Package 'colorSpec'

May 8, 2026

Type Package

Title Color Calculations with Emphasis on Spectral Data

Version 1.8-0

Encoding UTF-8

Date 2025-06-08

Maintainer Glenn Davis <gdavis@gluonics.com>

Description Calculate with spectral properties of light sources, materials, cameras, eyes, and scanners. Build complex systems from simpler parts using a spectral product algebra. For light sources, compute CCT, CRI, SSI, and IES TM-30 reports. For object colors, compute optimal colors and Logvinenko coordinates. Work with the standard CIE illuminants and color matching functions, and read spectra from text files, including CGATS files. Estimate a spectrum from its response. A user guide and 9 vignettes are included.

License GPL (>= 3)

LazyLoad yes

LazyData yes

Depends R (>= 4.0.0)

Imports logger

Suggests spacesXYZ (>= 1.5-1), rootSolve, MASS, quadprog, rgl, spacesRGB, zonohedra (>= 0.4-0), microbenchmark, arrangements, knitr, rmarkdown, png

Enhances plyr

NeedsCompilation no

Repository CRAN

VignetteBuilder knitr

BuildVignettes yes

ByteCompile no

Author Glenn Davis [aut, cre]

Date/Publication 2025-06-10 13:10:06 UTC

Contents

`colorSpec-package` *Package colorSpec - Color Calculations with Emphasis on Spectral Data*

Description

Package **colorSpec** is for working with spectral color data in R.

Details

Features:

1. a clear classification of the commonly seen spectra into 4 types - depending on the vector space to which they belong
2. flexible organization for the spectra in memory, using an S3 class - **colorSpec**
3. a product algebra for the **colorSpec** objects
4. uniform handling of biological eyes, electronic cameras, and general action spectra
5. a few advanced calculations, such as computing optimal colors (aka Macadam Limits)

6. inverse colorimetry, e.g. reflectance recovery from response
7. built-in essential tables, such as the CIE illuminants and color matching functions
8. a package logging system with log levels taken from the popular **Log4J**
9. support for reading a few spectrum file types, including CGATS
10. bonus files containing some other interesting spectra
11. minimal dependencies on other R packages

Non-features:

1. there is no support for many common 3D color spaces, such as CIELAB, HSL, etc.. For these spaces see packages **colorspace**, **colorscience**, **spacesRGB**, and **spacesXYZ**.
2. there are few non-linear operations
3. there is little support for scientific units; for these see package **colorscience**
4. photons are parameterized by wavelength in nanometers; other wavelength units (such as Angstrom and micron) and alternative parameterizations (such as wavenumber and electron-volt) are not available

Regarding the non-linear operations in 2, the only such operations are conversion of linear RGB to display RGB, conversion of absorbance to transmittance, and the reparameterized wavelength in `computeADL`. The electronic camera model is purely linear with no dark current offset or other deviations.

Many ideas are taken from packages **hyperSpec**, **hsdar**, **pavo**, and **zoo**.

Author(s)

Glenn Davis <gdavis@gluonics.com>

See Also

`colorSpec` for the S3 class provided by this package.

colorSpec User Guide

`ABC` *Standard Illuminants A, B, and C (1931)*

Description

`A.1nm`	standard Illuminant A, 360 to 780 nm at 1 nm intervals
`B.5nm`	standard Illuminant B, 320 to 780 nm at 5 nm intervals
`C.5nm`	standard Illuminant C, 320 to 780 nm at 5 nm intervals

Format

Each is a **colorSpec** object organized as a vector, with `quantity` equal to `"energy"`.

Details

All of these have been divided by 100, to make the values at 560nm near 1 instead of 100.

Source

http://www.cvrl.org

References

Günther Wyszecki and W. S. Stiles. **Color Science : Concepts and Methods, Quantitative Data and Formulae**. Second Edition. Wiley-Interscience. 1982.

A	Table I(3.3.4) pp. 754-758.
B	Table II(3.3.4) pp. 759.
C	Table II(3.3.4) pp. 759.

See Also

`D50` `D65`

Examples

```
summary(xyz1931.1nm)
white.point = product( D65.1nm, xyz1931.1nm, wave='auto' )
```

`actinometric` *convert a colorSpec object to be actinometric*

Description

Convert a radiometric **colorSpec** object to have quantity that is actinometric (number of photons). Test an object for whether it is actinometric.

Usage

```
## S3 method for class 'colorSpec'
actinometric( x, multiplier=1, warn=FALSE )

## S3 method for class 'colorSpec'
is.actinometric( x )
```

Arguments

x	a **colorSpec** object
multiplier	a scalar which is multiplied by the output, and intended for unit conversion
warn	if TRUE and a conversion actually takes place, the a WARN message is issued. This makes the user aware of the conversion, so units can be verified. This can be useful when actinometric() is called from another **colorSpec** function.

Details

If the quantity of x does not start with 'energy' then the quantity is not radiometric and so x is returned unchanged. Otherwise x is radiometric (energy-based), and must be converted.

If type(x) is 'light' then the most common radiometric energy unit is joule.
The conversion equation is:

$$Q = E * \lambda * 10^6/(N_A * h * c)$$

wher Q is the photon count, E is the energy of the photons, N_A is Avogadro's constant, h is Planck's constant, c is the speed of light, and λ is the wavelength. The output unit of photon count is (μmole of photons) = $(6.02214 * 10^{17}$ photons). If a different unit for Q is desired, then the output should be scaled appropriately. For example, if the desired unit of photon count is exaphotons, then set multiplier=0.602214.

If the quantity(x) is 'energy->electrical', then the most common radiometric unit of responsivity to light is coulombs/joule (C/J) or amps/watt (A/W). The conversion equation is:

$$QE = R_e * ((h * c)/e)/\lambda$$

where QE is the quantum efficiency, R_e is the energy-based responsivity, and e is the charge of an electron (in C).
If the unit of x is not C/J, then multiplier should be set appropriately.

If the quantity(x) is 'energy->neural' or 'energy->action', the most common radiometric unit of energy is joule (J).

The conversion equation is:

$$R_p = R_e * 10^{-6} * (N_A * h * c)/\lambda$$

where R_p is the photon-based responsivity, and R_e is the energy-based responsivity, The output unit of photon count is (μmole of photons) = $(6.02214 * 10^{17}$ photons). This essentially the reciprocal of the first conversion equation.

The argument multiplier is applied to the right side of all the above conversion equations.

Value

actinometric() returns a **colorSpec** object with quantity that is actinometric (photon-based) and not radiometric (energy-based). If type(x) is a material type ('material' or 'responsivity.material') then x is returned unchanged.

If quantity(x) starts with 'photons', then is.actinometric() returns TRUE, and otherwise FALSE.

Note

To log the executed conversion equation, execute `cs.options(loglevel='INFO')`.

Source

Wikipedia. **Photon counting**. https://en.wikipedia.org/wiki/Photon_counting

See Also

quantity, type, cs.options, radiometric

Examples

```
colSums( solar.irradiance ) # step size is 1nm, from 280 to 1000 nm. organized as a matrix
# AirMass.0  GlobalTilt AirMass.1.5
#  944.5458    740.3220    649.7749  # irradiance, watts*m^{-2}


colSums( actinometric(solar.irradiance) )
# AirMass.0  GlobalTilt AirMass.1.5
#  4886.920   3947.761   3522.149  # photon irradiance, (umoles of photons)*sec^{-1}*m^{-2}

colSums( actinometric(solar.irradiance,multiplier=0.602214) )
# AirMass.0  GlobalTilt AirMass.1.5
#  2942.972    2377.397    2121.088  # photon irradiance, exaphotons*sec^{-1}*m^{-2}
```

ANSI/IES TM-30 *Compute ANSI/IES TM-30 data, and Create Color Rendition Reports*

Description

Compute TM-30 data from a **colorSpec** object with the spectrum of a light source (`type()` is `'light'`). And plot full reports from this data. The full TM-30 standard is in the **References**.

These functions were guided by the **IES Spectral Calculator**, and checked against the downloadable .XLSX spreadsheets and .PDF reports available there.

Usage

```
## S3 method for class 'colorSpec'
computeTM30( x, reference=NULL )

## S3 method for class 'TM30'
plot( x, omi=c(0.75,0.2,1.25,0.25), ... )

## S3 method for class 'TM30'
print( x, ... )
```

Arguments

x	For computeTM30(), a colorSpec object with a single spectrum whose type is 'light'. For plot() and print(), an object returned by computeTM30(), with class 'TM30'.
reference	a **colorSpec** object used as the reference light source for x. If it is NULL, this reference is computed from the CCT of x, according to the instructions in **ANSI/IES TM-30-20**.
omi	a vector of 4 numbers, defining the outer margins of the plot, in inches; for details see par(). For the TM-30 report, text is written in the top and bottom outer margins using title(), so these margins may require some fine tuning.
...	further arguments ignored

Details

computeTM30() follows the recipes in **ANSI/IES TM-30-20**.

plot() reproduces the full reports at the **IES Spectral Calculator**.

print prints a short 9-line summary of the important TM-30-20 measures.

Value

computeTM30() returns a list of data with class 'TM30'. The data is designed to be a reflection of the data in the .XLSX spreadsheets created by the **IES Spectral Calculator**.

plot() and print() return TRUE or FALSE, invisibly.

References

ANSI/IES TM-30-20. **Technical Memorandum: IES Method for Evaluating Light Source Color Rendition**. https://store.ies.org/product/technical-memorandum-ies-method-for-evaluating-light-s

IES Spectral Calculator. https://www.ies.org/standards/standards-toolbox/tm-30-spectral-calculator/

See Also

par(), title()

Examples

```
## Not run:
F1 = subset( Fs.5nm, 1 )

pdf( "report.pdf", width=8.5, height=11 )
plot( computeTM30(F1) )
dev.off()

svg( "report.svg", width=8.5, height=11 )
plot( computeTM30(F1) )
dev.off()

## End(Not run)
```

applyspec *apply a function to each spectrum in a colorSpec object*

Description

apply a function to each spectrum in a **colorSpec** object

Usage

```
## S3 method for class 'colorSpec'
applyspec( x, FUN, ... )
```

Arguments

x	a **colorSpec** object with N wavelengths
FUN	a function that takes an N-vector as argument and returns an N-vector
...	additional arguments passed to FUN

Details

applyspec() simply calls apply() with the right MARGIN.

Value

a **colorSpec** object with the same dimensions, wavelength, quantity, and organization. If FUN does not return an N-vector, it is an ERROR and applyspec() returns NULL.

See Also

quantity, wavelength, linearize, organization, apply

Examples

```
#  convert absorbance to transmittance
path = system.file( "extdata/stains/Hematoxylin.txt", package='colorSpec' )
x = readSpectra( path )
x = applyspec( x, function(y) {10^(-y)} ) # this is what linearize(x) does
```

`as.data.frame` *Convert a colorSpec Object to a data.frame*

Description

convert a **colorSpec** object to a `data.frame`

Usage

```
## S3 method for class 'colorSpec'
as.data.frame( x, row.names=NULL, optional=FALSE, organization='auto', ... )
```

Arguments

`x`	a **colorSpec** object
`organization`	The organization of the returned `data.frame`, which can be `'row'`, `'col'`, or `'auto'`. If `'auto'`, then `'row'` or `'col'` is selected automatically, see **Details**
`row.names`	ignored
`optional`	ignored
`...`	extra arguments ignored

Details

If `organization` is `'auto'`, and the organization of `x` is `'df.row'`, then `organization` is set to `'row'` and the returned `data.frame` has the spectra in the rows. Otherwise the returned `data.frame` has the spectra in the columns.

Value

If the returned `data.frame` has the spectra in the rows, then the spectra are in a matrix in the last column (with name `spectra`), and any existing `extradata` are also returned in the other columns. The wavelengths are only present in character form, as the `colnames` of the matrix.
If the returned `data.frame` has the spectra in the columns, then the wavelengths are in the first column, and the spectra are in the other columns.

See Also

`as.matrix`, `extradata`

`atmosphere` *atmospheric transmittance along a horizontal path*

Description

Calculate transmittance along a horizontal optical path in the atmosphere, as a function of length (distance) and the molecular and aerosol properties. Because the path is horizontal, the atmospheric properties are assumed to be constant on the path. Only molecular scattering is considered. There is no modeling of molecular absorption; for visible wavelengths this is reasonable.

Usage

```
atmosTransmittance( distance, wavelength=380:720,
                    molecules=list(N=2.547305e25,n0=1.000293),
                    aerosols=list(metrange=25000,alpha=0.8,beta=0.0001) )
```

Arguments

`distance`	the length of the optical path, in meters. It can also be a numeric vector of lengths.
`wavelength`	a vector of wavelengths, in nm, for the transmittance calculations
`molecules`	a list of molecular properties, see **Details**. If this is `NULL`, then the molecular transmittance is identically 1.
`aerosols`	a list of aerosol properties, see **Details**. If this is `NULL`, then the aerosol transmittance is identically 1.

Details

The list `molecules` has 2 parameters that describe the molecules in the atmosphere. `N` is the molecular density of the atmosphere at sea level, in $molecules/meter^3$. The given default is the density at sea level. `n0` is the refractive index of pure molecular air (with no aerosols). For the molecular attenuation, the standard model for Rayleigh scattering is used, and there is no modeling of molecular absorption.

The list `aerosols` has 3 parameters that describe the aerosols in the atmosphere. The standard Angstrom aerosol attenuation model is:

$$attenuation(\lambda) = \beta * (\lambda/\lambda_0)^{-\alpha}$$

α is the Angstrom exponent, and is dimensionless. $attenuation$ and β have unit m^{-1}. And λ_0=550nm.

`metrange` is the *Meteorological Range* of the atmosphere in meters, as defined by *Koschmieder*. This is the distance at which the transmittance=0.02 at λ_0. If `metrange` is not `NULL` (the default is 25000) then both α and β are calculated to achieve this desired `metrange`, and the supplied α and β are ignored. α is calculated from `metrange` using the *Kruse* model, see **Note**. β is calculated so that the product of molecular and aerosol transmittance yields the desired `metrange`. In fact:

$$\beta = -\mu_0 - log(0.02)/V_r$$

where μ_0 is the molecular attenuation at λ_0, and V_r is the meteorological range. For a log message with the calculated values, execute `cs.options(loglevel='INFO')` before calling `atmosTransmittance()`.

Value

`atmosTransmittance()` returns a **colorSpec** object with `quantity` equal to `'transmittance'`. There is a spectrum in the object for each value in the vector `distance`. The `specnames` are set to `sprintf("dist=%gm",distance)`.
The final transmittance is the product of the molecular transmittance and the aerosol transmittance. If both `molecules` and `aerosols` are `NULL`, then the final transmittance is identically 1; the atmosphere has become a vacuum.

Note

The Kruse model for α as a function of V_r is defined in 3 pieces. For $0 \leq V_r < 6000$, $\alpha = 0.585 * (V_r/1000)^{1/3}$. For $6000 \leq V_r < 50000$, $\alpha = 1.3$. And for $V_r \geq 50000$, $\alpha = 1.6$. So α is increasing, but not strictly, and not continuously. V_r is in meters. See *Kruse* and *Kaushal*.

The built-in object `atmosphere2003` is transmittance along an optical path that is **NOT** horizontal, and extends to outer space. This is *much* more complicated to calculate.

References

Angstrom, Anders. On the atmospheric transmission of sun radiation and on dust in the air. *Geogr. Ann.*, no. 2. 1929.

Kaushal, H. and Jain, V.K. and Kar, S. **Free Space Optical Communication**. Springer. 2017.

Koschmieder, Harald. Theorie der horizontalen Sichtweite. *Beitrage zur Physik der Atmosphare*. 1924. **12**. pages 33-53.

P. W. Kruse, L. D. McGlauchlin, and R. B. McQuistan. **Elements of Infrared Technology: Generation, Transmission, and Detection**. J. Wiley & Sons, New York, 1962.

See Also

`solar.irradiance`, `specnames`

Examples

```
trans = atmosTransmittance( c(5,10,15,20,25)*1000 ) # 5 distances with atmospheric defaults

# verify that transmittance[550]=0.02 at dist=25000
plot( trans, legend='bottomright', log='y' )

# repeat, but this time assign alpha and beta explicitly
trans = atmosTransmittance( c(5,10,15,20,25)*1000, aero=list(alpha=1,beta=0.0001) )
```

bandSpectra *Compute Band-based Material Spectra, and Bands for Existing Material Spectra*

Description

A band-based material spectrum is a superimposition of bandpass filters, and (optionally) a bandstop filter. The 2 functions in this topic convert a vector of numbers between 0 and 1 to a band representation, and back again.

Usage

```
bandMaterial( lambda, wavelength=380:780 )

## S3 method for class 'colorSpec'
bandRepresentation( x )
```

Arguments

lambda	a numeric Mx2 matrix with wavelength pairs in the rows, or a vector that can be converted to such a matrix, by row. The two wavelengths in a row (the *transition wavelengths*) define either a bandpass or bandstop filter, and all the rows are superimposed to define the transmittance spectrum of the final material. If the 2 wavelengths are denoted by λ_1 and λ_2, and $\lambda_1 < \lambda_2$ then the filter is a bandpass filter. If the 2 wavelengths are swapped, then the spectrum is "flipped" and is a bandstop filter, and the band "wraps around" from long wavelengths to short. There can be at most 1 bandstop filter in the matrix, otherwise it is an error. The bands must be pairwise disjoint, otherwise it is an error. To get a material with transmittance identically 0, set lambda to a 0x2 matrix. To get a material with transmittance identically 1, set lambda to a 1x2 matrix with $\lambda_1 = \beta_0$ and $\lambda_2 = \beta_N$, where N is the number of wavelengths. See vignette **Convexity and Transitions** for the definition of β_0 and β_N and other mathematical details. lambda can also be a list of such matrices, which are processed separately, see **Value**.
wavelength	a vector of wavelengths for the returned object
x	a **colorSpec** object with type equal to 'material'

Details

bandRepresentation() is a right-inverse of bandMaterial(), see **Examples** and the test script test-bands.R. For more mathematical details, see the vignette **Convexity and Transitions**.

Value

bandMaterial() returns a **colorSpec** object with quantity equal to 'transmitance'. If lambda is a matrix, then the object has 1 spectrum. If lambda is a list of matrices with length N, then the object has N spectra.

bandRepresentation() returns a list of matrices with 2 columns. There is a matrix in the list for each spectrum in x.

See Also

rectangularMaterial(), vignette **Convexity and Transitions**

Examples

```
#  make a vector superimposing a bandpass and a bandstop filter, and of the proper length 401
vec = c( rep(1,100), 0.5, rep(0,40), .25, rep(1,50), 0.9, rep(0,100), 0.4, rep(1,107) )

#  convert that vector to a colorSpec object, with a single spectrum
spec = colorSpec( vec, wavelength=380:780, quantity='transmittance', specnames='sample' )

#  extract and print the 2 bands
lambda = bandRepresentation( spec ) ;  print(lambda)

##  $sample
##      lambda1 lambda2
##  BS   673.10   480.0
##  BP1  521.25   572.4

#  convert the 2 bands  (the transition wavelengths) back to a vector of length 401
#  and compare with the original vector
delta = vec - coredata( bandMaterial(lambda) )

range(delta)
##  [1] -9.092727e-14  2.275957e-14
```

---

bind *Combine colorSpec Objects*

Description

Take a sequence of **colorSpec** objects and combine their spectra

Usage

```
## S3 method for class 'colorSpec'
bind( ... )
```

Arguments

...	**colorSpec** objects with the same wavelength and quantity, and with distinct specnames (no duplicates)

Details

The organization of the returned object is the most complex of those in the inputs, where the order of complexity is:

'matrix' < 'df.col' < 'df.row'

If the selected organization is 'df.row', the extradata is combined in a way that preserves all the columns. Missing data is filled with NAs, analogous to rbind.fill().

The metadata of the returned object is copied from the first object in the input list.

Value

bind() returns a **colorSpec** object, or NULL in case of ERROR. If the bind is successful, the number of spectra in the output object is the sum of the number of spectra in the input objects.

See Also

wavelength, quantity, specnames, organization, extradata, metadata, rbind.fill()

Examples

```
Rosco = readSpectra( system.file( 'extdata/objects/Rosco.txt', package='colorSpec' ) )
Rosco = resample( Rosco, wave=wavelength(Hoya) )
numSpectra(Hoya)        # prints 4
numSpectra(Rosco)       # prints 42

filters = bind( Hoya, Rosco )
numSpectra(filters)     # prints 46

colnames( extradata(Hoya) )
## [1] "SAMPLE_NAME"  "FresnelReflectance"  "Thickness"

colnames( extradata(Rosco) )
## [1] "Model"  "SampleID"  "SAMPLE_NAME"  "Substrate"  "RefractiveIndex"  "Thickness"

##  The columns in common are "SAMPLE_NAME" and "Thickness"


colnames( extradata(filters) )
## [1] "FresnelReflectance" "Model" "RefractiveIndex" "SAMPLE_NAME"
## [5] "SampleID" "Substrate" "Thickness"
##
## "SAMPLE_NAME" and "Thickness" are combined in the usual way
## The other columns are present, and missing data is filled with NAs
```

`calibrate` *make a linear modification to a colorSpec responder*

Description

make a linear modification to a **colorSpec** responder with M spectra, so a specific stimulus (a single spectrum) creates a specific response (an M-vector). It is generalized form of *white balance*.
The options are complicated, but in all cases the returned object is `multiply(x,gmat)` where `gmat` is an internally calculated MxM matrix - called the *gain matrix*. Stated another way, the spectra in the output are linear combinations of spectra in the input `x`.
In case of ERROR, a message is logged and the original `x` is returned.

Usage

```
## S3 method for class 'colorSpec'
calibrate( x, stimulus=NULL, response=NULL, method=NULL )
```

Arguments

`x`	a **colorSpec** responder with M spectra. The `type` must be `'responsivity.light'` or `'responsivity.material'`.
`stimulus`	a **colorSpec** object with a single spectrum, with `type` either `'light'` or `'material'` to match `x`. The wavelength sequence of `stimulus` must be equal to that of `x`. If `stimulus` is `NULL`, then an appropriate default is chosen, see **Details**.
`response`	an M-vector, or a scalar which is then replicated to length M. Normally all entries are not `NA`, but it is OK to have exactly one that is not `NA`. In this special case, a single scaling factor is computed from that non-`NA` coordinate, and then applied to *all* M coordinates; the `method` must be `'scaling'`. This is useful for the recommended method for calibration in ASTM E308-01 section 7.1.2. The same type of scaling is also recommended method in CIE 15: Technical Report section 7.1. In this case `response=c(NA,100,NA)` so the special coordinate is the luminance Y. See the **Examples** below and the vignettes **Viewing Object Colors in a Gallery** and **The Effect of the Aging Human Lens on Color Vision**. All entries in `response`, that are not `NA`, must be positive. If `response` is `NULL`, then an appropriate default *may be* chosen, see **Details**.
`method`	an MxM *adaptation matrix*. `method` can also be `'scaling'` and it is then set to the MxM identity matrix, which scales each responsivity spectrum in `x` independently. If M=3, `method` can also be `'Bradford'`, `'Von Kries'`, `'MCAT02'`, or `'Bianco+Schettini'`, and it is then set to the popular corresponding *chromatic adaptation matrix*. For these special matrices, the spectra in `x` are **not** scaled independently; there is "cross-talk". If `method` is `NULL`, then an appropriate default is chosen, see **Details**.

Details

If stimulus is NULL, it is set to illuminantE() or neutralMaterial() to match x.

If response is NULL and the response of x is electrical or action, then response is set to an M-vector of all 1s. If response is NULL and the response of x is neural, then this is an ERROR and the user is prompted to supply a specific response.

If method is NULL, its assignment is complicated.
If M=3 and the response of x is neural, and the specnames of x partially match c('x','y','z') (case-insensitive), and none of the components of response are NA, then the neural response is assumed to be human, and the method is set to 'Bradford'.
Otherwise method is set to 'scaling'.

