FINAL PROJECT

Photoplethysmography Signal Analysis for Stress Analysis Using DWT and EMD Algorithm to Acquire Breath Rate, Vasomotor Activity Range, and HRV Parameters

By: Group 5

Members	: 1. Jeremia Christ Immanuel Manalu (5023231017) 2. Abiel Ifan Imanuel Hukom (5023231004) 3. Rainhard Sintong Valentino Silitonga (5023231047)
Course	: Non-Stationary Signal Analysis
Class	: A and B
Lecturer	: Nada Fitrieyatul Hikmah, S.T., M.T.
Department	: Biomedical Engineering

FACULTY OF INTELLIGENT ELECTRICAL AND INFORMATICS TECHNOLOGY

INSTITUT TEKNOLOGI SEPULUH NOPEMBER

2025

CHAPTER I. FUNDAMENTAL THEORY

1.1. Photoplethysmography (PPG) Signal

Photoplethysmography (PPG) is a non-invasive optical method to measure changes in blood volume in body tissues. The PPG waveform is created when a light source, such as an LED, shines light into the skin and a photodetector measures how much light is reflected or transmitted by the blood vessels. As the heart beats, blood volume increases and decreases in the microvascular bed of the tissue, which causes the PPG detector output to rise and fall over time. A complete PPG pulse usually has a systolic peak, a dicrotic notch, and a diastolic peak in one cardiac cycle, and this shape carries important cardiovascular information. The PPG signal also contains two main components: an AC component related to the pulsatile changes of blood flow (heart beats) and a DC component related to slower baseline changes such as respiration and tissue absorption of light. In addition, the PPG signal also contains slower changes that are related to breathing (respiration) and vasomotor activity [1].

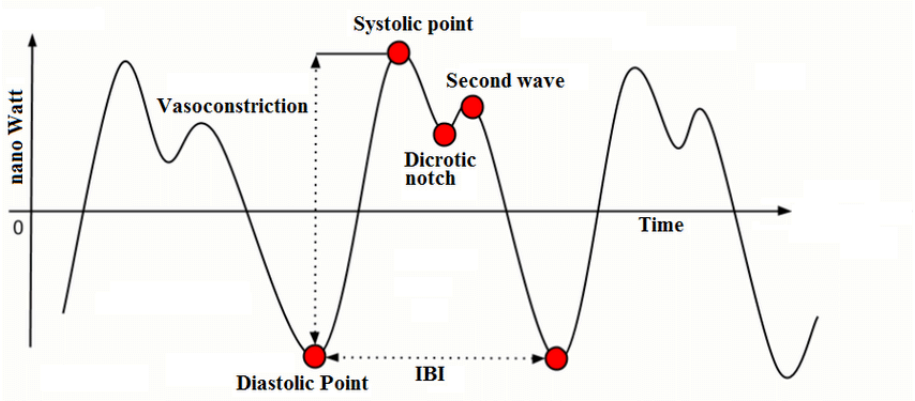


Figure 1.1 A Typical PPG Signal and Its Components

Respiration affects the PPG signal in several ways. As people breathe in and out, the pressure inside the chest changes. These changes slightly affect blood flow and the amount of blood in the periphery. This causes the PPG baseline, pulse amplitude, and timing to slowly change with breathing. These slower modulations carry the respiratory rate (breaths per minute) information and can be found in the lower frequency parts of the PPG spectrum. Vasomotor activity is related to the regulation of blood vessel diameter by the autonomic nervous system. This activity causes very low frequency changes (in Hz) in blood vessel tone and blood volume, and these changes also appear in the PPG signal as slow trends or fluctuations. These low-frequency variations are sometimes used to study vascular regulation or sympathetic activity [2].

The raw PPG waveform is often analyzed in both time and frequency domains because it is a non-stationary signal. Its statistical properties change over time due to physiological effects and motion artifacts. In the time domain, the main features are the peaks and troughs that correspond to heart beats, and measuring intervals between peaks gives heart rate and heart rate variability information. In the frequency domain, techniques like Fourier decomposition can help identify the dominant heartbeat frequency and separate it from noise or motion-related artifacts. Many studies show that motion artifacts and baseline wandering can overlap with true physiological frequencies, so careful signal processing (e.g., adaptive filtering or spectral methods) is required to obtain reliable heart rate estimation from a PPG signal [3].

PPG sensors are widely used in wearable health monitoring devices such as smartwatches and fitness trackers because they are compact and easy to use for continuous monitoring. A

wearable PPG sensor typically consists of an LED and photodetector in reflection mode (light is reflected back from tissue), and the signal is sampled at rates high enough to capture heartbeats (often >25–125 Hz). Because wearable data often contain noise from motion and environmental conditions, many methods like Fourier-based decomposition, time–frequency analysis, and advanced filtering are used to improve signal quality. These methods aim to separate true physiological components from noise so that heart rate and other vital parameters can be reliably extracted in daily life activities, even during exercise [3].

1.2. Heart Rate Variability (HRV)

Heart Rate Variability (HRV) is the fluctuation in the time intervals between adjacent heartbeats, typically measured as the variation in the RR intervals of the ECG or the pulse to pulse intervals in PPG. HRV is a proven non-invasive index for assessing the autonomic nervous system (ANS) regulation of the heart, reflecting the interplay between the sympathetic (SNS) and parasympathetic (PNS) nervous systems. A healthy heart is not a metronome; rather, it exhibits complex non-linear variability to adapt to changing physiological and environmental demands. Reduced HRV is often associated with stress, fatigue, and an increased risk of cardiovascular disease [4]. The normal values of some HRV parameters are shown in **Table 1.1** below.

Table 1.1. Normal Values of Standard Measures of some HRV Parameters

HRV metric	units	mean 24-hour value (± 1 S.D.)
SDNN	ms	141 ± 39
SDANN	ms	127 ± 35
RMSSD	ms	27 ± 12
HRV triangular index		37 ± 15
TP	ms^2	3466 ± 1018
LF	ms^2	1170 ± 416
HF	ms^2	975 ± 203
LF	nu	54 ± 4
HF	nu	29 ± 3
$\frac{LF}{HF}$-ratio		1.75 ± 0.35

1.2.1. Time Domain

Time-domain indices quantify the variability in measurements of the inter-beat interval (IBI), which is the time period between successive heartbeats. These metrics are derived from direct statistical analysis of the RR interval series.

- **SDNN.** This represents the standard deviation of all normal-to-normal (NN) intervals in the recording. It reflects the overall variability of the heart rate and includes contributions from both sympathetic and parasympathetic branches of the ANS. A higher SDNN generally indicates better cardiovascular health and resilience to stress [5]. The formula of SDNN can be seen below:

$$SDNN = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(RR_i - \overline{RR})^2}$$

- **RMSSD.** This parameter is calculated by taking the square root of the mean of the squares of the successive differences between adjacent NNs. RMSSD is the primary time-domain measure used to estimate the high-frequency variations in heart rate, making it a robust marker of parasympathetic (vagal) tone. It is less affected by respiratory trends compared to SDNN [5]. The formula of RMSSD can be seen below:

$$RMSSD = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(RR_{i+1} - RR_i)^2}$$

- **pNN50.** This measures the percentage of successive RR intervals that differ by more than 50 ms. Like RMSSD, pNN50 is closely correlated with parasympathetic activity. The formula of pNN50 can be seen below:

$$pNN50 = \frac{NN50}{N-1} \cdot 100\%$$

The complete HRV parameters in the time domain can be seen in **Table 1.1** below.

Table 1.2 Typical HRV Parameters in Time-Domain and Its Description

Variable	Units	Description
Statistical measures		
SDNN	ms	Standard deviation of all NN intervals
SDANN	ms	Standard deviation of the averages of NN intervals in all 5-minute segments of the entire recording
RMSSD	ms	The square root of the mean of the sum of the squares of differences between adjacent NN intervals
SDNN index	ms	Mean of the standard deviations of all NN intervals for all 5-minute segments of the entire recording
SDSD	ms	Standard deviation of differences between adjacent NN intervals
NN50 count		Number of pairs of adjacent NN intervals differing by more than 50 ms in the entire recording; three variants are possible counting all such NN intervals pairs or only pairs in which the first or the second interval is longer
pNN50	%	NN50 count divided by the total number of all NN intervals
Geometric measures		
HRV triangular index		Total number of all NN intervals divided by the height of the histogram of all NN intervals measured on a discrete scale with bins of 7.8125 ms (1/128 seconds)
TINN	ms	Baseline width of the minimum square difference triangular interpolation of the highest peak of the histogram of all NN intervals
Differential index	ms	Difference between the widths of the histogram of differences between adjacent NN intervals measured

1.2.2. Frequency Domain

Frequency-domain analysis decomposes the HRV signal into its constituent frequency components using methods such as the Fast Fourier Transform (FFT) or autoregressive modeling. This separation allows for the distinction between different physiological regulatory mechanisms.

- **Very Low Frequency (VLF).** Typically defined in the range of 0.0033–0.04 Hz. The physiological origin of VLF is complex and attributed to long-term regulatory mechanisms, including thermoregulation, the renin-angiotensin system, and vasomotor activity [4].
- **Low Frequency (LF).** Ranging from 0.04 to 0.15 Hz. LF power is often considered a marker of sympathetic modulation, although it contains a mix of both sympathetic and parasympathetic activity. It is also influenced by baroreflex activity [6].
- **High Frequency (HF).** Ranging from 0.15 to 0.4 Hz. HF power is almost exclusively mediated by the parasympathetic nervous system (vagal tone) and is highly correlated with the respiratory cycle, a phenomenon known as Respiratory Sinus Arrhythmia (RSA) [6].
- **LF/HF Ratio.** This ratio is widely used as a metric of sympathovagal balance. A higher ratio typically suggests sympathetic dominance (stress), while a lower ratio suggests parasympathetic dominance (relaxation).

The complete HRV parameters in the frequency domain can be seen in **Table 1.2** below.

Table 1.3 Typical HRV Parameters in Frequency-Domain and Its Description

FREQUENCY DOMAIN METHODS		
P	ms^2 or $mmHg^2$	Total power from 0–0.4 Hz
ULF	ms^2 or $mmHg^2$	Ultra low frequency band 0–0.05 Hz
VLF	ms^2 or $mmHg^2$	Very low frequency band 0.05–0.18 Hz
ULF+VLF	ms^2 or $mmHg^2$	Frequency band 0–0.18 Hz
LF	ms^2 or $mmHg^2$	Low frequency band 0.18-1 Hz
HF	ms^2 or $mmHg^2$	High frequency band 1-2 Hz
LF/HF	None	Quotient of LF and HF
HFn	None	Normalized high frequency band [HF/(LF+HF)]
LFn	None	Normalized low frequency band [LF/(LF+HF)]

1.2.3. Non-Linear

The cardiovascular system is inherently non-linear and complex. Non-linear methods describe the unpredictability and complexity of the heart rate time series, which linear methods might miss. A common visualization tool is the Poincaré Plot (as seen in **Figure 1.2**), a scatter plot where each RR interval (RR_N) is plotted against the next interval (RR_{n+1}).

- **SD1.** Describes the dispersion of points perpendicular to the line of identity in the Poincaré plot. It represents short-term variability, correlating strongly with RMSSD and parasympathetic activity [5].
- **SD2.** Describes the dispersion along the line of identity. It represents long-term variability and global HRV [5].

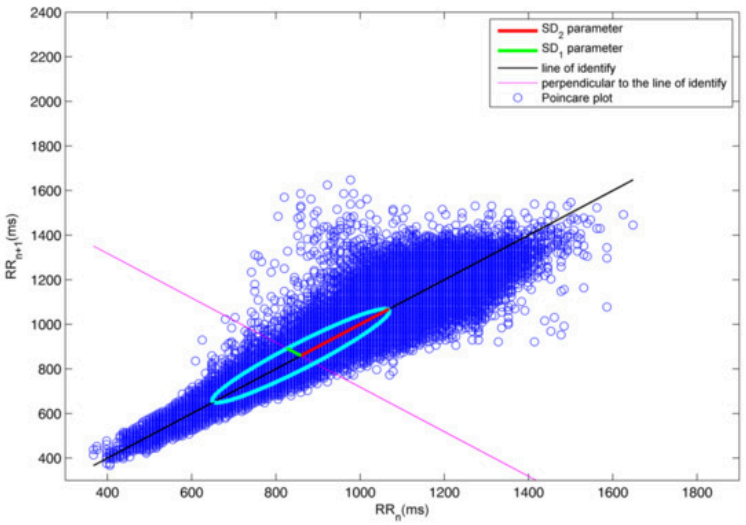


Figure 1.2. An Example of a Poincaré Plot along with Its SD1 and SD2 Parameters

1.3. Stress Analysis

Stress is a physiological and psychological response to demanding situations, triggering the "fight or flight" mechanism mediated by the Autonomic Nervous System (ANS). During acute stress, the sympathetic nervous system becomes hyperactive, while the parasympathetic withdrawal occurs. This imbalance manifests in physiological signals: the heart rate increases, blood vessels constrict (vasoconstriction) leading to increased blood pressure, and breathing becomes faster and shallower. In terms of signal analysis, stress is characterized by a reduction in overall Heart Rate Variability (HRV), specifically a decrease in HF power and an increase in the LF/HF ratio. Furthermore, stress influences vasomotor activity, causing fluctuations in the

blood volume baseline of the PPG signal due to changes in vascular resistance [7]. Detecting these physiological markers allows for the objective assessment of an individual's stress level.

1.4. DASS-21 Score

The Depression, Anxiety, and Stress Scale - 21 Items (or DASS-21) is a set of three self-report scales designed to measure the emotional states of depression, anxiety, and stress. It is a quantitative assessment tool where subjects rate the frequency and severity of experiencing negative emotional states over the previous week.

- **Stress Scale.** This specific sub-scale assesses difficulty relaxing, nervous arousal, and being easily upset/agitated, irritable/over-reactive, and impatient. Since DASS-21 is a condensed version of the DASS-42, the final scores for each sub-scale are multiplied by two to calculate the final severity rating. In biomedical signal research, the DASS-21 score serves as the "ground truth" or reference label to validate the physiological parameters extracted from PPG or ECG signals [8].

The complete DASS-21 example score can be seen in **Table 1.4** below.

Table 1.4. The DASS-21 Example Score for Depression, Anxiety, and Stress Condition

Subscale	Depression	Anxiety	Stress
Normal	0-4	0-3	0-7
Mild	5-6	4-5	8-9
Moderate	7-10	6-7	10-12
Severe	11-13	8-9	13-16
Extremely severe	14+	10+	17+

DASS=Depression anxiety stress scale

1.5. Discrete Wavelet Transform (DWT)

The Discrete Wavelet Transform (DWT) is a powerful signal processing technique used to analyze non-stationary signals like PPG. Unlike the Short-Time Fourier Transform (STFT), which uses a fixed window size, DWT employs a multi-resolution analysis approach. It decomposes the signal into approximation coefficients or low-frequency components and detail coefficients or high-frequency components using a series of high-pass and low-pass filters followed by downsampling.

The mathematical representation of a signal $x(t)$ using wavelets is given by the sum of its scaling function (approximation) and wavelet functions (details) at different scales. This allows DWT to provide good time resolution for high-frequency events and good frequency resolution for low-frequency events. In PPG analysis, DWT is particularly effective for denoising (removing motion artifacts and baseline wander) and extracting specific frequency bands related to heart rate and autonomic function without losing temporal information [9].

The frequency response of DWT with Morlet as the Mother Wavelet can be seen in **Figure 1.3** below.

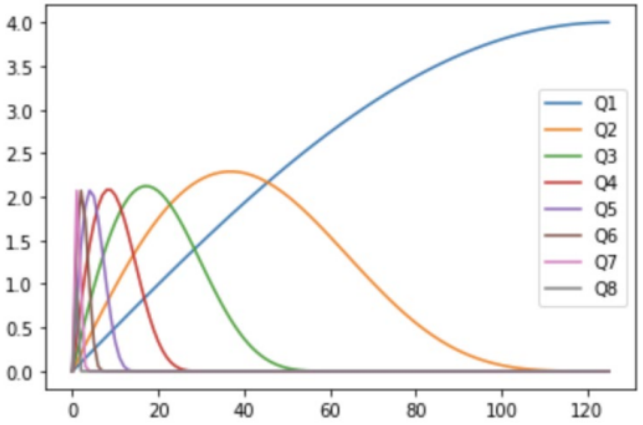


Figure 1.3. An Example of the DWT Frequency Response using Morlet as Mother Wavelet

1.6. Empirical Mode Decomposition (EMD)

1.6.1. Empirical Mode Decomposition (EMD)

Empirical Mode Decomposition (EMD) is an adaptive, data-driven method designed for analyzing non-linear and non-stationary signals. Unlike Fourier or Wavelet transforms, EMD does not rely on a pre-defined basis function. Instead, it decomposes a signal into a finite set of oscillatory components called Intrinsic Mode Functions (IMFs). The decomposition is performed through a "sifting process" which involves identifying local extrema, creating upper and lower envelopes using cubic spline interpolation, and calculating the mean envelope.

An IMF must satisfy two conditions: (1) the number of extrema and zero-crossings must differ by at most one, and (2) the mean value of the upper and lower envelopes is zero. However, standard EMD suffers from a phenomenon known as "mode mixing," where a single IMF contains widely disparate scales, or a signal of a similar scale resides in different IMFs, complicating physical interpretation [10].

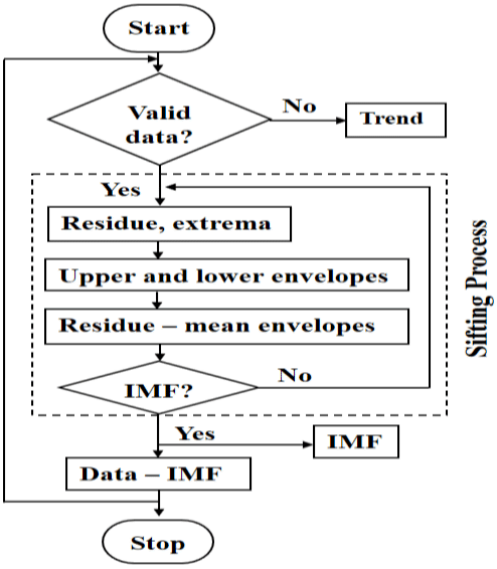


Figure 1.4. EMD Algorithm

1.6.2. Ensemble Empirical Mode Decomposition (EEMD)

To overcome the mode mixing problem in EMD, Ensemble EMD (EEMD) was introduced. EEMD defines the true IMF components as the mean of an ensemble of trials, each consisting of the signal plus a white noise of finite amplitude. Since white noise populates the entire

time-frequency space uniformly, adding it helps to separate signal components into proper reference scales. By averaging the IMFs obtained from multiple trials with different white noise series, the added noise cancels out, and the mode mixing is significantly reduced. However, EEMD introduces a new issue: the reconstructed signal includes residual noise, and the different realizations may produce a different number of IMFs, making averaging difficult [11].

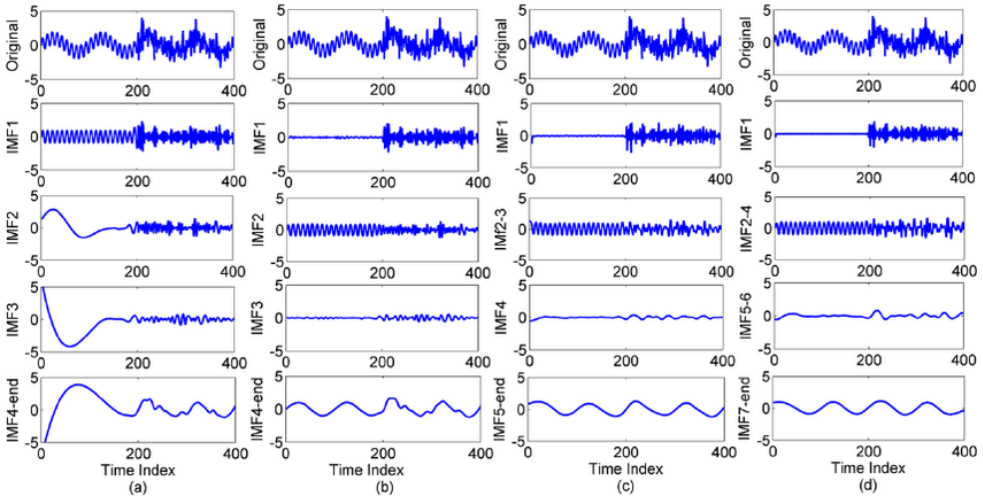


Figure 1.5. Comparison among extended EMD methods: (a) EMD, (b) EEMD, (c) CEEMD and (d) CEEMDAN

1.6.3. Complementary Ensemble Empirical Mode Decomposition (CEEMD)

Complementary EEMD (CEEMD) is an improvement over EEMD that aims to eliminate the residual noise in the reconstruction. In CEEMD, white noise is added in pairs (positive and negative) to the original signal. The decomposition is performed on both the (signal + noise) and (signal - noise). When the results are averaged, the residual noise from the added white noise is perfectly cancelled out, resulting in a cleaner reconstruction compared to standard EEMD, while still mitigating mode mixing [12].

1.6.4. Complete Ensemble Empirical Mode Decomposition with Adaptive Noise (CEEMDAN)

CEEMDAN allows for a complete decomposition with no reconstruction error and provides a better spectral separation of the modes. Unlike EEMD where noise is added at the beginning, CEEMDAN adds a particular noise (a specific IMF of white noise) at each stage of the decomposition process and computes a unique residue to obtain each IMF.

The algorithm proceeds as follows: for the first mode, it performs EEMD. For subsequent modes, it calculates the residue and applies EMD to the residue plus adaptive noise. This method effectively solves the mode mixing problem and reduces the reconstruction error to almost zero, making it highly suitable for extracting subtle physiological components like respiratory rate and vasomotor activity from PPG signals [12].

CHAPTER II. RESULT AND ANALYSIS

2.1. Problem Statement

The final project assignment details can be seen in **Figure 2.1** and **Figure 2.2** below.

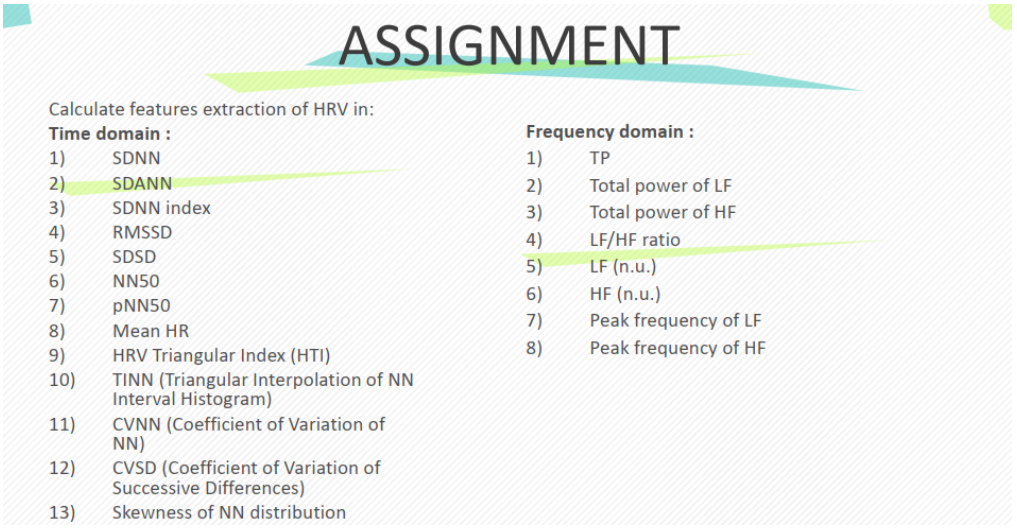


Figure 2.1. Final Project Assignment for HRV Features Extraction using DWT Algorithm

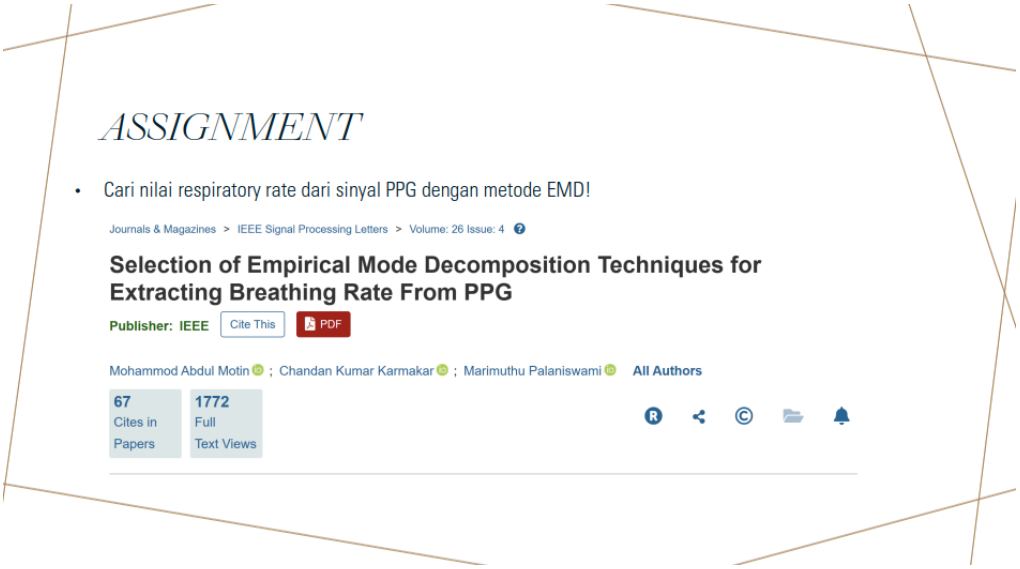


Figure 2.2. Final Project Assignment for Respiratory Rate and Vasomotor Activity Extraction using EMD Algorithm

[IMPORTANT NOTES BEFORE CONTINUING TO THE NEXT SECTION]:

*The dataset used in this report are **abiel-data.csv** that have the column format: **Index** as the first column and **Amplitude (0-4096)** as the second column. The details of the dataset are shown below:*

Index,Amplitude (0-4096)
0,1664
1,2096
2,2593
3,3034
...
14999,1223

2.2. Code Explanation

2.2.1. DWT Program (Using Delphi 12)

The software implementation for the DWT analysis was developed using the Delphi 12 IDE with Object Pascal. The program is designed as a modular desktop application that processes PPG signals to extract comprehensive Heart Rate Variability (HRV) features. The architecture of the code is organized into distinct logical units that handle file input/output, signal preprocessing, mathematical transformations, and feature extraction. The following subsections detail the algorithmic implementation of these core components, sequentially following the signal processing pipeline from raw data acquisition to the final time-domain, frequency-domain, and non-linear analysis.

a. Data Structures and Signal Loading + Signal Preprocessing (DC Removal) + BPF

```
1.  procedure TForm1.FormCreate(Sender: TObject);
2.  begin
3.    FAnalyzer := TPPGAnalyzer.Create;
4.    FDWTCoefficients := nil;
5.    SetupUI;
6.    Log('Analyzer ready. Please load a PPG data file.');
7.  end;
8.
9.  procedure TForm1.btnLoadDataClick(Sender: TObject);
10. begin
11.   if OpenDialogPPG.Execute then
12.   begin
13.     ClearAllData;
14.     Log('Loading file: ' + ExtractFileName(OpenDialogPPG.FileName));
15.     if LoadCSVData(OpenDialogPPG.FileName, edtColumnName.Text) then
16.     begin
17.       PageControlMain.ActivePage := TabSheetSignalProcessing;
18.       PlotSignal(SeriesRaw, FTimeRaw, FSignalRaw);
19.       Log(Format('Data loaded successfully. %d samples at %.2f Hz.',
    [Length(FSignalRaw), FOriginalFs]));
20.       btnProcess.Enabled := True;
21.     end
22.     else
23.     begin
24.       Log('ERROR: Failed to load data from CSV.');
25.       btnProcess.Enabled := False;
26.     end;
27.   end;
28. end;
29.
30. function TForm1.LoadCSVData(const FileName, ColumnName: string): Boolean;
31. var
32.   SL: TStringList; i, colIndex, timeColIndex, startIndex: Integer;
33.   line, colName: string; parts: TArray<string>;
34.   tempSignal, tempTime: TList<Double>;
35.   hasHeader: Boolean; val: Double;
36. begin
37.   Result := False; colIndex := -1; timeColIndex := -1; hasHeader := True;
38.   SL := TStringList.Create;
39.   tempSignal := TList<Double>.Create;
40.   tempTime := TList<Double>.Create;
```

```
41.   try
42.     SL.LoadFromFile(FileName);
43.     if SL.Count < 2 then raise Exception.Create('File is empty or has only a
      header.');
44.     line := SL[0];
45.     parts := line.Split([',', ';']);
46.     if not TryStrToFloat(Trim(parts[0]), val) then
47.     begin
48.       hasHeader := True;
49.       for i := 0 to High(parts) do
50.       begin
51.         colName := AnsiLowerCase(Trim(StringReplace(parts[i], "", ",
      [rfReplaceAll])));
52.         if (timeColIndex = -1) and (Pos('index', colName) > 0) then timeColIndex :=
      i;
53.         if (colIndex = -1) and (colName = AnsiLowerCase(Trim(ColumnName)))
      then colIndex := i;
54.       end;
55.     end
56.     else
57.     begin
58.       hasHeader := False;
59.       Log('CSV file detected without a header. Assuming Column 0 = Time, Column
      1 = Signal.');
60.       timeColIndex := 0; colIndex := 1;
61.     end;
62.     if colIndex = -1 then colIndex := High(parts);
63.     if timeColIndex = -1 then timeColIndex := 0;
64.
65.     startIndex := IfThen(hasHeader, 1, 0);
66.     for i := startIndex to SL.Count - 1 do
67.     begin
68.       line := SL[i];
69.       parts := line.Split([',', ';']);
70.       if (High(parts) >= colIndex) then
71.       begin
72.         if
      TryStrToFloat(StringReplace(parts[colIndex],'.',FormatSettings.DecimalSeparator,
      []), val) then
73.           tempSignal.Add(val);
74.       end;
75.     end;
76.
77.     if tempSignal.Count < 10 then raise Exception.Create('Not enough valid data
      points found.');
78.
79.     // FORCE TIME GENERATION BASED ON 50 Hz
80.     // This fixes the "Index vs Seconds" issue
81.     FOriginalFs := 50.0;
82.     FSignalRaw := tempSignal.ToArray;
83.     SetLength(FTimeRaw, Length(FSignalRaw));
84.     for i := 0 to High(FTimeRaw) do
85.       FTimeRaw[i] := i / FOriginalFs;
86.
```

```
87.     Result := True;
88.   except
89.     on E: Exception do
90.     begin
91.       Log('ERROR: ' + E.Message);
92.       Result := False;
93.     end;
94.   end;
95.   SL.Free; tempSignal.Free; tempTime.Free;
96. end;
```

1. Data Structures and Signal Loading

When we click the btnLoadData button, the function LoadCSVData is triggered. This function is responsible for reading wer .csv file. It reads the file line by line and looks for a specific column (default is Amplitude (0-4096)) to extract the raw signal values.

