A computational framework for rendering sRGB color from physical light spectra

B. SARIKAVAK-LISESIVDIN, S. B. LISESIVDIN*
Gazi University, Faculty of Science, Department of Physics, 06500, Teknikokullar, Ankara, Türkiye

This study introduces a computational method to convert any spectral power distribution (SPD) into its corresponding sRGB color, with practical applications in photonics, material science, and display engineering. Using Python libraries for numerical calculations, interpolation, and visualization, spectral data with the CIE color-matching functions were computed to derive tristimulus values. A matrix transformation from XYZ to the sRGB color space is then applied, followed by normalization and gamma correction to ensure precision. The method shows consistent results across various test spectra with different peak features. Four illustrative examples are provided to demonstrate the accuracy and adaptability of the approach.

(Received September 15, 2025; accepted June 2, 2026)

Keywords: Spectral power distribution, CIE 1931 color space, XYZ to sRGB conversion

1. Introduction

Color perception arises from the complex interplay between the spectral composition of light and the physiological response of the human visual system [1]. Spectral data, represented as spectral power distributions (SPD), capture the intensity of light across a range of wavelengths [2]. Colorimetric data conversion plays a crucial role in various fields, including lighting design [3], display calibration [4], remote sensing [5], and biological research [6], where accurate spectral interpretation is essential. It is therefore important to provide accessible tools for converting spectral data into perceptual color values [7].

The CIE 1931 color space has become a foundational reference for this purpose [8]. It relies on three color-matching functions, $\bar{x}(\lambda)$, $\bar{y}(\lambda)$, and $\bar{z}(\lambda)$, derived from empirical experiments [9,10]. By projecting an SPD onto color-matching functions, one can compute the XYZ tristimulus values, which form a standardized representation of color [11, 12]. These XYZ values are subsequently mapped to a given RGB color space—commonly sRGB—through a matrix transformation and color correction [13-15].

Despite the existence of established theoretical models, practical implementation challenges persist. For instance, interpolation artifacts can occur if the wavelength sampling of the SPD differs from that of the CIE functions [14]. Also, normalization steps can alter the spectrum, especially in cases of narrow-band distributions [16]. Because the sRGB space remains the most universally adopted for digital systems, studying the transformation of SPD to sRGB will help more successful rendering of SPD-based data in simulations and real-world applications.

In this work, we present a Python-based methodology that integrates widely available libraries to manage data I/O, numerical interpolation, and final color representation. Through three case studies—covering narrow-band, broad, and mixed-peak spectra—we demonstrate that the approach is robust, flexible, and suitable for a range of applications.

2. Methods

2.1. Data acquisition and interpolation

Spectral data were acquired or generated in the form of CSV files consisting of wavelengths and corresponding intensities. For compatibility with various input file structures, *pandas.read_csv* was employed with flexible delimiter detection (*delimiter=None, engine='python'*). This approach ensures the script can gracefully handle comma, tab, or space delimiters:

```
data = pd.read_csv(input_file,
delimiter=None, engine='python')
wavelengths = data.iloc[:, 0].to_numpy()
intensities = data.iloc[:, 1].to_numpy()
```

For consistency, all spectra were sampled between 380 nm and 780 nm, with a 1 nm interval where applicable. The CIE 1931 color-matching functions $\bar{x}(\lambda)$, $\bar{y}(\lambda)$, and $\bar{z}(\lambda)$ were also stored in a CSV file aligned with the same or broader wavelength range.

Potential discrepancies in wavelength sampling were addressed using one-dimensional interpolation via *scipy.interpolate.interp1d*:

$$\bar{x}_{interp}(\lambda) = interp1d(\lambda_{CIE}, \bar{x}(\lambda_{CIE}), \dots) \quad (1)$$

(and similarly for $\bar{y}(\lambda)$, and $\bar{z}(\lambda)$). Values out of range were set to zero.

2.2. Tristimulus value calculation

Once each wavelength in the input spectrum was mapped to the interpolated color-matching functions, the XYZ tristimulus values were computed as

$$X = k\sum_{\lambda} S(\lambda).\bar{x}(\lambda).\Delta\lambda \quad (2)$$

$$Y = k\sum_{\lambda} S(\lambda).\bar{y}(\lambda).\Delta\lambda \quad (3)$$

$$Z = k\sum_{\lambda} S(\lambda).\bar{z}(\lambda).\Delta\lambda \quad (4)$$

where $S(\lambda)$ is the spectral intensity, and k is a normalization constant [17]. In many practical implementations, $k = 1$ if intensities can be scaled to a maximum value of 1 [18, 19]. The sums are computed numerically in Python as

```
x = np.sum(intensities *
x_interp(wavelengths))
y = np.sum(intensities *
y_interp(wavelengths))
z = np.sum(intensities *
z_interp(wavelengths))
```

2.3. XYZ to sRGB transformation

The transformation from XYZ to sRGB uses the standard matrix multiplication with *np.dot*.