Value

a **colorSpec** object equal to multiply(x,gmat) where gmat is an internally calculated MxM matrix. The quantity() and wavelength() are preserved.
Note that gmat is not the same as the the MxM *adaptation matrix*. To inspect gmat execute summary() on the returned object. If method is 'scaling' then gmat is diagonal and the diagonal entries are the M gain factors needed to achieve the calibration.
Useful data is attached as attribute "calibrate".

Note

Chromatic adaptation transforms, such as 'Bradford', do not belong in the realm of spectra, for this is not really a spectral calculation. For more about this subject see the explanation in *Digital Color Management*, Chapter 15 - Myths and Misconceptions. These sophisticated adaptation transforms are provided in calibrate() because it is possible and convenient.

References

ASTM E308-01. Standard Practice for Computing the Colors of Objects by Using the CIE System. 2001.

CIE 15: Technical Report: Colorimetry, 3rd edition. CIE 15:2004.

Edward J. Giorgianni and Thomas E. Madden. **Digital Color Management: Encoding Solutions.** 2nd Edition John Wiley. 2009. Chapter 15 - Myths and Misconceptions.

See Also

is.regular(), multiply(), quantity(), wavelength(), colorSpec, summary(), illuminantE(), neutralMaterial(), product()

Examples

```
wave = 380:780

# make an art gallery illuminated by illuminant A, and with tristimulus XYZ as output
gallery = product( A.1nm, 'artwork', xyz1931.1nm, wave=wave )

#  calibrate simplistically,
```

```
#  so the perfect reflecting diffuser has the standard XYZ coordinates for Illuminant A
#  using the convention that Y=100 (instead of Y=1)
A = 100 * spacesXYZ::standardXYZ('A')
A
##         X   Y      Z
##  A 109.85 100 35.585


gallery.cal1 = calibrate( gallery, response=A, method='scaling' )

#  calibrate following the ASTM and CIE recommendation
gallery.cal2 = calibrate( gallery, response=c(NA,100,NA), method='scaling' )

#   make the Perfect Reflecting Diffuser for testing
prd = neutralMaterial( 1, wave=wave ) ; specnames(prd) = 'PRD'

#   compare responses to the PRD for gallery.cal1 and gallery.cal2
white.1 = product( prd, gallery.cal1 )
white.2 = product( prd, gallery.cal2 )
white.1 ; white.2 ; white.1 - white.2

##           X   Y      Z
##  PRD 109.85 100 35.585
##             X   Y        Z
##  PRD 109.8488 100 35.58151
##                X             Y           Z
##  PRD 0.001210456 -2.842171e-14 0.003489313


# make an RGB flatbead scanner from illuminant F11 and a Flea2 camera
scanner = product( subset(Fs.5nm,'F11'), 'paper', Flea2.RGB, wave='auto')
# adjust RGB gain factors (white balance) so the perfect reflecting diffuser yields RGB=(1,1,1)
scanner = calibrate( scanner )

# same flatbead scanner, but this time with some "white headroom"
scanner = product( subset(Fs.5nm,'F11'), 'paper', Flea2.RGB, wave='auto' )
scanner = calibrate( scanner, response=0.95 )
scanner
```

`canonicalOptimalColors`

compute the Canonical Optimal Colors

Description

Consider a **colorSpec** object `x` with `type` equal to `'responsivity.material'`. The set of all possible material reflectance functions (or transmittance functions) is convex, closed, and bounded (in any reasonable function space), and this implies that the set of all possible output responses from `x` is also convex, closed, and bounded. The latter set is called the *object-color solid* or *Rösch Farbkörper* for `x`. A color on the boundary of the *object-color solid* is called an *optimal color*

for x. The corresponding transmittance spectrum is called an *optimal spectrum* for x. The special points **W** (the response to the perfect reflecting diffuser) and **0** (the response to the perfect absorbing diffuser) are optimal.

Currently the function only works if the number of spectra in x is 3 (e.g. RGB or XYZ). In this case the *object-color solid* is a zonohedron whose boundary is the union of parallelograms, which may be coplanar. These parallelograms are indexed by distinct pairs of the wavelengths of x; if x has N wavelengths, then there are N*(N-1) parallelograms. The center of each parallelogram is called a *canonical optimal color*. Interestingly, the special points **W** and **0** are not canonical.

Usage

```
## S3 method for class 'colorSpec'
canonicalOptimalColors( x, lambda, spectral=FALSE )
```

Arguments

x	a **colorSpec** object with type equal to 'responsivity.material' and 3 spectra
lambda	a numeric Mx2 matrix whose rows contain distinct pairs of wavelengths of x, or a numeric vector that can be converted to such a matrix, by row. If any entry in lambda is not a wavelength of x, it is an error, and the corresponding returned data is set to NA.
spectral	if TRUE, the function returns a **colorSpec** object with the optimal spectra, see **Value**.

Details

The 3 responsivities are regarded not as continuous functions, but as step functions. This implies that the color solid is a zonohedron. In the preprocessing phase the zonohedral representation is calculated. The faces of the zonohedron are either parallelograms, or *compound faces* that can be partitioned into parallelograms. The centers of all these parallelograms are the canonical optimal colors.
The optimal spectra take value 1/2 at the 2 given wavelengths, and 0 or 1 elsewhere. If the 2 wavelengths are λ_1 and λ_2, and $\lambda_1 < \lambda_2$ then the spectrum is approximately a bandpass filter. If the 2 wavelengths are swapped, then the spectrum is "flipped" and is approximately a bandstop filter.
If the responsivities of x at wavelengths λ_1 and λ_2 are multiples of each other (or 0), it is an error, and the corresponding returned data is set to NA.

Value

If argument spectral=FALSE, canonicalOptimalColors() returns a data.frame with a row for each row in lambda. The columns in the output are:

lambda	the given matrix argument lambda
optimal	the computed optimal colors - an Mx3 matrix
transitions	the number of transitions in the optimal spectrum, this is a positive even number

If `rownames(lambda)` is not `NULL`, they are copied to the row names of the output.

If argument `spectral=TRUE`, it returns a **colorSpec** object with quantity `'transmittance'`. This object contains the optimal spectra, and the above-mentioned `data.frame` can then be obtained by applying `extradata()` to the returned object.

In case of global error, the function returns `NULL`.

References

Centore, Paul. *A zonohedral approach to optimal colours*. **Color Research & Application**. Vol. 38. No. 2. pp. 110-119. April 2013.

Logvinenko, A. D. An object-color space. **Journal of Vision**. 9(11):5, 1-23, (2009). `https://jov.arvojournals.org/article.aspx?articleid=2203976` doi:10.1167/9.11.5.

Schrödinger, E. (1920). Theorie der Pigmente von grösster Leuchtkraft. **Annalen der Physik**. 62, 603-622.

West, G. and M. H. Brill. Conditions under which Schrödinger object colors are optimal. **Journal of the Optical Society of America**. 73. pp. 1223-1225. 1983.

See Also

`probeOptimalColors()`, `bandRepresentation()`, `scanner.ACES`, `extradata()`, `type`, vignette **Convexity and Transitions**

Examples

```
wave    = seq(400,700,by=5)
D50.eye = product( D50.5nm, 'material', xyz1931.1nm, wavelength=wave )
canonicalOptimalColors( D50.eye, c(500,600, 550,560, 580,585) )
##    lambda.1 lambda.2   optimal.x   optimal.y   optimal.z transitions
##  1      500      600 47.02281830 80.07281030  4.33181530           2
##  2      550      560  5.18490614 10.09045773  0.06121505           2
##  3      580      585 26.91247649 21.49031008  0.03457904           6
```

---

chop *chop spectra into low and high parts*

Description

chop all spectra in a **colorSpec** object into low and high parts at a blending interval

Usage

```
## S3 method for class 'colorSpec'
chop( x, interval, adj=0.5 )
```

Arguments

x	a **colorSpec** object
interval	a numeric vector with length 2 giving the endpoints of the interval, in nm
adj	a number in [0,1] defining weights of low and high parts over the interval

Details

For each spectrum, the low and high parts sum to the original spectrum. The low part vanishes on the right of the interval, and the high part vanishes on the left.

Value

`chop(x)` returns a **colorSpec** object with twice the number of spectra in `x` and with `organization` equal to `'matrix'`. The names of the new spectra are formed by appending ".lo" and ".hi" to the original spectrum names.

See Also

`organization`,

Examples

```
# chop blue butane flame into diatomic carbon and hydrocarbon parts
path = system.file( "extdata/sources/BlueFlame.txt", package="colorSpec" )
blueflame = readSpectra( path, seq(375,650,0.5) )
plot( chop( blueflame, interval=c(432,435), adj=0.8 ) )

# chop 'white' LED into blue and yellow parts
path = system.file( "extdata/sources/Gepe-G-2001-LED.sp", package="colorSpec" )
LED = readSpectra( path )
plot( chop( LED, c(470,495) ) )
```

---

colorSpec *constructing and testing colorSpec Objects*

Description

The function `colorSpec()` is the constructor for **colorSpec** objects.

`is.colorSpec()` tests whether an object is a valid **colorSpec** object.
`as.colorSpec()` converts other variables to a **colorSpec** object, and is designed to be overridden by other packages.

Usage

```
colorSpec( data, wavelength, quantity='auto', organization='auto', specnames=NULL )

is.colorSpec(x)

## Default S3 method:
as.colorSpec( ... )
```

Arguments

data	a vector or matrix of the spectrum values. In case data is a vector, there is a single spectrum and the number of points in that spectrum is the length of the vector. In case data is a matrix, the spectra are stored in the columns, so the number of points in each spectrum is the number of rows in data, and the number of spectra is the number of columns in data. It is OK for the matrix to have only 0 or 1 column.
wavelength	a numeric vector of wavelengths for all the spectra, in nm. The length of this vector must be equal to NROW(data). The sequence must be increasing. The wavelength vector can be changed after construction.
quantity	a character string giving the quantity of all spectra in data; see quantity for a list of possible values. In case of 'auto', a guess is made from the specnames. The quantity can be changed after construction.
organization	a character string giving the desired organization of the returned **colorSpec** object. In case of 'auto', the organization is 'vector' or 'matrix' depending on data. Other possible organizations are 'df.col' or 'df.row'; see organization for discussion of all 4 possible organizations. The organization can be changed after construction.
specnames	a character vector with length equal to the number of spectra, and with no duplicates. If specnames is NULL and data is a vector, then specnames is set to deparse(substitute(data)). If specnames is NULL and data is a matrix, then specnames is set to colnames(data). If specnames is *still* not a character vector with the right length, or if there are duplicate names, then specnames is set to 'S1', 'S2', ... with a warning message. The specnames vector can be changed after construction.
x	an R object to test for being a valid **colorSpec** object.
...	arguments for use in other packages.

Details

Under the hood, a **colorSpec** object is either a vector, matrix, or data.frame. It is of S3 class 'colorSpec' with these extra attributes:

wavelength a numeric vector of wavelengths for all the spectra. If the organization of the object is 'df.col', then this is absent.

quantity a character string that gives the physical quantity of all spectra, see quantity() for a list of possible values.

`metadata` a named list for user-defined data. The names `'path'`, `'header'` and `'date'` are already reserved; see `metadata()`.

`step.wl` step between adjacent wavelengths in nm. This is assigned only when the wavelengths are regular; see `is.regular()`.

`specname` only assigned when the `organization` is `'vector'`, in which case it is equal to the single character string name of the single spectrum. Note that the word `specname` is singular. Also see `specnames()`.

And there are a few more attributes that exist only in special cases; see the **colorSpec User Guide**.

Value

`colorSpec()` returns a **colorSpec** object, or `NULL` in case of ERROR. Compare this function with `stats::ts()`.

`is.colorSpec()` returns `TRUE` or `FALSE`. It does more than check the `class`, and also checks `wavelength`, `quantity`, and `organization`. If `FALSE`, it logs (at `loglevel='DEBUG'`) the reason that `x` is invalid.

`as.colorSpec.default()` issues an ERROR message and returns `NULL`

See Also

`wavelength`, `quantity`, `organization`, `metadata`, `step.wl`, `specnames`, `is.regular`, `coredata`

Examples

```
#  make a synthetic Gaussian bandpass filter

center = 600
wave   = 400:700
trans  = exp( -(wave-center)^2 / 20^2 )

filter.bp   = colorSpec( trans, wave, quantity='transmittance', specnames='myfilter' )

organization( filter.bp )  # returns:  "vector"

# and now plot it
plot( filter.bp )
```

---

computeADL *compute ADL coordinates by ray tracing*

Description

Consider a **colorSpec** object `x` with `type` equal to `responsivity.material`. The set of all possible material reflectance functions (or transmittance functions) that take the value 0 or 1, and with 2 or 0 transitions is called the *2-transition spectrum space*. When there are 2 transitions, there are 2 types of spectra: *bandpass* and *bandstop*. When there are 0 transitions, the spectrum is either identically

0 or identically 1. When x is applied to this space, the corresponding surface in response space is called the *2-transition surface*. The special points **0** and **W** (the response to the perfect reflecting diffuser) are on this surface. The surface is symmetrical about the neutral gray midpoint **G=W/2**. Following *West and Brill*, colors on the surface are called *Schrödinger colors*. For details see: `zonohedra::raytrace2trans()`.

This surface is a subset of the *object-color solid* or *Rösch Farbkörper* for x. Points on the boundary of of the solid are called *optimal colors*, see `probeOptimalColors()`. In most cases, a point on the 2-transition surface (a Schrödinger color) is an optimal color, but this is not always true, see the vignette **Convexity and Transitions**.

Now consider a color response **R** not equal to **G**. There is a ray based at **G** and passing through **R** that intersects the 2-transition surface at a *Schrödinger color* **B** with Logvinenko coordinates (δ, ω). If these 2 coordinates are combined with α, where **R** = **G** + α(**B**-**G**), it yields the *Logvinenko coordinates* (α, δ, ω) of **R**. These coordinates are also denoted by ADL; see **References**. A color is Schrödinger iff α=1.

The coordinates of **0** are (α, δ, ω)=(1,0,NA). The coordinates of **W** are (α, δ, ω)=(1,1,NA). The coordinates of **G** are (α, δ, ω)=(0,NA,NA).

Usage

```
## S3 method for class 'colorSpec'
computeADL( x, response )
```

Arguments

x	a **colorSpec** object with type equal to `responsivity.material` and 3 spectra
response	a numeric Nx3 matrix with responses in the rows, or a numeric vector that can be converted to such a matrix, by row.

Details

For each response, a ray is computed and the ray tracing is done by `probeOptimalColors()`.

Value

`computeADL()` returns a `data.frame` with a row for each response. The columns in the data frame are:

response	the input response vector
lambda	lambda.1 and lambda.2 at the 2 transitions, in nm. lambda.1 < lambda.2 => bandpass, and lambda.1 > lambda.2 => bandstop.
ADL	the computed ADL coordinates of the response vector
omega	the reparameterized λ in the interval [0,1]; see *Logvinenko*

If an individual ray could not be traced, the row contains NA in appropriate columns.
In case of global error, the function returns NULL.

WARNING

Since this function is really a simple wrapper around `zonohedra::raytrace2trans()`. Please see the warnings about star-shaped regions there.

References

Logvinenko, A. D. An object-color space. **Journal of Vision**. 9(11):5, 1-23, (2009). https://jov.arvojournals.org/article.aspx?articleid=2203976. doi:10.1167/9.11.5.

Godau, Christoph and Brian Funt. XYZ to ADL: Calculating Logvinenko's Object Color Coordinates. Proceedings Eighteenth IS&T Color Imaging Conference. San Antonio. Nov 2009.

West, G. and M. H. Brill. Conditions under which Schrödinger object colors are optimal. **Journal of the Optical Society of America**. 73. pp. 1223-1225. 1983.

See Also

`type()`, `probeOptimalColors()`, `zonohedra::raytrace2trans()`, vignette **Convexity and Transitions**

Examples

```
D50.eye = product( D50.5nm, 'varmat', xyz1931.1nm, wave=seq(360,830,by=5) )
computeADL( D50.eye, c(30,50,70) )

##    response.X response.Y response.Z lambda.1 lambda.2 ADL.alpha ADL.delta  ADL.lambda
##  1         30         50         70 427.2011 555.5262 0.7371480 0.5384106 473.3594244

##     omega
## 0.3008816

## since alpha < 1, XYZ=c(30,50,70) is *inside* the Schrodinger color surface of D50.eye
```

---

computeCCT *Compute Correlated Color Temperature (CCT) of Light Spectra*

Description

Compute the CCT, in K, of a **colorSpec** object with `type` equal to `'light'`. The package **spacesXYZ** must be installed.

Usage

```
## S3 method for class 'colorSpec'
computeCCT( x, isotherms='robertson', locus='robertson', strict=FALSE )
```

Arguments

x	an **colorSpec** R object with type equal to 'light', and M spectra
isotherms	A character vector whose elements match one of the available isotherm families: 'robertson', 'mccamy', and 'native'. Matching is partial and case-insensitive. When more than one family is given, a matrix is returned, see **Value**. When isotherms='native' the isotherms are defined implicitly as lines perpendicular to the locus, see **Details** in spacesXYZ::CCTfromXYZ(). The character NA (NA_character_) is taken as a synonym for 'native'.
locus	valid values are 'robertson' and 'precision', see above. Matching is partial and case-insensitive.
strict	The CIE considers the CCT of a chromaticity uv to be meaningful only if the distance from uv to the Planckian locus is less than or equal to 0.05 [in CIE UCS 1960]. If strict=FALSE, then this condition is ignored. Otherwise, the distance is computed along the corresponding isotherm, and if it exceeds 0.05 the returned CCT is set to NA.

Details

In computeCCT(), for each spectrum, XYZ is computed using xyz1931.1nm, and the result passed to spacesXYZ::CCTfromXYZ(). If the quantity of x is 'photons' (actinometric) each spectrum is converted to 'energy' (radiometric) on the fly.

Value

computeCCT() returns a numeric vector of length M, where M is the number of spectra in x. The vector's names is set to specnames(x).
If the type of x is not 'light', then a warning is issued and all values are NA_real_.

References

McCamy, C. S. *Correlated color temperature as an explicit function of chromaticity coordinates*. Color Research & Application. Volume 17. Issue 2. pages 142-144. April 1992.

Robertson, A. R. Computation of correlated color temperature and distribution temperature. Journal of the Optical Society of America. 58. pp. 1528-1535 (1968).

Wyszecki, Günther and W. S. Stiles. **Color Science: Concepts and Methods, Quantitative Data and Formulae, Second Edition.** John Wiley & Sons, 1982. Table 1(3.11). pp. 227-228.

See Also

type(), quantity(), xyz1931, planckSpectra(), specnames(), spacesXYZ::CCTfromXYZ()

Examples

```
computeCCT( D65.1nm )                      # returns 6502.068
computeCCT( D65.1nm, isotherms='native' )  # returns 6503.323
computeCCT( A.1nm )                        # returns 2855.656
```

```
computeCCT( A.1nm, isotherms='native' )     # returns 2855.662
computeCCT( A.1nm, isotherms='mccamy' )     # returns 2857.188

moon = readSpectra( system.file( "extdata/sources/moonlight.txt", package='colorSpec' ) )
computeCCT( moon )                  # returns 4482.371
```

`computeCRI` *Compute Color Rendering Index (CRI) of Light Spectra*

Description

Compute the CIE 1974 color rendering index (CRI) of a light spectrum, called the *the test illuminant*.
From the given spectrum a *reference illuminant* is selected with the same CCT (Correlated Color Temperature). A selected set of 8 color samples is rendered in XYZ (1931) with both illuminants and 8 color differences are computed in CIE 1964 UVW color space. For each color sample a CRI is computed, where 100 is a perfect color match. The final CRI is the average of these 8 CRI values.

Usage

```
## S3 method for class 'colorSpec'
computeCRI( x, CCT=NULL, adapt=TRUE, tol=5.4e-3, attach=FALSE )

## S3 method for class 'colorSpec'
computeCRIdata( x, CCT=NULL, adapt=TRUE, tol=5.4e-3 )
```

Arguments

`x`	a non-empty **colorSpec** object with `type` equal to `'light'`. The spectra in `x` are the *test illuminants*. For `computeCRIdata()` there must be exactly 1 spectrum. For `computeCRI()` there can be multiple spectra.
`CCT`	the Correlated Color Temperature of the reference illuminant. If `CCT=NULL` (the default) then `CCT` is computed from the test illuminant. The user can override this with a target CCT, e.g. the advertised CCT for a particular light bulb. For `computeCRIdata()` there must be exactly one `CCT`. For `computeCRI()` the length of the vector must be either 1 or the number of spectra in `x`.
`adapt`	if `TRUE`, then a special chromatic adaption is performed, see **Details**
`tol`	for the CRI to be meaningful the chromaticities of the test and reference illuminants must be sufficiently close in the CIE 1960 uniform chromaticity space. If the tolerance is exceeded, the CRI is set to `NA_real_`. The default `tol=5.4e-3` is the one recommended by the CIE, but the argument allows the user to override it. To ignore this test, set `tol=Inf`.
`attach`	if `TRUE` and there is exactly 1 spectrum in `x`, then the list of intermediate calculations returned by `computeCRIdata()` is attached to the returned number, as attribute `data`. This attached list includes data for all 14 color samples, though only the first 8 are used to compute the CRI. If there is not exactly 1 spectrum in `x`, then `attach` is ignored.

Details

If not NULL, the CCT of x is computed by computeCCT() with default options.
When computing XYZs, the wavelengths of x and the color matching functions of xyz1931.1nm are used.
If adapt is TRUE the 8 color sample uv points are chromatically adapted from the test illuminant to the reference illuminant using a special von Kries type transformation; see *Oleari* and *Wikipedia*. The color sample UVW values are computed with the reference illuminant.
If adapt is FALSE the 8 color sample uv points are *not* chromatically adapted, and the color sample UVW values are computed with the test illuminant.

Value

computeCRI() returns a vector of CRI values with length equal to the number of spectra in x. All values are ≤ 100. In case of ERROR the CRI value is NA_real_. If attach is TRUE and x has exactly one spectrum, a large list of intermediate calculations is attached to the returned number.

computeCRIdata() returns a list of intermediate calculations, for the single spectrum in x. The items in the list are:

CCT	the Correlated Color Temperature of the reference illuminant.
illum.ref	the reference illuminant, which is a **colorSpec** object.
table1	a data frame with 2 rows, with CIE data for the test and reference illuminants (see below).
Delta_uv	the distance between the illuminants in the CIE 1960 uniform chromaticity space.
table2	a data frame with 14 rows, with CIE XYZ data of the 14 color samples.
table3	a data frame with 14 rows, with CIE uv data of the 14 color samples. If argument adapt is FALSE, then table3=
table4	a data frame with 14 rows, with UVW data of the 14 color samples, including the CRI of each sample.

The columns of table1 are:

XYZ	the CIE XYZ of the 2 illuminants.
xy	the CIE xy of the 2 illuminants.
uv	the CIE 1960 uv of the 2 illuminants. Delta_uv is the distance between these 2 points.

The first row of table1 is the given test illuminant, and the second row is the reference illuminant. The initial letter of the rowname of the reference indicates the type: the letter P means a Planckian illuminant, and the letter D means a Daylight illuminant.

The columns of table2 are:

referen	the CIE XYZ of the color sample, as illuminated by the reference illuminant.
test	the CIE XYZ of the color sample, as illuminated by the test illuminant.

The columns of table3 are:

before	the CIE 1960 uv of the color sample, as illuminated by the test illuminant.
after	the before values, after adaptation to the reference illuminant.

`difference` `after - before`

The columns of `table4` are:

`referen`	the UVW of the color sample, as illuminated by the reference illuminant.
`test`	the UVW of the color sample, as illuminated by the test illuminant.
`DeltaE`	the distance between the test and reference UVWs
`CRI`	the CRI of the color sample, which is a function of `DeltaE`.

The final CRI is the average of the `CRI` of the first 8 samples in `table4`, and the remaining samples are ignored.

Source

The reflectance spectra of the 14 color samples are taken from:
http://www.lrc.rpi.edu/programs/nlpip/lightinganswers/lightsources/scripts/NLPIP_LightSourceColor_Script.m

The wavelength vector is 360nm to 830nm with 5nm step. The same data over 380nm to 780nm is in Appendix 7 of *Hunt and Pointer.*

References

Oleari, Claudio, Gabriele Simone. **Standard Colorimetry: Definitions, Algorithms and Software**. John Wiley. 2016. pp. 465-470.

