

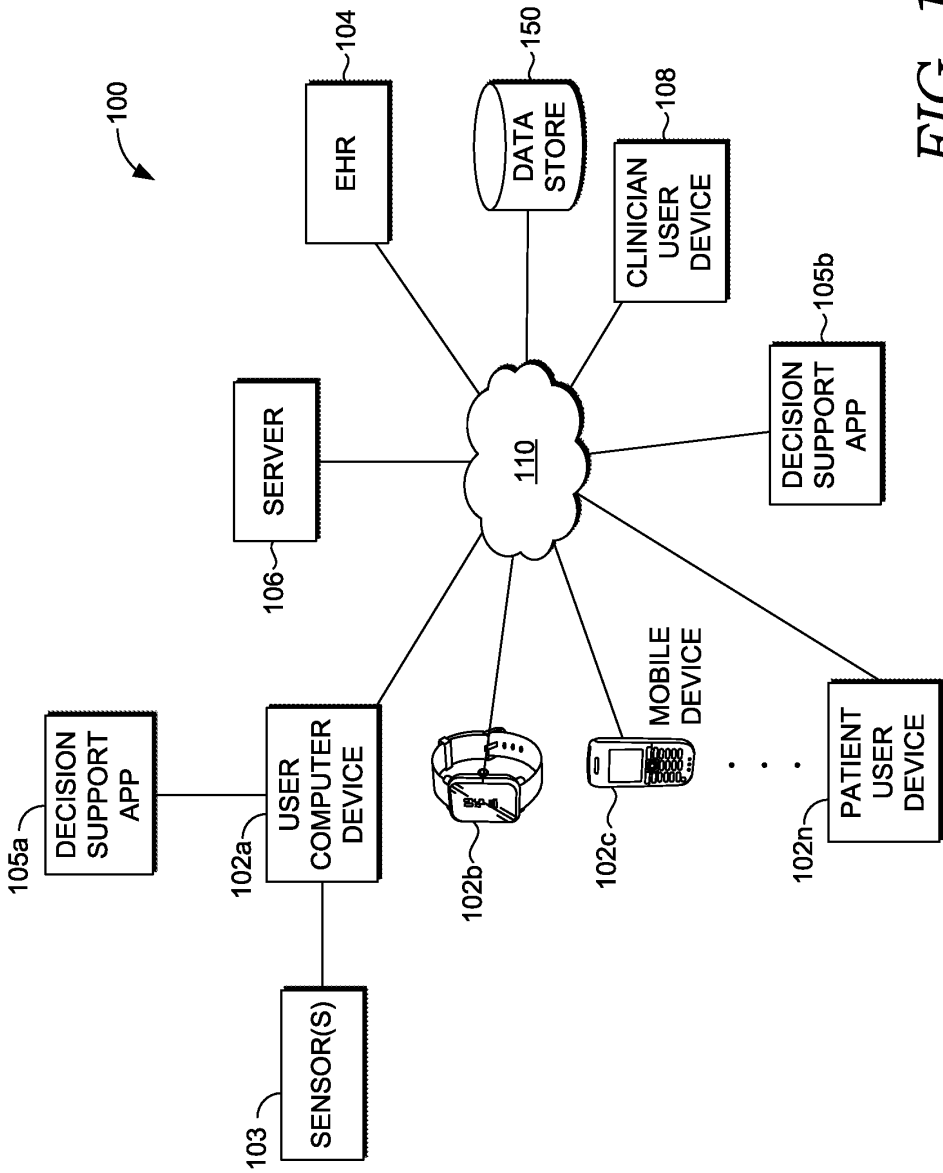


FIG. 1

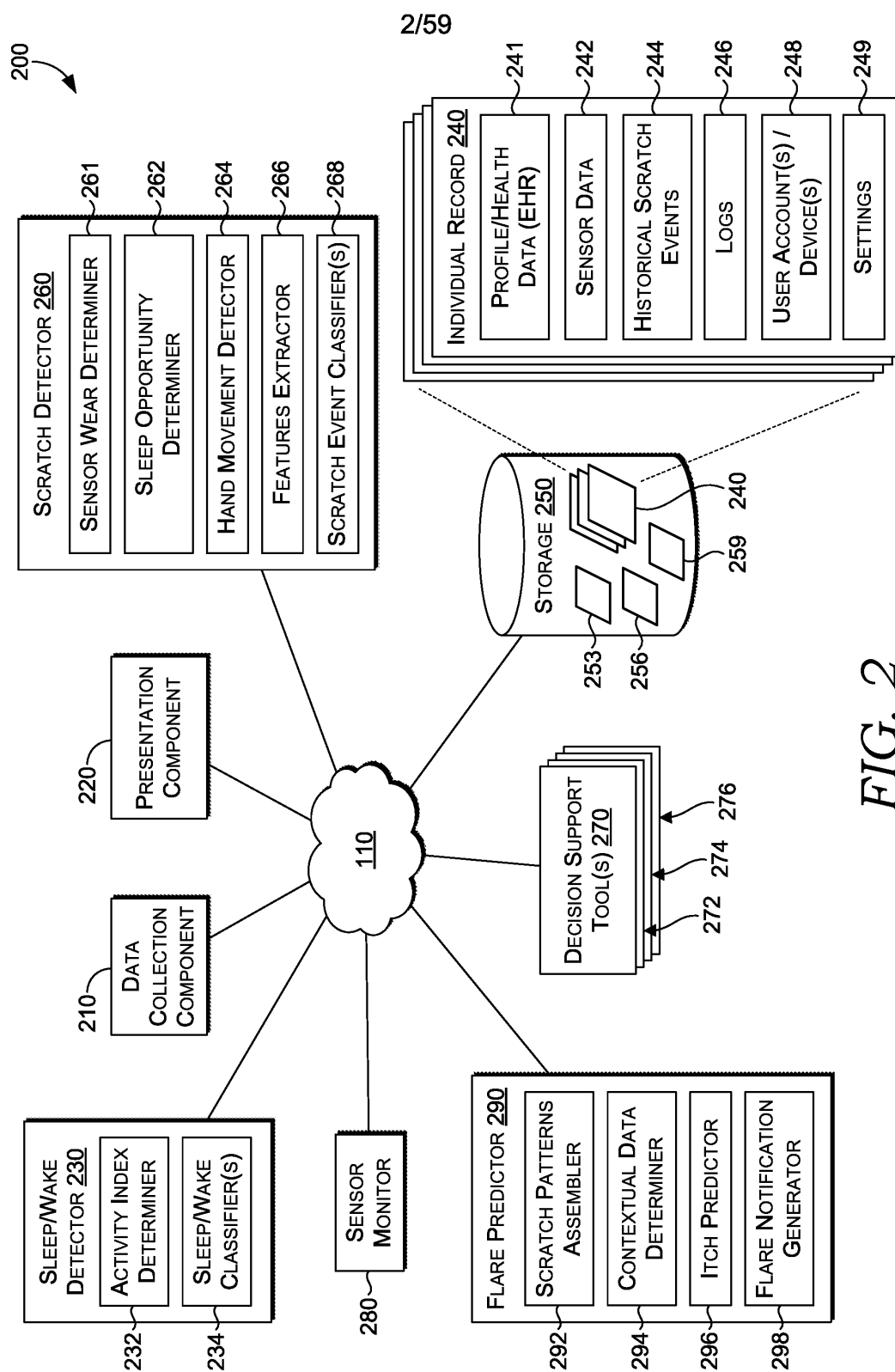
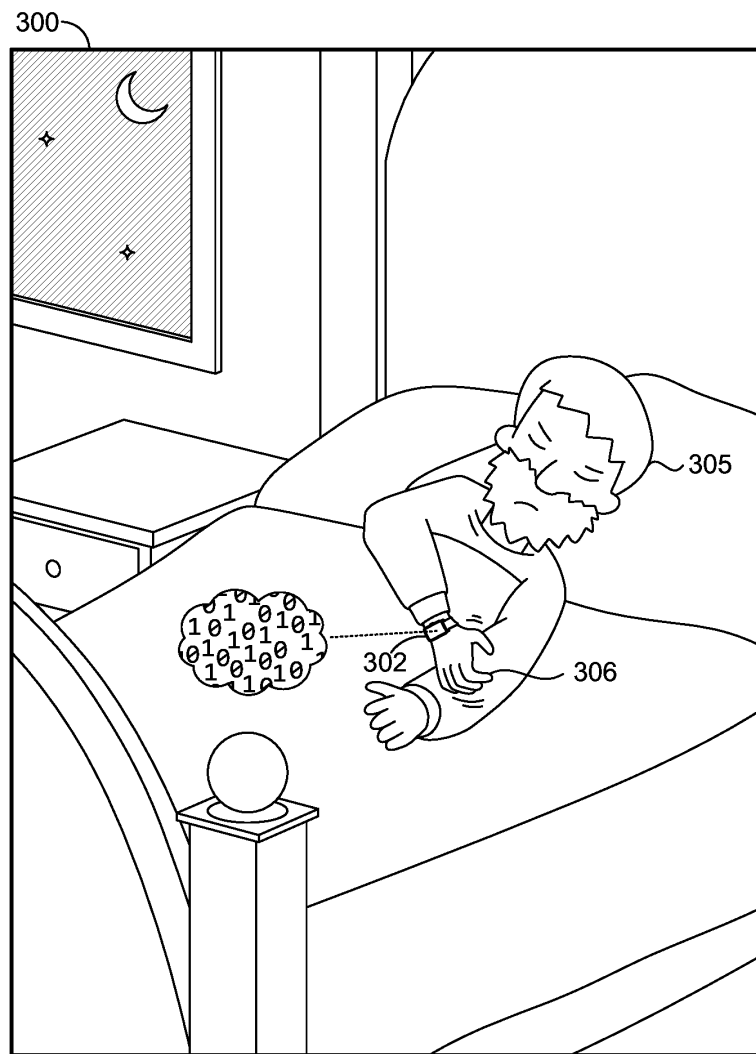