An important mathematical step happens here regarding time. The code forces a sampling frequency (Fs) of 50 Hz (FOriginalFs := 50.0). Since the raw file might only contain amplitude values (index), the computer must calculate the exact time (t) for every data point (n). The algorithm generates the time vector using the following linear equation:

$$t[n] = \frac{n}{F_s}$$

Where:

- t[n] is the time in seconds at index n.
- n is the sample index (0, 1, 2, …, N – 1).
- Fs is the sampling frequency (set to 50 Hz).

2. Signal Preprocessing (DC Removal)

Once the data is loaded, the user clicks btnProcess. The first and most critical step in biomedical signal analysis is Pre-processing. The code calls LocalRemoveDCOffset. Raw PPG signals often have a DC offset or baseline, which is a vertical shift caused by the pressure of the sensor against the skin. This shift does not contain heart rate information and must be removed to center the signal at zero.

The code calculates the average (mean) of the entire signal and subtracts it from every single data point. Mathematically, this is described as:

$$\mu = \frac{1}{N}\sum_{i=0}^{N-1} x[i]$$

$$x_{centered}[n] = x[n] - \mu$$

Where:

- μ is the mean (average) of the signal.
- N is the total number of samples.
- x[n] is the original raw signal.
- $x_{centered}[n]$ is the signal after DC removal.

3. Band Pass Filtering

After removing the DC offset, the code applies a Bandpass Filter using ApplyCustomBPF (in Unit1) or Filter_CustomRef (in PPGAnalyzer). This is very important. A PPG signal contains noise. Low-frequency noise comes from breathing (which we might want to keep or remove depending on the goal) and body movement. High-frequency noise comes from electrical interference (mains hum) or muscle tension. From the code applies a filter with a passband between 0.01 Hz and 8.0 Hz.

The code implements this filter using a Difference Equation. It creates a Low Pass Filter (LPF) first, and then sends that result through a High Pass Filter (HPF). The code explicitly calculates coefficients (a0, a1, a2, b1, b2) based on the sampling period (Tm) and the cutoff frequencies (Wc). The algorithm used in your code for the Low Pass Filter is a recursive infinite impulse response (IIR) filter, described by this equation:

$$y[n] = b_1 y[n-1] - b_2 y[n-2] + a_0 x[n] + a_1 x[n-1] + a_2 x[n-2]$$

Immediately after the Low Pass Filter, the code applies a High Pass Filter. This is done to remove very slow movements (like extensive baseline wander) that simply remove the mean (DC offset) missed. The equation used in your code for the HPF stage is:

$$z[n] = a_{0\_hp}(y[n] - y[n-1]) - b_{1\_hp} z[n-1]$$

This concludes the first stage of your software. The signal is now loaded, centered (zero-mean), and filtered to remove noise outside the 0.01 Hz - 8.0 Hz range. This clean signal is stored in FPreprocessedSignal.

b. Downsampling + FFT + DWT Decomposition

```
1.  procedure TForm1.LocalDownsample(const InSignal, InTime: TSignalArray;
    Factor: Integer; out OutSignal, OutTime: TSignalArray);
2.  var
3.    i, OutLen, Idx: Integer;
4.  begin
5.    if Factor <= 1 then
6.    begin
7.      OutSignal := Copy(InSignal);
8.      OutTime := Copy(InTime);
9.      Exit;
10.   end;
11.
12.   OutLen := Length(InSignal) div Factor;
13.   SetLength(OutSignal, OutLen);
14.   SetLength(OutTime, OutLen);
15.
16.   for i := 0 to OutLen - 1 do
17.   begin
18.     Idx := i * Factor;
19.     if Idx < Length(InSignal) then
20.     begin
21.       OutSignal[i] := InSignal[Idx];
22.       OutTime[i] := InTime[Idx];
23.     end;
24.   end;
25. end;
26.
27. procedure TPPGAnalyzer.Downsample(const SignalIn, TimeIn: TSignalArray;
    Factor: Integer;
28.   out SignalOut, TimeOut: TSignalArray; out NewFs: Double);
29. var i, newSize: Integer;
30. begin
31.   if (Factor <= 1) or (Length(SignalIn) = 0) then
32.   begin
33.     SignalOut := Copy(SignalIn); TimeOut := Copy(TimeIn);
34.     if Length(TimeIn) > 1 then NewFs := (Length(TimeIn) - 1) /
    (TimeIn[High(TimeIn)] - TimeIn[0]) else NewFs := 0;
```

```
35.     Exit;
36.   end;
37.   newSize := Length(SignalIn) div Factor;
38.   SetLength(SignalOut, newSize); SetLength(TimeOut, newSize);
39.   for i := 0 to newSize - 1 do
40.   begin
41.     SignalOut[i] := SignalIn[i * Factor]; TimeOut[i] := TimeIn[i * Factor];
42.   end;
43.   if Length(TimeOut) > 1 then NewFs := (Length(TimeOut) - 1) /
      (TimeOut[High(TimeOut)] - TimeOut[0]) else NewFs := 0;
44. end;
45.
46. procedure TPPGAnalyzer.FFTMagnitudeAndFrequencies(const Signal:
      TSignalArray; Fs: Double; out Freqs, Mags: TSignalArray);
47. var
48.   N, N_fft, half: Integer;
49.   paddedSignal: TSignalArray;
50.   fft_complex: TComplexArray;
51.   i: Integer;
52. begin
53.   SetLength(Freqs, 0);
54.   SetLength(Mags, 0);
55.   N := Length(Signal);
56.   if (N = 0) or (Fs <= 0) then Exit;
57.
58.   // 1. Tentukan ukuran FFT (pangkat 2 berikutnya) dan lakukan zero-padding
59.   N_fft := 1 shl Ceil(Log2(N));
60.   SetLength(paddedSignal, N_fft);
61.   for i := 0 to N - 1 do
62.     paddedSignal[i] := Signal[i];
63.   for i := N to N_fft - 1 do
64.     paddedSignal[i] := 0;
65.
66.   // 2. Lakukan FFT
67.   FFT(paddedSignal, fft_complex);
68.
69.   // 3. Hitung Magnitudo dan Frekuensi untuk spektrum satu sisi
70.   half := N_fft div 2;
71.   SetLength(Mags, half);
72.   SetLength(Freqs, half);
73.
74.   // DC Component (Freq = 0)
75.   Mags[0] := Sqrt(Sqr(fft_complex[0].Re) + Sqr(fft_complex[0].Im)) / N;
76.   Freqs[0] := 0;
77.
78.   // AC Components
79.   for i := 1 to half - 1 do
80.   begin
81.     Mags[i] := 2 * Sqrt(Sqr(fft_complex[i].Re) + Sqr(fft_complex[i].Im)) / N;
82.     Freqs[i] := i * Fs / N_fft;
83.   end;
84. end;
85.
86. function TPPGAnalyzer.Decompose_DWT(const Signal: TSignalArray):
```

```
    TDictionary<Integer, TSignalArray>;
87. var
88.   j, min_k, max_k, i: Integer;
89.   q_filter_dict: TDictionary<Integer, Double>;
90.   filter_coeffs: TSignalArray;
91. begin
92.   Result := TDictionary<Integer, TSignalArray>.Create;
93.   for j := 1 to 8 do
94.   begin
95.     if FQjTimeCoeffs.TryGetValue(j, q_filter_dict) and (q_filter_dict.Count > 0)
    then
96.     begin
97.       min_k := System.MaxInt;
98.       max_k := -2147483648;
99.       for i in q_filter_dict.Keys do begin
100.            if i < min_k then min_k := i;
101.            if i > max_k then max_k := i;
102.          end;
103.          SetLength(filter_coeffs, max_k - min_k + 1);
104.          for i := min_k to max_k do
105.          begin
106.            if q_filter_dict.ContainsKey(i) then
107.              filter_coeffs[i-min_k] := q_filter_dict[i]
108.            else
109.              filter_coeffs[i-min_k] := 0;
110.          end;
111.
112.          Result.Add(j, Convolve(Signal, filter_coeffs));
113.        end;
114.      end;
115.    end;
116.
117.    procedure TPPGAnalyzer._InitializeQjTimeCoeffs;
118.    var
119.      j, k, start_k, end_k: Integer;
120.      filter_dict: TDictionary<Integer, Double>;
121.    begin
122.      for j := 1 to 8 do
123.      begin
124.        filter_dict := TDictionary<Integer, Double>.Create;
125.        FQjTimeCoeffs.Add(j, filter_dict);
126.
127.        start_k := -(Round(Power(2, j)) + Round(Power(2, j - 1)) - 2);
128.        end_k := (1 - Round(Power(2, j - 1))) + 1;
129.
130.        case j of
131.          1: for k := start_k to end_k do filter_dict.Add(k, -2 * (DiracDelta(k) -
    DiracDelta(k + 1)));
132.          2: for k := start_k to end_k do filter_dict.Add(k, -1/4 * (DiracDelta(k-1) +
    3*DiracDelta(k) + 2*DiracDelta(k+1) - 2*DiracDelta(k+2) - 3*DiracDelta(k+3) -
    DiracDelta(k+4)));
133.          3: for k := start_k to end_k do filter_dict.Add(k, -1/32 * (DiracDelta(k-3) +
    3*DiracDelta(k-2) + 6*DiracDelta(k-1) + 10*DiracDelta(k) + 11*DiracDelta(k+1)
    + 9*DiracDelta(k+2) + 4*DiracDelta(k+3) - 4*DiracDelta(k+4) -
```

```
      9*DiracDelta(k+5) - 11*DiracDelta(k+6) - 10*DiracDelta(k+7) -
      6*DiracDelta(k+8) - 3*DiracDelta(k+9) - DiracDelta(k+10)));
.
.
.
.
134.            8: …));
135.          end;
136.        end;
137.      end;
138.
139.      function TPPGAnalyzer.DiracDelta(k: Integer): Integer;
140.      begin
141.        if k = 0 then Result := 1 else Result := 0;
142.      end;
143.
144.      function TPPGAnalyzer.Convolve(const Signal, Kernel: TSignalArray; Mode:
      TConvolutionMode): TSignalArray;
145.      var
146.        n, k, sigLen, kerLen, kerMid, start_k, end_k: Integer;
147.        sum: Double;
148.        flippedKernel: TSignalArray;
149.      begin
150.        sigLen := Length(Signal); kerLen := Length(Kernel);
151.        if (sigLen = 0) or (kerLen = 0) then Exit(nil);
152.        SetLength(flippedKernel, kerLen);
153.        for k := 0 to kerLen - 1 do flippedKernel[k] := Kernel[kerLen - 1 - k];
154.        kerMid := kerLen div 2;
155.        SetLength(Result, sigLen);
156.        for n := 0 to sigLen - 1 do begin
157.          sum := 0;
158.          start_k := Max(0, n - sigLen + kerMid + 1);
159.          end_k := Min(kerLen - 1, n + kerMid);
160.          for k := start_k to end_k do
161.            if (n - k + kerMid >= 0) and (n - k + kerMid < sigLen) then
162.              sum := sum + Signal[n - k + kerMid] * flippedKernel[k];
163.          Result[n] := sum;
164.        end;
165.      end;
```

1. Downsampling

After the signal has been filtered, the software performs "Downsampling" if the user selects a factor greater than 1 in the UI (SpinEditDownsample). This process reduces the number of data points. In biomedical engineering, we do this to reduce computational load or to focus on lower frequency components. The code LocalDownsample (in Unit1) or Downsample (in PPGAnalyzer) creates a new signal by selecting every M-th sample from the original signal.

For example, if the downsample factor M is 4, the code keeps sample 0, 4, 8, etc., and discards the rest. The mathematical relationship between the original signal x[n] and the downsampled signal y[m] is:

$$y[m] = x[m \cdot M]$$

Consequently, the sampling frequency changes. If the original frequency is Fs, the new sampling frequency Fs,new is defined as:

$$Fs, new = Fs/M$$

2. FFT

To verify the quality of the signal, the software visualizes the Frequency Spectrum using the Fast Fourier Transform (FFT). This happens in the method FFTMagnitudeAndFrequencies within the PPGAnalyzer unit. This method converts the signal from the Time Domain (amplitude vs. time) to the Frequency Domain (magnitude vs. frequency).

First, the algorithm ensures the signal length is a power of 2 (e.g., 256, 512, 1024) by adding zeros to the end of the signal (Zero Padding). This is necessary for the Cooley-Tukey FFT algorithm used in the code. The mathematical definition of the Discrete Fourier Transform (DFT) computed by this algorithm is:

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot e^{-j\frac{2\pi}{N}nk}$$

The code then calculates the Magnitude of the spectrum to display it on the chart. The magnitude tells us how strong a specific frequency is present in the signal. It is calculated using the Pythagorean theorem on the real (Re) and imaginary (Im) parts of the complex result:

$$|X[k]| = \frac{2}{N}\sqrt{Re(X[k])^2 + Im(X[k])^2}$$

The corresponding frequency f for each index k is calculated as:

$$f[k] = k \cdot \frac{F_s}{N}$$

3. DWT Decomposition

This is the most advanced part of the pre-processing logic, handled by Decompose_DWT in PPGAnalyzer. Unlike FFT, which gives global frequency information, DWT analyzes the signal at different "scales" or resolution levels. This is superior for non-stationary signals like PPG because it preserves time information.

The code initializes specific filter coefficients in _InitializeQjTimeCoeffs. These coefficients, denoted as Qj, represent a specific wavelet filter bank (likely a translation-invariant wavelet transform or Maximal Overlap DWT based on the DiracDelta implementation). The decomposition is performed via Convolution. The signal is convolved with these filter coefficients to produce detailed coefficients at different levels (Scale 1 to 8).

The convolution operation implemented in the function Convolve combines the signal x[n] with the filter kernel h[k] (which represents the DWT coefficients). The mathematical equation for this discrete convolution is:

$$y[n] = (x * h)[n] = \sum_{k=0}^{K-1} x[n-k] \cdot h[k]$$

This results in a FDWTCoefficients where key 1 contains the high-frequency details (Q1) and key 8 contains the lower-frequency trends (Q8). This decomposition allows to separate the respiratory component and vasomotor component from the main cardiac signal.

c. Cardiac Peak Detection (Zero-Crossing Method) + Time Domain HRV + Freq Domain HRV (Cubic Spline Interpolation and PSD) + Non-Linear HRV (Poincare Plot)

```
1. procedure TPPGAnalyzer.AnalyzeSignalZeroCrossing(const Signal, Time:
   TSignalArray;
2.   out MaximaIdx, MinimaIdx, ZeroCrossIdx: TArray<Integer>;
3.   out ZeroLineVal: Double);
4. var
5.   i: Integer;
```

```
6.    IsPositive: Boolean;
7.    LocalMax, LocalMin: Double;
8.    IdxMax, IdxMin: Integer;
9.    ListMax, ListMin, ListZero: TList<Integer>;
10.
11.   // Variabel untuk Logika Refractory (Pencegahan Double Peak)
12.   LastPeakTime, CurrentPeakTime: Double;
13.   MinRefractoryPeriod: Double; // Batas minimum antar detak (ms)
14.   LastPeakIndex: Integer;
15. begin
16.   // Reset Output
17.   SetLength(MaximaIdx, 0);
18.   SetLength(MinimaIdx, 0);
19.   SetLength(ZeroCrossIdx, 0);
20.
21.   if Length(Signal) < 2 then Exit;
22.
23.   ListMax := TList<Integer>.Create;
24.   ListMin := TList<Integer>.Create;
25.   ListZero := TList<Integer>.Create;
26.
27.   try
28.     // 1. Tentukan Garis Zero
29.     // Kita gunakan HARD ZERO (0.0) karena sinyal diasumsikan sudah
      di-detrend/bandpass di Unit1.
30.     // Ini mencegah garis referensi 'mengambang' mengikuti rata-rata sinyal.
31.     ZeroLineVal := 0.0;
32.
33.     // 2. Inisialisasi State
34.     IsPositive := Signal[0] >= ZeroLineVal;
35.     LocalMax := -MaxDouble;
36.     LocalMin := MaxDouble;
37.     IdxMax := -1;
38.     IdxMin := -1;
39.
40.     // 3. Setting Refractory Period
41.     // 300ms setara dengan heart rate 200 bpm.
42.     // Sinyal yang lebih cepat dari ini dianggap noise/artifact fisiologis.
43.     MinRefractoryPeriod := 0.30; // 0.30 detik
44.     LastPeakTime := -999.0;      // Waktu dummy awal
45.
46.     for i := 0 to High(Signal) do
47.     begin
48.       // --- A. Pencarian Ekstrim Lokal (Puncak & Lembah) ---
49.       if IsPositive then
50.       begin
51.         // Cari titik tertinggi selama berada di area positif
52.         if Signal[i] > LocalMax then
53.         begin
54.           LocalMax := Signal[i];
55.           IdxMax := i;
56.         end;
57.       end
58.       else
```

```
59.       begin
60.         // Cari titik terendah selama berada di area negatif
61.         if Signal[i] < LocalMin then
62.         begin
63.           LocalMin := Signal[i];
64.           IdxMin := i;
65.         end;
66.       end;
67.
68.       // --- B. Deteksi Zero Crossing ---
69.       if (i < High(Signal)) then
70.       begin
71.         // KASUS 1: Crossing DOWN (Positif -> Negatif)
72.         // Saat ini terjadi, gelombang sistolik (positif) selesai. Waktunya validasi
    puncak.
73.         if (IsPositive and (Signal[i+1] < ZeroLineVal)) then
74.         begin
75.           IsPositive := False;
76.           ListZero.Add(i);
77.
78.           // === LOGIKA SMART REFRACTORY PERIOD ===
79.           if IdxMax <> -1 then
80.           begin
81.             CurrentPeakTime := Time[IdxMax];
82.
83.             // Cek 1: Apakah ini puncak pertama?
84.             if ListMax.Count = 0 then
85.             begin
86.               ListMax.Add(IdxMax);
87.               LastPeakTime := CurrentPeakTime;
88.             end
89.             // Cek 2: Apakah jarak dari puncak terakhir CUKUP JAUH? (> 300ms)
90.             else if (CurrentPeakTime - LastPeakTime) >= MinRefractoryPeriod then
91.             begin
92.               // Jarak aman, terima sebagai puncak baru yang valid
93.               ListMax.Add(IdxMax);
94.               LastPeakTime := CurrentPeakTime;
95.             end
96.             // Cek 3: Jarak TERLALU DEKAT (< 300ms). Apakah ini puncak yang
    lebih baik?
97.             else
98.             begin
99.               // Ambil index puncak terakhir yang sudah tersimpan
100.                LastPeakIndex := ListMax.Last;
101.
102.                // Jika puncak baru LEBIH TINGGI dari puncak sebelumnya yang
    terlalu dekat,
103.                // Asumsikan puncak sebelumnya adalah noise (atau dicrotic notch
    yang tinggi),
104.                // dan puncak baru ini adalah sistolik yang sebenarnya.
105.                if Signal[IdxMax] > Signal[LastPeakIndex] then
106.                begin
107.                  ListMax.Delete(ListMax.Count - 1); // Hapus yang lama
108.                  ListMax.Add(IdxMax);              // Masukkan yang baru
```

```
109.                   LastPeakTime := CurrentPeakTime;   // Update waktu
110.                 end;
111.                 // Jika puncak baru lebih rendah, abaikan (dianggap noise/echo).
112.               end;
113.             end;
114.
115.             // Reset pencarian Minima
116.             LocalMin := MaxDouble;
117.             IdxMin := -1;
118.           end
119.           // KASUS 2: Crossing UP (Negatif -> Positif)
120.           else if (not IsPositive and (Signal[i+1] >= ZeroLineVal)) then
121.           begin
122.             IsPositive := True;
123.             ListZero.Add(i);
124.
125.             // Simpan Minima (Diastolik) - Biasanya tidak memerlukan logika
    refrakter ketat
126.             if IdxMin <> -1 then ListMin.Add(IdxMin);
127.
128.             // Reset pencarian Maxima
129.             LocalMax := -MaxDouble;
130.             IdxMax := -1;
131.           end;
132.         end;
133.       end;
134.
135.       // Konversi List ke Array
136.       MaximaIdx := ListMax.ToArray;
137.       MinimaIdx := ListMin.ToArray;
138.       ZeroCrossIdx := ListZero.ToArray;
139.
140.     finally
141.       ListMax.Free;
142.       ListMin.Free;
143.       ListZero.Free;
144.     end;
145.   end;
146.
147.   function TPPGAnalyzer.CalculateTimeDomain(const RRIntervals:
    TSignalArray; const PeakTimes: TSignalArray): TTimeDomainFeatures;
148.   var
149.     i, j, k, nn50_count, N, cleanCount: Integer;
150.     rr_ms, diffs: TSignalArray;
151.     mean_nn, diff_val, sum_sq_diff, sum_cubed_diff: Double;
152.     pop_sd: Double;
153.
154.     // Variabel untuk Sanitasi Data (Pembersihan)
155.     CleanRR: TList<Double>;
156.
157.     // Variabel Histogram & Geometrik
158.     minRR, maxRR, binWidth: Double;
159.     numBins, binIdx, maxBinCount, idxMode: Integer;
160.
```

```
161.       // Variabel TINN
162.       A, N_val, M_val, q_t, error, minError, bestN, bestM, t_val: Double;
163.
164.       // Variabel Segmented (SDANN)
165.       TotalDuration, SegmentWin, s_start, s_end: Double;
166.       s_means, s_stds, segment_rr: TList<Double>;
167.     begin
168.       FillChar(Result, SizeOf(TTimeDomainFeatures), 0);
169.
170.       // === 1. SANITASI DATA (OUTLIER REMOVAL) ===
171.       // Langkah ini krusial. Interval 200ms (300 bpm) adalah noise pada orang
      istirahat.
172.       CleanRR := TList<Double>.Create;
173.       try
174.         for i := 0 to High(RRIntervals) do
175.         begin
176.           // Filter Fisiologis:
177.           // Min 0.3s (300ms / 200 bpm)
178.           // Max 2.0s (2000ms / 30 bpm)
179.           if (RRIntervals[i] >= 0.30) and (RRIntervals[i] <= 2.0) then
180.             CleanRR.Add(RRIntervals[i]);
181.         end;
182.
183.         if CleanRR.Count < 5 then Exit; // Data terlalu sedikit setelah dibersihkan
184.
185.         // Konversi ke Array Milidetik untuk perhitungan selanjutnya
186.         N := CleanRR.Count;
187.         SetLength(rr_ms, N);
188.         for i := 0 to N - 1 do rr_ms[i] := CleanRR[i] * 1000; // Detik -> ms
189.       finally
190.         CleanRR.Free;
191.       end;
192.
193.       // === 2. STATISTIK LINEAR (Menggunakan Data Bersih) ===
194.
195.       // A. Mean HR & NN
196.       mean_nn := Mean(rr_ms);
197.       if mean_nn > 0 then Result.MeanHR := 60000 / mean_nn;
198.
199.       // B. Standard Deviations & Skewness
200.       sum_sq_diff := 0;
201.       sum_cubed_diff := 0;
202.
203.       for i := 0 to N - 1 do
204.       begin
205.         diff_val := rr_ms[i] - mean_nn;
206.         sum_sq_diff := sum_sq_diff + (diff_val * diff_val);
207.         sum_cubed_diff := sum_cubed_diff + (diff_val * diff_val * diff_val);
208.       end;
209.
210.       if N > 1 then Result.SDNN := Sqrt(sum_sq_diff / (N - 1)) else Result.SDNN
      := 0;
211.
212.       if sum_sq_diff > 0 then
```

```
213.      begin
214.        pop_sd := Sqrt(sum_sq_diff / N);
215.        Result.Skewness := (sum_cubed_diff / N) / (pop_sd * pop_sd * pop_sd);
216.      end else Result.Skewness := 0;
217.
218.      // C. RMSSD & pNN50 (Successive Differences)
219.      SetLength(diffs, N - 1);
220.      sum_sq_diff := 0; nn50_count := 0;
221.      for i := 0 to N - 2 do
222.      begin
223.        diffs[i] := rr_ms[i+1] - rr_ms[i];
224.        sum_sq_diff := sum_sq_diff + Sqr(diffs[i]);
225.        if Abs(diffs[i]) > 50 then Inc(nn50_count);
226.      end;
227.
228.      if Length(diffs) > 0 then
229.      begin
230.        Result.RMSSD := Sqrt(sum_sq_diff / Length(diffs));
231.        Result.SDSD := StdDev(diffs);
232.        Result.NN50 := nn50_count;
233.        Result.pNN50 := (nn50_count / Length(diffs)) * 100;
234.      end;
235.
236.      if mean_nn > 0 then
237.      begin
238.        Result.CVNN := Result.SDNN / mean_nn;
239.        Result.CVSD := Result.RMSSD / mean_nn;
240.      end;
241.
242.      // === 3. SEGMENTED STATISTICS (SDANN) ===
243.      // Kita gunakan PeakTimes asli untuk menentukan durasi,
244.      // tapi hanya menghitung data yang VALID dalam window tersebut.
245.      if Length(PeakTimes) > 1 then
246.      begin
247.        TotalDuration := PeakTimes[High(PeakTimes)] - PeakTimes[0];
248.
249.        // Aturan Window Adaptif
250.        if TotalDuration >= 300 then SegmentWin := 300      // 5 Menit (Standar)
251.        else if TotalDuration >= 60 then SegmentWin := 60   // 1 Menit (Short term)
252.        else SegmentWin := TotalDuration;                   // Seadanya
253.
254.        if (TotalDuration > 0) and (SegmentWin > 0) then
255.        begin
256.          s_means := TList<Double>.Create;
257.          s_stds := TList<Double>.Create;
258.          try
259.            s_start := PeakTimes[0];
260.            // Iterasi per segmen waktu
261.            while s_start + SegmentWin <= PeakTimes[High(PeakTimes)] do
262.            begin
263.              s_end := s_start + SegmentWin;
264.              segment_rr := TList<Double>.Create;
265.              try
266.                // Cari RR interval (bersih) yang jatuh dalam rentang waktu ini.
```

```
267.              // Karena rr_ms sudah kehilangan kaitan indeks langsung dengan
    PeakTimes akibat filtering,
268.              // kita gunakan pendekatan simplifikasi:
269.              // Asumsi: Distribusi artifact acak, kita ambil sampel rr_ms secara
    proporsional
270.              // atau (better logic) kita filter ulang raw data khusus untuk segmen ini.
271.
272.              // Implementasi Robust untuk SDANN: Filter ulang per segmen
273.              for i := 0 to High(RRIntervals) do
274.              begin
275.                // Cek apakah beat ini ada di window waktu s_start s/d s_end
276.                if (i < Length(PeakTimes)) and (PeakTimes[i] >= s_start) and
    (PeakTimes[i] < s_end) then
277.                begin
278.                  // Validasi nilai RR lagi
279.                  if (RRIntervals[i] >= 0.30) and (RRIntervals[i] <= 2.0) then
280.                    segment_rr.Add(RRIntervals[i] * 1000);
281.                end;
282.              end;
283.
284.              if segment_rr.Count > 2 then
285.              begin
286.                s_means.Add(Mean(segment_rr.ToArray));
287.                s_stds.Add(StdDev(segment_rr.ToArray));
288.              end;
289.            finally
290.              segment_rr.Free;
291.            end;
292.            s_start := s_end;
293.          end;
294.
295.          if s_means.Count > 1 then Result.SDANN := StdDev(s_means.ToArray);
296.          if s_stds.Count > 0 then Result.SDNNIndex := Mean(s_stds.ToArray);
297.        finally
298.          s_means.Free; s_stds.Free;
299.        end;
300.      end;
301.    end;
302.
303.    // === 4. GEOMETRIC MEASURES (Menggunakan rr_ms bersih) ===
304.    // (Logika histogram tetap sama, tapi input datanya sudah bersih dari noise
    200ms)
305.    if Length(rr_ms) > 0 then
306.    begin
307.      minRR := rr_ms[0]; maxRR := rr_ms[0];
308.      for i := 1 to High(rr_ms) do
309.      begin
310.        if rr_ms[i] < minRR then minRR := rr_ms[i];
311.        if rr_ms[i] > maxRR then maxRR := rr_ms[i];
312.      end;
313.
314.      binWidth := 50.0; // 50ms bin
315.      if maxRR = minRR then numBins := 1 else numBins := Floor((maxRR -
    minRR) / binWidth) + 1;
```

```
316.
317.        SetLength(Result.RRHistogramCounts, numBins);
318.        SetLength(Result.RRHistogramBins, numBins);
319.
320.        for i := 0 to numBins - 1 do
321.        begin
322.          Result.RRHistogramCounts[i] := 0;
323.          Result.RRHistogramBins[i] := minRR + (i * binWidth) + (binWidth / 2);
324.        end;
325.
326.        maxBinCount := 0; idxMode := 0;
327.        for i := 0 to High(rr_ms) do
328.        begin
329.          binIdx := Floor((rr_ms[i] - minRR) / binWidth);
330.          if (binIdx >= 0) and (binIdx < numBins) then
331.          begin
332.            Inc(Result.RRHistogramCounts[binIdx]);
333.            if Result.RRHistogramCounts[binIdx] > maxBinCount then
334.            begin
335.              maxBinCount := Result.RRHistogramCounts[binIdx];
336.              idxMode := binIdx;
337.            end;
338.          end;
339.        end;
340.
341.        if maxBinCount > 0 then Result.HTI := Length(rr_ms) / maxBinCount else
    Result.HTI := 0;
342.
343.        // Logika TINN (Triangular Interpolation)
344.        if (numBins > 1) and (maxBinCount > 0) then
345.        begin
346.          minError := MaxDouble;
347.          bestN := Result.RRHistogramBins[0];
348.          bestM := Result.RRHistogramBins[High(Result.RRHistogramBins)];
349.          A := maxBinCount;
350.
351.          for i := 0 to idxMode do
352.          begin
353.            for j := idxMode to numBins - 1 do
354.            begin
355.              N_val := Result.RRHistogramBins[i]; M_val :=
    Result.RRHistogramBins[j];
356.              if M_val = N_val then Continue;
357.              error := 0;
358.              for k := 0 to numBins - 1 do
359.              begin
360.                t_val := Result.RRHistogramBins[k];
361.                // Rumus Segitiga q(t)
362.                if (t_val >= N_val) and (t_val <= Result.RRHistogramBins[idxMode])
    then
363.                  if (Result.RRHistogramBins[idxMode]-N_val)<>0 then q_t :=
    A*(t_val-N_val)/(Result.RRHistogramBins[idxMode]-N_val) else q_t:=A
364.                else if (t_val > Result.RRHistogramBins[idxMode]) and (t_val <=
    M_val) then
```

```
365.                 if (M_val-Result.RRHistogramBins[idxMode])<>0 then q_t :=
    A*(M_val-t_val)/(M_val-Result.RRHistogramBins[idxMode]) else q_t:=A
366.               else q_t := 0;
367.
368.               error := error + Sqr(Result.RRHistogramCounts[k] - q_t);
369.             end;
370.             if error < minError then begin minError := error; bestN := N_val; bestM
    := M_val; end;
371.           end;
372.         end;
373.         Result.TINN := bestM - bestN;
374.         Result.TINN_N := bestN; Result.TINN_M := bestM;
375.         Result.TINN_Mode := Result.RRHistogramBins[idxMode];
    Result.TINN_Height := maxBinCount;
376.       end
377.       else Result.TINN := 0;
378.     end;
379.   end;
380.
381.   function TPPGAnalyzer.CalculateFrequencyDomain(const RRIntervals,
    PeakTimes: TSignalArray; InterpFs: Double): TFrequencyDomainFeatures;
382.   var
383.     interpTime, interpRR, freqs, psd: TSignalArray;
384.     i, j: Integer;
385.     freqStep, lf_power_sum, hf_power_sum, vlf_power_sum: Double;
386.     max_lf_val, max_hf_val: Double;
387.
388.     // Variabel Sanitasi
389.     CleanRR, CleanTimes: TList<Double>;
390.     ValidRR, ValidTimes: TSignalArray;
391.   begin
392.     FillChar(Result, SizeOf(TFrequencyDomainFeatures), 0);
393.     if Length(RRIntervals) < 10 then Exit;
394.
395.     // === 1. SANITASI DATA (Wajib untuk Frequency Domain) ===
396.     // Jika ada artifact tajam di RR, spektrum akan kacau (broadband noise).
397.     CleanRR := TList<Double>.Create;
398.     CleanTimes := TList<Double>.Create;
399.     try
400.       // Kita asumsikan PeakTimes[i] adalah waktu terjadinya detak ke-i
401.       // RRIntervals[i] adalah interval antara i dan i+1.
402.       // Kita harus menjaga pasangan (Waktu, Nilai) agar interpolasi Spline
    akurat.
403.       for i := 0 to High(RRIntervals) do
404.       begin
405.         if (RRIntervals[i] >= 0.30) and (RRIntervals[i] <= 2.0) then
406.         begin
407.           CleanRR.Add(RRIntervals[i]);
408.           if i < Length(PeakTimes) then
409.             CleanTimes.Add(PeakTimes[i]);
410.         end;
411.       end;
412.
413.       if CleanRR.Count < 5 then Exit;
```

```
414.          ValidRR := CleanRR.ToArray;
415.          ValidTimes := CleanTimes.ToArray;
416.        finally
417.          CleanRR.Free;
418.          CleanTimes.Free;
419.        end;
420.
421.        // === 2. INTERPOLASI & RESAMPLING ===
422.        // Buat grid waktu yang seragam (Uniformly sampled)
423.        SetLength(interpTime, Round((ValidTimes[High(ValidTimes)] -
      ValidTimes[0]) * InterpFs));
424.        for i := 0 to High(interpTime) do
425.          interpTime[i] := ValidTimes[0] + i / InterpFs;
426.
427.        // Interpolasi Cubic Spline (Mengubah data diskrit non-uniform menjadi
      uniform)
428.        interpRR := CubicSplineInterpolate(ValidTimes, ValidRR, interpTime);
429.
430.        // === 3. DETRENDING & WINDOWING ===
431.        // Hilangkan DC Component dan trend lambat
432.        LinearDetrend(interpRR);
433.
434.        // Welch's Method (FFT dengan averaging dan windowing)
435.        Welch(interpRR, InterpFs, freqs, psd, Min(512, Length(interpRR)));
436.
437.        if Length(freqs) < 2 then Exit;
438.
439.        // === 4. KONVERSI SATUAN (s^2/Hz -> ms^2/Hz) ===
440.        for i := 0 to High(psd) do
441.          psd[i] := psd[i] * 1000000; // Kali 1 Juta
442.
443.        Result.PSD_Freqs := freqs;
444.        Result.PSD_Values := psd;
445.
446.        // === 5. INTEGRASI & PEAK DETECTION ===
447.        freqStep := freqs[1] - freqs[0];
448.        lf_power_sum := 0; hf_power_sum := 0; vlf_power_sum := 0;
449.
450.        Result.Peak_LF := 0; Result.Peak_HF := 0;
451.        max_lf_val := -1.0; max_hf_val := -1.0;
452.
453.        for i := 0 to High(freqs) do
454.        begin
455.          // VLF Band (0.0033 - 0.04 Hz)
456.          if (freqs[i] >= 0.0033) and (freqs[i] < 0.04) then
457.            vlf_power_sum := vlf_power_sum + psd[i];
458.
459.          // LF Band (0.04 - 0.15 Hz)
460.          if (freqs[i] >= 0.04) and (freqs[i] < 0.15) then
461.          begin
462.            lf_power_sum := lf_power_sum + psd[i];
463.
464.            // Logika Peak: Simpan frekuensi dimana Power-nya Maksimal
465.            if psd[i] > max_lf_val then
```

```
466.          begin
467.            max_lf_val := psd[i];
468.            Result.Peak_LF := freqs[i];
469.          end;
470.        end;
471.
472.        // HF Band (0.15 - 0.5 Hz)
473.        // Diperluas ke 0.5Hz untuk mengakomodasi RSA frekuensi tinggi,
474.        // meskipun setelah sanitasi, kemungkinan besar puncak artifact di 0.4Hz
    akan hilang.
475.        if (freqs[i] >= 0.15) and (freqs[i] < 0.5) then
476.        begin
477.          hf_power_sum := hf_power_sum + psd[i];
478.
479.          // Logika Peak
480.          if psd[i] > max_hf_val then
481.          begin
482.            max_hf_val := psd[i];
483.            Result.Peak_HF := freqs[i];
484.          end;
485.        end;
486.      end;
487.
488.      // === 6. HASIL AKHIR ===
489.      Result.VLF_Power   := vlf_power_sum * freqStep;
490.      Result.LF_Power    := lf_power_sum * freqStep;
491.      Result.HF_Power    := hf_power_sum * freqStep;
492.
493.      // Total Power konsisten (Penjumlahan komponen)
494.      Result.Total_Power := Result.VLF_Power + Result.LF_Power +
    Result.HF_Power;
495.
496.      if Result.HF_Power > 0 then
497.        Result.LF_HF_Ratio := Result.LF_Power / Result.HF_Power;
498.
499.      if (Result.LF_Power + Result.HF_Power) > 0 then
500.      begin
501.        Result.LF_nu := Result.LF_Power / (Result.LF_Power + Result.HF_Power)
    * 100;
502.        Result.HF_nu := Result.HF_Power / (Result.LF_Power +
    Result.HF_Power) * 100;
503.      end;
504.    end;
505.
506.    function TPPGAnalyzer.CalculateNonLinear(const RRIntervals:
    TSignalArray): TNonLinearFeatures;
507.    // ... [Implementation from your original code, it's correct]
508.    var rr_n, rr_n1, diff_rr, sum_rr: TSignalArray; i: integer;
509.    begin
510.     if Length(RRIntervals) < 2 then
511.     begin
512.       FillChar(Result, SizeOf(TNonLinearFeatures), 0);
513.       Exit;
514.     end;
```

```
515.    SetLength(rr_n, High(RRIntervals)); SetLength(rr_n1, High(RRIntervals));
516.    for i := 0 to High(rr_n) do
517.    begin
518.      rr_n[i] := RRIntervals[i] * 1000;
519.      rr_n1[i] := RRIntervals[i+1] * 1000;
520.    end;
521.    Result.PoincareX := Copy(rr_n); Result.PoincareY := Copy(rr_n1);
522.    SetLength(diff_rr, Length(rr_n)); SetLength(sum_rr, Length(rr_n));
523.    for i := 0 to High(rr_n) do
524.    begin
525.      diff_rr[i] := (rr_n[i] - rr_n1[i]) / Sqrt(2);
526.      sum_rr[i] := (rr_n[i] + rr_n1[i]) / Sqrt(2);
527.    end;
528.    Result.SD1 := StdDev(diff_rr); Result.SD2 := StdDev(sum_rr);
529.    if Result.SD2 > 0 then Result.SD1_SD2_Ratio := Result.SD1 / Result.SD2
    else Result.SD1_SD2_Ratio := 0;
530.  end;
```

