brown`, `cyan`, `dark-grey`, `deep-orange`, `deep-purple`, `green`, `grey`, `indigo`, `khaki`, `light-blue`, `light-green`, `lime`, `orange`, `pink`, `purple`, `red`, `teal`, `yellow`

wesanderson — Wes Anderson palettes from wesandersonpalettes.tumblr.com (source), where *scheme* is as follows
qualitative: `BottleRocket1` (7), `BottleRocket2` (5), `Rushmore1` (5), `Royal1` (4), `Royal2` (5), `Zissou1` (5), `Darjeeling1` (5), `Darjeeling2` (5), `Chevalier1` (4), `FantasticFox1` (5), `Moonrise1` (4), `Moonrise2` (4), `Moonrise3` (5), `Cavalcanti1` (5), `GrandBudapest1` (4), `GrandBudapest2` (4), `IsleofDogs1` (6), `IsleofDogs2` (5), `FrenchDispatch1` (5)
sequential: `Zissou1` (5)

The palette names can be abbreviated and typed in lowercase letters (for example, `"BuGn"` could be typed as `"bugn"`, `"lin carcolor algorithm"` could be typed as `"lin car a"`). If abbreviation is ambiguous, the first matching name in the sorted list of palettes (including all palette types) will be used. Numbers in parentheses refer to the palette size (number of colors; a range means that the palette comes in different sizes).

ColorBrewer is a set of color schemes developed by Brewer et al. (2003); also see Brewer (2016). These colors are licensed under Apache License Version 2.0; see the copyright notes at ColorBrewer_updates.html.

Colormaps

Colormaps are palettes whose colors are obtained by linear segmentation around anchor points or by linear interpolation from a dense grid of RGB values. The syntax for colormaps is

S.[`add_`]`palette(`[`"`*name*`", ` *n*`, ` *range*`]`)

where

name selects the colormap. See below for available names.

n is the number of colors to be retrieved from the colormap. The default is 15.

range `= (`*lb*[`, ` *ub*]`)` specifies the range of the colormap to be used, with *lb* and *ub* within $[0, 1]$ (values smaller than 0 or larger than 1 will be interpreted as 0 or 1, respectively). The default is `(0,1)`. This default can also be selected by typing `.` (missing). If *lb* is larger than *ub*, the colors are retrieved in reverse order. Argument *range* has not effect for cyclic (circular) colormaps.

Currently available colormaps are as follows:

viridis	perceptually uniform colormaps from matplotlib.org (also see bids.github.io/colormap), where *viridis* is as follows sequential: `viridis`, `magma`, `inferno`, `plasma`, `cividis` (CVD-friendly) cyclic: `twilight`, `twilight shifted`
seaborn	perceptually uniform colormaps from seaborn.pydata.org, where *seaborn* is as follows sequential: `rocket`, `mako`, `flare`, `crest` diverging: `vlag`, `icefire`
`matplotlib` [*map*]	further colormaps from matplotlib.org (Hunter 2007), where *map* is `autumn`, `spring`, `summer`, `winter`, `bone`, `cool`, `copper`, `coolwarm`, `hot`, `jet` (default), or `turbo`
`CET` [*map*]	perceptually uniform colormaps by Kovesi (2015), where *map* is as follows (see colorcet.com/gallery.html for an overview) linear: `L01`, `L02`, `L03`, `L04`, `L05`, `L06`, `L07`, `L08`, `L09`, `L10`, `L11`, `L12`, `L13`, `L14`, `L15`, `L16`, `L17`, `L18`, `L19`, `L20` (default) rainbow: `R1`, `R2`, `R3`, `R4` isoluminant: `I1`, `I2`, `I3` diverging: `D01`, `D01A`, `D02`, `D03`, `D04`, `D06`, `D07`, `D08`, `D09`, `D10`, `D11`, `D12`, `D13` circular: `C1`, `C2`, `C3`, `C4`, `C5`, `C6`, `C7` CVD-friendly: `CBD1`, `CBL1`, `CBL2`, `CBC1`, `CBC2`
`scico` [*map*]	perceptually uniform CVD-friendly colormaps by Crameri (2018), where *map* is as follows sequential: `batlow` (default), `batlowW`, `batlowK`, `devon`, `lajolla`, `bamako`, `davos`, `bilbao`, `nuuk`, `oslo`, `grayC`, `hawaii`, `lapaz`, `tokyo`, `buda`, `acton`, `turku`, `imola` diverging: `broc`, `cork`, `vik`, `lisbon`, `tofino`, `berlin`, `roma`, `bam`, `vanimo` cyclic: `romaO`, `bamO`, `brocO`, `corkO`, `vikO`

The names can be abbreviated; if abbreviation is ambiguous, the first matching name in the sorted list of palettes (including all palette types) will be used.

Example

```
. mata: A = B = ColrSpace()
. mata: A.palette("viridis", 30)
. mata: B.palette("magma", 30, (.5,1))
. colorpalette, gropts(ysize(1) scale(*4)) nonumbers: mata(A) / mata(B)
```

Color generators

The syntax for the color generators is

$$S.\texttt{[add\_]palette(["}name\texttt{", }n\texttt{, }H\texttt{, }C\texttt{, }L\texttt{, }P\texttt{])}$$

where

name selects the type of color generator; see below for available names.

n specifies the number of colors to be generated. The default is 15.

H is a real vector specifying one or two hues in degrees of the color wheel.

C is a real vector specifying one or two chroma levels. For `hue` only the first level is relevant.

L is a real vector specifying one or two luminance/lightness levels. For `hue` only the first level is relevant.

P is a real vector specifying one or two power parameters. For `hue` only the first parameter is relevant: $P \neq 0$ causes `hue` to travel counter-clockwise around the color wheel.

The available color generators are as follows:

`hue` HCL colors with evenly spaced hues. The algorithm has been modeled after function `hue_pal()` from R's `scales` package by Hadley Wickham (see github.com/hadley/scales). The default parameters are H `= (15, 375)`, C `= 100`, and L `= 65`. If the difference between the two values of H is a multiple of 360, the second value will be reduced by $360/n$ (so that the space between the last and the first color is the same as between the other colors).

`HCL` [*scheme*] Qualitative, diverging, or sequential colors in the HCL space (radial CIE $L^*u^*v^*$). The algorithm has been modeled after R's `colorspace` package by Ihaka et al. (2016); also see Zeileis et al. (2009) and hclwizard.org. *scheme* can be one of the following.
qualitative: `qualitative` (default), `intense`, `dark`, `light`, `pastel`
sequential: `sequential`, `blues`, `greens`, `grays`, `oranges`, `purples`, `reds`, `heat`, `heat2`, `terrain`, `terrain2`, `viridis`, `plasma`, `redblue`
diverging: `diverging`, `bluered`, `bluered2`, `bluered3`, `greenorange`, `browngreen`, `pinkgreen`, `purplegreen`

`LCh` [*scheme*] Qualitative, diverging, or sequential colors in the LCh space (radial CIE $L^*a^*b^*$). The algorithm has been modeled in analogy to `HCL`. *scheme* can be one of the following.
qualitative: `qualitative` (default), `intense`, `dark`, `light`, `pastel`
sequential: `sequential`, `blues`, `greens`, `grays`, `oranges`, `purples`, `reds`, `heat`, `heat2`, `terrain`, `terrain2`, `viridis`, `plasma`, `redblue`
diverging: `diverging`, `bluered`, `bluered2`, `bluered3`, `greenorange`, `browngreen`, `pinkgreen`, `purplegreen`

`JMh` [*scheme*] Qualitative, diverging, or sequential colors in the $J'M'h$ space. The algorithm has been modeled in analogy to `HCL`. *scheme* can be one of the following.
qualitative: `qualitative` (default), `intense`, `dark`, `light`, `pastel`
sequential: `sequential`, `blues`, `greens`, `grays`, `oranges`, `purples`, `reds`, `heat`, `heat2`, `terrain`, `terrain2`, `viridis`, `plasma`, `redblue`
diverging: `diverging`, `bluered`, `bluered2`, `bluered3`, `greenorange`, `browngreen`, `pinkgreen`, `purplegreen`

`HSV` [*scheme*] Qualitative, diverging, or sequential colors in the HSV space. The algorithm has been modeled in analogy to `HCL`. *scheme* can be one of the following.
qualitative: `qualitative` (default), `intense`, `dark`, `light`, `pastel`, `rainbow`
sequential: `sequential`, `blues`, `greens`, `grays`, `oranges`, `purples`, `reds`, `heat`, `terrain`, `heat0`, `terrain0`
diverging: `diverging`, `bluered`, `bluered2`, `bluered3`, `greenorange`, `browngreen`, `pinkgreen`, `purplegreen`

`HSL` [*scheme*] Qualitative, diverging, or sequential colors in the HSL space. The algorithm has been modeled in analogy to `HCL`. *scheme* can be one of `qualitative` (default), `sequential`, or `diverging`.

The names of the generators and schemes can be abbreviated and typed in lowercase letters. If abbreviation is ambiguous, the first matching name in the sorted list of palettes and generators (including all palette types) will be used.

Given n (number of colors), H = `(`h_1`,`h_2`)` (hues), C = `(`c_1`,`c_2`)` (chroma levels), L = `(`l_1`,`l_2`)` (luminance levels), P = `(`p_1`,`p_2`)` (power coefficients), the `HCL` generator creates HCL colors $i = 1, \ldots, n$ according to the following formulas.