FIG. 2

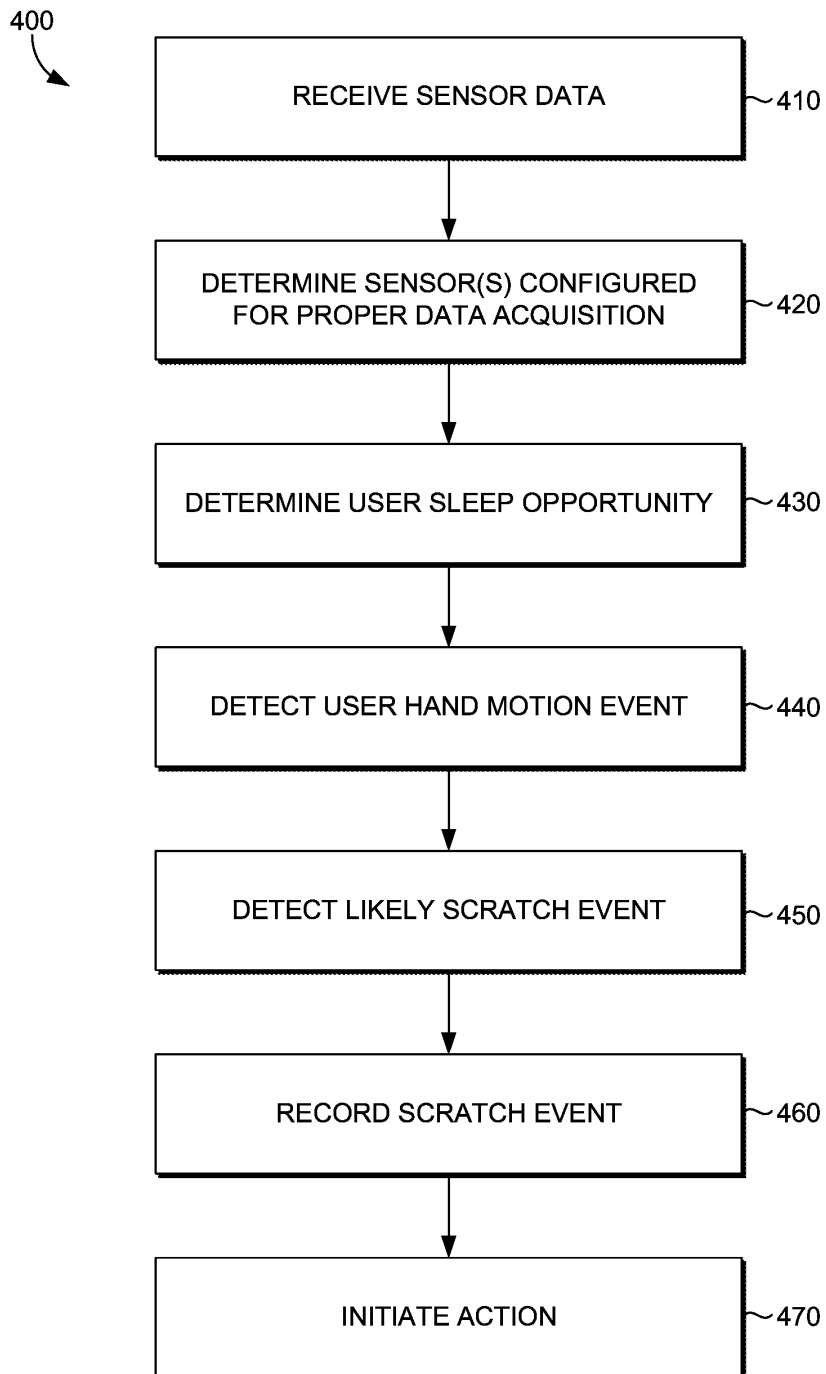
3/59

*FIG. 3A*

4/59

*FIG. 3B*

5/59

*FIG. 4A*

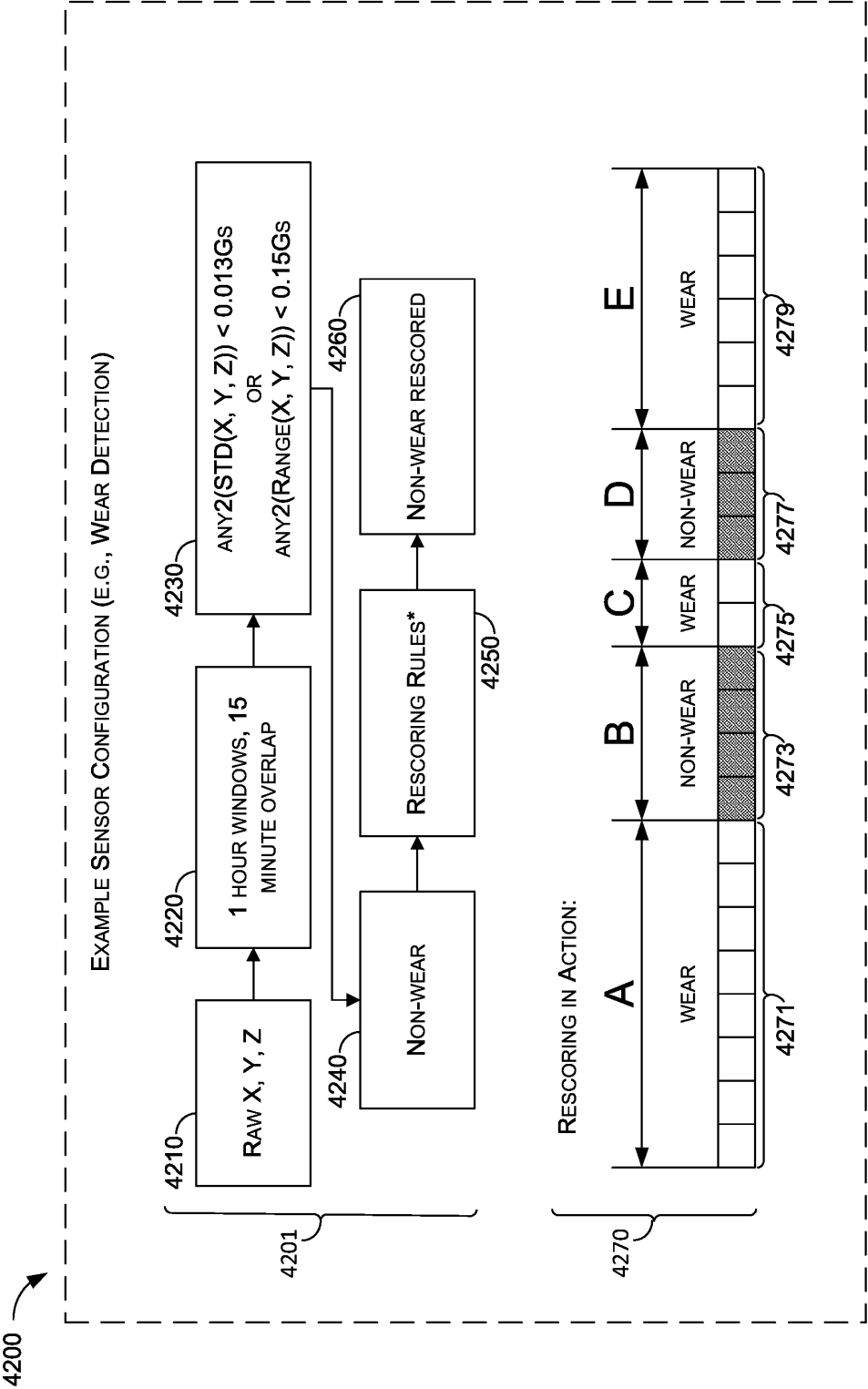


FIG. 4B

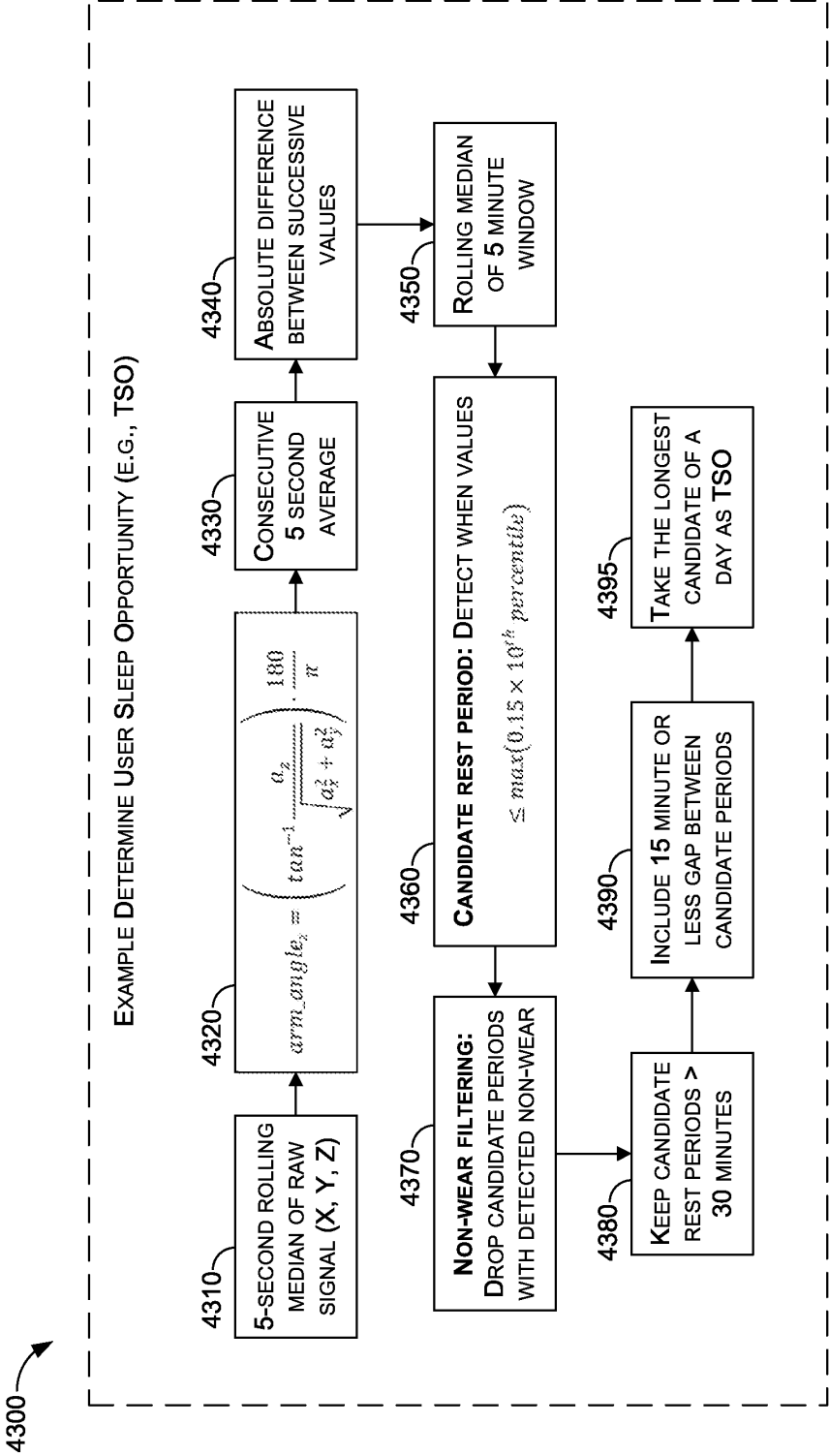


FIG. 4C

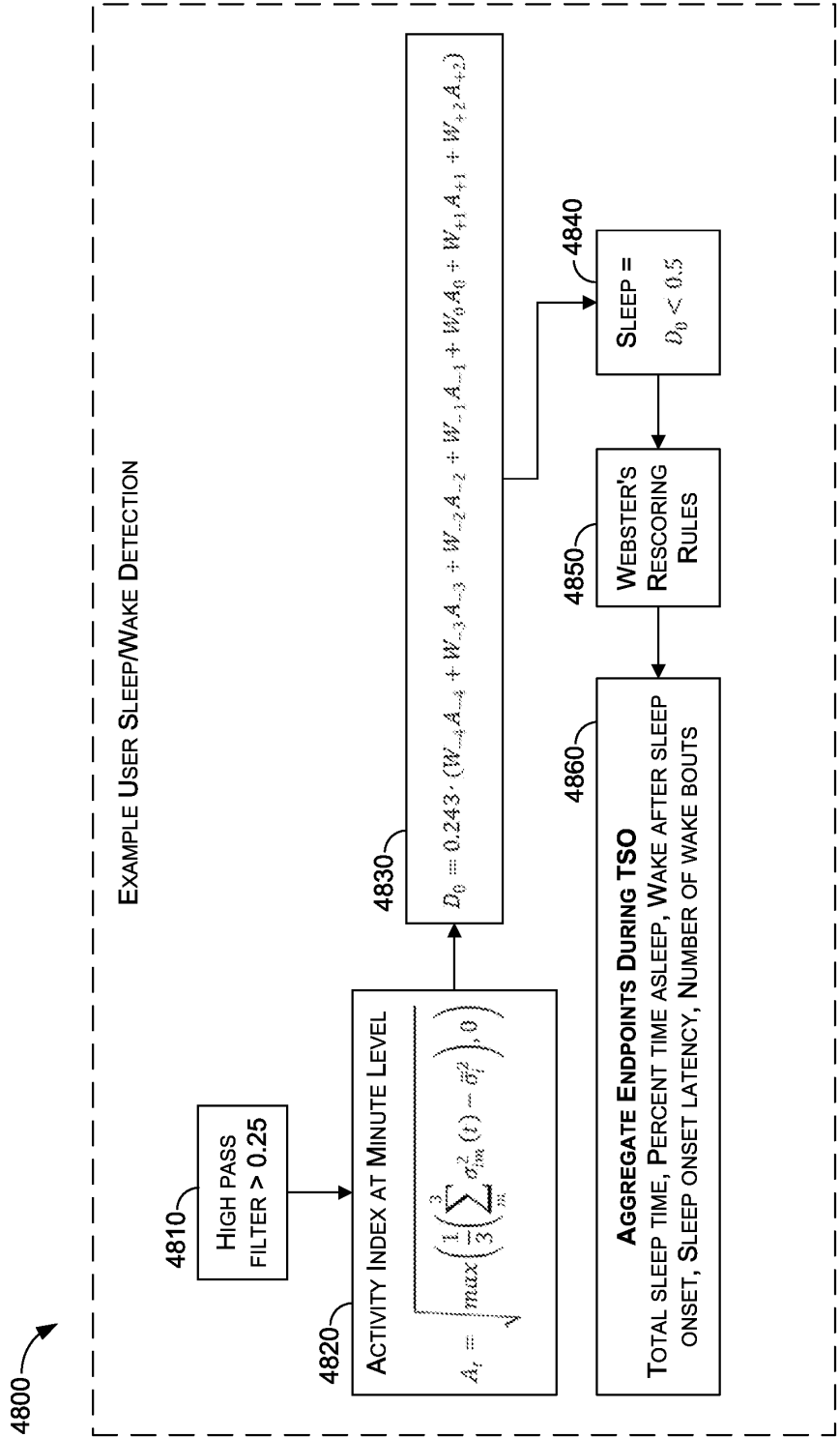


FIG. 4D

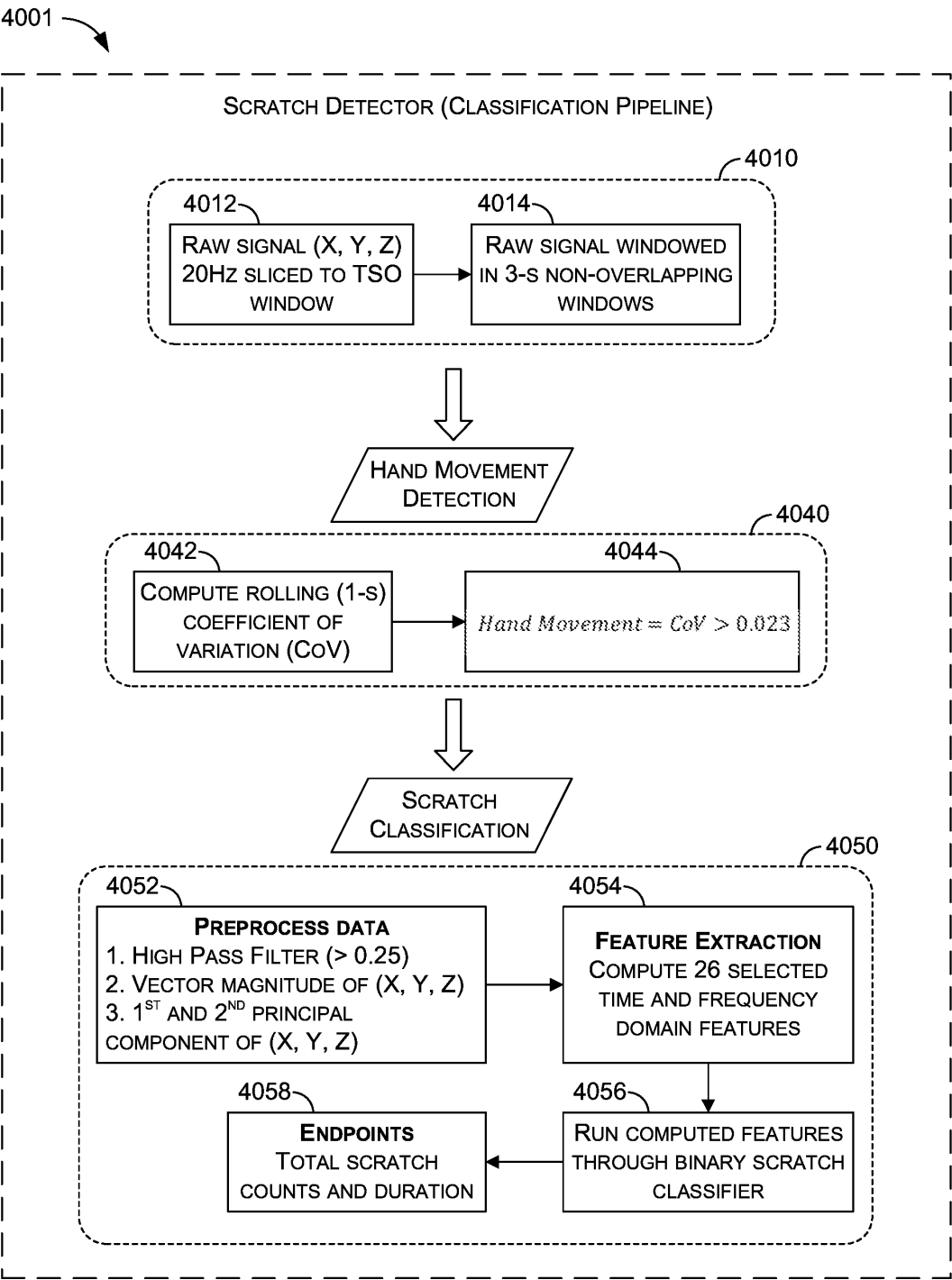
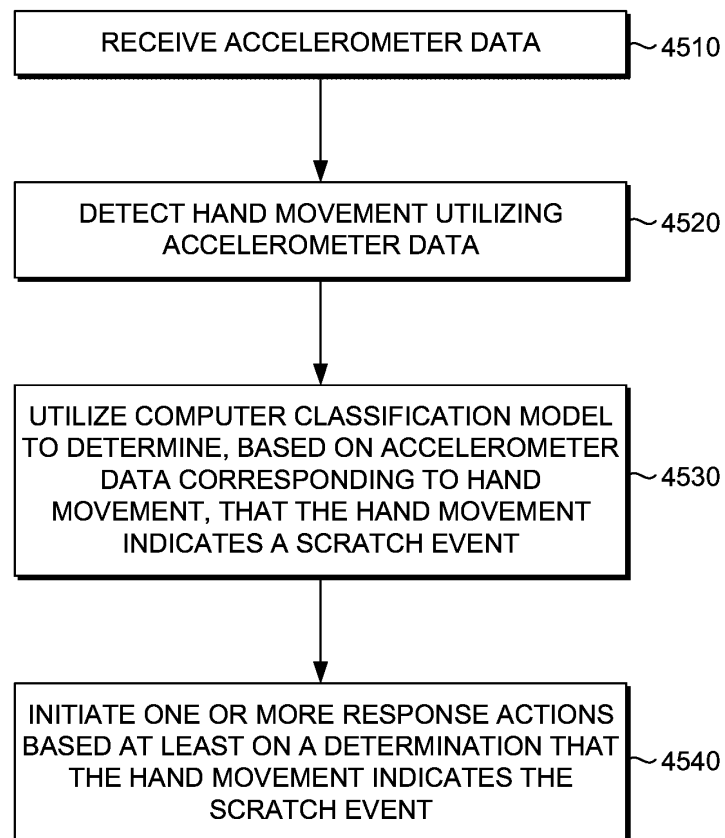


