Supplementary Material S2: Source Code Implementation

SAIM Universal Analysis Engine v9.2: Reproducibility Package

Takafumi Shiga

December 15, 2025

1. Data Dictionaries

1.1 Input Specification (Raw Sensors)

The engine integrates **Bilateral Inner** optical sensors to ensure robust hemodynamic tracking.

Variable	CSV Header	Description
Left Red/IR	`Optics13 / Optics7`	Left Inner Sensor Pair (660/850nm)
Right Red/IR	`Optics14 / Optics8`	Right Inner Sensor Pair (660/850nm)
EEG Bands	`Gamma, Delta, Alpha`	Absolute Band Powers (TP9, TP10)
Acceleration	`Accelerometer_X/Y/Z`	G-force ($1G = 9.81\,m/s^2$)
Heart Rate	`Heart_Rate`	Beats per minute (BPM)

1.2 Computed Metrics (Output Indices)

Metric	Name	Logic Summary
HEMO	Hemodynamic Capacity	Volatility of Bilateral HbO (MBLL)
FSI	Free State Index	Gamma/Delta log-ratio (Neural Precision)
SOM	Somatic Order	Inverse volatility of Acceleration
PE	Prediction Error Proxy	Alpha rhythm volatility (Uncertainty)
AUT	Autonomic Complexity	Heart Rate Entropy
NCI	Neural Complexity Index	Global Order Parameter

2. Source Code Listing

Note: This listing contains the complete, unabridged Python implementation used for the analysis.

Listing 1: SAIM_Engine_v9_2.py

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.signal import detrend
from scipy.stats import entropy
import warnings
import os

# Suppress warnings for clean output
warnings.filterwarnings('ignore')

# ============================================================
#   SAIM Config v9.2 (Bilateral Fusion & Adaptive Gating)
```

```
# ============================================================
class SAIMConfig:
    # --- Analysis Parameters ---
    WINDOW_SEC = 10
    STEP_SEC = 2
    MIN_DATA_POINTS = 5
    EPS = 1e-9

    # --- MBLL Parameters (See S1 Eq. 2) ---
    # Extinction Coefficient Matrix E [mM^-1 cm^-1]
    # Row 1: 660nm [HbO, HbR], Row 2: 850nm [HbO, HbR]
    E = np.array([
        [0.087, 0.730],
        [0.052, 0.032]
    ])
    DPF = 6.0  # Differential Pathlength Factor

    # --- Adaptive Weights (Priors) ---
    # Initial weights for Reliability Gating (See S1 Eq. 10)
    W_DEFAULTS = {
        'FSI': 0.35, 'SOM': 0.35, 'AUT': 0.15, 'HEMO': 0.15, 'PE': 0.30
    }

    # --- Phase Definitions ---
    PHASE_ORDER = [
        '01_Pre', '02_PostImmed', '03_PostStress', '04_PostRest',
        '05_RescueImmed', '06_RescuePost'
    ]

    LABELS_SHAM = {
        '01_Pre': 'I: Pre', '02_PostImmed': 'II: PostImmed',
        '03_PostStress': 'III: PostStress', '04_PostRest': 'IV: PostRest',
        '05_RescueImmed': 'V: RescueImmed', '06_RescuePost': 'VI: RescuePost'
    }
    LABELS_REAL = LABELS_SHAM

# ============================================================
#   Core Processing Class
# ============================================================
class SAIMAnalyzer:
    def __init__(self, subject_id, file_map, is_sham=False):
        self.subject_id = subject_id
        self.file_map = file_map
        self.is_sham = is_sham
        self.df_final = pd.DataFrame()
        self.active_metrics = set()
        self.labels = SAIMConfig.LABELS_SHAM

        # Pre-compute Inverse Matrix for MBLL (See S1 Eq. 1)
        try:
            self.E_inv = np.linalg.inv(SAIMConfig.E)
        except:
            self.E_inv = None
            print("[Error] Matrix inversion failed.")

