**Document Title**

Subject

Author

# HSI Processing documentation

Welcome to HSI Processing documentation!



Contents:

[Signature analysis X](#_6c7d2b4a8969bae08dac20ce20f09a02)

[Functions for analysis: X](#_d4909823c74dbcf26a9787b5c69b3c9d)

[get\_centroids\_and\_medoids() X](#_1abea44ee3ea3c20eee6d6b2836c5240)

[get\_cross\_correlation\_matrix() X](#_e23b18a4906c1020ec32b8c0673b861f)

[Clustering of hyperspectral data X](#_7f94f983b61999748812bce1c4899e57)

[Clustering methods: X](#_bb285c6040be8ba9b131d735c51918ae)

[CosClust X](#_423f5ce1455a9ab1cbe9f0e545a0a82d)

[HDBSCAN X](#_1d74e8e33fbd8e36ff91d7c9fe60d394)

[SCH X](#_4fdac6a1f7075331b59a79143096b71f)

[Methods of processing hyperspectral data X](#_70d3bf93886ef50d5f3c6853f91bae16)

[Processing methods: X](#_2f2ceed68632fa9c356a17f9678a76b0)

[normalize() X](#_cb386b59ae47c7a80774c5fbe8bcee44)

[rayleigh\_scattering() X](#_97dc421cfdfc245c4b74628401c899e5)

[sigma\_maximum\_filter() X](#_4677bdaea19204070f7095ed22618a23)

[Hyperspectral data reading tools X](#_0c24d9c6519995c2b1a3347b5f6183b9)

[Readers: X](#_bbda4f7e042a171ef7ca537202e127e2)

[open\_AVIRIS() X](#_5318c0050d7fa92fedb69b3de5305f1e)

[open\_ENVI() X](#_bc205aff12278a750929008f8c3cd706)

[open\_ERDAS() X](#_9df7df39cf8e59e4c9348b4439541ad2)

[open\_SpyFile() X](#_d011059ab784b30dd2b9efba1f60b5b8)

[open\_TIF() X](#_b9793d781dfdfc5082e9e480cf2267dd)

[Color image synthesis X](#_6a05ace1161008304031845629f5f5ec)

[Methods for RGB synthesis: X](#_95703a84eeb4c30d4f31240acc605668)

[hsi\_synthesize\_rgb() X](#_5caa67a6d5409b442dc9d4443ea99aaf)

[simple\_synthesize\_rgb() X](#_288617f63f11a8ac11bc461f1f614d44)

[labels\_to\_rgb() X](#_915760189d4def6e1fbb4afc161b159b)

[Class with a predefined set of colors: X](#_1fab9f3a13e34ea58a75f04edcd1d1b7)

[Color X](#_eaa74bf6e8ec889951e27070db177ae0)

[Empirical mode decomposition (EMD) X](#_942dcf191fd286491b2d81eb49215480)

[Methods for decomposition into EMD X](#_dbc7e03c86827bd1891254f8f788ec6d)

[SWEMD() X](#_2120079f72df660b2db6dad4b0e92fbf)

[SWEMD\_signal() X](#_cad30461ba5030c791e57c04dc72546e)

## Signature analysis

This collection of functions provides essential tools for analyzing and processing data in clustering and similarity contexts. These utilities include operations for calculating centroids and medoids, creating cross-correlation matrices, and additional methods that aid in understanding the structure and relationships within datasets.

### Functions for analysis:

|  |
| --- |
| **hsip.analysis.analysis.get\_centroids\_and\_medoids**(*labels*, *data*, *metric='cosine'*) |
| Compute centroids and medoids for clusters in labeled data.  This function calculates the centroids and medoids for each cluster in the provided data, given cluster labels. Centroids are the mean of all points in a cluster, while medoids are calculated as the point in the cluster closest to the centroid based on the specified distance metric.   |  |  | | --- | --- | | Parameters: | * **labels** (*np.ndarray*) – 1D array of cluster labels for the data points. Points with the same label are considered to belong to the same cluster. * **data** (*np.ndarray*) – 2D array of shape (n\_samples, n\_features) containing the data points. * **metric** (*str, optional*) – The distance metric used to calculate the medoids. Default is ‘cosine’. Other valid metrics include ‘euclidean’, ‘manhattan’, and others supported by scipy.spatial.distance.cdist. | | Returns: | **centroids** (*np.ndarray*) – 2D array of shape (n\_clusters, n\_features) containing the centroids of the clusters.**medoids** (*np.ndarray*) – 2D array of shape (n\_clusters, n\_features) containing the medoids of the clusters. |   Examples  Compute centroids and medoids for cosine similarity  **>>> import** **numpy** **as** **np** **>>> from** **hsip.analysis.analysis** **import** get\_centroids\_and\_medoids **>>>** labels = np.array([0, 0, 1, 1, 2, 2]) **>>>** data = np.array([ **...**  [1, 2], [2, 3], **...**  [3, 4], [4, 5], **...**  [5, 6], [6, 7] **...** ]) **>>>** centroids, medoids = get\_centroids\_and\_medoids(labels, data, metric='cosine') **>>>** print("Centroids:") **>>>** print(centroids) Centroids: [[1.5 2.5]  [3.5 4.5]  [5.5 6.5]] **>>>** print("Medoids:") **>>>** print(medoids) Medoids: [[1. 2.]  [3. 4.]  [5. 6.]] |