Negative values are clamped to zero as

```
rgb_linear = np.dot(rgb_transform_matrix,
[x, y, z])
rgb_linear = np.clip(rgb_linear, 0, None)
```

This ensures that no channel is negative before the application of gamma correction. Related transformation matrix is [21,22]

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 3.2406 & -1.5372 & -0.4986 \\ -0.9689 & 1.8758 & 0.0415 \\ 0.0557 & -0.2040 & 1.0570 \end{bmatrix} . \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \quad (5)$$

2.4 Gamma correction, visualization, and validation

Gamma correction compensates for a monitor's non-linear response, ensuring images are displayed accurately [23]. A gamma correction function is applied [11] as

$$R_{corrected} = \begin{cases} 12.92 \, . R_{linear}, & if \; R_{linear} \leq 0.0031308 \\ 1.055 \, . (R_{linear})^{\frac{1}{2.4}} - 0.055, & otherwise \end{cases} \quad (6)$$

(and similarly for G and B). Finally, the channels are scaled to a [0,255] range as

```
rgb_linear /= np.max(rgb_linear) #
Normalize
rgb_corrected = [gamma_correct(c) for c in
rgb_linear]
rgb_255 = [int(round(c * 255)) for c in
rgb_corrected]
```

Optionally, one may plot the CIE xy chromaticity diagram and map the calculated color to confirm accuracy. This step provides a visual check on whether the transformed values align with expected perceptual outcomes. In Python, such a diagram is typically generated via matplotlib, overlaying the computed chromaticity coordinates on the standard CIE diagram.

2.5. Python libraries used

The following Python libraries are used in the workflow:

numpy: Provides numerical arrays, matrix operations, and efficient summations [23].

pandas: Reads and handles CSV files with flexible delimiters [24].

matplotlib: Enables plotting of the CIE chromaticity diagram and visual inspection of results [25].

scipy.interpolate: Supplies interpolation routines (interp1d) to align input wavelengths with the CIE color-matching functions [26].

3. Computational workflow

Fig. 1 summarizes the step-by-step process for converting spectral power distributions (SPDs) into RGB color values. First, the spectral data (wavelength and intensity) are loaded from a CSV file, and then the CIE 1931 color-matching functions $\bar{x}$, $\bar{y}$, and $\bar{z}$, are imported and aligned via interpolation to match the input wavelengths. Next, the CIE XYZ tristimulus values are computed by integrating the product of the intensities and the interpolated color-matching functions. Once the XYZ values are obtained, they are transformed into the sRGB color space via a standard matrix multiplication. The resulting linear RGB values undergo normalization and gamma correction to ensure perceptual accuracy before being scaled to a typical 0–255 range. Finally, users have the option to visualize the outcome on a CIE xy chromaticity diagram, providing a clear indication of how the computed color would appear under standard viewing conditions.

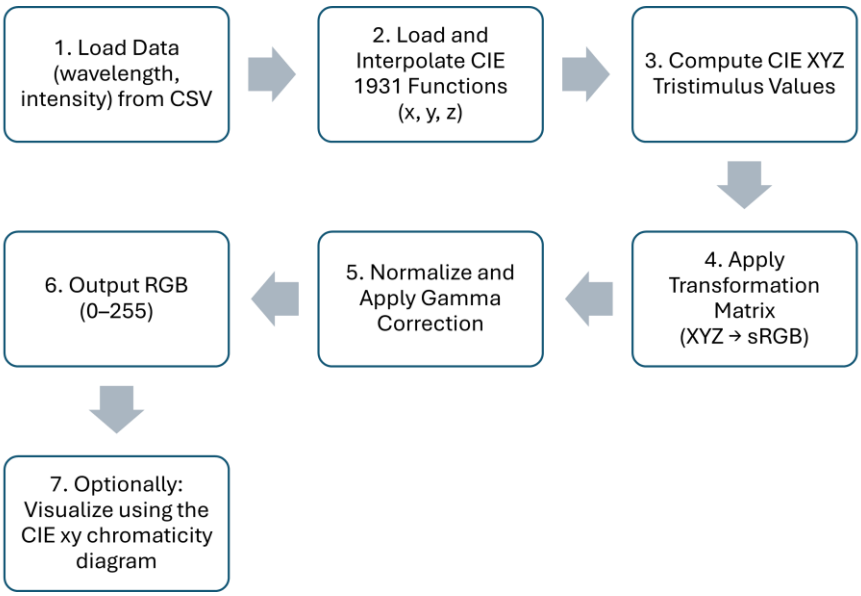

Fig. 1. A seven-step workflow for converting spectral data to RGB values based on the CIE 1931 color space (colour online)

4. Results and discussion

A synthetic narrow-band spectrum with a peak at 550 nm was generated and shown in Fig. 2. After interpolation and XYZ computation, the sRGB values conversion gives a definite green (0,255,0). This output exhibits a distinctly bright green tone, demonstrating the method's sensitivity to spectra with a given narrow-band single color.

