Program Studi Teknik Informatika
Institut Teknologi Sumatera

Mata Kuliah: **Digital Signal Processing (IF3024)** Tugas: **Final Project**

Daftar Anggota:

1. **Rustian Afencius Marbun (122140155)**
2. **Shintya Ayu Wardani (122140138)**

Real Time Detection of rPPG and Respiratory Signals

1 Pendahuluan

1.1 Latar Belakang

Final project pada mata kuliah Pengolahan Sinyal Digital (IF3024) bertujuan untuk membangun sistem deteksi sinyal rPPG dan sinyal respirasi secara real-time menggunakan webcam. Sistem ini bekerja tanpa kontak fisik dengan memproses video langsung dari kamera untuk menampilkan estimasi detak jantung dan laju napas dalam antarmuka grafis.

Untuk sinyal rPPG, digunakan algoritma Plane Orthogonal-to-Skin (POS) yang menganalisis perubahan warna pada area wajah (dahi) untuk mengekstraksi sinyal denyut nadi. Metode POS dipilih karena terbukti baik pada pencahayaan dan gerakan ringan [1]. Sedangkan untuk sinyal respirasi, sistem mendeteksi gerakan bahu dari landmark pose MediaPipe. Perubahan posisi ini di filter menggunakan band-pass untuk menangkap frekuensi pernapasan (0.1–0.5 Hz)[2].

1.2 Tujuan

Tujuan dibuatnya sistem ini adalah sebagai berikut:

- Mengembangkan sistem pemantauan detak jantung dan laju napas secara real-time tanpa kontak menggunakan webcam.
- Mengimplementasikan algoritma POS untuk ekstraksi sinyal rPPG dan MediaPipe Pose + Butterworth filter untuk sinyal respirasi.
- Menyajikan hasil estimasi dalam bentuk grafik dan teks melalui antarmuka GUI yang interaktif.

1.3 Komponen Pendukung

Dalam penyelesaiian final project ini, di perlukan alat dan bahan yang dapat mendukung dalam pembuatan program deteksi sinyal rPPG dan sinyal respirasi secara real time. Berikut komponen pendukung yang digunakan:

1.3.1 Bahasa Pemrograman

Dalam pengembangan program ini, bahasa pemrograman Python dipilih karena menyediakan library yang banyak digunakan dalam proyek berbasis computer vision.

1.3.2 Library

Berikut beberapa library yang digunakan dalam program ini :

1. **logging**: Mencatat log informasi, peringatan, dan kesalahan selama eksekusi aplikasi. Berguna untuk debugging dan pemantauan alur program.
2. **sys**: Mengakses parameter dan fungsi sistem. Digunakan untuk keluar dari aplikasi saat terjadi kesalahan kritis seperti kegagalan impor modul.
3. **tkinter**: Library GUI standar Python. Digunakan untuk membangun antarmuka pengguna seperti jendela utama, tombol, dan grafik.
4. **cv2 (OpenCV)**: Library visi komputer untuk pengambilan dan pemrosesan frame video dari kamera secara real-time, termasuk anotasi dan konversi warna.
5. **numpy**: Library komputasi numerik. Digunakan untuk manipulasi array, perhitungan statistik, dan pengolahan sinyal.
6. **matplotlib.pyplot**: Modul visualisasi data 2D. Digunakan untuk menampilkan grafik sinyal pernapasan dan detak jantung secara real-time.
7. **threading**: Mengelola eksekusi paralel (multithreading). Digunakan agar proses pengambilan frame tidak mengganggu GUI.
8. **queue**: Antrean thread-safe. Mengalirkan data frame antar-thread secara aman.
9. **time**: Mengatur delay, mengukur durasi, dan mencatat waktu pengambilan data.
10. **PIL (Pillow)**: Mengubah frame dari format OpenCV menjadi format gambar yang bisa ditampilkan di `tkinter`.
11. **collections.deque**: Buffer efisien untuk menyimpan dan membatasi jumlah sampel sinyal secara otomatis.
12. **mediapipe**: Framework real-time dari Google untuk deteksi pose dan wajah. Digunakan untuk mengekstrak landmark tubuh dan wajah.
13. **scipy**: Library ilmiah Python. Digunakan untuk pemrosesan sinyal lanjutan seperti filtering.
14. **pywt (PyWavelets)**: Digunakan untuk denoising sinyal menggunakan transformasi wavelet.
15. **dataclasses**: Modul Python untuk mendeklarasikan class data dengan sintaks yang ringkas dan bersih.
16. **typing**: Menyediakan petunjuk tipe (type hints) untuk meningkatkan keterbacaan dan pemeliharaan kode.
17. **asttokens, executing, stack-data, pure-eval**: Mendukung pelacakan eksekusi dan debugging stack trace.
18. **colorama, prompt-toolkit, pygments, wcwidth**: Untuk pewarnaan teks dan tampilan interaktif di terminal.
19. **comm, ipykernel, ipython, jupyter-client, jupyter-core, nest-asyncio, matplotlib-inline**: Untuk lingkungan Jupyter Notebook.
20. **debugpy**: Debugger Python yang digunakan oleh IDE seperti VSCode.

21. **decorator**: Untuk mempermudah pembuatan dekorator fungsi.
22. **exceptiongroup**: Untuk menangani banyak exception secara bersamaan.
23. **jedi, parso**: Untuk auto-completion dan parsing kode (biasanya digunakan di editor).
24. **packaging, platformdirs**: Untuk mengelola metadata paket dan direktori konfigurasi.
25. **psutil**: Untuk memantau resource sistem seperti CPU dan memori.
26. **pywin32**: Binding ke API Windows.
27. **pyzmq, tornado, traitlets**: Untuk komunikasi jaringan, digunakan dalam backend Jupyter.
28. **six**: Menjaga kompatibilitas lintas versi Python.
29. **typing-extensions**: Mendukung anotasi tipe baru pada Python versi lama.

2 Metode dan Analisis Matematis

2.1 Filter Respiration (Butterworth)

Untuk mengekstraksi sinyal respirasi, sistem ini menggunakan pergerakan vertikal bahu (koordinat Y dari landmark MediaPipe). Nilai ini direkam sebagai sinyal waktu, kemudian difilter menggunakan Butterworth band-pass filter orde 4 untuk mengisolasi frekuensi pernapasan manusia normal, yaitu sekitar 0.1–0.5 Hz (6–30 napas per menit)[2].

Fungsi Transfer

Filter Butterworth dipilih karena respon frekuensinya yang halus dan tanpa ripple pada passband maupun stopband[3]. Fungsi transfer analog dari Butterworth band-pass orde nn secara umum adalah:

$$H(s) = \frac{1}{\sqrt{1 + \left(\frac{s}{\omega_c}\right)^{2n}}}$$

Untuk implementasi digital, digunakan fungsi desain dari scipy.signal.butter() yang mengubah frekuensi cutoff menjadi bentuk digital (diskret) berdasarkan frekuensi sampling f_s, dan diterapkan menggunakan zero-phase filtering dengan scipy.signal.filtfilt() untuk menghindari distorsi fase.

Parameter Filter:

- **Orde**: 4
- **Frek. sampling** f_s: 30 Hz (asumsi webcam 30 FPS)
- **Cutoff**:
  - Low = 0.1 Hz
  - High = 0.5 Hz

2.2 Algoritma rPPG (POS)

Untuk mendeteksi detak jantung, sistem ini menggunakan sinyal warna dari wajah (area dahi). Perubahan warna ini diakibatkan oleh fluktuasi volume darah akibat denyut jantung. Salah satu metode yang digunakan untuk mengekstraksi sinyal tersebut adalah Plane Orthogonal-to-Skin (POS)[1].

Metode POS memproyeksikan sinyal RGB dari video wajah ke arah yang ortogonal terhadap vektor warna kulit, sehingga menghasilkan sinyal dengan rasio sinyal terhadap noise (SNR) yang tinggi.

Langkah-langkah POS:

1. Ekstraksi sinyal RGB dari area wajah selama durasi tertentu.
2. Normalisasi dan pengurangan rata-rata per kanal (detrending warna):

$$\tilde{C}(t) = C(t) - \mu(C(t))$$

3. Proyeksi sinyal warna ke bidang ortogonal:

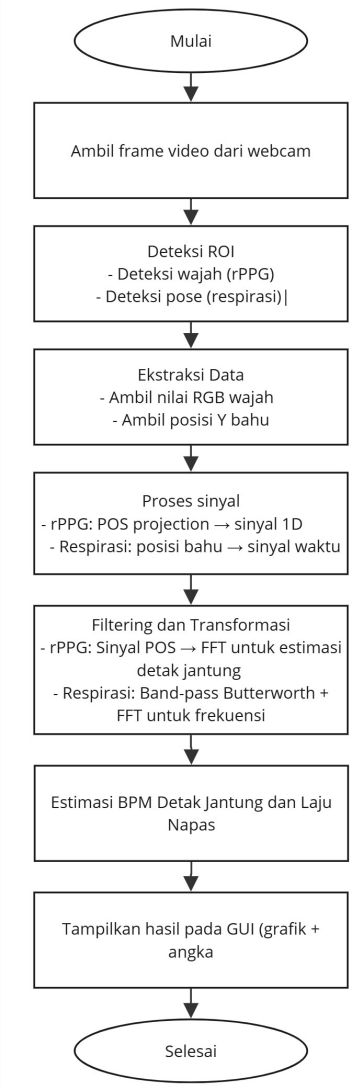
$$S(t) = P \cdot \tilde{C}(t)$$

4. Gabungkan dua sinyal hasil proyeksi dengan estimasi rasio energi untuk menghasilkan satu sinyal 1D rPPG yang dominan.
5. Estimasi detak jantung dari frekuensi dominan sinyal ini menggunakan FFT.