Günther Wyszecki and W. S. Stiles. **Color Science: Concepts and Methods, Quantitative Data and Formulae, Second Edition**. John Wiley & Sons, 1982. Table 1(3.11). p. 828.

Wikipedia. **Color rendering index**. https://en.wikipedia.org/wiki/Color_rendering_index

Hunt, R. W. G. and M. R. Pointer. **Measuring Colour**. 4th edition. John Wiley & Sons. 2011. Appendix 7.

See Also

`type()`, `xyz1931`, `computeCCT()`

Examples

```
computeCRI( Fs.5nm )
##      F1      F2      F3      F4  F5  F6      F7      F8      F9     F10     F11     F12
## 75.82257 64.15195 56.68144 51.36348  NA  NA 90.18452 95.50431 90.29347 81.03585 82.83362 83.06373

computeCRI( Fs.5nm, adapt=FALSE )
##      F1      F2      F3      F4  F5  F6      F7      F8      F9     F10     F11     F12
## 77.73867 65.38567 57.20553 50.65979  NA  NA 90.18551 95.96459 90.27063 82.86106 82.86306 83.10613

F2 =  subset(Fs.5nm,'F2')
computeCRI( F2 )
##       F2
```

```
## 64.15195

computeCRI( F2, CCT=4200 )
##       F2
## 63.96502

computeCRIdata( F2 )   # returns a very large list, ending with CRI = 64.15195

mean( computeCRIdata( F2 )$table4$CRI[1:8] ) ==  computeCRI( F2 )
##   F2
## TRUE
```

`computeSSI` *Compute the Spectrum Similarity Index of light spectra*

Description

Compute the Spectrum Similarity Index (SSI), an index between 0 and 100, of a **colorSpec** object with `type` equal to `'light'`. It compares a test spectrum with a reference spectrum (an ideal). The value 100 means a perfect match to the reference, and a smaller value mean a poorer match (similar to CRI). Only values in the interval [375,675] nm are used; for details see *Holm*.

Usage

```
## S3 method for class 'colorSpec'
computeSSI( x, reference=NULL, digits=0, isotherms='mccamy', locus='robertson' )
```

Arguments

`x`	a **colorSpec** object with `type` equal to `'light'`, and M test spectra
`reference`	a **colorSpec** object with `type` equal to `'light'`, and either 1 or M reference spectra. `reference` can also be `NULL` (the default), which means to generate each reference spectrum from the corresponding test spectrum.
`digits`	the number of digits after the decimal point in the returned vector. According to *Holm* the output should be rounded to the nearest integer, which corresponds to `digits=0`. To return full precision, set `digits=Inf`.
`isotherms`	this is only used when `reference=NULL`. It is passed to `computeCCT()` in order to compute the CCT of each test spectrum.
`locus`	this is only used when `reference=NULL`. It is passed to `computeCCT()` in order to compute the CCT of each test spectrum.

Details

If `reference` contains a single spectrum, then each test spectrum is compared to that one reference spectrum. If `reference` contains M spectra, then the i'th test spectrum is compared to the i'th reference spectrum.

If `reference=NULL` then for each test spectrum the CCT is computed and used to compute a reference spectrum with the same CCT. It is either a Planckian (black-body) or daylight illuminant, see *Holm* for details. The test spectrum and auto-computed reference spectrum are then compared.

Value

computeSSI() returns a numeric vector of length M, where M is the number of spectra in x. The vector's names is set from specnames(x) and a compact code for the corresponding reference spectrum.

If the type of x is not 'light', or reference is not valid, then the function returns NULL.

References

J. Holm and T. Maier and P. Debevec and C. LeGendre and J. Pines and J. Erland and G. Joblove and S. Dyer and B. Sloan and J. di Gennaro and D. Sherlock. *A Cinematographic Spectral Similarity Index*. SMPTE 2016 Annual Technical Conference and Exhibition. pp. 1-36. (2016).

See Also

type(), computeCCT(), planckSpectra(), daylightSpectra(), specnames()

Examples

```
computeSSI( planckSpectra( 1000*(2:6) )  )
##  P2000_SSI[2027K] P3000_SSI[3057K] P4000_SSI[D4063] P5000_SSI[D5061] P6000_SSI[D6063]
##                99               98               93               92               92
```

---

convolvewith *Convolve each spectrum in a colorSpec object with a kernel*

Description

This function convolves each spectrum in a colorSpec object with a kernel of odd length. Its primary purpose is to correct raw spectrometer data (with positive instrumental bandwidth) to have bandwidth=0. Two popular correction kernels for this, with lengths 3 and 5, are built-in options, see **Details**.

Usage

```
## S3 method for class 'colorSpec'
convolvewith( x, coeff )
```

Arguments

x	a **colorSpec** object with N wavelengths
coeff	a convolution kernel of odd length. The center entry of this vector is taken as index 0 in the convolution. coeff can also be the string 'SS3' which means to apply the Stearns&Stearns bandwidth correction kernel coeff=c(-1,14,-1)/12, see **Details**. coeff can also be the string 'SS5' which means to apply the Stearns&Stearns bandwidth correction kernel coeff=c(1,-12,120,-12,1)/98, see **Details**.

Details

The built-in kernels, `'SS3'` and `'SS5'`, were derived by *Stearns & Stearns* under specific hypotheses on the spectrometer profile, bandpass, and pitch; see **References**.
Missing values at both ends are filled by copying from the nearest valid value.
The function creates a function calling `stats::filter()` and passes that function to `applyspec()`.

Value

a **colorSpec** object with the same dimensions, `wavelength`, `quantity`, and `organization`. If `coeff` is invalid it is an ERROR and `convolvewith()` returns `NULL`.

References

Stearns, E.I., Stearns R.E. An example of a method for correction radiance data for bandpass error. *Color Research and Application*. 13-4. 257-259. 1988.

Schanda, Janos. CIE Colorimetry, in *Colorimetry: Understanding the CIE System*. Wiley Interscience. 2007. p. 124.

Oleari, Claudio, Gabriele Simone. *Standard Colorimetry: Definitions, Algorithms and Software*. John Wiley. 2016. p. 309.

See Also

`quantity`, `wavelength`, `linearize`, `applyspec`, `organization`

`coredata` *Extract the Core Data of a colorSpec Object*

Description

functions for extracting the core data contained in a **colorSpec** object.

Usage

```
## S3 method for class 'colorSpec'
coredata( x, forcemat=FALSE )

## S3 method for class 'colorSpec'
as.matrix( x, ... )
```

Arguments

`x`	a **colorSpec** object
`forcemat`	if `x` has only 1 spectrum, return a matrix with 1 column instead of a vector
`...`	extra arguments ignored

Value

coredata	If x has organization equal to 'vector' then it returns x, unless forcemat is TRUE when it returns x as a matrix with 1 column. If x has any other organization then it returns a matrix with spectra in the columns. All of the **colorSpec** attributes are stripped except the column names, and the row names are set to as.character(wavelength(x)).
as.matrix	a wrapper for coredata with forcemat=TRUE

See Also

organization

---

cs.options *Functions to set and retrieve colorSpec package options*

Description

colorSpec has one option. The option is stored in the **R** global list and is:

colorSpec.stoponerror

For details on what it does see logging.

It can be set using the built-in function base::options(). When R starts up, an option can be set using a call to base::options() in the file **Rprofile.site**. If **colorSpec** is later loaded, the value of the option will not be changed. If an option has not been assigned, then it is created with a default value.

The little helper function cs.options() makes setting the options a little easier in a few ways:

- it automatically prepends the string 'colorSpec.'
- partial matching of the option name is enabled
- an error is issued when the option value has the wrong type

Usage

```
cs.options( ... )
```

Arguments

...	named arguments are set; unnamed arguments are ignored with a warning. See **Examples**.

Value

returns a list with all the **colorSpec** options.

See Also

logging, options

Examples

```
cs.options( stop=TRUE )             # no problems, no warning
cs.options( stop='TRUE' )           # warns that value has the wrong type
cs.options( stop=FALSE, "DEBUG" )   # warns that the 2nd argument has no name
```

D50 *Standard Illuminant D50 (1964)*

Description

`D50.5nm` standard Illuminant D50, from 300 to 830 nm at 5 nm intervals.

Format

A **colorSpec** object organized as a vector, with 107 data points and `specnames` equal to `'D50'`.

Details

This spectrum is not copied from a table from a CIE publication, though it does match such a table. It is computed using the function `daylightSpectra()` by following the special CIE recipe given in the **References**. The temperature is set to (14388/14380) * 5000 = 5002.781 Kelvin. The coefficients of the daylight components S_0, S_1, and S_2 are rounded to 3 decimal places. This linear combination is computed at 10nm intervals and then linearly interpolated to 5nm intervals. The result is normalized to value 1 at 560nm (instead of the usual 100), and finally rounded to 5 decimal places. See **Examples**.

References

Günther Wyszecki and W.S. Stiles. **Color Science : Concepts and Methods, Quantitative Data and Formulae**. Second Edition. Wiley-Interscience. 1982. Table I(3.3.4) pp. 754-758

CIE 15: Technical Report: Colorimetry, 3rd edition. CIE 15:2004. Table T.1, pp 30-32, and Note 5 on page 69.

Schanda, Janos. CIE Colorimetry, in *Colorimetry: Understanding the CIE System*. Wiley Interscience. 2007. p. 42.

See Also

`ABC` , `D65` , `daylightSpectra`

Examples

```
#   the CIE recipe for computing D50.5nm
correction  = 14388 / 14380     # note 5, page 69 in CIE 15:2004
D50.10nm   = daylightSpectra( correction*5000, wavelength=seq(300,830,by=10), roundMs=TRUE )
D50.5nm     = resample( D50.10nm, seq(300,830,by=5), method='linear' )
D50.5nm     = round( D50.5nm, 5 )

summary( D50.5nm )
white.point = product( D50.5nm, xyz1931.1nm, wave='auto' )
```

D65 *Standard Illuminant D65 (1964)*

Description

D65.1nm	standard Illuminant D65, 300 to 830 nm at 1 nm intervals
D65.5nm	standard Illuminant D65, 380 to 780 nm at 5 nm intervals

Format

Each is a **colorSpec** object organized as a vector, with specnames equal to 'D65'.

Details

Both of these have been divided by 100, to make the values at 560nm equal to 1 instead of 100.

Source

http://www.cvrl.org

References

Günther Wyszecki and W.S. Stiles. **Color Science : Concepts and Methods, Quantitative Data and Formulae**. Second Edition. Wiley-Interscience. 1982. Table I(3.3.4) pp. 754-758

ASTM E 308-01. Standard Practice for Computing the Colors of Objects by Using the CIE System. Table 3. pages 3-4.

See Also

ABC, D50, daylightSpectra , daylight

Examples

```
summary( D65.1nm )
white.point = product( D65.1nm, xyz1931.1nm, wave='auto' )
```

daylight *Standard Daylight Components*

Description

daylight1964	spectral components S_0, S_1, S_2; from 300 to 830 nm at 5 nm intervals
daylight2013	smoothed spectral components S_0, S_1, S_2; from 300 to 830 nm at 1 nm intervals

Format

Each is a **colorSpec** object organized as a matrix with 3 columns

S0	component 0, the mean power spectrum
S1	component 1, the 1st characteristic spectrum
S2	component 2, the 2nd characteristic spectrum

Source

http://www.cie.co.at/publ/abst/datatables15_2004/CIE_sel_colorimetric_tables.xls

http://vision.vein.hu/~schanda/CIE%20TC1-74/

References

Günther Wyszecki and W.S. Stiles. **Color Science : Concepts and Methods, Quantitative Data and Formulae**. Second Edition. Wiley-Interscience. 1982. Table V(3.3.4) p. 762.

Smoothing spectral power distribution of daylights. Zsolt Kosztyan and Janos Schanda. Color Research & Application. Volume 38, Issue 5, pages 316-321, October 2013.

CIE 15: Technical Report: Colorimetry, 3rd edition. CIE 15:2004. Table T.2, pp 33-34

JUDD, D.B., MACADAM, D.L. and WYSZECKI, G., with the collaboration of BUDDE, H.W, CONDIT, H.R, HENDERSON, S.T, and SIMONDS, J.L. Spectral distribution of typical daylight as a function of correlated color temperature. J Opt. Soc. Am. 54, 1031-1040, 1964.

Zsolt Kosztyan and Janos Schanda. Smoothing spectral power distribution of daylights. Color Research & Application. Volume 38, Issue 5, pages 316-321, October 2013.

See Also

D65, D50, daylightSpectra

Examples

```
summary( daylight1964 )
day1964 = daylightSpectra( c(5000,6500), comp=daylight1964 )
day2013 = daylightSpectra( c(5000,6500), comp=daylight2013 )

plot( day1964, col='black' )
plot( day2013, col='black', add=TRUE )
```

DisplayRGB *Compute Display RGB from Linear RGB*

Description

All RGB displays have a non-linear "gamma function" of some sort. This function converts from linear RGB to an RGB appropriate for the gamma function of the display; which is also called the *electro-optical conversion function* (EOCF).

Usage

```
DisplayRGBfromLinearRGB( RGB, gamma='sRGB' )
```

Arguments

RGB	linear RGB values organized as a vector or matrix of any size; all 3 channels are treated the same way so size does not matter
gamma	either the string 'sRGB' or a positive number giving the gamma of the display.

Value

The function first clamps the input RGB to the interval [0,1]. If gamma='sRGB' (not case-sensitive) it then maps [0,1] to [0,1] using the special piecewise-defined sRGB function, see *Wikipedia*. In case gamma is a positive number, the function raises all values to the power 1/gamma. The dimensions and names of the input are copied to the output.
In case of error, the function returns the clamped input values.

WARNING

This function is deprecated. New software should use spacesRGB::SignalRGBfromLinearRGB() instead.

Source

Wikipedia. **sRGB**. https://en.wikipedia.org/wiki/SRGB

See Also

RGBfromXYZ

Examples

```
DisplayRGBfromLinearRGB( c(0.2, 0.5) )
# [1] 0.4845292 0.7353570     #  this is display sRGB, in [0,1]

DisplayRGBfromLinearRGB( c(-0.1, 0.2, 0.5, 1), 2.2 )
# [1] 0.0000000 0.4811565 0.7297401 1.0000000    #  gamma=2.2

x = seq( 0, 1, len=101)
plot( x, DisplayRGBfromLinearRGB(x), type='l' )
```

`emulate` *modify a colorSpec responder to emulate (approximate) another responder*

Description

The two possible modifications are:

- pre-multiplication by a transmitting filter
- post-multiplication by a matrix

Both of these are optional. If neither of these modifications is enabled, the original `x` is returned.

Usage

```
## S3 method for class 'colorSpec'
emulate( x, y, filter=FALSE, matrix=TRUE )
```

Arguments

`x`	a **colorSpec** responder with M spectra, to be modified. The `type` must be `'responsivity.light'` or `'responsivity.material'`
`y`	a **colorSpec** responder with N spectra, to be emulated by a modified `x`. It must have the same `type` and `wavelength` vector as `x`
`filter`	enable filter pre-multiplication.
`matrix`	enable matrix post-multiplication. If `matrix=TRUE` then the computed matrix `A` is MxN.

Details

If `filter=FALSE` and `matrix=TRUE` then the returned value is `multiply(x,A)`, where the matrix `A` is computed to minimize the difference with `y`, in the least squares sense (Frobenius matrix norm). The function `ginv()` is used here.

If `filter=TRUE` and `matrix=FALSE` then the returned value is `product(filter,x)`, where the object `filter` is computed to minimize the difference with `y`, in the least squares sense (Frobenius matrix norm). This calculation is fairly straightforward, but requires that the responsivity of `x` does not vanish at any wavelength. It also requires that `M=N`. The computed filter may be unrealistic, i.e. the transmittance may be > 1. If this happens a `WARN` message is issued.

If `filter=TRUE` and `matrix=TRUE` then the returned value is `product(filter,multiply(x,A))`, where `(filter,A)` are chosen with the above minimization criterion. If `N=1` then we must have `M=1` as well; the calculation is trivial and the emulation is exact. If $N \geq 2$, the calculation is iterative - solving alternatively for `filter` and `A` until convergence. The function `ginv()` is used on each iteration. This is a bilinear optimization. If convergence fails, it is an error and the function returns `NULL`. If convergence succeeds, there is 1 degree of freedom in the `(filter,A)` pair. If one is scaled by a positive constant, the other can be scaled by the inverse, and the returned object is the same. The filter is scaled so the maximum transmittance is 1.

If `filter=FALSE` and `matrix=FALSE` then the original `x` is returned, with a `WARN` message.

Value

a **colorSpec** object close to `y`, as in **Details**. The `quantity` is the same as `y`. The `specnames()` are the same as those of `y`, except that `".em"` is appended to each one. The function attaches attribute `"emulate"`, whose value is a list containing `filter` and/or `A` as appropriate.

Examples

see the vignette **Emulation of one Camera by another Camera**

See Also

`wavelength`, `type`, `quantity`, `multiply`, `product`, `specnames`

`extradata` *extradata of a colorSpec object*

Description

Retrieve or set the extradata of a **colorSpec** object.

Usage

```
## S3 method for class 'colorSpec'
extradata(x)

## S3 replacement method for class 'colorSpec'
extradata(x,add=FALSE) <- value
```

Arguments

x	a **colorSpec** object with M spectra
value	a data.frame with M rows. It is OK for value to have 0 columns, and value can also be NULL; see add.
add	If add is FALSE, then any existing extradata is discarded and replaced by value, except when value is NULL when x is left with no extradata. If add is TRUE, then value is appended to the existing extradata, except when value is NULL when x is left unchanged.

Details

If the organization of x is not 'df.row', then extradata cannot be stored in x and the assignment is ignored, with a warning. First change the organization to 'df.row', and *then* assign the extradata.

If the organization of x is 'df.row', but value does not have the right number of rows, the assignment is ignored, with a warning.

Value

extradata(x) returns a data.frame with M rows, where M is the number of spectra in x. The rownames are set to the specnames of x. If there is no extra data then the number of columns in this data.frame is 0.

Note

Do not confuse extradata and metadata.
metadata is unstructured data that is attached to the entire **colorSpec** object. extradata is structured data, with a row of data for each spectrum in the object.

See Also

metadata, organization

F96T12 *Photon Irradiance of F96T12 Fluorescent Bulb*

Description

F96T12
Sylvania F96T12 CW/VHO 215-Watt fluorescent bulb photon irradiance, measured with a LI-COR LI-1800 spectroradiometer, from 300 to 900 nm at 1 nm intervals.

Format

A **colorSpec** object organized as a vector, with 601 data points and specnames equal to 'F96T12'.

Details

The unit is (μmole of photons)$*sec^{-1} * m^{-2} * nm^{-1}$.

Source

Pedro J. Aphalo. https://www.mv.helsinki.fi/home/aphalo/photobio/lamps.html

See Also

ABC , D65 , daylightSpectra

Examples

```
sum( F96T12 )
# [1] 320.1132  photon irradiance, (micromoles of photons)*m^{-2}

sum( radiometric(F96T12) )
# [1] 68.91819  irradiance, watts*m^{-2}
```

Flea2.RGB *Flea2 Camera FL2-14S3C from Point Grey*

Description

`Flea2.RGB` an RGB responder to light, from 360 to 800 nm at 10 nm intervals

Format

A **colorSpec** object with `quantity` equal to `'energy->electrical'` and 3 spectra: `Red`, `Green`, and `Blue`.

Details

This data is read from the file **Flea2-spectral.txt** which was digitized from the plot in **Flea2-spectral.png**.

Source

https://store.imrnasia.com/fl2-14s3c-c/

See Also

quantity, vignette **Blue Flame and Green Comet**

Examples

```
#  Make a scanner from a tungsten source and a Flea2 camera
Flea2.scanner = product( A.1nm, "VARMATERIAL", Flea2.RGB, wavelength=420:680 )
Flea2.scanner = calibrate( Flea2.scanner )
```

Fluorescents — *Standard series F Illuminants F1, F2, F3, F4, F5, F6, F7, F8, F9, F10, F11, and F12*

Description

Fs.5nm contains 12 CIE Fluorescent Illuminants, from 380 to 780 nm, at 5nm intervals.

Format

Fs.5nm is a **colorSpec** object with 12 spectra. It is organized as a data frame with quantity equal to "energy".

Note

The series F illuminants do not seem to be normalized in a consistent way.

Source

http://www.rit-mcsl.org/UsefulData/Fluorescents.htm

See Also

ABC, D50, D65

Examples

```
#   plot only F4
plot( subset(Fs.5nm,"F4") )
```

HigherPasserines *Cone Fundamentals for the Higher Passerines*

Description

HigherPasserines Tetrachromatic Cone Fundamentals of Higher Passerine Birds

Format

A **colorSpec** object organized as a matrix with the 4 spectra:

UV	the UV wavelength responsivity
Short	the short wavelength responsivity
Medium	the medium wavelength responsivity
Long	the long wavelength responsivity

The wavelength is from 300 to 700 nm, at 1nm intervals.

Source

https://onlinelibrary.wiley.com/doi/full/10.1111/j.1095-8312.2005.00540.x

References

Endler & Mielke. Comparing entire colour patterns as birds see them. Biological Journal of the Linnean Society. Volume 86, Issue 4, pages 405-431, December 2005. Original Name of File: BIJ_540_Endler_Mielke_OnlineAppendix.txt.

See Also

lms2000

Examples

```
summary(HigherPasserines)
```

Hoya *standard Hoya filters*

Description

Hoya 4 standard Hoya filters; from 300 to 750 nm at 10nm intervals.

Format

A **colorSpec** object with quantity equal to 'transmittance' and 4 spectra:

R-60	long-pass red filter with cutoff about 600nm
G-533	band-pass green filter with peak about 533nm
B-440	band-pass blue filter with peak about 440nm
LB-120	Light-balancing Blue filter with mired shift equal to -120

Source

https://hoyaoptics.com/

See Also

quantity

Examples

```
#   compute response of ACES scanner to the Hoya filters
product( Hoya, scanner.ACES, wave='auto' )
```

inside *test whether points are inside the surface of* optimal colors

Description

Test points for being inside the surface of optimal colors, which is the boundary of the *object color solid*. It is essentially a wrapper around zonohedra::inside().

Usage

```
## S3 method for class 'colorSpec'
insideOptimalColors( x, p )
```

Arguments

x	a **colorSpec** object - with 1, 2, or 3 spectra
p	an NxM numeric matrix, where M is the number of spectra. The points to be tested are in the rows. p can also be a numeric vector that can be converted to such a matrix, by row.

Value

`insideOptimalColors()` returns a `data.frame` with N rows and these columns:

p	the given point
distance	the signed distance from the point to the boundary surface of optimal colors. When `distance` < 0, the point is in the interior, and the distance is the true Euclidean distance. For boundary points, `distance` is 0. When `distance` > 0, it may be larger than the true Euclidean distance, so `distance` is really a pseudo-distance.
inside	whether the point is inside the zonotope. `inside` = `distance<=0`

If the row names of p are unique, they are copied to the row names of the output.
In case of error, the function returns `NULL`.

See Also

`zonohedra::inside()`

Examples

```
wave = seq(400,800,by=4)

D50.eye = product( D50.5nm, 'varmat', xyz1931.1nm, wave=wave )

white = colSums( 4 * as.matrix(D50.eye) )

insideOptimalColors( D50.eye,  c(30,50,70,  0,0,0, white, white+1 )  )

##          p.1       p.2       p.3 inside  distance
##  1  30.00000  50.00000  70.00000   TRUE -7.528005
##  2   0.00000   0.00000   0.00000   TRUE  0.000000
##  3 101.24793 105.04021  86.52988   TRUE  0.000000
##  4 102.24793 106.04021  87.52988  FALSE  1.259912
```

`insideSchrodingerColors` *test whether points are inside the surface of Schrödinger colors*

Description

Consider a **colorSpec** object `x` with `type` equal to `'responsivity.material'`. The set of all possible material reflectance functions (or transmittance functions) that take the value 0 or 1, and with 2 or 0 transitions is called the *2-transition spectrum space*. When there are 2 transitions, there are 2 types of spectra: *bandpass* and *bandstop*. When there are 0 transitions, the spectrum is either identically 0 or identically 1. When `x` is applied to this space, the corresponding surface in response space is called the *2-transition surface*. The surface is closed. The special points **0** and **W** (the response to the perfect reflecting diffuser) are on this surface. The surface is symmetrical about the neutral gray midpoint **G=W/2**. Following *West and Brill*, colors on the surface are called *Schrödinger colors*. Denote the surface by S_2.

