**eda\_sympathetic.py**

* Two different "recipes" or methods (Posada & Ghisi) to achieve this goal.
* **signal\_psd\_welch**: A helper that performs Power Spectral Density (PSD) analysis using Welch's method, a standard and robust technique.
* **standardize**: A function to normalize data (z-scoring).

**Function Definition:**

**def eda\_sympathetic(eda\_signal, sampling\_rate=1000, frequency\_band=[0.045, 0.25], method="Posada", show=False):**

* **frequency\_band**: The target frequency range to analyze.
* **method**: Optional recipe ("Posada" or "Ghisi").
* **show**: Optional, to display a plot of the power spectrum.

**Low-pass Filter & Decimation:**

**downsampled\_1 = scipy.signal.decimate(...)**

**downsampled\_2 = scipy.signal.decimate(...)**

* Dramatically reduce the sampling rate in two stages: first from 400Hz to 40Hz (a factor of 10), and then from 40Hz to 2Hz (a factor of 20).

**Processing Steps:**

1. **Sanity Check**:
   * if len(eda\_signal) <= sampling\_rate \* 64:
2. **Resample to 400 Hz**
3. **Low-pass Filter**:
   * A Chebyshev Type I, 8th-order filter is used to remove high-frequency noise before the next step.
4. **Decimation (Downsampling)**
5. **High-pass Filter**
6. **Calculate Power Spectral Density (PSD):**

frequency, power = signal\_psd\_welch(...)```

---

\*\*Dispatcher Logic:\*\*

```python

out = {}

if method.lower() in ["ghiasi", "ghiasi2018"]:

out = \_eda\_sympathetic\_ghiasi(...)

elif method.lower() in ["posada", "posada-quintero", "quintero", "posada2016"]:

out = \_eda\_sympathetic\_posada(...)

else:

raise ValueError(...)

return out

This is the dispatcher logic.

* **out = {}**: Creates an empty dictionary to hold the results.
* **if/elif/else**: This block checks which method you chose (after converting it to lowercase).
* It calls the appropriate specialist helper function (\_eda\_sympathetic\_posada or \_eda\_sympathetic\_ghiasi) to do the actual work.
* **return out**: It returns the final dictionary containing the results. The result from the helper function is stored in out.
* If an invalid name is provided, it raises an error.

**Specialist 1: \_eda\_sympathetic\_posada**

**1. Sanity Check:**

if len(eda\_signal) <= sampling\_rate \* 64:

...

This method requires a lot of data to be reliable. This line checks if the data is at least 64 seconds long. If not, it warns the user and returns empty results (Not a Number/NaN).

**2. Resample to 400 Hz:**

**eda\_signal\_400hz = signal\_resample(...)**

* It first resamples the signal to a standard 400 Hz.

**3. Low-pass Filter:**

**sos = scipy.signal.cheby1(...)**

**eda\_signal\_filtered = scipy.signal.sosfilt(...)```**

**\* It applies a strong low-pass filter (an 8th-order Chebyshev filter) to remove high-frequency noise before the next step.**

**\*\*4. Decimation (Downsampling):\*\***

**```python**

**downsampled\_1 = scipy.signal.decimate(...)**

**downsampled\_2 = scipy.signal.decimate(...)**

* This is a key step. It dramatically reduces the sampling rate in two stages: first to 40 Hz (400/10) and then to 2 Hz (40/20). By sampling at only 2 Hz, the analysis is forced to focus only on very slow frequencies (below 1 Hz), which is where the sympathetic information is believed to be.

**5. High-pass Filter:**

**eda\_filtered = signal\_filter(...)**

* It applies a high-pass filter to the 2 Hz signal to remove any extremely slow DC drift that is not physiologically relevant.

**6. Calculate Power Spectral Density (PSD):**

frequency, power = signal\_psd\_welch(...)

* This is the core of the analysis. It takes the final, heavily processed 2 Hz signal and calculates its power spectrum, telling us how much power exists at each frequency.

**7. Calculate the Index:**

eda\_symp = signal\_power\_instant\_compute(psd, frequency\_band[0], frequency\_band[1])```

\* It takes the power spectrum and sums up all the power, but only within the target frequency band (e.g., 0.045 to 0.25 Hz). This sum is the "EDA-Sympathetic" index.

\*\*8. Normalize & Return:\*\*

```python

eda\_symp\_normalized = ...

out = {"EDA\_Sympathetic": eda\_symp, "EDA\_SympatheticN": eda\_symp\_normalized}

* It also calculates a normalized version by dividing the index by the total power, which can help in comparing results across different recordings. Finally, it returns these values in a dictionary.

**Specialist 2: \_eda\_sympathetic\_ghiasi**

def \_eda\_sympathetic\_ghiasi(...):

**1. Preprocessing:**

downsampled = signal\_resample(...)

normalized = standardize(...)

filtered = signal\_filter(...)

* This method uses a different preprocessing pipeline:
  + Resample to 50 Hz.
  + Normalize the signal's amplitude (z-score).
  + Then, apply a band-pass filter to keep only frequencies between 0.01 and 0.5 Hz.

**2. Time-Frequency Analysis:**

**..., bins = signal\_timefrequency(...)**

* This is the main difference from the Posada method. It uses the Short-Time Fourier Transform (STFT). Instead of getting one power spectrum for the whole signal, STFT gives you a "spectrogram."
* It shows how the power in different frequencies changes over time. bins contains the power values within your target frequency band for each moment in time.

**3. Calculate the Index:**

**eda\_symp = np.mean(bins)**

* The final index is simply the average power within the target frequency band across the entire signal duration.

**4. Normalize & Return:**

**eda\_symp\_normalized = eda\_symp / np.max(bins)**

**out = {"EDA\_Sympathetic": eda\_symp, "EDA\_SympatheticN": eda\_symp\_normalized}**

* It calculates the normalized version and returns the results in a dictionary.