1 Python Implementation

This document contains the python implementation of the Locally Stationary Wavelet model of Nason et al. (2000) and the factor model of Koch (2015). This code was used in the thesis of Guillaume Lafontaine "*Multiscale Factor Modelling of High Dimensional Locally Stationary Time Series*".
All the files can be access through Github on : `https://github.com/GuillaumeLaf/LSWFactorModel`

1.0.1 Custom_wavelet.py

```
import numpy as np
import pywt
import wavelet_utils as utils
import matplotlib.pyplot as plt

class Wavelet:
    discritization:np.ndarray
    def __init__(self, name:str, maxScale:int):
        """

        The Wavelet Class is simply an interface for the PyWavelet Wavelet class.
        This interface allows simplifications among our environment.
        We could easily get rid of the dependence on the PyWavelet package by hard coding wavelet filters,
                                                                    ...

        Parameters
        ----------
        name : str
            Name of the wavelet. The name must be the same as in the PyWavelet Library.
        maxScale : int
            The maximum scale of the wavelet. Usually computed based on available amount of data.
        """

        self.name = name
        self.pywtWavelet = pywt.Wavelet(name)
        self.maxScale = maxScale
        self.maxLength = self.getWaveletLength(self.maxScale + 1)
        # Maximum wavelet length. Note that the maximum length is twice the actual maximum length of the
                                                            coarsest wavelet scale (to make sure the
                                                            convolution with fft works)

        # High and Low Decomposition and Reconstruction filters (used to decompose a signal with a particular
                                                             wavelet)
        self.dec_lo = np.array(self.pywtWavelet.dec_lo)
        self.dec_hi = np.array(self.pywtWavelet.dec_hi)
        self.rec_lo = np.array(self.pywtWavelet.rec_lo)
        self.rec_hi = np.array(self.pywtWavelet.rec_hi)

    def discretizeToMaxScale(self):
        """
        Function that allows to get a discretize version of the wavelet
        Returns
        -------
        This function does not return anything.
        However it fills for each scale the array 'discritization' with the discritize wavelets.
        The first dimension of the array is the scale from 1st scale to 'MaxScale'.
        The second dimension is of size 'maxLength' and contains the discrete wavelet first padded in the
        end by zeros (as the discritized wavelet have differents lengths for different scales).
        """
```

```
        self.__initializeDiscritizationArrayIfNot()
        for i in range(self.maxScale):
            waveletLength = self.getWaveletLength(i)
            self.discritization[i, :waveletLength] = self.discretizeOneScale(i)

    def discretizeOneScale(self, scale:int):
        """

        Parameters
        ----------
        scale : int
            The scale at which we want to have a discritized version of the wavelet.
        Returns
        -------
        An array filled with a discrete wavelet without any padding
        """

        mother = self.pywtWavelet.wavefun(level=scale+1)[1]/np.sqrt(2**(scale+1))
        return np.trim_zeros(np.array(mother))

    def getWaveletLength(self, scale:int, cross_scale:int=0):
        """

        Parameters
        ----------
        scale : int
            Scale of the wavelet.
        cross_scale : int, optional
            cross Scale of the wavelet. The default is 0.
        Returns
        -------
        The length of the wavelet at one particular scale and cross scale.
        """

        filter_length = len(self.pywtWavelet.dec_lo)
        return int((2**(scale+cross_scale+1) - 1)*(filter_length - 1) + 1)

    def __initializeDiscritizationArrayIfNot(self):
        if not utils.isArrayInitialized(self, 'discritization'):
            self.discritization = np.zeros((self.maxScale, self.maxLength), dtype=np.float64)

class CrossCorrelationWavelet(Wavelet):
    A_operator:np.ndarray
    columnOrderIndexing:np.ndarray  # This array allows us to easily know where the CCWF of scale 'j' and
                                                        order 'i' is in the 'phi_operator' array
    phi_operator:np.ndarray     # Array which stacks the CCWF vertically based on scale and order
    def __init__(self, name:str, maxScale:int, order:int):
        super().__init__(name, maxScale)
        self.order = order
        self.discretizeToMaxScale()
        self.initializeA_operator()

    def getA_operatorAtOrder(self, i:int, r:int, trimmed:bool):
        """
        This function returns the inverse of the operator 'A' for a given two given 'orders' -i.e. "i" and "r
                                                                ".
        This function uses the 'columnOrderIndexing' list to extract the wanted orders from the array '
                                                            A_operator'.
```

```
        Parameters
        ----------
        i : int
            first order.
        r : int
            second order.
        trimmed : bool
            If False, this function returns a square matrix of dimension 'maxScale'
            filled with the operator 'A' and zeros where it is needed.
        Returns
        -------
        out : TYPE
            Either a square matrix if the 'trimmed' option is False.
            Otherwise, it returns the operator 'A' for two given 'orders' -i.e. "i" and "r".
        """

        idx_i_mask = (np.concatenate(self.columnOrderIndexing) == i)
        idx_r_mask = (np.concatenate(self.columnOrderIndexing) == r)

        operator_mask = np.outer(idx_i_mask, idx_r_mask)
        shape_i = self.maxScale - np.abs(i)
        shape_r = self.maxScale - np.abs(r)

        if not trimmed:
            out = np.zeros((self.maxScale, self.maxScale), dtype=np.float64)
            out[:shape_i, :shape_r] = self.A_operator[operator_mask].reshape(shape_i, shape_r)
        else:
            out = self.A_operator[operator_mask].reshape(shape_i, shape_r)
        return out

    def initializeA_operator(self):
        """
        Initialize the A operator used in the correction of the Evolutionary Wavelet Spectrum
        This function compute the Gramian Matrix of the phi_operator.
        Then it deletes the extra columns and rows of the operator 'A'.
        Finally it inverts the operator 'A' as required for the correction.
        Returns
        -------
        None.
        """

        self.initializePhi_operator()
        self.A_operator = self.phi_operator.T @ self.phi_operator
        self.A_operator = np.linalg.inv(self.A_operator)


    def initializePhi_operator(self):
        """
        Initialize the phi operator required to get the correction matrix 'A'.
        This operator is basically an array of all the Cross Correlation Wavelet Functions stacked vertically
                                                                .
        Taking the inner product of that operator with itself gives a Gramian matrix.
        Returns
        -------
        This function does not return anything.
        """

        self.phi_operator = np.empty((self.maxLength, ), dtype=np.float64)
```

```
        # See the docstring of the function '__stackCCWFatScale' to understand the importance of this list.
        col_order_j = []
        for j in range(self.maxScale):
            col_order_i = []
            self.__stackCCWFatScale(j, col_order_i)
            col_order_j.append(np.array(col_order_i))

        # Delete the first columns since it was only usefull to get the first stack
        self.phi_operator = np.delete(self.phi_operator, 0, axis=1)

        # Save the 'col_order_j' list permanently in the object.
        # However, it will later be modified when erasing some duplicate columns of the Gramian Matrix.
        self.columnOrderIndexing = np.array(col_order_j)

    def __stackCCWFatScale(self, j:int, col_order:list):
        """
        Stack the Cross Correlation Wavelet Function of a particular scale 'j' vertically in an array '
                                                                    phi_operator' for the order of the CCWF.