A dual-peak spectrum with prominent emissions at 520 nm and 600 nm was also tested and shown in Fig. 3. The final sRGB values, around (255, 248, 0), created a bright yellow hue, which is a combination of bright red and bright green emission. The interpolation and normalization steps effectively balanced the two peaks, confirming the reliability of this workflow for multi-peaked distributions.

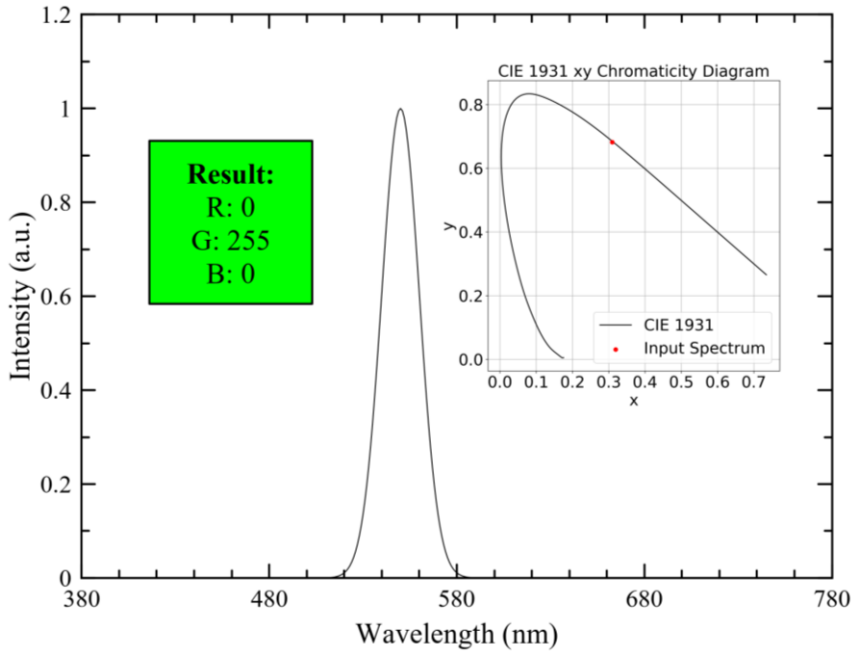

Fig. 2. A narrow Gaussian emission peak data. Inset: Resulted CIE 1931 xy chromaticity diagram and sRGB result (colour online)

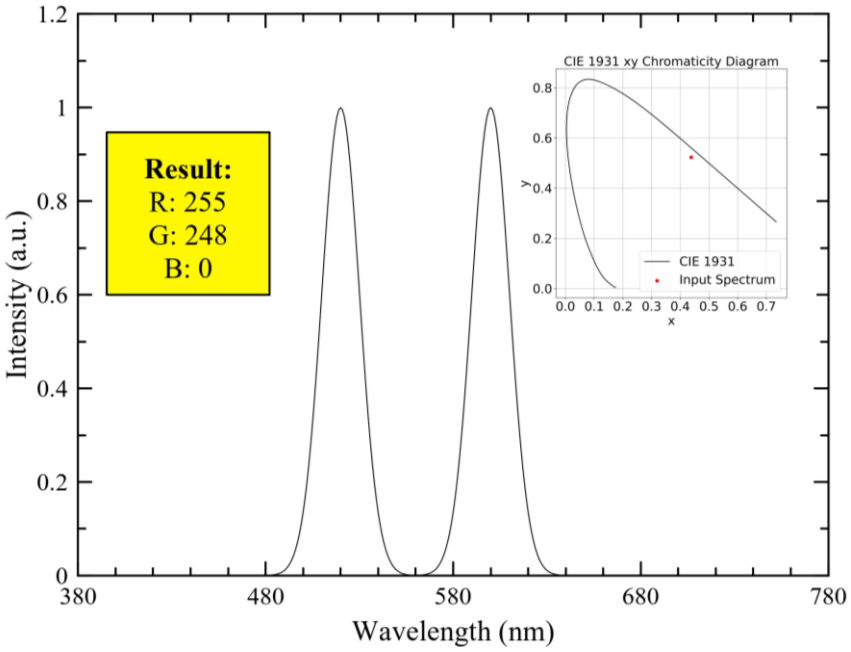

Fig. 3. Two narrow Gaussian emission peak data. Inset: Resulted CIE 1931 xy chromaticity diagram and sRGB result (colour online)