|  |
| --- |
| **hsip.analysis.analysis.get\_cross\_correlation\_matrix**(*data: ndarray*, *metric: str = 'euclidean'*) |
| Compute the cross-correlation matrix for a given dataset.  This function calculates the pairwise distances between all rows in the input data using the specified distance metric and returns the resulting cross-correlation matrix.   |  |  | | --- | --- | | Parameters: | * **data** (*np.ndarray*) – 2D array of shape (n\_samples, n\_features) containing the input data, where n\_samples is the number of data points and n\_features is the number of features. * **metric** (*str, optional*) – The distance metric to use for calculating pairwise distances. Default is ‘euclidean’. Supported metrics include ‘euclidean’, ‘manhattan’, ‘cosine’, and others supported by scipy.spatial.distance.cdist. | | Returns: | **cross\_corr\_mat** – 2D array of shape (n\_samples, n\_samples) containing the pairwise distances between all data points in the input data. | | Return type: | np.ndarray |   Examples  Compute cross-correlation matrix with the default Euclidean metric  **>>> import** **numpy** **as** **np** **>>> from** **hsip.analysis.analysis** **import** get\_cross\_correlation\_matrix **>>>** data = np.array([[0, 1], [1, 2], [2, 3], [3, 4]]) **>>>** cross\_corr\_mat = get\_cross\_correlation\_matrix(data) **>>>** print(cross\_corr\_mat) [[0. 1.41421356 2.82842712 4.24264069]  [1.41421356 0. 1.41421356 2.82842712]  [2.82842712 1.41421356 0. 1.41421356]  [4.24264069 2.82842712 1.41421356 0. ]] |

## Clustering of hyperspectral data

One of the main methods of processing the HSI is clustering.

### Clustering methods:

|  |
| --- |
| *class* **hsip.clustering.clustering.CosClust**(*threshold: float = 0.9*, *verbose=True*) |
| Bases: object  A clustering algorithm based on cosine similarity. Groups samples into clusters based on a threshold for cosine similarity and assigns labels to each sample.   |  |  | | --- | --- | | Parameters: | * **threshold** (*float, optional*) – The cosine similarity threshold for determining cluster membership. Default is 0.9. * **verbose** (*bool, optional*) – If True, displays progress and additional information during the clustering process. Default is True. |  |  | | --- | | **threshold** | | The cosine similarity threshold for clustering.   |  |  | | --- | --- | | Type: | float | |  |  | | --- | | **labels** | | The cluster labels assigned to each sample. Initialized as None and populated after fit is called.   |  |  | | --- | --- | | Type: | np.ndarray or None | |  |  | | --- | | **reference\_set** | | A list of reference samples representing each cluster.   |  |  | | --- | --- | | Type: | list | |  |  | | --- | | **centroids** | | 2D array of shape (n\_clusters, n\_features) containing the centroids of the clusters.   |  |  | | --- | --- | | Type: | np.ndarray | |  |  | | --- | | **medoids** | | 2D array of shape (n\_clusters, n\_features) containing the medoids of the clusters.   |  |  | | --- | --- | | Type: | np.ndarray | |  |  | | --- | | **fit**(*source\_data*) | | Performs clustering on the input data and returns cluster labels. |   Examples  **>>> import** **numpy** **as** **np** **>>> from** **hsip.clustering.clustering** **import** CosClust **>>>** data = np.random.rand(100, 50) *# 100 samples with 50 features each* **>>>** model = CosClust(threshold=0.8, verbose=**True**) **>>>** labels = model.fit(data) **>>>** print(labels) array([0, 1, 0, 2, ..., 1]) # Example output   |  | | --- | | **fit**(*source\_data: ndarray*) | | Performs clustering on the given data based on cosine similarity.   |  |  | | --- | --- | | Parameters: | **source\_data** (*np.ndarray*) – A 2D array of shape (n\_samples, n\_features) where n\_samples is the number of samples and n\_features is the number of features for each sample. | | Returns: | A 1D array of shape (n\_samples,) containing the cluster labels for each sample. | | Return type: | np.ndarray |   Notes  The algorithm proceeds in two steps:   1. Initial clustering based on the cosine similarity threshold. 2. Adjustment of labels based on the reference set of cluster representatives.  * Samples with no cluster assignment are labeled as -1. | |