        Parameters
        ----------
        j : int
            scale of the CCWF to be stacked.
        col_order : list
            List that saves the way the CCWF are stacked. This list contains the 'order' of the CCWF
                                                                associated with the scale 'j'.
            ex : col_order = [-2, -1, 0, 1, 2]
        Returns
        -------
        This function does not return anything.
        However it modifies the list 'col_order'. The list is passed by reference, not by value.
        """

        # Maximum and minimum order following Daniel Koch's notations
        mx = np.min([self.maxScale-j, self.order+1])
        mn = np.max([-j, -self.order])
        for i in range(mn, mx):
            if i >= 0:
                col_order.append(i)

                # The sign of the order is importance since the CCWF are not symmetric.
                # The CCWF with a negative order is the mirror around the y-axis of the positive order (for a
                                                                    given scale 'j')
                # Eventhough the 'negative is the mirror of the positive', the arrays are not mirror of each
                                                                other.
                self.phi_operator = np.column_stack((self.phi_operator, utils.fft_ConjugateConvolve(self.
                                                                discritization[j], self.discritization
                                                                [j+i])))
```

1.0.2 Evolutionary_Wavelet_Spectrum.py

```
import numpy as np
import pywt
import matplotlib.pyplot as plt
import custom_wavelets as w
import wavelet_utils as utils
import WaveletDecomposition as dec
import Smoother as smo
```

```
import numba as nb


plt.style.use('ggplot')

class EWS:
    spectrum:np.ndarray     #The spectrum does not include the approx. level (only the details coefficients)
    incrementsCorrelationMatrix:np.ndarray

    def __init__(self, decomposition:np.ndarray, isSpectrum:bool, order:int, wavelet:w.Wavelet):
        self.decomposition = decomposition
        self.isSpectrum = isSpectrum
        self.order = order
        self.crossWavelet = w.CrossCorrelationWavelet(wavelet.name, wavelet.maxScale, self.order)
        self.columnOrderIndexing = self.crossWavelet.columnOrderIndexing

        self.__getSpectrum()

    def updateDecomposition(self, decomposition:np.ndarray):
        """
        This function is usefull if we want to iterate over different decomposition without reinitializing
                                                                    the objects.
        I essentially use this function when checking for convergence of the EWS.

        Parameters
        ----------
        decomposition : np.ndarray
            New decomposition.

        Returns
        -------
        None.

        """

        self.decomposition = decomposition
        self.__getSpectrum()

    def setIncrementsCorrelationMatrix(self, correlationMatrix:np.ndarray):
        self.incrementsCorrelationMatrix = correlationMatrix
        self.__scaleSpectrumIfSimulation()

    def graph(self, order:int=0, sharey:bool=True):
        n_scales = np.where(np.concatenate(self.columnOrderIndexing) == order, 1, 0).sum()
        fig, ax = plt.subplots(n_scales, 1, figsize=(12, 15), sharex=True, sharey=sharey)
        ax = np.ravel(ax)
        for j in range(n_scales):
            ax[j].plot(self.getSpectrumOfScaleAndOrder(j, order))
            ax[j].set_ylabel(f'Scale -{j+1}')

    def correctSpectrum(self):
        """
        This function corrects the spectrum with the 'A_operator' computed in the 'custom_wavelet.py' module.

        Returns
        -------
        None.

        """
```

```
        temp_spectrum = np.zeros_like(self.spectrum)
        idx_i = np.concatenate(self.columnOrderIndexing)

        # Iterate over all possible/unique 'orders'.
        for i in set(idx_i):
            # Select the indices of the flattened array 'columnOrderIndexing' where the order is "i".
            idx = np.arange(len(idx_i))[idx_i == i]
            temp_spectrum[idx, :] = self.__correctSpectrumOfOrder(i, idx)
        self.spectrum = temp_spectrum

    def __correctSpectrumOfOrder(self, i:int, idx:np.ndarray):
        """
        This function corrects the spectrum of a particular order -i.e. "i".

        Parameters
        ----------
        i : int
            Order of the spectrum.
        idx : np.ndarray
            Array of indices of the flattened array 'columnOrderIndexing' where the order is "i".

        Returns
        -------
        correctedSpectrum : np.ndarray
            The corrected spectum of order "i".

        """

        correctedSpectrum = np.zeros((len(idx), self.spectrum.shape[1]), dtype=np.float64)
        for r in set(np.concatenate(self.columnOrderIndexing)):
            correctedSpectrum += self.crossWavelet.getA_operatorAtOrder(i, r, trimmed=True) @ self.
                                                                    getSpectrumOfOrder(r)
        return correctedSpectrum

    def smoothSpectrum(self, smoother:smo.Smoother):
        """
        This function smooths the spectrum according to the specified smoother.

        Parameters
        ----------
        smoother : smo.Smoother
            smoother used to smooth the spectrum.

        Returns
        -------
        None.

        """

        for i in range(self.spectrum.shape[0]):
            self.spectrum[i, :] = smoother.smooth(self.spectrum[i, :])

    def getSpectrumOfScaleAndOrder(self, j:int, i:int):
        """
        This function extracts the spectrum at a given scale "j" and order "i".

        Parameters
        ----------
```

```
        j : int
            Scale.
        i : int
            Order.

        Returns
        -------
        np.ndarray
            Spectrum of scale "j" and order "i".

        """

        orders = np.concatenate(self.columnOrderIndexing)
        mask = orders == i
        idx_mask = np.arange(orders.shape[0])
        idx_mask = idx_mask[mask]
        idx_j = idx_mask[j]
        return self.spectrum[idx_j]

    def getSpectrumOfOrder(self, i:int):
        """
        Extract the spectrum of order "i" given the flattened array 'columnOrderIndexing'.

        Parameters
        ----------
        i : int
            Order to be extracted.

        Returns
        -------
        np.ndarray
            Return the spectrum of the given order "i".

        """

        orders = np.concatenate(self.columnOrderIndexing)
        mask = orders == i
        return self.spectrum[mask, :].reshape(-1, self.spectrum.shape[1])

    def __getSpectrum(self):
        """
        Function that initialize the spectrum given the decomposition.
        If 'self.decomposition' is already a spectrum (as indicated by 'self.isSpectrum')
        then we must that the square root of it.

        Note that the 'spectrum' array is of dimension (length of columnOrderIndexing, length of signal)
        The 'spectrum' array is thus not necessarily in a logical order.
        We must once again use the 'columnOrderIndexing' to find the right scale and order.

        Returns
        -------
        None.

        """

        self.__initializeSpectrumArrayIfNot()
        if self.isSpectrum:
            self.__getCrossSpectrum(np.sqrt(self.decomposition))
        else:
```

```
            self.__getCrossSpectrum(self.decomposition)

    def __getCrossSpectrum(self, decomp:np.ndarray):
        """
        Function that sets the spectrum given the decomposition given as parameter.
        The spectrum is filled in accordance with the array 'columnOrderIndexing'.

        Note that even if negative orders are allowed they produce the same spectrum as positive orders,
                                                                hence creating storing redundant information.
        However, let the redundancy be, as to be more consistent with the spectrum of the CrossEWS class.

        Parameters
        ----------
        decomp : np.ndarray
            Decomposition used to create the spectrum.

        Returns
        -------
        None.