    def _load_clean_data(self, filepath):
        try:
            if not os.path.exists(filepath):
                if os.path.exists(os.path.basename(filepath)):
                    filepath = os.path.basename(filepath)
                else: return None

            df = pd.read_csv(filepath, low_memory=False)
            if 'TimeStamp' not in df.columns: return None

            df['TimeStamp'] = pd.to_datetime(df['TimeStamp'], errors='coerce')
            df = df.dropna(subset=['TimeStamp']).sort_values('TimeStamp').reset_index(drop=
                True)

            # Filter rows with hardware artifacts
            if 'Elements' in df.columns:
                df = df[df['Elements'].isna() | (df['Elements'] == '')]

```

```
            cols_to_check = ['Optics', 'Gamma', 'Delta', 'Alpha', 'Accelerometer', 'Heart']
            cols_to_num = [c for c in df.columns if any(k in c for k in cols_to_check)]
            for c in cols_to_num: df[c] = pd.to_numeric(df[c], errors='coerce')

            return df
        except: return None

    def _calculate_brain_hemo(self, df_win):
        """
        Derives Bilateral Hemodynamic Capacity (HEMO) via MBLL.
        Integrates Left Inner (13/7) and Right Inner (14/8) sensors.
        Reference: S1 Section 1.1 - 1.2
        """
        c = SAIMConfig

        # Define Sensor Pairs [Red_Col, IR_Col]
        sensor_pairs = [
            ('Optics13', 'Optics7'), # Left Inner
            ('Optics14', 'Optics8')  # Right Inner
        ]

        valid_hbo_stds = []

        for col_red, col_ir in sensor_pairs:
            if col_red not in df_win.columns or col_ir not in df_win.columns:
                continue

            try:
                raw_red = df_win[col_red].replace(0, np.nan).dropna()
                raw_ir  = df_win[col_ir].replace(0, np.nan).dropna()

                common_idx = raw_red.index.intersection(raw_ir.index)
                if len(common_idx) < c.MIN_DATA_POINTS: continue

                raw_red = raw_red.loc[common_idx]
                raw_ir  = raw_ir.loc[common_idx]

                # 1. Optical Density
                mean_red = raw_red.mean()
                mean_ir  = raw_ir.mean()
                if mean_red <= 0 or mean_ir <= 0: continue

                od_red = -np.log(raw_red / mean_red)
                od_ir  = -np.log(raw_ir / mean_ir)

                # 2. MBLL Inversion (See S1 Eq. 1)
                od_matrix = np.vstack((od_red.values / c.DPF, od_ir.values / c.DPF))
                conc_matrix = self.E_inv @ od_matrix
                hbo_series = conc_matrix[0, :] # Extract HbO

                valid_hbo_stds.append(np.std(hbo_series))

            except: continue

        # 3. Spatial Fusion & Activation (See S1 Eq. 3, 4)
        if not valid_hbo_stds:
            return np.nan

        # Average the volatility across valid hemispheres
        global_sigma = np.mean(valid_hbo_stds)

        scale_factor = 1000  # mM -> uM
        theta = 2.0          # Thermodynamic Noise Floor

        hemo = 1.0 / (1.0 + np.exp(-(np.log(global_sigma * scale_factor + c.EPS) - theta)))

        return hemo

    def _calc_metrics(self, df_win, fs_est):
        c = SAIMConfig

        # --- FSI (See S1 Eq. 5, 6) ---
        g_cols = [col for col in df_win.columns if 'Gamma' in col]
```

```
        d_cols = [col for col in df_win.columns if 'Delta' in col]
        fsi = np.nan
        if g_cols and d_cols:
            g_val = df_win[g_cols].mean().mean()
            d_val = df_win[d_cols].mean().mean()
            is_log = (df_win[g_cols].min().min() < 0)
            try:
                fsi_raw = (g_val - d_val) if is_log else np.log((g_val+c.EPS)/(d_val+c.EPS))
                fsi = 1 / (1 + np.exp(-fsi_raw))
            except: pass

        # --- SOM (See S1 Eq. 7) ---
        acc_cols = [col for col in df_win.columns if 'Accelerometer' in col]
        som = np.nan
        if acc_cols:
            mag = np.sqrt(np.sum(df_win[acc_cols]**2, axis=1))
            som = 1.0 / (1.0 + np.std(mag))