- qualitative: `HCL[`i`]` = `(`$h_1 + (h_2 - h_1) \times (i-1)/(n-1)$`,` c_1`,` l_1`)`
- sequential: let $j = (n-i)/(n-1)$, then

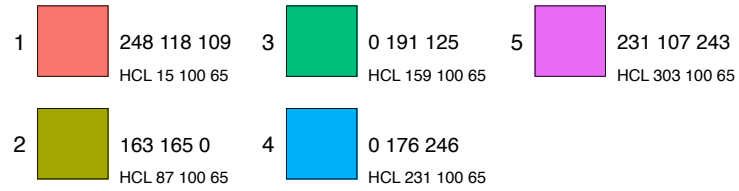
  `HCL[`i`]` = `(`$h_2 - (h_2 - h_1) \times j$`,` $c2 - (c2 - c1) \times j^{p_1}$`,` $l2 - (l2 - l1) \times j^{p_2}$`)`
- diverging: let $j = (n - 2i + 1)/(n-1)$, then

  `HCL[`i`]` = `(cond(`$j > 0$`,` h_1`,` h_2`),` $c_1 \times |j|^{p_1}$`,` $l_2 - (l_2 - l_1) \times |j|^{p_2}$`)`

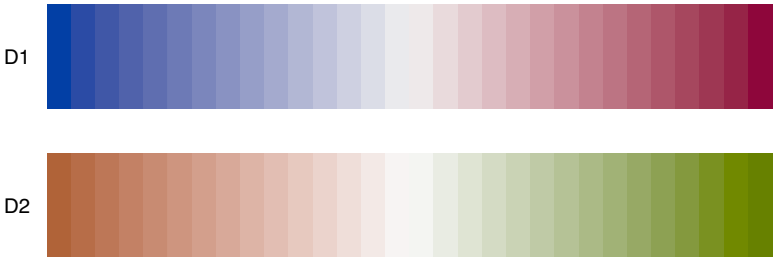
The `LCh`, `JMh`, `HSV`, and `HSL` generator use analogous formulas. For qualitative colors, if h_2 is omitted, it is set to $h_2 = h_1 + 360 \times (n-1)/n$. See file `colrspace_library_generators.sthlp` for the parameter settings of the different generators.

Examples

```
. mata: S = ColrSpace()
. mata: S.palette("hue", 5)
. colorpalette mata(S), rows(2)
```




```
. mata: D1 = ColrSpace()
. mata: D1.palette("HCL diverging", 30)
. mata: D2 = ColrSpace()
. mata: D2.palette("HCL diverging", 30, (30, 100), 70, (50, 98))
. colorpalette, gropts(ysize(2) scale(*2)) nonumbers labels(D1 D2): ///
>     mata(D1) / mata(D2)
```



6.3 Set/retrieve opacity and intensity

Set opacity

To set the opacity of the colors in S, type

$$S.\texttt{[add\_]opacity[\_added]}(\textit{opacity}\texttt{[, }\textit{noreplace}\texttt{]})$$

S.`opacity()` sets opacty for all existing colors; use S.`opacity_added()` if you only want to set opacity for the colors that have been added last. Furthermore, use S.`add_opacity()` or S.`add_opacity_added()` to leave the existing colors unchanged and append a copy of the colors with the new opacity settings. Arguments are as follows.

opacity is a real vector of opacity values as percentages in $[0, 100]$. A value of 0 makes the color fully transparent, a value of 100 makes the color fully opaque. If the number of specified opacity values is smaller than the number of existing colors, the opacity values will be recycled; if the number of opacity values is larger than the number of

colors, the colors will be recycled. To skip assigning opacity to a particular color, you may set the corresponding element in *opacity* to . (missing).

noreplace $\neq 0$ specifies that existing opacity values should not be replaced. By default, *S*.`opacity()` resets opacity for all colors irrespective of whether they already have an opacity value or not.

Alternatively, you may type

S.[`add_`]`alpha`[`_added`](*alpha*[`,` *noreplace*])

where *alpha* contains opacity values specified as proportions in $[0, 1]$.

Retrieve opacity

To retrieve a real colvector containing the opacity values (as percentages) of the colors in *S*, type

opacity `=` *S*.`opacity`[`_added`]`()`

opacity will be equal to . (missing) for colors that do not have an opacity value. *S*.`opacity()` returns the opacity values of all colors; *S*.`opacity_added()` only returns the opacity values of the colors that have been added last.

Alternatively, you may type

alpha `=` *S*.`alpha`[`_added`]`()`

to retrieve opacity values as proportions.

Set intensity

To set the intensity adjustment multipliers of the colors in *S*, type

S.[`add_`]`intensity`[`_added`](*intensity*[`,` *noreplace*])

S.`intensity()` sets the intensity multipliers for all existing colors; use *S*.`intensity_added()` if you only want to set intensity for the colors that have been added last. Furthermore, use *S*.`add_intensity()` and *S*.`add_intensity_added()` to leave the existing colors unchanged and append a copy of the colors with the new intensity settings. Arguments are as follows.

intensity is a real vector of intensity adjustment multipliers in $[0, 255]$. A multiplier smaller than 1 makes the color lighter, a multiplier larger than one make the color darker. If the number of specified intensity multipliers is smaller than the number of existing colors, the intensity multipliers will be recycled; if the number of intensity multipliers is larger than the number of colors, the colors will be recycled. To skip

assigning an intensity multiplier to a particular color, you may set the corresponding element in *intensity* to . (missing).

noreplace $\neq 0$ specifies that existing intensity adjustment multipliers should not be replaced. By default, *S*.intensity() resets the intensity multipliers for all colors irrespective of whether they already have an intensity multiplier or not.

Note that *S*.intensity() does not manipulate the stored coordinates of a color, it just adds an extra piece of information. This extra information, the intensity multiplier, is added to a color specification when exporting the colors using *S*.colors(). If you want to actually transform the stored color values instead of just recording an intensity multiplier, you can use function *S*.intensify().

Retrieve intensity

To retrieve a real colvector containing the intensity adjustment multipliers of the colors in *S*, type

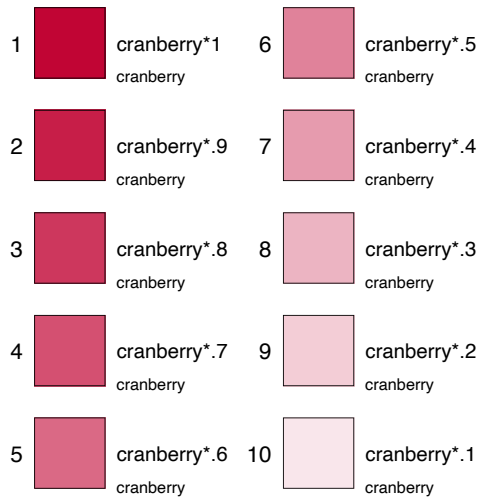
intensity = *S*.intensity[_added]()

intensity will be equal to . (missing) for colors that do not have an intensity multiplier. *S*.intensity() returns the intensity multipliers of all colors; *S*.intensity_added() only returns the intensity multipliers of the colors that have been added last.

Examples

```
: S = ColrSpace()
: S.palette("s2", 4)
: S.opacity((., 80, ., 60))
: S.intensity((.7, ., .8, .))
: S.Colors()
                    1
  ┌──────────────────┐
1 │          navy*.7 │
2 │         maroon%80 │
3 │  forest_green*.8 │
4 │      dkorange%60 │
  └──────────────────┘
```

```
. mata: S = ColrSpace()
. mata: S.colors("cranberry")
. mata: S.intensity(range(1,.1,.10))
. colorpalette mata(S)
```



6.4 Recycle, select, and order colors

Recycle colors

To recycle the colors in S, type

$$S.\left[\texttt{add\_}\right]\texttt{recycle}\left[\texttt{\_added}\right](n)$$

where n is a real scalar specifying the number of desired colors. S.`recycle()` will create n colors by recycling the colors until the desired number of colors is reached. If n is smaller than the number of existing colors, S.`recycle()` will select the first n colors. S.`recycle()` operates on all existing colors; use S.`recycle_added()` if you only want to recycle the colors added last. Furthermore, use S.`add_recycle()` or S.`add_recycle_added()` to leave the existing colors unchanged and append the recycled colors.

Example

```
: S = ColrSpace()
: S.colors("black red yellow")
: S.recycle(7)
: S.colors()
  black red yellow black red yellow black
: S.recycle(2)
: S.colors()
  black red
```

Select colors

To select (and order) colors in S, type

$S.$[add_]select[_added](p)

where p is a real vector of the positions of the colors to be selected (permutation vector). Positive numbers refer to colors from the start; negative numbers refer to colors from the end (numbers out of range will ignored). Colors not covered in p will be dropped and the selected colors will be ordered as specified in p. S.select() operates on all existing colors; use S.select_added() if you only want to manipulate the colors added last. Furthermore, use S.add_select() or S.add_select_added() to leave the existing colors unchanged and append the selected colors.

Example

```
: S = ColrSpace()
: S.colors("black red yellow blue green")
: S.select((4,3,4))
: S.colors()
  blue yellow blue
```

Drop colors

To drop individual colors in S (without changing the order of the remaining colors), type

$S.$[add_]drop[_added](p)

where p is a real vector of the positions of the colors to be dropped (permutation vector). Positive numbers refer to colors from the start; negative numbers refer to colors from the end (numbers out of range will ignored). S.drop() operates on all existing colors; use S.drop_added() if you only want to manipulate the colors added last. Furthermore, use S.add_drop() or S.add_drop_added() to leave the existing colors unchanged and append a copy of the colors that have not been dropped.

Example

```
: S = ColrSpace()
: S.colors("black red yellow blue green")
: S.drop(-2)     // (drop second last)
: S.colors()
  black red yellow green
```

Reorder colors

To order the colors in S, type

$S.$[add_]order[_added](*p*)

where p is a real vector specifying the desired order of the colors (permutation vector). Positive numbers refer to colors from the start; negative numbers refer to colors from the end (numbers out of range will ignored). Colors not covered in p will be placed last, in their original order. S.order() operates on all existing colors; use S.order_added() if you only want to manipulate the colors added last. Furthermore, use S.add_order() or S.add_order_added() to leave the existing colors unchanged and append the reordered colors.

Example

```
: S = ColrSpace()
: S.colors("black red yellow blue green")
: S.order((4,3,4))
: S.colors()
  blue yellow blue black red green
```

Reverse the order of colors

To reverse the order of the colors in S, type:

$S.$[add_]reverse[_added]()

S.reverse() operates on all existing colors; use S.reverse_added() if you only want to manipulate the colors added last. Furthermore, use S.add_reverse() or S.add_reverse_added() to leave the existing colors unchanged and append the reversed colors. S.reverse() is equivalent to S.order(S.N()::1) or S.select(S.N()::1).

Example

```
: S = ColrSpace()
: S.colors("black red yellow blue green")
: S.reverse()
: S.colors()
  green blue yellow red black
```

Shift colors

To shift the positions of colors up or down, wrapping positions around at the top and bottom, type

$S.$[add_]shift[_added](*k*)

where k specifies the size of the shift. If k is in $(-1, 1)$, the colors are shifted by trunc($k \times n$) positions, where n is the total number of colors (proportional shift); if $|k| \geq 1$, the colors are shifted by trunc(k) positions. Specify $k > 0$ ($k < 0$) for a

shift in upward (downward) direction. *S*.`shift()` operates on all existing colors; use *S*.`shift_added()` if you only want to manipulate the colors added last. Furthermore, use *S*.`add_shift()` or *S*.`add_shift_added()` to leave the existing colors unchanged and append the shifted colors.

