JMO Spectrum Library Documentation

Version 1.23, July 4th, 2020

Julius Muschaweck, JMO GmbH, Zugspitzstr. 66, 82131 Gauting, Germany. julius@jmoptics.de

Rationale

We deal routinely with spectra in illumination optics. We need to analyze LED spectra, compute color coordinates and color rendering values from spectra, integrate them, add spectra, multiply spectra with scalar weights and with other spectra (like transmission spectra), interpolate a given measured spectrum with non-equidistant wavelength values to a regular 1 nm array, and so on.

In practice, this is tedious: spectra come in various formats, and the problem of dealing with two spectra als tabulated values which have two different sets of wavelengths is annoying. In addition, many colorimetric calculations, from computing simple x/y coordinates all the way to the color rendering index, are not easily accessable.

This open source Matlab library is designed to make these engineering tasks easy and transparent. It is compatible with GNU Octave, and with Matlab for Mac and Linux.

License -- public domain

Version history

Version 1.24, July 4, 2020

Fixed bug in AddSpectra, where two spectra with different additional fields could not be added.

Version 1.23, Jan 25, 2020

Fixed a bug in CRI (van Kries transformation was first computed but then ignored.

Added IntegrateSpectrum, to integrate a spectrum, with an optional weighting spectrum

Version 1.22, Jan 24, 2020

Added ComputeSpectrumColorimetry, convenience function to compute XYZ, CCT and color rendering values in one call

Added Vlambda, returning V(lambda), CIE 1924 as a spectrum

Added PlotCIExyBorder, plotting the CIE x/y "color horseshoe", with optional ticks and other options

MakeSpectrum now accepts optional Name/Value pairs to set fields like e.g. 'name'

CCT and CCT_from_xy have additional return values `ok` and `errmsg`, to suppress errors and warnings if desired

Version 1.21, Jan 24, 2020

Added CCT, convenience function to compute CCT from spectrum directly, not via x/y

Added Example_WhiteLED.m as an example script composing a white LED from blue and yellow, with supporting spectra blue.sre and yellow.sre

Added LDomPurity, to compute dominant wavelength and purity of a spectrum

Added ReadLightToolsSpectrumFile, to read LightTools .sre files, as supplied by many LED vendors. Reads also simple ASCII 2-column tables with any text header

Version 1.2, Jan. 22, 2020

Added CIE standard illuminants

Added CRI (color rendering index) computations

Added AssignNewWavelength function

Version 1.01, Oct. 13:

Added WriteLightToolsSpectrumFile

Version 1.0, Sept. 9, 2019: Initial version

Getting started

To define a spectrum and compute its CIE xy color coordinates, simply say

```
clear s;
s.lam = [360 830];
s.val = [1 1];
% this is a flat spectrum, known as CIE standard illuminant E.
% By definition, it should have color coordinates x == y == 1/3-
s.XYZ = CIE1931_XYZ(s);
% This is a common pattern: Compute information from a spectrum, and then
% add this information to the same spectrum as an additional field
s
```

```
s = struct with fields:
    lam: [360 830]
    val: [1 1]
    XYZ: [1×1 struct]
```

```
s.XYZ
```

```
ans = struct with fields:
    X: 106.8654
    Y: 106.8569
    Z: 106.8919
    x: 0.3333
    y: 0.3333
    z: 0.3334
```

Design decisions

Spectrum

In this library, a *spectrum* is a struct with at least two fields named `lam` and `val`, which meet the following requirements:

1. `lam` is a 1D vector of numeric values which are not complex, > 0, and strictly ascending
2. `val` is a 1D vector of numeric values which are not complex.
3. `lam` and `val` are of same length and have at least two elements

It is desirable for `lam` and `val` to be column vectors.

There is a function `rv = SpectrumSanityCheck(rhs)`, which tests all these requirements and, if met, returns the same struct except that `lam` and `val` are converted to column vectors if necessary.

A *spectrum* models the function *S(lambda)* which represents a physical scalar function of wavelength. Like spectral radiant flux, spectral irradiance, spectral radiant intensity, spectral radiance, spectral transmission, spectral absorption, spectral efficiency. The tabulated values in `val` are linearly interpolated. Outside the range given by `lam`, *S(lambda)* == 0. Accordingly, there are no spectra consisting of truly monochromatic lines in this library. If you want to approximate line spectra, make very narrow triangles. Spectra are continuous, except at the wavelength boundaries, where the jump to zero

`lam` is considered to have units of nanometers in all library functions that make use of this unit, e.g. color calculations.

Why structs and not classes? Classes are nice to guarantee that properties like `lam` and `val` always are present, and would allow methods which operate directly on spectra. However, structs are simpler and more versatile. I, as the library designer, cannot know which additional information a user (I myself, for example), wants to attach to a given spectrum. Name, date, name of LED, color coordinates and more. To make a spectrum s, I can simply say

```
clear s
nfig = 0; %figure number to be used in examples
s.lam = [360 830];
s.val = [1 1];
s.name = "CIE standard illuminant E";
s.hopp = "topp";
s
```

```
s = struct with fields:
     lam: [360 830]
     val: [1 1]
    name: "CIE standard illuminant E"
    hopp: "topp"
```

to give it an appropriate name and an arbitrary value in an arbitrary additional field. Of course, a user can always create a subclass to add properties to a given class, but it is just so much simpler to assign to an additional struct field in a single line. In my experience, these conventions for spectra are simple and few enough to be easily remembered and adhered to.

Conventions

In my code, I like to have long unabbreviated variable and function names. Except for a few standards:

`rv` is my name for the return value of a function

`spec` is my name for a single spectrum argument of a function.

`lhs` and `rhs` are short for 'left hand side' and 'right hand side', the arguments of a binary function.

Proper library function names start with capital letters, e.g. `MakeSpectrum`. Internal helper functions start with a small "i", e.g. `iLinInterpolProto`.

A spectrum struct is valid if it has fields lam and `val` which fulfill the requirements above. In addition, I assume a spectrum may have the following fields:

`name`: A short character string with a name.

`description`: A longer character string with a description.