This function computes whether given points are inside S_2. It only supports `x` with 3 channels. It is essentially a wrapper around `zonohedra::inside2trans()`.

Usage

```
## S3 method for class 'colorSpec'
insideSchrodingerColors( x, p )
```

Arguments

`x`	a **zonohedron** object
`p`	an Nx3 numeric matrix. The points to be tested are in the rows. `p` can also be a numeric vector that can be converted to such a matrix, by row.

Details

If the surface has no self-intersections, the the definition of whether a point `p` is "inside" is fairly straightforward: it is where the linking number of `p` and the surface is non-zero. In fact, if it is non-zero then it must be +1 or -1. The *linking number* is analogous the *winding number* in 2D.

Unfortunately, there is currently no test for whether the surface *has* self-intersections, For a bad surface with self-intersections, the linking number might be any integer. Since there is no such test, we simply use the same non-zero linking number rule always.

The computed `linkingnumber` is returned so that the user can apply the non-zero rule, or the even-odd rule, as appropriate for their situation. These 2 rules are analogous to the two winding number rules used for polygons in computer graphics, see **Point in polygon**.

The case where a point is *on* the surface (i.e. the `distance` to the surface is 0) is problematic. The `linkingnumber` is then undefined, and we currently set `inside` to be undefined as well. Thus `inside` should be interpreted as *strictly inside*. However, in some situations, the user may want to consider `inside` to be `TRUE` in this problematic case. Or the user may want to consider points that are within a very small epsilon of the surface, where roundoff might have occurred, to have `inside=FALSE` or `inside=NA`. So the both the computed `linkingnumber` and `distance` are returned so the user can use them to make their own definition of what "inside" means.

Value

`insideSchrodingerColors()` returns a `data.frame` with N rows and these columns:

`p`	the given point
`distance`	the distance from the point to the surface. This is the true Euclidean distance, and not a signed distance. If the point is on the surface, the `distance` should be 0 up to numerical precision.
`linkingnumber`	the linking number of the point and the surface. If the point is *on* the surface (`distance==0`), the (mathematical) linking number is undefined, and the computed `linkingnumber` is `NA` (integer).
`inside`	whether the point is inside the surface; a logical. This is currently set to `linkingnumber != 0`. If the `linkingnumber` is `NA` (integer), then `inside` is `NA` (logical).
`timecalc`	the time to do the calculations, in seconds

If the row names of `p` are unique, they are copied to the row names of the output.
In case of error, the function returns `NULL`.

References

Point in polygon — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Point_in_polygon&oldid=1139808558. 2023.

Spivak, Michael. *A Comprehensive Introduction to Differential Geometry*. Volume 1. 3rd edition. Publish or Perish. 1999.

West, G. and M. H. Brill. Conditions under which Schrödinger object colors are optimal. **Journal of the Optical Society of America**. 73. pp. 1223-1225. 1983.

See Also

`zonohedra::inside2trans()`

Examples

```
D50.eye = product( D50.5nm, 'varmat', xyz1931.1nm, wave=seq(360,830,by=5) )

insideSchrodingerColors( D50.eye,  c(30,50,70,  0,0,0)  )

##    p.1 p.2 p.3 distance linkingnumber inside     timecalc
##  1  30  50  70 7.516577             1   TRUE 4.858999e-04
##  2   0   0   0 0.000000            NA     NA 9.399839e-06
```

`interpolate` *interpolate spectra*

Description

interpolate along a 1-parameter path of spectra

Usage

```
## S3 method for class 'colorSpec'
interpolate( x, p, pout, pname=deparse(substitute(p)) )
```

Arguments

`x`	a **colorSpec** object, typically with multiple spectra
`p`	a numeric vector with `length(p)==numSpectra(x)`. The value `p[i]` is associated with the `i`'th spectrum in `x`.
`pout`	a numeric vector of parameter values at which interpolation of the spectra in `x` take place
`pname`	the name of the parameter `p`

Details

Each spectrum in `x` can be thought of as a point in a high-dimensional space, and each point has a real-valued parameter associated with it. The function performs natural spline interpolation on these points, one coordinate at a time. For each wavelength value it calls `spline` with `method='natural'`.

Value

`interpolate(x)` returns a **colorSpec** object `y` with a spectrum for each value in `pout`. The organization of `y` is `'df.row'`, and `extradata(y)` has a single column which is a copy of `pout`. The name of the column is `pname`. The names in `specnames(y)` are `<pname>=<pout>`. Other properties of `y`, e.g. `wavelength`, `quantity`, ..., are the same as `x`.
In case of ERROR, the function returns `NULL`.

See Also

`organization`, `wavelength`, `extradata`, `spline`

Examples

```
path = system.file( "extdata/stains/PhenolRed-Fig7.txt", package="colorSpec" )
wave = 350:650
phenolred = readSpectra( path, wavelength=wave )
pH = as.numeric( sub( '[^0-9]+([0-9]+)$', '\\1', specnames(phenolred) ) )
pHvec = seq(min(pH),max(pH),by=0.05)
phenolinterp = interpolate( phenolred, pH, pHvec )
```

`invert` *estimate spectra from responses, effectively inverting the operator from spectrum to response*

Description

Given a light responder (e.g. an eye or a camera), two light spectra that produce the same response from the responder are called *metamers* for that responder. Similarly, given a material responder (e.g. a scanner), two reflectance spectra that produce the same response from the responder are called *metamers* for that responder.

For a given responder and response, there are typically infinitely many *metamers*. The set of all of them is often called the *metameric suite*. The goal of the function `invert()` is to calculate a "good" metamer in the "suite". *Koenderink* calls this topic *inverse colorimetry*. In the case that the estimated spectrum is a reflectance spectrum, the topic is often called *reflectance estimation* or *reflectance recovery*, see *Bianco*.

The *centroid method*, which is the default and the featured method in this package, computes the centroid of the set of all the metamers (if any). The centroid is computed in an infinite-dimensional context and is expounded further in *Davis*.

The *Hawkyard method*, see *Hawkyard* and *Bianco*, has been around a long time. The centroid and Hawkyard methods have similarities, e.g. both are low-dimensional with the number of variables equal to the number of responses (usually 3). The Hawkyard method is very fast, but has a key problem, see below.

The *Transformed Least Slope Squared* (TLSS) method was developed by Scott Burns, see **References**. This is my name for it, not his. What I call TLLS is actually is a combination of Burns' LHTSS and LLSS methods; the one that `invert()` chooses depends on `type(x)`, see below. Both of these are high-dimensional, with the number of variables equal to #(responses) + #(wavelengths).

The first argument to `invert()` is the responder `x`, and the second is the matrix `response` of responses (e.g. XYZs or RGBs).

The goal is to return a "good" spectrum for each response so that:

$$\texttt{product(invert(x,response), x)} \cong \texttt{response}$$

The error is returned as column `estim.precis`, see below.

First consider the case where `x` has type `type='responsivity.material'`. The goal is to compute a reflectance spectra. All the methods will fail if the response is on the object-color boundary (an *optimal color*) or outside the boundary. They *may* also fail if the response is inside the object-color

solid (the *Rösch Farbkörper*) and very close to the boundary.
The centroid method solves a non-linear system that contains a *Langevin-function-based* squashing function, see *Davis* for details. When successful it always returns a feasible spectrum with small `estim.precis`.
The Hawkyard method is linear and very fast, but in raw form it may return a non-feasible reflectance spectrum. In this case `invert()` simply clamps to the interval [0,1] and so `estim.precis` can be large.
The TLSS method solves a non-linear system that contains the squashing function $(\tanh(z) + 1)/2$, see *Burns* for details. When successful it always returns a feasible spectrum with small `estim.precis`.

Now consider the case where `x` has `type='responsivity.light'`. The goal is to compute the spectrum of a light source. All the methods will fail if chromaticity of the response is on the boundary of the inverted-U (assuming `x` models the human eye) or outside the boundary. They *may* also fail if the response is inside the inverted-U and very close to the boundary.
The centroid method works on a relatively small range of chromaticities; it will fail if the response is too far from the response to Illuminant E. See *Davis* for the details. When successful it always returns an everywhere positive spectrum with small `estim.precis`. This method has the desirable property that if the response is multiplied by a positive number, the computed spectrum is multiplied by that same number.
The Hawkyard method does not work in this case.
The TLSS method solves a non-linear system that contains the squashing function $\exp(z)$, see *Burns* for the details. When successful it always returns an everywhere positive spectrum with small `estim.precis`. This method succeeds on a larger set of chromaticities than the centroid method. It also has the desirable scaling multiplication property mentioned above.

The centroid and Hawkyard methods have an equalization option, which is controlled by the argument `alpha` and is enabled by default, see below. When enabled, if the response comes from a constant spectrum (a perfectly neutral gray material, or a multiple of Illuminant E), then the computed spectrum is that same constant spectrum (up to numerical precision). I call this the *neutral-exact property*. Equalization is a complicated mechanism, for details see *Davis*. For the TLSS method, the neutral-exact property is intrinsic, and `alpha` is ignored.
NOTE: If the responder has only one output channel (e.g. a monochrome camera) and equalization is enabled, then *all* responses are inverted to a constant spectrum. This may or may not be desirable.

Usage

```
## S3 method for class 'colorSpec'
invert( x, response, method='centroid', alpha=1 )
```

Arguments

`x`	a **colorSpec** object with `type(x)` = `'responsivity.material'` or `'responsivity.light'` and M responsivities. The wavelengths must be regular (equidistant).
`response`	a numeric NxM matrix, or a numeric vector that can be converted to such matrix, by row. The N responses are contained in the rows. The `rownames(response)` are copied to the output `specnames`.
`method`	either `'centroid'` or `'Hawkyard'` or `'TLSS'`. `'Hawkyard'` is only valid when `type(x)` is `'responsivity.material'`. Matching is partial and case-insensitive.

`alpha` a vector of M weighting coefficients, or a single number that is replicated to length M. When `method='centroid'`, `alpha` is used for *equalizing* the responsivities, which is recommended. For `alpha` to be valid, the linear combination of the M responsitivies, with coefficients `alpha`, must be positive. To disable equalization (not recommended) and use the original responsivities, set `alpha=NULL`. Similarly, when `method='Hawkyard'`, `alpha` is used for equalizing the responsivities, which is also recommended. When `method='TLSS'`, `alpha` is ignored.

Details

For `method='centroid'` the function calls the non-linear root-finder `rootSolve::multiroot()`, which is general purpose and "full Newton".

For `method='Hawkyard'` the function solves a linear system by inverting a small matrix (#[responses] x #[responses]). The spectra are then clamped to [0,1].

For `method='TLSS'` the function solves a constrained least-squares problem using Lagrange multipliers. A critical point is found using a "full Newton" iteration. The original MATLAB code is posted at *Burns*, and was ported from MATLAB to R with only trivial changes. When computing a reflectance spectrum, the Hawkyard method is used for the initial guess, after little extra clamping. This improved guess cuts the number of iterations substantially, and the extra computation time is negligible.

Value

If `type(x)='responsivity.material'` it returns a **colorSpec** object with `type = 'material'` (`quantity = 'reflectance'`).

If `type(x)='responsivity.light'` it returns a **colorSpec** object with `type = 'light'` (`quantity='energy'` or `quantity='photons'` depending on `quantity(x)`).

In either case, the returned object has `organization = 'df.row'` and the `extradata` is a `data.frame` with these columns:

`response` the input matrix of desired responses

`estim.precis` the difference between the desired response and actual response. It is the mean of the absolute value of the differences. See `rootSolve::multiroot()`

`time.msec` the time to compute the spectrum, in msec. When `method='Hawkyard'`, all N spectra are computed at once, so all N spectra are assigned the same mean time.

`iters` the number of iterations that were required to find the relevant root. This is not present when `method='Hawkyard'`.

`clamped` a logical indicating whether the reflectance was clamped to [0,1]. This is present only when `method='Hawkyard'`.

If a response could not be estimated, the row contains `NA` in appropriate columns, and a warning is issued.

In case of global error, the function returns `NULL`.

Known Issues

If `type(x)='responsivity.light'` the centroid method may fail (not converge) if the response is too far from that of Illuminant E.

References

Davis, Glenn. A Centroid for Sections of a Cube in a Function Space, with Application to Colorimetry. https://arxiv.org/abs/1811.00990. [math.FA]. 2018.

Bianco, Simone. Reflectance spectra recovery from tristimulus values by adaptive estimation with metameric shape correction. vol. 27, no 8. *Journal of the Optical Society of America A*. pages 1868-1877. 2010 https://opg.optica.org/josaa/abstract.cfm?uri=josaa-27-8-1868.

Burns, Scott A. Generating Reflectance Curves from sRGB Triplets. http://scottburns.us/reflectance-curves-from-srgb/.

Hawkyard, C. J. Synthetic reflectance curves by additive mixing. *Journal of the Society of Dyers and Colourists*. vol. 109. no. 10. Blackwell Publishing Ltd. pp. 323-329. 1993.

Koenderink, J.J. Color for the Sciences. MIT Press. 2010.

See Also

`type()`, `quantity()`, `organization()`, `specnames()`, `product()`, `extradata()`, `rootSolve::multiroot()`, vignette **Estimating a Spectrum from its Response**

Examples

```
wave = 400:700
E.eye = product( illuminantE(1,wave), "material", xyz1931.1nm, wavelength=wave )
path = system.file( 'extdata/targets/CC_Avg30_spectrum_CGATS.txt', package='colorSpec' )
MacbethCC = readSpectra( path, wavelength=wave )
XYZ = product( MacbethCC, E.eye, wavelength=wave )
est.eq   = invert( E.eye, XYZ, method='centroid', alpha=1 )
extra    = extradata(est.eq)
range(extra$estim.precis)       # prints   0.000000e+00 3.191741e-08
```

---

lightResponsivitySpectra

compute standard light responsivity spectra

Description

Some action spectra standards are defined by simple equations; the erythemal spectrum for human sunburn is one of them.

Usage

```
erythemalSpectrum( wavelength=250:400 )
```

Arguments

`wavelength` a vector of wavelengths, in nm

Details

This erythemal spectrum is defined in 4 pieces: $\lambda \leq 298$, $298 \leq \lambda \leq 328$, $328 \leq \lambda \leq 400$, and $400 < \lambda$. The unit is nm. The spectrum is used in the definition of the international standard **UV Index**.

Value

For `erythemalSpectrum()`
A **colorSpec** object with `quantity` equal to `'energy->action'`. The responsivity is 0 for $\lambda > 400$ nm, so this putting this spectrum in the category of human vision is a bit of a stretch.

Source

https://en.wikipedia.org/wiki/Ultraviolet_index

References

McKinlay, A.F., and B.L. Diffey. A reference action spectrum for ultraviolet induced erythema in human skin. CIE Res. Note, 6(1), 17-22. (1987)

See Also

daylight, `quantity`, materialSpectra, lightSpectra

`LightSpectra` *compute standard light spectra*

Description

Two families of standard illuminants that are parameterized by temperature are the Planckian spectra (black-body spectra), and daylight spectra. For the daylight spectra, a smoothed version is available. Illuminant E, a third and trivial spectrum, is also available.

Usage

```
planckSpectra( temperature, wavelength=300:830, normalize=TRUE, c2=1.4388e-2 )

daylightSpectra( temperature, wavelength=NULL,
                    components=colorSpec::daylight1964, roundMs=FALSE )

referenceSpectraTM30( temperature, wavelength=380:780, ... )

illuminantE( energy=1, wavelength=380:780 )
```

Arguments

temperature	a vector of temperatures, in Kelvin
wavelength	a vector of wavelengths. For planckSpectra() and illuminantE() this is required. For daylightSpectra() this is optional. The default wavelength=NULL means to use the wavelengths in components, and otherwise components is resampled at the given wavelength vector.
normalize	a logical value. If TRUE the Planck spectra are normalized to have value 1 at 560nm. If FALSE then the quantity returned is radiant exitance with unit $W * m^{-2} * nm^{-1}$.
c2	the value of hc/k in Planck's law. h is the Planck constant; c is the speed of light in m/sec; and k is the Boltzmann constant. The default value of 1.4388e-2 $m * K$ was recommended by the CIE in 2005; in 1986 the CIE recommended c2=1.438e-2. If c2='calc' then c2 is calculated directly from the 3 physical constants, as recommended by CODATA 2014.
components	a **colorSpec** object with the daylight components S_0, S_1, and S_2. The default is daylight1964 and a smoothed version daylight2013 is also available.
roundMs	a logical value. The original CIE method for the daylight spectra requires rounding intermediate coefficients M1 and M2 to 3 decimal places. This rounding is necessary to reproduce the tabulated values in Table T.1 of the CIE publication in **References**.
...	other args - components and roundMs - passed to daylightSpectra
energy	a vector of energy levels

Details

For planckSpectra() the valid range of temperatures is 0 to Inf (∞) K, but with exceptions at the endpoints. For a negative temperature the spectrum is set to all NAs.
If temperature=0 and normalize=TRUE, the spectrum is set to all NAs. If temperature=0 and normalize=FALSE, the spectrum is set to all 0s.
Conversely, if temperature=Inf and normalize=FALSE, the spectrum is set to all NAs. If temperature=Inf and normalize=TRUE, the spectrum is set to the pointwise limit $(560/\lambda)^4$ (which appears blue).

For daylightSpectra() the valid range of temperatures is 4000 to 25000 K. For a temperature outside this range the spectrum is set to all NAs.

For referenceSpectraTM30() the valid range of temperatures is 0 to 25000 K. The spectrum is computed from the recipe in standard IES TM-30, see **ANSI/IES TM-30-20**. For a temperature $\leq$ 4000K, the reference is the Planck spectrum. For a temperature $\geq$ 5000K, the reference is the Daylight spectrum. And for a temperature between 4000K and 5000K, the reference is a weighted blend of the Planck and Daylight spectra at that temperature.

The equations for daylightSpectra() and planckSpectra() are complex and can be found in the **References**.

IlluminantE() is trivial - all constant energy.

Value

For planckSpectra(), daylightSpectra(), and referenceSpectraTM30() :
a **colorSpec** object with quantity equal to 'energy', and organization equal to 'matrix' or 'vector'. The specnames are 'PNNNN' or 'DNNNN' for planckSpectra() and daylightSpectra() respectively. And for referenceSpectraTM30() 'blended CCT=NNNN' is used.
The number of spectra in the object is the number of temperatures = length(temperature).

For illuminantE() :
A **colorSpec** object with quantity equal to 'energy'.
The number of spectra in the object is the number of energy levels = length(energy).

References

Günther Wyszecki and W.S. Stiles. **Color Science : Concepts and Methods, Quantitative Data and Formulae**. Second Edition. Wiley-Interscience. 1982. page 146.

CIE 15: Technical Report: Colorimetry, 3rd edition. CIE 15:2004. Table T.1, pp 30-32, and Note 5 on page 69.

Schanda, Janos. CIE Colorimetry, in *Colorimetry: Understanding the CIE System*. Wiley Interscience. 2007. p. 42.

ANSI/IES TM-30-20. **Technical Memorandum: IES Method for Evaluating Light Source Color Rendition**. https://store.ies.org/product/technical-memorandum-ies-method-for-evaluating-light-s

See Also

daylight, resample, organization, quantity, materialSpectra

---

linearize *linearize a colorSpec object - to make it ready for colorimetric calculations*

Description

linearize spectra and return modified object

Usage

```
## S3 method for class 'colorSpec'
linearize( x )
```

Arguments

x	a **colorSpec** object

Details

If the quantity(x) is not 'absorbance' then x is returned unchanged.

If the quantity(x) is 'absorbance' then absorbance is converted to transmittance using

$$transmittance = 10^{-absorbance}$$

Surprisingly, there does not seem to be a similar logarithmic version of reflectance. Plots with log(responsivity) is somewhat common, but does not seem to have a separate name. I have not seen log(radiometric power).

Value

linearize returns a **colorSpec** object with linear quantities.

See Also

quantity

---

lms1971 *Cone Fundamentals - 2-degree (1971)*

Description

lms1971.5nm	the Vos & Walraven (1971) 2° cone fundamentals from 380 to 780 nm, at 5nm intervals

Format

A **colorSpec** object organized as a matrix with 3 columns:

long	the long wavelength responsivity
medium	the medium wavelength responsivity
short	the short wavelength responsivity

Source

http://www.cvrl.org/database/text/cones/vw.htm

References

Vos, J. J. & Walraven, P. L. On the derivation of the foveal receptor primaries. **Vision Research**. 11 (1971) pp. 799-818.

See Also

lms2000

Examples

```
summary(lms1971.5nm)
white.point = product( D65.1nm, lms1971.5nm, wave='auto' )
```

`lms2000` *Cone Fundamentals - 2-degree (2000)*

Description

`lms2000.1nm` the Stockman and Sharpe (2000) 2° cone fundamentals from 390 to 830 nm, at 1nm intervals

Format

A **colorSpec** object organized as a matrix with 3 columns:

`long`	the long wavelength responsivity
`medium`	the medium wavelength responsivity
`short`	the short wavelength responsivity

Source

http://www.cvrl.org/cones.htm

References

Stockman, A., Sharpe, L. T., & Fach, C. C. (1999). The spectral sensitivity of the human short-wavelength cones. **Vision Research**. 39, 2901-2927.

Stockman, A., & Sharpe, L. T. (2000). Spectral sensitivities of the middle- and long-wavelength sensitive cones derived from measurements in observers of known genotype. **Vision Research**. 40, 1711-1737.

See Also

lms1971

Examples

```
summary(lms2000.1nm)
white.point = product( D65.1nm, lms2000.1nm, wave='auto' )
```

logging *Logging in colorSpec package*

Description

Logging is done using the **logger** package. Logging output goes to `stderr()`, just like the message stream; but see `sink()` (and the pitfalls of using it).

Logging Options

`colorSpec.stoponerror` If the this option is `TRUE` (the default), a log event with level `ERROR` stops execution; otherwise, execution keeps going. For interactive use, `TRUE` is probably better. For long batch jobs, `FALSE` might be appropriate, since then a single error may not force a complete repeat.
A `FATAL` event always stops execution.

For examples on changing this option, see `cs.options`.

References

Wikipedia. **Log4j**. https://en.wikipedia.org/wiki/Log4j

See Also

`options`, `cs.options`, `sink`, `stderr`

Examples

```
options( colorSpec.stoponerror=TRUE )

# or equivalently
cs.options( stop=TRUE )
```

luminsivity *Luminous Efficiency Functions (photopic and scotopic)*

Description

`luminsivity.1nm` Four luminous efficiency functions, from 360 to 830 nm, at 1nm step

Format

A **colorSpec** object, with quantity `'energy->neural'`, and with 4 spectra:

`photopic1924` The luminous efficiency function adopted by the CIE in 1924, and defining the *standard photopic observer*. It is only to be used when light levels are high enough that the sensitivity of the eye is mediated by cones, and not rods. It is the same as the *y-bar* function in `xyz1931.1nm`. It is used to define the *candela* in the *International System* (SI) and is the only one of these functions to appear in the SI. It was downloaded from http://www.cvrl.org/database/data/lum/vl1924e_1.csv where it is defined from 360 to 830 nm.

`scotopic1951` The luminous efficiency function adopted by the CIE in 1951, and defining the *standard scotopic observer*. It is only to be used when light levels are low enough to exclude the activation of cones. It has no effective role in colorimetry. It was downloaded from http://www.cvrl.org/database/data/lum/scvle_1.csv where it is defined from 380 to 780 nm. It has been padded with 0s to 360 to 830 nm.