|  |
| --- |
| *class* **hsip.clustering.clustering.HDBSCAN**(*\*\*kwargs*) |
| Bases: object  A wrapper class for the HDBSCAN clustering algorithm.  The HDBSCAN class simplifies the usage of the HDBSCAN clustering algorithm by encapsulating the core functionality of the hdbscan.HDBSCAN class. It provides a streamlined interface for clustering data and retrieving cluster labels.   |  | | --- | | **labels** | | Cluster labels assigned to each data point after fitting the model. Initially set to None.   |  |  | | --- | --- | | Type: | np.ndarray or None | |  |  | | --- | | **centroids** | | 2D array of shape (n\_clusters, n\_features) containing the centroids of the clusters.   |  |  | | --- | --- | | Type: | np.ndarray | |  |  | | --- | | **medoids** | | 2D array of shape (n\_clusters, n\_features) containing the medoids of the clusters.   |  |  | | --- | --- | | Type: | np.ndarray | |  |  | | --- | | **fit**(*source\_data: np.ndarray*) → np.ndarray | | Fits the HDBSCAN clustering model to the provided data and computes cluster labels. |  |  |  | | --- | --- | | Parameters: | * **min\_cluster\_size** (*int, optional*) – The minimum size of clusters. Default is 5. * **min\_samples** (*int, optional*) – The minimum number of samples in a neighborhood for a point to be considered a core point. * **cluster\_selection\_epsilon** (*float, optional*) – The distance threshold for cluster selection. Default is 0.0. |   Examples  **>>> import** **numpy** **as** **np** **>>> from** **hsip.clustering.clustering** **import** HDBSCAN **>>>** source\_data = np.random.rand(100, 5) **>>>** model = HDBSCAN(min\_cluster\_size=10) **>>>** labels = model.fit(source\_data) **>>>** print(labels) [0 0 1 -1 2 2 1 1 3 ...]   |  | | --- | | **fit**(*source\_data: ndarray*) | |

|  |
| --- |
| *class* **hsip.clustering.clustering.SCH**(*linkage\_method: str = 'complete'*, *linkage\_metric: str = 'cosine'*, *linkage\_optimal\_ordering: bool = False*, *fcluster\_t: float = 0.25*, *fcluster\_criterion: str = 'distance'*, *fcluster\_depth: int = 2*) |
| Bases: object  A class for performing hierarchical clustering using SciPy’s linkage and fcluster methods.  The SCH class provides an interface for hierarchical clustering with customizable linkage and clustering parameters.   |  | | --- | | **labels** | | Cluster labels assigned to each data point after fitting the model. Initially set to None.   |  |  | | --- | --- | | Type: | np.ndarray or None | |  |  | | --- | | **linkage\_method** | | Linkage method used for hierarchical clustering. Supported methods include “single”, “complete”, “average”, “weighted”, “centroid”, “median”, and “ward”.   |  |  | | --- | --- | | Type: | str | |  |  | | --- | | **linkage\_metric** | | Distance metric used to compute pairwise distances between data points. Common metrics include “euclidean”, “cosine”, “cityblock”, and “hamming”.   |  |  | | --- | --- | | Type: | str | |  |  | | --- | | **linkage\_optimal\_ordering** | | If True, the linkage matrix will be reordered to minimize the distances between successive leaves.   |  |  | | --- | --- | | Type: | bool | |  |  | | --- | | **fcluster\_t** | | The threshold to apply when forming flat clusters. The meaning of t depends on the fcluster\_criterion.   |  |  | | --- | --- | | Type: | float | |  |  | | --- | | **fcluster\_criterion** | | The criterion to use in forming flat clusters. Supported criteria include “inconsistent”, “distance”, and “maxclust”.   |  |  | | --- | --- | | Type: | str | |  |  | | --- | | **fcluster\_depth** | | The maximum depth to perform inconsistency calculation if fcluster\_criterion=”inconsistent”. Ignored for other criteria.   |  |  | | --- | --- | | Type: | int | |  |  | | --- | | **centroids** | | 2D array of shape (n\_clusters, n\_features) containing the centroids of the clusters.   |  |  | | --- | --- | | Type: | np.ndarray | |  |  | | --- | | **medoids** | | 2D array of shape (n\_clusters, n\_features) containing the medoids of the clusters.   |  |  | | --- | --- | | Type: | np.ndarray | |  |  | | --- | | **fit**(*source\_data: np.ndarray*) → np.ndarray | | Fits the hierarchical clustering model to the provided data and computes cluster labels. |  |  |  | | --- | --- | | Parameters: | * **linkage\_method** (*str, optional*) – Linkage method for clustering. Default is “complete”. * **linkage\_metric** (*str, optional*) – Distance metric for clustering. Default is “cosine”. * **linkage\_optimal\_ordering** (*bool, optional*) – Whether to reorder linkage matrix for optimal leaf ordering. Default is False. * **fcluster\_t** (*float, optional*) – Threshold for forming flat clusters. Default is 0.25. * **fcluster\_criterion** (*str, optional*) – Criterion for forming flat clusters. Default is “distance”. * **fcluster\_depth** (*int, optional*) – Depth for inconsistency calculation when fcluster\_criterion=”inconsistent”. Default is 2. |   Examples  **>>> import** **numpy** **as** **np** **>>> from** **hsip.clustering.clustering** **import** SCH **>>>** source\_data = np.random.rand(10, 5) **>>>** model = SCH() **>>>** labels = model.fit(source\_data) **>>>** print(labels) [1 1 2 2 3 3 4 4 5 5]   |  | | --- | | **fit**(*source\_data: ndarray*) | |