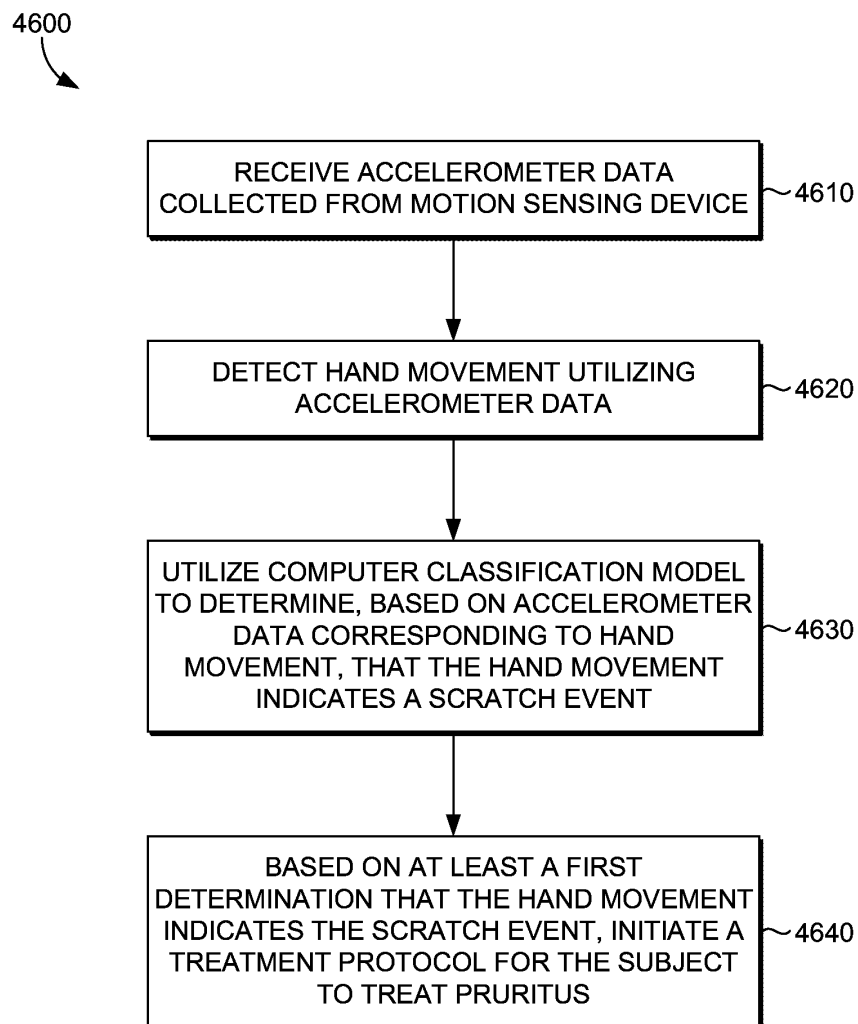
FIG. 4E

10/59

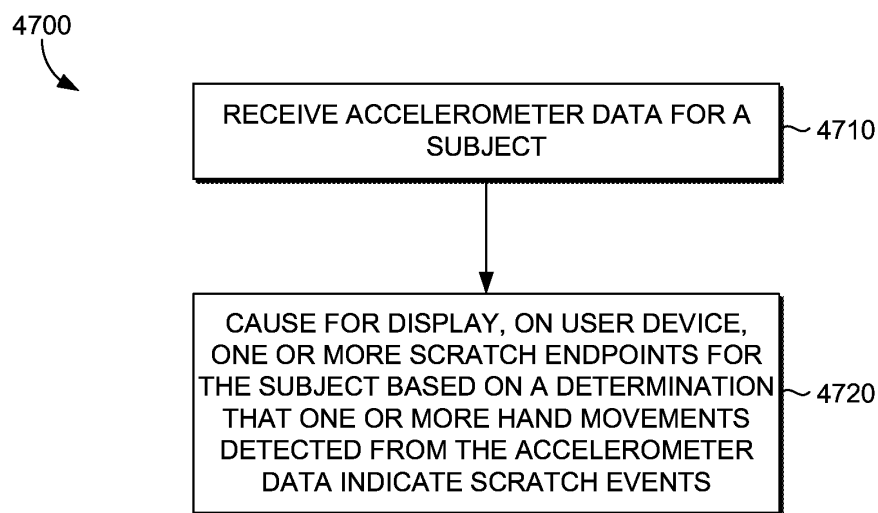
4500

*FIG. 4F*

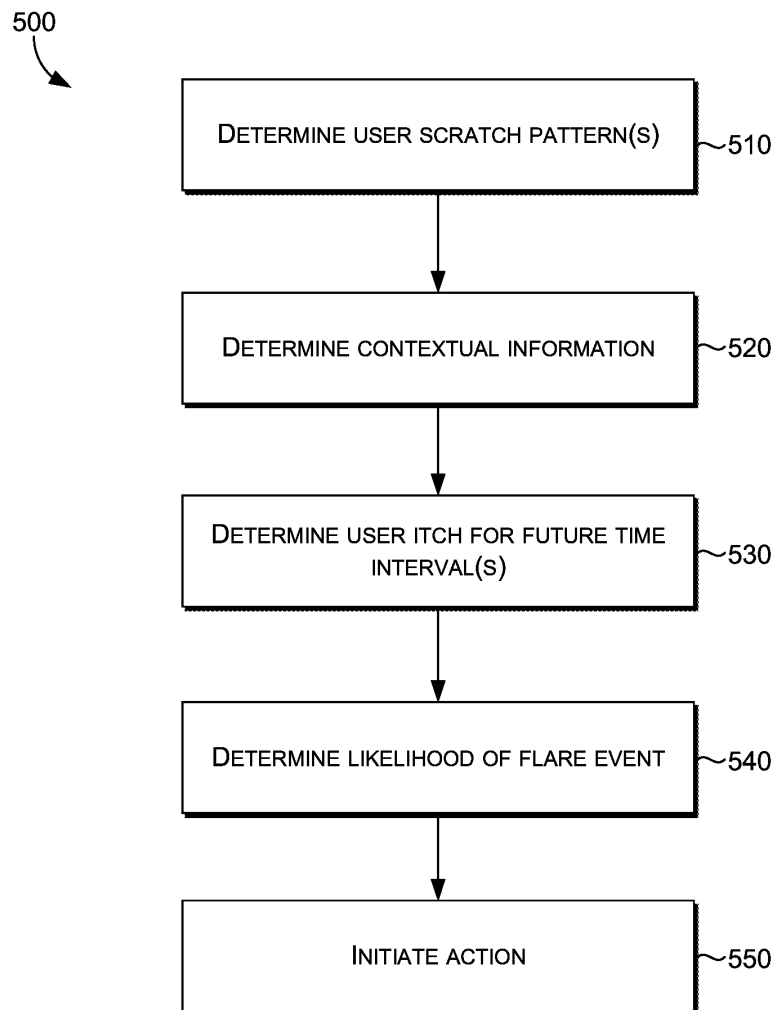
11/59

*FIG. 4G*

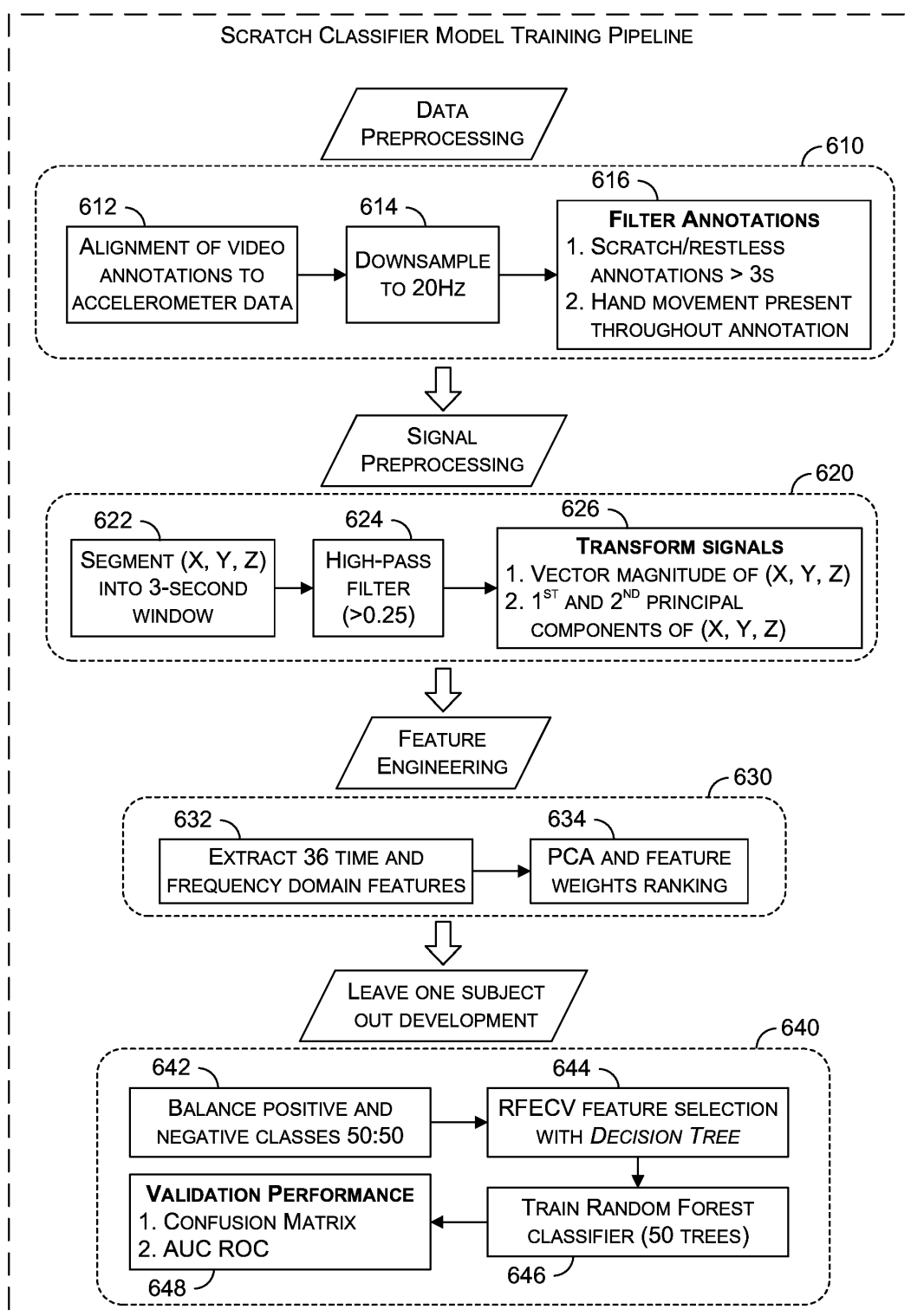
12/59

*FIG. 4H*

13/59

*FIG. 5*

14/59



600

FIG. 6A

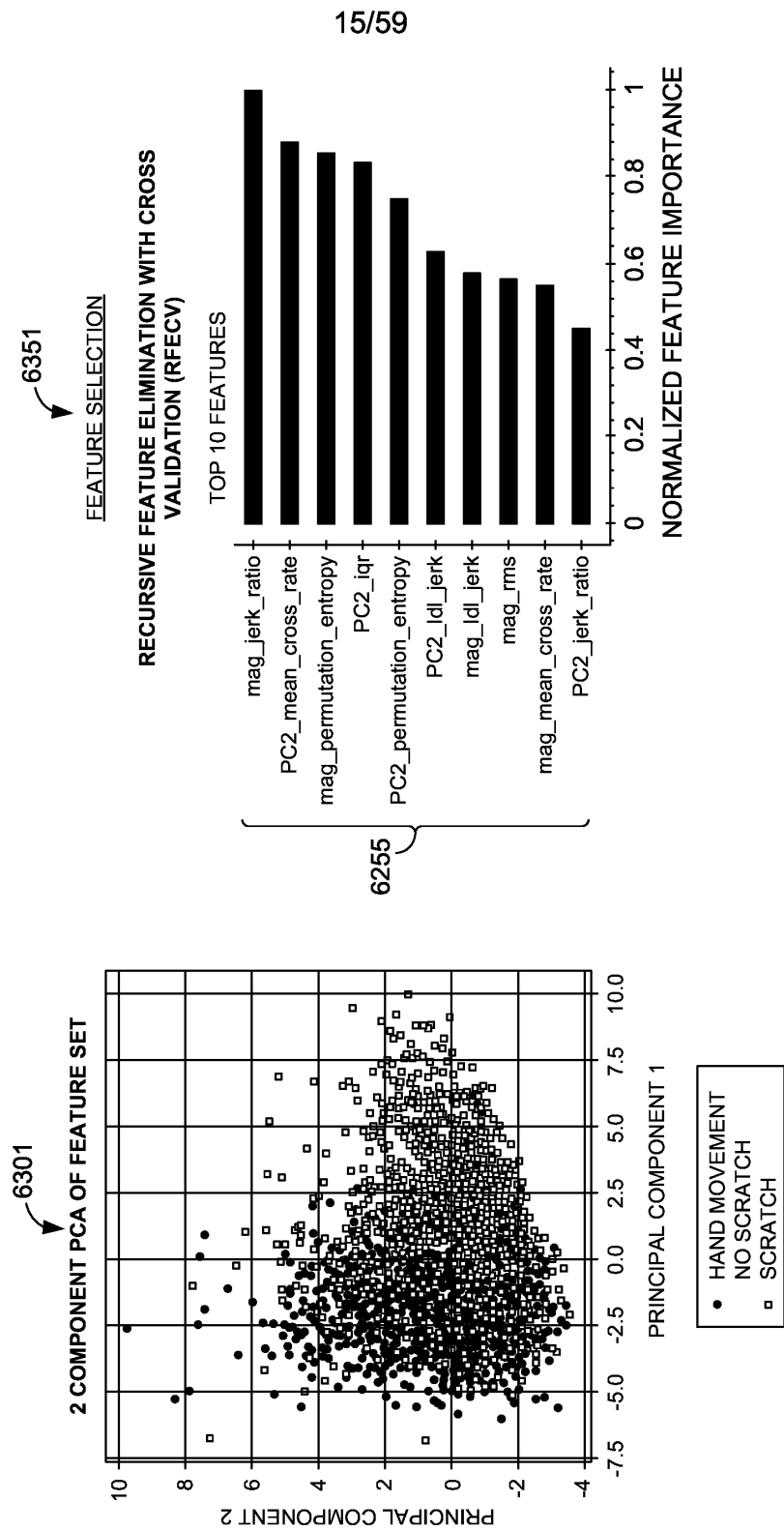


FIG. 6B

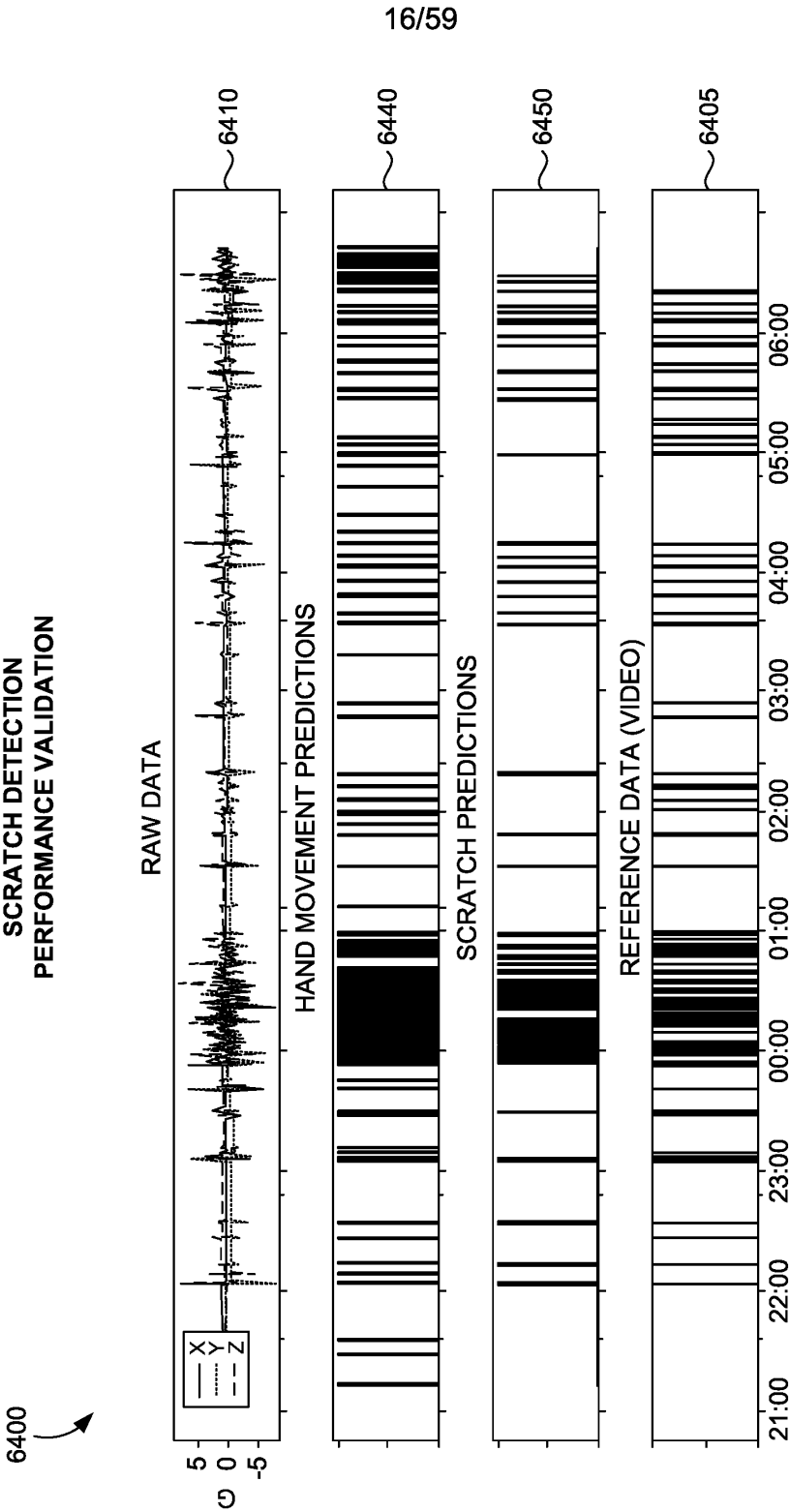
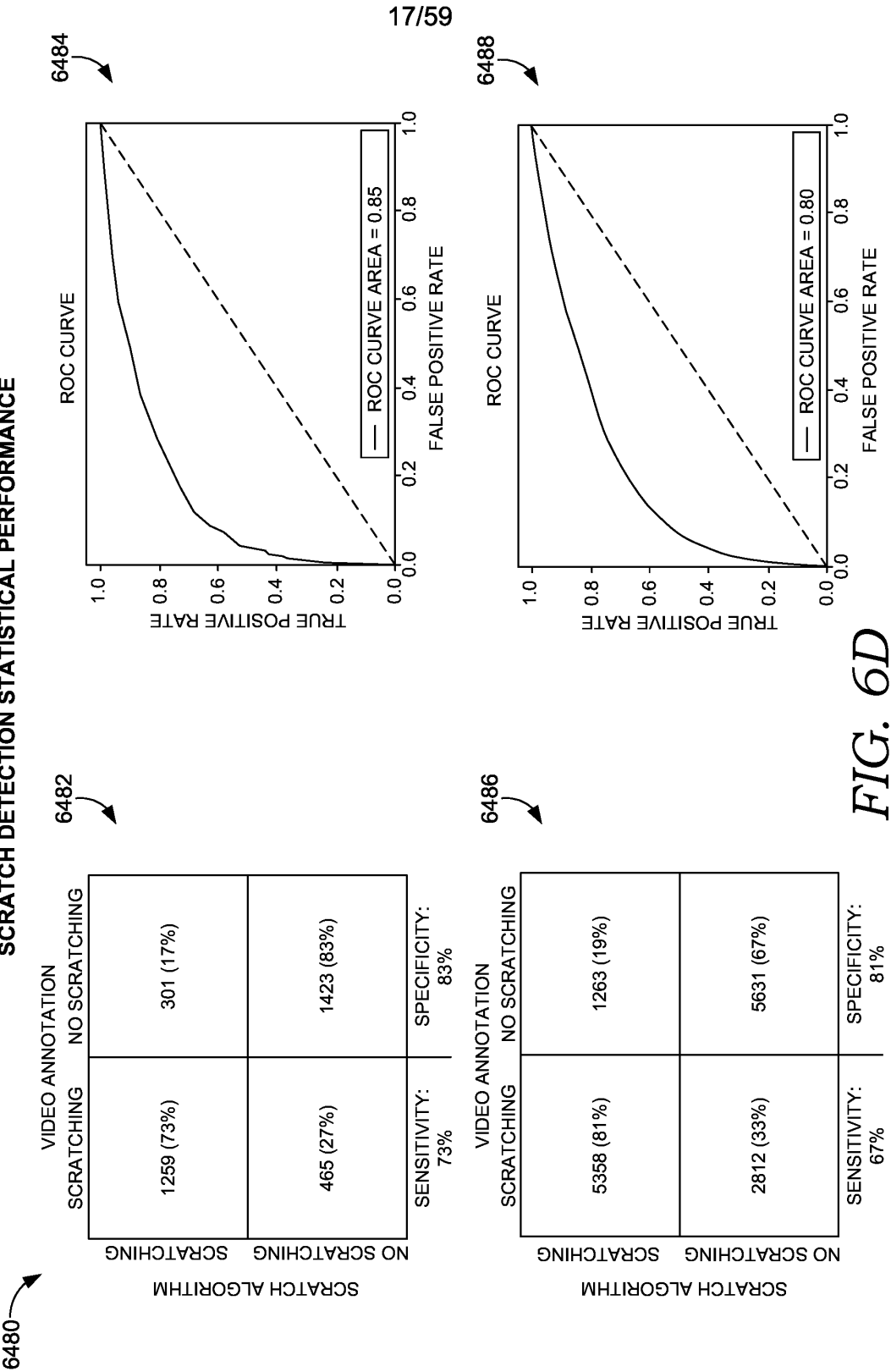