`XYZ`: A struct with tristimulus and color coordinate fields `X, Y, Z, x, y, z`, typically created by code like `s.XYZ = CIE1931_XYZ(s)`

Function reference

Alphabetic list

AddSpectra adds two spectra

AddWeightedSpectra adds multiple spectra with weights

AssignNewWavelength replaces the wavelength of a spectrum, interpolating the values correctly

CCT, convenience function to compute CCT from spectrum directly, not via x/y

CCT_from_xy computes correlated color temperature and uv-distance to Planck locus from spectrum

CIE1931_lam_x_y_z.mat contains CIE 1931 data for color matching functions, monochromatic border and Planck locus

CIE1931_XYZ computes CIE 1931 XYZ color coordinates and tristimulus values from spectrum

CIE_Illuminant returns a large selection of CIE standard illuminants

CIE_Illuminant_D computes the CIE standard daylight spectrum as function of color temperature

CODATA2018 returns a struct with CODATA 2018 values for relevant physical constants (speed of light, Boltzmann constant etc.)

ComputeSpectrumColorimetry, convenience function to compute XYZ, CCT and color rendering values in one call

CRI computes Color Rendering Index values according to CIE 13.3-1995

Example_WhiteLED.m as an example script composing a white LED from blue and yellow, with supporting spectra blue.sre and yellow.sre

GaussSpectrum creates a Gaussian spectrum for given mean and standard deviation

IntegrateSpectrum, integrate a spectrum, with an optional weighting spectrum

IsOctave determines if running on GNU Octave or Matlab

LDomPurity, to compute dominant wavelength and purity of a spectrum

LinInterpol computes linear interpolation like `interp1,` but about five times faster on Matlab on Windows

LinInterpolAdd4Async computes linear interpolation of the sum of four functions

MakeSpectrum creates a valid spectrum from wavelength and value data

MultiplySpectra multiplies two spectra, e.g. an LED spectrum with a transmission spectrum

PlanckLocus returns a wealth of information about the Planck locus, including interpolation function objects and Judd lines of equal CCT

PlanckSpectrum creates a blackbody spectrum with various normalizations to choose from

PlotCIExyBorder, plotting the CIE x/y "color horseshoe", with optional ticks and other options

ReadLightToolsSpectrumFile, to read LightTools .sre files, as supplied by many LED vendors. Reads also simple ASCII 2-column tables with any text header

SpectrumSanityCheck checks if a spectrum fulfills the requirements

TestLinInterpol tests the LinInterpol function

Vlambda, returning V(lambda), CIE 1924 as a spectrumCRI is a class for computing color rendering index values

WriteLightToolsSpectrumFile writes a spectrum to an ASCII file in LightTools® format.

`AddSpectra`

Adds two spectra without weights. Convenience function with simpler interface than AddWeightedSpectra

```
% function rv = AddSpectra(lhs, rhs)
```

Input: `lhs, rhs`*:* Both must be valid spectra.

Output: `rv`*:* Sum of `rhs + lhs`. When both spectra do not overlap, the wavelength ranges are concatenated, and the range in between is padded with zero. If they do overlap, then `rv.lam` contains all values from both input spectra, with duplicate values removed, and what is added are the linearly interpolated values from both input spectra. Thus, the sum spectrum is a perfect model of the underlying continuous function which is the sum of the continuous, linearly interpolated input spectra.

Additional fields present in `lhs` or `rhs` will be stripped, `rv` will have only fields `lam` and `val.`

Usage Example:

```
clear s1 s2 sumspec
s1 = MakeSpectrum([400,500,600],[1, 2, 4]);
s2 = MakeSpectrum([400,560,610],[4, 4, 1]);
sumspec = AddSpectra(s1,s2);
nfig = nfig + 1;
figure(nfig);
clf;
hold on;
```

```
plot(s1.lam, s1.val,'Marker','x');
plot(s2.lam, s2.val,'Marker','x');
plot(sumspec.lam, sumspec.val,'Marker','x');
legend({'s1','s2','sumspec'},'Location','NorthWest');
axis([390 610 0 8]);
grid on;
xlabel('lam');
ylabel('val');
title('AddSpectra demo');
```

AddWeightedSpectra

Computes weighted sum of several spectra.

```
% function rv = AddWeightedSpectra(spectra,weights)
```

Input:

`spectra`: Nonempty *c*ell array of spectra

`weights`*:* Vector of numeric non-complex values, same length as `spectra`.

Output: `rv`*:* Weighted sum of input spectra. When spectra do not overlap, the wavelength ranges are concatenated, and the range in between is padded with zero. If they do overlap, then `rv.lam` contains all values from all input spectra, with duplicate values removed, and sorted, and what is added are the linearly interpolated values from all input spectra. Thus, the sum spectrum is a perfect model of the function which is the sum of the continuous, linearly interpolated input spectra.

Additional fields present in `spectra` will be stripped, `rv` will have only fields `lam` and `val`.

Usage Example:

Add a red, a green, and a blue spectrum created by GaussSpectrum

```
clear red green blue sumspec
red = GaussSpectrum(linspace(550,700),620,15);
green = GaussSpectrum(linspace(430,730),530,20);
blue = GaussSpectrum(linspace(400,500),450,8);
sumspec = AddWeightedSpectra({red, green, blue},[2, 5, 1.5]);
nfig = nfig + 1;
figure(nfig);
clf;
hold on;
plot(red.lam, red.val,'r');
plot(green.lam, green.val,'g');
plot(blue.lam, blue.val,'b');
plot(sumspec.lam, sumspec.val,'k');
legend({'red','green','blue','sumspec'},'Location','NorthWest');
grid on;
xlabel('lam');
ylabel('val');
title('AddWeightedSpectra demo');
```

AssignNewWavelength

Assign a new wavelength vector to a spectrum, interpolating the old values.

```
% function rv = AssignNewWavelength(spectrum, lam_new)
```

Input:

`spectrum:` a spectrum

`lam_new:` the new wavelength array, ascending doubles.

Output:

`rv:` A copy of the old spectrum with all fields, but `lam` is replaced and `val` is interpolated. For `lam_new` values outside the old interval, `val(i) == 0`

Usage Example:

```
s_old = MakeSpectrum([400 500], [0 100]);
s_new = AssignNewWavelength(s_old, [450 451 452])
```

```
s_new = struct with fields:
    lam: [450 451 452]
    val: [50 51 52]
```

CCT

convenience function to compute CCT from spectrum directly, not via x/y

```
% function [iCCT, dist_uv, ok, errmsg] = CCT(spectrum)
```

Input:

`spectrum:` a spectrum with fields lam and val

Output:

`iCCT:` Correlated color temperature in Kelvin

`dist_uv:` Distance to Planck locus in u/v color space. When `dist_uv>0.05,` a warning is issued, when `dist_uv` > 0.09, an error. Positive when x/y is above Planck locus (on the "green" side), negative when below.

