Coolpi API Doc

version 0.1.12

Contents

1 Description

The COlour Operations Library for the Processing of Images (coolpi) is an open-source toolbox programmed in Python for the treatment of colorimetric and spectral data. It includes classes, methods and functions developed and tested following the colorimetric standards published by the Commission Internationale de l'Éclairage (CIE, 2018).

The coolpi package has been developed as part of the INDIGO project (In-ventory and DI-sseminate G-raffiti along the d-O-naukanal) carried out by the Ludwig Boltzmann Institute in

close collaboration with the GEO Department of TU Wien University.

The achievement of colour-accurate digital images is one of the primary research topics within the INDIGO project. Therefore, the coolpi package also includes specific procedures for digital image processing and colour correction, particularly from images in RAW format.

Although the coolpi package has been designed mainly for cultural heritage documentation applications based on digital imaging techniques, we are confident that its applicability can be extended to any discipline where colour accurate registration is required.

The coolpi package is freely available under the GNU General Public License terms.

1.1 Modules

The coolpi library is structured in the following oriented objected programming (OOP) modules:

- Auxiliary: scripts with common operations for the coolpi modules.
- Colour: CIE, Colour and Spectral classes, with the basic colorimetric tools based on CIE formulation or additional published standards.
- Image: ColourChecker and Image classes with the methods and functions for image processing.

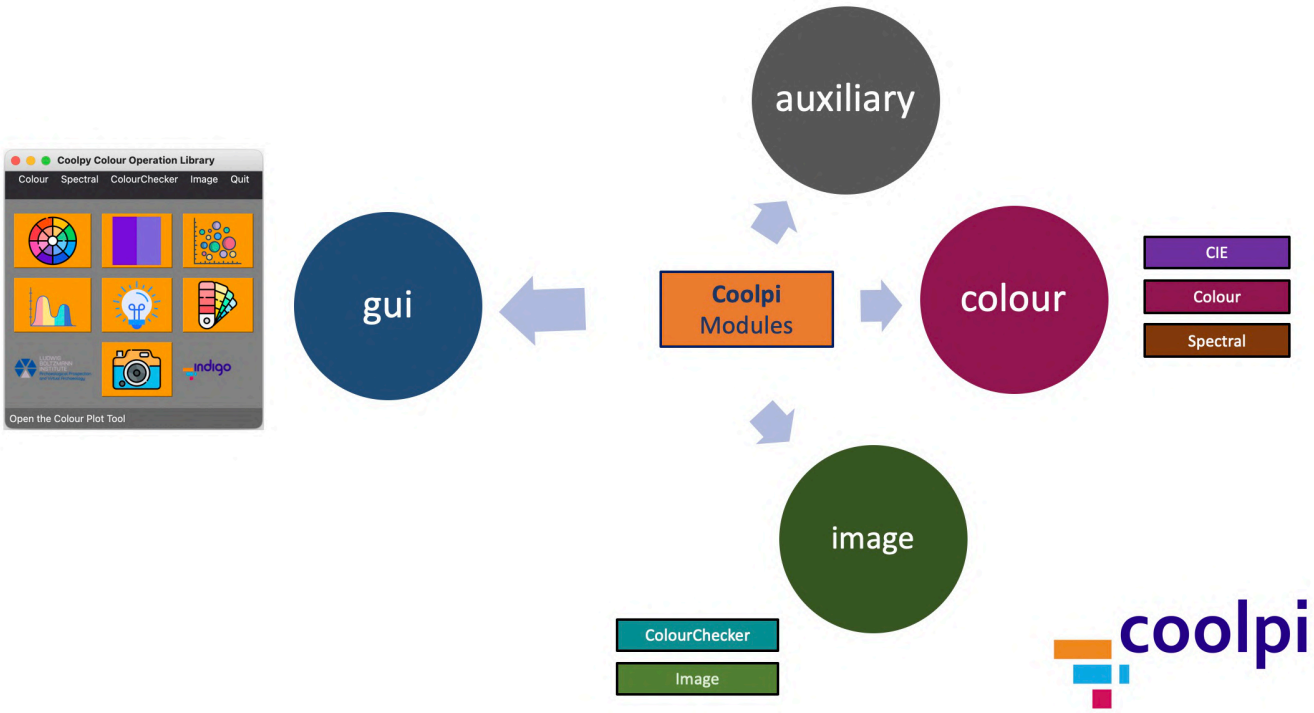

Figure 1: *Modules*

The coolpi auxilary module integrates functions that are used in the classes to carry out operations related to data loading and checking, creation and display of colorimetric and spectral graphs, and so on. It also includes the errors module, with the exceptions associated with each of the classes.

The recommended way to import the modules is as follows:

```
>>> import coolpi.auxiliary.common_operations as cop
>>> import coolpi.auxiliary.load_data as ld
>>> import coolpi.auxiliary.export_data as ed
>>> import coolpi.auxiliary.plot as cpt
```

Info

The auxiliary functions are designed to support the coolpi library classes, they are not intended to be used independently by the user. However, they can be imported and used directly from Python if desired.

The colour module is one of the pillars of the coolpi package, and is based on the colorimetric recommendations of the CIE (CIE, 2018). This module includes the CIE, Colour and Spectral main classes, and the implementation of the basic tools for the colorimetric and spectral treatment of the data.

The acquisition of colour-accurate digital images is one of the primary research topics in the international graffiti project INDIGO. Thus, the image module implemented in coolpi provides

the ColourChecker and Image classes, with the methods and functions necessary to process and obtain accurate-colour data from digital images, especially in RAW format.

In addition, a graphical interface GUI has been designed that integrates the main functionalities of the coolpi library, especially designed for non-programmer users.

2 Installation

The coolpi package can be installed directly from PyPi running the pip command on the system shell:

Installation

pip install coolpi

⚠ **Alert**

The coolpi package is based on Python 3.9. It is therefore recommended not to work with lower python versions, as the correct functioning of the library is not guaranteed.

ⓘ **Info**

The coolpi project links are as follows:

- *Coolpi PyPi*
- *Coolpi Documentation*
- *GitHub*
- *Source*
- *Jupyter Notebooks*
- *INDIGO project*
- *Ludwig Boltzmann Institut*
- *GEO Department of TU Wien University*

2.1 Dependencies

For the proper operation of coolpi, the following packages must be installed together:

- Create plots and figures: matplotlib 3.5.2
- Scientific computing: numpy 1.22.4
- Computer vision: opencv-python 4.6.0.66
- Data Analysis: pandas 1.4.2
- Qt (GUI): pyside6 6.3.0
- RAW image processing: rawpy 0.17.1
- Scientific computing: scipy 1.8.1
- Statistical data visualization: seaborn 0.11.2

> ⚠ **Alert**
>
> *The dependencies should have been installed automatically along with coolpi. Please check that everything is correct.*

3 CIE

The Commission Internationale de l'Éclairage (CIE) establishes standards of response functions, models and procedures of specification relevant to photometry, colorimetry, colour rendering, visual performance and visual assessment of light and lighting (CIE, Division 1: Vision and Colour).

The coolpi package follows in a rigorous manner the recommendations published by the CIE concerning the standard colorimetric observers, illuminants, the computation of tristimulus values, the colour space conversions formulae and colour difference equations among other colorimetric practices (CIE, 2018).

The CIE objects implemented into the coolpi package are based on the abstract class *CIE*, and can include other abstract classes according to their requeriments. The *CIE* main classes are: *Observer*, *SComponents*, *CMF*, *CFB*, and *RGBCMF*.

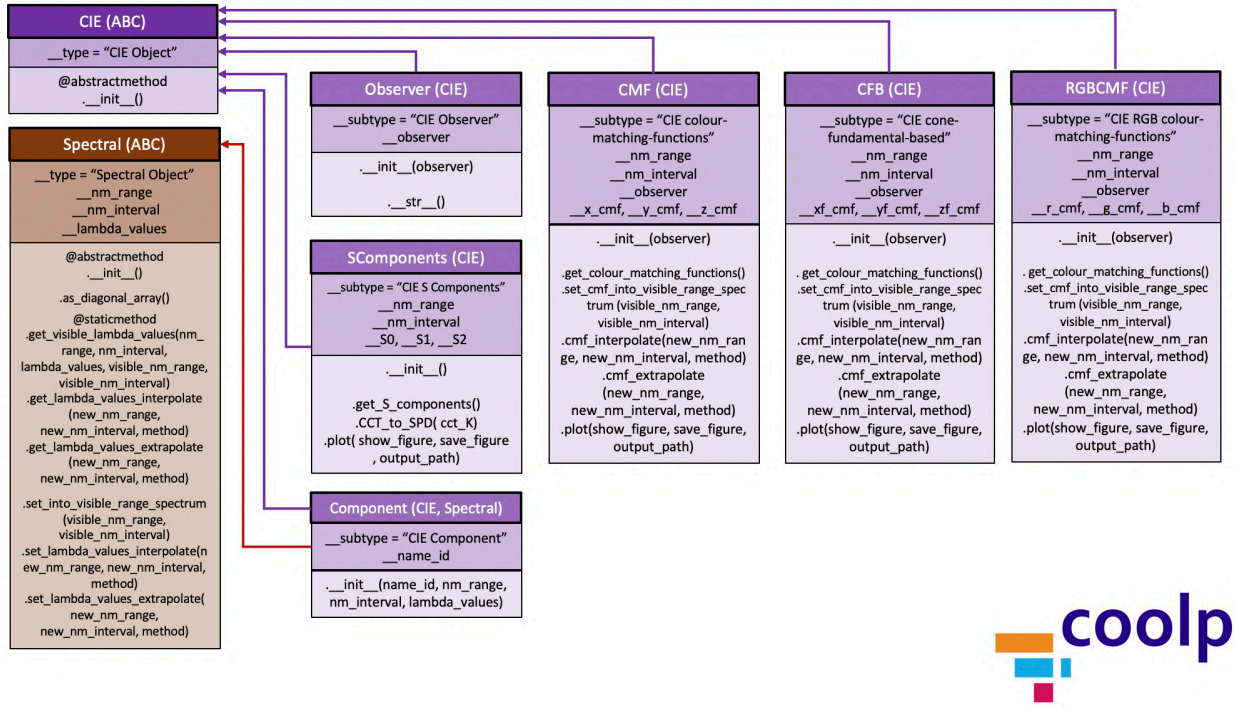

Figure 2: *UML Diagram for the CIE classes*

Info

For further explanation of some of the calculations applied, we highly recommend users to consult the standards published by the CIE, particularly the Technical Report CIE 015:2018, Colorimetry, 4th Edition (CIE, 2018). This publication provides the recommendations of the CIE concerning colorimetry, particularly the use of the standard colorimetric observers and standard illuminants, colour spaces, colour difference metrics and other colorimetric practices and formulae.

Practical use of *CIE* classes

Users are encouraged to previously take a look at the Jupyter Notebook:

`01_CIE_objects.ipynb`

International Commission on Illumination
Commission Internationale de l'Eclairage
Internationale Beleuchtungskommission

3.1 Observer

class Observer

Observer(observer=2)

The *Observer* class represents a valid CIE standard colorimetric observer.

The colour sensitivity of the human visual system (HVS) changes according to the angle of view. Based on visual colour matching experiments, the CIE originally defined the standard observer in 1931 using a 2º field of view. Subsequently, in 1964, the CIE defined a new standard observer, based on a 10º field of view experiments.

Therefore, the *Observer* class implements both observers defined by the CIE. The 2º observer refers to an ideal observer whose colour-matching properties correspond to the CIE $\bar{x}(\lambda)$, $\bar{y}(\lambda)$ and $\bar{z}(\lambda)$ colour-matching-functions (CMFs). The 10º standard observer refers to an ideal observer whose colour-matching properties correspond to the CIE colour-matching-functions adopted in 1964 (Luo, 2016).

The 1931 2° and 1964 10° standard observer functions are used in the calculation of tristimulus values to reflect the human response to the visible spectrum.

Info

CIE 015:2018. 3. Recommendations concerning standard observer data (pp.3-5) (CIE, 2018).

3.1.1 Create an instance

Observer(observer=2)

Parameter:

observer: `str`, `int` *CIE standard observer. Default: 2*

The *Observer* class receives a valid `str` or `int` type for the specification of the 2º or 10º degree CIE standard observer.

```
>>> from coolpi.colour.cie_colour_spectral import Observer
>>> obs = Observer("2")
>>> obs_10 = Observer(10)
```

In case the observer is not specified, the default value is 2 (so an instance of the 2º standard colorimetric observer is created).

```
>>> obs = Observer()
>>> print(obs)
2º standard observer (CIE 1931)
```

> ⚠ **Alert**
>
> *It is only allowed to create instances for CIE standard observers 2º and 10º degree. Otherwise, a CIEObserverError is raised.*

```
>>> Obs = Observer(3)
CIEObserverError: The observer should be a CIE standard 1931 or 1964 observer (2º or 10º)
```

3.1.2 Attributes

The class *attributes* are: *type* (returns the description of the main class), *subtype* (returns the description of the object), and *observer* (returns `int` 2 or 10).

```
>>> print(obs.type)
CIE Object
>>> print(obs.subtype)
CIE Observer
>>> print(obs.observer)
2
```

3.1.3 Method

> **`__str__`**
>
> *Returns the string representation of the object. This method is called when the `print()` or `str()` function is invoked on an object.*

```
>>> print(obs) # __str__ method
2º standard observer (CIE 1931)
```

3.2 Component

> ***class Component***
>
> *Component(name_id, nm_range, nm_interval, lambda_values)*

The *Component* class represents a valid CIE spectral component.

It is an auxiliary class designed to support the CIE *SComponents*, *CMF*, CFB and *RGBCMF* classes. Users can create an instance of the *Component* class, however, it makes no sense from a practical point of view.

The class attributes are: *type* (returns a `str` with the description of the main class), *subtype* (returns a `str` with the description of the object), *name_id* (returns a `str` with the *Component* description), *nm_range* (returns a `list` with the range for the spectral λ values), *nm_interval* (returns an `int` with the λ intervale in nm), and finally the *lambda_values* (returns a `list` with the *Component* spectral data).

3.3 SComponents

class SComponents

SComponents()

.get_S_components()

.CCT_to_SPD(cct_K)

.plot(show_figure, save_figure, output_path)

The *SComponents* class represents the CIE $S_0(\lambda)$, $S_1(\lambda)$ and $S_2(\lambda)$ components.

The *SComponents* class is the implementation of the CIE $S_0(\lambda)$, $S_1(\lambda)$ and $S_2(\lambda)$ components of daylight used in the calculation of relative spectral distribution (SPD) of CIE daylight illuminants of different correlated colour temperatures (CCT or Tcp).

It is implemented for wavelengths from range $\lambda = 300\ nm$ to $\lambda = 830\ nm$, at 5 nm intervals, derived from the data tables published by the CIE.

Info

CIE 015:2018. 11.1. Table 6. S Components of the relative SPD of daylight used in the calculation of relative SPD of CIE daylight illuminants of different correlated colour temperatures (CCT). (pp.54-55) (CIE, 2018).

3.3.1 Create an instance

SComponents()

No parameters are required to instantiate the class.

To *instantiate* the *SComponents* class:

```
>>> from coolpi.colour.cie_colour_spectral import SComponents
>>> sc = SComponents()
```

3.3.2 Attributes

The class *attributes* are: *type* (returns the description of the main class), *subtype* (returns the description of the object), *nm_range* (returns a `list` with the range for the spectral λ values), *nm_interval* (returns the λ intervale in nm), and finally the *S0*, *S1* and *S2* (returns each of the *SComponent* as a *Component* object).

```
>>> print(sc.type)
CIE Object
>>> print(sc.subtype)
CIE S Components
>>> print(sc.nm_range)
[300, 830]
>>> print(sc.nm_interval)
5
```

Each of the $S_0(\lambda)$, $S_1(\lambda)$ and $S_2(\lambda)$ components can be accessed directly as follows:

```
>>> S0 = sc.S0
>>> S1 = sc.S1
>>> S2 = sc.S2
>>> print(S0.name_id, S0.nm_range, S0.nm_interval)
S0 [300, 830] 5
```

3.3.3 Methods

To get $S_0(\lambda)$, $S_1(\lambda)$ and $S_2(\lambda)$ components:

> ***SComponents.get_S_components()***
>
> *Method to get each of the $S_0(\lambda)$, $S_1(\lambda)$ and $S_2(\lambda)$ components as a Component object.*
>
> *Returns:*
>
> $S_0(\lambda)$, $S_1(\lambda)$, $S_2(\lambda)$: `Component` *S0, S1, S2.*

To get the *SComponents*:

```
>>> S0, S1, S2 = sc.get_S_components()
```

Attributes:

```
>>> print(S0.name_id)
S0
>>> print(S0.nm_range)
[300, 830]
>>> print(S0.nm_interval)
5
>>> print(S0.lambda_values) # list of lambda values
[0.04, 3.02, 6, 17.8, 29.6, 42.45, 55.3, 56.3, 57.3, 59.55, 61.8, 61.65, 61.5, 65.15, 68.8,
66.1, 63.4, 64.6, 65.8, 80.3, 94.8, 99.8, 104.8, 105.35, 105.9, 101.35, 96.8, 105.35,
113.9,
119.75, 125.6, 125.55, 125.5, 123.4, 121.3, 121.3, 121.3, 117.4, 113.5, 113.3, 113.1,
111.95,
110.8, 108.65, 106.5, 107.65, 108.8, 107.05, 105.3, 104.85, 104.4, 102.2, 100, 98, 96,
95.55,
95.1, 92.1, 89.1, 89.8, 90.5, 90.4, 90.3, 89.35, 88.4, 86.2, 84, 84.55, 85.1, 83.5, 81.9,
82.25, 82.6, 83.75, 84.9, 83.1, 81.3, 76.6, 71.9, 73.1, 74.3, 75.35, 76.4, 69.85, 63.3,
67.5,
71.7, 74.35, 77, 71.1, 65.2, 56.45, 47.7, 58.15, 68.6, 66.8, 65, 65.5, 66, 63.5, 61, 57.15,
53.3, 56.1, 58.9, 60.4, 61.9]
```

SComponents.get_S_components_lambda_values()

Method to get the lambda values for the $S_0(\lambda)$, $S_1(\lambda)$ and $S_2(\lambda)$ components as `list` objects.

Returns:

$S_0(\lambda)$, $S_1(\lambda)$, $S_2(\lambda)$: `list` *S0, S1, S2 lambda values.*

For some calculations, the spectral values of the components can be accessed directly using the method `get_S_components_lambda_values`.

To get the spectral lambda values for each $S_0(\lambda)$, $S_1(\lambda)$ and $S_2(\lambda)$ components:

```
>>> S0_lambda, S1_lamdda, S2_lambda = sc.get_S_components_lambda_values()
```

Computation of the SPD from the CCT:

SComponents.CCT_to_SPD(cct_K)

Method to obtain the SPD of the illuminant from the correlated colour temperature in º Kelvin entered as a parameter.

Parameter:

cct_K: `float` *CCT (º Kelvin) into the range (4000-25000).*

Returns:

nm_range: `list` *range in* nm.

nm_interval: `list` *interval in* nm.

spd: `list` *Computed SPD.*

To compute the SPD of an illuminant from a CCT in º Kelvin:

```
>>> nm_range, nm_interval, spd = sc.CCT_to_SPD(6500)
```

Info

For a CCT in the range (4000, 25000) the CIE formulation is used.

CIE015:2018. 4.1.2 Other D illuminants. Eq. 4.7 to 4.11 (p. 12); Note 6 (p. 13) (CIE, 2018).

__str__

Returns the string representation of the object. This method is called when the `print()` *or* `str()` *function is invoked on an object.*

```
>>> print(sc) # __str__ method
CIE S0, S1 and S2 components
```

3.3.4 Plot

SComponents.plot(show_figure=True, save_figure=False, output_path=None)

Method to create and display the plot of the $S_0(\lambda)$, $S_1(\lambda)$ *and* $S_2(\lambda)$ *components using the matplotlib Python package.*

Parameters:

show_figure: `bool` *If True, the plot is shown. Default:* `True`.

save_figure: `bool` *If True, the figure is saved. Default:* `False`.

output_path: `os` *path for the ouput figure. Default:* `None`.

To *plot* the $S_0(\lambda)$, $S_1(\lambda)$ and $S_2(\lambda)$ components:

```
>>> sc.plot()
```

It is possible to save the figure by specifying `save_figure = True`, and entering the `output_path` as follows:

```
>>> sc.plot(show_figure = False, save_figure = True, output_path = path_figure)
```

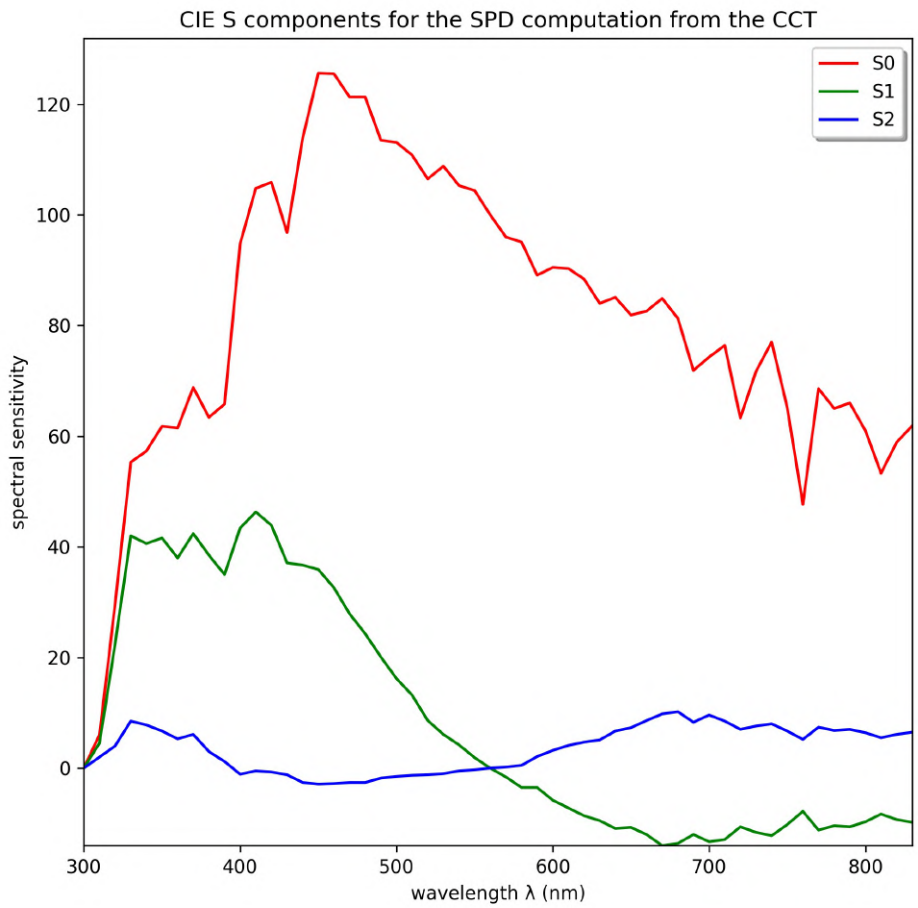

Figure 3: *SComponents Plot*

3.4 CMF

class CMF

CMF(observer=2)

.get_colour_matching_functions()

.set_cmf_into_visible_range_spectrum (visible_nm_range, visible_nm_interval)

.cmf_interpolate(new_nm_range, new_nm_interval, method)

.cmf_extrapolate(new_nm_range, new_nm_interval, method)

.plot(show_figure, save_figure, output_path)

The *CMF* class represents the CIE $\bar{x}(\lambda)$, $\bar{y}(\lambda)$ and $\bar{z}(\lambda)$ colour-matching-functions (CMFs).

For correlation with visual colour matching of fields subtending between about 1º and about 4º (2º standard colorimetric observer), or larger than 4º at the eye of the observer (10º standard colorimetric observer), it is recommended that colorimetric specification of colour stimuli should be based on the $\bar{x}(\lambda)$, $\bar{y}(\lambda)$ and $\bar{z}(\lambda)$ colour-matching-functions (CMFs) defined by the CIE.

The CMFs are used to compute the CIE XYZ tristimulus values from the SPD of the illuminant and the spectral reflectance data, following the CIE recommendations.

The *CMF* $\bar{x}(\lambda)$, $\bar{y}(\lambda)$ and $\bar{z}(\lambda)$ CMFs components are implemented for wavelengths from range $\lambda = 380\ nm$ to $\lambda = 780\ nm$, at 5 nm intervals, derived from the data tables published by the CIE.

Info

CIE 015:2018. 3.1. CIE 1931 standard colorimetric observer (p. 3) (CIE, 2018).

CIE 015:2018. 11.1. Table 1. CMF and corresponding xy coordinates of the CIE 1931 colorimetric observer (pp.43-44). Table 2. CMF and corresponding xy coordinates of the CIE 1964 colorimetric observer (pp.45-46) (CIE, 2018).

3.4.1 Create an instance

CMF(observer=2)

Parameter:

observer: `str, int, Observer` *CIE standard observer. Default: 2.*

The *CMF* class receives a valid `str` or `int` type for the specification of the 2º or 10º degree CIE standard observer.

```
>>> from coolpi.colour.cie_colour_spectral import CMF
>>> cmf_2 = CMF(2)
```

In case the observer is not specified, the default value is 2 (so an instance of the *CMF* for the CIE 1931 2º standard colorimetric observer is created).

```
>>> cmf_2 = CMF()
>>> print(cmf_2)
CIE colour-matching-functions implementation for the 2º standard observer (CIE 1931).
```

Also, the *CMF* class can be instantiated using an *Observer* instance as input parameter:

```
>>> from coolpi.colour.cie_colour_spectral import Observer
>>> obs_10 = Observer(10)
>>> cmf_10 = CMF(obs_10)
```

3.4.2 Attributes

The class *attributes* are: *type* (returns a `str` with the description of the main class), *subtype* (returns a `str` with the description of the object), *nm_range* (returns a `list` with the range for the spectral λ values), *nm_interval* (returns an `int` with the λ intervale in nm), and finally the *x_cmf*, *y_cmf* and *y_cmf* (returns each of the CMFs as a *Component* object).

```
>>> print(cmf_2.type)
CIE Object
>>> print(cmf_2.subtype)
CIE colour-matching-functions
>>> print(cmf_2.nm_range)
[380, 780]
>>> print(cmf_2.nm_interval)
5
>>> print(cmf_2.observer) # __str__ method
2º standard observer (CIE 1931)
```

Each of the CMFs can be accessed directly as follows:

```
>>> x_cmf = cmf_2.x_cmf
>>> y_cmf = cmf_2.y_cmf
>>> z_cmf = cmf_2.z_cmf
>>> print(x_cmf.name_id, x_cmf.nm_range, x_cmf.nm_interval)
x_cmf [380, 780] 5
```

3.4.3 Methods

To get the colour-matching-functions:

> ***CMF.get_colour_matching_functions()***
>
> *Method to get the $\bar{x}(\lambda)$, $\bar{y}(\lambda)$ and $\bar{z}(\lambda)$ colour-matching-functions as a Component object.*

Returns:

$\bar{x}(\lambda)$, $\bar{y}(\lambda)$, $\bar{z}(\lambda)$: `Component` *CMFs.*

To get the $\bar{x}(\lambda)$, $\bar{y}(\lambda)$ and $\bar{z}(\lambda)$ colour-matching-functions:

```
>>> x_cmf, y_cmf, z_cmf = cmf_2.get_colour_matching_functions()
```

Attributes:

```
>>> print(x_cmf.name_id)
x_cmf
>>> print(x_cmf.nm_range)
[380, 780]
>>> print(x_cmf.nm_interval)
5
>>> print(x_cmf.lambda_values) # list of lambda values
```

To set the colour-matching-functions into the visible spectrum:

CMF.set_cmf_into_visible_range_spectrum(visible_nm_range=[400,700], visible_nm_interval=10)

Method to set the spectral data into the visible spectrum.

Parameters:

visible_nm_range: `list` *visible range [min, max] in* nm*. Default: [400,700].*

visible_nm_interval: `int` *visible interval in* nm*. Default: 10.*

To set the *CFM* $\bar{x}(\lambda)$, $\bar{y}(\lambda)$ and $\bar{z}(\lambda)$ into the visible range spectrum:

```
>>> cmf_2.set_cmf_into_visible_range_spectrum(visible_nm_range=[400,700],
    visible_nm_interval=10)
>>> cmf_2.plot()
```

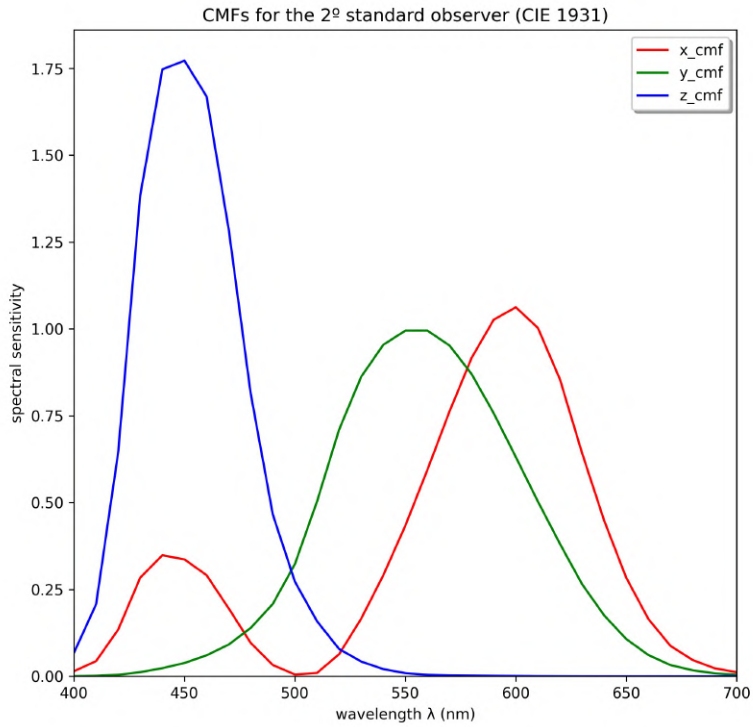

Figure 4: *CMF for the 2º standard observer (visible spectrum)*

Interpolation:

CMF.cmf_interpolate(new_nm_range, new_nm_interval, method="Akima")

Method to interpolate the spectral data of the $\bar{x}(\lambda)$, $\bar{y}(\lambda)$ and $\bar{z}(\lambda)$ colour-matching-functions.

Parameters:

new_nm_range: `list` *new range [min, max] to interpolate.*

new_nm_interval: `int` *new interval in nm.*

method: `str` *Interpolation method. Default: "Akima".*

The interpolation methods implemented are: "Linear", "Spline", "CubicHermite", "Fifth", "Sprague" and "Akima"

CMF interpolation:

```
>>> cmf_2.cmf_interpolate([380,780], 1, method="Akima")
>>> cmf_2.plot()
```

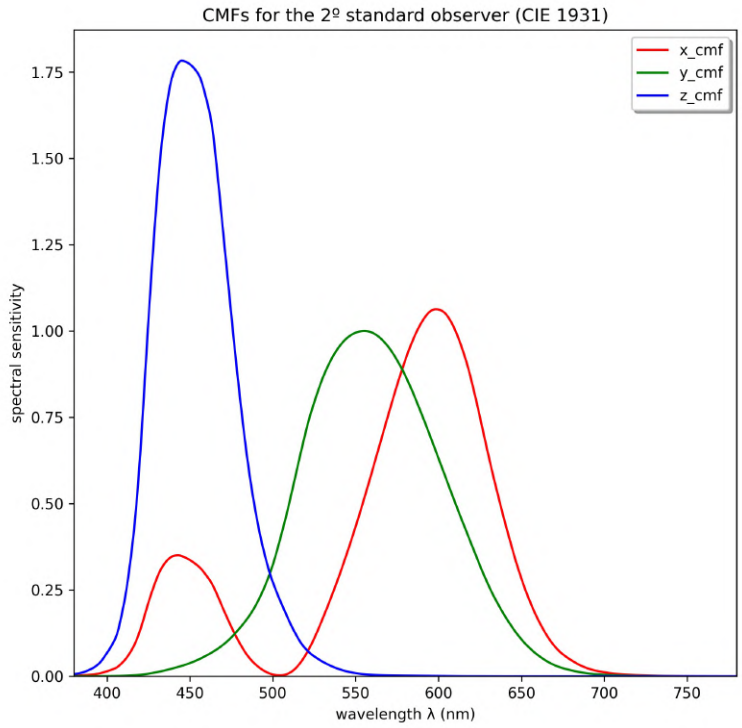

Figure 5: *CMF for the 2º standard observer (interpolation data)*

Extrapolation:

CMF.cmf_extrapolate(new_nm_range, new_nm_interval, method="Spline")

Method to extrapolate the spectral data of the $\bar{x}(\lambda)$, $\bar{y}(\lambda)$ and $\bar{z}(\lambda)$ colour-matching-functions.

Parameters:

new_nm_range: `list` *new range [min, max] to extrapolate.*

new_nm_interval: `int` *new interval in nm. Default: 10.*

method: `str` *Extrapolation method. Default: "Spline".*

The extrapolation methods implemented are: "Spline", "CubicHermite", "Fourth" and "Fifth"

⚠ **Alert**

Extrapolation of measured data may cause errors and should be used only if it can be demonstrated that the resulting errors are insignificant for the purpose of the user.

CIE 015:2018. 7.2.3 Extrapolation and interpolation (p. 24) (CIE, 2018).

CMF extrapolation:

```
>>> cmf_2.cmf_extrapolate([340,830], 1, method="Spline")
>>> cmf_2.plot()
```

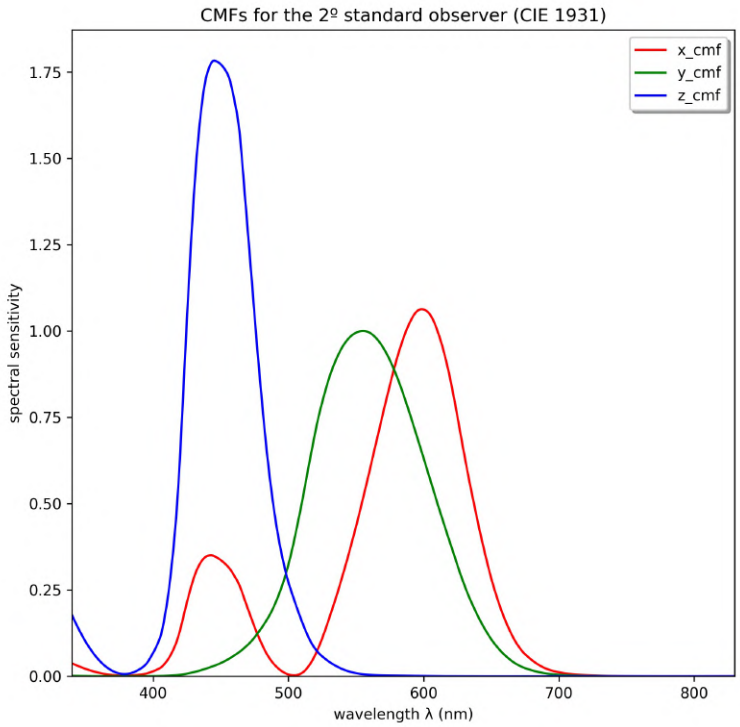

Figure 6: *CMF for the 2º standard observer (extrapolation data)*

__str__

__str__

Returns the string representation of the object. This method is called when the `print()` or `str()` function is invoked on an object.

```
>>> print(cmf_2) # __str__ method
CIE colour-matching-functions for the 2º standard observer (CIE 1931)
```

3.4.4 Plot

CMF.plot(show_figure=True, save_figure=False, output_path=None)

Method to create and display the plot of the $\bar{x}(\lambda)$, $\bar{y}(\lambda)$ and $\bar{z}(\lambda)$ colour-matching-functions using the matplotlib Python package.

Parameters:

show_figure: `bool` *If True, the plot is shown. Default:* `True`.

save_figure: `bool` *If True, the figure is saved. Default:* `False`.

output_path: `os` *path for the ouput figure. Default:* `False`.

To *plot* the $\bar{x}(\lambda)$, $\bar{y}(\lambda)$ and $\bar{z}(\lambda)$ colour-matching-functions:

```
>>> cmf_2.plot(show_figure = True, save_figure = False, output_path = None)
```

It is possible to save the figure by specifying `save_figure = True`, and entering the `output_path` as follows:

```
>>> cmf_2.plot(show_figure = False, save_figure = True, output_path = path_figure)
```

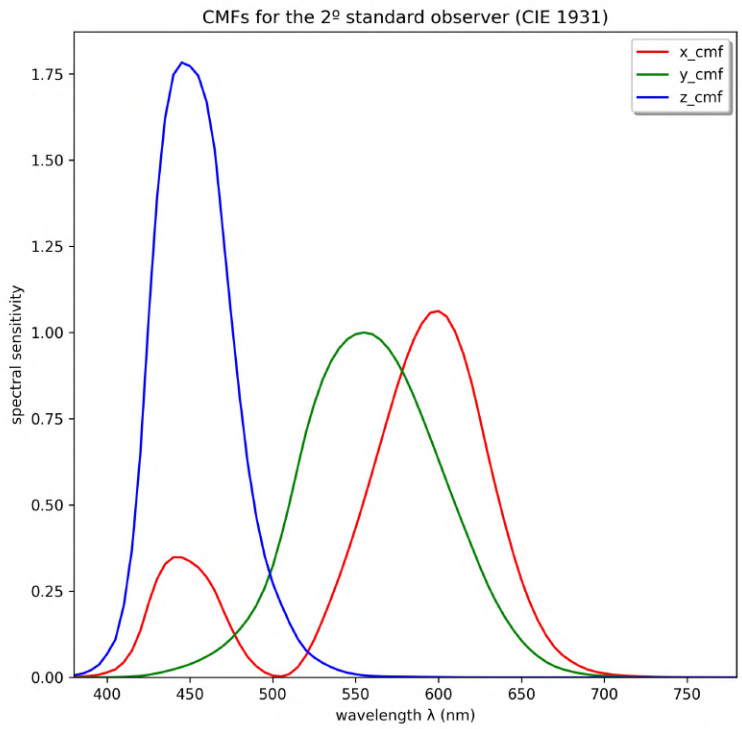

Figure 7: *CMF for the 2º standard observer (CIE 1931)*

```
>>> cmf_10.plot(show_figure = True, save_figure = False, output_path = None)
```

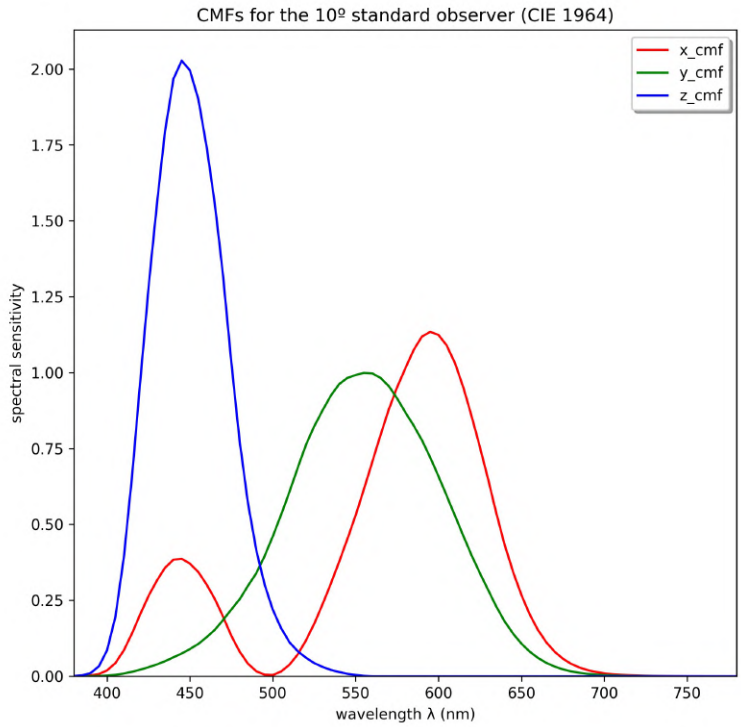

Figure 8: *CMF for the 10º standard observer (CIE 1964)*

3.5 CFB

class CFB

CFB(observer=2)

.get_colour_matching_functions()

.set_cmf_into_visible_range_spectrum (visible_nm_range, visible_nm_interval)

.cmf_interpolate(new_nm_range, new_nm_interval, method)

.cmf_extrapolate(new_nm_range, new_nm_interval, method)

.plot(show_figure, save_figure, output_path)

The *CFB* class represents the CIE cone-fundamental-based (cfb) CMFs.

The three cone fundamentals are the spectral sensitivities of the long- (L-), middle- (M-), and short- (S-) wavelength cones measured relative to light entering the cornea, known as the fundamental CMFs or $\bar{l}(\lambda)$, $\bar{m}(\lambda)$ and $\bar{s}(\lambda)$. In coolpi we have used the notation that appears in the CIE tables, that is, $\bar{x}_f(\lambda)$, $\bar{y}_f(\lambda)$ and $\bar{z}_f(\lambda)$ colour-matching-functions.

The *CFB* are implemented for wavelengths from range $\lambda = 390\ nm$ to $\lambda = 780\ nm$, at 5 nm intervals, derived from the data tables published by the CIE.

Info

CIE 015:2018. 3.3.1. Cone-fundamental-based tristimulus functions (p. 6) (CIE, 2018).

CIE 015:2018. 11.1. Table 3. CFB for 2º field size (pp.47-48). Table 4. CFB for 10º field size (pp.49-50) (CIE, 2018).

3.5.1 Create an instance

CFB(observer=2)

Parameter:

observer: `str, int, Observer` *CIE standard observer. Default: 2.*

The *CFB* class receives a valid `str` or `int` type for the specification of the 2º or 10º degree CIE standard observer.

```
>>> from coolpi.colour.cie_colour_spectral import CFB
>>> cfb_2 = CFB(2)
```

In case the observer is not specified, the default value is 2 (so an instance of the *CFB* for the CIE 1931 2º standard colorimetric observer is created).

Also, the *CFB* class can be instantiated using an *Observer* instance as input parameter:

```
>>> from coolpi.colour.cie_colour_spectral import Observer
>>> obs_10 = Observer(10)
>>> cfb_10 = CFB(obs_10)
```

3.5.2 Attributes

The class *attributes* are: *type* (returns a `str` with the description of the main class), *subtype* (returns a `str` with the description of the object), *nm_range* (returns a `list` with the range for the spectral λ values), *nm_interval* (returns an `int` with the λ intervale in nm), and finally the *xf_cmf*, *yf_cmf* and *yf_cmf* (returns each of the CMFs as a *Component* object).

```
>>> print(cfb_2.type)
CIE Object
>>> print(cfb_2.subtype)
CIE cone-fundamental-based
>>> print(cfb_2.nm_range)
[390, 780]
>>> print(cfb_2.nm_interval)
5
>>> print(cfb_2.observer) # __str__ method
2º standard observer (CIE 1931)
```

Each of the *CFB* colour-matching-functions can be accessed directly as follows:

```
>>> xf_cmf = cfb_2.xf_cmf
>>> yf_cmf = cfb_2.yf_cmf
>>> zf_cmf = cfb_2.zf_cmf
>>> print(xf_cmf.name_id, xf_cmf.nm_range, xf_cmf.nm_interval)
xf_cmf [390, 780] 5
```

3.5.3 Methods

To get the cfb colour-matching-functions:

CFB.get_colour_matching_functions()

Method to get the $\bar{x}_F(\lambda)$, $\bar{y}_F(\lambda)$ and $\bar{z}_F(\lambda)$ colour-matching-functions as a Component object.

Returns:

$\bar{x}_F(\lambda)$, $\bar{y}_F(\lambda)$ *and* $\bar{z}_F(\lambda)$ *:* `Component` *CFB CMFs.*

To get the $\bar{x}_F(\lambda)$, $\bar{y}_F(\lambda)$ and $\bar{z}_F(\lambda)$ colour-matching-functions:

```
>>> xf_cmf, yf_cmf, zf_cmf = cfb_2.get_colour_matching_functions()
```

Attributes:

```
>>> print(xf_cmf.name_id)
xf_cmf
>>> print(xf_cmf.nm_range)
[390, 780]
>>> print(xf_cmf.nm_interval)
5
>>> #print(xf_cmf.lambda_values) # list of lambda values
```

To set the cfb colour-matching-functions into the visible spectrum:

CFB.set_cmf_into_visible_range_spectrum(visible_nm_range=[400,700], visible_nm_interval=10)

Method to set the spectral data into the visible spectrum.

Parameters:

visible_nm_range: `list` *[min, max] range. Default: [400,700].*

visible_nm_interval: `int` *Interval in nm. Default: 10.*

To set the *CFB* $\bar{x}_F(\lambda)$, $\bar{y}_F(\lambda)$ and $\bar{z}_F(\lambda)$ colour-matching-functions into the visible range spectrum:

```
>>> cfb_2.set_cmf_into_visible_range_spectrum(visible_nm_range = [400,700],
visible_nm_interval = 10)
>>> cfb_2.plot(show_figure = True, save_figure = False, output_path = None)
```

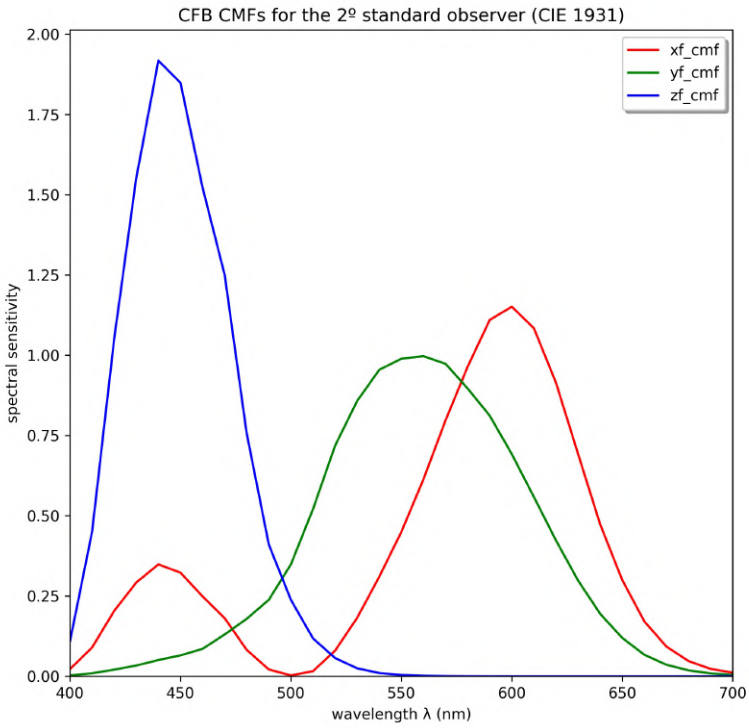

Figure 9: *CFB for the 2º standard observer (visible spectrum)*

Interpolation:

CFB.cmf_interpolate(new_nm_range, new_nm_interval, method="Akima")

Method to interpolate the spectral data of the $\bar{x}_F(\lambda)$, $\bar{y}_F(\lambda)$ *and* $\bar{z}_F(\lambda)$ *colour-matching-functions.*

Parameters:

new_nm_range: `list` *[min, max] range to interpolate.*

new_nm_interval: `int` *Interval in nm. Default: 10.*

method: `str` *Interpolation method. Default: "Akima".*

The interpolation methods implemented are: "Linear", "Spline", "CubicHermite", "Fifth", "Sprague" and "Akima".

CFB colour-matching-functions interpolation:

```
>>> cfb_2.cmf_interpolate([390,780], 1, method="Akima")
>>> cfb_2.plot()
```

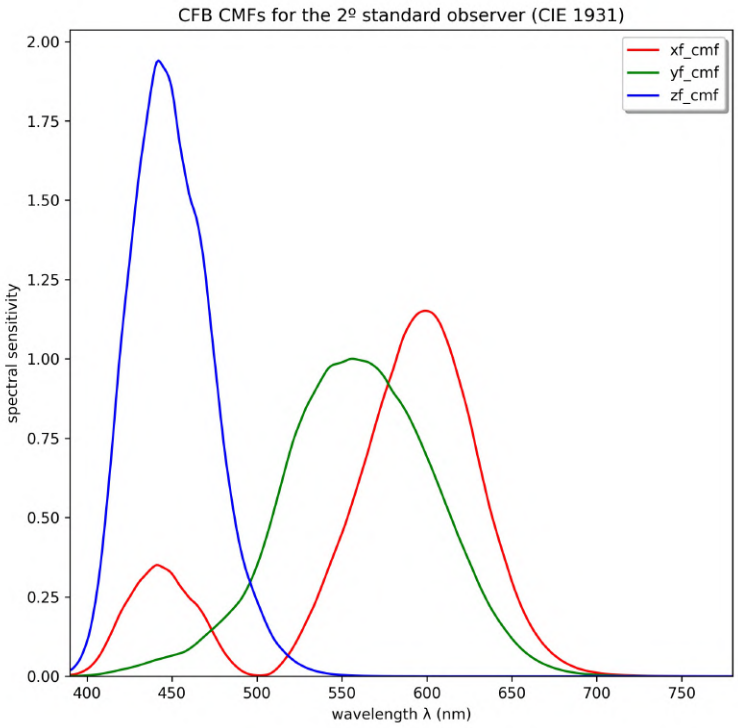

Figure 10: *CFB for the 2º standard observer (interpolation data)*

Extrapolation:

CFB.cmf_extrapolate(new_nm_range, new_nm_interval, method="Spline")

Method to extrapolate the spectral data of the $\bar{x}_F(\lambda)$, $\bar{y}_F(\lambda)$ and $\bar{z}_F(\lambda)$ colour-matching-functions.

Parameters:

new_nm_range: `list` *[min, max] range to extrapolate.*

new_nm_interval: `int` *Interval in nm. Default: 10.*

method: `str` *Extrapolation method. Default: "Spline".*

The extrapolation methods implemented are: "Spline", "CubicHermite", "Fourth" and "Fifth".

⚠ **Alert**

Extrapolation of measured data may cause errors and should be used only if it can be demonstrated that the resulting errors are insignificant for the purpose of the user.

CIE 015:2018. 7.2.3 Extrapolation and interpolation (p. 24) (CIE, 2018).

CFB colour-matching-functions extrapolation:

```
>>> cfb_2.cmf_extrapolate([340,830], 1, method="Spline")
>>> cfb_2.plot()
```

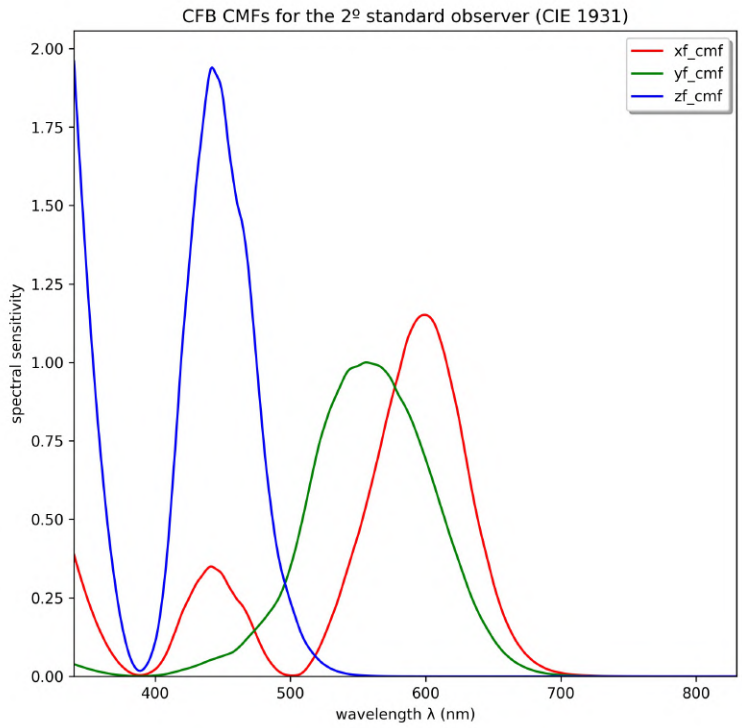

Figure 11: *CFB for the 2º standard observer (extrapolation data)*

`__str__`:

__str__

Returns the string representation of the object. This method is called when the `print()` or `str()` function is invoked on an object.

```
>>> print(cfb_2) # __str__ method
CIE cone-fundamental-based CMFs implementation for the 2º standard observer (CIE 1931)
```

3.5.4 Plot

CFB.plot(show_figure=True, save_figure=False, output_path=None)

Method to create and display the plot of the $\bar{x}_F(\lambda)$, $\bar{y}_F(\lambda)$ and $\bar{z}_F(\lambda)$ colour-matching-functions using the matplotlib Python package.

Parameters:

show_figure: `bool` *If True, the plot is shown. Default:* `True`*.*

save_figure: `bool` *If True, the figure is saved. Default:* `False`*.*

output_path: `os` *path for the ouput figure. Default:* `None`*.*

To *plot* the $\bar{x}_F(\lambda)$, $\bar{y}_F(\lambda)$ and $\bar{z}_F(\lambda)$ colour-matching-functions:

```
>>> cfb_2.plot(show_figure = True, save_figure = False, output_path = None)
```

It is possible to save the figure by specifying `save_figure = True`, and entering the `output_path` as follows:

```
>>> cfb_2.plot(show_figure = False, save_figure = True, output_path = path_figure)
```

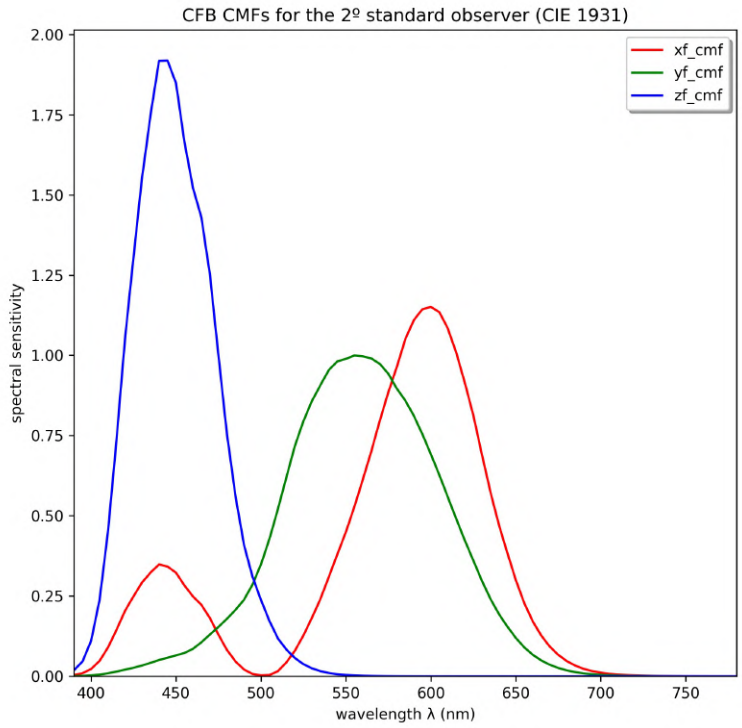

Figure 12: *CFB for the 2º standard observer (CIE 1931)*

```
>>> cfb_10.plot(show_figure = True, save_figure = False, output_path = None)
```

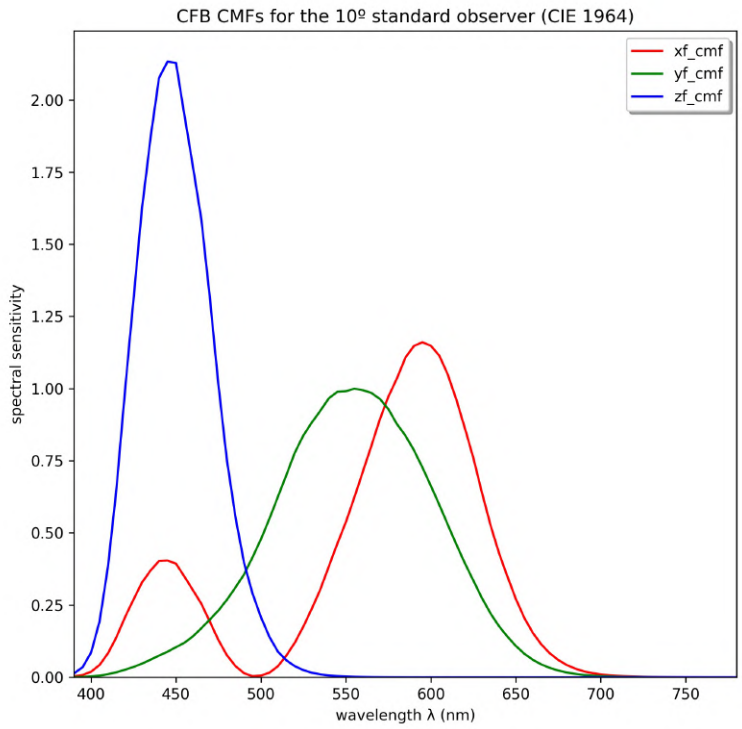

Figure 13: *CFB for the 10º standard observer (CIE 1964)*

3.6 RGBCMF

class RGBCMF

RGBCMF(observer=2)

.get_colour_matching_functions()

.set_cmf_into_visible_range_spectrum (visible_nm_range, visible_nm_interval)

.cmf_interpolate(new_nm_range, new_nm_interval, method)

.cmf_extrapolate(new_nm_range, new_nm_interval, method)

.plot(show_figure, save_figure, output_path)

The *RGBCMF* class represents the CIE rgb colour-matching-functions.

The CMFs $\bar{x}(\lambda)$, $\bar{y}(\lambda)$ and $\bar{z}(\lambda)$ were originally derived from $\bar{r}(\lambda)$, $\bar{g}(\lambda)$ and $\bar{b}(\lambda)$ CMFs referring to the spectral reference stimuli $[R]$, $[G]$, $[B]$.

The *RGBCMF* class is implemented for wavelengths from range $\lambda = 380\ nm$ to $\lambda = 780\ nm$, at 5 nm intervals, derived from the data tables published by the CIE.

Info

CIE 015:2018. Annex B. B1. Determination of the $\bar{r}(\lambda)$, $\bar{g}(\lambda)$ and $\bar{b}(\lambda)$ colour-matching-functions (p. 82) (CIE, 2018).

CIE 015:2018. Table B.1 (pp.85-86). Table B.2 (pp.87-88) (CIE, 2018).

3.6.1 Create an instance

RGBCMF(observer=2)

Parameter:

observer: `str`, `int`, `Observer` *2 or 10 CIE standard observer. Default: 2.*

The *RGBCMF* class receives a valid `str` or `int` type for the specification of the 2º or 10º degree CIE standard observer.

```
>>> from coolpi.colour.cie_colour_spectral import RGBCMF
>>> rgbcmf_2 = RGBCMF(2)
```

In case the observer is not specified, the default value is 2 (so an instance of the *RGBCMF* for the CIE 1931 2º standard colorimetric observer is created).

Also, the *RGBCMF* class can be instantiated using an *Observer* instance as input parameter:

```
>>> from coolpi.colour.cie_colour_spectral import Observer
>>> obs_10 = Observer(10)
>>> rgbcmf_10 = RGBCMF(obs_10)
```

3.6.2 Attributes

The class *attributes* are: *type* (returns a `str` with the description of the main class), *subtype* (returns a `str` with the description of the object), *nm_range* (returns a `list` with the range for the spectral λ values), *nm_interval* (returns an `int` with the λ intervale in *nm*), and finally the *r_cmf*, *g_cmf* and *b_cmf* (returns each of the CMFs as a *Component* object).

```
>>> print(rgbcmfb_2.type)
CIE Object
>>> print(rgbcmfb_2.subtype)
CIE RGB Colour-matching-functions
>>> print(rgbcmfb_2.nm_range)
[380, 780]
>>> print(rgbcmfb_2.nm_interval)
5
>>> print(rgbcmfb_2.observer) # __str__ method
2º standard observer (CIE 1931)
```

Each of the *RGBCMF* colour-matching-functions can be accessed directly as follows:

```
>>> r_cmf = rgbcmfb_2.r_cmf
>>> g_cmf = rgbcmfb_2.g_cmf
>>> b_cmf = rgbcmfb_2.b_cmf
>>> print(r_cmf.name_id, r_cmf.nm_range, r_cmf.nm_interval)
r_cmf [380, 780] 5
```

3.6.3 Methods

To get the rgb colour-matching-functions:

> ***RGBCMF.get_colour_matching_functions()***
>
> *Method to get the $\bar{r}(\lambda)$, $\bar{g}(\lambda)$ and $\bar{b}(\lambda)$ colour-matching-functions as a Component object.*

Returns:

$\bar{r}(\lambda)$, $\bar{g}(\lambda)$, $\bar{b}(\lambda)$: `Component` *RGB CMFs.*

To get the $\bar{r}(\lambda)$, $\bar{g}(\lambda)$ and $\bar{b}(\lambda)$ colour-matching-functions:

```
>>> r_cmf, g_cmf, b_cmf = rgbcmfb_2.get_colour_matching_functions()
```

Attributes:

```
>>> print(r_cmf.name_id)
r_cmf
>>> print(r_cmf.nm_range)
[380, 780]
>>> print(r_cmf.nm_interval)
5
>>> print(r_cmf.lambda_values) # list of lambda values
```

To set the rgb colour-matching-functions into the visible spectrum:

RGBCMF.set_cmf_into_visible_range_spectrum(visible_nm_range=[400-700], visible_nm_interval=10)

Method to set the spectral data into the visible spectrum.

Parameters:

visible_nm_range: `list` *[min, max] range. Default: [400,700].*

visible_nm_interval: `int` *Interval in nm. Default: 10.*

To set the *RGBCMF* $\bar{r}(\lambda)$, $\bar{g}(\lambda)$ and $\bar{b}(\lambda)$ colour-matching-functions into the visible range spectrum:

```
>>> rgbcmfb_2.set_cmf_into_visible_range_spectrum(visible_nm_range = [400,700],
visible_nm_interval = 10)
>>> rgbcmfb_2.plot(show_figure = True, save_figure = False, output_path = None)
```

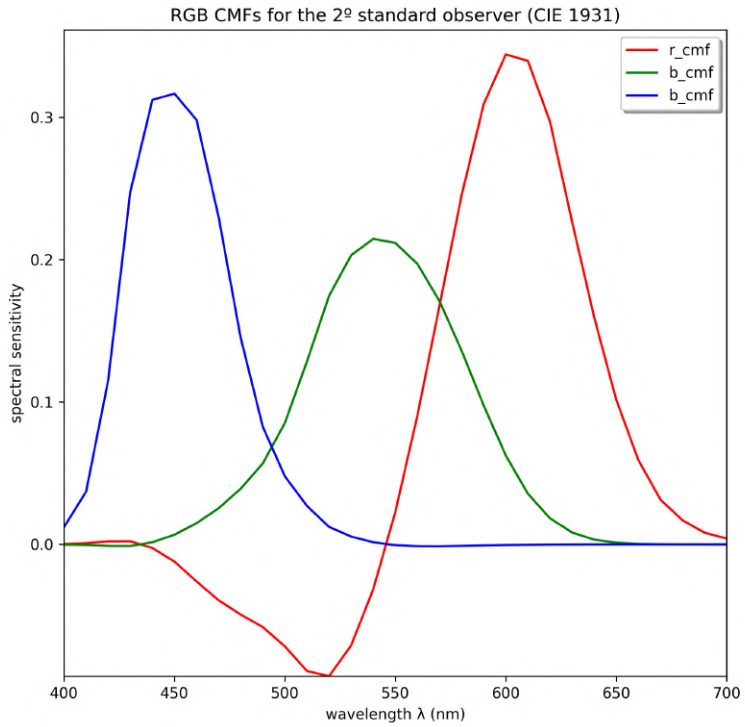

Figure 14: *RGBCMF for the 2º standard observer (visible spectrum)*

Interpolation:

RGBCMF.cmf_interpolate(new_nm_range, new_nm_interval, method="Akima")

Method to interpolate the spectral data of the $\bar{r}(\lambda)$, $\bar{g}(\lambda)$ and $\bar{b}(\lambda)$ colour-matching-functions.

Parameters:

new_nm_range: `list` *[min, max] range to interpolate.*

new_nm_interval: `int` *Interval in nm. Default: 10.*

method: `str` *Interpolation method. Default: "Akima".*

The interpolation methods implemented are: "Linear", "Spline", "CubicHermite", "Fifth", "Sprague" and "Akima".

RGBCMF colour-matching-functions interpolation:

```
>>> rgbcmfb_2.cmf_interpolate([380,780], 1, method="Akima")
>>> rgbcmfb_2.plot()
```

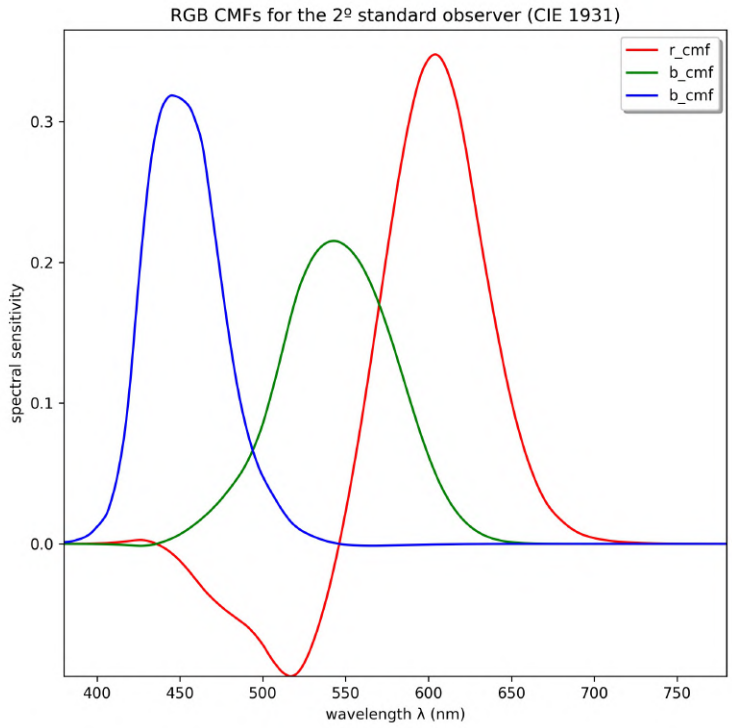

Figure 15: *RGBCMF for the 2º standard observer (interpolation data)*

Extrapolation:

RGBCMF.cmf_extrapolate(new_nm_range, new_nm_interval, method="Spline")

Method to extrapolate the spectral data of the $\bar{r}(\lambda)$, $\bar{g}(\lambda)$ and $\bar{b}(\lambda)$ colour-matching-functions.