        """

        counter = 0
        for idx_scale, scale in enumerate(self.columnOrderIndexing):
             for order in scale:
                 self.spectrum[counter, :] = decomp[idx_scale, :] * decomp[idx_scale + order, :]
                 counter += 1

    def __getIncrementsCorrelationScalingSpectrum(self):
        return np.array([[self.incrementsCorrelationMatrix[j, j+i, z]
                          for j, orders in enumerate(self.columnOrderIndexing)
                          for i in orders]
                         for z in range(self.decomposition.shape[1])]).T

    def __scaleSpectrumIfSimulation(self):
        if self.__isSimulation():
            scalingSpectrum = self.__getIncrementsCorrelationScalingSpectrum()
            self.spectrum = np.multiply(self.spectrum, scalingSpectrum)

    def __initializeSpectrumArrayIfNot(self):
            if not utils.isArrayInitialized(self, 'spectrum'):
                self.spectrum = np.zeros((np.concatenate(self.columnOrderIndexing).shape[0], self.
                                                                    decomposition.shape[1]), dtype=np.
                                                                    float64)


    def __isSimulation(self):
        return utils.isArrayInitialized(self, 'incrementsCorrelationMatrix')    # If it a simulation then we
                                                                need to have a correlation matrix. Allow us to
                                                                 differentiate between simulation and real
                                                                decomposition of a signal

class CrossEWS:
    spectrum:np.ndarray     #The spectrum does not include the approx. level (only the details coefficients)
    incrementsCorrelationMatrix:np.ndarray

    def __init__(self, decomposition:np.ndarray, isSpectrum:bool, order:int, wavelet:w.Wavelet):
        """
        This class is really a one to one correspondence with the EWS class.
        It is only adapted to the multivariate case of the LSW model.
```

```
        Parameters
        ----------
        decomposition : np.ndarray
            Wavelet Decomposition. Size of the array is : (n_signals, n_details, length_signals)
        isSpectrum : bool
            Flag if the decomposition array is already a spectrum. This is used when doing some simulations
        order : int
            Order of the LSW model.
        wavelet : w.Wavelet
            Wavelet used to decompose the signal. This is an instance of the Wavelet class

        Returns
        -------
        None.

        """

        self.decomposition = decomposition      # (n_signals, n_details, lenght_signal)
        self.isSpectrum = isSpectrum
        self.order = order
        self.crossWavelet = w.CrossCorrelationWavelet(wavelet.name, wavelet.maxScale, self.order)
        self.columnOrderIndexing = self.crossWavelet.columnOrderIndexing

        self.__getSpectrum()

    def updateDecomposition(self, decomposition:np.ndarray):
        self.decomposition = decomposition
        self.__getSpectrum()

    def graph(self, u:int, v:int, order:int=0, sharey:bool=True):
        n_scales = np.where(np.concatenate(self.columnOrderIndexing) == order, 1, 0).sum()
        fig, ax = plt.subplots(n_scales, 1, figsize=(12, 15), sharex=True, sharey=True)
        ax = np.ravel(ax)
        for j in range(n_scales):
            ax[j].plot(self.getSpectrumOfScaleAndOrder(u, v, j, order))
            ax[j].set_ylabel(f'Scale -{j+1}')

    def smoothSpectrum(self, smoother:smo.Smoother):
        """
        Smooth the spectrum with the provided Smoother

        Parameters
        ----------
        smoother : smo.Smoother
            Smoother object.

        Returns
        -------
        None.

        """

        for u in range(self.spectrum.shape[0]):
            for v in range(self.spectrum.shape[0]):
                for i in range(self.spectrum.shape[2]):
                        self.spectrum[u, v, i, :] = smoother.smooth(self.spectrum[u, v, i, :])

    def correctSpectrum(self):
```

```
        """
        Correct the spectrum at every scale and order.

        Returns
        -------
        None.

        """
        idx_i = np.concatenate(self.columnOrderIndexing)
        for u in range(self.spectrum.shape[0]):
            for v in range(self.spectrum.shape[0]):
                temp_spectrum = np.zeros_like(self.spectrum[u, v])
                for i in set(idx_i):
                    idx = np.arange(len(idx_i))[idx_i == i]
                    temp_spectrum[idx, :] = self.__correctSpectrumOfOrder(u, v, i, idx)
                self.spectrum[u, v] = temp_spectrum

    def __correctSpectrumOfOrder(self, u:int, v:int, i:int, idx:np.ndarray):
        """
        Correct the spectrum at a given order "i" and between two processes "u" and "v".

        Parameters
        ----------
        u : int
            First process.
        v : int
            Second process.
        i : int
            order that will be corrected.
        idx : np.ndarray
            Index array containing the location in the spectrum array of the EWS between "u" and "v" at order
                                                                                    "i".

        Returns
        -------
        correctedSpectrum : TYPE
            Corrected EWS between "u" and "v" at order "i".

        """

        correctedSpectrum = np.zeros((len(idx), self.spectrum.shape[3]), dtype=np.float64)
        for r in set(np.concatenate(self.columnOrderIndexing)):
            correctedSpectrum += self.crossWavelet.getA_operatorAtOrder(i, r, trimmed=True) @ self.
                                                                    getSpectrumOfOrder(u, v, r)
        return correctedSpectrum

    def getSpectrumOfScaleAndOrder(self, u:int, v:int, j:int, i:int):
        """
        Function that returns the EWS between two processes "u" and "v" for a given scale "j" and order "i".

        Parameters
        ----------
        u : int
            First process.
        v : int
            Second process.
        j : int
            Scale.
        i : int
```

```
        Order.

    Returns
    -------
    np.ndarray
        Return the EWS.

    """
    orders = np.concatenate(self.columnOrderIndexing)
    mask = orders == i
    idx_mask = np.arange(orders.shape[0])
    idx_mask = idx_mask[mask]
    idx_j = idx_mask[j]
    return self.spectrum[u, v, idx_j]

def getSpectrumOfOrder(self, u:int, v:int, i:int):
    """
    Function that returns the EWS between two processes "u" and "v" for a given order "i".

    Parameters
    ----------
    u : int
        First process.
    v : int
        Second process.
    i : int
        Order.

    Returns
    -------
    np.ndarray
        Return the EWS.

    """

    orders = np.concatenate(self.columnOrderIndexing)
    mask = orders == i
    return self.spectrum[u, v, mask].reshape(-1, self.spectrum.shape[3])

def getSpectrumForAllSignalsAtTimeZ(self, j:int, i:int, z:int):
    """
    Function that allows us to recover the EWS for all processes at a given scale "j", order "i" and time
                                                                "z".
    This function is used when constructing the huge "S" stacked matrix (analogue to the covariance
                                                            matrix).

    Parameters
    ----------
    j : int
        Scale.
    i : int
        Order.
    z : int
        Time.

    Returns
    -------
    TYPE
        EWS for all processes at a given scale "j", order "i" and time "z".
```

```
        """

        orders = np.concatenate(self.columnOrderIndexing)
        idx = np.arange(len(orders))[orders == i]
        return self.spectrum[:, :, idx[j], z]

    def __getSpectrum(self):
        """
        The series of function attached to this one are used to compute the EWS from the processes'
                                                                    decompositions.

        Returns
        -------
        None.