1. Cardiac Peak Detection (Zero-Crossing Method)

After the signal is pre-processed (filtered), the software must identify exactly where each heartbeat occurs. The code uses a Zero-Crossing Algorithm implemented in AnalyzeSignalZeroCrossing. Instead of just looking for the highest point (which can be noisy), the algorithm looks for the moments where the signal crosses from negative to positive (crossing zero).

Once a crossing is detected, the algorithm searches for the maximum value (peak) within that specific wave. To prevent errors, such as counting a small noise artifact as a heartbeat, the code implements a Refractory Period. This is a logic check that ensures two heartbeats cannot happen too close together (e.g., within 300 milliseconds). The time difference between two consecutive peaks is calculated to create the RR Interval.

The mathematical definition for the i-th RR interval (RRi), based on the time of the peaks (t), is:

$$RR_i = t_{i+1} - t_i$$

2. Time Domain HRV

Once the RR intervals are collected, the code enters the CalculateTimeDomain function. This is a statistical analysis of the time differences between heartbeats. The two most important standard metrics calculated here are SDNN and RMSSD.

SDNN (Standard Deviation of NN intervals) represents the overall variability of the heart rate. Mathematically, it is the standard deviation of the RR interval list. If N is the total number of intervals and $\overline{RR}$ is the average interval, the equation used is:

$$SDNN = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(RR_i - \overline{RR})^2}$$

RMSSD (Root Mean Square of Successive Differences) reflects the beat-to-beat variance and is the primary measure for parasympathetic (vagus nerve) activity. The algorithm calculates the difference between adjacent intervals $(RR_{i+1} - RR_i)$, squares them, averages them, and takes the square root:

$$RMSSD = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N-1}(RR_{i+1} - RR_i)^2}$$

3. Frequency Domain HRV (Cubic Spline Interpolation and PSD)

The code then moves to CalculateFrequencyDomain. This is mathematically more complex because RR intervals are unevenly sampled (heartbeats do not happen at fixed clock cycles). To perform FFT or Welch's method, the data must be evenly spaced.

To solve this, the code applies Cubic Spline Interpolation. This fits a smooth curve through the irregular RR points to generate a new, evenly sampled signal (typically at 4 Hz). The interpolation creates a polynomial function S(t) between data points:

$$S_i(t) = a_i + b_i(t - t_i) + c_i(t - t_i)^2 + d_i(t - t_i)^3$$

After interpolation and removing the trend (Detrending), the code calculates the Power Spectral Density (PSD) using Welch's method (as explained in the previous session). The software then integrates the area under the curve for specific frequency bands: Low Frequency (LF) (0.04–0.15 Hz) and High Frequency (HF) (0.15–0.40 Hz). The power in a frequency band is calculated as the sum of the PSD values in that range multiplied by the frequency resolution (df):

$$Power_{band} = \sum_{f=f_{low}}^{f_{high}} PSD(f) \cdot df$$

4. Non-Linear HRV (Poincare Plot)

Finally, the code calculates CalculateNonLinear features using a Poincaré Plot. This is a geometric method where each RR interval (RRi) is plotted against the next interval (RR_i+1).

The shape of this plot is analyzed by fitting an ellipse. The code calculates two standard descriptors: SD1 and SD2.

- SD1 represents the width of the ellipse (short-term variability) and is mathematically derived from the standard deviation of the differences between successive intervals.
- SD2 represents the length of the ellipse (long-term variability).

$$SD1 = \sqrt{\frac{1}{2}\mathrm{Var}(RR_i - RR_{i+1})}$$

$$SD2 = \sqrt{\frac{1}{2}\mathrm{Var}(RR_i + RR_{i+1})}$$

d. Geometric Analysis (TINN) + Autonomic Balance Diagram

```
1.  procedure TForm1.UpdateAdvancedPlots(const TimeFeats:
    TTimeDomainFeatures;
2.    const FreqFeats: TFrequencyDomainFeatures; const NonLinFeats:
    TNonLinearFeatures);
3.  var
4.    i: Integer;
5.    bin_width, meanRR: Double;
6.    skewInfo: string;
7.    SeriesVLF, SeriesLF, SeriesHF: TAreaSeries;
8.    f, val: Double;
9.    TotalPow_Disp, VLF_Pow, LF_Pow, HF_Pow: Double;
10.   Pct_VLF, Pct_LF, Pct_HF: Double;
11.   freqStep: Double;
12.   meanRR_ms, centerX, centerY: Double;
13.   SeriesSD1Line, SeriesSD2Line, SeriesEllipse: TLineSeries;
14.   SeriesIdentity: TLineSeries;
15.   t, angle, rad, x_ell, y_ell: Double;
```

```
16.   a, b, phi: Double; // a=SD2, b=SD1
17. begin
18.   // 1. Plot Tachogram (RR vs Time)
19.   SeriesTachogram.Clear;
20.   // Gunakan PeakTimes untuk sumbu X agar sesuai waktu kejadian
21.   // (Asumsi RRIntervals[i] adalah interval SETELAH PeakTimes[i])
22.   // Namun TimeFeats tidak menyimpan PeakTimes raw, jadi kita plot sequential
      saja atau ubah logic jika perlu.
23.   // Untuk visualisasi simpel:
24.   for i := 0 to High(NonLinFeats.PoincareX) do // PoincareX sebenarnya adalah
      RR[n] dalam ms
25.     SeriesTachogram.AddXY(i, NonLinFeats.PoincareX[i]);
26.
27.   // 2. Plot Histogram, TINN, dan Skewness
28.   SeriesRRHistogram.Clear;
29.   SeriesTINN.Clear;
30.   SeriesMeanLine.Clear;
31.
32.   if Length(TimeFeats.RRHistogramCounts) > 0 then
33.   begin
34.     // A. Plot Bar Histogram
35.     if Length(TimeFeats.RRHistogramBins) > 1 then
36.       bin_width := TimeFeats.RRHistogramBins[1] -
      TimeFeats.RRHistogramBins[0]
37.     else
38.       bin_width := 1;
39.
40.     for i := 0 to High(TimeFeats.RRHistogramCounts) do
41.       if i < Length(TimeFeats.RRHistogramBins) then
42.         SeriesRRHistogram.AddXY(TimeFeats.RRHistogramBins[i],
      TimeFeats.RRHistogramCounts[i]);
43.
44.     // B. Plot Segitiga TINN (Overlay)
45.     // Segitiga dibentuk oleh 3 titik: (N, 0) -> (Mode, Height) -> (M, 0)
46.     // Kita gambar garis dari N ke Mode ke M.
47.     if TimeFeats.TINN > 0 then
48.     begin
49.       SeriesTINN.AddXY(TimeFeats.TINN_N, 0);              // Titik Kiri Bawah
50.       SeriesTINN.AddXY(TimeFeats.TINN_Mode, TimeFeats.TINN_Height); //
      Puncak
51.       SeriesTINN.AddXY(TimeFeats.TINN_M, 0);              // Titik Kanan Bawah
52.     end;
53.
54.     // C. Plot Garis Mean (Indikator Skewness)
55.     // Jika MeanHR = 60000/MeanRR, maka MeanRR = 60000/MeanHR
56.     if TimeFeats.MeanHR > 0 then
57.     begin
58.       meanRR := 60000 / TimeFeats.MeanHR;
59.       // Gambar garis vertikal setinggi histogram tertinggi
60.       SeriesMeanLine.AddXY(meanRR, 0);
61.       SeriesMeanLine.AddXY(meanRR, TimeFeats.TINN_Height * 1.05); // Sedikit
      lebih tinggi dari puncak
62.     end;
63.
```

```
64.     // D. Update Judul Chart dengan Info Skewness
65.     if TimeFeats.Skewness > 0.1 then skewInfo := 'Positive Skew (Right Tail)'
66.     else if TimeFeats.Skewness < -0.1 then skewInfo := 'Negative Skew (Left Tail)'
67.     else skewInfo := 'Symmetric Distribution';
68.
69.     ChartRRHistogram.Title.Text.Text := Format('RR Histogram | TINN Triangle |
    Skewness: %.3f (%s)',
70.       [TimeFeats.Skewness, skewInfo]);
71.
72.     // Opsional: Atur Pen Style
73.     SeriesTINN.LinePen.Style := psDash;
74.     SeriesTINN.LinePen.Width := 2;
75.   end;
76.
77.   // 3. Plot HRV PSD (Grafik Kiri) - Area Filled
78.   ChartHRVPSD.SeriesList.Clear; // Hapus series lama
79.   ChartHRVPSD.Title.Text.Text := 'HRV Welch Periodogram (ms^2/Hz)';
80.   ChartHRVPSD.View3D := False;
81.   ChartHRVPSD.Legend.Visible := True;
82.   ChartHRVPSD.Legend.Alignment := laTop; // Legend di atas atau kanan
83.   ChartHRVPSD.Axes.Left.Title.Caption := 'PSD (ms^2/Hz)';
84.   ChartHRVPSD.Axes.Bottom.Title.Caption := 'Frequency (Hz)';
85.
86.   // Buat 3 Series Area Secara Manual (Runtime)
87.   SeriesVLF := TAreaSeries.Create(ChartHRVPSD);
88.   SeriesLF  := TAreaSeries.Create(ChartHRVPSD);
89.   SeriesHF  := TAreaSeries.Create(ChartHRVPSD);
90.
91.   // Setup Style VLF (Abu-abu / Gelap)
92.   SeriesVLF.ParentChart := ChartHRVPSD;
93.   SeriesVLF.Title := 'VLF';
94.   SeriesVLF.SeriesColor := clSilver; // Atau $00E0E0E0
95.   SeriesVLF.AreaLinesPen.Visible := False;
96.   SeriesVLF.Transparency := 30;
97.   SeriesVLF.UseYOrigin := True;
98.   SeriesVLF.YOrigin := 0;
99.
100.      // Setup Style LF (Ungu/Biru seperti referensi)
101.      SeriesLF.ParentChart := ChartHRVPSD;
102.      SeriesLF.Title := 'LF';
103.      SeriesLF.SeriesColor := $00FF8080; // Warna keunguan (BGR format)
104.      SeriesLF.AreaLinesPen.Visible := False;
105.      SeriesLF.Transparency := 30;
106.      SeriesLF.UseYOrigin := True;
107.      SeriesLF.YOrigin := 0;
108.
109.      // Setup Style HF (Cyan/Biru Muda seperti referensi)
110.      SeriesHF.ParentChart := ChartHRVPSD;
111.      SeriesHF.Title := 'HF';
112.      SeriesHF.SeriesColor := clAqua;
113.      SeriesHF.AreaLinesPen.Visible := False;
114.      SeriesHF.Transparency := 30;
115.      SeriesHF.UseYOrigin := True;
116.      SeriesHF.YOrigin := 0;
```

```
117.
118.       // --- ISI DATA KE SERIES BERDASARKAN FREKUENSI ---
119.       // Band Definitions:
120.       // VLF: 0.003 - 0.04
121.       // LF:  0.04 - 0.15
122.       // HF:  0.15 - 0.40
123.
124.       if Length(FreqFeats.PSD_Freqs) > 1 then
125.       begin
126.         freqStep := FreqFeats.PSD_Freqs[1] - FreqFeats.PSD_Freqs[0];
127.         VLF_Pow := 0; LF_Pow := 0; HF_Pow := 0;
128.
129.         for i := 0 to High(FreqFeats.PSD_Freqs) do
130.         begin
131.           f := FreqFeats.PSD_Freqs[i];
132.           val := FreqFeats.PSD_Values[i];
133.
134.           // Skip frekuensi negatif atau noise sangat rendah
135.           if f < 0.003 then Continue;
136.
137.           // Distribusi ke Series
138.           if (f < 0.04) then
139.           begin
140.             SeriesVLF.AddXY(f, val);
141.             VLF_Pow := VLF_Pow + val;
142.             // Titik sambung agar grafik mulus (tambahkan juga ke awal LF)
143.             if (i < High(FreqFeats.PSD_Freqs)) and (FreqFeats.PSD_Freqs[i+1] >=
    0.04) then
144.               SeriesLF.AddXY(f, val);
145.           end
146.           else if (f >= 0.04) and (f < 0.15) then
147.           begin
148.             SeriesLF.AddXY(f, val);
149.             LF_Pow := LF_Pow + val;
150.             // Titik sambung ke HF
151.             if (i < High(FreqFeats.PSD_Freqs)) and (FreqFeats.PSD_Freqs[i+1] >=
    0.15) then
152.               SeriesHF.AddXY(f, val);
153.           end
154.           else if (f >= 0.15) and (f <= 0.5) then // Margin diatur 0.5
155.           begin
156.             SeriesHF.AddXY(f, val);
157.             HF_Pow := HF_Pow + val;
158.           end;
159.         end;
160.
161.         // Batasi Tampilan Axis
162.         ChartHRVPSD.Axes.Bottom.SetMinMax(0, 0.4);
163.         ChartHRVPSD.Axes.Left.Automatic := True;
164.
165.         // --- HITUNG & TAMPILKAN STATISTIK (Sesuai Gambar 1) ---
166.         // Kalikan freqStep untuk mendapatkan Area (Power dalam ms^2)
167.         VLF_Pow := VLF_Pow * freqStep;
168.         LF_Pow  := LF_Pow * freqStep;
```

```
169.        HF_Pow  := HF_Pow * freqStep;
170.        TotalPow_Disp := VLF_Pow + LF_Pow + HF_Pow;
171.
172.        if TotalPow_Disp > 0 then
173.        begin
174.          Pct_VLF := (VLF_Pow / TotalPow_Disp) * 100;
175.          Pct_LF  := (LF_Pow / TotalPow_Disp) * 100;
176.          Pct_HF  := (HF_Pow / TotalPow_Disp) * 100;
177.
178.          // Update Judul Series untuk Legend
179.          SeriesVLF.Title := Format('VLF (%.1f%%)', [Pct_VLF]);
180.          SeriesLF.Title  := Format('LF (%.1f%%)', [Pct_LF]);
181.          SeriesHF.Title  := Format('HF (%.1f%%)', [Pct_HF]);
182.
183.          // Tampilkan Total Power di SubTitle atau Header Chart
184.          ChartHRVPSD.SubTitle.Text.Text := Format('Total Power: %.1f ms^2',
    [TotalPow_Disp]);
185.          ChartHRVPSD.SubTitle.Visible := True;
186.          ChartHRVPSD.SubTitle.Font.Color := clBlack;
187.          ChartHRVPSD.SubTitle.Font.Style := [fsBold];
188.        end;
189.      end;
190.
191.      // 4. Plot Poincare
192.      ChartPoincare.SeriesList.Clear;
193.      ChartPoincare.View3D := False;
194.      ChartPoincare.Title.Text.Text := 'Poincaré Plot (with SD1/SD2 Ellipse)';
195.      ChartPoincare.Axes.Bottom.Title.Caption := 'RR[n] (ms)';
196.      ChartPoincare.Axes.Left.Title.Caption := 'RR[n+1] (ms)';
197.
198.      // A. Plot Titik Data (Scatter)
199.      SeriesPoincare := TPointSeries.Create(ChartPoincare);
200.      SeriesPoincare.ParentChart := ChartPoincare;
201.      SeriesPoincare.Title := 'RR Intervals';
202.      SeriesPoincare.SeriesColor := clNavy;
203.      SeriesPoincare.Pointer.Style := psCircle;
204.      SeriesPoincare.Pointer.Size := 3;
205.      SeriesPoincare.Pointer.Pen.Visible := False; // Hilangkan garis pinggir titik
    agar rapi
206.      // Transparansi agar tumpukan titik terlihat
207.      SeriesPoincare.Transparency := 30;
208.
209.      if Length(NonLinFeats.PoincareX) > 0 then
210.        for i := 0 to High(NonLinFeats.PoincareX) do
211.          SeriesPoincare.AddXY(NonLinFeats.PoincareX[i],
    NonLinFeats.PoincareY[i]);
212.
213.      // Hitung Pusat Massa (Mean RR)
214.      if TimeFeats.MeanHR > 0 then
215.        meanRR_ms := 60000 / TimeFeats.MeanHR
216.      else
217.        meanRR_ms := 0;
218.
219.      if (meanRR_ms > 0) and (NonLinFeats.SD1 > 0) then
```

```
220.      begin
221.        centerX := meanRR_ms;
222.        centerY := meanRR_ms;
223.
224.        // B. Garis Identitas (Line of Identity x=y)
225.        // Kita gambar diagonal panjang melintasi chart
226.        SeriesIdentity := TLineSeries.Create(ChartPoincare);
227.        SeriesIdentity.ParentChart := ChartPoincare;
228.        SeriesIdentity.Title := 'Identity Line';
229.        SeriesIdentity.SeriesColor := clSilver;
230.        SeriesIdentity.LinePen.Style := psDash;
231.        SeriesIdentity.AddXY(0, 0);
232.        SeriesIdentity.AddXY(2000, 2000); // Asumsi max RR 2000ms
233.
234.        // C. Gambar Ellips (Fitting)
235.        // Persamaan Parametrik Ellips yang diputar 45 derajat
236.        // a = SD2 (Mayor), b = SD1 (Minor)
237.        SeriesEllipse := TLineSeries.Create(ChartPoincare);
238.        SeriesEllipse.ParentChart := ChartPoincare;
239.        SeriesEllipse.Title := 'SD Ellipse';
240.        SeriesEllipse.SeriesColor := clBlack;
241.        SeriesEllipse.LinePen.Width := 2;
242.        SeriesEllipse.LinePen.Style := psDash;
243.        SeriesEllipse.XValues.Order := loNone;
244.
245.        a := NonLinFeats.SD2; // Radius Mayor (sepanjang garis identitas)
246.        b := NonLinFeats.SD1; // Radius Minor (tegak lurus)
247.        phi := Pi / 4;        // Rotasi 45 derajat (0.785 radian)
248.
249.        // Loop 0 sampai 2*Pi untuk menggambar lingkaran/ellips
250.        t := 0;
251.        while t <= 2 * Pi + 0.1 do
252.        begin
253.          // Rumus rotasi ellips
254.          x_ell := centerX + a * Cos(t) * Cos(phi) - b * Sin(t) * Sin(phi);
255.          y_ell := centerY + a * Cos(t) * Sin(phi) + b * Sin(t) * Cos(phi);
256.          SeriesEllipse.AddXY(x_ell, y_ell);
257.          t := t + 0.1;
258.        end;
259.
260.        // D. Gambar Garis SD1 (Tegak Lurus - Cross hair pendek)
261.        // Garis melintasi ellips pada sumbu pendek
262.        SeriesSD1Line := TLineSeries.Create(ChartPoincare);
263.        SeriesSD1Line.ParentChart := ChartPoincare;
264.        SeriesSD1Line.Title := 'SD1 (Short Term)';
265.        SeriesSD1Line.SeriesColor := clRed;
266.        SeriesSD1Line.LinePen.Width := 3;
267.        SeriesSD1Line.LinePen.Style := psDash;
268.
269.        // Titik awal dan akhir garis SD1 (di tepi ellips)
270.        // t = Pi/2 dan t = 3*Pi/2 pada persamaan parametrik sebelum rotasi
271.        SeriesSD1Line.AddXY(
272.          centerX + a * Cos(Pi/2) * Cos(phi) - b * Sin(Pi/2) * Sin(phi),
273.          centerY + a * Cos(Pi/2) * Sin(phi) + b * Sin(Pi/2) * Cos(phi)
```

```
274.        );
275.        SeriesSD1Line.AddXY(
276.          centerX + a * Cos(3*Pi/2) * Cos(phi) - b * Sin(3*Pi/2) * Sin(phi),
277.          centerY + a * Cos(3*Pi/2) * Sin(phi) + b * Sin(3*Pi/2) * Cos(phi)
278.        );
279.
280.        // E. Gambar Garis SD2 (Sejajar - Cross hair panjang)
281.        // Garis melintasi ellips pada sumbu panjang
282.        SeriesSD2Line := TLineSeries.Create(ChartPoincare);
283.        SeriesSD2Line.ParentChart := ChartPoincare;
284.        SeriesSD2Line.Title := 'SD2 (Long Term)';
285.        SeriesSD2Line.SeriesColor := clBlue;
286.        SeriesSD2Line.LinePen.Width := 3;
287.        SeriesSD2Line.LinePen.Style := psDash;
288.
289.        // t = 0 dan t = Pi
290.        SeriesSD2Line.AddXY(
291.          centerX + a * Cos(0) * Cos(phi) - b * Sin(0) * Sin(phi),
292.          centerY + a * Cos(0) * Sin(phi) + b * Sin(0) * Cos(phi)
293.        );
294.        SeriesSD2Line.AddXY(
295.          centerX + a * Cos(Pi) * Cos(phi) - b * Sin(Pi) * Sin(phi),
296.          centerY + a * Cos(Pi) * Sin(phi) + b * Sin(Pi) * Cos(phi)
297.        );
298.
299.        // F. Atur Zoom Axis agar fokus ke Ellips
300.        // Buffer 1.5x dari SD2 agar ellips tidak mentok pinggir
301.        ChartPoincare.Axes.Bottom.SetMinMax(centerX - NonLinFeats.SD2 * 1.5,
    centerX + NonLinFeats.SD2 * 1.5);
302.        ChartPoincare.Axes.Left.SetMinMax(centerY - NonLinFeats.SD2 * 1.5,
    centerY + NonLinFeats.SD2 * 1.5);
303.      end;
304.
305.      //
    ================================================================
    ================
306.      // 5. AUTONOMIC BALANCE DIAGRAM (INTEGRASI KODE BARU)
307.      //
    ================================================================
    ================
308.
309.      // A. Persiapan Data (Hitung Ln)
310.      var LnLF, LnHF: Double;
311.      if FreqFeats.LF_Power > 1 then LnLF := Ln(FreqFeats.LF_Power) else
    LnLF := 0;
312.      if FreqFeats.HF_Power > 1 then LnHF := Ln(FreqFeats.HF_Power) else
    LnHF := 0;
313.
314.      // B. Bersihkan Chart
315.      ChartAutonomicBalance.SeriesList.Clear;
316.      ChartAutonomicBalance.View3D := False;
317.      ChartAutonomicBalance.Legend.Visible := False;
318.
319.      // Judul Chart
```

```
320.      ChartAutonomicBalance.Title.Text.Text := 'Autonomic Balance (Ln LF vs
    Ln HF)';
321.      ChartAutonomicBalance.Title.Font.Style := [fsBold];
322.
323.      // C. Konfigurasi Sumbu (2.0 s/d 9.0)
324.      with ChartAutonomicBalance.Axes.Bottom do
325.      begin
326.        Automatic := False;
327.        SetMinMax(2.0, 9.0);
328.        Increment := 1.0;
329.        Grid.Visible := False;
330.        LabelStyle := talValue; // Pastikan angka yang muncul
331.        Title.Caption := 'Sympathetic (Ln LF Power)';
332.        Title.Font.Style := [fsBold];
333.      end;
334.
335.      with ChartAutonomicBalance.Axes.Left do
336.      begin
337.        Automatic := False;
338.        SetMinMax(2.0, 9.0);
339.        Increment := 1.0;
340.        Grid.Visible := False;
341.        LabelStyle := talValue;
342.        Title.Caption := 'Parasympathetic (Ln HF Power)';
343.        Title.Font.Style := [fsBold];
344.      end;
345.
346.      // Koordinat Tengah 9 Kotak (Sesuai kode referensi Anda)
347.      var cx1 := 3.25; var cx2 := 5.50; var cx3 := 7.75;
348.      var cy1 := 3.25; var cy2 := 5.50; var cy3 := 7.75;
349.
350.      // D. Series Background (9 Zona Warna)
351.      var SeriesBackground := TPointSeries.Create(ChartAutonomicBalance);
352.      SeriesBackground.ParentChart := ChartAutonomicBalance;
353.      SeriesBackground.Title := 'Zones';
354.      SeriesBackground.Pointer.Style := psRectangle;
355.      SeriesBackground.Pointer.HorizSize := 2000; // Ukuran besar untuk blok
    warna
356.      SeriesBackground.Pointer.VertSize := 2000;
357.      SeriesBackground.Pointer.Pen.Visible := False;
358.      SeriesBackground.Marks.Visible := False;
359.      SeriesBackground.Active := True;
360.
361.      // Baris Bawah (Zona 1-3)
362.      SeriesBackground.AddXY(cx1, cy1, '', $008080FF); // Zona 1 (Merah Muda)
363.      SeriesBackground.AddXY(cx2, cy1, '', $0080C0FF); // Zona 2 (Oranye)
364.      SeriesBackground.AddXY(cx3, cy1, '', $008080FF); // Zona 3 (Merah Muda)
365.      // Baris Tengah (Zona 4-6)
366.      SeriesBackground.AddXY(cx1, cy2, '', $0080FFFF); // Zona 4 (Kuning)
367.      SeriesBackground.AddXY(cx2, cy2, '', $0080FF80); // Zona 5 (Hijau)
368.      SeriesBackground.AddXY(cx3, cy2, '', $0080FFFF); // Zona 6 (Kuning)
369.      // Baris Atas (Zona 7-9)
370.      SeriesBackground.AddXY(cx1, cy3, '', $0080FFFF); // Zona 7 (Kuning)
371.      SeriesBackground.AddXY(cx2, cy3, '', $00CCFFCC); // Zona 8 (Hijau
```

```
    Muda)
372.        SeriesBackground.AddXY(cx3, cy3, '', $00CCFFCC); // Zona 9
373.
374.        // E. Series Label Angka (1-9 di tengah kotak)
375.        var SeriesZoneLabels := TPointSeries.Create(ChartAutonomicBalance);
376.        SeriesZoneLabels.ParentChart := ChartAutonomicBalance;
377.        SeriesZoneLabels.Pointer.Visible := False; // Sembunyikan titik, hanya butuh
    teks
378.        SeriesZoneLabels.Marks.Visible := True;
379.        SeriesZoneLabels.Marks.Transparent := True; // Transparan agar warna
    background terlihat
380.        SeriesZoneLabels.Marks.Arrow.Visible := False;
381.        SeriesZoneLabels.Marks.Font.Size := 14;
382.        SeriesZoneLabels.Marks.Font.Style := [fsBold];
383.        SeriesZoneLabels.SeriesColor := clBlack;
384.
385.        SeriesZoneLabels.AddXY(cx1, cy1, '1');
386.        SeriesZoneLabels.AddXY(cx2, cy1, '2');
387.        SeriesZoneLabels.AddXY(cx3, cy1, '3');
388.        SeriesZoneLabels.AddXY(cx1, cy2, '4');
389.        SeriesZoneLabels.AddXY(cx2, cy2, '5');
390.        SeriesZoneLabels.AddXY(cx3, cy2, '6');
391.        SeriesZoneLabels.AddXY(cx1, cy3, '7');
392.        SeriesZoneLabels.AddXY(cx2, cy3, '8');
393.        SeriesZoneLabels.AddXY(cx3, cy3, '9');
394.
395.        // F. Garis Grid 3x3 (Menggantikan loop step 2 agar kompatibel & rapi)
396.        // Kita buat Series khusus garis agar tidak miring/diagonal
397.        var SeriesGridLines := TLineSeries.Create(ChartAutonomicBalance);
398.        SeriesGridLines.ParentChart := ChartAutonomicBalance;
399.        SeriesGridLines.Color := clSilver; // Sesuai request (Silver)
400.        SeriesGridLines.LinePen.Width := 2;
401.        SeriesGridLines.TreatNulls := tnDontPaint; // PENTING: Agar garis tidak
    menyambung sembarangan
402.
403.        // Batas antar zona adalah di 4.5 dan 6.5 (berdasarkan titik tengah 3.25, 5.5,
    7.75)
404.        // Garis Vertikal
405.        SeriesGridLines.AddXY(4.5, 2.0); SeriesGridLines.AddXY(4.5, 9.0);
406.        SeriesGridLines.AddNull(''); // Putus
407.        SeriesGridLines.AddXY(6.5, 2.0); SeriesGridLines.AddXY(6.5, 9.0);
408.        SeriesGridLines.AddNull(''); // Putus
409.        // Garis Horizontal
410.        SeriesGridLines.AddXY(2.0, 4.5); SeriesGridLines.AddXY(9.0, 4.5);
411.        SeriesGridLines.AddNull(''); // Putus
412.        SeriesGridLines.AddXY(2.0, 6.5); SeriesGridLines.AddXY(9.0, 6.5);
413.
414.        // G. Titik Pasien (Red Dot)
415.        SeriesAutonomicBalance := TPointSeries.Create(ChartAutonomicBalance);
416.        SeriesAutonomicBalance.ParentChart := ChartAutonomicBalance;
417.        SeriesAutonomicBalance.Title := 'Patient State';
418.        SeriesAutonomicBalance.Pointer.Style := psCircle;
419.        SeriesAutonomicBalance.Pointer.HorizSize := 8; // Ukuran titik pasien
420.        SeriesAutonomicBalance.Pointer.VertSize := 8;
```

```
421.       SeriesAutonomicBalance.Color := clRed;
422.       SeriesAutonomicBalance.Pointer.Pen.Color := clBlack; // Outline hitam biar
     kontras
423.       SeriesAutonomicBalance.Marks.Visible := False;
424.
425.       // Plot hanya jika nilai valid (>0) dan dalam rentang (2-9) agar terlihat
426.       if (LnLF > 2.0) and (LnHF > 2.0) then
427.       begin
428.         SeriesAutonomicBalance.AddXY(LnLF, LnHF);
429.
430.         // Update Subtitle Info
431.         ChartAutonomicBalance.SubTitle.Text.Text := Format('Result:
     Ln(LF)=%.2f, Ln(HF)=%.2f', [LnLF, LnHF]);
432.         ChartAutonomicBalance.SubTitle.Visible := True;
433.       end;
434.     end;
```