Parameters:

new_nm_range: `list` *[min, max] range to extrapolate.*

new_nm_interval: `int` *Interval in nm. Default: 10.*

method: `str` *Extrapolation method. Default: "Spline".*

The extrapolation methods implemented are: "Spline", "CubicHermite", "Fourth" and "Fifth".

⚠ **Alert**

Extrapolation of measured data may cause errors and should be used only if it can be demonstrated that the resulting errors are insignificant for the purpose of the user.

CIE 015:2018. 7.2.3 Extrapolation and interpolation (p.24) (CIE, 2018).

RGBCMF colour-matching-functions extrapolation:

```
>>> rgbcmfb_2.cmf_extrapolate([340,830], 1, method="Spline")
>>> rgbcmfb_2.plot()
```

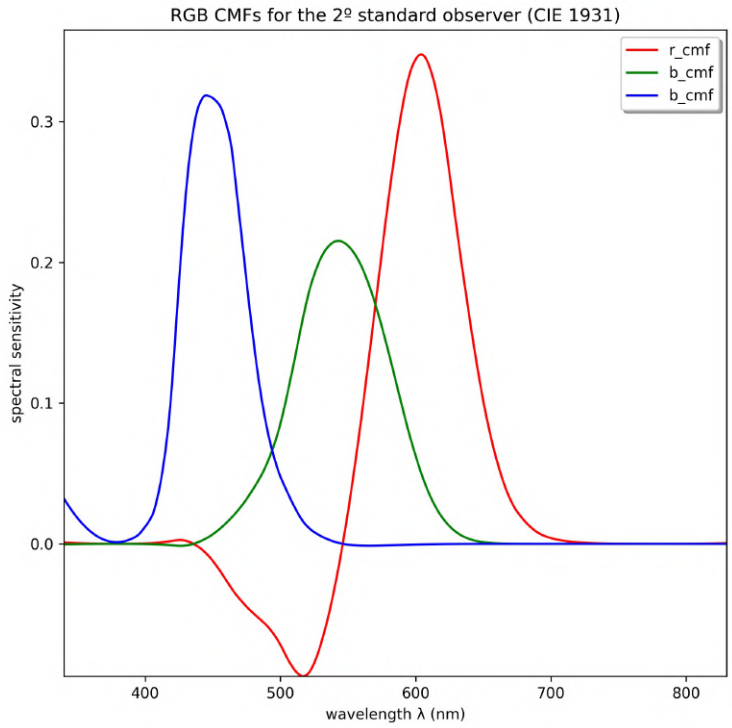

Figure 16: *RGBCMF for the 2º standard observer (extrapolation data)*

`__str__`:

`__str__`

Returns the string representation of the object. This method is called when the `print()` or `str()` function is invoked on an object.

```
>>> print(rgbcmfb_2) # __str__ method
CIE RGB colour-matching-functions implementation for the 2º standard observer (CIE 1931)
```

3.6.4 Plot

RGBCMF.plot(show_figure=True, save_figure=False, output_path=None)

Method to create and display the plot of the $\bar{r}(\lambda)$, $\bar{g}(\lambda)$ and $\bar{b}(\lambda)$ colour-matching-functions using the matplotlib Python package.

Parameters:

show_figure: `bool` *If True, the plot is shown. Default:* `True`.

save_figure: `bool` *If True, the figure is saved. Default:* `False`.

output_path: `os` *path for the ouput figure. Default:* `None`.

To *plot* the $\bar{r}(\lambda)$, $\bar{g}(\lambda)$ and $\bar{b}(\lambda)$ colour-matching-functions:

```
>>> rgbcmfb_2.plot(show_figure = True, save_figure = False, output_path = None)
```

It is possible to save the figure by specifying `save_figure = True`, and entering the `output_path` as follows:

```
>>> rgbcmfb_2.plot(show_figure = False, save_figure = True, output_path = path_figure)
```

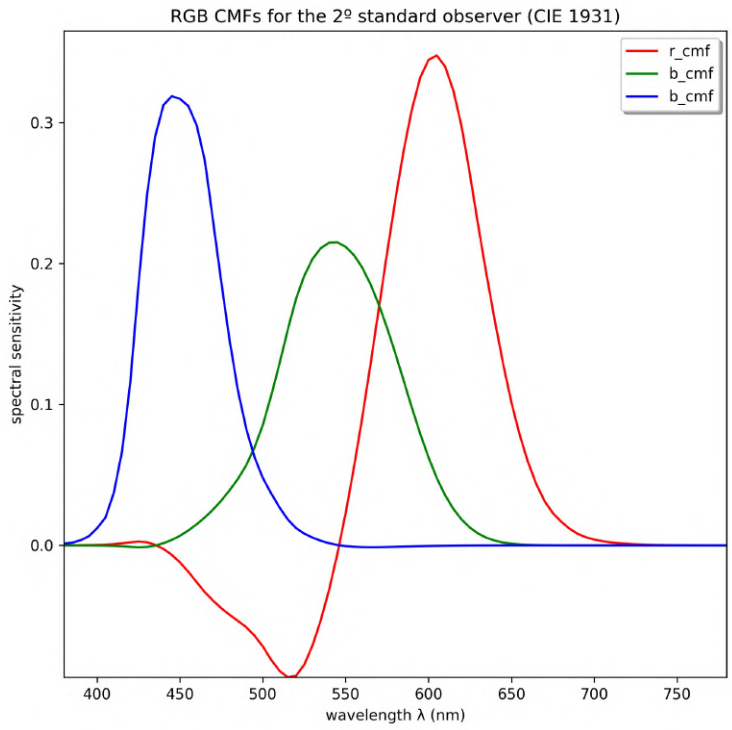

Figure 17: *RGBCMF for the 2º standard observer (CIE 1931)*

```
>>> rgbcmfb_10.plot(show_figure = True, save_figure = False, output_path = None)
```

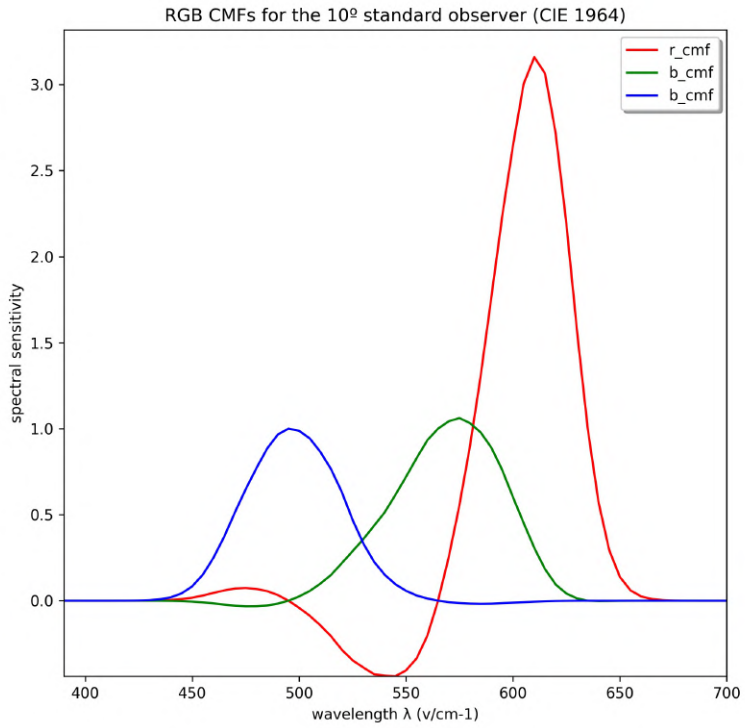

Figure 18: *RGBCMF for the 10º standard observer (CIE 1964)*

4 Colour

The *Colour* classes support the main colour spaces defined by the CIE. Since 1931, the CIE has developed systems to express colour numerically (CIE, 2018). Colour spaces define the quantitative relationship between physical pure colours in the electromagnetic visible spectrum, and physiological perceived colours in human vision. The mathematical relationships that define colour spaces are essential tools for colour management. Thus, a colour space describes an environment in which colors can be represented, ordered, compared, or computed numerically for practical purposes. Color spaces can be three-dimensional (CIE XYZ) or two-dimensional (CIE xy).

The Colour classes implemented are: *CIEXYZ*, *CIExyY*, *CIEuvY*, *CIELAB*, *CIELCHab*, *CIELUV* and *CIELCHuv*.

A particular *WhitePoint(Colour)* class has been defined to support illuminant classes (*Illuminant*, *IlluminantFromCCT* and *MeasuredIlluminant*). The *WhitePoint* class refers to illuminant white point coordinates, usually known as X_n, Y_n, Z_n.

Also, an *sRGB* class has been included, following the IEC/4WD 61966-2-1 specification (IEC, 1999). Although the sRGB is not strictly a CIE colour space, the sRGB data is computed from XYZ tristimulus values under the CIE standard illuminant D65, so it can be

considered as an independent and physically-based colour space. In addition, the sRGB is a color space suitable for being displayed on any device which allows working with the sRGB format.

Figure 19: *Colour classes*

Colour objects include class methods to perform basic colorimetric operations such as the transformations between colour spaces, computing colour difference metrics, the application of chromatic adaptation transforms, lambda operations (e.g. interpolation or extrapolation methods), and ploting among other functionalities.

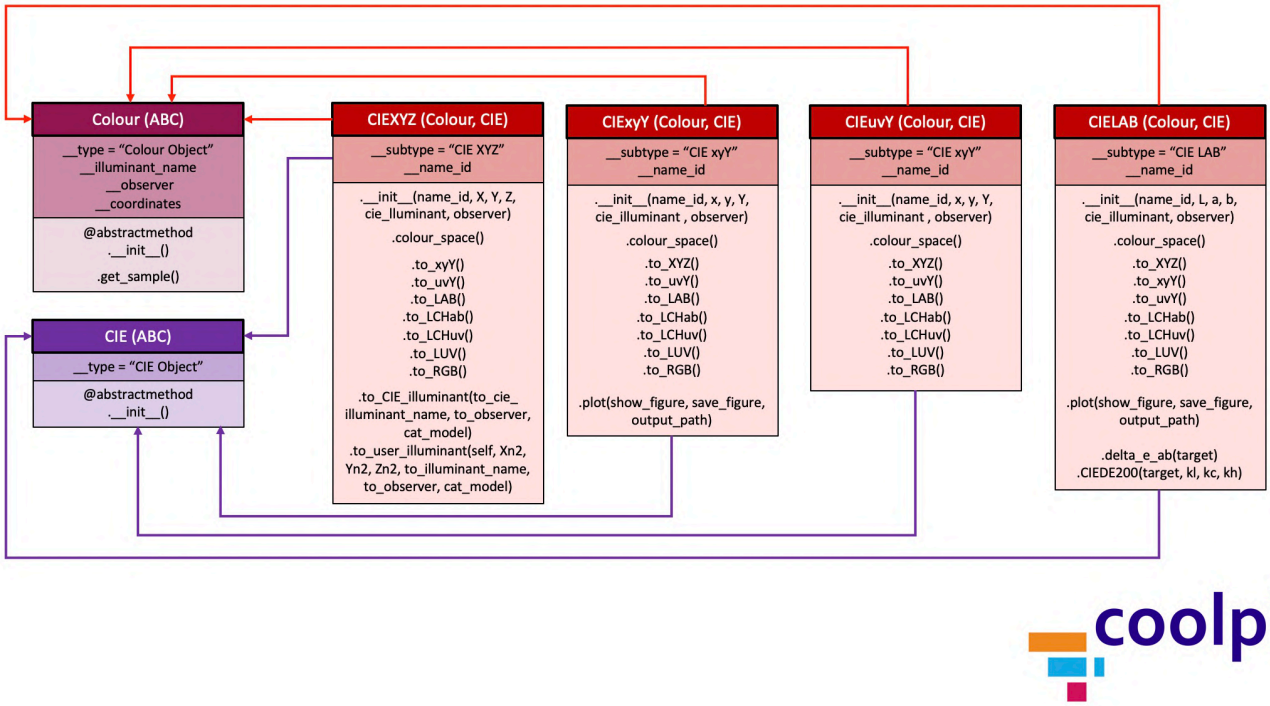

Figure 20: *UML Diagram for the Colour classes (A)*

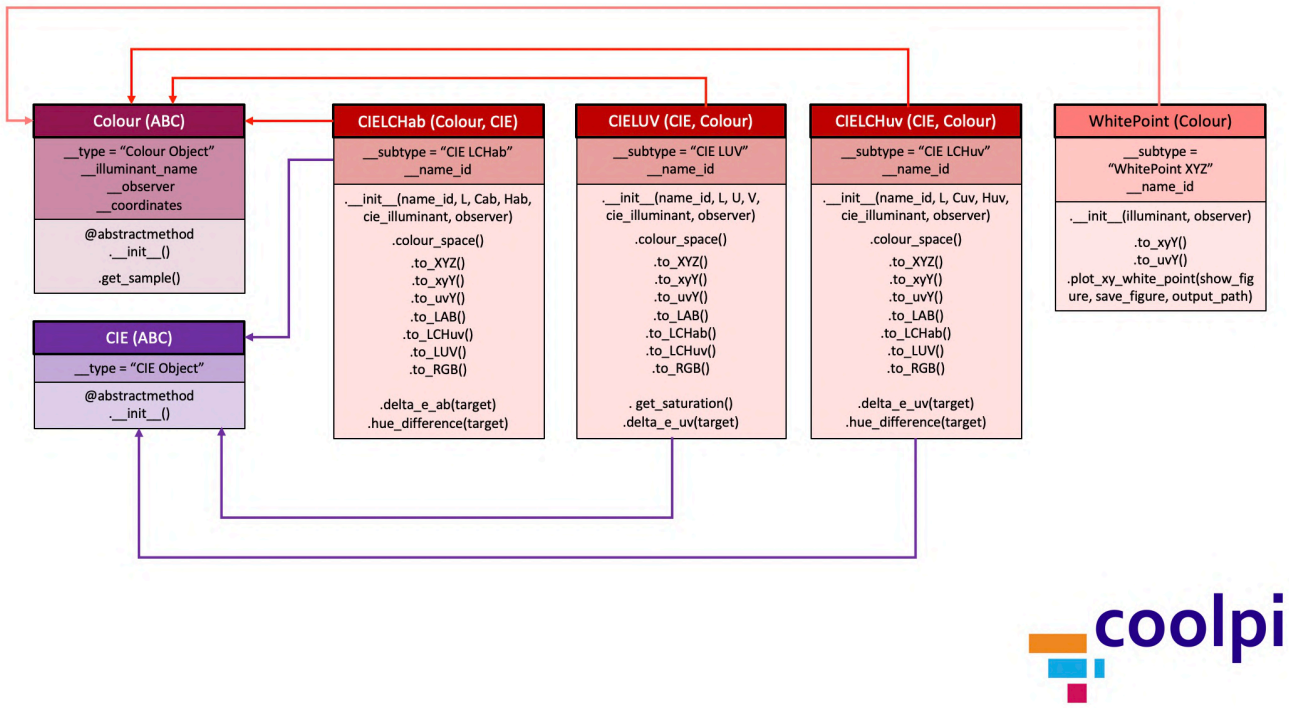

Figure 21: *UML Diagram for the Colour classes (B)*

Although these functionalities are included in the colour objects, they can be used as independent functions by the user. In this case, we recommend that the modules be imported as follows:

```
>>> import coolpi.colour.cat_models as cat
>>> import coolpi.colour.colour_difference as cde
>>> import coolpi.colour.colour_space_conversion as csc
>>> import coolpi.colour.lambda_operations as lo
```

Practical use of *Colour* classes

Users are encouraged to previously take a look at the Jupyter Notebook:

02a_Colour_objects.ipynb

4.1 CIEXYZ

class CIEXYZ

CIEXYZ(name_id, X, Y, Z, cie_illuminant="D65", observer=2)

.to_xyY()

.to_uvY()

.to_LAB()

.to_LCHab()

.to_LUV()

.to_LCHuv()

.to_RGB(rgb_name_space)

.to_CIE_illuminant(to_cie_ illuminant_name, to_observer, cat_model)

.to_user_illuminant(X_{n2}, Y_{n2}, Z_{n2}, to_illuminant_name, to_observer, cat_model)

The *CIEXYZ* class represents a colour object in the CIE XYZ colour space.

The CIE 1931 XYZ colour space allows any colour stimulus (usually expressed in terms of radiance at fixed wavelength intervals in the visible spectrum) to be represented with three parameters X, Y and Z, known as tristimulus values. The XYZ tristimulus values are fundamental measures of colour and are directly used in a number of colour management operations. This colour space serves as a reference to define many other colour spaces.

The CIE XYZ colour space is based on the additive colour mixing principle. Thus, all colour signals can be matched by the additive mixture of three primaries. In this colour space, the primary colours used are not real colours, in the sense that they cannot be generated with any light spectrum.

The second coordinate Y represents the luminance, that is, the total radiation reflected in the visible spectrum. Z is quasi-equal to blue stimulation (or the S cone response of the human eye), and X is a linear combination of cone response curves chosen to be nonnegative (Schanda, 2007).

The XYZ tristimulus values can be obtained directly from the spectral reflectance measurement of a colour sample by integrating the product of the reflectance with the spectral power distribution (SPD) of a light source (or illuminant) and with the CIE colour-matching-functions (CIE, 2018).

Info

CIE 015:2018. 7.1. Calculation of tristimulus values (pp.21-22). (CIE, 2018).

4.1.1 Create an instance

CIEXYZ(name_id, X, Y, Z, cie_illuminant="D65", observer=2)

Parameters:

name_id: `str` *Sample ID.*

X: `float` *X tristimulus value.*

Y: `float` *Y tristimulus value.*

Z: `float` *Z tristimulus value.*

cie_illuminant: `str`, `Illuminant` *CIE Illuminant. Default: "D65".*

observer: `str`, `int`, `Observer` *CIE standard observer. Default: 2.*

To *create an instance* of the *CIEXYZ* class, simply enter the required parameters as follows:

```
>>> from coolpi.classes.cie_colour_spectral import CIEXYZ
>>> sample_XYZ = CIEXYZ(name_id="Sample_1", X=10.234, Y=5.871, Z=22.109,
    cie_illuminant="D65", observer=2)
```

An *Illuminant* instance can also be passed as a parameter:

```
>>> from coolpi.colour.cie_colour_spectral import Illuminant
>>> D65 = Illuminant("D65")
>>> sample_XYZ = CIEXYZ(name_id="Sample_1", X=10.234, Y=5.871, Z=22.109,
    cie_illuminant=D65, observer=2)
```

The colour sample should be referred to a valid CIE standard illuminant and observer:

```
>>> sample_XYZ = CIEXYZ(name_id="Sample_1", X=10.234, Y=5.871, Z=22.109,
    cie_illuminant="D30", observer=2)
CIEIlluminantError: The input illuminant name is not a valid CIE standard illuminant
>>> sample_XYZ = CIEXYZ(name_id="Sample_1", X=10.234, Y=5.871, Z=22.109,
    cie_illuminant="D65", observer=20)
CIEObserverError: The observer should be a CIE standard 1931 or 1964 observer (2º or 10º)
```

⚠ **Alert**

It is only allowed to create instances for a CIE standard illuminant and observer. Otherwise, a CIEIlluminantError or CIEObserverError is raised.

4.1.2 Attributes

The class *attributes* are: *type* (returns a `str` with the description of the main class), *subtype* (returns a `str` with the description of the object), *colour_space* (returns a `str` with the colour space (equivalent to `subtype`), *name_id* (returns a `str` with the sample id), *coordinates* (returns a `list` with the three coordinates as `float` of the colour sample), *illuminant* (returns an *Illuminant* instance), and *observer* (returns an *Observer* instance).

```
>>> print(sample_XYZ.type)
Colour Object
>>> print(sample_XYZ.colour_space()) # subtype
CIE XYZ
>>> print(sample_XYZ.subtype) # returns colour space
CIE XYZ
>>> print(sample_XYZ.name_id)
Sample_1
>>> print(sample_XYZ.coordinates)
[10.234, 5.871, 22.109]
>>> print(sample_XYZ.illuminant) # str method
Illuminant object: CIE D65 standard illuminant
>>> print(sample_XYZ.observer) # observer str method
2º standard observer (CIE 1931)
>>> print(sample_XYZ.get_sample()) # colour data as dict
{'Sample_1': [10.234, 5.871, 22.109]}
```

4.1.3 Methods

Colour Space Conversion (CSC):

It is only necessary to execute the class method that performs the colour conversion to the output colour space, for example:

```
>>> sample_xyY = sample_XYZ.to_xyY() # returns a CIExyY class object
>>> type(sample_xyY)
coolpi.colour.cie_colour_spectral.CIExyY
```

After applying the colour conversion, a new *Colour* object is returned.

> **Info**
>
> *Similarly, the transformation can be applied between the other implemented colour spaces.*

The alternative way to obtain the colour coordinates in the desired output colour space is:

```
>>> print("xyY   = ", sample_XYZ.to_xyY().coordinates)
xyY   =  [0.2678076097765217, 0.1536347935311666, 5.871]
>>> print("uvY   = ", sample_XYZ.to_uvY().coordinates)
uvY   =  [0.24866060039118973, 0.320963881768372, 5.871]
>>> print("LAB   = ", sample_XYZ.to_LAB().coordinates)
LAB   =  [29.084647940987004, 43.545094568382126, -39.82173471482208]
>>> print("LCHab = ", sample_XYZ.to_LCHab().coordinates)
LCHab =  [29.084647940987004, 59.008014851094664, 317.5572656376388]
>>> print("LUV   = ", sample_XYZ.to_LUV().coordinates)
LUV   =  [29.084647940987004, 19.22024307991132, -55.72304917618993]
>>> print("LCHuv = ", sample_XYZ.to_LCHuv().coordinates)
LCHuv =  [29.084647940987004, 58.94468554113221, 289.03056690015865]
```

If a conversion to RGB space is desired:

```
>>> print("sRGB     = ", sample_XYZ.to_RGB(rgb_name_space = "sRGB").coordinates)
sRGB     =  [0.39755114527023666, 0.1523320829991103, 0.5141947956274906]
>>> print("AdobeRGB = ", sample_XYZ.to_RGB(rgb_name_space = "Adobe").coordinates)
AdobeRGB =  [0.3484283664374471, 0.1522825496876888, 0.5051836729448721]
>>> print("AppleRGB = ", sample_XYZ.to_RGB(rgb_name_space = "Apple").coordinates)
AppleRGB =  [0.3835002641849221, 0.12346656707261819, 0.5212116703733893]
```

Note:

The colour conversion between CIE XYZ and the other CIE spaces is implemented as methods of the class itself. However, it is possible to apply the transformation functions between colour spaces by importing the module `colour_space_conversion`.

Please, see interactive Jupyter Notebook:

`02b_Colour_Space_Conversion.ipynb`

Info

A more detailed explanation of the formulas and methodology implemented to perform the conversion between color spaces can be found in the CSC section.

Chromatic Adaptation Transforms (CATs):

CIEXYZ.to_CIE_illuminant(to_cie_illuminant_name, to_observer, cat_model="von Kries")

Parameters:

to_cie_illuminant_name: `str` Target CIE illuminant name.

to_observer: `int` *Observer (2 or 10).*

cat_model: `str` *CAT method. Default: “von Kries”.*

Returns:

XYZ: `CIEXYZ` *CIEXYZ referred to the new illuminant.*

Chromatic Adaptation Transforms (CATs) are implemented into the *CIEXYZ* class. The models implemented are: “von Kries”, “Bradford”, “Sharp”, “CMCCAT200", “CAT02", “BS” and “BSPC”.

The method returns a new *CIEXYZ* object, for example:

```
>>> sample_XYZ_d50 = sample_XYZ.to_CIE_illuminant(to_cie_illuminant_name ="D50",
    to_observer=2, cat_model="Bradford")
>>> type(sample_XYZ_d50)
coolpi.colour.cie_colour_spectral.CIEXYZ
```

A new *CIEXYZ* instance is created with the XYZ values referred to the new illuminant.

Attributes:

```
>>> print(sample_XYZ_d50.colour_space())
CIE XYZ
>>> print(sample_XYZ_d50.name_id)
Sample_1
>>> print(sample_XYZ_d50.coordinates)
[9.893278796703505, 5.7763373686957005, 16.63362389551362]
>>> print(sample_XYZ_d50.illuminant)
Illuminant object: CIE D50 standard illuminant
>>> print(sample_XYZ_d50.observer)
2º standard observer (CIE 1931)
>>> print(sample_XYZ_d50.get_sample())
{'Sample_1': [9.893278796703505, 5.7763373686957, 16.63362389551362]}
```

CIEXYZ.to_user_illuminant(Xn2, Yn2, Zn2, cat_model=“Sharp”)

Parameters:

Xn2, Yn2, Zn2: `float` *Target illuminant White Point*

cat_model: `str` *CAT method. Default: “Sharp”*

*Returns:

X2, Y2, Z2: `float` *XYZ referred to the new illuminant*

Also, it is possible to apply a CAT transform using a user illuminant white point (X_{n2}, Y_{n2}, Z_{n2}).

The method returns the XYZ coordinates under the new illuminant.

```
>>> X2, Y2, Z2 = sample_XYZ.to_user_illuminant(Xn2=91.4, Yn2=102.3, Zn2=37.8,
    cat_model="Bradford")
>>> print(f"The XYZ values in the new illuminant are: {X2}, {Y2}, {Z2}")
The XYZ values in the new illuminant are: 8.233072778635329, 5.34612439356663,
7.078762278967476
```

__str__:

> **__str__**
>
> *Returns the string representation of the object. This method is called when the `print()` or `str()` function is invoked on an object.*

```
>>> print(sample_XYZ) # str method
CIEXYZ object: Sample_1 : X =10.234, Y=5.871, Z=22.109
```

4.2 CIExyY

> ***class CIExyY***
>
> *CIExyY(name_id, x, y, Y, cie_illuminant="D65", observer=2)*
>
> *.to_XYZ()*
>
> *.to_uvY()*
>
> *.to_LAB()*
>
> *.to_LCHab()*
>
> *.to_LUV()*
>
> *.to_LCHuv()*
>
> *.to_RGB()*
>
> *.plot(show_figure, save_figure, output_path)*

The *CIExyY* class represents a colour object in the CIE xyY colour space.

For some practical applications it is convenient to use a two-dimensional representation of the colours in such a way that their components are separated into luminance and chroma. The CIE xyY colour space was designed to achieve this goal.

The xyY colour space expresses the CIE XYZ tristimulus values in terms of x,y chromaticity coordinates. The x,y chromaticity coordinates are computed as a normalization of XYZ tristimulus values. Thus, the x,y chromaticity coordinates represent the relative amounts of the tristimulus values (Hunt and Pointer, 2011). The CIE Y value (or luminance) is added for practical purposes.

Info

CIE 015:2018. 7.3. Calculation of chromaticity coordinates (p.26). (CIE, 2018)

4.2.1 Create an instance

CIExyY(name_id, x, y, Y, cie_illuminant="D65", observer=2)

Parameters:

name_id: `str` *Sample ID.*

x: `float` *x chromaticity coordinate.*

y: `float` *y chromaticity coordinate.*

Y: `float` *Y tristimulus value.*

cie_illuminant: `str`, `Illuminant` *CIE Illuminant. Default: "D65".*

observer: `str`, `int`, `Observer` *CIE standard observer. Default: 2.*

To *create an instance* of the *CIExyY* class, simply enter the required parameters as follows:

```
>>> from coolpi.classes.cie_colour_spectral import CIExyY
>>> sample_xyY = CIExyY(name_id="Sample_1", x=0.26781, y=0.15364, Y=5.871,
    cie_illuminant="D65", observer=2)
```

An *Illuminant* instance can also be passed as a parameter:

```
>>> from coolpi.colour.cie_colour_spectral import Illuminant
>>> D65 = Illuminant("D65")
>>> sample_xyY = CIExyY(name_id="Sample_1", x=0.26781, y=0.15364, Y=5.871,
    cie_illuminant=D65, observer=2)
```

The colour sample should be referred to a valid CIE standard illuminant and observer:

```
>>> sample_xyY = CIExyY(name_id="Sample_1", x=0.26781, y=0.15364, Y=5.871,
    cie_illuminant="D30", observer=2)
CIEIlluminantError: The input illuminant name is not a valid CIE standard illuminant
>>> sample_xyY = CIExyY(name_id="Sample_1", x=0.26781, y=0.15364, Y=5.871,
    cie_illuminant="D65", observer=20)
CIEObserverError: The observer should be a CIE standard 1931 or 1964 observer (2º or 10º)
```

Alert

It is only allowed to create instances for a CIE standard illuminant and observer. Otherwise, a CIEIlluminantError or CIEObserverError is raised.

4.2.2 Attributes

The class *attributes* are: *type* (returns a `str` with the description of the main class), *subtype* (returns a `str` with the description of the object), *colour_space* (returns a `str` with the colour space (equivalent to `subtype`), *name_id* (returns a `str` with the sample id), *coordinates* (returns a `list` with the three coordinates as `float` of the colour sample), *illuminant* (returns an *Illuminant* instance), and *observer* (returns an *Observer* instance).

```
>>> print(sample_xyY.type)
Colour Object
>>> print(sample_xyY.colour_space()) # subtype
CIE xyY
>>> print(sample_xyY.subtype) # returns colour space
CIE xyY
>>> print(sample_xyY.name_id)
Sample_1
>>> print(sample_xyY.coordinates)
[0.26781, 0.15364, 5.871]
>>> print(sample_xyY.illuminant) # str method
Illuminant object: CIE D65 standard illuminant
>>> print(sample_xyY.observer) # str method
2º standard observer (CIE 1931)
>>> print(sample_xyY.get_sample()) # colour data as dict
2º standard observer (CIE 1931)
{'Sample_1': [0.26781, 0.15364, 5.871]}
```

4.2.3 Methods

Colour Space Conversion (CSC):

It is only necessary to execute the class method that performs the colour conversion to the output colour space, for example:

```
>>> sample_uvY = sample_xyY.to_uvY() # returns a CIEuvY class object
>>> type(sample_uvY)
coolpi.colour.cie_colour_spectral.CIEuvY
```

After applying the colour conversion, a new *Colour* object is returned.

Info

Similarly, the transformation can be applied between the other implemented colour spaces.

The alternative way to obtain the colour coordinates in the desired output colour space is:

```
>>> print("XYZ   = ", sample_xyY.to_XYZ().coordinates)
XYZ   =  [10.233744532673784, 5.871, 22.107960492059362]
>>> print("uvY   = ", sample_xyY.to_uvY().coordinates)
uvY   =  [0.24865948942215288, 0.32097046002144813, 5.871]
>>> print("LAB   = ", sample_xyY.to_LAB().coordinates)
LAB   =  [29.084647940987004, 43.54311522120996, -39.81989232614932]
>>> print("LCHab = ", sample_xyY.to_LCHab().coordinates)
LCHab =  [29.084647940987004, 59.005310846005145, 317.55728877376833]
>>> print("LUV   = ", sample_xyY.to_LUV().coordinates)
LUV   =  [29.084647940987004, 19.21982302204829, -55.72056193591771]
>>> print("LCHuv = ", sample_xyY.to_LCHuv().coordinates)
LCHuv =  [29.084647940987004, 58.942197273712985, 289.03096925944504]
```

If a conversion to RGB space is desired:

```
>>> print("sRGB     = ", sample_xyY.to_RGB(rgb_name_space = "sRGB").coordinates)
sRGB     =  [0.3975466946551014, 0.1523408468071324, 0.5141831884504302]
>>> print("AdobeRGB = ", sample_xyY.to_RGB(rgb_name_space = "Adobe").coordinates)
AdobeRGB =  [0.3484256104933206, 0.15229131747801383, 0.5051723823847176]
>>> print("AppleRGB = ", sample_xyY.to_RGB(rgb_name_space = "Apple").coordinates)
AppleRGB =  [0.38349633557812984, 0.12347918390545604, 0.5211998170812592]
```

Note:

The colour conversion between CIE xyY and the other CIE spaces is implemented as methods of the class itself. However, it is possible to apply the transformation functions between colour spaces by importing the module `colour_space_conversion`.

Please, see interactive Jupyter Notebook:

`02b_Colour_Space_Conversion.ipynb`

Info

A more detailed explanation of the formulas and methodology implemented to perform the conversion between color spaces can be found in the CSC section.

__str__:

__str__

Returns the string representation of the object. This method is called when the `print()` or `str()` function is invoked on an object.

```
>>> print(sample_xyY) # str method
CIExyY object: Sample_1 : x =0.26781, y=0.15364, Y=5.871
```

4.2.4 Plot

CIEXxyY.plot(show_figure=True, save_figure=False, output_path=None)

Method to create and display the x,y into the CIE 1931 x,y Chromaticity Diagram using Matplotlib.

Parameters:

show_figure: `bool` *If True, the plot is shown. Default:* `True`.

save_figure: `bool` *If True, the figure is saved. Default:* `False`.

output_path: `os` *path for the ouput figure. Default:* `None`.

CIExyY objects can be represented in the CIE 1931 x,y Chromaticity Diagram as follows:

```
>>> sample_xyY.plot()
```

In addition, it is posible to save the figure setting True the opcion `save_figure` and introducing a valid `output_path`.

```
>>> sample_xyY.plot(show_figure = False, save_figure = True, output_path = path_figure)
```

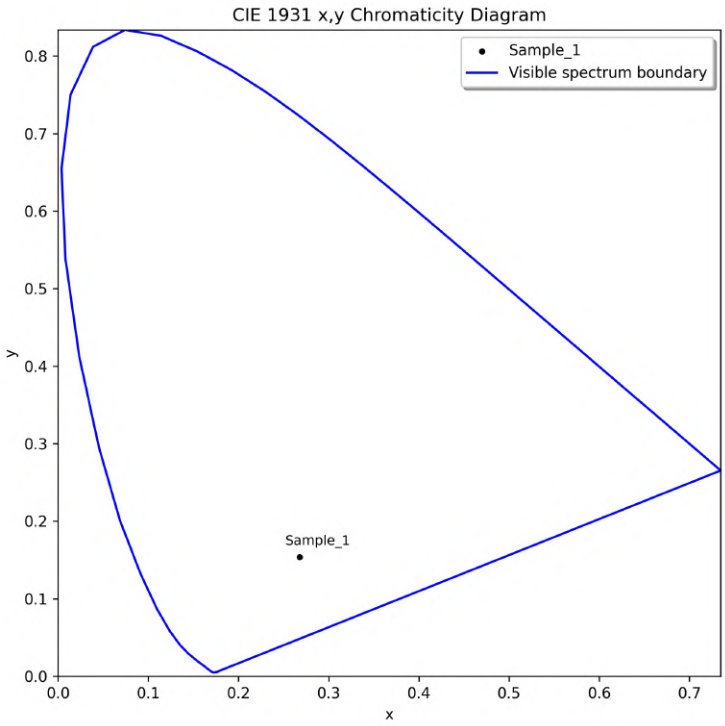

Figure 22: *CIE 1931 x,y Chromaticity Diagram*

4.3 CIEuvY

class CIEuvY

CIEuvY(name_id, u, v, Y, cie_illuminant="D65", observer=2)

.to_XYZ()

.to_xyY()

.to_LAB()

.to_LCHab()

.to_LCHuv()

.to_LUV()

.to_RGB()

The *CIEuvY* class represents a colour object in the CIE u'v' colour space.

The CIE 1976 UCS (uniform-chromaticity-scale) colour space (now obsolete) is actually a

modification of the CIE 1960 colour space based on the UCS diagram proposed by MacAdam. The CIE 1960 colour space uses new u,v chromaticity coordinates derived from the CIE 1931 x,y coordinates. The CIE 1960 was adopted not only because of its improved uniformity but for the simplicity of the transform equations (compared with the initial Judd model). The CIE 1960 UCS does not define a luminance (or lightness component), but the CIE Y is added for practical purposes (Ohta and Robertson, 2005). Although the CIE 1960 UCS is now outdated, it is still used for computations related to colour correlated temperature.

The CIE 1960 diagram was replaced by the CIE 1976 u',v' UCS.

Info

CIE 015:2018. 8.1. CIE 1976 uniform chromaticity scale (UCS) diagram (p.27). (CIE, 2018)

4.3.1 Create an instance

CIEuvY(name_id, u, v, Y, cie_illuminant="D65", observer=2)

Parameters:

name_id: `str` *Sample ID.*

u: `float` *u' chromaticity coordinate.*

v: `float` *v' chromaticity coordinate.*

Y: `float` *Y tristimulus value.*

cie_illuminant: `str`, `Illuminant` *CIE Illuminant. Default: "D65".*

observer: `str`, `int`, `Observer` *CIE standard observer. Default: 2.*

To *create an instance* of the *CIEuvY* class, simply enter the required parameters as follows:

```
>>> from coolpi.colour.cie_colour_spectral import CIEuvY
>>> sample_uvY = CIEuvY(name_id="Sample_1", u=0.24866, v=0.32096, Y=5.871,
    cie_illuminant="D65", observer=2)
```

An *Illuminant* instance can also be passed as a parameter:

```
>>> from coolpi.colour.cie_colour_spectral import Illuminant
>>> D65 = Illuminant("D65")
>>> sample_uvY = CIEuvY(name_id="Sample_1", u=0.24866, v=0.32096, Y=5.871,
    cie_illuminant=D65, observer=2)
```

The colour sample should be referred to a valid CIE standard illuminant and observer:

```
>>> sample_uvY = CIEuvY(name_id="Sample_1", u=0.24866, v=0.32096, Y=5.871,
    cie_illuminant="D30", observer=2)
CIEIlluminantError: The input illuminant name is not a valid CIE standard illuminant
>>> sample_uvY = CIEuvY(name_id="Sample_1", u=0.24866, v=0.32096, Y=5.871,
    cie_illuminant="D65", observer=20)
CIEObserverError: The observer should be a CIE standard 1931 or 1964 observer (2º or 10º)
```

Alert

It is only allowed to create instances for a CIE standard illuminant and observer. Otherwise, a CIEIlluminantError or CIEObserverError is raised.

4.3.2 Attributes

The class *attributes* are: *type* (returns a `str` with the description of the main class), *subtype* (returns a `str` with the description of the object), *colour_space* (returns a `str` with the colour space (equivalent to `subtype`), *name_id* (returns a `str` with the sample id), *coordinates* (returns a `list` with the three coordinates as `float` of the colour sample), *illuminant* (returns an *Illuminant* instance), and *observer* (returns an *Observer* instance).

```
>>> print(sample_uvY.type)
Colour Object
>>> print(sample_uvY.colour_space()) # subtype
CIE u'v'Y
>>> print(sample_uvY.subtype) # returns color space
CIE u'v'Y
>>> print(sample_uvY.name_id)
Sample_1
>>> print(sample_uvY.coordinates)
[0.24866, 0.32096, 5.871]
>>> print(sample_uvY.illuminant) # str method
Illuminant object: CIE D65 standard illuminant
>>> print(sample_uvY.observer) # str method
2º standard observer (CIE 1931)
>>> print(sample_uvY.get_sample()) # color data as dict
{'Sample_1': [0.24866, 0.32096, 5.871]}
```

4.3.3 Methods

Colour Space Conversion (CSC):

It is only necessary to execute the class method that performs the colour conversion to the output colour space, for example:

```
>>> sample_LAB = sample_uvY.to_LAB() # returns a CIELAB class object
>>> type(sample_LAB)
coolpi.colour.cie_colour_spectral.CIELAB
```

After applying the colour conversion, a new *Colour* object is returned.

Info

Similarly, the transformation can be applied between the other implemented colour spaces.

The alternative way to obtain the colour coordinates in the desired output colour space is:

```
>>> print("XYZ   = ", sample_uvY.to_XYZ().coordinates)
XYZ   =  [10.234099062188434, 5.871, 22.109630654910273]
>>> print("xyY   = ", sample_uvY.to_xyY().coordinates)
xyY   =  [0.26780508819376303, 0.15363185984730635, 5.871]
>>> print("LAB   = ", sample_uvY.to_LAB().coordinates)
LAB   =  [29.084647940987004, 43.5458620880538, -39.82285243808329]
>>> print("LCHab = ", sample_uvY.to_LCHab().coordinates)
LCHab =  [29.084647940987004, 59.00933554359986, 317.55696768515674]
>>> print("LUV   = ", sample_uvY.to_LUV().coordinates)
LUV   =  [29.084647940987004, 19.220016071748365, -55.72451687445427]
>>> print("LCHuv = ", sample_uvY.to_LCHuv().coordinates)
LCHuv =  [29.084647940987004, 58.94599900493335, 289.02989312888906]
```

If a conversion to RGB space is desired:

```
>>> print("sRGB     = ", sample_uvY.to_RGB(rgb_name_space = "sRGB").coordinates)
sRGB     =  [0.3975512398171586, 0.15232908950474702, 0.5142018048920176]
>>> print("AdobeRGB = ", sample_uvY.to_RGB(rgb_name_space = "Adobe").coordinates)
AdobeRGB =  [0.34842811042674693, 0.15227955537015053, 0.5051905145986682]
>>> print("AppleRGB = ", sample_uvY.to_RGB(rgb_name_space = "Apple").coordinates)
AppleRGB =  [0.383500167319504, 0.12346212215521132, 0.5212188124193071]
```

Note:

The colour conversion between CIE uvY and the other CIE spaces is implemented as methods of the class itself. However, it is possible to apply the transformation functions between colour spaces by importing the module `colour_space_conversion`.

Please, see interactive Jupyter Notebook:

02b_Colour_Space_Conversion.ipynb

Info

A more detailed explanation of the formulas and methodology implemented to perform the conversion between color spaces can be found in the CSC section.

__str__:

__str__

Returns the string representation of the object. This method is called when the `print()` or `str()` function is invoked on an object.

```
>>> print(sample_uvY)
CIEuvY object: Sample_1 : u' =0.24866, v'=0.32096, Y=5.871
```

4.4 CIELAB

class CIELAB

CIELAB(name_id, L, a, b, cie_illuminant="D65", observer=2)

.colour_space()

.to_XYZ()

.to_xyY()

.to_uvY()

.to_LCHab()

.to_LCHuv()

.to_LUV()

.to_RGB()

.plot(show_figure, save_figure, output_path)

.delta_e_ab(target)

.CIEDE2000(target, kl, kc, kh)

The *CIELAB* class represents a colour object in the CIELAB colour space.

The CIE XYZ and CIE xyY color spaces are not visually uniform. Thus, equal distances between colour stimuli calculated in these colour spaces do not necessarily reflect equal perceived differences. To address this shortcoming, the CIE developed in 1976 two colour spaces intended to be perceptually uniform, the CIELAB and CIELUV.

The CIE 1976 $L^*a^*b^*$ colour space provides a three-dimensional colour space where the $a^* - b^*$ axes form a plane to which the L^* axis is orthogonal. It separates the colour into lightness (L^*) and chromatic information (a^*, b^*) in two red/green (a^*) and yellow/blue (b^*) axes. The L^* defines black at 0 and diffuse white at 100; negative a^* values indicate green and positive values indicate red; and negative b^* values indicate blue and positive values indicate yellow.

Since the CIELAB colour space is approximately uniform, it is together with the CIELUV the colour space used to calculate the colour difference metrics (or ΔE^*_{ab}) between colour samples in classical colorimetry.

The $L^*a^*b^*$ coordinates can be obtained from XYZ tristimulus values using the well-established formulation provided by the CIE (CIE, 2018).

Info

CIE 015:2018. 8.2.1. CIE 1976 $L^*a^*b^*$ colour space; CIELAB colour space (pp.28-30) (CIE, 2018).

4.4.1 Create an instance

CIELAB(name_id, L, a, b, cie_illuminant="D65", observer=2)

Parameters:

name_id: `str` *Sample ID.*

L: `float` *L coordinate.*

a: `float` *a coordinate.*

b: `float` *b coordinate.*

cie_illuminant: `str`, `Illuminant` *CIE Illuminant. Default: "D65".*

observer: `str`, `int`, `Observer` *CIE standard observer. Default: 2.*

To *create an instance* of the *CIELAB* class, simply enter the required parameters as follows:

```
>>> from coolpi.classes.cie_colour_spectral import CIELAB
>>> sample_LAB = CIELAB(name_id="Sample_1", L=29.08465, a=43.545095, b=-39.821735,
    cie_illuminant="D65", observer=2)
```

An *Illuminant* instance can also be passed as a parameter:

```
>>> from coolpi.colour.cie_colour_spectral import Illuminant
>>> D65 = Illuminant("D65")
>>> sample_LAB = CIELAB(name_id="Sample_1", L=29.08465, a=43.545095, b=-39.821735,
    cie_illuminant=D65, observer=2)
```

The colour sample should be referred to a valid CIE standard illuminant and observer:

```
>>> sample_LAB = CIELAB(name_id="Sample_1", L=29.08465, a=43.545095, b=-39.821735,
    cie_illuminant="D30", observer=2)
CIEIlluminantError: The input illuminant name is not a valid CIE standard illuminant
>>> sample_LAB = CIELAB(name_id="Sample_1", L=29.08465, a=43.545095, b=-39.821735,
    cie_illuminant="D65", observer=20)
CIEObserverError: The observer should be a CIE standard 1931 or 1964 observer (2º or 10º)
```

⚠ **Alert**

It is only allowed to create instances for a CIE standard illuminant and observer. Otherwise, a CIEIlluminantError or CIEObserverError is raised.

4.4.2 Attributes

The class *attributes* are: *type* (returns a `str` with the description of the main class), *subtype* (returns a `str` with the description of the object), *colour_space* (returns a `str` with the colour space (equivalent to `subtype`), *name_id* (returns a `str` with the sample id), *coordinates* (returns a `list` with the three coordinates as `float` of the colour sample), *illuminant* (returns an *Illuminant* instance), and *observer* (returns an *Observer* instance).

```
>>> print(sample_LAB.type)
Colour Object
>>> print(sample_LAB.colour_space()) # subtype
CIE LAB
>>> print(sample_LAB.subtype) # returns color space
CIE LAB
>>> print(sample_LAB.name_id)
Sample_1
>>> print(sample_LAB.coordinates)
[29.08465, 43.545095, -39.821735]
>>> print(sample_LAB.illuminant) # str method
Illuminant object: CIE D65 standard illuminant
>>> print(sample_LAB.observer) # str method
2º standard observer (CIE 1931)
>>> print(sample_LAB.get_sample()) # color data as dict
{'sample_1': [29.08465, 43.55509, -39.82173]}
```

4.4.3 Methods

Colour Space Conversion (CSC):

It is only necessary to execute the class method that performs the colour conversion to the output colour space, for example:

```
>>> sample_LCHab = sample_LAB.to_LCHab() # returns a CIELCHab class object
>>> type(sample_LCHab)
coolpi.colour.cie_colour_spectral.CIELCHab
```

After applying the colour conversion, a new *Colour* object is returned.

> **Info**
>
> *Similarly, the transformation can be applied between the other implemented colour spaces.*

The alternative way to obtain the colour coordinates in the desired output colour space is:

```
>>> print("XYZ   = ", sample_LAB.to_XYZ().coordinates)
XYZ   =  [10.234001201189463, 5.871000804384624, 22.109002163921144]
>>> print("xyY   = ", sample_LAB.to_xyY().coordinates)
xyY   =  [0.2678076119895039, 0.1536347978176864, 5.871000804384624]
>>> print("uvY   = ", sample_LAB.to_uvY().coordinates)
uvY   =  [0.24866059973237598, 0.3209638872208844, 5.871000804384624]
>>> print("LCHab = ", sample_LAB.to_LCHab().coordinates)
LCHab =  [29.08465, 59.008015362061194, 317.5572657161243]
>>> print("LUV   = ", sample_LAB.to_LUV().coordinates)
LUV   =  [29.084649999999996, 19.220244191487748, -55.72305105943939]
>>> print("LCHuv = ", sample_LAB.to_LCHuv().coordinates)
LCHuv =  [29.084649999999996, 58.94468768390675, 289.03056732468815]
```

If a conversion to RGB space is desired:

```
>>> print("sRGB     = ", sample_LAB.to_RGB(rgb_name_space = "sRGB").coordinates)
sRGB     =  [0.3975511679443915, 0.1523321016451217, 0.5141948184670979]
>>> print("AdobeRGB = ", sample_LAB.to_RGB(rgb_name_space = "Adobe").coordinates)
AdobeRGB =  [0.34842838757648475, 0.15228256833385487, 0.5051836955284814]
>>> print("AppleRGB = ", sample_LAB.to_RGB(rgb_name_space = "Apple").coordinates)
AppleRGB =  [0.3835002864652972, 0.12346658709318589, 0.521211693412172]
```

Note:

The colour conversion between CIELAB and the other CIE spaces is implemented as methods of the class itself. However, it is possible to apply the transformation functions between colour spaces by importing the module `colour_space_conversion`.

Please, see interactive Jupyter Notebook:

`02b_Colour_Space_Conversion.ipynb`

Info

A more detailed explanation of the formulas and methodology implemented to perform the conversion between color spaces can be found in the CSC section.

Colour difference:

CIELAB.delta_e_ab(target)

Method to compute the CIE76 ΔE_{ab} colour difference between two colour samples in CIELAB units.

Parameters:

target: `CIELAB` *CIELAB target sample.*

Returns:

delta_e_ab: `float` ΔE_{ab} *in CIELAB units.*

CIELAB.CIEDE2000(target, kl=1, kc=1, kh=1)

Method to compute the CIEDE2000 colour difference between two colour samples in CIELAB coordinates.

Parameters:

target: `CIELAB` *CIELAB target sample.*

kl, kc, kh: `float` *kl, kc, kh parametric factors. Default: values set to 1.*

Returns:

AE_00: `float` *CIEDE2000 in CIELAB units.*

Two colour difference metrics have been implemented, the CIE76 (or ΔE_{ab}) and the improved CIEDE2000, following the recommendations of the CIE (CIE, 2018).

Given two *CIELAB* colour samples, it is posible to compute the ΔE_{ab} colour difference between them as follows:

```
>>> S1_LAB = CIELAB(name_id="Sample_1", L=29.084648, a=43.545095, b=-39.821735,
    cie_illuminant="D65", observer=2)
>>> S2_LAB = CIELAB(name_id="Sample_2", L=35.893646, a=15.903706, b=-29.659472,
    cie_illuminant="D65", observer=2)
>>> AE_ab = S1_LAB.delta_e_ab(S2_LAB)
>>> print(AE_ab)
30.22714721726968
>>> AE_ab = S2_LAB.delta_e_ab(S1_LAB)
>>> print(AE_ab)
30.22714721726968
```

To compute the CIEDE2000 colour difference:

```
>>> AE_00 = S1_LAB.CIEDE2000(S2_LAB)
>>> print(AE_00)
13.205811623182113
```

The *CIELAB* reference and target objects should be measured under the same illuminant:

```
>>> S3_LAB = CIELAB(name_id="Sample_3", L=35.893646, a=15.903706, b=-29.659472,
    cie_illuminant="D50", observer=2)
>>> AE_ab = S1_LAB.delta_e_ab(S3_LAB)
CIEIlluminantError: The CIELAB sample must be referred to the same illuminant D65
```

The reference and the colour target should be referred to the same colour space:

```
>>> S3_XYZ = CIEXYZ(name_id="Sample_3", X=10.345, Y=20.367, Z=36.972, cie_illuminant="D65",
observer=2)
>>> AE_ab = S1_LAB.delta_e_ab(S3_XYZ)
ClassTypeError: The input target is not a CIELAB colour object
```

⚠ **Alert**

For the calculation of the colour difference, both samples must be in the same colour space, and should be measured under the same illuminant. Otherwise, a

ClassTypeError or CIEIlluminantError is raised.

Info

Further details about the colour difference metrics implemented can be found in the Colour-difference section.

__str__:

__str__

Returns the string representation of the object. This method is called when the `print()` or `str()` function is invoked on an object.

```
>>> print(sample_LAB) # str method
CIELAB object: Sample_1 : L* =29.08465, a*=43.545095, b*=-39.821735
```

4.4.4 Plot

CIELAB.plot(show_figure=True, save_figure=False, output_path=None)

*Method to create and display the L*a*b into the CIELAB diagram using Matplotlib.*

Parameters:

show_figure: `bool` *If True, the plot is shown. Default:* `True`.

save_figure: `bool` *If True, the figure is saved. Default:* `False`.

output_path: `os` *path for the ouput figure. Default:* `None`.

CIELAB objects can be represented as follows:

```
>>> sample_LAB.plot()
```

In addition, it is posible to save the figure setting `True` the opcion `save_figure` and introducing a valid `output_path`.

```
>>> sample_LAB.plot(show_figure = False, save_figure = True, output_path = path_figure)
```

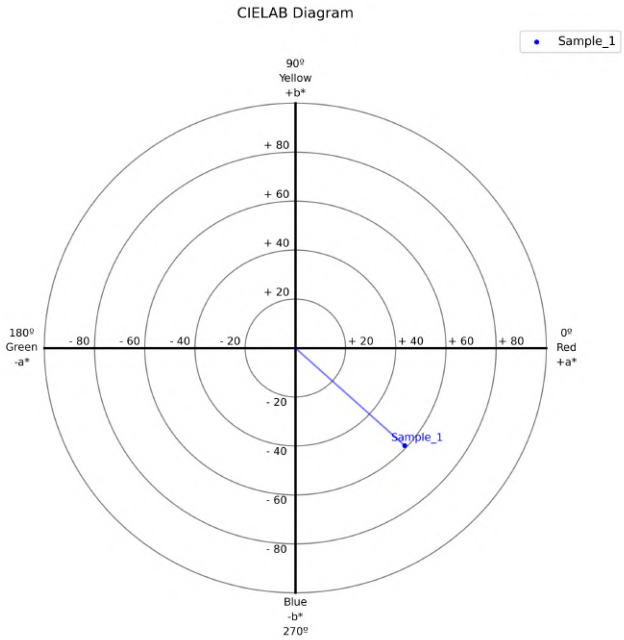

Figure 23: *CIELAB Diagram*

4.5 CIELCHab

class CIELCHab

CIELCHab(name_id, L, Cab, Hab, cie_illuminant="D65", observer=2)

.colour_space()

.to_XYZ()

.to_xyY()

.to_LAB()

.to_LUV()

.to_LCHuv()

.to_RGB()

.delta_e_ab(target)

.hue_difference(target)

The *CIELCHab* class represents a colour object in the CIE LCHab colour space.

The CIELAB colour space can be defined not only in terms of Cartesian coordinates but also in polar coordinates.

The CIE LCHab lightness (L^*), chroma (C^*_{ab}) and hue angle (h_{ab}) are calculated from CIELAB $L^*a^*b^*$ coordinates.

> **Info**
>
> *CIE 015:2018. 8.2.1.2 Correlates of lightness, chroma and hue. Equations 8.12 and 8.13 (p. 29) (CIE, 2018).*

4.5.1 Create an instance

> ***CIELCHab(name_id, L, Cab, Hab, cie_illuminant="D65", observer=2)***
>
> *Parameters:*
>
> *name_id:* `str` *Sample ID.*
>
> *L:* `float` *L coordinate.*
>
> *Cab:* `float` *Chroma coordinate.*
>
> *Hab:* `float` *Hue coordinate.*
>
> *cie_illuminant:* `str`, `Illuminant` *CIE Illuminant. Default: "D65".*
>
> *observer:* `str`, `int`, `Observer` *CIE standard observer. Default: 2.*

To *create an instance* of the *CIELCHab* class, simply enter the required parameters as follows:

```
>>> from coolpi.colour.cie_colour_spectral import CIELCHab
>>> sample_LCHab = CIELCHab(name_id="Sample_1", L=29.08465, Cab=59.008015, Hab=317.557266,
    cie_illuminant="D65", observer=2)
```

An *Illuminant* instance can also be passed as a parameter:

```
>>> from coolpi.colour.cie_colour_spectral import Illuminant
>>> D65 = Illuminant("D65")
>>> sample_LCHab = CIELCHab(name_id="Sample_1", L=29.08465, Cab=59.008015, Hab=317.557266,
    cie_illuminant=D65, observer=2)
```

The colour sample should be referred to a valid CIE standard illuminant and observer:

```
>>> sample_LCHab = CIELCHab(name_id="Sample_1", L=29.08465, Cab=59.008015, Hab=317.557266,
    cie_illuminant="D30", observer=2)
CIEIlluminantError: The input illuminant name is not a valid CIE standard illuminant
>>> sample_LCHab = CIELCHab(name_id="Sample_1", L=29.08465, Cab=59.008015, Hab=317.557266,
    cie_illuminant="D65", observer=20)
CIEObserverError: The observer should be a CIE standard 1931 or 1964 observer (2º or 10º)
```

Alert

It is only allowed to create instances for a CIE standard illuminant and observer. Otherwise, a CIEIlluminantError or CIEObserverError is raised.

4.5.2 Attributes

The class *attributes* are: *type* (returns a `str` with the description of the main class), *subtype* (returns a `str` with the description of the object), *colour_space* (returns a `str` with the colour space (equivalent to `subtype`), *name_id* (returns a `str` with the sample id), *coordinates* (returns a `list` with the three coordinates as `float` of the colour sample), *illuminant* (returns an *Illuminant* instance), and *observer* (returns an *Observer* instance).

```
>>> print(sample_LCHab.type)
Colour Object
>>> print(sample_LCHab.colour_space()) # subtype
CIE LCHab
>>> print(sample_LCHab.subtype) # returns color space
CIE LCHab
>>> print(sample_LCHab.name_id)
Sample_1
>>> print(sample_LCHab.coordinates)
[29.08465, 59.008015, 317.557266]
>>> print(sample_LCHab.illuminant) # str method
Illuminant object: CIE D65 standard illuminant
>>> print(sample_LCHab.observer) # str method
2º standard observer (CIE 1931)
>>> print(sample_LCHab.get_sample()) # color data as dict
{'Sample_1': [29.08465, 59.008015, 317.557266]}
```

4.5.3 Methods

Colour Space Conversion (CSC):

It is only necessary to execute the class method that performs the colour conversion to the output colour space, for example:

```
>>> sample_LUV = sample_LCHab.to_LUV() # returns a CIELUV class object
>>> type(sample_LUV)
coolpi.colour.cie_colour_spectral.CIELUV
```

After applying the colour conversion, a new *Colour* object is returned.

Info

Similarly, the transformation can be applied between the other implemented colour spaces.

The alternative way to obtain the colour coordinates in the desired output colour space is:

```
>>> print("XYZ   = ", sample_LCHab.to_XYZ().coordinates)
XYZ   =  [10.234001192169652, 5.871000804384624, 22.10900190432898]
>>> print("xyY   = ", sample_LCHab.to_xyY().coordinates)
xyY   =  [0.26780761363592964, 0.15363479889760845, 5.871000804384624]
>>> print("uvY   = ", sample_LCHab.to_uvY().coordinates)
uvY   =  [0.24866060070314994, 0.3209638887568151, 5.871000804384624]
>>> print("LAB   = ", sample_LCHab.to_LAB().coordinates)
LAB   =  [29.08465, 43.54509493011557, -39.821734539914864]
>>> print("LUV   = ", sample_LCHab.to_LUV().coordinates)
LUV   =  [29.084649999999996, 19.220244558537818, -55.723050478703314]
>>> print("LCHuv = ", sample_LCHab.to_LCHuv().coordinates)
LCHuv =  [29.084649999999996, 58.9446872545959, 289.03056784603535]
```

If a conversion to RGB space is desired:

```
>>> print("sRGB     = ", sample_LCHab.to_RGB(rgb_name_space = "sRGB").coordinates)
sRGB     =  [0.3975511693849829, 0.15233210155791194, 0.5141948156003442]
>>> print("AdobeRGB = ", sample_LCHab.to_RGB(rgb_name_space = "Adobe").coordinates)
AdobeRGB =  [0.3484283887766336, 0.1522825682460794, 0.505183692716826]
>>> print("AppleRGB = ", sample_LCHab.to_RGB(rgb_name_space = "Apple").coordinates)
AppleRGB =  [0.3835002879135414, 0.12346658710006109, 0.5212116905001472]
```

Note:

The colour conversion between CIE LCHab and the other CIE spaces is implemented as methods of the class itself. However, it is possible to apply the transformation functions between colour spaces by importing the module `colour_space_conversion`.

Please, see interactive Jupyter Notebook:

`02b_Colour_Space_Conversion.ipynb`

Info

A more detailed explanation of the formulas and methodology implemented to perform the conversion between color spaces can be found in the CSC section.

Colour difference:

CIELCHab.delta_e_ab(target)

Method to compute the CIE76 ΔE_{ab} colour difference between two colour samples in CIELAB units.

Parameters:

target: `CIELCHab` *CIELCHab target sample.*

Returns:

delta_e_ab: `float` ΔE_{ab} *in CIELAB units.*

CIELCHab.hue_difference(target)

Method to compute the ΔH_{ab} difference between two CIELCHab colour samples.

Parameters:

target: `CIELCHab` *CIELCHab target sample.*

Returns:

Delta_H: `float` ΔH_{ab}.

Given two *CIELCHab* colour samples, it is posible to compute the ΔE_{ab} colour difference between them as follows:

```
>>> S1_LCHab = CIELCHab(name_id="Sample_1", L=29.08465, Cab=59.008015, Hab=317.557266,
    cie_illuminant="D65", observer=2)
>>> S2_LCHab = CIELCHab(name_id="Sample_2", L=35.893646, Cab=33.654304, Hab=298.200655,
    cie_illuminant="D65", observer=2)
>>> AE_ab = S1_LCHab.delta_e_ab(S2_LCHab)
>>> print(AE_ab)
30.22714623748055
```

To compute the ΔH_{ab} difference between the two CIE LCHab colour samples:

```
>>> AH_ab = S1_LCHab.hue_difference(S2_LCHab)
>>> print(AH_ab)
-14.98356705402615
```

The *CIELCHab* reference and target objects should be measured under the same illuminant:

```
>>> S3_LCHab = CIELCHab(name_id="Sample_3", L=35.893646, Cab=33.654304, Hab=298.200655,
    cie_illuminant="D50", observer=2)
>>> AE_ab = S1_LCHab.delta_e_ab(S3_LCHab)
CIEIlluminantError: The CIELCHab sample must be referred to the same illuminant D65
```

The reference and the colour target should be referred to the same colour space:

```
>>> S3_XYZ = CIEXYZ(name_id="Sample_3", X=10.345, Y=20.367, Z=36.972, cie_illuminant="D65",
observer=2)
>>> AE_ab = S1_LCHab.delta_e_ab(S3_XYZ)
ClassTypeError: The input target is not a CIELAB colour object
```

Alert

For the calculation of the colour difference, both samples must be in the same colour space, and should be measured under the same illuminant. Otherwise, a ClassTypeError or CIEIlluminantError is raised.

Info

Further details about the colour difference metrics implemented can be found in the Colour-Difference section.

`__str__`:

`__str__`

Returns the string representation of the object. This method is called when the `print()` or `str()` function is invoked on an object.

```
>>> print(sample_LCHab) # str method
CIELCHab object: Sample_1 : L* =29.08465, C*ab=59.008015, h*ab=317.557266
```

4.6 CIELUV

class CIELUV

CIELUV(name_id, L, U, V, cie_illuminant="D65", observer=2)

.to_XYZ()

.to_xyY()

.to_uvY()

.to_LAB()

.to_LCHab()

.to_LCHuv()

.to_RGB()

.get_saturation()

.delta_e_uv(target)

The *CIELUV* class represents a colour object in the CIELUV colour space.

The CIE $L^*u^*v^*$ (or simply CIELUV) is a three-dimensional, approximately uniform colour space developed in 1976 by the CIE. The CIELUV coordinates can be calculated from the tristimulus values XYZ or the x,y chromaticity coordinates through a non-linear transformation (CIE, 2018).

Since the CIELUV colour space is approximately uniform, it is together with CIELAB the colour space used to calculate the colour difference metrics (or ΔE^*_{ab}) between colour samples in classical colorimetry.

Info

CIE 015:2018. 8.2.2. CIE 1976 $L^*u^*v^*$ colour space; CIELUV colour space (p. 30) (CIE, 2018).

4.6.1 Create an instance

CIELUV(name_id, L, U, V, cie_illuminant="D65", observer=2)

Parameters:

name_id: `str` *Sample ID.*

L: `float` *L coordinate.*

U: `float` u^* *coordinate.*

V: `float` v^* *coordinate.*

cie_illuminant: `str`, `Illuminant` *CIE Illuminant. Default: "D65".*

observer: `str`, `int`, `Observer` *CIE standard observer. Default: 2.*

To *create an instance* of the *CIELUV* class, simply enter the required parameters as follows:

```
>>> from coolpi.colour.cie_colour_spectral import CIELUV
>>> sample_LUV = CIELUV(name_id="Sample_1", L=29.08465, U=19.220243, V=-55.723049,
    cie_illuminant="D65", observer=2)
```

An *Illuminant* instance can also be passed as a parameter:

```
>>> from coolpi.colour.cie_colour_spectral import Illuminant
>>> D65 = Illuminant("D65")
>>> sample_LUV = CIELUV(name_id="Sample_1", L=29.08465, U=19.220243, V=-55.723049,
    cie_illuminant=D65, observer=2)
```

The colour sample should be referred to a valid CIE standard illuminant and observer:

```
>>> sample_LUV = CIELUV(name_id="Sample_1", L=29.08465, U=19.220243, V=-55.723049,
    cie_illuminant="D30", observer=2)
CIEIlluminantError: The input illuminant name is not a valid CIE standard illuminant
>>> sample_LUV = CIELUV(name_id="Sample_1", L=29.08465, U=19.220243, V=-55.723049,
    cie_illuminant="D65", observer=20)
CIEObserverError: The observer should be a CIE standard 1931 or 1964 observer (2º or 10º)
```


Alert

It is only allowed to create instances for a CIE standard illuminant and observer. Otherwise, a CIEIlluminantError or CIEObserverError is raised.

4.6.2 Attributes

The class *attributes* are: *type* (returns a `str` with the description of the main class), *subtype* (returns a `str` with the description of the object), *colour_space* (returns a `str` with the colour space (equivalent to `subtype`), *name_id* (returns a `str` with the sample id), *coordinates* (returns a `list` with the three coordinates as `float` of the colour sample), *illuminant* (returns an *Illuminant* instance), and *observer* (returns an *Observer* instance).

```
>>> print(sample_LUV.type)
Colour Object
>>> print(sample_LUV.colour_space()) # subtype
CIE LUV
>>> print(sample_LUV.subtype) # returns color space
CIE LUV
>>> print(sample_LUV.name_id)
Sample_1
>>> print(sample_LUV.coordinates)
[29.08465, 19.220243, -55.723049]
>>> print(sample_LUV.illuminant) # str method
Illuminant object: CIE D65 standard illuminant
>>> print(sample_LUV.observer) # str method
2º standard observer (CIE 1931)
>>> print(sample_LUV.get_sample()) # color data as dict
{'Sample_1': [29.08465, 19.220243, -55.723049]}
```

4.6.3 Methods

Colour Space Conversion (CSC):

It is only necessary to execute the class method that performs the colour conversion to the output colour space, for example:

```
>>> sample_LCHuv = sample_LUV.to_LCHuv() # returns a CIELCHuv class object
>>> type(sample_LCHuv)
coolpi.colour.cie_colour_spectral.CIELCHuv
```

After applying the colour conversion, a new *Colour* object is returned.