`ok:` `true` if CCT can be properly computed, i.e. `dist_uv <= 0.9` and CCT within 1000..Inf range. `false` if something went really wrong

`errmsg:` empty `''` if everything is ok, `'error: ...'` if `ok==false,` `'warning: ...'` e.g. if `dist_uv > 0.05'`, which makes the CCT computation technically illegal.

If `ok` is not queried as a return value, warnings and errors are raised. If `ok` is queried, they are suppressed.

Usage Example:

```
test = CIE_Illuminant('D65'); % 6500K CIE daylight spectrum, slightly above Planck
```

```
[iCCT, dist_uv] = CCT(test) %#ok<ASGLU> % not exactly 6500, depends on step size of XYZ integra
```

```
iCCT = 6.5026e+03
dist_uv = 0.0032
```

CCT_from_xy

Compute correlated color temperature of an xy color point. Uses parabolic interpolation between nearest Planck locus points

```
% function [CCT, dist_uv, ok, errmsg] = CCT_from_xy(x,y)
```

Input: `x, y` scalar numbers, x/y color coordinates

Output:

`CCT:` Correlated color temperature in Kelvin

`dist_uv:` Distance to Planck locus in u/v color space. When `dist_uv>0.05,` a warning is issued, when `dist_uv` > 0.09, an error. Positive when x/y is above Planck locus (on the "green" side), negative when below.

`ok: true` if CCT can be properly computed, i.e. `dist_uv <= 0.9` and CCT within 1000..Inf range. `false` if something went really wrong

`errmsg:` empty `''` if everything is ok, `'error: ...'` if `ok==false, 'warning: ...'` e.g. if `dist_uv > 0.05'`, which makes the CCT computation technically illegal.

If ok is not queried as a return value, warnings and errors are raised. If ok is queried, they are suppressed.

Usage Example:

```
clear T pl jl uv den x y CCT dist_uv
T = 3456;
pl = PlanckLocus;
jl = pl.JuddLine_func(T);
% compute uv coordinates for color point with CCT == T,
% 0.04 away from Planck
uv = [jl.u, jl.v] + 0.04 * [jl.du, jl.dv];
% compute xy coordinates by standard transformation
den = 2*uv(1) - 8*uv(2) + 4;
x = 3*uv(1)/den;
y = 2 * uv(2) / den;
% compute CCT
[iCCT, dist_uv] = CCT_from_xy(x,y);
iCCT - T % should be zero, is about 1.3 mK
```

```
ans = 0.0013
```

```
dist_uv - 0.04 % should be zero
```

```
ans = -4.3846e-07
```

CIE1931_lam_x_y_z.mat

A .mat file which contains a struct named `'CIE1931XYZ'` with CIE 1931 data: x/y/z color matching functions, monochromatic border x/y coordinates with corresponding wavelengths, and coordinates of the Planck locus with corresponding absolute temperatures.

Usage Example:

```
clear CIE1931XYZ;
load('CIE1931_lam_x_y_z.mat','CIE1931XYZ')
CIE1931XYZ
```

```
CIE1931XYZ = struct with fields:
        lam: [471×1 double]
          x: [471×1 double]
          y: [471×1 double]
          z: [471×1 double]
    xBorder: [471×1 double]
    yBorder: [471×1 double]
    PlanckT: [1001×1 double]
    Planckx: [1001×1 double]
    Plancky: [1001×1 double]
```

CIE1931_XYZ

Computes CIE 1931 color coordinates.

```
% function rv = CIE1931_XYZ(spec)
```

Input: `spec` is a *spectrum* struct, see above for requirements.

Output: A struct with fields `X Y Z x y z`.

Capital `X Y Z` are the CIE tristimulus values, i.e. the result of integrating `spec` with the CIE 1931 standard x y z color matching functions.

`x y z` are the corresponding color coordinates, `x = X / (X + Y + Z)` etc.

Usage example:

```
clear s;
s.lam = [360 830];
s.val = [1 1];
% this is a flat spectrum, known as CIE standard illuminant E.
% By definition, it should have color coordinates x == y == 1/3-
s.XYZ = CIE1931_XYZ(s);
% This is a common pattern: Compute information from a spectrum, and then
% add this information to the same spectrum as an additional field
s
```

```
s = struct with fields:
    lam: [360 830]
    val: [1 1]
    XYZ: [1×1 struct]
```

```
s.XYZ
```

```
ans = struct with fields:
    X: 106.8654
```

```
    Y: 106.8569
    Z: 106.8919
    x: 0.3333
    y: 0.3333
    z: 0.3334
```

CIE_Illuminant

Returns CIE standard illuminants.

Available spectra are:
'A';'D65';'C';'E','D50';'D55';'D75';'FL1';'FL2';'FL3';'FL4';'FL5';'FL6';'FL7';'FL8';'FL9';'FL10';'FL11';'FL12';

'FL3_1';'FL3_2';'FL3_3';'FL3_4';'FL3_5';'FL3_6';'FL3_7';'FL3_8';'FL3_9';'FL3_10';'FL3_11';'FL3_12';
'FL3_13';'FL3_14';'FL3_15'; 'HP1';'HP2';'HP3';'HP4';'HP5'

```
% function rv = CIE_Illuminant(name,varargin)
```

Input: name is the desired name, one of the available names listed above

varargin: Name/Value pair 'lam',lam to define the desired wavelength table to which the illuminant spectrum will be interpolated. Default is 360:830 nm in 1 nm steps

Output: spectrum struct with fields lam (copy of input, or default 360:830), val and name (copy of input name)

Usage Example:

```
FL4 = CIE_Illuminant('FL4')
```

```
FL4 = struct with fields:
     lam: [1×471 double]
     val: [1×471 double]
    name: 'CIE standard illuminant FL4'
```

```
clf;plot(FL4.lam,FL4.val);
title('CIE standard FL4 fluorescent spectrum')
```