Kelebihan POS:

1. Robust terhadap variasi pencahayaan
2. Efektif dalam kondisi gerakan ringan
3. Lebih baik dibanding metode dasar seperti CHROM atau GREEN

3 Alur Pemrosesan



Gambar 1: Alur Sistem

4 Implementasi

4.1 Instalasi Library

4.1.1 Menggunakan Python venv

File konfigurasi yang ada di dalam direktori virtual environment Python yang dibuat. File ini bukanlah alat untuk menginstal library secara langsung.

```

    python -m venv nama_env
```

Kode 1: Perintah create environment

4.1.2 Menggunakan Requirements.txt

file teks yang berisi daftar semua library Python dan versi spesifiknya yang dibutuhkan oleh suatu proyek.

```
    pip install -r requirements.txt
```

Kode 2: Perintah install requirements.txt

4.2 Folder Gui

Folder gui merupakan paket Python yang berisi komponen-komponen Graphical User Interface (GUI) untuk aplikasi pemantauan.

4.2.1 Modul pycache

- folder yang secara otomatis dibuat oleh Python ketika menjalankan script Python.
- Fungsi utamanya untuk menyimpan bytecode terkompilasi (.pyc file) dari modul-modul Python.
- Tujuannya untuk mempercepat waktu startup pada eksekusi selanjutnya, karena Python tidak perlu mengkompilasi ulang file sumber (.py file) setiap kali.

4.2.2 Modul init.py

Baris *UnifiedVitalSignsApp*, *PlotManager* mendefinisikan apa saja yang akan diekspor (dapat diakses) ketika paket gui diimpor menggunakan from gui import *.

```
from gui.main_app import UnifiedVitalSignsApp
from gui.plot_manager import PlotManager

    __all__ = ['UnifiedVitalSignsApp', 'PlotManager']
```

Kode 3: init.py

4.2.3 Modul main-app.py

Modul ini berisi kelas utama aplikasi GUI, yaitu UnifiedVitalSignsApp.

1. Library

```
    import cv2
    import numpy as np
    import tkinter as tk
    from tkinter import ttk, messagebox
    import threading
    import queue
    import time
    from PIL import Image, ImageTk
    from typing import Optional, Tuple

        from processors import RPPGProcessor, RespirationProcessor
        from gui.plot_manager import PlotManager

```

Kode 4: library main-app.py

2. Kelas UnifiedVitalSignsApp mengoordinasikan GUI, pemrosesan video, dan visualisasi real-time untuk pemantauan detak jantung (menggunakan rPPG) dan laju pernapasan (menggunakan deteksi pose).

```
class UnifiedVitalSignsApp:
    """
    Unified GUI application for real-time vital signs monitoring.

    This application provides a comprehensive interface for monitoring both
    heart rate (using rPPG) and respiration rate (using pose detection)
    from video input with real-time visualization and controls.
    """

    def __init__(self):
        """Initialize the unified vital signs monitoring application."""
        # Setup main window
        self._setup_main_window()

        # Initialize application state
        self._initialize_application_state()

        # Initialize signal processors
        self._initialize_processors()

        # Setup GUI components
        self._setup_gui_components()

        # Initialize threading components
        self._initialize_threading()

        # Bind window close event
        self.root.protocol("WM_DELETE_WINDOW", self.on_closing)

```

Kode 5: Class UnifiedVitalSignsApp

3. Inisialisasi jendela utama aplikasi (setup-main-window).

```
def _setup_main_window(self) -> None:
    """Setup the main application window with proper sizing and title."""
    self.root = tk.Tk()
    self.root.title("Heart Rate & Respiration Realtime Monitoring")
    self.root.geometry("1400x900")
    self.root.minsize(1200, 800)  # Set minimum window size

    # Configure main window style
    style = ttk.Style()
    style.theme_use('clam')  # Use a modern theme

def _initialize_application_state(self) -> None:
        """Initialize application state variables."""
        self.is_running = False
        self.cap: Optional[cv2.VideoCapture] = None
        self.video_label: Optional[tk.Label] = None

```

Kode 6: Inisiasi Utama

4. Inisialisasi processor sinyal untuk rPPG dan pernapasan (initialize processors). Pengaturan komponen GUI seperti panel kontrol, tampilan video, dan area plot (setup–gui-components, setup-control-panel, setup-content-area, dll.).

```
    def _initialize_processors(self) -> None:
        """Initialize signal processing modules."""
        # Initialize rPPG processor for heart rate detection
        self.rppg_processor = RPPGProcessor(fps=30, window_length=1.6)

        # Initialize respiration processor for breathing analysis
        self.resp_processor = RespirationProcessor(fps=30)

    def _setup_gui_components(self) -> None:
            """Setup all GUI components and layout."""
            # Create main container frame
            main_frame = ttk.Frame(self.root)
            main_frame.pack(fill=tk.BOTH, expand=True, padx=10, pady=10)

            # Setup control panel
            self._setup_control_panel(main_frame)

            # Setup content area with video and plots
            self._setup_content_area(main_frame)

    def _setup_control_panel(self, parent: tk.Widget) -> None:
            . . .

    def _setup_content_area(self, parent: tk.Widget) -> None:
            . . .

    def _initialize_threading(self) -> None:
        """Initialize threading components for video capture and processing."""
        self.frame_queue = queue.Queue(maxsize=10)
        self.capture_thread: Optional[threading.Thread] = None

```

Kode 7: Inisialisasi processor

5. Logika untuk memulai (start-monitoring) dan menghentikan (stop-monitoring) proses pemantauan, termasuk inisialisasi kamera dan pengelolaan thread.

```
    def start_monitoring(self) -> None:
        . . .
    def stop_monitoring(self) -> None:
        . . .
        # Clear video display
        cv2.destroyAllWindows()

```

Kode 8: Logika monitoring

6. Pengelolaan queue frame (capture-frames, process-frames) untuk memisahkan pengambilan video dari pemrosesan.

```
    def _capture_frames(self) -> None:
        . . .

    def _process_frames(self) -> None:
        . . .
        # Schedule next frame processing
        if self.is_running:
            self.root.after(33, self._process_frames)  # ~30 FPS
```

```

```

Kode 9: Pengelolaan frame

7. Memperbarui tampilan nilai vital sign (update-vital-sign-display) dan status aplikasi (update-status-display).

```
    def _update_vital_signs_display(self, face_detected: bool, pose_detected: bool,
                                    hr: float, rr: float) -> None:
        . . .
    def _update_status_display(self, face_detected: bool, pose_detected: bool) -> None:
        . . .
            self.plot_manager.update_respiration_data(
                    timestamp, raw_val, filtered_val
                )

```

Kode 10: Tampilan vital sign

8. Mengelola tampilan frame video yang diproses (display-frame).

```
    def _display_frame(self, frame: np.ndarray) -> None:
        . . .

    def _resize_frame_to_fit(self, frame: np.ndarray, width: int, height: int) -> np.ndarray:
        . . .

        return cv2.resize(frame, (new_width, new_height))

```

Kode 11: Tampilan frame video

9. Penanganan penutupan aplikasi (on-closing) untuk memastikan pelepasan sumber daya yang benar.

```
    def on_closing(self) -> None:
        """
        Handle application closing event.

        This method ensures proper cleanup of resources when the
        application window is closed.
        """
        # Stop monitoring if running
        self.stop_monitoring()

        # Close any MediaPipe resources
        if hasattr(self.resp_processor, 'landmarker') and self.resp_processor.landmarker:
            self.resp_processor.landmarker.close()

        # Destroy OpenCV windows
        cv2.destroyAllWindows()

        # Close tkinter application
        self.root.quit()
        self.root.destroy()

    def run(self) -> None:
            """
        Start the unified vital signs monitoring application.
```

```
26
27          This method starts the main GUI event loop and runs the application
28          until the user closes the window.
29          """
30          try:
31              self.root.mainloop()
32              except KeyboardInterrupt:
33                  print("Application interrupted by user")
34                  self.on_closing()
35
36
```

Kode 12: Close aplikasi

4.2.4 Modul plot-manager.py

Modul yang mengelola semua fungsionalitas plotting Matplotlib untuk visualisasi data tanda-tanda vital secara real-time.