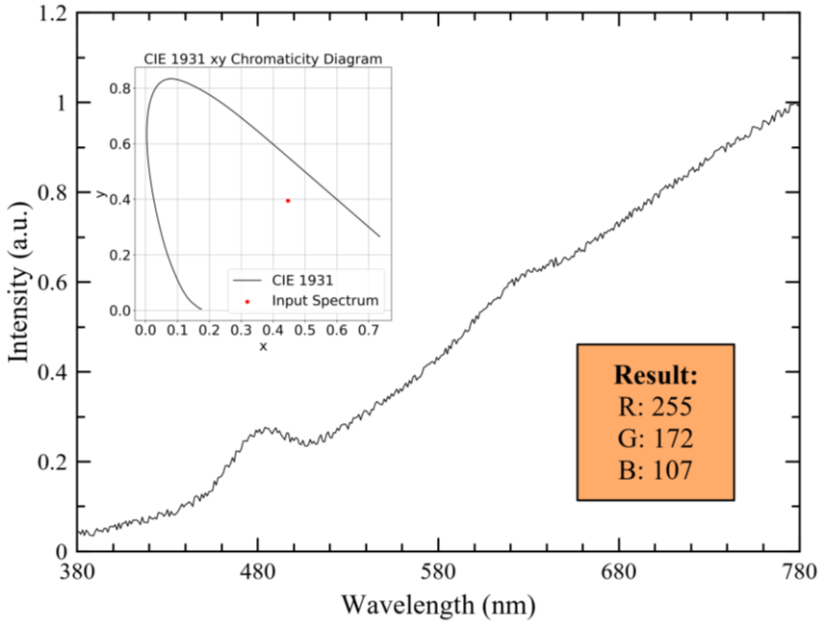

Fig. 4. A hypothetical incandescent light synthetic spectrum data. Inset: Resulted CIE 1931 xy chromaticity diagram and sRGB result (colour online)

A hypothetical incandescent light synthetic SPD was produced and shown in Fig. 4. The resulting sRGB values were around (255,172,107), appearing as a vivid yellowish-orange hue. This outcome validates the algorithm's ability to handle broad and smoothly varying spectra, a common scenario in practical lighting assessments.

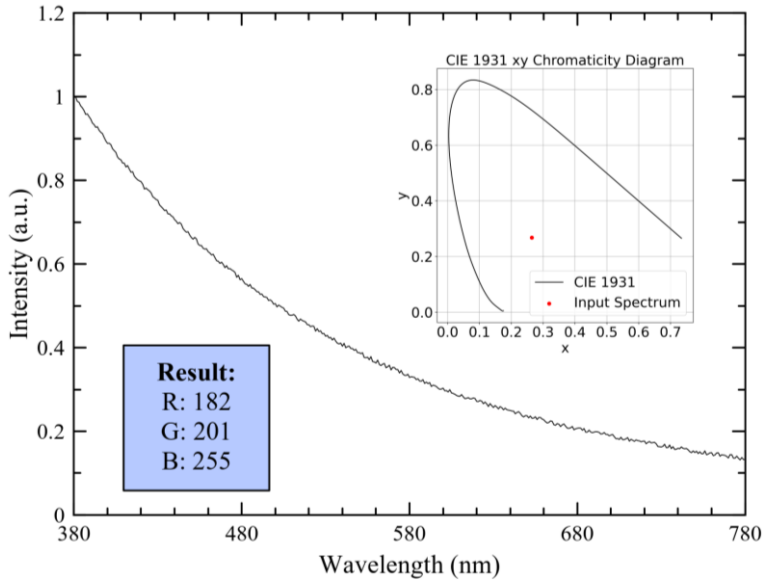

Fig. 5. A hypothetical blue star emission synthetic spectrum data. Inset: Resulted CIE 1931 xy chromaticity diagram and sRGB result (colour online)

The emission data of an object with a temperature of 15000K, created to behave like an emission of a hypothetical blue star, with a Planckian distribution with random noise, is given in Figure 5. The resulting sRGB values were around (182,201,255), appearing as bluish-white light. This outcome validates the algorithm's ability to handle also bluish colors, in addition to green, red, and their mixtures.

Table 1 summarizes the intermediate and final color calculations for each example discussed in this study. Computed XYZ rows show how each spectral dataset maps to the CIE XYZ space after integration with the color-matching functions. These values are then converted to Linear RGB before clamping, reflecting the direct output of the XYZ-to-sRGB matrix multiplication. In some cases, negative channels appear, indicating that a portion of the spectrum lies outside the standard sRGB primaries until correction is applied. The subsequent Normalized Linear RGB row shows the result of clamping negative values to zero and scaling the highest channel to 1.0, ensuring consistent relative proportions. Finally, the Final RGB row lists the gamma-corrected integer values in the familiar 0–255 range, which correspond to visually interpretable colors. Readers should note how initial XYZ values sometimes yield negative RGB channels before clamping, and how gamma correction and normalization resolve these into final sRGB colors. This highlights the algorithm's robustness across various SPD profiles.