CIE_Illuminant_D

Returns CIE standard daylight illuminant D as function of CCT.

```
% function rv = CIE_Illuminant_D(CCT,varargin)
```

Input: CCT is the desired color temperature. Must be in [4000, 25000].

varargin: Name/Value pair 'lam',lam to define the desired wavelength table to which the illuminant spectrum will be interpolated. Default is 360:830 nm in 1 nm steps

Output: spectrum struct with fields lam (copy of input, or default 360:830), val and name (copy of input name)

Usage Example:

```
D6100 = CIE_Illuminant_D(6100,'lam',300:5:830);
clf;plot(D6100.lam,D6100.val);
title('CIE standard D (daylight) spectrum for CCT = 6100');
```

CODATA2018

A struct with the most relevant physical constants for optics, as defined by CODATA

```
% function cd = CODATA2018()
```

Input: None

Output: Struct with fields:

`b`: Wien's wavelength displacement law constant

`bprime`: Wien frequency displacement law constant

`c`: speed of light in vacuum

`e`: elementary charge

`h`: Planck constant

`k`: Boltzmann constant

`me`: electron mass

`mn`: neutron mass

`mp`: proton mass

`NA`: Avogadro constant

`R`: molar gas constant

`Vm`: molar volume of ideal gas

`sigma`: Stefan-Boltzmann constant

`c1`: first radiation constant (2 pi h c^2)

`c1L`: first radiation constant for spectral radiance (2 h c^2)

`c2`: second radiation constant (h c / k)

Each field is a struct with fields `name, value, reluncertainty, absuncertainty, unit.` Most uncertainties are zero, since 2018.

Usage Example:

```
clear cd lambda freq pe c
cd = CODATA2018();
lambda = 500e-9; % in meters
c = cd.c.value % speed of light
```

```
c = 299792458
```

```
freq = c/lambda% frequency of 500 nm light about 600 THz
```

```
freq = 5.9958e+14
```

```
pe = freq * cd.h.value % energy of a 500 nm photon, in Joule
```

```
pe = 3.9729e-19
```

ComputeSpectrumColorimetry

convenience function to compute XYZ, CCT and color rendering values in one call

```
% function rv = ComputeSpectrumColorimetry(s, varargin)
```

Takes the spectrum struct and adds fields with computed colorimetric values:

XYZ: struct with fields x, y, z, X, Y, Z as returned from CIE1931_XYZ(s)

CCT, dist_uv_Planck: color temperature and distance to Planck locus (may be NaN if out of CCT range)

CRI_all: struct with all 16 Ri and Ra as returned from CRI.FullCRI(s) (may be NaN if out of CCT range)

Ra: number, general CRI value

Also performs optional normalization to peak = 1

Input:

`s:` spectrum, struct with fields `lam` and `val.`

`varargin:` Name/Value pair: 'Normalize', 'peak' | 'off' (default)

Normalize -> 'peak' scales spectrum .val to have peak = 1

Normalize -> 'off' leaves .val unchanged

Output:

`rv:` the spectrum, with added fields and possibly scaled `.val` field. Other fields in `s`, like e.g. `.name,` remain unchanged

Usage Example:

```
test = MakeSpectrum([400 700],[1 1],'hopp',42); % hopp is just some field to show it remains
ComputeSpectrumColorimetry(test)
```

```
ans = struct with fields:
               lam: [2×1 double]
               val: [2×1 double]
              hopp: 42
               XYZ: [1×1 struct]
               CCT: 5.4542e+03
    dist_uv_Planck: -0.0041
           CRI_all: [1×1 struct]
                Ra: 95.5554
              Ldom: 556.7580
            purity: 0.0022
```

CRI

A class to compute color rendering indices. See https://en.wikipedia.org/wiki/Color_rendering_index, and CIE 13.3-1995 Technical Report

```
% classdef CRI < handle
```

Constructor:

```
% function this = CRI()
cri = CRI()
```

```
cri =
  CRI with properties:

    CRISpectra_: [1×16 struct]
    strict_5nm_: 0
```

loads the CRI reflectivity spectra into the read-only variable CRISpectra_

SetStrict_5nm

```
% function prev = SetStrict_5nm(this, yesno)
prev = cri.SetStrict_5nm(true)
```

```
prev = logical
   0
```

```
% do something
cri.SetStrict_5nm(prev); % restore previous state
```

sets the internal wavelength interval to 5 nm or 1 nm

Input: yesno: logical (true -> 5nm, false -> 1nm)

Output: prev: logical, the value before the call

SingleRi

computes a special Ri value

```
% function rv = SingleRi(this, spectrum, i)
FL4 = CIE_Illuminant('FL4');
D65 = CIE_Illuminant('D65');
cri.SingleRi(FL4,9) % Really really bad R9, fluorescent and saturated red don't work well
```

```
ans = -110.6599
```

```
cri.SingleRi(D65,9) % By definition, CIE D should work perfectly for CCT > 5000
```

```
ans = 99.9894
```

Ra

computes the general Ra, the by far most used value

```
% function [rv, Ri_1_8] = Ra(this, spectrum)
cri.Ra(FL4) % CRI is designed to give 51
```

```
ans = 51.5163
```

```
cri.Ra(D65) % by definition, 100
```

```
ans = 99.9979
```

Input: a spectrum

Output: rv: double, the Ra value.

Ri_1_8: array of eight doubles, the individual Ri values of which Ra is the mean

FullCRI

computes the CRI for all sixteen reflectivity spectra. R1..R14 are defined in the standard, R15 is Asian skin, and R16 (my personal addition) is perfect white

```
% function rv = FullCRI(this, spectrum)
cri.FullCRI(FL4)
```

```
ans = struct with fields:
    Ri: [16×1 double]
    Ra: 51.5163
```

Input: a spectrum

Output: rv: a struct with fields Ri (array of 16 doubles, the individual Ri values), and Ra (the general index)

PlotCRISpectra

plots the 16 reflectivity spectra

```
% function PlotCRISpectra(this)
cri.PlotCRISpectra();
```

CRISpectra_

a read-only property. An array of 16 structs with fields describing the individual spectra:

```
cri.CRISpectra_
```

```
ans = 1×16 struct
```