## Methods of processing hyperspectral data

Working with HSI often requires functionality for both pre- and post-processing of data.

### Processing methods:

|  |
| --- |
| **hsip.processing.processing.normalize**(*array: ndarray*) |
| Normalizes a given NumPy array to the range [0, 1]. The function scales the input array such that the minimum value in the array becomes 0 and the maximum value becomes 1. The formula used is:  formula   |  |  | | --- | --- | | Parameters: | **array** (*np.ndarray*) – A NumPy array to be normalized. The array can be of any shape or type that supports element-wise arithmetic operations. | | Returns: | A NumPy array of the same shape as the input, with values normalized to the range [0, 1]. | | Return type: | np.ndarray |   Examples  **>>> import** **numpy** **as** **np** **>>> from** **hsip.processing.processing** **import** normalize **>>>** array = np.array([1, 2, 3, 4, 5]) **>>>** normalized\_array = normalize(array) **>>>** print(normalized\_array) [0. 0.25 0.5 0.75 1. ] **>>>** array = np.array([[1, 2], [3, 4], [5, 6]]) **>>>** normalized\_array = normalize(array) **>>>** print(normalized\_array) [[0. 0.2 ]  [0.4 0.6 ]  [0.8 1. ]] |

|  |
| --- |
| **hsip.processing.processing.rayleigh\_scattering**(*spectral\_data: ndarray*, *inplace=False*, *verbose=True*) |
| Compute the Rayleigh scattering signature for a spectral dataset.   |  |  | | --- | --- | | Parameters: | * **spectral\_data** (*np.ndarray*) – A multi-dimensional NumPy array where the last dimension represents spectral bands. * **inplace** (*bool, default=False*) – If True, the function modifies spectral\_data by subtracting the computed Rayleigh signature. * **verbose** (*bool, verbose=True*) – If True, displays a progress bar. | | Returns: | A 1D array containing the Rayleigh scattering signature for each spectral band. | | Return type: | np.ndarray |   Examples  Apply a sigma filter to a 3D spectral dataset:  **>>> import** **numpy** **as** **np** **>>> from** **hsip.processing.processing** **import** rayleigh\_scattering **>>>** data = np.random.rand(100, 100, 10) \* 10 *# Example spectral data* **>>>** rayleigh\_signature = rayleigh\_scattering(data, inplace=**True**) **>>>** rayleigh\_signature *# Rayleigh scattering signature* array([...]) |

|  |
| --- |
| **hsip.processing.processing.sigma\_maximum\_filter**(*spectral\_data: ndarray*, *sigma: float = 3*, *thresholds: ndarray | None = None*) |
| Applies a sigma-based maximum filter to the input spectral data, capping values based on a calculated threshold of mean + sigma \* standard deviation.   |  |  | | --- | --- | | Parameters: | * **spectral\_data** (*np.ndarray*) – The input spectral data array. It is expected to have a shape where the last dimension corresponds to the spectral bands. * **sigma** (*float, optional*) – The multiplier for the standard deviation used in calculating the threshold, by default 3. * **thresholds** (*np.ndarray, optional*) – An array to store the calculated thresholds. If provided, it must have a shape matching the last dimension of spectral\_data. If None, thresholds are computed and returned internally, by default None. | | Returns: | The filtered spectral data with values capped by the calculated thresholds. | | Return type: | np.ndarray |   Examples  Apply a sigma filter to a 3D spectral dataset:  **>>> import** **numpy** **as** **np** **>>> from** **hsip.processing.processing** **import** sigma\_maximum\_filter **>>>** data = np.random.rand(100, 100, 10) \* 10 *# Example spectral data* **>>>** thresholds = np.zeros(data.shape[-1:], dtype=np.float32) **>>>** result\_with\_thresholds = sigma\_maximum\_filter(data, sigma=2, thresholds=thresholds) **>>>** thresholds *# Updated thresholds* array([...]) |

## Hyperspectral data reading tools

Hyperspectral data are distributed in different formats. Using these functions it is possible to read HSI in the following formats: SpyFile, ENVI, AVIRIS, ERDAS/Lan, Tif.