`photopic1978` The luminous efficiency function for photopic vision, with adjustments in the blue region by Judd (1951) and Vos (1978). It was published by the CIE in 1988. It was downloaded from http://www.cvrl.org/database/data/lum/vme_1.csv where it is defined from 380 to 780 nm. It has been padded with 0s to 360 to 830 nm.

`photopic2008` The CIE (2008) *physiologically-relevant* luminous efficiency function for photopic vision, by Stockman, Jagle, Pirzer, & Sharpe. It was downloaded from http://www.cvrl.org/database/data/lum/linCIE2008v2e_1.csv where it is defined from 390 to 830 nm. It has been padded with 0s to 360 to 830 nm.

Note

Luminsivity is a self-coined *portmanteau word*: `luminsivity` = `luminous` * `responsivity`. The word is unrelated to *emissivity*. The term *luminous responsivity* is not common, but appears on page 15 of *Grum*. The term *luminous efficiency function* is standard, but too long. The term *luminosity function* is common, but *luminosity* is ambiguous and also appears in astronomy and scattering theory.
The object `luminsivity.1nm` is used by the function `photometric()`.

Source

Colour & Vision Research Laboratory. Institute of Opthalmology. University College London. UK. http://www.cvrl.org/

References

Grum, Franc and Richard J. Becherer. **Radiometry**. Optical Radiation Measurements, Volume 1. Academic Press. 1979.

Stockman, A., Jagle, H., Pirzer, M., & Sharpe, L. T. (2008). The dependence of luminous efficiency on chromatic adaptation. Journal of Vision, 8, 16:1, 1-26.

See Also

`xyz1931.1nm`, `photometric`

Examples

```
summary(luminsivity.1nm)
product( D65.1nm, luminsivity.1nm, wave='auto' )
```

`materialSpectra` *compute standard material spectra*

Description

Compute neutral gray material constant reflectance/transmittance, and rectangular spectra. Also compute absorbance of the human lens, as a function of age.

Usage

```
neutralMaterial( gray=1, wavelength=380:780 )
rectangularMaterial( lambda, alpha=1, wavelength=380:780 )

lensAbsorbance( age=32, wavelength=400:700 )
```

Arguments

`gray`	a numeric N-vector of gray levels, in the interval [0,1]. `gray=1` represents the *Perfect Reflecting Diffuser*.
`lambda`	a numeric Nx2 matrix with wavelength pairs in the rows, or a vector that can be converted to such a matrix, by row. The two wavelengths are the two *transition wavelengths* of the returned spectrum, see **Details**.
`alpha`	a numeric N-vector of *chromatic amplitudes* in the interval [-1,1]. N must be equal to `nrow(lambda)`. `alpha` can also be a single number, which is then replicated to length `nrow(lambda)`. The *chromatic amplitude* is defined by *Logvinenko* and controls the size of both transitions, see **Details**.
`age`	a numeric N-vector of ages in years; all ages must be ≥ 20.
`wavelength`	a vector of wavelengths for the returned object

Details

A *rectangular spectrum*, or *rectangular metamer*, is easiest to define when $\alpha = 1$ and $\lambda_1 < \lambda_2$. In this case it is a band-pass filter with transmittance=1 for $\lambda \in [\lambda_1, \lambda_2]$ and transmittance=0 otherwise. To create a long-pass filter, just set λ_2 to `Inf`, or any large wavelength outside the spectrum range; and similarly for a short-pass filter.
When $0 < \alpha < 1$ the spectrum is a weighted mixture of this band-pass filter with a perfect neutral gray filter with transmittance=0.5 at all λ, using α and $1 - \alpha$ as the two weights. The minimum transmittance is $(1 - \alpha)/2$ and the maximum is $(1 + \alpha)/2$, and their difference, the *chromatic amplitude*, is α. It is still a band-pass filter.
If $\alpha = 0$ the spectrum is a perfect neutral with transmittance=0.5.
To "flip" the spectrum to its complement (change band-pass to band-stop, etc.), change α to a negative number, or swap λ_1 and λ_2. If $\lambda_1 == \lambda_2$ then the spectrum is undefined and a warning is issued (unless $\alpha = 0$).

Value

`neutralMaterial()` returns a **colorSpec** object with `quantity` equal to `'reflectance'`. The reflectance of each spectrum is constant and taken from `gray`. There are N spectra in the object - one for each gray level.

`rectangularMaterial()` returns a **colorSpec** object with `quantity` equal to `'transmitance'`. The transmitance of each spectrum is a step function with 0, 1 or 2 transitions (jumps) defined by the corresponding row in `lambda`. If `rownames(lambda)` is not `NULL`, they are copied to `specnames` of the output. Otherwise the `specnames` are computed from the shape of the spectrum using these acronyms: LP (long-pass), SP (short-pass), BP (band-pass), BS (band-stop), and N (neutral, in case `alpha==0`).

`lensAbsorbance()` returns a **colorSpec** object with `quantity` equal to `'absorbance'`. The absorbance model for the human lens is taken from *Pokorny*. There are N spectra in the object - one for each age (N=`length(age)`).

Logvinenko

It is clear that there are 3 degrees-of-freedom in the spectra returned by `rectangularMaterial()`. *Logvinenko* shows that these spectra in fact form a 3D ball, which he calls the *rectangle color atlas*. He also shows that if a material responder satisfies the 2-transition condition, then these spectra uniquely generate *all* colors in the corresponding object color solid. For more on this, see the vignette **Estimating a Spectrum from its Response**.

Ostwald

Every spectrum returned by `rectangularMaterial()` is an Ostwald ideal spectrum. In Ostwald's terminology, the *color content* = *chromatic amplitude* = α. And the *black content* = *white content* = $(1 - \alpha)/2$. Note that the sum of these 3 contents is 1. However, Ostwald allows *black content* and *white content* to be unequal, as long as the sum of the 3 contents is 1, and all are non-negative. Thus there is one extra degree-of-freedom for Ostwald's ideal spectra, for a total of 4 degrees-of-freedom. If an additional argument (or arguments) were added to `rectangularMaterial()`, then it could return all Ostwald ideal spectra.

References

Foss, Carl E. and Dorothy Nickerson and Walter C. Granville. Analysis of the Ostwald Color System. *J. Opt. Soc. Am.*. vol. 34. no. 7. pp. 361-381. July, 1944.

Logvinenko, A. D. An object-color space. **Journal of Vision**. 9(11):5, 1-23, (2009). `https://jov.arvojournals.org/article.aspx?articleid=2203976`. doi:10.1167/9.11.5.

Pokorny, Joel, Vivianne C. Smith, and Margaret Lutze. Aging of the Human Lens. **Applied Optics**. Vol. 26, No. 8. 15 April 1987. Table I. Page 1439.

See Also

lightSpectra, `quantity()`, `specnames()`, `computeADL()`, vignette **Estimating a Spectrum from its Response**

Examples

```
#   make a perfect reflecting diffuser (PRD)
prd = neutralMaterial( 1 )

#   make a perfect transmitting filter (PTF)
ptf = prd
quantity(ptf) = 'transmittance'

#   make a band-stop filter (for interval [500,550])
#   with 1% transmittance in the band, and 99% outside the band
bs = rectangularMaterial( c(500,550), -0.98, 400:700 )
bs = rectangularMaterial( c(550,500),  0.98, 400:700 )  # equivalent to previous line

#   compare transmittance at 3 ages: 20, 32, and 80 years
plot( linearize(lensAbsorbance( c(20,32,80) )), col='black', lty=1:3 )
```

---

mean *calculate mean of multiple spectra*

Description

compute mean of all spectra in a **colorSpec** object

Usage

```
## S3 method for class 'colorSpec'
mean( x, ... )
```

Arguments

x	a colorSpec object
...	further arguments ignored

Details

This function might be useful when capturing many spectra on a spectrometer and averaging them to reduce noise.

Value

a **colorSpec** object with single spectrum = average of all spectra in **colorSpec**.

metadata *metadata of a colorSpec object*

Description

Retrieve or set the metadata of a **colorSpec** object.

Usage

```
## S3 method for class 'colorSpec'
metadata(x, ...)

## S3 replacement method for class 'colorSpec'
metadata(x, add=FALSE ) <- value
```

Arguments

x	a **colorSpec** R object
...	optional names of metadata to return
value	a named list. If add is FALSE, value replaces any existing metadata. If add is TRUE, value is appended to the existing list of metadata. If a name already exists, its value is updated using modifyList(). Unnamed items in value are ignored.
add	if add=FALSE, any existing metadata is discarded. If add=TRUE then existing metadata is preserved, using modifyList().

Details

The metadata list is stored as attr(x,'metadata'). After construction this list is empty.

Value

metadata(x) with no additional arguments returns the complete named list of metadata. If arguments are present, then only those metadata items are returned.

Note

Do not confuse extradata and metadata.
metadata is unstructured data that is attached to the entire **colorSpec** object. extradata is structured data, with a row of data for each spectrum in the object.

See Also

extradata, modifyList

Examples

```
## Not run:
# get list of *all* metadata
metadata(x)

# get just the file 'path'
metadata( x, 'path' )

# set the 'date'
metadata( x ) = list( date="2016-04-01" )

## End(Not run)
```

`multiply` *multiply a colorSpec object by scalar, vector, or matrix*

Description

multiply spectra by coefficients and return modified object

Usage

```
## S3 method for class 'colorSpec'
multiply( x, s )

## S3 method for class 'colorSpec'
normalize( x, norm='L1' )
```

Arguments

`x`	a **colorSpec** object with M spectra
`s`	a scalar, an M-vector, or an MxP matrix. In the case of a matrix, assigning `colnames(s)` is recommended; see **Details**.
`norm`	one of `'L1'`, `'L2'`, or `'Linf'`, specifying one of the standard vector norms $L^1, L^2, or L^{inf}$. `norm` can also be a numeric wavelength (e.g. 560 nm), and then the spectrum is scaled to have value 1 at this wavelength. Of course, this is not a true vector norm.

Details

For `multiply()`:
If `s` is an MxP matrix, say **S**, and one thinks of the spectra as organized in an NxM matrix **X**, then the new spectra are defined by the matrix **XS**, which is NxP. If the P column names of `s` are set, then they are copied to the `specnames` of the output. Otherwise, default spectrum names are assigned as in `colorSpec()`, with a warning.
If `s` is an M-vector, then **S**=`diag(s)` is computed and used in the previous sentence. This has the

effect of multiplying spectrum i by s[i].
If s is a scalar then every spectrum is multiplied by s.
The multiplication may produce negative entries, but no check is made for this.
WARNING: An M-vector and an Mx1 matrix may yield quite different results.

For `normalize()`:
`normalize()` calls `multiply()` with s = an M-vector. If the norm of a spectrum is 0, then it is left unchanged.

Value

`multiply` returns a **colorSpec** object with the matrix of spectra of x multiplied by s.

`normalize` returns a **colorSpec** object with each spectrum of x scaled to have given norm equal to 1.

In both functions, the `quantity` and `wavelength` are preserved.

Note

If x is organized as a matrix, and s is a scalar, the one can use the simpler and equivalent `s*x`.

See Also

`product()`, `quantity()`, `wavelength()`, `specnames()`, `colorSpec()`

---

officialXYZ *Query the Official XYZ values for Standard Illuminants*

Description

In careful calcuations with standard illuminants, it is often helpful to have the 'official' values of XYZ, i.e. with the right number of decimal places.

Usage

```
officialXYZ( name )
```

Arguments

name	a subvector of `c('A','B','C','D50','D50.ICC','D55','D65','D75','E','F2','F7','F11')`, which are the names of some standard illuminants

Details

All XYZ values are taken from the ASTM publication in **References**, except `B` which is taken from *Wyszecki & Stiles* and `D50.ICC` which is taken from ICC publications. The latter is different than that of ASTM.

Value

An Mx3 matrix where M is the length of `name`. Each row filled with the official XYZ, but if the illuminant name is not recognized the row is all `NA`s. The matrix `rownames` are set to `name`, and `colnames` to `c('X','Y','Z')`.

WARNING

This function is deprecated. New software should use `spacesRGB::standardXYZ()` instead.

Note

The input names are case-sensitive. The output XYZ is normalized so that `Y=1`.

References

ASTM E 308 - 01. Standard Practice for Computing the Colors of Objects by Using the CIE System. (2001).

Günther Wyszecki and W. S. Stiles. Color Science: Concepts and Methods, Quantitative Data and Formulae, Second Edition. John Wiley & Sons, 1982. Table I(3.3.8) p. 769.

See Also

`ABC`, `D50`, `D65`, `Fluorescents`, `illuminantE`

Examples

```
officialXYZ( c('A','D50','D50.ICC','D65') )
#                 X Y         Z
# A       1.0985000 1 0.3558500
# D50     0.9642200 1 0.8252100
# D50.ICC 0.9642029 1 0.8249054
# D65     0.9504700 1 1.0888300
```

`organization` *organization of a colorSpec object*

Description

Retrieve or set the organization of a **colorSpec** object.

Usage

```
## S3 method for class 'colorSpec'
organization(x)

## S3 replacement method for class 'colorSpec'
organization(x) <- value
```

Arguments

x	a **colorSpec** R object
value	a valid organization: 'vector', 'matrix', 'df.col', or 'df.row'.

Details

If organization(x) is "vector", then x is a vector representing a single spectrum. Compare this with stats::ts().

If organization(x) is "matrix", then x is a matrix and the spectra are stored in the columns.

If organization(x) is "df.col", then x is a data.frame with M+1 columns, where M is the number of spectra. The wavelengths are stored in column 1, and the spectra in columns 2:(M+1). This organization is good for printing to the console, and writing to files.

If the organization of x is "df.row", then x is a data.frame with N rows, where N is the number of spectra. The spectra are stored in the last column, which is a matrix with the name "spectra". The other columns preceding spectra (if present) contain extra data associated with the spectra; see extradata.

Value

organization(x) returns a valid organization: 'vector', 'matrix', 'df.col', or 'df.row'.

Note

In organization(x) <- value
if x has more than 1 spectrum, then value equal to 'vector' is invalid and ignored.
If organization(x) is equal to 'df.row' and also has extradata, then changing the organization silently discards the extradata.

See Also

colorSpec; extradata

Examples

```
organization(Hoya)              # returns 'df.row'
organization(Hoya) = 'matrix'   # extradata in Hoya is silently discarded
```

---

photometric *convert illuminant spectra to photometric units*

Description

Convert radiometric units of power or energy to photometric units, using 4 standard photometric weighting curves. Actinometric units (number of photons) are converted to radiometric units (energy of photons) on-the-fly.

Usage

```
## S3 method for class 'colorSpec'
photometric( x, photopic=683, scotopic=1700, multiplier=1 )
```

Arguments

x	a **colorSpec** object with type equal to 'light', and with M spectra
photopic	the conversion factor for photopic vision, in lumen/watt. The CIE standard is 683, and another common value is 683.002.
scotopic	the conversion factor for scotopic vision, in lumen/watt. The CIE standard is 1700, and another common value is 1700.06.
multiplier	an conversion factor intended for conversion of units, and applied to both photopic and scotopic vision. For example if the input unit of x is $watt * sr^{-1}$, and the desired output unit is $candlepower$, then set multiplier=1/0.981.

Details

The function computes the product of x with luminsivity.1nm. This product is an Mx4 matrix, where M is the number of spectra in x. There are 3 columns for photopic vision, and 1 column for scotopic vision. These columns are multiplied by the appropriate conversion factors and the resulting Mx4 matrix is returned.

The 5 power-based input quantities and corresponding photometric outputs are:

radiant power $[watt]$	—>	**luminous flux** $[lumen]$
irradiance $[watt * m^{-2}]$	—>	**illuminance** $[lumen * m^{-2} = lux]$
radiant exitance $[watt * m^{-2}]$	—>	**luminous exitance** $[lumen * m^{-2} = lux]$
radiant intensity $[watt * sr^{-1}]$	—>	**luminous intensity** $[lumen * sr^{-1} = candela]$
radiance $[watt * sr^{-}1 * m^{-2}]$	—>	**luminance** $[candela * m^{-2} = nit]$

The 2 *common* energy-based input quantities and corresponding photometric outputs are:

radiant energy $[joule]$	—>	**luminous energy** $[talbot = lumen - second]$
radiant exposure $[joule * m^{-2}]$	—>	**luminous exposure** $[talbot * m^{-2} = lux - second]$

and there are 3 more obtained by integrating over time. For example "time-integrated radiance" —> "time integrated luminance". But I have not been able to find names for these 3. The *talbot* is the unofficial name for a lumen-second.

Value

photometric() returns an Mx4 matrix, where M is the number of spectra in x. The rownames are specnames(x), and the colnames are specnames(luminsivity.1nm).
In case of ERROR it returns NULL.

Note

To get the right output quantity and units, the user must know the input quantity and units. If the units are different than those in the above list, then set `multiplier` appropriately.
It is up to the user to determine whether *photopic* or *scotopic* vision (or neither) is appropriate. The intermediate *mesopic* vision is currently a subject of research by the CIE, and might be added to this function in the future.

References

Poynton, Charles. **Digital Video and HD - Algorithms and Interfaces**. Morgan Kaufmann. Second Edition. 2012. Appendix B, pp. 573-580.

See Also

`quantity`, `type`, `luminsivity.1nm`, `radiometric`

Examples

```
photometric( solar.irradiance )  # unit is watt*m^{-2}

#             photopic1924 scotopic1951 photopic1978 photopic2008  # units are lux
# AirMass.0       133100.41     313883.2    133843.65     140740.3
# GlobalTilt      109494.88     250051.5    110030.31     115650.0
# AirMass.1.5      97142.25     215837.1     97571.57     102513.7
```

---

plot *plot spectra*

Description

plot the spectra in a **colorSpec** object as lines or points

Usage

```
## S3 method for class 'colorSpec'
plot( x, color=NULL, subset=NULL, main=TRUE, legend=TRUE, CCT=FALSE, add=FALSE, ... )
```

Arguments

`x`	a colorSpec object
`color`	If `color=NULL` then colors are computed from the spectra themselves. If `type(x)` is `'material'` the color is computed using illuminant `D65.1nm` and responder `BT.709.RGB` with no further normalization. Otherwise the spectrum color is faked by changing its `quantity` to `'energy'` and taking the `product` with `BT.709.RGB`. The resulting RGBs are normalized to have a maximum of 1. This RGB normalization is done *before* processing the `subset` argument. If `color='auto'` then a suitable set of colors is generated using `colorRamp()`. Otherwise `color` is passed on to `lines.default()` as the `col` argument, e.g. `col='black'`.

subset	specifies a subset of x to plot; see subset() for acceptable arguments.
main	If main=TRUE then a main title is generated from the file 'path' in the metadata list, or from deparse(substitute(x)). If main=FALSE then no main title is displayed. And if main is a string then that string is used as the main title.
legend	If legend=TRUE then a pretty legend using specnames(x) is placed in the 'topright' corner of the plot. If legend is a string it is interpreted as naming a corner of the plot and passed as such to the function legend. If legend=FALSE then no legend is drawn.
CCT	If CCT=TRUE and the type of x is 'light' then the CCT of each spectrum is added to the legend; see computeCCT(). The package **spacesXYZ** must be installed.
add	If add=TRUE then the lines are added to an existing plot, and these arguments are ignored: main, ylab, xlim, ylim, and log; see **Details**.
...	other graphical parameters, see **Details**

Details

Commonly used graphical parameters are:

type passed to lines.default(), with default type='l'. Other valid values are 'p' (points), 'b', 'c', 'o', 'h', 'S', 's', and 'n', see plot() for their meanings.
An additional type='step' is available. This option draws each spectrum as a *step function*, similar to 'S' and 's', except that the jumps are *between* the wavelengths (with appropriate extensions at min and max wavelengths). The function segments() is used for the drawing. For type='step', lwd and lty should be vectors of length 1 or 2. If the length of lwd is 1, then horizontal segments are draw with that width, but vertical segments are not drawn. If the length of lwd is 2, then vertical segments are draw with width lwd[2]. If the length of lty is 2, then the styles are applied to the horizontal and vertical segments in that order. If the length of lty is 1, then that style is applied to both horizontal and vertical segments. For examples of this plotting option, see the vignette **Convexity and Transitions**.

lwd, lty passed to lines.default(), except when type='step' when they are passed to segments(). In the former case these can be vectors, and components are passed sequentially to each spectrum, similar to matplot(). In the latter case, see the description in type. The default value for both is 1.

pch passed to lines.default(), but it only has meaning when type='p', 'b', or 'o'. This can be a vector, and components are passed sequentially to each spectrum.

ylab If ylab is a string then it is passed on to plot.default(), otherwise suitable default string is generated.

xlim, ylim If xlim and ylim are 2-vectors, they are passed to plot.default. If one of the components is NA then a suitable default is supplied.

log passed on to plot.default(). Care must be taken for y because many spectra are 0 at some wavelengths, and even negative. Use ylim in such cases.

Value

TRUE or FALSE

See Also

computeCCT(), subset(), lines(), segments(), plot(), matplot(), colorRamp()

Examples

```
plot( 100 * BT.709.RGB )
plot( xyz1931.1nm, add=TRUE, lty=2, legend=FALSE )
```

---

plotOptimals *Plot Optimal Colors*

Description

Consider a **colorSpec** object x with type equal to 'responsivity.material' and 3 responsivity spectra. The function plotOptimals3D() makes a plot of the *object-color solid* for x. This solid is a *zonohedron* in 3D. The 3D drawing package **rgl** is required.
Consider a **colorSpec** object x with type equal to 'responsivity.material' and 2 responsivity spectra. The function plotOptimals2D() makes a plot of the *object-color solid* for x. This solid is a *zonogon* in 2D. The 3D drawing package **rgl** is *not* required.
The set of all possible material reflectance functions (or transmittance functions) is convex, closed, and bounded (in any reasonable function space), and this implies that the set of all possible output responses from x is also convex, closed, and bounded. The latter set is called the *object-color solid*, or *Rösch Farbkörper*, for x. A color on the boundary of the *object-color solid* is called an *optimal color*.

These functions are essentially wrappers around zonohedra::plot.zonogon() and zonohedra::plot.zonohedron().

Usage

```
## S3 method for class 'colorSpec'
plotOptimals3D( x, ... )

## S3 method for class 'colorSpec'
plotOptimals2D( x, ... )
```

Arguments

x	a **colorSpec** object with type equal to 'responsivity.material' and 2 or 3 spectra, as appropriate.
...	more arguments passed to zonohedra::plot.zonogon() or zonohedra::plot.zonohedron() as appropriate. For plotOptimals3D(), examples are type and both. For plotOptimals2D(), examples are orientation and elabels.

Value

The functions return TRUE or FALSE.

Note

If all responsivity functions of x are non-negative, the *object-color solid* of x is inside the box. If the responsivity functions of x have negative lobes, the *object-color solid* of x extends outside the box. Indeed, the box may actually be *inside* the optimals.

References

Centore, Paul. *A Zonohedral Approach to Optimal Colours*. **Color Research & Application**. Vol. 38. No. 2. pp. 110-119. April 2013.

Logvinenko, A. D. An object-color space. Journal of Vision. 9(11):5, 1-23, (2009). https://jov.arvojournals.org/article.aspx?articleid=2203976. doi:10.1167/9.11.5.

West, G. and M. H. Brill. Conditions under which Schrödinger object colors are optimal. **Journal of the Optical Society of America**. 73. pp. 1223-1225. 1983.

See Also

type(), probeOptimalColors(), sectionOptimalColors(), zonohedra::plot.zonogon(), zonohedra::plot.zonohed
vignette **Plotting Chromaticity Loci of Optimal and Schrodinger Colors**

Examples

```
human = product( D50.5nm, 'slot', xyz1931.5nm, wave=seq(400,770,by=5) )
plotOptimals3D( human )

plotOptimals2D( subset(human,2:3) )     # y and z only

scanner = product( D50.5nm, 'slot', BT.709.RGB, wave=seq(400,770,by=5) )
plotOptimals3D( scanner )
```

---

print *Convert colorSpec object to readable text*

Description

display a **colorSpec** object as readable text. Output goes to stdout().