1. Library

```
1
2       import numpy as np
3       import matplotlib.pyplot as plt
4       from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
5       from matplotlib.animation import FuncAnimation
6       from collections import deque
7       import tkinter as tk
8       from typing import Tuple, List, Optional
9
```

Kode 13: Library plot-manager.py

2. Kelas PlotManager menangani pembuatan, pembaruan, dan animasi plot untuk rPPG dan sinyal pernapasan

```
1
2       class PlotManager:
3           """
4           Manages real-time plotting for rPPG and respiration signals.
5
6           This class handles the creation, updating, and animation of matplotlib
7           plots for visualizing heart rate and respiration monitoring data.
8           """
9
10          def __init__(self, rppg_parent: tk.Widget, resp_parent: tk.Widget):
11              """
12              Initialize the plot manager with parent widgets for embedding plots.
13
14              Args:
15                  rppg_parent: Parent widget for rPPG plots
16                  resp_parent: Parent widget for respiration plots
17              """
18              self.rppg_parent = rppg_parent
19              self.resp_parent = resp_parent
20
21              # Initialize plot data buffers
22              self._initialize_data_buffers()
23
24              # Setup plots
25              self._setup_rppg_plots()
26              self._setup_respiration_plots()
```

```
27
28          # Start animations
29          self._start_animations()
30
```

Kode 14: Library plot-manager.py

- Inisialisasi buffer data (initialize data buffers) menggunakan deque untuk menyimpan data sinyal terkini.
- Pengaturan plot rPPG (setup rppg plots) dengan tiga subplot (sinyal RGB, sinyal rPPG mentah, dan sinyal rPPG terfilter).
- Pengaturan plot pernapasan (setup respiration plots) dengan dua subplot (sinyal posisi bahu mentah dan sinyal pernapasan terfilter).
- Memulai animasi Matplotlib (start animations) menggunakan FuncAnimation untuk pembaruan real-time.
- Metode untuk memperbarui data plot rPPG (update rppg data) dan pernapasan (update respiration data).
- Fungsi pembaruan internal (update rppg plots, update resp plots) yang dipanggil oleh animasi untuk menggambar ulang grafik.
- Manajemen batas sumbu (update rppg axislimits, update resp axis limits) agar plot selalu menampilkan jendela waktu yang relevan dan penskalaan sumbu Y yang sesuai.
- Fungsionalitas untuk menghapus semua data plot (clear all data).

```
1
2   def _initialize_data_buffers(self) -> None:
3           . . .
4           # Respiration signal buffers
5           self.resp_time_data = deque(maxlen=buffer_size)
6           self.raw_resp_data = deque(maxlen=buffer_size)
7           self.filtered_resp_data = deque(maxlen=buffer_size)
8
9   def _setup_rppg_plots(self) -> None:
10          . . .
11
12  def _setup_filtered_rppg_subplot(self) -> None:
13          . . .
14
15  def _setup_filtered_respiration_subplot(self) -> None:
16          . . .
17          # Configure subplot appearance
18          ax.grid(True, alpha=0.3, linestyle='--')
19          ax.legend(loc='upper right', fontsize=9)
20          ax.set_facecolor('#f8f8f8')
21
22  def _start_animations(self) -> None:
23          . . .
24          # Animation for respiration plots (update every 100ms)
25          self.resp_ani = FuncAnimation(
26              self.resp_fig,
27              self._update_resp_plots,
28              interval=100,
29              blit=False,
30              cache_frame_data=False
31          )
32
33  def update_rppg_data(self, timestamp: float, r_val: float, g_val: float,
```

```
                            b_val: float, rppg_val: float, filtered_val: float) -> None:
        . . .

    def update_respiration_data(self, timestamp: float, raw_val: float,
                                filtered_val: float) -> None:
        . . .
        self.resp_time_data.append(timestamp)
        self.raw_resp_data.append(raw_val)
        self.filtered_resp_data.append(filtered_val)

    def _update_rppg_plots(self, frame) -> List:
        . . .

    def _update_rppg_axis_limits(self, time_array: np.ndarray) -> None:
        . . .
        except (IndexError, ValueError):
                # Fallback to auto-scaling
                ax.relim()
                ax.autoscale_view()

    def _update_resp_plots(self, frame) -> List:
        . . .

    def _update_resp_axis_limits(self, time_array: np.ndarray) -> None:
        . . .
            except (IndexError, ValueError):
                ax.relim()
                ax.autoscale_view()

    def clear_all_data(self) -> None:
        . . .
        # Force plot updates
        self._update_rppg_plots(None)
        self._update_resp_plots(None)
        self.rppg_canvas.draw_idle()
        self.resp_canvas.draw_idle()

    def save_plots(self, filename_prefix: str = "vital_signs") -> None:
        """
        Save current plots to image files.

        Args:
            filename_prefix: Prefix for saved filenames
        """
        try:
            # Save rPPG plots
            self.rppg_fig.savefig(f"{filename_prefix}_heart_rate.png",
                                  dpi=300, bbox_inches='tight')

            # Save respiration plots
            self.resp_fig.savefig(f"{filename_prefix}_respiration.png",
                                  dpi=300, bbox_inches='tight')

            print(f"Plots saved as {filename_prefix}_*.png")

        except Exception as e:
            print(f"Error saving plots: {e}")

```

Kode 15: inisialisasi

4.3 Folder Processors

Folder processors adalah paket Python yang berisi logika utama untuk pemrosesan sinyal vital, dibagi menjadi sub-paket untuk deteksi pernapasan dan deteksi detak jantung (rPPG).

4.3.1 Respiration

Modul ini mengimplementasikan deteksi laju pernapasan dengan menganalisis gerakan vertikal landmark bahu, yang mencerminkan ekspansi dan kontraksi dada selama siklus pernapasan. Ini menggunakan filter bandpass untuk mengisolasi frekuensi pernapasan yang relevan, biasanya dalam rentang 0.1 Hz hingga 0.5 Hz.

1. Modul Respiration-processor-modular.py

   Modul ini merupakan versi yang di-refactor dari processor pernapasan, menggunakan komponen-komponen yang lebih kecil dan terfokus.

```
import numpy as np
import logging
from typing import Optional, Tuple
from utils.pose_detector import PoseDetectionHandler, PoseLandmarks
from utils.signal_buffer import SignalBufferManager, BufferConfig
from processors.respiration.respiratory_analyzer import RespiratorySignalAnalyzer,
FilterConfig, AnalysisResult
from utils.visualization_helper import VisualizationHelper, VisualizationConfig
```

Kode 16: Library Respiration-processor-modular.py

2. Class RespirationProcessorModular

   - PoseDetectionHandler: Untuk mendeteksi landmark pose.
   - SignalBufferManager: Untuk mengelola buffer sinyal.
   - RespiratorySignalAnalyzer: Untuk menganalisis sinyal dan menghitung laju pernapasan.
   - VisualizationHelper: Untuk menggambar visualisasi pada frame.

```
class RespirationProcessorModular:
    """
    Modular respiration processor using coordinated components.

    This refactored version delegates specific responsibilities to focused
    components, making the code more maintainable and testable.
    """

    def __init__(self, fps: int = 30, model_path: Optional[str] = None):
        . . .
        logging.info("Modular respiration processor initialized")

    def _initialize_components(self, model_path: Optional[str]) -> None:
        . . .
        logging.info("All components initialized successfully")

    def process_frame(self, frame: np.ndarray) -> Tuple[np.ndarray, bool, float]:
        . . .

    def _calculate_landmark_quality(self, landmarks: PoseLandmarks) -> float:
        . . .
```

```
            return np.clip(overall_quality, 0.0, 1.0)

        def _analyze_respiratory_signal(self) -> None:
            . . .
                # Store filtered signal for visualization
                if result.filtered_signal:
                    self.signal_buffer.add_filtered_sample(result.filtered_signal[-1])

                logging.info(f"Respiration Rate: {self.current_rr:.1f} BPM (confidence: {
    result.confidence:.3f})")
            else:
                logging.debug("Respiratory analysis failed or low confidence")

        def update_filter_params(self, lowcut: float, highcut: float) -> None:
            """
            Update bandpass filter parameters.

            Args:
                lowcut: Low frequency cutoff in Hz
                highcut: High frequency cutoff in Hz
            """
            self.signal_analyzer.update_filter_config(lowcut, highcut)
            logging.info(f"Filter parameters updated: {lowcut:.2f} - {highcut:.2f} Hz")

        def reset_signals(self) -> None:
            """Reset all signal buffers and analysis state."""
            self.signal_buffer.clear_buffers()
            self.current_rr = 0
            self.frame_idx = 0
            self.last_analysis_result = None
            logging.info("Respiration signals reset")

        def get_signal_quality(self) -> float:
            """
            Get current signal quality metric.

            Returns:
                Signal quality score (0-1, higher is better)
            """
            return self.signal_buffer.get_overall_quality()

        def get_buffer_statistics(self) -> dict:
            . . .
            return stats

        def is_pose_detection_available(self) -> bool:
            """
            Check if pose detection is available.

            Returns:
                True if pose detection is working, False otherwise
            """
            return self.pose_detector.is_available()

        def get_signal_data(self, num_samples: Optional[int] = None) -> dict:
            """
            Get current signal data for external analysis or visualization.

            Args:
                num_samples: Number of recent samples to retrieve

            Returns:
```

```
            Dictionary containing signal arrays
        """
        signal_data = self.signal_buffer.get_signal_data(num_samples)

        return {
            'raw_signal': signal_data.raw_signal,
            'filtered_signal': signal_data.filtered_signal,
            'timestamps': signal_data.timestamps,
            'quality_scores': signal_data.quality_scores
        }

    def configure_visualization(self, **kwargs) -> None:
        """
        Configure visualization parameters.

        Args:
            **kwargs: Visualization configuration parameters
        """
        self.visualizer.update_config(**kwargs)
        logging.info("Visualization configuration updated")

    def validate_current_signal(self) -> Tuple[bool, str]:
        """
        Validate current signal data.

        Returns:
            Tuple of (is_valid, validation_message)
        """
        signal_data = self.signal_buffer.get_raw_signal()
        return self.signal_analyzer.validate_signal(signal_data)

```

Kode 17: Class respiration modular

3. Modul Respiration-processor.py

   Modul ini mengimplementasikan deteksi laju pernapasan dengan menganalisis gerakan vertikal landmark bahu.

```

import numpy as np
import mediapipe as mp
import cv2
import logging
from collections import deque
from typing import Optional, Tuple

from utils.signal_utils import (
    apply_bandpass_filter,
    find_dominant_frequency
)

```

Kode 18: Library Respiration-processor.py

4. Class RespirationProcessor

   - Menginisialisasi sistem deteksi pose MediaPipe (setup pose detection), dengan opsi fallback ke deteksi pose dasar jika landmarker tingkat lanjut tidak tersedia.
   - Mengelola buffer sinyal (initialize signal buffers) untuk menyimpan data pernapasan mentah dan yang sudah terfilter.