Example

```
: S = ColrSpace()
: S.colors("black red yellow blue green")
: S.shift(2)
: S.colors()
  blue green black red yellow
```

6.5 Interpolate and mix

Interpolation

To apply linear interpolation to the colors in *S*, type:

S.[`add_`]`ipolate`[`_added`](*n*[, *space*, *range*, *power*, *positions*, *padded*])

Opacity values and intensity adjustment multipliers, if existing, will also be interpolated. *S*.`ipolate()` takes all existing colors as input and replaces them with the interpolated colors; use *S*.`ipolate_added()` if you only want to interpolate the colors added last. Furthermore, use *S*.`add_ipolate()` or *S*.`add_ipolate_added()` to leave the existing colors unchanged and append the interpolated colors. Arguments are as follows.

n is a real scalar specifying the number of destination colors. *S*.`ipolate()` will interpolate the existing (origin) colors to *n* new colors (thus increasing or decreasing the number of colors, depending on whether *n* is larger or smaller than the number of origin colors).

space selects the color space in which the colors are interpolated. *space* can be `"RGB"`, `"lRGB"`, `"HSV"`, `"HSL"`, `"CMYK"`, `"XYZ"`, `"xyY"`, `"Lab"`, `"LCh"`, `"Luv"`, `"HCL"`, `"CAM02` [*mask*]`"`, `"JMh` [*coefs*]`"`, or `"Jab` [*coefs*]`"` (lowercase spelling allowed). The default is `"Jab"`. This default can also be selected by typing `""`. When interpolating from one hue to the next (relevant for `"HSV"`, `"HSL"`, `"LCh"`, `"HCL"`, `"JMh"`, and `"CAM02"` when *mask* contains `h`), *S*.`ipolate()` will travel around the color wheel in the direction in which the two hues are closer to each other (based on the original order of colors in *S*; the rule may be violated if colors are reordered through argument *positions*).

range = (*lb*[, *ub*]) specifies range of the destination colors. The default is `(0,1)`. This default can also be selected by typing `.` (missing). If *lb* is larger than *ub*, the destination colors will be arranged in reverse order. Extrapolation will be applied if the specified range exceeds $[0, 1]$.

power is a real scalar affecting the distribution of the destination colors across *range*.

The default is to distribute them evenly. This default can also be selected by typing . (missing) or setting *power* to 1. A *power* value larger than 1 squishes the positions towards *lb*. If interpolating between two colors, this means that the first color will dominate most of the interpolation range (slow to fast transition). A value between 0 and 1 squishes the positions towards *ub*, thus making the second color the dominant color for most of the range (fast to slow transition). Another way to think of the effect of *power* is that it moves the center of the color gradient up (if *power* is larger than 1) or down (if *power* is between 0 and 1).

positions is a real vector specifying the positions of the origin colors. The default is to place them on a regular grid from 0 and 1. This default can also be selected by typing . (missing). If *positions* has less elements than there are colors, default positions are used for the remaining colors. If the same position is specified for multiple colors, these colors will be averaged before applying interpolation.

padded $\neq 0$ requests padded interpolation. By default, if *padded* is omitted or equal to 0, the first color and the last color are taken as the end points of the interpolation range; these colors thus remain unchanged (as long as default settings are used for *range* and *position*). If *padded* $\neq 0$, the positions of the colors are interpreted as interval midpoints, such that the interpolation range is padded by half an interval on each side. This causes the destination colors to be spread out slightly more (less) than the origin colors, if the number of destination colors is larger (smaller) than the number of origin colors.

Circular interpolation is used if *S*.`pclass()` is equal to `"circular"` or `"cyclic"`. In this case, arguments *range*, *power*, *positions*, and *padded* will be ignored. For regular (noncircular) interpolation, which is applied if *S*.`pclass()` is different from `"circular"` or `"cyclic"`, these arguments can be used to fine-tune the interpolation.

Examples

```
. mata: Jab = ColrSpace()
. mata: Jab.colors("#337ab7 #f0ad4e")
. mata: JMh = J(1, 1, Jab)    // (make copy)
. mata: Jab.ipolate(30)
. mata: JMh.ipolate(30, "JMh")
. colorpalette, gropts(ysize(1) scale(*4)) nonumbers: ///
>     mata(Jab) / mata(JMh)
```


```
. mata: A = ColrSpace()
. mata: A.colors("#fafa6e #2A4858")
. mata: B = C = D = J(1, 1, A)            // (make copies)
. mata: A.ipolate(30, "HCL")
```

```
. mata: B.ipolate(30, "HCL", (.1,.9))     // (select range)
. mata: C.ipolate(30, "HCL", ., 1.5)      // (make 1st color dominant)
. mata: D.ipolate(30, "HCL", ., .6)       // (make 2nd color dominant)
. colorpalette, gropts(ysize(2) scale(*2)) nonumbers: ///
>     mata(A) / mata(B) / mata(C) / mata(D)
```


```
. mata: A = ColrSpace()
. mata: A.colors("black red yellow")
. mata: B = C = J(1, 1, A)
. mata: A.ipolate(30)                             // (red in middle)
. mata: B.ipolate(30, "", ., ., (0, .3, 1))       // (shift left)
. mata: C.ipolate(30, "", ., ., (0, .7, 1))       // (shift right)
. colorpalette, gropts(ysize(1.5) scale(*2.67)) nonumbers: ///
>     mata(A) / mata(B) / mata(C)
```

Mixing

To mix (i.e. average) the colors in S, type:

$$S.\texttt{[add\_]mix[\_added]([}\mathit{space}\texttt{, }w\texttt{])}$$

Opacity values and intensity adjustment multipliers, if defined, will also be mixed (i.e. averaged). S`.mix()` takes all existing colors as input and replaces them with the mixed color; use S`.mix_added()` if you only want to mix the colors added last. Furthermore, use S`.add_mix()` or S`.add_mix_added()` to leave the existing colors unchanged and append the mixed color. Arguments are as follows.

space selects the color space in which the colors are mixed. *space* can be `"RGB"`,

"lRGB", "HSV", "HSL", "CMYK", "XYZ", "xyY", "Lab", "LCh", "Luv," "HCL", "CAM02 [*mask*]", "JMh [*coefs*]", or "Jab [*coefs*]" (lowercase spelling allowed). The default is "Jab". This default can also be selected by typing "". When mixing hues (relevant for "HSV", "HSL", "LCh", "HCL", "JMh", and "CAM02" when *mask* contains h), *S*.mix() will compute the mean of angles as described at en.wikipedia.org/wiki/Mean_of_circular_quantities (using weighted sums of the cartesian coordinates if weights are specified); this is slightly different from the procedure employed by *S*.ipolate().

w is a real vector containing weights. Color mixing works by transforming the colors to the selected color space, taking the means of the attributes across colors, and then transforming the resulting "average" color back to the original space. *w* specifies the weights given to the individual colors when computing the means. If *w* contains less elements than there are colors, the weights will be recycled. Omit *w*, or specify *w* as 1 or as . (missing) to use unweighted means.

Examples

```
: S = ColrSpace()
: S.colors("black red yellow")
: S.get()
        1     2     3
  ------------------------
1 |     0     0     0
2 |   255     0     0
3 |   255   255     0
  ------------------------

: S.mix("lRGB")
: S.get()
        1     2     3
  ------------------------
1 |   213   156     0
  ------------------------

: S.colors("black red yellow")
: S.mix("lRGB", (.5, 1, 1))
: S.get()
        1     2     3
  ------------------------
1 |   231   170     0
  ------------------------
```

6.6 Intensify, saturate, luminate

Intensify

To adjust the intensity of the colors in *S*, type

S.[add_]intensify[_added](*m*)

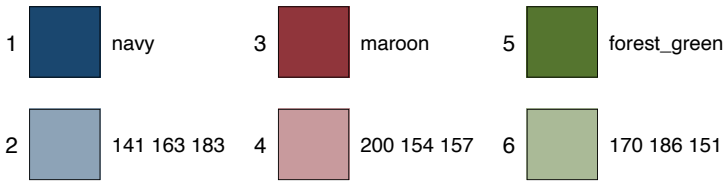
where *m* is a real vector of intensity adjustment multipliers in $[0, 255]$. A multiplier

smaller than 1 makes the color lighter, a multiplier larger than one make the color darker. If the number of specified multipliers is smaller than the number of colors, the multipliers will be recycled; if the number of multipliers is larger than the number of colors, the colors will be recycled. To skip adjusting the intensity of a particular color, you may set the corresponding multiplier to . (missing). *S*.`intensify()` operates on all existing colors; use *S*.`intensify_added()` if you only want to manipulate the colors added last. Furthermore, use *S*.`add_intensify()` or *S*.`add_intensify_added()` to leave the existing colors unchanged and append the manipulated colors.

`ColrSpace` uses the same algorithm as is used in official Stata to adjust the color intensity. Applying *S*.`intensify()` thus results in colors that look the same as colors that have been specified using intensity multiplier syntax (see help [G] ***colorstyle***). The algorithm works by increasing or decreasing the RGB values proportionally, with rounding to the nearest integer and adjustment to keep all values within $[0, 255]$.

Example

```
. mata: S = ColrSpace()
. mata: S.colors("navy maroon forest_green")
. mata: S.select((1,1,2,2,3,3))       // duplicate colors
. mata: S.intensify((., .5))
. colorpalette mata(S), rows(2)
```



Saturate

To change the saturation (colorfulness) of the colors in *S*, type:

S.[`add_`]`saturate`[`_added`](*d*[`,` *method*`,` *level*])

S.`saturate()` operates on all existing colors; use *S*.`saturate_added()` if you only want to manipulate the colors added last. Furthermore, use *S*.`add_saturate()` or *S*.`add_saturate_added()` to leave the existing colors unchanged and append the manipulated colors. Arguments are as follows.