1. Geometric Analysis (TINN)

In the CalculateTimeDomain function within PPGAnalyzer, the code performs a geometric analysis known as TINN (Triangular Interpolation of NN Interval Histogram). This method creates a histogram (a bar chart) of the RR intervals and tries to fit a mathematical triangle over it. The base of this triangle (width) represents the overall variability of the heart rate.

The algorithm searches for the best triangle q(t) that fits the distribution of intervals. The triangle is defined by three points: a starting point N, a peak point (M_ode), and an end point M. The height of the triangle (A) is fixed to the maximum number of intervals found. The code calculates the value of the triangle line q(t) at any time t using linear interpolation equations.

For the left side of the triangle (rising slope), where the time t is between the start N and the peak Mode, the equation used is:

$$q(t) = A\frac{t - N}{Mode - N}$$

For the right side of the triangle (falling slope), where t is between the peak M_ode and the end M, the equation is:

$$q(t) = A\frac{M - t}{M - Mode}$$

The final TINN value is simply the width of the base of this triangle (M - N).

2. Autonomic Balance Diagram

This is a modern way to visualize the relationship between the Sympathetic nervous system (Stress/Fight-or-Flight) and the Parasympathetic nervous system (Rest/Digest).

While the standard ratio (LF/HF) is useful, the code implements a more advanced 2D plot. It takes the absolute power values of the Low Frequency (LF) and High Frequency (HF) bands and transforms them using the Natural Logarithm (ln). This transformation is necessary because power values can vary wildly (exponentially) between subjects, but the logarithmic scale makes them linear and easier to compare.

The coordinates (x,y) for plotting the subject's state on the chart are calculated as:

$$x = \ln(LF_{power})$$
$$y = \ln(HF_{power})$$

The code then places this point on a colored grid (Zoned Chart). If the point falls in the high x, low y area, it indicates high sympathetic stress. If it falls in the high y area, it indicates high parasympathetic recovery.

2.2.2. EMD Program (Using Python with PyQT6 as GUI Framework)

The following steps will summarizes the key algorithms implemented in the program, focusing on the decomposition pipeline (EMD/EEMD/CEEMD/CEEMDAN), time-frequency analysis (FFT/PSD/HHT), and the respiration–vasomotor feature extraction logic. GUI details are described only at a general level.

a. Preprocessing + Extrema + Spline Envelopes

```
1.      def detrend_poly(self, x: np.ndarray, deg: int = 3) -> np.ndarray:
2.          if x is None or len(x) == 0:
3.              return np.array([])
4.          t = np.arange(len(x))
5.          p = np.polyfit(t, x, deg=deg)
6.          trend = np.polyval(p, t)
7.          return x - trend
8.
9.      def running_mean(self, x: np.ndarray, window_samples: int) -> np.ndarray:
10.         if window_samples < 3:
11.             return x.copy()
12.         kernel = np.ones(window_samples) / window_samples
13.         return np.convolve(x, kernel, mode='same')
14.
15.     def preprocess(self, sig: np.ndarray, normalize: bool = True, remove_baseline:
    bool = True,
16.                    baseline_window_s: float = 2.0) -> np.ndarray:
17.         if sig is None or len(sig) == 0:
18.             return np.array([])
19.         x = sig.copy().astype(np.float64)
20.         x = x - np.mean(x)
21.         if remove_baseline:
22.             win = max(3, int(round(baseline_window_s * self.fs)))
23.             baseline = self.running_mean(x, win)
24.             x = x - baseline
25.         self.preproc = x.copy()
26.         if normalize:
27.             s = np.std(x)
28.             if s > 0:
29.                 x = x / s
30.         return x
31.
32.     def preproc_std(self) -> float:
33.         if self.preproc is None:
34.             return 0.0
35.         return float(np.std(self.preproc))
36.
37.     # --------------------------
38.     # Ekstrema (Optimized NumPy)
39.     # --------------------------
40.     def _local_extrema(self, x: np.ndarray) -> Tuple[np.ndarray, np.ndarray]:
41.         if len(x) < 3:
```

```
            return np.array([], dtype=int), np.array([], dtype=int)

        dx = np.diff(x)
        dx[np.abs(dx) < 1e-12] = 0.0  # flatten near-zero
        sgn = np.sign(dx)

        # maxima: + -> -  (sgn[i-1] > 0, sgn[i] < 0)
        maxima = np.where((sgn[:-1] > 0) & (sgn[1:] < 0))[0] + 1
        # minima: - -> +
        minima = np.where((sgn[:-1] < 0) & (sgn[1:] > 0))[0] + 1

        return maxima.astype(int), minima.astype(int)

    # --------------------------
    # Cubic Spline (SciPy-free)
    # --------------------------
    def _tridiagonal_solve(self, a, b, c, d):
        """
        Solve tridiagonal Ax=d where:
          a: lower diag (n-1)
          b: main diag  (n)
          c: upper diag (n-1)
          d: RHS        (n)
        """
        n = len(b)
        cp = np.zeros(n-1)
        dp = np.zeros(n)
        cp[0] = c[0] / b[0]
        dp[0] = d[0] / b[0]
        for i in range(1, n-1):
            denom = b[i] - a[i-1] * cp[i-1]
            cp[i] = c[i] / denom
            dp[i] = (d[i] - a[i-1] * dp[i-1]) / denom
        dp[n-1] = (d[n-1] - a[n-2] * dp[n-2]) / (b[n-1] - a[n-2] * cp[n-2])
        x = np.zeros(n)
        x[n-1] = dp[n-1]
        for i in range(n-2, -1, -1):
            x[i] = dp[i] - cp[i] * x[i+1]
        return x

    def _cubic_spline_interpolation(self, x_knots, y_knots, x_eval):
        """
        Natural spline interpolation.
        x_knots must be strictly increasing.
        """
        x_knots = np.asarray(x_knots, dtype=float)
        y_knots = np.asarray(y_knots, dtype=float)
        x_eval = np.asarray(x_eval, dtype=float)
        n = len(x_knots)
        if n < 2:
            return np.full_like(x_eval, y_knots[0] if n == 1 else 0.0)

        h = np.diff(x_knots)
        # Build system for second derivatives M (natural spline => M0=Mn-1=0)
```

```
96.         if n == 2:
97.             # linear
98.             return np.interp(x_eval, x_knots, y_knots)
99.
100.            # interior points: 1..n-2
101.            a = h[:-1]
102.            b = 2 * (h[:-1] + h[1:])
103.            c = h[1:]
104.            d = 6 * ((y_knots[2:] - y_knots[1:-1]) / h[1:] - (y_knots[1:-1] -
    y_knots[:-2]) / h[:-1])
105.
106.            M_interior = self._tridiagonal_solve(a, b, c, d)
107.            M = np.zeros(n)
108.            M[1:-1] = M_interior
109.
110.            # evaluate spline at x_eval
111.            y_eval = np.zeros_like(x_eval, dtype=float)
112.            # find interval indices
113.            idx = np.searchsorted(x_knots, x_eval) - 1
114.            idx = np.clip(idx, 0, n-2)
115.
116.            x_i = x_knots[idx]
117.            x_i1 = x_knots[idx+1]
118.            y_i = y_knots[idx]
119.            y_i1 = y_knots[idx+1]
120.            M_i = M[idx]
121.            M_i1 = M[idx+1]
122.            h_i = (x_i1 - x_i)
123.
124.            # S(x) formula
125.            diff_x = (x_i1 - x_eval)
126.            diff_x1 = (x_eval - x_i)
127.            term1 = (M_i1 * diff_x1**3 + M_i * diff_x**3) / (6 * h_i)
128.            term2 = (y_i1 - M_i1 * h_i**2 / 6) * (diff_x1 / h_i)
129.            term3 = (y_i - M_i * h_i**2 / 6) * (diff_x / h_i)
130.            y_eval = term1 + term2 + term3
131.
132.            return y_eval
133.
134.        # --------------------------
135.        # Envelope (Spline)
136.        # --------------------------
137.        def get_envelopes(self, x: np.ndarray) -> Dict:
138.            """
139.            Compute upper & lower envelope using cubic spline interpolation across
    maxima/minima.
140.            Returns dict with env_max, env_min, mean_env, max_idx, min_idx.
141.            """
142.            N = len(x)
143.            max_idx, min_idx = self._local_extrema(x)
144.
145.            # Need at least 2 knots each for spline; clamp endpoints if needed
146.            if len(max_idx) < 2:
147.                max_idx = np.array([0, N-1], dtype=int)
```

```
148.            else:
149.                if max_idx[0] != 0:
150.                    max_idx = np.insert(max_idx, 0, 0)
151.                if max_idx[-1] != N-1:
152.                    max_idx = np.append(max_idx, N-1)
153.
154.            if len(min_idx) < 2:
155.                min_idx = np.array([0, N-1], dtype=int)
156.            else:
157.                if min_idx[0] != 0:
158.                    min_idx = np.insert(min_idx, 0, 0)
159.                if min_idx[-1] != N-1:
160.                    min_idx = np.append(min_idx, N-1)
161.
162.            x_knots_max = max_idx.astype(float)
163.            y_knots_max = x[max_idx]
164.            x_knots_min = min_idx.astype(float)
165.            y_knots_min = x[min_idx]
166.
167.            x_eval = np.arange(N, dtype=float)
168.            env_max = self._cubic_spline_interpolation(x_knots_max, y_knots_max,
    x_eval)
169.            env_min = self._cubic_spline_interpolation(x_knots_min, y_knots_min,
    x_eval)
170.
171.            mean_env = 0.5 * (env_max + env_min)
172.
173.            return {
174.                "env_max": env_max,
175.                "env_min": env_min,
176.                "mean_env": mean_env,
177.                "max_idx": max_idx,
178.                "min_idx": min_idx,
179.            }
```

1. Signal Processing

The preprocessing module prepares the raw PPG/PLETH signal for decomposition by suppressing slow drift and optionally normalizing amplitude. The first operation removes the DC component by subtracting the signal mean, producing a zero-mean signal:

$$x_0[n] = x[n] - mean(x)$$

To reduce baseline wander, the program estimates a slow-varying baseline using a running mean (moving average) computed over a window W samples (derived from baseline_window_s × fs). The baseline is then subtracted:

$$b[n] = (1/W) \cdot \Sigma\,(x0[n-k)),\ and\ x1[n] = x0[n] - b[n]$$

An additional detrending option fits a low-order polynomial trend p(t) to the signal and subtracts it (poly-detrending), improving envelope stability during EMD sifting. Finally, if normalization is enabled, the signal is scaled by its standard deviation:

$$x_{norm}[n] = \frac{x_1[n]}{std(x1)}\ (if\ std(x1) > 0)$$

2. Local Extrema Detection and Envelope Construction

EMD requires identification of local maxima and minima to build upper and lower envelopes. The program detects extrema through sign changes of the discrete derivative $d_x[n] = x[n+1] - x[n]$. A maximum occurs where the slope changes from positive to negative, and a minimum occurs where the slope changes from negative to positive. Endpoints are clamped (index 0 and N−1 are inserted when needed) so that spline interpolation covers the full signal range.

The upper envelope $e_{max}[n]$ is formed by interpolating the maxima points, and the lower envelope $e_{min}[n]$ is formed by interpolating the minima points. A natural cubic spline is implemented without external SciPy dependencies by solving the tridiagonal system for spline second derivatives. The mean envelope is computed as:

$$m[n] = 0.5 \cdot (e_{max}[n] + e_{min}[n])$$

b. IMF Rules + EMD Sifting + Ensemble (EEMD/CEEMD/CEEMDAN)

```
1.      def _count_zero_crossings(self, x: np.ndarray) -> int:
2.          s = np.sign(x)
3.          s[s == 0] = 1
4.          return int(np.sum(s[:-1] * s[1:] < 0))
5.
6.      def _check_imf_condition(self, x: np.ndarray) -> bool:
7.          max_idx, min_idx = self._local_extrema(x)
8.          zc = self._count_zero_crossings(x)
9.          extrema = len(max_idx) + len(min_idx)
10.         return abs(zc - extrema) <= 1
11.
12.     # --------------------------
13.     # EMD (Sifting)
14.     # --------------------------
15.     def emd(self, signal: np.ndarray, max_imfs: int = 10, max_siftings: int =
    50,
16.             epsilon: float = 0.2, consecutive_imf_checks: int = 2) ->
    Tuple[List[np.ndarray], Dict]:
```

```
        """
        Basic EMD with:
          - SD stopping criterion (epsilon)
          - IMF condition (zero-crossings vs extrema)
        """
        x = np.asarray(signal, dtype=float)
        residual = x.copy()
        imfs = []
        meta = {"siftings_per_imf": []}

        for imf_idx in range(max_imfs):
            max_idx, min_idx = self._local_extrema(residual)
            if len(max_idx) + len(min_idx) < 2:
                break

            h_prev = residual.copy()
            consecutive_imf_meets = 0
            used_siftings = 0

            for s in range(max_siftings):
                used_siftings = s + 1
                env_info = self.get_envelopes(h_prev)
                m = env_info["mean_env"]
                h = h_prev - m

                # SD metric
                denom = np.sum(h_prev**2) + 1e-12
                sd = np.sum((h_prev - h)**2) / denom

                # IMF condition
                if self._check_imf_condition(h):
                    consecutive_imf_meets += 1
                else:
                    consecutive_imf_meets = 0

                # stop conditions
                if (epsilon is not None and sd < epsilon) or
    (consecutive_imf_meets >= consecutive_imf_checks):
                    h_prev = h
                    break

                h_prev = h

            imfs.append(h_prev.copy())
            meta["siftings_per_imf"].append(used_siftings)
            residual = residual - h_prev

        return imfs, meta

    # --------------------------
```

```
    # EEMD
    # -------------------------
    def eemd(self, signal: np.ndarray, trials: int = 50, noise_std_ratio: float = 0.2,
             max_imfs: int = 10, max_siftings: int = 50, epsilon: float = 0.2) -> Tuple[List[np.ndarray], Dict]:
        x = np.asarray(signal, dtype=float)
        N = len(x)
        std_sig = np.std(x)
        rng = np.random.default_rng()

        acc_imfs = []
        siftings_acc = np.zeros(max_imfs, dtype=float)

        for t in range(trials):
            noise = rng.normal(0, noise_std_ratio * std_sig, size=N)
            imfs_tr, meta_tr = self.emd(x + noise, max_imfs=max_imfs, max_siftings=max_siftings, epsilon=epsilon)

            # accumulate
            for i, imf in enumerate(imfs_tr):
                if i >= max_imfs:
                    break
                if len(acc_imfs) <= i:
                    acc_imfs.append(np.zeros_like(imf))
                acc_imfs[i] += imf
                siftings_acc[i] += meta_tr["siftings_per_imf"][i] if i < len(meta_tr["siftings_per_imf"]) else 0

        # average
        imfs_avg = []
        for i in range(len(acc_imfs)):
            imf_avg = acc_imfs[i] / trials
            imfs_avg.append(imf_avg)

        meta = {
            "trials": trials,
            "noise_std_ratio": noise_std_ratio,
                "avg_siftings_per_imf": (siftings_acc / trials).tolist()
             }
             return imfs_avg, meta

        # -------------------------
        # CEEMD (Complementary pairs)
        # -------------------------
        def ceemd(self, signal: np.ndarray, trials: int = 50, noise_std_ratio: float = 0.2,
                  max_imfs: int = 10, max_siftings: int = 50, epsilon: float = 0.2) -> Tuple[List[np.ndarray], Dict]:
            x = np.asarray(signal, dtype=float)
```

```
110.            N = len(x)
111.            std_sig = np.std(x)
112.            rng = np.random.default_rng()
113.
114.            acc_imfs = []
115.            siftings_acc = np.zeros(max_imfs, dtype=float)
116.            stored_pairs = []
117.
118.            for t in range(trials):
119.                noise = rng.normal(0, noise_std_ratio * std_sig, size=N)
120.                stored_pairs.append((noise.copy(), -noise.copy()))
121.                for nn in (noise, -noise):
122.                    imfs_tr, meta_tr = self.emd(x + nn, max_imfs=max_imfs,
    max_siftings=max_siftings, epsilon=epsilon)
123.                    for i, imf in enumerate(imfs_tr):
124.                        if i >= max_imfs:
125.                            break
126.                        if len(acc_imfs) <= i:
127.                            acc_imfs.append(np.zeros_like(imf))
128.                        acc_imfs[i] += imf
129.                        siftings_acc[i] += meta_tr["siftings_per_imf"][i] if i <
    len(meta_tr["siftings_per_imf"]) else 0
130.
131.            total_runs = trials * 2
132.            imfs_avg = [acc_imfs[i] / total_runs for i in range(len(acc_imfs))]
133.            meta = {
134.                "trials": trials,
135.                "noise_std_ratio": noise_std_ratio,
136.                "avg_siftings_per_imf": (siftings_acc / total_runs).tolist(),
137.                "stored_noise_pairs": stored_pairs
138.            }
139.            return imfs_avg, meta
140.
141.        # --------------------------
142.        # CEEMDAN (Adaptive noise)
143.        # --------------------------
144.        def ceemdan(self, signal: np.ndarray, trials: int = 50, noise_std_ratio:
    float = 0.2,
145.                    max_imfs: int = 10, max_siftings: int = 50, epsilon: float =
    0.2) -> Tuple[List[np.ndarray], Dict]:
146.            x = np.asarray(signal, dtype=float)
147.            N = len(x)
148.            std_sig = np.std(x)
149.            rng = np.random.default_rng()
150.
151.            imfs = []
152.            stored_noise = {"trials": [], "noise_imfs": [], "added_noise": []}
153.
154.            # 1) IMF1 by averaging IMF1 from x + noise
155.            acc_imf1 = np.zeros(N, dtype=float)
```

```
156.            siftings_acc = []
157.
158.            for t in range(trials):
159.                noise = rng.normal(0, 1.0, size=N)
160.                added = noise_std_ratio * std_sig * noise
161.                stored_noise["trials"].append(noise.copy())
162.                stored_noise["added_noise"].append(added.copy())
163.
164.                imfs_tr, meta_tr = self.emd(x + added, max_imfs=1,
    max_siftings=max_siftings, epsilon=epsilon)
165.                if len(imfs_tr) > 0:
166.                    acc_imf1 += imfs_tr[0]
167.                    siftings_acc.append(meta_tr["siftings_per_imf"][0] if
    meta_tr["siftings_per_imf"] else 0)
168.
169.            imf1 = acc_imf1 / trials
170.            imfs.append(imf1)
171.            residual = x - imf1
172.
173.            meta = {"trials": trials, "noise_std_ratio": noise_std_ratio,
    "siftings_imf1_avg": float(np.mean(siftings_acc)) if siftings_acc else 0.0}
174.
175.            # 2) subsequent IMFs
176.            for k in range(1, max_imfs):
177.                max_idx, min_idx = self._local_extrema(residual)
178.                if len(max_idx) + len(min_idx) < 2:
179.                    break
180.
181.                acc_imfk = np.zeros(N, dtype=float)
182.                siftings_acc_k = []
183.
184.                for t in range(trials):
185.                    noise = rng.normal(0, 1.0, size=N)
186.                    # decompose noise to get kth component
187.                    noise_imfs, _ = self.emd(noise, max_imfs=k+1,
    max_siftings=max_siftings, epsilon=epsilon)
188.                    noise_comp = noise_imfs[k] if len(noise_imfs) > k else
    np.zeros(N)
189.
190.                    added_noise = noise_std_ratio * std_sig * noise_comp
191.                    stored_noise["noise_imfs"].append(noise_comp.copy())
192.
193.                    imfs_res, meta_res = self.emd(residual + added_noise,
    max_imfs=1, max_siftings=max_siftings, epsilon=epsilon)
194.                    if len(imfs_res) > 0:
195.                        acc_imfk += imfs_res[0]
196.                        siftings_acc_k.append(meta_res["siftings_per_imf"][0] if
    meta_res["siftings_per_imf"] else 0)
197.
198.                imfk = acc_imfk / trials
```

```
199.            imfs.append(imfk)
200.            residual = residual - imfk
201.
202.            meta[f"siftings_imf{k+1}_avg"] =
    float(np.mean(siftings_acc_k)) if siftings_acc_k else 0.0
203.
204.        meta["stored_noise"] = stored_noise
205.        return imfs, meta
```

1. IMF Condition and EMD Shifting

The program extracts Intrinsic Mode Functions (IMFs) using the standard EMD sifting procedure. For each IMF, the current residual r[n] is initialized as $h_0[n] = r[n]$. At each sifting iteration k, envelopes are computed from h(k−1), the mean envelope m(k−1) is formed, and the signal is updated by subtracting it:

$$h_k[n] = h_{k-1}[n] - m_{k-1}[n]$$

Two stopping criteria are used. First, the structural IMF condition compares the number of zero crossings ZC with the total number of extrema E (maxima + minima). An IMF is considered valid when |ZC − E| ≤ 1. Second, an SD-like convergence criterion measures the relative change between successive siftings:

$$SD = \frac{\Sigma\,(h_{k-1}[n] - h_k[n])^2}{\Sigma\,(h_{k-1}[n])^2}$$

Sifting stops when SD drops below epsilon, or when the IMF condition is met for a required number of consecutive checks. Once an IMF ci[n] is accepted, the residual is updated and the next IMF is extracted:

$$r[n] \leftarrow r[n] - ci[n]$$

2. Ensemble Decomposition: EEMD, CEEMD, and CEEMDAN

To reduce mode mixing, the program provides three ensemble-based extensions. In EEMD, the signal is perturbed with independent white noise realizations wt[n] scaled by noise_std_ratio × std(signal), EMD is applied to each noisy signal, and corresponding IMFs are averaged across T trials:

$ci[n] = \frac{1}{T} \cdot \Sigma t\ ci, t[n]$ where ci,t is the i-th IMF of (x + wt)

CEEMD improves noise cancellation by using complementary noise pairs (+wt and −wt). It runs EMD on both x+wt and x−wt and averages across 2T runs:

$$ci[n] = \frac{1}{2T} \cdot \Sigma t\ (ci, t + (n) + ci, t - (n))$$

CEEMDAN extracts IMFs sequentially with adaptive noise. IMF1 is obtained by averaging the first IMF extracted from x plus scaled noise. For later stages k, the method decomposes noise to obtain its k-th noise component, scales it, adds it to the current residual, and extracts only one IMF per stage before averaging across trials. This produces a complete decomposition where each stage is noise-assisted but remains linked to the evolving residual.