Table 1. Computed XYZ, Linear RGB before clamping, Normalized linear RGB, Final RGB values of investigated examples in the study

Examples		**1**	**2**	**3**	**4**
Computed XYZ	**X**	11.0693	27.6380	46.6477	39.8153
	Y	24.4054	33.0230	41.2842	40.3682
	Z	0.30345	2.47044	16.5217	70.3507
Linear RGB before clamping	**R**	-1.7959	37.5692	79.4667	31.8947
	G	35.0672	35.2685	32.9297	40.0651
	B	-4.0413	-2.5859	11.6397	68.3433
Normalized linear RGB	**R**	0.0000	1.0000	1.0000	0.4666
	G	1.0000	0.9387	0.4143	0.5862
	B	0.0000	0.0000	0.1464	1.0000
Final RGB	**R**	0	255	255	182
	G	255	248	172	201
	B	0	0	107	255

5. Conclusions

This study presented a robust, Python-based methodology for converting spectral power distributions into perceptually accurate RGB colors under the CIE 1931 color space. Through a systematic combination of interpolation, XYZ tristimulus calculations, matrix-based color transformation, and gamma correction, our workflow enables reliable rendering of spectral data for various applications. The results indicate that the proposed approach handles both narrow-band and broad emission spectra effectively, producing visually consistent sRGB outputs. Moreover, the method's reliance on open-source libraries allows for straightforward adaptation and extension to alternative color spaces or specialized domains. Future work may involve integrating additional perceptual models, supporting real-time spectral analysis, and further refining performance for hyperspectral data. By providing code that is easily inspected and tested, this work encourages reproducible research and fosters collaborative development in the growing intersection of color science and computational techniques.

Acknowledgments

This work supports open data and reproducible science. The main code described in this paper, along with the other codes used to generate all synthetic data samples, is published under the MIT license in the public repository:

https://github.com/sblisesivdin/spectrum-to-rgb.

References

[1] S. Benkner, A. Herzog, S. Klir, W. D. Van Driel, T. Q. Khanh, IEEE Access **10**, 83612 (2022).
[2] M. Koedam, J.J. Opstelten, Light. Res. Technol. **3**(3), 205 (1971).
[3] M. Karlen, C. Spangler, J. R. Benya, Lighting design basics, John Wiley & Sons, New York, 2017.
[4] C. Meininger, T. Lianza, G. Annese, Fundamentals and Applications of Colour Engineering, John Wiley & Sons, 99 (2023).
[5] S. P. Mertikas, P. Partsinevelos, C. Mavrocordatos, N. A. Maximenko, Pollution Assessment for Sustainable Practices in Applied Sciences and Engineering, Butterworth-Heinemann, 107 (2021).
[6] G. G. Hammes, Spectroscopy for the biological sciences, John Wiley & Sons, New York, 2005.
[7] G. Sharma, R. Bala (Eds.), Digital color imaging handbook, CRC Press, Boca Raton, 2017.
[8] Y. Ohno, Proc. NIP Digit. Fabr. Conf. **16**, 540 (2000).
[9] M. da Fonseca, I. Samengo, Neural Comput. **33**(9), 2578 (2021).
[10] E. Allen, J. Opt. Soc. Am. **56**(9), 1256 (1966).
[11] M. Magnusson, J. Sigurdsson, S. E. Armansson, M. O. Ulfarsson, H. Deborah, J. R. Sveinsson, Proc. IEEE Int. Geosci. Remote Sens. Symp. (IGARSS), 2045 (2020).
[12] C. Wyman, P. P. Sloan, P. Shirley, J. Comput. Graph. Tech. **2**(2), 11 (2013).
[13] A. Molada-Tebar, G. J. Verhoeven, D. Hernández-López, D. González-Aguilera, Sensors **24**(6), 1743 (2024).
[14] S. Süsstrunk, R. Buckley, S. Swen, Proc. Color Imag. Conf. **7**, 127 (1999).
[15] M. Afifi, A. Abdelhamed, A. Abuolaim, A. Punnappurath, M. S. Brown, IEEE T. Pattern Anal. **44**(9), 4688 (2021).
[16] M. Kretkowski, R. Jablonski, Y. Shimodaira, IEICE T. Inf. Syst. **93**(3), 651 (2010).
[17] R. Lukac (Ed.), Perceptual digital imaging: methods and applications, CRC Press, Boca Raton, 151 (2017).
[18] E. I. Stearns, Text. Chem. Color. **17**(8), 53 (1985).
[19] V. Mohammadi, K. Ansari, P. Gouton, H. Attig, Sensors **24**(23), 7728 (2024).
[20] E. Reinhard, E.A. Khan, A.O. Akyuz, G. Johnson, Color imaging: fundamentals and applications, CRC Press, Boca Raton, 2008.
[21] S. Westland, C. Ripamonti, V. Cheung, Computational colour science using MATLAB, John Wiley & Sons, New York, 2012.
[22] International Electrotechnical Commission, IEC 61966-2-1, Colour management – Default RGB colour space - sRGB, 2003.
[23] C. R. Harris, K. J. Millman, S. J. Van Der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, Nature **585**(7825), 357 (2020).
[24] W. McKinney, Proc. 9th Python Sci. Conf., 51 (2010).
[25] J. D. Hunter, Comput. Sci. Eng. **9**(3), 90 (2007).
[26] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. Van Der Walt, Nat. Methods **17**(3), 261 (2020).