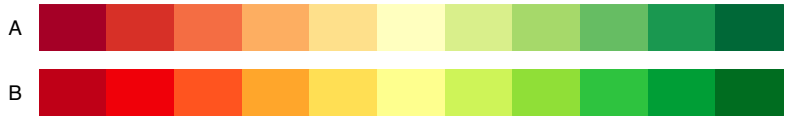
d is a real vector of saturation adjustments addends. Positive values increase saturation, negative values decrease saturation. If the number of specified addends is smaller than the number of colors, the addends will be recycled; if the number of addends is larger than the number of colors, the colors will be recycled. To skip adjusting a particular color, you may set the corresponding addend to . (missing). Typically, reasonable addends are in a range of about ± 50.

method selects the color space in which the colors are manipulated. It can be `"LCh"`, `"HCL"`, `"JCh"` (shorthand for `CAM02 JCh`), or `"JMh"` (lowercase spelling allowed). The default is `"LCh"`. This default can also be selected by typing `""`. *S*`.saturate()` works by converting the colors to the selected color space, adding d to the C channel (or M' in case of $J'M'h$), and then converting the colors back (after resetting negative chroma values to zero).

level $\neq 0$ specifies that d provides chroma levels, not addends. In this case, the C channel will be set to d. Reasonable values typically lie in a range of 0–100, although higher values are possible. Negative values will be reset to 0.

Example

```
. mata: A = ColrSpace()
. mata: A.palette("RdYlGn")
. mata: B = J(1, 1, A)    // make copy of A
. mata: B.saturate(25)
. colorpalette, gropts(ysize(1) scale(*4)) nonumbers labels(A B): ///
>     mata(A) / mata(B)
```



S`.saturate()` has been inspired by the `saturate()` and `desaturate()` functions in Gregor Aisch's chroma.js.

Luminate

To change the luminance of the colors in S, type

S.[`add_`]`luminate`[`_added`](*d*[`,` *method*`,` *level*])

S`.luminate()` operates on all existing colors; use *S*`.luminate_added()` if you only want to manipulate the colors added last. Furthermore, use *S*`.add_luminate()` or *S*`.add_luminate_added()` to leave the existing colors unchanged and append the manipulated colors. Arguments are as follows.

d is a real vector of luminance adjustments addends. Positive values increase luminance, negative values decrease luminance. If the number of specified addends is smaller than the number of colors, the addends will be recycled; if the number of addends is larger than the number of colors, the colors will be recycled. To skip adjusting a particular color, you may set the corresponding addend to `.` (missing). Typically, reasonable addends are in a range of about ± 50.

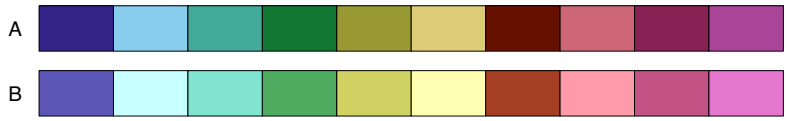
method selects the color space in which the colors are manipulated. It can be `"Lab"`, `"LCh"`, `"Luv"`, `"HCL"`, `"JCh"` (shorthand for `CAM02 JCh`), `"JMh"` or `"Jab"` (lowercase

spelling allowed). The default is `"JMh"`. This default can also be selected by typing `""`. *S*`.luminate()` works by converting the colors to the selected color space, adding *d* to the L channel (or J in case of CIECAM02 JCh, J' in case of $J'M'h$ or $J'a'b'$), and then converting the colors back (after resetting negative luminance values to zero). Results will be identical between `"Lab"` and `"LCh"`, between `"Luv"` and `"HCL"`, and between `"JMh"` and `"Jab"`.

level $\neq 0$ specifies that *d* provides luminance levels, not addends. In this case, the L channel will be set to *d*. Reasonable values typically lie in a range of 0–100. Negative values will be reset to 0.

Example

```
. mata: A = ColrSpace()
. mata: A.palette("ptol", 10)
. mata: B = J(1, 1, A)    // make copy of A
. mata: B.luminate(20)
. colorpalette, lc(black) gropts(ysize(1) scale(*4)) nonumbers labels(A B): ///
>     mata(A) / mata(B)
```



S`.luminate()` has been inspired by the `darken()` and `brighten()` functions in Gregor Aisch's chroma.js.

6.7 Grayscale conversion

To convert the colors in *S* to gray, type

S.[`add_`]`gray`[`_added`]`(`[*p*`,` *method*]`)`

S`.gray()` transforms all existing colors; use *S*`.gray_added()` if you only want to transform the colors added last. Furthermore, use *S*`.add_gray()` or *S*`.add_gray_added()` to leave the existing colors unchanged and append the transformed colors. Arguments are as follows.

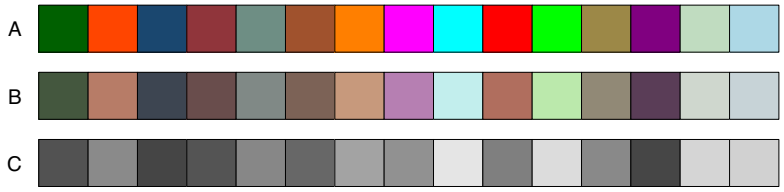
p is a real vector of proportions of gray, with *p* in $[0, 1]$. The default is $p = 1$ (complete conversion to gray). If the number of specified proportions is smaller than the number of colors, the proportions will be recycled; if the number of proportions is larger than the number of colors, the colors will be recycled. To skip converting a particular color, you may set the corresponding proportion to `.` (missing).

method specifies the color space in which the colors are manipulated. It can be `"LCh"`, `"HCL"`, `"JCh"` (shorthand for `CAM02 JCh`), or `"JMh"` (lowercase spelling allowed).

The default is "LCh". This default can also be selected by typing "". Grayscale conversion works by converting the colors the selected color space, reducing the C channel (or M' in case of $J'M'h$) towards zero, and then converting the colors back.

Example

```
. mata: A = ColrSpace()
. mata: A.palette("s1")
. mata: B = C = J(1, 1, A)    // make copies
. mata: B.gray(.7)
. mata: C.gray()
. colorpalette, lc(black) gropts(ysize(1.5) scale(*2.67)) nonumbers ///
>     labels(A B C): mata(A) / mata(B) / mata(C)
```



Grayscale conversion is also supported by function S.convert().

6.8 Color vision deficiency simulation

To convert the colors in S such that they look how they would appear to people suffering from color vision deficiency (color blindness), type

S.[add_]cvd[_added]([p, *method*])

S.cvd() transforms all existing colors; use S.cvd_added() if you only want to transform the colors added last. Furthermore, use S.add_cvd() or S.add_cvd_added() to leave the existing colors unchanged and append the transformed colors. Arguments are as follows.

p is a real vector of deficiency severities, with p in $[0, 1]$. The default is $p = 1$ (maximum severity, i.e. deuteranopia, protanopia, or tritanopia, respectively). If the number of specified severities is smaller than the number of colors, the severities will be recycled; if the number of severities is larger than the number of colors, the colors will be recycled. To skip converting a particular color, you may set the corresponding severity to . (missing).

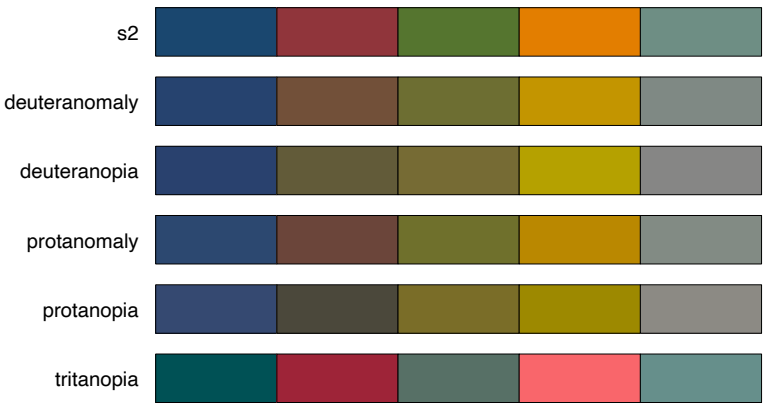
method specifies the type of color vision deficiency. It can be "deuteranomaly", "protanomaly", or "tritanomaly" (abbreviations allowed). The default is "deuteranomaly". This default can also be selected by typing "". See en.wikipedia.org/wiki/Color_blindness for basic information on the different types of color blindness.

`ColrSpace` implements color vision deficiency simulation based on Machado et al. (2009), using the transformation matrices provided at www.inf.ufrgs.br/~oliveira (employing linear interpolation between matrices for intermediate severity values). The transformations matrix for a specific combination of (scalar) p and *method* can be retrieved as follows:

M = S.cvd_M([p, *method*])

Example

```
. mata: A = ColrSpace()
. mata: A.palette("s2", 5)
. mata: d = D = p = P = T = J(1, 1, A)    // make copies
. mata: d.cvd(.5);      d.name("deuteranomaly")
. mata: D.cvd();        D.name("deuteranopia")
. mata: p.cvd(.5, "p"); p.name("protanomaly")
. mata: P.cvd(1, "p");  P.name("protanopia")
. mata: T.cvd(1, "t");  T.name("tritanopia")
. colorpalette, lc(black) gropts(ysize(3) scale(*1.33)) nonumbers: m(A) / m(d) /
>  m(D) / m(p) / m(P) / m(T)
```



Color vision deficiency simulation is also supported by function S.convert().

6.9 Color differences and contrast ratios

Color differences

To compute differences between colors in S, type

D = S.delta[_added]([P, "*method*", *noclip*])

where P is a $r \times 2$ matrix with each row selecting two colors to be compared. For example, P `= (3,5)` would compare the 3rd and the 5th color; P `= (1,2) \ (3,5)` would make two comparisons: 1st to 2nd and 3rd to 5th. The default, if P is omitted, is to make $n-1$ consecutive comparisons, where n is the number of existing colors: 1st to 2nd, 2nd to 3rd, ..., $(n-1)$th to nth; this is equivalent to P `= ((1::`S`.N()-1), (2::`S`.N()))`. This default can also be selected by typing `.` (missing). S`.delta()` operates on all existing colors, that is, P selects among all colors; in S`.delta_added()` P only selects among the colors added last. Further options are as follows.

method selects the method used to compute the color differences. It can be one of the following (lowercase spelling and abbreviations allowed):

`Jab` [*coefs*]	compute the differences from the perceptually uniform CIECAM $J'a'b'$ space as described by Luo and Li (2013, chapter 2.6.1)
`E76`	1976 CIELAB Delta E definition (euclidean distance in the CIE $L^*a^*b^*$ color space)
`E94`	1994 CIELAB Delta E definition (based on the description by Lindbloom 2017b, but using a modification to make the differences symmetric as suggested by Hunt 2004, 670)
`E2000`	2000 CIELAB Delta E definition (based on the description given by Lindbloom 2017c)
space	compute the differences as euclidean distances in the respective color space, where *space* may be `RGB`, `RGB1`, `lRGB`, `XYZ`, `XYZ1`, `xyY1`, `Lab`, `LCh`, `Luv`, `HCL`, `JCh` (shorthand for `CAM02 JCh`), or `JMh` [*coefs*]

The default is `Jab`. This default can also be selected by typing `""`. For background information on color difference also see en.wikipedia.org/wiki/Color_difference.

noclip $\neq 0$ prevents converting the colors to valid RGB values before computing the differences. By default, S`.delta()` translates the colors to linear RGB and clips the coordinates at 0 and 1, before converting the colors to the color space selected by *method*, so that the computed differences are consistent with how the colors are perceived on an RGB device. Specify *noclip* $\neq 0$ to skip this extra step.