Fields	lam	val	name	description	munsell
1	1×471 double	1×471 double	'TCS01'	'Light grey...	'7.5 R 6/4'
2	1×471 double	1×471 double	'TCS02'	'Dark greyi...	'5 Y6/4'
3	1×471 double	1×471 double	'TCS03'	'Strong yel...	'5 GY 6/8'
4	1×471 double	1×471 double	'TCS04'	'Moderate y...	'2.5 G 6/6'
5	1×471 double	1×471 double	'TCS05'	'Light blui...	'10 BG 6/4'

Fields	lam	val	name	description	munsell
6	1×471 double	1×471 double	'TCS06'	'Light blue'	'5 PB 6/8'
7	1×471 double	1×471 double	'TCS07'	'Light violet'	'2.5 P 6/8'
8	1×471 double	1×471 double	'TCS08'	'Light redd...	'10 P 6/8'
9	1×471 double	1×471 double	'TCS09'	'Strong red'	'4.5 R 4/13'
10	1×471 double	1×471 double	'TCS10'	'Strong yellow'	'5 Y 8/10'
11	1×471 double	1×471 double	'TCS11'	'Strong green'	'4.5 G 5/8'
12	1×471 double	1×471 double	'TCS12'	'Strong blue'	'3 PB 3/11'
13	1×471 double	1×471 double	'TCS13'	'Light yell...	'5 YR 8/4'
14	1×471 double	1×471 double	'TCS14'	'Moderate o...	'5 GY 4/4'

⋮

Example_WhiteLED.m

example script composing a white LED from blue and yellow, with supporting spectra blue.sre and yellow.sre. Run from Matlab environment and look at variables and figures.

```
% ExampleWhiteLED()
```

GaussSpectrum

Creates a normalized Gaussian spectrum with given mean and standard deviation

```
% function rv = GaussSpectrum(lam_vec,mean,sdev,varargin)
```

Input:

`lam_vec:` A vector of positive reals, strictly ascending

`mean:` Scalar positive number. Mean value of the distribution. May or may not be in the `lam_vec` range.

`sdev:` Scalar positive number. Standard deviation

`varargin:` Optional string argument '`val_only`'.

Output:

`rv:` Spectrum struct, with additional `name` field. Except if optional argument '`val_only`' is present: Then, rv is a column vector of the values.

Usage Example:

see also example for `AddWeightedSpectra`.

```
GaussSpectrum(400:500,450,10)
```

```
ans = struct with fields:
     val: [101×1 double]
     lam: [101×1 double]
```

```
    name: 'Gauss spectrum for mean 450 nm and sdev 10 nm, normalized to peak = 1'
```

IntegrateSpectrum

Compute weighted integral of spectrum. If weight omitted, compute just the integral.

```
% function rv = IntegrateSpectrum(spectrum, weight)
```

Spectra in this library are always continuous, and assumed to be linear between the data points. First, spectrum and weight are multiplied. The result will have the wavelengths interweaved, and the multiplication result will be the product of the linearly interpolated individual values. The multiplication result is assumed to be linear between these interweaved points, just like every spectrum in this library. Accordingly, we integrate the multiplication result by simple trapezoidal rule.

This means, however, that even for simple linear spectra and weights, the integral depends on the wavelength resolution.

Input:

`spectrum:` A valid spectrum

`weight:` Optional argument, also a valid spectrum

Output:

`rv:` The integral ov `.val` over `.lam` of the multiplied spectrum (see `MultiplySpectra` for more info about details).If `eight` is omitted, just integrate the spectrum as-is.

Usage Example:

```
sunlight = PlanckSpectrum(200:10*1000, 5800); % approximate extraterrestrial solar spectrum, fr
visible = MakeSpectrum([360 780],[1 1]);
P_full = IntegrateSpectrum(sunlight);
P_visible = IntegrateSpectrum(sunlight, visible);
fprintf('the visible fraction is %.1f %%', P_visible / P_full * 100);
```

```
the visible fraction is 48.8 %
```

IsOctave

Determines if code is running under GNU Octave (or Matlab)

```
% function rv = IsOctave()
```

Output: Returns logical 1 when running under GNU Octave, else returns logical 0

Usage Example:

```
IsOctave()
```

```
ans = logical
   0
```

LDomPurity

Compute dominant wavelength and purity of a spectrum using `E = [ 1/3, 1/3]` as white point

```
% function [ldom, purity] = LDomPurity(rhs)
```

Input:

`rhs:` may be spectrum (struct with lam and val), or XYZ (struct with x and y), or array with length 2

Output:

`ldom:` dominant wavelength in nm, i.e. the wavelength where the E -> x/y line intersects the monochromatic border. Negative if E -> x/y intersects magenta line, not the monochromatic border. 555 if x/y == E within circle of eps = 2.2e-16.

`purity:` purity, i.e. the E->x/y distance relative to E->border. 0 if x/y == E, 1 if x/y on border. Negative if E -> x/y intersects magenta line, not the monochromatic border.

Usage Example:

```
blue = GaussSpectrum(360:830,460,20);
[ldom,purity] = LDomPurity(blue) %#ok<ASGLU> % yes, LDom > LPeak for blue
```

```
ldom = 465.7497
purity = 0.9694
```

```
red = GaussSpectrum(360:830,630,20);
[ldom,purity] = LDomPurity(red) %#ok<ASGLU> % yes, LDom < LPeak for red
```

```
ldom = 614.0548
purity = 0.9999
```

```
magenta = AddSpectra(blue,red);
m_XYZ = CIE1931_XYZ(magenta);
[ldom,purity] = LDomPurity(m_XYZ) %#ok<ASGLU> % both values are negative
```

```
ldom = -554.0733
purity = -0.5550
```

```
green_xy = [0.16,0.80];
[ldom,purity] = LDomPurity(green_xy) %#ok<ASGLU> % x/y near border for about 530 nm
```

```
ldom = 530.5086
purity = 0.9925
```

```
[ldom,purity] = LDomPurity([1/3,1/3]) % ldom is arbitrary
```

```
ldom = 555
purity = 0
```

LinInterpol

Computes linearly interpolated values of scalar tabulated function. Very similar to built in `interp1`, but uses faster C++ DLL under Matlab (on my machine, a factor of five faster).

Used internally as a helper function for higher level library functions, but exposed as a proper library function because it may well be useful to speed up computation in another context.

```
% function yq = LinInterpol(xx,yy,xq)
```

Input:

`xx` is a 1D vector of numeric values which are not complex, and strictly ascending

`yy` is a 1D vector of numeric values which are not complex.