*Corresponding author: bora@gazi.edu.tr

Appendix

In addition to repository information given in Acknowledgements section, source code of *spectrum-to-rgb.py* and examples are given:

spectrum-to-rgb.py:

```
import csv
import os
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d

"""
spectrum-to-rgb.py
------------------
Python-based tool for converting spectral power
distributions (SPDs) to RGB colors using the CIE
1931 color space.

MIT License

Copyright (c) 2025-2026 by Beyza Lisesivdin and
Sefer Bora Lisesivdin

Full license information can be found in
LICENSE.md

Usage:
$ ./spectrum-to-rgb.py <filename>

"""

def load_csv(file_path):
    """
    Load a CSV file with automatic delimiter
detection.
    Returns a numpy array of the data.
    """
    with open(file_path, 'r') as file:
        # Detect delimiter
        sample = file.read(1024)
        file.seek(0)
        dialect = csv.Sniffer().sniff(sample,
delimiters=",\t ")
        delimiter = dialect.delimiter
        print(f"Detected delimiter in
{file_path}: '{delimiter}'")

    # Load data
    return np.genfromtxt(file_path,
delimiter=delimiter)

def load_spectrum(csv_file):
    """
    Load the spectrum data from a CSV file.
    The first column should be wavelength in nm,
    and the second column should be intensity.
    """
    data = load_csv(csv_file)
    if data.ndim != 2 or data.shape[1] != 2:
        raise ValueError(f"Invalid spectrum file
format: {csv_file}. Expected two columns.")
    return data[:, 0], data[:, 1]

def xyz_to_rgb(x, y, z):
    """
    Convert CIE XYZ color space to sRGB in the
range 0-255.
    """
    # Transformation matrix for sRGB
    mat = np.array([
        [3.2406, -1.5372, -0.4986],
        [-0.9689,  1.8758,  0.0415],
        [0.0557, -0.2040,  1.0570]
    ])

    # Convert XYZ to linear RGB
    rgb = np.dot(mat, [x, y, z])

    # Debug: Linear RGB before clamping
    print(f"Linear RGB before clamping: {rgb}")

    # Clip negative values
    rgb = np.clip(rgb, 0, None)

    # Normalize to the brightest channel
    max_val = np.max(rgb)
    if max_val > 0:  # Avoid division by zero
        rgb /= max_val

    # Debug: Normalized linear RGB
    print(f"Normalized linear RGB: {rgb}")

    # Apply gamma correction
    def gamma_correct(value):
        if value <= 0.0031308:
            return 12.92 * value
        else:
            return 1.055 * (value ** (1 / 2.4)) -
0.055

    rgb = [gamma_correct(c) for c in rgb]

    # Debug: Gamma-corrected RGB
    print(f"Gamma-corrected RGB: {rgb}")

    # Scale to 0-255
    rgb_255 = [int(round(c * 255)) for c in rgb]

    # Debug: Final RGB
    print(f"Final RGB (0-255): {rgb_255}")

    return rgb_255



def spectrum_to_xyz(wavelengths, intensities,
cie_file):
    """
    Convert a spectrum to CIE 1931 XYZ values.
    """
    cie_data = load_csv(cie_file)
    cie_wavelengths = cie_data[:, 0]
    cie_x = cie_data[:, 1]
    cie_y = cie_data[:, 2]
    cie_z = cie_data[:, 3]

    # Interpolate color matching functions to the
input wavelengths
    x_interp = interp1d(cie_wavelengths, cie_x,
bounds_error=False, fill_value=0)
    y_interp = interp1d(cie_wavelengths, cie_y,
bounds_error=False, fill_value=0)
    z_interp = interp1d(cie_wavelengths, cie_z,
bounds_error=False, fill_value=0)

    # Debug: Print interpolated values
    print(f"Interpolated x̄:
{x_interp(wavelengths)}")
    print(f"Interpolated ȳ:
{y_interp(wavelengths)}")
    print(f"Interpolated z̄:
{z_interp(wavelengths)}")

    x = np.sum(intensities *
x_interp(wavelengths))
    y = np.sum(intensities *
y_interp(wavelengths))
    z = np.sum(intensities *
z_interp(wavelengths))
```

```
    # Debug: Print XYZ values
    print(f"Computed XYZ: X={x}, Y={y}, Z={z}")
    return x, y, z

def plot_chromaticity_diagram(xy_point,
cie_file):
    """
    Plot the CIE 1931 xy chromaticity diagram and
mark the given xy point.
    """
    cie_data = load_csv(cie_file)
    cie_x = cie_data[:, 1]
    cie_y = cie_data[:, 2]
    cie_z = cie_data[:, 3]

    # Normalize to get xy values
    total = cie_x + cie_y + cie_z
    chromaticity_x = cie_x / total
    chromaticity_y = cie_y / total

    plt.figure(figsize=(8, 8))
    plt.plot(chromaticity_x, chromaticity_y,
label='CIE 1931 Boundary', color='black')
    plt.scatter(*xy_point, color='red',
label='Input Spectrum', zorder=5)
    plt.xlabel('x')
    plt.ylabel('y')
    plt.title('CIE 1931 xy Chromaticity Diagram')
    plt.grid(True)
    plt.legend()
    plt.show()

def main():
    import sys

    # Expect exactly one command-line argument
(the spectrum CSV)
    if len(sys.argv) != 2:
        print("Usage: python spectrum-to-rgb.py
<spectrum_file.csv>")
        sys.exit(1)

    # Get the directory of the current script
    script_dir =
os.path.dirname(os.path.abspath(__file__))

    # Paths for CIE file and input spectrum file
    cie_file = os.path.join(script_dir,
'CIE_xyz_1931_2deg.csv')
    #csv_file = os.path.join(script_dir,
input("Enter the name of the spectrum CSV file
(in the same folder): "))
    csv_file = sys.argv[1]  # The user-specified
spectrum filename
    if not os.path.exists(csv_file):
        print(f"Error: file '{csv_file}' not
found.")
        sys.exit(1)

    try:
        # Load spectrum data
        wavelengths, intensities =
load_spectrum(csv_file)
        print("Spectrum Data Loaded:",
wavelengths, intensities)

        if wavelengths.size == 0 or
intensities.size == 0:
            raise ValueError("Spectrum file is
empty or improperly formatted.")

        # Normalize intensities
        max_intensity = np.max(intensities)
        if max_intensity > 0:
            intensities = intensities /
max_intensity
        else:
            raise ValueError("Spectrum
intensities are all zero.")

        intensities = intensities /
np.max(intensities)
        # Convert spectrum to XYZ
        x, y, z = spectrum_to_xyz(wavelengths,
intensities, cie_file)

        # Normalize to xy chromaticity
        total = x + y + z
        if total == 0:
            raise ValueError("Spectrum produces
zero total intensity in XYZ space.")

        chromaticity_x = x / total
        chromaticity_y = y / total

        # Convert XYZ to RGB
        rgb = xyz_to_rgb(x, y, z)

        print(f"Chromaticity (x, y):
({chromaticity_x:.4f}, {chromaticity_y:.4f})")
        print(f"RGB Value (0-255): {rgb}")

        # Plot the CIE xy chromaticity diagram

plot_chromaticity_diagram((chromaticity_x,
chromaticity_y), cie_file)
    except Exception as e:
        print(f"Error: {e}")

if __name__ == "__main__":
    main()
```