        """

        self.__initializeSpectrumArrayIfNot()
        if self.isSpectrum:
            self.__getCrossSpectrum(np.sqrt(self.decomposition))
        else:
            self.__getCrossSpectrum(self.decomposition)

    def __getCrossSpectrum(self, decomp:np.ndarray):
        for u in range(self.spectrum.shape[0]):
            for v in range(self.spectrum.shape[0]):
                self.__getCrossSpectrumBetweenTS(u, v, decomp)

    def __getCrossSpectrumBetweenTS(self, u:int, v:int, decomp:np.ndarray):
        counter = 0
        for idx_scale, scale in enumerate(self.columnOrderIndexing):
            for order in scale:
                # if order >= 0:
                self.spectrum[u, v, counter] = decomp[u, idx_scale, :] * decomp[v, idx_scale + order, :]
                # else:
                #     self.spectrum[u, v, counter] = decomp[v, idx_scale, :] * decomp[u, idx_scale + order,
                                                                                      :]
                counter += 1

    def __initializeSpectrumArrayIfNot(self):
            if not utils.isArrayInitialized(self, 'spectrum'):
                self.spectrum = np.zeros((self.decomposition.shape[0], self.decomposition.shape[0], np.
                                          concatenate(self.columnOrderIndexing).
                                          shape[0], self.decomposition.shape[2])
                                          , dtype=np.float64)
```

1.0.3 Smoother.py

```
import numpy as np
import matplotlib.pyplot as plt
import custom_wavelets as w
import WaveletDecomposition as dec
import wavelet_utils as utils

class Smoother:
    def __init__(self):
        """
```

```
        Interface class used to aggregate multiple smoothing methods.
        This class will make adding new type of smoothing method easier.

        Returns
        -------
        None.

        """

        pass

    def smooth(self, signal:np.ndarray):
        pass

class Kernel_smoother(Smoother):
    def __init__(self, name:str, window:int):
        """
        Smoother Object based on a kernel.

        Parameters
        ----------
        name : str
            Name of the kernel.
        window : int
            Window of the kernel.

        Returns
        -------
        None.

        """

        self.name = name
        self.window = window
        self.kernel = np.empty((window, ), dtype=np.float64)
        self.kernelFunction = np.vectorize(self.__getKernelFunctionFromName())
        self.discritize()

    def discritize(self):
        """
        Discretize the continuous kernel.

        Returns
        -------
        None.

        """

        grid = np.linspace(-1, 1, self.window)
        self.kernel = self.kernelFunction(grid)
        self.kernel = self.kernel / np.sum(self.kernel)

    def smooth(self, signal:np.ndarray):
        """
        Smooth the signal provided

        Parameters
        ----------
        signal : np.ndarray
```

```
            Signal that needs to be smoothed.

        Returns
        -------
        np.ndarray
            Smoothed signal.

        """
        return utils.fft_convolve(signal, self.kernel)

    def graph(self):
        """
        Graph the kernel coefficients

        Returns
        -------
        None.

        """

        fig, ax = plt.subplots(1,1, figsize=(12, 8))
        ax.plot(self.kernel, 'o')
        ax.axhline(y=0)

    def __getKernelFunctionFromName(self):
        """
        Function that allows to get the kernel equation with its name.

        Returns
        -------
        Func
            Function of the kernel.

        """

        return {
            'Simple': lambda x: 1.,
            'Triangular': lambda x: (1.- np.abs(x)),
            'Epanechnikov': lambda x: 3./4. * (1 - x*x),
            'Gaussian': lambda x: 3./np.sqrt(2*np.pi) * np.exp(-4.5*x*x),
            'Silverman': lambda x: 2.5 * np.exp(-np.abs(5*x)/np.sqrt(2)) * np.sin(np.abs(5*x)/np.sqrt(2) + np
                                                                                  .pi/4.)
            }.get(self.name, lambda x:1)

class SWT_smoother(Smoother):
    decomposition:dec.WaveletDecomposition
    def __init__(self, wavelet:w.Wavelet, smooth_type:str):
        """
        This smoother is based on the de-noising wavelet method of Donoho (1995) "De-noising by soft-
                                                                thresholding"

        Parameters
        ----------
        wavelet : w.Wavelet
            Wavelet object that will be used to smooth.
        smooth_type : str
            smoohting type : soft - hard

        Returns
```

```
        -------
        None.

        """

        self.wavelet = wavelet
        self.thresholder = Threshold(smooth_type)
        self.thresh_func = np.vectorize(self.__thresholdDetail, signature='(n)->(n)')

    def smooth(self, signal:np.ndarray):
        """
        Smooth the provided signal.
        Note that the stationary wavelet transform rotates each scale of the decomposition.
        To recover the smoothed signal we therefore need to rotate the decomposition backwards beginning from
                                                                    the end.
        We also assume the first 3 scales to only be noise. Thus, we discard them completely -i.e. setting
                                                                  the wavelet coefficients to zero.
        The signal also need to be a multiple of 2, hence we add zeros at the end of the array if nedded.
        Finally we apply the thresholding method (soft, hard).

        Parameters
        ----------
        signal : np.ndarray
            Signal that needs to be smoothed.

        Returns
        -------
        np.ndarray
            Smoothed Signal.

        """

        signal, n_extend = self.__matchSignalLengthToPower2(signal)
        self.decomposition = dec.WaveletDecomposition(signal, self.wavelet)
        self.decomposition.rotateDecomposition()
        self.decomposition.details[:3] = 0.0
        self.decomposition.details[3:] = self.thresh_func(self.decomposition.details[3:])

        if n_extend == 0:
            return self.decomposition.SWTReconstruct()
        else:
            return self.decomposition.SWTReconstruct()[:-n_extend]

    def __thresholdDetail(self, detail:np.ndarray):
        return self.thresholder.threshold(detail)

    def __matchSignalLengthToPower2(self, signal:np.ndarray):
        # This function is self explanatory
        i = 0
        while len(signal) > 2**i:
            i += 1
        n_extend = 2**i - len(signal)
        zeros = np.zeros((n_extend,), dtype=np.float64)
        return np.concatenate([signal, zeros]), n_extend

class Threshold:
    thresh:np.float64
    func:None
    def __init__(self, name:str):
```

```
        """
        Class containing the thresholding methods.
        This class is once again used to facilitate adding new types of thresholding methods.

        Parameters
        ----------
        name : str
            Name of the thresholding method.

        Returns
        -------
        None.

        """

        self.name = name

    def threshold(self, signal:np.ndarray):
        self.__initializeThreshAndFuncIfNot(signal)
        return self.func(signal)

    def __getUniversalThresholdValue(self, signal:np.ndarray):
        """
        Function that computes the universal threshold value for the provided array

        Parameters
        ----------
        signal : np.ndarray
            Array.

        Returns
        -------
        TYPE
            Universal Threshold.

        """

        return np.std(signal) * np.sqrt(2. * np.log(len(signal)))

    def __getThresholdFunctionFromName(self):
        """
        Function that allows to retrieve easily the tresholding method from its name.
        Note that the function returned here only apply to a single scalar and will have to be vectorized
                                                                    later.

        Returns
        -------
        Function
            Tresholding function.