> **Info**
>
> *Similarly, the transformation can be applied between the other implemented colour spaces.*

The alternative way to obtain the colour coordinates in the desired output colour space is:

```
>>> print("XYZ   = ", sample_LUV.to_XYZ().coordinates)
XYZ   =  [10.23400089782265, 5.871000804384624, 22.109001333800745]
>>> print("xyY   = ", sample_LUV.to_xyY().coordinates)
xyY   =  [0.2678076119944678, 0.1536348023747352, 5.871000804384624]
>>> print("uvY   = ", sample_LUV.to_uvY().coordinates)
uvY   =  [0.2486605965811294, 0.3209638926676893, 5.871000804384624]
>>> print("LAB   = ", sample_LUV.to_LAB().coordinates)
LAB   =  [29.084649999999996, 43.54509264954989, -39.82173352874585]
>>> print("LCHab = ", sample_LUV.to_LCHab().coordinates)
LCHab =  [29.084649999999996, 59.008012634661114, 317.55726523015403]
>>> print("LCHuv = ", sample_LUV.to_LCHuv().coordinates)
LCHuv =  [29.08465, 58.9446853485151, 289.0305668825717]
```

If a conversion to RGB space is desired:

```
>>> print("sRGB     = ", sample_LUV.to_RGB(rgb_name_space = "sRGB").coordinates)
sRGB     =  [0.3975511597612691, 0.15233211277188624, 0.5141948091403663]
>>> print("AdobeRGB = ", sample_LUV.to_RGB(rgb_name_space = "Adobe").coordinates)
AdobeRGB =  [0.34842838195041914, 0.15228257946662999, 0.5051836864980597]
>>> print("AppleRGB = ", sample_LUV.to_RGB(rgb_name_space = "Apple").coordinates)
AppleRGB =  [0.38350027892160043, 0.12346660287196215, 0.5212116838595396]
```

Note:

The colour conversion between CIELUV and the other CIE spaces is implemented as methods of the class itself. However, it is possible to apply the transformation functions between colour spaces by importing the module `colour_space_conversion`.

Please, see interactive Jupyter Notebook:

`02b_Colour_Space_Conversion.ipynb`

Info

A more detailed explanation of the formulas and methodology implemented to perform the conversion between color spaces can be found in the CSC section.

Colour difference:

CIELUV.get_saturation()

Method to compute the CIELUV s_{uv} saturation.

Returns:

s_uv: `float` *s_{uv} saturation.*

CIELUV.delta_e_uv(target)

Method to compute the CIE76 ΔE_{uv} colour difference between two colour samples in CIELUV units.

Parameters:

target: `CIELUV` *CIELUV target sample.*

Returns:

delta_e_uv: `float` ΔE_{uv} *in CIELUV units.*

To compute the s_{uv} (saturation) of a given *CIELUV* colour sample:

```
>>> s_uv = sample_LUV.get_saturation()
>>> print(s_uv)
2.0266596073363474
```

Given two *CIELUV* colour objects, it is posible to compute the ΔE_{uv} colour difference between them as follows:

```
>>> S1_LUV = CIELUV(name_id="S1", L=29.08465, U=19.220243, V=-55.723049,
    cie_illuminant="D65", observer=2)
>>> S2_LUV = CIELUV(name_id="S2", L=35.893646, U=-1.021978, V=-42.663638,
    cie_illuminant="D65", observer=2)
>>> AE_uv = S1_LUV.delta_e_uv(S2_LUV)
>>> print(AE_uv)
25.033141097508683
>>> AE_uv = S2_LUV.delta_e_uv(S1_LUV)
>>> print(AE_uv)
25.033141097508683
```

The *CIELUV* reference and target objects should be measured under the same illuminant:

```
>>> S3_LUV = CIELUV(name_id="Sample_3", L=35.893646, U=-1.021978, V=-42.663638,
    cie_illuminant="D50", observer=2)
>>> AE_ab = S1_LUV.delta_e_uv(S3_LUV)
CIEIlluminantError: The CIELUV sample must be referred to the same illuminant D65
```

The reference and the colour target should be referred to the same colour space:

```
>>> S3_XYZ = CIEXYZ(name_id="Sample_3", X=10.345, Y=20.367, Z=36.972, cie_illuminant="D65",
observer=2)
>>> AE_av = S1_LUV.delta_e_av(S3_XYZ)
ClassTypeError: The input target is not a CIELAB colour object
```

Alert

For the calculation of the colour difference, both samples must be in the same colour space, and should be measured under the same illuminant. Otherwise, a ClassTypeError or CIEIlluminantError is raised.

Info

Further details about the colour difference metrics implemented can be found in the Colour-Difference section.

__str__:

> **__str__**
>
> *Returns the string representation of the object. This method is called when the `print()` or `str()` function is invoked on an object.*

```
>>> print(sample_LUV) # str method
CIELUV object: Sample_1 : L* =29.08465, u*=19.220243, v*=-55.723049
```

4.7 CIELCHuv

> ***class CIELCHuv***
>
> *CIELCHuv(name_id, L, Cuv, Huv, cie_illuminant="D65", observer=2)*
>
> *.to_XYZ()*
>
> *.to_xyY()*
>
> *.to_uvY()*
>
> *.to_LAB()*
>
> *.to_LCHab()*
>
> *.to_LUV()*
>
> *.to_RGB()*
>
> *.delta_e_uv(target)*
>
> *.hue_difference(target)*

The *CIELCHuv* class represents a colour object in the CIE LCHuv colour space.

The CIELUV colour space can be defined not only in terms of Cartesian coordinates but also in polar coordinates.

The CIELCHuv lightness (L^*), chroma (C^*_{uv}) and hue angle (h_{uv}) are calculated from CIELUV $L^*u^*v^*$ coordinates.

> **Info**

CIE 015:2018. 8.2.2.2 Correlates of lightness, saturation, chroma and hue. Equations 8.31 to 8.33 (p. 31) (CIE, 2018).

4.7.1 Create an instance

CIELCHuv(name_id, L, Cuv, Huv, cie_illuminante="D65", observer=2)

Parameters:

name_id: `str` *Sample ID.*

L: `float` *L coordinate.*

Cuv: `float` *Chroma coordinate.*

Huv: `float` *Hue coordinate.*

cie_illuminant: `str`, `Illuminant` *CIE Illuminant. Default: "D65".*

observer: `str`, `int`, `Observer` *CIE standard observer. Default: 2.*

To *create an instance* of the *CIELCHuv* class, simply enter the required parameters as follows:

```
>>> from coolpi.colour.cie_colour_spectral import CIELCHuv
>>> sample_LCHuv = CIELCHuv(name_id="Sample_1", L=29.08465, Cuv=58.944686, Huv=289.030567,
    cie_illuminant="D65", observer=2)
```

An *Illuminant* instance can also be passed as a parameter:

```
>>> from coolpi.colour.cie_colour_spectral import Illuminant
>>> D65 = Illuminant("D65")
>>> sample_LCHuv = CIELCHuv(name_id="Sample_1", L=29.08465, Cuv=58.944686, Huv=289.030567,
    cie_illuminant=D65, observer=2)
```

The colour sample should be referred to a valid CIE standard illuminant and observer:

```
>>> sample_LCHuv = CIELCHuv(name_id="Sample_1", L=29.08465, Cuv=58.944686, Huv=289.030567,
    cie_illuminant="D30", observer=2)
CIEIlluminantError: The input illuminant name is not a valid CIE standard illuminant
>>> sample_LCHuv = CIELCHuv(name_id="Sample_1", L=29.08465, Cuv=58.944686, Huv=289.030567,
    cie_illuminant="D65", observer=20)
CIEObserverError: The observer should be a CIE standard 1931 or 1964 observer (2º or 10º)
```

⚠ **Alert**

It is only allowed to create instances for a CIE standard illuminant and observer. Otherwise, a CIEIlluminantError or CIEObserverError is raised.

4.7.2 Attributes

The class *attributes* are: *type* (returns a `str` with the description of the main class), *subtype* (returns a `str` with the description of the object), *colour_space* (returns a `str` with the colour space (equivalent to `subtype`), *name_id* (returns a `str` with the sample id), *coordinates* (returns a `list` with the three coordinates as `float` of the colour sample), *illuminant* (returns an *Illuminant* instance), and *observer* (returns an *Observer* instance).

```
>>> print(sample_LCHuv.type)
Colour Object
>>> print(sample_LCHuv.colour_space()) # subtype
CIE LCHuv
>>> print(sample_LCHuv.subtype) # returns colour space
CIE LCHuv
>>> print(sample_LCHuv.name_id)
Sample_1
>>> print(sample_LCHuv.coordinates)
[29.08465, 58.944686, 289.030567]
>>> print(sample_LCHuv.illuminant) # str method
Illuminant object: CIE D65 standard illuminant
>>> print(sample_LCHuv.observer) # str method
2º standard observer (CIE 1931)
>>> print(sample_LCHuv.get_sample()) # colour data as dict
{'Sample_1': [29.08465, 58.944686, 289.030567]}
```

4.7.3 Methods

Colour Space Conversion (CSC):

It is only necessary to execute the class method that performs the colour conversion to the output colour space, for example:

```
>>> sample_XYZ = sample_LCHuv.to_XYZ() # returns a CIEXYZ class object
>>> type(sample_XYZ)
coolpi.colour.cie_colour_spectral.CIEXYZ
```

After applying the colour conversion, a new *Colour* object is returned.

Info

Similarly, the transformation can be applied between the other implemented colour spaces.

The alternative way to obtain the colour coordinates in the desired output colour space is:

```
>>> print("XYZ   = ", sample_LCHuv.to_XYZ().coordinates)
XYZ   =  [10.234000981992285, 5.871000804384624, 22.10900156642102]
>>> print("xyY   = ", sample_LCHuv.to_xyY().coordinates)
xyY   =  [0.26780761197695807, 0.15363480110111952, 5.871000804384624]
>>> print("uvY   = ", sample_LCHuv.to_uvY().coordinates)
uvY   =  [0.248660597445017, 0.32096389114299995, 5.871000804384624]
>>> print("LAB   = ", sample_LCHuv.to_LAB().coordinates)
LAB   =  [29.084649999999996, 43.545093301686265, -39.82173394102766]
>>> print("LCHab = ", sample_LCHuv.to_LCHab().coordinates)
LCHab =  [29.084649999999996, 59.00801339413619, 317.55726536206396]
>>> print("LUV   = ", sample_LCHuv.to_LUV().coordinates)
LUV   =  [29.08465, 19.220243326636297, -55.72304957648575]
```

If a conversion to RGB space is desired:

```
>>> print("sRGB     = ", sample_LCHuv.to_RGB(rgb_name_space = "sRGB").coordinates)
sRGB     =  [0.3975511620151892, 0.15233210968884317, 0.5141948117534585]
>>> print("AdobeRGB = ", sample_LCHuv.to_RGB(rgb_name_space = "Adobe").coordinates)
AdobeRGB =  [0.3484283834979808, 0.15228257638192652, 0.5051836890284868]
>>> print("AppleRGB = ", sample_LCHuv.to_RGB(rgb_name_space = "Apple").coordinates)
AppleRGB =  [0.3835002809982222, 0.12346659849864408, 0.5212116865356863]
```

Note:

The colour conversion between CIE LCHuv and the other CIE spaces is implemented as methods of the class itself. However, it is possible to apply the transformation functions between colour spaces by importing the module `colour_space_conversion`.

Please, see interactive Jupyter Notebook:

`02b_Colour_Space_Conversion.ipynb`

Info

A more detailed explanation of the formulas and methodology implemented to perform the conversion between color spaces can be found in the CSC section.

Colour difference:

CIELCHuv.delta_e_uv(target)

Method to compute the CIE76 ΔE_{uv} colour difference between two colour samples in CIELUV units.

Parameters:

target: `CIELCHuv` *CIELCHuv target sample.*

Returns:

delta_e_uv: `float` ΔE_{uv} *in CIELUV units.*

CIELCHuv.hue_difference(target)

Method to compute the ΔH_{uv} *difference between two CIELCHuv colour samples.*

Parameters:

target: `CIELCHuv` *CIELCHuv target sample.*

Returns:

Delta_H: `float` ΔH_{uv}.

Given two *CIELCHuv* colour samples, it is posible to compute the ΔE_{uv} colour difference between them as follows:

```
>>> S1_LCHuv = CIELCHuv(name_id="Sample_1", L=29.08465, Cuv=58.944686, Huv=289.030567,
    cie_illuminant="D65", observer=2)
>>> S2_LCHuv = CIELCHuv(name_id="Sample_2", L=35.893646, Cuv=42.675878, Huv=268.627781,
    cie_illuminant="D65", observer=2)
>>> AE_uv = S1_LCHuv.delta_e_uv(S2_LCHuv)
>>> print(AE_uv)
25.033141326676905
>>> AE_uv = S2_LCHuv.delta_e_uv(S1_LCHuv)
>>> print(AE_uv)
25.033141326676905
```

To compute the ΔH_{uv} difference between the two *CIELCHuv* colour samples:

```
AH_uv = S1_LCHuv.hue_difference(S2_LCHuv)
print(AH_uv)
-17.765743001982752
```

The *CIELCHuv* reference and target objects should be measured under the same illuminant:

```
>>> S3_LCHuv = CIELCHuv(name_id="Sample_3", L=35.893646, Cuv=42.675878, Huv=268.627781,
    cie_illuminant="D50", observer=2)
>>> AE_uv = S1_LCHuv.delta_e_uv(S3_LCHuv)
CIEIlluminantError: The CIELCHuv sample must be referred to the same illuminant D65
```

The reference and the colour target should be referred to the same colour space:

```
>>> S3_XYZ = CIEXYZ(name_id="Sample_3", X=10.345, Y=20.367, Z=36.972, cie_illuminant="D65",
observer=2)
>>> AE_uv = S1_LCHuv.delta_e_av(S3_XYZ)
ClassTypeError: The input target is not a CIELAB colour object
```


Alert

For the calculation of the colour difference, both samples must be in the same colour space, and should be measured under the same illuminant. Otherwise, a ClassTypeError or CIEIlluminantError is raised.

Info

Further details about the colour difference metrics implemented can be found in the Colour-Difference section.

__str__:

__str__

Returns the string representation of the object. This method is called when the `print()` or `str()` function is invoked on an object.

```
>>> print(sample_LCHuv) # str method
CIELCHuv object: Sample_1 : L* =29.08465, C*uv=58.944686, h*uv=289.030567
```

4.8 sRGB

class sRGB

sRGB(name_id, sR, sG, sB, observer=2)

.to_XYZ()

The *sRGB* class represents a colour object in the sRGB colour space.

The sRGB colour space is the most widely used on digital devices. The sRGB or standard RGB colour space was developed by HP and Microsoft in 1996 to use on multimedia systems. The International Electrotechnical Commission standardised the sRGB colour space in 1998 (IEC 61966-2-1:1999).

sRGB uses the same color primaries and white point as ITU-R BT.709.

The *sRGB* class is implemented only for colour samples measured under the CIE D65 standard illuminant.

The transformation from sRGB values to CIE XYZ tristimulus values (or the reverse transform) are performed following the IEC formulation (IEC 61966-2-1:1999).

Info

International Electrotechnical Commission, 1999. IEC/4WD 61966-2-1: Colour Measurement and Management in Multimedia Systems and Equipment - Part 2-1: Default RGB Colour Space - sRGB (IEC, 1999).

4.8.1 Create an instance

sRGB(name_id, sR, sG, sB, observer=2)

Parameters:

name_id: `str` *Sample ID.*

sR: `float` *red coordinate in range [0-1].*

sG: `float` *green coordinate in range [0-1].*

sB: `float` *blue coordinate in range [0-1].*

observer: `str`, `int`, `Observer` *CIE standard observer. Default: 2.*

To *create an instance* of the *sRGB* class, simply enter the required parameters as follows:

```
>>> from coolpi.colour.cie_colour_spectral import sRGB
>>> sample_sRGB = sRGB(name_id="Sample_1", sR=0.397551, sG=0.152332, sB=0.514194,
observer=2)
```

The *sRGB* sample should be referred to a valid CIE standard observer:

```
>>> sample_sRGB = sRGB(name_id="Sample_1", sR=0.397551, sG=0.152332, sB=0.514194,
observer=20)
CIEObserverError: The observer should be a CIE standard 1931 or 1964 observer (2º or 10º)
```

4.8.2 Attributes

The class *attributes* are: *type* (returns a `str` with the description of the main class), *subtype*

(returns a `str` with the description of the object), *colour_space* (returns a `str` with the colour space (equivalent to `subtype`), *name_id* (returns a `str` with the sample id), *coordinates* (returns a `list` with the three coordinates as `float` of the colour sample), *illuminant* (returns an *Illuminant* instance), and *observer* (returns an *Observer* instance).

```
>>> print(sample_sRGB.type)
Colour Object
>>> print(sample_sRGB.colour_space()) # subtype
sRGB
>>> print(sample_sRGB.subtype) # returns colour space
sRGB
>>> print(sample_sRGB.name_id)
Sample_1
>>> print(sample_sRGB.coordinates)
[0.397551, 0.152332, 0.514194]
>>> print(sample_sRGB.illuminant)
Illuminant object: CIE D65 standard illuminant
>>> print(sample_sRGB.observer)  # str method
2º standard observer (CIE 1931)
>>> print(sample_sRGB.get_sample()) # colour data as dict
{'Sample_1': [0.397551, 0.152332, 0.514194]}
```

4.8.3 Methods

Colour Space Conversion (CSC):

Only the conversion between the sRGB and CIE XYZ colour space is implemented:

```
>>> sample_XYZ = sample_sRGB.to_XYZ() # returns a CIEXYZ class object
>>> type(sample_XYZ)
coolpi.colour.cie_colour_spectral.CIEXYZ
```

After applying the colour conversion, a new *Colour* object is returned.

To get the CIE XYZ coordinates after the transform:

```
>>> sprint("XYZ   = ", sample_sRGB.to_XYZ().coordinates)
XYZ   =  [10.234239815225408, 5.871220041030704, 22.109061465530786]
```

Note:

Please, see interactive Jupyter Notebook:

02b_Colour_Space_Conversion.ipynb

Info

A more detailed explanation of the formulas and methodology implemented to

perform the conversion between color spaces can be found in the CSC section.

__str__:

> **__str__**
>
> *Returns the string representation of the object. This method is called when the `print()` or `str()` function is invoked on an object.*

```
>>> print(sample_sRGB) # str method
sRGB object: Sample_1 : sR =0.397551, sG=0.152332, sB=0.514194
```

5 Spectral

The coolpi package includes classes for specific objects with spectral properties based on the *Spectral* abstract class, such as illuminant and spectral colour.

The main *Spectral* objects are:

- *Illuminant*: Spectral power distribution (SPD) for CIE standard illuminants.
- *IlluminantFromCCT*: SPD of illuminant computed from colour correlated temperature (CTT or Tcp measured in ºK).
- *MeasuredIlluminant*: SPD measured with a spectrometer or similar specific instruments for light measurements.
- *SpectralColour*: Spectral data for colour samples.
- *Reflectance*: Reflectance data measured using a spectrophotometer.

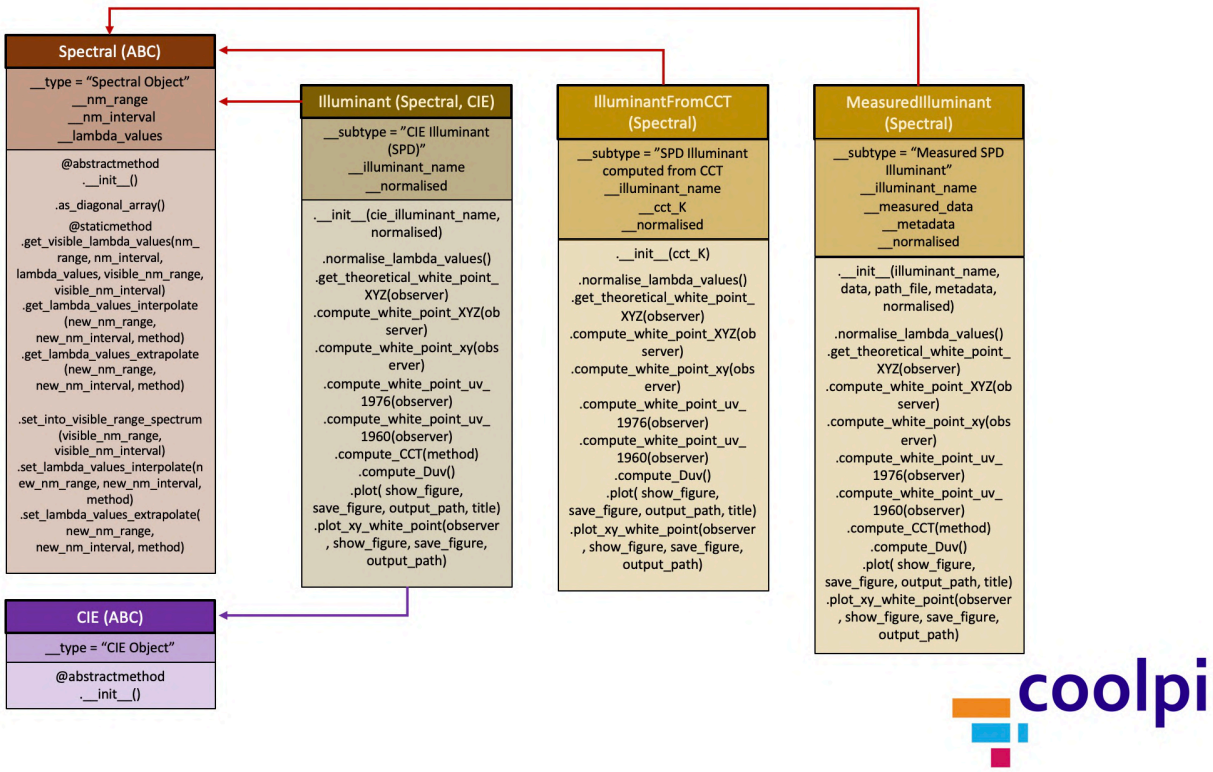

Figure 24: *UML Diagram for the Spectral Illuminant classes*

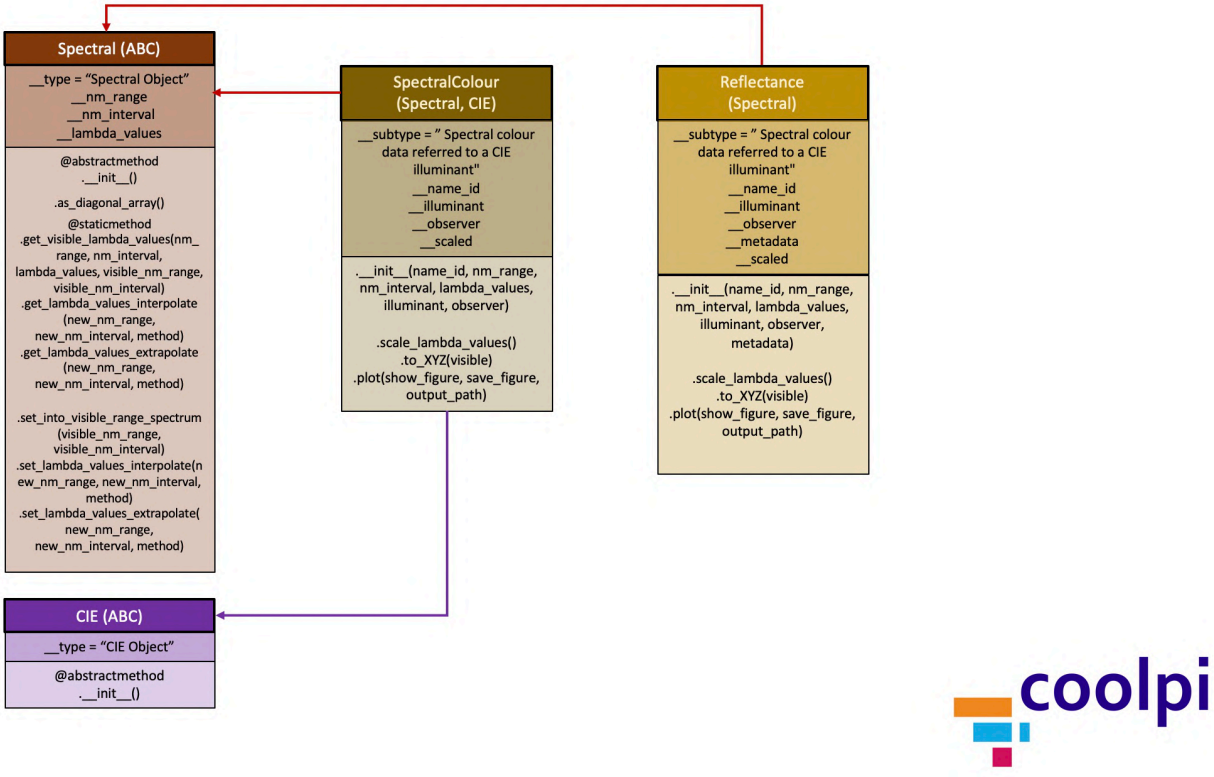

Figure 25: *UML Diagram for the Spectral Colour classes*

The spectral power distribution of a given illuminant can be converted into XYZ tristimulus values. The set of the tristimulus coordinates of an illuminant is known as white point (usually denoted as X_n, Y_n, Z_n). Thus, coolpi includes a *WhitePoint* class to support coolpi illuminant classes.

Practical use of *Spectral* classes

Users are encouraged to previously take a look at the Jupyter Notebook:

`03_Spectral_objects.ipynb`

5.1 Illuminant

class Illuminant

Illuminant(cie_illuminant_name, normalised=False))

.as_diagonal_array()

.set_into_visible_range_spectrum(visible_nm_range, visible_nm_interval)()

.normalise_lambda_values()

.get_theoretical_white_point_XYZ(observer)

.compute_white_point_XYZ(observer)

.compute_white_point_xy(observer)

.compute_white_point_uv_1976(observer)

.compute_white_point_uv_1960(observer)

.compute_CCT(method)

.compute_Duv()

.plot(show_figure, save_figure, output_path)

.plot_xy_white_point(observer, show_figure, save_figure, output_path)

The *Illuminant* class represents a valid CIE standard illuminant defined by its relative spectral power distribution (SPD).

The term *source* refers to a physical emitter of light, such as a lamp or the sun and the sky. The term *illuminant* refers to a specific SPD, not necessarily provided directly by a source, and not necessarily realisable as a source (CIE, 2018).

The *Illuminant* class allows to create an instance of the following CIE standard illuminants:

- A: CIE standard illuminant A is intended to represent typical, domestic, tungsten-

filament lighting. Its relative SPD corresponds aproximatelty with a CCT of 2855.5 ºK.

- C: This illuminant is intended to represent average daylight with a CCT of approximately 6800º K.
- D: Daylight illuminants: D50, D55, D65, D75.
- ID: Indoor daylight illuminants. ID50, ID65.
- F: Fluorescent lamps. Standard: FL1, FL2, FL3, FL4, FL5, FL6; Broad-band: FL7, FL8, FL9; Narrow-band: FL10, FL11, FL12; Standard halophosphate: FL3.1, FL3.2, FL3.3; DeLuxe type: FL3.4, FL3.5, FL3.6; Three-band: FL.3.7, FL.3.8, FL3.9, FL3.10, FL3.11; Multi-band: FL3.12, FL3.13, FL.3.14; D65 simulator: FL3.15.
- HP: High presssure discharge lamp illuminants. Standard high pressure sodium lamp: HP1; Colour enhanced high pressure sodium lamp; High pressure metal halide lamps: HP2, HP3, HP4, HP5.
- LED: LED lamps (Photosphor-type LEDs with different CCTs): LED-B1, LED-B2, LED-B3, LED-B4, LED-B5; Hybrid-type: LED-BH1; RGB-type: LED-RGB1; Violet-pumped phosphor-type: LED-V1, LED-V2.

Info

CIE 015:2018. 4.1.1.1 CIE standard illuminant A (p.10), 4.1.4 Illuminant C (p.14), 4.1.1.2 CIE standard illuminant D65 (p.11) (CIE, 2018).

Note

For the definition of the spectral power distribution (SPD) of CIE illuminants, the tabulated data published by the CIE were used.

CIE 015:2018. 11.2. Table 5. (p.51); Table 7 (p.56); Table 10.1 (p.59); Table 10.2 (p.61); Table 10.3 (p.63); Table 11 (p.65); Table 12.1 (p.67) and Table 12.2 (p.69) (CIE, 2018).

It is highly recommended to use the CIE standard illuminant D65, which has a relative SPD representing a phase of daylight with a CCT of approximately 6503 ºK (also called the nominal correlated colour temperature of the daylight illuminant). When D65 cannot be used, it is recommended that one of the daylight illuminants D50, D55 or D75 be used (CIE, 2018).

From a practical point of view, illuminants are most used in colorimetry to obtain the CIE XYZ tristimulus values from reflected or transmitted spectral data of colour objects under a specific light source.

5.1.1 Create an instance

Illuminant(cie_illuminant_name, normalised = `False`)

Parameters:

cie_illuminant_name: `str` *CIE Illuminant name.*

normalised: `bool`, *If True, the SPD is normalised. Default:* `False`.

To *create an instance* of the *Illuminant* class, simply enter the required parameters as follows:

```
>>> from coolpi.classes.cie_colour_spectral import Illuminant
>>> D65 = Illuminant(cie_illuminant_name="d65", normalised = False)
>>> type(D65)
coolpi.colour.cie_colour_spectral.Illuminant
```

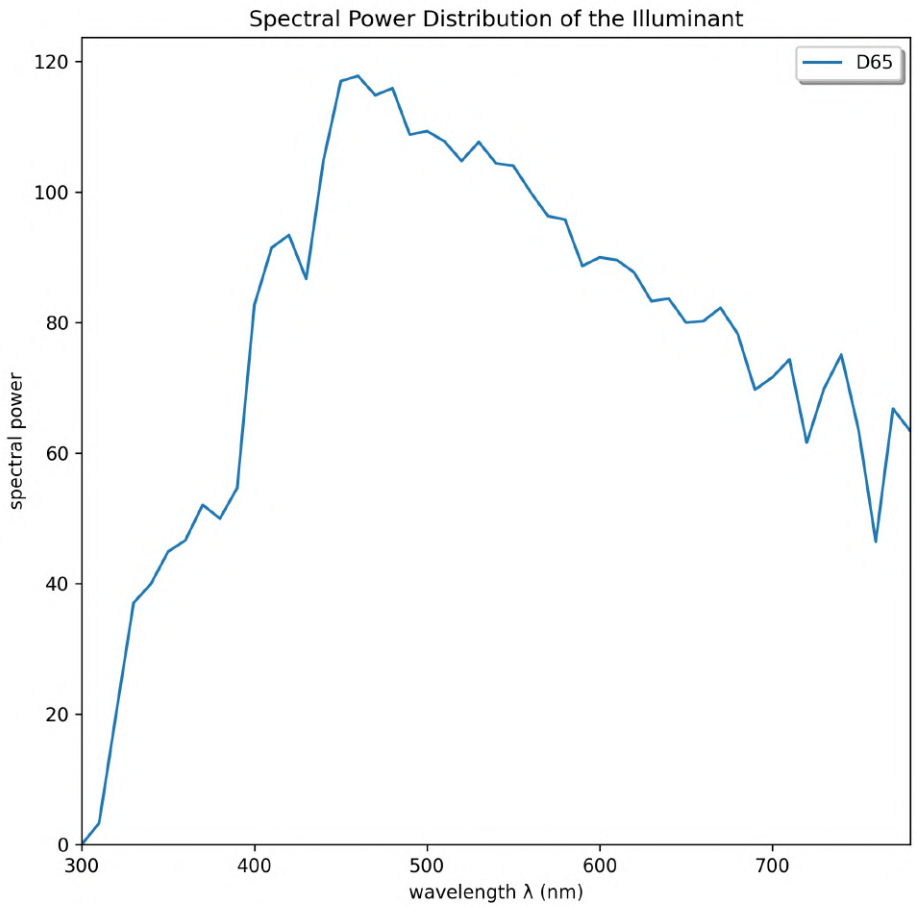

Figure 26: *Illuminant*

The *Illuminant* object should be instantiated for a valid CIE standard illuminant:

```
>>> from coolpi.classes.cie_colour_spectral import Illuminant
>>> D30 = Illuminant(cie_illuminant_name="d30", normalised = False)
CIEIlluminantError: The input illuminant name is not a valid CIE standard illuminant
```

> ⚠ **Alert**
>
> *It is only allowed to create Illuminant instances for a valid CIE standard illuminant. Otherwise, a CIEIlluminantError is raised.*

5.1.2 Attributes

The class *attributes* are: *type* (returns a `str` with the description of the main class), *subtype* (returns a `str` with the description of the object), *illuminant_name* (returns a `str` with the CIE illuminant), *nm_range* (returns a `list` with the min and max range value in nm), *nm_interval* (returns an `int` with the lambda interval in nm), *lambda_values* (returns a `list` with the illuminant SPD) and *normalised* (returns a `bool`).

```
>>> print(D65.type)
Spectral Object
>>> print(D65.subtype)
CIE Illuminant (SPD)
>>> print(D65.illuminant_name)
D65
>>> print(D65.nm_range)
[300, 780]
>>> print(D65.nm_interval)
5
>>> print(D65.lambda_values)
[0.0341, 1.6643, 3.2945, 11.7652, 20.236, 28.6447, 37.0535, 38.5011, 39.9488, 42.4302,
44.9117, 45.775, 46.6383, 49.3637, 52.0891, 51.0323, 49.9755, 52.3118, 54.6482, 68.7015,
82.7549, 87.1204, 91.486, 92.4589, 93.4318, 90.057, 86.6823, 95.7736, 104.865, 110.936,
117.008, 117.41, 117.812, 116.336, 114.861, 115.392, 115.923, 112.367, 108.811, 109.082,
109.354, 108.578, 107.802, 106.296, 104.79, 106.239, 107.689, 106.047, 104.405, 104.225,
104.046, 102.023, 100, 98.1671, 96.3342, 96.0611, 95.788, 92.2368, 88.6856, 89.3459,
90.0062, 89.8026, 89.5991, 88.6489, 87.6987, 85.4936, 83.2886, 83.4939, 83.6992, 81.863,
80.0268, 80.1207, 80.2146, 81.2462, 82.2778, 80.281, 78.2842, 74.0027, 69.7213, 70.6652,
71.6091, 72.979, 74.349, 67.9765, 61.604, 65.7448, 69.8856, 72.4863, 75.087, 69.3398,
63.5927, 55.0054, 46.4182, 56.6118, 66.8054, 65.0941, 63.3828]
>>> print(D65.normalised)
False
```

5.1.3 Methods

As a numpy diagonal array:

> ***Illuminant.as_diagonal_array()***

Method to get the spectral data as a numpy diagonal array.

Returns:

as_diag: `numpy.ndarray` *SPD data as a diagonal array.*

To get the SPD as a diagonal array:

```
>>> D65_as_np = D65.as_diagonal_array()
>>> type(D65_as_np)
numpy.ndarray
```

```
>>> print(D65_as_np.shape)
(97, 97)
```

To set the SPD into the visible spectrum:

Illuminant.set_into_visible_range_spectrum(visible_nm_range=[400,700], visible_nm_interval=10)

Method to set the spectral data into the visible spectrum.

Parameters:

visible_nm_range: `list` *[min, max] range in nm. Default: [400,700].*

visible_nm_interval: `int` *Interval in nm. Default: 10.*

To set the SPD into the visible range spectrum:

```
>>> D65.set_into_visible_range_spectrum(visible_nm_range = [400,700],
    visible_nm_interval = 10)
```

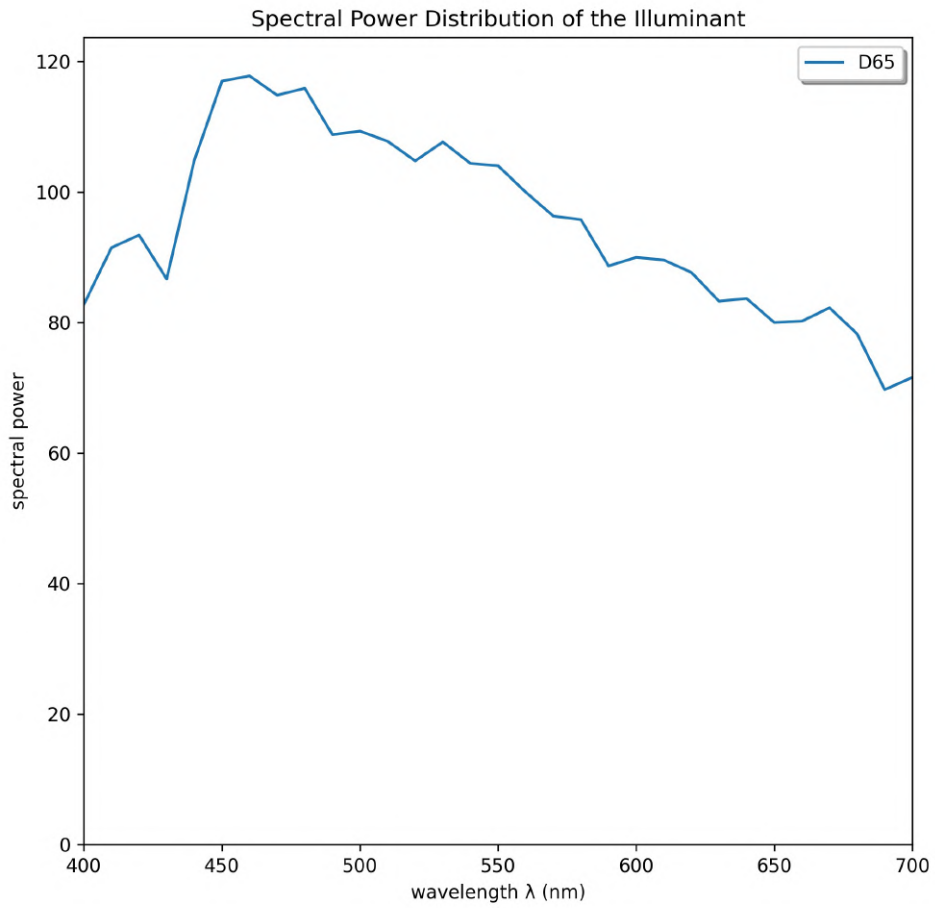

Figure 27: *Illuminant (visible spectrum)*

To normalise the SPD:

> ***Illuminant.normalise_lambda_values()***
>
> *Method to normalise the SPD data of the illuminant.*

To normalise the SPD:

```
>>> D65 = Illuminant(cie_illuminant_name="d65", normalised = True)
>>> print(D65.normalised)
True
```

An alternative way to normalise the spectral data:

```
>>> D65.normalise_lambda_values()
>>> print(D65.normalised)
True
```

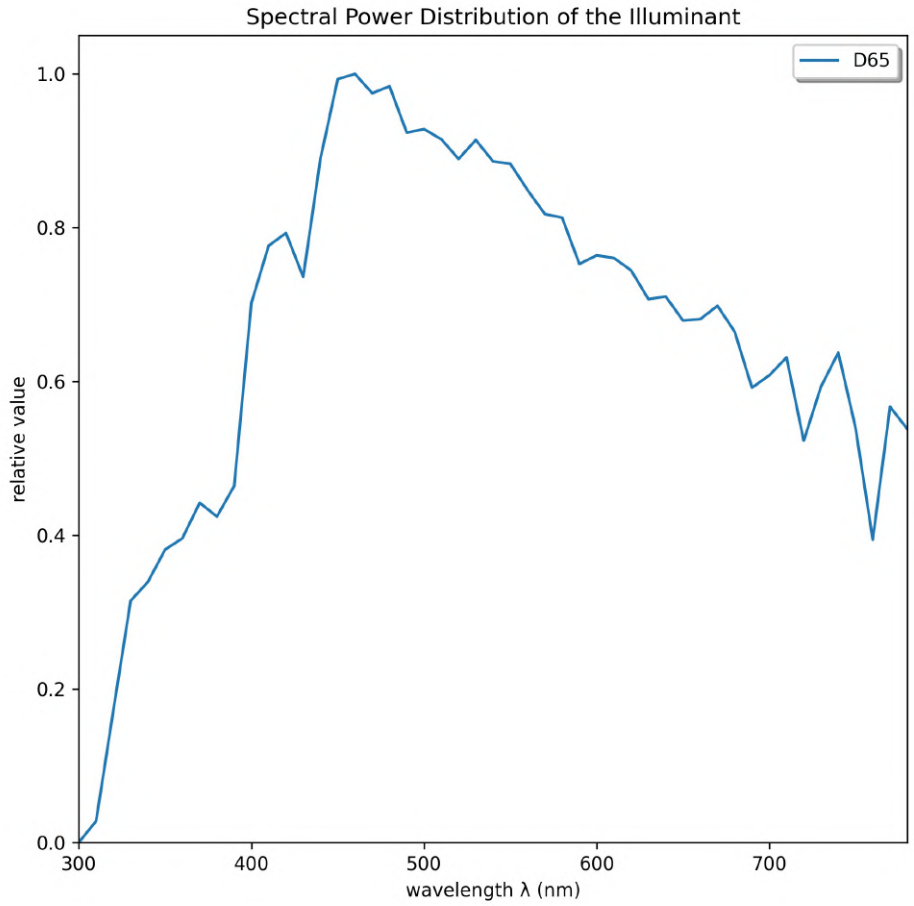

Figure 28: *Illuminant (normalised)*

`WhitePoint computations:`

Illuminant.get_theoretical_white_point_XYZ(observer=2)

Method to get the theoretical illuminant WhitePoint in CIE XYZ coordinates.

Theoretical data are available only for CIE Illuminants: A, C, D50, D55, D65, D75.

Returns `None` for all the remaining illuminants.

Parameters:

observer: `str`, `int`, `Observer` *CIE standard observer. Default: 2.*

Returns:

X_n, Y_n, Z_n: `float` *Theoretical CIE XYZ WhitePoint data.*

To get the theoretical CIE XYZ data of the *Illuminant* WhitePoint:

```
>>> Xn, Yn, Zn = D65.get_theoretical_white_point_XYZ(observer=2)
>>> print(Xn, Yn, Zn)
95.04 100.0 108.88
```

It is possible to introduce an *Observer* object to the class method as an `observer` parameter as follows:

```
>>> from coolpi.colour.cie_colour_spectral import Observer
>>> obs = Observer(10)
>>> Xn, Yn, Zn = D65.get_theoretical_white_point_XYZ(observer=obs)
>>> print(Xn, Yn, Zn)
94.81 100.0 107.32
```

Theoretical data are available only for CIE Illuminants: A, C, D50, D55, D65, D75. The class method returns `None` for all the remaining illuminants:

```
>>> LED1 = Illuminant("LED-B1")
>>> Xn, Yn, Zn = LED1.get_theoretical_white_point_XYZ(observer=2)
>>> print(Xn, Yn, Zn)
None None None
```

Illuminant.compute_white_point_XYZ(observer=2)

Method to compute the illuminant WhitePoint in CIE XYZ coordinates.

Parameters:

observer: `str`, `int`, `Observer` *CIE standard observer. Default: 2.*

Returns:

X_n, Y_n, Z_n: `float` *Computed XYZ WhitePoint.*

To compute the XYZ values of the *Illuminant* WhitePoint (it provides similar results compared to theoretical data):

```
>>> Xn, Yn, Zn = D65.compute_white_point_XYZ(observer=2)
>>> print(Xn, Yn, Zn)
95.04296621098943 100.00000000000007 108.88005680506745
```

It is possible to introduce an *Observer* object to the class method as an `observer` parameter as follows:

```
>>> from coolpi.colour.cie_colour_spectral import Observer
>>> obs = Observer(10)
>>> Xn, Yn, Zn = D65.compute_white_point_XYZ(observer=obs)
>>> print(Xn, Yn, Zn)
94.81179251714113 99.99999999999999 107.31944255522858
```

Illuminant.compute_white_point_xy(observer=2)

Method to compute the illuminant WhitePoint in CIE x,y coordinates.

Parameters:

observer: `str`, `int`, `Observer` *CIE standard observer. Default: 2.*

Returns:

x_n, y_n: `float` *Computed x,y WhitePoint.*

To compute the x,y chromaticity coordinates of the *Illuminant* WhitePoint:

```
>>> xn, yn = D65.compute_white_point_xy(observer=2)
>>> print(xn, yn)
0.31272052136033174 0.32903068351855935
```

It is possible to introduce an *Observer* object to the class method as an `observer` parameter as follows:

```
>>> from coolpi.colour.cie_colour_spectral import Observer
>>> obs = Observer(10)
>>> xn, yn = D65.compute_white_point_xy(observer=10)
>>> print(xn, yn)
0.3138099657072225 0.330981998521427
```

Illuminant.compute_white_point_uv_1976(observer=2)

Method to compute the illuminant WhitePoint in CIE u',v' coordinates.

Parameters:

observer: `str`, `int`, `Observer` *CIE standard observer. Default: 2.*

Returns:

u'_n, v'_n: `float` *Computed u',v' WhitePoint.*

To compute the u',v' chromaticity coordinates of the *Illuminant* WhitePoint:

```
>>> un, vn = D65.compute_white_point_uv_1976(observer=2)
>>> print(un, vn)
0.1978327527562152 0.4683394378846908
```

It is possible to introduce an *Observer* object to the class method as an `observer` parameter

as follows:

```
>>> from coolpi.colour.cie_colour_spectral import Observer
>>> obs = Observer(10)
>>> un, vn = D65.compute_white_point_uv_1976(observer=obs)
>>> print(un, vn)
0.19785740922985115 0.4695398736256153
```

Illuminant.compute_white_point_uv_1960(observer=2)

Method to compute the illuminant WhitePoint in CIE u,v coordinates.

Parameters:

observer: `str`, `int`, `Observer` *CIE standard observer. Default: 2.*

Returns:

u_n, v_n: `float` *Computed u,v WhitePoint.*

To compute the u,v chromaticity coordinates of the *Illuminant* WhitePoint:

```
>>> un, vn = D65.compute_white_point_uv_1960(observer=2)
>>> print(un, vn)
0.1978327527562152 0.31222629192312723
```

It is possible to introduce an *Observer* object to the class method as an `observer` parameter as follows:

```
>>> from coolpi.colour.cie_colour_spectral import Observer
>>> obs = Observer(10)
>>> un, vn = D65.compute_white_point_uv_1960(observer=obs)
>>> print(un, vn)
0.19785740922985115 0.3130265824170768
```

CCT computations:

Illuminant.compute_CCT(method="McCamy")

Method to compute the CCT (º K) from the x,y chromaticity coordinates.

Parameters:

method: `str` *Method. Default: "McCamy".*

Returns:

cct_K: `float` *Computed CCT in ºK.*

To compute the CCT of the *Illuminant*:

```
cct_k = D65.compute_CCT(method="McCamy")
print(cct_k)
6503.738306232476
cct_k = D65.compute_CCT(method="Hernandez")
print(cct_k)
6499.391166487143
cct_k = D65.compute_CCT(method="Ohno")
print(cct_k)
6502.887111814883
```

Info

McCamy, C.S. 1992. Correlated color temperature as an explicit function of chromaticity coordinates, Color Res. Appl. 17, 142-144 (McCamy, 1992).

Hernandez-Andres, J., Lee, R. L., & Romero, J. 1999. Calculating correlated color temperatures across the entire gamut of daylight and skylight chromaticities. Applied optics, 38(27), 5703-5709 (Hernandez-Andres et al, 1999).

Ohno, Yoshi. 2014. Practical Use and Calculation of CCT and Duv, LEUKOS, 10:1, 47-55 (Ohno, 2014).

Illuminant.compute_Duv()

Method to compute Δ_{uv}.

Returns:

Duv: `float` *Computed* Δ_{uv}.

To compute the Δ_{uv} of the *Illuminant*:

```
>>> Duv = D65.compute_Duv()
>>> print(Duv)
0.0032128098677425
```

Info

Ohno, Yoshi. 2014. Practical Use and Calculation of CCT and Duv, LEUKOS, 10:1, 47-55 (Ohno, 2014).

`__str__`:

> **`__str__`**
>
> *Returns the string representation of the object. This method is called when the `print()` or `str()` function is invoked on an object.*

```
>>> print(D65) # str method
Illuminant object: CIE D65 standard illuminant
```

5.1.4 Plot

> ***Illuminant.plot(show_figure=True, save_figure=False, output_path=None)***
>
> *Method to create and display the SPD of the illuminant using Matplotlib.*
>
> *Parameters:*
>
> *show_figure: `bool` If True, the plot is shown. Default: `True`.*
>
> *save_figure: `bool` If True, the figure is saved. Default: `False`.*
>
> *output_path: `os` path for the ouput figure. Default: `None`.*

Illuminant objects can be represented as follows:

```
>>> D65.plot()
```

In addition, it is posible to save the figure setting `True` the opcion `save_figure` and introducing a valid `output_path`.

```
>>> D65.plot(show_figure = False, save_figure = True, output_path = path_figure)
```

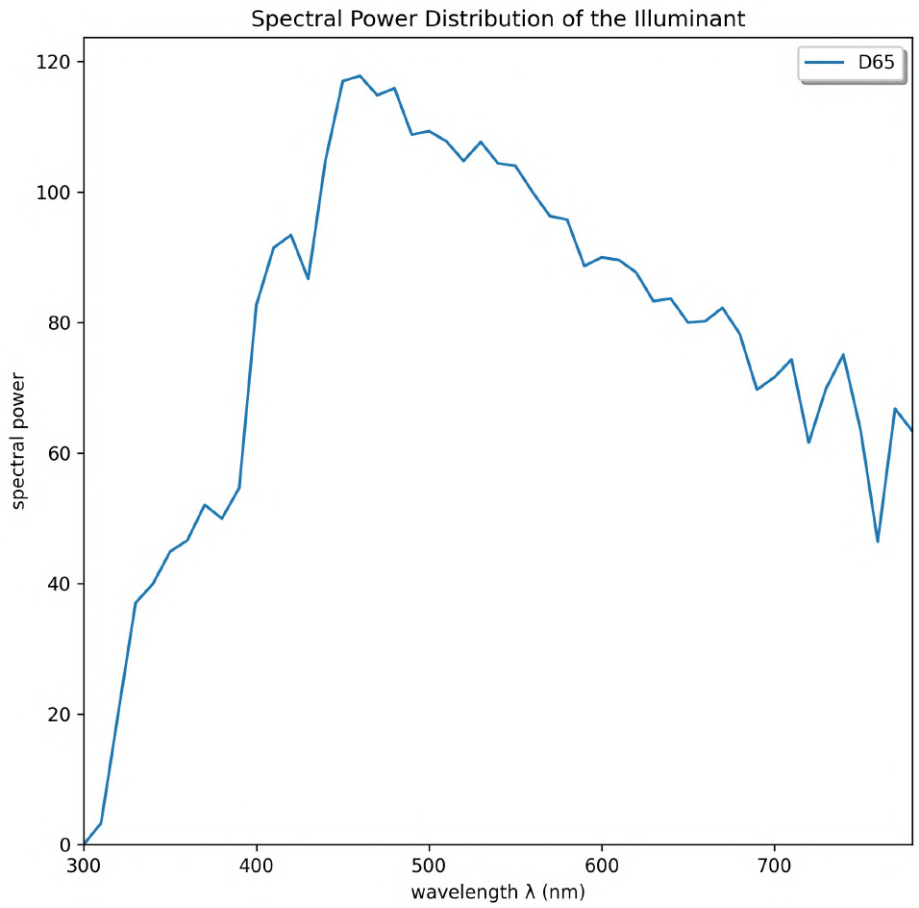

Figure 29: *Illuminant Plot*

Illuminant.plot_xy_white_point(observer=2, show_figure=True, save_figure=False, output_path=None)

Method to create and display the x,y WhitePoint into the CIE 1931 x,y Chromaticity Diagram using Matplotlib.

Parameters:

observer: `str`, `int`, `Observer` *CIE standard observer. Default: 2.*

show_figure: `bool` *If True, the plot is shown. Default:* `True`.

save_figure: `bool` *If True, the figure is saved. Default:* `False`.

output_path: `os` *path for the ouput figure. Default:* `None`.

The x,y values of the *Illuminant* WhitePoint can be represented as follows:

```
>>> D65.plot_xy_white_point(observer=2)
```

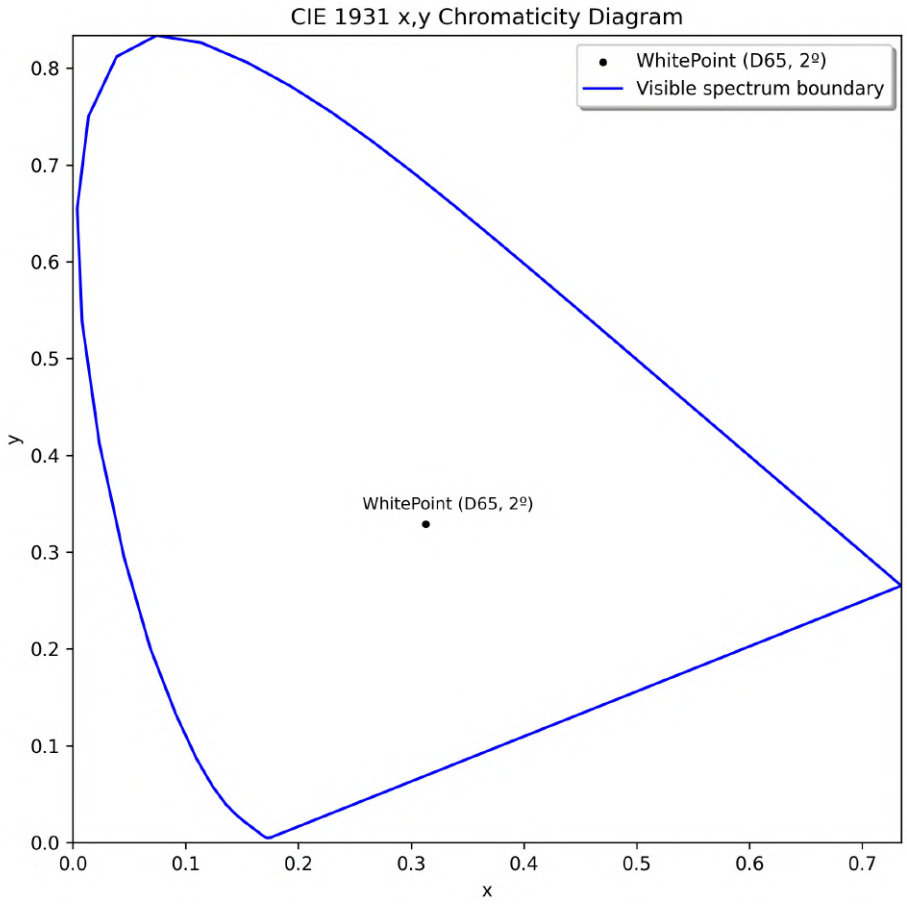

Figure 30: *Illuminant x,y WhitePoint (2º observer)*

```
>>> from coolpi.colour.cie_colour_spectral import Observer
>>> obs = Observer(10)
>>> D65.plot_xy_white_point(observer=obs)
```

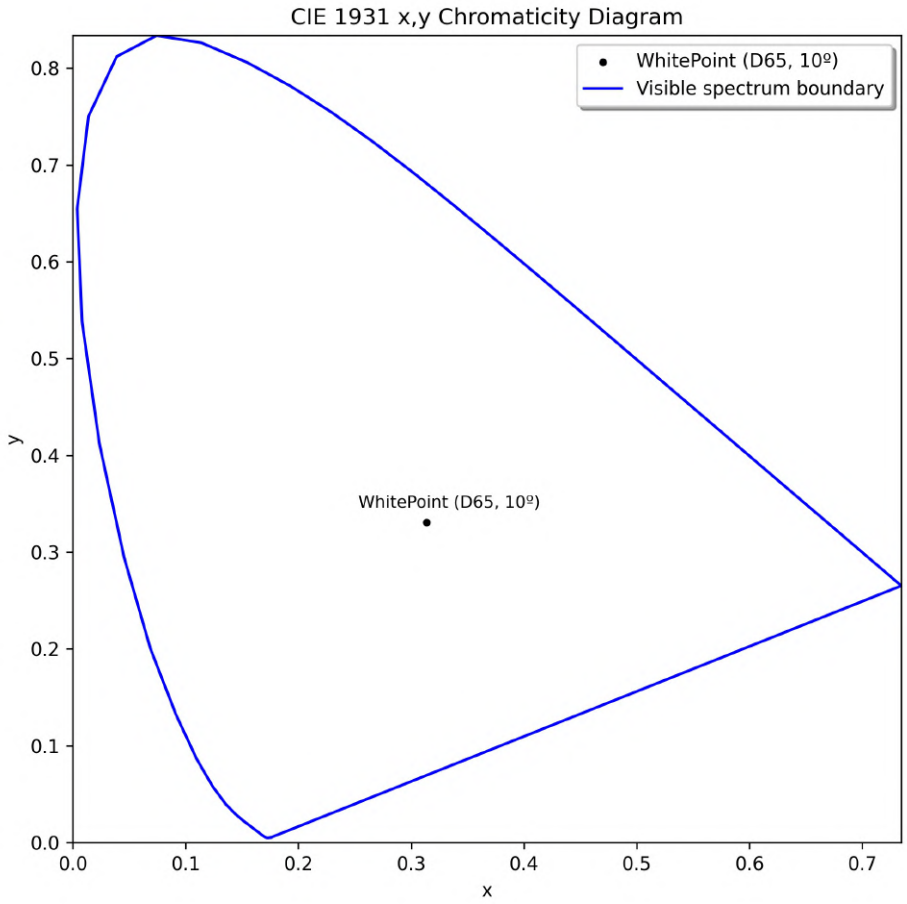

Figure 31: *Illuminant x,y WhitePoint (10º observer)*

In addition, it is posible to save the figure setting `True` the opcion `save_figure` and introducing a valid `output_path`.

```
>>> D65.plot_xy_white_point(observer=2, show_figure=False, save_figure=True,
    output_path=path_figure)
```

5.2 IlluminantFromCCT

class IlluminantFromCCT

IlluminantFromCCT(cct_K)

.as_diagonal_array()

.set_into_visible_range_spectrum(visible_nm_range, visible_nm_interval)

.normalise_lambda_values()

.compute_white_point_XYZ(observer)

.compute_white_point_xy(observer)

.compute_white_point_uv_1976(observer)

.compute_white_point_uv_1960(observer)

.compute_Duv()

.plot(show_figure, save_figure, output_path)

.plot_xy_white_point(observer, show_figure, save_figure, output_path)

The *IlluminantFromCCT* class represents an illuminant whose SPD is computed from a valid correlated colour temperature (specified in ºK).

When none of the CIE D illuminants can be used, a daylight illuminant at a nominal CCT temperature (into the range 4000-25000 ºK) can be calculated, using the CIE equations (CIE, 2018).

⚠ **Alert**

These equations provide an illuminant whose CCT is approximately equal to the nominal value, but not exactly so.

CIE 015:2018. 4.1.2 Other D illuminants (p.12) (CIE, 2018).

5.2.1 Create an instance

IlluminantFromCCT(cct_K)

Parameters:

cct_K: `float` *CCT (º K).*

To *create an instance* of the *IlluminantFromCCT* class, simply enter the required CCT (º K) as follows:

```
>>> from coolpi.colour.cie_colour_spectral import IlluminantFromCCT
>>> cct_6500 = IlluminantFromCCT(cct_K=6500)
>>> type(cct_6500)
coolpi.colour.cie_colour_spectral.IlluminantFromCCT
```

The input CCT should be into the valid range (4000-25000):

```
>>> cct_1000 = IlluminantFromCCT(1000)
CCTNotInValidRangeError: CCT out of range [4000-25000] ºK
```

> **Alert**
>
> *It is only allowed to create instances for IlluminantFromCCT into the range (4000-25000). Otherwise, a CCTNotInValidRangeError is raised.*

Comparison between the CIE standard illuminant D65 and the one computed from the CCT:

```
>>> from coolpi.classes.cie_colour_spectral import Illuminant
>>> D65 = Illuminant("D65") # D65 illuminant
>>> from coolpi.colour.cie_colour_spectral import IlluminantFromCCT
>>> cct_6500 = IlluminantFromCCT(6500) # Computed from CCT
>>> import coolpi.auxiliary.plot as cpt
>>> illuminants = {D65.illuminant_name: (D65.nm_range, D65.nm_interval, D65.lambda_values),
    cct_6500.illuminant_name:(cct_6500.nm_range, cct_6500.nm_interval,
cct_6500.lambda_values)}
>>> cpt.plot_illuminant(illuminants) # Plot both illuminants
```

The SPD computed from the CCT using the CIE formulation provides results close to the theoretical SPD.

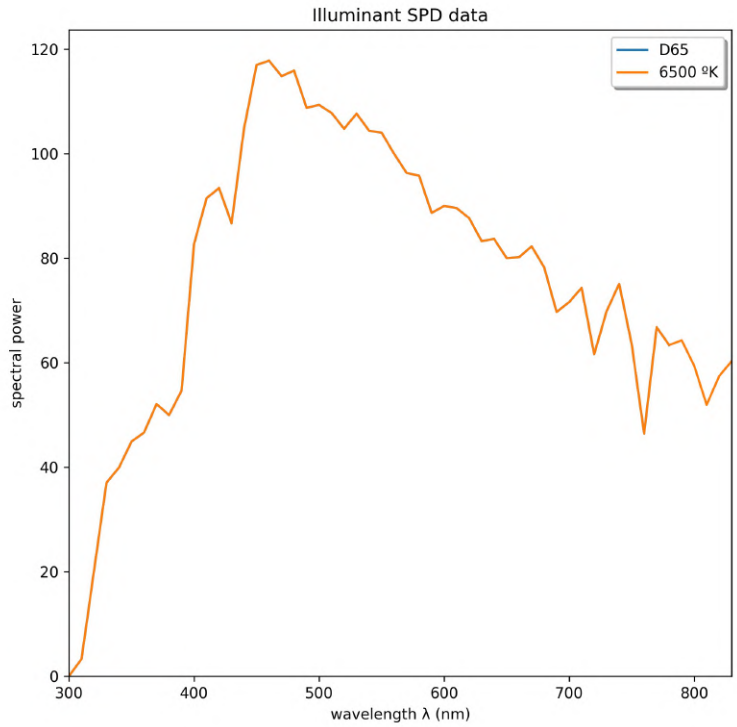

Figure 32: *CIE Illuminant D65 SPD data versus SPD computed from a CCT of 6500*

5.2.2 Attributes

The class *attributes* are: *type* (returns a `str` with the description of the main class), *subtype* (returns a `str` with the description of the object), *cct_K* (returns a `float` with the CCT), *illuminant_name* (returns a `str` with the illuminant name or description), *nm_range* (returns a `list` with the min and max range value in nm), *nm_interval* (returns an `int` with the lambda interval in nm), *lambda_values* (returns a `list` with the illuminant SPD) and *normalised* (returns a `bool`).

```
>>> print(cct_6500.type)
Spectral Object
>>> print(cct_6500.subtype)
SPD Illuminant computed from CCT
>>> print(cct_6500.cct_K)
6500
>>> print(cct_6500.illuminant_name)
6500 ºK
>>> print(cct_6500.nm_range)
[300, 830]
>>> print(cct_6500.nm_interval)
5
>>> print(cct_6500.lambda_values)
[0.0341070107442173, 1.664964028639965, 3.29582104653571, 11.76958016911639, 20.2433392916,
28.65523613908625, 37.06713298647541, 38.5145824753376, 39.9620319641998, 42.4437277348063,
44.92542350541294, 45.7881822682530, 46.65094103109305, 49.37706093877433, 52.103180846455,
51.04589598635213, 49.9886111262486, 52.3244630530405, 54.6603149798325, 68.71528464939324,
82.77025431895399, 87.1362741422727, 91.5022939655915, 92.4747861394841, 93.44727831337678,
90.07136853322397, 86.69545875307116, 95.7867783754486, 104.87809799782606, 110.9493270107,
117.0205560236943, 117.4222712280235, 117.8239864323528, 116.3474998513616, 114.8710132703,
115.4013823033908, 115.9317513364112, 112.3748639340858, 108.8179765317603,
109.08917375638,
109.36037098100098, 108.58343235663901, 107.80649373227, 106.2997310874192,
104.79296844256,
106.2423674525021, 107.691766462442, 106.0494014057284, 104.4070363490139,
104.2269704126757,
104.0469044763376, 102.0234522381688, 100, 98.1668067516856, 96.333613503371,
96.06016126520,
95.78670902703365, 92.23540647866838, 88.6841039303031, 89.34393031050946,
90.00375669071585,
89.79990363121277, 89.59605057170968, 88.64556674002509, 87.69508290834051,
85.4898495295043,
83.28461615066809, 83.4895682262552, 83.69452030184232, 81.85831689992642,
80.02211349801054,
80.11570232827667, 80.20929115854283, 81.240463710058, 82.27163626157318,
80.27488073192401,
78.27812520227486, 73.99707740822733, 69.7160296141798, 70.65961844444594,
71.60320727471208,
72.9732978841554, 74.34338849359871, 67.97138775048364, 61.599387007368584,
65.73997328312619,
69.88055955888377, 72.48112876062918, 75.08169796237459, 69.33498182013226,
63.58826567788993,
55.00153246163541, 46.41479924538091, 56.60766232812017, 66.80052541085942,
65.08940408138075,
63.37828275190208, 63.83883487963534, 64.2993870073686, 61.873533204750466,
59.44767940213234,
55.70153246163541, 51.95538552113849, 54.69597179689608, 57.43655807265366,
58.872394801259624,
60.30823152986558]
>>> print(cct_6500.normalised)
False
```

5.2.3 Methods

As a numpy diagonal array:

IlluminantFromCCT.as_diagonal_array()

Method to get the spectral data as a numpy diagonal array.