FIG. 6C



SLEEP-RELATED SIGNAL ANALYSIS

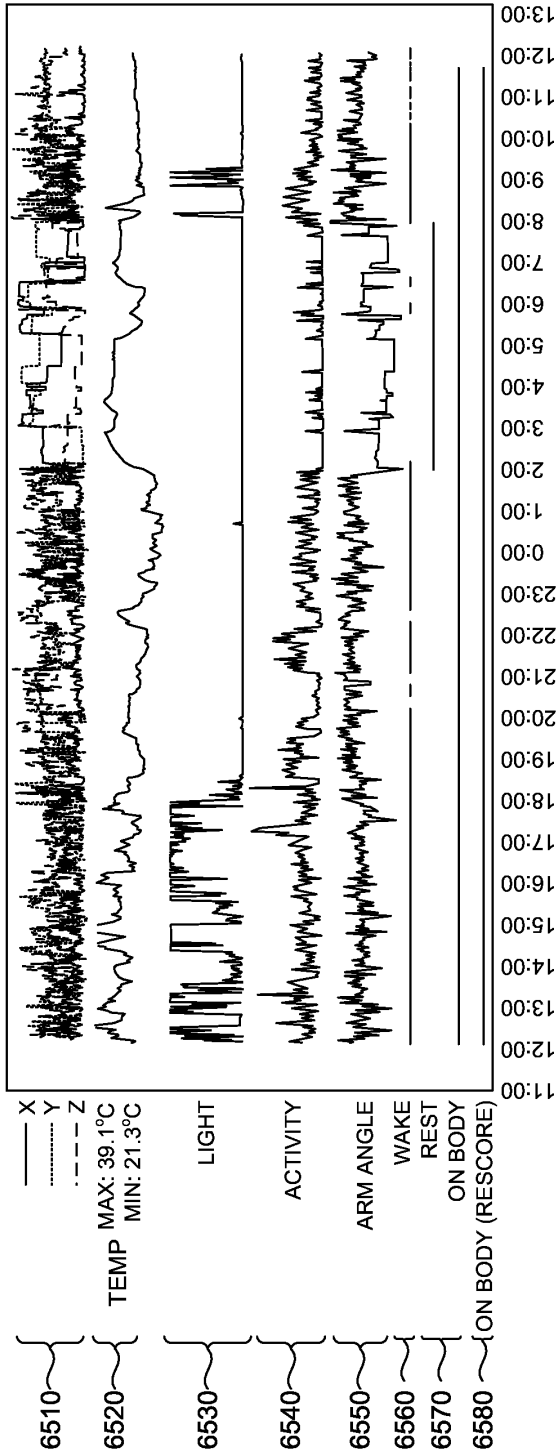


FIG. 6E

6500

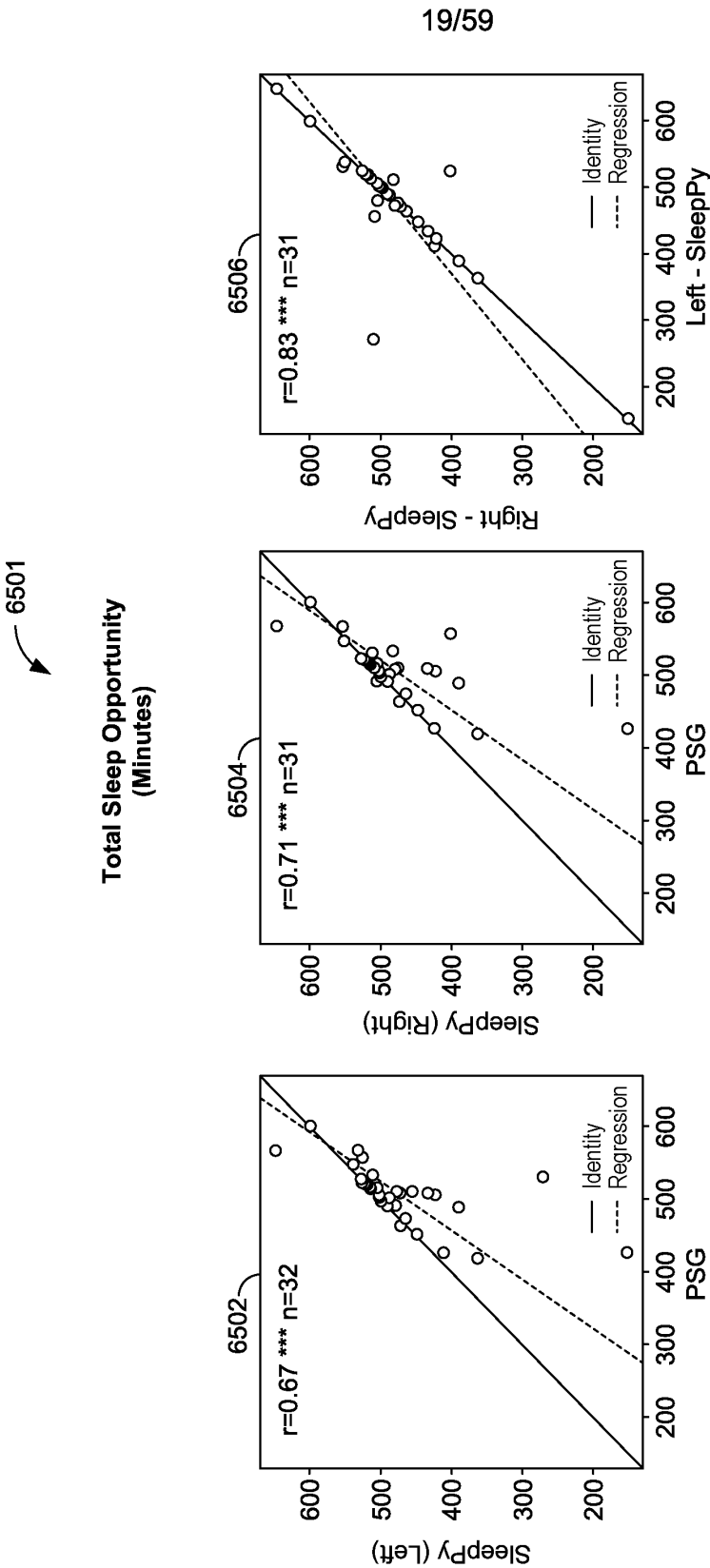


FIG. 6F

20/59

7400

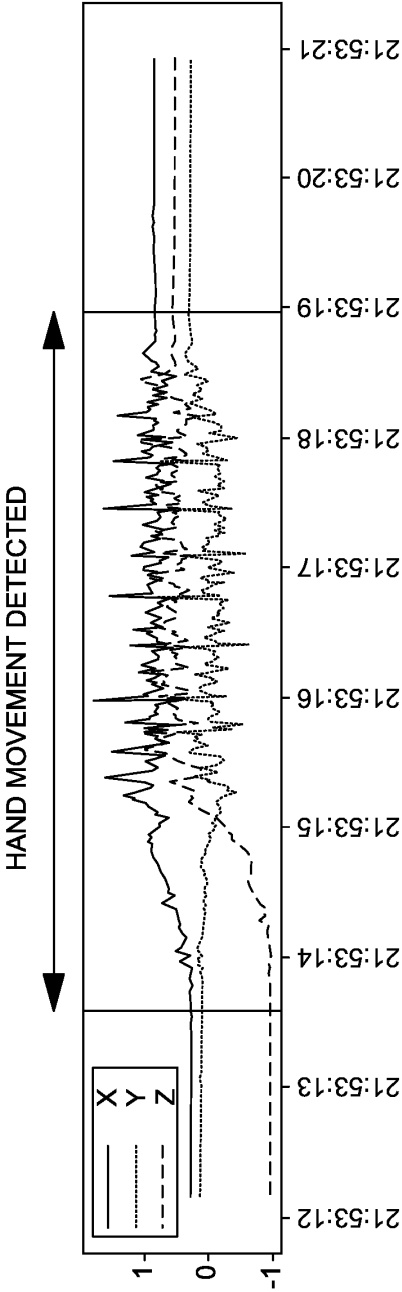


FIG. 7A

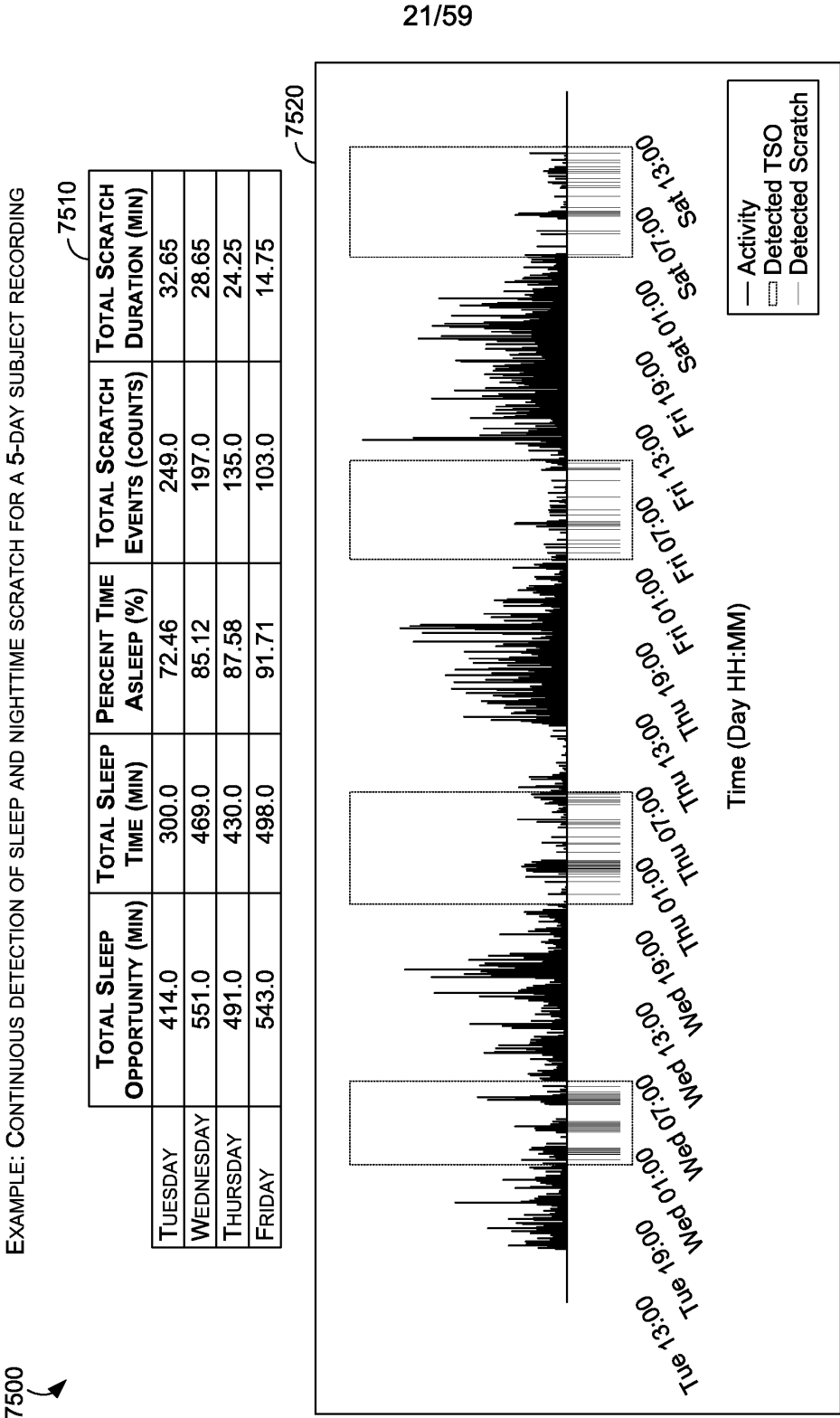


FIG. 7B

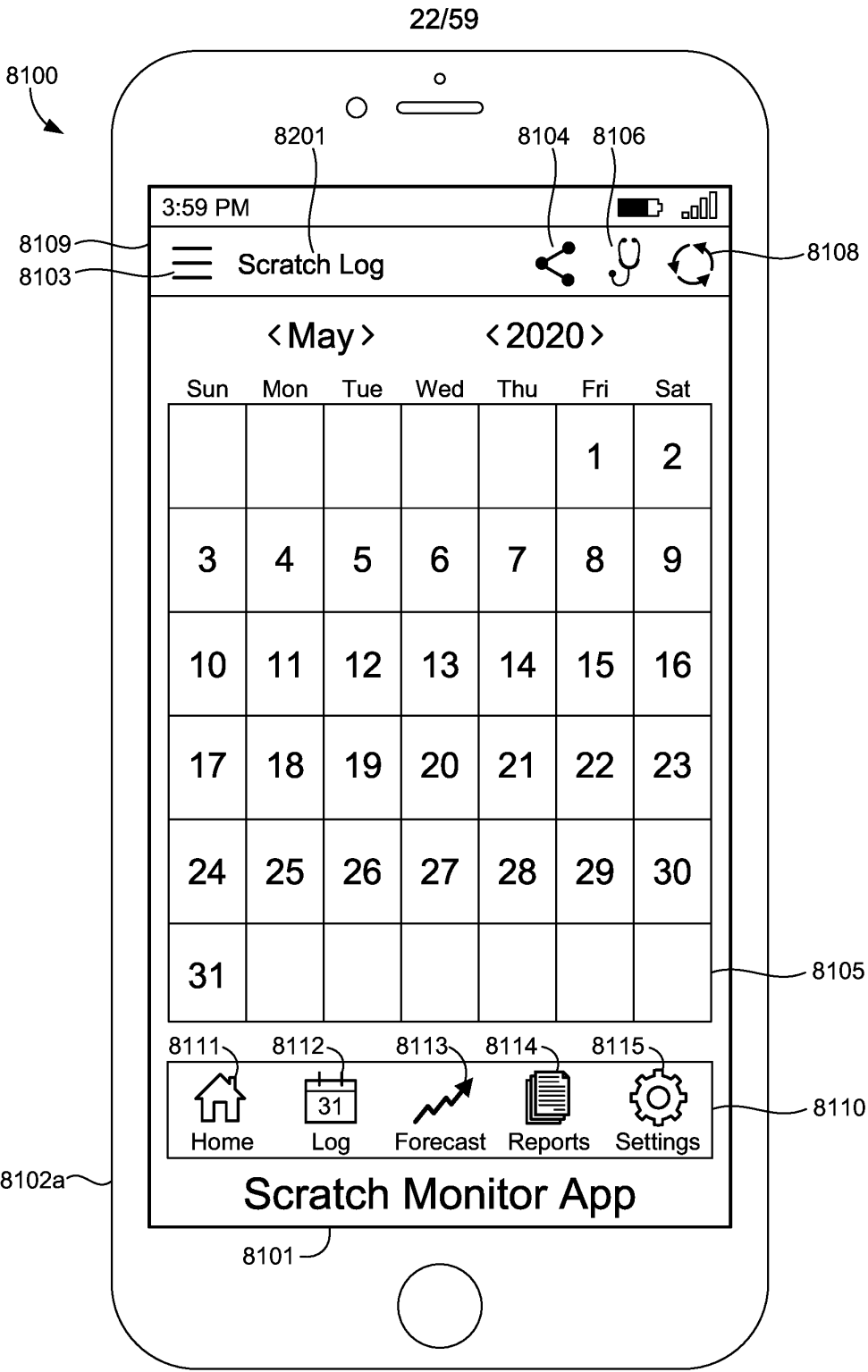


FIG. 8A

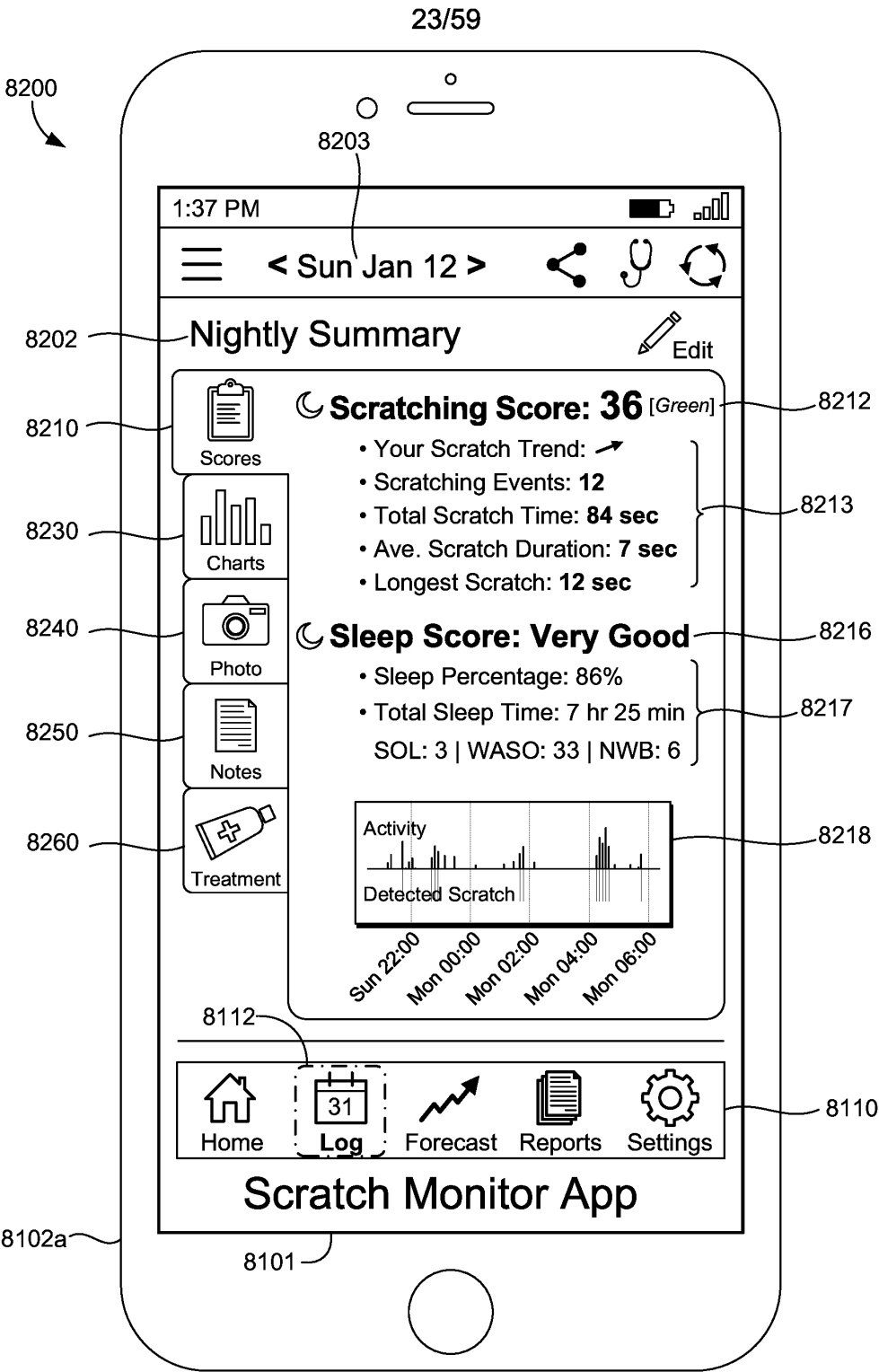


FIG. 8B

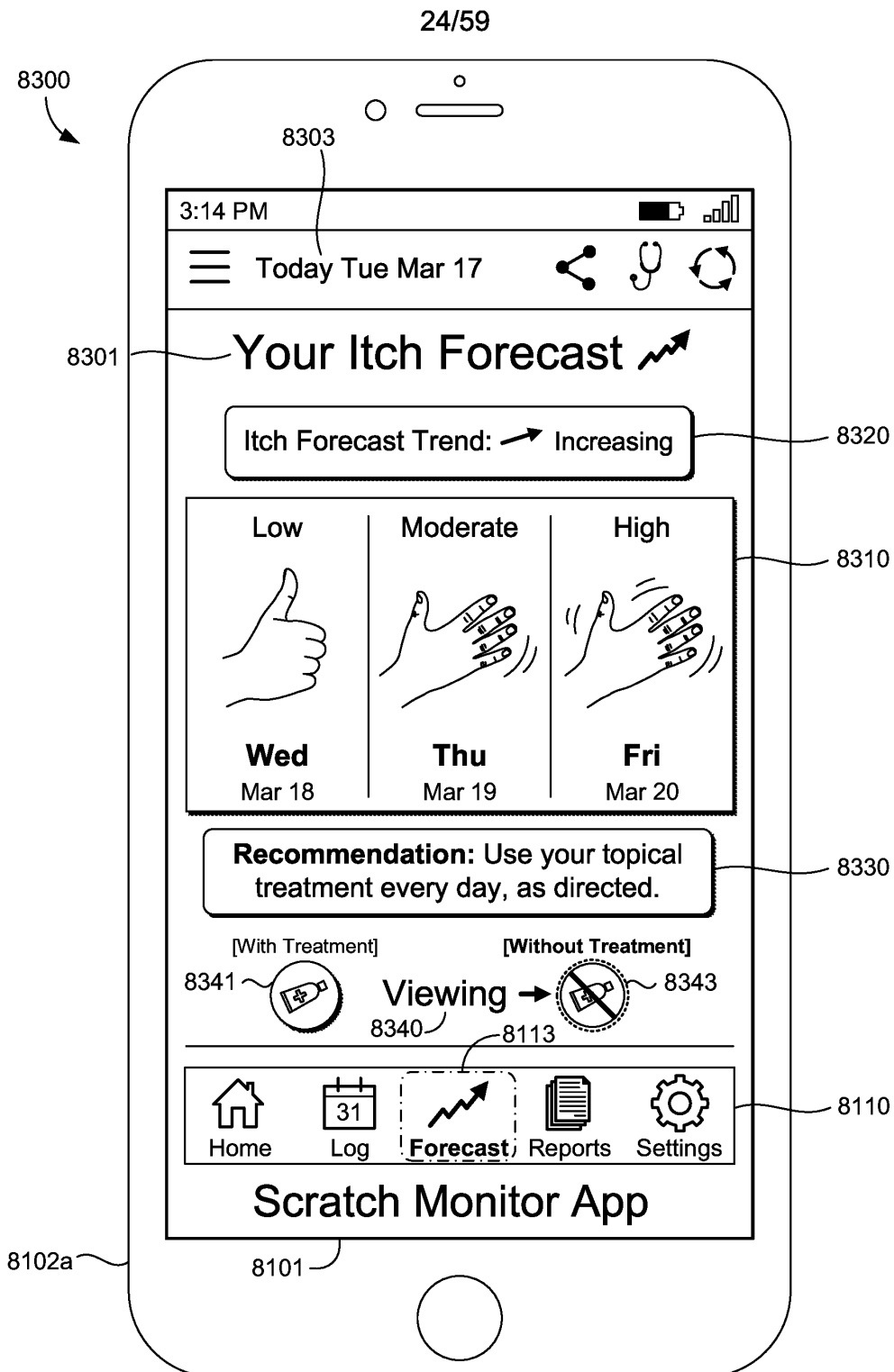


FIG. 8C

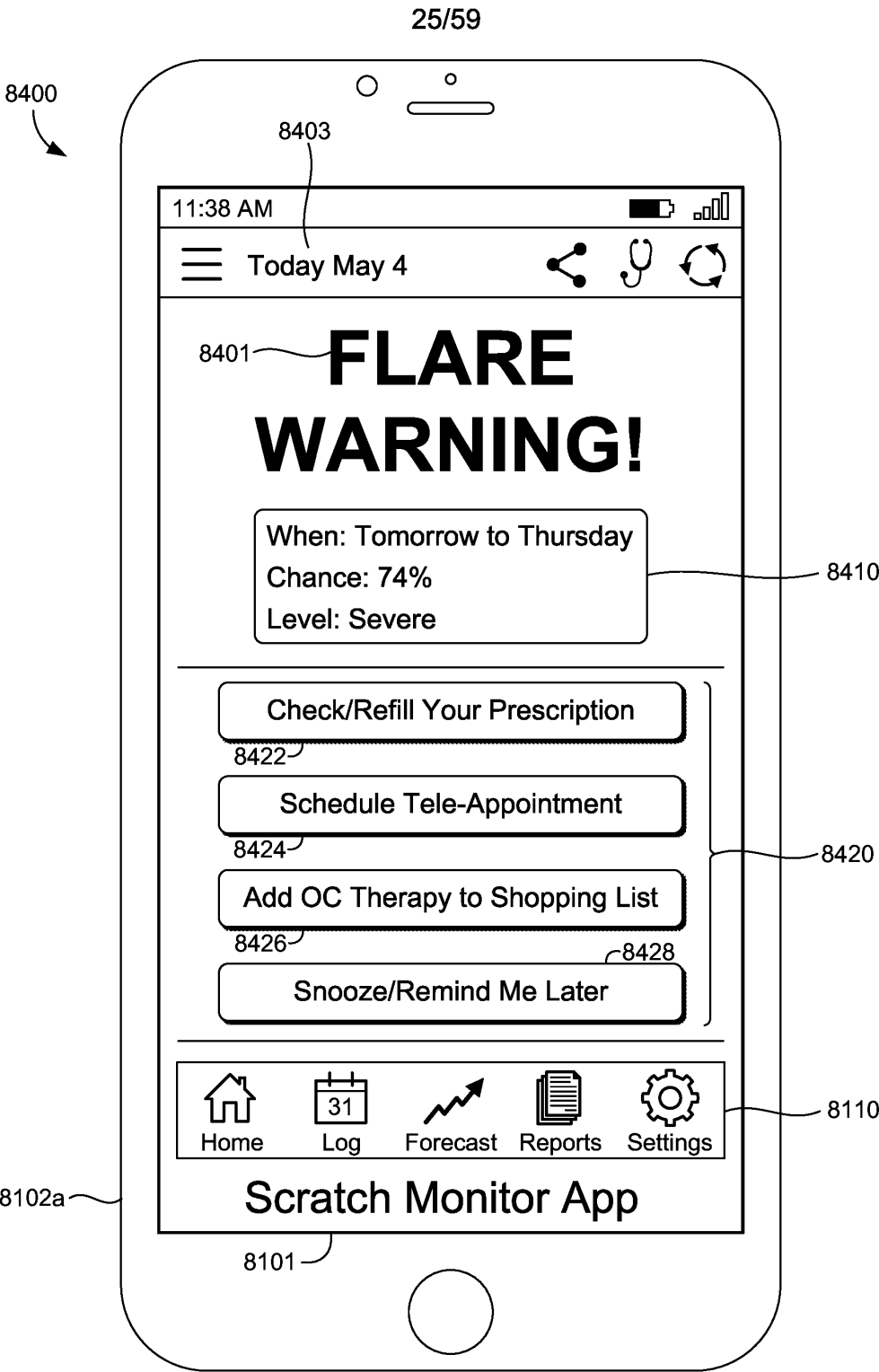


FIG. 8D

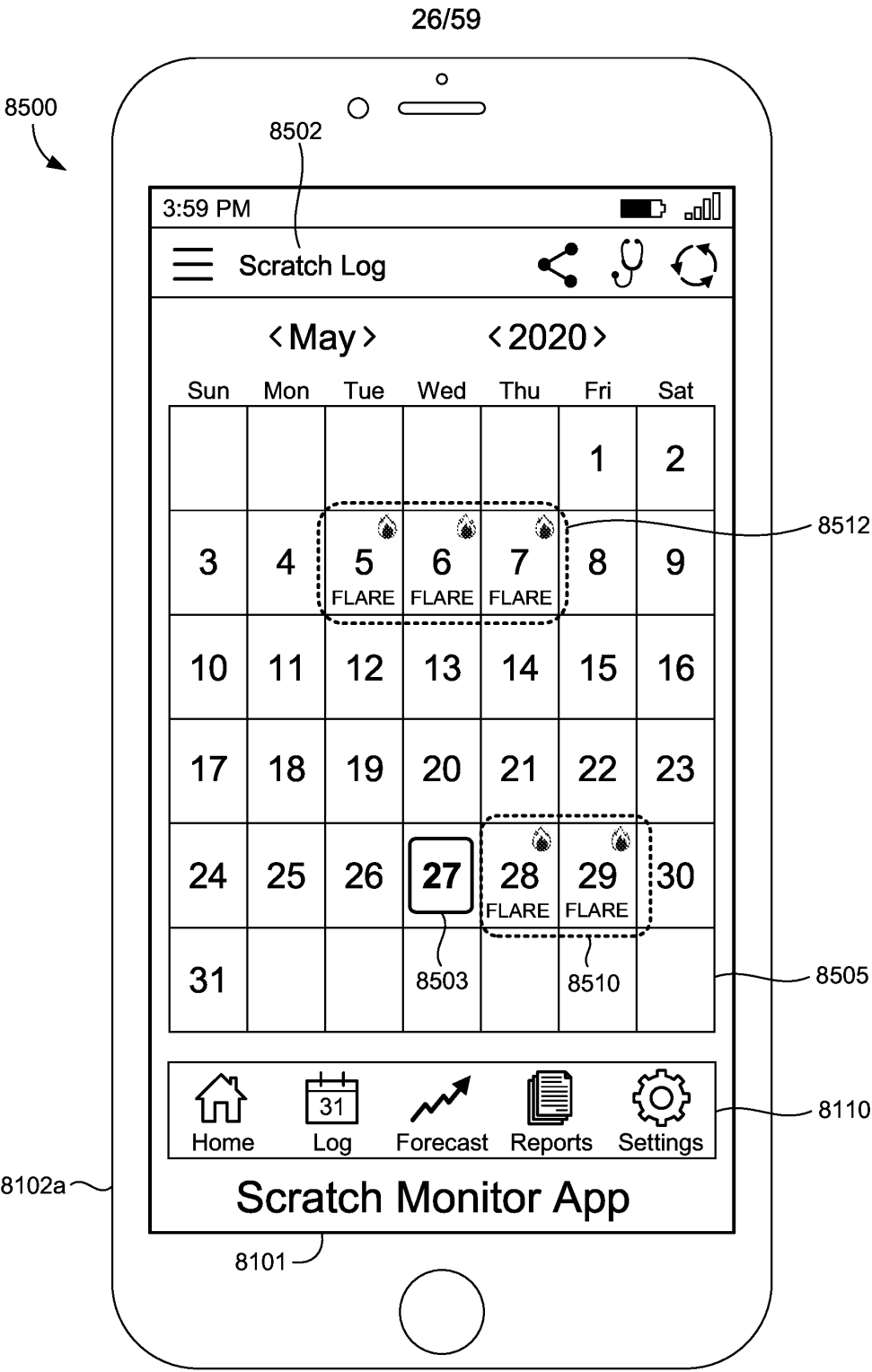


FIG. 8E

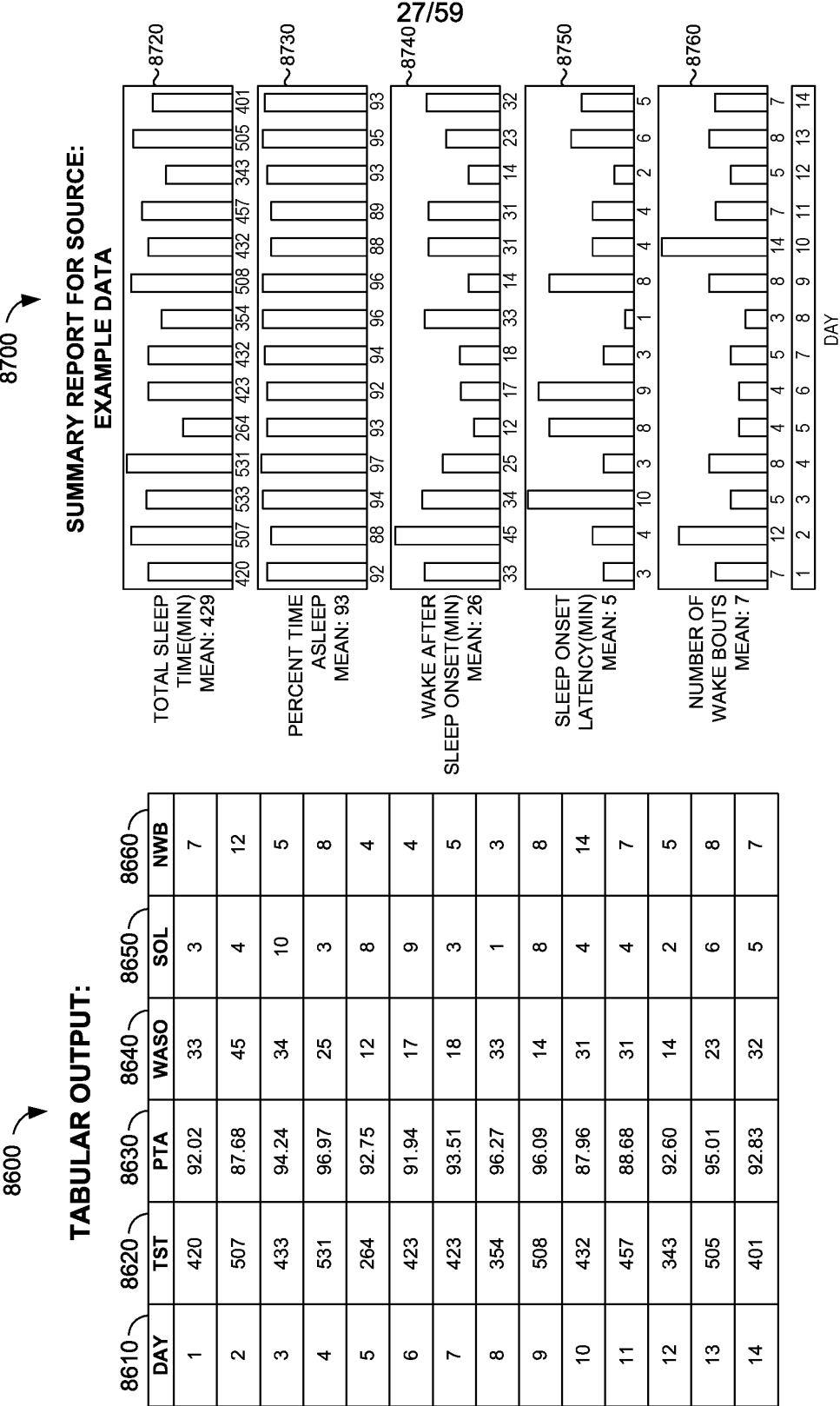


FIG. 8F

EXAMPLE DISPLAYED ANALYTICS FOR SLEEP

28/59

```
#####
#
#  ScratchSleep Combined (pre-alpha)
#
#####

import matplotlib.pyplot as plt
import matplotlib.dates as mdates
from matplotlib.dates import DateFormatter
import matplotlib.backends.backend_pdf
from sleepy import SleepPy
from scratchpy import ScratchPy
from ScratchSleepSolution.utils import build_logger
import pandas as pd
import numpy as np

class ScratchSleep(SleepPy):
    """
    This package pulls the SleepPy and ScratchPy packages into a single pipeline for the analysis of sleep and
    scratch behaviors using triaxial accelerometer data. The pipeline is built to accept GeneActiv .bin or .csv files
    and process it to produce clinical end points for sleep and scratch on a 24 hour basis(measured from noon to noon).

    """

    def __init__(
        self,
        input_file,
        results_directory,
        sampling_frequency=None,
        plot=False,
        verbose=False,
        export_scratch_data=False,
        logger=None,
    ):
        """
        Class initialization.

        Parameters
        -----
        input_file : string
            Full path to the input file (geneactiv .bin or .csv).
        results_directory : string
            Full path to the output directory.
        sampling_frequency : float
            Sampling rate of the data in Hz.
        plot : boolean
            Enables plotting.
        verbose : boolean
            Enables progress statements.
        export_scratch_data : boolean
            Exports scratch-specific predictions, plots, and endpoints.
        logger : logger object
            Use custom logger.
        """

```

CONTINUES IN FIG. 9B

*FIG. 9A*

29/59

CONTINUES FROM FIG. 9A

```

"""
super().__init__(input_file, results_directory, sampling_frequency)
self.allowed_rest_break = 150
self.minimum_rest_threshold = 0.1
self.min_t = 25.0
self.minimum_hours = 12
self.fs = sampling_frequency
self.scratch_endpoints = []
self.export_scratch_data = export_scratch_data
self.plot = plot
self.verbose = verbose
self.movement_based_nonwear = 0.0010
# Set up logging
if "_" not in self.src_name:
    raise ValueError("Input filename format incorrect. Quitting.")
if len(self.src_name.split("_")) < 8:
    raise ValueError("Input filename format incorrect. Quitting.")
self.study = self.src_name.split("_")[0]
self.subject = self.src_name.split("_")[5]
self.device_location = self.src_name.split("_")[6]
self.visit = self.src_name.split("_")[7]
log_filename = (
    results_directory
    + "/analytics_pipeline_"
    + self.study
    + "_"
    + self.subject
    + "_"
    + self.device_location
    + "_"
    + self.visit
    + ".log"
)
if logger is None:
    self.logger = build_logger(log_name=__name__, log_path=log_filename,)
else:
    self.logger = logger
return

def run(self):
    """
    Runs the full scratch and sleep analysis end to end on the input data

    """
    if self.verbose:
        self.logger.info(
            "[ANALYTICS_PIPELINE] [INFO] Study: [{}] | Subject: [{}] | "
            "Location: [{}] | Visit: [{}] | Running Scratch and Sleep Analytics "
            "Pipeline on single wrist.".format(