All readers except the TIF reader are wrappers for the opening functions from the [Spectral Python library](https://www.spectralpython.net/).

### Readers:

|  |
| --- |
| **hsip.reader.reader.open\_AVIRIS**(*path\_rfl: str*, *path\_spc: str*) |
| Opens a hyperspectral image in AVIRIS format.   |  |  | | --- | --- | | Parameters: | * **path\_rfl** (*str*) – Path to the AVIRIS reflectance data file (.rfl). * **path\_spc** (*str*) – Path to the AVIRIS spectral calibration file (.spc). | | Returns: | A SpyFile object containing the AVIRIS hyperspectral data. | | Return type: | SpectralLibrary |   Examples  **>>> from** **hsip.reader.reader** **import** open\_AVIRIS **>>>** hsi = open\_AVIRIS("example.rfl", "example.spc") **>>>** print(hsi) SpyFile: [shape=(100, 100, 224), dtype=float32] |

|  |
| --- |
| **hsip.reader.reader.open\_ENVI**(*path\_hdr: str*, *path\_img: str*) |
| Opens a hyperspectral image in ENVI format and converts it to a NumPy array.   |  |  | | --- | --- | | Parameters: | * **path\_hdr** (*str*) – Path to the ENVI header file (.hdr). * **path\_img** (*str*) – Path to the ENVI image file (.img). | | Returns: | A NumPy array of the hyperspectral data, with dtype=float. | | Return type: | np.ndarray |   Notes   * The ENVI image is opened as a memory-mapped array and then loaded fully into memory as a float array.   Examples  **>>> from** **hsip.reader.reader** **import** open\_ENVI **>>>** hsi = open\_ENVI("example.hdr", "example.img") **>>>** print(hsi.shape) (100, 100, 224) # Example dimensions |

|  |
| --- |
| **hsip.reader.reader.open\_ERDAS**(*path: str*) |
| Opens a hyperspectral image in ERDAS Imagine format.   |  |  | | --- | --- | | Parameters: | **path** (*str*) – Path to the ERDAS Imagine file to be opened. | | Returns: | A SpyFile object containing the hyperspectral data. | | Return type: | SpectralLibrary |   Examples  **>>> from** **hsip.reader.reader** **import** open\_ERDAS **>>>** hsi = open\_ERDAS("example.img") **>>>** print(hsi) SpyFile: [shape=(100, 100, 224), dtype=float32] |

|  |
| --- |
| **hsip.reader.reader.open\_SpyFile**(*path\_lan: str*) |
| Opens a hyperspectral image in LAN format.   |  |  | | --- | --- | | Parameters: | **path\_lan** (*str*) – Path to the LAN file to be opened. | | Returns: | A SpyFile object containing the hyperspectral data. | | Return type: | SpectralLibrary |   Examples  **>>> from** **hsip.reader.reader** **import** open\_SpyFile **>>>** hsi = open\_SpyFile("example.lan") **>>>** print(hsi) SpyFile: [shape=(100, 100, 224), dtype=float32] |

|  |
| --- |
| **hsip.reader.reader.open\_TIF**(*path\_tif: str*) |
| Opens a hyperspectral image in GeoTIFF format.   |  |  | | --- | --- | | Parameters: | **path\_tif** (*str*) – Path to the GeoTIFF file to be opened. | | Returns: | A NumPy array containing the hyperspectral data. | | Return type: | np.ndarray |   Notes   * The GeoTIFF data is loaded into memory using the tifffile library and copied as a NumPy array.   Examples  **>>>** >>> **from** **hsip.reader.reader** **import** open\_TIF **>>>** hsi = open\_TIF("example.tif") **>>>** print(hsi.shape) (100, 100, 224) # Example dimensions |

## Color image synthesis

Just like hyperspectral images, color images also consist of multiple channels. This functionality allows for the reconstruction of color images from HSI data with colors that closely resemble reality. Additionally, it enables the creation of color images composed of the most informative channels, which can be used for semantic clustering.