Returns:

as_diag: `numpy.ndarray` *SPD data as a diagonal array.*

To get the SPD as a diagonal array:

```
>>> cct_6500_as_np = cct_6500.as_diagonal_array()
>>> type(cct_6500_as_np)
numpy.ndarray
```

```
>>> print(cct_6500_as_np.shape)
(107, 107)
```

To set the SPD into the visible spectrum:

IlluminantFromCCT.set_into_visible_range_spectrum(visible_nm_range=[400,700], visible_nm_interval=10)

Method to set the spectral data into the visible spectrum.

Parameters:

visible_nm_range: `list` *[min, max] range in nm. Default: [400,700].*

visible_nm_interval: `int` *Interval in nm. Default: 10.*

To set the SPD into the visible range spectrum:

```
>>> cct_6500.set_into_visible_range_spectrum(visible_nm_range = [400,700],
    visible_nm_interval = 10)
```

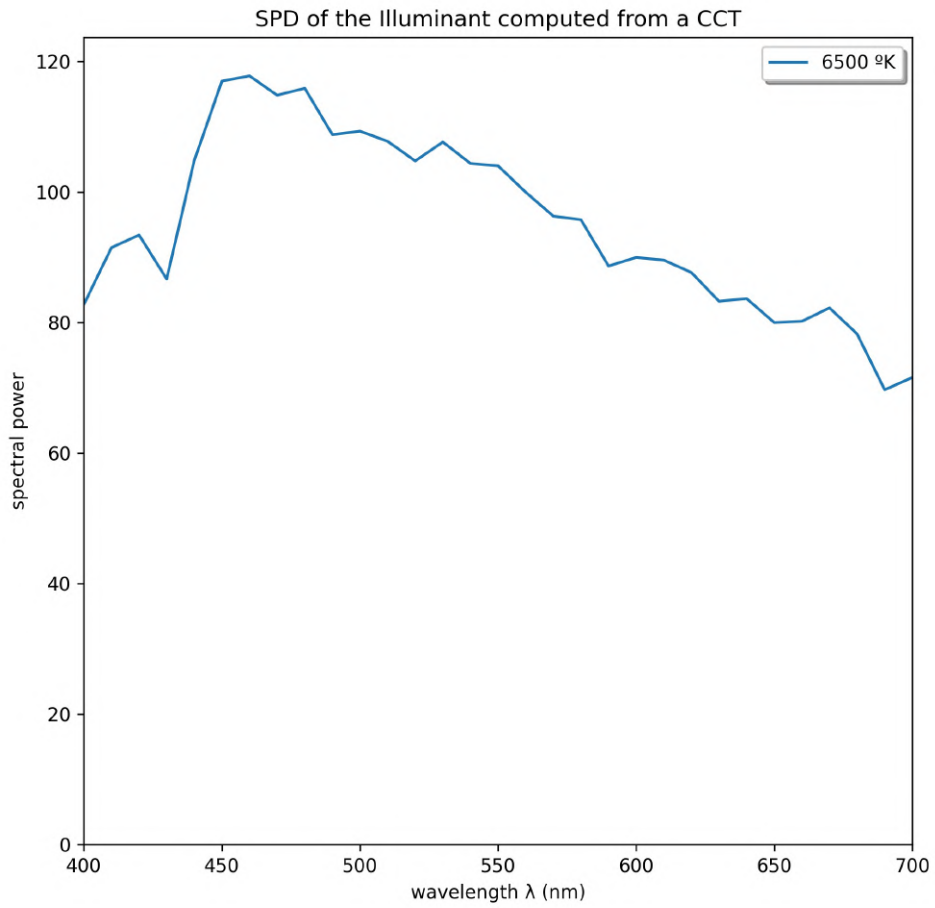

Figure 33: *IlluminantFromCCT (visible spectrum)*

To normalise SPD:

> ***IlluminantFromCCT.normalise_lambda_values()***
>
> *Method to normalise the SPD data of the illuminant.*

To normalise the SPD:

```
>>> cct_6500.normalise_lambda_values()
>>> print(cct_6500.normalised)
True
```

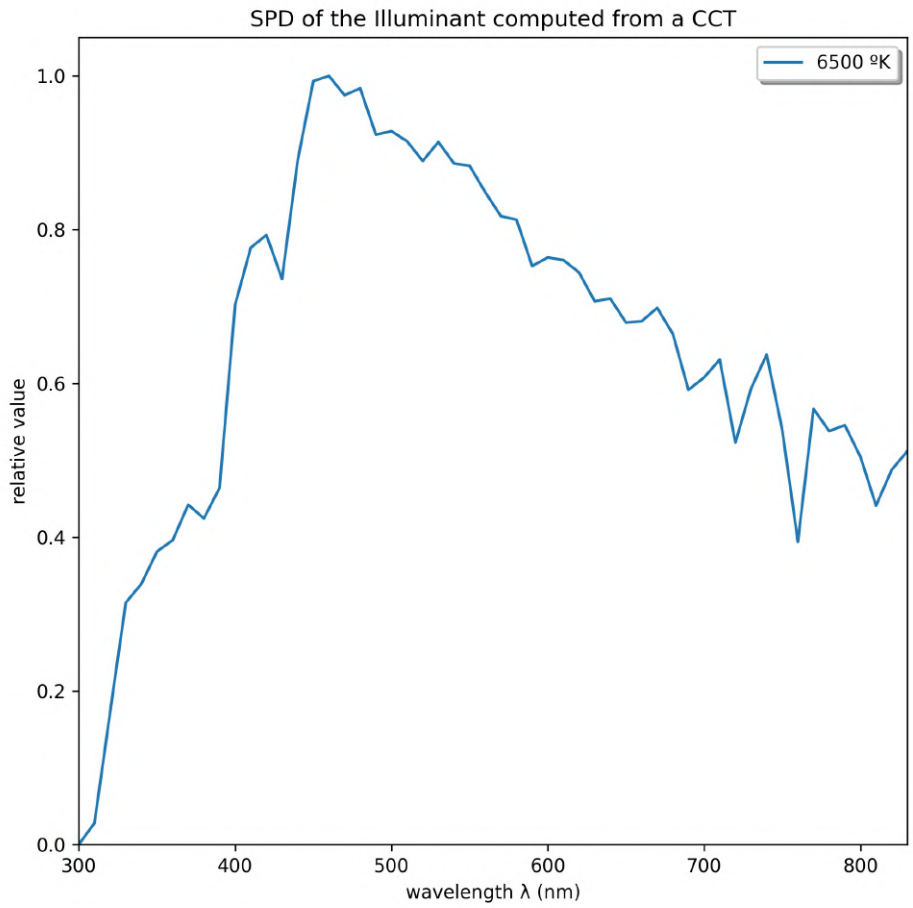

Figure 34: *IlluminantFromCCT (normalised)*

`WhitePoint computations:`

IlluminantFromCCT.compute_white_point_XYZ(observer=2)

Method to compute the illuminant WhitePoint in CIE XYZ coordinates.

Parameters:

observer: `str, int, Observer` *CIE standard observer. Default: 2.*

Returns:

X_n, Y_n, Z_n: `float` *Computed XYZ WhitePoint.*

To compute the XYZ of the *IlluminantFromCCT* WhitePoint:

```
>>> Xn, Yn, Zn = D65.compute_white_point_XYZ(observer=2)
>>> print(Xn, Yn, Zn)
95.04296621098943 100.00000000000007 108.88005680506745
```

It is possible to introduce an *Observer* object to the class method as an `observer` parameter as follows:

```
>>> from coolpi.colour.cie_colour_spectral import Observer
>>> obs = Observer(10)
>>> Xn, Yn, Zn = cct_6500.compute_white_point_XYZ(observer=obs)
>>> print(Xn, Yn, Zn)
94.81179251714113 99.99999999999999 107.31944255522858
```

IlluminantFromCCT.compute_white_point_xy(observer=2)

Method to compute the illuminant WhitePoint in CIE x,y coordinates.

Parameters:

observer: `str`, `int`, `Observer` *CIE standard observer. Default: 2.*

Returns:

x_n, y_n: `float` *Computed x,y WhitePoint.*

To compute the x,y chromaticity coordinates of the *IlluminantFromCCT* WhitePoint:

```
>>> xn, yn = cct_6500.compute_white_point_xy(observer=2)
>>> print(xn, yn)
0.31270854682321936 0.3290189140334981
```

It is possible to introduce an *Observer* object to the class method as an `observer` parameter as follows:

```
>>> from coolpi.colour.cie_colour_spectral import Observer
>>> obs = Observer(10)
>>> xn, yn = cct_6500.compute_white_point_xy(observer=10)
>>> print(xn, yn)
0.3137976216295172 0.33097065541803566
```

IlluminantFromCCT.compute_white_point_uv_1976(observer=2)

Method to compute the illuminant WhitePoint in CIE u',v' coordinates.

Parameters:

observer: `str`, `int`, `Observer` *CIE standard observer. Default: 2.*

Returns:

u'_n, v'_n: `float` *Computed u',v' WhitePoint.*

To compute the u',v' chromaticity coordinates of the *IlluminantFromCCT* WhitePoint:

```
>>> un, vn = cct_6500.compute_white_point_uv_1976(observer=2)
>>> print(un, vn)
0.1978288469952197 0.46833137243684275
```

It is possible to introduce an *Observer* object to the class method as an `observer` parameter as follows:

```
>>> from coolpi.colour.cie_colour_spectral import Observer
>>> obs = Observer(10)
>>> un, vn = cct_6500.compute_white_point_uv_1976(observer=obs)
>>> print(un, vn)
0.19785310137770626 0.4695320288716597
```

IlluminantFromCCT.compute_white_point_uv_1960(observer=2)

Method to compute the illuminant WhitePoint in CIE u,v coordinates.

Parameters:

observer: `str, int, Observer` *CIE standard observer. Default: 2.*

Returns:

u_n, v_n: `float` *Computed u,v WhitePoint.*

To compute the u,v chromaticity coordinates of the *IlluminantFromCCT* WhitePoint:

```
>>> un, vn = cct_6500.compute_white_point_uv_1960(observer=2)
>>> print(un, vn)
0.1978288469952197 0.31222091495789517
```

It is possible to introduce an *Observer* object to the class method as an `observer` parameter as follows:

```
>>> from coolpi.colour.cie_colour_spectral import Observer
>>> obs = Observer(10)
>>> un, vn = cct_6500.compute_white_point_uv_1960(observer=obs)
>>> print(un, vn)
0.19785310137770626 0.31302135258110647
```

CCT computations:

IlluminantFromCCT.compute_Duv()

Method to compute the Δ_{uv}.

Returns:

Duv: `float` *Computed* Δ_{uv}.

To compute the Δ_{uv} of the *IlluminantFromCCT*:

```
>>> Duv = cct_6500.compute_Duv()
>>> print(Duv)
0.0032128142198512427
```

Info

Ohno, Yoshi. 2014. Practical Use and Calculation of CCT and Duv, LEUKOS, 10:1, 47-55 (Ohno, 2014).

`__str__`:

`__str__`

Returns the string representation of the object. This method is called when the `print()` *or* `str()` *function is invoked on an object.*

```
>>> print(cct_6500)
IlluminantFromCCT object: CCT 6500 º K
```

5.2.4 Plot

IlluminantFromCCT.plot(show_figure=True, save_figure=False, output_path=None)

Method to create and display the SPD of the illuminant using Matplotlib.

Parameters:

show_figure: `bool` *If True, the plot is shown. Default:* `True`.

save_figure: `bool` *If True, the figure is saved. Default:* `False`.

output_path: `os` *path for the ouput figure. Default:* `None`.

IlluminantFromCCT objects can be represented as follows:

```
>>> cct_6500.plot()
```

In addition, it is posible to save the figure setting `True` the opcion `save_figure` and introducing a valid `output_path`.

```
>>> cct_6500.plot(show_figure = False, save_figure = True,
    output_path = path_figure)
```

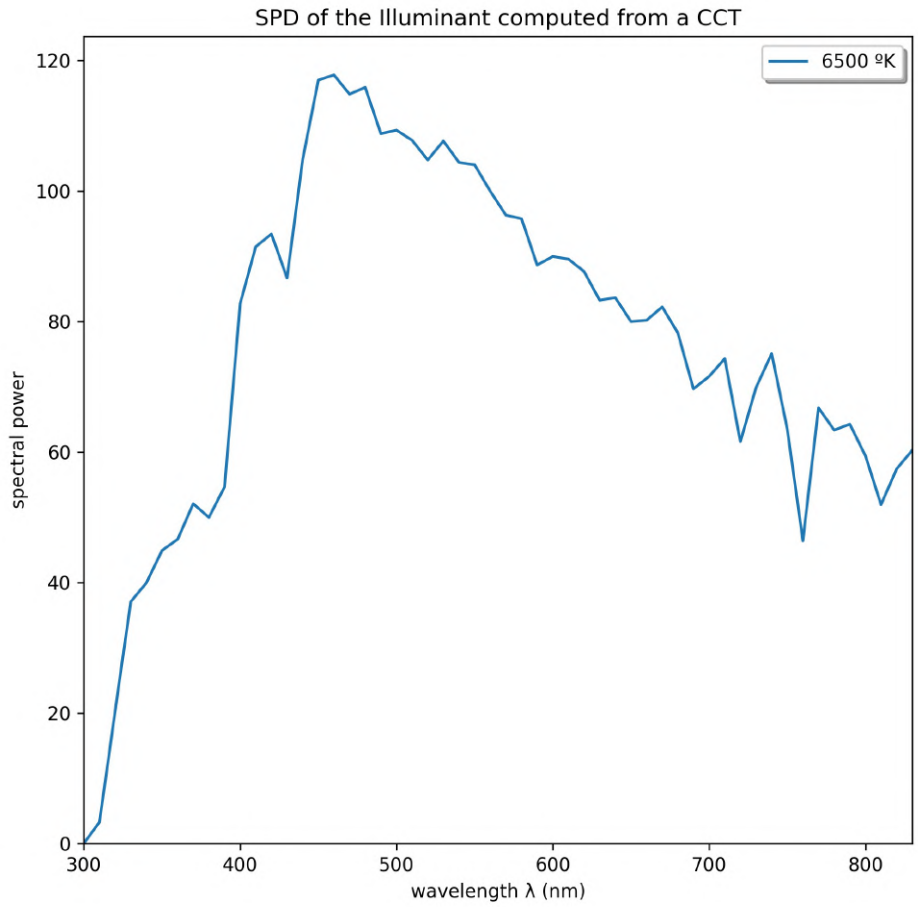

Figure 35: *IlluminantFromCCT Plot*

IlluminantFromCCT.plot_xy_white_point(observer=2, show_figure=True, save_figure=False, output_path=None)

Method to create and display the x,y WhitePoint into the CIE 1931 x,y Chromaticity Diagram using Matplotlib.

Parameters:

observer: `str`, `int`, `Observer` *CIE standard observer. Default: 2.*

show_figure: `bool` *If True, the plot is shown. Default:* `True`.

save_figure: `bool` *If True, the figure is saved. Default:* `False`.

output_path: `os` *path for the ouput figure. Default:* `None`.

The x,y values of the *IlluminantFromCCT* WhitePoint can be represented as follows:

```
>>> cct_6500.plot_xy_white_point(observer=2)
```

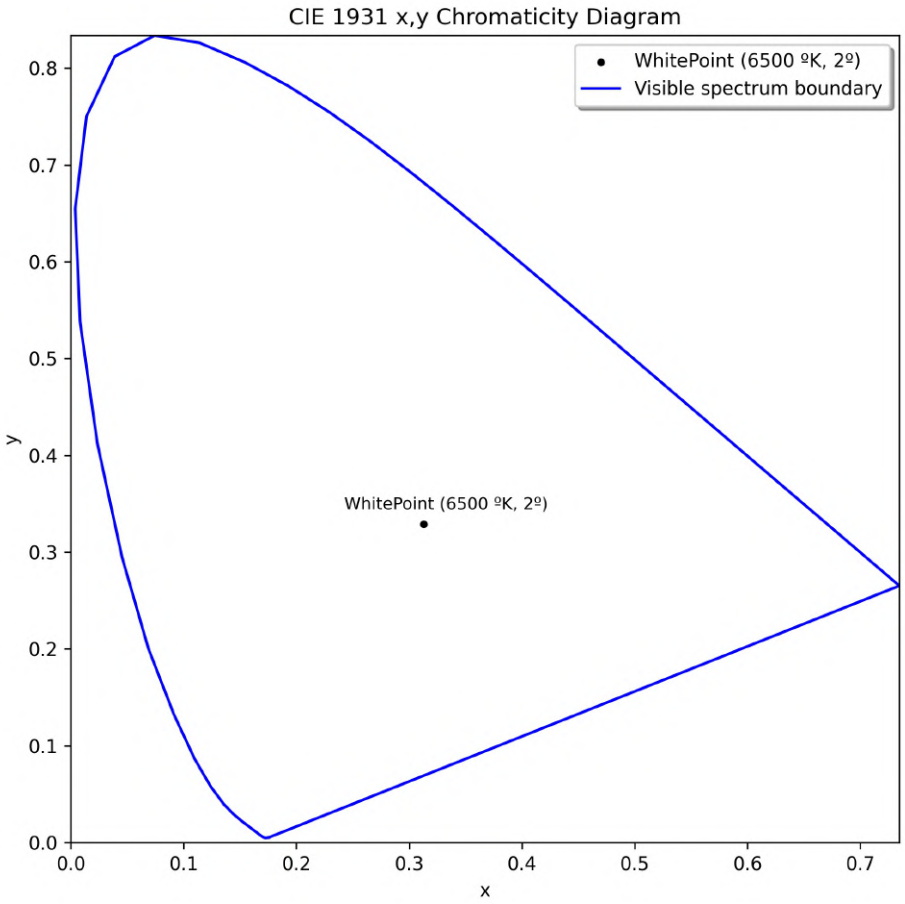

Figure 36: *IlluminantFromCCT x,y WhitePoint (2º observer)*

```
>>> from coolpi.colour.cie_colour_spectral import Observer
>>> obs = Observer(10)
>>> cct_6500.plot_xy_white_point(observer=obs)
```

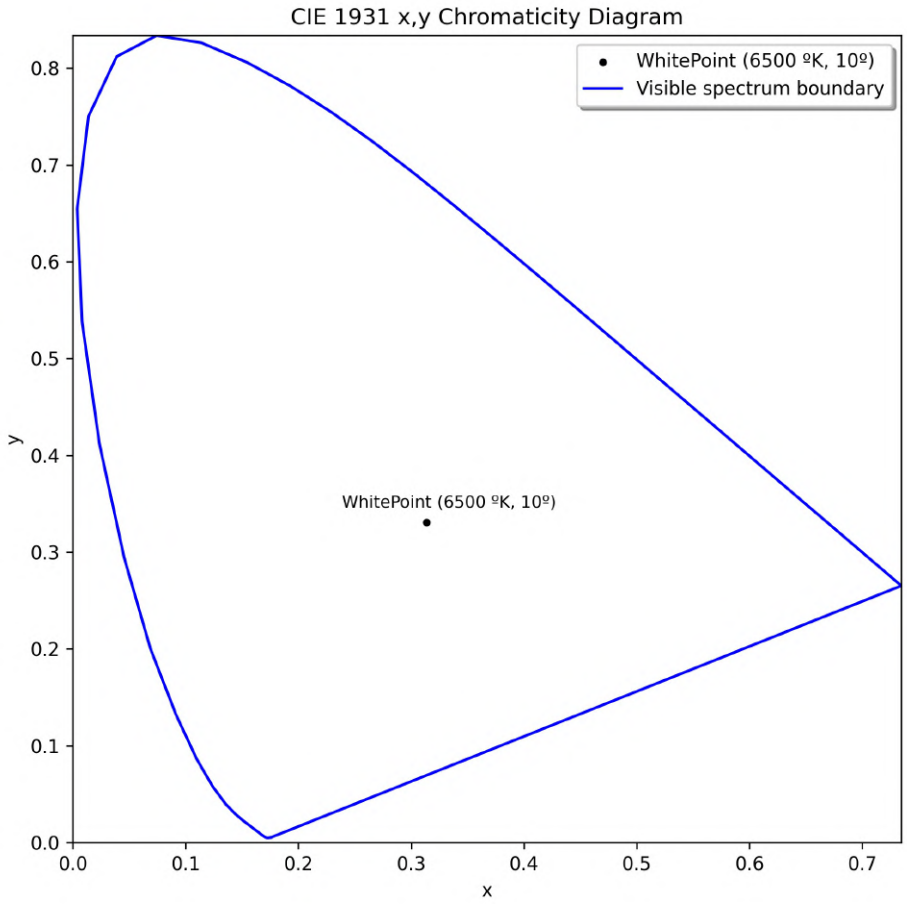

Figure 37: *IlluminantFromCCT xy WhitePoint (10º observer)*

In addition, it is posible to save the figure setting `True` the opcion `save_figure` and introducing a valid `output_path`.

```
>>> cct_6500.plot_xy_white_point(observer=2, show_figure = False,
    save_figure = True, output_path = path_figure)
```

5.3 MeasuredIlluminant

class MeasuredIlluminant

MeasuredIlluminant(illuminant_name, data=None, path_file=None, metadata={}, normalised=`False`)

.set_instrument_measurement_as_metadata()

.as_diagonal_array()

.set_into_visible_range_spectrum(visible_nm_range, visible_nm_interval)

.normalise_lambda_values()

.get_theoretical_white_point_XYZ(observer)

.compute_white_point_XYZ(observer)

.compute_white_point_xy(observer)

.compute_white_point_uv_1976(observer)

.compute_white_point_uv_1960(observer)

.compute_CCT(method)

.compute_Duv()

.plot(show_figure, save_figure, output_path)

.plot_xy_white_point(observer, show_figure, save_figure, output_path)

The *MeasuredIlluminant* class represents an illuminant whose SPD has been measured using specific instruments.

The SPD of light sources can be obtained by means of specific instrumentation, such as spectrometers. The measured data will vary depending on the instrumentation used. In order to cover the different devices available, the *MeasuredIlluminant* class has been implemented in coolpi to define illuminants for which at least these data is available: spectral power distribution data, spectral range and interval (in nm). The rest of the measures can be inserted in the class as a `dict`, which can be easily accessed by users.

5.3.1 Create an instance

MeasuredIlluminant(illuminant_name, data=None, path_file=None, metadata={}, normalised=False)

Parameters:

illuminant_name: `str` *Illuminant name or description.*

data: `dict` *Measured data.*

Required keys: SPD nm range [min, max], nm interval (nm) as `int` and SPD values as a `list`.*

path_file: `os` *Valid path. CSV for Sekonic. JSON in other case. Default:* `None`

metadata: `dict` *Instrument measurement information. Default: {}.*

normalised: `bool`, *If True, the SPD is normalised. Default:* `False`.

Generally, colorimetric instrumentation allows downloading measurements in easily uploaded Python file formats, such as CSV. The coolpi library includes a specific function to load CSV files obtained with Sekonic devices. For other instruments, it is posible to load the measured data using JSON files. In addition, it is possible to instantiate the class directly by entering the measurement data as a `dict`.

Regardless of the method used to instantiate the class, the following data are required as a minimum: `nm_range`, `nm_interval`, and `lambda_values`. Otherwise, a `DictLabelError` is raised.

To *create an instance* of the *MeasuredIlluminant* class, simply enter the required parameters as follows:

The `MeasuredIlluminant` class can be instantiated from: data (measured data as `dict`); from path_file (valid `os` path: CSV extension for Sekonic devices or JSON for other instruments).

From measured data as `dict`:

```
>>> from coolpi.colour.cie_colour_spectral import MeasuredIlluminant
>>> data_as_dict = {
    "nm_range": [380, 780],
    "nm_interval" :5,
    "lambda_values": [0.0114073, 0.00985631, 0.0072386, 0.00728911, 0.01057201, 0.01290828,
        0.01089283, 0.00820896, 0.00928692, 0.01575225, 0.03204091, 0.04716717, 0.04427407,
        0.03362948, 0.03129305, 0.0341507, 0.03702938, 0.0393975, 0.04113271, 0.0425808,
        0.04451207, 0.04628982, 0.04587397, 0.04293254, 0.03907673, 0.03530976, 0.03190796,
        0.02873687, 0.02581335, 0.02336659, 0.02249494, 0.02874496, 0.04532121, 0.05484243,
        0.04516013, 0.02744621, 0.0213983, 0.02104433, 0.02324293, 0.02704061, 0.02970082,
        0.02967269, 0.02855473, 0.02819227, 0.02867136, 0.03075241, 0.03364546, 0.03476707,
        0.03369897, 0.03211911, 0.03017166, 0.02761547, 0.02518627, 0.02336014, 0.02189655,
        0.02034273, 0.01835747, 0.01599688, 0.01375351, 0.01198202, 0.01058324, 0.0093942,
        0.00819748, 0.00698151, 0.00614299, 0.00563017, 0.00501737, 0.00425217, 0.00352747,
        0.00306196, 0.00279519, 0.00244556, 0.00205841, 0.00189265, 0.00200176, 0.00212645,
        0.00203685, 0.00170677, 0.00143012, 0.00107608, 0.00085404]
    }
>>> instrument_metadata = {"Date": [[2022,6,19], [9,29,43]], "Measuring Mode": "Ambient",
    "Viewing Angle":2}
>>> JND65 = MeasuredIlluminant(illuminant_name="JND65", data=data_as_dict, path_file =
None,
    metadata= instrument_metadata, normalised=False)
```

The `dict` should contain at least the following keys: “nm_range”, “nm_interval”, and

“lambda_values”. Otherwise, a `DictLabelError` is raised. In addition, optional keys can also be added: (e.g. CCT, Δ_{uv}).

```
>>> data_as_dict_incomplete = {
    "nm_range": [380, 780],
    "nm_interval" :5
    }
>>> JND65 = MeasuredIlluminant(illuminant_name="JND65", data=data_as_dict_incomplete,
    path_file = None, metadata= instrument_metadata, normalised=False)
DictLabelError: Error in the dict or file with measurement data: Incomplete data or wrong
labels.
```

From a CSV file with the measured data taken using a Sekonic instrument:

```
>>> from coolpi.colour.cie_colour_spectral import MeasuredIlluminant
>>> file_spd = ["res", "spd", "SPD_14_JND65_2022-06-19_02°_6658K.csv"]
>>> path_spd = os.path.join(*file_spd)
>>> instrument_metadata = {"Date": [[2022,6,19], [9,29,43]], "Measuring Mode": "Ambient",
    "Viewing Angle":2}
>>> JND65 = MeasuredIlluminant(illuminant_name="JND65", path_file = path_spd,
    metadata= instrument_metadata, normalised=False)
```

From a generic JSON file with the measured data taken with any instrument:

```
>>> from coolpi.colour.cie_colour_spectral import MeasuredIlluminant
>>> file_spd = ["res", "json", "JND65_Sekonic.json"]
>>> path_spd = os.path.join(*file_spd)
>>> instrument_metadata = {"Date": [[2022,6,19], [9,29,43]], "Measuring Mode": "Ambient",
    "Viewing Angle":2}
>>> JND65 = MeasuredIlluminant(illuminant_name="JND65", path_file = path_spd,
    metadata= instrument_metadata, normalised=False)
```

The JSON file should contain at least the following keys: “nm_range”, “nm_interval”, and “lambda_values”. Otherwise, a `DictLabelError` is raised.

Info

The MeasuredIlluminant metadata can be set separately with the method:

`.set_instrument_measurement_as_metadata(metadata)`

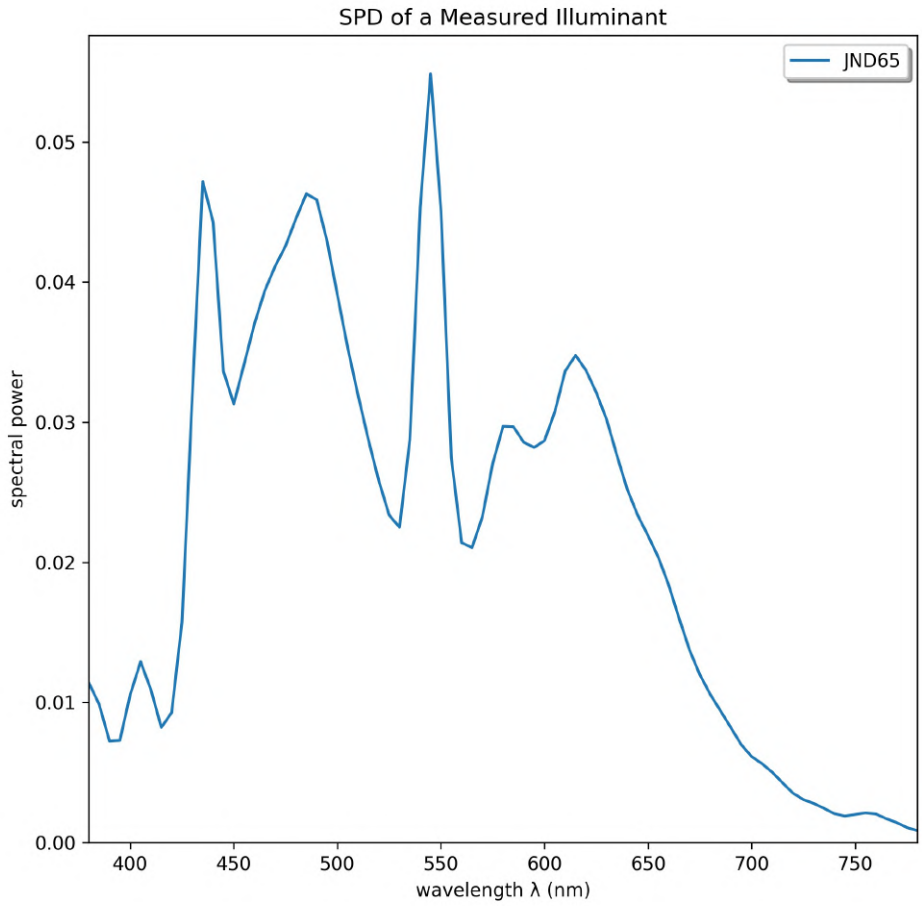

Figure 38: *MeasuredIlluminant*

5.3.2 Attributes

The class *attributes* are: *type* (returns a `str` with the description of the main class), *subtype* (returns a `str` with the description of the object), *illuminant_name* (returns a `str` with the illuminant name or description), *nm_range* (returns a `list` with the min and max range value in *nm*), *nm_interval* (returns an `int` with the lambda interval in *nm*), *lambda_values* (returns a `list` with the illuminant SPD), *normalised* (returns a `bool`), *measured_data* (returns a `dict` with the measured data) and *metadata* (returns a `dict` with the information about the measurement conditions).

```
>>> print(JND65.type)
Spectral Object
>>> print(JND65.subtype)
Measured SPD Illuminant
>>> print(JND65.illuminant_name)
JND65
>>> print(JND65.nm_range)
[380, 780]
>>> print(JND65.nm_interval)
5
>>> print(JND65.lambda_values)
[0.0114073  0.00985631 0.0072386  0.00728911 0.01057201 0.01290828
 0.01089283 0.00820896 0.00928692 0.01575225 0.03204091 0.04716717
 0.04427407 0.03362948 0.03129305 0.0341507  0.03702938 0.0393975
 0.04113271 0.0425808  0.04451207 0.04628982 0.04587397 0.04293254
 0.03907673 0.03530976 0.03190796 0.02873687 0.02581335 0.02336659
 0.02249494 0.02874496 0.04532121 0.05484243 0.04516013 0.02744621
 0.0213983  0.02104433 0.02324293 0.02704061 0.02970082 0.02967269
 0.02855473 0.02819227 0.02867136 0.03075241 0.03364546 0.03476707
 0.03369897 0.03211911 0.03017166 0.02761547 0.02518627 0.02336014
 0.02189655 0.02034273 0.01835747 0.01599688 0.01375351 0.01198202
 0.01058324 0.0093942  0.00819748 0.00698151 0.00614299 0.00563017
 0.00501737 0.00425217 0.00352747 0.00306196 0.00279519 0.00244556
 0.00205841 0.00189265 0.00200176 0.00212645 0.00203685 0.00170677
 0.00143012 0.00107608 0.00085404]
>>> print(JND65.normalised)
False
>>> print(JND65.measured_data.keys())
dict_keys(['nm_range', 'Date', 'Measuring Mode', 'Viewing Angle [º]', 'CCT',
'Delta_uv', 'Illuminance [lx]', 'XYZ', 'xyz', "u'v'", 'lambda_values_5nm',
'lambda_values_1nm'])
>>> print(JND65.metadata.keys())
dict_keys(['Date', 'Measuring Mode', 'Viewing Angle'])
```

5.3.3 Methods

As a numpy diagonal array:

MeasuredIlluminant.as_diagonal_array()

Method to get the spectral data as a numpy diagonal array.

Returns:

as_diag: `numpy.ndarray` *SPD data as a diagonal array.*

To get the SPD as a diagonal array:

```
>>> JND65_as_np = JND65.as_diagonal_array()
>>> type(JND65_as_np)
numpy.ndarray
```

```
>>> print(JND65_as_np.shape)
>>> (81, 81)
```

To set the SPD into the visible spectrum:

> ***MeasuredIlluminant.set_into_visible_range_spectrum(visible_nm_range=[400,700], visible_nm_interval=10)***
>
> *Method to set the spectral data into the visible spectrum.*
>
> *Parameters:*
>
> *visible_nm_range:* `list` *[min, max] range in nm. Default: [400,700].*
>
> *visible_nm_interval:* `int` *Interval in nm. Default: 10.*

To set the SPD into the visible range spectrum:

```
>>> JND65.set_into_visible_range_spectrum(visible_nm_range = [400,700],
    visible_nm_interval = 10)
```

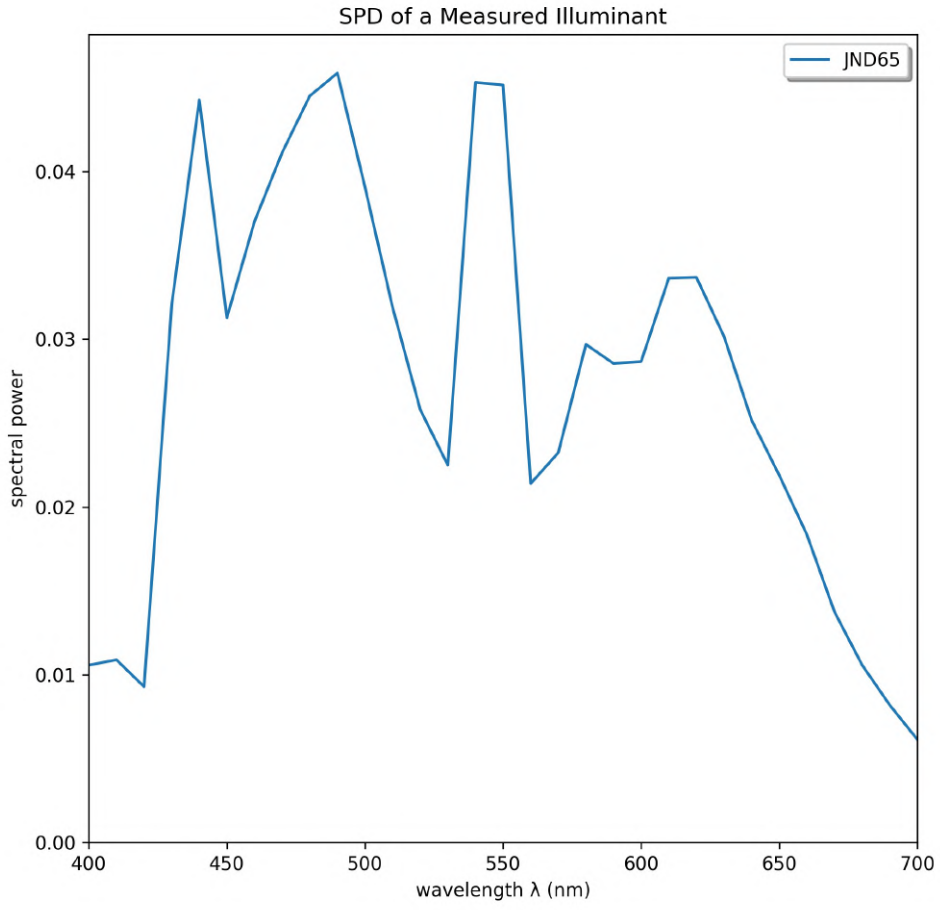

Figure 39: *MeasuredIlluminant (visible spectrum)*

To normalise the SPD:

> ***MeasuredIlluminant.normalise_lambda_values()***
>
> *Method to normalise the SPD data of the illuminant.*

To normalise the SPD:

```
>>> JND65 = MeasuredIlluminant(illuminant_name="JND65", path_file = path_spd,
    metadata= instrument_metadata, normalised=True)
>>> print(JND65.normalised)
True
```

An alternative wat to normalise the spectral data:

```
>>> JND65.normalise_lambda_values()
>>> print(JND65.normalised)
True
```

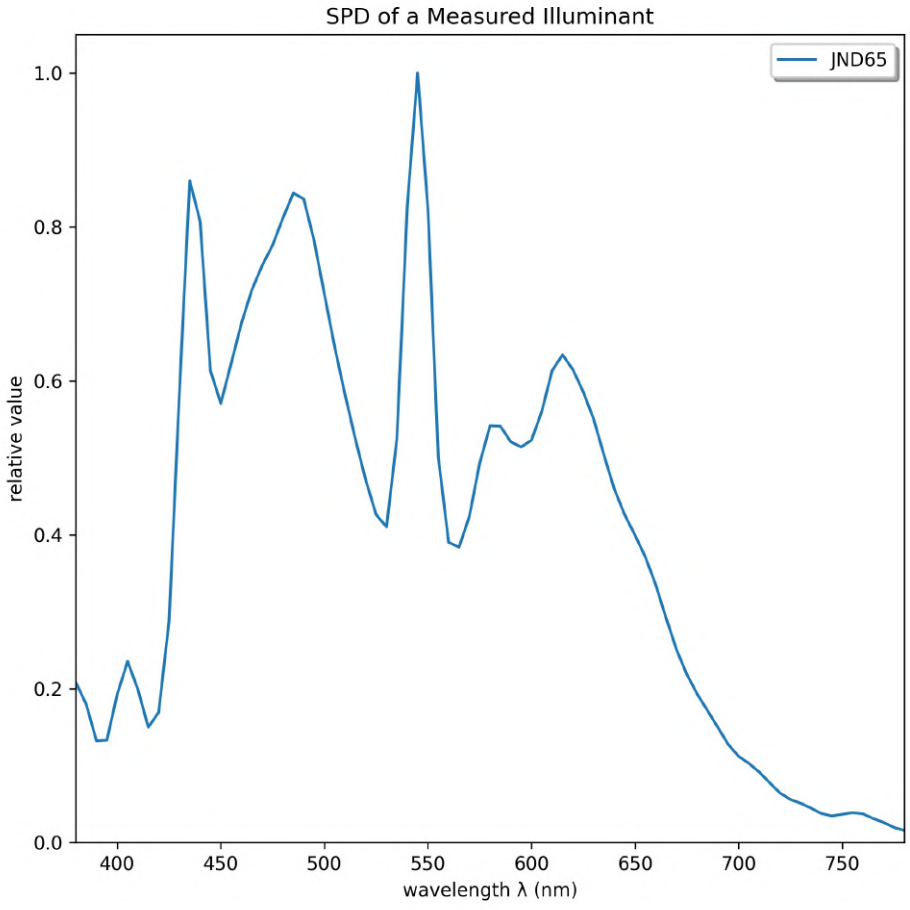

Figure 40: *MeasuredIlluminant (normalised)*

WhitePoint computations:

MeasuredIlluminant.compute_white_point_XYZ(observer=2)

Method to compute the illuminant WhitePoint in CIE X, Y, Z coordinates.

Parameters:

observer: `str`, `int`, `Observer` *CIE standard observer. Default: 2.*

Returns:

X_n, Y_n, Z_n: `float` *Computed XYZ WhitePoint.*

To compute the XYZ values of the *MeasuredIlluminant* WhitePoint:

```
>>> Xn, Yn, Zn = JND65.compute_white_point_XYZ(observer=2)
>>> print(Xn, Yn, Zn)
96.49001962118156 100.0 113.77215723444507
```

Compare to measured data (if available):

```
>>> Xn_, Yn_, Zn_ = JND65.measured_data["XYZ"]
>>> print(Xn_/Yn_*100, Yn_/Yn_*100, Zn_/Yn_*100) # Normalised to Yn_ and scaled
96.49001954952843 100.0 113.77215849328286
```

It is possible to introduce an *Observer* object to the class method as an `observer` parameter as follows:

```
>>> from coolpi.colour.cie_colour_spectral import Observer
>>> obs = Observer(10)
>>> Xn, Yn, Zn = JND65.compute_white_point_XYZ(observer=obs)
>>> print(Xn, Yn, Zn)
94.66708943782882 100.00000000000006 108.51020312717426
```

MeasuredIlluminant.compute_white_point_xy(observer=2)

Method to compute the illuminant WhitePoint in CIE x,y coordinates.

Parameters:

observer: `str`, `int`, `Observer` *CIE standard observer. Default: 2.*

Returns:

x_n, y_n: `float` *Computed x,y WhitePoint.*

To compute the x,y chromaticity coordinates of the *MeasuredIlluminant* WhitePoint:

```
>>> xn, yn = JND65.compute_white_point_xy(observer=2)
>>> print(xn, yn)
0.31099510935901464 0.3223080589888747
```

Compare to measured data (if available):

```
>>> xn_, yn_, zn_ = JND65.measured_data["xyz"]
>>> print(xn_, yn_, zn_)
0.311 0.3223 0.3667
```

It is possible to introduce an *Observer* object to the class method as an `observer` parameter as follows:

```
>>> from coolpi.colour.cie_colour_spectral import Observer
>>> obs = Observer(10)
>>> xn, yn = JND65.compute_white_point_xy(observer=obs)
>>> print(xn, yn)
0.3122499334858055 0.3298400060042736
```

MeasuredIlluminant.compute_white_point_uv_1976(observer=2)

Method to compute the illuminant WhitePoint in CIE u',v' coordinates.

Parameters:

observer: `str, int, Observer` *CIE standard observer. Default: 2.*

Returns:

u'_n, v'_n: `float` *Computed u',v' WhitePoint.*

To compute the u',v' chromaticity coordinates of the *MeasuredIlluminant* WhitePoint:

```
>>> un, vn = JND65.compute_white_point_uv_1976(observer=2)
>>> print(un, vn)
0.19917369469689275 0.4644426592795847
```

Compare to measured data (if available):

```
>>> un_, vn_ = JND65.measured_data["u'v'"]
>>> print(un_, vn_)
0.1992 0.4644
```

It is possible to introduce an *Observer* object to the class method as an `observer` parameter as follows:

```
>>> from coolpi.colour.cie_colour_spectral import Observer
>>> obs = Observer(10)
>>> un, vn = JND65.compute_white_point_uv_1976(observer=obs)
>>> print(un, vn)
0.19720279739119692 0.46870173865606235
```

MeasuredIlluminant.compute_white_point_uv_1960(observer=2)

Method to compute the illuminant WhitePoint in CIE u,v coordinates.

Parameters:

observer: `str`, `int`, `Observer` *CIE standard observer. Default: 2.*

Returns:

u_n, v_n: `float` *Computed u,v WhitePoint.*

To compute the u,v chromaticity coordinates of the *MeasuredIlluminant* WhitePoint:

```
>>> un, vn = JND65.compute_white_point_uv_1960(observer=2)
>>> print(un, vn)
0.19917369469689275 0.30962843951972313
```

It is possible to introduce an *Observer* object to the class method as an `observer` parameter as follows:

```
>>> from coolpi.colour.cie_colour_spectral import Observer
>>> obs = Observer(10)
>>> un, vn = JND65.compute_white_point_uv_1960(observer=obs)
>>> print(un, vn)
0.19720279739119692 0.31246782577070825
```

CCT computations:

MeasuredIlluminant.compute_CCT(method="McCamy")

Method to compute the CCT (º K) from the x,y chromaticity coordinates.

Parameters:

method: `str` *Method. Default: "McCamy".*

Returns:

cct_K: `float` *Computed CCT in ºK.*

To compute the CCT of the *MeasuredIlluminant*:

```
>>> cct_k = JND65.compute_CCT(method="McCamy")
>>> print(cct_k)
6655.347717016843
>>> cct_k = JND65.compute_CCT(method="Hernandez")
>>> print(cct_k)
6655.424286531472
>>> cct_k = JND65.compute_CCT(method="Ohno")
>>> print(cct_k)
6656.262990065892
```

Compare to measured data (if available):

```
>>> cct_k_ = JND65.measured_data["CCT"]
>>> print(cct_k_)
6658
```

Info

McCamy, C.S. 1992. Correlated color temperature as an explicit function of chromaticity coordinates, Color Res. Appl. 17, 142-144 (McCamy, 1992).

Hernandez-Andres, J., Lee, R. L., & Romero, J. 1999. Calculating correlated color temperatures across the entire gamut of daylight and skylight chromaticities. Applied optics, 38(27), 5703-5709 (Hernandez-Andres et al, 1999).

Ohno, Yoshi. 2014. Practical Use and Calculation of CCT and Duv, LEUKOS, 10:1, 47-55 (Ohno, 2014).

MeasuredIlluminant.compute_Duv()

Method to compute the Δ_{uv}.

Returns:

Duv: `float` Computed Δ_{uv}.

To compute the Δ_{uv} of the *MeasuredIlluminant*:

```
>>> Duv = JND65.compute_Duv()
>>> print(Duv)
0.0006092222311397388
```

Compare to measured data (if available):

```
>>> Duv_ = JND65.measured_data["Delta_uv"]
>>> print(Duv_)
0.0006
```

Info

Ohno, Yoshi. 2014. Practical Use and Calculation of CCT and Duv, LEUKOS, 10:1, 47-55 (Ohno, 2014).

__str__:

__str__

Returns the string representation of the object. This method is called when the `print()` or `str()` function is invoked on an object.

```
>>> print(JND65) # str method
MeasuredIlluminant object: Illuminant JND65
```

5.3.4 Plot

MeasuredIlluminant.plot(show_figure=True, save_figure=False, output_path=None)

Method to create and display the SPD of the illuminant using Matplotlib.

Parameters:

show_figure: `bool` If True, the plot is shown. Default: `True`.

save_figure: `bool` If True, the figure is saved. Default: `False`.

output_path: `os` path for the ouput figure. Default: `None`.

MeasuredIlluminant objects can be represented as follows:

```
>>> JND65.plot()
```

In addition, it is posible to save the figure setting `True` the opcion `save_figure` and introducing a valid `output_path`.

```
>>> JND65.plot(show_figure = False, save_figure = True, output_path = path_figure)
```

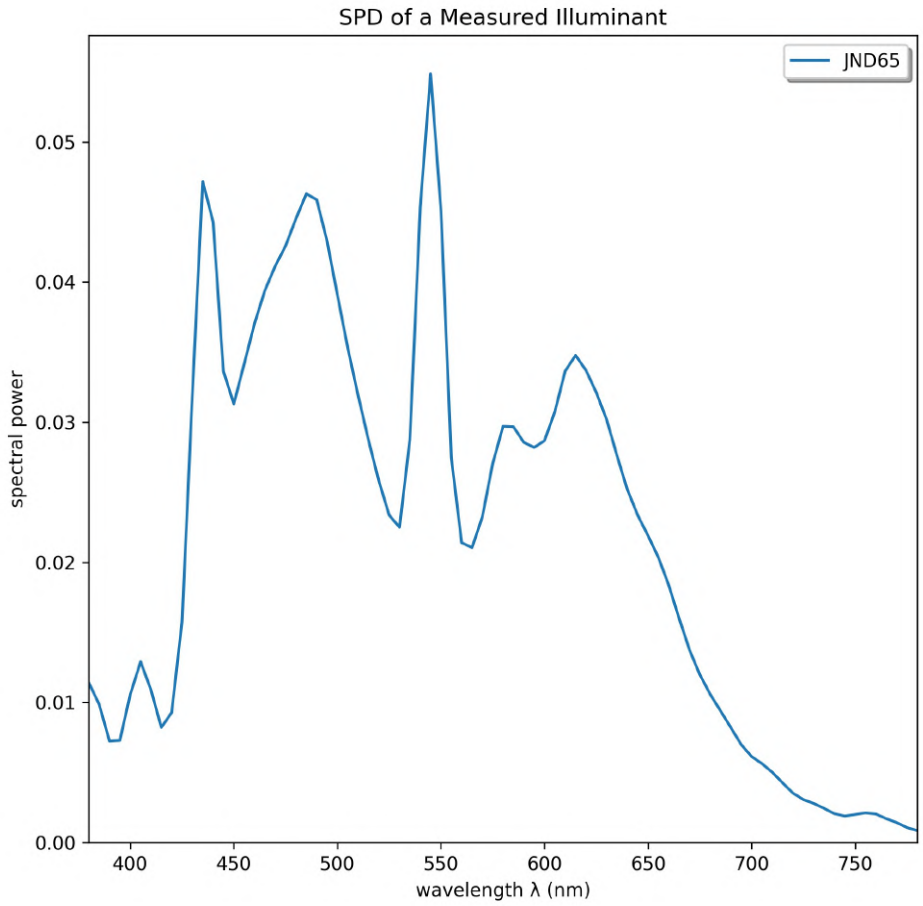

Figure 41: *MeasuredIlluminant Plot*

MeasuredIlluminant.plot_xy_white_point(observer=2, show_figure=True, save_figure=False, output_path=None)

Method to create and display the x,y WhitePoint into the CIE 1931 x,y Chromaticity Diagram using Matplotlib.

Parameters:

observer: `str`, `int`, `Observer` *CIE standard observer. Default: 2.*

show_figure: `bool` *If True, the plot is shown. Default:* `True`.

save_figure: `bool` *If True, the figure is saved. Default:* `False`.

output_path: `os` *path for the ouput figure. Default:* `None`.

The x,y values of the *MeasuredIlluminant* WhitePoint can be represented as follows:

```
>>> JND65.plot_xy_white_point(observer=2)
```

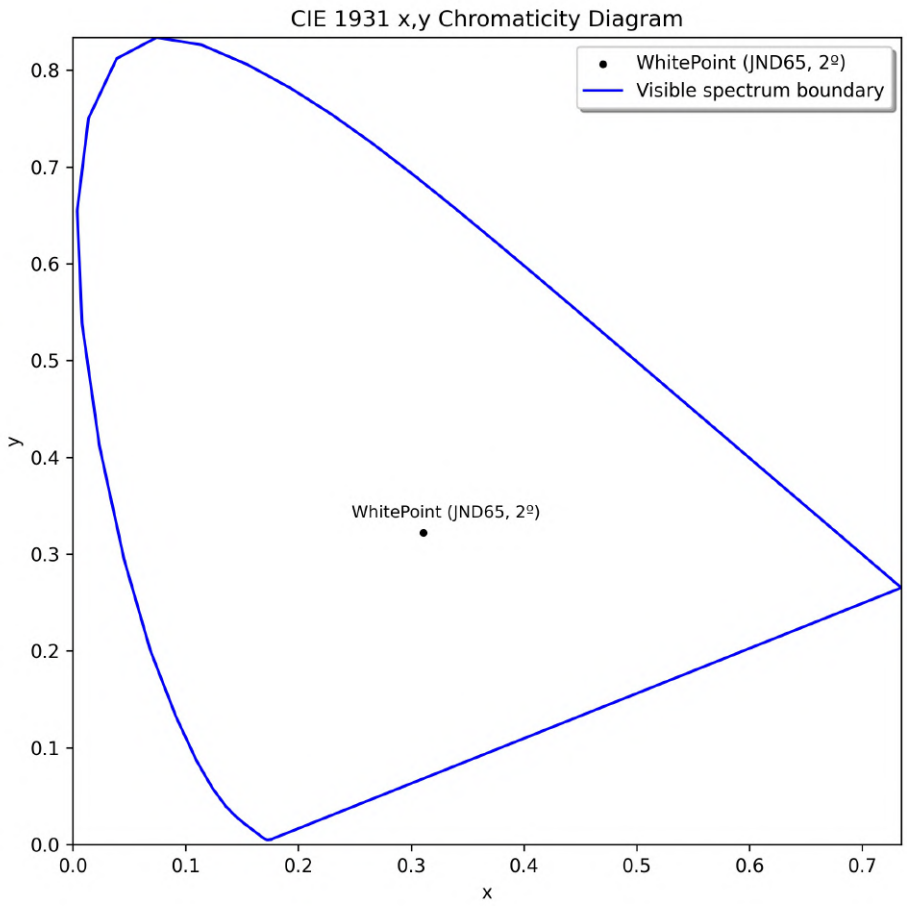

Figure 42: *MeasuredIlluminant x,y WhitePoint (2º observer)*

```
>>> from coolpi.colour.cie_colour_spectral import Observer
>>> obs = Observer(10)
>>> JND65.plot_xy_white_point(observer=obs)
```

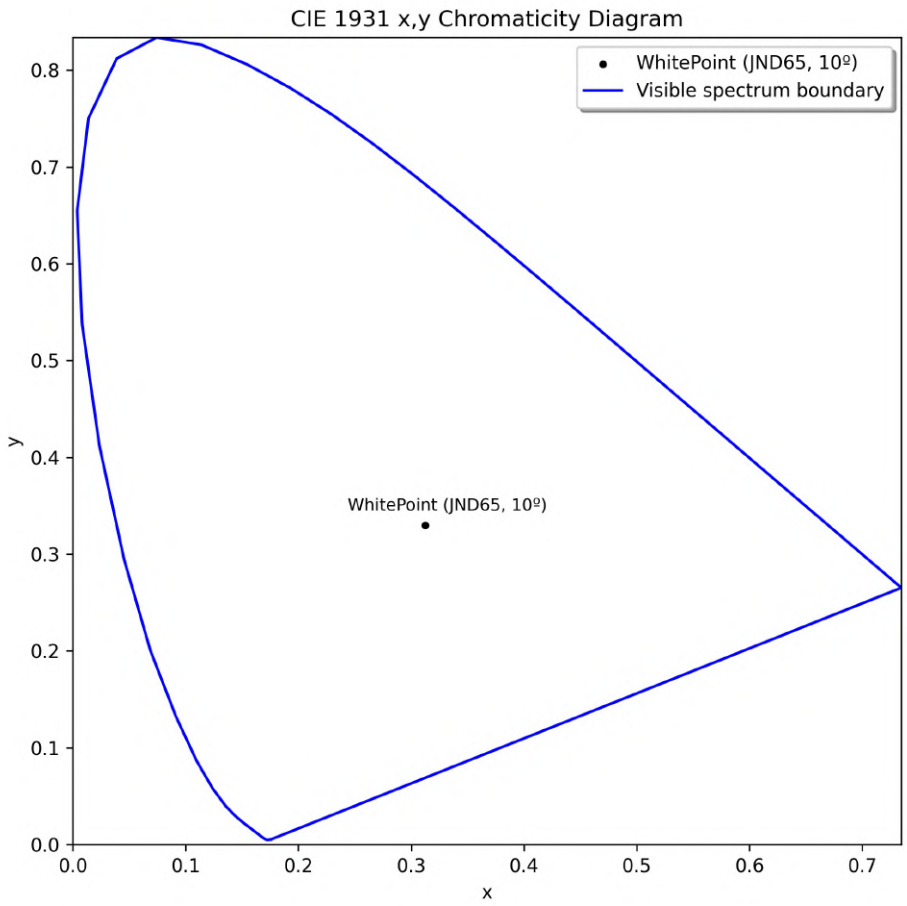

Figure 43: *MeasuredIlluminant x,y WhitePoint (10º observer)*

In addition, it is posible to save the figure setting `True` the opcion `save_figure` and introducing a valid `output_path`.

```
>>> JND65.plot_xy_white_point(observer=2, show_figure = False,
    save_figure = True, output_path = path_figure)
```

5.4 WhitePoint

class WhitePoint

WhitePoint(illuminant="D65", observer=2)

.to_xyY()

.to_uvY()

.plot_xy_white_point(show_figure, save_figure, output_path)

The *WhitePoint* class represents a white point (or reference target) object of a given

illuminant in XYZ coordinates.

The white point is defined by the CIE as the “achromatic reference stimulus in a chromaticity diagram that corresponds to the stimulus that produces an image area that has the perception of white” (CIE, 2018).

Theoretical data are available for the CIE illuminants: A, C, D50, D55, D65 and D75. Computed data is performed for all the remaining illuminants.

> **Info**
>
> *CIE 015:2018. Table 9.1 and 9.2. Colorimetric data for CIE illuminants (p.58) (CIE, 2018).*

5.4.1 Create an instance

> ***WhitePoint(illuminant=“D65", observer=2)***
>
> *Parameters:*
>
> *illuminant:* `str`, `Illuminant`, `IlluminantFromCCT`, `MeasuredIlluminant` — *Illuminant. Default: “D65".*
>
> *observer:* `str`, `int`, `Observer` — *CIE standard observer. Default: 2.*

To *create an instance* of the *WhitePoint* class, simply enter the required parameters as follows:

```
>>> from coolpi.colour.cie_colour_spectral import WhitePoint
>>> wp_d65 = WhitePoint(illuminant="D65", observer=2)
>>> type(wp_d65)
coolpi.colour.cie_colour_spectral.WhitePoint
```

It is possible to introduce an *Observer* object to the class method as an `observer` parameter as follows:

```
>>> from coolpi.colour.cie_colour_spectral import Observer
>>> obs = Observer(10)
>>> wp_d65_10 = WhitePoint(cie_illuminant_name="D65", observer=obs)
>>> type(wp_d65_10)
coolpi.colour.cie_colour_spectral.WhitePoint
```

To get the X_n, Y_n, Z_n values:

```
>>> Xn, Yn, Zn = wp_d65.coordinates
>>> print(Xn, Yn, Zn)
95.04 100.0 108.88
```

Theoretical data are available only for the CIE illuminants: A, C, D50, D55, D65 and D75. For all the remaining illuminants implemented, the X, Y, Z values of the illuminant white point are computed.

For a CIE *Illuminant* without theoretical data available:

```
>>> wp_ledb1 = WhitePoint(cie_illuminant_name="LED-B1", observer=2)
>>> Xn, Yn, Zn = wp_ledb1.coordinates
>>> print(Xn, Yn, Zn)
111.8078554407522 100.0 33.41107903970757
```

For an *IlluminantFromCCT* instance:

```
>>> from coolpi.colour.cie_colour_spectral import IlluminantFromCCT
>>> spd_cct = IlluminantFromCCT(6500)
>>> wp_cct = WhitePoint(illuminant=spd_cct, observer=2)
>>> Xn, Yn, Zn = wp_cct.coordinates
>>> print(Xn, Yn, Zn)
95.04272656841287 99.99999999999993 108.89116821618512
```

For a *MeasuredIlluminant* instance (example JND65):

```
wp_measured = WhitePoint(illuminant=JND65, observer=2)
Xn, Yn, Zn = wp_measured.coordinates
print(Xn, Yn, Zn)
```

The *WhitePoint* instance should be specified for a valid CIE standard illuminant and observer:

```
>>> wp_d65 = WhitePoint(illuminant="D30", observer=2)
CIEIlluminantError: The input illuminant name is not a valid CIE standard illuminant
>>> wp_d65 = WhitePoint(illuminant="D65", observer=20)
CIEObserverError: The observer should be a CIE standard 1931 or 1964 observer (2º or 10º)
```

Alert

It is only allowed to create instances for a CIE standard illuminant and observer. Otherwise, a CIEIlluminantError or CIEObserverError is raised.

5.4.2 Attributes

The class *attributes* are: *type* (returns a `str` with the description of the main class), *subtype* (returns a `str` with the description of the object), *colour_space* (returns a `str` with the colour space), *name_id* (returns a `str` with the sample id), *illuminant* (returns the *Illuminant*, *IlluminantFromCCT* or *MeasuredIlluminant* instance), *coordinates* (returns a `list` with the three coordinates of the white point) and *observer* (returns an *Observer* instance).

```
>>> print(wp_d65.type)
Colour Object
>>> print(wp_d65.colour_space()) # subtype
CIE XYZ
>>> print(wp_d65.subtype)
WhitePoint XYZ
>>> print(wp_d65.name_id)
WhitePoint for illuminant D65 and 2 º observer
>>> print(wp_d65.coordinates)
[95.04, 100.0, 108.88]
>>> print(wp_d65.illuminant) # str method
Illuminant object: CIE D65 standard illuminant
>>> print(wp_d65.observer) # str method
2º standard observer (CIE 1931)
>>> print(wp_d65.get_sample()) # colour data as dict
{'WhitePoint for illuminant D65 and 2 º observer': [95.04, 100.0, 108.88]}
```

Info

Theoretical data are available only for the CIE illuminants: A, C, D50, D55, D65 and D75. For all the remaining illuminants implemented, the XYZ values of the illuminant white point are computed.

5.4.3 Methods

Colour Space Conversion (CSC):

Only conversions to CIE xyY and CIE uvY colour spaces are allowed for a *WhitePoint* object.

```
>>> wp_d65_xyY = wp_d65.to_xyY() # returns a CIExyY class object
>>> type(wp_d65_xyY)
>>> coolpi.colour.cie_colour_spectral.CIExyY
```

After applying the colour conversion, a new *Colour* object is returned.

The alternative way to obtain the colour coordinates in the desired output colour space is:

```
>>> print("xyY   = ", wp_d65.to_xyY().coordinates)
xyY   =  [0.3127138720715978, 0.3290339563042906, 100.0]
>>> print("uvY   = ", wp_d65.to_uvY().coordinates)
uvY   =  [0.19782690146122145, 0.46834020232296747, 100.0]
```

__str__:

> **__str__**
>
> *Returns the string representation of the object. This method is called when the `print()` or `str()` function is invoked on an object.*

```
>>> print(wp_d65) #str method
WhitePoint for D65 illuminant and 2º observer: Xn=95.04, Yn=100.0, Zn=108.88
```

5.4.4 Plot

> ***WhitePoint.plot_xy_white_point(show_figure=True, save_figure=False, output_path=None)***
>
> *Method to create and display the x,y into the CIE 1931 x,y Chromaticity Diagram using Matplotlib.*
>
> *Parameters:*
>
> *show_figure:* `bool` *If True, the plot is shown. Default:* `True`*.*
>
> *save_figure:* `bool` *If True, the figure is saved. Default:* `False`*.*
>
> *output_path:* `os` *path for the ouput figure. Default:* `None`*.*

WhitePoint objects can be represented in the CIE 1931 x,y Chromaticity Diagram as follows:

```
>>> wp_d65.plot()
```

In addition, it is posible to save the figure setting True the opcion `save_figure` and introducing a valid `output_path`.

```
>>> wp_d65.plot(show_figure = False, save_figure = True, output_path = path_figure)
```

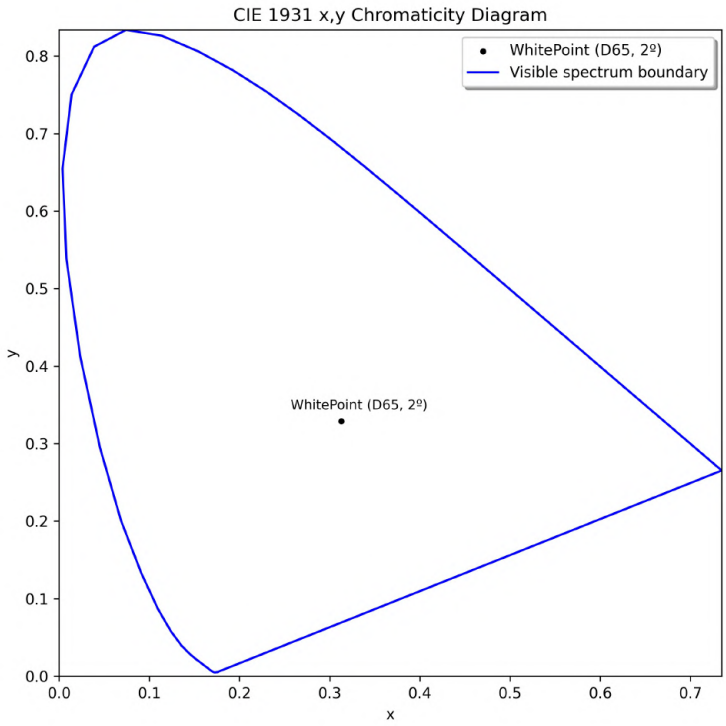

Figure 44: *WhitePoint Plot*

5.5 SpectralColour

class SpectralColour

SpectralColour(name_id, nm_range, nm_interval, lambda_values, illuminant="D65", observer=2)

.as_diagonal_array()

.set_into_visible_range_spectrum(visible_nm_range, visible_nm_interval)

.scale_lambda_values()

.to_XYZ(visible)

.plot(show_figure, save_figure, output_path)

The *SpectralColour* class represents a colour sample defined by its spectral data referred to a CIE illuminant.

5.5.1 Create an instance

SpectralColour(name_id, nm_range, nm_interval, lambda_values, illuminant="D65", observer=2)

Parameters:

name_id: `str` *Sample ID.*

nm_range: `list` *spectral nm range [min, max].*

nm_interval: `int` *spectral interval in nm.*

lambda_values: `list` *spectral data.*

illuminant: `str`, `Illuminant` *CIE illuminant. Default: "D65"`*

observer: `str`, `int`, `Observer` *CIE standard observer. Default: 2.*

To *create an instance* of the *SpectralColour* class, simply enter the required parameters as follows:

```
>>> from coolpi.colour.cie_colour_spectral import SpectralColour
>>> lambda_data = [6.0475, 6.0475, 6.0475, 6.0475, 6.0475, 5.97, 5.8775, 5.845, 5.8625,
    5.9225, 5.99, 6.06, 6.245, 6.49, 7.175, 9.435, 13.045, 15.6575, 17.1975, 19.415, 24.47,
    33.2475, 44.34, 53.75, 58.15, 58.6625, 58.1725, 57.9075, 57.5325, 57.32, 57.91, 59.6,
    61.6025, 63.5, 64.97, 64.97, 64.97, 64.97, 64.97]
>>> sample_spc = SpectralColour(name_id="Sample_1", nm_range=[360, 740], nm_interval=10,
    lambda_values= lambda_data, illuminant="D65", observer=2)
```

The *SpectralColour* should be specified for a valid CIE standard illuminant and observer:

```
>>> sample_spc = SpectralColour(name_id="Sample_1", nm_range=[360, 740], nm_interval=10,
    lambda_values= lambda_data, illuminant="D30", observer=2)
CIEIlluminantError: The input illuminant is not a valid CIE standard illuminant
>>> sample_spc = SpectralColour(name_id="Sample_1", nm_range=[360, 740], nm_interval=10,
    lambda_values= lambda_data, illuminant="D65", observer=20)
CIEObserverError: The observer should be a CIE standard 1931 or 1964 observer (2º or 10º)
```

⚠ **Alert**

It is only allowed to create SpectralColour instances for a valid CIE standard illuminant and observer. Otherwise, a CIEIlluminantError or CIEObserverError is raised.