`xx` and *`yy`* are of same length and have at least two elements

`xq` is a 1D vector of numeric values which are not complex, and strictly ascending (the latter is NOT a requirement for `interp1`)

These preconditions are not checked.

Output:

`yq` is a vector of same length as `xq`, with linearly interpolated values. Zero if outside the `xx` range.

Usage Example:

```
LinInterpol([1 2],[3 4],[-100,1,2,1.7])
```

```
ans = 1×4
         0    3.0000    4.0000    3.7000
```

`LinInterpolAdd4Async`

Computes linearly interpolated values of the sum of four input functions. Uses C++ DLL with multithreading under Matlab. On my machine, a factor of five faster than summing the result of `interp1` calls, but not faster than summing the results of four `LinInterpol` calls. I still leave the function in place, there may be a difference for other input values and/or on another machine.

```
% function yq = LinInterpolAdd4Async(xx0,yy0,xx1,yy1,xx2,yy2,xx3,yy3,xq)
```

Input:

`xx0` is a 1D vector of numeric values which are not complex, and strictly ascending

`yy0` is a 1D vector of numeric values which are not complex.

`xx0` and `yy0` are of same length and have at least two elements.

Same applies to the `xx1/yy1, xx2/yy2, xx3/yy3` pairs, but the `xx_` arrays may all be different.

`xq` is a 1D vector of numeric values which are not complex, and strictly ascending (the latter is NOT a requirement for `interp1`!!)

Output: `yq` is a vector of same length as `xq`, with the sum of the four linearly interpolated values.

Usage Example:

```
LinInterpolAdd4Async([0 1],[1 1.1], [0 1],[2 2], [0 1],[3 3], [0 2],[4 4], [0, 1, 0.5])
```

```
ans = 1×3
   10.0000   10.1000   10.0500
```

MakeSpectrum

Creates a spectrum struct out of arrays `lam` and `val` and checks if they meet the requirements

```
% function rv = MakeSpectrum(lam, val, varargin)
```

Input:

`lam` is a 1D vector of numeric values which are not complex, > 0, and strictly ascending

`val` is a 1D vector of numeric values which are not complex.

`lam` and `val` are of same length and have at least two elements.

The requirements are checked, an error thrown if violated

`varargin:` Name/Value pairs for additional fields

Output:

`rv` is a spectrum struct with column vector fields `lam` and `val`.

Usage Example:

```
clear s
s = MakeSpectrum([400 700], [1 1],'name','flat spectrum') %#ok<NASGU>
```

```
s = struct with fields:
     lam: [2×1 double]
     val: [2×1 double]
    name: 'flat spectrum'
```

MultiplySpectra

Multiply two spectra, e.g. an LED spectrum with a transmission spectrum

```
% function rv = MultiplySpectra(lhs, rhs)
```

Input: `lhs, rhs:` Valid spectrum structs. May or may not overlap.

Output: `rv:` Spectrum struct, modeling the product of *lhs(lambda) * rhs(lambda)*. The `rv.lam` field covers the overlap region, if any, where it contains all wavelengths from both inputs. The `rv.val` field contains the product of the respective interpolated values. There is no need to have zero values outside the overlap range, as zero is always assumed outside anyway. When there is no overlap, `rv` contains a single interval as the overall min/max wavelength range, with zero value.

Usage Example:

```
clear s1 s2 prodspec;
s1 = GaussSpectrum(400:500, 450,10);
```

```
s2 = MakeSpectrum([430 470],[0 2]);
prodspec = MultiplySpectra(s1, s2);
nfig = nfig + 1;
figure(nfig);
clf;
hold on;
plot(s1.lam, s1.val);
plot(s2.lam, s2.val);
plot(prodspec.lam, prodspec.val);
legend({'s1','s2','prodspec = s1*s2'},'Location','northwest');
```

PlanckLocus

Compute the x/y, u/v, u'/v' coordinates of the Planck locus, as well as interpolation function objects for x/y, u/v, and the Judd lines (lines of equal correlated color temperature) in u/v.

```
% function rv = PlanckLocus()
```

Input: None

Output: Struct with fields:

`nT`: Number of temperature points (1001)

`invT`: Inverse absolute temperature values, from near zero (1e-11) to 0.002 (1/K), in equidistant steps. (The Planck locus points are approximately equidistant in 1/T, and not at all equidistant in T)

`T`: Absolute temperature values, from 1e11 K down to 500 K

`x`: CIE 1931 x color coordinate values of the Planck locus curve

`y`: CIE 1931 y color coordinate values of the Planck locus curve

`u`: CIE u color coordinate values of the Planck locus curve (to be used only for CCT computation purposes)

`v`: CIE v color coordinate values of the Planck locus curve (to be used only for CCT computation purposes)

`up`: CIE u' (u_prime) color coordinate values of the Planck locus curve

`vp`: CIE v' (v_prime) color coordinate values of the Planck locus curve

`xy_func`: A function object to compute interpolated values of the x/y color coordinates of the Planck locus. To be called like `xy = rv.xy_func(T)` where T is a scalar of vector of absolute temperatures, and returns an array of size `[2, length(T)]` with the interpolated x values as first column and the interpolated y values as second column. Returns `NaN` when `T` is out of range 500 .. 1e11

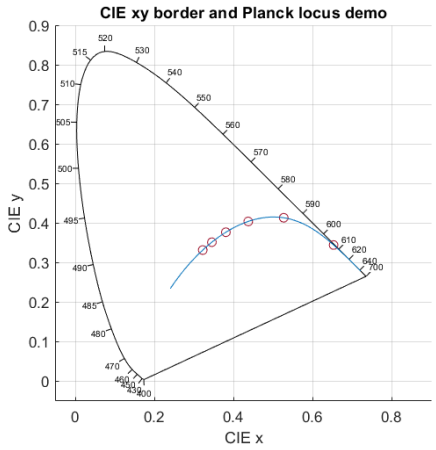
`uv_func`: Same as `xy_func`, except it returns u/v values.