Usage

```
## S3 method for class 'colorSpec'
print( x, ...)

## S3 method for class 'colorSpec'
summary( object, long=TRUE, ... )
```

Arguments

x	a colorSpec object
object	a colorSpec object
long	logical indicating whether to print metadata, calibrate, sequence, and emulate data
...	further arguments ignored

Details

If long=FALSE, summary() prints a summary of the wavelength vector, and names of all spectra. For each spectrum it prints the range of values, LambdaMax, and extradata if any. If long=TRUE it also prints data listed above (if any).
The function print() simply calls summary() with long=FALSE.

Value

Both functions return (invisibly) the character vector that was just printed to stdout().

See Also

extradata, print, summary, stdout

Examples

```
print( xyz1931.1nm )

xyz1931.1nm     # same thing, just calls print()
```

---

probeOptimalColors *compute optimal colors by ray tracing*

Description

Consider a **colorSpec** object x with type equal to 'responsivity.material'. The set of all possible material reflectance functions (or transmittance functions) is convex, closed, and bounded (in any reasonable function space), and this implies that the set of all possible output responses from x is also convex, closed, and bounded. The latter set is called the *object-color solid* or *Rösch Farbkörper* for x. A color on the boundary of the *object-color solid* is called an *optimal color*. The special points **W** (the response to the perfect reflecting diffuser) and **0** are on the boundary of this set. The interior of the line segment of neutrals joining **0** to **W** is in the interior of the *object-color solid*. It is natural to parameterize this segment from 0 to 1 (from **0** to **W**).

A ray r that is based at a point on the interior of the neutral line segment must intersect the boundary of the *object-color solid* in a unique optimal color. The purpose of the function probeOptimalColors() is to compute that intersection point.

Currently the function only works if the number of spectra in x is 3 (e.g. RGB or XYZ).

Before **colorSpec** v 0.8-1 this function used a 2D root-finding method that could only find optimal colors whose spectra contain 0, 1, or 2 transitions. But starting with v0.8-1, we have switched to zonohedral representation of the object-color solid, which makes it possible to discover more than 2 transitions. The inspiration for this change is the article by *Centore*. To *inspect* these computed spectra, the argument `spectral` must be set to `TRUE`. The function is essentially a wrapper around `zonohedra::raytrace()`.

Usage

```
## S3 method for class 'colorSpec'
probeOptimalColors( x, gray, direction, aux=FALSE, spectral=FALSE )
```

Arguments

`x`	a **colorSpec** object with `type` equal to `'responsivity.material'` and 3 spectra
`gray`	vector of numbers in the open interval (0,1) that define neutral grays on the line segment from black to white; this neutral gray point is the basepoint of a probe ray
`direction`	a numeric Nx3 matrix with directions of the probe rays in the rows, or a numeric vector that can be converted to such a matrix, by row.
`aux`	a logical that specifies whether to return extra performance and diagnostic data; see **Details**
`spectral`	if `TRUE`, the function returns a **colorSpec** object with the optimal spectra, see **Value**.

Details

Each gray level and each direction defines a ray. So the total number of rays traced is `length(gray)` * `nrow(direction)`. The 3 responsivities are regarded not as continuous functions, but as step functions. This implies that the color solid is a zonohedron. In the preprocessing phase the zonohedral representation is calculated. The faces of the zonohedron are either parallelograms, or *compound faces* that can be partitioned into parallelograms. The centers of all these parallelograms are computed, along with their normals and plane constants.
This representation of the color solid is very strict regarding the 2-transition assumption. During use, one can count on there being some spectra with more than two transitions. Forcing the best 2-transition spectrum is a possible topic for the future.

Value

If argument `spectral=FALSE`, `probeOptimalColors()` returns a `data.frame` with a row for each traced ray. There are `length(gray)` * `nrow(direction)` rays. The columns in the output are:

`gray`	the graylevel defining the $basepoint$ of the ray. $basepoint = gray * W$
`direction`	the $direction$ of the ray
`s`	computed scalar so that $basepoint + s * direction$ is optimal

`optimal`	the optimal color on the boundary; $optimal = basepoint + s * direction$

If `aux` is `TRUE`, these extra columns related to performance and diagnostics are added:

`timetrace`	time to trace the ray, in seconds
`facetgens`	# of generators of the zonogon facet that the ray intersects. Typically it is 2, which means the facet is a parallelogram.

If `spectral` is `TRUE`, these extra columns are added:

`distance`	the signed distance from `optimal` to the boundary; it should be very small.
`transitions`	the number of transitions in the optimal spectrum; it is a non-negative even integer.

If `spectral` is `TRUE`, `probeOptimalColors()` returns a **colorSpec** object with quantity `'reflectance'`. This object contains the optimal spectra, and can be used to inspect the spectra with more than 2 transitions, which will happen. The above-mentioned `data.frame` can then be obtained by applying `extradata()` to the returned object.

If an individual ray could not be traced (which should be rare), the row contains `NA` in appropriate columns.
In case of global error, the function returns `NULL`.

WARNING

The preprocessing calculation of the zonohedron dominates the total time. And this time goes up rapidly with the number of wavelengths. We recommend using a wavelength step of 5nm, as in the **Examples**. For best results, batch a lot of rays into a single function call and then process the output.

References

Centore, Paul. *A zonohedral approach to optimal colours*. **Color Research & Application**. Vol. 38. No. 2. pp. 110-119. April 2013.

Logvinenko, A. D. An object-color space. **Journal of Vision**. 9(11):5, 1-23, (2009). `https://jov.arvojournals.org/article.aspx?articleid=2203976`. doi:10.1167/9.11.5.

Schrödinger, E. (1920). Theorie der Pigmente von grösster Leuchtkraft. **Annalen der Physik**. 62, 603-622.

West, G. and M. H. Brill. Conditions under which Schrödinger object colors are optimal. **Journal of the Optical Society of America**. 73. pp. 1223-1225. 1983.

See Also

`type()`, vignette **Plotting Chromaticity Loci of Optimal Colors**, `scanner.ACES`, `extradata()`, `zonohedra::raytrace()`

Examples

```
wave    = seq(400,700,by=5)
D50.eye = product( D50.5nm, 'material', xyz1931.1nm, wavelength=wave )
probeOptimalColors( D50.eye, c(0.2,0.5,0.9), c(1,2,1, -1,-2,-1) )

##    gray direction.1 direction.2 direction.3         s  optimal.1  optimal.2
##  1  0.2           1           2           1 32.306207  52.533143  85.612065
##  2  0.2          -1          -2          -1  8.608798  11.618138   3.782055
##  3  0.5           1           2           1 20.993144  71.560483  94.485416
##  4  0.5          -1          -2          -1 20.993144  29.574196  10.512842
##  5  0.9           1           2           1  4.333700  95.354911 103.165832
##  6  0.9          -1          -2          -1 35.621938  55.399273  23.254556

##     optimal.3 lambda.1 lambda.2    dol.delta    dol.omega   dol.lambda
##  1  49.616046 451.8013 598.9589   0.63409966   0.48287469 536.97618091
##  2   8.701041 636.3031 429.4659   0.08458527   0.99624955 674.30015903
##  3  64.267740 441.9105 615.0822   0.78101041   0.49048222 538.73234859
##  4  22.281453 615.0822 441.9105   0.21898959   0.99048222 662.20606601
##  5  82.227974 422.9191 648.7404   0.95800430   0.49825407 540.49590064
##  6  42.272337 593.2415 455.2425   0.42035428   0.97962398 650.57382749


# create a 0-1 spectrum with 2 transitions
rectspec = rectangularMaterial( lambda=c(579.8697,613.7544), alpha=1, wave=wave )

# compute the corresponding color XYZ
XYZ = product( rectspec, D50.eye )
XYZ
##                              X        Y          Z
##  BP_[579.87,613.754] 33.42026 21.96895 0.02979764

# trace a ray from middle gray through XYZ
white.XYZ   = product( neutralMaterial(1,wave=wave), D50.eye )
direction   = XYZ - white.XYZ/2

res = probeOptimalColors( D50.eye, 0.5, direction, spectral=TRUE )
res = extradata(res)
res$s
##  [1] 1.00004
## And since s=1.00004 > 1,
## XYZ is actually in the interior of the color solid, and not on the boundary.
## The boundary point res$optimal is a little-bit further along the ray,
## and the corresponding spectrum has more than 2 transitions.

res$optimal
##              X        Y          Z
##  [1,] 33.41958 21.96774 0.02808178

res$transitions
##  [1] 8
```

product *Compute the product of colorSpec objects*

Description

Take a sequence of **colorSpec** objects and compute their product. Only certain types of sequences are allowed. The return value can be a new **colorSpec** object or a matrix; see **Details**.

Usage

```
## S3 method for class 'colorSpec'
product( ... )
```

Arguments

`...` unnamed arguments are **colorSpec** objects, and possibly a single character string, see **Details**. Possible named arguments are:

`wavelength` The default `wavelength='identical'` means that all the **colorSpec** objects must have the same wavelength sequence; if they do not it is an ERROR. `wavelength` can be a new wavelength sequence, and all the objects are then `resample`d at these new wavelengths. `wavelength` can also be `'auto'` or `NULL` which means to compute a suitable wavelength sequence from those of the objects, see **Details**. It is OK to abbreviate the string `wavelength` (e.g. to `wave`); see **Examples**. It is OK for the wavelength sequence to be irregular; when the return value is a matrix the integration weights the spectrum values appropriately.

`method`**,** `span`**,** `extrapolation`**,** `clamp` passed to `resample()` with no checking or changes

`integration` only applies when the return type is matrix. The default option is `'rectangular'`, which means to weight the spectrum value equally at all wavelengths; this is the ASTM E308-01 recommendation. The other option is `'trapezoidal'`, which means to give the 2 endpoint wavelength values 1/2 the weight of the others. Trapezoidal integration is provided mostly for compatibility with other software.

Details

To explain the allowable product sequences it is helpful to introduce some simple notation for the objects:

notation	**colorSpec** `type`	**description of the object**
L	`light`	a light source
M	`material`	a material

R_L	`responsivity.light`	a light responder (aka detector)
R_M	`responsivity.material`	a material responder (e.g. a scanner)

It is also helpful to define a sequence of positive integers to be *conformable* iff it has at most one value greater than 1. For example, a sequence of all 1s is conformable. A sequence of all q's is conformable. The sequences `c(1,3)` and `c(1,1,4,1,1,4,1)` are conformable, but `c(1,1,4,1,3,4,1)` is not.

There are 6 types of sequences for which the product is defined:

1. $M_1 * M_2 * ... * M_m \mapsto M'$
The product of m materials is another material. Think of a stack of m transmitting filters effectively forming a new filter. If we think of each object as a matrix (with the spectra in the columns), then the product is element-by-element using R's `*` - the Hadamard product. The numbers of spectra in the terms must be conformable. If some objects have 1 spectrum and all the others have q, then the column-vector spectrums are repeated q times to form a matrix with q columns. If the numbers of spectra are not conformable, it is an ERROR and the function returns `NULL`.
As an example, suppose M_1 has 1 spectrum and M_2 has q spectra, and $m = 2$. Then the product is a material with q spectra. Think of an IR-blocking filter followed by the RGB filters in a 3-CCD camera.

2. $L * M_1 * M_2 * ... * M_m \mapsto L'$
The product of a light source followed by m materials is a light source. Think of a light source followed by a stack of m transmitting filters, effectively forming a new light source. The numbers of spectra in the terms must be conformable as in sequence 1, and the matrices are multiplied element by element.
As an example, suppose L has 1 spectrum and M_1 has q spectra, and $m = 1$. Then the product is a light source with q spectra. Think of a light source followed by a filter wheel with q filters.

3. $M_1 * M_2 * ... * M_m * R_L \mapsto R'_L$
The product of m materials followed by a light responder, is a light responder. Think of a stack of m transmitting filters in front of a camera, effectively forming a new camera. The numbers of spectra in the terms must be conformable as in sequence 1, and the matrices are multiplied element by element.
As an example, suppose R_L has 1 spectrum and M_1 has q spectra, and $m = 1$. Then the product is a light responder with q spectra. Think of a 3-CCD camera in which all 3 CCDs have exactly the same responsivity and so can be modeled with a single object R_L.

4. $L * M_1 * ... * \bullet * ... * M_m * R_L \mapsto R'_M$
This is the strangest product. The bullet symbol $\bullet$ means that a variable material is inserted at that slot in the sequence (or light path). For each material spectrum inserted there is a response from R_L. Therefore the product of this sequence is a material responder R_M. Think of a light source L going through a transparent object $\bullet$ on a flatbed scanner and into a camera R_L. For more about the mathematics of this product, see the **colorSpec-guide.pdf** in the doc directory. These material

responder spectra are the same as the *effective spectral responsivities* in *Digital Color Management*. The numbers of spectra in the terms must be conformable as in sequence 1, and the product is a material responder with q spectra.
In the function `product()` the location of the • is marked by any character string whatsoever - it's up to the user who might choose something that describes the typical material (between the light source and camera). For example one might choose:
`scanner = product( A.1nm, 'photo', Flea2.RGB, wave='auto')`
to model a scanner that is most commonly used to scan photographs. Other possible strings could be `'artwork'`, `'crystal'`, `'varmaterial'`, or even `'slot'`. See the vignette **Viewing Object Colors in a Gallery** for a worked-out example.

5. $L * M_1 * M_2 * ... * M_m * R_L \mapsto matrix$
The product of a light source, followed by m materials, followed by a light responder, is a matrix! The numbers of spectra in the terms must splittable into a conformable left part (L' from sequence 2.) and a conformable right part (R'_L from sequence 3.). There is a row for each spectrum in L', and a column for each spectrum in R'_L. Suppose the element-by-element product of the left part is $n \times p$ and the element-by-element product of the right part is and $n \times q$, where n is the number of wavelengths. Then the output matrix is the usual matrix product `%*%` of the transpose of the left part times and right part, which is $p \times q$.
As an example, think of a light source followed by a reflective color target with 24 patches followed by an RGB camera. The sequence of spectra counts is `c(1,24,3)` which is splittable into `c(1,24)` and `c(3)`. The product matrix is 24×3. See the vignette **Viewing Object Colors in a Gallery** for a worked-out example.
Note that is OK for there to be no materials in this product; it is OK if $m = 0$. See the vignette **Blue Flame and Green Comet** for a worked-out example.

6. $M_1 * M_2 * ... * M_m * R_M \mapsto matrix$
The product of m materials followed by a material responder, is a matrix ! The sequence of numbers of spectra must be splittable into left and right parts as in sequence 4, and the product matrix is formed the same way. One reason for computing this matrix in 2 steps is that one can `calibrate` the material responder separately in a customizable way. See the vignette **Viewing Object Colors in a Gallery** for a worked-out example with a flatbed scanner.

Note that sequences 5. and 6. are the only ones that use the usual matrix product `%*%`. They may also use the Hadamard matrix product `*`, as in sequences 1 to 4.

The argument `wavelength` can also be `'auto'` or `NULL`. In this case the intersection of all the wavelength intervals of the objects is computed. If the intersection is empty, it is an ERROR and the function returns `NULL`. The wavelength step `step.wl` is taken to be the smallest step over all the object wavelength sequences. If the minimum `step.wl` is less than 1 nanometer, it is rounded off to the nearest power of 2 (e.g 1, 0.5, 0.25, ...).

Value

`product()` returns either a **colorSpec** object or a matrix, see **Details**.

If `product()` returns a **colorSpec** object, the `organization` of the object is `'matrix'` or `'vector'`; any `extradata` is lost. However, all terms in the product are saved in `attr(*,'sequence')`. One can use `str()` to inspect this attribute.

If `product()` returns a matrix, this matrix can sometimes be ambiguous, see **Note**.

All actinometric terms are converted to radiometric on-the-fly and the returned **colorSpec** object is also radiometric.

In case of ERROR it returns `NULL`.

Note

The product for sequences 1, 2, and 3 is associative. After all matrices are filled out to have q columns, the result is essentially a Hadamard product of matrices, which is associative. Also note that a subsequence of sequences 2 and 3 might be sequence 1.

The product for sequence 4 is never associative, since subproducts that contain the variable $\bullet$ are undefined. However the result is essentially a Hadamard product of matrices, and is unambiguous.

The product for sequence 5 is associative in special cases, but not in general. The problem is that the left and right splitting point is not unique. If all objects have only a single spectrum, then it *is* associative, and therefore unambiguous. If the left part has a different number of multiple spectra than the right part, then it is not associative in general since some ways of grouping the product may be undefined.
Moreover, in some cases the product can be ambiguous. Suppose that the vector of spectrum counts is `c(1,3,1)`; this could come from a single light source, followed by 3 filters (e.g. RGB), followed by a graylevel camera. There are 2 ways to split this: `"1|3,1"` and `"1,3|1"`. The first split is interpreted as the light source into a camera with 3 channels. The second split is interpreted as 3 colored light sources into a graylevel camera. In the first split the returned matrix is a `1x3` row vector. In the second split the returned matrix is a `3x1` column vector. For the vector `"1,3,1"`, one can show that the computed components in the 2 vectors are equal, so the ambiguity is benign. But consider the longer sequence `"1,3,3,1"`. There are 3 ways to split this, and the returned matrices are `1x3`, `3x3`, and `3x1`. So this ambiguity is obviously a problem. Whenever there is an ambiguity, the function chooses a splitting in which the left part is as long as possible, and issues a warning message. The user should inspect the result carefully. To avoid the ambiguity, the user should break the product into smaller pieces and call `product()` multiple times.

The product for sequence 6 is essentially the same as sequence 5, and the function issues a warning message when appropriate. Note that a subproduct is defined only if it avoids the final multiplication with R_M.

References

Edward J. Giorgianni and Thomas E. Madden. **Digital Color Management: Encoding Solutions.** 2nd Edition John Wiley. 2009. Figure 10.11a. page 141.

Wikipedia. **Hadamard product (matrices)**. https://en.wikipedia.org/wiki/Hadamard_product_(matrices)

ASTM E308-01. Standard Practice for Computing the Colors of Objects by Using the CIE System. (2001).

See Also

wavelength, type, resample, calibrate, radiometric, step.wl

Examples

```
#  sequence 1.
path = system.file( "extdata/objects/Midwest-SP700-2014.txt", package='colorSpec' )
blocker.IR = readSpectra( path )
product( blocker.IR, Hoya, wave='auto' )


#  sequence 2.
product( subset(solar.irradiance,1), atmosphere2003, blocker.IR, Hoya, wave='auto' )


#  sequence 3.
plumbicon = readSpectra( system.file( "extdata/cameras/plumbicon30mm.txt", package='colorSpec' ) )
product( blocker.IR, subset(Hoya,1:3), plumbicon, wave='auto' )


#   sequence 4.
#   make an RGB scanner
bluebalancer = subset(Hoya,'LB')
# combine tungsten light source A.1nm with blue light-balance filter
# use the string 'artwork' to mark the variable material slot
scanner = product( A.1nm, bluebalancer, 'artwork', Flea2.RGB, wave='auto' )


#  sequence 5.
product( D65.1nm, Flea2.RGB, wave='auto' )  # a 1x3 matrix, no materials
product( D65.1nm, neutralMaterial(0.01), Flea2.RGB, wave='auto' ) # a 1x3 matrix, 1 material
path = system.file( "extdata/sources/Lumencor-SpectraX.txt", package='colorSpec' )
lumencor = readSpectra( path, wave=340:660 )
product( lumencor, Flea2.RGB, wave='auto' )   # a 7x3 matrix, no materials


#  sequence 6.
scanner = calibrate( scanner )
target = readSpectra( system.file( "extdata/targets/N130501.txt", package='colorSpec') )
product( target, scanner, wave='auto' )  #  a 288x3 matrix
```

---

ptransform *make a linear transformation to a colorSpec responder*

Description

apply a linear transformation to a **colorSpec** responder with M spectra, so that multiples of M given *primary* vectors are transformed to the standard basis of R^M. And a given *white* vector is

transformed to the M-vector of all 1s.
The returned object is always `multiply(x,A)` where `A` is an internally calculated MxM matrix. The name `ptransform` is short for *projective transformation*.
In case of ERROR, a message is logged and NULL returned.

Usage

```
## S3 method for class 'colorSpec'
ptransform( x, primary, white, digits=Inf )
```

Arguments

`x`	a **colorSpec** responder with M spectra. `type(x)` must be `'responsivity.light'` or `'responsivity.material'`.
`primary`	an MxM matrix whose rows define the M primary vectors in the response space of `x`. It is OK for each row to have a single value that is NA. In this case the NA value is changed so that the sum of the row is 1. This is done because typically the rows represent chromaticities in the response space of `x`. After this change, the rows of `primary` must form a basis of R^M. `rownames(primary)` must be defined; when M=3 they are typically `c('R','G','B')`.
`white`	an M-vector in the response space of `x`, that is typically the ideal white point of a color display. When `white` is expressed in the basis defined by `primary`, the coordinates must all be non-zero. `white` can also be a **colorSpec** object with a single spectrum suitable as stimulus for `x`; in this case the vector `white` is set to `product( white, x, wavelength='auto' )`.
`digits`	if a positive integer, then `white` is rounded to this number of decimal digits, but in a non-standard way, see **Details**. This is typically done so the internally calculated MxM matrix `A` agrees with that from a color standard, see **Examples**.

Details

The formal mathematical requirements for `primary` and `white` are:

- The rows of `primary` must form a basis of R^M. Equivalently, if P denotes the matrix `primary`, then P is invertible.
- The coordinates of `white` in this basis are all non-zero. Equivalently, if x is the solution of $xP = white$, then every component of x is non-zero.

Assuming both of these are true, then there is a unique matrix A so that

- A transforms a multiple of the i'th row of P to the i'th standard basis vector of R^M.
- A transforms `white` to the M-vector of all 1s.

This statement is essentially the fundamental theorem of (analytic) projective geometry; see *Bumcrot* page 87, and *Semple* page 398. The rows of P plus $white$ define a *projective frame*; the former are the *fundamental points* and the latter is the *unit point*.

If digits is a positive integer, the chromaticity of white is computed by dividing white by sum(white). The latter must be non-zero, or else it is an ERROR. This chromaticity is rounded to digits decimal digits, while preserving the sum of 1. This *rounded chromaticity* is non-zero, and defines a line through 0. The vector white is projected onto this line to get the new and rounded white, with the rounded chromaticity. See **Examples**.

Value

a **colorSpec** object equal to multiply(x,A) where A is an internally calculated MxM matrix. The quantity and wavelength are preserved. The specnames of the returned object are set to tolower(rownames(primary)).
The user may want to change the quantity of the returned object; see **Examples**.

References

Bumcrot, Robert J. **Modern Projective Geometry**. Holt, Rinehart, and Winston. 1969.

IEC 61966-2-1:1999. Multimedia systems and equipment - Colour measurement and management. Part 2-1: Colour management - Default RGB colour space - sRGB. https://webstore.iec.ch/publication/6169

Semple, J. G. and G. T. Kneebone. **Algebraic Projective Geometry**. Oxford. 1952.