        """

        return {
            'hard':self.__hardThresholdingFunction,
            'soft':self.__softThresholdingFunction
            }.get(self.name, self.__softThresholdingFunction)

    def __hardThresholdingFunction(self, x):
        """
```

```
        Hard tresholding function.

        Parameters
        ----------
        x : TYPE
            Scalar on which to apply the treshold.

        Returns
        -------
        TYPE
            Tresholded input.

        """

        if np.abs(x)  > self.thresh:
            return x
        else:
            return 0.

    def __softThresholdingFunction(self, x):
        """
        Soft tresholding function.

        Parameters
        ----------
        x : TYPE
            Scalar on which to apply the treshold.

        Returns
        -------
        TYPE
            Tresholded input.

        """

        if np.abs(x) - self.thresh >= 0.:
            return np.sign(x) * (np.abs(x) - self.thresh)
        else:
            return 0.

    def __initializeThreshAndFuncIfNot(self, signal:np.ndarray):
        if not utils.isArrayInitialized(self, 'thresh'):
            self.thresh = self.__getUniversalThresholdValue(signal)
            self.func = np.vectorize(self.__getThresholdFunctionFromName())
```

1.0.4 WaveletDecomposition.py

```
import numpy as np
import pywt
import custom_wavelets as w
import wavelet_utils as utils
import matplotlib.pyplot as plt

class WaveletDecomposition:
    details:np.ndarray
    approx:np.ndarray

    def __init__(self, signal:np.ndarray, wavelet:w.Wavelet):
```

```
        """
        This class is used to structure all the functions and information needed for a wavelet decomposition.
        Currently this class only contains the stationary wavelet transform but could be easily extended to
                                                                    other wavelet transforms.

        Parameters
        ----------
        signal : np.ndarray
            Signal that needs to be decomposed.
        wavelet : w.Wavelet
            Wavelet object used for the decomposition.

        Returns
        -------
        None.

        """

        self.signal = signal
        self.wavelet = wavelet

        self.SWTdecompose(norm=False)

    # First scale at the beginning of array
    def SWTdecompose(self, norm:bool):
        """
        This function decomposes the signal with the Stationary Wavelet Transform.
        Note that we can decompose array of arbitrary length, not just multiples of 2.

        Parameters
        ----------
        norm : bool
            Indicates whether the decomposition will be normalized.
            If True, compute the MODWT.
            If False, compute the SWT.

        Returns
        -------
        None.

        """

        self.__initializeDetailsAndApproxArrayIfNot()
        low_pass = self.wavelet.dec_lo
        high_pass = self.wavelet.dec_hi

        approx = np.copy(self.signal)
        for i in range(self.wavelet.maxScale):
            details = utils.fft_convolve(approx, high_pass)/self.__getNormalizeConstant(norm)
            approx = utils.fft_convolve(approx, low_pass)/self.__getNormalizeConstant(norm)

            details = np.roll(details, -(2**i))
            approx = np.roll(approx, -(2**i))

            low_pass = utils.upsample(low_pass)
            high_pass = utils.upsample(high_pass)

            self.details[i] = details
```

```
        self.approx = approx

    def SWTReconstruct(self):
        """
        Reconstruct the signal given the decomposition stored in 'details' and 'approx' array.
        Like the decompostion function 'SWTdecompose', this function supports arbitrary length signals.

        Returns
        -------
        reconstruction : TYPE
             The reconstructed signal.

        """

        for i in range(self.wavelet.maxScale):
            if i == 0:
                reconstruction = np.fft.fft(self.approx)
            else:
                reconstruction = np.fft.fft(reconstruction)
            rec_lo = self.wavelet.rec_lo
            rec_hi = self.wavelet.rec_hi

            for j in range(self.wavelet.maxScale-1-i):
                rec_lo = utils.upsample(rec_lo)
                rec_hi = utils.upsample(rec_hi)

            rec_lo = utils.adjustSecondArraySizeToFirst(self.signal, rec_lo)
            rec_hi = utils.adjustSecondArraySizeToFirst(self.signal, rec_hi)

            rec_lo = np.fft.fft(rec_lo)
            rec_hi = np.fft.fft(rec_hi)

            d = self.details[self.wavelet.maxScale - i - 1]
            d = np.fft.fft(d)

            convolutionApprox = reconstruction * rec_lo
            convolutionDetail = d * rec_hi
            convolutionApprox = np.real(np.fft.ifft(convolutionApprox))
            convolutionDetail = np.real(np.fft.ifft(convolutionDetail))

            reconstruction = (convolutionApprox + convolutionDetail)/2.
        return reconstruction

    def rotateDecomposition(self):
        """
        This function rotates the approximate and detail coefficients array since it was rotated during the
                                                                decomposition

        Returns
        -------
        None.

        """

        for i in range(self.wavelet.maxScale-1, -1, -1):
            if i == self.wavelet.maxScale-1:
                self.approx = np.roll(self.approx, -(2**(i+2)))
            self.details[i] = np.roll(self.details[i], -(2**(i+2)))
```

```
    def __getNormalizeConstant(self, norm:bool):
        """
        This function return the normalizing constant. In order words, if we want to get the MODWT "norm"
                                                                argument should be true.

        Parameters
        ----------
        norm : bool
            DESCRIPTION.

        Returns
        -------
        TYPE
            DESCRIPTION.

        """

        if norm:
            return np.sqrt(2)
        else:
            return 1.0

    def __initializeDetailsAndApproxArrayIfNot(self):
        if not utils.isArrayInitialized(self, 'details'):
            self.details = np.zeros((self.wavelet.maxScale, len(self.signal)), dtype=np.float64)
            self.approx = np.zeros((1, len(self.signal)), dtype=np.float64)
```

wavelet_utils.py

```
import numpy as np
import matplotlib.pyplot as plt
import numba as nb

"""
    This file contains a handful of functions that are used throughout this repository.
"""

def np_convolve(in1:np.ndarray, in2:np.ndarray):
    return np.convolve(in1, in2)

def fft_convolve(in1:np.ndarray, in2:np.ndarray):
    """
    Compute the convolution of two arrays with the Fast Fourier Transform.

    Parameters
    ----------
    in1 : np.ndarray
        First array.
    in2 : np.ndarray
        Second array. This array is supposed to be a filter

    Returns
    -------
    TYPE
        Convolution of 'in1' and 'in2'.

    """

    # This function allows to also convolve a filter 'in2' with an array 'in1'
    in2 = adjustSecondArraySizeToFirst(in1, in2)
```

```
    in1_fft = np.fft.fft(in1)
    in2_fft = np.fft.fft(in2)
    convolution = in1_fft * in2_fft
    convolution = np.fft.ifft(convolution)
    return np.real(convolution)

def fft_ConjugateConvolve(in1:np.ndarray, in2:np.ndarray):
    """
    Compute the cross-correlation of two array with the Fast Fourier Transform.
    Unlike the convolution, we take the conjugate of the filter 'in2'.

    Parameters
    ----------
    in1 : np.ndarray
        First array.
    in2 : np.ndarray
        Second array. This array is supposed to be a filter.

    Returns
    -------
    TYPE
        Cross-correlation between 'in1' and 'in2'.