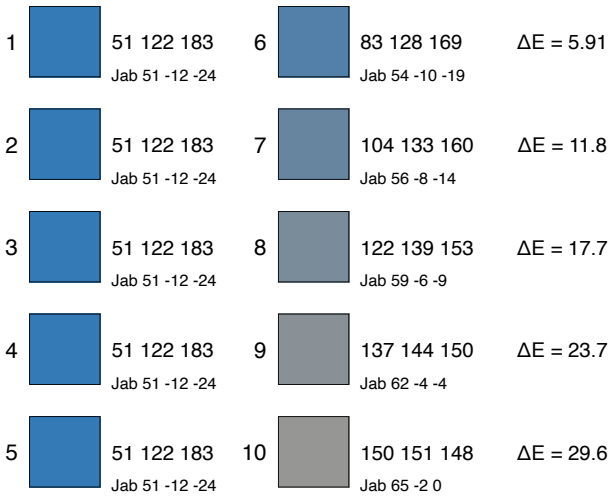
Opacity settings and intensity adjustment multipliers are ignored when computing the color differences.

Example

```
: S = ColrSpace()
: S.colors("#337ab7 #f0ad4e")
: S.ipolate(6, "", (0, .5))
: S.delta((J(5,1,1), (2::6)))     // compare 1st to other colors
                  1
    ┌──────────────┐
  1 │   5.91377881 │
  2 │  11.82755762 │
  3 │  17.74133643 │
  4 │  23.65511524 │
  5 │  29.56889405 │
    └──────────────┘
```

Graphical illustration:

```
. mata: D = S.delta((J(5,1,1), (2::6)))
. mata: D = `"""´ :+ "{&Delta}E = " :+ strofreal(D,"%9.3g") :+ `"""´
. mata: D = strofreal(1::5) :+ " 3 " :+  D
. mata: st_local("D", invtokens(D´))
. colorpalette mata(S), order(1 1 1 1 1) gropts(text(`D´))
```



Contrast ratios

To compute contrast ratios between colors in S, type

$$R \texttt{ = } S\texttt{.contrast}\big[\texttt{\_added}\big]\texttt{(}\big[P\big]\texttt{)}$$

where P is a $r \times 2$ matrix with each row selecting two colors to be compared. For example, P `= (3,5)` would compare the 3rd and the 5th color; P `= (1,2) \ (3,5)` would make two comparisons: 1st to 2nd and 3rd to 5th. The default, if P is omitted, is to make $n-1$ consecutive comparisons, where n is the number of existing colors: 1st to 2nd, 2nd to 3rd, ..., $(n-1)$th to nth; this is equivalent to P `= ((1::`S`.N()-1), (2::`S`.N()))`. This default can also be selected by typing `.` (missing). S`.contrast()` operates on all existing colors, that is, P selects among all colors; in S`.contrast_added()` P only selects among the colors added last.

The contrast ratios are computed according to the Web Content Accessibility Guidelines (WCAG) 2.0 at www.w3.org. Let Y_0 be the Y attribute of the lighter color, and Y_1 be the Y attribute of the darker color, in CIE XYZ space (in $Y_{\text{white}} = 100$ scaling). The contrast ratio is then defined as $(Y_0 + 5)/(Y_1 + 5)$. Typically, a contrast ratio of at

least 4.5 is recommended between foreground text and background fill.

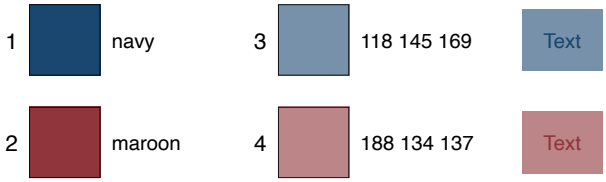
Opacity settings and intensity adjustment multipliers are ignored when computing the contrast ratios.

Example

Say, you want to print text inside bars and want the text and the bar fill to have the same basic color. One idea is to use colors with reduced intensity for the fill and print the text in the original color. *S*.`contrast()` may be helpful for finding out by how much you need to reduce intensity so that there is enough contrast between text and bar fill.

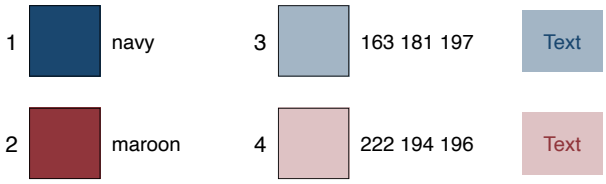
```
. mata: S = ColrSpace()
. mata: S.colors("navy maroon")
. mata: S.add_intensify(.6)
. mata: S.contrast((1,3) \ (2,4))     // not enough contrast
                 1
  1   2.939889559
  2   2.504861745

. mata: C = S.Colors()
. mata: t = `" 3 "Text", c(%s) box m(medium) bc(%s)"´
. mata: st_local("t1", sprintf("1"+t, C[1], C[3]))
. mata: st_local("t2", sprintf("2"+t, C[2], C[4]))
. colorpalette mata(S), gropts(text(`t1´) text(`t2´)) rows(2)
```




```
. mata: S.select((1,2))
. mata: S.add_intensify((.4,.3))
. mata: S.contrast((1,3) \ (2,4))     // contrast now ok
                 1
  1   4.586021616
  2   4.574782323

. mata: C = S.Colors()
. mata: t = `" 3 "Text", c(%s) box m(medium) bc(%s)"´
. mata: st_local("t1", sprintf("1"+t, C[1], C[3]))
. mata: st_local("t2", sprintf("2"+t, C[2], C[4]))
. colorpalette mata(S), gropts(text(`t1´) text(`t2´)) rows(2)
```



6.10 Import/export colors in various spaces

Import colors

An alternative to *S*.`colors()` is to import colors into *S* using the following functions:

S.`set(`*C*[`,` *space*]`)`
S.`add(`*C*[`,` *space*]`)`
S.`reset`[`_added`]`(`*C*[`,` *space*`,` *p*]`)`

S.`set()` replaces preexisting colors by the new colors; use *S*.`add()` if you want to append the new colors to the existing colors. *S*.`reset()` can be used to reset the values of colors, without reinitializing opacity and intensity adjustment; *S*.`reset_added()` is like *S*.`reset()` but only operates on the colors that have been added last. The arguments are as follows.

C provides the color values. If *space* is equal to `"HEX"`, *C* is a string vector of length *n* containing *n* hex RGB values; if *space* is equal to `"CMYK"`, `"CMYK1"`, `"RGBA"`, or `"RGBA1"`, *C* is a $n \times 4$ real matrix; if *space* is equal to `"CAM02`[*mask*]`"`, *C* is a $n \times$ `strlen(`*mask*`)` real matrix; in all other cases, *C* is a $n \times 3$ real matrix of *n* color values in the respective space. For *S*.`reset()` the number of colors in *C* must match the length of *p*.

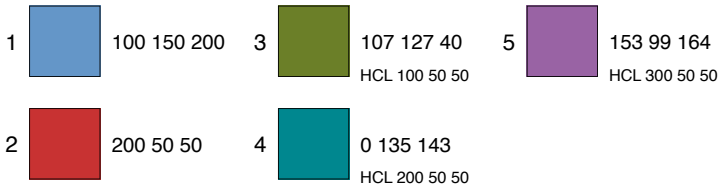
space is a string scalar specifying the color space of *C*. It can be `"HEX"`, `"RGB"`, `"RGB1"`, `"lRGB"`, `"HSV"`, `"HSL"`, `"CMYK"`, `"CMYK1"`, `"XYZ"`, `"XYZ1"`, `"xyY"`, `"xyY1"`, `"Lab"`, `"LCh"`, `"Luv"`, `"HCL"`, `"CAM02`[*mask*]`"`, `"JMh`[*coefs*]`"`, `"Jab`[*coefs*]`"`, `"RGBA"`, or `"RGBA1"` (lowercase spelling allowed). The default is `"RGB"`. This default can also be selected by typing `""`.

p is a real vector of the positions of the colors to be modified. Positive numbers refer to colors from the start; negative numbers refer to colors from the end. *S*.`reset()` aborts with error if *p* addresses positions that do not exists. If *p* is omitted, the default is to modify all colors. This default can also be selected by typing `.` (missing).

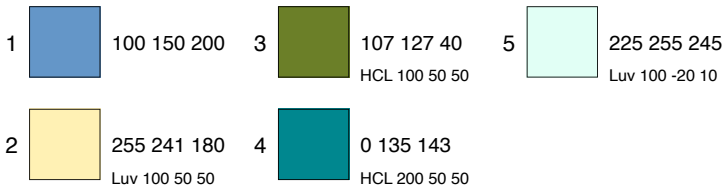
Example

```
. mata: S = ColrSpace()
. mata: S.set((100,150,200) \ (200,50,50), "RGB")
. mata: S.add((100,50,50) \ (200,50,50) \ (300,50,50), "HCL")
```

```
. colorpalette mata(S), rows(2)
```




```
. mata: S.reset((100,50,50) \ (100,-20,10), "Luv", (2,-1))
. colorpalette mata(S), rows(2)
```



Export colors

To retrieve the colors from S in a particular color space, type

$$C \texttt{ = } S\texttt{.get}\big[\texttt{\_added}\big]\texttt{(}\big[\texttt{, } \textit{space}\big]\texttt{)}$$

where *space* is a string scalar specifying the color space. It can be `"HEX"`, `"RGB"`, `"RGB1"`, `"lRGB"`, `"HSV"`, `"HSL"`, `"CMYK"`, `"CMYK1"`, `"XYZ"`, `"XYZ1"`, `"xyY"`, `"xyY1"`, `"Lab"`, `"LCh"`, `"Luv"`, `"HCL"`, `"CAM02` $[$*mask*$]$`"`, `"JMh` $[$*coefs*$]$`"`, `"Jab` $[$*coefs*$]$`"`, `"RGBA"`, or `"RGBA1"` (lowercase spelling allowed). The default is `"RGB"`. This default can also be selected by typing `""`. S`.get()` returns all colors; S`.get_added()` only returns the colors that have been added last.