        # --- PE (See S1 Eq. 8) ---
        a_cols = [col for col in df_win.columns if 'Alpha' in col]
        pe = np.nan
        if a_cols:
            a_val = np.nanmean(df_win[a_cols], axis=1)
            mask = np.isfinite(a_val)
            a_clean = a_val[mask]
            if len(a_clean) > c.MIN_DATA_POINTS:
                try:
                    pe = np.std(detrend(a_clean)) / (np.mean(np.abs(a_clean)) + c.EPS)
                except: pe = np.nan

        # --- AUT (See S1 Eq. 9) ---
        aut = np.nan
        if 'Heart_Rate' in df_win.columns:
            hr = df_win['Heart_Rate'].dropna().values
            if len(hr) > 3:
                counts, _ = np.histogram(hr, bins='fd')
                aut = entropy(counts/np.sum(counts)) / (np.log(len(counts)+1) + c.EPS)

        # --- HEMO ---
        hemo = self._calculate_brain_hemo(df_win)

        return fsi, som, pe, aut, hemo, np.nan

    def run_analysis(self):
        mode_label = "SHAM" if self.is_sham else "REAL"
        print(f"--- Processing {self.subject_id} [{mode_label}] ---")

        c = SAIMConfig
        raw_data = []

        # Data Loading Loop
        for key in c.PHASE_ORDER:
            if key not in self.file_map: continue
            df = self._load_clean_data(self.file_map[key])
            if df is None: continue

            label = self.labels.get(key, key)
            print(f" > Processing {key} ({label})...")

            curr, end = df['TimeStamp'].iloc[0], df['TimeStamp'].iloc[-1]
            while curr < end:
                nxt = curr + pd.Timedelta(seconds=c.WINDOW_SEC)
                df_win = df[(df['TimeStamp'] >= curr) & (df['TimeStamp'] < nxt)]

                if len(df_win) >= c.MIN_DATA_POINTS:
                    metrics = self._calc_metrics(df_win, 1.0)
                    raw_data.append((label, key, *metrics))
                curr += pd.Timedelta(seconds=c.STEP_SEC)

        if not raw_data:
            print("[Error] No valid data extracted.")
            return

```

```
        df_raw = pd.DataFrame(raw_data, columns=['Phase', 'PhaseKey', 'FSI', 'SOM', 'PE', 'AUT', 'HEMO', 'RR'])

        # --- Adaptive Reliability Gating (See S1 Eq. 10) ---
        # Logic: Include metric only if variance > 1e-6 (Noise Floor)
        valid_weights = {
            m: c.W_DEFAULTS[m] for m in ['FSI', 'SOM', 'AUT', 'HEMO']
            if df_raw[m].count() > 0 and df_raw[m].std() > 1e-6
        }

        total_w = sum(valid_weights.values())
        if total_w > 0:
            for k in valid_weights: valid_weights[k] /= total_w

        self.active_metrics = set(valid_weights.keys())
        print(f" > Active Metrics for NCI: {list(self.active_metrics)}")

        # --- NCI Calculation (See S1 Eq. 11) ---
        results = []
        for _, row in df_raw.iterrows():
            score = sum(row[m] * w for m, w in valid_weights.items() if not np.isnan(row[m]))

            if not np.isnan(row['PE']):
                score -= row['PE'] * c.W_DEFAULTS['PE']

            nci = 1.0 / (1.0 + np.exp(-score))

            active_vals = [row[m] for m in valid_weights.keys() if not np.isnan(row[m])]
            f_val = (1.0 - np.mean(active_vals)) + (row['PE'] * 0.5) if active_vals else np.nan

            results.append({
                'Phase': row['Phase'], 'PhaseKey': row['PhaseKey'],
                'NCI_Z': nci, 'F_Z': f_val, 'PE_Z': row['PE'], 'HEMO_Z': row['HEMO']
            })

        self.df_final = pd.DataFrame(results)
        self._normalize_and_plot()