                self.study, self.subject, self.device_location, self.visit
            )
        )
    )

```

CONTINUES IN FIG. 9C

*FIG. 9B*

30/59

CONTINUES FROM FIG. 9B

```

try:
    self.run_sleep()
    if self.verbose:
        self.logger.info(
            "[ANALYTICS_PIPELINE] [SUCCESS] Study: {} | Subject: {} | "
            "Location: {} | Visit: {} | Ran SleepPy.".format(
                self.study, self.subject, self.device_location, self.visit
            )
        )
except Exception as e:
    if self.verbose:
        self.logger.info(
            "[ANALYTICS_PIPELINE] [FAILURE] Study: {} | Subject: {} | "
            "Location: {} | Visit: {} | SleepPy run fail.".format(
                self.study, self.subject, self.device_location, self.visit
            ),
            exc_info=True,
        )

try:
    self.run_scratch()
    if self.verbose:
        self.logger.info(
            "[ANALYTICS_PIPELINE] [SUCCESS] Study: {} | Subject: {} | "
            "Location: {} | Visit: {} | Ran ScratchPy.".format(
                self.study, self.subject, self.device_location, self.visit
            )
        )
except Exception as e:
    if self.verbose:
        self.logger.info(
            "[ANALYTICS_PIPELINE] [FAILURE] Study: {} | Subject: {} | "
            "Location: {} | Visit: {} | ScratchPy run fail.".format(
                self.study, self.subject, self.device_location, self.visit
            ),
            exc_info=True,
        )

try:
    self.export_scratch_sleep()
    if self.verbose:
        self.logger.info(
            "[ANALYTICS_PIPELINE] [SUCCESS] Study: {} | Subject: {} | "
            "Location: {} | Visit: {} | Exported scratch/sleep results.".format(
                self.study, self.subject, self.device_location, self.visit
            )
        )
except Exception as e:
    if self.verbose:
        self.logger.info(
            "[ANALYTICS_PIPELINE] [FAILURE] Study: {} | Subject: {} | "
            "Location: {} | Visit: {} | Export scratch/sleep results fail".format(
                self.study, self.subject, self.device_location, self.visit
            ),
            exc_info=True,
        )
return

```

*FIG. 9C*

CONTINUES IN FIG. 9D

31/59

CONTINUES FROM FIG. 9C

```

def run_sleep(self):
    """
    Runs the sleep analysis portion of the package on the input data.

    """
    if self.verbose:
        self.logger.info(
            "[SLEEPY] [INFO] Study: {} | Subject: {} | "
            "Location: {} | Visit: {} | Running SleepPy analysis.".format(
                self.study, self.subject, self.device_location, self.visit
            )
        )

    self.load()
    if self.verbose:
        self.logger.info(
            "[SLEEPY] [INFO] Study: {} | Subject: {} | "
            "Location: {} | Visit: {} | Loaded data.".format(
                self.study, self.subject, self.device_location, self.visit
            )
        )

    self.calculate_major_rest_periods()
    if self.verbose:
        self.logger.info(
            "[SLEEPY] [INFO] Study: {} | Subject: {} | "
            "Location: {} | Visit: {} | Calculated total sleep opportunity for all nights.".format(
                self.study, self.subject, self.device_location, self.visit
            )
        )

    self.calculate_sleep_predictions()
    if self.verbose:
        self.logger.info(
            "[SLEEPY] [INFO] Study: {} | Subject: {} | "
            "Location: {} | Visit: {} | Calculated sleep/wake predictions for all nights.".format(
                self.study, self.subject, self.device_location, self.visit
            )
        )

    self.calculate_endpoints()
    if self.verbose:
        self.logger.info(
            "[SLEEPY] [INFO] Study: {} | Subject: {} | "
            "Location: {} | Visit: {} | Calculated sleep endpoints for all nights.".format(