Example

```
: S = ColrSpace()
: S.palette("s2", 5)
: S.Colors()
                1
  +--------------+
1 |         navy |
2 |       maroon |
3 | forest_green |
4 |     dkorange |
5 |         teal |
  +--------------+

: S.get()
        1     2     3
  +-------------------+
1 |   26    71   111  |
  +-------------------+
```

```
  2 |  144    53    59
  3 |   85   117    47
  4 |  227   126     0
  5 |  110   142   132
```

```
: S.get("RGB1")
                1             2             3
  1 |  .1019607843   .2784313725   .4352941176
  2 |  .5647058824   .2078431373    .231372549
  3 |  .3333333333   .4588235294   .1843137255
  4 |  .8901960784   .4941176471             0
  5 |   .431372549   .5568627451   .5176470588
```

```
: S.get("CAM02 QsH")
                1             2             3
  1 |  55.65919632   69.52529473   303.5288896
  2 |  65.37940215   72.51359985   399.3754444
  3 |  72.56743268   62.83103583   157.0976797
  4 |  91.67834891   71.71306279   56.76191955
  5 |  83.33062233   39.46129676   221.8105571
```

```
: S.opacity((100,90,80,70,60))
: S.get("RGBa")
       1     2     3     4
  1 |  26    71   111     1
  2 | 144    53    59    .9
  3 |  85   117    47    .8
  4 | 227   126     0    .7
  5 | 110   142   132    .6
```

6.11 Color converter and other utilities

Convert colors without storing

Instead of storing colors in S using S.`set()` and then retrieving the colors in a particular space using function S.`get()`, colors can also be converted directly from from one space to another using the S.`convert()` function. S.`convert()` will not store any colors or otherwise manipulate the content of S. The syntax is:

$$C \texttt{ = } S\texttt{.convert(}C_0\texttt{, } \textit{from}\texttt{, } \textit{to}\texttt{)}$$

where C_0 is a matrix of input colors values in color space *from*, and *to* is a destination color space. *from* and *to* can be `"HEX"`, `"RGB"`, `"RGB1"`, `"lRGB"`, `"HSV"`, `"HSL"`, `"CMYK"`, `"CMYK1"`, `"XYZ"`, `"XYZ1"`, `"xyY"`, `"xyY1"`, `"Lab"`, `"LCh"`, `"Luv"`, `"HCL"`, `"CAM02`$\left[mask\right]$`"`, `"JMh`$\left[coefs\right]$`"`, or `"Jab`$\left[coefs\right]$`"` (lowercase spelling allowed). The default is `"RGB"`. This default can also be selected by typing `""`. If *from* is `"HEX"`, C_0 is a string vector containing n hex colors. In all other cases, C_0 is a $n \times c$ real matrix of n color values in the respective coding scheme. See the diagram in Figure 1 for the paths along which

the colors will be translated.

Example

```
: S = ColrSpace()
: RGB = (25, 70, 120) \ (150, 60, 60)
: S.convert(RGB, "RGB", "xyY")
                  1             2             3
  1    .195937035   .1947051783   5.942364968
  2   .4925235637   .3295598015   10.04388472

: S.convert(RGB, "RGB", "JMh")
                  1             2             3
  1   30.26849693   22.86056577   249.2738221
  2   42.09699618   24.98595382   22.81761075

: Jab = S.convert(RGB, "RGB", "Jab")
: S.convert(Jab, "Jab", "HSV")
                  1             2             3
  1   211.5789474   .7916666667   .4705882353
  2           360            .6   .5882352941

: HCL = S.convert(Jab, "Jab", "HCL")
: S.convert(HCL, "HCL", "RGB")
        1     2     3
  1    25    70   120
  2   150    60    60
```

S.convert() can also be used for grayscale conversion or color vision deficiency simulation. The syntax is

C = *S*.convert(*C*$_0$, *from*, "gray"[, *p*, *method*])
C = *S*.convert(*C*$_0$, *from*, "cvd"[, *p*, *method*])

where p is a real scalar in $[0, 1]$ specifying the proportion of gray or the severity of color vision deficiency. The default is $p = 1$ (complete conversion to gray, maximum CVD severity). This default can also be selected by typing . (missing). *method* selects the conversion method or CVD type; see functions *S*.gray() and *S*.cvd() for details.

Check validity of color specification

To check whether a color specification is valid you can type

color = *S*.cvalid(*colorspec*)

where *colorspec* is a single color specification as described for *S*.colors(). If *colorspec*

is valid, *color* will be set to the (expanded) name of the color, or the RGB code of the color if no color name is available. If *colorspec* is invalid, *color* will be set to empty string.

Obtain list of named colors

To obtain a list of named colors provided by `ColrSpace` (excluding Stata's system colors), type

list `=` *S*`.namedcolors(`[*pattern*`,` *case*]`)`

list will be a $n \times 2$ string matrix with color names in the first column and hex codes in the second column. Specify *pattern* to filter the list; only color names matching the specified pattern will be listed in this case. The syntax for *pattern* is as explained in [M-5] **strmatch()**. By default, case will be ignored; specify *case* $\neq 0$ for case-sensitive filtering.

Examples

```
: S = ColrSpace()
: S.namedcolors("*lime*")
                         1                      2
  ┌──────────────────────────────────────────────────┐
1 │                  Lime                 #00FF00  │
2 │             LimeGreen                 #32CD32  │
3 │   w3-2017-golden-lime                 #9c9a40  │
4 │    w3-2018-lime-punch                 #BFD641  │
5 │     w3-2018-limelight                 #F1EA7F  │
6 │          w3-food-lime                 #bffe28  │
7 │               w3-lime                 #cddc39  │
8 │          w3-win8-lime                 #a4c400  │
  └──────────────────────────────────────────────────┘

. mata: S.Colors(S.namedcolors("*lime*")[,1])
. colorpalette mata(S), rows(3)
```

Check whether palette exists

To check whether *name* matches an existing palette you can type

name = *S*.pexists(*name*[, *libname*])

name will be set to the (expanded) name of the palette if a matching palette was found. If no matching palette is found, *name* will be set to empty string. See *S*.palette() for information on palettes. *libname* will be replaced by the name of the ColrSpace library in which the palette was found. If no matching palette is found, *libname* will be left unchanged.

Obtain list of available palettes

To obtain a list of available palettes, type

list = *S*.palettes([*pattern*, *case*])

list will be a $n \times 2$ string matrix with palette names in the first column and library names in the second column. The library names provide information on the ColrSpace library in which a palette definition can be found. Specify *pattern* to filter the list; only palettes matching the specified pattern will be listed in this case. The syntax for *pattern* is as explained in [M-5] **strmatch()**. By default, case will be ignored; specify *case* $\neq 0$ for case-sensitive filtering.

Interpolation

In addition to *S*.ipolate(), ColrSpace also provides interpolation functions that do not involve translation between colorspaces and do not store any colors in *S*. These direct interpolation functions are

C = *S*.colipolate(*C*$_0$, *n*[,*range*, *power*, *positions*, *padded*])

for regular interpolation and

C = *S*.colipolate_c(*C*$_0$, *n*)

for circular interpolation, where C_0 is an $n_0 \times c$ matrix of n_0 origin colors that are interpolated to n destination colors. Other arguments are as for *S*.ipolate().

Recycling

In addition to *S*.recycle(), ColrSpace also provides a recycling function that does not store any colors in *S*. This direct recycling function is

C = *S*.colrecycle(*C*$_0$, *n*)

where C_0 is an $n_0 \times c$ matrix of n_0 input colors values that are recycled to n output colors.

Linear segmented colormaps

Function

RGB1 = *S*.lsmap(*R*, *G*, *B*, *n*[, *range*])

can be used to create linear segmented colormaps. Some of the colormaps above are implemented in terms of this function. R, G, and B are matrices specifying the anchor points of the segments (each row consist of three values: the anchor, the value of the color on the left of the anchor, and the value of the color on the right). See the corresponding tutorial page at matplotlib.org for details. *S*.lsmap() does not check the consistency of the specified matrices and may return invalid results if consistency is violated.

Clipping

Function

C = *S*.clip(*C*$_0$, *a*, *b*)

can be used for clipping, where C_0 is a real matrix of input values, a is a real scalar specifying the lower bound, and b is a real scalar specifying the upper bound. Values in C_0 smaller than a will be set to a; values larger than b will be set to b; values between a and b as well as missing values will be left as is.

7 Settings

7.1 Display overview of color space settings

To display an overview of the current color space settings of S, type

S.settings()

Example:

```
: S = ColrSpace()
: S.settings()
  rgb_gamma():  gamma = 2.4
                offset = .055
            transition = .0031308
                 slope = 12.92
```

```
rgb_white():      X = 95.047
                  Y = 100
                  Z = 108.883
rgb_xy():      red x = .64
               red y = .33
             green x = .3
             green y = .6
              blue x = .15
              blue y = .06
xyzwhite():       X = 95.047
                  Y = 100
                  Z = 108.883
viewcond():     Y_b = 20
                L_A = 4.07436654
                  F = 1
                  c = .69
                N_c = 1
ucscoefs():     K_L = 1
                c_1 = .007
                c_2 = .0228
chadapt():   method = "Bfd"
```

To restore the default color space settings, type

S.`clearsettings()`

7.2 RGB working space

To set the RGB working space, type

S.`rgbspace("`*name*`")`

where *name* is one of the following.