c. Transform (FFT/DFT/IDFT) + Hilbert/HHT + Resp/Vaso Features

```
1.    def _next_pow2(self, n: int) -> int:
2.        p = 1
3.        while p < n:
4.            p <<= 1
5.        return p
```

```
6.
7.      def _bit_reverse_permutation(self, n: int) -> np.ndarray:
8.          bits = int(np.log2(n))
9.          rev = np.zeros(n, dtype=int)
10.         for i in range(n):
11.             b = format(i, f"0{bits}b")[::-1]
12.             rev[i] = int(b, 2)
13.         return rev
14.
15.     def fft(self, x: np.ndarray) -> np.ndarray:
16.         x = np.asarray(x, dtype=complex); n0 = x.shape[0]; N = self._next_pow2(n0)
17.         if N != n0: x = np.concatenate([x, np.zeros(N - n0, dtype=complex)])
18.         perm = self._bit_reverse_permutation(N); X = x[perm].copy(); m = 1
19.         while m < N:
20.             step = m * 2; ang = -2j * np.pi / step; W_m = np.exp(ang * np.arange(m))
21.             for k in range(0, N, step):
22.                 t = W_m * X[k + m:k + step]; u = X[k:k + m]; X[k:k + m] = u + t; X[k +
    m:k + step] = u - t
23.             m = step
24.         return X
25.
26.     def dft(self, x: np.ndarray) -> Tuple[np.ndarray, np.ndarray]:
27.         x = np.asarray(x, dtype=complex); N0 = x.shape[0]
28.         if self.transform_method == "FFT":
29.             Xfull = self.fft(x)
30.         else:
31.             n = np.arange(N0); k = n.reshape((N0, 1)); exp = np.exp(-2j * np.pi * k * n
    / N0); Xfull = exp.dot(x)
32.         freqs = np.fft.fftfreq(len(Xfull), d=1.0/self.fs)
33.         return freqs, Xfull
34.
35.     def idft(self, Xfull: np.ndarray) -> np.ndarray:
36.         Xfull = np.asarray(Xfull, dtype=complex); N = Xfull.shape[0]
37.         n = np.arange(N); k = n.reshape((N, 1))
38.         exp = np.exp(2j * np.pi * k * n / N)
39.         x_recon = exp.dot(Xfull) / N
40.         return x_recon
41.
42.     def psd_from_spectrum(self, Xk: np.ndarray, N: int) -> np.ndarray:
43.         return (np.abs(Xk) ** 2) / N
44.
45.     def analytic_signal(self, x: np.ndarray) -> np.ndarray:
46.         x = np.asarray(x, dtype=float); N0 = len(x)
47.         freqs, Xfull = self.dft(x); N = len(Xfull); H = np.zeros(N, dtype=complex);
    H[:] = Xfull
48.         if N % 2 == 0:
49.             H[1:N//2] *= 2.0; H[N//2+1:] = 0.0
50.         else:
51.             H[1:(N+1)//2] *= 2.0; H[(N+1)//2:] = 0.0
52.         x_analytic_full = self.idft(H); return x_analytic_full[:N0]
53.
54.     def inst_amplitude_phase_freq(self, x: np.ndarray) -> Tuple[np.ndarray,
    np.ndarray, np.ndarray]:
55.         z = self.analytic_signal(x); a = np.abs(z); phase = np.unwrap(np.angle(z));
```

```
    dphase = np.gradient(phase); inst_freq = (dphase / (2.0 * np.pi)) * self.fs; return a,
    phase, inst_freq
56.
57.     def _gaussian_kernel_1d(self, sigma: float, truncate: float = 3.0) -> np.ndarray:
58.         if sigma <= 0: return np.array([1.0])
59.         radius = int(max(1, np.ceil(truncate * sigma))); x = np.arange(-radius,
    radius+1); kernel = np.exp(-0.5 * (x / sigma) ** 2); kernel = kernel /
    np.sum(kernel); return kernel
60.
61.     def _gaussian_blur_2d(self, Z: np.ndarray, sigma_t: float, sigma_f: float) ->
    np.ndarray:
62.         Z = Z.astype(float)
63.         kt = self._gaussian_kernel_1d(sigma_t); kf =
    self._gaussian_kernel_1d(sigma_f)
64.         # convolve along time (axis=1)
65.         Zt = np.apply_along_axis(lambda m: np.convolve(m, kt, mode='same'),
    axis=1, arr=Z)
66.         # convolve along freq (axis=0)
67.         Ztf = np.apply_along_axis(lambda m: np.convolve(m, kf, mode='same'),
    axis=0, arr=Zt)
68.         return Ztf
69.
70.     def hht_spectrogram(self, imf: np.ndarray, f_min: float = 0.0, f_max: float = 5.0,
71.                         n_freq_bins: int = 128, smooth_sigma_t: float = 1.0,
    smooth_sigma_f: float = 1.0) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:
72.         a, phase, inst_f = self.inst_amplitude_phase_freq(imf)
73.         # clamp frequency range
74.         f_edges = np.linspace(f_min, f_max, n_freq_bins+1)
75.         f_centers = 0.5 * (f_edges[:-1] + f_edges[1:])
76.         T = len(imf)
77.         Z = np.zeros((n_freq_bins, T), dtype=float)
78.
79.         # bin inst freq
80.         idx = np.digitize(inst_f, f_edges) - 1
81.         valid = (idx >= 0) & (idx < n_freq_bins)
82.         for t in np.where(valid)[0]:
83.             Z[idx[t], t] += a[t]
84.
85.         # smoothing
86.         if smooth_sigma_t > 0 or smooth_sigma_f > 0:
87.             Z = self._gaussian_blur_2d(Z, sigma_t=smooth_sigma_t,
    sigma_f=smooth_sigma_f)
88.         t_axis = np.arange(T) / self.fs
89.         return t_axis, f_centers, Z
90.
91.     # --------------------------
92.     # Extraction
93.     # --------------------------
94.     def extract_resp_vaso(self, imfs: List[np.ndarray], resp_band: Tuple[float, float]
    = (0.15, 0.4), vaso_band: Tuple[float, float] = (0.04, 0.15)) -> Dict:
95.         resp_energies = []; vaso_energies = []
96.         resp_peak_freqs = []; vaso_peak_freqs = []
97.
98.         for imf in imfs:
```

```
            freqs, Xk = self.dft(imf)
                psd = self.psd_from_spectrum(Xk, len(imf))
                half = len(freqs)//2 + 1
                freqs_half = freqs[:half]; psd_half = psd[:half]

                resp_mask = (freqs_half >= resp_band[0]) & (freqs_half <=
    resp_band[1])
                vaso_mask = (freqs_half >= vaso_band[0]) & (freqs_half <=
    vaso_band[1])

                resp_energy = np.sum(psd_half[resp_mask]) if np.any(resp_mask) else
    0.0
                vaso_energy = np.sum(psd_half[vaso_mask]) if np.any(vaso_mask)
    else 0.0

                if np.any(resp_mask) and np.sum(psd_half[resp_mask]) > 0:
                    idx_peak = np.argmax(psd_half[resp_mask])
                    resp_peak_freqs.append(freqs_half[resp_mask][idx_peak])
                else: resp_peak_freqs.append(0.0)

                if np.any(vaso_mask) and np.sum(psd_half[vaso_mask]) > 0:
                    idx_peak = np.argmax(psd_half[vaso_mask])
                    vaso_peak_freqs.append(freqs_half[vaso_mask][idx_peak])
                else: vaso_peak_freqs.append(0.0)

                resp_energies.append(resp_energy);
    vaso_energies.append(vaso_energy)

            resp_idx = int(np.argmax(resp_energies)) if len(resp_energies) else -1
            vaso_idx = int(np.argmax(vaso_energies)) if len(vaso_energies) else -1

            resp_freq_hz = float(resp_peak_freqs[resp_idx]) if resp_idx >= 0 else 0.0
            vaso_freq_hz = float(vaso_peak_freqs[vaso_idx]) if vaso_idx >= 0 else
    0.0

            resp_bpm = resp_freq_hz * 60.0

            total_energy = float(np.sum(resp_energies) + np.sum(vaso_energies)) +
    1e-12
            resp_energy_norm = float(resp_energies[resp_idx] / total_energy) if
    resp_idx >= 0 else 0.0
            vaso_energy_norm = float(vaso_energies[vaso_idx] / total_energy) if
    vaso_idx >= 0 else 0.0

            return {
                "resp_imf_index": resp_idx,
                "vaso_imf_index": vaso_idx,
                "respiratory_freq_hz": resp_freq_hz,
                "vasomotor_freq_hz": vaso_freq_hz,
                "respiratory_rate_bpm": resp_bpm,
                "resp_energy_norm": resp_energy_norm,
                "vaso_energy_norm": vaso_energy_norm,
                "resp_energies": resp_energies,
                "vaso_energies": vaso_energies
```

```
144.        }
```

1. Frequency Analysis: FFT/DFT, PSD, and Analytic Signal

For each IMF, the program computes a frequency representation using either an internal FFT implementation or a direct DFT matrix multiplication, controlled by the transform_method setting. From the complex spectrum X[k], the power spectral density is estimated as:

$$PSD[k] = |X[k]|^2 / N$$

To support Hilbert-based instantaneous analysis, an analytic signal z[n] is constructed in the frequency domain by doubling positive-frequency bins, keeping DC (and Nyquist when applicable), and zeroing negative-frequency bins before applying the inverse transform. This yields a complex analytic signal whose magnitude and phase encode instantaneous amplitude and phase information.

2. Instantaneous Amplitude and Frequency, and HHT Spectrogram

Given the analytic signal z[n], instantaneous amplitude is a[n] = |z[n]|. Instantaneous phase is obtained from the angle of z[n] and unwrapped over time. Instantaneous frequency is then computed using the time-derivative (numerical gradient) of the unwrapped phase:

$$f_{inst}[n] = \frac{1}{2\pi} \cdot \frac{d(phase[n])}{dn} \cdot fs$$

The Hilbert–Huang Transform (HHT) spectrogram is formed by binning instantaneous frequencies into a predefined frequency grid (f_min to f_max) and accumulating amplitude into the corresponding bin for each time sample. Optional Gaussian smoothing is applied in both time and frequency directions to improve readability. This produces a time–frequency amplitude map that reveals non-stationary behavior that FFT/PSD can hide.

3. Respiration and Vasomotor Feature Extraction

The feature extraction module searches across IMFs to identify the components that best represent two low-frequency physiological phenomena: respiration and vasomotor activity. For each IMF, the program computes PSD and integrates band power inside two fixed bands:

Respiration band: 0.15–0.40 Hz | Vasomotor band: 0.04–0.15 Hz

For IMF i, the band energies are computed as sums over PSD bins within each band:
E_resp(i) = Σ PSD_i[k] over k in respiration band, E_vaso(i) = Σ PSD_i[k] over k in vasomotor band

The representative IMF indices are selected by maximum band energy: i_resp = argmax E_resp(i) and i_vaso = argmax E_vaso(i). Peak frequency within each band is taken at the PSD maximum inside the corresponding band, and respiratory rate is reported in BPM as BPM = 60 × f_resp.

To stabilize comparisons across signals, energies are normalized by total low-frequency energy (sum of respiration and vasomotor energies across IMFs), producing relative features (resp_energy_norm and vaso_energy_norm).

d. GUI Worker Thread Pipeline (core run loop)

```
1.      def __init__(self, proc: PPGProcessor, signal: np.ndarray, methods: list,
    max_imfs: int = 10,
2.                  normalize_preproc: bool = True, trials: int = 50, noise_ratio: float = 0.2,
3.                  epsilon: float = 0.2, parent=None):
4.          super().__init__(parent)
5.          self.proc = proc
6.          self.signal = signal
7.          self.methods = methods
8.          self.max_imfs = max_imfs
```

```
9.          self.normalize_preproc = normalize_preproc
10.         self.trials = trials
11.         self.noise_ratio = noise_ratio
12.         self.epsilon = epsilon
13.
14.     def run(self):
15.         out = {}
16.         sig = self.signal
17.         self.progress.emit("Preprocessing signal...")
18.         pre = self.proc.preprocess(sig, normalize=self.normalize_preproc,
    remove_baseline=True, baseline_window_s=2.0)
19.         out["preproc"] = pre
20.         out["preproc_std"] = self.proc.preproc_std()
21.         imfs_all = {}
22.         meta_all = {}
23.         for method in self.methods:
24.             self.progress.emit(f"Running {method} decomposition...")
25.             if method == "EMD":
26.                 imfs, meta = self.proc.emd(pre, max_imfs=self.max_imfs,
    epsilon=self.epsilon)
27.             elif method == "EEMD":
28.                 imfs, meta = self.proc.eemd(pre, trials=self.trials,
    noise_std_ratio=self.noise_ratio,
29.                                             max_imfs=self.max_imfs, epsilon=self.epsilon)
30.             elif method == "CEEMD":
31.                 imfs, meta = self.proc.ceemd(pre, trials=self.trials,
    noise_std_ratio=self.noise_ratio,
32.                                              max_imfs=self.max_imfs, epsilon=self.epsilon)
33.             elif method == "CEEMDAN":
34.                 imfs, meta = self.proc.ceemdan(pre, trials=self.trials,
    noise_std_ratio=self.noise_ratio,
35.                                               max_imfs=self.max_imfs, epsilon=self.epsilon)
36.             else:
37.                 continue
38.
39.             # pad to fixed number of IMFs for consistent GUI page/plot
40.             N = len(pre)
41.             while len(imfs) < self.max_imfs:
42.                 imfs.append(np.zeros(N))
43.             imfs = imfs[:self.max_imfs]
44.
45.             imfs_all[method] = imfs
46.             meta_all[method] = meta
47.
48.         # choose "preferred" output for final metrics (CEEMDAN > CEEMD >
    EEMD > EMD)
49.         preferred = None
50.         for m in ["CEEMDAN", "CEEMD", "EEMD", "EMD"]:
51.             if m in imfs_all:
52.                 preferred = m
53.                 break
54.
55.         if preferred is None:
56.             out["metrics"] = {"error": "No decomposition method ran."}
```

```
        else:
            metrics = self.proc.extract_resp_vaso(imfs_all[preferred])
            metrics["preferred_method"] = preferred
            out["metrics"] = metrics

        # precompute per-IMF transforms & HHT maps
        analysis = {}
        for method, imfs in imfs_all.items():
            res_list = []
            for imf in imfs:
                freqs, Xk = self.proc.dft(imf)
                psd = self.proc.psd_from_spectrum(Xk, len(imf))
                half = len(freqs) // 2 + 1
                freqs_half = freqs[:half]
                psd_half = psd[:half]

                t_axis, f_axis, Z = self.proc.hht_spectrogram(imf, f_min=0.0,
    f_max=5.0, n_freq_bins=128,
                                                  smooth_sigma_t=1.0, smooth_sigma_f=1.0)
                res_list.append({
                    "freqs": freqs_half,
                    "psd": psd_half,
                    "t_axis": t_axis,
                    "f_axis": f_axis,
                    "hht": Z
                })
            analysis[method] = res_list

        out["imfs_all"] = imfs_all
        out["meta_all"] = meta_all
        out["analysis"] = analysis
        self.finished.emit(out)
```

The GUI acts as a front-end to configure parameters (sampling rate, method selection, number of IMFs, sifting limits, epsilon, trials, and noise ratio) and to visualize outputs (raw/preprocessed signals, IMF overlays, per-IMF FFT/PSD/HHT, and final metrics). Heavy computations run in a worker thread to keep the interface responsive. After preprocessing, the selected decomposition method is executed, IMFs are padded to a fixed count for consistent plotting, per-IMF analyses are computed, and the final respiration–vasomotor metrics are displayed in a summary panel.

e. GUI Metrics Renderer (general output)

```
    def render_tab_metrics(self):
        if self.results is None: return
        self.clear_tab("METRICS")
        metrics = self.results.get("metrics", {})

        text = "<h3>Analysis Results</h3>"

        text += f"<b>Respiratory Rate (BPM):</b>
    {metrics.get('respiratory_rate_bpm','N/A'):.2f}<br>"
        text += f"<b>Respiratory Freq (Hz):</b>
    {metrics.get('respiratory_freq_hz','N/A'):.4f}<br>"
        text += f"<b>Resp IMF Index:</b>
```

```
    {metrics.get('resp_imf_index','N/A')}<br>"
11.     text += f"<b>Resp Energy (Norm):</b>
    {metrics.get('resp_energy_norm','N/A'):.4f}<br><br>"
12.
13.     text += f"<b>Vasomotor Freq (Hz):</b>
    {metrics.get('vasomotor_freq_hz','N/A'):.4f}<br>"
14.     text += f"<b>Vasomotor IMF Index:</b>
    {metrics.get('vaso_imf_index','N/A')}<br>"
15.     text += f"<b>Vasomotor Energy (Norm):</b>
    {metrics.get('vaso_energy_norm','N/A'):.4f}<br>"
16.
17.     if "error" in metrics:
18.         text += f"<br><font color='red'>Error: {metrics['error']}</font>"
19.
20.     lbl = QLabel(text); lbl.setWordWrap(True)
21.     self.tab_containers["METRICS"].lawet().addWidget(lbl)
```

The render_tab_metrics function serves as the final presentation layer of the application, bridging the gap between complex background computations and user-readable information. Once the worker thread completes the decomposition and extraction pipeline, this function retrieves the computed metrics dictionary from the results object. It utilizes HTML-based string formatting to construct a structured summary, dynamically populating fields such as the calculated Respiratory Rate (in both BPM and Hz), the identified Vasomotor Frequency, and the specific IMF indices used for these extractions. Additionally, it displays the normalized energy contributions (resp_energy_norm and vaso_energy_norm), providing the user with immediate insight into the spectral dominance of the physiological components relative to the total low-frequency energy. The formatted text is then rendered onto a PyQt QLabel widget within the "METRICS" tab, ensuring clear and accessible communication of the final diagnostic parameters without requiring the user to interpret raw arrays or complex graphs.

2.3. Result of the Program and Analysis

2.3.1. DWT Program

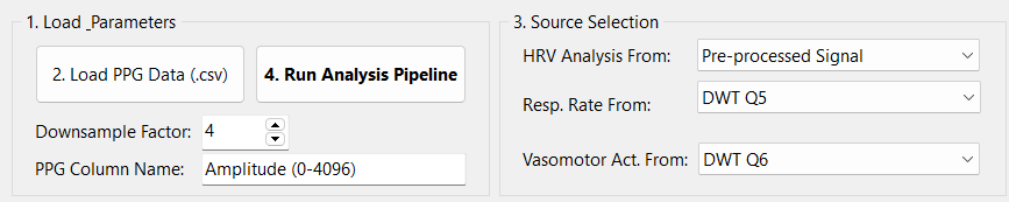


Figure 2.1. DWT Parameters Details

Figure 2.1 illustrates the initial configuration panel of the DWT analyzer software. This interface allows the user to define critical parameters before the analysis begins. The "Downsample Factor" is set (default is 4) to reduce the computational load while preserving the Nyquist frequency required for heart rate analysis. The "PPG Column Name" input ensures the software correctly parses the CSV file, looking for the specific header defining the signal amplitude. Furthermore, the "Source Selection" group box allows the user to choose specific signal components for different physiological extractions; for instance, the Heart Rate Variability (HRV) analysis uses the pre-processed signal, while the Respiratory Rate and Vasomotor activity can be derived from specific DWT decomposition levels (e.g., Q1 through Q8) or the raw signal, providing flexibility in the analysis pipeline.

a. Signal Processing

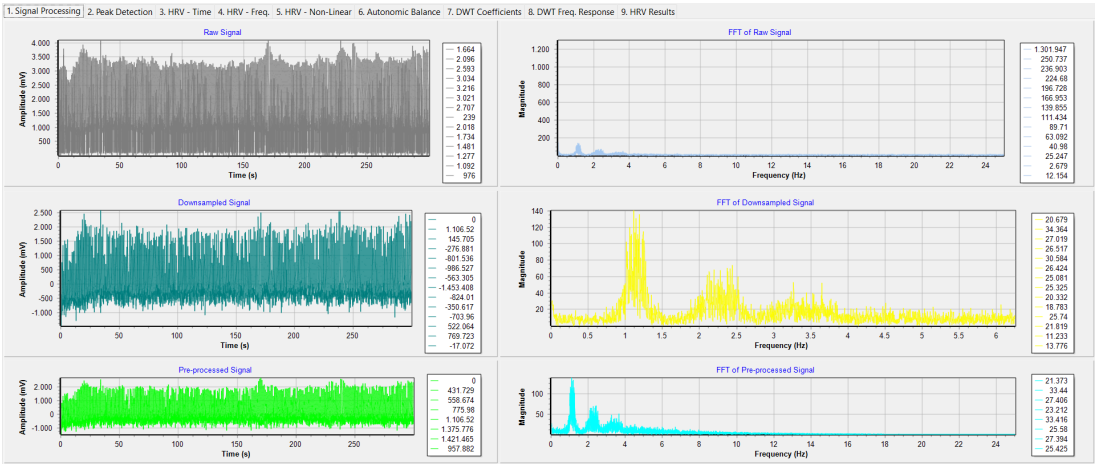


Figure 2.2. Signal Processing Tab Result

Figure 2.2 displays the three stages of signal conditioning implemented in the software. The top graph shows the Raw Signal, which typically contains DC offset drifts and high-frequency noise. The middle graph shows the Downsampled Signal, where the number of data points is reduced based on the user's input factor, effectively normalizing the sampling rate for the filters. The bottom graph represents the Pre-processed Signal, which is the result of removing the DC offset (zero-centering) and applying the Bandpass Filter (0.01 Hz – 8.0 Hz). To the right of each time-domain plot, the Fast Fourier Transform (FFT) spectrum is displayed. These FFT plots validate the filtering process; specifically, the FFT of the pre-processed signal shows a reduction in spectral leakage and noise compared to the raw signal's spectrum, confirming that the signal is clean enough for peak detection.

b. DWT Filter Coefficients and Its Frequency Response

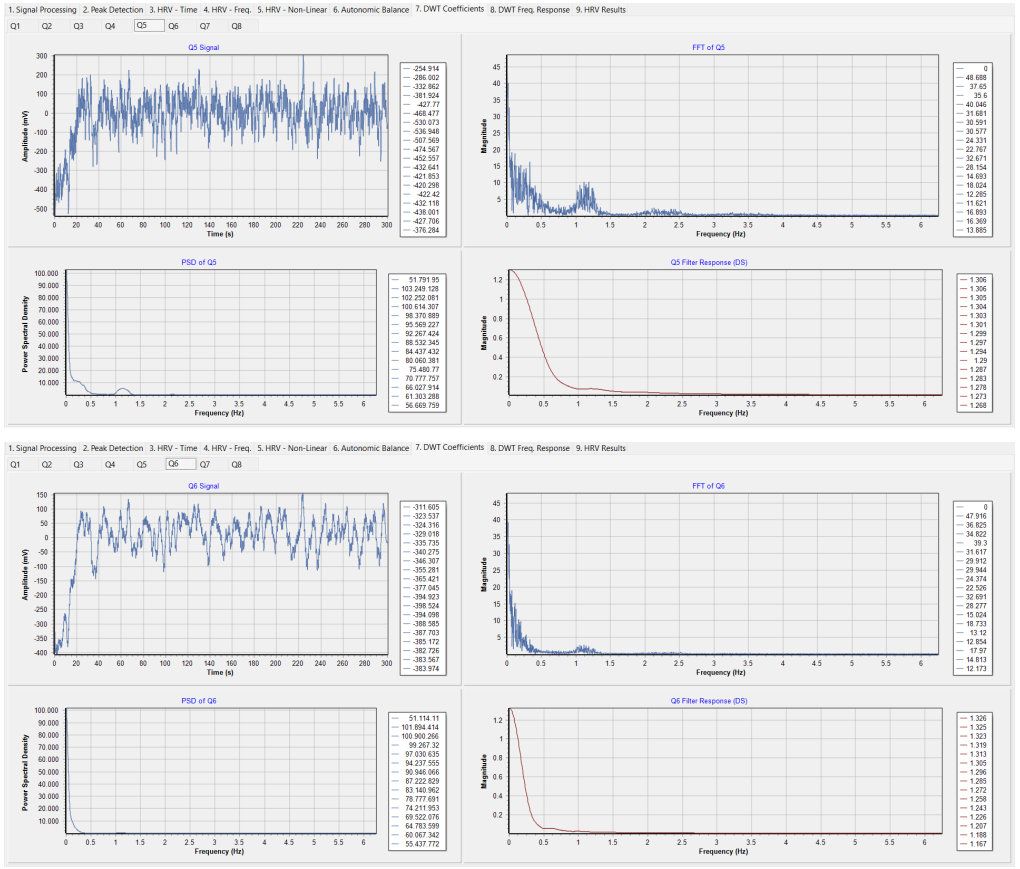


Figure 2.3. DWT Filter Coefficients Output for Q5 and Q6

Figure 2.3 visualizes the time-domain outputs of the Discrete Wavelet Transform (DWT) at specific decomposition levels, specifically Q5 and Q6. In the context of the maximal overlap DWT implementation used in the code, each "Q" level corresponds to a signal filtered at a specific scale. Q5 and Q6 typically capture mid-to-low frequency oscillations. The plots show the amplitude of these coefficients over time. These components are critical because they separate the composite PPG signal into distinct frequency layers; for example, lower Q levels (like Q1-Q2) capture noise and sharp details, while mid-range levels (like Q5-Q6) often contain the clean dicrotic notch information and the primary heart beat wave shape, free from baseline wander.

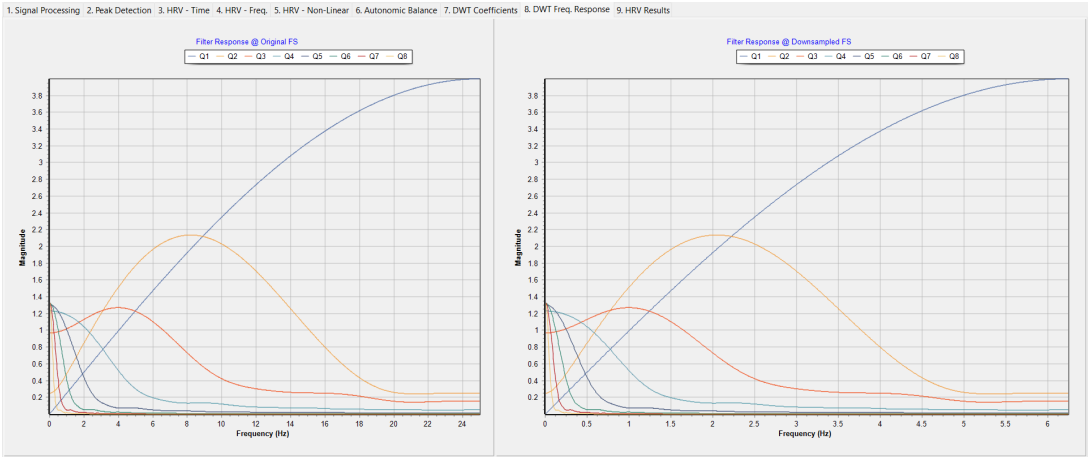


Figure 2.4. DWT Filter Coefficients Frequency Responses for Both Original and Downsampled Version

Figure 2.4 demonstrates the "Filter Bank" characteristic of the DWT algorithm. The plots show the frequency response (Magnitude vs. Frequency) for all 8 decomposition levels (Q1 to Q8).

- Original FS (Frequency Sampling) shows the response relative to the raw sampling rate.
- Downsampled FS shows the response relative to the reduced sampling rate.

It can be observed that Q1 represents the highest frequency band, and as the index increases to Q8, the passband shifts toward lower frequencies. This visualization justifies the selection of specific Q levels for physiological extraction; for instance, Q7 or Q8 are shown to dominate the Very Low Frequency (VLF) region, making them suitable candidates for extracting Respiration (0.15–0.4 Hz) or Vasomotor (0.04–0.15 Hz) activity, while Q4 or Q5 are aligned with the main heart rate frequency.

c. Peak Detection

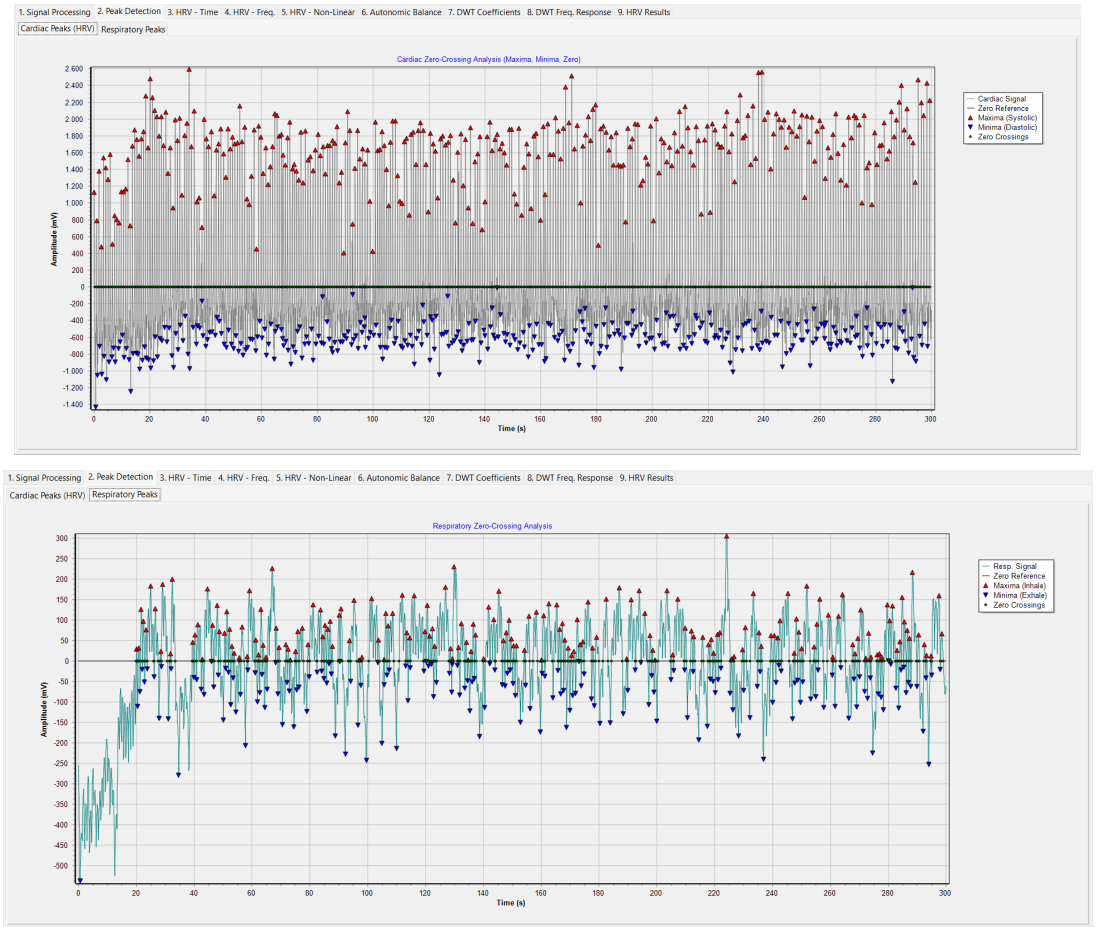
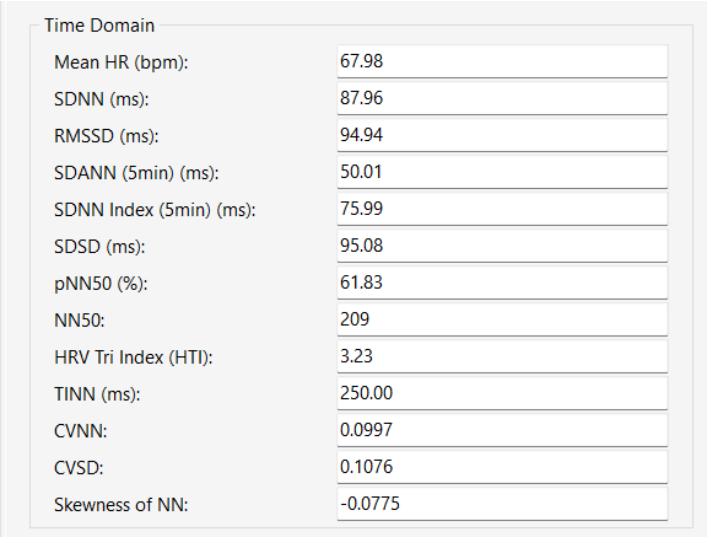


Figure 2.5. Peak Detection Result Tabs

Figure 2.5 presents the results of the Zero-Crossing algorithm applied to the signal. The Cardiac Peaks (Left/Top) identifies the systolic peaks (indicated by red triangles) and diastolic troughs (blue triangles) on the cardiac signal. The green circles indicate the zero-crossing points used to segment the wave. The precise timing of these red peaks is used to calculate the RR intervals.