`JuddLine_func`: A function object to compute interpolated values of the Judd line parameters. To be called like `jl = rv.JuddLine_func(T)` where T is a scalar, absolute temperature. Returns a struct with fields `u, v, du, dv` where `[u,v]` are the u/v coordinates of the Planck locus point for `T`, and `[du,dv]` is the Judd line direction, normalized to length 1. By definition, the Judd line is perpendicular to the Planck locus in u/v color

space, and all color points on a Judd line are deemed to have same CCT (correlated color temperature). Note that the "allowed" meaningful range is +/- 0.05 u/v length units away from the Planck locus.

Usage Example:

```
clear pl xb yb T xy
pl = PlanckLocus();
fh = figure();clf;
PlotCIExyBorder('Figure',fh);
hold on;
T = [1000 2000 3000 4000 5000 6000];
xy = pl.xy_func(T);
scatter(xy(:,1),xy(:,2));
plot(pl.x,pl.y);
axis equal;
grid on;
axis([-0.05 0.9 -0.05 0.9]);
xlabel('CIE x');
ylabel('CIE y');
title('CIE xy border and Planck locus demo')
```



`PlanckSpectrum`

Generate blackbody spectrum for some absolute temperature.

```
% function rv = PlanckSpectrum(lam_vec, T, varargin)
```

Input:

`lam_vec`: A wavelength vector, numeric, non-complex, positive, strictly ascending

`T`: Scalar real number, absolute temperature in K. May be `inf`, then returns `lam_vec.^(-4)`, scaled to `localpeak1` (this is the high temperature limit of the shape of the long wavelength tail).

`varargin`: Name/Value pairs.

`'normalize'` -> string, default `'globalpeak1'`. Allowed values:

`'globalpeak1'`: scaled such that global peak would be 1.0 even if outside lambda range

'localpeak1': scaled such that the peak value is 1.0 for the given lam_vec. Not exactly identical if global peak is in range, due to discretization

'localflux1': scaled such that integral over lam_vec is 1.0

'radiance': scaled such rv is blackbody spectral radiance, W/(wlu m²sr). wlu is wavelengthUnit, see below

'exitance': scaled such that rv is blackbody spectral exitance, W/(wlu m²). wlu is wavelengthUnit, see belo

'wavelengthUnit' -> positive real, wavelength unit in meters, default 1e-9 (nanometers).

1e-9 : lam_vec given in nm, returned spectrum is W / (nm m² sr) or W / (nm m²) for radiance/exitance scaling. rv.XYZ will be CIE XYZ values X, Y, Z, x, y

Output: Spectrum struct with fields

lam: Same as lam_vec, but column vector.

val: The spectral values, column vector of same length, normalized according to 'normalize' (default: global peak 1.0).

name: string, an appropriate name

Usage Example:

```
clear bb T
nfig = nfig + 1;
figure(nfig);
clf;
hold on;
for T = [3000 4000 5000 6000]
    bb = PlanckSpectrum(200:2000,T,'normalize','radiance');
    plot(bb.lam,bb.val);
end
xlabel('\lambda (nm)');
ylabel('spectral radiance L_\lambda (W/(m² nm sr)')
legend({'3000K','4000K','5000K','6000K'})
title('Blackbody spectral radiance')
```

PlotCIExyBorder

plotting the CIE x/y "color horseshoe", with optional ticks and other options

```
% function [ah, fh] = PlotCIExyBorder(varargin)
```

Plot the CIE x/y "color horseshoe" into a new figure or a given figure or axes handle. Various options are available to control the look. Re-use the figure and/or axes handle for further plotting into the same figure

Input:

varargin: Name/Value pairs:

'Handle': valid figure handle to use for the plot. Current hold state will be restored

`'Axes'` : valid axes handle to use for the plot. Current hold state will be restored. Overrides `'Handle'`

`'LineSpec'` : valid `LineSpec` string (see `plot` documentation), e.g. `'--b'` for dashed blue lines, see 'plot' documentation.

`'PlotOptions'`: cell array of valid plot options, e.g. `{'Color',[0.5 0.5 0.5],'LineWidth',2}`

`'Ticks'` : array of wavelength values where ticks and labels are plotted. Reasonable default. Say `...,'Ticks',[],...` to suppress

`'TickFontSize'`: number, obvious meaning. Default: `6`

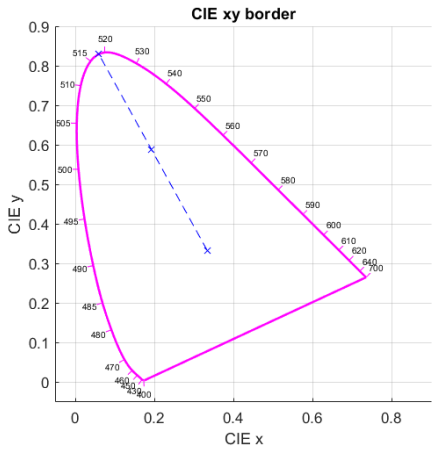
Output:

`ah`: the axis handle

`fh`: the figure handle

Usage Example: (see also the example for PlanckLocus)

```
[ah, fh] = PlotCIExyBorder('LineSpec','m','PlotOptions',{'LineWidth',1.5});
green = ComputeSpectrumColorimetry(GaussSpectrum(450:600,520,30));
% find the point on the monochromatic border corresponding to lDom
load('CIE1931_lam_x_y_z.mat','CIE1931XYZ');
xPure = LinInterpol(CIE1931XYZ.lam, CIE1931XYZ.xBorder, green.Ldom);
yPure = LinInterpol(CIE1931XYZ.lam, CIE1931XYZ.yBorder, green.Ldom);
% plot line from white point, to x/y of green, to border at green.lDom
plot([1/3,green.XYZ.x,xPure], [1/3,green.XYZ.y,yPure],'--bx');
```



ReadLightToolsSpectrumFile

Read LightTools .sre files, as supplied by many LED vendors. Reads also simple ASCII 2-column tables with any text header

```
% function rv = ReadLightToolsSpectrumFile( fn )
```

LightTools .sre files are simple ASCII 2 column text files, with a text header. The files `blue.sre` and `yellow.sre` are examples.

LightTools .sre files have the option to be photometric, i.e. values are luminous flux, whereas all spectra in this library are radiometric. If it is photometric, ReadLightToolsSpectrumFile rescales the values properly to radiometric. There is also an option to be discrete, i.e. a spectrum of narrow lines. ReadLightToolsSpectrumFile converts such files to a continuous spectrum with very narrow triangular peaks.