- Mengonfigurasi parameter filter bandpass (initialize filter parameters) untuk rentang frekuensi pernapasan (0.1 Hz - 0.5 Hz).
- Memproses setiap frame video untuk mengekstrak sinyal pernapasan, mendeteksi pose, dan menggambar visualisasi (process frame).
- Menghitung laju pernapasan dari gerakan bahu menggunakan analisis frekuensi (calculate respiration rate).
- Menyediakan fungsionalitas untuk memperbarui parameter filter (update filter params) dan mereset sinyal (reset signals).

```

class RespirationProcessor:
    . . .

    def __init__(self, fps: int = 30):
        . . .
        # Initialize respiration rate calculation
        self.current_rr = 0
        self.frame_idx = 0

    def _setup_pose_detection(self) -> None:
        . . .
            self.pose_available = True
            self.landmarker = None

    def _initialize_signal_buffers(self) -> None:
        """Initialize signal storage buffers for respiratory data."""
        # Store up to 1 minute of data for analysis
        self.max_buffer_size = self.fps * 60
        self.raw_y_buffer = deque(maxlen=self.max_buffer_size)
        self.filtered_y_buffer = deque(maxlen=self.max_buffer_size)
        self.time_buffer = deque(maxlen=self.max_buffer_size)

    def _initialize_filter_parameters(self) -> None:
        """Initialize bandpass filter parameters for respiration frequencies."""
        # Normal respiration rate: 12-20 breaths per minute (0.2-0.33 Hz)
        # Allow wider range to capture individual variations
        self.lowcut = 0.1   # Hz (6 breaths per minute)
        self.highcut = 0.5  # Hz (30 breaths per minute)

    def process_frame(self, frame: np.ndarray) -> Tuple[np.ndarray, bool, float]:
        . . .

    def _process_with_landmarker(self, frame: np.ndarray, processed_frame: np.ndarray,
                                 h: int, w: int) -> Tuple[bool, np.ndarray]:
        . . .
        pose_detected = False

        # Convert to RGB and create MediaPipe image
        rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
        mp_image = mp.Image(image_format=mp.ImageFormat.SRGB, data=rgb)

        # Calculate timestamp for video processing
        timestamp_ms = int((self.frame_idx / self.fps) * 1000)
            . . .
        except Exception as e:
            logging.error(f"Pose landmarker processing error: {e}")

        return pose_detected, processed_frame

    def _process_with_basic_pose(self, frame: np.ndarray, processed_frame: np.ndarray,
```

```
                                    h: int, w: int) -> Tuple[bool, np.ndarray]:
        . . .

    def _draw_pose_landmarks(self, frame: np.ndarray, lx: int, ly: int,
                             rx: int, ry: int) -> None:
        . . .

    def _store_respiratory_data(self, avg_y: float) -> None:
        . . .

    def _calculate_respiration_rate(self) -> None:
        . . .
        except Exception as e:
            logging.error(f"Respiration calculation error: {e}")

    def update_filter_params(self, lowcut: float, highcut: float) -> None:
        """
        Update bandpass filter parameters for respiration analysis.

        Args:
            lowcut: Low frequency cutoff in Hz
            highcut: High frequency cutoff in Hz

        Raises:
            ValueError: If parameters are invalid
        """
        if lowcut >= highcut:
            raise ValueError("Low cutoff must be less than high cutoff")

        if lowcut < 0.05 or highcut > 1.0:
            raise ValueError("Cutoff frequencies must be in range [0.05, 1.0] Hz")

        self.lowcut = lowcut
        self.highcut = highcut
        logging.info(f"Updated filter parameters: {lowcut:.2f} - {highcut:.2f} Hz")

    def reset_signals(self) -> None:
        """Reset all signal buffers and respiration rate estimates."""
        self.raw_y_buffer.clear()
        self.filtered_y_buffer.clear()
        self.time_buffer.clear()
        self.current_rr = 0
        self.frame_idx = 0
        logging.info("Respiration signals reset")

    def get_signal_quality(self) -> float:
        """
        Calculate and return current signal quality metric.

        Returns:
            Signal quality score (0-1, higher is better)
        """
        if len(self.raw_y_buffer) < 10:
            return 0.0

        # Calculate signal variability as quality indicator
        recent_data = list(self.raw_y_buffer)[-30:]  # Last 30 samples
        signal_std = np.std(recent_data)
        signal_range = np.max(recent_data) - np.min(recent_data)

        # Normalize quality score (higher movement = better signal)
        quality = min(1.0, signal_range / 50.0)  # Assuming 50 pixels max movement
```

```
114
115             return quality
116
117
```

Kode 19: Class RespirationProcessor

5. Modul Respiration-analyzer.py

   Modul ini fokus pada analisis sinyal pernapasan untuk perhitungan laju pernapasan.

```
1
2   import numpy as np
3   import logging
4   from typing import List, Optional, Tuple
5   from dataclasses import dataclass
6
7   from utils.signal_utils import (
8       apply_bandpass_filter,
9       find_dominant_frequency
10  )
11
```

Kode 20: Library Respiration-analyzer.py

6. Class RespiratorySignalAnalyzer

   - Menerapkan filter bandpass untuk mengisolasi frekuensi pernapasan (filter signal).
   - Menemukan frekuensi dominan dalam rentang pernapasan (find respiratory frequency).
   - Menghitung kualitas sinyal (calculate signal quality) dan kepercayaan diri (calculate confidence) dalam hasil analisis.
   - Menyediakan metode analyze signal yang mengembalikan objek AnalysisResult yang berisi laju pernapasan terhitung, frekuensi dominan, kualitas sinyal, dan kepercayaan diri.

```
1
2   @dataclass
3   class FilterConfig:
4       """
5       Configuration for respiratory signal filtering.
6
7       Attributes:
8           lowcut: Low frequency cutoff in Hz (minimum breathing rate)
9           highcut: High frequency cutoff in Hz (maximum breathing rate)
10          fs: Sampling frequency in Hz
11      """
12      lowcut: float = 0.1   # 6 breaths per minute
13      highcut: float = 0.5  # 30 breaths per minute
14      fs: float = 30.0      # 30 fps
15
16
17  @dataclass
18  class AnalysisResult:
19      """
20      Result of respiratory signal analysis.
21
22      Attributes:
23          respiration_rate: Calculated respiration rate in breaths per minute
24          dominant_frequency: Dominant frequency in Hz
25          signal_quality: Quality score of the analysis (0-1)
26          confidence: Confidence in the result (0-1)
```

```
            filtered_signal: Filtered signal used for analysis
        """
        respiration_rate: float
        dominant_frequency: float
        signal_quality: float
        confidence: float
        filtered_signal: List[float]


    class RespiratorySignalAnalyzer:
        . . .

        def __init__(self, filter_config: Optional[FilterConfig] = None):
            . . .
            logging.info(f"Respiratory analyzer initialized: {self.config.lowcut:.2f}-{self.
    config.highcut:.2f} Hz")

        def _validate_filter_config(self) -> None:
            """Validate filter configuration parameters."""
            if self.config.lowcut >= self.config.highcut:
                raise ValueError("Low cutoff must be less than high cutoff")

            if self.config.lowcut < 0.05 or self.config.highcut > 1.0:
                raise ValueError("Cutoff frequencies must be in range [0.05, 1.0] Hz")

            if self.config.fs <= 0:
                raise ValueError("Sampling frequency must be positive")

        def analyze_signal(self, signal: List[float]) -> Optional[AnalysisResult]:
            . . .
            except Exception as e:
                logging.error(f"Respiratory analysis error: {e}")
                return None

        def _filter_signal(self, signal: List[float]) -> List[float]:
            . . .

        def _find_respiratory_frequency(self, signal: List[float]) -> float:
            . . .
            except Exception as e:
                logging.error(f"Frequency analysis error: {e}")
                return 0.0

        def _calculate_signal_quality(self, signal: List[float]) -> float:
            . . .
            # Calculate SNR
            if noise_estimate > 0:
                snr = signal_power / noise_estimate
                # Normalize SNR to 0-1 range
                quality = min(1.0, snr / 10.0)  # SNR of 10 = quality of 1.0
            else:
                quality = 0.5  # Default if noise can't be estimated

            return quality

        def _calculate_confidence(self, signal: List[float], dominant_freq: float,
                                  signal_quality: float) -> float:
            . . . .
            # Consider signal length (longer signals are more reliable)
            length_factor = min(1.0, len(signal) / 100.0)  # Full confidence at 100+ samples
            confidence *= length_factor

```

```
            # Ensure confidence is in valid range
            return np.clip(confidence, 0.0, 1.0)

        def update_filter_config(self, lowcut: float, highcut: float) -> None:
            . . .

        def get_filter_config(self) -> FilterConfig:
            """
            Get current filter configuration.

            Returns:
                Current FilterConfig object
            """
            return self.config

        def validate_signal(self, signal: List[float]) -> Tuple[bool, str]:
            . . .

        def get_analysis_parameters(self) -> dict:
            """
            Get current analysis parameters.

            Returns:
                Dictionary containing analysis configuration
            """
            return {
                'filter_lowcut': self.config.lowcut,
                'filter_highcut': self.config.highcut,
                'sampling_frequency': self.config.fs,
                'min_analysis_length': self.min_analysis_length,
                'confidence_threshold': self.confidence_threshold
            }