    """

    in2 = adjustSecondArraySizeToFirst(in1, in2)

    in1_fft = np.fft.fft(in1)
    in2_fft = np.fft.fft(in2)
    convolution = in1_fft * np.conjugate(in2_fft)
    convolution = np.fft.ifft(convolution)
    return np.real(convolution)

def adjustSecondArraySizeToFirst(in1:np.ndarray, in2:np.ndarray):
    if in1.size > in2.size:
        in2 = np.append(in2, np.repeat(0, in1.size-in2.size))
    return in2

def adjustFirstArraySizeToSecond(in1:np.ndarray, in2:np.ndarray):
    if in1.size < in2.size:
        in1 = np.append(in1, np.repeat(0, in2.size-in1.size))
    return in2

def isArrayInitialized(obj, name):
    return hasattr(obj, name)


"""
    The following functions could have been placed inside their corresponding classes.
    However, in order to speed up the computation using Numba, we have to extract them from the class
                                                                environment.
    The 'nb.njit' decorator allows to get speed of the function comparable to C.
"""

@nb.njit()
def upsample(arr):
    """
    Upsample the given array by placing zeros between each elements.
```

```
    Parameters
    ----------
    arr : TYPE
        Array that needs to be upsampled.

    Returns
    -------
    upsampledArray : np.ndarray

    """

    upsampledArray = np.zeros((arr.size*2,), dtype=np.float64)
    upsampledArray[0::2] = np.copy(arr)
    return upsampledArray

@nb.njit(nogil=True)
def rollMatrixByRow(matrix:np.ndarray):
    """
    This function allows to roll an entire matrix by row.

    Parameters
    ----------
    matrix : np.ndarray
        DESCRIPTION.

    Returns
    -------
    matrix : np.ndarray

    """

    for i in range(matrix.shape[0]):
        matrix[i, :] = np.roll(matrix[i, :], -i)
    return matrix

@nb.njit(nogil=True, fastmath=True)
def initializeA_operatorOfOrders(crossCorrelationScaleI:np.ndarray, crossCorrelationScaleR:np.ndarray,
                                                        maxScale:int):
    """
    Initialize the correction matrix 'A' between scales "i" and "r".

    Parameters
    ----------
    crossCorrelationScaleI : np.ndarray
        Cross correlation array at scale 'i'.
    crossCorrelationScaleR : np.ndarray
        Cross correlation array at scale 'r'.
    maxScale : int
        maximum scale.

    Returns
    -------
    out : TYPE
        correction matrix 'A.

    """

    out = np.zeros((maxScale, maxScale), dtype=np.float64)
```

```
    for j in range(maxScale):
        for l in range(maxScale):
            out[j, l] = crossCorrelationScaleI[j] @ crossCorrelationScaleR[l]
    return out
```

LSW_model.py

```
import numpy as np
import pywt
import matplotlib.pyplot as plt
import wavelet_utils as utils
import custom_wavelets as w
import WaveletDecomposition as dec
import Evolutionary_Wavelet_Spectrum as ews
import numba as nb

class LSW:
    simulation:np.ndarray
    isSignalASpectrum:bool
    maxScale:int
    decomposition:dec.WaveletDecomposition
    evol_spectrum:ews.EWS
    incrementsCorrelationMatrix:np.ndarray
    lengthSignal:int

    def __init__(self, signal:np.ndarray, wavelet_name:str, order:int=0):
        """
        This class contains everything related to the Locally Stationary Wavelet model of Nason and al. (2000
                                                                ).


        Parameters
        ----------
        signal : np.ndarray
            Signal that is assumed to follow a LSW process.
            The signal could be a spectrum since it is possible to simulate a process given a particular
                                                                Evolutionary Wavelet Spectrum.
        wavelet_name : str
            Name of the wavelet used to model the process.
        order : int, optional
            Order of the model according to Koch (2015). The default is 0.

        Returns
        -------
        None.

        """

        self.signal = signal
        self.order = order
        self.wavelet_name = wavelet_name

        self.__isSignalASpectrum()
        self.__computeMaxScale()

        self.wavelet = w.Wavelet(wavelet_name, self.maxScale)

        self.__initializeDecomposition()

    def getConstantIncrementsCorrelationRows(self, firstRow:np.ndarray):
        """
```

```
        Get a constant correlation matrix for the increments of the process.

        Parameters
        ----------
        firstRow : np.ndarray
            First row of the correlation matrix.
            Note that the correlation matrix will be adapted in order to have a semi-positive matrix.

        Returns
        -------
        np.ndarray
            return the repeated row for the length of the signal.

        """

        firstRow = firstRow[np.newaxis, :]
        return np.repeat(firstRow, self.lengthSignal, axis=0)

    def addIncrementsCorrelationMatrix(self, RowsForEachZ:np.ndarray):
        """


        Parameters
        ----------
        RowsForEachZ : np.ndarray
            This array specifies the first row of the correlation matrix of the LSW increments.
            This array should be of shape : (FirstRowCorrelation, length_signal)

        Returns
        -------
        None.

        """

        self.__initializeIncrementCorrelationMatrixIfNot()
        # Check if the first row provided has the same shape of the number of scales.
        # If not, add zeros at the end of the 'FirstRow' array.
        self.__checkSignalLengthMatch(RowsForEachZ.shape[0])
        for z in range(self.lengthSignal):
            buildingMatrix = incrCorrMatrixAtTimeZ(RowsForEachZ[z], self.maxScale)

            # This line allows to get a semi-positive matrix
            buildingMatrix = buildingMatrix @ buildingMatrix.T

            self.incrementsCorrelationMatrix[:, :, z] = getCorrelationMatrixFromCovMatrix(buildingMatrix)

        self.__initializeSpectrum()

    def getRandomizedCoeffs(self):
        """
        This function gets the randomized coefficients for each time 'z'.

        Returns
        -------
        randomizedCoeffs : np.ndarray

        """

        originalSpectrum = self.evol_spectrum.getSpectrumOfOrder(0)
```

```
        randomizedCoeffs = np.empty((self.maxScale, self.lengthSignal), dtype=np.float64)
        for z in range(self.lengthSignal):
            randomizedCoeffs[:, z] = self.randomizeCoeffsAtTimeZ(z, originalSpectrum[:, z])
        return randomizedCoeffs

    def randomizeCoeffsAtTimeZ(self, z:int, spectrumAtTimeZ:np.ndarray):
        """
        Get the randomized coefficients from the EWS at one particular time 'z'.
        Note that we provide the EWS but the random coefficient is defined from the sqrt of the EWS.
        By having an identity matrix for the covariance of the increments, we recover the original model of
                                                                    Nason and al. (2000).
        If the covariance matrix is not identity, we get the model developped by Koch (2015).

        Parameters
        ----------
        z : int
            The particular time.
        spectrumAtTimeZ : np.ndarray
            Spectrum at time 'z'

        Returns
        -------
        randomizedCoeffsAtTimeZ : np.ndarray
            DESCRIPTION.

        """

        mean = np.repeat(0, len(spectrumAtTimeZ))
        cov = self.incrementsCorrelationMatrix[:, :, z]
        randomizedCoeffsAtTimeZ = np.sqrt(spectrumAtTimeZ) * np.random.multivariate_normal(mean, cov, size=1)
                                                                    .ravel()
        return randomizedCoeffsAtTimeZ

    def simulateScale(self, scaleCoeffs:np.ndarray, scale:int):
        """
        Simulate one particular scale. The simulation is done via convolving the randomized coefficients with
                                                                    the given discrete wavelet.