See Also

quantity, wavelength, colorSpec, multiply, product

Examples

```
############ Example for sRGB   ############

# assign the standard sRGB primaries
P = matrix( c(0.64,0.33,NA,  0.3,0.6,NA, 0.15,0.06,NA ), 3, 3, byrow=TRUE )
rownames(P) = c('R','G','B')
P
#   [,1] [,2] [,3]
# R 0.64 0.33   NA
# G 0.30 0.60   NA
# B 0.15 0.06   NA

white = product( D65.1nm, xyz1931.1nm, wave='auto' )
white
#           X        Y        Z
# D65 100.437 105.6708 115.0574

white/sum(white)
#             X         Y         Z
# D65 0.3127269 0.3290232 0.3582499

# But the sRGB standard D65 is xy=(0.3127,0.3290)
# so when the next line is executed,
# the calculated 3x3 matrix will *NOT* agree with the sRGB standard
y = ptransform( xyz1931.1nm, P, white, digits=Inf )
```

```
product( D65.1nm, y, wave='auto' )
#     R G B
# D65 1 1 1     # this is exactly what we want, but the internal 3x3 matrix is a little off

# now repeat, but this time round the white chromaticity to
# xy=(0.3127,0.3290) in order to get the matrix right
y = ptransform( xyz1931.1nm, P, white, digits=4 )

rgb = product( D65.1nm, y, wave='auto' )
rgb
#            R        G        B
# D65 1.000238 1.000053 0.999835   # off in the 4'th digit  (WARN: this is linear RGB)

255 * rgb
#            R        G        B
# D65 255.0607 255.0134 254.9579   # good enough for 8-bit RGB

65535 * rgb
#            R        G        B
# D65 65550.59 65538.44 65524.18   # NOT good enough for 16-bit RGB

# So for 16-bit RGB, one can get the white RGB right, or the 3x3 matrix right, but not both !


############ Example for ProPhoto RGB   ############

# assign the standard ProPhoto RGB primaries
P = matrix( c(0.7347,0.2653,NA,  0.1596,0.8404,NA, 0.0366,0.0001,NA ), 3, 3, byrow=TRUE )
rownames(P) = c('R','G','B')
P
#     [,1]   [,2] [,3]
# R 0.7347 0.2653   NA
# G 0.1596 0.8404   NA
# B 0.0366 0.0001   NA

white = product( D50.5nm, xyz1931.5nm, wave='auto' )
white
#            X       Y        Z
# D50 101.2815 105.042 86.67237

white / sum(white)
#             X         Y         Z
# D50 0.3456755 0.3585101 0.2958144

# but the ProPhoto RGB standard is xy=(0.3457,0.3585);  proceed anyway
y = ptransform( xyz1931.5nm, P, white, digits=Inf )

product( D50.5nm, y, wave='auto' )
#     R G B
# D50 1 1 1    #  this is exactly what we want, but the internal 3x3 matrix is a little off

# the following line is an equivalent way to compute y.
```

```
# pass D50.5nm directly as the 'white' argument
y = ptransform( xyz1931.5nm, P, D50.5nm )
```

---

quantity *quantity of a colorSpec object*

Description

Retrieve or set the quantity of a **colorSpec** object.

Usage

```
## S3 method for class 'colorSpec'
quantity(x)

## S3 replacement method for class 'colorSpec'
quantity(x) <- value

## S3 method for class 'colorSpec'
type(x)
```

Arguments

x	a **colorSpec** R object
value	a valid quantity string; see Details.

Details

There are 4 valid types, which are further divided into 14 valid quantities. All of these are strings:

For type='light'
quantity = 'energy' (radiometric), or 'photons' (actinometric)

For type='responsivity.light'
quantity = 'energy->electrical', 'energy->neural', 'energy->action',
'photons->electrical', 'photons->neural', or 'photons->action'

For type='material'
quantity = 'reflectance', 'transmittance', or 'absorbance'

For type='responsivity.material'
quantity = 'material->electrical', 'material->neural', or 'material->action'

Also see the **colorSpec User Guide**.

Value

quantity() returns the quantity of x

type() returns the type of x, which depends only on the quantity.

Note

The **colorSpec** quantity is more general than the physical SI quantity; for example quantity='energy' really includes 10 distinct SI quantities and maybe more. The unit is left arbitrary in most cases. Exceptions are reflectance, transmittance, and absorbance which are dimensionless.

Changing the quantity should only be done if one knows what one is doing. It does not change the underlying numbers. For example, changing photons to energy does not do numerical conversion. For this specific conversion, see radiometric().

Similarly, see linearize() for conversion from absorbance to transmittance.

See Also

colorSpec, radiometric, linearize

---

radiometric *convert a colorSpec object from actinometric to radiometric*

Description

Convert a **colorSpec** object to have quantity that is radiometric (energy of photons) - to prepare it for colorimetric calculations. Test an object for whether it is radiometric.

Usage

```
## S3 method for class 'colorSpec'
radiometric( x, multiplier=1, warn=FALSE )

## S3 method for class 'colorSpec'
is.radiometric( x )
```

Arguments

x	a **colorSpec** object
multiplier	a scalar which is multiplied by the output, and intended for unit conversion
warn	if TRUE and a conversion actually takes place, the a WARN message is issued. This makes the user aware of the conversion, so units can be verified. This can be useful when radiometric() is called from another **colorSpec** function.

Details

If the quantity of x does not start with 'photons' then the quantity is not actinometric and so x is returned unchanged. Otherwise x is actinometric (photon-based).

If type(x) is 'light' then the most common actinometric unit of photon count is (μmole of photons) = ($6.02214x10^{17}$ photons). The conversion equation is:

$$E = Q * 10^{-6} * N_A * h * c/\lambda$$

where E is the energy of the photons, Q is the photon count, N_A is Avogadro's constant, h is Planck's constant, c is the speed of light, and λ is the wavelength in meters. The output energy unit is joule.
If the unit of Q is not (μmole of photons), then the output should be scaled appropriately. For example, if the unit of photon count is exaphotons, then set multiplier=1/0.602214.

If the quantity(x) is 'photons->electrical', then the most common actinometric unit of responsivity to light is quantum efficiency (QE). The conversion equation is:

$$R_e = QE * \lambda * e/(h * c)$$

where R_e is the energy-based responsivity, QE is the quantum efficiency, and e is the charge of an electron (in C). The output responsivity unit is coulombs/joule (C/J) or amps/watt (A/W).
If the unit of x is not quantum efficiency, then multiplier should be set appropriately.

If the quantity(x) is 'photons->neural' or 'photons->action', the most common actinometric unit of photon count is (μmole of photons) = ($6.02214x10^{17}$ photons). The conversion equation is:

$$R_e = R_p * \lambda * 10^6/(N_A * h * c)$$

where R_e is the energy-based responsivity, R_p is the photon-based responsivity. This essentially the reciprocal of the first conversion equation.

The argument multiplier is applied to the right side of all the above conversion equations.

Value

radiometric() returns a **colorSpec** object with quantity that is radiometric (energy-based) and not actinometric (photon-based). If type(x) is a material type ('material' or 'responsivity.material') then x is returned unchanged.

If quantity(x) starts with 'energy', then is.radiometric() returns TRUE, and otherwise FALSE.

Note

To log the executed conversion equation, execute cs.options(loglevel='INFO').

Source

Wikipedia. **Photon counting**. https://en.wikipedia.org/wiki/Photon_counting

See Also

quantity, type, F96T12, cs.options, actinometric

Examples

```
sum( F96T12 )    # the step size is 1nm, from 300 to 900nm
# [1] 320.1132  photon irradiance, (micromoles of photons)*m^{-2}*sec^{-1}

sum( radiometric(F96T12) )
# [1] 68.91819  irradiance, watts*m^{-2}
```

`readCGATS` *read tables from files in ANSI/CGATS.17 format*

Description

The CGATS text format supports a preamble followed by N tables, where N $\geq$ 1. Each table can have a separate header. A table may or may not contain spectral data, see **Note**. The function converts each table to a `data.frame` with attributes; see **Details**.

Usage

```
readCGATS( path, collapsesingle=FALSE )
```

Arguments

`path`	the path name of a single file, in CGATS format
`collapsesingle`	If `path` has only one table (N=1) and `collapsesingle` is `TRUE`, then return the single `data.frame` (instead of a list with 1 `data.frame`). If `path` has multiple tables (N $\geq$ 2), then `collapsesingle` is ignored.

Details

The returned list is given attributes: `"path"`, `"preamble"`, and (if present) `"date"`, `"created"`, `"originator"`, and `"file_descriptor"`. The attribute values are all character vectors. The value of attribute `"path"` is the argument `path`, and the other values are extracted from `"preamble"`. The length of `"preamble"` is (typically) greater than 1, and the others have length 1. Each line of the preamble is a keyword-value pair. The keyword `ORIGINATOR` is converted to attribute `"originator"`. The keyword `FILE_DESCRIPTOR` is converted to attribute `"file_descriptor"`. The keyword `CREATED` is converted to attributes `"created"` and `"date"`. The list is also given `names`. If the keyword `TABLE_NAME` is present in the table header, then its value is used. Otherwise the names are `"TABLE_1"`, `"TABLE_2"`, ...

Each `data.frame` in the list is assigned attributes: `"header"`, and (if present) `"descriptor"`. The length of `"header"` is (typically) greater than 1, and `"descriptor"` has length 1. Each line of the table header is a keyword-value pair. The keywords `DESCRIPTOR` and `TABLE_DESCRIPTOR` are converted to attribute `"descriptor"`.

For the lines between `BEGIN_DATA` and `END_DATA`, two conventions for separating the values are supported:

- In the standard convention, fields are separated by contiguous spaces or tabs, and character strings (which may have embedded spaces or even tabs) are enclosed by double-quotes. This is is the convention in the CGATS standard. The function `scan()` is used here.
- In the non-standard convention, fields are separated by a *single* tab, and character strings (which may have embedded spaces but not tabs) are *not* enclosed by double-quotes. This convention is often easier to work with in spreadsheet software. The function `strsplit()` is used here.

The function `readCGATS()` selects the separation convention by examining the line after `BEGIN_DATA_FORMAT`. If this line is split by a single tab and the number of fields matches that given on the `NUMBER_OF_FIELDS` line, then the non-standard convention is selected; otherwise, the standard convention is selected.

Value

`readCGATS()` returns a list of `data.frames` - one `data.frame` for each table found in `path`. The list and each individual `data.frame` have attributes, see **Details**.

If `path` has only a single table (the majority of files have only 1) and `collapsesingle` is `TRUE`, then the attributes of the list are copied to those of the `data.frame`, and the `data.frame` is then returned. The name of the table is lost.

If there is an error in any table, then the function returns `NULL`.

Note

In the `BEGIN_DATA_FORMAT` line(s), field names may not be quoted and may not have embedded spaces.
The CGATS standard allows duplicated field names, and `readCGATS()` returns them as they appear, with no attempt to append numbers in order to make them unique. Examples of field names which may be duplicated are: `SPECTRAL_NM`, `SPECTRAL_DEC`, and `SPECTRAL_PCT`; for more on these see `readSpectraCGATS()`.
No attempt is made to recognize those tables that contain spectral data. For conversion of spectral data to **colorSpec** objects, see `readSpectraCGATS()`.

References

ANSI/CGATS.17. Graphic technology - Exchange format for colour and process control data using XML or ASCII text. https://webstore.ansi.org/ 2009.

ISO/28178. Graphic technology - Exchange format for colour and process control data using XML or ASCII text. https://www.iso.org/standard/44527.html. 2009.

CGATS.17 Text File Format. http://www.colorwiki.com/wiki/CGATS.17_Text_File_Format.

See Also

`readSpectraCGATS`, `scan`, `strsplit`, `names`

Examples

```
#   read file with 2 tables of non-spectral data
A70 = readCGATS( system.file( "extdata/targets/A70.ti3", package='colorSpec' ) )
length(A70)         # [1] 2   # the file has 2 tables
```

```
ncol( A70[[1]] )    # [1] 7   # the 1st table has 7 columns
ncol( A70[[2]] )    # [1] 4   # the 2nd table has 4 columns
```

---

readSpectra *read colorSpec objects from files*

Description

These functions read **colorSpec** objects from files. In case of ERROR, they return NULL. There are 5 different file formats supported; see **Details**.

Usage

```
readSpectra( pathvec, ... )

readSpectraXYY( path )
readSpectraSpreadsheet( path )
readSpectrumScope( path )
readSpectraCGATS( path )
readSpectraControl( path )
```

Arguments

pathvec	a character vector to (possibly) multiple files. The file extension and a few lines from each file are read and a guess is made regarding the file format.
...	optional arguments passed on to resample(). The most important is wavelength. If these are missing then resample() is not called.
path	a path to a single file with the corresponding format: XYY, Spreadsheet, Scope, CGATS, or Control. See Details. If the function cannot recognize the format, it returns NULL.

Details

readSpectra() reads the first few lines of the file in order to determine the format, and then calls the corresponding format-specific function. If readSpectra() cannot determine the format, it returns NULL. The 5 file formats are:

XYY

There is a column header line matching ^(wave|wv?l) (not case sensitive) followed by the the names of the spectra. All lines above this one are taken to be metadata. The separarator on this header line can be space, tab, or comma; the line is examined and the separator found there is used in the lines with data below. The organization of the returned object is 'df.col'. This is probably the most common file format; see the sample file ciexyz31_1.csv.

Spreadsheet

There is a line matching "^(ID|SAMPLE|Time)". This line and lines below must be tab-separated.

Fields matching '^[A-Z]+([0-9.]+)nm$' are taken to be spectral data and other fields are taken to be extradata. All lines above this one are taken to be metadata. The organization of the returned object is 'df.row'. This is a good format for automated acquisition, using a spectrometer, of many spectra. See the sample file N130501.txt from **Wolf Faust**.

Scope
This is a file format used by **Ocean Optics** spectrometer software. There is a line
>»»Begin Processed Spectral Data<««
followed by wavelength and energy separated by a tab. There is only 1 spectrum per file. The organization of the returned object is 'vector'. See the sample file pos1-20x.scope.

CGATS
This is a complex format that is best understood by looking at some samples, such as extdata/objects/Rosco.txt; see also the **References**. The function readCGATS() is first called to get all the tables, and then for each table the column names are examined. There are 2 conventions for presenting the spectral data:

- In the standard convention the fields SPECTRAL_DEC or SPECTRAL_PCT have the spectral values. The former is the true value, and the latter is the true value x 100. Each value column is preceded a corresponding wavelength column, which has the field name SPECTRAL_NM. Note that these field names are highly duplicated. In principle, this convention allows each record in a CGATS table to have a different wavelength vector. However, this complication is rejected by readSpectraCGATS(), which treats it as an ERROR.
- In the non-standard convention the field names that match the pattern "^(nm|SPEC_|SPECTRAL_)[_A-Z]*([0-9.]+)$" are considered to be spectral value data, and other fields are considered extradata. The wavelength is the numerical value of the 2nd parenthesized expression ([0-9.]+) in nanometers. Note that every record in this CGATS table has the same wavelength vector. Although this convention is non-standard, it appears in files from many companies, including X-Rite.

If a data.frame has spectral data, it is converted to a **colorSpec** object and placed in the returned list. The organization of the resulting **colorSpec** object is 'df.row'. If the data.frame of extradata contains a column SAMPLE_NAME, SAMPLE_ID, SampleID, or Name, (examined in that order), then that column is taken to be the specnames of the object. If a table has no spectral data, then it is ignored. If the CGATS file has no tables with spectral data, then it is an ERROR and the function returns NULL.

Control
This is a personal format used for digitizing images of plots from manufacturer datasheets and academic papers. It is structured like a .INI file. There is a [Control] section establishing a simple linear map from pixels to the wavelength and spectrum quantities. Only 3 points are really necessary. It is OK for there to be a little rotation of the plot axes relative to the image. This is followed by a section for each spectrum, in XY pixel units only. Conversion to wavelength and spectral quantities is done on-the-fly after they are read. Extrapolation can be a problem, especially when the value is near 0. To force constant extrapolation (see resample()), repeat the control point (knot) at the endpoint. See the sample file Lumencor-SpectraX.txt. The organization of the returned objects is 'vector'.

Value

readSpectra() returns a single **colorSpec** object or NULL in case of ERROR. If there are multiple files in pathvec and they cannot be combined using bind() because their wavelengths are different, it is an ERROR. To avoid this ERROR, the wavelength argument can be used for resampling to common wavelengths. If there are multiple files, the organization of the returned object is 'df.row' and the first column is the path from which the spectrum was read.

The functions readSpectraXYY(), readSpectraSpreadsheet(), and readSpectraScope(), return a single **colorSpec** object, or NULL in case of ERROR.

The functions readSpectraCGATS() and readSpectraControl() are more complicated. These 2 file formats can contain multiple spectra with different wavelength sequences, so both functions return a *list* of **colorSpec** objects, even when that list has length 1. If no spectral objects are found, they return NULL.

If readSpectra() calls readSpectraCGATS() or readSpectraControl() and receives a list of **colorSpec** objects, readSpectra() attempts to bind() them into a single object. If they all have the same wavelength vector, then the bind() succeeds and the single **colorSpec** object is returned. Otherwise the bind() fails, and it is an ERROR. To avoid this error readSpectra() can be called with a wavelength argument. The multiple spectra are resampled using resample() and *then* combined using bind(), which makes it much more convenient to read such files.

Note

During import, each read function tries to guess the quantity from spectrum names or other cues. For example the first line in **N130501.txt** is IT8.7/1, which indicates that the quantity is 'transmittance' (a reflective target is denoted by IT8.7/2). If a confident guess cannot be made, it makes a wild guess and issues a warning. If the quantity is incorrect, one can assign the correct value after import. Alternatively one can add a line to the header part of the file with the keyword 'quantity' followed by some white-space and the correct value. It is OK to put the value in quotes. See example files under folder **extdata**.

References

CGATS.17 Text File Format. http://www.colorwiki.com/wiki/CGATS.17_Text_File_Format.

ANSI/CGATS.17. Graphic technology - Exchange format for colour and process control data using XML or ASCII text. https://webstore.ansi.org/ 2009.

ISO/28178. Graphic technology - Exchange format for colour and process control data using XML or ASCII text. https://www.iso.org/standard/44527.html. 2009.

See Also

wavelength, quantity, metadata, resample, bind, readCGATS

Examples

```
#   read file with header declaring the quantity to be "photons->neural"
bird = readSpectra( system.file( "extdata/eyes/BirdEyes.txt", package='colorSpec' ) )
quantity(bird)   # [1] "photons->neural"
```

resample *resample a colorSpec Object to new wavelengths*

Description

interpolate or smooth to new wavelengths. Simple extrapolation and clamping is also performed.

Usage

```
## S3 method for class 'colorSpec'
resample( x, wavelength, method='auto', span=0.02, extrapolation='const', clamp='auto' )
```

Arguments

x	a **colorSpec** object
wavelength	vector of new wavelengths, in nanometers
method	interpolation methods available are 'sprague', 'spline', and 'linear'. Also available is 'auto' which means to use 'sprague' if x is regular, and 'spline' otherwise. An available smoothing method is 'loess'. See **Details**.
span	smoothing argument passed to loess() during interpolation, and not used by other methods. The default value span=0.02 is suitable for .scope spectra but may be too small in many other cases.
extrapolation	extrapolation methods available are 'const' and 'linear'. These can be abbreviated to the initial letter. Also available is a numeric value, which is used for simple padding. See **Details**. Also available is a vector or list of length 2 that combines 2 of the above. The first item is used on the low side (shorter wavelengths), and the second item is used on the high side (longer wavelengths).
clamp	clamp methods available are 'auto', TRUE, and FALSE. Also available is a numeric vector of length 2, which defines the clamping interval. See **Details**.

Details

If method is 'sprague', the quintic polynomial in *De Kerf* is used. Six weights are applied to nearby data values: 3 on each side. The 'sprague' method is only supported when x is regular.

If method is 'spline', the function stats::spline() is called with method='natural'. The 'spline' method is supported even when x is irregular.

If method is 'linear', the function stats::approx() is called. Two weights are applied to nearby data values: 1 on each side. The 'linear' method is supported even when x is irregular.

If method is 'loess', the function stats::loess() is called with the given span parameter. Smoothing is most useful for noisy data, e.g. raw data from a spectrometer. I have found that span=0.02 works well for Ocean Optics .scope files, but this may be too small in other cases, which triggers an error in stats::loess(). The 'loess' method is supported even when x is irregular.

If extrapolation is 'const', the extreme values at each end are simply extended. If extrapolation is 'linear', the line defined by the 2 extreme values at each end is used for extrapolation. If the ultimate and penultimate wavelengths are equal, then this line is undefined and the function reverts to 'const'.

If clamp is 'auto', output values are clamped to the physically realizable interval appropriate for x. This is the interval [0,1] when quantity(x) is 'reflectance' or 'transmittance', and the interval $[0,\infty)$ otherwise. Exception: If an input spectrum in x violates a limit, then clamping the output spectrum to this limit is *NOT* enforced. This happens most frequenty for theoretical (or matrixed) cameras, such as BT.709.RGB.

If clamp is TRUE, the result is the same as 'auto', but with no exceptions. If clamp is FALSE, then no clamping is done.

If clamp is a numerical interval, then clamping is done to that interval, with no exceptions. The two standard intervals mentioned above can be expressed in **R** as c(0,1) and c(0,Inf) respectively.

Value

resample(x) returns a **colorSpec** object with the new wavelength. Other properties, e.g. organization, quantity, ..., are preserved.
In case of ERROR, the function returns NULL.

References

De Kerf, Joseph L. F. **The interpolation method of Sprague-Karup**. *Journal of Computational and Applied Mathematics*. volume I, no 2, 1975. equation (S).

See Also

organization(), quantity(), wavelength(), is.regular(), theoreticalRGB, spline(), approx(), loess

Examples

```
path = system.file( "extdata/sources/pos1-20x.scope", package='colorSpec' )
y = readSpectra( path )
# plot noisy data in gray
plot( y, col='gray' )
# plot smoothed plot in black on top of the noisy one to check quality
plot( resample( y, 200:880, meth='loess', span=0.02 ), col='black', add=TRUE )
```

---

responsivityMetrics *Compute Metrics for a Light Responder (e.g. a camera) or a Material Responder (e.g. a scanner)*

Description

This function computes a few technical metrics regarding some geometric objects related to a responder: the spherical chromaticity polygon, cone, convex cone, and color-solid.

Currently the function only works if the number of spectra in `x` is 3 (e.g. RGB or XYZ). In this case the rows of `as.matrix(x)` (after weighting by step size) are called the *generators*; they are vectors in R^3 and we require that they are all in some open linear halfspace (unless a generator is 0). The 0-based rays through the generators intersect a plane inside the halfspace to form the vertices of the *chromaticity polygon* P. The 0-based rays through points of the interior of P form a cone, and the convex hull of this cone is a convex cone. The central projection of P onto the unit sphere is the *spherical chromaticity polygon* P_S. If `type` is `'responsivity.material'`, then `x` has an *object-color solid* or *Rösch Farbkörper*, which is a zonohedron Z. See *Centore* and vignette **Convexity and Transitions** for details.

Some simplification of the generators is performed during pre-processing. Generators that are 0 (in all channels) are removed, and a group of generators that are all positive multiples of each other is replaced by their sum. The 3-vectors are called the *condensed generators*. These simplifications do not change any of the geometric objects defined above.

Usage

```
## S3 method for class 'colorSpec'
responsivityMetrics( x )
```

Arguments

`x`	a **colorSpec** object with `type` equal to `'responsivity.light'` or `'responsivity.material'`, and 3 spectra

Value

`responsivityMetrics()` returns a list with these items:

`generators`	a pair of integers, the 1st is the number of original generators, and the 2nd is the number of condensed generators
`zeros`	vector of wavelengths at which the responsivity is 0 (in all 3 channels)
`multiples`	a list of vectors of wavelengths; the responsivities in each vector (group) are positive multiples of each other
`salient`	a logical where TRUE means that there is some open linear halfspace that contains all the non-zero generators. If all the responsivities are non-negative, which is the usual case, then `salient=TRUE`.
`normal`	If `salient=TRUE`, then the inward pointing unit normal for the previous halfspace. Otherwise, `normal=NA`.

If `salient=TRUE`, then the list also contains:

`concavities`	a `data.frame` with 2 columns: `wavelength` and `extangle`, where `extangle` is the external angle at the wavelength (for the spherical chromaticity polygon P_S), and is negative. A negative angle means that P_S is concave at that vertex.

coneangle	the solid angle of the cone generated by the generators. This is identical to the area of the spherical chromaticity polygon, with concavities preserved.
cxconeangle	the solid angle of the convex cone generated by the generators, with no concavities. This is identical to the area of the convex hull of the spherical chromaticity polygon. If all responsivities are non-negative, which is the usual case, then this solid angle is less than the solid angle of an octant, which is $\pi/2$. We always have coneangle <= cxconeangle.

If the type of x is 'responsivity.material' then the list also contains:

area	the surface area of the object-color solid of x
volumesch	the volume enclosed by the Schrödinger colors of x
volumeopt	the volume of the object-color solid of x. We always have volumesch <= volumeopt.

In case of global error, the function returns NULL.

Note

To determine the value of salient, the package **quadprog** might be required.

References

Centore, Paul. *A zonohedral approach to optimal colours*. **Color Research & Application**. Vol. 38. No. 2. pp. 110-119. April 2013.