### Methods for RGB synthesis:

|  |
| --- |
| **hsip.rgb.rgb.hsi\_synthesize\_rgb**(*spectral\_data: ndarray*, *rgb\_bands: list | ndarray | None = None*, *wavelengths: list | ndarray | None = None*) |
| Synthesizes an RGB image from hyperspectral data.   |  |  | | --- | --- | | Parameters: | * **spectral\_data** (*np.ndarray*) – Hyperspectral image data, expected to be a 3D array of shape (height, width, bands). * **rgb\_bands** (*list of int, optional*) – A list containing the band indices for red, green, and blue channels, in that order. Must be of length 3. If provided, wavelengths is ignored. * **wavelengths** (*list of float, optional*) – A list containing the wavelengths corresponding to each band in spectral\_data. If provided, the closest wavelengths to 650 nm (red), 550 nm (green), and 450 nm (blue) are used. | | Returns: | A 3D array of shape (height, width, 3) representing the synthesized RGB image. | | Return type: | np.ndarray |   Examples  **>>>** *# Using band indices directly:* **>>> from** **hsip.rgb.rgb** **import** hsi\_synthesize\_rgb **>>>** spectral\_data = np.random.rand(100, 100, 224) *# Example hyperspectral data* **>>>** rgb\_bands = [50, 100, 150] *# Example red, green, blue bands* **>>>** rgb\_image = synthesize\_rgb(spectral\_data, rgb\_bands=rgb\_bands) **>>>** print(rgb\_image.shape) (100, 100, 3) **>>>** *# Using wavelengths:* **>>>** wavelengths = np.linspace(400, 700, 224) *# Example wavelength data* **>>>** rgb\_image = synthesize\_rgb(spectral\_data, wavelengths=wavelengths) **>>>** print(rgb\_image.shape) (100, 100, 3) |

|  |
| --- |
| **hsip.rgb.rgb.simple\_synthesize\_rgb**(*band\_data: list*, *sig\_max\_filt: float | None = None*) |
| Generate an RGB image from three hyperspectral bands.  This function synthesizes a simple RGB image from three provided hyperspectral bands, corresponding to the red, green, and blue channels. Optionally, it applies a sigma maximum filter to each band for noise reduction before normalization and scaling.   |  |  | | --- | --- | | Parameters: | * **band\_data** (*list of np.ndarray*) – A list containing exactly three 2D arrays, each representing one band of hyperspectral data for the red, green, and blue channels. Each array must have the same shape. * **sig\_max\_filt** (*float, optional*) – A sigma value to apply the sigma\_maximum\_filter function to each band. If provided, it is used for noise reduction before synthesizing the RGB image. | | Returns: | **rgb\_image** – A 3D array of shape (height, width, 3) representing the synthesized RGB image. The output is of type np.uint8 with pixel values scaled to the range [0, 255]. | | Return type: | np.ndarray |   Examples  Generate an RGB image with sigma maximum filtering  **>>> import** **numpy** **as** **np** **>>> from** **hsip.rgb.rgb** **import** simple\_synthesize\_rgb **>>>** band\_red = np.random.rand(100, 100) **>>>** band\_green = np.random.rand(100, 100) **>>>** band\_blue = np.random.rand(100, 100) **>>>** band\_data = [band\_red, band\_green, band\_blue] **>>>** rgb\_image = simple\_synthesize\_rgb(band\_data, sig\_max\_filt=3) **>>>** print(rgb\_image.shape) (100, 100, 3) |

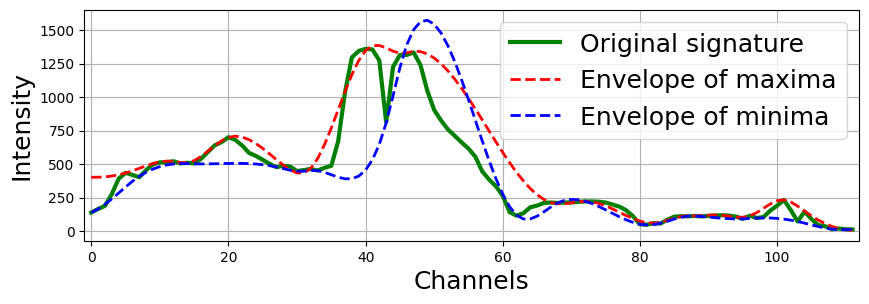
|  |
| --- |
| **hsip.rgb.labels.labels\_to\_rgb**(*labels: ndarray*, *RGB\_image: ndarray | None = None*) |
| Converts a label mask into an RGB image representation.  This function generates an RGB image where each unique label in the input labels array is assigned a specific color. If an RGB\_image is provided, the color for each label is determined by the mean color of the corresponding regions in the RGB\_image. If not, predefined colors from a global colors\_set are used. Warning: colors\_set - has exactly 17 colors in the set, if there are more unique labels, the colors for some will be repeated!   |  |  | | --- | --- | | Parameters: | * **labels** (*np.ndarray*) – A 2D or 3D array representing the label mask. Each unique value corresponds to a different class. * **RGB\_image** (*np.ndarray, optional*) – A 3D array of shape (height, width, 3) representing an existing RGB image. If provided, the color for each label is calculated as the mean RGB value of the regions corresponding to that label. | | Returns: | A 3D RGB image of shape (height, width, 3) with colors assigned to each label. | | Return type: | np.ndarray |   Notes   * If RGB\_image is not provided, the colors are assigned from a predefined colors\_set array. * This function is useful for visualizing label masks in a colorful, human-readable format.   Examples  Example 1: Using a predefined color set:  **>>> from** **hsip.rgb.labels** **import** labels\_to\_rgb **>>>** labels = np.array([[0, 0, 1], **...**  [1, 2, 2]]) **>>>** rgb\_image = labels\_to\_rgb(labels) **>>>** print(rgb\_image.shape) (2, 3, 3)  Example 2: Using an existing RGB image to compute colors:  **>>> from** **hsip.rgb.labels** **import** labels\_to\_rgb **>>>** labels = np.array([[0, 0, 1], **...**  [1, 2, 2]]) **>>>** RGB\_image = np.random.randint(0, 255, size=(2, 3, 3), dtype=np.uint8) **>>>** rgb\_image = labels\_to\_rgb(labels, RGB\_image=RGB\_image) **>>>** print(rgb\_image.shape) (2, 3, 3) |