        Parameters
        ----------
        scaleCoeffs : np.ndarray
            Randomized coefficients of the LSW model.
        scale : int
            Scale at which we simulate the LSW model.

        Returns
        -------
        np.ndarray
            Simulated scale.

        """

        discreteWavelet = self.evol_spectrum.crossWavelet.discritization[scale, :self.lengthSignal]
        return utils.fft_convolve(scaleCoeffs, discreteWavelet)

    def simulate(self):
        """
        This function allows to simulate the LSW process from the Evolutionary Wavelet Spectrum (provided or
                                                                    estimated)
```

```
        Returns
        -------
        None.

        """

        self.__initializeSpectrumIfNot()
        self.simulation = np.zeros((self.lengthSignal,), dtype=np.float64)
        randomizedCoeffs = self.getRandomizedCoeffs()
        for i in range(self.maxScale):
            self.simulation += self.simulateScale(randomizedCoeffs[i, :], i)

    def graph(self):
        fig, ax = plt.subplots(1,1,figsize=(17, 8))
        ax.plot(self.simulation)

    def __initializeSpectrum(self):
        if self.isSignalASpectrum:
            self.evol_spectrum = ews.EWS(self.signal, isSpectrum=self.isSignalASpectrum, order=self.order,
                                                                    wavelet=self.wavelet)
        else:
            self.evol_spectrum = ews.EWS(self.decomposition.details, isSpectrum=self.isSignalASpectrum, order
                                                                    =self.order, wavelet=self.wavelet)
        self.evol_spectrum.setIncrementsCorrelationMatrix(self.incrementsCorrelationMatrix)

    def __isSignalASpectrum(self):
        if self.signal.ndim == 1:
            self.isSignalASpectrum = False
        else:
            self.isSignalASpectrum = True

    def __computeMaxScale(self):
        filterLength = len(np.array(pywt.Wavelet(self.wavelet_name).dec_lo))
        if self.isSignalASpectrum:
            self.maxScale = self.signal.shape[0]
            self.lengthSignal = self.signal.shape[1]
            self.__getMaxScaleBelowTwoTimeSignalLength(filterLength)
            self.signal = self.signal[:self.maxScale]
        else:
            self.maxScale = int(np.floor(np.log(len(self.signal))/np.log(2)))
            self.lengthSignal = len(self.signal)
            self.__getMaxScaleBelowTwoTimeSignalLength(filterLength)

    def __getMaxScaleBelowTwoTimeSignalLength(self, filterLength:int):
        while (2**self.maxScale - 1)*(filterLength-1)+1 > 2*self.lengthSignal:
            self.maxScale -= 1

    def __initializeDecomposition(self):
        if not self.isSignalASpectrum:
            self.decomposition = dec.WaveletDecomposition(self.signal, self.wavelet)

    def __initializeIncrementCorrelationMatrixIfNot(self):
        if not  utils.isArrayInitialized(self, 'incrementsCorrelationMatrix'):
            self.incrementsCorrelationMatrix = np.zeros((self.maxScale, self.maxScale, self.lengthSignal),
                                                                    dtype=np.float64)

    def __initializeSpectrumIfNot(self):
        if not utils.isArrayInitialized(self, 'evol_spectrum'):
```

```
            self.__initializeSpectrum()

    def __checkSignalLengthMatch(self, length:int):
        if length != self.lengthSignal:
            raise ValueError("Length of the array doesn't match length of signal")

"""
    The following functions could have been placed inside their corresponding classes.
    However, in order to speed up the computation using Numba, we have to extract them from the class
                                                                environment.
    The 'nb.njit' decorator allows to get speed of the function comparable to C.
"""

@nb.njit(nogil=True)
def matchFirstRowLengthToMaxScale(FirstRow:np.ndarray, maxScale:int):
    if len(FirstRow) >= (maxScale):
        return FirstRow[:maxScale]
    else:
        return np.concatenate((FirstRow, np.repeat(0.0, maxScale-len(FirstRow))))

@nb.njit(nogil=True)
def incrCorrMatrixAtTimeZ(FirstRow:np.ndarray, maxScale:int):
    """
    This function constructs the correlation matrix from the first row provided.

    Parameters
    ----------
    FirstRow : np.ndarray
        First row that will be replicated in the covariance matrix.
    maxScale : int
        Maximum scale of the wavelet decomposition in the LSW process.

    Returns
    -------
    np.ndarray
        Covariance matrix.

    """

    FirstRow = matchFirstRowLengthToMaxScale(FirstRow, maxScale)
    buildingMatrix = np.zeros((maxScale, maxScale), dtype=np.float64)

    for i in range(maxScale):
        buildingMatrix[i] = FirstRow
        FirstRow = np.roll(FirstRow, 1)

    buildingMatrix = np.triu(buildingMatrix)
    return buildingMatrix + buildingMatrix.T - np.diag(np.diag(buildingMatrix))

@nb.njit(nogil=True, fastmath=True)
def getCorrelationMatrixFromCovMatrix(cov:np.ndarray):
    """
    From a covariance matrix, get the associated correlation matrix.

    Parameters
    ----------
    cov : np.ndarray
        Covariance matrix.
```

```
    Returns
    -------
    np.ndarray
        Correlation matrix.