    def _normalize_and_plot(self):
        # Baseline Z-Scoring against Phase I (Pre)
        base_df = self.df_final[self.df_final['PhaseKey'] == '01_Pre']
        if base_df.empty: base_df = self.df_final

        for m in ['NCI_Z', 'PE_Z', 'F_Z', 'HEMO_Z']:
            mean, std = base_df[m].mean(), base_df[m].std()
            if np.isnan(std) or std < 1e-9: std = 1.0

            self.df_final[m] = (self.df_final[m] - mean) / std
            self.df_final[f'{m}_Trend'] = self.df_final[m].interpolate().rolling(5, center=True).mean()
            self.df_final[f'{m}_Std'] = self.df_final[m].interpolate().rolling(5, center=True).std()

        # --- Visualization (Reproducible Plotting Logic) ---
        sns.set_style("whitegrid")
        fig = plt.figure(figsize=(14, 18))
        plt.suptitle(f"SAIM Analysis v9.2: {self.subject_id}", fontsize=16, fontweight='bold')

        # Subplot 1: NCI & PE (Neuro-Somatic Dynamics)
        ax1 = plt.subplot(3, 1, 1)
        df_p = self.df_final.reset_index(drop=True)
        # NCI Trend with Standard Deviation Shading
        ax1.plot(df_p.index, df_p['NCI_Z_Trend'], color='#00BFFF', label='NCI (Integration)', lw=2.5)
        ax1.fill_between(df_p.index, df_p['NCI_Z_Trend']-df_p['NCI_Z_Std'],
                         df_p['NCI_Z_Trend']+df_p['NCI_Z_Std'], color='#00BFFF', alpha=0.15)
        # PE Trend
        ax1.plot(df_p.index, df_p['PE_Z_Trend'], color='#FF4500', label='PE (Error)', lw=2.5)
        self._add_boundaries(ax1, df_p)
        ax1.set_title(f"1. Neuro-Somatic Dynamics (Active: {list(self.active_metrics)})", fontsize=14)
        ax1.legend(loc='upper left')
```

```
        # Subplot 2: Metabolic Cost (F vs HEMO)
        ax2 = plt.subplot(3, 1, 2)
        ax2.plot(df_p.index, df_p['F_Z_Trend'], color='magenta', label='F (Total Cost)', lw=2.5)
        if 'HEMO' in self.active_metrics:
            ax2.plot(df_p.index, df_p['HEMO_Z_Trend'], color='red', label='HEMO (Capacity)', lw=2.5, linestyle='--')
        self._add_boundaries(ax2, df_p)
        ax2.set_title("2. Metabolic Cost & Flow", fontsize=14)
        ax2.legend(loc='upper left')

        # Subplot 3: Statistical Distribution
        ax3 = plt.subplot(3, 1, 3)
        df_melt = self.df_final.melt(id_vars='Phase', value_vars=['NCI_Z', 'PE_Z', 'F_Z', 'HEMO_Z'],
                                     var_name='Metric', value_name='Z-Score')
        sns.boxplot(data=df_melt, x='Phase', y='Z-Score', hue='Metric', palette='viridis', ax=ax3, showfliers=False)
        ax3.axhline(0, color='black', linestyle='-')
        ax3.set_title("3. Statistical Distribution", fontsize=14)

        plt.tight_layout(rect=[0, 0.03, 1, 0.95])

        # Save Outputs
        out_png = f"SAIM_Result_{self.subject_id}_v9.2.png"
        out_csv = f"SAIM_Data_{self.subject_id}_v9.2.csv"
        plt.savefig(out_png)
        self.df_final.to_csv(out_csv, index=False)
        print(f"Analysis Complete. Graphs saved to {out_png}")
        print(f"Data saved to {out_csv}")

    def _add_boundaries(self, ax, df):
        """Adds vertical separators between experimental phases."""
        boundaries = df.groupby('PhaseKey').apply(lambda x: x.index[-1]).tolist()[:-1]
        for b in boundaries: ax.axvline(b, color='gray', linestyle=':', alpha=0.8)

if __name__ == "__main__":
    # Example execution entry point
    # auto_run_subject()
    pass
```