```

Kode 21: Class RespiratorySignalAnalyzer

4.3.2 rPPG

1. Modul rppg-processor.py

   Modul ini mengimplementasikan metode Plane-Orthogonal-to-Skin (POS) untuk mengekstraksi sinyal detak jantung dari video wajah.

```

    import numpy as np
    import mediapipe as mp
    import cv2
    import time
    from collections import deque
    from typing import Optional, Tuple, List

    from utils.signal_utils import (
        apply_bandpass_filter,
        apply_savgol_filter,
        wavelet_denoise,
        normalize_signal,
        detect_peaks_with_validation,
        calculate_signal_quality,
        smooth_signal_exponential
    )
```

```
18
```

Kode 22: Library rppg-processor.py

2. Class RPPGProcessor

- Menginisialisasi deteksi wajah MediaPipe untuk mengidentifikasi ROI (Region of Interest) pada dahi (setup face detection).
- Mengekstrak ROI wajah (extract face roi).
- Mengelola buffer sinyal RGB dan sinyal rPPG yang diekstraksi (initialize signal buffers).
- Menerapkan metode POS (cpu pos) untuk memproyeksikan sinyal RGB ke bidang ortogonal terhadap warna kulit, meminimalkan artefak gerakan.
- Menerapkan berbagai filter dan pemrosesan sinyal (appl filter) seperti bandpass filtering, wavelet denoising, Savitzky-Golay smoothing, dan exponential smoothing.
- Menghitung detak jantung dari sinyal rPPG yang terfilter menggunakan deteksi puncak (calculate heart rate).
- Menyediakan metode untuk mereset sinyal (reset signals).

```
1
2   class RPPGProcessor:
3
4       def __init__(self, fps: int = 30, window_length: float = 1.6):
5           . . .
6           # Configure signal processing parameters
7           self._initialize_processing_parameters()
8
9       def _setup_face_detection(self) -> None:
10          """Initialize MediaPipe face detection with optimized settings."""
11          self.mp_face_detection = mp.solutions.face_detection
12          self.face_detection = self.mp_face_detection.FaceDetection(
13              model_selection=1,  # Full-range model for better accuracy
14              min_detection_confidence=0.5
15          )
16
17      def _initialize_signal_buffers(self) -> None:
18          """Initialize all signal storage buffers."""
19          self.r_signal = deque(maxlen=self.max_buffer_size)
20          self.g_signal = deque(maxlen=self.max_buffer_size)
21          self.b_signal = deque(maxlen=self.max_buffer_size)
22          self.rppg_signal = deque(maxlen=self.max_buffer_size)
23          self.filtered_rppg = deque(maxlen=self.max_buffer_size)
24          self.timestamps = deque(maxlen=self.max_buffer_size)
25
26      def _initialize_hr_parameters(self) -> None:
27          """Initialize heart rate calculation parameters."""
28          self.current_hr = 0
29          self.hr_history = deque(maxlen=15)  # Store recent HR estimates
30          self.hr_timestamps = deque(maxlen=15)
31          self.last_valid_hr_time = 0
32
33      def _initialize_processing_parameters(self) -> None:
34          """Initialize signal processing and filtering parameters."""
35          # Smoothing parameters
36          self.smooth_window_size = 9
37          self.wavelet_name = 'sym4'
38          self.wavelet_level = 3
39
40          # Peak detection parameters
```

```
            self.min_peak_distance = int(self.fps * 0.5)  # Minimum 0.5s between peaks
            self.peak_prominence = 0.3
            self.min_signal_quality = 1.2

            # Timing control
            self.force_recalc_interval = 1.0  # Force recalculation every second
            self.last_calculation_time = 0
            self.prev_filtered_value = 0
            self.last_filter_time = 0

        def cpu_pos(self, signal_array: np.ndarray) -> np.ndarray:
            . . .
                # Add to output with overlap-add
                H[:, m:(n + 1)] = np.add(H[:, m:(n + 1)], Hnm)

            return H[0, :]

        def extract_face_roi(self, frame: np.ndarray) -> Tuple[Optional[np.ndarray], Optional
    [Tuple[int, int, int, int]]]:
            . . .
            # Extract ROI
            roi = frame[forehead_y:forehead_y+forehead_height,
                        forehead_x:forehead_x+forehead_width]
            bbox_coords = (forehead_x, forehead_y, forehead_width, forehead_height)

            return roi, bbox_coords

        def process_frame(self, frame: np.ndarray) -> Tuple[np.ndarray, bool, float]:
            . . .
                # Process signals if sufficient data available
                if len(self.r_signal) >= self.window_size:
                    self._update_rppg_signal()
                    self._calculate_heart_rate()

            return processed_frame, face_detected, self.current_hr

        def _update_rppg_signal(self) -> None:
            """Update rPPG signal using the POS method."""
            if len(self.r_signal) < self.window_size:
                return

            # Prepare RGB array for POS processing
            rgb_array = np.array([
                list(self.r_signal),
                list(self.g_signal),
                list(self.b_signal)
            ])

            # Reshape for POS method: (batch_size=1, channels=3, time_samples)
            rgb_array = rgb_array.reshape(1, 3, -1)

            # Extract rPPG signal using POS method
            rppg = self.cpu_pos(rgb_array)

            # Store the latest rPPG value
            if len(rppg) > 0:
                self.rppg_signal.append(rppg[-1])

                # Apply filtering if sufficient data available
                if len(self.rppg_signal) >= self.window_size:
                    self._apply_filter()

```

```
102         def _apply_filter(self) -> None:
103             . . .
104             except Exception as e:
105                 print(f"Filtering error: {e}")
106                 # Fallback: use raw signal if filtering fails
107                 if len(self.rppg_signal) > 0:
108                     self.filtered_rppg.append(self.rppg_signal[-1])
109 
110         def _calculate_heart_rate(self) -> None:
111             . . .
112             except Exception as e:
113                 print(f"Heart rate calculation error: {str(e)}")
114 
115         def reset_signals(self) -> None:
116             """Reset all signal buffers and heart rate estimates."""
117             self.r_signal.clear()
118             self.g_signal.clear()
119             self.b_signal.clear()
120             self.rppg_signal.clear()
121             self.filtered_rppg.clear()
122             self.timestamps.clear()
123             self.hr_history.clear()
124             self.hr_timestamps.clear()
125             self.current_hr = 0
126             self.last_valid_hr_time = 0
127             self.prev_filtered_value = 0
128 
```

Kode 23: Class RPPGProcessor

4.4 Folder Utils

Folder utils adalah paket Python yang berisi modul-modul utilitas atau pembantu yang menyediakan fungsionalitas umum yang dapat digunakan kembali di seluruh aplikasi. Ini membantu menjaga kode utama tetap bersih dan terfokus pada logika.

4.4.1 Modul pose-detector.py

Modul ini menyediakan handler khusus untuk pengaturan dan pemrosesan deteksi pose.

1. Library

```
1 
2     import cv2
3     import numpy as np
4     import mediapipe as mp
5     import logging
6     from typing import Optional, Tuple, Union
7     from dataclasses import dataclass
8 
```

Kode 24: Library pose-detector.py

2. Class PoseDetectionHandler

   Class PoseDetectionHandler merangkum semua logika deteksi pose, mendukung MediaPipe Tasks yang lebih canggih dan estimasi pose dasar dengan fallback jika diperlukan.

   - Menginisialisasi deteksi pose MediaPipe (setup pose detection), mencoba menggunakan landmarker tingkat lanjut terlebih dahulu dan beralih ke deteksi pose dasar jika terjadi kesalahan.

- Mendeteksi landmark pose dalam frame video (detect pose) dan mengembalikan objek PoseLandmarks yang berisi koordinat bahu dan confidence score.
- Memastikan apakah deteksi pose tersedia dan siap digunakan (is available).
- Objek PoseLandmarks digunakan sebagai dataclass untuk menyimpan data landmark pose yang terstruktur.

```

@dataclass
class PoseLandmarks:
    """
    Container for pose landmark data.

    Attributes:
        left_shoulder_x: X coordinate of left shoulder
        left_shoulder_y: Y coordinate of left shoulder
        right_shoulder_x: X coordinate of right shoulder
        right_shoulder_y: Y coordinate of right shoulder
        average_y: Average Y coordinate of both shoulders
        confidence: Detection confidence score
    """
    left_shoulder_x: float
    left_shoulder_y: float
    right_shoulder_x: float
    right_shoulder_y: float
    average_y: float
    confidence: float = 1.0

class PoseDetectionHandler:

    def __init__(self, model_path: Optional[str] = None):
        . . .

    def _setup_pose_detection(self) -> None:
        """
        Initialize MediaPipe pose detection with fallback options.

        Attempts to use the advanced pose landmarker if available,
        falls back to basic pose estimation for compatibility.
        """
        try:
            # Try to use advanced MediaPipe Tasks pose landmarker
            self._setup_advanced_landmarker()
            logging.info("Using advanced pose landmarker")

        except Exception as e:
            # Fallback to basic pose estimation
            logging.info(f"Advanced pose landmarker not available ({e}), using basic pose
")
            self._setup_basic_pose()

    def _setup_advanced_landmarker(self) -> None:
        """Setup advanced MediaPipe Tasks pose landmarker."""
        from mediapipe.tasks.python import BaseOptions
        from mediapipe.tasks.python.vision import (
            PoseLandmarker, PoseLandmarkerOptions, RunningMode
        )

        options = PoseLandmarkerOptions(
            base_options=BaseOptions(model_asset_path=self.model_path),
            running_mode=RunningMode.VIDEO,
            num_poses=1
```

```
            )
            self.landmarker = PoseLandmarker.create_from_options(options)
            self.pose_available = True

        def _setup_basic_pose(self) -> None:
            """Setup basic MediaPipe pose estimation."""
            self.mp_pose = mp.solutions.pose
            self.pose = self.mp_pose.Pose(
                static_image_mode=False,
                model_complexity=1,
                smooth_landmarks=True,
                min_detection_confidence=0.5,
                min_tracking_confidence=0.5
            )
            self.pose_available = True
            self.landmarker = None

        def detect_pose(self, frame: np.ndarray, frame_idx: int, fps: float) -> Optional[
    PoseLandmarks]:
            . . .
            if self.landmarker:
                return self._detect_with_landmarker(frame, frame_idx, fps, h, w)
            else:
                return self._detect_with_basic_pose(frame, h, w)

        def _detect_with_landmarker(self, frame: np.ndarray, frame_idx: int,
                                    fps: float, h: int, w: int) -> Optional[PoseLandmarks]:
            . . .
            except Exception as e:
                logging.error(f"Pose landmarker processing error: {e}")

            return None

        def _detect_with_basic_pose(self, frame: np.ndarray, h: int, w: int) -> Optional[
    PoseLandmarks]:
            . . .
            except Exception as e:
                logging.error(f"Basic pose processing error: {e}")

            return None

        def is_available(self) -> bool:
            """
            Check if pose detection is available and ready.

            Returns:
                True if pose detection is available, False otherwise
            """
            return self.pose_available