                self.study, self.subject, self.device_location, self.visit
            )
        )

```

CONTINUES IN FIG. 9E

*FIG. 9D*

32/59

CONTINUES FROM FIG. 9D

```

        if self.plot:
            self.visualize()
        if self.verbose:
            vis_fp_days = "{}/{}_visual_report.pdf".format(self.src_name)
            vis_fp_summary = "{}/{}_summary_report.pdf".format(self.src_name)
            self.logger.info(
                "[SLEEPY] [INFO] Study: {} | Subject: {} | "
                "Location: {} | Visit: {} | Generated sleep analysis reports.".format(
                    self.study, self.subject, self.device_location, self.visit
                )
            )
            self.logger.info(
                "[SLEEPY] [INFO] Study: {} | Subject: {} | "
                "Location: {} | Visit: {} | Saved: {}".format(
                    self.study,
                    self.subject,
                    self.device_location,
                    self.visit,
                    vis_fp_days,
                )
            )
            self.logger.info(
                "[SLEEPY] [INFO] Study: {} | Subject: {} | "
                "Location: {} | Visit: {} | Saved: {}".format(
                    self.study,
                    self.subject,
                    self.device_location,
                    self.visit,
                    vis_fp_summary,
                )
            )
        return

def run_scratch(self):
    """
    Runs the scratch analysis portion of the package on the input data.
    """
    if self.verbose:
        self.logger.info(
            "[ANALYTICS_PIPELINE] [INFO] Study: {} | Subject: {} | "
            "Location: {} | Visit: {} | Running ScratchPy analysis.".format(
                self.study, self.subject, self.device_location, self.visit
            )
        )
    if self.plot:
        pdf = matplotlib.backends.backend_pdf.PdfPages(
            self.dst + "{}/{}_scratch_report.pdf".format(self.src_name)
        )

```

CONTINUES IN FIG. 9F

*FIG. 9E*

33/59

CONTINUES FROM FIG. 9E

```

for day in range(len(self.raw_days)):
    mrp = self.major_rest_periods[day]
    # If MRP exists, run slice data and run scratch
    if mrp:
        raw_day = self.raw_days[day][mrp[0] : mrp[1]]
        if self.verbose:
            self.logger.info(
                "[SCRATCHPY] [INFO] Study: [{}] | Subject: [{}] | "
                "Location: [{}] | Visit: [{}] | Sliced data into total sleep opportunity for day: {}".format(
                    self.study,
                    self.subject,
                    self.device_location,
                    self.visit,
                    day + 1,
                )
            )
        # No MRP, return np.nan for scratch endpoints
    else:
        if self.verbose:
            self.logger.info(
                "[SCRATCHPY] [ISSUE] Study: [{}] | Subject: [{}] | "
                "Location: [{}] | Visit: [{}] | No TSO. No scratch predictions. "
                "for day: {}".format(
                    self.study,
                    self.subject,
                    self.device_location,
                    self.visit,
                    day + 1,
                )
            )
        self.scratch_endpoints.append(
            [
                ("total_scratch_events", np.nan),
                ("total_scratch_duration_seconds", np.nan),
                ("total_no_scratch_events", np.nan),
                ("total_no_scratch_duration_seconds", np.nan),
            ]
        )
    continue

raw_day_df = raw_day.copy()
raw_day_df["ts"] = raw_day.index
raw_day_df.reset_index(inplace=True, drop=True)
raw_day_df = raw_day_df[["ts", "X", "Y", "Z"]]
raw_day_df.columns = ["ts", "x", "y", "z"]