ReadLightToolsSpectrumFile can also be used to read a spectrum from an ASCII file which starts with any number of text header lines, i.e. lines whose first non-whitespace character is not a '0'..'9' digit, followed by lines with two numbers each.

The last line may or may not be terminated with a newline, i.e. the last line may be empty.

Input:

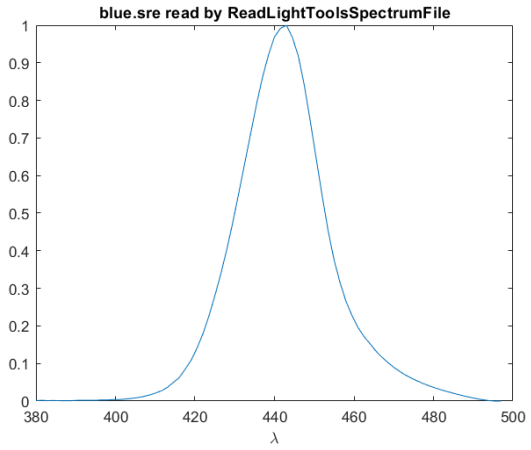
`fn:` File name. Extension must be supplied, is not automatically added.

Output:

`rv:` The spectrum

Usage Example:

```
blue = ReadLightToolsSpectrumFile( 'blue.sre' );
figure; clf;
plot(blue.lam, blue.val);
xlabel('\lambda'); title('blue.sre read by ReadLightToolsSpectrumFile');
```



SpectrumSanityCheck

Checks requirements for a spectrum variable, and returns a sanitized version of the variable, with column vectors `lam` and `val`.

```
% function [ok, msg, rv] = SpectrumSanityCheck(spec, varargin)
```

Input:

`spec` is a spectrum which should adhere to the requirements above.

`varargin` are optional name/value pairs:

'doThrow', logical, default: true. If there is an error: If flag is true, throws error exception, else returns false in ok and an error message in *msg*.

'doStrip', logical, default: false: If true returns rv with only the lam and val fields, else returns rv with all fields of input spec

Output:

ok: Logical. True if there are no requirement violations, else false

msg: Character string. Empty string if there are no requirement violations, else diagnostic message.

rv: A spectrum struct where lam and val are same as in spec except that both are always column vectors. When the doStrip flag is false, also has all other fields of spec. At requirement violation, is empty array.

Usage Example:

```
clear good bad ok1 msg1 rv1 ok2 msg2 rv2
good = MakeSpectrum([400 700], [1 1]);
[ok1, msg1, rv1] = SpectrumSanityCheck(good)
```

```
ok1 = logical
   1
msg1 =

     []
rv1 = struct with fields:
    lam: [2×1 double]
    val: [2×1 double]
```

```
bad = struct('lam', [1, 2, 1], 'val', [0 0 0]);
[ok2, msg2, rv2] = SpectrumSanityCheck(bad, 'doThrow', false)
```

```
ok2 = logical
   0
msg2 =
'lam must be strictly ascending'
rv2 =

     []
```

TestLinInterpol

A script to test the LinInterpol and LinInterpolAdd4Async functions, also measuring run times.

Input: none

Output: diagnostic text

Usage Example:

```
TestLinInterpol
```

```
LinInterpol elapsed time:1.102e-05
interp1 elapsed time: 8.4465e-05, diffnorm = 0
LinInterpolAdd4Async elapsed time: 7.7939e-05
4 x LinInterpol elapsed time: 4.3233e-05, diffnorm = 0
4 x interp1 elapsed time: 0.00033919, diffnorm = 0
```

Vlambda

Returns V(lambda), CIE 1924 as a spectrum

```
% function rv = Vlambda()
```

In steps of 1 nm from 360 nm to 830 nm

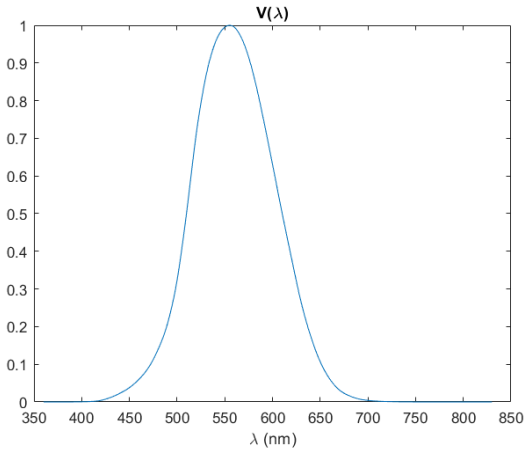
Input: none

Output:

`rv`: V(lambda) as spectrum in steps of 1 nm from 360 nm to 830 nm

Usage Example:

```
vl = Vlambda();
figure(); clf;
plot(vl.lam, vl.val);
xlabel('\lambda (nm)');
title('V(\lambda)')
```



WriteLightToolsSpectrumFile

Write LightTools .sre spectrum file, to assign to a source

```
% function WriteLightToolsSptrumFile(spectrum, filename, varargin)
```

Input:

`spectrum`: A valid spectrum with .lam and .val fields. See Spectrum requirements.

If spectrum has other fields which are simple strings or numeric values, they will be written as named comments. E.g. if spectrum.name == 'My LED' then '# name: My LED' will be written.

`filename`: name of the spectrum file. `'.sre'` is appended if necessary

optional arguments: Name-Value pairs

`'mode'`: string, only allowed value is `'discrete'`. Default spectrum mode is continuous.

`'comment'` : string or cell array of strings, will be written as comment(s)

Output: none

Usage Example:

```
s = PlanckSpectrum(400:700,5600);
WriteLightToolsSpectrumFile(s,'Planck5600.sre');
```

will write `Planck5600.sre`, with the first lines being

```
# LightTools spectrum file, created 13-Oct-2019 15:16:16

# name: Planck blackbody spectrum for T = 5600 K, normalized to global peak = 1

dfat 1.0

dataname: Planck blackbody spectrum for T = 5600 K, normalized to global peak = 1

continuous

radiometric

400 0.838583

401 0.841573
```

template for copy paste

short

```
%
```

long

Input:

`rhs`: the right hand side

Output:

`lhs`: the left hand side

Usage Example:

```

```