```

Kode 25: Class PoseDetectionHandler

4.4.2 Modul signal-buffer.py

Modul ini menyediakan manajemen buffer sinyal yang efisien untuk pemrosesan sinyal pernapasan.

1. Library

```

    import numpy as np
    import logging
```

```
4     from collections import deque
5     from typing import List, Optional, Tuple
6     from dataclasses import dataclass, field
7
```

Kode 26: Library signal-buffer.py

2. Class SignalBufferManager

- Menginisialisasi buffer sinyal (deque) dengan ukuran maksimum yang ditentukan dalam BufferConfig.
- Menambahkan sampel baru ke buffer sinyal mentah, dengan timestamp dan quality score opsional (add sample).
- Menambahkan sampel sinyal yang sudah terfilter (add filtered sample).
- Mengambil data sinyal mentah, terfilter, timestamp, dan quality score (get raw signal, get filtered signal, get timestamps, get quality scores, get signal data).
- Memeriksa apakah buffer memiliki cukup data untuk analisis (has sufficient data).
- Mengosongkan semua buffer (clear buffers).
- Menghitung kualitas sinyal secara keseluruhan (get overall quality) dan statistik buffer (get buffer statistics).
- Dataclass BufferConfig dan SignalData digunakan untuk mengonfigurasi dan menyimpan data sinyal secara terstruktur.

```
1
2     @dataclass
3     class BufferConfig:
4         . . .
5         max_size: int = 1800  # 1 minute at 30 fps
6         min_analysis_size: int = 30  # Minimum samples for analysis
7         fps: float = 30.0
8
9     @dataclass
10    class SignalData:
11        . . .
12        raw_signal: List[float] = field(default_factory=list)
13        filtered_signal: List[float] = field(default_factory=list)
14        timestamps: List[float] = field(default_factory=list)
15        quality_scores: List[float] = field(default_factory=list)
16
17    class SignalBufferManager:
18            . . .
19            logging.info(f"Signal buffer manager initialized with max size: {self.config.
      max_size}")
20
21        def add_sample(self, value: float, quality_score: Optional[float] = None) -> None:
22            . . .
23            logging.debug(f"Added sample: value={value:.2f}, timestamp={timestamp:.2f}s")
24
25        def add_filtered_sample(self, value: float) -> None:
26            """
27            Add a filtered signal sample.
28
29            Args:
30                value: Filtered signal value to add
31            """
32            self.filtered_buffer.append(value)
33
```

```
    def get_raw_signal(self, num_samples: Optional[int] = None) -> List[float]:
        . . .

    def get_filtered_signal(self, num_samples: Optional[int] = None) -> List[float]:
        . . .

    def get_timestamps(self, num_samples: Optional[int] = None) -> List[float]:
        . . .

    def get_signal_data(self, num_samples: Optional[int] = None) -> SignalData:
        . . .

    def get_quality_scores(self, num_samples: Optional[int] = None) -> List[float]:
        """
        Get signal quality scores.

        Args:
            num_samples: Number of recent scores to retrieve (all if None)

        Returns:
            List of quality scores
        """
        if num_samples is None:
            return list(self.quality_buffer)
        else:
            return list(self.quality_buffer)[-num_samples:] if len(self.quality_buffer)
>= num_samples else list(self.quality_buffer)

    def has_sufficient_data(self) -> bool:
        """
        Check if buffer contains sufficient data for analysis.

        Returns:
            True if sufficient data is available, False otherwise
        """
        return len(self.raw_buffer) >= self.config.min_analysis_size

    def get_buffer_size(self) -> int:
        """
        Get current buffer size.

        Returns:
            Number of samples currently in buffer
        """
        return len(self.raw_buffer)

    def get_buffer_duration(self) -> float:
        """
        Get current buffer duration in seconds.

        Returns:
            Duration of buffered data in seconds
        """
        return len(self.raw_buffer) / self.config.fps

    def clear_buffers(self) -> None:
        """Clear all signal buffers and reset frame count."""
        self.raw_buffer.clear()
        self.filtered_buffer.clear()
        self.time_buffer.clear()
        self.quality_buffer.clear()
        self.frame_count = 0
```

```
95
96          logging.info("Signal buffers cleared")
97
98      def get_overall_quality(self) -> float:
99          """
100         Calculate overall signal quality from recent samples.
101
102         Returns:
103             Overall quality score (0-1, higher is better)
104         """
105         if len(self.quality_buffer) == 0:
106             return 0.0
107
108         # Use recent samples for quality assessment
109         recent_quality = list(self.quality_buffer)[-30:] if len(self.quality_buffer) >=
    30 else list(self.quality_buffer)
110         return np.mean(recent_quality)
111
112     def _calculate_sample_quality(self, value: float) -> float:
113         . . .
114         # Combine quality metrics
115         overall_quality = (range_quality + consistency_quality) / 2
116
117         return np.clip(overall_quality, 0.0, 1.0)
118
119     def get_buffer_statistics(self) -> dict:
120         . . .
121         return {
122             'size': len(raw_data),
123             'duration': self.get_buffer_duration(),
124             'mean': np.mean(raw_data),
125             'std': np.std(raw_data),
126             'range': np.max(raw_data) - np.min(raw_data),
127             'quality': self.get_overall_quality()
128         }
129
```

Kode 27: Class SignalBufferManager

4.4.3 Modul signal-utils.py

Modul ini berisi berbagai fungsi utilitas untuk pemrosesan sinyal.

```
1
2   import numpy as np
3   import scipy.signal as signal
4   import pywt
5   from typing import Tuple, List, Optional
6
```

Kode 28: Library signal-utils

2.
- apply bandpass filter: Menerapkan filter bandpass ke sinyal untuk mengisolasi rentang frekuensi tertentu.
- apply savgol filter: Menerapkan filter Savitzky Golay untuk penghalusan sinyal.
- wavelet denoise: Melakukan denoising (pengurangan noise) menggunakan transformasi wavelet.
- normalize signal: Normalisasi sinyal untuk konsistensi.
- detect peaks with validation: Mendeteksi puncak dalam sinyal dengan validasi.

- calculate signal quality: Menghitung metrik kualitas sinyal.
- smooth signal exponential: Penghalusan sinyal menggunakan exponential smoothing.
- find dominant frequency: Menemukan frekuensi dominan dalam sinyal, kemungkinan menggunakan Fast Fourier Transform (FFT).

```
1
2   def apply_bandpass_filter(data: List[float], lowcut: float, highcut: float,
3                         fs: float, order: int = 4) -> np.ndarray:
4   . . .
5   try:
6       b, a = signal.butter(order, [low, high], btype='band')
7       return signal.filtfilt(b, a, data)
8   except Exception as e:
9       raise ValueError(f"Filter design failed: {e}")
10
11
12  def apply_savgol_filter(data: np.ndarray, window_length: int = 15,
13                          polyorder: int = 2) -> np.ndarray:
14      . . .
15      # Ensure window length is odd and valid
16      if window_length % 2 == 0:
17          window_length -= 1
18      window_length = max(3, min(window_length, len(data) - 2))
19
20      return signal.savgol_filter(data, window_length, polyorder)
21
22
23  def wavelet_denoise(data: np.ndarray, wavelet: str = 'sym4',
24                      levels: int = 3) -> np.ndarray:
25      . . .
26      except Exception as e:
27          print(f"Wavelet denoising failed: {e}")
28          return data
29
30
31  def normalize_signal(signal_data: List[float]) -> np.ndarray:
32      . . .
33      if std < 1e-9:
34          raise ValueError("Signal has zero variance - cannot normalize")
35
36      return (signal_array - mean) / std
37
38
39  def find_dominant_frequency(signal_data: np.ndarray, fs: float,
40                              freq_range: Optional[Tuple[float, float]] = None) -> float:
41      . . .
42      # Find peak frequency
43      if len(fft_vals) == 0 or np.max(fft_vals) == 0:
44          return 0.0
45
46      peak_idx = np.argmax(fft_vals)
47      return freqs[peak_idx]
48
49
50  def detect_peaks_with_validation(signal_data: np.ndarray, fs: float,
51                                   min_distance_sec: float = 0.5,
52                                   prominence: float = 0.2,
53                                   min_width_sec: float = 0.08) -> Tuple[np.ndarray, List[
    float]]:
54      . . .
55      # Validate peak intervals for physiological plausibility
56      validated_rates = []
```

```
        if len(peaks) >= 2:
            for i in range(1, len(peaks)):
                interval_samples = peaks[i] - peaks[i-1]
                if interval_samples > 0:
                    # Convert to rate (beats/breaths per minute)
                    rate = 60.0 * fs / interval_samples
                    # Accept rates in physiological range (40-180 BPM)
                    if 40 <= rate <= 180:
                        validated_rates.append(rate)

        return peaks, validated_rates


    def calculate_signal_quality(signal_data: np.ndarray) -> float:
        . . .
        # Combine amplitude and SNR for quality score
        quality_score = amplitude_range * np.log10(snr + 1)
        return max(0.0, quality_score)


    def smooth_signal_exponential(current_value: float, previous_value: float,
                                  alpha: float = 0.3) -> float:
       . . .
        if not 0 < alpha < 1:
            raise ValueError("Alpha must be between 0 and 1")

        return alpha * current_value + (1 - alpha) * previous_value

```

Kode 29: Pemrosesan sinyal.