```

CONTINUES IN FIG. 9G

*FIG. 9F*

34/59

CONTINUES FROM FIG. 9F

```

scratch = ScratchPy(
    raw_data_df=raw_day_df,
    results_directory=self.dst,
    sampling_frequency=20.0,
)
scratch.run()

if self.verbose:
    self.logger.info(
        "[SCRATCHPY] [INFO] Study: {} | Subject: {} | "
        "Location: {} | Visit: {} | Calculated scratch predictions for day {}".format(
            self.study,
            self.subject,
            self.device_location,
            self.visit,
            day + 1,
        )
    )

if self.plot:
    scratch.visualize(prepare_only=True)
    self.visualize_scratch(data_to_plot=scratch.data_to_plot, pdf=pdf)

if self.verbose:
    self.logger.info(
        "[SCRATCHPY] [INFO] Study: {} | Subject: {} | "
        "Location: {} | Visit: {} | Appended scratch report for day {} to final report.".format(
            self.study,
            self.subject,
            self.device_location,
            self.visit,
            day + 1,
        )
    )

if self.export_scratch_data:
    scratch.export(
        output_file_prefix=self.src_name + "_day_{}".format(day + 1)
    )
    if self.verbose:
        self.logger.info(
            "[SCRATCHPY] [INFO] Study: {} | Subject: {} | "
            "Location: {} | Visit: {} | Exported scratch predictions and endpoints for day {}".format(
                self.study,
                self.subject,
                self.device_location,
                self.visit,
                day + 1,
            )
        )

```

CONTINUES IN FIG. 9H

*FIG. 9G*

35/59

CONTINUES FROM FIG. 9G

```

        self.scratch_endpoints.append(
            sorted(scratch.scratch_endpoints.items(), reverse=True)
        )
    if self.plot:
        pdf.close()
        plt.close("all")

    return

def export_scratch_sleep(self):
    """
    Exports the combined scratch and sleep endpoints to .csv.
    """
    header = (
        ["date"] + self.endpoints[0] + [i[0] for i in self.scratch_endpoints[0]]
    )
    days = list(range(0, len(self.raw_days_to_plot)))
    export_df = []
    for day in days:
        idx_start = self.raw_days_to_plot[day].index[0]
        idx_start = (
            idx_start.replace(hour=12, minute=0, second=0, microsecond=0)
            - pd.to_timedelta("24h")
        )
        if idx_start
            < idx_start.replace(hour=12, minute=0, second=0, microsecond=0)
        else idx_start.replace(hour=12, minute=0, second=0, microsecond=0)
    )
    if self.major_rest_periods[day]:
        export_df.append(
            [str(idx_start.date())]
            + self.endpoints[day + 1]
            + [i[1] for i in self.scratch_endpoints[day]]
        )
    else:
        export_df.append(
            [str(idx_start.date())]
            + self.endpoints[day + 1]
            + [np.nan for i in self.scratch_endpoints[day]]
        )
    export_df = pd.DataFrame(export_df)
    export_df.columns = header
    export_fp = self.dst + "{}_sw_output_table.csv".format(self.src_name)
    export_df.to_csv(export_fp, index=False)

    if self.verbose:
        self.logger.info(
            "[ANALYTICS_PIPELINE] [INFO] Study: {} | Subject: {} | "
            "Location: {} | Visit: {} | Saved scratch and sleep endpoints: {}".format(
                self.study,
                self.subject,
                self.device_location,
                self.visit,
                export_fp,
            )
        )
    return

```

*FIG. 9H*

CONTINUES IN FIG. 9I

36/59

CONTINUES FROM FIG. 9H

```

@staticmethod
def visualize_scratch(data_to_plot, pdf):
    """
    Generate a plot of a given instance of scratch algorithm results.
    """

    # Dates
    start_date = data_to_plot.index[0].strftime("%B %d, %Y, %r")
    stop_date = data_to_plot.index[-1].strftime("%B %d, %Y, %r")

    # PLOT!
    fig, axs = plt.subplots(nrows=3, ncols=1, figsize=(12, 8))
    plt.suptitle("Scratch Report (" + start_date + " - " + stop_date + ")")

    # Raw Data
    axs[0].plot(data_to_plot.index.values, data_to_plot.x, color="r", label="X")
    axs[0].plot(data_to_plot.index.values, data_to_plot.y, color="g", label="Y")
    axs[0].plot(data_to_plot.index.values, data_to_plot.z, color="b", label="Z")
    axs[0].axis.set_major_locator(mdates.HourLocator(interval=1))
    axs[0].axis.set_major_formatter(DateFormatter("%H:%M"))
    axs[0].legend(loc="center left", bbox_to_anchor=(1, 0.5))
    axs[0].set_ylabel("Raw Data (G)")

    # Hand Movement Predictions
    axs[1].plot(
        data_to_plot.index.values,
        data_to_plot.hand_movement_predictions,
        color="k",
        linewidth=0.1,
    )
    axs[1].axis.set_major_locator(mdates.HourLocator(interval=1))
    axs[1].axis.set_major_formatter(DateFormatter("%H:%M"))
    axs[1].set_ylabel("Hand Movement\nPredictions")
    axs[1].set_ylim([0.5, 1.5])
    axs[1].set_yticks([], [])

    # Scratch Predictions
    axs[2].plot(
        data_to_plot.index.values,
        data_to_plot.scratch_predictions,
        color="k",
        linewidth=0.1,
    )
    axs[2].axis.set_major_locator(mdates.HourLocator(interval=1))
    axs[2].axis.set_major_formatter(DateFormatter("%H:%M"))
    axs[2].set_ylabel("Scratch\nPredictions")
    axs[2].set_ylim([0.5, 1.5])
    axs[2].set_yticks([], [])
    axs[2].set_xlabel("Time (HH:MM)")
    pdf.savefig(fig.number)

```

*FIG. 9I*

37/59

```
#####
#
# Scratch Detection (pre-alpha)
#
#####

import numpy as np
import pandas as pd
import datetime
import matplotlib.pyplot as plt
from matplotlib.dates import DateFormatter
import matplotlib.dates as mdates
import pkg_resources as pkr
from joblib import load
from motion_detection_algorithm import detect_motion_from_acceleration
from scratchpy import helper_functions

class ScratchPy:
    def __init__(
        self,
        raw_data_df,
        results_directory,
        sampling_frequency=20.0,
        verbose=False,
    ):
        """
        Scratch algorithm object. This object houses the procedure for computing scratch predictions from a wrist-worn
        accelerometer device.

        Parameters
        -----
        raw_data_df : Pandas DataFrame
            DataFrame of raw accelerometer data. Measured in G's. Columns = [ts {datetime}, x {float}, y {float}, z {float}]
        results_directory : str
            File path to output directory
        verbose : bool
            Bool to allow for printing of results
        """
        .
        .

```

CONTINUES IN FIG. 10B

*FIG. 10A*

38/59

CONTINUES FROM FIG. 10A

```

# Raw data to process
self.raw_data = raw_data_df
# Results directory to save final results
self.results_directory = results_directory
# Window size to process data (non-overlapping windows)
self.window_size = 3.0
# Properties of hand movement motion detection algorithm
self.motion_detection_filter_cutoff = 6
self.motion_detection_threshold = 0.022558169349678834
# Sampling frequency of raw data
self.fs = sampling_frequency
# Total number of accelerometer samples in a given window
self.no_samples_per_window = int(self.fs * self.window_size)
# Load serialized scratch model
model_file_path = pkr.resource_filename(
    "scratchpy.model", "model_trained_all_subjects.joblib"
)
self.scratch_model = load(model_file_path)
# Selected features for Scratch model
self.selected_features = [
    "PC2_iqr",
    "PC2_spectral_entropy",
    "PC1_iqr",
    "PC2_spectral_flatness",
    "PC1_dom_freq_magnitude",
    "PC2_sparc",
    "PC2_mean_cross_rate",
    "PC2_jerk_ratio",
    "PC2_signal_entropy",
    "PC2_dom_freq_magnitude",
    "mag_jerk_ratio",
    "PC1_spectral_flatness",
    "mag_sparc",
    "PC1_skewness",
    "PC1_permutation_entropy",
    "mag_rms",
    "PC1_signal_entropy",
    "PC2_permutation_entropy",
    "mag_signal_entropy",
    "PC1_ldl_jerk",
    "PC2_skewness",
    "mag_range",
    "PC1_sparc",
    "mag_permutation_entropy",
    "PC1_dom_freq_value",
    "mag_iqr",
]

```

CONTINUES IN FIG. 10C

*FIG. 10B*

39/59

CONTINUES FROM FIG. 10B

```

# Hand movement predictions
self.hand_movement_predictions = None
# Include print statements outlining run of package
self.verbose = verbose
# Total recording duration
self.recording_duration_minutes = None
# Epoch Results
self.epoch_timestamps = []
self.epoch_scratch_predictions = []
# Endpoint Results
self.scratch_endpoints = None
# Output filename prefix
self.output_file_prefix = ""
# Data for plotting
self.data_to_plot = None

def quality_check_input_data(self):
    """
    Run a series of quality checks on input data.
    """
    # ----- #
    # [CHECK] Sampling rate is 20 Hz
    # ----- #
    if self.fs != 20.0:
        print(
            "[QC ERROR] Sampling frequency not equal to 20 hz. Required 20hz input sampling rate."
        )
        return False

    # ----- #
    # [CHECK] Raw data columns = ["ts", "x", "y", "z"]
    # ----- #
    if self.raw_data.columns.tolist() != ["ts", "x", "y", "z"]:
        print(
            "[QC ERROR] Incorrect raw data column format. "
            "Columns of raw data must be formatted as: ['ts', 'x', 'y', 'z']"
        )
        return False

    # ----- #
    # [CHECK] Raw data timestamp is datetime
    # ----- #
    if not isinstance(self.raw_data.ts.values[0], np.datetime64):
        print("[QC ERROR] Timestamps of raw data are not np.datetime64.")
        return False

```

CONTINUES IN FIG. 10D

*FIG. 10C*

40/59

CONTINUES FROM FIG. 10C :

```

# ----- #
# [CHECK] duration of recording >= 3 seconds
# ----- #
start_ts = self.raw_data.ts.values[0]
stop_ts = self.raw_data.ts.values[-1]
total_dur = (pd.to_datetime(stop_ts) - pd.to_datetime(start_ts)).total_seconds()
if total_dur < 3.0:
    # Minimum of 3 seconds of data required to run algorithm
    print("[QC ERROR] Not enough data. Minimum recording duration is 3 seconds.")
    return False
else:
    self.recording_duration_minutes = total_dur / 60.0

return True

def run(self):
    """
    Run the scratch algorithm module.
    """
    # ----- #
    # Run quality checks on input data
    # ----- #
    if self.verbose:
        print("Running quality checks...")
    qc_result = self.quality_check_input_data()
    if not qc_result:
        return

    # ----- #
    # 1. Run motion detection algorithm (hand movement)
    # ----- #
    if self.verbose:
        print("Running hand movement detector...")
    hand_movement_predictions, _ = detect_motion_from_acceleration(
        sensor_data=self.raw_data,
        fs=self.fs,
        filter_cutoff=self.motion_detection_filter_cutoff,
        cov_threshold=self.motion_detection_threshold,
    )
    # Convert to DataFrame
    hand_movement_predictions_df = pd.DataFrame()
    hand_movement_predictions_df["hand_movement_predictions"] = pd.Series(
        hand_movement_predictions
    )
    self.hand_movement_predictions = hand_movement_predictions_df

    # Remove first 1 second of raw data (due to rolling CoV calculation)
    self.raw_data = self.raw_data.loc[self.fs:]
    self.raw_data.reset_index(drop=True, inplace=True)

```

: CONTINUES IN FIG. 10E

*FIG. 10D*

41/59

CONTINUES FROM FIG. 10D

```

# ----- #
# 2. Window raw data and hand movement predictions into 3 second windows
# ----- #
if self.verbose:
    print("Windowing data...")
windowed_data = helper_functions.non_overlapping_windows(
    df=self.raw_data, window_size=self.no_samples_per_window
)
windowed_hand_movement = helper_functions.non_overlapping_windows(
    df=hand_movement_predictions_df, window_size=self.no_samples_per_window
)
total_windows = windowed_hand_movement.shape[0]

# ----- #
# 3. Iterate over each window and compute scratch prediction
# ----- #
if self.verbose:
    print("Running model on each window...")
epoch_timestamps = []
epoch_scratch_predictions = []
for window in range(0, total_windows):
    # Get hand movement predictions for window
    hand_movement_window = windowed_hand_movement[window]
    hand_movement_window = hand_movement_window.reshape(
        self.no_samples_per_window,
    )

    raw_data_window = pd.DataFrame(
        windowed_data[window], columns=["ts", "x", "y", "z"]
    )
    # Assemble timestamps for window
    epoch_timestamps.extend(
        [
            raw_data_window.ts[0],
            raw_data_window.ts[0] + datetime.timedelta(seconds=1),
            raw_data_window.ts[0] + datetime.timedelta(seconds=2),
        ]
    )

```

CONTINUES IN FIG. 10F

*FIG. 10E*

42/59

CONTINUES FROM FIG. 10E

```

# ----- #
# 4. Check if window is 100% hand movement
# ----- #
if (np.unique(hand_movement_window) == [1])[0]:
    # Window is 100% hand movement
    # Signal preprocessing
    preprocessed_data = helper_functions.signal_preprocess_data(
        sliced_data=raw_data_window, fs=self.fs
    )

    # Signal Features
    ftc = helper_functions.compute_features(
        preprocessed_data=preprocessed_data, fs=self.fs
    )
    # Slice features for selected features
    ftc = ftc[self.selected_features]

    # Run through model
    prediction = list(self.scratch_model.predict(ftc.values))
    epoch_scratch_predictions.extend(prediction * int(self.window_size))

else:
    # Window is not 100% hand movement => No Scratch
    epoch_scratch_predictions.extend([0] * int(self.window_size))

# Set Epoch Results
self.epoch_timestamps = epoch_timestamps
self.epoch_scratch_predictions = epoch_scratch_predictions

# Calculate and set endpoint results
scratch_endpoints = helper_functions.calculate_scratch_endpoints(
    epoch_scratch_predictions,
)

self.scratch_endpoints = scratch_endpoints

def visualize(self, output_file_prefix="", prep_only=False):
    """
    Generate a plot of a given instance of scratch algorithm results.
    Plot will be extracted to user defined output directory.

    Parameters
    -----
    output_file_prefix : str
        Identifier to add as prefix to save filename.
    prep_only : bool
        Flag for preparing plot data only, deactivates plotting or saving.
    """

```

CONTINUES IN FIG. 10G

*FIG. 10F*

43/59

CONTINUES FROM FIG. 10F .

```

# ----- #
# [CHECK] Necessary data is available
# ----- #

if not len(self.epoch_scratch_predictions):
    print("[ERROR] Cannot export, no scratch predictions...")
    return

if output_file_prefix:
    self.output_file_prefix = output_file_prefix

plot_save_filepath = (
    self.results_directory
    + "/"
    + self.output_file_prefix
    + "_scratch_report.png"
)

# Plot Results
if self.verbose and not prep_only:
    print("Generating plot...")
elif self.verbose and prep_only:
    print("Generating plot data...")
# Assemble predictions in 3 second windows instead of 1
predictions_plot = self.epoch_scratch_predictions[::3]
# Assemble predictions based on number of samples in window (1 pred per sample)
preds_by_samples = []
for i in predictions_plot:
    preds_by_samples.extend([i] * self.no_samples_per_window)

# Slice raw data to number of samples that give full windows
raw_data_to_plot = self.raw_data.iloc[: len(preds_by_samples)]

# Slice hand movement predictions to number of samples that give full windows
hm_preds = self.hand_movement_predictions.iloc[: len(preds_by_samples)]

# Final DataFrame housing raw data, hand movement predictions, scratch predictions to plot
data_to_plot = pd.concat([raw_data_to_plot, hm_preds], axis=1)
data_to_plot["scratch_predictions"] = pd.Series(preds_by_samples)
data_to_plot["ts"] = pd.to_datetime(data_to_plot["ts"])
data_to_plot.set_index(keys="ts", drop=True, inplace=True)
self.data_to_plot = data_to_plot

# Prepare dataframe only
if prep_only:
    if self.verbose:
        print("Plot data generated.")
    return

# Dates
start_date = data_to_plot.index[0].strftime("%Y-%m-%d")
stop_date = data_to_plot.index[-1].strftime("%Y-%m-%d")

# PLOT!
fig, axs = plt.subplots(nrows=3, ncols=1, figsize=(12, 8))
plt.suptitle("Scratch Report (" + start_date + " - " + stop_date + ")")