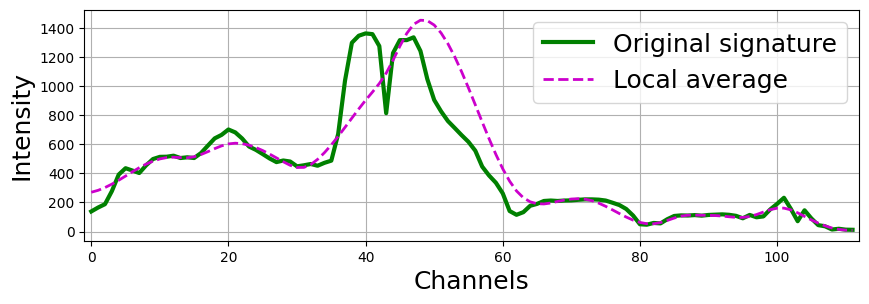
### Class with a predefined set of colors:

|  |
| --- |
| *class* **hsip.rgb.colors.Color** |
| Bases: object  A utility class for managing and accessing a predefined set of colors.  This class provides a collection of RGB color values in the range [0, 1], and methods to retrieve single or multiple colors by index or slice. It also supports modular indexing to wrap around the color set.   |  | | --- | | **colors** | | A 2D array of shape (n\_colors, 3) containing RGB color values in the range [0, 1].   |  |  | | --- | --- | | Type: | np.ndarray | |  |  | | --- | | **\_\_get\_color\_\_**(*index*) | | Retrieves a single color or a subset of colors by index or slice. |  |  | | --- | | **\_\_getitem\_\_**(*index*) | | Alias for \_\_get\_color\_\_, allows indexing with square brackets. |  |  | | --- | | **\_\_len\_\_**() | | Returns the total number of colors in the palette. |  |  | | --- | | **\_\_iter\_\_**() | | Returns an iterator over the colors. |  |  |  | | --- | --- | | Parameters: | **None** |   Examples  **>>>** *# Create an instance of the `Color` class and retrieve colors:* **>>> from** **hsip.rgb.colors** **import** Color **>>>** color\_palette = Color() **>>>** *# Access a single color using an index:* **>>>** color\_palette[0] array([0.90196078, 0.09803922, 0.29411765]) **>>>** *# Access multiple colors using a slice:* **>>>** color\_palette[1:4] array([[0.23529412, 0.70588235, 0.29411765],  [0. , 0.50980392, 0.78431373],  [0.96078431, 0.50980392, 0.18823529]]) **>>>** *# Modular indexing wraps around:* **>>>** color\_palette[20] array([0.23529412, 0.70588235, 0.29411765]) # Index 20 maps to 1 (20 % len(colors)) **>>>** *# Iterate through the colors:* **>>> for** color **in** color\_palette: **...**  print(color) array([0.90196078, 0.09803922, 0.29411765]) array([0.23529412, 0.70588235, 0.29411765]) ... |

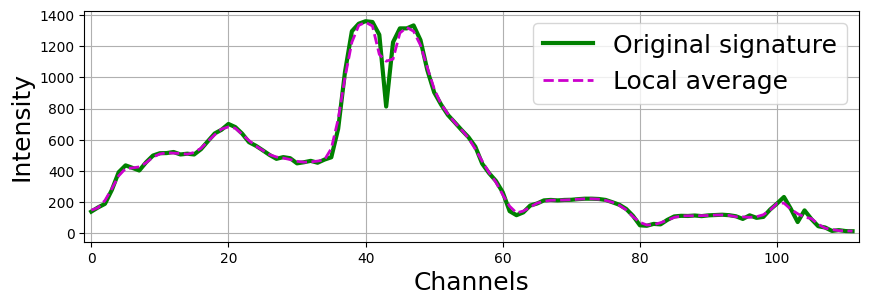
## Empirical mode decomposition (EMD)