4.4.4 Modul visualization-helper.py

Modul ini bertanggung jawab untuk membantu visualisasi pada frame video yang diproses.

1. Library

```

    import cv2
    import numpy as np
    from typing import Tuple, List, Optional
    from dataclasses import dataclass

    from utils.pose_detector import PoseLandmarks

```

Kode 30: Library visualization-helper.py

2. Class VisualizationHelper

   - draw pose landmarks: Menggambar landmark pose pada frame.
   - draw respiration rate: Menampilkan laju pernapasan pada frame.
   - create signal overlay: Membuat overlay sinyal pada frame.
   - Dataclass VisualizationConfig untuk mengonfigurasi parameter visualisasi.

```

    @dataclass
    class VisualizationConfig:
        """
        Configuration for visualization elements.
```

```
6
7           Attributes:
8               landmark_color: Color for pose landmarks (BGR format)
9               line_color: Color for connecting lines (BGR format)
10              text_color: Color for text labels (BGR format)
11              landmark_radius: Radius of landmark circles
12              line_thickness: Thickness of connecting lines
13              text_font: OpenCV font type
14              text_scale: Font scale factor
15              text_thickness: Text line thickness
16          """
17          landmark_color: Tuple[int, int, int] = (255, 0, 0)  # Blue
18          line_color: Tuple[int, int, int] = (0, 255, 255)    # Yellow
19          text_color: Tuple[int, int, int] = (255, 255, 255)  # White
20          landmark_radius: int = 4
21          line_thickness: int = 2
22          text_font: int = cv2.FONT_HERSHEY_SIMPLEX
23          text_scale: float = 0.5
24          text_thickness: int = 1
25
26
27      class VisualizationHelper:
28
29          def __init__(self, config: Optional[VisualizationConfig] = None):
30              . . .
31              # Draw confidence indicator if available
32              if landmarks.confidence < 1.0:
33                  self._draw_confidence_indicator(annotated_frame, landmarks.confidence)
34
35              return annotated_frame
36
37          def _draw_landmark(self, frame: np.ndarray, x: int, y: int, label: str) -> None:
38              """
39              Draw a single landmark point with label.
40
41              Args:
42                  frame: Frame to draw on
43                  x: X coordinate
44                  y: Y coordinate
45                  label: Text label for the landmark
46              """
47              # Draw landmark circle
48              cv2.circle(frame, (x, y), self.config.landmark_radius,
49                         self.config.landmark_color, -1)
50
51              # Draw label above the landmark
52              label_y = max(y - 10, 20)  # Ensure label is visible
53              cv2.putText(frame, label, (x - 20, label_y),
54                          self.config.text_font, self.config.text_scale,
55                          self.config.text_color, self.config.text_thickness)
56
57          def _draw_roi_indicator(self, frame: np.ndarray, landmarks: PoseLandmarks) -> None:
58              """
59              Draw region of interest indicator.
60
61              Args:
62                  frame: Frame to draw on
63                  landmarks: Pose landmarks for ROI calculation
64              """
65              # Calculate ROI center
66              center_x = int((landmarks.left_shoulder_x + landmarks.right_shoulder_x) / 2)
67              center_y = int(landmarks.average_y)
```

```
68
69          # Draw ROI label
70          cv2.putText(frame, "Respiration ROI", (center_x - 50, center_y - 30),
71                      self.config.text_font, self.config.text_scale,
72                      self.config.text_color, self.config.text_thickness)
73
74      def _draw_confidence_indicator(self, frame: np.ndarray, confidence: float) -> None:
75          . . .
76          # Confidence text
77          cv2.putText(frame, f"Conf: {confidence:.2f}", (bar_x, bar_y - 5),
78                      self.config.text_font, self.config.text_scale,
79                      self.config.text_color, self.config.text_thickness)
80
81      def _get_confidence_color(self, confidence: float) -> Tuple[int, int, int]:
82          """
83          Get color based on confidence level.
84
85          Args:
86              confidence: Confidence score (0-1)
87
88          Returns:
89              BGR color tuple
90          """
91          if confidence > 0.8:
92              return (0, 255, 0)    # Green
93          elif confidence > 0.5:
94              return (0, 255, 255)  # Yellow
95          else:
96              return (0, 0, 255)    # Red
97
98      def draw_respiration_rate(self, frame: np.ndarray, respiration_rate: float,
99                                signal_quality: float = 1.0) -> np.ndarray:
100         . . .
101         # Draw quality indicator
102         self._draw_quality_indicator(annotated_frame, signal_quality, text_x, text_y -
    25)
103
104         return annotated_frame
105
106     def _get_quality_color(self, quality: float) -> Tuple[int, int, int]:
107         """
108         Get color based on signal quality.
109
110         Args:
111             quality: Quality score (0-1)
112
113         Returns:
114             BGR color tuple
115         """
116         if quality > 0.7:
117             return (0, 255, 0)    # Green
118         elif quality > 0.4:
119             return (0, 255, 255)  # Yellow
120         else:
121             return (0, 128, 255)  # Orange
122
123     def _draw_quality_indicator(self, frame: np.ndarray, quality: float,
124                                 x: int, y: int) -> None:
125         """
126         Draw signal quality indicator.
127
128         Args:
```

```
            frame: Frame to draw on
            quality: Quality score (0-1)
            x: X position for indicator
            y: Y position for indicator
        """
        quality_text = f"Quality: {quality:.2f}"
        color = self._get_quality_color(quality)

        cv2.putText(frame, quality_text, (x, y),
                    self.config.text_font, self.config.text_scale,
                    color, self.config.text_thickness)

    def draw_breathing_indicator(self, frame: np.ndarray, breathing_phase: str,
                                 intensity: float = 0.5) -> np.ndarray:
        . . .
        # Draw text
        cv2.putText(annotated_frame, text, (indicator_x, indicator_y),
                    self.config.text_font, self.config.text_scale * 1.2,
                    color, self.config.text_thickness + 1)

        return annotated_frame

    def create_signal_overlay(self, frame: np.ndarray, signal_data: List[float],
                              max_points: int = 100) -> np.ndarray:
        . . .
        # Add overlay label
        cv2.putText(annotated_frame, "Signal",
                    (w - max_points * 2 - 15, overlay_y_start - 15),
                    self.config.text_font, self.config.text_scale,
                    self.config.text_color, self.config.text_thickness)

        return annotated_frame

    def update_config(self, **kwargs) -> None:
        """
        Update visualization configuration.

        Args:
            **kwargs: Configuration parameters to update
        """
        for key, value in kwargs.items():
            if hasattr(self.config, key):
                setattr(self.config, key, value)
            else:
                raise ValueError(f"Invalid configuration parameter: {key}")

    def get_config(self) -> VisualizationConfig:
        """
        Get current visualization configuration.

        Returns:
            Current VisualizationConfig object
        """
        return self.config


```

Kode 31: Class

4.4.5 Modul Main.py

File main.py adalah entry point utama untuk aplikasi pemantauan tanda-tanda vital. Ini berfungsi sebagai orkestrator yang menginisialisasi sistem dan menjalankan aplikasi GUI.

1. Library

```
1
2   import logging
3   import sys
4   from tkinter import messagebox
5
6   # Import modular components
7   try:
8       from gui.main_app import UnifiedVitalSignsApp
9   except ImportError as e:
10      print(f"Error importing GUI components: {e}")
11      print("Please ensure all required packages are installed and modules are in the
    correct location.")
12      sys.exit(1)
13
```

Kode 32: Library Main.py

2. Inisialisasi sistem

- Impor Modul: Mengimpor modul penting seperti logging, sys, tkinter.messagebox, dan UnifiedVitalSignsApp dari gui main app, serta menangani kegagalan impor.
- Fungsi setup logging(): Mengonfigurasi pencatatan log ke konsol dan file, dengan level INFO dan format lengkap, serta mencatat pesan saat aplikasi dimulai.
- Fungsi check dependencies(): Memverifikasi ketersediaan modul penting (cv2, numpy, scipy, dll). Jika ada yang hilang, tampilkan pesan kesalahan GUI dan log error.
- Fungsi main(): Menjalankan urutan inisialisasi aplikasi: logging, pengecekan dependency, peluncuran GUI (app.run()), dan penanganan kesalahan fatal secara terpusat.
- Blok ": Memastikan main() hanya dijalankan jika main.py dieksekusi langsung, bukan saat diimpor.

```
1
2   # Import modular components
3   try:
4       from gui.main_app import UnifiedVitalSignsApp
5   except ImportError as e:
6       print(f"Error importing GUI components: {e}")
7       print("Please ensure all required packages are installed and modules are in the
    correct location.")
8       sys.exit(1)
9
10
11  def setup_logging():
12      """
13      Configure logging for application monitoring.
14
15      Sets up both console and file logging with appropriate formatting
16      to track application events and potential issues.
17      """
18      logging.basicConfig(
19          level=logging.INFO,
20          format="[%(asctime)s] %(levelname)s - %(message)s",
21          handlers=[
```