`Adobe 1998`	Adobe RGB (1998)
`Apple`	Apple RGB
`Best`	Best RGB
`Beta`	Beta RGB
`Bruce`	Bruce RGB
`CIE`	CIE 1931 RGB
`ColorMatch`	ColorMatch RGB
`Don 4`	Don RGB 4
`ECI v2`	ECI RGB v2
`Ekta PS5`	Ekta Space PS5
`Generic`	Generic RGB
`HDTV`	HDTV (HD-CIF)
`NTSC`	NTSC RGB (1953)

`PAL/SECAM`	PAL/SECAM RGB
`ProPhoto`	ProPhoto RGB
`SGI`	SGI RGB
`SMPTE-240M`	SMPTE-240M RGB
`SMPTE-C`	SMPTE-C RGB
`sRGB`	Standard RGB using primaries from Lindbloom (2017d)
`sRGB2`	Standard RGB using equation F.8 (XYZ to RGB matrix) from IEC (2003)
`sRGB3`	Standard RGB using equation F.7 (RGB to XYZ matrix) from IEC (2003)
`Wide Gamut`	Adobe Wide Gamut RGB
`Wide Gamut BL`	Wide Gamut variant from Lindbloom (2017d)

The names can be abbreviated and typed in lowercase letters. If abbreviation is ambiguous, the first matching name in the alphabetically ordered list will be used. See the `ColrSpace` source code for the definitions of the spaces. The definitions have been taken from Pascale (2003) and Lindbloom (2017d). Also see en.wikipedia.org/wiki/RGB_color_space. The default is *S*`.rgbspace("sRGB")`. This default can also be selected by typing *S*`.rgbspace("")`. Other color management systems may use slightly different definition of standard RGB. For example, the `colorspacious` Python library by Smith (2018) uses a definition equivalent to `"sRGB2"`. The advantage of `"sRGB"` is that RGB white (`255 255 255`) translates to the reference white in XYZ, which is not exactly true for `"sRGB2"` or `"sRGB3"`.

An RGB working space consists of three elements: the parameters of the gamma compression used to transform lRGB (linear RGB) to RGB, the reference white, and the working space primaries used to transform XYZ to lRGB. Instead of choosing a named RGB working space, the elements can also be set directly as described below.

Gamma compression

To set the gamma compression parameters, type

S`.rgb_gamma(`*args*`)`

where *args* is

gamma
or *gamma*`,` *offset*`,` *transition*`,` *slope*
or `(`*gamma*`,` *offset*`,` *transition*`,` *slope*`)`
or `"`*gamma*`"`
or `"`*gamma offset transition slope*`"`

If only *gamma* is provided, simple gamma encoding $C' = C^{1/gamma}$ is applied. If *offset*, *transition*, and *slope* are also provided, the detailed gamma encoding $C' = (1 + offset) \times C^{1/gamma} - offset$ if $C > transition$ and else $C' = C \times slope$ is used. A

typical value for *gamma* is 2.2; see Novak (2016) for an excellent explanation of gamma compression; also see en.wikipedia.org/wiki/Gamma_correction.

Reference white

The reference white can be set by

S.`rgb_white(`*args*`)`

where *args* is as described in for function *S*.`xyzwhite()`. If the reference white of the RGB working space differs from the XYZ reference white, `ColrSpace` applies chromatic adaption when translating between XYZ and lRGB.

Working space primaries

To set the working space primaries, type

S.`rgb_xy(`*xy*`)`

where *xy* is a 3×2 matrix containing the red, green, and blue xy primaries. `ColrSpace` uses the method described in Lindbloom (2017e) to compute the lRGB-to-XYZ transformation matrix from the white point and the primaries, and sets the XYZ-to-lRGB matrix to the inverse of the lRGB-to-XYZ matrix. Alternatively, you can type

S.`rgb_M(`*M*`)`

where *M* is a 3×3 matrix, to directly set the lRGB-to-XYZ matrix to *M* and the XYZ-to-lRGB matrix to `luinv(`*M*`)` (see [M-5] **luinv()**), or

S.`rgb_invM(`*invM*`)`

to set the XYZ-to-lRGB matrix to *invM* and the lRGB-to-XYZ matrix to `luinv(`*invM*`)`.

Retrieve settings

To retrieve the current RGB working space settings, you can type

```
gamma = S.rgb_gamma()
white = S.rgb_white()
   xy = S.rgb_xy()
    M = S.rgb_M()
invM  = S.rgb_invM()
```

7.3 XYZ reference white

To set the reference white for the CIE XYZ color space, type

S.`xyzwhite(`*args*`)`

where *args* is

X, *Y*, *Z*
or (*X*, *Y*, *Z*)
or "*X* *Y* *Z*"
or *x*, *y*
or (*x*, *y*)
or "*x* *y*"
or "*name*"

where X, Y, and Z are the XYZ coordinates of the white point (with $Y = 100$), x and y are the xyY coordinates of the white point (assuming $Y = 100$), and *name* is one of the following:

CIE 1931 2° observer	CIE 1964 10° observer	Description
`A`	`A 10 degree`	Incandescent/Tungsten 2856K
`B`	`B 10 degree`	Direct sunlight at noon 4874K (obsolete)
`B`	`B 10 degree`	Direct sunlight at noon 4874K (obsolete)
`B BL`		B 2 degree variant from Lindbloom (2017a)
`C`	`C 10 degree`	North sky daylight 6774K (obsolete)
`D50`	`D50 10 degree`	Horizon light 5003K (used for color rendering)
`D55`	`D55 10 degree`	Mid-morning/mid-afternoon daylight 5503K (used for photography)
`D65`	`D65 10 degree`	Noon daylight 6504K (new version of north sky daylight)
`D75`	`D75 10 degree`	North sky daylight 7504K
`9300K`		High eff. blue phosphor monitors 9300K
`E`		Uniform energy illuminant 5454K
`F1`	`F1 10 degree`	Daylight fluorescent 6430K
`F2`	`F2 10 degree`	Cool white fluorescent 4200K
`F3`	`F3 10 degree`	White fluorescent 3450K
`F4`	`F4 10 degree`	Warm white fluorescent 2940K
`F5`	`F5 10 degree`	Daylight fluorescent 6350K
`F6`	`F6 10 degree`	Lite white fluorescent 4150K
`F7`	`F7 10 degree`	Broad-band daylight fluorescent, 6500K
`F8`	`F8 10 degree`	D50 simulator, Sylvania F40 design 50, 5000K
`F9`	`F9 10 degree`	Cool white deluxe fluorescent 4150K

`F10`	`F10 10 degree`	Philips TL85, Ultralume 50, 5000K
`F11`	`F11 10 degree`	Narrow-band white fluorescen, Philips TL84, Ultralume 40, 4000K
`F12`	`F12 10 degree`	Philips TL83, Ultralume 30, 3000K

The names can be abbreviated and typed in lowercase letters (for example, `"D55 10 degree"` could be typed as `"d55 10"`). If abbreviation is ambiguous, the first matching name in the alphabetically ordered list will be used. See the `ColrSpace` source code for the definitions of the white points. The definitions have been taken from Pascale (2003), Lindbloom (2017a), and en.wikipedia.org/wiki/Standard_illuminant. The default is S.`xyzwhite("D65")`. This default can also be selected by typing S.`xyzwhite(.)` or S.`xyzwhite("")`. To retrieve a 1×3 rowvector containing the XYZ coordinates of the current white point, you can type

white `=` S.`xyzwhite()`

7.4 CIECAM02 viewing conditions

To set the CIECAM02 viewing conditions, type

S.`viewcond(`*args*`)`

where *args* is

Y_b, L_A, F, c, N_c
or Y_b, L_A, (F, c, N_c)
or (Y_b, L_A, F, c, N_c)
or "Y_b L_A F c N_c"
or Y_b, L_A, "*surround*"
or "Y_b L_A *surround*"

with *surround* equal to `average` ($F = 1$, $c = .69$, $N_c = 1$), `dim` ($F = .9$, $c = .59$, $N_c = .9$), or `dark` ($F = .8$, $c = .525$, $N_c = .8$) (abbreviations allowed). The default is $Y_b = 20$, $L_A = 64/(5\pi)$, and average surround. These defaults can also be selected by typing S.`viewcond(.)` or S.`viewcond("")`, or by setting Y_b to `.`, L_A to `.`, and *surround* to `.` or empty string. To retrieve a 1×5 rowvector of the current viewing condition parameters, type

viewcond `=` S.`viewcond()`

See Luo and Li (2013) for details on CIECAM02 viewing conditions.

7.5 Default coefficients for J'M'h and J'a'b'

To set the default uniform color space coefficients for $J'M'h$ and $J'a'b'$, type

S.ucscoefs(*args*)

where *args* is

K$_L$, *c*$_1$, *c*$_2$
or (*K*$_L$, *c*$_1$, *c*$_2$)
or "*K*$_L$ *c*$_1$ *c*$_2$"
or "*name*"

with *name* equal to UCS ($K_L = 1$, $c_1 = .007$, $c_2 = .0228$), LCD ($K_L = .77$, $c_1 = .007$, $c_2 = .0053$), or SCD ($K_L = 1.24$, $c_1 = .007$, $c_2 = .0363$) (abbreviations and lowercase letters allowed). To to retrieve a 1×3 rowvector of the current default coefficients, type

ucscoefs = *S*.ucscoefs(*args*)

See Luo and Li (2013, chapter 2.6.1) and Luo et al. (2006) for details on these coefficients.

7.6 Chromatic adaption method

To set the chromatic adaption method type

S.chadapt(*method*)

where *method* is "Bfd" (Bradford), "identity" (XYZ Scaling), "vKries" (Von Kries), or "CAT02" (abbreviations and lowercase letters allowed). The default is *S*.chadapt("Bfd"), which can also be selected by typing *S*.chadapt(""). The Bradford, XYZ Scaling, and Von Kries methods use the procedure described in Lindbloom (2017a), the "CAT02" method uses the procedure described in Luo and Li (2013, page 33). To retrieve a string scalar containing the current method, type

method = *S*.chadapt()

ColrSpace uses chromatic adaption internally whenever such a translation is necessary. However, you can also apply chromatic adaption manually by typing

XYZnew = *S*.XYZ_to_XYZ(*XYZ*, *from*, *to*)

where *XYZ* is an $n \times 3$ matrix of XYZ values to be adapted, *from* is the origin whitepoint, and *to* is the destination whitepoint; any single-argument whitepoint specification as described in for *S*.xyzwhite() is allowed. Function *S*.XYZ_to_XYZ() does not change or store any colors in *S*.

To retrieve the predefined transformation matrices on which chromatic adaption is based, type

M = *S*.tmatrix([*name*])

where *name* is `"Bfd"`, `"identity"`, `"vKries"`, `"CAT02"`, or `"HPE"` (Hunt-Pointer-Estevez) (abbreviations and lowercase letters allowed). The default is *S*.`tmatrix("Bfd")`, which can also be selected by typing *S*.`tmatrix("")`. The `"HPE"` matrix is not used for chromatic adaption, but has been included in *S*.`tmatrix()` for convenience. It is used when translating colors from XYZ to CIECAM02; see Luo and Li (2013).