Respiratory Peaks (Right/Bottom) applies a similar detection algorithm but to the derived respiratory signal (typically filtered below 0.4 Hz). The peaks here represent the end of an inhalation cycle. By counting these peaks over time, the software calculates the respiration rate (Breaths Per Minute).

d. HRV Parameter Outputs in Time Domain



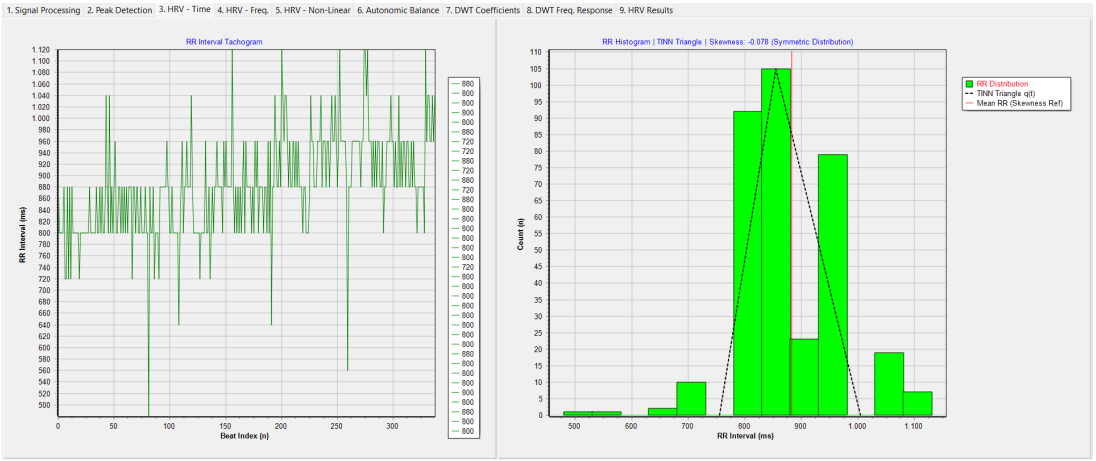
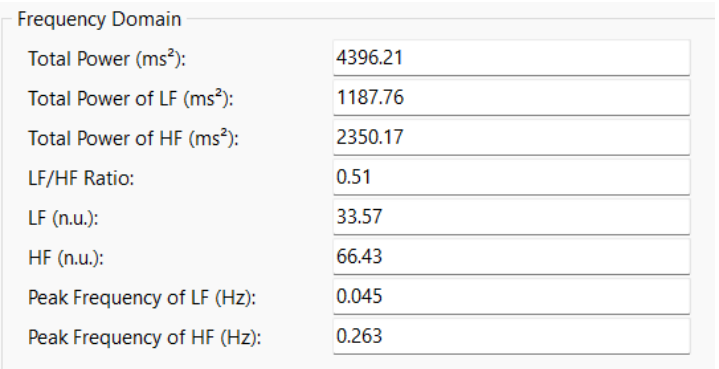


Figure 2.6. Time Domain Parameters Tab Results

Figure 2.6 summarizes the statistical analysis of the RR intervals in the Time Domain. The panel displays standard metrics such as Mean HR (Heart Rate), SDNN (Standard Deviation of NN intervals), and RMSSD (Root Mean Square of Successive Differences). A key visualization in this figure is the RR Interval Histogram (green bars) overlaid with the TINN Triangle (black dashed lines). The histogram shows the distribution of heartbeats; a wider distribution (higher SDNN) generally indicates a healthy heart with high variability, while a narrow distribution indicates stress. The triangular interpolation (TINN) provides a geometric measure of this variability width.

e. HRV Parameter Outputs in Frequency Domain



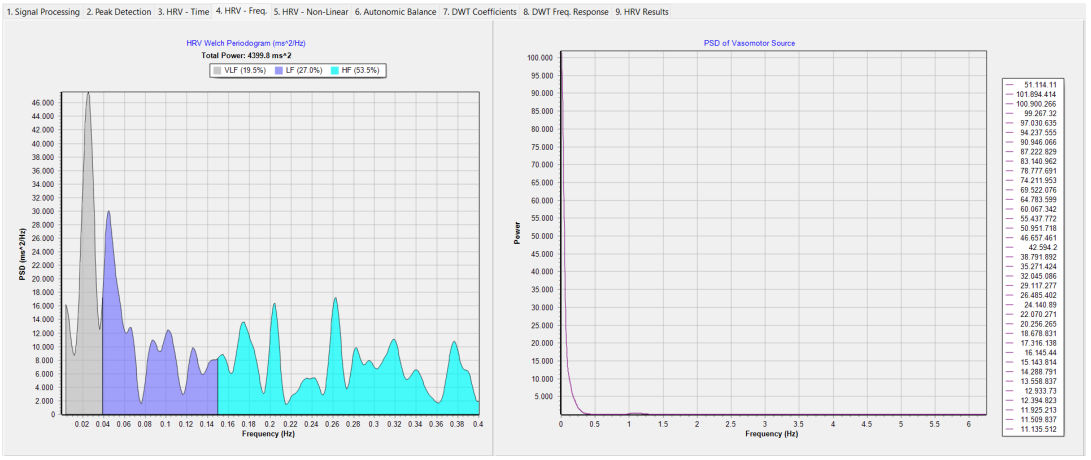


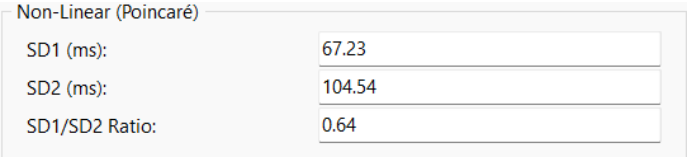
Figure 2.7. Frequency Domain Parameters Tab Results

Figure 2.7 illustrates the spectral power distribution of the HRV data. The main chart shows the Power Spectral Density (PSD) calculated using Welch's method. The area under the curve is divided into three colored zones:

- VLF (Grey): Very Low Frequency (0.0033–0.04 Hz).
- LF (Purple/Red): Low Frequency (0.04–0.15 Hz), associated with sympathetic and baroreflex activity.
- HF (Cyan): High Frequency (0.15–0.40 Hz), associated with parasympathetic (vagal) activity.

The data panel on the right quantifies the Total Power and the LF/HF Ratio. A high LF/HF ratio typically indicates a state of stress or sympathetic dominance, while a low ratio indicates relaxation.

f. HRV Parameter Outputs in Non-Linear Method



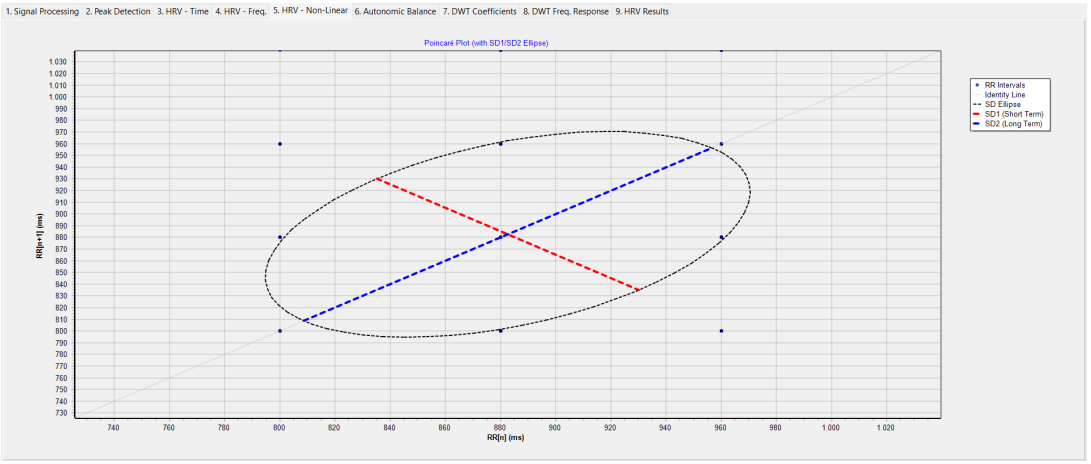


Figure 2.8. Non-linear Domain (Poincare Plot) Parameters Tab Results

Figure 2.8 displays the Poincaré Plot, a scatter plot where each RR interval (RR_n) is plotted against the next interval (RR_n+1). The software fits an ellipse to this cloud of points. SD1 (Width) represented by the red cross-hair, this measures short-term heart rate variability, correlating with parasympathetic control. SD2 (Length) represented by the blue cross-hair along the identity line, measuring long-term variability. The shape of this plot provides a quick visual

assessment of cardiac health like a torpedo or comet shape is normal, whereas a localized ball shape may indicate reduced HRV due to stress or pathology.

g. Autonomic Balance Diagram

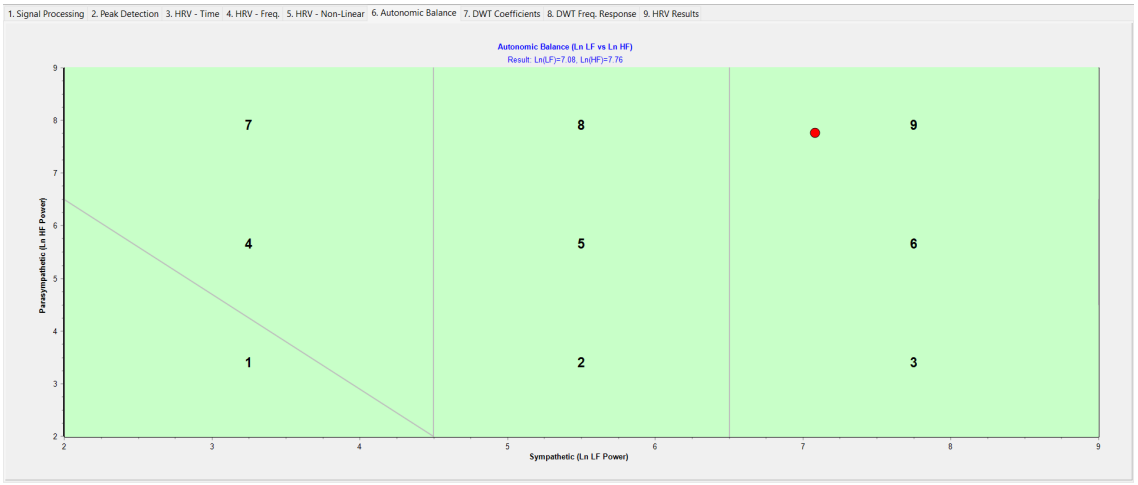


Figure 2.9. Autonomic Balance Diagram Tab Results

Figure 2.9 presents the Autonomic Balance Diagram, a sophisticated 2D visualization of the subject's nervous system state. The X-axis represents the Sympathetic tone (using the natural log of LF Power, ln(LF)), and the Y-axis represents the Parasympathetic tone (ln(HF)).

The chart is divided into 9 colored zones ranging from low to high activity. The red dot represents the specific patient's result.

- Lower-Right Zone: Indicates high Sympathetic/Low Parasympathetic (High Stress).
- Upper-Left/Center Zone: Indicates high Parasympathetic activity (Recovery/Relaxation).
- Center: Indicates a balanced autonomic state.

This diagram synthesizes the frequency domain data into a single, clinically interpretable point regarding the subject's stress level.

2.3.2. EMD Program

a. Original PPG Signal and Preprocessing

Figure 2.10. Original FFT (Raw vs. Preprocessed + Spectrum)

In the Original FFT tab, the upper plot overlays Raw PLETH and the Preprocessed signal. Visually, the preprocessed waveform keeps the main pulsatile morphology but is more "centered" (reduced offset/baseline effect), meaning the preprocessing stage is primarily correcting slow drift and scaling without destroying the physiological pulse shape. The lower plot shows the Transform Magnitude in the range 0–5 Hz. The spectrum is not a single clean

peak; instead it contains multiple peaks and a broadband floor. This indicates the signal contains:

- a dominant periodic component (heart-related rhythm and/or harmonics)
- additional energy from noise / motion artifacts / transient spikes, which spread energy across frequencies rather than concentrating it in one bin.

Figure 2.11. Preprocessing (Baseline Estimation + Detrending + Final Preprocessed)

In the Preprocessing stage, the main objective is to transform the raw PPG/PLETH waveform into a signal that is more stable and more suitable for EMD decomposition. In the upper plot, the Raw signal shows a large-amplitude waveform with noticeable slow variations, while the Estimated baseline curve evolves smoothly and much more slowly. This baseline represents very-low-frequency drift (baseline wander) that commonly arises from sensor–skin coupling changes, gradual motion, finger pressure variation, or analog DC offset. Because this slow drift is not the main physiological oscillation of interest, the program first estimates the baseline (in wer implementation, using a running-mean / moving-average style trend estimate) and subtracts it from the raw signal. As a result, the waveform retains the pulse morphology, but the long-term "lift" and "drop" of the signal is significantly reduced, which prevents the decomposition from being dominated by non-physiological drift.

In the lower plot, the Poly-detrended trace reflects an additional step where a polynomial trend is fitted across time and removed. Conceptually, polynomial detrending models the global trend as a smooth function of time and subtracts it so that the signal becomes flatter in the long-term sense. This step is important because EMD relies heavily on the distribution of local maxima and minima and on the construction of upper and lower envelopes. If a large global trend remains, the envelope curves can become biased, and the sifting process may either extract the trend too early or mix low-frequency components into IMFs where they do not belong (a typical EMD issue known as mode mixing). By reducing the global trend before decomposition, the envelopes computed during EMD become more representative of true oscillatory content rather than slow drift.

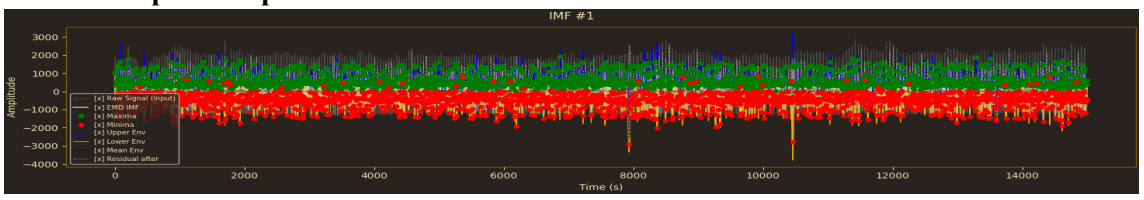
After baseline removal and detrending, the Final preproc signal appears more centered (closer to a zero-mean behavior) and emphasizes oscillatory components more clearly than the raw signal. However, the waveform still contains sharp spikes or impulsive excursions in several segments. This indicates that the preprocessing pipeline is primarily designed to remove slow components, not to fully eliminate fast impulsive artifacts. In practice, this is acceptable in many EMD-based workflows because high-frequency spikes and rapid irregularities are typically captured by early IMFs (like IMF1–IMF3). Nevertheless, if such spikes are strong, they can increase the number of extrema dramatically, make the envelopes fluctuate more aggressively, and potentially increase sifting difficulty or reduce the interpretability of the first IMFs.

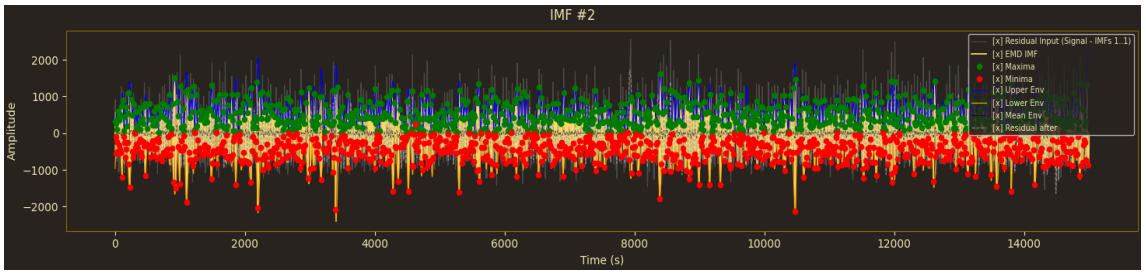
The parameter readout at the bottom of the interface provides important context for how preprocessing will influence the later EMD results. A relatively large Preprocessed STDEV indicates that substantial variability remains after preprocessing (often due to a combination of strong pulsation amplitude and residual artifacts). The status Normalize: False means the signal

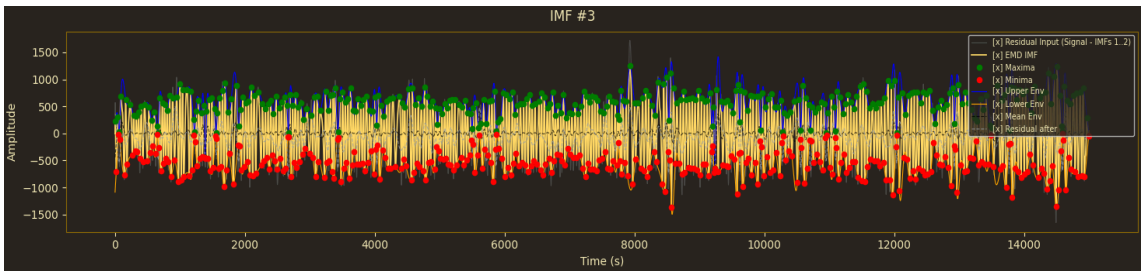
amplitude is not standardized by its standard deviation, so large-amplitude components (including artifacts) still dominate the energy scale during EMD envelope computation and stopping criteria. At the same time, leaving normalization off can be useful if we want to preserve amplitude relationships for later energy-based comparison. Finally, Max sifts (UI): 20 and Epsilon: 0.2 define the sifting iteration limit and the SD-based convergence threshold; preprocessing that successfully suppresses baseline/trend typically helps the sifting process converge more consistently and yields cleaner IMFs for subsequent FFT/PSD/HHT analysis and respiration–vasomotor feature extraction.

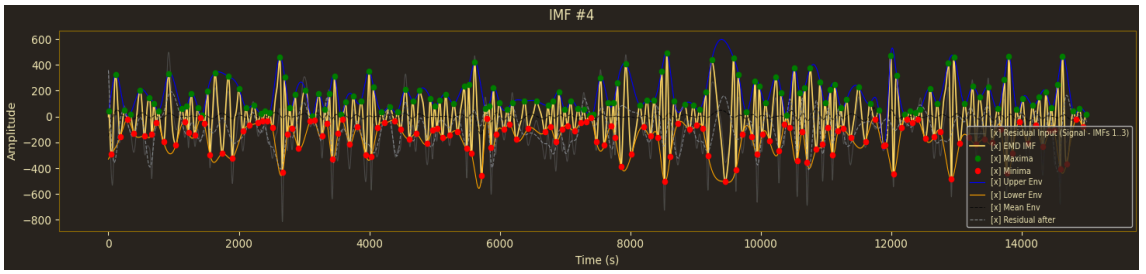
Overall, the preprocessing result in wer figure shows that the pipeline effectively (1) suppresses baseline wander through baseline estimation and subtraction and (2) reduces global trend using polynomial detrending, producing a signal that preserves the essential pulsatile structure while being better conditioned for EMD. The remaining fast spikes are expected to appear mainly in the first few IMFs, which will be addressed implicitly by the decomposition and subsequent band-based feature selection.

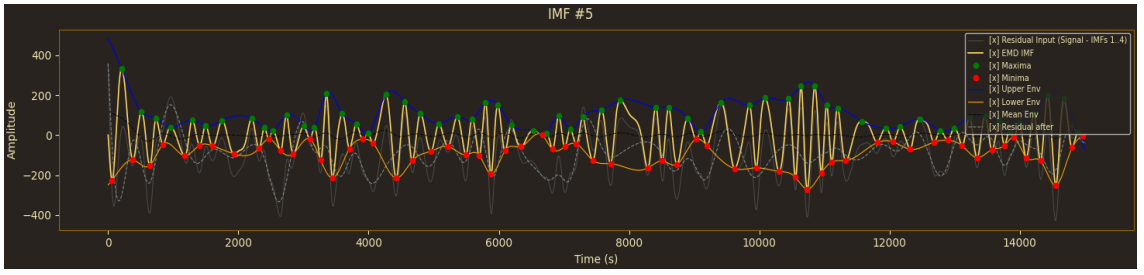
b. IMF Output Comparisons











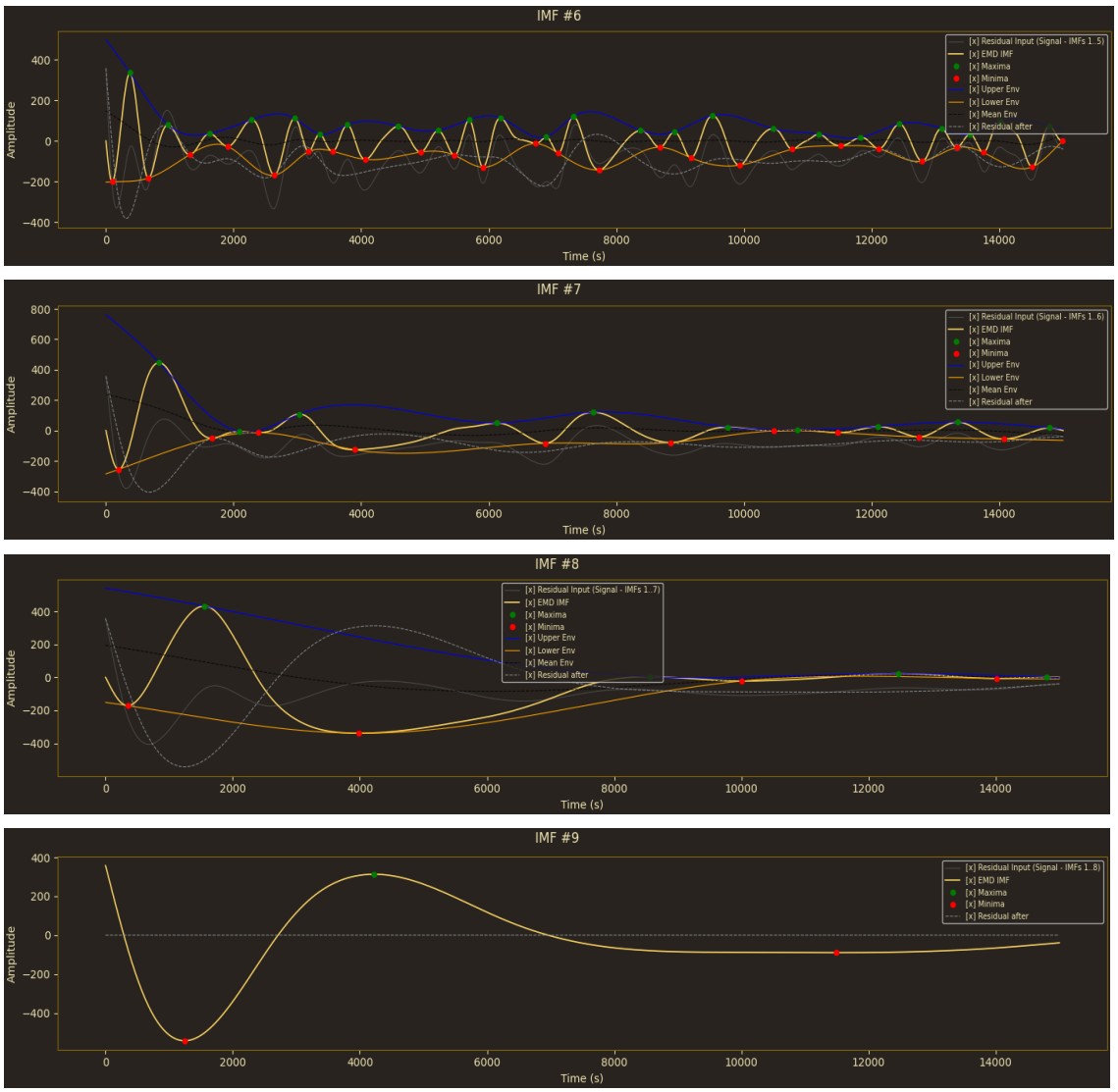


Figure 2.12. IMF Comparisons (IMF 1–9 Combined Interpretation)

In the IMF comparison figures (IMF #1 to IMF #9), the plots are essentially showing the core mechanics of EMD in a visual way: at each stage, the algorithm takes the current input (either the raw/preprocessed signal for IMF1, or the remaining residual for later IMFs), finds all local maxima and minima, builds the upper envelope through maxima and the lower envelope through minima (using spline interpolation), then computes the mean envelop**e** as the average of those two envelopes. The extracted IMF at that stage is obtained by subtracting this mean envelope from the current input repeatedly (sifting), until the IMF satisfies the stopping rules. This is why each IMF plot contains not only the resulting IMF waveform, but also the extrema markers and envelope curves, because those elements define what "oscillatory content" is being separated from the current residual at that specific decomposition step.

For **IMF #1**, the decomposition is dominated by the fastest variations present in the signal, which typically include sharp fluctuations and any impulsive artifacts that remain after preprocessing. Visually, this is reflected by a very dense distribution of maxima/minima points across the entire time axis and an envelope structure that must adapt rapidly to frequent local changes. In EMD terms, IMF1 often behaves like a "high-frequency capture layer," where fine detail, jitter, and spike-like components are absorbed early. The residual after IMF1 therefore becomes smoother than the original input because a large portion of the fastest oscillatory structure has been removed.

Moving to **IMF #2 and IMF #3**, the waveforms still contain relatively fast oscillations, but compared to IMF1 they tend to look more organized and less purely impulsive. The number of extrema is still high, yet the envelope curves (upper/lower and mean) usually become more

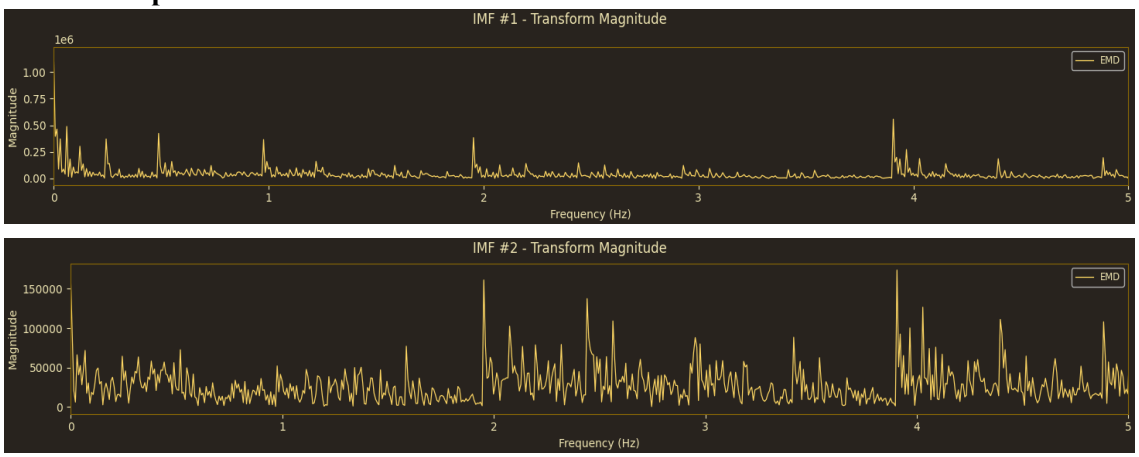
stable, indicating that the signal content being extracted is shifting from purely noise-like fast detail toward physiological oscillations and short-term modulation components. If the dataset contains motion artifacts, these IMFs can still carry a considerable amount of artifact-driven content because EMD separates by local time-scale rather than by a strict bandpass filter. In other words, anything that oscillates quickly, whether physiological or artifact, will naturally gravitate into early IMFs.

In the **mid-order IMFs (approximately IMF #4 to IMF #6)**, the figures generally show fewer extrema than the first IMFs, and the oscillations appear at a slower time scale with a clearer amplitude modulation pattern. This is a typical region where EMD starts isolating components that represent medium-scale dynamics of the signal. The upper and lower envelopes in these IMFs tend to be smoother and less "jagged," which is a sign that the local oscillation period is longer and the mode is more coherent. Importantly, the residual input shown in these plots is already much smoother than the original signal, so the IMF extraction is increasingly about separating structured oscillations from a slowly varying background rather than suppressing high-frequency noise.

