**eda\_findpeaks.py**

**Purpose:** The purpose is to look at the phasic EDA signal (waves) and identify the locations of potential SCRs.

Imports:

import numpy as np

import pandas as pd

from ..signal import signal\_filter, signal\_findpeaks, signal\_smooth, signal\_zerocrossings

* **signal\_filter, signal\_smooth**: Helper functions to apply filters or smooth a signal (like ironing out wrinkles).
* **signal\_findpeaks**: A general-purpose peak-finding function.
* **signal\_zerocrossings**: A function to find where a signal crosses the zero line. This is very useful when analyzing a signal's derivative (its "speed").

**The Main eda\_findpeaks Function**

def eda\_findpeaks(eda\_phasic, sampling\_rate=1000, method="neurokit",

amplitude\_min=0.1):

if isinstance(eda\_phasic, pd.DataFrame):

* This is the data formatting step. If you pass in a full DataFrame, it tries to find and use the column named EDA\_Phasic.

method = method.lower()

if method in ["gamboa2008", "gamboa"]:

info = \_eda\_findpeaks\_gamboa2008(eda\_phasic)

# ... (elif blocks for all other methods) ...

else:

raise ValueError(...)

return info

**The Specialist Detector Functions**

A **peak** is any "hill" in the signal that is reasonably tall compared to the other hills.

def \_eda\_findpeaks\_neurokit(eda\_phasic, amplitude\_min=0.1):

peaks = signal\_findpeaks(eda\_phasic, relative\_height\_min=amplitude\_min, relative\_max=True)

info = {

"SCR\_Onsets": peaks["Onsets"],

"SCR\_Peaks": peaks["Peaks"],

"SCR\_Height": eda\_phasic[peaks["Peaks"]]

}

return info

* **peaks = signal\_findpeaks(...)**: It calls a general-purpose peak detector. relative\_height\_min=amplitude\_min tells it to only keep peaks that are at least 10% (by default) as tall as the tallest peak in the whole signal.
* **info = {...}**: It then takes the signal results from signal\_findpeaks and puts them into a dictionary with the standard SCR "keys". SCR\_Height is simply the value of the signal at the location of each peak.

A peak or real SCR isn't just a spike; it's a "hill" that takes time to climb and time to descend. A peak only counts if it has a sustained rise of at least 0.5 seconds and a sustained fall of at least 0.5 seconds.

def \_eda\_findpeaks\_vanhalem2020(eda\_phasic, sampling\_rate=1000):

# Smooth

eda\_phasic = signal\_filter(..., method="savgol", ...)

info = signal\_findpeaks(eda\_phasic)

peaks = info["Peaks"]

threshold = 0.5 \* sampling\_rate # 0.5 sec in samples

* It first lightly smooths the data to remove minor jitters. Then it finds all "possible" peaks with the general signal\_findpeaks function.

# Define each peak as a consistent increase of 0.5s

increase = info["Peaks"] - info["Onsets"]

peaks = peaks[increase > threshold]

* It calculates the duration of the rising slope for each peak (increase).
* It then filters the list, keeping only the peaks where the rise duration was longer than the 0.5-second threshold.

# check if each peak is followed by a constant decrease of 0.5s

decrease = info["Offsets"][idx] - peaks

# ... (code to filter peaks where decrease > threshold) ...

* It calculates the duration of the fall (decrease).
* It again filters the lists, keeping only the peaks that also have a fall duration longer than the threshold.
* Finally, it packages the fully filtered onsets, peaks, and heights into the info dictionary.

\_eda\_findpeaks\_gamboa2008 (2nd Derivative)

def \_eda\_findpeaks\_gamboa2008(eda\_phasic):

derivative = np.diff(np.sign(np.diff(eda\_phasic)))

* **np.diff(eda\_phasic)**: This calculates the signal's "speed" (the first derivative).
* **np.sign(...)**: This checks if the signal is positive (going up), negative (going down), or zero.
* **np.diff(...)**: Taking the difference again on the sign gives you the "change in speed," which is a proxy for the second derivative. A value less than 0 indicates a peak.

pi = np.nonzero(derivative < 0)[0] + 1 # peak indices

ni = np.nonzero(derivative > 0)[0] + 1 # trough (onset) indices

* It finds all the locations of peaks (pi) and troughs (ni).

peaks = pi[:-1]

onsets = ni[:-1]

* This section cleans up the lists to make sure every peak has a corresponding onset, ensuring they are properly paired.
* The final paired lists are then returned in the info dictionary.

**\_eda\_findpeaks\_Kim2008**

A **peak** is the highest point in the signal between two zero-crossings of its smoothed derivative (its "speed").

def \_eda\_findpeaks\_Kim2009(...):

df = np.diff(eda\_phasic) # Calculate derivative (speed)

df = signal\_smooth(signal\_df, ...) # Smooth the speed signal

zeros = signal\_zerocrossings(df) # Find where the speed is zero

* It calculates the derivative, smooths it heavily to remove noise, and then finds all the points where the smoothed derivative crosses zero. These zero-crossings roughly correspond to the onsets and offsets of the SCRs.

for i in range(0, len(zeros) - 1, 2):

scrs += [eda\_phasic[zeros[i]:zeros[i+1]]]

pks += [zeros[i] + np.argmax(eda\_phasic[zeros[i]:zeros[i+1]])]

* It loops through the zeros list two at a time (in pairs). Each pair represents the start and end of a potential SCR.
* Inside the loop, for each window between two zero-crossings, it finds the location of the maximum value. This location is the peak (pks).

# exclude SCRs with small amplitude

masked\_amps > (amplitude\_min \* np.nanmax(amps))

* Finally, it filters out all the detected SCRs that have an amplitude below the minimum threshold.

**\_eda\_findpeaks\_nabian2018**

This method is very similar to the Kim (2009) method. It also looks for peaks between zero-crossings of the smoothed derivative, but it uses a slightly different filtering logic.

def \_eda\_findpeaks\_nabian2018(eda\_phasic):

# ... (calculate smoothed derivative & find positive/negative crossings)

...

pos\_crossing = signal\_zerocrossing(..., direction="positive")

neg\_crossing = signal\_zerocrossing(..., direction="negative")

* Like Kim, it calculates the smoothed derivative. But instead of all zero-crossings, it specifically finds where the derivative goes from negative to positive (pos\_crossing, i.e., onsets) and from positive to negative (neg\_crossing, i.e., offsets).

# ... (loop through pairs of crossings) ...

for i, j in zip(pos\_crossings, neg\_crossings):

if len(amps\_list) == 0:

# append first peak

else:

diff = amp - eda\_phasic[i]

if not diff < 0.1 \* max(amps\_list):

# append the peak...

* It loops through paired onsets (i) and offsets (j).
* The filtering logic is **iterative**: a detected SCR is only kept if its amplitude is not less than 10% of the maximum amplitude found *so far*. This is subtly different from the Kim method, which compares the maximum of the entire signal.
* The final list of peaks, onsets, and heights that passed this test are returned in the info dictionary.