    """

    std = np.sqrt(np.diag(cov))
    std = np.outer(std, std)
    return cov / std
```

1.0.5 Factor_model_LSW.py

```
import numpy as np
import matplotlib.pyplot as plt
import LSW_model as lsw
import Evolutionary_Wavelet_Spectrum as ews
import WaveletDecomposition as dec
import custom_wavelets as wav
import Smoother as smo
import wavelet_utils as utils
import pywt
from scipy.sparse.linalg import eigs

class LSW_FactorModel:
    maxScale:int
    crossEWS:ews.CrossEWS
    loadings:np.ndarray
    factors:np.ndarray
    commonComp:np.ndarray
    def __init__(self, signals:np.ndarray, wavelet_name:str, order:int, n_factors:int):
        self.signals = signals      # (n_signals, length)
        self.n_signals = self.signals.shape[0]
        self.length_signal = self.signals.shape[1]
        self.wavelet_name = wavelet_name
        self.order = order
        self.n_factors = n_factors

        self.__computeMaxScale()
        self.wavelet = wav.Wavelet(self.wavelet_name, self.maxScale)
        self.wavelet.discretizeToMaxScale()

        self.__initializeModel()

    def __initializeModel(self):
        decomp = []
        for i in range(self.n_signals):
            decomp.append(dec.WaveletDecomposition(self.signals[i], self.wavelet).details)
        decomp = np.array(decomp)
        self.crossEWS = ews.CrossEWS(decomp, isSpectrum=False, order=self.order, wavelet=self.wavelet)
        # self.crossEWS.correctSpectrum()

    def smoothSpectrum(self, smoother:smo.Smoother):
        self.crossEWS.smoothSpectrum(smoother)

    def getLoadings(self):
        self.__initializeLoadingsIfNot()
        self.__initializeFactorsIfNot()
```

```
    for z in range(self.length_signal):
        S = self.createBlockSatTimeZ(z)
        eigVectors = eigs(S, k=self.n_factors, which='LM')[1]
        loadings = np.sqrt(self.n_signals*self.maxScale) * np.real(eigVectors)
        self.loadings[:, :, z] = loadings
        self.factors[:, z] = self.getFactors(z, loadings)

def createBlockSatTimeZ(self, z:int):
    block = [[self.getCorrectMatrix(scale1, scale2, z) for scale2 in range(self.maxScale)] for scale1 in
                                                        range(self.maxScale)]
    S = np.block(block)
    return S

def getCorrectMatrix(self, scale1:int, scale2:int, z:int):
    order = scale2 - scale1
    if np.abs(order) > self.order:
        return np.zeros((self.n_signals, self.n_signals), dtype=np.float64)
    elif order >= 0:
        return self.crossEWS.getSpectrumForAllSignalsAtTimeZ(scale1, order, z)
    else:
        return self.crossEWS.getSpectrumForAllSignalsAtTimeZ(scale2, -order, z).T

def getFactors(self, z:int, loadings:np.ndarray):
    decomp = np.ravel(self.crossEWS.decomposition[:, :, z], order='C')
    return (loadings.T @ decomp)/(self.n_signals*self.maxScale)
    # S = np.where(S < 0.0, 0.0, S)
    # diag = np.sqrt(np.diag(S))
    # return (loadings.T @ diag)/(self.n_signals*self.maxScale)

def getLoadingsForScaleAndElement(self, scale:int, element:int):
    idx = scale*self.n_signals + element
    return self.loadings[idx, :, :]      # out : (n_factors, length_signal)

def getCommonCompAtScaleAndElement(self, scale:int, element:int):
    commonCompMatrix = self.getLoadingsForScaleAndElement(scale, element).T @ self.factors
    return np.diag(commonCompMatrix)  # Only the diagonal element contains the common component.

def getCommonCompInTimeDomainForElement(self, element:int):
    for j in range(self.maxScale):
        discrete_wav = self.wavelet.discritization[j, :self.length_signal]
        commonCompAtScale = self.getCommonCompAtScaleAndElement(j, element)
        self.commonComp[element, :] += utils.fft_convolve(commonCompAtScale, discrete_wav)

def getCommonComp(self):
    self.__initializeCommonCompIfNot()
    for u in range(self.n_signals):
        self.getCommonCompInTimeDomainForElement(u)

def __computeMaxScale(self):
    filterLength = len(np.array(pywt.Wavelet(self.wavelet_name).dec_lo))
    self.maxScale = int(np.floor(np.log(self.length_signal)/np.log(2)))
    self.__getMaxScaleBelowTwoTimeSignalLength(filterLength)

def __getMaxScaleBelowTwoTimeSignalLength(self, filterLength:int):
    while (2**self.maxScale - 1)*(filterLength-1)+1 > 2*self.length_signal:
        self.maxScale -= 1

def __initializeLoadingsIfNot(self):
```

```
        if not utils.isArrayInitialized(self, 'loadings'):
            # loadings are sorted by scale then by element (outer : scale, inner : elements)
            self.loadings = np.empty((self.maxScale*self.n_signals, self.n_factors, self.length_signal),
                                                                    dtype=np.float64)

    def __initializeFactorsIfNot(self):
        if not utils.isArrayInitialized(self, 'factors'):
            self.factors = np.empty((self.n_factors, self.length_signal), dtype=np.float64)

    def __initializeCommonCompIfNot(self):
        if not utils.isArrayInitialized(self, 'commonComp'):
            self.commonComp = np.zeros((self.n_signals, self.length_signal), dtype=np.float64)
```

1.0.6 Code examples

EWS convergence

```
import numpy as np
import matplotlib.pyplot as plt
import LSW_model as lsw
import Evolutionary_Wavelet_Spectrum as ews
import WaveletDecomposition as dec
import custom_wavelets as wav
import Smoother as smo
from tqdm import tqdm

l = 1024    # Length of the time serie
mS = int(np.floor(np.log(l)/np.log(2))) # Maximum scale of decomposition
spect = np.zeros((mS, l), dtype=np.float64)
spect[0][256:589] = 1.0
spect[0][768:] = (np.sin(2*np.pi*np.linspace(0,1,l) - np.pi/4)**2 + 0.5)[768:]
spect[2][:256] = (np.sin(np.pi*np.linspace(0,1,l) - np.pi/4)**2 + 0.5)[:256]
spect[3][284:] = (np.sin(5*np.pi*np.linspace(0,1,l) - np.pi/4)**2 + 0.5)[284:]
spect[1]= (np.sin(5*np.pi*np.linspace(0,1,1024) - np.pi)**2 + 0.5)

n = 250     # Number of iteration
order = 0   # Order of the LSW model

lsw1 = lsw.LSW(spect, 'db1', order=order)
incrCorrRows = lsw1.getConstantIncrementsCorrelationRows(np.array([1.]))    # Here, it gives us the identity
                                                    matrix as the covariance between increments
lsw1.addIncrementsCorrelationMatrix(incrCorrRows)
avg_simulation = np.zeros_like(lsw1.evol_spectrum.spectrum)
# smoother = smo.Kernel_smoother('Gaussian', 75)
smoother = smo.SWT_smoother(wav.Wavelet('db10', 6), 'soft')
for i in tqdm(range(n)):

    lsw1.simulate()

    decomp_simulation = dec.WaveletDecomposition(lsw1.simulation, lsw1.wavelet)
    if i == 0:
        spect_simulation = ews.EWS(decomp_simulation.details, isSpectrum=False, order=order, wavelet=lsw1.
                                                                wavelet)
    spect_simulation.updateDecomposition(decomp_simulation.details)

    # spect_simulation.smoothSpectrum(smoother)
    spect_simulation.correctSpectrum()

    avg_simulation += (spect_simulation.spectrum/n)
```

```
avg_ews = ews.EWS(decomp_simulation.details, isSpectrum=False, order=order, wavelet=lsw1.wavelet)
avg_ews.spectrum = avg_simulation
avg_ews.graph(sharey=True)
```

Factor Model

```
import numpy as np
import matplotlib.pyplot as plt
import LSW_model as lsw
import Evolutionary_Wavelet_Spectrum as ews
import WaveletDecomposition as dec
import custom_wavelets as wav
import Smoother as smo
import Factor_model_LSW as fmodel
from tqdm import tqdm
import pandas as pd

data = pd.read_csv('euribor_daily.csv', delimiter=';', index_col=[0], parse_dates=True)
data.drop('date', inplace=True, axis=1)
data.index = data.index.rename('Time')

# data = data - data.shift(1)   # If we want ot model the returns
data.dropna(inplace=True)
data = data.iloc[::-1]

# smoother = smo.Kernel_smoother('Gaussian', 100)
smoother = smo.SWT_smoother(wav.Wavelet('db6', 8), 'soft')

np_data = np.flip(data.T.to_numpy(), axis=0)

# When the max. scale is '0' is will compute the max. scale given the number of observations
fm = fmodel.LSW_FactorModel(np_data, 'db1', order=0, n_factors=2, maxScale=0)
fm.smoothSpectrum(smoother)
fm.getLoadings()
fm.getCommonComponents()
```