Empirical Mode Decomposition (EMD) is a signal processing method used to decompose a signal into a set of intrinsic mode functions (IMFs), which are oscillatory components obtained adaptively from the data. The method iteratively identifies local extrema, constructs upper and lower envelopes using spline interpolation, and computes a local mean to extract IMFs. A significant drawback of EMD lies in the accuracy of defining the local mean and envelopes, particularly near the boundaries of the data, where end effects can lead to distortions. These boundary issues and the reliance on spline interpolation may result in inaccuracies in the extracted modes.





The 1D-EMD decomposition algorithm, adapted for hyperspectral image (HSI) analysis, incorporates several updates compared to the classical approach: 1. Precise calculation of the local mean using a moving average window instead of the arithmetic mean of the envelope maxima and minima of the signal. 2. Introduction of a rule for computing the local mean when the window extends beyond the range of spectral channels. 3. Flexibility to adapt the window size to scales smaller than the distance between adjacent zero-crossings in the current empirical mode (EM) or the initial window width. 4. Noise suppression in EMs by increasing the initial window size and applying it multiple times.



### Methods for decomposition into EMD

|  |
| --- |
| **hsip.swemd.swemd.SWEMD**(*data: ndarray*, *number\_of\_modes: int = 4*, *windows\_size: list = [3]*, *verbose: bool = True*) |
| Returns for a 1D signal its IMF and windows for each of them.   |  |  | | --- | --- | | Parameters: | * **x** (*np.ndarray*) – An array of dimension 3 (height \* width \* bands), dimension 2 (n\_samples \* bands), or just a single sample. * **number\_of\_modes** (*int, default=4*) – The number of IMFs to calculate for the input signal. * **windows\_size** (*list or tuple of int, default=3*) – The size of windows for each mode, starting with the first. If the list type is passed, then each element will indicate the size of the window starting with the first IMF. The list of passed sizes can be less than specified in number\_of\_modes, in which case subsequent sizes of sliding windows will be calculated automatically. The list can be passed a value equal to -1, in which case the window size will also be calculated automatically. If int is passed, this number will be the size of the sliding window only for the first IMF. | | Returns: | **IMFs** (*list*) – Empirical modes for each sample.**err\_windows\_size** (*list*) – Window sizes for each level of empirical modes. |   Examples  EMD calculation: window lengths equal three for the 1st and 2nd modes, 5 for the 3rd mode, and automatic for the rest.  **>>> import** **numpy** **as** **np** **>>> from** **hsip.swemd.swemd** **import** SWEMD **>>>** data = np.random.rand(1000, 1000, 100) \* 10 *# Example spectral data* **>>>** IMFs, windows = SWEMD(data, number\_of\_modes=8, windows\_size=[3, 3, 5]) **>>>** print(IMFs.shape, windows.shape) (8, 1000, 1000, 100), (8, 1000, 1000) |

|  |
| --- |
| **hsip.swemd.swemd.SWEMD\_signal**(*iSample: ndarray*, *number\_of\_modes: int = 4*, *windows\_size: list = [3]*) |
| Returns IMFs for each sample.   |  |  | | --- | --- | | Parameters: | * **x** (*np.ndarray*) – One-dimensional signal. * **number\_of\_modes** (*int, default=4*) – The number of IMFs to calculate for the input signal. * **windows\_size** (*list or tuple of int, default=3*) – The size of windows for each mode, starting with the first. If the list type is passed, then each element will indicate the size of the window starting with the first IMF. The list of passed sizes can be less than specified in number\_of\_modes, in which case subsequent sizes of sliding windows will be calculated automatically. The list can be passed a value equal to -1, in which case the window size will also be calculated automatically. If int is passed, this number will be the size of the sliding window only for the first IMF. | | Returns: | **IMFs** (*list*) – Empirical modes for each sample.**err\_windows\_size** (*list*) – Window sizes for each level of empirical modes. |   Examples  EMD calculation: window lengths equal three for the 1st and 2nd modes, 5 for the 3rd mode, and automatic for the rest.  **>>> import** **numpy** **as** **np** **>>> from** **hsip.swemd.swemd** **import** SWEMD\_signal **>>>** data = np.random.rand(100) \* 10 *# Example spectral data* **>>>** IMFs, windows = SWEMD(data, number\_of\_modes=8, windows\_size=[3, 3, 5]) **>>>** print(IMFs.shape, windows.shape) (100), (8) |