**eda\_phasic.py**

The main function of the eda\_phasic function is to decompose a cleaned EDA signal into its two fundamental parts:

1. **Tonic component (SCL or EDA\_Tonic)**: The general tide level at the beach.
2. **Phasic component (EDA\_phasic or SCR)**: Individual waves crashing on the shore.

**Imports:**

**import numpy as np**

**import pandas as pd**

**import scipy.linalg**

**import scipy.signal**

**from ..signal import signal\_filter, signal\_resample, signal\_smooth**

* **scipy.linalg & scipy.signal**: Powerful tools from the SciPy library for advanced math, specifically linear algebra and signal processing.
* **signal\_filter, signal\_resample, signal\_smooth**: Helper functions from the NeuroKit library that perform common signal processing tasks.

**The Main Function**

def eda\_phasic(eda\_signal, sampling\_rate=1000, method="highpass", \*\*kwargs):

* **method="highpass"**: The specific algorithm to use. The default is "highpass". Other options include "smoothmedian", "cvxeda", and "sparse".
* **\*\*kwargs**: A "catch-all" for any extra settings you want to pass to the chosen method (e.g., you might want to change the cutoff for the "highpass" filter).

**"""Electrodermal Activity (EDA) Decomposition"""**

This is a detailed docstring explaining the purpose of the function, the different methods it offers, its parameters, and showing examples of how to use it and compare the methods.

**Dispatcher Logic**

method = method.lower() # Remove capitalized letters

if method in ["cvxeda", "convex"]:

tonic, phasic = \_eda\_phasic\_cvxeda(eda\_signal, sampling\_rate)

elif method in ["sparse", "sparseda"]:

tonic, phasic = \_eda\_phasic\_sparseda(eda\_signal, sampling\_rate)

elif method in ["median", "smoothmedian"]:

tonic, phasic = \_eda\_phasic\_mediansmooth(eda\_signal, sampling\_rate, \*\*kwargs)

elif method in ["neurokit", "highpass", "biopac", "acknowledge"]:

tonic, phasic = \_eda\_phasic\_highpass(eda\_signal, sampling\_rate, \*\*kwargs)

else:

raise ValueError(

"NeuroKit error: eda\_phasic(): 'method' should be one of ... "

)

* **method = method.lower()**: Makes the method name lowercase for consistency.
* **The if/elif/else chain**: Checks which method you chose.
* Based on your choice, it calls the corresponding helper function (e.g., \_eda\_phasic\_cvxeda for "cvxeda").
* It saves the results into the tonic and phasic variables.
* If you enter a method that doesn't exist, it raises an error.

return pd.DataFrame({"EDA\_Tonic": tonic, "EDA\_Phasic": phasic})

Finally, it takes the tonic and phasic signals, puts them into a pandas DataFrame with column names, and returns it.

**The Helper "Specialist" Functions**

This method assumes the tonic signal is a very smooth version of the original signal.

def \_eda\_phasic\_mediansmooth(eda\_signal, sampling\_rate=1000, smoothing\_factor=4):

size = smoothing\_factor \* sampling\_rate

tonic = signal\_smooth(eda\_signal, kernel="median", size=size)

phasic = eda\_signal - tonic

return np.array(tonic), np.array(phasic)

* **size = smoothing\_factor \* sampling\_rate**: Calculates the size of the smoothing window in samples. A smoothing factor of 4 means a 4-second window.
* tonic = signal\_smooth(..., kernel="median", ...): It creates the tonic signal by applying a median filter. This filter slides a window along the signal and, for each position, takes the median (middle) value of all the points in the window. This is very effective at removing sharp peaks (the phasic part).
* phasic = eda\_signal - tonic: It calculates the phasic signal by simply subtracting the smooth tonic baseline from the original signal. What's left over are the fast responses.

**Example:**  
If the signal is [1, 1, 5, 1, 1]. A median filter would see the 5 as an outlier and replace it with 1, resulting in a tonic signal of [1, 1, 1, 1, 1].  
Subtracting this from the original leaves the phasic part: [0, 0, 4, 0, 0].

**\_eda\_phasic\_highpass(...)**

This method assumes the phasic signal contains the high-frequency parts of the signal and the tonic contains the low-frequency parts.

def \_eda\_phasic\_highpass(eda\_signal, sampling\_rate=100, cutoff=0.05):

phasic = signal\_filter(eda\_signal, sampling\_rate=sampling\_rate,

lowcut=cutoff, method="butter")

tonic = signal\_filter(eda\_signal, sampling\_rate=sampling\_rate,

highcut=cutoff, method="butter")

return tonic, phasic

* **phasic = signal\_filter(..., lowcut=cutoff, ...)**: It creates the phasic signal by applying a high-pass filter. This filter removes everything that changes slower than 0.05 Hz (the slow tonic drift) and keeps everything that changes faster (the quick responses).
* tonic = signal\_filter(..., highcut=cutoff, ...): It creates the tonic signal by applying a low-pass filter. This does the opposite: it keeps the slow-moving baseline and removes the fast-changing phasic responses.

**Advanced Methods**

An EDA signal is a combination of a slow tonic drift and a series of phasic responses that have a specific shape (a "Bateman function").

These methods then use a powerful math technique called **convex optimization** to find the tonic and phasic components that best reconstruct the original signal.

* \_eda\_phasic\_sparsEDA(...) and its helpers (lasso, etc.) (Experimental)
  + The function and its helpers (lasso, update\_chol, downdate\_chol) are extremely complex and implement an algorithm called LARS-lasso.

**\_eda\_phasic\_sparsEDA Breakdown:**

1. **Preprocessing:** It standardizes the signal by resampling it to 8 Hz and adding padding at the beginning and end.
2. **Build a Dictionary (R matrix).**
3. **Analyze in Chunks:** It slides a window across the signal, analyzing it piece by piece.
4. **Call Lasso:** For each chunk, it calls a lasso function. The goal of the lasso function is to pick the absolute minimum number of building blocks from the dictionary (R) that are needed to deconstruct that chunk of the signal.
5. **Post-processing.**
6. **Final Output:** It assembles the resulting tonic (SCL) and phasic (SCR) components and resamples them back to the original sampling rate.

**2. Lasso & its helpers (updateChol, downdateChol):**

* These functions implement the core optimization algorithm. You can think of them as the "engine" of sparsEDA.
* lasso iteratively tries to find the best and sparsest solution.
* updateChol and downdateChol are highly technical linear algebra functions that allow the algorithm to efficiently add or remove building blocks from its potential solution without having to restart the math from scratch every time. They make the process much faster.