Example 1_Green_Narrow_Gaussian.py:

```
import numpy as np
import pandas as pd

"""
Example 1 - Green Narrow Gaussian Peak

MIT License

Copyright (c) 2025-2026 by Beyza Lisesivdin and
Sefer Bora Lisesivdin

Full license information can be found in
LICENSE.md

Usage:
$ python Example1_Green_Narrow_Gaussian.py

"""

# Generating synthetic data for Example 1: Narrow
Gaussian peak at 550 nm
wavelengths_example1 = np.arange(380, 781, 1)  #
Wavelengths from 380 nm to 780 nm
intensity_example1 = np.exp(-0.5 *
((wavelengths_example1 - 550) / 10)**2)  #
Gaussian centered at 550 nm

# Saving Example 1 to a CSV
example1_data = pd.DataFrame({'Wavelength (nm)':
wavelengths_example1, 'Intensity':
intensity_example1})
example1_file_path =
"Narrow_Green_Gaussian_Spectrum.csv"
example1_data.to_csv(example1_file_path,
index=False)
```

Example 2_Two_Narrow_Gaussian.py:

```
import numpy as np
import pandas as pd

"""
Example 2 - Two Narrow Gaussian Peaks

MIT License

Copyright (c) 2025-2026 by Beyza Lisesivdin and
Sefer Bora Lisesivdin

Full license information can be found in
LICENSE.md

Usage:
$ python Example2_Two_Narrow_Gaussian.py

"""

# Generating synthetic spectrum with two narrow
Gaussian peaks
wavelengths_gaussian = np.arange(380, 781, 1)  #
Wavelengths from 380 nm to 780 nm

# Two narrow Gaussian peaks at 520 nm and 600 nm
intensity_gaussian = (
    np.exp(-0.5 * ((wavelengths_gaussian - 520) /
10)**2) +  # Peak at 520 nm
    np.exp(-0.5 * ((wavelengths_gaussian - 600) /
10)**2)    # Peak at 600 nm
)

# Normalize intensity to [0, 1]
intensity_gaussian = intensity_gaussian /
np.max(intensity_gaussian)

# Saving the spectrum to a CSV
gaussian_data = pd.DataFrame({'Wavelength (nm)':
wavelengths_gaussian, 'Intensity':
intensity_gaussian})
gaussian_file_path =
"Two_Narrow_Gaussian_Spectrum.csv"
gaussian_data.to_csv(gaussian_file_path,
index=False)

gaussian_file_path
```