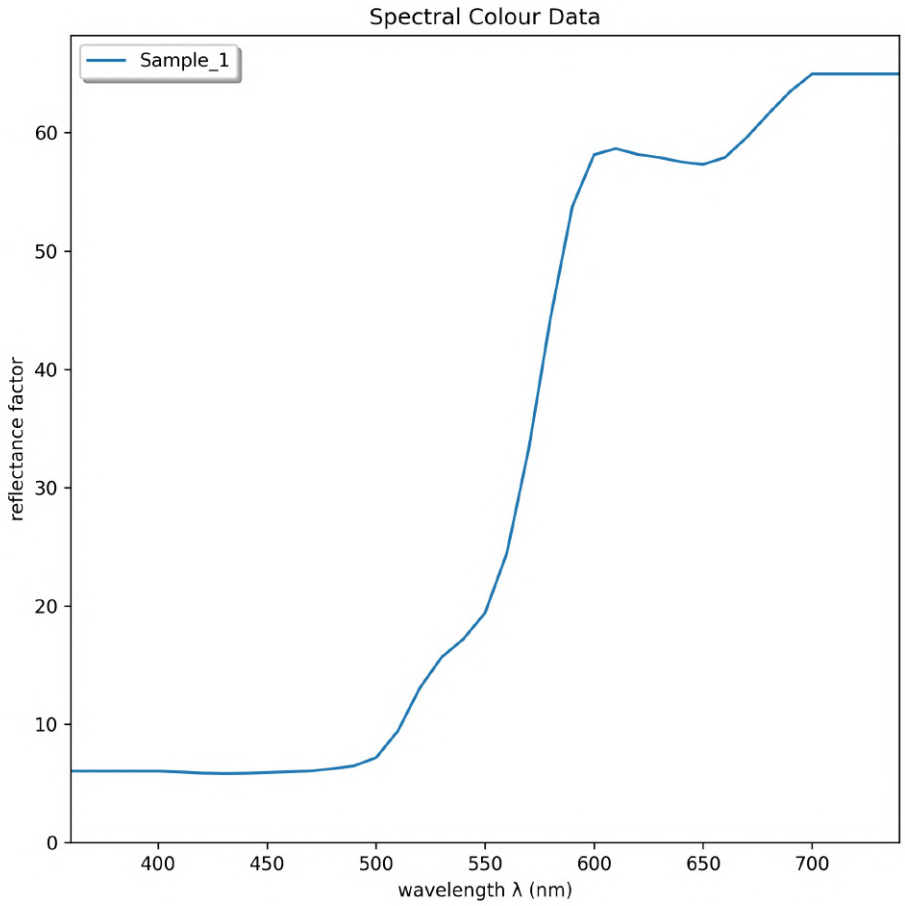

Figure 45: *SpectralColour*

5.5.2 Attributes

The class *attributes* are: *type* (returns a `str` with the description of the main class), *subtype* (returns a `str` with the description of the object), *name_id* (returns a `str` with the sample id), *illuminant* (returns the *Illuminant* instance), *observer* (returns the *Observer* instance), *nm_range* (returns a `list` with the min and max range value in *nm*), *nm_interval* (returns an `int` with the lambda interval in *nm*), *lambda_values* (returns a `list` with the spectral data) and *scaled* (returns a `bool`).

```
>>> print(sample_spc.type)
Spectral Object
>>> print(sample_spc.subtype)
SpectralColour data referred to a CIE illuminant
>>> print(sample_spc.name_id)
Sample_1
>>> print(sample_spc.illuminant) # str method
Illuminant object: CIE D65 standard illuminant
>>> print(sample_spc.observer) # str method
2º standard observer (CIE 1931)
>>> print(sample_spc.nm_range)
[360, 740]
>>> print(sample_spc.nm_interval)
10
>>> print(sample_spc.lambda_values)
[6.0475, 6.0475, 6.0475, 6.0475, 6.0475, 5.97, 5.8775, 5.845, 5.8625, 5.9225, 5.99,
6.06, 6.245, 6.49, 7.175, 9.435, 13.045, 15.6575, 17.1975, 19.415, 24.47, 33.2475,
44.34, 53.75, 58.15, 58.6625, 58.1725, 57.9075, 57.5325, 57.32, 57.91, 59.6, 61.6025,
63.5, 64.97, 64.97, 64.97, 64.97, 64.97]
>>> print(sample_spc.scaled)
False
```

5.5.3 Methods

As a numpy diagonal array:

> ***SpectralColour.as_diagonal_array()***
>
> *Method to get the spectral data as a numpy diagonal array.*
>
> *Returns:*
>
> *as_diag:* `numpy.ndarray` *spectral data as a diagonal array.*

To get the spectral data as a diagonal array:

```
sample_spc_as_np = sample_spc.as_diagonal_array()
type(sample_spc_as_np)
numpy.ndarray
```

```
>>> print(sample_spc_as_np.shape)
>>> (39, 39)
```

To set the spectral data into the visible spectrum:

> ***SpectralColour.set_into_visible_range_spectrum(visible_nm_range=[400,700], visible_nm_interval=10)***
>
> *Method to set the spectral data into the visible spectrum.*

Parameters:

visible_nm_range: `list` *[min, max] range in nm. Default: [400,700].*

visible_nm_interval: `int` *Interval in nm. Default: 10.*

To set the *SpectralColour* spectral data into the visible range spectrum:

```
>>> sample_spc.set_into_visible_range_spectrum(visible_nm_range=[400,700],
    visible_nm_interval=10)
```

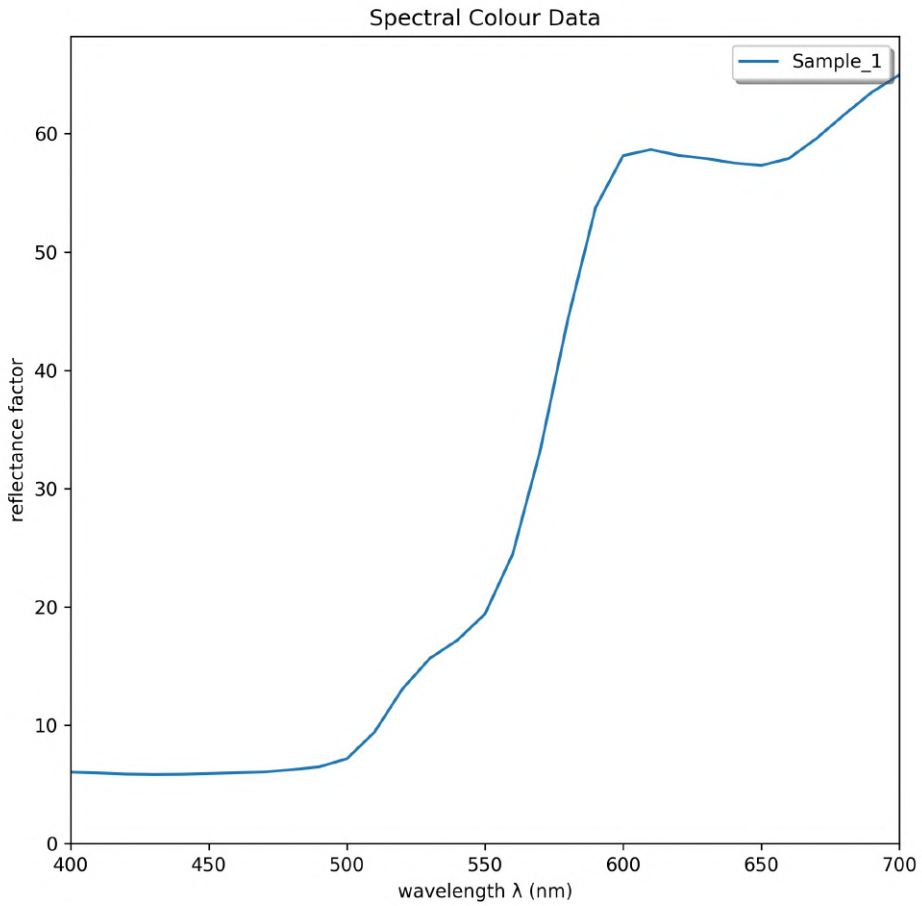

Figure 46: *SpectralColour (visible spectrum)*

To scale the spectral data into the range (0,1):

MeasuredIlluminant.scale_lambda_values()

Method to scale the spectral data into the range (0,1).

To scale the spectral data into the range (0,1):

```
>>> sample_spc.scale_lambda_values()
>>> print(sample_spc.scaled)
True
```

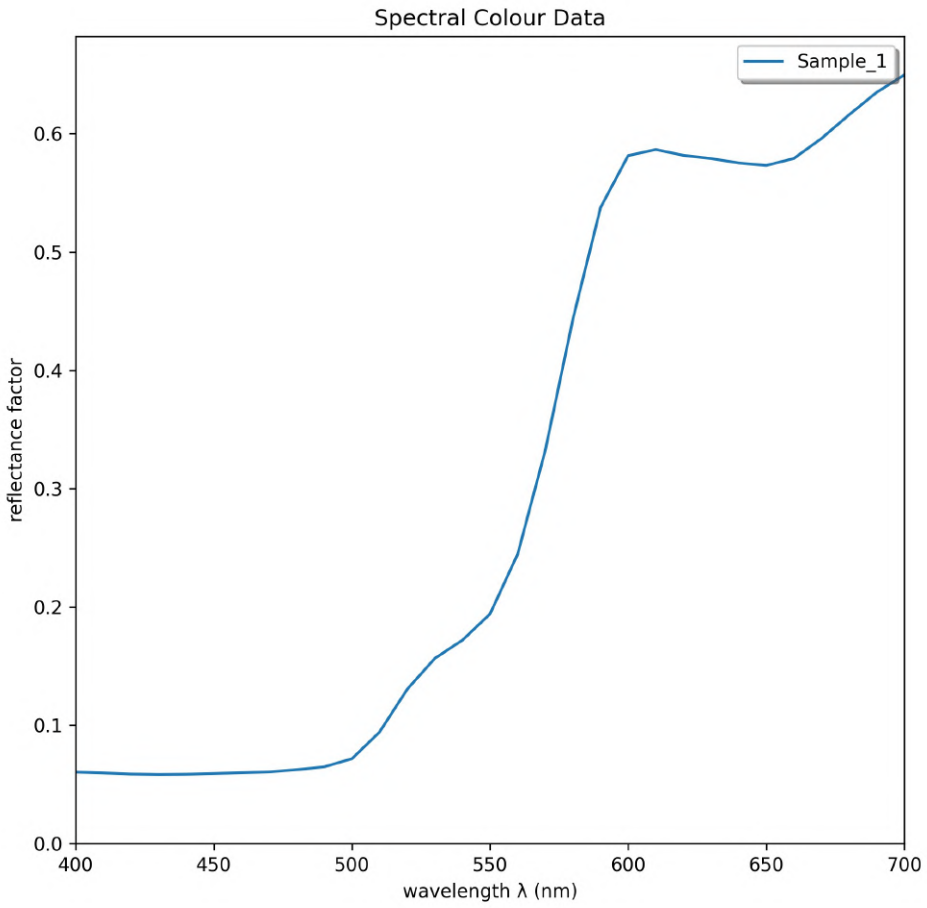

Figure 47: *SpectralColor (scaled)*

Spectral data to CIE XYZ:

SpectralColour.to_XYZ(visible = False)

Method to obtain the CIE XYZ tristimulus values from spectral data.

Parameter:

visible: `bool` *If True, use only the visible spectrum data. Default:* `False`

Returns:

X,Y,Z: `float` *CIE XYZ coordinates.*

To compute the CIE XYZ from the spectral data:

Using all the spectral data:

```
>>> X, Y, Z = sample_spc.to_XYZ() # returns a CIEXYZ instance
>>> print(X, Y, Z)
36.87034965110855, 29.52534750336318, 6.768737346875687
```

Using only the spectral data into the visible range spectrum:

```
>>> X, Y, Z = sample_spc.to_XYZ(visible=True)
>>> print(X, Y, Z)
36.8208393765158 29.51360348192101 6.757552323290141
```

__str__:

> **__str__**
>
> *Returns the string representation of the object. This method is called when the `print()` or `str()` function is invoked on an object.*

```
>>> print(sample_spc) # str method
SpectralColour object: Sample_1
```

5.5.4 Plot

> ***SpectralColour.plot(show_figure=True, save_figure=False, output_path=None)***
>
> *Method to create and display the spectral data of a colour sample using Matplotlib.*
>
> *Parameters:*
>
> *show_figure:* `bool` *If True, the plot is shown. Default:* `True`.
>
> *save_figure:* `bool` *If True, the figure is saved. Default:* `False`.
>
> *output_path:* `os` *path for the ouput figure. Default: `None.*

SpectralColour objects can be represented as follows:

```
>>> sample_spc.plot()
```

In addition, it is posible to save the figure setting `True` the opcion `save_figure` and introducing a valid `output_path`.

```
>>> sample_spc.plot(show_figure = False, save_figure = True, output_path = path_figure)
```

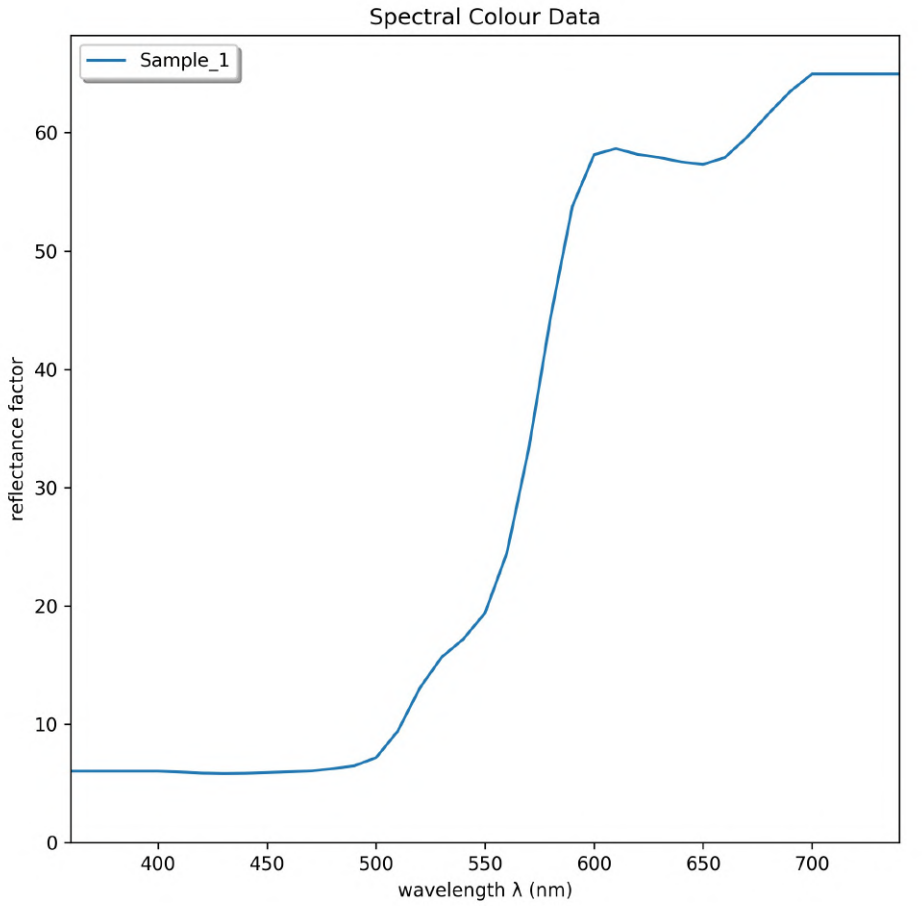

Figure 48: *SpectralColour Plot*

5.6 Reflectance

class Reflectance

Reflectance(name_id, nm_range, nm_interval, lambda_values, illuminant="D65", observer=2, metadata={})

.as_diagonal_array()

.set_into_visible_range_spectrum(visible_nm_range, visible_nm_interval)

.scale_lambda_values()

.to_XYZ(visible)

.plot(show_figure, save_figure, output_path)

The *Reflectance* class represents a colour sample defined by its spectral data

measured by a spectrophotometer and referred to a given illuminant.

5.6.1 Create an instance

Reflectance(name_id, nm_range, nm_interval, lambda_values, illuminant="D65", observer=2, metadata={})

Parameters:

name_id: `str` *Sample ID.*

nm_range: `list` *spectral nm range [min, max].*

nm_interval: `int` *spectral interval in nm.*

lambda_values: `list` *spectral data.*

illuminant: `dict` *CIE illuminant. Default: "D65".*

observer: `str, int, Observer` *CIE standard observer. Default: 2.*

metadata: `dict` *Instrument measurement information. Default: {}.*

To *create an instance* of the *Reflectance* class, simply enter the required parameters as follows:

```
>>> from coolpi.colour.cie_colour_spectral import Reflectance
>>> lambda_data = [6.0475, 6.0475, 6.0475, 6.0475, 6.0475, 5.97, 5.8775, 5.845,
    5.8625, 5.9225, 5.99, 6.06, 6.245, 6.49, 7.175, 9.435, 13.045, 15.6575, 17.1975,
    19.415, 24.47, 33.2475, 44.34, 53.75, 58.15, 58.6625, 58.1725, 57.9075, 57.5325,
    57.32, 57.91, 59.6, 61.6025, 63.5, 64.97, 64.97, 64.97, 64.97, 64.97]
>>> rfc_metadata = {"NameColorChart": "Calibrite colorchecker CLASSIC",
    "Manufacturer":["Calibrite", "Made in USA", 850028833087, 2021],
    "Measurement Date": [[2022, 3, 11], [10, 13, 40]], "Instrument":"Konica Minolta CM-
600d",
    "Illuminant": "D65", "Observer": 2, "Geometry": "di:8, de:8", "Specular
Component":"SCI",
    "Measurement Area": "MAV(8mm)"}
>>> from coolpi.colour.cie_colour_spectral import IlluminantFromCCT
>>> cct_6500 = IlluminantFromCCT(6500)
>>> sample_rfc = Reflectance(name_id="Sample_1", nm_range=[360, 740], nm_interval=10,
    lambda_values= lambda_data, illuminant=cct_6500, observer=2)
```

Info

The Reflectance metadata can be set separately with the method:

```
    .set_instrument_measurement_as_metadata(metadata)
```

The *Reflectance* can be instantiated not only for a valid CIE standard illuminant (`str`) or *Illuminant* object, but for *IlluminantFromCCT* or *MeasuredIlluminant* objects.

In case of specifying a not valid CIE standard illuminant or observer:

```
>>> sample_rfc = Reflectance(name_id="Sample_1", nm_range=[360, 740], nm_interval=10,
    lambda_values= lambda_data, illuminant="D30", observer=2)
CIEIlluminantError: The input illuminant is not a valid CIE standard illuminant
>>> sample_rfc = Reflectance(name_id="Sample_1", nm_range=[360, 740], nm_interval=10,
    lambda_values= lambda_data, illuminant="D65", observer=20)
CIEObserverError: The observer should be a CIE standard 1931 or 1964 observer (2º or 10º)
```

Alert

It is only allowed to create Reflectance instances for a valid CIE standard observer. Otherwise, a CIEObserverError is raised. In case of specifying a not valid CIE standard illuminant, a CIEIlluminantError is raised.

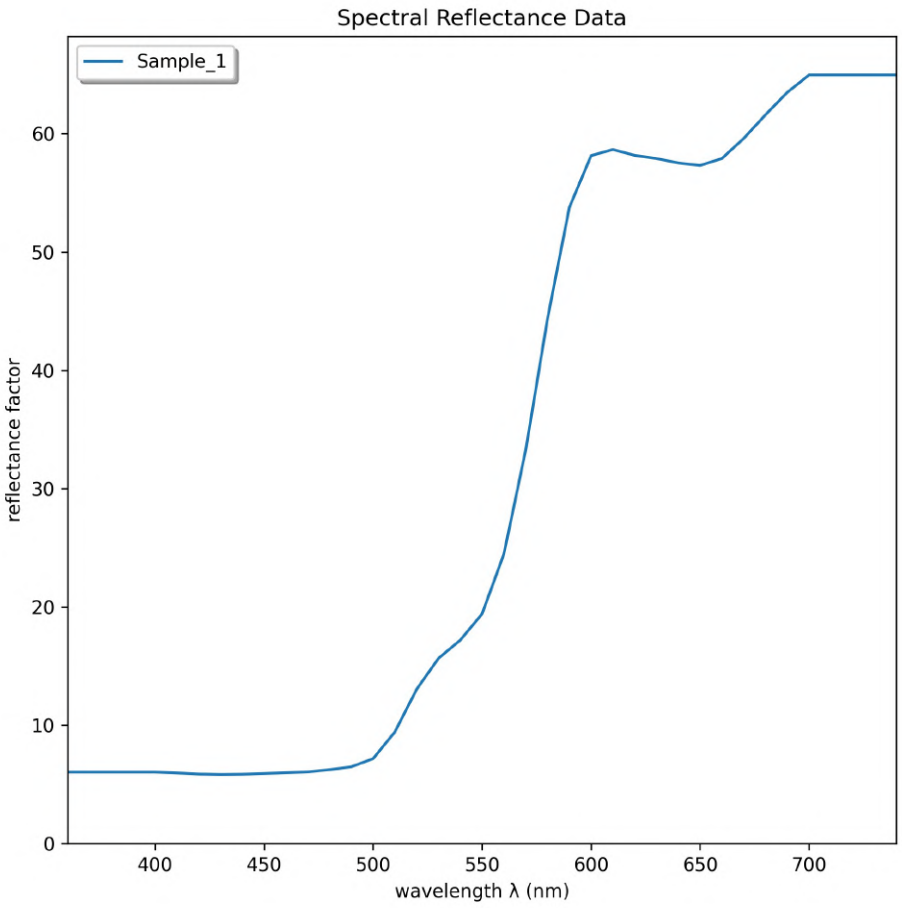

Figure 49: *Reflectance*

5.6.2 Attributes

The class *attributes* are: *type* (returns a `str` with the description of the main class), *subtype* (returns a `str` with the description of the object), *name_id* (returns a `str` with the sample id), *illuminant* (returns the *Illuminant*, *IlluminantFromCCT* or *MeasuredIlluminant* instance), *observer* (returns the *Observer* instance), *nm_range* (returns a `list` with the min and max range value in *nm*), *nm_interval* (returns an `int` with the lambda interval in *nm*), *lambda_values* (returns a `list` with the spectral data), *scaled* (returns a `bool`) and *metadata* (returns a `dict` with the measurement details).

```
>>> print(sample_rfc.type)
Spectral Object
>>> print(sample_rfc.subtype)
Reflectance data of a colour sample
>>> print(sample_rfc.name_id)
Sample_1
>>> print(sample_rfc.illuminant)  # str method
IlluminantFromCCT object: CCT 6500 º K
>>> print(sample_rfc.observer) # str method
2º standard observer (CIE 1931)
>>> print(sample_rfc.nm_range)
[360, 740]
>>> print(sample_rfc.nm_interval)
10
>>> print(sample_rfc.lambda_values)
[6.0475, 6.0475, 6.0475, 6.0475, 6.0475, 5.97, 5.8775, 5.845, 5.8625, 5.9225,
5.99, 6.06, 6.245, 6.49, 7.175, 9.435, 13.045, 15.6575, 17.1975, 19.415, 24.47,
33.2475, 44.34, 53.75, 58.15, 58.6625, 58.1725, 57.9075, 57.5325, 57.32, 57.91,
59.6, 61.6025, 63.5, 64.97, 64.97, 64.97, 64.97, 64.97]
>>> print(sample_rfc.scaled)
False
>>> print(sample_rfc.metadata.keys())
dict_keys(['NameColorChart', 'Manufacturer', 'Measurement Date', 'Instrument',
'Illuminant', 'Observer', 'Geometry', 'Specular Component', 'Measurement Area'])
```

5.6.3 Methods

As a numpy diagonal array:

Reflectance.as_diagonal_array()

Method to get the reflectance data as a numpy diagonal array.

Returns:

as_diag: `numpy.ndarray` *reflectance data as a diagonal array.*

To get the reflectance data as a diagonal array:

```
>>> sample_rfc_as_np = sample_rfc.as_diagonal_array()
>>> type(sample_rfc_as_np)
numpy.ndarray
```

```
>>> print(sample_rfc_as_np.shape)
(39, 39)
```

To set the reflectance data into the visible spectrum:

> ***Reflectance.set_into_visible_range_spectrum(visible_nm_range=[400,700], visible_nm_interval=10)***
>
> *Method to set the reflectance data into the visible spectrum.*
>
> *Parameters:*
>
> *visible_nm_range:* `list` *[min, max] range in nm. Default: [400,700].*
>
> *visible_nm_interval:* `int` *Interval in nm. Default: 10.*

To set the reflectance data into the visible range spectrum:

```
>>> sample_rfc.set_into_visible_range_spectrum(visible_nm_range=[400,700],
    visible_nm_interval=10)
```

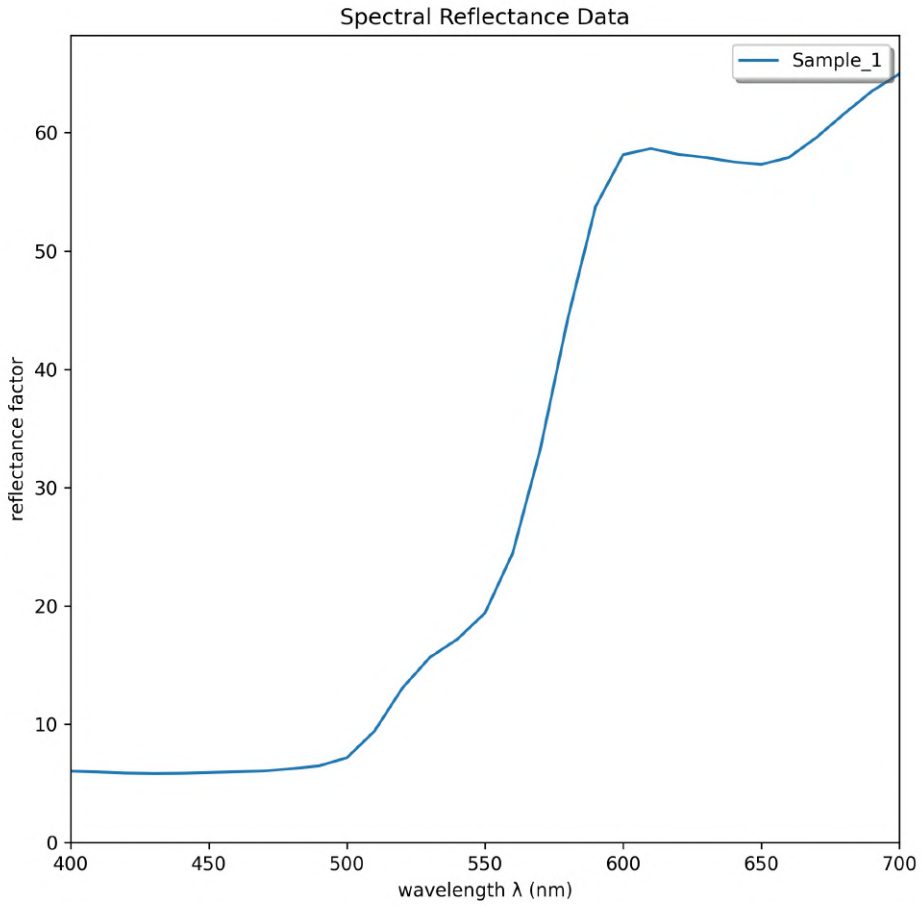

Figure 50: *Reflectance (visible spectrum)*

To scale the reflectance data into the range (0, 1):

> ***Reflectance.scale_lambda_values()***
>
> *Method to scale the reflectance data into the range (0,1).*

To scale the reflectance data into the range (0,1):

```
>>> sample_rfc.scale_lambda_values()
>>> print(sample_rfc.lambdscaleda_values)
True
```

Figure 51: *Reflectance (scaled)*

Reflectance data to CIE XYZ:

> ***Reflectance.to_XYZ(visible=False)***
>
> *Method to obtain the XYZ tristimulus values from the reflectance data.*

Parameter:

visible: `bool` *If True, use only the visible spectrum data. Default:* `False`.

Returns:

X,Y,Z,: `float` *CIE XYZ coordinates.*

To compute the CIE XYZ from the reflectance data:

Using all the reflectance data:

```
>>> X, Y, Z = sample_rfc.to_XYZ()
>>> print(X, Y, Z)
36.87034965110855 29.52534750336318 6.768737346875687
```

Using only the reflectance data into the visible range spectrum:

```
X, Y, Z= sample_rfc.to_XYZ(visible=True)
print(X, Y, Z)
36.8208393765158 29.51360348192101 6.757552323290141
```

`__str__`:

`__str__`

Returns the string representation of the object. This method is called when the `print()` *or* `str()` *function is invoked on an object.*

```
>>> print(sample_rfc) # str method
Reflectance object: Sample_1
```

5.6.4 Plot

Reflectance.plot(show_figure=True, save_figure=False, output_path=None)

Method to create and display the reflectance data of a colour sample using Matplotlib.

Parameters:

show_figure: `bool` *If True, the plot is shown. Default:* `True`.

save_figure: `bool` *If True, the figure is saved. Default:* `False`.

output_path: `os` *path for the ouput figure. Default:* `None`.

Reflectance objects can be represented as follows:

```
>>> sample_rfc.plot()
```

In addition, it is posible to save the figure setting `True` the opcion `save_figure` and introducing a valid `output_path`.

```
>>> sample_rfc.plot(show_figure = False, save_figure = True, output_path = path_figure)
```

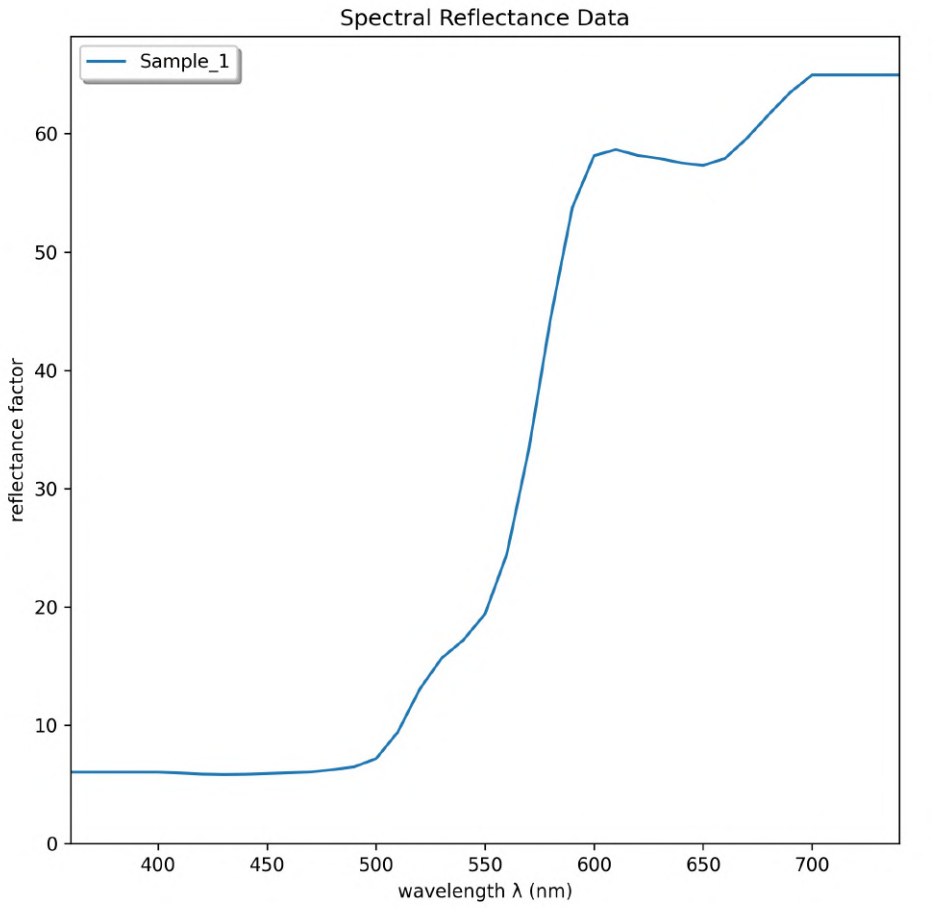

Figure 52: *Reflectance Plot*

6 CSC

The Colour Space Conversion (CSC) functions are implemented as class methods for *Colour* objects. We highly recommend the use of objects to perform the transform between colour spaces. However, the conversion functions can be used directly in Python.

The following figure shows the relationship between direct and reverse conversion that can be carried out between colour spaces by using the well-known CIE formulation (CIE, 2018)). All other non-direct conversions should be done by using auxiliary colour conversions until the desired output colour space is reached.

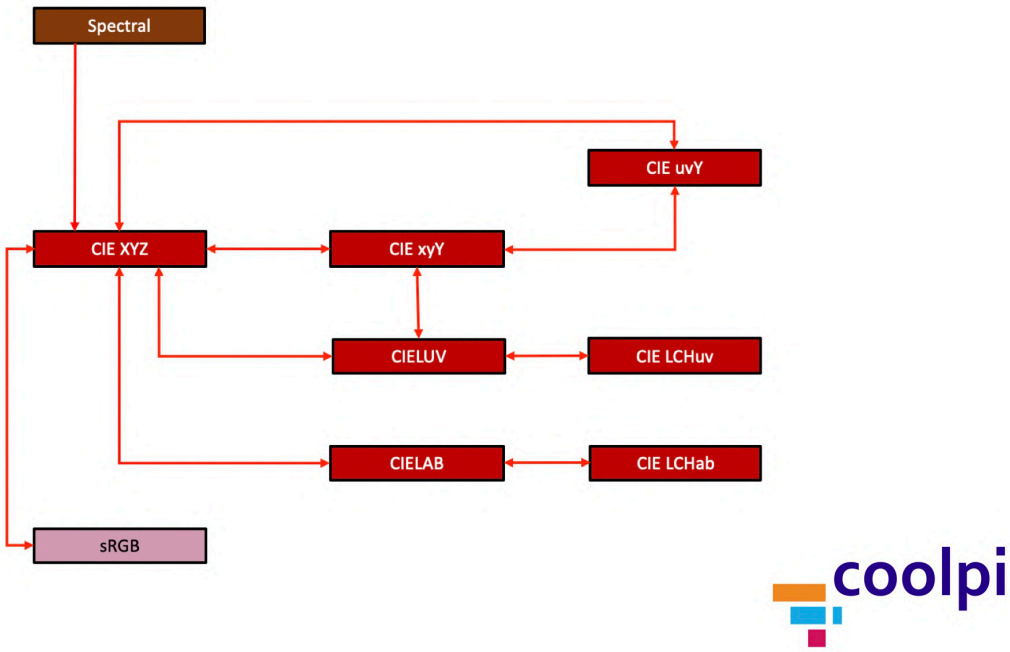

Figure 53: *Colour Space Conversion: Relationship between colour spaces*

The Colour Space Conversion Module can be imported as follows:

```
>>> import coolpi.colour.colour_space_conversion as csc
```

Alternatively, the specific function can be imported to perform the desired conversion between colour spaces as follows:

```
>>> from coolpi.colour.colour_space_conversion import  XYZ_to_xyY
```

Practical use of *CSC* functions

Users are encouraged to previously take a look at the Jupyter Notebooks:

02b_CSC_Colour_Space_Conversion.ipynb

02c_CSC-Test_data_(Ohta&Robertson_2005).ipynb.ipynb

6.1 CIE XYZ to CIE xyY

csc.XYZ_to_xyY(X, Y, Z)

Function to compute the transformation between CIE XYZ and CIE xyY colour spaces.

Parameters:

X, Y, Z: `float` *CIE XYZ tristimulus values.*

Returns:

x, y, Y: `float` *CIE xyY coordinates.*

Chromaticity coordinates can be computed from CIE XYZ tristimulus values as follows:

$$x = \frac{X}{X+Y+Z}$$

$$y = \frac{Y}{X+Y+Z}$$

$$z = \frac{Z}{X+Y+Z}$$

Because of the relation $x + y + z = 1$, usually only the x and y coordinates are used.

Info

CIE 015:2018. 7.3 Calculation of chromaticity coordinates (p.26) (CIE, 2018)).

Example:

Given a colour sample in CIE XYZ colour space:

```
>>> x, y, Y = csc.XYZ_to_xyY(X=36.821, Y=29.514, Z=6.758)
>>> print(x, y, Y)
0.5037554895817657 0.40378695634328876 29.514
```

6.1.1 Reverse transform

csc.xyY_to_XYZ(x, y, Y)

Function to compute the transformation between CIE xyY and CIE XYZ colour spaces.

Parameters:

x, y, Y: `float` *CIE xyY coordinates.*

Returns:

X, Y, Z: `float` *CIE XYZ tristimulus values.*

CIE XYZ tristimulus values can be obtained from x,y chromaticity coordinates using the following equations:

$$X = Y \cdot \frac{x}{y}$$

$$Y = Y$$

$$Z = Y \cdot \frac{1-x-y}{y}$$

Info

Kruschwitz, J. D. 2018. Field guide to colorimetry and fundamental color modeling. Society of Photo-Optical Instrumentation Engineers. SPIE. p. 18 (Kruschwitz, 2018).

Example:

Given a colour sample in CIE xyY colour space:

```
>>> X, Y, Z = csc.xyY_to_XYZ(x=0.503756, y=0.403787,Y=29.514)
>>> print(X, Y, Z)
36.821033326976846 29.514 6.75795877034179
```

6.2 CIE XYZ to CIE u'v'Y

csc.XYZ_to_uvY(X, Y, Z)

Function to compute the transformation between CIE XYZ and CIE 1976 u'v' UCS colour spaces.

Parameters:

X, Y, Z: `float` *CIE XYZ tristimulus values.*

Returns:

u, v, Y: `float` *CIE u',v',Y coordinates.*

The uniform chromaticity scale (UCS) coordinates can be obtained directly from CIE XYZ tristimulus values as follows:

$$u' = \frac{4X}{(X+15Y+3Z)}$$

$$v' = \frac{9Y}{(X+15Y+3Z)}$$

where X, Y, Z are the tristimulus values. The third chromaticity coordinate w' is equal to $(1 - u' - v')$.

For practical purposes, the Y tristimulus value is added to the u' and v' coordinates.

Info

CIE015:2018. 8.1. CIE 1976 uniform chromaticity scale (UCS) diagram. Eq. 8.1. (p.27) (CIE, 2018))

Example:

Given a colour sample in CIE XYZ colour space:

```
>>> u, v, Y = csc.XYZ_to_uvY(X=36.821, Y=29.514, Z=6.758)
>>> print(u, v, Y)
0.29468292634127313 0.5314592691149549 29.514
```

6.2.1 Reverse transform

csc.uvY_to_XYZ(u, v, Y)

Function to compute the transformation between CIE 1976 $u'v'$ and CIE XYZ colour spaces.

Solved using a linear equation system.

Parameters:

u, v, Y: `float` *CIE u',v',Y coordinates,*

Returns:

X, Y, Z: `float` *CIE XYZ tristimulus values.*

The transformation between the CIE u'v'Y and CIE XYZ colour spaces can be carried out by simply solving the linear system of the equations formed with the input tristimulus values.

Info

For further details about the process, please refer to the source code.

Example:

Given a colour sample in CIE u'v'Y colour space:

```
>>> X, Y, Z = csc.uvY_to_XYZ(u=0.294683, v=0.531459, Y=29.514)
>>> print(X, Y, Z)
36.82102784880865 29.514 6.758075079168849
```

6.3 CIE XYZ to CIELAB

csc.XYZ_to_LAB(X, Y, Z, Xn=95.04, Yn=100.00, Zn=108.88)

Function to compute the transformation between CIE XYZ and CIELAB colour spaces.

A reference X_n, Y_n, Z_n *WhitePoint is required to perform this conversion.*

By default, the white point of the CIE illuminant D65 is used.

Parameters:

X, Y, Z: `float` *CIE XYZ tristimulus value.*

X_n, Y_n, Z_n*:* `float` *White point,* $Y_n = 100$*. Default: D65.*

Returns:

L, a, b: `float` *CIELAB coordinate.s*

The CIELAB colour space is a three-dimensional, approximately uniform, colour space produced by plotting in rectangular coordinates, L^*, a^*, b^* quantities defined by the

equations:

$$L^* = 116 \cdot f(\frac{Y}{Y_n}) - 16$$

$$a^* = 500 \cdot [f(\frac{X}{X_n}) - f(\frac{Y}{Y_n})]$$

$$b^* = 200 \cdot [f(\frac{Y}{Y_n}) - f(\frac{Z}{Z_n})]$$

where

$$f(\frac{X}{X_n}) = (\frac{X}{X_n})^{\frac{1}{3}} \quad \text{if} \quad (\frac{X}{X_n}) > (\frac{24}{116})^3$$

$$f(\frac{X}{X_n}) = (\frac{841}{108}) \cdot (\frac{X}{X_n}) + (\frac{16}{116}) \quad \text{if} \quad (\frac{X}{X_n}) \leq (\frac{24}{116})^3$$

and

$$f(\frac{Y}{Y_n}) = (\frac{Y}{Y_n})^{\frac{1}{3}} \quad \text{if} \quad (\frac{Y}{Y_n}) > (\frac{24}{116})^3$$

$$f(\frac{Y}{Y_n}) = (\frac{841}{108}) \cdot (\frac{Y}{Y_n}) + (\frac{16}{116}) \quad \text{if} \quad (\frac{Y}{Y_n}) \leq (\frac{24}{116})^3$$

and

$$f(\frac{Z}{Z_n}) = (\frac{Z}{Z_n})^{\frac{1}{3}} \quad \text{if} \quad (\frac{Z}{Z_n}) > (\frac{24}{116})^3$$

$$f(\frac{Z}{Z_n}) = (\frac{841}{108}) \cdot (\frac{Z}{Z_n}) + (\frac{16}{116}) \quad \text{if} \quad (\frac{Z}{Z_n}) \leq (\frac{24}{116})^3$$

where X, Y, Z are the tristimulus values of the test object colour stimulus considered, and X_n, Y_n, Z_n are the tristimulus values of a specific white object colour stimulus or the light source with $Y_n = 100$.

Info

CIE015:2018. 8.2.1. CIELAB colour space. 8.2.1.1. Basic coordinates. Eq. 8.3-8.11. (p.28) (CIE, 2018)

Example:

Given a colour sample in CIE XYZ colour space:

```
>>> L, a, b = csc.XYZ_to_LAB(X=36.821, Y=29.514, Z=6.7589) # Default D65
>>> print(L, a, b)
61.23260441015891 31.602680131515292 53.969245402697965
```

6.3.1 Reverse transform

csc.LAB_to_XYZ(L, a, b, Xn=95.04, Yn=100.00, Zn=108.88)

Function to compute the transformation between CIELAB and CIE XYZ colour spaces.

A reference X_n, Y_n, Z_n WhitePoint is required to perform this conversion.

Parameters:

L, a, b: `float` *CIELAB coordinates.*

X_n, Y_n, Z_n: `float` *White point, $Y_n = 100$. Default: D65.*

Returns:

X, Y, Z: `float` *CIE XYZ tristimulus values.*

The transformation between the L^*, a^*, b^* coordinates to the X, Y, Z tristimulus values is given by the following equations:

$$f(\frac{Y}{Y_n}) = \frac{(L^*+16)}{116}$$

$$f(\frac{X}{X_n}) = \frac{a^*}{500} + f(\frac{Y}{Y_n})$$

$$f(\frac{Z}{Z_n}) = f(\frac{Y}{Y_n}) - \frac{b^*}{200}$$

where

$$X = X_n \cdot [f(\frac{X}{X_n})]^3 \qquad \text{if} \quad f(\frac{X}{X_n}) > (\frac{24}{116})$$

$$X = X_n \cdot [f(\frac{X}{X_n}) - \frac{16}{116}]\frac{108}{481} \qquad \text{if} \quad f(\frac{X}{X_n}) \leq (\frac{24}{116})$$

and

$$Y = Y_n \cdot [f(\frac{Y}{Y_n})]^3 \qquad \text{if} \quad f(\frac{Y}{Y_n}) > (\frac{24}{116}) \quad \text{or } L^* > 8$$

$$Y = Y_n \cdot [f(\frac{Y}{Y_n}) - \frac{16}{116}]\frac{108}{481} \qquad \text{if} \quad f(\frac{Y}{Y_n}) \leq (\frac{24}{116}) \quad \text{or } L^* \leq 8$$

and

$$Z = Z_n \cdot [f(\frac{Z}{Z_n})]^3 \qquad \text{if} \quad f(\frac{Z}{Z_n}) > (\frac{24}{116})$$

$$Z = Z_n \cdot [f(\frac{Z}{Z_n}) - \frac{16}{116}]\frac{108}{481} \qquad \text{if} \quad f(\frac{Z}{Z_n}) \leq (\frac{24}{116})$$

where L^*, a^*, b^* are the values of the test object colour stimulus considered, and X_n, Y_n, Z_n are the tristimulus values of a specific white object colour stimulus or the light source with $Y_n = 100$.

Info

CIE015:2018. Annex C. Reverse transformation from values L^*, a^*, b^* to tristimulus values X, Y, Z (p.89) (CIE, 2018)

Example:

Given a colour sample in CIELAB colour space:

```
>>> X, Y, Z = csc.LAB_to_XYZ(L=61.232604, a=31.60268, b=53.969245) # Default D65
>>> print(X, Y, Z)
10.233974030270863 5.870981271147834 22.108933775862585
```

6.4 CIE XYZ to CIELUV

csc.XYZ_to_LUV(X, Y, Z, Xn=95.04, Yn=100.00, Zn=108.88)

Function to compute the transformation between CIE XYZ and CIELUV colour spaces.

A reference X_n, Y_n, Z_n WhitePoint is required to perform this conversion.

Parameters:

X, Y, Z: `float` *CIE XYZ tristimulus values.*

X_n, Y_n, Z_n: `float` *White point, $Y_n = 100$. Default: D65.*

Returns:

L, U, V: `float` *CIELUV coordinates.*

The CIE 1976 L^*, u^*, v^*, or simply CIELUV colour space is a three-dimensional, approximately uniform, colour space produced by plotting in rectangular coordinates, L^*, u^*, v^* quantities defined by the equations:

$$L^* = 116 \cdot f(\frac{Y}{Y_n}) - 16$$

where

$$f(\frac{Y}{Y_n}) = (\frac{Y}{Y_n})^{\frac{1}{3}} \qquad \text{if} \quad (\frac{Y}{Y_n}) > (\frac{24}{116})^3$$

$$f(\frac{Y}{Y_n}) = (\frac{841}{108}) \cdot (\frac{Y}{Y_n}) + (\frac{16}{116}) \qquad \text{if} \quad (\frac{Y}{Y_n}) \leq (\frac{24}{116})^3$$

and

$$u' = \frac{4X}{(X+15Y+3Z)}$$

$$v' = \frac{9Y}{(X+15Y+3Z)}$$

$$u^* = 13 \cdot L^* \cdot (u' - u'_n)$$

$$v^* = 13 \cdot L^* \cdot (v' - v'_n)$$

where u',v',Y describe the colour stimulus considered and Y_n, u'_n, v'_n describe a specified white object colour stimulus.

Info

CIE 015:2018. 8.2.2.1. Basic coordinates. Eq.8.26 to 8.30 (p.30) (CIE, 2018)

Example:

Given a colour sample in CIE XYZ colour space:

```
>>> L, U, V = csc.XYZ_to_LUV(X=36.821, Y=29.514, Z=6.758) # Default D65
>>> print(L, U, V)
61.23260441015891 77.09970653086928 50.24428301895813
```

6.4.1 Reverse transformation

csc.LUV_to_XYZ(L, U, V, Xn=95.04, Yn=100.00, Zn=108.88)

Function to compute the transformation between CIELUV and CIE XYZ colour spaces.

A reference X_n, Y_n, Z_n WhitePoint is required to perform this conversion

Parameters:

L, U, V: `float` *CIELUV coordinates.*

X_n, Y_n, Z_n: `float` *White point, $Y_n = 100$. Default: D65.*

Returns:

X, Y, Z: `float` *CIE XYZ tristimulus values.*

Example:

Given a colour sample in CIELUV colour space:

```
>>> X, Y, Z = csc.LUV_to_XYZ(L=61.232604, U=77.099707, V=50.244283) # Default D65
>>> print(X, Y, Z)
36.820999540426826 29.513999529780317 6.7579997249050034
```

6.5 CIE XYZ to sRGB

csc.XYZ_to_RGB(X, Y, Z, rgb_space="sRGB", scaled = False)

Function to compute the transformation between CIE XYZ and RGB colour spaces (sRGB, AdobeRGB, AppleRGB).

Implemented only for CIE D65 standard illuminant

Parameters:

X, Y, Z: `float` *CIE XYZ tristimulus values.*

rgb_space: `str` *Output RGB colour space. Default: "sRGB".*

scaled: `bool` *If `True`, CIE XYZ in range (0-1). Default: `False`.*

Returns:

R, G, B: `float` *RGB coordinates*

The transformation from CIE XYZ tristimulus values to sRGB colour space is performed following the IEC formulation (IEC, 1999).

Info

International Electrotechnical Commission, 1999. IEC/4WD 61966-2-1: Colour Measurement and Management in Multimedia Systems and Equipment - Part 2-1: Default RGB Colour Space - sRGB.

3.3 Transformation from 1931 CIE XYZ values to RGB values (IEC, 1999).

Example:

Given a colour sample in CIE XYZ colour space:

```
>>> sR, sG, sB = csc.XYZ_to_RGB(X=36.821, Y=29.514, Z=6.758)
>>> print(sR,sG,sB)
0.8574573325855739 0.4841575515644259 0.19554223093601664
```

6.5.1 Reverse transform

csc.RGB_to_XYZ(R, G, B, rgb_space="sRGB", scaled = False)

Function to compute the transformation between RGB colour spaces (sRGB, AdobeRGB, AppleRGB) and CIE XYZ.

Implemented only for CIE D65 standard illuminant

Parameters:

R, G, B: `float` *RGB coordinates.*

rgb_space: `str` *Output RGB colour space. Default: "sRGB".*

scaled: `bool` *If* `True`*, CIE XYZ in range (0-1). Default:* `False`*.*

Returns:

X, Y, Z: `float` *CIE XYZ tristimulus values.*

The transformation from sRGB values to CIE XYZ tristimulus coordinates is performed following the IEC formulation (IEC, 1999).

Info

International Electrotechnical Commission, 1999. IEC/4WD 61966-2-1: Colour Measurement and Management in Multimedia Systems and Equipment - Part 2-1: Default RGB Colour Space - sRGB.

3.2 Transformation from RGB values to 1931 CIE XYZ values(IEC, 1999).

Example:

Given a colour sample in sRGB colour space:

```
>>> X, Y, Z = csc.RGB_to_XYZ(R=0.8574573, G=0.4841576, B=0.1955422)
>>> print(X,Y,Z)
36.821664390790275 29.51556496878112 6.758527341276241
```

6.6 CIE xyY to CIE u'v'Y

csc.xy_to_uv(x, y)

Function to compute the transformation between CIE 1931 x,y chromaticity coordinates and CIE 1976 u',v' UCS.

Parameters:

x, y: `float` *CIE x,y chromaticity coordinates.*

Returns:

u, v: `float` *CIE u',v' chromaticity coordinates.*

The CIE 1976 u',v' UCS chromaticity coordinates can be obtained from CIE 1931 x,y chromaticity coordinates as follows:

$$u' = \frac{4x}{(-2x+12y+3)}$$

$$v' = \frac{9y}{(-2x+12y+3)}$$

Info

CIE015:2018. 8.1. CIE 1976 uniform chromaticity scale (UCS) diagram. Note. 3. Eq. 8.2 (p. 27) (CIE, 2018)

Example:

Given a colour sample in CIE $u'v'$ colour space:

```
>>> u, v = csc.xy_to_uv(x=0.503756, y=0.403787)
>>> print(u,v)
0.29468324633822035 0.5314593651998879
```

6.6.1 Reverse transform

csc.uv_to_xy(u, v)

Function to compute the transformation between the CIE 1976 u',v' and x,y chromaticity coordinates.

Solved using a linear equation system.

Parameters:

u, v: `float` *CIE u',v' chromaticity coordinates.*

Returns:

x, y: `float` *CIE x,y chromaticity coordinates.*

The transformation between the CIE 1976 u',v' chromaticity coordinates and CIE 1931 x,y chromaticity coordinates can be obtained by simply solving the linear system of equations formed with the input u',v' coordinates.

Info

For further details about the process, please refer to the source code.

Example:

Given a colour sample in CIE u'v' colour space:

```
>>> x, y = csc.uv_to_xy(u=0.294683, v=0.531459)
>>> print(x,y)
0.5037551612098116 0.40378638774005415
```

6.7 CIE xyY to CIELUV

csc.xyY_to_LUV(x, y, Y, Xn=95.04, Yn=100.00, Zn=108.88)

Function to compute the transformation between the CIE xyY and CIELUV colour spaces.

A reference X_n, Y_n, Z_n WhitePoint is required to perform this conversion.

Parameters:

x, y, Y: `float` *CIE x,y chromaticity coordinates.*

X_n, Y_n, Z_n: `float` *White point,* $Y_n = 100$*. Default: D65.*

Returns:

L, U, V: `float` *CIELUV coordinates.*

The CIELUV coordinates can be obtained from CIE xyY as follows:

$$L^* = 116 f(\frac{Y}{Y_n}) - 16$$

where

$$f(\frac{Y}{Y_n}) = (\frac{Y}{Y_n})^{\frac{1}{3}} \quad \text{if} \quad (\frac{Y}{Y_n}) > (\frac{24}{116})^3$$

$$f(\frac{Y}{Y_n}) = (\frac{841}{108})(\frac{Y}{Y_n}) + (\frac{16}{116}) \quad \text{if} \quad (\frac{Y}{Y_n}) \leq (\frac{24}{116})^3$$

and

$$u' = \frac{4x}{(-2x+12y+3)}$$

$$v' = \frac{9y}{(-2x+12y+3}$$

$$u^* = 13L^*(u' - u'_n)$$

$$v^* = 13L^*(v' - v'_n)$$

where u',v',Y describe the colour stimulus considered and Y_n, u'_n, v'_n describe a specified white object colour stimulus.

Info

CIE 015:2018. 8.2.2.1. Basic coordinates. Eq.8.26 to 8.30 (p.30). And Note 3. Eq. 8.2 (p.27) (CIE, 2018)

Example:

Given a colour sample in CIE xyY colour space:

```
>>>L, U, V = csc.xyY_to_LUV(x=0.503756, y=0.403787, Y=29.514) # Default D65
>>> print(L, U, V)
61.23260441015891 77.09996125607354 50.24435950485714
```

6.7.1 Reverse transform

csc.LUV_to_xyY(L, U, V, Xn=95.04, Yn=100.00, Zn=108.88)

Function to compute the transformation between the CIELUV and CIE xyY colour spaces.

Parameters:

L, U, V: `float` *CIELUV coordinates.*

X_n, Y_n, Z_n*:* `float` *White point,* $Y_n = 100$*. Default: D65.*

Returns:

x, y, Y: `float` *CIE x,y chromaticity coordinates.*

Example:

Given a colour sample in CIELUV colour space:

```
>>> x, y, Y = csc.LUV_to_xyY(L=61.232604, U=77.099707, V=50.244283) # Default D65
>>> print(x, y, Y)
0.5037554915983159 0.4037869565662692 29.513999529780317
```

6.8 CIELAB to CIE LCHab

csc.LAB_to_LCHab(L, a, b)

Function to compute the transformation between CIELAB and CIE LCHab colour spaces.

Parameters:

L, a, b: `float` *CIELAB coordinates.*

Returns:

L, C, Hab: `float` *CIE LCHab coordinates. Hab in degrees.*

The CIE LCHab lightness (L^*), chroma (C^*_{ab}) and hue angle (h_{ab}) are calculated from CIELAB $L^*a^*b^*$ coordinates as follows:

$$L^* = L^*$$

$$C^*_{ab} = \sqrt{(a^*)^2 + (b^*)^2}$$

$$h_{ab} = arctan(\frac{b^*}{a^*})$$

Info

CIE 015:2018. 8.2.1.2 Correlates of lightness, chroma and hue. Equations 8.12 and 8.13 (p. 29) (CIE, 2018)

Example:

Given a colour sample in CIELAB colour space:

```
>>> L, Cab, Hab = csc.LAB_to_LCHab(L=61.232604, a=31.60268, b=53.969245)
>>> print(L, Cab, Hab)
61.232604 62.54125669550001 59.64811202045994
```

6.8.1 Reverse transformation

csc.LCHab_to_LAB(L, Cab, Hab)

Function to compute the transformation between CIE LCHab and CIELAB colour spaces.

Parameters:

L, Cab, Hab: `float` *CIE LCHab coordinates. Hab in degrees.*

Returns:

L, a, b: `float` *CIELAB coordinates.*

Since the CIE LCHab coordinates represent the polar coordinates of the CIELAB system, it is easy to convert to a Cartesian system using basic trigonometric formulae as follows:

$$L^* = L^*$$

$$a^* = C_{ab} \cdot cos(H_{ab})$$

$$b^* = C_{ab} \cdot sin(H_{ab})$$

Example:

Given a colour sample in CIE LCHab colour space:

```
>>> L, a, b = csc.LCHab_to_LAB(L=61.232604, Cab=62.541257, Hab=59.648112)
>>> print(L, a, b)
61.232604 31.602680173138754 53.969245251479585
```

6.9 CIELUV to CIE LCHuv

csc.LUV_to_LCHuv(L, U, V)

Function to compute the transformation between CIELUV and CIE LCHuv colour spaces.

Parameters:

L, U, V: `float` *CIELUV coordinates*

Returns:

L, Cuv, Huv: `float` *CIE LCHuv coordinates. Huv in degrees*

The CIE LCHuv lightness (L^*), chroma (C^*_{uv}) and hue angle (h_{uv}) are calculated from CIELUV coordinates as follows:

$$L^* = L^*$$

$$C^*_{uv} = \sqrt{(u^*)^2 + (v^*)^2}$$

$$h_{uv} = arctan(\frac{v^*}{u^*})$$

Info

CIE 015:2018. 8.2.2.2 Correlates of lightness, chroma and hue. Equations 8.32 and 8.33 (p. 31) (CIE, 2018)

Example:

Given a colour sample in CIELUV colour space:

```
>>> L, Cuv, Huv = csc.LUV_to_LCHuv(L=61.232604, U=77.099707, V=50.244283)
>>> print(L, Cuv, Huv)
61.232604 92.02637009939019 33.09145484438754
```

6.9.1 Reverse transform

csc.LCHuv_to_LUV(L, Cuv, Huv)

Function to compute the transformation between CIE LCHuv and CIELUV colour spaces.

Parameters:

L, Cuv, Huv: `float` *CIE LCHub coordinates. Huv in degrees.*

Returns:

L, U, V: `float` *CIELUV coordinates.*

Since the CIE LCHuv coordinates represent the polar coordinates of the CIELUV system, it is easy to convert to a Cartesian system using basic trigonometric formulae as follows:

$$L^* = L^*$$

$$u^* = C_{uv} \cdot cos(H_{uv})$$

$$v^* = C_{uv} \cdot sin(H_{uv})$$

Example:

Given a colour sample in CIE LCHuv colour space:

```
>>> L, U, V = csc.LCHuv_to_LUV(L=61.232604, Cuv=92.02637, Huv=33.091455)
>>> print(L, U, V)
61.232604 77.09970678026993 50.24428315513418
```

6.10 Spectral to CIEXYZ

csc.spectral_to_XYZ(reflectance, spd, x_cmf, y_cmf, z_cmf)

Function to compute the CIE XYZ tristimulus values from spectral data.

Parameters:

reflectance: `list` *Spectral data for the colour sample.*

spd: `list` *Spectral data for the illuminant.*

x_cmf, y_cmf, z_cmf: `list` *CIE CMFs.*

Returns:

X, Y, Z: `float` *CIE XYZ tristimulus values.*

Spectral radiant energy is converted into CIE XYZ tristimulus values by integrating the product of the spectral distribution of a sample ($\varphi_\lambda(\lambda)$) with the spectral power distribution of a light source (or illuminant) and with the $\bar{x}(\lambda)$, $\bar{y}(\lambda)$, $\bar{z}(\lambda)$ CMFs using the CIE formulation:

$$X = k \cdot \sum_\lambda \cdot \varphi_\lambda(\lambda) \cdot \bar{x}(\lambda) \cdot \Delta\lambda$$

$$Y = k \cdot \sum_\lambda \cdot \varphi_\lambda(\lambda) \cdot \bar{y}(\lambda) \cdot \Delta\lambda$$

$$Z = k \cdot \sum_\lambda \cdot \varphi_\lambda(\lambda) \cdot \bar{z}(\lambda) \cdot \Delta\lambda$$

where

$$k = \frac{100}{\sum_\lambda \cdot S(\lambda) \cdot \bar{y}(\lambda) \cdot \Delta\lambda}$$

The summatory is computed over the visible spectrum (about 360 to 830 nm or 300 to 700 depending on the device used).

The tristimulus values can be obtained in a similar way for the cone-fundamental-based CMFs. Please refer to the technical documentation published by the CIE for a more detailed discussion on the conversion of spectral data to CIE XYZ (CIE, 2018).

Info

CIE 015:2018. 7.1 Calculation of tristimulus values (pp.21-22) (CIE, 2018).

Example:

Given the reflectance data, the SPD of the illuminant and the CMFs (for a valid CIE observer):

```
>>> reflectance = [6.0475, 5.97, 5.8775, 5.845, 5.8625, 5.9225, 5.99, 6.06, 6.245, 6.49,
    7.175, 9.435, 13.045, 15.6575, 17.1975, 19.415, 24.47, 33.2475, 44.34, 53.75, 58.15,
    58.6625, 58.1725, 57.9075, 57.5325, 57.32, 57.91, 59.6, 61.6025, 63.5, 64.97]
```

To get the SPD of the illuminant (CIE standard D65 for this example):

```
>>> from coolpi.colour.cie_colour_spectral import Illuminant
>>> D65 = Illuminant("D65")
>>> D65.set_into_visible_range_spectrum()
>>>> spd = D65.lambda_values
```

To get the CMFs for the CIE 2º standard observer:

```
>>> from coolpi.colour.cie_colour_spectral import CMF
>>> cmf_2 = CMF(observer=2)
>>> cmf_2.set_cmf_into_visible_range_spectrum()
>>> x_cmf, y_cmf, z_cmf = cmf_2.get_colour_matching_functions()
>>> x_cmf = x_cmf.lambda_values
>>> y_cmf = y_cmf.lambda_values
>>> z_cmf = z_cmf.lambda_values
```

To compute the CIE XYZ trsitimulus values:

```
>>> X, Y, Z = csc.spectral_to_XYZ(reflectance, spd, x_cmf, y_cmf, z_cmf)
>>> print(X, Y, Z)
36.8208393765158 29.51360348192101 6.757552323290141
```

6.11 Additional examples

If the user wishes to perform a conversion between colour spaces without a straightforward transformation, intermediate conversions must be used until the desired colour space is reached.

Example:

Given a sample in CIE LCHab coordinates, to perform the conversion to sRGB:

```
>>> L, a, b = csc.LCHab_to_LAB(L=61.232604, Cab=62.541257, Hab=59.648112)
>>> X, Y, Z = csc.LAB_to_XYZ(L,a,b)
>>> sR, sG, sB = csc.XYZ_to_RGB(X,Y,Z, rgb_space="sRGB")
>>> print(sR, sG, sB)
0.8574549106403581 0.48415796750328627 0.1955735207277029
```

Given a sample in CIE u'v'Y coordinates, to perform the conversion to CIE LCHuv:

```
>>> X, Y, Z = csc.uvY_to_XYZ(u=0.294683, v=0.531459, Y=29.514)
>>> L, U, V = csc.XYZ_to_LUV(X, Y, Z)
>>> L, Cuv, Huv = csc.LUV_to_LCHuv(L, U, V)
>>> print(L, Cuv, Huv)
61.23260441015891 92.02630188035951 33.091323340827486
```

7 Colour-difference

The three-dimensional colour space produced by plotting the CIE XYZ tristimulus values in rectangular coordinates is not visually uniform, in other words, the CIE XYZ colour space does not match well to the human perception of colour differences. Thus, colour differences equally perceived could correspond to very different distances computed in the CIE XYZ colour space. The same problem occurs in the CIE xyY colour space, where the distances between colour samples are also not directly related to their apparent perceptual differences.

In classical colorimetry, colour difference metrics are determined using formulas based on the CIELAB colour space, since it is more perceptually uniform than the CIE XYZ space. This is partly due to the application of non-linear functions in the transformation from CIE XYZ tristimulus values to CIELAB coordinates. Thus, euclidean distances in CIELAB colour space can be used to represent approximately the perceived magnitude of colour differences between colour stimuli of the same size and shape, viewed under the same light conditions (CIE, 2018).

Two colour difference metrics based on CIELAB coordinates have been implemented in coolpi: the CIE76 or ΔE^*_{ab} equation and the improved CIEDE2000 or ΔE^*_{00} formula.

The ΔE^*_{uv} based on CIELUV coordinates has also been implemented although its practical use is less widespread.

Users are highly recommended to use objects for the calculation of colour differences between samples. However, the functions can also be used separately if desired. In this case, we recommend that the coolpi `colour_difference` module is imported as follows:

```
>>> import coolpi.colour.colour_difference as cde
```

An interactive notebook has been prepared to help users become familiar with the methods and functions involved in colour difference calculations. Also, an additional notebook can be consulted with Test data for the CIEDE2000 colour difference.

Practical use of *CSC* functions

Users are encouraged to previously take a look at the Jupyter Notebooks:

04a_Colour-difference.ipynb

04b_CIEDE2000_Test_data_(Sharma_et_al_2005).ipynb

Info

CIE 015:2018. 8.2.1.3 Colour differences (pp.29-30), 8.2.2.3 Colour differences (p.31), 8.3.1. CIEDE2000 colour-difference formula (pp.32-33) (CIE, 2018).

Sharma, G., Wu, W., and Dalal, E. N. 2005. The CIEDE2000 color-difference formula: Implementation notes, supplementary test data, and mathematical observations. Color Research & Application 30(1), 21-30. (Sharma et al., 2005).