8 Alphabetical index of functions

Below is a sorted list of all functions provided by `ColrSpace`. Several of these functions also come in variants such as *S*.`add_`*name*`()`, *S*.*name*`_added()`, or *S*.`add_`*name*`_added()`, where *name* is the function name.

`ColrSpace()`	initialize a `ColrSpace` object
S.`add()`	add colors in a particular space
S.`alpha()`	set/retrieve opacity
S.`chadapt()`	set chromatic adaption method
S.`clear()`	remove all colors and meta data
S.`clearindex()`	clear internal look-up tables
S.`clearsettings()`	clear color space settings
S.`clip()`	helper function for clipping
S.`contrast()`	compute contrast ratios
S.`delta()`	compute color differences
S.`describe()`	displays contents of *S*
S.`drop()`	drop colors
S.`colipolate()`	helper function for interpolation
S.`colipolate_c()`	helper function for circular interpolation
S.`colors()`	string input/output (scalar)
S.`Colors()`	string input/output (vector)
S.`colrecycle()`	helper function for recycling
S.`convert()`	convert colors between spaces
S.`cvalid()`	check whether color is valid
S.`cvd()`	color vision deficiency simulation
S.`cvd_M()`	helper function to retrieve CVD matrix
S.`get()`	retrieve colors in a particular space
S.`gray()`	gray scale conversion
S.`info()`	color description input/output (scalar)
S.`Info()`	color description input/output (vector)

S.intensify()	adjust color intensity
S.intensity()	set/retrieve intensity adjustment
S.ipolate()	interpolate colors
S.isipolate()	whether interpolation has been applied
S.lsmap()	helper function to create linear segmented colormaps
S.luminate()	adjust luminance of colors
S.mix()	mix colors
S.N()	retrieve number of colors
S.name()	set/retrieve name of color collection
S.namedcolors()	return index of available named colors
S.names()	color names input/output (scalar)
S.Names()	color names input/output (vector)
S.note()	set/retrieve description of color collection
S.opacity()	set/retrieve opacity
S.order()	order colors
S.palette()	retrieve colors from named palette
S.palettes()	return index of available palettes
S.pclass()	set/retrieve class of color collection
S.pexists()	check whether named palette exists
S.recycle()	recycle colors
S.reset()	reset colors in a particular space
S.reverse()	reverse the order of colors
S.rgbspace()	set RGB working space
S.rgb_gamma()	set/retrieve gamma correction
S.rgb_invM()	set/retrieve XYZ-to-lRGB matrix
S.rgb_M()	set/retrieve lRGB-to-XYZ matrix
S.rgb_white()	set/retrieve RGB reference white
S.rgb_xy()	set/retrieve RGB primaries
S.saturate()	adjust saturation (chroma) of colors
S.select()	select colors
S.set()	set colors in a particular space
S.settings()	display color space settings
S.shift()	shift positions of colors
S.source()	set/retrieve source of color collection
S.tmatrix()	retrieve transformation matrices
S.ucscoefs()	set default coefficients for $J'M'h$ and $J'a'b'$

S.viewcond()	set/retrieve CIECAM02 viewing conditions
S.xyzwhite()	set/retrieve XYZ reference white
S.XYZ_to_XYZ()	apply chromatic adaption

9 Source code and certification script

lcolrspace.mlib has been compiled in Stata 14.2. The source code can be found in file colrspace_source.sthlp. Palette definitions, parameters of color generators, and definitions of named colors are kept in additional source files. These files are colrspace_library_palettes.sthlp, colrspace_library_lsmaps.sthlp, colrspace_library_rgbmaps.sthlp, colrspace_library_generators.sthlp, and colrspace_library_namedcolors.sthlp.

Users can extend the set of available palettes and colors by providing personal library files. These files should be stored somewhere along the [R] **adopath** (for example in the PERSONAL directory), so Stata can find them, and they must be named as above but with a "_personal" suffix (e.g. colrspace_library_palettes_personal.sthlp). Each library file has its peculiar syntax; see the explanations in the file headers.

A certification script testing internal consistency and comparing results to some test values and results from the colorspacious Python library by Smith (2018) (see file gold_values.py at Github) as well as to results obtained from the color calculators at colorizer.org and www.brucelindbloom.com, can be found at github.com/benjann/colrspace/blob/master/colrspace_cscript.do.

10 References

Bischof, D. 2017a. G538SCHEMES: module to provide graphics schemes for http://fivethirtyeight.com. Statistical Software Components S458404, Boston College Department of Economics. https://ideas.repec.org/c/boc/bocode/s458404.html.

———. 2017b. New graphic schemes for Stata: plotplain and plottig. *The Stata Journal* 17(3): 748–759.

Brewer, C. A. 2016. *Designing Better Maps. A Guide for GIS Users*. 2nd ed. Redlands, CA: Esri Press.

Brewer, C. A., G. W. Hatchard, and M. A. Harrower. 2003. ColorBrewer in Print: A Catalog of Color Schemes for Maps. *Cartography and Geographic Information Science* 30(1): 5–32.

Briatte, F. 2013. SCHEME-BURD: Stata module to provide a ColorBrewer-inspired graphics scheme with qualitative and blue-to-red diverging colors. Statistical Software Components S457623, Boston College Department of Economics. https://ideas.repec.org/c/boc/bocode/s457623.html.

Bundesamt für Statistik. 2017. Layoutrichtlinien. Gestaltungs und Redaktionsrichtlinien für Publikationen, Tabellen und grafische Assets. Technical Report Version 1.1.1, Bundesamt für Statistik, Neuchâtel.

Crameri, F. 2018. Scientific colour maps. Zenodo. DOI: 10.5281/zenodo.1243862.

Hunt, R. W. G. 2004. *The Reproduction of Colour*. 6th ed. West Sussex: John Wiley & Sons.

Hunter, J. D. 2007. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering* 9(3): 90–95.

Ihaka, R., P. Murrell, K. Hornik, J. C. Fisher, R. Stauffer, and A. Zeileis. 2016. colorspace: Color Space Manipulation. R package version 1.3-2. http://CRAN.R-project.org/package=colorspace.

International Electrotechnical Commission (IEC). 2003. International Standard IEC 61966-2-1:1999/AMD1:2003. Amendment 1 – Multimedia systems and equipment – Color measurement and management – Part 2-1: Color management – Default RGB color space – sRGB. http://www.sis.se/api/document/preview/562720/.

Jann, B. 2018. Color palettes for Stata graphics. *The Stata Journal* 18(4): 765–785.

Juul, S. 2003. Lean mainstream schemes for Stata 8 graphics. *The Stata Journal* 3(3): 295–301.

Kovesi, P. 2015. Good Colour Maps: How to Design Them. arXiv:1509.03700 [cs.GR].

Lin, S., J. Fortuna, C. Kulkarni, M. Stone, and J. Heer. 2013. Selecting Semantically-Resonant Colors for Data Visualization. *Computer Graphics Forum* 32(3pt4): 401–410.

Lindbloom, B. J. 2017a. Chromatic Adaptation. Revision 06 Apr 2017. http://www.brucelindbloom.com/Eqn_ChromAdapt.html.

———. 2017b. Delta E (CIE 1994). Revision 07 Apr 2017. http://www.brucelindbloom.com/Eqn_DeltaE_CIE94.html.

———. 2017c. Delta E (CIE 2000). Revision 08 Apr 2017. http://www.brucelindbloom.com/Eqn_DeltaE_CIE2000.html.

———. 2017d. RGB Working Space Information. Revision 06 Apr 2017. http://www.brucelindbloom.com/WorkingSpaceInfo.html.

———. 2017e. RGB/XYZ Matrices. Revision 07 Apr 2017. http://www.brucelindbloom.com/Eqn_RGB_XYZ_Matrix.html.

Luo, M. R., G. Cui, and C. Li. 2006. Uniform Colour Spaces Based on CIECAM02 Colour Appearance Model. *COLOR research and application* 31(4): 320–330.

Luo, M. R., and C. Li. 2013. CIECAM02 and Its Recent Developments. In *Advanced Color Image Processing and Analysis*, ed. C. Fernandez-Maloigne, 19–58. New York: Springer.

Machado, G. M., M. M. Oliveira, and L. A. F. Fernandes. 2009. A Physiologically-based Model for Simulation of Color Vision Deficiency. *IEEE Transactions on Visualization and Computer Graphics* 15(6): 1291–1298.

Morris, T. 2013. SCHEME-MRC: Stata module to provide graphics scheme for UK Medical Research Council. Statistical Software Components S457703, Boston College Department of Economics. https://ideas.repec.org/c/boc/bocode/s457703.html.

———. 2015. SCHEME-TFL: Stata module to provide graph scheme, based on Transport for London's corporate colour pallette. Statistical Software Components S458103, Boston College Department of Economics. https://ideas.repec.org/c/boc/bocode/s458103.html.

Novak, J. 2016. What every coder should know about gamma. 2016 Sep 21. https://blog.johnnovak.net/2016/09/21/what-every-coder-should-know-about-gamma/.

Okabe, M., and K. Ito. 2002. Color Universal Design (CUD). How to make figures and presentations that are friendly to Colorblind people. http://jfly.iam.u-tokyo.ac.jp/color/.

Pascale, D. 2003. *A review of RGB color spaces ... from xyY to R'G'B'*. Montreal: The BabelColor Company. URL http://www.babelcolor.com/index_htm_files/A%20review%20of%20RGB%20color%20spaces.pdf.

Pisati, M. 2007. SPMAP: Stata module to visualize spatial data. Statistical Software Components S456812, Boston College Department of Economics. http://ideas.repec.org/c/boc/bocode/s456812.html.

Smith, N. J. 2018. colorspacious 1.1.2: A powerful, accurate, and easy-to-use Python library for doing colorspace conversions. http://pypi.org/project/colorspacious.

Tol, P. 2012. Colour Schemes. SRON Technical Note, Doc. no. SRON/EPS/TN/09-002. https://personal.sron.nl/~pault/colourschemes.pdf.

Zeileis, A., K. Hornik, and P. Murrell. 2009. Escaping RGBland: Selecting Colors for Statistical Graphics. *Computational Statistics & Data Analysis* 53: 3259–3270.