Example 3_Incandescent.py:

```
import numpy as np
import pandas as pd

"""
Example 3 - Incandescent Light source

MIT License

Copyright (c) 2025-2026 by Beyza Lisesivdin and
Sefer Bora Lisesivdin

Full license information can be found in
LICENSE.md

Usage:
$ python Example3_Incandescent.py

"""
# Generating synthetic spectral profile for an
incandescent light source
# The profile follows a Planckian distribution
with a more complex curve to simulate realistic
incandescent behavior.

def planckian_distribution(wavelengths,
temperature=2700):
    """Generate a Planckian distribution for a
given temperature."""
    h = 6.626e-34  # Planck's constant, J*s
    c = 3.0e8      # Speed of light, m/s
    k = 1.38e-23   # Boltzmann's constant, J/K
    wavelengths_m = wavelengths * 1e-9  # Convert
nm to meters
    return (2 * h * c**2) / (wavelengths_m**5) /
(np.exp(h * c / (wavelengths_m * k *
temperature)) - 1)

# Wavelength range (380 nm to 780 nm)
wavelengths_incandescent = np.arange(380, 781, 1)

# Simulate incandescent light with some noise and
smoothing for more complexity
intensity_incandescent =
planckian_distribution(wavelengths_incandescent,
temperature=2700)
intensity_incandescent = intensity_incandescent /
np.max(intensity_incandescent)  # Normalize to
[0, 1]

# Adding minor peaks and smoothing
noise = 0.02 *
np.random.rand(len(intensity_incandescent))  #
Random noise
extra_peaks = (
    0.1 * np.exp(-0.5 *
((wavelengths_incandescent - 480) / 15)**2) +
    0.05 * np.exp(-0.5 *
((wavelengths_incandescent - 620) / 20)**2)
)
intensity_incandescent = intensity_incandescent +
noise + extra_peaks
intensity_incandescent = intensity_incandescent /
np.max(intensity_incandescent)  # Renormalize to
[0, 1]

# Saving the incandescent light spectrum to a CSV
incandescent_data = pd.DataFrame({'Wavelength
(nm)': wavelengths_incandescent, 'Intensity':
intensity_incandescent})
incandescent_file_path =
"Incandescent_Light_Spectrum.csv"
incandescent_data.to_csv(incandescent_file_path,
index=False)
```

Example 4_Blue_star.py:

```
import numpy as np
import pandas as pd

"""
Example 4 - Blue star emission

MIT License

Copyright (c) 2025-2026 by Beyza Lisesivdin and
Sefer Bora Lisesivdin

Full license information can be found in
LICENSE.md

Usage:
$ python Example4_Blue_star.py

"""

def planckian_blue_star_spectrum(
    wavelength_min=380,
    wavelength_max=780,
    temperature=15000,  # Typical color
temperature for a hot, blue star
```

```
    noise_level=0.01,
    filename="Blue_Star_Spectrum.csv"
):

    # Physical constants
    h = 6.62607015e-34  # Planck's constant (J*s)
    c = 3.0e8           # Speed of light (m/s)
    kB = 1.380649e-23   # Boltzmann constant
(J/K)

    # Create an array of wavelengths in meters
    wavelengths_nm = np.arange(wavelength_min,
wavelength_max + 1, 1)
    wavelengths_m = wavelengths_nm * 1e-9  #
convert nm to m

    # Compute Planck's law (arbitrary units)
    # B(λ, T) = (2hc^2 / λ^5) * 1 /
[exp(hc/(λkBT)) - 1]
    radiance = (2 * h * c**2) /
(wavelengths_m**5) / (
        np.exp((h * c) / (wavelengths_m * kB *
temperature)) - 1
    )

    # Normalize peak to 1
    radiance /= np.max(radiance)

    # Add small random noise for realism
    noise = noise_level *
np.random.rand(len(radiance))
    radiance += noise

    # Re-normalize to ensure max remains 1
    radiance /= np.max(radiance)

    # Create a DataFrame
    df = pd.DataFrame({
        "Wavelength (nm)": wavelengths_nm,
        "Intensity": radiance
    })

    # Save to CSV
    df.to_csv(filename, index=False)

    print(f"Planckian 'blue star' spectrum saved
to '{filename}'")

if __name__ == "__main__":
    planckian_blue_star_spectrum()
```