7.1 ΔE^*_{ab}

The CIE 1976 colour difference formula (known as CIE76 or simply ΔE^*_{ab}) was the first equation developed by the CIE that allowed obtaining the colour difference between two stimulus based on the computation of the Euclidean distance from its CIELAB coordinates.

Although other improved metrics have been developed in recent years, the ΔE^*_{ab} remains a widely used equation for the calculation of colour differences in a wide range of scientific disciplines.

The CIE76 colour difference can be obtained between two colour stimuli in CIELAB coordinates. Alternatively, the coordinates of the colour stimuli may be referenced to the CIELCHab colour space. It is essential that both samples have been measured under the same illuminant. The CIE recommends the use of the D65 standard illuminant (CIE, 2018).

Practical examples to compute colour difference between samples

Users are encouraged to previously take a look at the following Jupyter Notebook:

`04a_Colour_difference.ipynb`

7.1.1 Samples in CIELAB coordinates

Differences between two colour samples should be obtained as follows:

$\Delta L^* = L^*_2 - L^*_1$ CIELAB lightness difference

$\Delta a^* = a^*_2 - a^*_1$ CIELAB a^* difference

$\Delta b^* = b^*_2 - b^*_1$ CIELAB b^* difference

where L_1^*, a_1^*, b_1^* and L_2^*, a_2^*, b_2^* are the two color stimuli considered in CIELAB coordinates.

Given a pair of colour stimuli defined in CIELAB space, the colour difference between them can be computed using the ΔE_{ab}^* equation (or CIE76) as follows:

$$\Delta E_{ab}^* = \sqrt{(\Delta L^*)^2 + (\Delta a^*)^2 + (\Delta b^*)^2}$$

where ΔL^*, Δa^*, and Δb^* are the differences between the L^*, a^*, and b^* coordinates of the two colour stimuli.

Info

CIE 015:2018. 8.2.1.3 Colour differences. Eq. 8.14 to 8.16 and 8.21 (p.29) (CIE, 2018).

Example:

Import the coolpi `colour_difference` module as follows:

```
>>> import coolpi.colour.colour_difference as cde
```

To set the CIELAB coordinates of the two colour stimuli:

```
>>> L1, a1, b1 = 29.08465, 43.545095, -39.821735
>>> L2, a2, b2 = 35.89365, 15.903706, -29.659472
```

If you would like to obtain the lightness, a^* and b^* differences:

```
>>> AL = L2 - L1
>>> Aa = a2 - a1
>>> Ab = b2 - b1
>>> print("CIELAB lightness difference = ", AL)
CIELAB lightness difference =  6.809000000000001
>>> print("CIELAB a difference         = ", Aa)
CIELAB a difference         =  -27.641389000000004
>>> print("CIELAB b difference         = ", Ab)
CIELAB b difference         =  10.162262999999996
```

cde.delta_E_ab(L1, a1, b1, L2, a2, b2)

Function to compute the CIE76 colour difference between two colour samples in CIELAB units.

Parameters:

L1, a1, b1: `float` *CIELAB coordinates of sample 1.*

L2, a2, b2: `float` *CIELAB coordinates of sample 2.*

Returns:

delta_e_ab: `float` *CIE76 colour difference value in CIELAB units.*

Compute the ΔE^*_{ab} colour difference between two colour samples as follows:

```
>>> AEab = cde.delta_E_ab(L1, a1, b1, L2, a2, b2)
>>> print(AEab)
30.22714766779178
```

Info

For the calculation of colour differences using Colour objects, please refer to the CIELAB class section for a more detailed explanation.

7.1.2 Samples in CIELCHab coordinates

An alternative way to compute the ΔE^*_{ab} colour difference is through coordinates in the CIELCHab colour space.

$\Delta L^* = L^*_2 - L^*_1$ CIELAB lightness difference

$\Delta C^*_{ab} = C^*_{ab2} - C^*_{ab1}$ CIELAB chroma difference

$\Delta h_{ab} = h_{ab2} - h_{ab1}$ CIELAB hue-angle difference

The CIELAB hue difference is calculated as follows:

$\Delta H^*_{ab} = 2 \cdot \sqrt{(C^*_{ab2} \cdot C^*_{ab1})} \cdot sin(\frac{\Delta h_{ab}}{2})$ CIELAB hue difference

where subscripts 2 and 1 refer to the two colour samples between which the colour difference is to be calulated.

The colour difference between the two stimulus can be obtained using the following equation:

$$\Delta E^*_{ab} = \sqrt{(\Delta L^*)^2 + (\Delta C^*_{ab})^2 + (\Delta H^*_{ab})^2}$$

However, both computations for the ΔE^*_{ab} are equivalent.

Info

CIE 015:2018. 8.2.1.3 Colour differences. Eq. 8.17 to 8.19, and Eq. 8.22 (p.30) (CIE, 2018)

Example:

Import the coolpi `colour_difference` module as follows:

```
>>> import coolpi.colour.colour_difference as cde
```

To set the CIELCHab coordinates of the two colour stimuli:

```
>>> L1, Cab1, Hab1 = 29.08465, 59.008015, 317.557266
>>> L2, Cab2, Hab2 = 35.89365, 33.654303, 298.200655
```

cde.compute_hue_angle_difference(C1, H1, C2, H2)

Function to compute the Hue difference for colour samples in CIELAB or CIELUV coordinates.

Parameters:

C1, H1: `float` *Chroma and Hue (degree) sample 1.*

C2, H2: `float` *Chroma and Hue (degree) sample 1.*

Returns:

Ah: `float` *Ahab for CIELAB or Ahuv for CIELUV.*

If you would like to obtain the lightness, chroma and hue differences:

```
>>> AL = L2 - L1
>>> ACab = Cab2 - Cab1
>>> Ahab = cde.compute_hue_angle_difference(Cab1, Hab1, Cab2, Hab2)
>>> print("CIELAB lightness difference = ", AL)
CIELAB lightness difference =  6.809000000000001
>>> print("CIELAB chroma difference    = ", ACab)
CIELAB chroma difference    =  -25.353712
>>> print("CIELAB hue-angle difference = ", Ahab)
CIELAB hue-angle difference =  -0.3378365939777529
```

cde.compute_Delta_H_difference(C1, C2, Ah)

Function to compute the Delta_Hue difference for colour samples in CIELAB or

CIELUV coordinates.

Parameters:

C1: `float` *Chroma sample 1.*

C2: `float` *Chroma sample 2.*

h: `float` *Hue-angle difference in degrees.*

Returns:

Delta_H: `float` *Delta_Hab for CIELAB or Delta_Huv for CIELUV.*

The ΔH^*_{ab} can be computed as follows:

```
>>> AHab = cde.compute_Delta_hue_angle(Cab1, Cab2, Ahab)
>>> print("CIELAB hue difference    = ", AHab)
CIELAB hue difference    =  -14.98356683141618
```

cde.delta_E_ab_cielchab(L1, Cab1, Hab1, L2, Cab2, Hab2)

Function to compute the CIE76 colour difference between two colour samples in CIELHab coordinates.

Parameters:

L1, Cab1, Hab1: `float` *CIELCHab coordinates of sample 1.*

L2, Cab2, Hab2: `float` *CIELCHab coordinates of sample 2.*

Returns:

delta_e_ab: `float` *CIE76 colour difference value in CIELAB units.*

Compute the ΔE^*_{ab} colour difference between two colour samples as follows:

```
>>> AEab = cde.delta_E_ab_cielchab(L1, Cab1, Hab1, L2, Cab2, Hab2)
>>> print(AEab)
30.227147866949988
```

Info

For the calculation of colour differences using Colour objects, please refer to the CIELCHab class section for a more detailed explanation.

7.2 CIEDE2000

CIEDE2000 (or ΔE^*_{00}) is a colour-difference formula recommended by the CIE in 2001 since it provides an improved procedure for the computation of color differences for industrial applications (Sharma et.al, 2005). The CIEDE2000 formula corrects the non-uniformity of the CIELAB colour space adding lightness, chroma, and hue weighting functions and a scaling factor for CIELAB a* axis (Luo et al., 2001).

The ΔE^*_{00} colour difference equation must be applied as follows:

$$\Delta E_{00} = \sqrt{(\frac{\Delta L'}{k_L \cdot S_L})^2 + (\frac{\Delta C'}{k_C \cdot S_C})^2 + (\frac{\Delta H'}{k_H \cdot S_H})^2 + R_T \cdot (\frac{\Delta C'}{k_C \cdot S_C}) \cdot (\frac{\Delta H'}{k_H \cdot S_H})^2}$$

where the S_L, S_C, S_H and R_T are the weighting functions, and k_L, k_C and k_H are the parametric factors or correction terms for variations in experimental conditions (under reference conditions they are set to $k_L = k_C = k_H = 1$).

For the implementation of the CIEDE2000 formula into the coolpi package, the practical recommendations provided by Sharma et al. have been considered.

> **Info**
>
> *CIE 015:2018. 8.3.1. CIEDE2000 colour-difference formula Eq. 8.36 to 8.45 (pp.33-34) (CIE, 2018).*
>
> *Sharma, G., Wu, W., and Dalal, E. N. 2005. The CIEDE2000 color-difference formula: Implementation notes, supplementary test data, and mathematical observations. Color Research & Application 30(1), 21-30 (Sharma et.al, 2005).*

> **Practical examples to compute colour difference between samples**
>
> *Users are encouraged to previously take a look at the following Jupyter Notebook:*
>
> *04a_Colour_difference.ipynb*
>
> *04b_CIEDE2000_Test_data(Sahrma_et_al_2005).ipynb*

Example:

Import the coolpi `colour_difference` module as follows:

```
>>> import coolpi.colour.colour_difference as cde
```

To set the CIELAB coordinates of the two colour stimuli:

```
>>> L1, a1, b1 = 29.08465, 43.545095, -39.821735
>>> L2, a2, b2 = 35.89365, 15.903706, -29.659472
```

cde.CIEDE2000(L1, a1, b1, L2, a2, b2, kl=1, kc=1, kh=1)

Function to compute the CIEDE2000 colour difference between two colour samples in CIELAB coordinates.

Parameters:

L1, a1, b1: `float` *CIELAB coordinates of sample 1.*

L2, a2, b2: `float` *CIELAB coordinates of sample 2.*

kl, kc, kh: `float` *kl, kc, kh parametric factors. Default: values set to 1.*

Returns:

AE00: `float` *CIEDE2000 colour difference value in CIELAB units.*

Compute the ΔE^*_{00} colour difference between the two colour samples as follows:

```
>>> AE00 = cde.CIEDE2000(L1, a1, b1, L2, a2, b2, kl=1, kc=1, kh=1)
>>> print(AE00)
13.205812360718502
```

Info

For the calculation of colour differences using Colour objects, please refer to the CIELAB class section for a more detailed explanation.

7.3 ΔE^*_{uv}

7.3.1 Samples in CIELUV coordinates

The CIELUV colour difference between two colour stimuli is obtained as the Euclidean distance between the points representing them in the CIELUV colour space, and it is designed as ΔE^*_{uv}:

$$\Delta E^*_{uv} = \sqrt{(\Delta L^*)^2 + (\Delta u^*)^2 + (\Delta v^*)^2}$$

where ΔL^*, Δu^*, and Δv^* are the differences between the L^*, u^*, and v^* coordinates of the two colour samples.

Info

CIE 015:2018. 8.2.2.3 Colour differences. 8.35 (p.31) (CIE, 2018)

Practical examples to compute colour difference between samples

Users are encouraged to previously take a look at the following Jupyter Notebook:

04a_Colour_difference.ipynb

Example:

Import the coolpi `colour_difference` module as follows:

```
>>> import coolpi.colour.colour_difference as cde
```

To set the CIELUV coordinates of the two colour stimuli:

```
>>> L1, U1, V1 = 29.08465, 19.220243, -55.723049
>>> L2, U2, V2 = 35.89365, -1.021978, -42.663638
```

The lightness, u^* and v^* differences can be computed as follows:

```
>>> AL = L2 - L1
>>> AU = U2 - U1
>>> AV = V2 - V1
>>> print("CIELUV lightness difference = ", AL)
CIELUV lightness difference =  6.809000000000001
>>> print("CIELUV u difference         = ", AU)
CIELUV u difference         =  -20.242221
>>> print("CIELUV v difference         = ", AV)
CIELUV v difference         =  13.059411000000004
```

cde.delta_E_uv(L1, U1, V1, L2, U2, V2)

Function to compute the CIE76 colour difference between two colour samples in CIELUV units.

Parameters:

L1, U1, V1: `float` *CIELUV coordinates of sample 1.*

L2, U2, V2: `float` *CIELUV coordinates of sample 2.*

Returns:

delta_e_uv: `float` *CIE76 colour difference value in CIELUV units.*

Compute the ΔE^*_{uv} colour difference between two colour samples as follows:

```
>>> AEuv = cde.delta_E_uv(L1, U1, V1, L2, U2, V2)
>>> print(AEuv)
25.03314218550604
```

Info

For the calculation of colour differences using Colour objects, please refer to the CIELUV class section for a more detailed explanation.

7.3.2 Samples in CIELCHuv coordinates

Approximate correlates of saturation, chroma and hue can be obtained from CIELUV coordinates as follows:

$s_{uv} = 13 \cdot \sqrt{(u' - v'_n)^2 + (v' - v'_n)^2}$ CIE1976 CIELUV saturation

$C^*_{uv} = \sqrt{u^{*2} + v^{*2}}$ CIE1976 CIELUV chroma

$h^*_{uv} = arctg(\frac{v^*}{u^*})$ CIE1976 CIELUV hue-angle

where the h^*_{uv} has to be calculated taking into account its quadrant.

The CIELUV hue difference is calculated as follows:

$$\Delta h_{uv} = h_{uv,2} - h_{uv,1}\,.$$

$$\Delta H^*_{uv} = 2 \cdot \sqrt{(C^*_{uv,2} \cdot C^*_{uv,1})} \cdot sin(\frac{\Delta h_{uv}}{2})$$

where subscripts 2 and 1 refer to the two colour samples between which the colour difference is to be calulated.

Differences between two colour samples should be obtained as follows:

$$\Delta E^*_{uv} = \sqrt{(\Delta L^*)^2 + (\Delta C^*_{uv})^2 + (\Delta H^*_{uv})^2}$$

Info

CIE 015:2018. 8.2.2.2 Correlates of lightness, saturation, chroma and hue. Eq. 8.31 to 8.34 (p.31) (CIE, 2018)

Example:

Import the coolpi `colour_difference` module as follows:

```
>>> import coolpi.colour.colour_difference as cde
```

To set the CIELCHuv coordinates of the two colour stimuli:

```
>>> L1, Cuv1, Huv1 = 29.08465, 58.94469, 289.030567
>>> L2, Cuv2, Huv2 = 35.89365, 42.67588, 268.627782
```

The lightness, chroma and hue differences can be obtained as follows:

```
>>> AL = L2 - L1
>>> ACuv = Cuv2 - Cuv1
>>> Ahuv = cde.compute_hue_angle_difference(Cuv1, Huv1, Cuv2, Huv2)
>>> print("CIELUV lightness difference = ", AL)
CIELUV lightness difference =  6.809000000000001
>>> print("CIELUV chroma difference    = ", ACuv)
CIELUV chroma difference    =  -16.268810000000002
>>> print("CIELUV hue-angle difference = ", Ahuv)
CIELUV hue-angle difference =  -0.35609577482651117
```

The ΔH^*_{uv} can be computed as follows:

```
>>> AHuv = cde.compute_Delta_H_difference(Cuv1, Cuv2, Ahuv)
>>> print("CIELUV hue difference    = ", AHuv)
CIELUV hue difference    =  -17.76574315954105
```

cde.delta_E_uv_cielchuv(L1, Cuv1, Huv1 , L2, Cuv2, Huv2)

Function to compute the CIE76 colour difference between two colour samples in CIELHuv coordinates.

Parameters:

L1, Cuv1, Huv1: `float` *CIELCHuv coordinates of sample 1.*

L2, Cuv2, Huv2: `float` *CIELCHuv coordinates of sample 2.*

Returns:

delta_e_uv: `float` *CIE76 colour difference value in CIELUV units.*

Compute the ΔE^*_{uv} colour difference between two colour samples as follows:

```
>>> AEuv = cde.delta_E_uv_cielchuv(L1, Cuv1, Huv1 , L2, Cuv2, Huv2)
>>> AEuv
25.03314382627319
```

Info

For the calculation of colour differences using Colour objects, please refer to the CIELCHuv class section for a more detailed explanation.

8 Image

The colour data registered by consumer digital cameras (usually in the well-known RGB colour model) are not strictly colorimetric. The built-in imaging sensor of the camera does not satisfy the Luther-Ives condition, which means that its three spectral sensitivity curves (one for Red, one for Green and one for Blue) do not entirely mimic those of the human eye. Moreover, these spectral curves are device-dependent; in other words, they differ from camera to camera. Finally, every camera brand processes the images in their proprietary way to yield a pleasing photograph, not a colour-accurate one.

A digital camera is thus not suitable for rigorous colour determination without any colour correction procedure. The coolpi image module includes the classes, methods and functions required for processing and obtaining colour-accurate data from digital images, especially in RAW format.

The main classes implemented in the image module can be divided into *ColourChecker* and *Image* classes.

Colour checkers are a widely used tool in digital photography, especially for white balance adjustment and colour calibration.

Depending on the type of data stored in the *ColourChecker* class, these can be:

- ColourCheckerSpectral: Measured spectral data from a given colour checker.
- ColourCheckerXYZ: Measured XYZ data from a given colour checker, or computed from a *ColourCheckerSpectral* or *ColourcheckerLAB* instance.
- ColourcheckerLAB: Measured LAB data from a given colour checker, or computed from a *ColourCheckerSpectral* or *ColourcheckerXYZ* instance.
- ColourCheckerRGB: RGB data from a given colour checker extracted from a digital image.

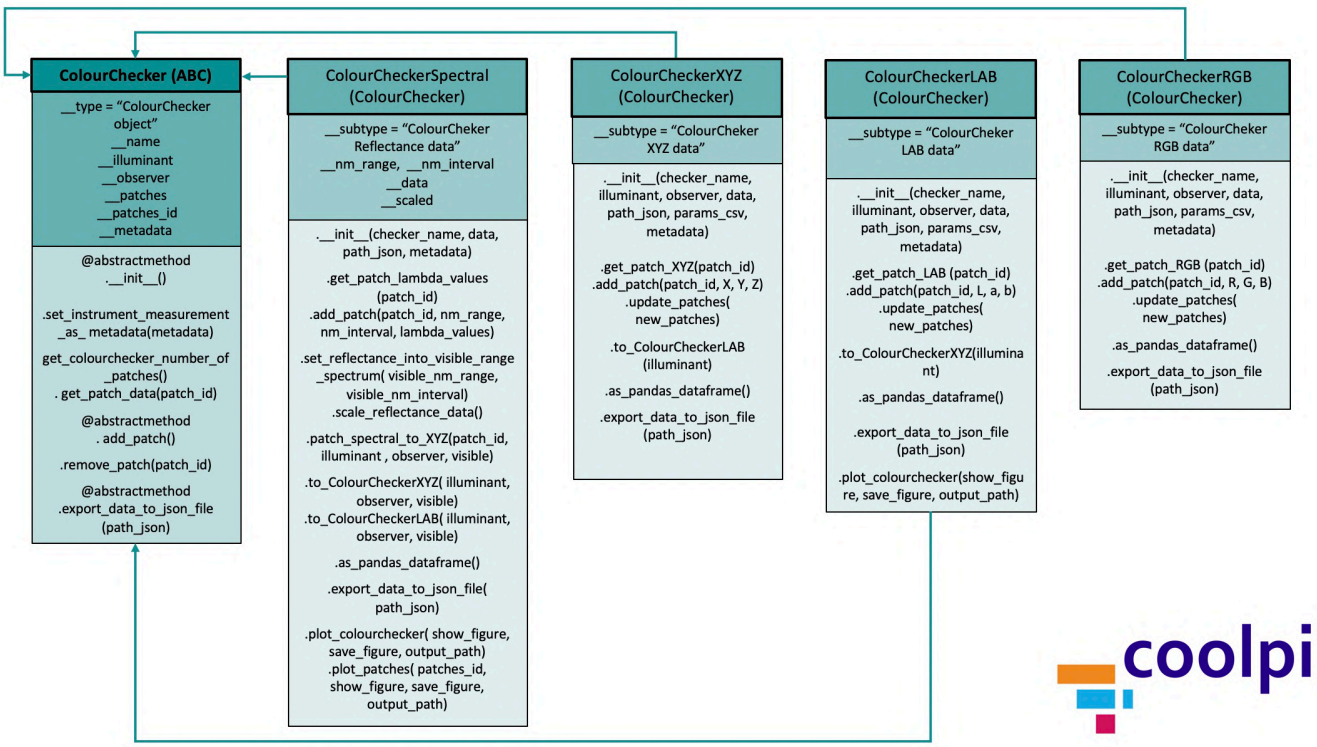

Figure 54: *UML Diagram for the ColourChecker classes*

ColourChecker classes can be imported separately, according to the needs of users as follows:

```
>>> from coolpi.image.image_objects import ColourCheckerSpectral
```

Practical use of *ColourChecker* classes

Users are encouraged to previously take a look at the Jupyter Notebook:

05_ColourChecker_objects.ipynb

For working with digital images, coolpi includes the following classes:

- Image classes: RawImage and ProcessedImage classes.

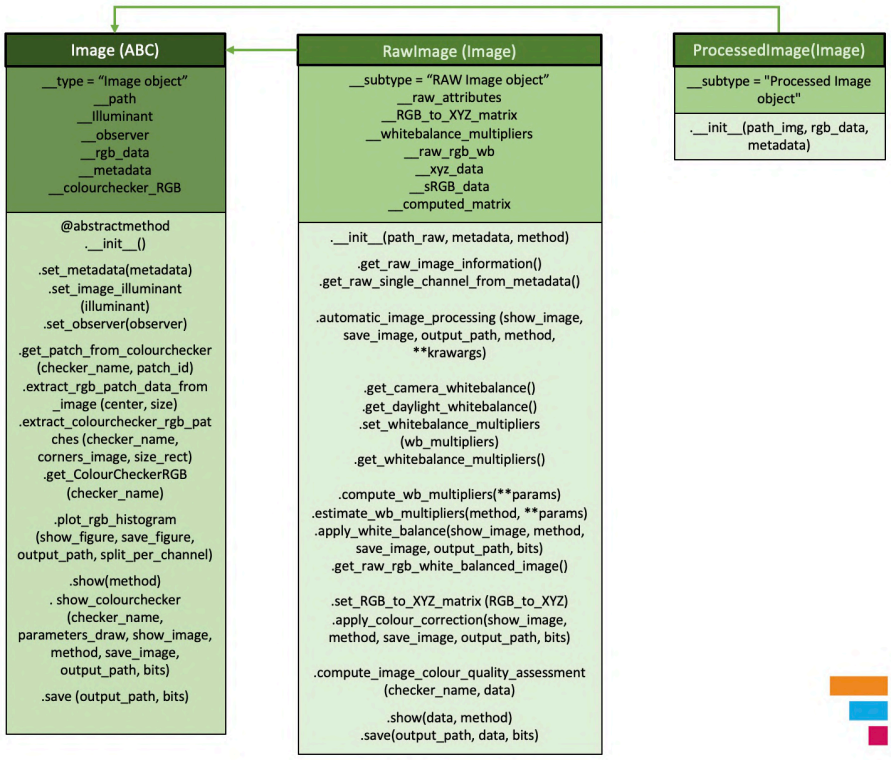

Figure 55: *UML Diagram for the Image classes*

Image classes can be imported separately, according to the needs of users as follows:

```
>>> from coolpi.image.image_objects import RawImage
```

Practical use of *ColourChecker* classes

Users are encouraged to previously take a look at the Jupyter Notebook:

`06_Image_objects.ipynb`

8.1 ColourCheckerSpectral

class ColourCheckerSpectral

ColourCheckerSpectral(checker_name, data=None, path_json=None, metadata={})

.set_instrument_measurement_as_metadata(metadata)

.get_colourchecker_number_of_patches()

.get_patch_data(patch_id)

.get_patch_lambda_values(patch_id)

```
.add_patch(patch_id, nm_range, nm_interval, lambda_values)

.remove_patch(patch_id)

.set_reflectance_into_visible_range_spectrum(visible_nm_range,
    visible_nm_interval)

.scale_reflectance_data()

.patch_spectral_to_XYZ(patch_id, new_illuminant, new_observer)

.to_ColourCheckerXYZ(illuminant, observer)

.to_ColourCheckerLAB(illuminant)

.as_pandas_dataframe()

.export_data_to_json_file(path_json)

.plot_colourchecker(show_figure, save_figure, output_path)

.plot_patches(patches_id, show_figure, save_figure, output_path)
```

The *ColourCheckerSpectral* class represents the spectral data of the colour patches of a given colour checker measured using a colorimetric instrument.

Spectral data for a variety of colour charts are available in the coolpi repository. Attempts have been made to implement the most widespread colour charts provided, not only for digital imaging related applications, but also for other applications (such as soil science, like the Munsell Soil Color Book).

The colour charts implemented in the coolpi package, provided by different manufacturers, are as follows:

- Calibrite CLASSIC (CCC).
- Calibrite PASSPORT PHOTO 2 (CCPP2).
- Calibrite PASSPORT VIDEO (CCPPV).
- Calibrite DIGITAL SG (CCDSG).
- GretagMacbeth (GM).
- Munsell Soil Color Book Ed. 1994 (MSCB_1994).
- Munsell Soil Color Book Ed. 2009/2022 (MSCB_2009).
- Pantone Metallics (PM).
- QPCARD 202 (QPGQP202).

- RAL K5 CLASSIC (RALK5).
- RAL K7 CLASSIC (RALK7).
- Spyder Checker (SCK100).
- X-Rite PASSPORT PHOTO (XRCCPP).
- X-Rite Digital SG (XRCCSG).

> ⚠ **Alert**
>
> *We recommend spectral measurements of the colour charts on a regular basis. Over time, patches may deteriorate or become dirty, and their spectral values may not necessarily match those obtained in previous measurements.*

8.1.1 Create an instance

> ***ColourCheckerSpectral(checker_name, data=None, path_json=None,metadata={})***
>
> *Parameters:*
>
> *checker_name:* `str` *Colour checker name or description.*
>
> *data:* `dict` *Spectral data. Default:* `None`.
>
> *path_json:* `os` *JSON file. Default:* `None`.
>
> *metadata:* `dict` *Instrument measurement information. Default: {}.*

To *create an instance* of the *ColourCheckerSpectral* class, simply enter the required parameters as follows:

The *ColourCheckerSpectral* class can be instantiated from: `checker_name` (automatic loading data from coolpi resources if the colour checker is implemented), data (measured spectral data as `dict`) or from a `path_file` (valid `os` path: JSON file with the spectral data).

Automatic loading of spectral data from coolpi resources:

```
>>> from coolpi.image.colourchecker import ColourCheckerSpectral
>>> XRCCPP = ColourCheckerSpectral("XRCCPP")
>>> type(XRCCPP)
coolpi.image.colourchecker.ColourCheckerSpectral
```

The information on the measurement conditions is automatically included as `metadata`

attribute as `dict` type.

In case of specifying a *ColourChecker* name not included in the coolpi resources:

```
>>> CMP = ColourCheckerSpectral("CMP Refcard Color Master2")
ClassInstantiateError: ColourCheckerSpectral class could not be instantiated.
```

From spectral data as `dict` (only one patch as sample):

```
>>> from coolpi.image.colourchecker import ColourCheckerSpectral
>>> data_as_dict = {"Illuminant": "D65", "Observer": 2, "nm_range":[360, 740],
    "nm_interval":10, "patches": {"B5":
[6.7000,6.7000,6.7000,6.7000,6.7000,6.8300,6.9000,7.0100,

7.2275,7.6625,8.4500,9.6950,12.3225,16.6675,24.8275,35.6100,45.2950,50.2225,51.3925,50.5600


46.0500,42.8025,38.7050,34.5425,31.5350,29.9850,29.2650,28.7025,28.3525,28.7850,30.2350,31.

    34.4100,34.4100,34.4100,34.4100,34.4100]}} # only one patch as a sample
>>> XRCCPP = ColourCheckerSpectral(checker_name="XRCCPP", data=data_as_dict)
>>> type(XRCCPP)
coolpi.image.colourchecker.ColourCheckerSpectral
```

The data `dict` should contain at least the following keys: “Illuminant”, “Observer”, “nm_range”, “nm_interval” and “patches”. Otherwise, a `DictLabelError` is raised.

```
>>> data_as_dict_incomplete = {"Illuminant": "D65", "nm_range":[360, 740],
"nm_interval":10, "patches":
    "B5":
[6.7000,6.7000,6.7000,6.7000,6.7000,6.8300,6.9000,7.0100,7.2275,7.6625,8.4500,9.6950,12.322

>>> XRCCPP = ColourCheckerSpectral(checker_name="XRCCPP", data=data_as_dict_incomplete)
DictLabelError: Error in the dict or file with measurement data: Incomplete data or wrong
labels.
```

From a generic JSON file with the measured data taken with any instrument:

```
>>> from coolpi.image.colourchecker import ColourCheckerSpectral
>>> file_rfc = ["res", "json", "XRCCPP_reflectance.json"]
>>> path_rfc = os.path.join(*file_rfc)
>>> colourchecker_metadata = {"NameColorChart": "Calibrite colorchecker CLASSIC",
    "Manufacturer":["Calibrite", "Made in USA", 850028833087, 2021], "Measurement Date":
    [[2022, 3, 11], [10, 13, 40]], "Instrument": "Konica Minolta CM-600d", "Illuminant":
    "D65", "Observer": 2, "Geometry": "di:8, de:8", "Specular Component":"SCI",
    "Measurement Area": "MAV(8mm)"}
>>> XRCCPP = ColourCheckerSpectral("XRCCPP", path_json = path_rfc,
    metadata=colourchecker_metadata)
>>> type(XRCCPP)
coolpi.image.colourchecker.ColourCheckerSpectral
```

The JSON file should contain at least the following keys: “Illuminant”, “Observer”, “nm_range”, “nm_interval” and “patches” (with the spectral data as `dict`, i.e. “patches”=

{patch_id (as `str`): reflectance (as `list`)}).

Info

The ColourCheckerSpectral metadata can be set separately with the method:

`.set_instrument_measurement_as_metadata(metadata)`

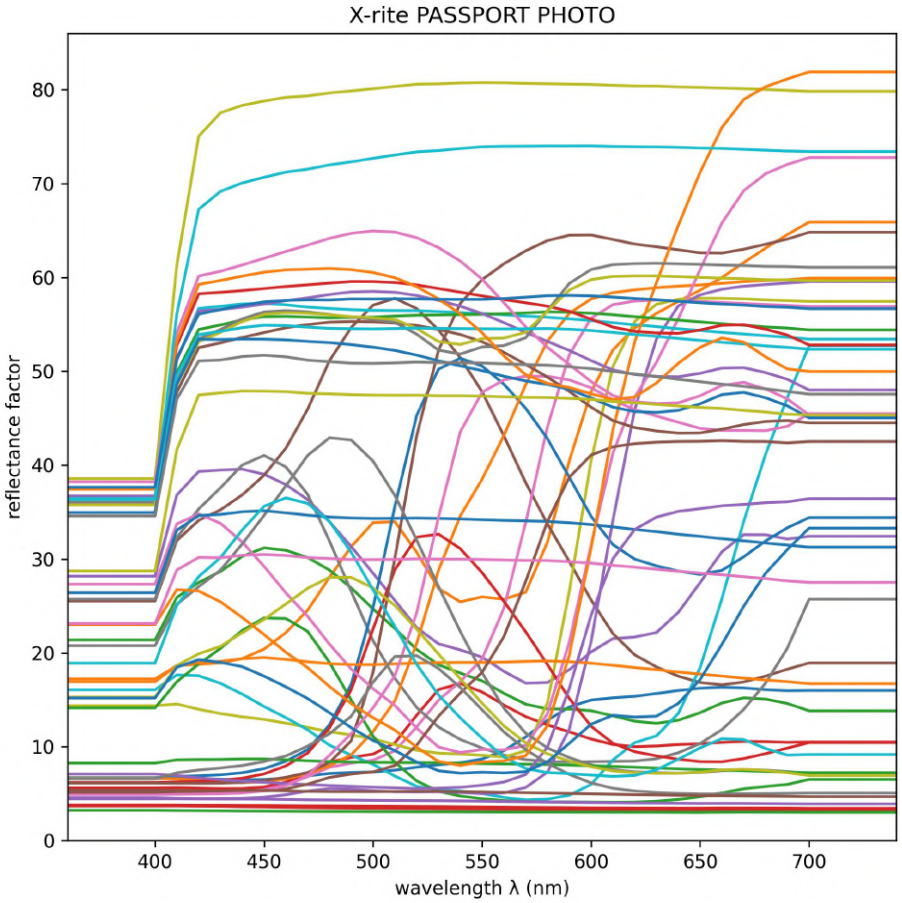

Figure 56: *ColourCheckerSpectral*

Alert

We recommend spectral measurements of the colour charts on a regular basis. Over time, patches may deteriorate or become dirty, and their spectral values may not necessarily match those obtained in previous measurements.

8.1.2 Attributes

The class *attributes* are: *type* (returns a `str` with the description of the main class), *subtype*

(returns a `str` with the description of the object), *name* (returns a `str` with the *ColourChecker* name or description), *illuminant* (returns the *Illuminant*, *IlluminantFromCCT* or *MeasuredIlluminant* instance), *observer* (returns the *Observer* instance), *nm_range* (returns a `list` with the min and max range value in *nm*), *nm_interval* (returns an `int` with the lambda interval in *nm*), *patches* (returns a `dict` with the spectral data), *patches_id* (returns a `list` with the patches id of the *ColourChecker*), *data* (returns a `dict` with the full measured data), *scaled* (returns a `bool`) and *metadata* (returns a `dict` with the measurement details).

```
>>>print(XRCCPP.type)
ColourChecker object
>>> print(XRCCPP.subtype)
ColourCheker Reflectance data
>>> print(XRCCPP.name)
X-rite PASSPORT PHOTO (24+26+1 patches)
>>> print(XRCCPP.illuminant)
Illuminant object: CIE D65 standard illuminant
>>> print(XRCCPP.observer)
2º standard observer (CIE 1931)
>>> print(XRCCPP.nm_range)
[360, 740]
>>> print(XRCCPP.nm_interval)
10
>>> #print(XRCCPP.patches) # reflectance data as dict
>>> print(XRCCPP.patches_id)
dict_keys(['A1', 'A2', 'A3', 'A4', 'A5', 'A6', 'B1', 'B2', 'B3', 'B4', 'B5', 'B6',
'C1', 'C2', 'C3', 'C4', 'C5', 'C6', 'D1', 'D2', 'D3', 'D4', 'D5', 'D6', 'E1', 'E2',
'E3', 'E4', 'E5', 'E6', 'E7', 'E8', 'F1', 'F2', 'F3', 'F4', 'F5', 'G1', 'G2', 'G3',
'G4', 'G5', 'H1', 'H2', 'H3', 'H4', 'H5', 'H6', 'H7', 'H8', 'WR'])
>>> print(XRCCPP.data.keys()) # Full data as dict
dict_keys(['Metadata', 'lambda_nm_interval', 'lambda_nm_range', 'patches'])
>>> print(XRCCPP.scaled)
False
>>> print(XRCCPP.metadata.keys()) # Measurement information as dict
dict_keys(['NameColorChart', 'Manufacturer', 'Measurement Date', 'Instrument',
'Illuminant',
'Observer', 'Geometry', 'Specular Component', 'Measurement Area'])
```

8.1.3 Methods

Patch management:

ColourCheckerSpectral.get_colourchecker_number_of_patches()

Method to get the total number of colour patches of the colour checker.

Returns:

num_patches: `int` *Total number of patches.*

To get the total number of patches:

```
>>> num_patches = XRCCPP.get_colourchecker_number_of_patches()
>>> print(num_patches)
51
```

ColourCheckerSpectral.get_patch_lambda_values(patch_id)

Method to get the spectral data of a given patch as `list`.

Parameter:

patch_id: `str` Patch ID.

Returns:

lambda_values: `list` Spectral data.

To get the lambda values of a patch:

```
>>> lambda_values = XRCCPP.get_patch_lambda_values("A1")
>>> print(lambda_values)
[6.5975, 6.5975, 6.5975, 6.5975, 6.5975, 6.6825, 6.615, 6.535, 6.48, 6.5125, 6.6425, 6.9,
7.15, 7.25, 7.3125, 7.5, 7.8175, 8.08, 8.2925, 8.69, 9.57, 11.04, 12.815, 14.2425, 14.97,
15.21, 15.3675, 15.6675, 15.9925, 16.2325, 16.3475, 16.2625, 16.085, 15.97, 15.9975,
15.9975,
15.9975, 15.9975, 15.9975]
```

The input `patch_id` should be a valid patch. Otherwise, a `PatchError` is raised.

```
>>> lambda_values = XRCCPP.get_patch_lambda_values("P1")
PatchError: Patch id not present in the current ColourChecker
```

ColourCheckerSpectral.get_patch_data(patch_id)

Method to get the spectral data of a given patch as a Reflectance object.

Parameter:

patch_id: `str` Patch ID.

Returns:

reflectance: `Reflectance` Spectral data.

To get a patch as a *Reflectance* object:

```
>>> rfc = XRCCPP.get_patch_data("A1")
>>> type(rfc)
coolpi.colour.cie_colour_spectral.Reflectance
```

The input `patch_id` should be a valid patch. Otherwise, a `PatchError` is raised.

```
>>> rfc = XRCCPP.get_patch_data("P1")
PatchError: Patch id not present in the current ColourChecker
```

ColourCheckerSpectral.add_patch(patch_id, nm_range, nm_interval, lambda_values)

Method to add a patch into the colour checker object.

Parameter:

patch_id: `str` *Patch ID.*

nm_range: `list` *spectral nm range [min, max].*

nm_interval: `int` *spectral interval in nm.*

lambda_values: `list` *spectral data.*

To add a patch:

```
>>> rfc_lambda = [100 for i in range(0,39)]
>>>XRCCPP.add_patch(patch_id="P1", nm_range=[360, 740], nm_interval=10,
lambda_values=rfc_lambda)
>>> print(XRCCPP.get_colourchecker_number_of_patches())
52
```

The new patch should be in the same `nm_range` and `nm_interval` as the *ColourCheckerSpectral*. Otherwise, a `PatchError` is raised.

```
>>> rfc_lambda = [100 for i in range(0,30)]
>>> XRCCPP.add_patch(patch_id="P2", nm_range=[360, 740], nm_interval=10,
lambda_values=rfc_lambda)
PatchError: Wrong reflectance lambda_values.
```

ColourCheckerSpectral.remove_patch(patch_id)

Method to remove a patch from the colour checker.

Parameter:

patch_id: `str` *Patch ID.*

To remove a patch from the colour checker:

```
>>> XRCCPP.remove_patch("P1")
>>>print(XRCCPP.get_colourchecker_number_of_patches())
51
```

To set the reflectance data of the colour patches into the visible spectrum:

ColourCheckerSpectral.set_reflectance_into_visible_range_spectrum(visible_nm_range=[400,700], visible_nm_interval=10)

Method to set the reflectance data of the colour patches into the visible spectrum.

Parameters:

visible_nm_range: `list` *[min, max] range in nm. Default: [400,700].*

visible_nm_interval: `int` *Interval in nm. Default: 10.*

To set the reflectance data of the colour patches into the visible range spectrum:

```
>>> XRCCPP.set_reflectance_into_visible_range_spectrum(visible_nm_range = [400,700],
visible_nm_interval = 10)
```

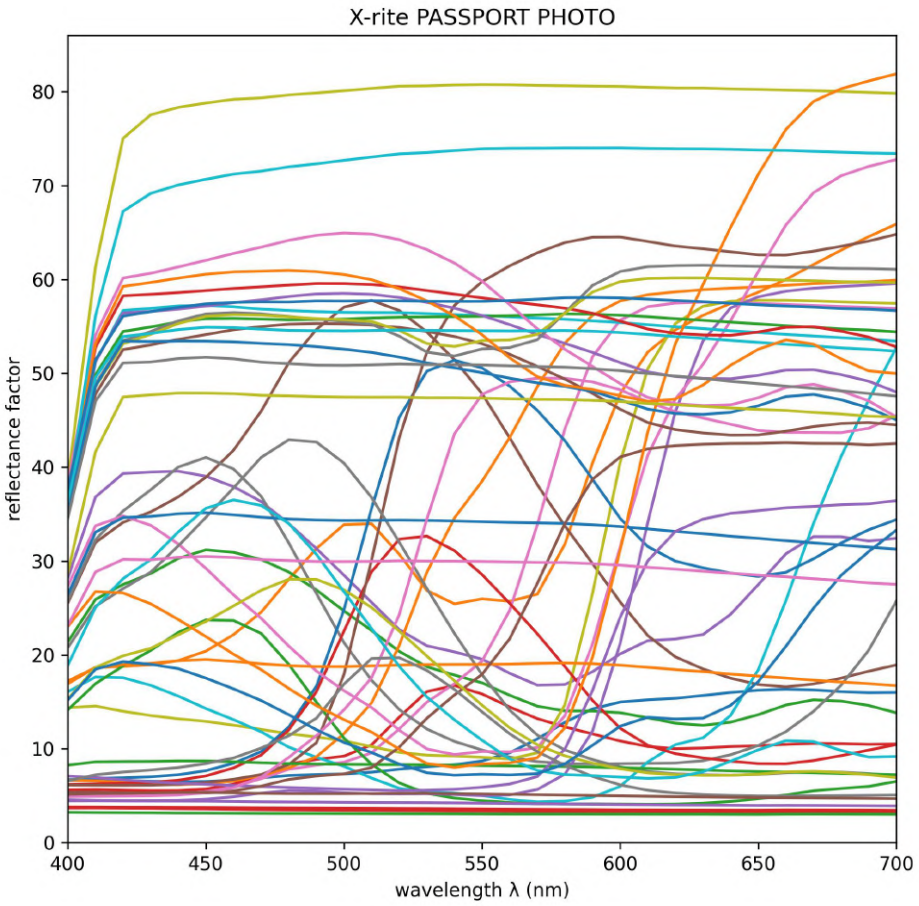

Figure 57: *ColourCheckerSpectral (visible spectrum)*

To scale the reflectance data of the colour patches into the range (0,1):

> ***ColourCheckerSpectral.scale_reflectance_data()()***
>
> *Method to scale the reflectance data of the colour patches into the range (0,1).*

To scale the reflectance data into the range (0,1):

```
>>> XRCCPP.scale_reflectance_data()
>>> print(XRCCPP.scaled)
True
```

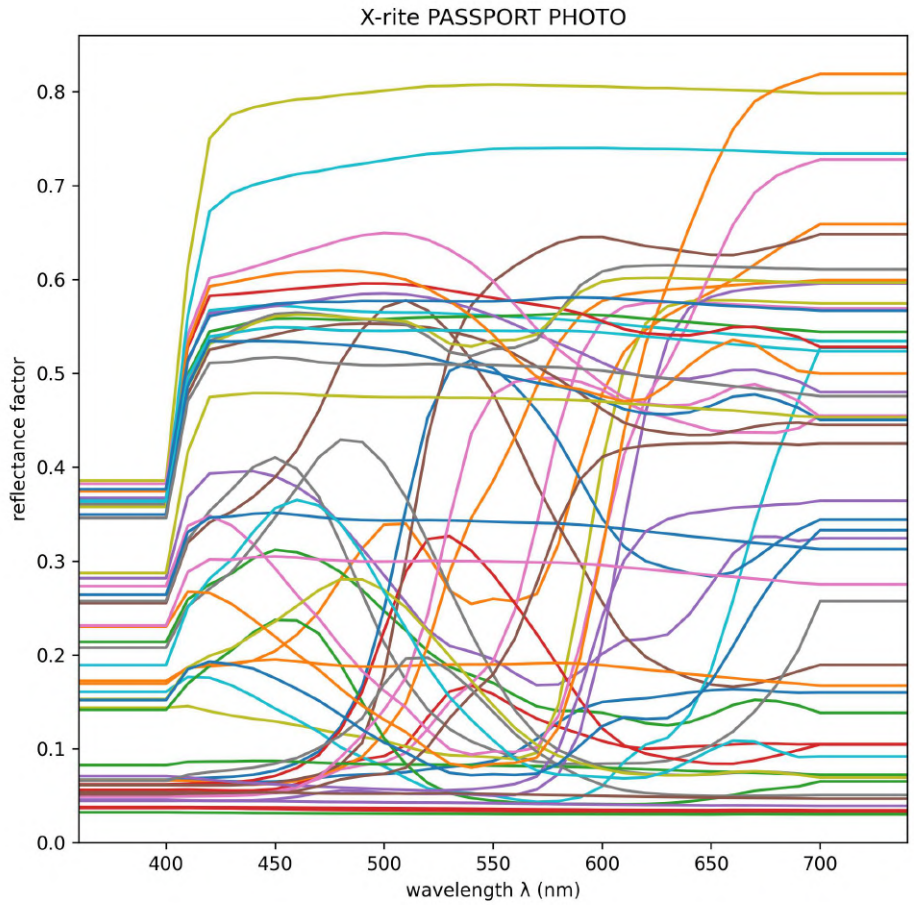

Figure 58: *ColourCheckerSpectral (scaled)*

Colour Space Conversion (CSC):

ColourCheckerSpectral.patch_spectral_to_XYZ(patch_id, illuminant="D65", observer=2, visible=False)

Method to obtain the XYZ tristimulus values of a given colour patch from the reflectance data.

Parameter:

patch_id: `str` *Patch ID.*

illuminant: `str`, `Illuminant`, `IlluminantFromCCT`, `MeasuredIlluminant` *Illuminant. Default: "D65".*

observer: `str`, `int`, `Observer` *CIE Observer. Default: 2.*

visible: `bool` *If True, use only the visible spectrum data. Default:* `False`.

Returns:

X,Y,Z: `float` *Computed XYZ values.*

To compute XYZ using the same illuminant and observer as used in the measurement of the spectral data:

```
>>> X, Y, Z = XRCCPP.patch_spectral_to_XYZ(patch_id="A1")
>>> print(X,Y,Z)
11.461291517251293 10.499093307882513 7.333347299183066
```

For a new illuminant and observer:

```
>>> X, Y, Z = XRCCPP.patch_spectral_to_XYZ(patch_id="A1", illuminant="D50", observer=10)
>>> print(X,Y,Z)
>>> 11.908376423618433 10.534252547651414 5.4715573789714975
```

Also, it is possible to compute the XYZ values referred to *IlluminantFromCCT* or *MeasuredIlluminant* objects.

Export data:

ColourCheckerSpectral.to_ColourCheckerXYZ(illuminant = "D65", observer = 2, visible=False)

Method to export the XYZ data computed form the spectral data as a ColourCheckerXYZ.

Parameter:

illuminant: `str`, `Illuminant`, *Illuminant. Default: "D65".*

`IlluminantFromCCT`,

`MeasuredIlluminant`

observer: `str`, `int`, `Observer` *CIE Observer. Default: 2*

visible: `bool` *If True, use only the visible spectrum data. Default:* `False`.

Returns:

colour_checker_XYZ: `ColourCheckerXYZ` *Computed XYZ values.*

To export the XYZ data computed form the spectral data as a *ColourCheckerXYZ*:

```
>>> XRCCPP_XYZ = XRCCPP.to_ColourCheckerXYZ(illuminant="D65", observer=2)
>>> type(XRCCPP_XYZ)
coolpi.image.colourchecker.ColourCheckerXYZ
```

ColourCheckerSpectral.to_ColourCheckerLAB(illuminant="D65", observer=2, visible=False)

Method to export the LAB data computed from the spectral data as a ColourcheckerLAB.

Parameter:

illuminant: `str`, `Illuminant`, *Illuminant. Default: "D65".*

`IlluminantFromCCT`,

`MeasuredIlluminant`

observer: `str`, `int`, `Observer` *CIE Observer. Default: 2*

visible: `bool` *If True, use only the visible spectrum data. Default:* `False`.

Returns:

colour_checker_LAB: `ColourCheckerLAB` *Computed LAB values.*

To export the LAB data computed from the spectral data as a *ColourCheckerLAB*:

```
>>> XRCCPP_LAB = XRCCPP.to_ColourCheckerLAB(illuminant="D65", observer=2)
>>> type(XRCCPP_LAB)
coolpi.image.colourchecker.ColourCheckerLAB
```

ColourCheckerSpectral.as_pandas_dataframe()

Method to export the spectral data as a pandas `DataFrame` *object.*

Returns:

dataframe: `DataFrame` *Spectral data.*

To export the spectral data as a pandas `DataFrame`:

```
>>> XRCCPP_pd = XRCCPP.as_pandas_dataframe()
>>> type(XRCCPP_pd)
pandas.core.frame.DataFrame
```

ColourCheckerSpectral.export_data_to_json_file(path_json)

Method to export the spectral data to a JSON file.

Parameter:

path_json: `os` *path for the ouput JSON file.*

To export the spectral data as a JSON file:

```
>>> XRCCPP.export_data_as_json_file(path_json)
```

`__str__`:

`__str__`

Returns the string representation of the object. This method is called when the `print()` *or* `str()` *function is invoked on an object.*

```
>>> print(XRCCPP) # str method
ColourCheckerSpectral object: X-rite PASSPORT PHOTO (24+26+1 patches)
```

8.1.4 Plot

ColourCheckerSpectral.plot_colourchecker(show_figure, save_figure, output_path)

Method to create and display the spectral data of a colour checker using Matplotlib.

Parameters:

show_figure: `bool` *If True, the plot is shown. Default:* `True`.

save_figure: `bool` *If True, the figure is saved. Default:* `False`.

output_path: `os` *path for the ouput figure. Default:* `None`.

ColourCheckerSpectral objects can be represented as follows:

```
>>> XRCCPP.plot_colourchecker()
```

In addition, it is posible to save the figure setting `True` the opcion `save_figure` and introducing a valid `output_path`.

```
>>> XRCCPP.plot_colourchecker(show_figure = False, save_figure = True, output_path =
path_figure)
```

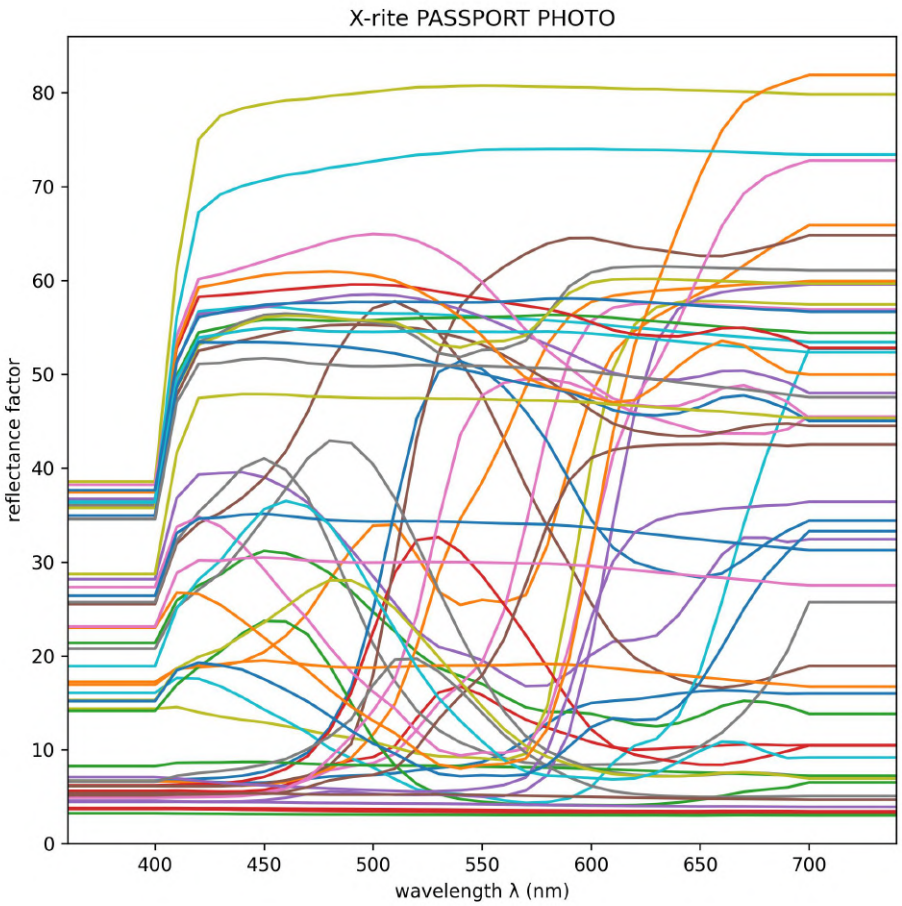

Figure 59: *ColourCheckerSpectral Plot*

ColourCheckerSpectral.plot_patches(patches_id, show_figure, save_figure, output_path)

Method to plot the spectral data of a set of given colour patches.

Parameters:

patches_id: `list` *Patches ID to plot.*

show_figure: `bool` *If True, the plot is shown. Default:* `True`.

save_figure: `bool` *If True, the figure is saved. Default:* `False`.

output_path: `os` *path for the ouput figure. Default:* `None`.

A set of colour patches can be represented as follows:

```
>>> XRCCPP.plot_patches(["A1","A2"])
```

In addition, it is posible to save the figure setting `True` the opcion `save_figure` and introducing a valid `output_path`.

```
>>> XRCCPP.plot_patches(["A1","A2"], show_figure = False, save_figure = True, output_path =
path_figure)
```

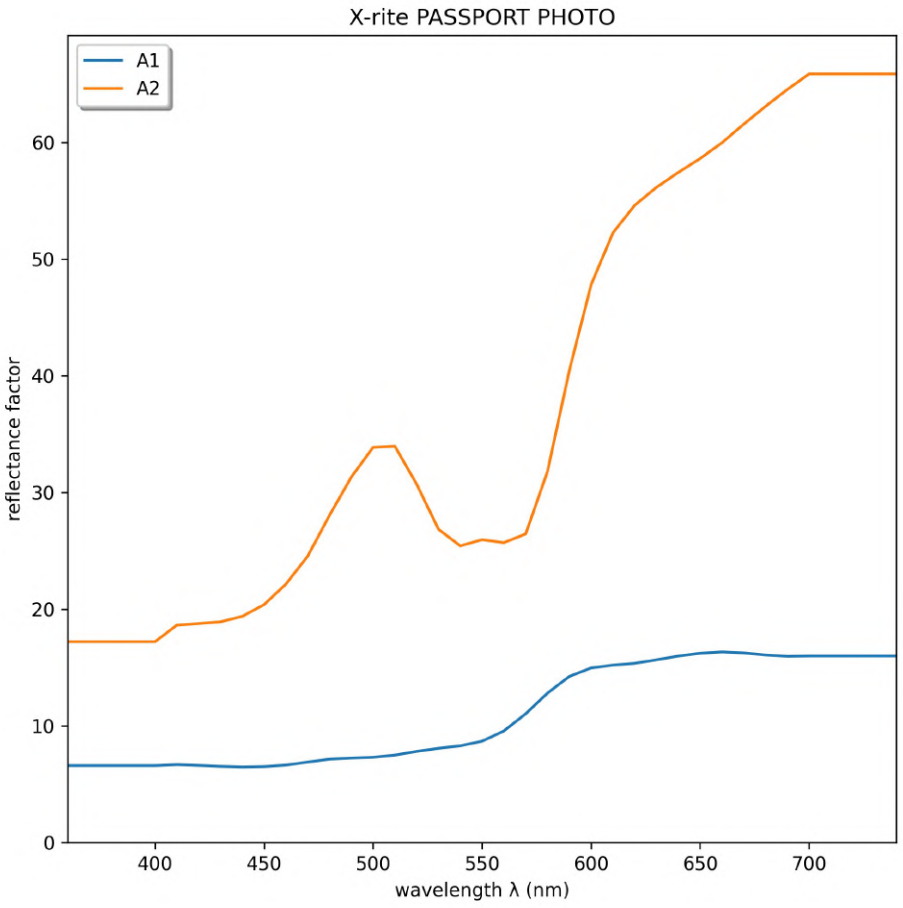

Figure 60: *ColourCheckerSpectral Plot Samples*

8.2 ColourCheckerXYZ

class ColourCheckerXYZ

ColourCheckerXYZ(checker_name, illuminant="D65", observer="2", data=None, path_json=None, params_csv=None, metadata={})

.set_instrument_measurement_as_metadata(metadata)

.get_colourchecker_number_of_patches()

.get_patch_data(patch_id)

.add_patch(patch_id, X, Y, Z)

.remove_patch(patch_id)

.to_ColourCheckerLAB(illuminant)

.as_pandas_dataframe()

.export_data_to_json_file(path_json)

The *ColourCheckerXYZ* class represents the XYZ data of the colour patches of a given colour checker measured using a colorimetric instrument, or computed from measured data (e.g. CIELAB or Reflectance).

8.2.1 Create an instance

ColourCheckerXYZ(checker_name, illuminant="D65", observer="2", data=None, path_json=None, params_csv=None, metadata={})

Parameters:

checker_name: `str` *Colour checker name or description.*

illuminant: `str`, `Illuminant`, *Illuminant. Default: "D65".*

`IlluminantFromCCT`,

`MeasuredIlluminant`,

observer: `str`, `int`, `Observer` *CIE Observer. Default: 2*

data: `dict` *Measured XYZ data. Default:* `None`.

path_json: `os` *JSON file. Default:* `None`.

params_csv: `dict` *Params to load a CSV file. Default:* `None`.

metadata: `dict` *Instrument measurement information. Default: {}.*

To *create an instance* of the *ColourCheckerXYZ* class, simply enter the required parameters as follows:

The *ColourCheckerXYZ* class can be instantiated from: data (XYZ data as `dict`); from a `path_json` (valid `os` path: JSON file with the XYZ data) or loaded from a CSV. In this case, a `params_csv` should be passed with the following format:

params_csv = dict(path_csv=path_file, csv_cols={"label":pos_label, "X":pos_X, "Y":pos_Y, "Z":pos_Z}, head=True)

⚠ **Alert**

Remember, Python starts counting from 0.

From XYZ data as `dict`:

```
>>> from coolpi.image.colourchecker import ColourCheckerXYZ
>>> data_as_dict = {
    "A1": [11.3116861299, 10.2886174806, 7.01505571903],
    "B2": [13.4997007812, 11.9790184311, 37.6369446057],
    "C3": [19.887537955, 11.7003493539, 5.10489787605],
    "D4": [18.0291683689, 19.1801885705, 21.2166071106]
    }
>>> colourchecker_metadata = {"NameColorChart": "GretagMacbeth", "Instrument":
    "Konica Minolta CM-600d", "Illuminant": "D65", "Observer": 2}
>>> GM_XYZ = ColourCheckerXYZ(checker_name="GM", illuminant="D65",
    observer=2, data = data_as_dict, metadata=colourchecker_metadata)
```

From a CSV file:

```
>>> from coolpi.image.colourchecker import ColourCheckerXYZ
>>> params_file = dict(path_csv=path_file, csv_cols={"label":0, "X":6, "Y":7,
    "Z":8}, head=True)
>>> GM_XYZ = ColourCheckerXYZ(checker_name="GM", illuminant="D65", observer=2,
    params_csv = params_file, metadata=colourchecker_metadata)
```

From a JSON file:

```
>>> from coolpi.image.colourchecker import ColourCheckerXYZ
>>> GM_XYZ = ColourCheckerXYZ(checker_name="GM", illuminant="D65", observer=2,
    path_json = path_file, metadata=colourchecker_metadata)
```

8.2.2 Attributes

The class *attributes* are: *type* (returns a `str` with the description of the main class), *subtype* (returns a `str` with the description of the object), *name* (returns a `str` with the *ColourChecker* name or description), *illuminant* (returns the *Illuminant*, *IlluminantFromCCT* or *MeasuredIlluminant* instance), *observer* (returns the *Observer* instance), *patches* (returns a `dict` with the spectral data), *patches_id* (returns a `list` with the patches id of the *ColourChecker*) and *metadata* (returns a `dict` with the measurement details).

```
>>> print(GM_XYZ.type)
ColourChecker object
>>> print(GM_XYZ.subtype)
ColourChecker XYZ data
>>> print(GM_XYZ.name)
GM
>>> print(GM_XYZ.illuminant) # str method
Illuminant object: CIE D65 standard illuminant
>>> print(GM_XYZ.observer) # str method
2º standard observer (CIE 1931)
>>>#print(GM_XYZ.patches) # XYZ data as dict
>>>print(GM_XYZ.patches_id)
dict_keys(['A1', 'A2', 'A3', 'A4', 'A5', 'A6', 'B1', 'B2', 'B3', 'B4', 'B5', 'B6',
'C1', 'C2', 'C3', 'C4', 'C5', 'C6', 'D1', 'D2', 'D3', 'D4', 'D5', 'D6'])
>>>print(GM_XYZ.metadata.keys()) # Measurement information as dict
dict_keys(['NameColorChart', 'Instrument', 'Illuminant', 'Observer'])
```

8.2.3 Methods

Patch management:

> ***ColourCheckerXYZ.get_colourchecker_number_of_patches()***
>
> *Method to get the total number of colour patches of the colour checker.*
>
> *Returns:*
>
> *num_patches:* `int` *Total number of patches.*

To get the total number of patches:

```
>>> num_patches = GM_XYZ.get_colourchecker_number_of_patches()
>>>print(num_patches)
24
```

> ***ColourCheckerXYZ.get_patch_data(patch_id)***
>
> *Method to get the XYZ data of a given patch.*
>
> *Parameter:*

patch_id: `str` Patch ID.

Returns:

X,Y,Z: `float` XYZ data.

To get the XYZ values of a given patch:

```
>>> X, Y, Z = GM_XYZ.get_patch_data("A1")
>>> print(X, Y, Z)
11.3116861299 10.2886174806 7.01505571903
```

The input `patch_id` should be a valid patch. Otherwise, a `PatchError` is raised.

```
>>> X, Y, Z = GM_XYZ.get_patch_data("P1")
PatchError: Patch id not present in the current ColourChecker
```

ColourCheckerXYZ.add_patch(patch_id, X, Y, Z)

Method to add a patch into the colour checker.

Parameter:

patch_id: `str` Patch ID.

X,Y,Z: `float` XYZ values.

To add a patch:

```
>>> GM_XYZ.add_patch(patch_id="P1", X=100, Y=100, Z=100)
>>> print(GM_XYZ.get_colourchecker_number_of_patches())
25
```

ColourCheckerXYZ.remove_patch(patch_id)

Method to remove a patch from the colour checker.

Parameter:

patch_id: `str` Patch ID.

To remove a patch from the colour checker:

```
>>> GM_XYZ.remove_patch("P1")
>>>print(GM_XYZ.get_colourchecker_number_of_patches())
24
```

Export data:

ColourCheckerXYZ.to_ColourCheckerLAB(illuminant = “D65")

Method to export the LAB data computed from the XYZ values as a ColourCheckerLAB.

Parameter:

illuminant: `str`, `Illuminant`, `IlluminantFromCCT`, `MeasuredIlluminant` Illuminant. Default: “D65".

Returns:

colour_checker_LAB: `ColourCheckerLAB` Computed LAB values.

To export the LAB data computed from the XYZ data as a *ColourCheckerLAB*:

```
>>> GM_LAB = GM_XYZ.to_ColourCheckerLAB(illuminant="D65")
>>> type(GM_LAB)
coolpi.image.colourchecker.ColourCheckerLAB
```

ColourCheckerXYZ.as_pandas_dataframe()

Method to export the XYZ data as a pandas `DataFrame` object.

Returns:

dataframe: `DataFrame` Spectral data.

To export the XYZ data as a pandas `DataFrame`:

```
>>> GM_XYZ_pd = GM_XYZ.as_pandas_dataframe()
>>> type(GM_XYZ_pd)
pandas.core.frame.DataFrame
```

ColourCheckerXYZ.export_data_to_json_file(path_json)

Method to export the XYZ data to a JSON file.

Parameter:

path_json: `os` *path for the ouput JSON file.*

To export the XYZ data as a JSON file:

```
>>> GM_XYZ.export_data_to_json_file(path_json)
```

`__str__`:

`__str__`

Returns the string representation of the object. This method is called when the `print()` *or* `str()` *function is invoked on an object.*

```
>>> print(GM_XYZ) # str method
ColourCheckerXYZ object: GM
```

8.3 ColourcheckerLAB

class ColourcheckerLAB

ColourcheckerLAB(checker_name, illuminant="D65", observer="2", data=None, path_json=None, params_csv=None, metadata={})

.set_instrument_measurement_as_metadata(metadata)

.get_colourchecker_number_of_patches()

.get_patch_data(patch_id)

.add_patch(patch_id, L, a, b)

.remove_patch(patch_id)

.to_ColourCheckerXYZ(illuminant)

.as_pandas_dataframe()

.export_data_to_json_file(path_json)

.plot_colourchecker(show_figure, save_figure, output_path)

The *ColourCheckerLAB* class represents the LAB data of the colour patches of a given colour checker measured using a colorimetric instrument, or computed from

measured data (e.g. XYZ or Reflectance).

8.3.1 Create an instance

ColourcheckerLAB(checker_name, illuminant="D65", observer="2", data=None, path_json=None, params_csv=None, metadata={})

Parameters:

checker_name: `str` *Colour checker name or description.*

illuminant: `str`, `Illuminant`, `IlluminantFromCCT`, `MeasuredIlluminant` *Illuminant. Default: "D65".*

observer: `str`, `int`, `Observer` *CIE Observer. Default: 2*

data: `dict` *Measured LAB data. Default:* `None`.

path_json: `os` *JSON file. Default:* `None`.

params_csv: `dict` *Params to load a CSV file. Default:* `None`.

metadata: `dict` *Instrument measurement information. Default: {}.*

To *create an instance* of the *ColourcheckerLAB* class, simply enter the required parameters as follows:

The *ColourcheckerLAB* class can be instantiated from: data (LAB data as `dict`); from a `path_json` (valid `os` path: JSON file with the LAB data) or loaded from a CSV. In this case, a `params_csv` should be passed with the following format:

params_csv = dict(path_csv=path_file, csv_cols={"label":pos_label, "L":pos_L, "A":pos_a, "B":pos_b}, head=True)

⚠ **Alert**

Remember, Python starts counting from 0.