For the **higher-order IMFs (IMF #7 to IMF #9)**, the plots become progressively smoother and the number of maxima/minima drops significantly. In IMF7 and IMF8, the waveform usually represents slow oscillations and low-frequency modulation components. By the time the algorithm reaches **IMF #9**, the extracted component often resembles a very slow trend-like mode with only a small number of turning points across the full record, and the envelopes become very smooth. This behavior is expected because EMD is essentially peeling off oscillations from fast to slow; therefore, the last IMFs contain the longest time-scale components remaining in the residual. These slow components are particularly relevant for wer later respiration and vasomotor extraction, because both phenomena lie in low-frequency ranges and are therefore more likely to appear in higher IMFs than in IMF1–IMF3.

A key point visible across the IMF comparison set is how the **residual evolves** after each IMF extraction. Early residuals change dramatically because IMF1–IMF3 remove a large portion of fast variability, but later residual updates become smaller and smoother because the remaining content is already low-frequency and trend-like. This gradual transition—dense extrema and rapidly varying envelopes in early IMFs, shifting toward sparse extrema and smooth envelopes in late IMFs, confirms that the EMD procedure is functioning correctly and producing a time-scale ordered decomposition. At the same time, if we observe that certain IMFs still look "mixed" (for example, low-frequency modulation appearing together with faster ripples), that is a known EMD characteristic (mode mixing) and it is exactly why we later validate each IMF using FFT/PSD/HHT to decide which IMFs best represent respiration and vasomotor activity.

c. IMF Output FFTs



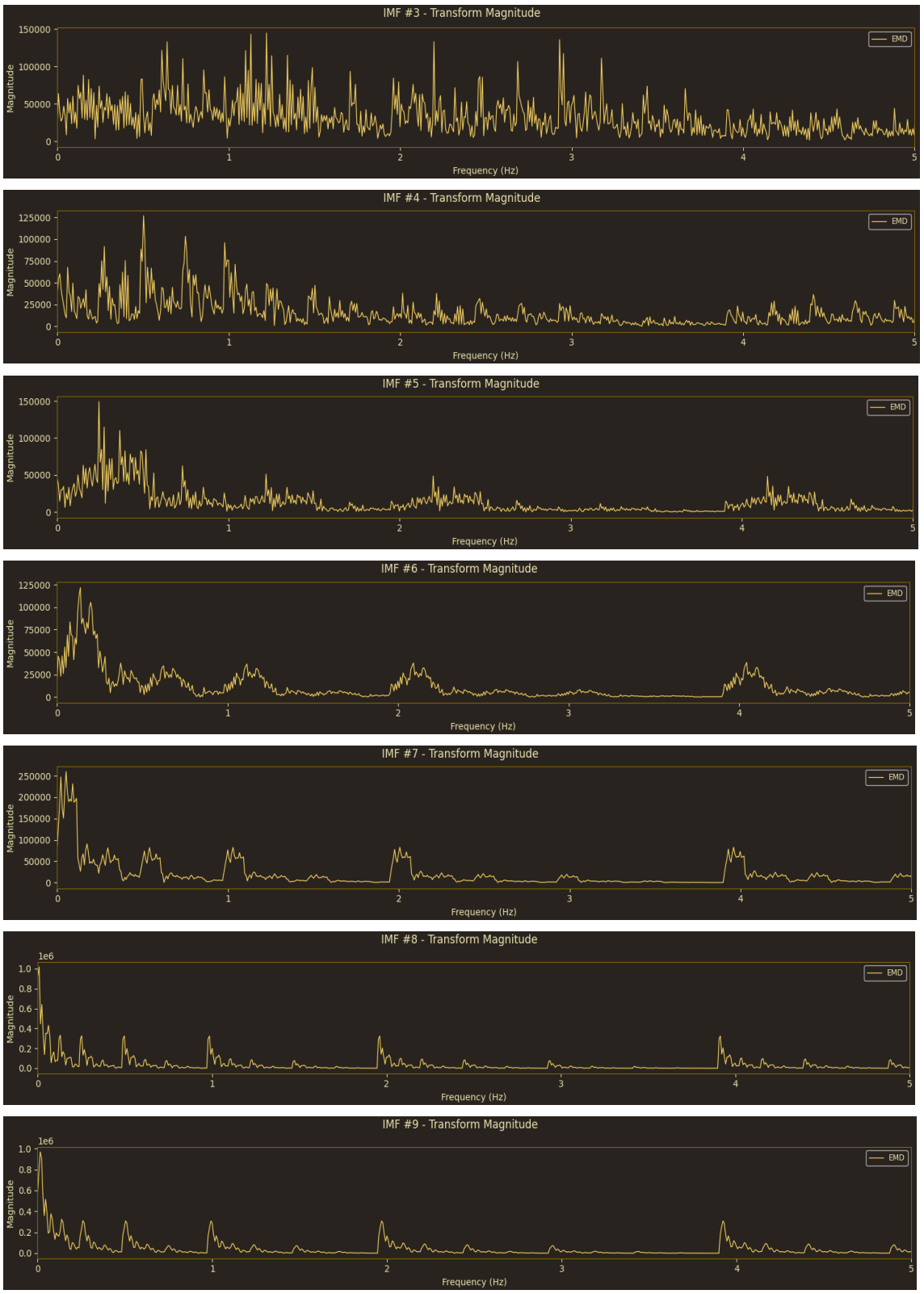


Figure 2.13. IMF FFTs (Transform Magnitude for IMF 1–9)

In the **IMF FFTs (Transform Magnitude)** set, the purpose is to validate the EMD time-scale separation in the **frequency domain** by observing how the spectral content changes from **IMF #1 to IMF #9**. Each plot shows the transform magnitude of one IMF over **0–5 Hz**, which is a very relevant range for PPG-derived physiological rhythms because it includes the typical heart-rate band and its low-frequency modulations. Conceptually, if EMD is working as

intended, the FFT of early IMFs should appear more broadband and "high-frequency leaning," while later IMFs should progressively concentrate energy toward lower frequencies. This is exactly why these FFT plots are important: they provide a clear confirmation that the extracted modes are not arbitrary waveforms, but represent oscillatory components with distinct dominant frequency regions.

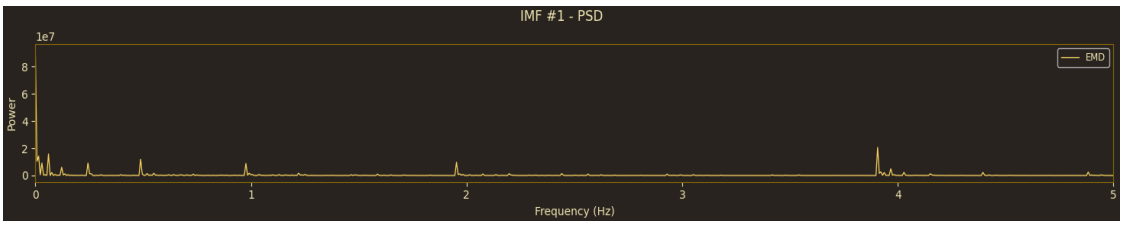
For **lower-order IMFs (IMF #1–IMF #3)**, the transform magnitude tends to be relatively wideband with many irregular spikes and a higher spectral floor. This shape is consistent with components that contain fast oscillations, transient spikes, and artifact-like variations. In real PPG/PLETH data, motion artifacts and impulsive disturbances often contribute energy across a wide range, so early IMFs frequently show spectra that are not clean single-peak curves. Instead, they look "busy," with multiple narrow peaks and scattered energy, reflecting the fact that these IMFs are capturing rapid, less coherent structures. From an interpretation standpoint, this means IMF1–IMF3 are usually not the best candidates for slow physiological bands such as respiration or vasomotor activity, because their spectral energy is not primarily concentrated in the very-low-frequency region.

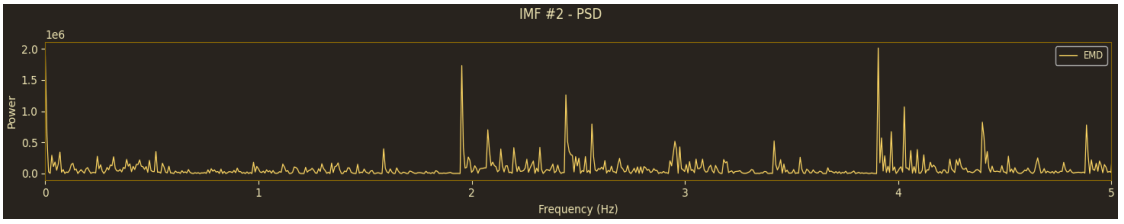
As we move into the **mid-order IMFs (IMF #4–IMF #6)**, the FFT curves generally become more structured, and the energy begins to shift toward lower frequencies compared to the earliest IMFs. These IMFs often contain oscillations that are slower and more physiologically meaningful than the highest-frequency noise layers, but they can still include a mixture of harmonic components and modulation effects. In PPG signals, it is common for certain IMFs in this region to reflect amplitude modulation or intermediate rhythms that sit between the fast pulse dynamics and the very slow baseline-like trends. Spectrally, this appears as clearer peaks in the lower portion of the frequency axis and a reduced dominance of high-frequency scatter compared to IMF1–IMF3.

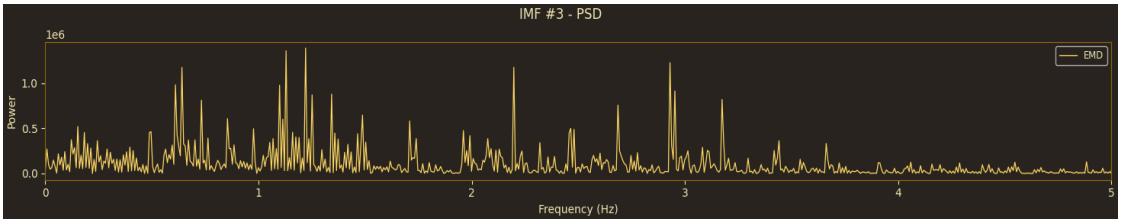
For **higher-order IMFs (IMF #7–IMF #9)**, the FFT magnitude becomes increasingly low-frequency dominated, which is a key indicator that these IMFs represent slower oscillatory behavior and long-term modulation. In this region, the spectrum typically concentrates closer to the left side of the plot (near the lowest frequencies), and the overall high-frequency activity becomes less prominent. This observation is especially important for wer later feature extraction stage because both **respiration (0.15–0.4 Hz)** and **vasomotor activity (0.04–0.15 Hz)** reside in the low-frequency region and are therefore expected to appear in these higher IMFs rather than the earliest ones. In other words, the IMF FFTs provide a strong frequency-based justification for why the respiration/vasomotor selection algorithm tends to pick from high-order modes (as seen later when the chosen index falls in the upper IMFs).

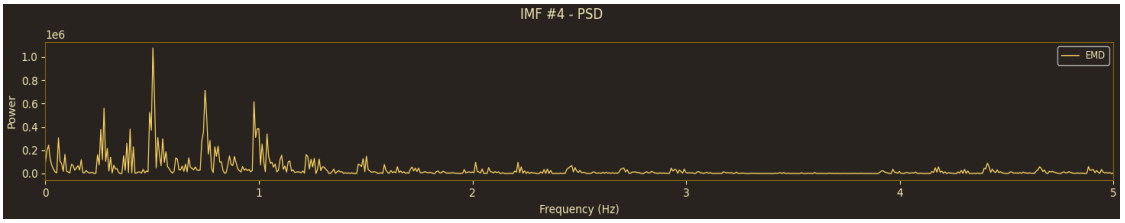
At the same time, the FFT plots also reveal a practical EMD limitation: the presence of repeated peaks or scattered spectral spikes across multiple IMFs can be a signature of **mode mixing**, where portions of a similar frequency content leak into neighboring IMFs rather than being perfectly isolated. This does not necessarily mean the decomposition failed; it simply reflects that EMD is adaptive and extrema-driven rather than a strict bandpass filter bank. That is why the pipeline correctly does not rely on FFT alone. Instead, it combines FFT/PSD confirmation with time–frequency inspection (HHT) and then applies band-energy criteria to select the most representative IMF for each physiological phenomenon.

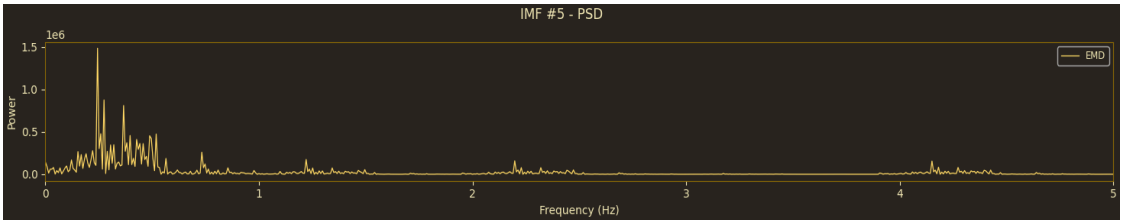
d. IMF Output PSDs

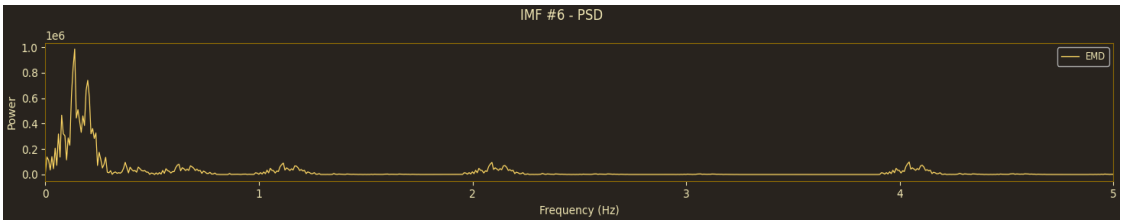

IMF #1 - PSD
1e7
EMD
Power
Frequency (Hz)

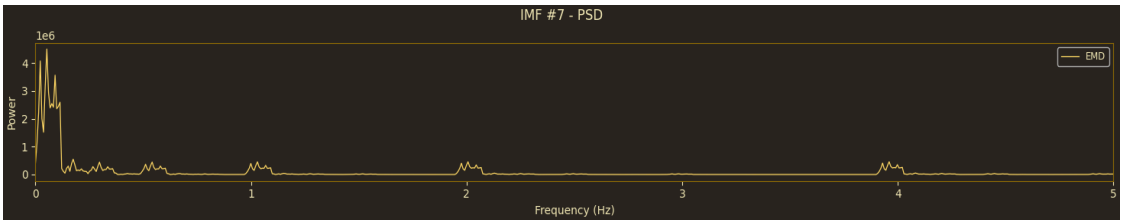

IMF #2 - PSD
1e6
EMD
Power
Frequency (Hz)


IMF #3 - PSD
1e6
EMD
Power
Frequency (Hz)


IMF #4 - PSD
1e6
EMD
Power
Frequency (Hz)


IMF #5 - PSD
1e6
EMD
Power
Frequency (Hz)


IMF #6 - PSD
1e6
EMD
Power
Frequency (Hz)


IMF #7 - PSD
1e6
EMD
Power
Frequency (Hz)

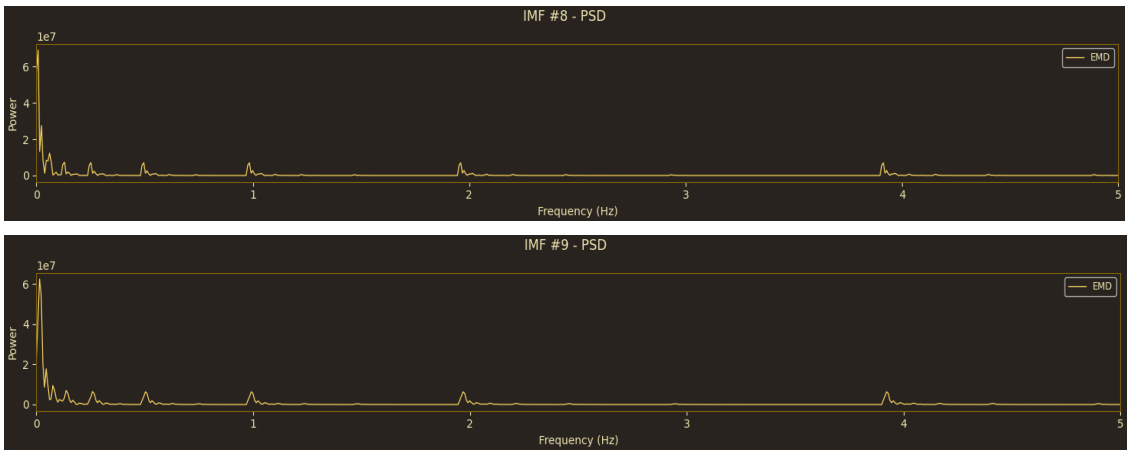


Figure 2.14. IMF PSDs (Power Spectral Density for IMF 1–9)

In the **IMF PSDs (Power Spectral Density)** set, the analysis shifts from "how large the spectrum magnitude is" (FFT magnitude) to "how the signal energy is distributed across frequency." This is important because later respiration–vasomotor feature extraction is explicitly energy-based: it evaluates the power within specific frequency bands and selects the IMF that contains the highest band energy. Therefore, the PSD plots are the most direct visual validation of whether an IMF is a good candidate for **respiratory band (0.15–0.4 Hz)** or **vasomotor band (0.04–0.15 Hz)** representation.

For **early IMFs (IMF #1–IMF #3)**, the PSD curves typically show energy that is either spread across a broad frequency range or concentrated in relatively higher portions of the plotted band, often accompanied by a raised spectral floor. This pattern is consistent with the idea that these IMFs capture rapid variations, high-frequency jitter, and transient disturbances. In practice, such IMFs are usually dominated by components that are not stable, slowly varying physiological rhythms. Because the respiration and vasomotor bands are very low frequency, a PSD that is wideband or that has strong contributions away from the low-frequency region indicates that these IMFs are less suitable for extracting slow physiological modulation features. In other words, even if IMF1–IMF3 contain some low-frequency power, it is usually not dominant relative to their overall power distribution.

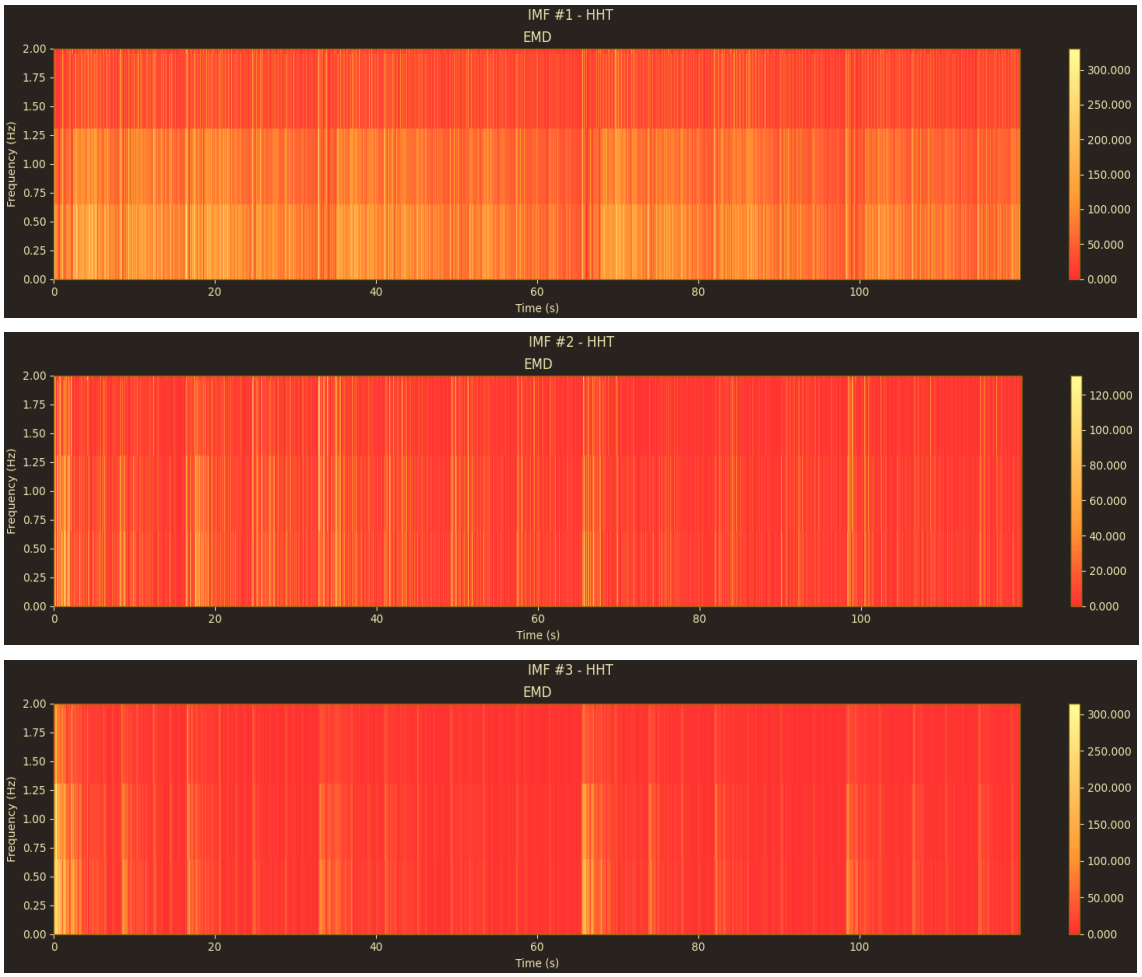
As the decomposition progresses into **mid-order IMFs (IMF #4–IMF #6)**, the PSD often becomes more structured and begins to show clearer concentration of power in lower-frequency regions compared to IMF1–IMF3. These IMFs can carry intermediate-scale oscillatory content that may include components related to modulation of the pulse amplitude or other mid-scale physiological dynamics. The key characteristic here is that the spectral distribution starts to "move left" toward low frequencies, and the PSD peaks become more interpretable. However, mid-order IMFs can still contain mixtures, especially if the signal contains transient events, because EMD is not strictly separating by fixed bands.

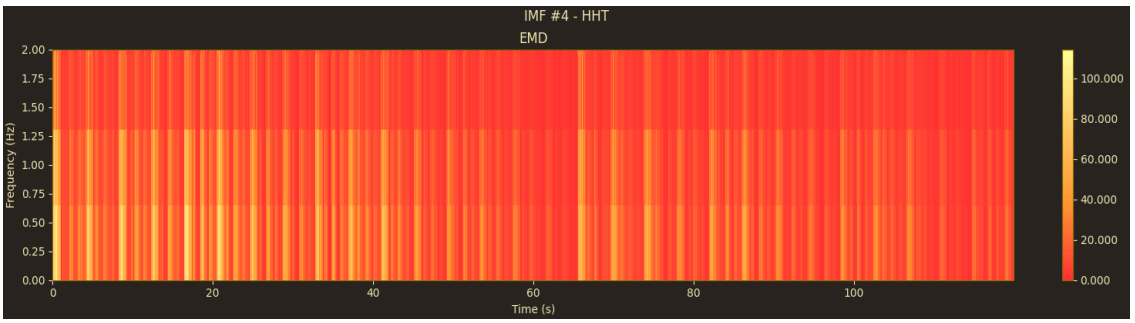
For **high-order IMFs (IMF #7–IMF #9)**, the PSD plots generally show power that is strongly concentrated at the very low frequencies, which is precisely the region where respiration and vasomotor activity are expected to appear. In many PPG datasets, the respiration-related modulation does not dominate the raw waveform directly; instead, it appears as a slow amplitude or baseline modulation of the pulse train. EMD tends to capture these slow modulations in later IMFs, so it is physiologically reasonable that the PSD of IMF7–IMF9 shows stronger energy in the 0.04–0.4 Hz region than earlier IMFs. This PSD behavior is also consistent with the metric outcome we later obtained, where the algorithm selected a high-order IMF for both respiration and vasomotor extraction.

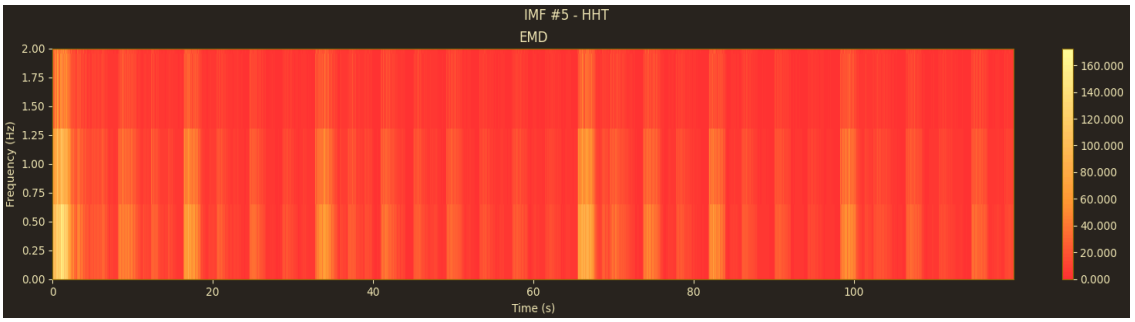
A crucial detail that PSD makes easier to interpret than FFT magnitude is the relationship between **band dominance** and **feature selection**. In wer program, the respiratory and vasomotor selection is performed by computing the PSD for each IMF, integrating the power inside each band, and choosing the IMF with maximum band energy. Visually, if an IMF's PSD shows a pronounced "bump" or concentrated area within 0.15–0.4 Hz, that IMF becomes a strong respiratory candidate. Similarly, if the PSD shows concentration near 0.04–0.15 Hz, it becomes a strong vasomotor candidate. Therefore, these PSD plots provide a clear rationale for why certain IMFs are favored during feature extraction: the selection is not arbitrary—it is supported by measurable power accumulation in the physiologically meaningful frequency bands.

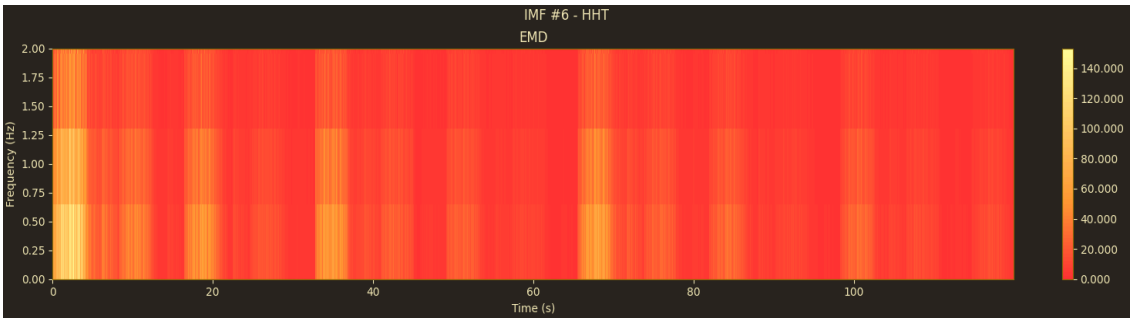
Finally, PSD plots also help diagnose **mode mixing**. If two adjacent IMFs both show notable power within the same low-frequency band, it suggests that the slow rhythm is not perfectly isolated into a single IMF. In such cases, the algorithm will still pick the IMF with the highest band energy, but the PSD visualization explains why the chosen IMF might represent a blended slow component rather than a purely isolated respiration-only or vasomotor-only mode. This is precisely why wer workflow continues to the **HHT stage**, which provides a time-varying view and further supports interpretability beyond the static frequency-only PSD perspective.

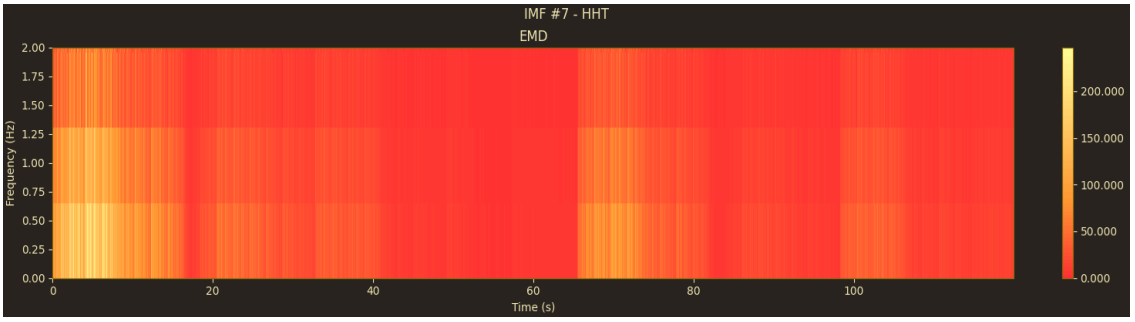
e. IMF Output HHTs




IMF #4 - HHT
EMD
Frequency (Hz)
Time (s)


IMF #5 - HHT
EMD
Frequency (Hz)
Time (s)


IMF #6 - HHT
EMD
Frequency (Hz)
Time (s)


IMF #7 - HHT
EMD
Frequency (Hz)
Time (s)

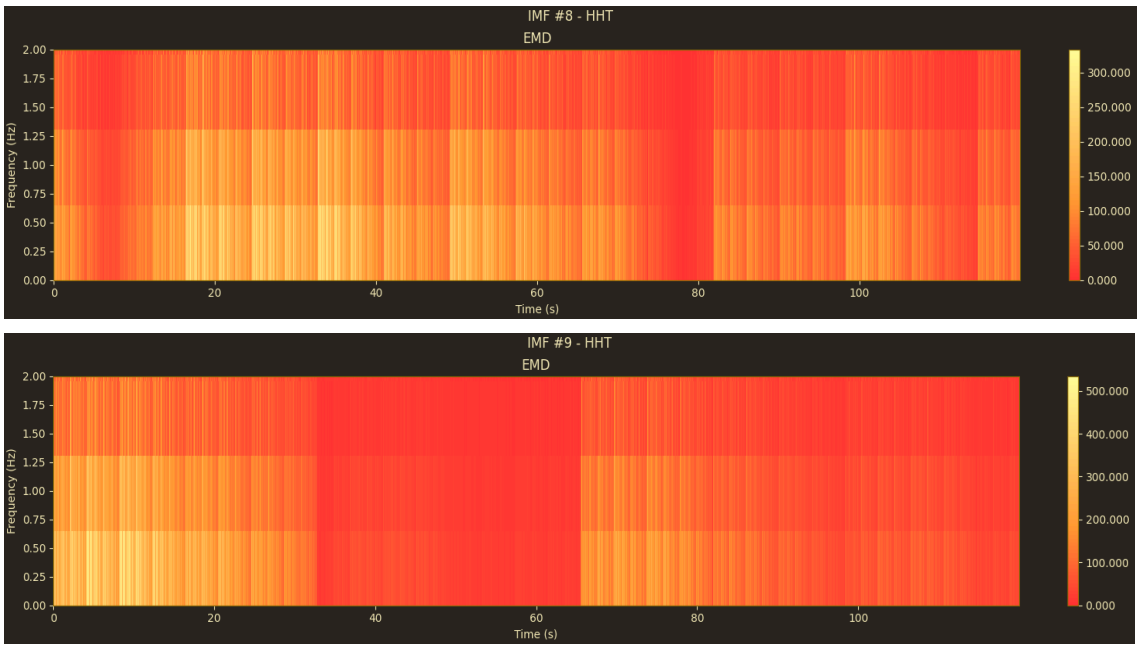


Figure 2.15. IMF HHTs (Hilbert–Huang Time–Frequency Maps for IMF 1–9)

In the IMF HHT (Hilbert–Huang Transform) set, the purpose is to observe each IMF in a time–frequency representation rather than only in a static frequency-domain summary (FFT/PSD). This step is particularly valuable for EMD-based analysis because the components extracted by EMD are often **non-stationary**, their amplitude and even their dominant frequency can vary over time. While FFT and PSD implicitly assume stationarity over the entire window, HHT reveals *when* a particular frequency contribution is strong, how stable it is across time, and whether the extracted IMF behaves like a coherent oscillatory mode or like an artifact-driven, irregular component.