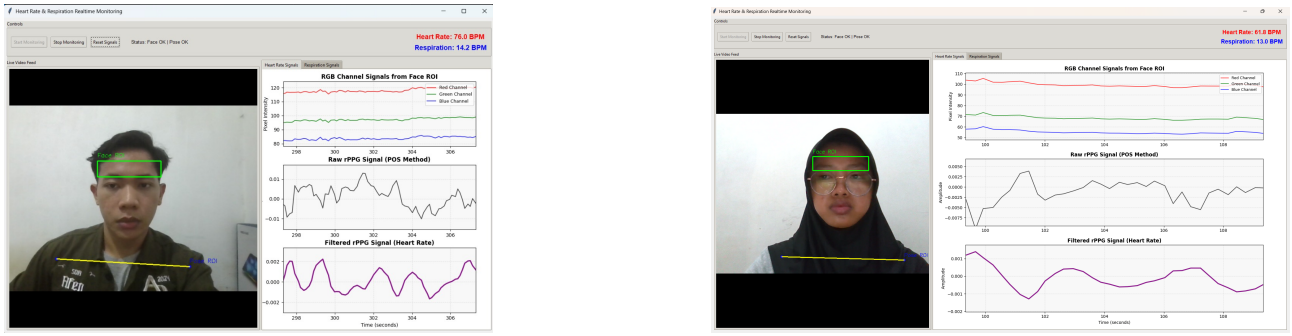
```
                logging.StreamHandler(),
                logging.FileHandler('vital_signs_monitor.log')
            ]
        )

        # Log application startup
        logging.info("Vital Signs Monitor application starting...")


    def check_dependencies():
        """
        Check if all required dependencies are available.

        Returns:
            bool: True if all dependencies are available, False otherwise
        """
        required_modules = [
            'cv2', 'numpy', 'scipy', 'matplotlib', 'tkinter',
            'mediapipe', 'PIL', 'pywt'
        ]

        missing_modules = []

        for module in required_modules:
            try:
                __import__(module)
            except ImportError:
                missing_modules.append(module)

        if missing_modules:
            error_msg = f"Missing required modules: {', '.join(missing_modules)}"
            logging.error(error_msg)
            messagebox.showerror(
                "Missing Dependencies",
                f"{error_msg}\n\nPlease install the required packages using:\npip install -r
    requirements.txt"
            )
            return False

        return True


    def main():
        """
        Main entry point for the Vital Signs Monitor application.

        This function:
        1. Sets up logging
        2. Checks for required dependencies
        3. Creates and runs the main application
        4. Handles any startup errors gracefully
        """
        try:
            # Setup application logging
            setup_logging()

            # Check dependencies
            if not check_dependencies():
                logging.error("Dependency check failed. Exiting application.")
                return

            logging.info("All dependencies verified successfully")
```

```
83
84          # Create and run the main application
85          logging.info("Initializing main application...")
86          app = UnifiedVitalSignsApp()
87
88          logging.info("Starting application main loop...")
89          app.run()
90
91      except Exception as e:
92          error_msg = f"Fatal error during application startup: {e}"
93          logging.error(error_msg, exc_info=True)
94
95          # Show user-friendly error message
96          try:
97              messagebox.showerror(
98                  "Application Error",
99                  f"An error occurred while starting the application:\n\n{e}\n\nCheck the
    log file for more details."
100             )
101         except:
102             # If even tkinter fails, print to console
103             print(f"FATAL ERROR: {error_msg}")
104
105         sys.exit(1)
106
107     finally:
108         logging.info("Vital Signs Monitor application shutdown complete")
109
110
111 if __name__ == "__main__":
112     """
113     Application entry point.
114
115     This ensures the application only runs when this file is executed directly,
116     not when imported as a module.
117     """
118     main()
119
```

Kode 33: Menjalankan GUI

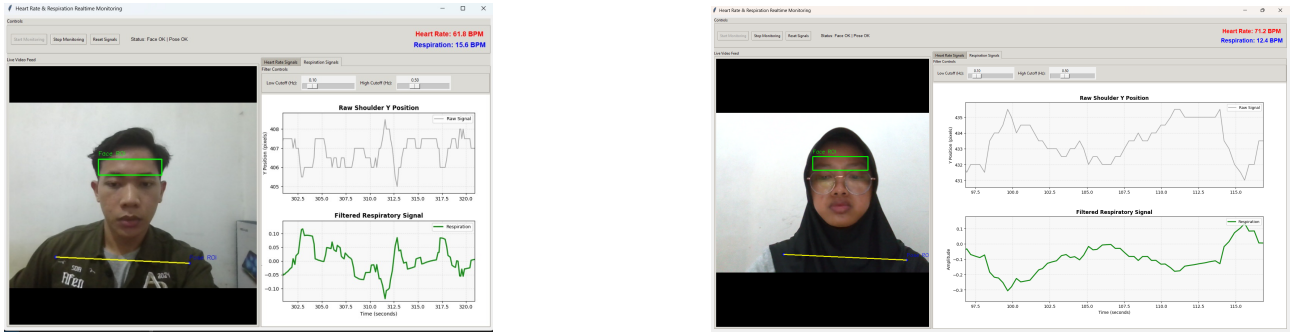
5 Hasil

Sistem deteksi detak jantung dan pernapasan real-time telah berhasil diimplementasikan dengan antarmuka GUI yang menampilkan video feed langsung, kontrol monitoring, dan visualisasi sinyal secara simultan. Sistem dapat mendeteksi ROI pada area wajah untuk ekstraksi sinyal rPPG dan area bahu untuk deteksi pergerakan respirasi menggunakan MediaPipe. Dari hasil pengujian, sistem berhasil mengekstrak sinyal RGB dari ketiga channel warna dengan intensitas yang stabil dan konsisten. Setelah pemrosesan menggunakan metode POS dan filtering, sinyal rPPG yang dihasilkan menampilkan pola periodik yang jelas, menunjukkan bahwa sistem mampu mendeteksi variasi detak jantung dengan akurasi yang baik dalam rentang normal fisiologis.



(a) Afen-Hasil Sinyal rPPG (b) Shintya-Hasil Sinyal rPPG

Gambar 2: Hasil Sinyal rPPG



(a) Afen-Hasil Sinyal Respirasi (b) Shintya-Hasil Sinyal Respirasi

Gambar 3: Hasil Sinyal Respirasi

Untuk sinyal respirasi, sistem berhasil mendeteksi pergerakan vertikal bahu yang kemudian difilter menggunakan Butterworth band-pass filter untuk menghasilkan sinyal respirasi yang bersih dan stabil. Hasil pengukuran respiration rate menunjukkan konsistensi dalam rentang normal pernapasan manusia. Sistem dilengkapi dengan kontrol filter yang dapat disesuaikan dan mampu melakukan monitoring kontinyu dengan stabilitas sinyal yang baik. Meskipun sistem menunjukkan kinerja yang memuaskan untuk monitoring non-invasive menggunakan webcam standar, terdapat keterbatasan dalam hal sensitivitas terhadap kondisi pencahayaan dan memerlukan posisi subjek yang relatif stabil untuk hasil optimal, namun secara keseluruhan sistem dapat berfungsi sebagai alat monitoring vital sign yang efektif dan mudah digunakan.

6 Kesimpulan

Sistem deteksi detak jantung dan pernapasan real-time menggunakan webcam telah berhasil diimplementasikan dengan menggabungkan metode Plane Orthogonal-to-Skin (POS) untuk ekstraksi sinyal rPPG dan MediaPipe Pose untuk deteksi pergerakan respirasi. Sistem mampu melakukan monitoring simultan terhadap kedua parameter vital dengan antarmuka yang user-friendly, menampilkan visualisasi sinyal real-time, dan menghasilkan pengukuran yang berada dalam rentang normal fisiologis. Penerapan filtering Butterworth band-pass dan teknik denoising wavelet berhasil menghasilkan sinyal yang bersih dan stabil untuk analisis lebih lanjut. Meskipun terdapat keterbatasan terkait sensitivitas terhadap kondisi pencahayaan dan memerlukan posisi subjek yang stabil, sistem ini membuktikan potensi teknologi computer vision dan digital signal processing dalam aplikasi monitoring kesehatan non-invasive yang dapat diakses menggunakan perangkat sederhana seperti webcam, membuka peluang untuk pengembangan lebih lanjut dalam bidang telemedicine dan healthcare monitoring.

References

[1] W. Wang, A. C. den Brinker, S. Stuijk, and G. de Haan, "Algorithmic principles of remote ppg," *IEEE Transactions on Biomedical Engineering*, vol. 64, no. 7, pp. 1479–1491, 2017. [Online]. Available: https://ieeexplore.ieee.org/document/7565547

[2] C. Massaroni, D. Lo Presti, D. Formica, S. Silvestri, and E. Schena, "Non-contact monitoring of breathing pattern and respiratory rate via rgb signal measurement," *Sensors*, vol. 19, no. 12, 2019. [Online]. Available: https://www.mdpi.com/1424-8220/19/12/2758

[3] SciPy Developers, "scipy.signal.butter — scipy v1.11.3 manual," https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.butter.html, 2023, accessed: 2025-05-31.