From LAB data as `dict`:

```
>>> from coolpi.image.colourchecker import ColourCheckerLAB
>>> data_as_dict = {
    "A1": [39.3252, 11.7671, 13.4596],
    "B2": [41.6404, 17.3296, -45.8501],
    "C3": [42.5921, 48.7030, 25.8693],
    "D4": [52.0973, -0.1819, -0.6007]
    }
>>> colourchecker_metadata = {"NameColorChart": "Calibrite colorchecker CLASSIC",
    "Manufacturer":["Calibrite", "Made in USA", 850028833087, 2021], "Measurement Date":
    [[2022, 3, 11], [10, 13, 40]], "Instrument": "Konica Minolta CM-600d", "Illuminant":
    "D65", "Observer": 2, "Geometry": "di:8, de:8", "Specular Component":"SCI",
    "Measurement Area": "MAV(8mm)"}
>>> CCC_LAB = ColourCheckerLAB(checker_name="CCC", illuminant="D65", observer=2,
    data = data_as_dict, metadata=colourchecker_metadata)
```

From a CSV file:

```
>>> from coolpi.image.colourchecker import ColourCheckerLAB
>>> params_file = dict(path_csv=path_file, csv_cols={"label":0, "L":1, "A":2, "B":3},
    head=True)
>>> CCC_LAB = ColourCheckerLAB(checker_name="CCC", illuminant="D65", observer=2,
    params_csv = params_file, metadata=colourchecker_metadata)
```

From a JSON file:

```
>>> from coolpi.image.colourchecker import ColourCheckerLAB
>>> CCC_LAB = ColourCheckerLAB(checker_name="CCC", illuminant="D65", observer=2,
    path_json = path_file, metadata=colourchecker_metadata)
```

8.3.2 Attributes

The class *attributes* are: *type* (returns a `str` with the description of the main class), *subtype* (returns a `str` with the description of the object), *name* (returns a `str` with the *ColourChecker* name or description), *illuminant* (returns the *Illuminant*, *IlluminantFromCCT* or *MeasuredIlluminant* instance), *observer* (returns the *Observer* instance), *patches* (returns a `dict` with the spectral data), *patches_id* (returns a `list` with the patches id of the *ColourChecker*) and *metadata* (returns a `dict` with the measurement details).

```
>>> print(CCC_LAB.type)
ColourChecker object
>>> print(CCC_LAB.subtype)
ColourChecker LAB data
>>> print(CCC_LAB.name)
CCC
>>> print(CCC_LAB.illuminant)
Illuminant object: CIE D65 standard illuminant
>>> print(CCC_LAB.observer)
2º standard observer (CIE 1931)
>>> #print(CCC_LAB.patches) # XYZ data as dict
>>> print(CCC_LAB.patches_id)
dict_keys(['A1', 'A2', 'A3', 'A4', 'A5', 'A6', 'B1', 'B2', 'B3', 'B4', 'B5', 'B6',
'C1', 'C2', 'C3', 'C4', 'C5', 'C6', 'D1', 'D2', 'D3', 'D4', 'D5', 'D6'])
>>> print(CCC_LAB.metadata.keys()) # Measurement information as dict
dict_keys(['NameColorChart', 'Manufacturer', 'Measurement Date', 'Instrument',
'Illuminant', 'Observer', 'Geometry', 'Specular Component', 'Measurement Area'])
```

8.3.3 Methods

Patch management:

ColourcheckerLAB.get_colourchecker_number_of_patches()

Method to get the total number of colour patches of the colour checker.

Returns:

num_patches: `int` *Total number of patches.*

To get the total number of patches:

```
>>> num_patches = CCC_LAB.get_colourchecker_number_of_patches()
>>> print(num_patches)
24
```

ColourcheckerLAB.get_patch_data(patch_id)

Method to get the LAB data of a given patch.

Parameter:

patch_id: `str` *Patch ID.*

Returns:

L,a,b: `float` *LAB data.*

To get the LAB values of a given patch:

```
>>> L, a, b = CCC_LAB.get_patch_data("A1")
>>> print(L, a, b)
39.3252 11.7671 13.4596
```

The input `patch_id` should be a valid patch. Otherwise, a `PatchError` is raised.

```
>>> L, a, b = CCC_LAB.get_patch_data("P1")
PatchError: Patch id not present in the current ColourChecker
```

ColourcheckerLAB.add_patch(patch_id, L, a, b)

Method to add a patch into the colour checker object.

Parameter:

patch_id: `str` *Patch ID.*

L,a,b: `float` *LAB values.*

To add a patch:

```
>>> CCC_LAB.add_patch(patch_id="P1", L=0, a=0, b=0)
>>> print(CCC_LAB.get_colourchecker_number_of_patches())
25
```

ColourcheckerLAB.remove_patch(patch_id)

Method to remove a patch from the colour checker.

Parameter:

patch_id: `str` *Patch ID.*

To remove a patch from the colour checker:

```
>>> CCC_LAB.remove_patch("P1")
>>> print(CCC_LAB.get_colourchecker_number_of_patches())
24
```

Export data:

ColourcheckerLAB.to_ColourCheckerXYZ(illuminant = “D65")

Method to export the XYZ data computed from the LAB data as a

ColourCheckerXYZ.

Parameter:

illuminant: `str`, `Illuminant`, *Illuminant. Default: "D65".*

`IlluminantFromCCT`,

`MeasuredIlluminant`

Returns:

colour_checker_XYZ: `ColourCheckerXYZ` *Computed XYZ values.*

To export the XYZ data computed from the LAB data as a *ColourCheckerXYZ*:

```
>>> CCC_XYZ = CCC_LAB.to_ColourCheckerXYZ(illuminant="D65")
>>> type(CCC_XYZ)
coolpi.image.colourchecker.ColourCheckerXYZ
```

ColourcheckerLAB.as_pandas_dataframe()

Method to export the LAB data as a pandas `DataFrame` *object.*

Returns:

dataframe: `DataFrame` *Spectral data.*

To export the LAB data as a pandas `DataFrame`:

```
>>> CCC_LAB_pd = CCC_LAB.as_pandas_dataframe()
>>> type(CCC_LAB_pd)
pandas.core.frame.DataFrame
```

ColourcheckerLAB.export_data_to_json_file(path_json)

Method to export the LAB data to a JSON file.

Parameter:

path_json: `os` *path for the ouput JSON file.*

To export the LAB data as a JSON file:

```
>>> CCC_LAB.export_data_to_json_file(path_json)
```

__str__:

> **__str__**
>
> *Returns the string representation of the object. This method is called when the `print()` or `str()` function is invoked on an object.*

```
>>> print(CCC_LAB) # str method
ColourCheckerLAB object: CCC
```

8.3.4 Plot

> ***ColourcheckerLAB.plot_colourchecker(show_figure, save_figure, output_path)***
>
> *Method to create and display the LAB data of a colour checker using Matplotlib.*
>
> *Parameters:*
>
> *show_figure:* `bool` *If True, the plot is shown. Default:* `True`*.*
>
> *save_figure:* `bool` *If True, the figure is saved. Default:* `False`*.*
>
> *output_path:* `os` *path for the ouput figure. Default:* `None`*.*

ColourcheckerLAB objects can be represented as follows:

```
>>> CCC_LAB.plot_colourchecker()
```

In addition, it is posible to save the figure setting `True` the opcion `save_figure` and introducing a valid `output_path`.

```
>>> CCC_LAB.plot_colourchecker(show_figure = False, save_figure = True, output_path =
path_figure)
```

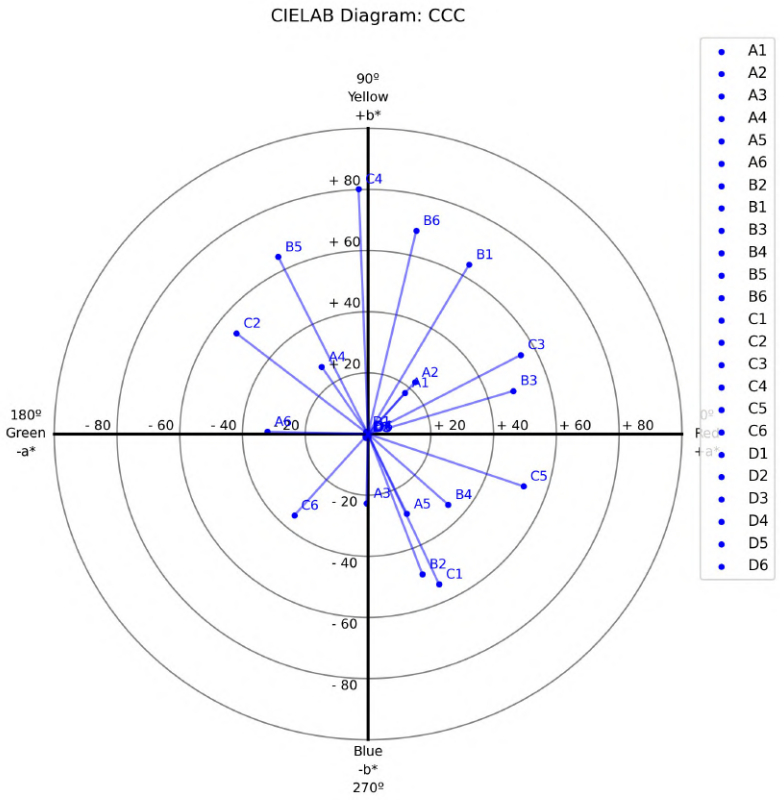

Figure 61: *ColourcheckerLAB Plot*

8.4 ColourCheckerRGB

class ColourCheckerRGB

ColourCheckerRGB(checker_name, illuminant="D65", observer="2", data=None, path_json=None, params_csv=None, metadata={})

.set_instrument_measurement_as_metadata(metadata)

.get_colourchecker_number_of_patches()

.get_patch_data(patch_id)

.add_patch(patch_id, L, a, b)

.remove_patch(patch_id)

.as_pandas_dataframe()

.export_data_to_json_file(path_json)

The *ColourCheckerRGB* class represents the RGB data of the colour patches of a given colour checker extracted from an image.

8.4.0.1 Create an instance

ColourCheckerRGB(checker_name, illuminant="D65", observer="2", data=None, path_json=None, params_csv=None, metadata={})

Parameters:

checker_name: `str` *Colour checker name or description.*

illuminant: `str`, `Illuminant`, `IlluminantFromCCT`, `MeasuredIlluminant` *Illuminant. Default: "D65".*

observer: `str`, `int`, `Observer` *CIE Observer. Default: 2*

data: `dict` *Measured LAB data. Default:* `None`.

path_json: `os` *JSON file. Default:* `None`.

params_csv: `dict` *Params to load a CSV file. Default:* `None`.

metadata: `dict` *Instrument measurement information. Default: {}.*

To *create an instance* of the *ColourCheckerRGB* class, simply enter the required parameters as follows:

The *ColourcheckerRGB* class can be instantiated from: data (RGB data as `dict`); from a `path_json` (valid `os` path: JSON file with the RGB data) or loaded from a CSV. In this case, a `params_csv` should be passed with the following format:

params_csv = dict(path_csv=path_file, csv_cols={"label":pos_label, "R":pos_R, "G":pos_G, "B":pos_B}, head=True)

⚠ **Alert**

Remember, Python starts counting from 0.

From RGB data as `dict`:

```
>>> from coolpi.image.colourchecker import ColourCheckerRGB
>>> data_as_dict = {
    'A1': (482.780625, 389.610625, 117.115625),
    'A2': (1614.445, 1226.3875, 415.245),
    'A3': (535.3375, 809.37, 439.4275),
    'A4': (424.7675, 594.4634375, 161.335),
    'A5': (937.2725, 1077.8575, 586.9075),
    'A6': (976.41875, 2012.144375, 825.36375),
    'B1': (1701.5975, 995.8575, 164.24),
    'B2': (333.04875, 526.209375, 418.59),
    'B3': (1487.425, 622.858125, 198.82875),
    'B4': (397.1575, 298.0428125, 178.135),
    'B5': (1371.31, 1962.948125, 389.39875),
    'B6': (2316.55625, 1778.48375, 274.41),
    'C1': (157.275, 272.4553125, 272.934375),
    'C2': (424.585, 939.61, 251.305),
    'C3': (1199.7975, 398.519375, 110.97375),
    'C4': (2439.41, 2327.4025, 347.6475),
    'C5': (1561.56625, 749.494375, 380.11875),
    'C6': (323.215, 946.37125, 608.13375),
    'D1': (2691.9025, 3185.595, 1204.645),
    'D2': (1891.29875, 2260.2575, 867.435),
    'D3': (1243.5, 1498.68875, 577.32),
    'D4': (690.24375, 826.01, 315.67875),
    'D5': (344.115, 418.266875, 165.17125),
    'D6': (142.025, 169.595, 69.49)}
>>> colourchecker_metadata = {"NameColorChart": "X-Rite Colorchecker PASSPORT PHOTO",
    "Manufacturer":["X-RITE", "Made in USA", 710762440005, "09/2014"], "Measurement Date":
    [[2022, 3, 11], [11, 12, 26]], "Instrument": "Nikon D5600", "Illuminant": JNA,
"Observer": 2}
>>> XRCCPP_RGB = ColourCheckerRGB(checker_name="XRCCPP", illuminant=JNA, observer=2,
    data = data_as_dict, metadata=colourchecker_metadata)
```

From a CSV file:

```
>>> from coolpi.image.colourchecker import ColourCheckerRGB
>>> params_file = dict(path_csv=path_file, csv_cols={"label":0, "R":1, "G":2, "B":3},
    head=True)
>>> XRCCPP_RGB = ColourCheckerRGB(checker_name="XRCCPP", illuminant="D65",
    observer=2, params_csv = params_file, metadata=colourchecker_metadata)
```

From a JSON file:

```
>>> from coolpi.image.colourchecker import ColourCheckerRGB
>>> XRCCPP_RGB = ColourCheckerRGB(checker_name="XRCCPP", illuminant="D65",
    observer=2, path_json = path_file, metadata=colourchecker_metadata)
```

8.4.1 Attributes

The class *attributes* are: *type* (returns a `str` with the description of the main class), *subtype*

(returns a `str` with the description of the object), *name* (returns a `str` with the *ColourChecker* name or description), *illuminant* (returns the *Illuminant*, *IlluminantFromCCT* or *MeasuredIlluminant* instance), *observer* (returns the *Observer* instance), *patches* (returns a `dict` with the spectral data), *patches_id* (returns a `list` with the patches id of the *ColourChecker*) and *metadata* (returns a `dict` with the measurement details).

```
>>> print(XRCCPP_RGB.type)
ColourChecker object
>>> print(XRCCPP_RGB.subtype)
ColourChecker RGB data
>>> print(XRCCPP_RGB.name)
XRCCPP
>>> print(XRCCPP_RGB.illuminant) # str method
MeasuredIlluminant object: Illuminant JNA.
>>> print(XRCCPP_RGB.observer) # str method
2º standard observer (CIE 1931)
>>>#print(XRCCPP_RGB.patches) # XYZ data as dict
>>> print(XRCCPP_RGB.patches_id)
dict_keys(['A1', 'A2', 'A3', 'A4', 'A5', 'A6', 'B1', 'B2', 'B3', 'B4', 'B5',
'B6', 'C1', 'C2', 'C3', 'C4', 'C5', 'C6', 'D1', 'D2', 'D3', 'D4', 'D5', 'D6'])
>>> print(XRCCPP_RGB.metadata.keys()) # Measurement information as dict
dict_keys(['NameColorChart', 'Manufacturer', 'Measurement Date', 'Instrument',
'Illuminant', 'Observer'])
```

8.4.2 Methods

Patch management:

> ***ColourCheckerRGB.get_colourchecker_number_of_patches()***
>
> *Method to get the total number of colour patches of the colour checker.*
>
> *Returns:*
>
> *num_patches:* `int` *Total number of patches.*

To get the total number of patches:

```
>>> num_patches = XRCCPP_RGB.get_colourchecker_number_of_patches()
>>> print(num_patches)
24
```

> ***ColourCheckerRGB.get_patch_data(patch_id)***
>
> *Method to get the LAB data of a given patch.*
>
> *Parameter:*

patch_id: `str` *Patch ID.*

Returns:

R,G,B: `float` *RGB data.*

To get the RGB values of a given patch:

```
>>> R, G, B = XRCCPP_RGB.get_patch_data("A1")
>>> print(R, G, B)
482.780625 389.610625 117.115625
```

The input `patch_id` should be a valid patch. Otherwise, a `PatchError` is raised.

```
>>> R, G, B = XRCCPP_RGB.get_patch_data("P1")
PatchError: Patch id not present in the current ColourChecker
```

ColourCheckerRGB.add_patch(patch_id, R, G, B)

Method to add a patch into the colour checker object.

Parameter:

patch_id: `str` *Patch ID.*

R,G,B: `float` *RGB values.*

To add a patch:

```
>>> XRCCPP_RGB.add_patch(patch_id="P1", R=0, G=0, B=0)
>>> print(XRCCPP_RGB.get_colourchecker_number_of_patches())
25
```

ColourCheckerRGB.remove_patch(patch_id)

Method to remove a patch from the colour checker.

Parameter:

patch_id: `str` *Patch ID.*

To remove a patch from the colour checker:

```
>>> XRCCPP_RGB.remove_patch("P1")
>>> print(XRCCPP_RGB.get_colourchecker_number_of_patches())
24
```

Export data:

ColourCheckerRGB.as_pandas_dataframe()

Method to export the RGB data as a pandas `DataFrame` object.

Returns:

dataframe: `DataFrame` Spectral data.

To export the RGB data as a pandas `DataFrame`:

```
>>> XRCCPP_RGB_pd = XRCCPP_RGB.as_pandas_dataframe()
>>> type(XRCCPP_RGB_pd)
pandas.core.frame.DataFrame
```

ColourCheckerRGB.export_data_to_json_file(path_json)

Method to export the RGB data to a JSON file.

Parameter:

path_json: `os` path for the ouput JSON file.

To export the RGB data as a JSON file:

```
>>> XRCCPP_RGB.export_data_to_json_file(path_json)
```

`__str__`:

`__str__`

Returns the string representation of the object. This method is called when the `print()` or `str()` function is invoked on an object.

```
>>> print(XRCCPP_RGB) # str method
ColourCheckerRGB object: XRCCPP
```

8.5 RawImage

class RawImage

RawImage(path_raw, metadata={}, method="postprocess")

.set_metadata(metadata)

.set_image_illuminant(illuminant)

.set_observer(observer)

.get_raw_image_information()

.get_raw_single_channel_from_metadata()

.extract_rgb_patch_data_from_image(center, size)

.extract_colourchecker_rgb_patches(checker_name, corners_image, size_rect)

.get_ColourCheckerRGB(checker_name)

.get_patch_from_colourchecker(checker_name, patch_id)

*.automatic_image_processing(show_image, save_image, output_path, method, **krawargs)*

.get_camera_whitebalance()

.get_daylight_whitebalance()

*.compute_wb_multipliers(**params)*

*.estimate_wb_multipliers(method, **params)*

.set_whitebalance_multipliers(wb_multipliers)

.get_whitebalance_multipliers()

.apply_white_balance(show_image, method, save_image, output_path, bits)

.get_raw_rgb_white_balanced_image()

.set_RGB_to_XYZ_matrix(RGB_to_XYZ)

.apply_colour_correction(show_image, method, save_image, output_path, bits)

.compute_image_colour_quality_assessment(colourchecker_name)

.plot_rgb_histogram(show_figure, save_figure, output_path, split_per_channel)

.show(method)

.show_colourchecker(colourchecker_name, show_figure, save_figure,

output_path, method, parameters_draw, bits)

.save(output_path, bits)

The *RawImage* class represents the RGB RAW data extracted from a RAW image.

8.5.1 Create an instance

RawImage(path_raw, metadata={}, method="postprocess")

Parameters:

path_raw: `os` *RAW image path.*

metadata: `dict` *Image information as metadata. Default: {}.*

method: `str` *Method to get the RAW RGB demosaiced data.*

Default: "postprocess".

To *create an instance* of the *RawImage* class, simply enter the required parameters as follows:

```
>>> metadata = {"Camera": "Nikon D5600", "image_size":[4008, 6008],
    "Date": [[2022, 6, 19], [8, 29, 00]], "ColorChecker": "XRCCPP",
    "illuminant":JNA, "observer": 2}
>>> raw_image = RawImage(path_raw, metadata, method = "postprocess")
```

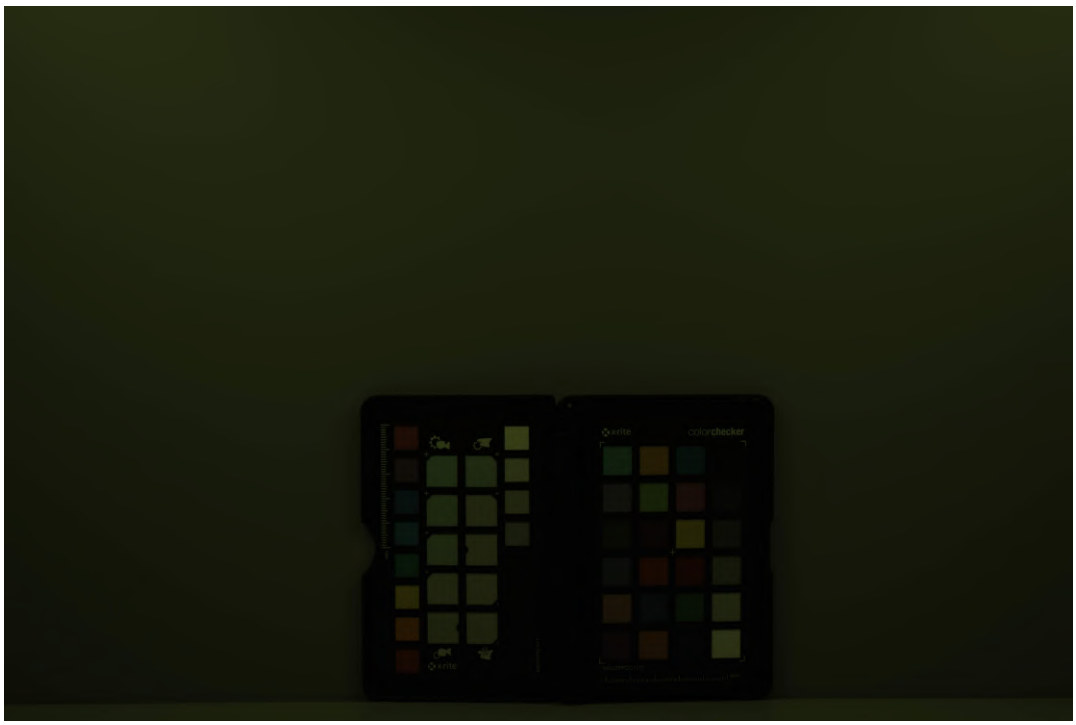

Figure 62: *RawImage*

The *RawImage* metadata can be set separately with the method `.set_metadata(metadata)`. If the illuminant under which the image has been taken is known (e.g. the spectral power distribution of the illuminant has been measured using specific instrumentation), it is recommended to introduce the illuminant and the CIE observer (default 2) as keys in the metadata for colorimetric computations. However, the illuminant and observer can be set separately with the methods `.set_image_illuminant(illuminant)` and `.set_observer(observer)`.

There are two different methods for the extraction of RAW RGB data from RAW images, called “raw_image” and “postprocess”. Both methods are based on the rawpy package. The “raw_image” method uses the “raw_image” data (RAW data as a single channel without any preprocessing), and computes the RAW RGB demosaiced image using its own functions. The “postprocess” is based on the `rawpy.postprocess` functions but controlling the parameters to ensure minimal processing of the RAW data. Thus, the RAW RGB data extracted is as close as possible to the original data stored by the camera sensor.

Since the results are identical using both methods (data from `raw_image`, or `postprocessing`), we will use the `rawpy.postprocess` function: in this way it is not necessary to take into account the sensor rotation angle of some cameras (e.g. Fujifilm). Thus, the RAW RGB data are obtained automatically.

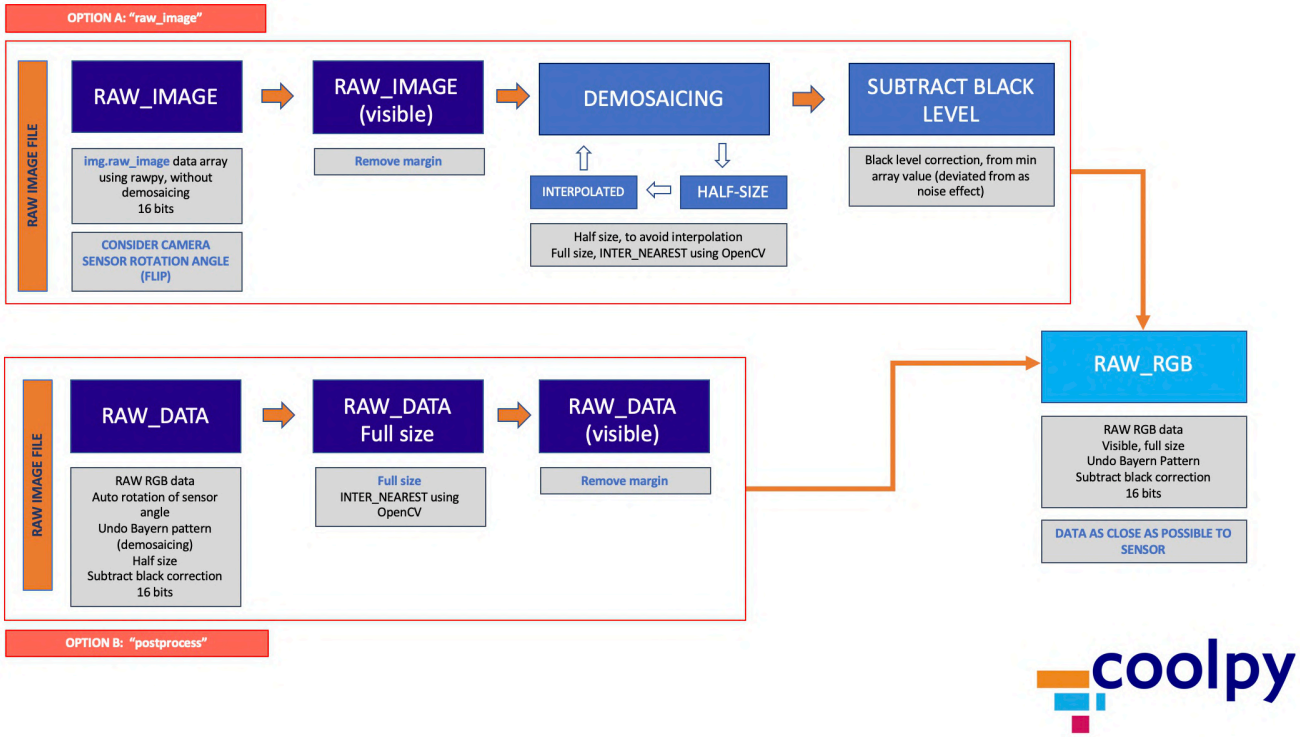

Figure 63: *RawImage RAW RGB data extraction*

8.5.2 Attributes

The class *attributes* are: *type* (returns a `str` with the description of the main class), *subtype* (returns a `str` with the description of the object), *path* (returns an `os` with the image path), *raw_attributes* (returns a `dict` with the properties of the RAW image), *rgb_data* (returns a `numpy.ndarray` with the RAW RGB demosaiced data), *metadata* (returns a `dict` with the details on image acquisition), *illuminant* (returns the *Illuminant*, *IlluminantFromCCT* or *MeasuredIlluminant* instance), and *observer* (returns the *Observer* instance)

```
>>> print(raw_image.type)
Image object
>>> print(raw_image.subtype)
RAW Image object
>>> print(raw_image.path)
res/img/INDIGO_2022-06-19_NikonD5600_0017.NEF
>>> print(raw_image.raw_attributes.keys()) # same as .get_raw_image_information()
dict_keys(['black_level_per_channel', 'camera_whitebalance', 'colour_desc',
'daylight_whitebalance', 'num_colours', 'raw_colours', 'raw_image', 'raw_image_visible',
'raw_pattern', 'xyz_cam_matrix', 'tone_curve', 'white_level', 'raw_image_size',
'processed_image_size'])
>>> print(raw_image.rgb_data.shape)
(4008, 6008, 3)
>>> print(raw_image.metadata)
{'Camera': 'Nikon D5600', 'image_size': [4008, 6008], 'Date': [[2022, 6, 19], [8, 29, 0]],
'ColorChecker': 'XRCCPP', 'illuminant': JNA, 'observer': 2}
>>> print(raw_image.illuminant)
MeasuredIlluminant object: Illuminant JNA.
>>> print(raw_image.observer)
2º standard observer (CIE 1931)
```

Alert

For advanced users, it is possible to access the RAW data before demosaicing by using the method `.get_raw_single_channel_from_metadata()`, but the sensor rotation angle shall be taken into account if applicable.

8.5.3 Methods

Automatic Image Processing:

RawImage.automatic_image_processing(show_image, save_image, output_path, method, **krawargs)

Method for the automatic processing of the RAW image (using `rawpy.postprocess`).

Parameters:

show_image: `bool` If True, the plot is shown. Default: `True`.

save_image: `bool` If True, the figure is saved. Default: `False`.

output_path: `os` path for the ouput figure. Default: `None`.

method: `str` path for the ouput figure. Default: `None`.

krawargs: `dict` *Optional parameters, e.g. to use a custom white balance.*

Returns:

processed_img: `ProcessedImage` *Processed image.*

For the automatic processing of a *RawImage* object, simply enter the required parameters as follows:

```
>>> rgb_processed = raw_image.automatic_image_processing(show_image = True,
save_image = False, output_path = None, method="matplotlib")
```

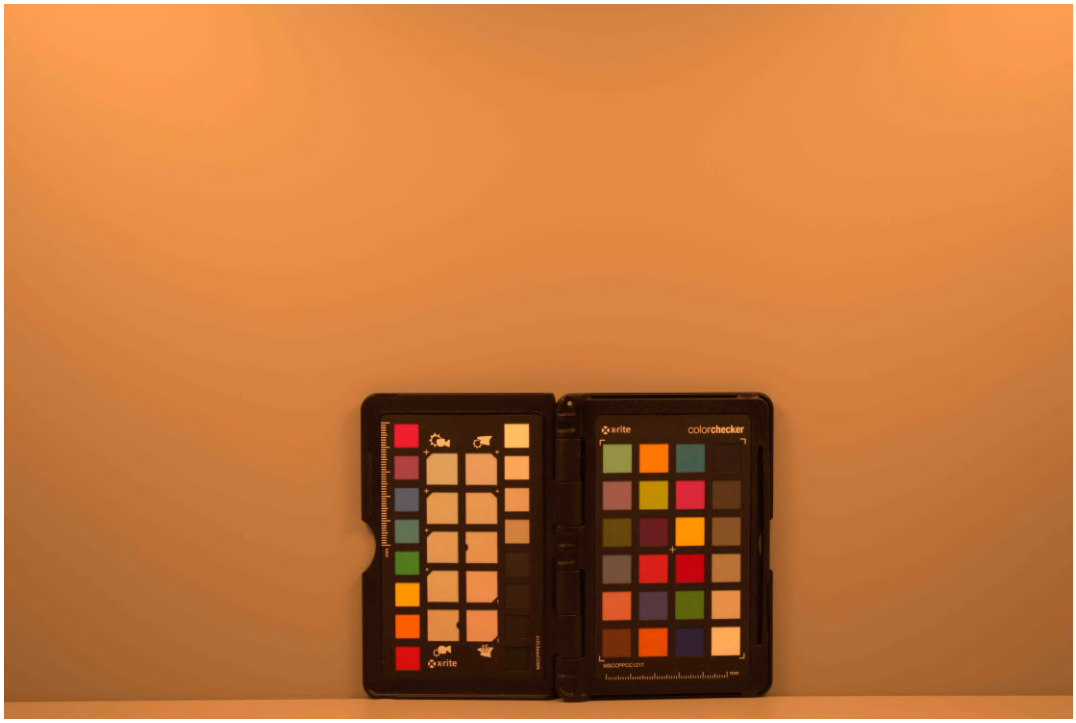

Figure 64: *RawImage Automatic Image Processing*

The default parameters used for the image processing are: AHD demosaicing algorithm, full size, without applying noise reduction and any white balance algorithm, sRGB as output color space, and as an 8 bits image.

However, custom parameters can be introduced as optional `krawargs`, e.g. to apply custom white balance multipliers:

```
>>> krawargs = dict(use_camera_wb=False, use_auto_wb=False,
    user_wb=[1.21, 1.0, 2.60, 1.0], output_bps=16)
>>> rgb_processed_custom = raw_image.automatic_image_processing(show_image = True,
    save_image = False, output_path = None, method="matplotlib", **krawargs)
```

Figure 65: *RawImage Custom Image Processing*

> ⚠ **Alert**
>
> *The RawImage automatic image processing does not include a colour correction pipeline.*

RGB patch data extraction:

> ***RawImage.extract_rgb_patch_data_from_image(center, size)***
>
> *Method to extract the RGB data of a patch from an image.*
>
> *Parameters:*
>
> *center:* `list` *Center coordinates of the patch as [x=col, y=row].*
>
> *size:* `int` *Patch size in px.*
>
> *Returns:*
>
> *r,g,b:* `float` *RGB average data.*

To extract the RGB data of a single patch from the image:

```
>>> r_grey, g_grey, b_grey = raw_image.extract_rgb_patch_data_from_image([2469,3490], 70) #
"F1"
>>> print(r_grey, g_grey, b_grey)
1780.4232653061224 2112.1951020408164 801.9648979591836
```

RawImage.extract_colourchecker_rgb_patches(checker_name, corners_image, size_rect=40)

Method to extract the RGB data of a colour checker from an image.

Parameters:

checker_name: `str` *Colour checker name.*

corners_image: `dict` *Colour checker corner coordinates as {"TopLeft":[x,y], "TopRight":[x,y], "BottomRight":[x,y], "BottomLeft":[x,y]}.*

size_rect: `int` *Size in px. Default: 40.*

The following colour checkers have been implemented:

- Calibrite PASSPORT PHOTO 2: "CCPP2_24", "CCPP2_26".
- Calibrite PASSPORT VIDEO: "CCPPV_24", "CCPPV_3".
- Munsell Soil Color Book (Ed.1994): Colour checker name and page, e.g. "MSCB1994_GLEY1", "MSCB1994_5Y".
- Munsell Soil Color Book (Ed.2009): Colour checker name and page, e.g. "MSCB2009_GLEY1", "MSCB2009_GLEY2".
- Spyder Checker: "SCK100_48", "SCK100_7".
- X-rite PASSPORT PHOTO: "XRCCPP_24", "XRCCPP_26".

Some of the colour checkers available in coolpi have been separated into smaller sections (either because of their shape, or because they are in single sheets like the Munsell Book).

The corner coordinates shall refer to the markings or crosses if present. In case the colour checker does not have any reference mark, the colour checker should be positioned horizontally so that there are more patches horizontally than vertically. The input coordinates then should correspond to the corners of the outer boundary formed by the colour patches.

For the Munsell Book, the corners to be entered will be the boundaries of the page without changing its position.

To extract the 24 colour patches from the XRCCPP colour checker:

```
>>> colourchecker_name = "XRCCPP_24"
>>> corners = {"TopLeft":[3342.76,3680.96], "TopRight":[3345.91,2441.74],
    "BottomRight":[4162.87,2433.49], "BottomLeft":[4156.83,3669.2]}
>>> raw_image.extract_colourchecker_rgb_patches(checker_name=colourchecker_name,
corners_image=corners, size_rect=70)
```

Figure 66: *RawImage showing the colour checker patches (enhanced image)*

RawImage.get_ColourCheckerRGB(checker_name)

Method to get the RGB data from a colour patch of a ColourCheckerRGB extracted from the image.

Parameters:

checker_name: `str` *Colour checker name.*

Returns:

colourchecker_rgb: `ColourCheckerRGB` *Colour checker RGB data.*

To get a *ColourCheckerRGB* (it must have been extracted from the image beforehand):

```
>>> colourchecker = raw_image.get_ColourCheckerRGB("XRCCPP")
>>> type(colourchecker)
coolpi.image.colourchecker.ColourCheckerRGB
```

To get the *ColourCheckerRGB* attributes:

```
>>> print(colourchecker.type)
ColourChecker object
>>> print(colourchecker.subtype)
ColourChecker RGB data
>>> print(colourchecker.name)
XRCCPP
>>> print(colourchecker.illuminant)
MeasuredIlluminant object: Illuminant JNA.
>>> print(colourchecker.observer)
2º standard observer (CIE 1931)
>>> #print(XRCCPP_RGB.patches) # RGB data as dict
>>> print(colourchecker.patches_id)
dict_keys(['A1', 'A2', 'A3', 'A4', 'A5', 'A6', 'B1', 'B2', 'B3', 'B4', 'B5', 'B6',
'C1', 'C2', 'C3', 'C4', 'C5', 'C6', 'D1', 'D2', 'D3', 'D4', 'D5', 'D6'])
>>> print(colourchecker.metadata.keys()) # Measurement information as dict
dict_keys(['Camera', 'image_size', 'Date', 'ColorChecker', 'illuminant', 'observer'])
```

RawImage.get_patch_from_colourchecker(checker_name, patch_id)

Method to get the RGB data from a colour patch of a ColourCheckerRGB extracted from the image.

Parameters:

checker_name: `str` *Colour checker name.*

patch_id: `dict` *Patch ID.*

Returns:

r,g,b: `float` *RGB data.*

To get the RGB data of a given patch from a *ColourCheckerRGB*:

```
>>> raw_image.get_patch_from_colourchecker("XRCCPP", "D3")
(1242.0461224489795, 1497.0938775510203, 575.5077551020408)
```

In case of introducing a colour checker name or patch which is not in the image, a `ColourCheckerError` or a `PatchError` is raised.

```
>>>raw_image.get_patch_from_colourchecker("XRCCP", "P1")
ColourCheckerError: ColourChecker not present on image: dict_keys(['XRCCPP'])
>>>raw_image.get_patch_from_colourchecker("XRCCPP", "P1")
PatchError: Patch id not present in the ColourChecker XRCCPP
```

White Balance multipliers:

RawImage.get_camera_whitebalance()

Method to get the camera white balance multipliers from `raw_attributes`.

Returns:

wb_camera: `list` Camera wb multipliers.

To get the camera white balance multipliers from `raw_attributes`:

```
>>> wb_camera = raw_image.get_camera_whitebalance()
>>> print(wb_camera)
[2.05078125, 1.0, 1.3984375, 1.0]
```

RawImage.get_daylight_whitebalance()

Method to get the camera daylight white balance multipliers from `raw_attributes`.

Returns:

wb_daylight: `list` Daylight wb multipliers.

To get the camera daylight white balance multipliers from `raw_attributes`:

```
>>> wb_daylight = raw_image.get_daylight_whitebalance()
>>> print(wb_daylight)
[2.178001880645752, 0.9379902482032776, 1.199445366859436, 0.0]
```

*RawImage.compute_wb_multipliers(**params)*

Method to compute the RAW wb multipliers from a given patch.

To compute the wb multipliers from a patch of a ColourCheckerRGB extracted from image use:

`params` = dict(colourchecker_name=str, patch_id=str)

To compute the wb multipliers from r,g,b values use:

`params` = dict(patch_rgb=[r,g,b])

Parameters:

params: `dict` Parameters.

Returns:

wb_multipliers: `list` Computed wb multipliers.

To compute the wb multipliers using a grey/white patch data from a *ColourCheckerRGB*:

```
>>> params = dict(colourchecker_name ="XRCCPP", patch_id="D3")
>>> wb_multipliers = raw_image.compute_wb_multipliers(**params)
>>> print(wb_multipliers)
[1.205344834215299, 1.0, 2.6013444028839836, 1.0]
```

To compute the wb multipliers using the rgb patch data:

```
>>> F1 = raw_image.extract_rgb_patch_data_from_image([2469,3490], 70) # "F1"
>>> params = dict(patch_rgb=F1)
>>> wb_multipliers = raw_image.compute_wb_multipliers(**params)
>>> print(wb_multipliers)
[1.1863443615906974, 1.0, 2.6337750036390215, 1.0]
```

RawImage.estimate_wb_multipliers(method, **params)

Method to estimate the raw wb multipliers of an image.

Methods implemented: “wb_algorithm”, “illuminant”.

To estimate the wb multipliers using a wb algorithm use:

`params` = dict(algorithm=str, remove_colourckecker=bool, corners_colourchecker=dict)

wb algorithms implemented: “Average”, “GreyWorld”, “MaxWhite”, “Retinex”.

To estimate from the illuminant:

`params` = dict(use_transform_matrix=“embedded”)

The estimation method uses the “embedded” or the “computed” RGB to XYZ transform matrix.

Parameters:

method: `str` Method.

params: `dict` Parameters.

Returns:

wb_multipliers: `list` Estimated wb multipliers.

To estimate the wb multipliers of the image using a white balance algorithm:

```
>>> params = dict(algorithm="GreyWorld", remove_colourckecker=False,
    corners_colourchecker=None) # "Average", "GreyWorld", "MaxWhite", "Retinex"
>>> wb_multipliers = raw_image.estimate_wb_multipliers(method="wb_algorithm", **params)
>>> print(wb_multipliers)
[1.1817885097403635, 1.0, 2.613837972697569, 1.0]
```

It is possible to remove a *ColourCheckerRGB* from the image to estimate the wb multipliers as follows:

```
>>> params = dict(algorithm="GreyWorld", remove_colourckecker=True,
    corners_colourchecker= {"TopLeft":[1903,2120], "BottomRight":[4357,3940]})
>>> wb_multipliers = raw_image.estimate_wb_multipliers(method="wb_algorithm", **params)
>>> print(wb_multipliers)
[1.1851094404443938, 1.0, 2.6137520731738633, 1.0]
```

To estimate the wb multipliers of the image from the illuminant:

```
>>> wb_multipliers = raw_image.estimate_wb_multipliers(method="illuminant")
>>> print(wb_multipliers)
[1.2151604453345477, 1.0, 2.8910047176851696, 1.0]
```

If the *RawImage* object doesnot have an RGB to XYZ colour transformation matrix assigned, the camera embedded matrix is used for the estimation of the white balance multipliers.

To use a computed colour transformation matrix, it must be entered beforehand. Otherwise, a `ClassMethodError` is raised.

It is possible to set the computed RGB to XYZ matrix as follows:

```
>>> RGB_to_XYZ_WPP = np.array([[0.6953,0.0599,0.1632], [0.2817, 0.7431, -0.1077],
    [0.0915,-0.3027,1.1328]])
>>> raw_image.set_RGB_to_XYZ_matrix(RGB_to_XYZ_WPP)
```

Then, it is possible to compute the wb multipliers as follows:

```
>>> params = dict(use_transform_matrix="computed")
>>> wb_multipliers = raw_image.estimate_wb_multipliers(method="illuminant", **params)
>>> print(wb_multipliers)
[1.1809646428091218, 1.0, 2.6040767044506463, 1.0]
```

RawImage.set_whitebalance_multipliers(wb_multipliers)

Method to set the white balance multipliers of an image.

Options: “camera”, “daylight”, None ([1,1,1,1] is set) or custom = [r_gain, g_gain, b_gain, g_gain2]

Parameters:

wb_multipliers: `str, list` *wb multipliers.*

To set the white balance multipliers of an image:

```
>>> raw_image.set_whitebalance_multipliers(wb_multipliers =
    [1.205344834215299, 1.0, 2.6013444028839836, 1.0])
```

RawImage.get_whitebalance_multipliers()

Method to get the white balance multipliers of an image.

Returns:

wb_multipliers: `list` *wb multipliers.*

To get the white balance multipliers of an image:

```
>>> wb_multipliers = raw_image.get_whitebalance_multipliers()
>>> print(wb_multipliers)
[1.205344834215299, 1.0, 2.6013444028839836, 1.0]
```

White Balanced Image:

RawImage.apply_white_balance(show_image=True, method="OpenCV", save_image=False, output_path=None, bits=16)

Method to apply the wb multipliers to obtain the white balanced image.

Parameters:

show_image: `bool` *If True, the image is shown. Default:* `True`.

method: `str` *Method to show the image. Default: "OpenCV"*

save_image: `bool` *If True, the image is saved. Default:* `False`.

output_path: `os` *path for the ouput figure. Default:* `None`.

bits: `int` *Output image bits. Default: 16.*

To obtain the white balanced image:

```
>>> raw_image.apply_white_balance(show_image=True, method="matplotlib")
```

Figure 67: *RawImage White Balanced Image*

RawImage.get_raw_rgb_white_balanced_image()

Method to get the white balanced image.

Returns:

raw_rgb_wb: `numpy.ndarray` *White balanced image.*

To get the white balanced image:

```
>>> raw_rgg_wb = raw_image.get_raw_rgb_white_balanced_image()
>>> type(raw_rgg_wb)
numpy.ndarray
```

RAW Colour Image Processing:

RawImage.set_RGB_to_XYZ_matrix(RGB_to_XYZ)

Method to set the computed RGB to XYZ transform array.

Parameters:

RGB_to_XYZ: `list`, `numpy.ndarray` *Computed RGB to XYZ colour transform matrix.*

To set the computed RGB to XYZ colour transform matrix:

```
>>> RGB_to_XYZ_WPP = np.array([[0.6953,0.0599,0.1632], [0.2817, 0.7431, -0.1077],
    [0.0915,-0.3027,1.1328]])
>>> raw_image.set_RGB_to_XYZ_matrix(RGB_to_XYZ_WPP)
```

RawImage.apply_colour_correction(show_image=True, method="OpenCV", save_image=False, output_path=None, bits=16)

Method to obtain the colour-corrected image.

Parameters:

show_image: `bool` *If True, the image is shown. Default:* `True`.

method: `str` *Method to show the image. Default: "OpenCV".*

save_image: `bool` *If True, the image is saved. Default:* `False`.

output_path: `os` *path for the ouput figure. Default:* `None`.

bits: `int` *Output image bits. Default: 16.*

The following figure shows schematically the workflow implemented in coolpi to obtain colour-corrected images.

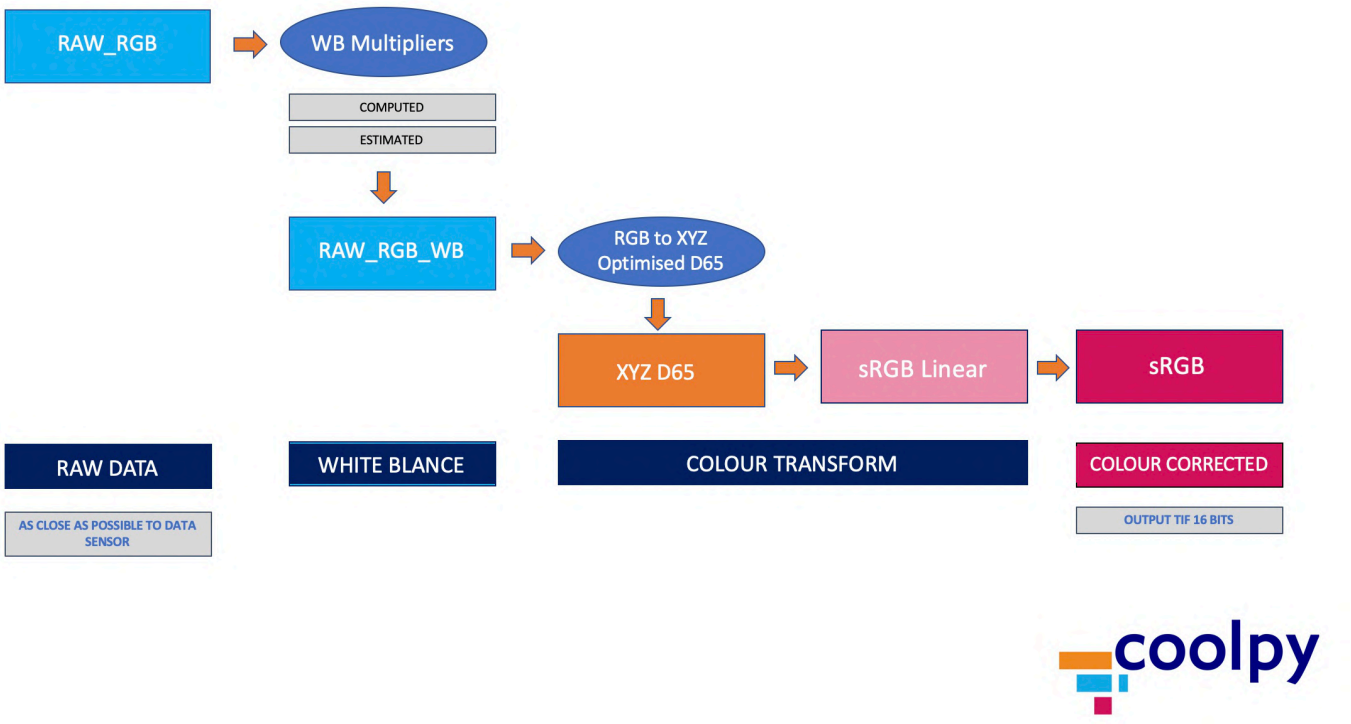

Figure 68: *RAW Colour-corrected Image Workflow*

To obtain the colour-corrected image, it is necessary to have entered the RGB to XYZ transformation matrix to be used. If not, the one provided by the camera manufacturer will be

used by default if it is available in the `raw_attributes` of the *RawImage*.

```
>>> RGB_to_XYZ_WPP = np.array([[0.6953,0.0599,0.1632], [0.2817, 0.7431, -0.1077],
    [0.0915,-0.3027,1.1328]])
>>> raw_image.set_RGB_to_XYZ_matrix(RGB_to_XYZ_WPP)
```

In addition, the wb multipliers must have been specified beforehand.

```
>>> raw_image.set_whitebalance_multipliers(wb_multipliers =
    [1.205344834215299, 1.0, 2.6013444028839836, 1.0])
```

Once the transformation matrix has been entered, the colour-corrected image can be obtained as follows:

```
>>> raw_image.apply_colour_correction(show_image=True, method="matplotlib")
```

Figure 69: *RawImage Colour-corrected Image*

⚠ **Alert**

White balance plays a crucial role for the proper correct-colour correction of digital images. Please read the previous sections on the calculation or estimation of the wb multipliers for a correct application of the white balance.

RawImage.get_colour_corrected_image()

Method to get the colour-corrected image.

Returns:

sRGB_non_linear: `numpy.ndarray` *sRGB corrected image.*

To get the colour-corrected image:

```
>>> sRGB_non_linear = raw_image.get_colour_corrected_image()
>>> type(sRGB_non_linear)
type(sRGB_non_linear)
```

RawImage.compute_image_colour_quality_assessment(checker_name, data=None)

Method to perform the quality assessment of the colour-corrected image obtained.

Parameters:

checker_name: `str` *Colour checker name.*

data: `ColourCheckerXYZ` *Reference CIE XYZ data. Default:* `None`.

Returns:

colourchecker_metrics: `DataFrame` *CIE XYZ residuals and colour-difference metrics.*

Once the colour-corrected image has been obtained, colorimetric analysis of the result can be performed. For the evaluation of the image it is required that a colour checker appears in the image. In this case, please extract first the RGB data from the image by the class method `RawImmage.extract_colourchecker_rgb_patches(checker_name, corners_image, size_rect)`. Thus, the transformation from RAW RGB to CIE XYZ (under CIE illuminant D65) can then be performed using the transformation matrix set to the *RawImage* object.

To compute the image colour quality assessment:

```
>>> colourchecker_metrics =
raw_image.compute_image_colour_quality_assessment(colourchecker_name="XRCCPP")
>>> #colourchecker_metrics.describe() # basic stats
>>> colourchecker_metrics["CIEDE2000"].mean()
4.500208139893018
```

The computed CIE XYZ data shall be compared with the CIE XYZ measured with colorimetric instruments as reference (loaded automatically from coolpi resources if available, or introduced by users as `data` parameter).

```
>>> from coolpi.image.image_objects import ColourCheckerSpectral
>>> XRCCPP = ColourCheckerSpectral("XRCCPP")
>>> colourchecker_metrics =
raw_image.compute_image_colour_quality_assessment(checker_name="XRCCPP",
    data = XRCCPP_XYZ)
>>> colourchecker_metrics["CIEDE2000"].mean()
4.500208139893018
```

`__str__`:

> **`__str__`**
>
> *Returns the string representation of the object. This method is called when the `print()` or `str()` function is invoked on an object.*

```
>>> print(raw_image) # str method
RawImage object from path: res/img/INDIGO_2022-06-19_NikonD5600_0017.NEF
```

8.5.4 Plot

> ***RawImage.plot_rgb_histogram(show_figure=True, save_figure=False, output_path=None, split_per_channel=False)***
>
> *Method to create and display the RGB Histogram using Matplotlib.*
>
> *Parameters:*
>
> *show_figure: `bool` If True, the plot is shown. Default: `True`.*
>
> *save_figure: `bool` If True, the figure is saved. Default: `False`.*
>
> *output_path: `os` path for the ouput figure. Default: `None`.*
>
> *split_per_channel: `bool` If True, show the histogram per channel. Default: `False`.*

To plot the RAW RGB Histogram:

```
>>> raw_image.plot_rgb_histogram()
```

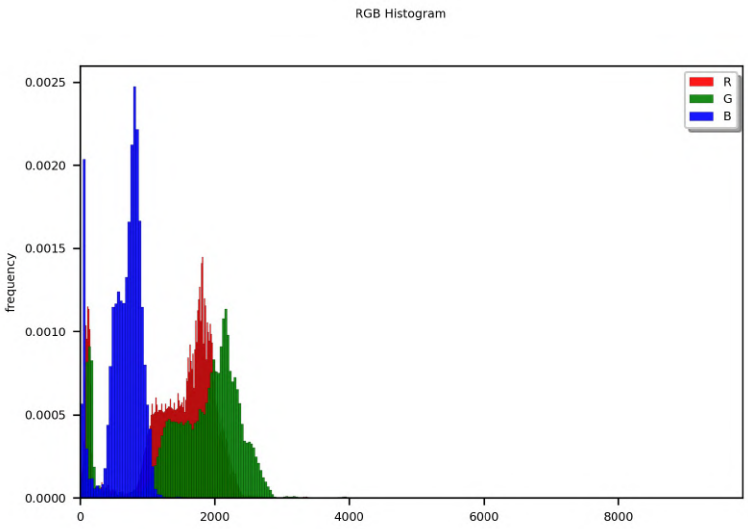

Figure 70: *RawImage Histogram*

Also, it is possible to plot the RAW RGB Histogram per channel as follows:

```
>>> raw_image.plot_rgb_histogram(split_per_channel=True)
```

In addition, it is posible to save the figure setting `True` the opcion `save_figure` and introducing a valid `output_path`:

```
>>> raw_image.plot_rgb_histogram(show_figure=False, save_figure=True,
output_path=path_hist, split_per_channel=True)
```

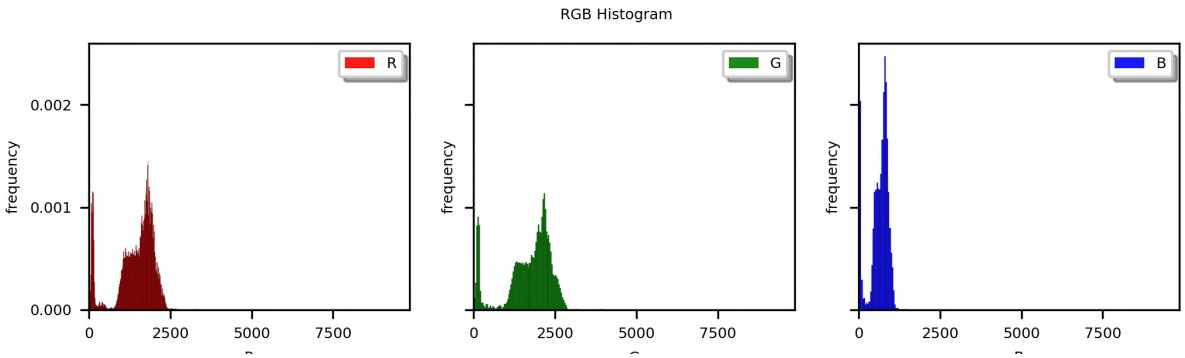

Figure 71: *RawImage Histogram per channel*

8.5.5 Show

RawImage.show_colourchecker(checker_name, parameters_draw=None, show_image=True, method="OpenCV", save_image=False, output_path=None, bits=16)

Method to show the patches of a ColourCheckerRGB extracted from the image using OpenCV tools.

Parameters:

checker_name: `str` *Colour checker name.*

parameters_draw: `dict` *Custom parameters to create the drawing. Default:* `None`.

show_image: `bool` *If True, the image is shown. Default:* `True`.

save_figure: `bool` *If True, the image is saved. Default:* `False`.

output_path: `os` *path for the ouput image. Default:* `None`.

bits: `int` *Output image bits. Default: 16.*

Once a *ColourCheckerRGB* object has been extracted from the *RawImage*, it is possible to draw the extracted patches on a new image as a visual aid to check the process.

To show the *ColourcheckerRGB*:

```
>>> raw_image.show_colourchecker("XRCCPP", show_image=True, method="matplotlib")
```

In addition, it is posible to save the figure setting `True` the opcion `save_figure` and introducing a valid `output_path`:

```
>>> raw_image.show_colourchecker("XRCCPP", show_image=False, save_image=True,
output_path=path_out, bits=16)
```

Figure 72: *RawImage showing the colour checker patches (enhanced image)*

RawImage.show(data = “raw”, method=“OpenCV”)

Method to display the RAW RGB image (“raw”), the white balanced image (“wb”) or the final sRGB image (“sRGB”).

Parameters:

data: `str` *Data to show. Default: “raw”.*

method: `str` *Method to display the image. Default: “OpenCV”.*

To display the RAW RGB image (“raw”), the white balanced image (“wb”) or the final sRGB image (“sRGB”), simply enter the required parameters as follows:

```
>>> raw_image.show(data="raw", method="matplotlib")
```

8.5.6 Save

RawImage.save(output_path, data = “raw”, bits=16)

Method to save the RAW RGB image (“raw”), the white balanced image (“wb”) or the final sRGB image (“sRGB”).

Parameters:

output_path: `os` *path for the ouput image. Default:* `None`.

data: `str` *Data to show. Default: “raw”.*

bits: `int` *Output image bits. Default: 16.*

To save the RAW RGB image (“raw”), the white balanced image (“wb”) or the final sRGB image (“sRGB”), simply enter the required parameters as follows:

```
>>> raw_image.save(output_path=path_out, data = "raw", bits=16)
```

8.6 ProcessedImage

class ProcessedImage

ProcessedImage(path_img=None, rgb_data=None, metadata={})

.set_metadata(metadata)

.set_image_illuminant(illuminant)

.set_observer(observer)

.extract_rgb_patch_data_from_image(center, size)

.extract_colourchecker_rgb_patches(checker_name, corners_image, size_rect)

.get_ColourCheckerRGB(checker_name)

.get_patch_from_colourchecker(checker_name, patch_id)

.plot_rgb_histogram(show_figure, save_figure, output_path, split_per_channel)

.show(method)

.show_colourchecker(colourchecker_name, show_figure, save_figure,

output_path, method, parameters_draw, bits)

.save(output_path, bits)

The *ProcessedImage* class represents the RGB data extracted from an non-raw image.

8.6.1 Create an instance

ProcessedImage(path_img=None, rgb_data=None, metadata={})

Parameters:

path_img: `os` *Colour checker name or description.*

rgb_data: `numpy.ndarray` *RGB data, e.g. from RawImage.*

metadata: `dict` *Image information as metadata. Default: {}.*

To *create an instance* of the *ProcessedImage* class, simply enter the required parameters as follows:

```
>>> metadata = {"Camera": "Nikon D5600", "Date": [[2022, 6, 19], [8, 29, 00]],
    "ColorChecker": "XRCCPP", "illuminant":JNA, "observer": 2}
>>> processed_image = ProcessedImage(path_img = path_tif, rgb_data=None,
    metadata=metadata)
```

Figure 73: *ProcessedImage*

The *ProcessedImage* metadata can be set separately with the method `.set_metadata(metadata)`. If the illuminant under which the image has been taken is known (e.g. the spectral power distribution of the illuminant has been measured using specific instrumentation), it is recommended to introduce the illuminant and the CIE observer (default 2) as keys in the metadata for colorimetric computations. However, the illuminant and observer can be set separately with the methods `.set_image_illuminant(illuminant)` and `.set_observer(observer)`.

8.6.2 Attributes

The class *attributes* are: *type* (returns a `str` with the description of the main class), *subtype* (returns a `str` with the description of the object), *path* (returns an `os` with the image path), *rgb_data* (returns a `numpy.ndarray` with the RAW RGB demosaiced data), *metadata* (returns a `dict` with the details on image acquisition), *illuminant* (returns the *Illuminant*, *IlluminantFromCCT* or *MeasuredIlluminant* instance), and *observer* (returns the *Observer* instance)

```
>>> print(processed_image.type)
Image object
>>> print(processed_image.subtype)
Processed Image object
>>> print(processed_image.path)
res/img/custom_image_processing_wb_16bits.tif
>>> print(processed_image.rgb_data.shape)
(4016, 6016, 3)
>>> print(processed_image.metadata)
{'Camera': 'Nikon D5600', 'Date': [[2022, 6, 19], [8, 29, 0]],
'ColorChecker': 'XRCCPP', 'illuminant':JNA, 'observer': 2}
>>> print(processed_image.illuminant)
MeasuredIlluminant object: Illuminant JNA.
>>> print(processed_image.observer)
2º standard observer (CIE 1931)
```

8.6.3 Methods

ProcessedImage.extract_rgb_patch_data_from_image(center, size)

Method to extract the RGB data of a patch from an image.

Parameters:

center: `list` *Center coordinates of the patch as [x=col, y=row].*

size: `int` *Patch size in px.*

Returns:

r,g,b: `float` *RGB average data.*

To extract the RGB data of a single patch from the image:

```
>>> r_grey, g_grey, b_grey = processed_image.extract_rgb_patch_data_from_image([2469,3490],
70) # "F1"
>>> print(r_grey, g_grey, b_grey)
57463.82326530612 56555.61612244898 55993.384081632656 # 16 bits
```

ProcessedImage.extract_colourchecker_rgb_patches(checker_name, corners_image, size_rect=40)

Method to extract the RGB data of a colour checker from an image.

Parameters:

checker_name: `str` *Colour checker name.*

corners_image: `dict` *Colour checker corner coordinates as {"TopLeft":[x,y], "TopRight":[x,y], "BottomRight":[x,y], "BottomLeft":[x,y]}.*

size_rect: `int` *Size in px. Default: 40.*

The following colour checkers have been implemented:

- Calibrite PASSPORT PHOTO 2: "CCPP2_24", "CCPP2_26".
- Calibrite PASSPORT VIDEO: "CCPPV_24", "CCPPV_3".
- Munsell Soil Color Book (Ed.1994): Colour checker name and page, e.g. "MSCB1994_GLEY1", "MSCB1994_5Y".
- Munsell Soil Color Book (Ed.2009): Colour checker name and page, e.g. "MSCB2009_GLEY1", "MSCB2009_GLEY2".
- Spyder Checker: "SCK100_48", "SCK100_7".
- X-rite PASSPORT PHOTO: "XRCCPP_24", "XRCCPP_26".

Some of the colour checkers available in coolpi have been separated into smaller sections (either because of their shape, or because they are in single sheets like the Munsell Book).

The corner coordinates shall refer to the markings or crosses if present. In case the colour checker does not have any reference mark, the colour checker should be positioned horizontally so that there are more patches horizontally than vertically. The input coordinates then should correspond to the corners of the outer boundary formed by the colour patches.

For the Munsell Book, the corners to be entered will be the boundaries of the page without changing its position.

To extract the 24 colour patches from the XRCCPP colour checker:

```
>>> colourchecker_name = "XRCCPP_24"
>>> corners = {"TopLeft":[3342.76,3680.96], "TopRight":[3345.91,2441.74],
    "BottomRight":[4162.87,2433.49], "BottomLeft":[4156.83,3669.2]}
>>> raw_image.extract_colourchecker_rgb_patches(checker_name=colourchecker_name,
corners_image=corners, size_rect=70)
```

Figure 74: *ProcessedImage showing the colour checker patches*

ProcessedImage.get_ColourCheckerRGB(checker_name)

Method to get the RGB data from a colour patch of a ColourCheckerRGB extracted from the image.

Parameters:

checker_name: `str` *Colour checker name.*

Returns:

colourchecker_rgb: `ColourCheckerRGB` *Colour checker RGB data.*

To get a *ColourCheckerRGB* (it must have been extracted from the image beforehand):

```
>>> colourchecker = processed_image.get_ColourCheckerRGB("XRCCPP")
>>> type(colourchecker)
coolpi.image.colourchecker.ColourCheckerRGB
```

To get the *ColourCheckerRGB* attributes:

```
>>> print(colourchecker.type)
ColourChecker object
>>> print(colourchecker.subtype)
ColourChecker RGB data
>>> print(colourchecker.name)
XRCCPP
>>> print(colourchecker.illuminant)
MeasuredIlluminant object: Illuminant JNA.
>>> print(colourchecker.observer)
2º standard observer (CIE 1931)
>>> #print(XRCCPP_RGB.patches) # RGB data as dict
>>> print(colourchecker.patches_id)
dict_keys(['A1', 'A2', 'A3', 'A4', 'A5', 'A6', 'B1', 'B2', 'B3', 'B4', 'B5', 'B6',
'C1', 'C2', 'C3', 'C4', 'C5', 'C6', 'D1', 'D2', 'D3', 'D4', 'D5', 'D6'])
>>> print(colourchecker.metadata.keys()) # Measurement information as dict
dict_keys(['Camera', 'Date', 'ColorChecker', 'illuminant', 'observer'])
```

> ***ProcessedImage.get_patch_from_colourchecker(checker_name, patch_id)***
>
> *Method to get the RGB data from a colour patch of a ColourCheckerRGB extracted from the image.*
>
> *Parameters:*
>
> *checker_name:* `str` *Colour checker name.*
>
> *patch_id:* `dict` *Patch ID.*
>
> *Returns:*
>
> *r,g,b:* `float` *RGB data.*

To get the RGB data of a given patch from a *ColourCheckerRGB*:

```
>>> processed_image.get_patch_from_colourchecker("XRCCPP", "D3")
(47612.109591836735, 47455.98775510204, 47447.257755102044) # 16 bits
```

In case of introducing a colour checker name or patch which is not in the image, a `ColourCheckerError` or a `PatchError` is raised.

```
>>>processed_image.get_patch_from_colourchecker("XRCCP", "P1")
ColourCheckerError: ColourChecker not present on image: dict_keys(['XRCCPP'])
>>>processed_image.get_patch_from_colourchecker("XRCCPP", "P1")
PatchError: Patch id not present in the ColourChecker XRCCPP
```

`__str__`:

> **`__str__`**

Returns the string representation of the object. This method is called when the `print()` or `str()` function is invoked on an object.

```
>>> print(processed_image) # str method
ProcessedImage object from path: res/img/custom_image_processing_wb_16bits.tif
```

8.6.4 Plot

ProcessedImage.plot_rgb_histogram(show_figure=True, save_figure=False, output_path=None, split_per_channel=False)

Method to create and display the RGB Histogram using Matplotlib.