See Also

type, vignette **Convexity and Transitions**

Examples

```
D65.eye = product( D65.5nm, 'varmat', xyz1931.1nm, wave=wavelength(D65.5nm) )
responsivityMetrics( D65.eye )
```

```
##  $generators
##  [1] 81 65
##
##  $zeros
##  numeric(0)
##
##  $multiples
##  $multiples[[1]]
##   [1] 700 705 710 715 720 725 730 735 740 745 750 755 760 765 770 775 780
##
##  $salient
##  [1] TRUE

##  $normal
##  [1] 0.5773503 0.5773503 0.5773503

##  $concavities
```

```
##     wavelength      extangle
##  2         385 -0.3870584861
##  4         395 -0.4393794850
##  13        440 -0.0255522232
##  14        445 -0.0075938225
##  41        580 -0.0010472855
##  42        585 -0.0006600542
##  44        595 -0.0017602134
##  45        600 -0.0004783000
##  46        605 -0.0026858850
##  49        620 -0.0037210312
##  53        640 -0.0011878846
##  54        645 -0.0029247630
##
##  $coneangle
##  [1] 1.185259
##
##  $cxconeangle
##  [1] 1.185282
##
##  $area
##  [1] 39873.89
##
##  $volumesch
##  [1] 511128.5
##
##  $volumeopt
##  [1] 511129.5
```

`scanner` *standard RGB scanners*

Description

`scanner.ACES` is an RGB responder to material; an ACES/SMPTE standard for scanning RGB film. The 3 spectra are defined from 368 to 728 nm, at 2nm intervals.

Format

A **colorSpec** object with `quantity` equal to `'material->electrical'` and 3 spectra: `r`, `g`, and `b`.

Details

The responsivities have been scaled (by `calibrate`) so the response to the *perfect transmitting filter* (PTF) is RGB=(1,1,1).

References

Technical Bulletin TB-2014-005. Informative Notes on SMPTE ST 2065-2 - Academy Printing Density (APD). Spectral Responsivities, Reference Measurement Device and Spectral Calculation.

SMPTE ST 2065-3 Academy Density Exchange Encoding (ADX). Encoding Academy Printing Density (APD) Values.

The Academy of Motion Picture Arts and Sciences. Science and Technology Council. Academy Color Encoding System (ACES) Project Committee. Version 1.0 December 19, 2014. Annex A Spectral Responsivities.

See Also

quantity, calibrate

Examples

```
#   compute response of ACES scanner to the Hoya filters
product( Hoya, scanner.ACES, wave='auto' )

##                  R            G          B
## R-60   0.902447043 2.022522e-05 0.00000000
## G-533  0.038450857 4.900983e-01 0.05431134
## B-440  0.008466317 1.686241e-02 0.42863320
## LB-120 0.184408941 3.264111e-01 0.53492533
```

---

sectionOptimalColors *compute sections of an optimal color surface by hyperplanes*

Description

Consider a **colorSpec** object x with type equal to 'responsivity.material'. The set of all possible material reflectance functions (or transmittance functions) is convex, closed, and bounded, and this implies that the set of all possible output responses from x is also convex, closed, and bounded. The latter set is called the *object-color solid* or *Rösch Farbkörper* for x. If the dimension of the response of x is 2, this solid is a convex polygon that is centrally symmetric - a *zonogon*. If the dimension of the response of x is 3 (e.g. RGB or XYZ), this solid is a special type of centrally symmetric convex polyhedron called a *zonohedron*, see *Centore*. This function only supports dimensions 2 and 3. Denote this object-color solid by **Z**.

A color on the boundary of **Z** is called an *optimal color*. Let the equation of a hyperplane be given by:

$$< v, normal >= \beta$$

where $normal$ is orthogonal to the hyperplane, and β is the plane constant, and v is a variable. The purpose of the function sectionOptimalColors() is to compute the intersection of the hyperplane and ∂**Z**.

In dimension 3 this hyperplane is a 2D plane, and the intersection is generically a convex polygon. If the plane is a supporting plane, then the intersection is a face (a vertex, an edge, or a facet) of **Z**; in this case the function computes the center of the face.

In dimension 2 this hyperplane is a line, and the intersection is generically 2 points. If the line is supporing line, then the intersection is a vertex or an edge of **Z**; in the case of an edge the function computes the center of the edge.

Of course, the intersection can also be empty.

The function is essentially a wrapper around `zonohedra::section.zonohedron()` and `zonohedra::section.zonogon()`.

Usage

```
## S3 method for class 'colorSpec'
sectionOptimalColors( x, normal, beta )
```

Arguments

`x`	a **colorSpec** object with `type` equal to `'responsivity.material'` and M spectra, where M=2 or 3.
`normal`	a nonzero vector of dimension M, that is the normal to the family of parallel hyperplanes
`beta`	a vector of numbers of positive length. The number `beta[k]` defines the k'th plane `<v,normal>` = `beta[k]`.

.

Details

In the case that the dimension of `x` is 3, then **Z** is a zonohedron. For processing details see: `zonohedra::section.zonohedron()`.

In the case that the dimension of `x` is 2, then **Z** is a zonogon. For processing details see: `zonohedra::section.zonogon()`.

Value

The function returns a list with an item for each value in vector `beta`. Each item in the output is a list with these items:

`beta`	the value of the plane constant β
`section`	an NxM matrix, where N is the number of points in the section, and M is the dimension of `normal`. The points of the section are the rows of the matrix. If the intersection is empty, then N=0.

In case of global error, the function returns `NULL`.

References

Centore, Paul. *A Zonohedral Approach to Optimal Colours*. **Color Research & Application**. Vol. 38. No. 2. pp. 110-119. April 2013.

Logvinenko, A. D. An object-color space. **Journal of Vision**. 9(11):5, 1-23, (2009). `https://jov.arvojournals.org/article.aspx?articleid=2203976`. doi:10.1167/9.11.5.

See Also

vignette **Plotting Chromaticity Loci of Optimal Colors**, `probeOptimalColors()`, `zonohedra::section.zonohedron()`, `zonohedra::section.zonogon()`

Examples

```
wave = seq(420,680,by=5)
Flea2.scanner = product( A.1nm, "material", Flea2.RGB, wavelength=wave )
seclist = sectionOptimalColors( Flea2.scanner, normal=c(0,1,0), beta=10 )
nrow( seclist[[1]]$section )
##  [1] 89


seclist[[1]]$section[ 1:5, ]
##  the polygon has 89 vertices, and the first 5 are:
##           Red Green      Blue
##  233 109.2756    10 3.5391342
##  185 109.5729    10 2.5403628
##  136 109.8078    10 1.7020526
##  86  109.9942    10 1.0111585
##  35  110.1428    10 0.4513051
```

`sectionSchrodingerColors`

compute sections of a Schrodinger color surface by hyperplanes

---

Description

Consider a **colorSpec** object `x` with `type` equal to `'responsivity.material'`. The set of all possible material reflectance functions (or transmittance functions) that take the value 0 or 1, and with 2 or 0 transitions is called the *2-transition spectrum space*. When there are 2 transitions, there are 2 types of spectra: *bandpass* and *bandstop*. When there are 0 transitions, the spectrum is either identically 0 or identically 1. When `x` is applied to this space, the corresponding surface in response space is called the *2-transition surface*. The special points **0** and **W** (the response to the perfect reflecting diffuser) are on this surface. The surface is symmetrical about the neutral gray midpoint **G=W/2**. Following *West and Brill*, colors on the surface are called *Schrödinger colors*. Denote the surface by S_2. For details see: `zonohedra::raytrace2trans()`. This function only supports `x` with 3 channels.

Let the equation of a hyperplane be given by:

$$< v, normal >= \beta$$

where $normal$ is orthogonal to the hyperplane, and β is the plane constant, and v is a variable. The purpose of the function `sectionSchrodingerColors()` is to compute the intersection of the hyperplane and S_2. The intersection is generically a single polygon, which is not necessarily convex.

In pathological cases, the intersection can be many polygons, but this function only returns one of them, with a warning that there are more. Of course, the intersection can also be empty.

The function is essentially a wrapper around `zonohedra::section2trans()`.

Usage

```
## S3 method for class 'colorSpec'
sectionSchrodingerColors( x, normal, beta )
```

Arguments

x	a **colorSpec** object with type equal to 'responsivity.material' and 3 spectra.
normal	a nonzero vector of dimension 3, that is the normal to the family of parallel hyperplanes
beta	a vector of numbers of positive length. The number beta[k] defines the k'th plane <v,normal> = beta[k].

.

Value

The function returns a list with an item for each value in vector beta. Each item in the output is a list with these items:

beta	the value of the plane constant β
section	an Nx3 matrix, where N is the number of points in the section. The points of the section are the rows of the matrix. If the intersection is empty, then N=0.

In case of global error, the function returns NULL.

References

West, G. and M. H. Brill. Conditions under which Schrödinger object colors are optimal. **Journal of the Optical Society of America**. 73. pp. 1223-1225. 1983.

See Also

vignette **Plotting Chromaticity Loci of Optimal and Schrodinger Colors**, sectionOptimalColors(), zonohedra::section2trans()

Examples

```
wave = seq(420,680,by=5)
Flea2.scanner = product( A.1nm, "material", Flea2.RGB, wavelength=wave )
seclist = sectionSchrodingerColors( Flea2.scanner, normal=c(0,1,0), beta=10 )
nrow( seclist[[1]]$section )
##  [1] 106


seclist[[1]]$section[ 1:5, ]
##  the polygon has 106 vertices, and the first 5 are:
##              Red Green     Blue
##  [1,]  0.4054689    10 28.09329
##  [2,]  5.2833637    10 27.76564
```

```
##  [3,] 10.9896471    10 27.45328
##  [4,] 17.2285339    10 27.17679
##  [5,] 23.7280652    10 26.94359
```

`solar.irradiance` *Standard Solar Irradiance - Extraterrestrial and Terrestrial*

Description

`solar.irradiance` Three power spectra; from 280 to 1000 nm at 1 nm intervals. The unit is $W * m^{-2} * nm^{-1}$.

`atmosphere2003` a transmittance spectrum = the quotient of 2 spectra from `solar.irradiance`

Format

`solar.irradiance` is a **colorSpec** object with `quantity` equal to `'energy'` and with 3 spectra:

`AirMass.0` Extraterrestrial Radiation (solar spectrum at top of atmosphere) at mean Earth-Sun distance

`GlobalTilt` spectral radiation from solar disk plus sky diffuse and diffuse reflected from ground on south facing surface tilted 37 deg from horizontal

`AirMass.1.5` the sum of Direct and Circumsolar irradiance, when the optical path is 1.5 times that of the sun at zenith, see **Details**

`atmosphere2003` is a **colorSpec** object with `quantity` equal to `'transmittance'` and with 1 spectrum:

`AirMass.1.5` the quotient `AirMass.1.5` / `AirMass.0` from `solar.irradiance`

Details

Direct is Direct Normal Irradiance Nearly parallel (0.5 deg divergent cone) radiation on surface with surface normal tracking (pointing to) the sun, excluding scattered sky and reflected ground radiation.

Circumsolar is Spectral irradiance within +/- 2.5 degree (5 degree diameter) field of view centered on the 0.5 deg diameter solar disk, but excluding the radiation from the disk.

Note

The reference spectra in ASTM G173-03 are designed for Photovoltaic Performance Evaluation.

The original wavelength sequence in ASTM G173-03 is irregular. The interval is 0.5 nanometer from 280 to 400 nm, 1 nm from 400 to 1700 nm, an intermediate wavelength at 1702 nm, and 5 nm from 1705 to 4000 nm. To create the object `solar.irradiance` with a regular step size, the original was resampled from 280 to 1000 nm at 1nm intervals.

Source

Reference Solar Spectral Irradiance: ASTM G-173. http://rredc.nrel.gov/solar/spectra/am1.5/astmg173/astmg173.html

References

ASTM G173-03 Reference Spectra Derived from SMARTS v. 2.9.2.
Standard Tables for Reference Solar Spectral Irradiances: Direct Normal and Hemispherical on 37-deg Tilted Surface (2003)

See Also

D65, D50, daylightSpectra, resample, vignette **Blue Flame and Green Comet**

---

specnames *specnames of a colorSpec object*

Description

Retrieve or set the specnames of a **colorSpec** object. Retrieve the number of spectra.

Usage

```
## S3 method for class 'colorSpec'
specnames(x)

## S3 replacement method for class 'colorSpec'
specnames(x) <- value

## S3 method for class 'colorSpec'
numSpectra(x)
```

Arguments

x	a **colorSpec** R object
value	a character vector with length equal to the number of spectra in x.

Details

If the organization of x is "vector" then x is a vector and value is a single string, which is stored as attr(x,'specname').

If the organization of x is "matrix", then x is a matrix and value is stored as colnames(x).

If the organization of x is "df.col", then x is a data.frame with N+1 columns, where N is the number of spectra. value is stored as colnames(x)[2:(N+1)].

If the organization of x is "df.row", then x is a data.frame and value is stored as row.names(x).

Value

`specnames()` returns a character vector with the names of the spectra.

`numSpectra(x)` is equal to `length(specnames(x))` but much more efficient.

See Also

`rownames`, `colnames`

`standardRGB` *Convert from XYZ to some standard RGB spaces*

Description

To display an XYZ value, it typically must be converted to a standard RGB space. This is the function to do it.

Usage

```
RGBfromXYZ( XYZ, space )
```

Arguments

`XYZ`	a 3-vector, or a matrix with 3 columns with XYZs in the rows
`space`	the name of the RGB space - either `'sRGB'` or `'Adobe RGB'`. The match is case-insensitive, and spaces in the string are ignored.

Details

The input `XYZ` is multiplied by the appropriate 3x3 conversion matrix (for **sRGB** or **Adobe RGB**). These matrices are taken from *Lindbloom* and not from the corresponding *Wikipedia* articles; for the reason why see **Note**.

Value

An Mx3 matrix where M is the number of rows in `XYZ`, or M=1 if XYZ is a 3-vector. Each row of the returned matrix is filled with linear RGB in the appropriate RGB space. Values outside the unit cube are not clamped. To compute non-linear display RGB, see `DisplayRGBfromLinearRGB()`. In case of error the function returns `NULL`.

WARNING

This function is deprecated. New software should use `spacesRGB::RGBfromXYZ()` instead. The new function returns "signal RGB" instead of linear RGB.

Note

An RGB space is normally defined by the xy chromaticities of the 3 primaries and the white point. We follow *Lindbloom* in using the 'official' XYZ of the white point from ASTM E308. Using this XYZ of the white point makes the color space a little more consistent with other areas of color. For example, from IEC 61966-2-1 we have D65 xyY=(0.3127,0.3290,1) -> XYZ=(0.9504559,1,1.0890578). But from ASTM E308, D65 XYZ=(0.95047,1,1.08883), which is a little different.

Source

IEC 61966-2-1:1999. Multimedia systems and equipment - Colour measurement and management. Part 2-1: Colour management - Default RGB colour space - sRGB. https://webstore.iec.ch/publication/6169

Lindbloom, Bruce. RGB/XYZ Matrices. http://brucelindbloom.com/index.html?Eqn_RGB_XYZ_Matrix.html

Wikipedia. **sRGB**. https://en.wikipedia.org/wiki/SRGB

Wikipedia. **Adobe RGB**. https://en.wikipedia.org/wiki/Adobe_RGB_color_space

See Also

D65, officialXYZ, DisplayRGBfromLinearRGB

Examples

```
RGBfromXYZ( officialXYZ('D65'), 'sRGB' )
#      R G B
# [1,] 1 1 1    # not really 1s, but difference < 1.e-7

RGBfromXYZ( c(.3127,0.3290,0.3583)/0.3290, 'sRGB' )
#              R        G       B
# [1,] 0.9998409 1.000023 1.00024    difference > 1.e-5
```

`subset` *extract a subset of a colorSpec Object*

Description

extract a subset of the spectra in a **colorSpec** object.
The subset can be specified by indexes, by a logical vector, or by a regular expression matching the specnames

Usage

```
## S3 method for class 'colorSpec'
subset( x, subset, ... )
```

Arguments

x	a **colorSpec** object
subset	an integer vector, a logical vector, or a regular expression
...	additional arguments ignored

Details

If subset is an integer vector, each integer must be between 1 and M, where M the number of spectra in x. No duplicates are allowed. The number of spectra returned is equal to length(subset). It is OK for the length to be 0, in which case the function returns the empty subset.

If subset is a logical vector, its length must be equal to M. The number of spectra returned is equal to the number of TRUEs in subset.

If subset is a regular expression, the number of spectra returned is equal to the number of specnames(x) matched by the expression.

Value

subset(x) returns a **colorSpec** object with the same organization as x. Exception: if the organization of x is 'vector' and the subset is empty, then the returned object is a matrix with 0 columns.

Note

subset() can also be used for re-ordering the spectra; just set argument subset to the desired permutation vector.

See Also

organization

Examples

```
tritanope = subset( lms2000.1nm, 1:2 )  # keep long and medium cone fundamentals, but drop the short

sml2000.1nm = subset( lms2000.1nm, 3:1 ) #   reorder from short to long
```

---

theoreticalRGB *Theoretical RGB Cameras - BT.709.RGB, Adobe.RGB, and ACES.RGB*

Description

These are 3 built-in **colorSpec** objects, with `quantity` equal to `'energy->electrical'`.

`BT.709.RGB` a theoretical RGB responder to light. The 3 responsivity spectra are constructed so that the RGBs from this theoretical camera, when displayed on an sRGB display after correct EOTF adjustment, would emit light with the same XYZs as the captured scene (up to a constant multiple). All three responsivities have negative lobes.

`Adobe.RGB` a theoretical RGB responder to light. The 3 responsivity spectra are constructed so that the RGBs from this theoretical camera, when displayed on an Adobe RGB display after correct EOTF adjustment, would emit light with the same XYZs as the captured scene (up to a constant multiple). All three responsivities have negative lobes.

`ACES.RGB` a theoretical RGB responder to light. Unlike the two above cameras, the responsivities are non-negative and so this camera could be built, in principle. These are the ACES RICD (Reference Input Capture Device) spectral sensitivities.

Format

All are **colorSpec** objects with `quantity` equal to `'energy->electrical'` and 3 spectra: `r`, `g`, and `b`. The wavelengths are 360 to 830 nm at 1 nm intervals.

Details

All responsitivity spectra are linear combinations of the spectra in `xyz1931.1nm`. These 3 theoretical cameras satisfy the *Maxwell-Ives criterion* by construction.
For `BT.709.RGB` and `Adobe.RGB`, the responsivities are scaled so the response to `D65.1nm` is RGB=(1,1,1). These responsivities have negative lobes.
The BT.709 primaries and white point are the same as those of sRGB (though the EOTF functions are different).
Adobe RGB and sRGB share the same Red, Blue, and White chromaticities, and only differ in the Green. This implies that for both cameras the Green output is 0 at Red and Blue, and 1 at White. This in turn implies that the Green output is identical for both cameras for all input spectra, and so the Green responsivity spectra are identical for both cameras.
For `ACES.RGB` the responsivities are area normalized as in Annex C of S-2008-001. They are scaled so that the response to Illuminant E is RGB=(1,1,1). For an example of white-balancing, as in Annex B, see the examples below.

References

Poynton, Charles. **Digital Video and HD - Algorithms and Interfaces**. Morgan Kaufmann. Second Edition. 2012. Figure 26.5 on page 302.

Academy Color Encoding Specification (ACES). S-2008-001. 2011. Annex B, pp. 23-25. Annex C, pp. 26-33.

See Also

`quantity()`, `D65.1nm`, `xyz1931.1nm`, `ptransform()`, `calibrate()`, vignette **Blue Flame and Green Comet**

Examples

```
#######    BT.709.RGB is created using the following recipe  ########
P = matrix( c(0.64,0.33,NA,  0.3,0.6,NA, 0.15,0.06,NA ), 3, 3, byrow=TRUE )
rownames(P) = c('R','G','B')
BT.709.RGB  = ptransform( xyz1931.1nm, P, D65.1nm )
quantity(BT.709.RGB) = "energy->electrical"

#######    Adobe.RGB recipe is the same, except for the matrix P  ########
P = matrix( c(0.64,0.33,NA,  0.21,0.71,NA, 0.15,0.06,NA ), 3, 3, byrow=TRUE )
rownames(P) = c('R','G','B')
Adobe.RGB  = ptransform( xyz1931.1nm, P, D65.1nm )
quantity(Adobe.RGB) = "energy->electrical"

#######  white-balancing ACES.RGB for CIE Standard Illuminant D60 ########
# in a scene illuminated by daylight illuminant D6003,
# and with a perfect-reflecting-diffuser in that scene,
# object 'camera1' would have response RGB=(1,1,1) for that diffuser.
D6003 = daylightSpectra( 6000*1.4388/1.4380, wavelength=wavelength(ACES.RGB) )
camera1 = calibrate( ACES.RGB, D6003, 1, method='scaling' )
```

`wavelength` *wavelength vector of a colorSpec object*

Description

Retrieve or set the wavelengths of a **colorSpec** object. Retrieve the number of wavelengths, and whether the wavelength sequence is regular.

Usage

```
## S3 method for class 'colorSpec'
wavelength(x)

## S3 replacement method for class 'colorSpec'
wavelength(x) <- value

## S3 method for class 'colorSpec'
numWavelengths(x)

## S3 method for class 'colorSpec'
is.regular(x)
step.wl(x)
```

Arguments

`x`	a **colorSpec** R object
`value`	a numeric vector with length equal to the number of wavelengths in `x`. The wavelengths must be increasing. The unit must be nanometers.

Details

If the organization of x is 'df.col', then x is a data.frame and the wavelength vector is stored in the first column of x.
Otherwise, the wavelength vector is stored as attr(x,'wavelength').

Value

wavelength() returns a numeric vector with the wavelengths of the spectra, in nanometers.

numWavelengths(x) is equal to length(wavelength(x)) but much more efficient.

is.regular() returns TRUE or FALSE, depending on whether the step between consecutive wavelengths is a constant. A truncation error of 1.e-6 nm is tolerated here. For example, the X-Rite ColorMunki spectrometer in hi-res mode has a step of 3.33333nm, and it is considered regular.

step.wl() returns the *mean* step in nm, whether the wavelengths are regular or not.

See Also

colorSpec

---

xyz1931 *CIE Color Matching Functions - 2-degree (1931)*

Description

xyz1931.1nm	the 1931 2° functions from 360 to 830 nm, at 1nm intervals
xyz1931.5nm	the 1931 2° functions from 380 to 780 nm, at 5nm intervals

Format

Each is a **colorSpec** object organized as a matrix with 3 variables.

x	the x-bar responsivity function
y	the y-bar responsivity function
z	the z-bar responsivity function

Source

http://www.cvrl.org

References

Günther Wyszecki and W.S. Stiles. **Color Science : Concepts and Methods, Quantitative Data and Formulae**. Second Edition. Wiley-Interscience. 1982. Table I(3.3.1). pp. 723-735.

ASTM E 308 - 01. Standard Practice for Computing the Colors of Objects by Using the CIE System. Table 1

See Also

xyz1964

Examples

```
summary(xyz1931.1nm)
white.point = product( D65.1nm, xyz1931.1nm, wave='auto' )
```

---

xyz1964 *CIE Color Matching Functions - 10-degree (1964)*

Description

xyz1964.1nm	the 10° 1964 functions from 360 to 830 nm, at 1nm intervals
xyz1964.5nm	the 10° 1964 functions from 380 to 780 nm, at 5nm intervals

Format

Each is a **colorSpec** object organized as a matrix with 3 columns.

x	the x-bar responsivity function
y	the y-bar responsivity function
z	the z-bar responsivity function

Source

http://www.cvrl.org

References

Günther Wyszecki and W.S. Stiles. **Color Science : Concepts and Methods, Quantitative Data and Formulae**. Second Edition. Wiley-Interscience. 1982. Table I(3.3.1). pp. 723-735.

ASTM E 308 - 01. Standard Practice for Computing the Colors of Objects by Using the CIE System. Table 1

See Also

xyz1931

Examples

```
summary(xyz1964.1nm)
white.point = product( D65.1nm, xyz1964.1nm, wave='auto' )
```

Index