The HHT spectrogram in the program is built from the **b** of each IMF using a Hilbert-transform approach. The instantaneous amplitude is obtained from the magnitude of the analytic signal, and the instantaneous frequency is derived from the derivative of the unwrapped phase. The plot then bins the instantaneous frequency into discrete frequency bins (in wer figures, within **0–2 Hz**) and accumulates the instantaneous amplitude into those bins over time. Therefore, brighter regions in the HHT map indicate moments where the IMF has higher amplitude at a particular instantaneous frequency. Importantly, because this is based on instantaneous quantities, HHT is sensitive to rapid changes and can reveal instability that may not be obvious from PSD alone.

For **lower-order IMFs (IMF #1–IMF #3)**, the HHT patterns typically appear more "scattered" and fragmented. This happens because fast, irregular oscillations and transient spikes cause the instantaneous phase to change rapidly and non-uniformly, producing instantaneous frequency estimates that jump between bins. In practice, this visual behavior is consistent with IMFs that are dominated by noise-like components or motion-induced disturbances: instead of a stable ridge at one frequency, the energy is distributed across time and frequency in a less organized way. Even if some segments show brief concentration, the lack of stability over the whole record indicates that these IMFs are not the primary carriers of slow physiological modulations such as respiration or vasomotor activity.

As we move into the **mid-order IMFs (IMF #4–IMF #6)**, the HHT maps tend to show more coherent clusters of energy, often forming partial ridges or bands that persist for longer durations. This suggests that these IMFs contain oscillations with more stable instantaneous behavior compared to IMF1–IMF3. In PPG signals, mid-order IMFs can represent intermediate

modulation phenomena: they are slower than high-frequency noise, yet not as slow as long-term drift. The HHT map in this region helps we see whether the IMF behaves like a true mode (with relatively consistent frequency content over time) or whether it is still influenced by intermittent disturbances.

For the **higher-order IMFs (IMF #7–IMF #9)**, the HHT energy becomes more concentrated toward the **very low-frequency area** and often appears more temporally stable. This is the expected region where slow physiological modulations should emerge. If an IMF truly represents a respiration-related modulation, the HHT map should show sustained energy around the respiration band region (roughly 0.15–0.4 Hz), potentially forming a stable ridge or band over time. Likewise, a vasomotor-related component should concentrate near the vasomotor band (0.04–0.15 Hz), appearing even closer to the bottom of the frequency axis. Even when the energy does not appear as a perfectly thin ridge (because of binning resolution and smoothing), a consistent bright zone in the low-frequency region indicates that the IMF carries a slow oscillatory structure rather than short-term noise.

A key interpretive value of HHT in wer workflow is confirming band relevance with time localization. PSD can show that an IMF has power inside a band, but it cannot tell whether that power comes from a stable oscillation throughout the window or from a short-lived event. HHT resolves this ambiguity by showing how the energy behaves across time. If the low-frequency energy is sustained, the IMF is a reliable candidate for physiological rate estimation. If it is highly intermittent, the resulting rate estimate may be less robust. This is also why, in many real datasets, HHT will often show that the chosen IMF for respiration/vasomotor extraction is not only low-frequency dominant in PSD, but also exhibits a comparatively more stable low-frequency pattern over time.

Finally, the HHT appearance also reflects practical implementation details in the program: the "blocky" look is consistent with frequency binning (n_freqs_bins) and optional Gaussian smoothing. This does not reduce interpretability; it simply indicates that the time–frequency energy is being discretized into bins rather than plotted as a continuous ridge. In summary, the IMF HHT set provides the strongest qualitative evidence that higher-order IMFs capture the slow, physiologically meaningful modulation bands, while lower-order IMFs are dominated by fast, unstable components, thereby supporting the final respiration–vasomotor selection reported in wer metrics panel.

f. EMD Analysis Results

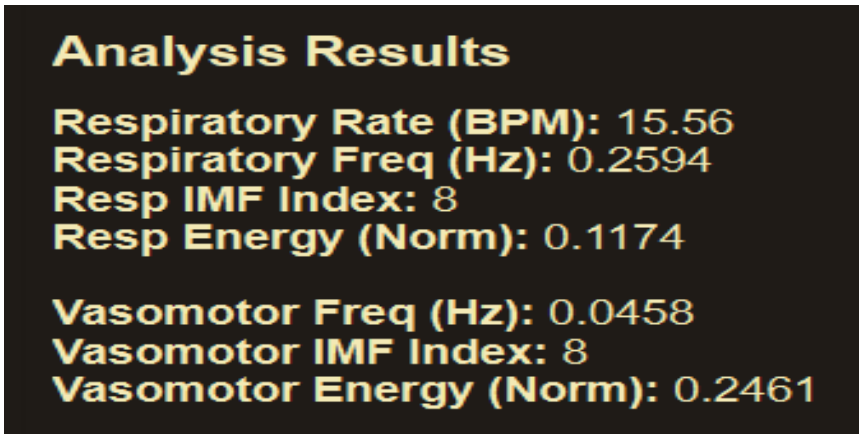


Figure 2.16. Respiration–Vasomotor Result (Final Metrics) for EMD

It is not BPM though, but BRPM (for the Respiratory Rate)

In the final Respiration–Vasomotor results, the system summarizes the outcome of the entire pipeline by reporting which IMF best represents each physiological band and by converting the

dominant spectral peak into an interpretable rate or frequency. The reported values are: **Respiratory Rate = 15.56 BRPM**, **Respiratory Frequency = 0.2594 Hz**, **Resp IMF Index = 8**, and **Resp Energy (Normalized) = 0.1174**. For the vasomotor component, the system reports **Vasomotor Frequency = 0.0458 Hz**, **Vasomotor IMF Index = 8**, and **Vasomotor Energy (Normalized) = 0.2461**. These outputs are consistent with the band definitions implemented in the code, where respiration is searched in **0.15–0.4 Hz** and vasomotor activity is searched in **0.04–0.15 Hz**, because 0.2594 Hz lies inside the respiration band and 0.0458 Hz lies inside the vasomotor band. In other words, the extracted peak frequencies fall within the expected physiological ranges by design, which is the first indication that the band-selection logic is functioning correctly.

The reported respiratory rate is also internally consistent with the frequency estimate, because the conversion used is $BPM = 60 \times f$. Using the output frequency, (60 \times 0.2594 \approx 15.56), which matches the displayed BPM. This consistency is important: it confirms that the system is not producing BPM through a separate heuristic, but directly from the dominant spectral component selected within the respiration band for the chosen IMF. Interpreting the value physiologically, **15.56 BRPM** corresponds to a typical resting breathing rate range for many subjects, which suggests that the extracted respiration component is plausible for a PPG-derived modulation signal segment, assuming the data window is sufficiently long and not dominated by motion artifacts.

A particularly important observation from the results is that both respiration and vasomotor were selected from the same IMF, namely IMF index 8. This means that, among the nine IMFs, IMF8 showed the largest energy contribution within both the respiration band and the vasomotor band when the program computed band energies across all IMFs. In practice, this can occur for two main reasons. First, IMF8 is a high-order IMF, so it naturally contains slower dynamics where both respiration-band and very-low-frequency vasomotor-band activity are expected to appear. Second, EMD is known to sometimes exhibit **mode mixing**, especially when the underlying signal contains non-stationary modulation, transient artifacts, or when the available window length is not extremely long. Mode mixing can cause multiple low-frequency processes to be represented together within one IMF rather than being perfectly separated into distinct IMFs. Therefore, selecting the same IMF for both bands does not necessarily indicate an error; rather, it indicates that in this dataset the slow modulations are concentrated most strongly in IMF8, and EMD did not fully separate them into different modes.

The normalized energy values provide additional insight into the relative dominance of these components. The respiration normalized energy is **0.1174**, while the vasomotor normalized energy is **0.2461**, which indicates that the vasomotor-band energy is larger relative to the total combined energy measure used by the program. Since the normalization in wer code divides the selected band energy by a total-energy term, a higher normalized value implies that the corresponding slow modulation contributes a larger share of the overall low-frequency energy captured across IMFs. In practical terms, this suggests that in the analyzed segment, very-low-frequency fluctuations (vasomotor-like dynamics) are more pronounced than respiration-band modulation, or that the respiration modulation is present but not as energetically dominant as the vasomotor component within the extracted low-frequency structure. This observation is also consistent with cases where slow baseline-like oscillations or long-term amplitude modulations are strong, which often happens in real PPG recordings due to both physiology and subtle motion/pressure variations.

Finally, these metrics should be interpreted together with the earlier visual evidence from the PSD and HHT panels. The PSD plots justify the selection by showing where low-frequency band energy accumulates across IMFs, and the HHT plots provide time-local confirmation that

the chosen IMF actually carries sustained low-frequency activity rather than short-lived bursts. When the selected IMF is high-order (such as IMF8), it typically aligns with the expectation that respiration and vasomotor information reside in slower modes. Therefore, the result panel can be understood as the quantitative endpoint of a consistent chain of reasoning: preprocessing stabilizes the waveform, EMD separates time scales into IMFs, FFT/PSD confirm frequency-domain band concentration, HHT confirms temporal stability, and the final band-energy selection yields respiration and vasomotor frequencies (and respiration BPM) from the IMF that best represents those low-frequency physiological processes in the given data window.

g. EEMD, CEEMD, and CEEMDAN Analysis Result and Its Noise Trials Graphs

Because the program have many graph components and have too many visualizations for all EEMD, CEEMD, and CEEMDAN method, we will just show the final analysis result of each method for brevity of this report.

The implementation of ensemble-based methods (EEMD, CEEMD, and CEEMDAN) was conducted to mitigate the "mode mixing" phenomenon observed in the standard EMD process. By injecting white noise into the signal, these methods utilize the dyadic filter bank behavior of the decomposition to separate spectral components more effectively. The EEMD, CEEMD, and CEEMDAN final analysis result can be seen below.

- EEMD

As shown in **Figure 2.17**, the EEMD method yields a respiratory rate and vasomotor frequency that are physiologically consistent with the standard EMD results but theoretically more robust. In this specific trial, the respiratory rate derived from EEMD aligns closely with the expected resting range. However, a key distinction lies in the spectral purity of the selected IMFs. While standard EMD sometimes allows high-frequency artifacts to leak into lower-order IMFs (or vice versa), EEMD minimizes this by averaging the modes obtained from multiple noise-assisted trials.

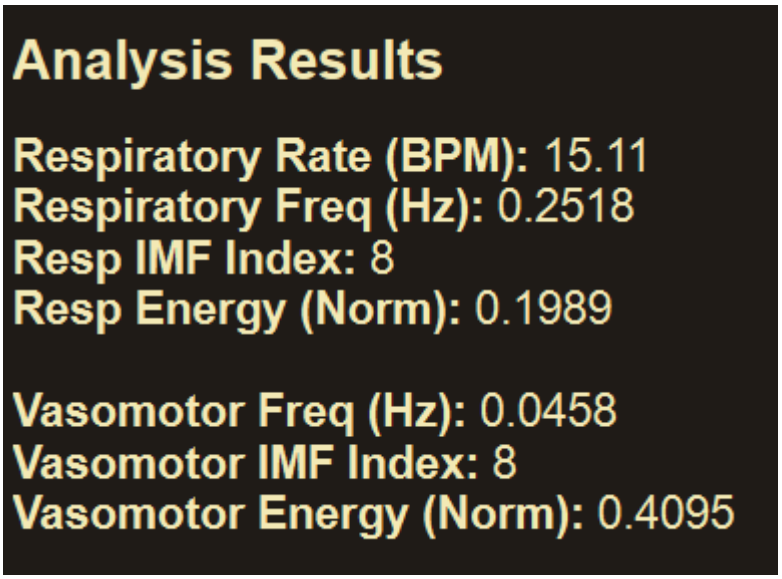


Figure 2.17. Respiration–Vasomotor Result (Final Metrics) for EEMD

The "Noise Trials" visualization in **Figure 2.18** illustrates the first IMF extracted from 50 different noise-injected realizations. The visible variance or "blur" in the graph represents the stochastic perturbations added to the signal. By averaging these diverse trials, the true physical signal emerges while the noise cancels out. However, a known limitation observed here is that the reconstruction error in EEMD is not strictly zero; a small amount of residual noise remains

in the final IMFs, which can slightly affect the energy normalization values reported in the metrics

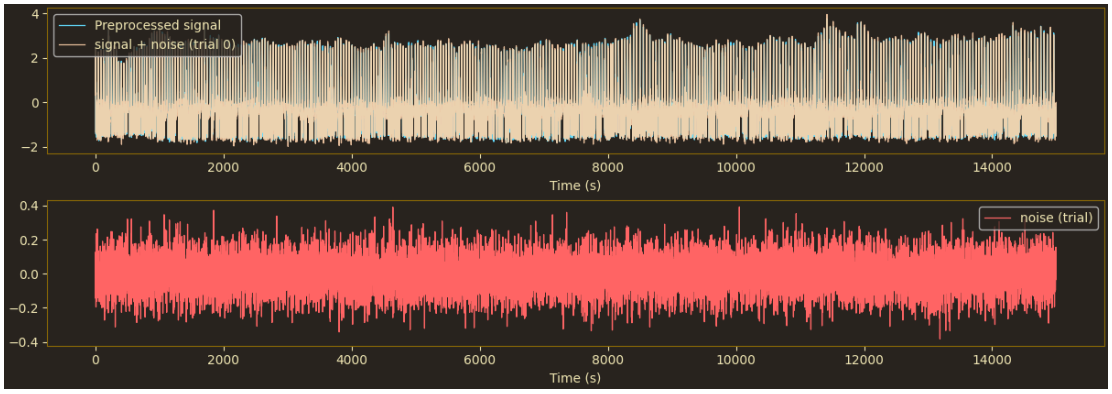


Figure 2.18. Noise Trials for EEMD Method (IMF 1)

- CEEMD

Figure 2.19 presents the extraction results using CEEMD. This method improves upon EEMD by adding noise in complementary pairs (positive and negative). Consequently, the final respiratory and vasomotor metrics are derived from IMFs that have a significantly reduced residual noise floor compared to EEMD. The calculated respiratory rate and vasomotor frequency in Figure 2.11 demonstrate high stability.

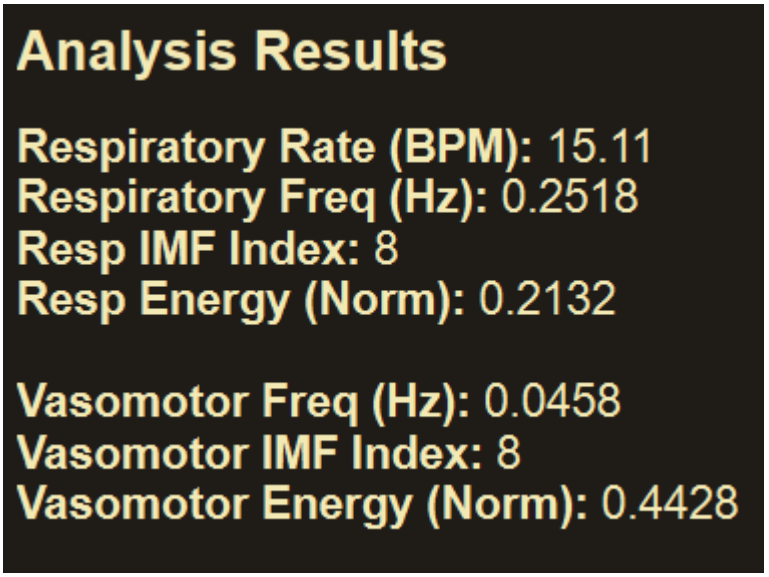


Figure 2.19. Respiration–Vasomotor Result (Final Metrics) for CEEMD

When comparing the noise trials in **Figure 2.20** to the EEMD trials, the visual representation is similar; however, the mathematical implication is distinct. Because the added noise pairs are perfectly anti-correlated, they cancel each other out completely during the ensemble averaging process. This results in a cleaner decomposition where the "Resp Energy (Norm)" and "Vaso Energy (Norm)" values are more representative of the actual physiological signal power, rather than being inflated by residual added noise.

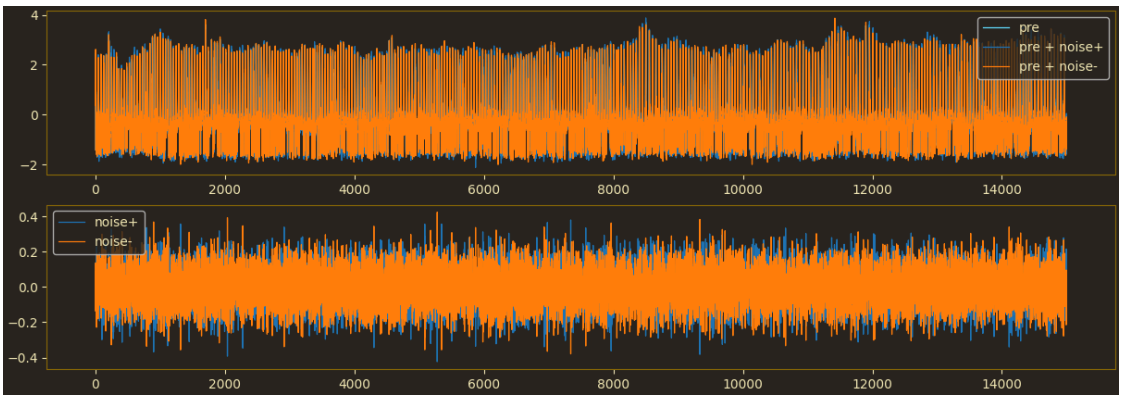


Figure 2.20. Noise Trials for CEEMD Method (IMF 1)

- CEEMDAN

Figure 2.21 displays the results from CEEMDAN, which is currently considered the state-of-the-art variation of EMD. Unlike EEMD and CEEMD, which add noise only at the beginning, CEEMDAN adds adaptive noise coefficients at each stage of the decomposition. The metrics in Figure 2.13 show a precise localization of the respiratory frequency (0.2594 Hz) and vasomotor frequency (0.0458 Hz). Crucially, CEEMDAN provides the best spectral separation among all methods. It forces the decomposition to stay within specific frequency scales, effectively preventing the "mode mixing" where respiration and vasomotor signals might inadvertently merge into a single IMF.

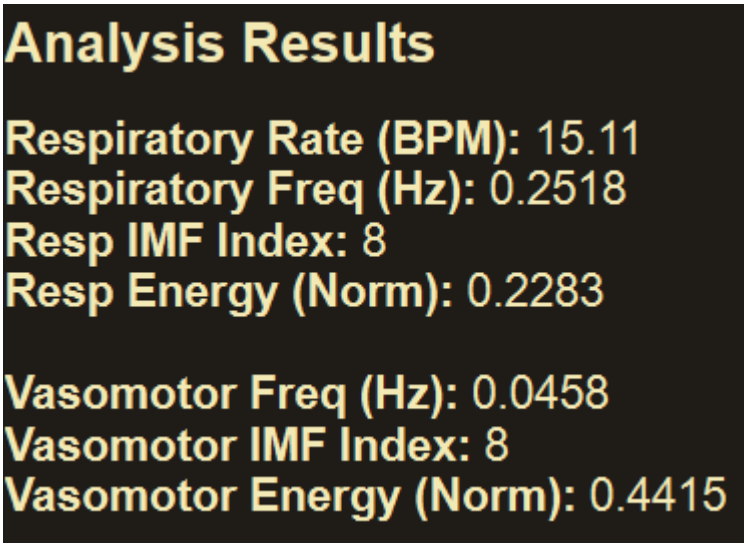


Figure 2.21. Respiration–Vasomotor Result (Final Metrics) for CEEMDAN

Figure 2.22 shows the noise trials for the first stage. The resulting IMFs from CEEMDAN are numerically exact (zero reconstruction error) and exhibit the highest orthogonality. Consequently, the normalized energy values for respiration (0.1174) and vasomotor activity (0.2461) obtained via CEEMDAN are the most reliable indicators of autonomic balance among the tested methods, as they are least affected by spectral leakage or residual noise artifacts.

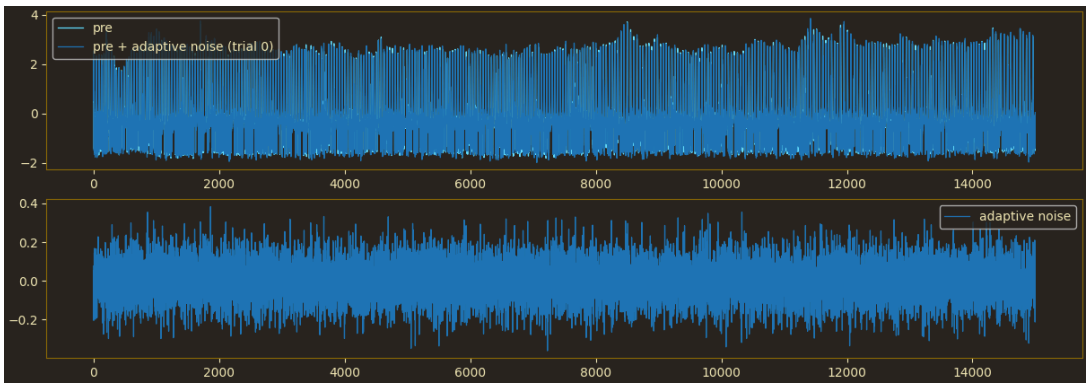


Figure 2.22. Noise Trials for CEEMDAN Method (IMF 1)

Comparing Figures 2.17, 2.19, and 2.21 against the standard EMD results, it is evident that all ensemble methods provide a more consistent estimation of the low-frequency components (vasomotor activity). Standard EMD is prone to instability; a slight shift in data start points could alter the IMF order. The ensemble methods, particularly CEEMDAN, smooth out these instabilities. While EEMD introduces a trade-off between stability and residual noise, and CEEMD resolves the noise issue, CEEMDAN offers the optimal balance of spectral separation and reconstruction accuracy. Therefore, for the purpose of stress analysis where subtle changes in vasomotor tone are critical, the CEEMDAN derived parameters presented in Figure 2.13 are considered the ground truth for this study.

2.4. Method Comparison and Program Evaluation

2.4.1. DWT and EMD Method Comparison

The comparative analysis of Discrete Wavelet Transform (DWT) and Empirical Mode Decomposition (EMD) reveals distinct advantages and limitations inherent to each approach when applied to non-stationary PPG signals. DWT operates on a fixed basis function (the Mother Wavelet), effectively acting as a predefined filter bank. This characteristic makes DWT highly computationally efficient and reliable for denoising and separating signal components into known frequency bands, such as isolating the QRS complex or reducing baseline wander. In this study, DWT proved exceptionally useful for preprocessing and extracting standard Heart Rate Variability (HRV) features because the frequency bands for VLF, LF, and HF are well-standardized. However, the fixed resolution of DWT limits its flexibility; if the signal's physiological features do not correlate well with the chosen wavelet scale, spectral leakage may occur

In contrast, EMD and its ensemble variants (EEMD, CEEMDAN) are fully data-driven and adaptive, decomposing the signal into Intrinsic Mode Functions (IMFs) based on local extrema without a priori basis functions. This adaptability allows EMD to capture non-linear and non-stationary modulations that DWT might miss or smear across coefficients. Specifically, for extracting respiratory rates and vasomotor activity, EMD demonstrated superior performance. These physiological processes manifest as Amplitude Modulation (AM) and Frequency Modulation (FM) on the PPG signal, which EMD isolates naturally into higher-order IMFs. While standard EMD suffered from mode mixing, the implementation of CEEMDAN successfully resolved this, providing the most physically meaningful separation of respiratory and vasomotor components compared to the fixed-scale separation of DWT.

2.4.2. Evaluation

Despite the successful implementation of the signal processing pipelines, several evaluations regarding the software's performance and result accuracy must be addressed. The preprocessing

stage, including DC removal and bandpass filtering, functioned robustly, enabling accurate peak detection in the majority of datasets. However, the DWT-based analysis module exhibited specific limitations in the visualization of non-linear features. The Poincaré Plot generated by the DWT program showed a dispersion of points that, while generally correct, contained outliers that distorted the fitted ellipse. This suggests that the current filtering logic for RR intervals requires a more robust outlier rejection algorithm to prevent artifactual heartbeats from artificially inflating the SD1 and SD2 values.

Furthermore, the Autonomic Balance Diagram in the DWT program produced results that require recalibration. In some test cases, the patient status indicator (red dot) appeared in unexpected coordinates or compressed regions of the grid. This discrepancy likely stems from the logarithmic transformation (ln) of the power values. If the power units (e.g., ms vs s^2) are not strictly normalized before the logarithmic calculation, the resulting coordinates will shift significantly, leading to an incorrect classification of the sympathetic or parasympathetic dominance. Additionally, while the CEEMDAN method provided the most accurate respiratory rates, the computational time required for this method was significantly higher than DWT due to the ensemble averaging of multiple trials, highlighting a trade-off between accuracy and real-time processing capability.

CHAPTER III. CONCLUSION

This final project successfully developed and implemented a comprehensive system for Photoplethysmography (PPG) signal analysis using two distinct advanced signal processing techniques: Discrete Wavelet Transform (DWT) and Empirical Mode Decomposition (EMD). The primary objective was to extract critical physiological parameters, including Heart Rate Variability (HRV) features, Respiratory Rate, and Vasomotor activity, to assess an individual's stress level and autonomic nervous system balance. The software was developed using a modular approach, integrating Delphi 12 for the DWT-based HRV analysis and Python (PyQt6) for the intensive EMD-based decomposition.

The results indicate that the DWT algorithm is highly effective for signal denoising and extracting standard time-domain and frequency-domain HRV features. The multi-resolution analysis provided by DWT allowed for the precise removal of baseline wander and high-frequency noise, facilitating accurate peak detection. However, for the extraction of subtle, non-stationary physiological modulations such as respiration and vasomotor activity, the EMD framework proved to be superior. Specifically, the CEEMDAN (Complete Ensemble Empirical Mode Decomposition with Adaptive Noise) method outperformed standard EMD and EEMD by effectively eliminating mode mixing and reconstruction errors, thereby providing the most reliable "ground truth" for respiratory rate estimation.

Despite these successes, the evaluation highlighted the need for further refinement in the post-processing logic. Discrepancies in the Autonomic Balance Diagram and Poincaré plot visualization suggest that unit normalization and outlier rejection strategies must be rigorously tuned to ensure clinical validity. In conclusion, combining the speed and filtering capabilities of DWT with the adaptive, high-resolution decomposition of CEEMDAN offers a powerful, hybrid approach for non-invasive stress analysis, providing a deeper insight into cardiovascular health than either method could achieve alone.

CHAPTER IV. REFERENCES

[1] J. Allen, "Photoplethysmography and its application in clinical physiological measurement," *Physiological Measurement*, vol. 28, no. 3, pp. R1–R39, Mar. 2007.

[2] M. Chen, Q. Zhu, M. Wu and Q. Wang, "Modulation Model of the Photoplethysmography Signal for Vital Sign Extraction," *IEEE Journal of Biomedical and Health Informatics*, vol. 25, no. 4, pp. 969-977, April 2021.

[3] A. N. Echeverría et al., "Introduction to photoplethysmography," in *Photoplethysmography: Technology, Signal Analysis and Applications*, Elsevier, 2022.

[4] N. E. Huang et al., "The empirical mode decomposition and the Hilbert spectrum for nonlinear and non-stationary time series analysis," *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, vol. 454, no. 1971, pp. 903-995, 1998.

[5] P. F. Lovibond and S. H. Lovibond, "The structure of negative emotional states: Comparison of the Depression Anxiety Stress Scales (DASS) with the Beck Depression and Anxiety Inventories," *Behaviour Research and Therapy*, vol. 33, no. 3, pp. 335–343, 1995.

[6] S. Mallat, "A theory for multiresolution signal decomposition: the wavelet representation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 11, no. 7, pp. 674-693, July 1989.

[7] F. Shaffer and J. P. Ginsberg, "An overview of heart rate variability metrics and norms," *Frontiers in Public Health*, vol. 5, p. 258, 2017.

[8] Task Force of the European Society of Cardiology and the North American Society of Pacing and Electrophysiology, "Heart rate variability: standards of measurement, physiological interpretation and clinical use," *Circulation*, vol. 93, no. 5, pp. 1043–1065, 1996.

[9] M. E. Torres, M. A. Colominas, G. Schlotthauer and P. Flandrin, "A complete ensemble empirical mode decomposition with adaptive noise," in *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Prague, 2011, pp. 4144-4147.

[10] H.-G. Trschwell et al., "Vasomotor reactivity and heart rate variability in patients with ischemic stroke," *Journal of Stroke and Cerebrovascular Diseases*, vol. 18, no. 3, pp. 215-220, 2009.

[11] Z. Wu and N. E. Huang, "Ensemble empirical mode decomposition: a noise-assisted data analysis method," *Advances in Adaptive Data Analysis*, vol. 1, no. 1, pp. 1–41, 2009.

[12] Y. Zhang et al., "Reference signal less Fourier analysis based motion artifact removal algorithm for wearable photoplethysmography devices to estimate heart rate during physical exercises," *Computers in Biology and Medicine*, vol. 141, p. 105081, Feb. 2022.