Parameters:

show_figure: `bool` *If True, the plot is shown. Default:* `True`.

save_figure: `bool` *If True, the figure is saved. Default:* `False`.

output_path: `os` *path for the ouput figure. Default:* `None`.

split_per_channel: `bool` *If True, show the histogram per channel. Default:* `False`.

To plot the RGB Histogram:

```
>>> processed_image.plot_rgb_histogram()
```

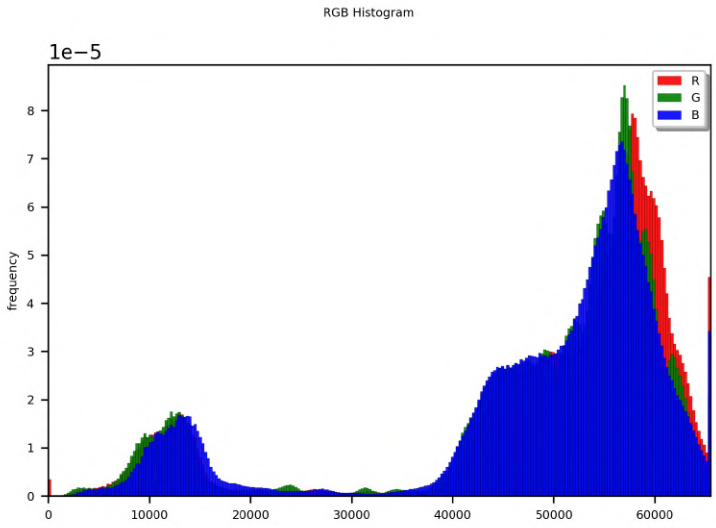

Figure 75: *ProcessedImage Histogram*

Also, it is possible to plot the RAW RGB Histogram per channel as follows:

```
>>> processed_image.plot_rgb_histogram(split_per_channel=True)
```

In addition, it is posible to save the figure setting `True` the opcion `save_figure` and introducing a valid `output_path`:

```
>>> processed_image.plot_rgb_histogram(show_figure=False, save_figure=True,
output_path=path_hist, split_per_channel=True)
```

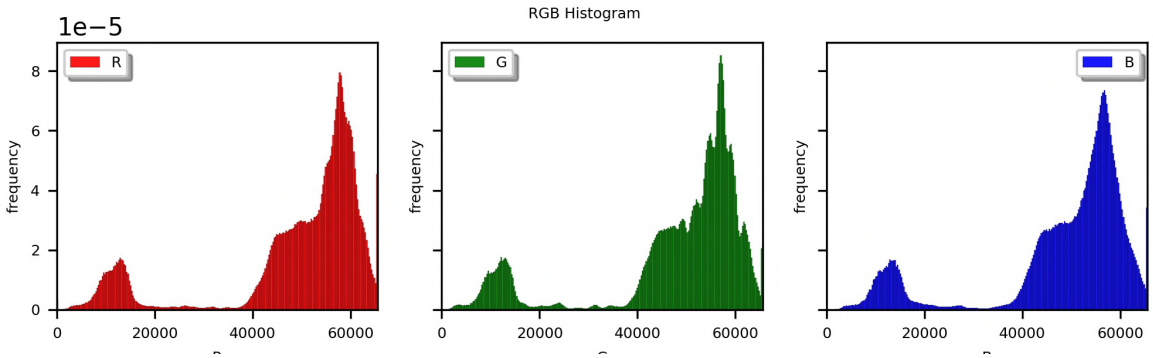

Figure 76: *ProcessedImage Histogram per channel*

8.6.5 Show

ProcessedImage.show_colourchecker(checker_name, parameters_draw=None, show_image = True, method =“OpenCV”, save_image = False, output_path = None, bits=16)

Method to show the patches of a ColourCheckerRGB extracted from the image using OpenCV tools.

Parameters:

checker_name: `str` *Colour checker name.*

parameters_draw: `dict` *Custom parameters to create the drawing. Default:* `None`.

show_image: `bool` *If True, the image is shown. Default:* `True`.

save_figure: `bool` *If True, the image is saved. Default:* `False`.

output_path: `os` *path for the ouput image. Default:* `None`.

bits: `int` *Output image bits. Default: 16.*

Once a *ColourCheckerRGB* object has been extracted from the *ProcessedImage*, it is

possible to draw the extracted patches on a new image as a visual aid to check the process.

To show the *ColourcheckerRGB*:

```
>>> raw_image.show_colourchecker("XRCCPP", show_image=True, method="matplotlib")
```

In addition, it is posible to save the figure setting `True` the opcion `save_figure` and introducing a valid `output_path`:

```
>>> raw_image.show_colourchecker("XRCCPP", show_image=False, save_image=True,
output_path=path_out, bits=16)
```

Figure 77: *ProcessedImage showing the colour checker patches*

ProcessedImage.show(method="OpenCV")

Method to display the RGB image data.

Parameters:

method: `str` *Method to display the image. Default: "OpenCV".*

To display the RGB image data, simply enter the required parameters as follows:

```
>>> processed_image.show(method="matplotlib")
```

8.6.6 Save

RawImage.save(output_path, bits=16)

Method to save the RGB image data.

Parameters:

output_path: `os` *path for the ouput image. Default:* `None`*.*

bits: `int` *Output image bits. Default: 16.*

To save the RGB image data, simply enter the required parameters as follows:

```
>>> processed_image.save(output_path=path_out, bits=16)
```

9 Notebooks

A series of interactive Jupyter Notebooks have been prepared. They include practical examples to help users become familiar with the classes, methods and functions implemented in the coolpi package.

- 01 CIE objects
- 02a Colour objects
- 02b CSC - Colour Space Conversion
- 02c CSC - Data test (Ohta&Robertson 2005)
- 03a Spectral objects
- 03b CTT Calculations Test data (Fontecha et al. 2002)
- 04a Colour-difference
- 04b CIEDE2000 - Test data (Sharma et al., 2005)
- 05 ColourChecker objects
- 06 Image objects

Info

Users can find the interative Jupyter Notebooks in the notebook folder of the coolpi repository on GitHub.

Alert

In order to use the iterative notebooks, JupyterLab, or its extension in the code editor used, must be installed beforehand.

10 GUI

A graphical user interface has been designed together with the coolpi package. The aim is to help especially non-programmers to use in an easy and practical way the functionalities implemented in the coolpi library. Efforts have been made to develop the graphical interface in a way that makes it intuitive and friendly to use.

10.1 Run

To run the coolpi-gui, simply type the following command lines:

```
>>> from coolpi.gui.app import GUI
>>> gui = GUI()
>>> gui.run()
```

10.2 Home Screen

The coolpi-gui includes the following tools:

- CSC: Colour Space Conversion
- CDE: Colour ΔE
- CPT: Colour Plot Tool
- SPC: Spectral Colour
- SPD: Illuminant SPD
- CCI: ColourChecker Inspector
- RCIP: RAW Colour Image Processing

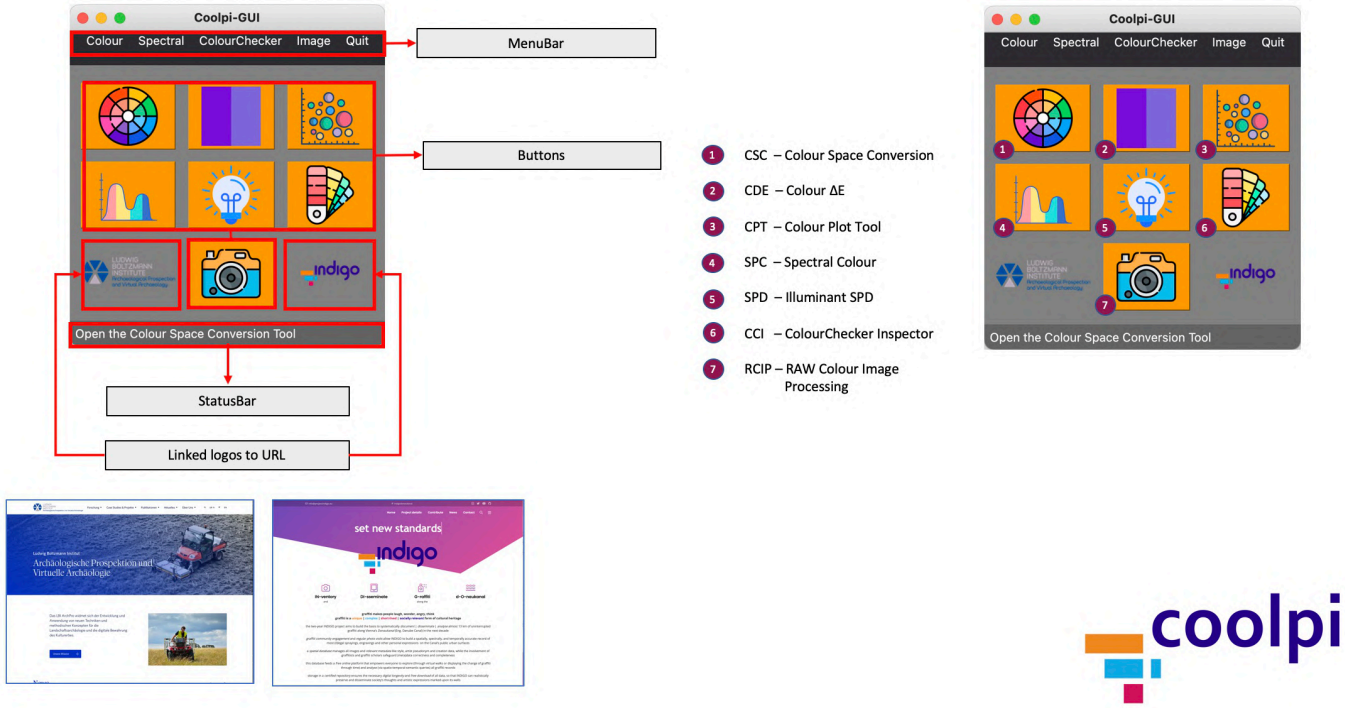

Figure 78: *Home screen*

10.3 CSC: Colour Space Conversion

Tool to calculate the conversion between CIE colour spaces.

Colour Sample: Select the illuminant, the observer and the CIE colour space. Insert the

`Sample ID` or description of the colour sample and its three coordinates.

Calculate: Click the *Calculate* button to perform the conversion between the CIE colour spaces. The coordinates referred to the output color spaces will be automatically updated on the screen.

Export: Click the *Export* button to save the computed colour sample coordinates in a JSON exchange file. This button is active once the conversion is performed.

Clear: Click the *Clear* button to delete the data and start a new colour space conversion with a different colour sample.

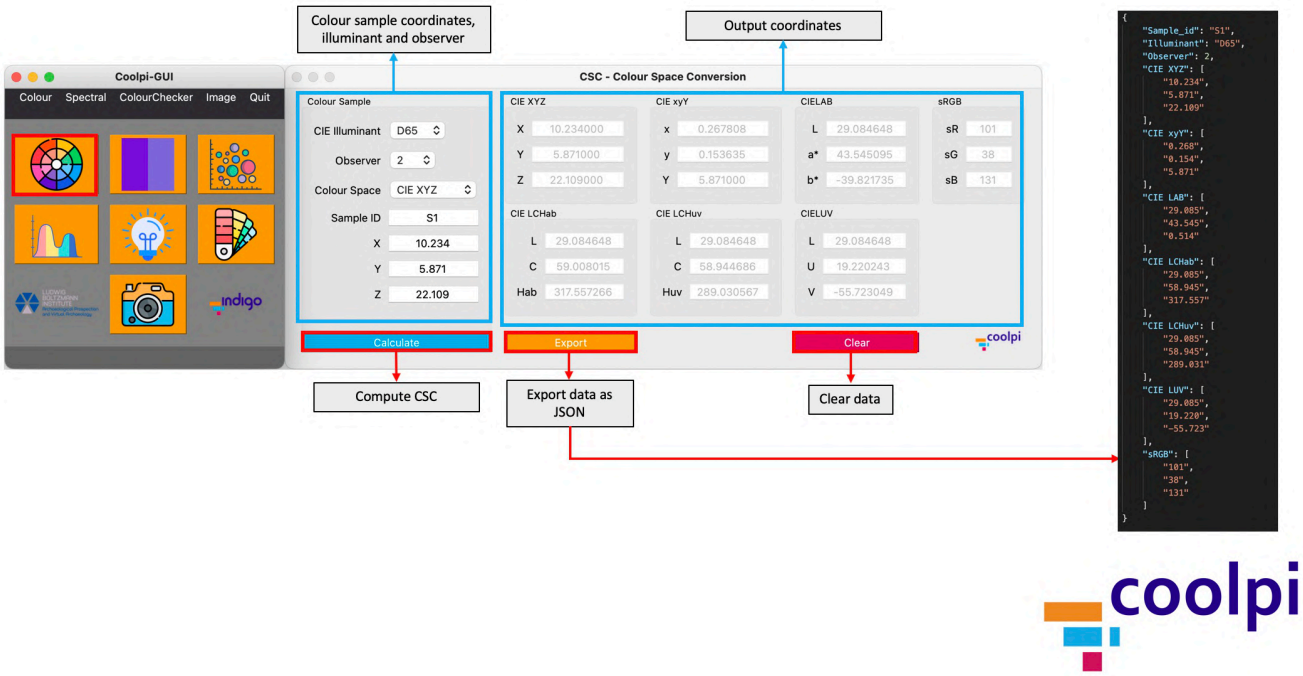

Figure 79: *CSC - Colour Space Conversion*

10.4 CDE: Colour ΔE

Tool to obtain the colour difference between two colour samples.

Colour Sample A: Select the illuminant, the observer and the CIE colour space. Insert the `Sample ID` or description of the colour sample A and its three coordinates.

Colour Sample B: Select the illuminant, the observer and the CIE colour space. Insert the `Sample ID` or description of the colour sample B and its three coordinates.

Calculate: Click the *Calculate* button to obtain the colour difference between the input samples A and B. Before calculating the colour difference between the samples, the

conversion to CIELAB D65 shall be performed in case the coordinates entered are referred to a different colour space and illuminant. The CIELAB D65 coordinates of both samples as well as the obtained ΔE_{ab} and CIEDE2000 (in CIELAB units) will be displayed automatically on the screen. As a visual aid, a CIELAB Diagram with samples A and B will be plotted on the screen. In addition, the sRGB for each sample (under the D65 illuminant) are shown on the screen to help users identify the samples easily.

Zoom: Click the *Zoom* button to display the `CIELAB Diagram` of the colour samples in a new window. This button is active once the colour difference is performed.

Save: Click the *Save* button to export the `CIELAB Diagram` as an image. This button is active once the colour difference is performed.

Clear: Click the *Clear* button to delete the data and start a new colour difference computation with two different colour samples.

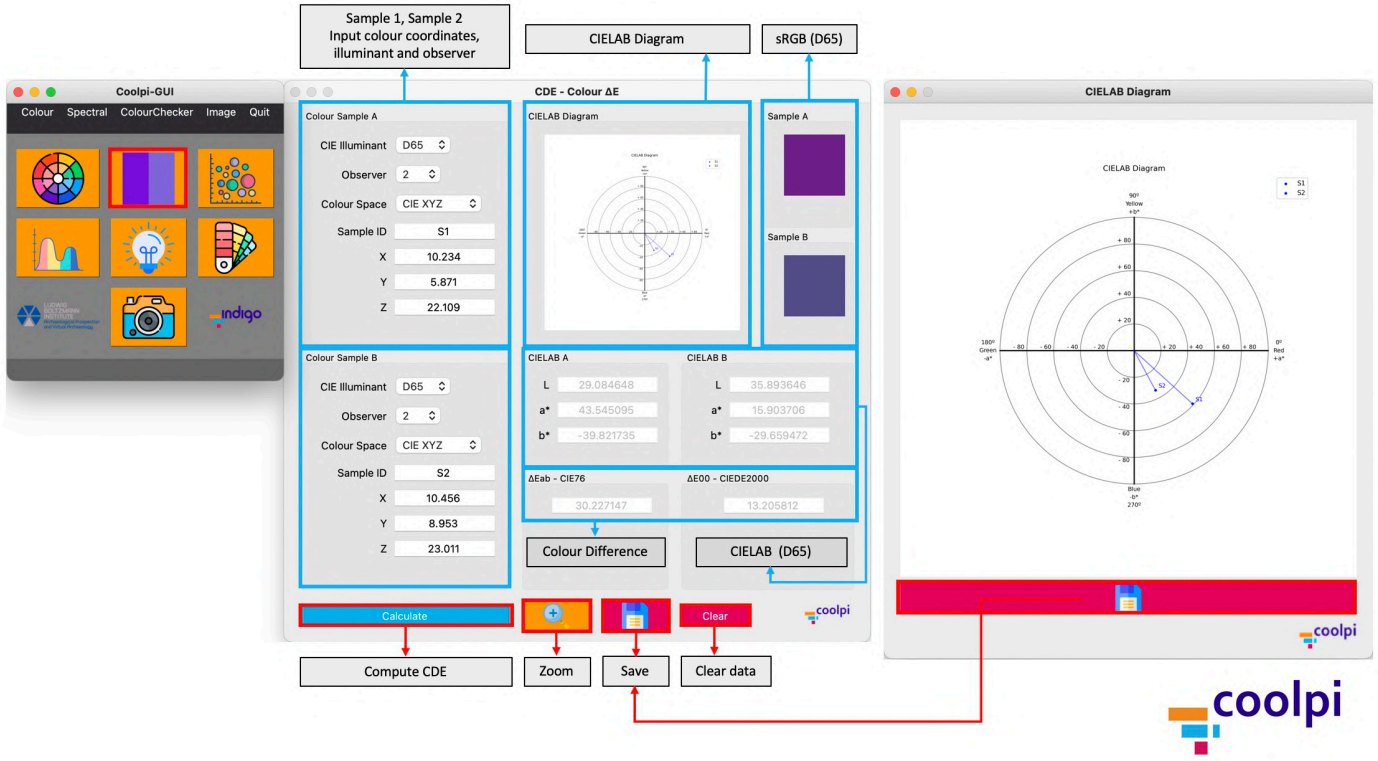

Figure 80: *CDE - Colour* ΔE

10.5 CPT: Colour Plot Tool

Tool to display the CIE 1931 x,y Chromaticity Diagram, the CIELAB Diagram and the sRGB plot of a given colour sample.

Colour Sample: Select the illuminant, the observer and the CIE colour space. Insert the `Sample ID` or description of the colour sample and its three coordinates.

Plot: Click the *Plot* button to show the `CIE 1931 x,y Chromaticity Diagram`, the `CIELAB Diagram` and the `sRGB Plot` of the colour sample.

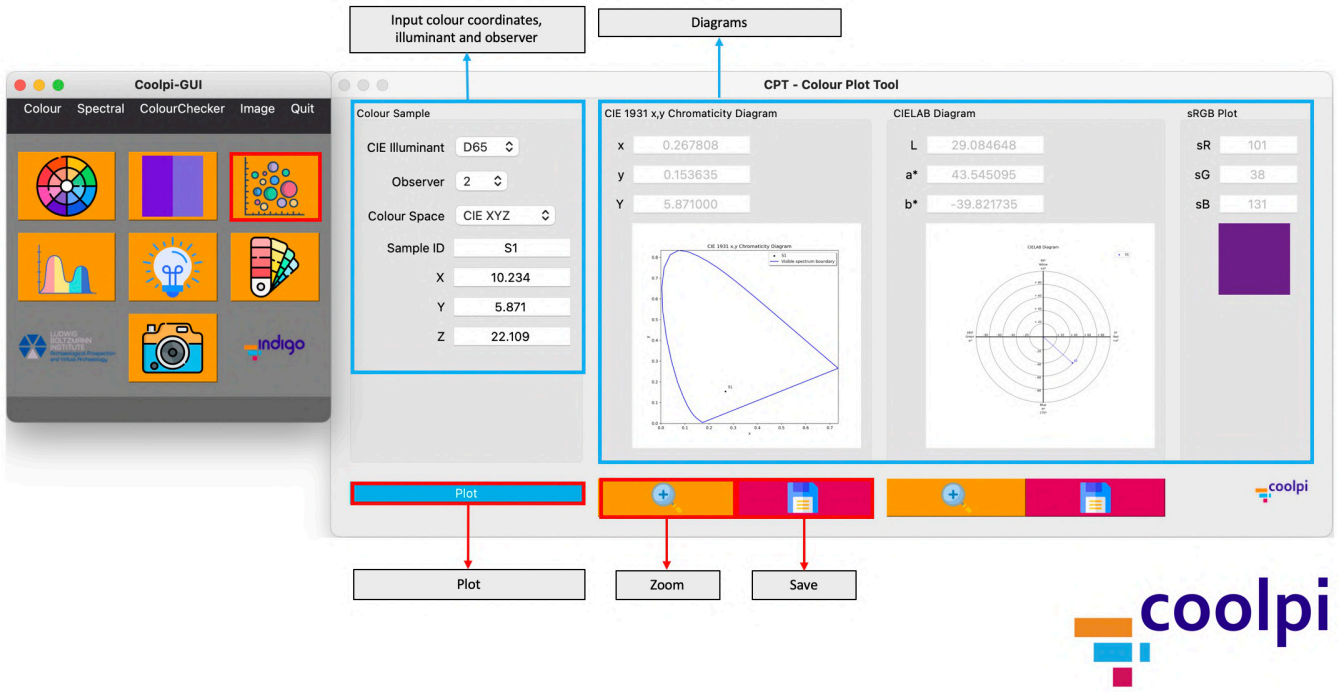

Figure 81: *CPT - Colour Plot Tool*

Zoom: Click the *Zoom* button to display the `CIE 1931 x,y Chromaticity Diagram` or the `CIELAB Diagram` of the colour sample in a new window. This button is active once the *Plot* button has been pressed.

Save: Click the *Save* button to export the `CIE 1931 x,y Chromaticity Diagram` or the `CIELAB Diagram` as an image. This button is active once the *Plot* button has been pressed.

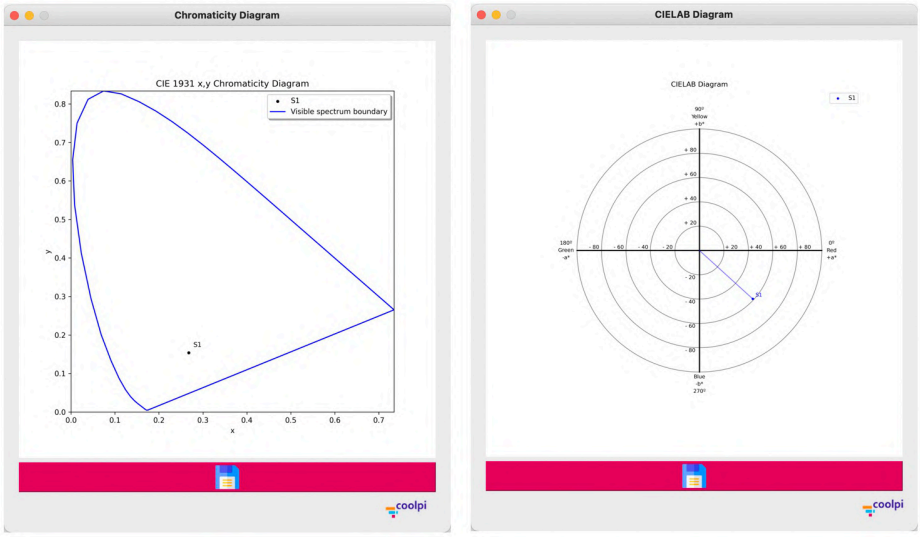

Figure 82: *Zoom: CIE 1931 x,y Chromaticity Diagram and CIELAB Diagram*

10.6 SPC: Spectral Colour

Tool for dealing with spectral colour samples.

Load from JSON: Click the *Load from JSON* button to load the spectral data of the colour sample. The JSON file must contain the following keys: `Sample_id` (`str` with the name or description), `Illuminant` (CIE illuminant name as `str`), `Observer` (CIE observer as `str` or `int`), `nm_range` (`list` with the min and max range value in nm), `nm_interval` (lambda interval in nm as `int`), and `lambda_values` (spectral colour data as `list`).

```
{
"Sample_id": "CCC-C3",
"Illuminant": "D65",
"Observer": 2,
"nm_range": [360, 740],
"nm_interval": 10,
"lambda_values": [4.9650,4.9650,4.9650,4.9650,4.9650,4.8950,4.8500,4.8425,4.8725,5.0000,5.3000,5.8025,6.1825,6.1625,
5.9450,5.7775,5.7225,5.7125,5.5925,5.4025,5.3850,5.9075,7.8525,13.5475,24.6675,38.6225,48.9025,54.4750,58.2550,61.1775,
63.5725,65.1550,66.0950,66.7600,67.2575,67.2575,67.2575,67.2575,67.2575]
}
```

Figure 83: *JSON file format: SPC data*

Once the spectral data has been loaded, the computed CIE XYZ, CIE xy and CIELAB coordinates will be displayed automatically on the screen. In addition, the plot with the spectral data together with the sRGB data (under D65) will be displayed on the screen. The sRGB colour of the input sample is displayed to help users identify the spectral colour.

Zoom: Click the *Zoom* button to display the `Spectral Colour Plot` figure with the spectral data of the colour sample in a new window. This button is active once the spectral data of the colour sample is loaded from the JSON file.

Save: Click the *Save* button to export the `Spectral Colour Plot` of the spectral data for the colour sample as an image. This button is active once the spectral data of the colour sample is loaded from the JSON file.

Clear: Click the *Clear* button to delete the data and start with a new spectral data colour sample.

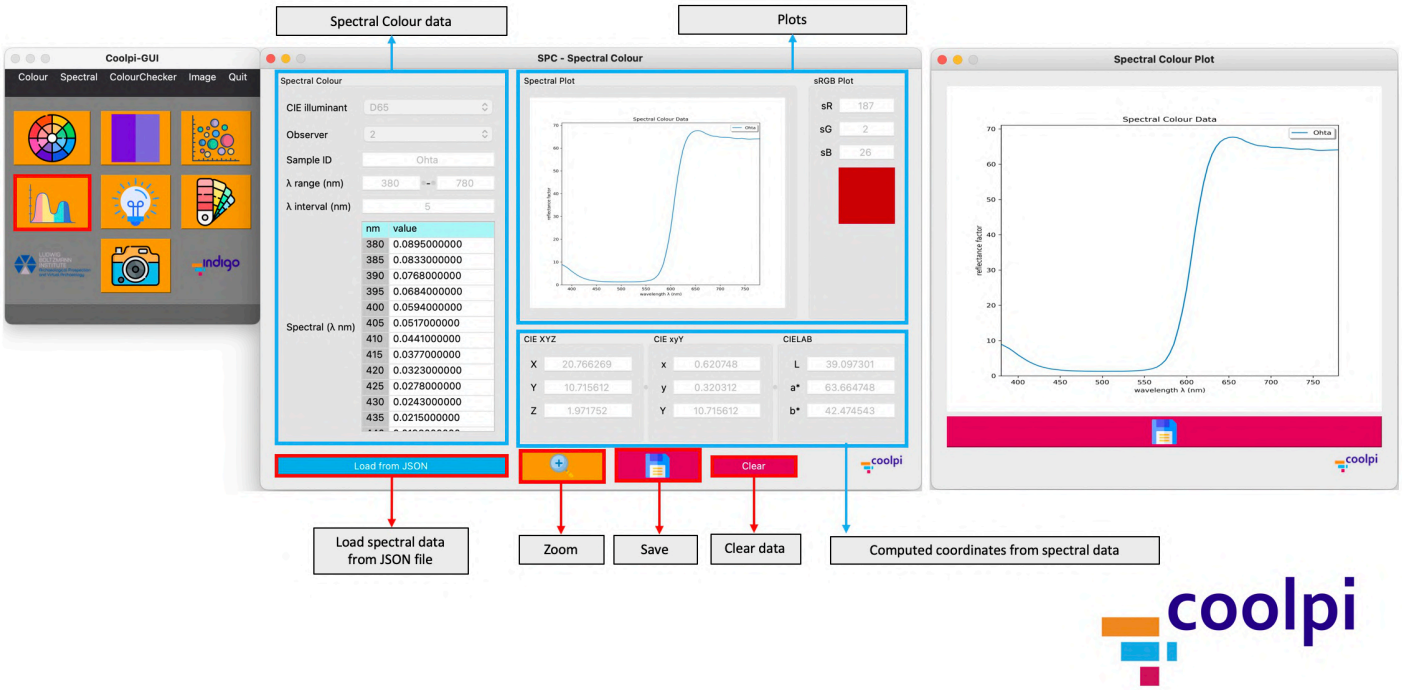

Figure 84: *SPC - Spectral Colour*

10.7 SPD: Illuminant SPD

Tool for dealing with CIE illuminants, illuminant whose SPD is computed from a valid correlated colour temperature and measured illuminants.

Input Illuminant: Select the type of illuminant: CIE, CCT (SPD computed from CCT) or Measured.

- For CIE illuminants, select the name of the desired illuminant in the `CIE illuminant` combobox.

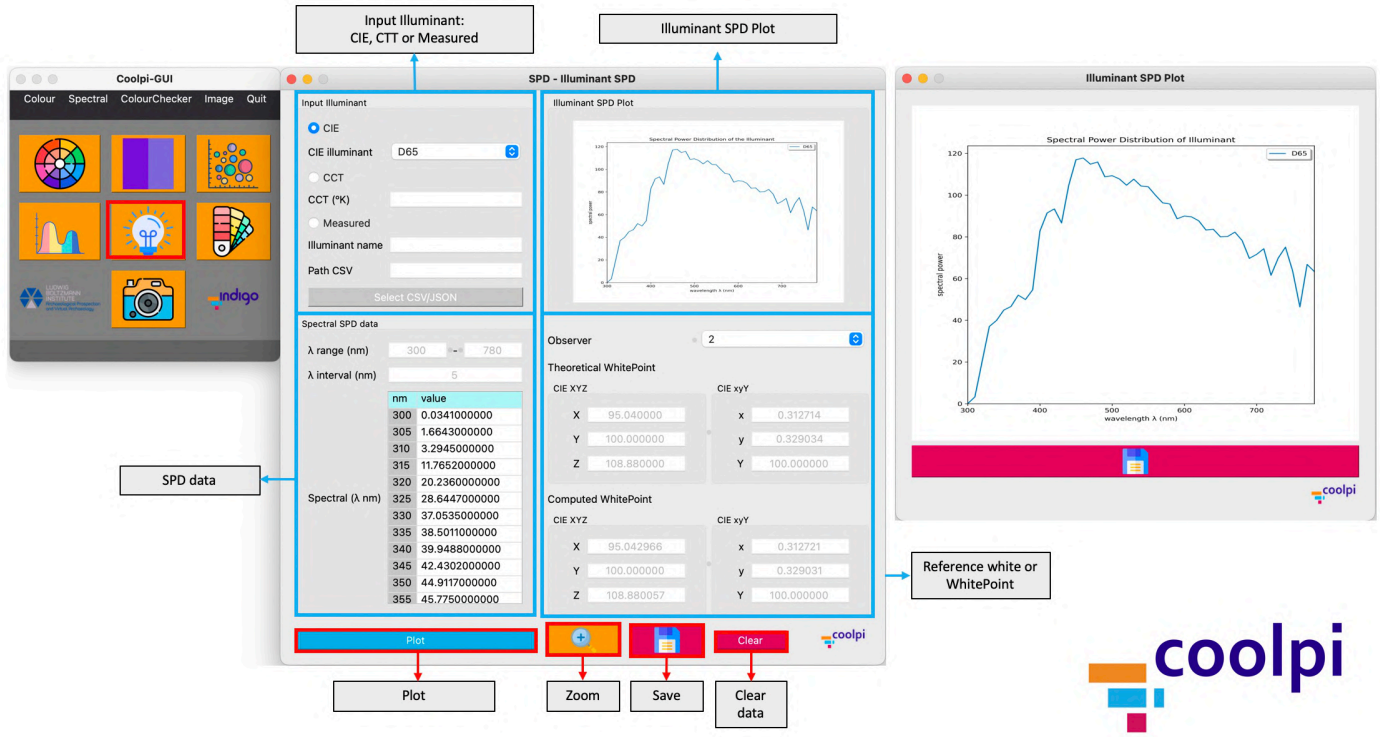

Figure 85: *CIE Illuminant*

- For CCT illuminants, simply enter the CCT (in Kelvin degrees) in the `CCT K` box. The CCT should be into the range 4000-25000 (ºK).

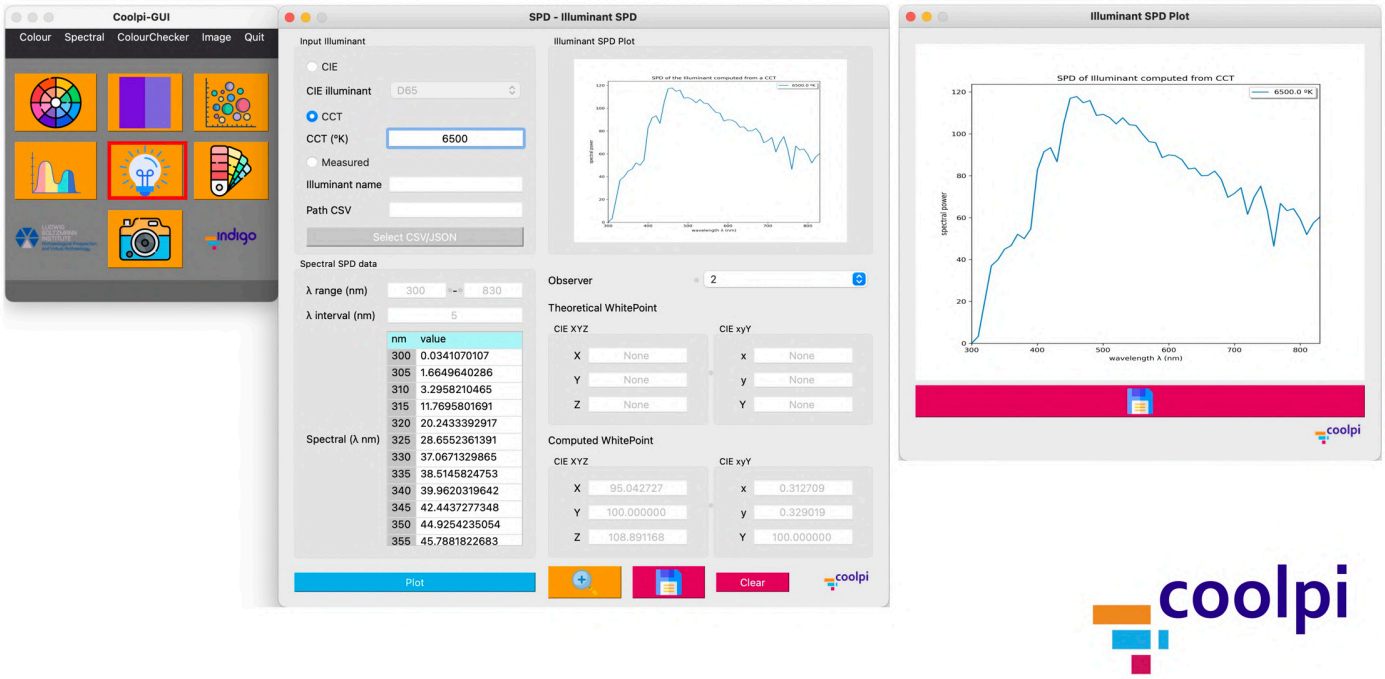

Figure 86: *Illuminant computed from CCT*

- For Measured illuminants, set the illuminant name in the `Illuminant name` box. Then click on the `Select CSV/JSON` button. Choose the file with the illuminant data. For measurements carried out with Sekonic instruments select the CSV file provided by the

instrument. For other instruments (or in case the CSV file fails), it is possible to load the SPD data from a generic JSON file. The JSON file must contain the following keys: `nm_range` (`list` with the min and max range value in *nm*), `nm_interval` (lambda interval in *nm* as `int`), and `lambda_values` (spectral power distribution data as `list`).

```
{
"nm_range": [380, 780],
"nm_interval" :5,
"lambda_values": [0.0114073, 0.00985631, 0.0072386, 0.00728911, 0.01057201, 0.01290828,
    0.01089283, 0.00820896, 0.00928692, 0.01575225, 0.03204091, 0.04716717, 0.04427407,
    0.03362948, 0.03129305, 0.0341507, 0.03702938, 0.0393975, 0.04113271, 0.0425808,
    0.04451207, 0.04628982, 0.04587397, 0.04293254, 0.03907673, 0.03530976, 0.03190796,
    0.02873687, 0.02581335, 0.02336659, 0.02249494, 0.02874496, 0.04532121, 0.05484243,
    0.04516013, 0.02744621, 0.0213983, 0.02104433, 0.02324293, 0.02704061, 0.02970082,
    0.02967269, 0.02855473, 0.02819227, 0.02867136, 0.03075241, 0.03364546, 0.03476707,
    0.03369897, 0.03211911, 0.03017166, 0.02761547, 0.02518627, 0.02336014, 0.02189655,
    0.02034273, 0.01835747, 0.01599688, 0.01375351, 0.01198202, 0.01058324, 0.0093942,
    0.00819748, 0.00698151, 0.00614299, 0.00563017, 0.00501737, 0.00425217, 0.00352747,
    0.00306196, 0.00279519, 0.00244556, 0.00205841, 0.00189265, 0.00200176, 0.00212645,
    0.00203685, 0.00170677, 0.00143012, 0.00107608, 0.00085404]
}
```

Figure 87: *JSON file format: SPD data*

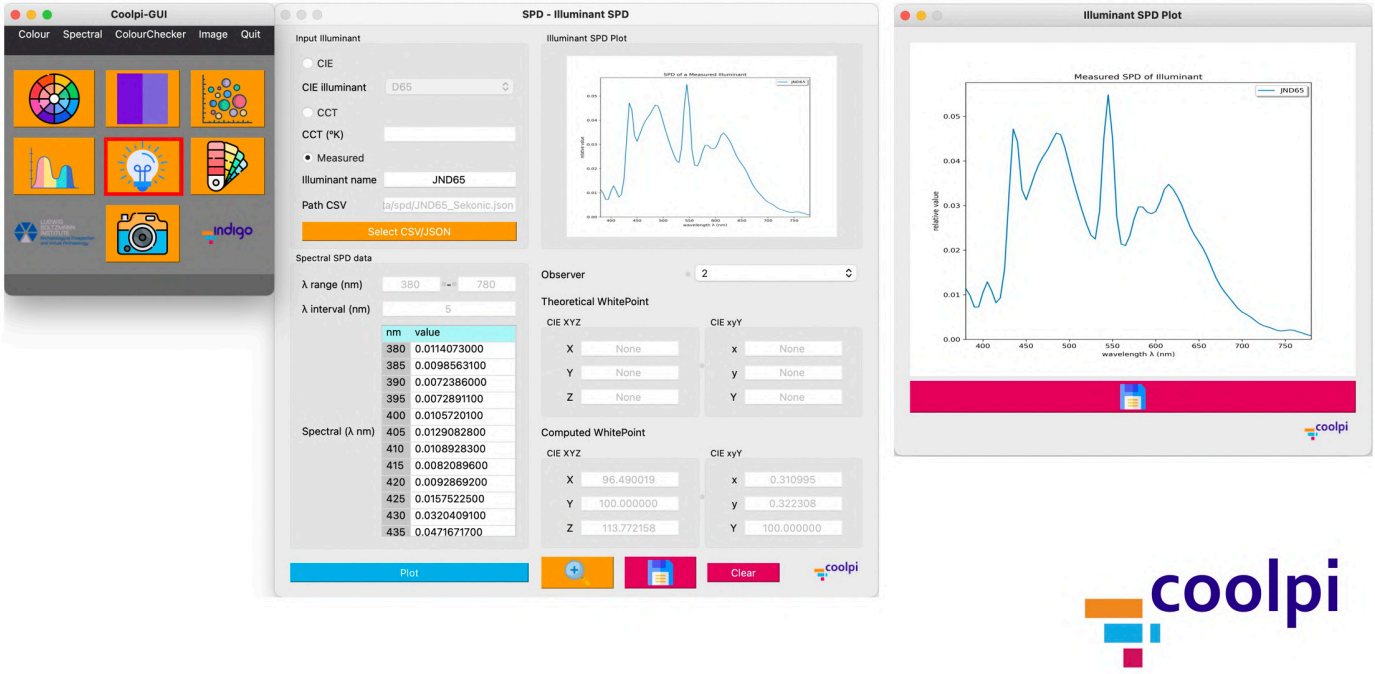

Figure 88: *Measured Illuminant*

Plot: Click on the *Plot* button to display the illuminant spectral power distribution figure. In addition to the figure, the illuminant data (i.e. range, interval and SPD) shall be loaded on the screen.

If the theoretical data is available for the illuminant white point (or reference white), they shall be displayed on the screen (only for illuminants CIE A, C, D50, D55, D65, D75). Regardless of the illuminant type, the reference white in CIE XYZ and xy chormaticity coordinates will be calculated and shown. By default they will be computed for the CIE 1931 2º standard observer, however, the observer can be selected directly in the `Observer` combobox.

Zoom: Click the *Zoom* button to display the `Illuminant SPD Plot` for the input illuminant in a new window. This button is active once the spectral power distribution of the illuminant has been loaded and displayed on the screen.

Save: Click the *Save* button to export the `Illuminant SPD Plot` as an image. This button is active once the spectral power distribution of the illuminant has been loaded and displayed on the screen.

Clear: Click the *Clear* button to delete the data and start with a new illuminant.

10.8 CCI: Colour Checker Inspector Tool

Tool to deal with colour checkers.

Colour checkers are a widely used tool in digital photography, especially for white balance adjustment and colour calibration. There are different colour checkers, provided by different manufacturers. The most widespread are probably those made by X-Rite (now Calibrite) and Datacolor. In coolpi-gui we have tried to include the most widespread ones, not only for digital imaging applications, but also for other disciplines (e.g. the Munsell Soil Color Book for the measurement of colour of soil samples).

The following ColourCheckers have been implemented in the coolpi-gui:

- Calibrite CLASSIC (CCC).
- Calibrite PASSPORT PHOTO 2 (CCPP2).
- Calibrite PASSPORT VIDEO (CCPPV).
- Calibrite DIGITAL SG (CCDSG).
- GretagMacbeth (GM).
- Munsell Soil Color Book Ed. 1994 (MSCB_1994).
- Munsell Soil Color Book Ed. 2009/2022 (MSCB_2009).
- Pantone Metallics (PM).
- QPCARD 202 (QPGQP202).
- RAL K5 CLASSIC (RALK5).
- RAL K7 CLASSIC (RALK7).

- Spyder Checker (SCK100).
- X-Rite PASSPORT PHOTO (XRCCPP).
- X-Rite Digital SG (XRCCSG).

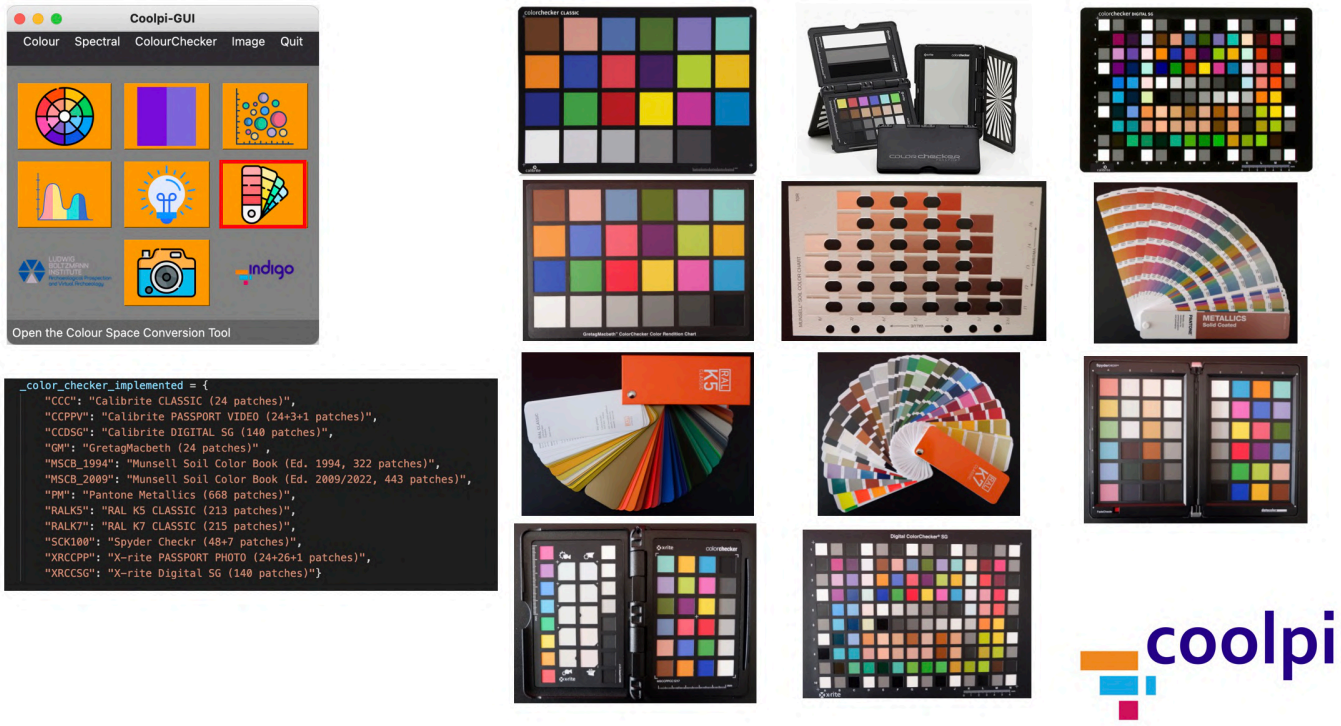

Figure 89: *Colour Checkers Implemented*

ColourChecker: The ColourChecker spectral data can be loaded from the coolpi resources (Option: `From Resources`) or by a JSON file (Option: `Measured`). Select the ColourChecker name from the combobox for use the coolpi resources data. Click the `Load from JOSN` button and select a valid path to the JSON file to import the ColourCheckerRGB measured data. Once selected, the following information about the colour checker will appear on the screen: Instrument (instrument used for the spectral measurement), Meas. Date (date of the measurement), Illuminant, Observer, Num. Patches (total number of colour samples), λ range (min, max range in nm) and λ interval in nm. In the `Spectral Data` section, the patch identifiers together with their spectral data shall be loaded.

Figure 90: *CCI - ColourChecker Inspector - From Resources*

For a correct import of the measured spectral data of the colour checker, the JOSN file shall have the folowing keys: “NameColorChart”, “Instrument”, “Measurement Date”, “Illuminant”, “Observer”, “nm_range” and “nm_interval”. Otherwise, a warning message will be displayed on the screen.

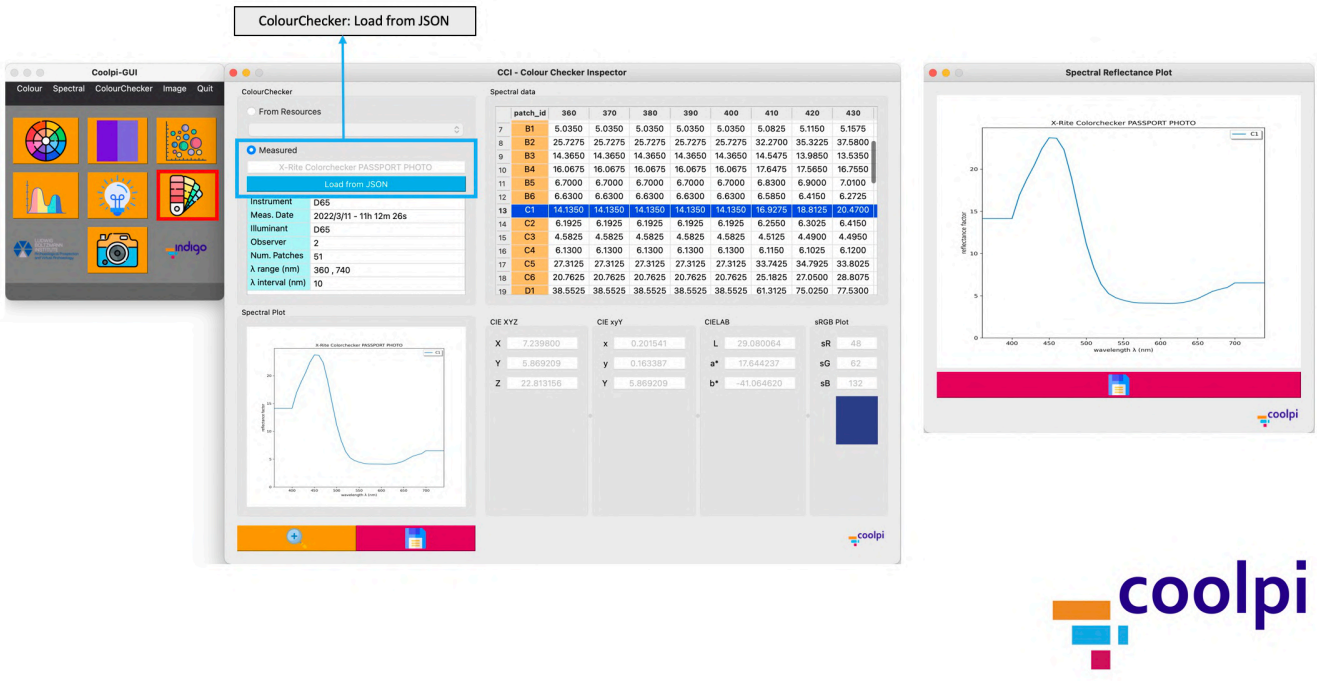

Figure 91: *CCI - ColourChecker Inspector - Measured*

Spectral Plot: Select a patch by clicking in the `Spectral Data` section. Once a patch is

selected, the graph of its spectral data will be loaded on the screen. The `Spectral Plot` will be updated automatically when the selected patch is changed.

In addition to the graphic, the CIE XYZ coordinates computed from the spectral data, as well as the CIE xy, CIELAB and sRGB (under D65) coordinates of the selected patch will be loaded on the screen. The coordinates will be updated automatically when another patch is selected.

Zoom: Click the *Zoom* button to display the `Spectral Reflectance Plot` of the selected patch in a new window. This button is active once the spectral data of the selected patch has been loaded and displayed on the screen.

Save: Click the *Save* button to export the `Spectral Reflectance Plot` of the selected patch as an image. This button is active once the spectral data of the selected patch has been loaded and displayed on the screen.

> ⚠ **Alert**
>
> *We recommend spectral measurements of the colour charts on a regular basis. Over time, patches may deteriorate or become dirty, and their spectral values may not necessarily match those obtained in previous measurements.*

10.9 RCIP: RAW Colour Image Processing

Tool to get colour-accurate data from RAW images.

Load Image: Click the `Load Image` button to import the desired RAW image. Once selected, the `Processing Options` will be active. The image path will be automatically updated in the `RAW Imagage` path tag.

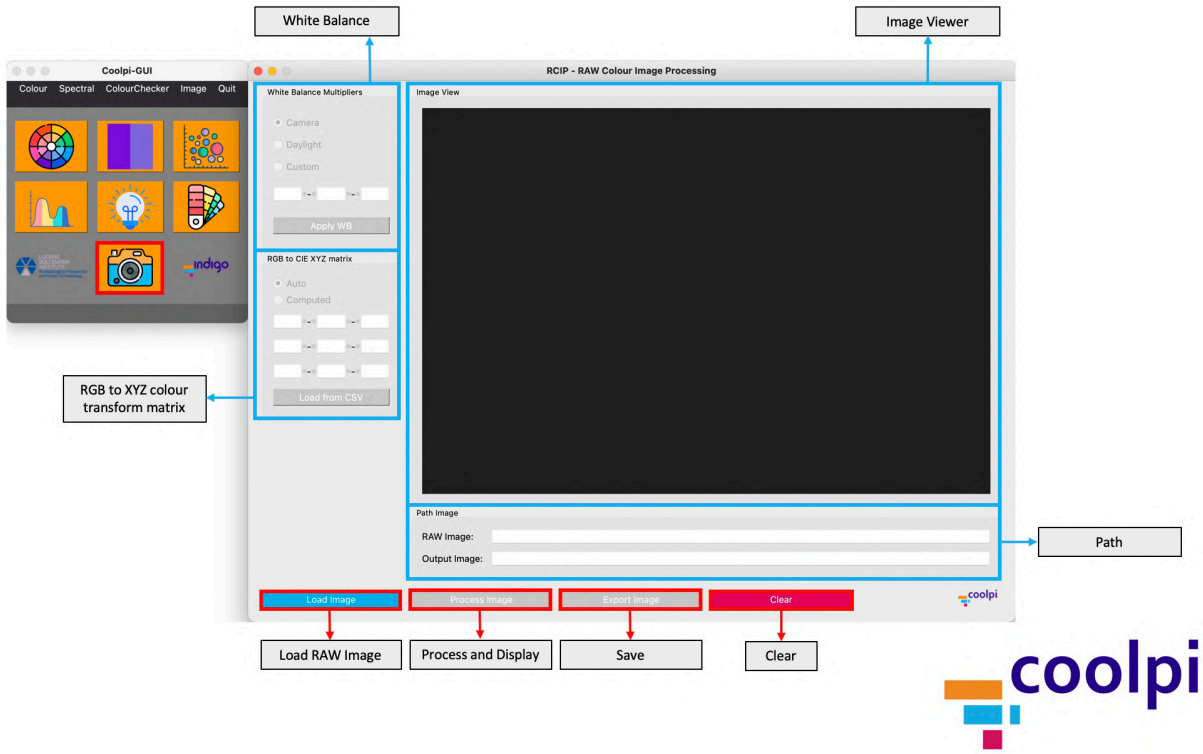

Figure 92: *RCIP - RAW Colour Image Processing*

In addition, the RAW RGB image will be displayed on the screen.

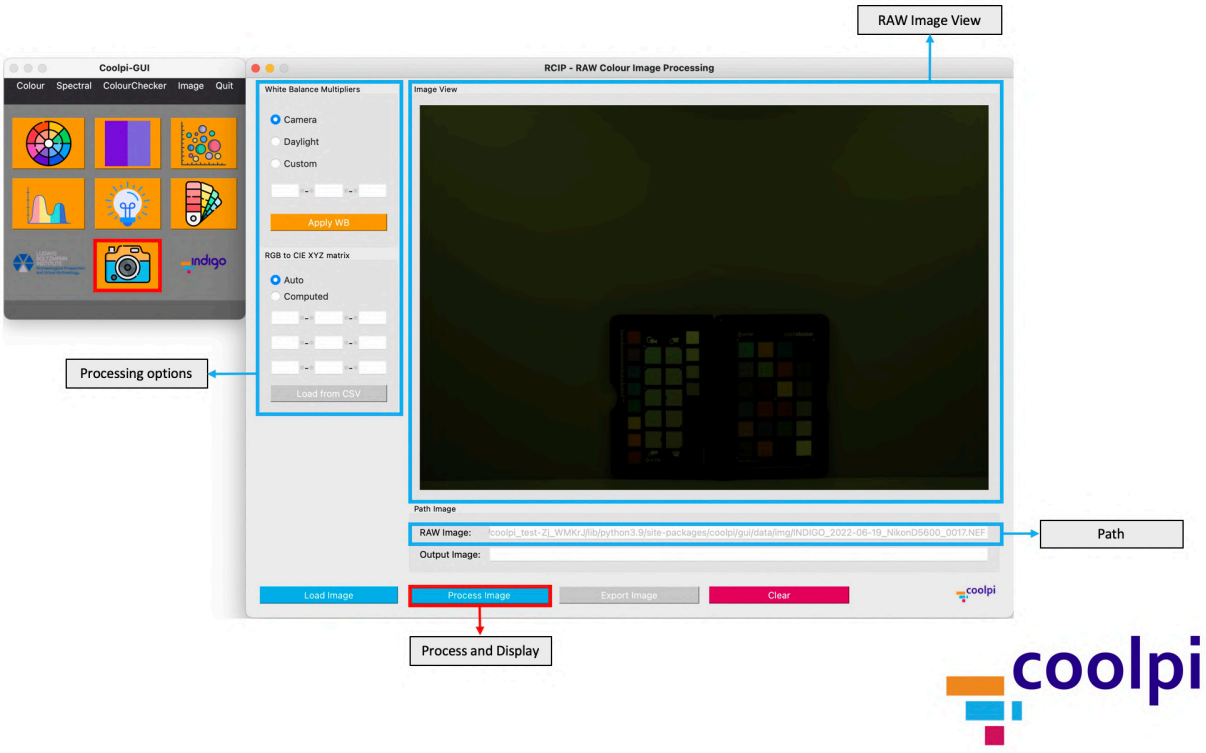

Figure 93: *RCIP - RAW Image View*

Image View: This is the image viewer to display the original raw image, the image after applying white balance, or the final colour-corrected image.

White Balance Multipliers: Select the desired option for the white balance: `Camera`, `Daylight`

or `Custom`. For a correct colour-correction of the RAW image, we recommend the proper computation of the white balance multipliers beforehand.

RGB to CIE XYZ matrix: Select the desired option for the colour transform matrix: `Auto` or `Computed`. If the `Auto` option is selected, the camera embedded colour transform array will be used. If a computed RGB to CIE XYZ transformation matrix is available, it can be added manually or imported using a CSV file by clicking the button `Load from CSV`.

Process Image: Click the *Process Image* to process the RAW image using the current `Processing Options`.

Export Image: Click the *Export Image* button to export the colour-corrected image. The image can be exported as JPEG (8bits), or PNG/TIF (16 bits).

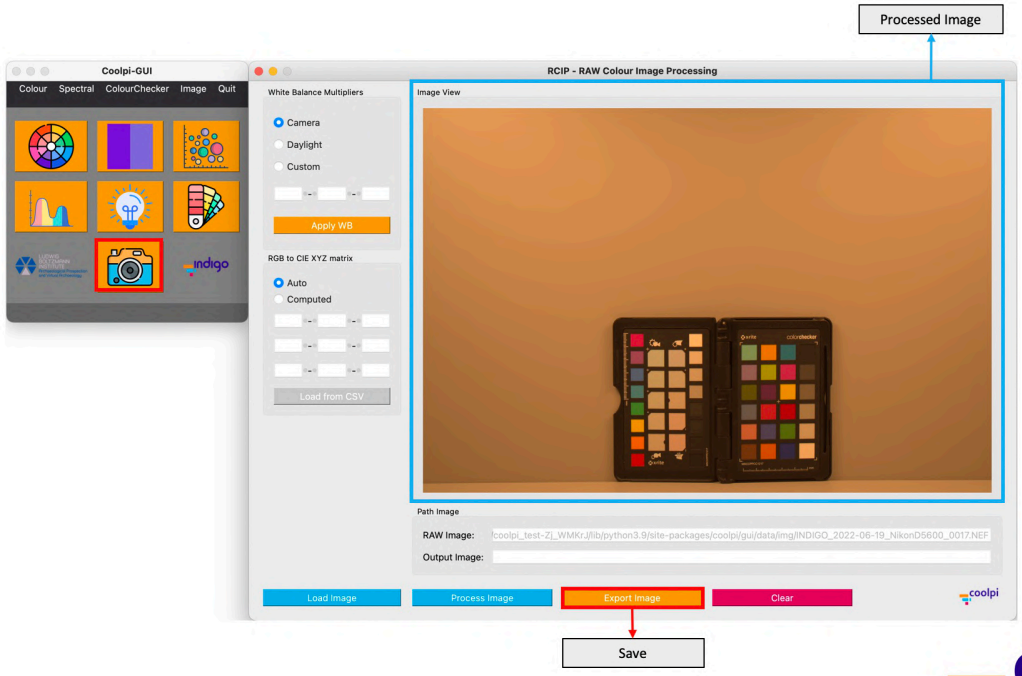

Figure 94: *RCIP - Automatic Processed Image*

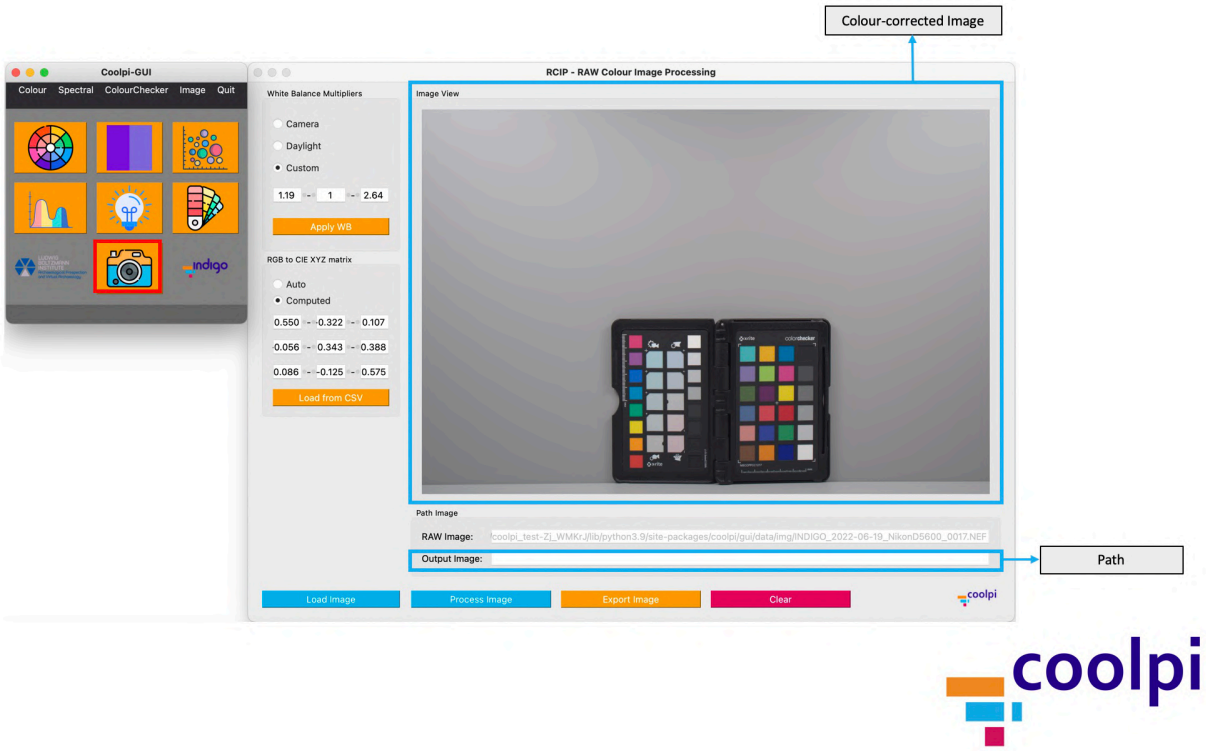

Figure 95: *RCIP - Colour-corrected Image*

Info

For further details on the computation of the white balance multipliers and the RAW image processing, please refer to the RawImage class section).

11 References

- CIE. Colorimetry, 4th Edition. 2018. Commission Internationale de l'Éclairage, Vienna, Austria.
- Hernandez-Andres, J., Lee, R. L., & Romero, J. 1999. Calculating correlated color temperatures across the entire gamut of daylight and skylight chromaticities. Applied optics, 38(27), 5703-5709
- Hunt, R. W. G. and Pointer, M. R. 2011. Measuring colour. John Wiley & Sons.
- International Electrotechnical Commission. 1999. IEC/4WD 61966-2-1: Colour Measurement and Management in Multimedia Systems and Equipment - Part 2-1: Default RGB Colour Space - sRGB
- Kruschwitz, J. D. 2018. Field guide to colorimetry and fundamental color modeling. Society of Photo-Optical Instrumentation Engineers. SPIE
- Luo, M. R., Cui, G., and Rigg, B. 2001. The development of the CIE 2000 colour-difference formula: CIEDE2000. Color Research & Application, 26(5), 340-350
- Luo, M. R. 2016. Encyclopedia of color science and technology. Springer New York.
- McCamy, C.S. 1992. Correlated color temperature as an explicit function of chromaticity coordinates, Color Res. Appl. 17, 142-144.
- Molada-Tebar, A.; Lerma, J.L.; Marqués-Mateu, Á. 2018. Camera characterization for improving color archaeological documentation. Color Res. Appl. (43), pp. 47-57.
- Molada-Tebar, A.; Marqués-Mateu, Á.; Lerma, J.L. 2019. Correct use of color for cultural heritage documentation. ISPRS Ann. Photogramm. Remote Sens. Spatial Inf. Sci., IV-2/W6, 107-113.
- Molada-Tebar, A.; Riutort-Mayol, G.; Marqués-Mateu, Á.; Lerma, J.L. (2019). A Gaussian Process Model for Color Camera Characterization: Assessment in Outdoor Levantine Rock Art Scenes. Sensors, 19, pp.4610.
- Molada-Tebar, A.; Marqués-Mateu, Á.; Lerma, J.L.; Westland, S. (2020) Dominant Color Extraction with K-Means for Camera Characterization in Cultural Heritage Documentation. Remote Sens. 12, pp.520.
- Ohta, N., and Robertson, A. 2005. Colorimetry: fundamentals and applications. John Wiley & Sons.
- Ohno, Yoshi. 2014. Practical Use and Calculation of CCT and Duv, LEUKOS, 10:1, 47-55.
- Sharma, G., Wu, W., & Dalal, E. N. 2005. The CIEDE2000 color-difference formula: Implementation notes, supplementary test data, and mathematical observations. Color Research & Application 30(1), 21-30
- Schanda, János. 2007. Colorimetry: understanding the CIE system. John Wiley &

Sons.
- Verhoeven, G., Wild, B., Schlegel, J., Wieser, M., Pfeifer, N., Wogrin, S., Eysn, L., Carloni, M., Koschiček-Krombholz, B., Molada-Tebar, A., Otepka-Schremmer, J., Ressl, C., Trognitz, M. and Watzinger, A. 2022. Project Indigo-Document, Disseminate & Analyse a Graffiti-Scape. ISPRS-International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences, 46, 513-520.

12 Credits

The coolpi package has been developed by Adolfo Molada Tebar.