```

*FIG. 10G*

CONTINUES IN FIG. 10H

44/59

CONTINUES FROM FIG. 10G

```

# Raw Data
axs[0].plot(data_to_plot.index.values, data_to_plot.x, color="r", label="X")
axs[0].plot(data_to_plot.index.values, data_to_plot.y, color="g", label="Y")
axs[0].plot(data_to_plot.index.values, data_to_plot.z, color="b", label="Z")
axs[0].xaxis.set_major_locator(mdates.HourLocator(interval=1))
axs[0].xaxis.set_major_formatter(DateFormatter("%H:%M"))
axs[0].legend(loc="center left", bbox_to_anchor=(1, 0.5))
axs[0].set_ylabel("Raw Data (G)")

# Hand Movement Predictions
axs[1].plot(
    data_to_plot.index.values,
    data_to_plot.hand_movement_predictions,
    color="k",
    linewidth=0.1,
)
axs[1].xaxis.set_major_locator(mdates.HourLocator(interval=1))
axs[1].xaxis.set_major_formatter(DateFormatter("%H:%M"))
axs[1].set_ylabel("Hand Movement\nPredictions")
axs[1].set_ylim([0.5, 1.5])
axs[1].set_yticks([], [])

# Scratch Predictions
axs[2].plot(
    data_to_plot.index.values,
    data_to_plot.scratch_predictions,
    color="k",
    linewidth=0.1,
)
axs[2].xaxis.set_major_locator(mdates.HourLocator(interval=1))
axs[2].xaxis.set_major_formatter(DateFormatter("%H:%M"))
axs[2].set_ylabel("Scratch\nPredictions")
axs[2].set_ylim([0.5, 1.5])
axs[2].set_yticks([], [])
axs[2].set_xlabel("Time (HH:MM)")

plt.savefig(plot_save_filepath)
if self.verbose:
    print("\tSaved: " + plot_save_filepath)

def export(self, output_file_prefix=""):
    """
    Export the results of the scratch algorithm module. Endpoints and epoch (1 second) predictions will
    be extracted to user defined output directory.

    Parameters
    -----
    output_file_prefix : str
        Identifier to add as prefix to save filename.
    """

```

CONTINUES IN FIG. 10I

*FIG. 10H*

45/59

CONTINUES FROM FIG. 10H

```

# ----- #
# [CHECK] Necessary data is available
# ----- #

if not len(self.epoch_scratch_predictions):
    print("[ERROR] Cannot export, no scratch predictions...")
    return

if output_file_prefix:
    self.output_file_prefix = output_file_prefix

if self.verbose:
    print("Exporting results...")

epoch_predictions_save_filepath = (
    self.results_directory
    + "/"
    + self.output_file_prefix
    + "_scratch_epoch_predictions.csv"
)
endpoint_predictions_save_filepath = (
    self.results_directory
    + "/"
    + self.output_file_prefix
    + "_scratch_endpoint_predictions.csv"
)

# Epoch Predictions
epoch_predictions = pd.DataFrame()
epoch_predictions["ts"] = pd.Series(self.epoch_timestamps)
epoch_predictions["predictions"] = pd.Series(self.epoch_scratch_predictions)
epoch_predictions.to_csv(epoch_predictions_save_filepath, index=False)
if self.verbose:
    print("\tSaved: ", epoch_predictions_save_filepath)

# Endpoint Predictions
save_endpoints = {}
for i in self.scratch_endpoints:
    save_endpoints[i] = [self.scratch_endpoints[i]]
endpoint_predictions = pd.DataFrame.from_dict(save_endpoints)
endpoint_predictions.to_csv(endpoint_predictions_save_filepath, index=False)
if self.verbose:
    print(
        "\tSaved: ", endpoint_predictions_save_filepath,
    )

```

*FIG. 10I*

46/59

```
#####
#
# Sleep Detection & Determine TSO (pre-alpha)
#
#####

import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
import matplotlib.backends.backend_pdf
from sleepypy.majorrest import MajorRestPeriod
from sleepypy.colekripke import ColeKripke
from inertial_sensor_routines.gnactv import tabular_conversion
from sleepypy.utils import (
    downsample,
    resample_by_interpolation,
    bin2df,
    high_pass_filter,
    activity_index,
    load_geneactiv_csv,
    non_overlapping_windows,
    total_sleep_time,
    percent_time_asleep,
    number_of_wake_bouts,
    wake_after_sleep_onset,
    sleep_onset_latency,
    build_logger,
)

sns.set()

class SleepPy(object):

    def __init__(
        self,
        input_file,
        results_directory,
        sampling_frequency=None,
        aws_object=None,
        start_buffer="0s",
        stop_buffer="0s",
        start_time="",
        stop_time="",
        temperature_threshold=25.0,
        minimum_rest_block=30,
        allowed_rest_break=60,
        minimum_rest_threshold=0.1,
        maximum_rest_threshold=1.0,
        minimum_rest_period=None,
        movement_based_nonwear=None,
        minimum_hours=6,
        downsample=True,
        plot=True,
        fast_load=True,
        logger=None,
    ):
        :
```

*FIG. 11A*

CONTINUES IN FIG. 11B

47/59

CONTINUES FROM FIG. 11A

```

"""
Class initialization.

Parameters
-----
input_file : string
    Full path to the file to be processed.
results_directory : string
    Full path to the directory where the results should be saved.
sampling_frequency : float
    Sampling frequency of the input data, required if not using fast load.
aws_object : data object
    Data object to be processed from aws (in place of source file path).
start_buffer : string
    Number of seconds to ignore from the beginning of the recording. format: "0s"
stop_buffer : string
    Number of seconds to ignore at the end of the recording. format: "0s"
start_time : string
    Time stamp from which to start looking at data. format: "%Y-%m-%d %H:%M:%S%f"
stop_time : string
    Time stamp at which to stop looking at data. format: "%Y-%m-%d %H:%M:%S%f"
temperature_threshold : float
    Lowest temperature for which a data point is considered valid.
minimum_rest_block : int
    Number of minutes required to consider a rest period valid.
allowed_rest_break : int
    Number of minutes allowed to interrupt major rest period.
minimum_rest_threshold : float
    Minimum allowed threshold for determining major rest period.
maximum_rest_threshold : float
    Maximum allowed threshold for determining major rest period.
minimum_rest_period : float
    Minimum length allowed for major rest period.
movement_based_nonwear : float
    Threshold for movement based nonwear.
minimum_hours : float
    Minimum number of hours required to consider a day useable.
downsample : boolean
    True triggers downsampling to 20hz.
plot : boolean
    True triggers plotting of data.
fast_load : boolean
    True triggers inertial sensor routines fortran loading of bin data. Locates sampling frequency in data.
logger : logger object
    Replace default logger with logger object if desired.
"""

```

CONTINUES IN FIG. 11C

*FIG. 11B*

48/59

CONTINUES FROM FIG. 11B

```

if aws_object is not None:
    self.src = aws_object
else:
    self.src = input_file
self.extension = input_file.split(".")[-1]
self.dst = results_directory
self.src_name = input_file.split("/")[-1][0:-4]
self.sub_dst = results_directory
self.fs = sampling_frequency
self.window_size = 60
self.high_pass_cutoff = 0.25
self.start_buffer = start_buffer
self.stop_buffer = stop_buffer
self.start_time = start_time
self.stop_time = stop_time
self.min_t = temperature_threshold
self.minimum_rest_block = minimum_rest_block
self.allowed_rest_break = allowed_rest_break
self.minimum_rest_threshold = minimum_rest_threshold
self.maximum_rest_threshold = maximum_rest_threshold
self.minimum_rest_period = minimum_rest_period
self.movement_based_nonwear = movement_based_nonwear
self.minimum_hours = minimum_hours
self.downsample = downsample
self.plot = plot
self.fast_load = fast_load
if logger is None:
    self.logger = build_logger(log_name=__name__)
else:
    self.logger = logger

# DATA
self.raw_days = []
self.raw_days_to_plot = []
self.arm_angle_days = []
self.arm_angle_thresholds = []
self.rest_period_days = []
self.activity_index_days = []
self.sleep_wake_prediction_days = []
self.major_rest_periods = []
self.endpoints = [
    [
        "total_sleep_time",
        "percent_time_asleep",
        "waso",
        "sleep_onset_latency",
        "number_wake_bouts",
        "total_sleep_opportunity",
    ]
]

```

CONTINUES IN FIG. 11D

*FIG. 11C*

49/59

CONTINUES FROM FIG. 11C

```

def run(self):
    """
    Runs the full package end to end on the input data.

    """
    if self.fast_load:
        self.logger.info("Running fast loader. Inferring sampling frequency...")
    else:
        self.logger.info("Loading data...")
    self.load()
    self.logger.info("Detecting major rest periods...")
    self.calculate_major_rest_periods()
    self.logger.info("Making sleep predictions...")
    self.calculate_sleep_predictions()
    self.logger.info("Calculating sleep endpoints...")
    self.calculate_endpoints()
    if self.plot:
        self.logger.info("Building plots...")
        self.visualize()
    self.logger.info("Exporting results...")
    self.export()

    return

def load(self):
    """
    Loads full data into 24 hour days.

    """
    # FULL DATA
    data = (
        load_geneactiv_csv(self.src)
        if self.extension == "csv"
        else (
            bin2df(self.src)
            if not self.fast_load
            else tabular_conversion(
                self.src, acc_units="g", save_file=None, return_sampling_rate=True,
            )
        )
    )

    if self.fast_load:
        df = data[0]
        self.fs = data[1]
        self.logger.info(f"Detected frequency: {data[1]}")
    else:
        df = data

    # CHECK DATA LENGTH
    if ((len(df) / float(self.fs)) / 3600.0) <= self.minimum_hours:
        exception_string = f"Total length of data did not meet your minimum time requirement of {self.minimum_hours}
hours"
        raise Exception(exception_string)

```

*FIG. 11D*

CONTINUES IN FIG. 11E

50/59

CONTINUES FROM FIG. 11D

```

# DOWNSAMPLE
if self.downsample:
    self.logger.info(f"Downsampling data from {self.fs} to 20hz.")
    if self.fs > 20.0:
        if (self.fs / 20.0).is_integer():
            df = downsample(df, int(self.fs) // 20)
        else:
            df = resample_by_interpolation(df, int(self.fs), 20)
    self.fs = 20.0

# APPLY BUFFERS
df = df.loc[
    df.index[0]
    + pd.Timedelta(self.start_buffer) : df.index[-1]
    - pd.Timedelta(self.stop_buffer)
]

# APPLY START AND STOP TIMES
if self.start_time and self.stop_time:
    self.start_time = pd.to_datetime(
        self.start_time, format="%Y-%m-%d %H:%M:%S:%f"
    )
    self.stop_time = pd.to_datetime(
        self.stop_time, format="%Y-%m-%d %H:%M:%S:%f"
    )
    df = df.loc[self.start_time : self.stop_time]
elif self.start_time:
    self.start_time = pd.to_datetime(
        self.start_time, format="%Y-%m-%d %H:%M:%S:%f"
    )
    df = df.loc[self.start_time :]
elif self.stop_time:
    self.stop_time = pd.to_datetime(
        self.stop_time, format="%Y-%m-%d %H:%M:%S:%f"
    )
    df = df.loc[: self.stop_time]

# SPLIT BY DAY
self.logger.info("Splitting raw data into 24 hour periods...")
self.raw_days = [
    day[1]
    for day in df.groupby(pd.Grouper(level=0, freq="24h", base=12))
    if ((len(day[1]) / float(self.fs)) / 3600.0) >= self.minimum_hours
]

# MAKE SURE WE ARE LEFT WITH USEFUL DAYS
if not self.raw_days:
    exception_string = f"No processable days found. Please examine your input data. Ensure there are enough hours
based on your minimum criteria of {self.minimum_hours} hours."
    raise Exception(exception_string)

# DAYS FOR PLOTTING
self.raw_days_to_plot = [day.resample("60s").median() for day in self.raw_days]
return

```

*FIG. 11E*

CONTINUES IN FIG. 11F

51/59

CONTINUES FROM FIG. 11E

```

def calculate_major_rest_periods(self):
    """
    Gets the major rest periods for each 24 hour day.
    """
    for day in self.raw_days:
        mrp = MajorRestPeriod(
            day.copy(),
            self.fs,
            minimum_rest_block=self.minimum_rest_block,
            minimum_rest_threshold=self.minimum_rest_threshold,
            allowed_rest_break=self.allowed_rest_break,
            temperature_threshold=self.min_t,
            maximum_rest_threshold=self.maximum_rest_threshold,
            minimum_rest_period=self.minimum_rest_period,
            movement_based_nonwear=self.movement_based_nonwear,
        )
        mrp.run()
        self.arm_angle_days.append(mrp.df_angle)
        self.arm_angle_thresholds.append(mrp.arm_angle_thresh)
        self.rest_period_days.append(mrp.rest_periods)
        self.major_rest_periods.append(mrp.mrp)
    return

def calculate_sleep_predictions(self):
    """
    Gets the sleep/wake predictions for each 24 hour day.
    """
    for day in self.raw_days:
        # PREPROCESS, ACTIVITY INDEX
        day = day.copy()[["X", "Y", "Z"]]
        windowed = non_overlapping_windows(day.copy(), int(60 * self.fs))
        filtered = high_pass_filter(
            windowed,
            sampling_rate=self.fs,
            hp_cutoff=self.high_pass_cutoff,
            order=3,
        )
        ai = activity_index(filtered)

        # SLEEP WAKE PREDICTIONS
        ck = ColeKripke(ai.flatten())
        predictions = ck.predict()

        # ADD TIME BACK
        idx = pd.date_range(day.index[0], day.index[-1], freq="60s")[
            0 : len(predictions)
        ]
        predictions = pd.DataFrame(predictions)
        predictions.index = idx
        ai = pd.DataFrame(ai)
        ai.index = idx

        # STORE DATA
        self.activity_index_days.append(ai)
        self.sleep_wake_prediction_days.append(predictions)
    return

```

*FIG. 11F*

CONTINUES IN FIG. 11G

52/59

CONTINUES FROM FIG. 11F

```

def calculate_endpoints(self):
    """
    Calculates the clinical sleep endpoints for each 24 hour day.

    """
    for day in range(len(self.sleep_wake_prediction_days)):
        mrp = self.major_rest_periods[day]
        if mrp:
            predictions = self.sleep_wake_prediction_days[day][
                mrp[0] : mrp[1]
            ].values
            self.endpoints.append(
                [
                    total_sleep_time(predictions),
                    percent_time_asleep(predictions),
                    wake_after_sleep_onset(predictions),
                    sleep_onset_latency(predictions),
                    number_of_wake_bouts(predictions),
                    int((mrp[1] - mrp[0]).total_seconds() / 60.0),
                ]
            )
        else:
            self.endpoints.append([np.nan] * 6)
    return

def visualize(self):
    """
    Builds all visual reports and exports them to pdf.

    """
    # OUTPUT PDF FOR ALL DAYS
    pdf = matplotlib.backends.backend_pdf.PdfPages(
        self.dst + "/" + "{}_visual_report.pdf".format(self.src_name)
    )

    # PROCESS DAYS
    days = list(range(0, len(self.raw_days_to_plot)))
    for day in days:
        # LOAD DATA, GET SHARED AXIS
        raw = self.raw_days_to_plot[day]
        idx_start = raw.index[0]
        idx_start = (
            idx_start.replace(hour=12, minute=0, second=0, microsecond=0)
            - pd.to_timedelta("24h")
        )
        if idx_start
            < idx_start.replace(hour=12, minute=0, second=0, microsecond=0)
            else idx_start.replace(hour=12, minute=0, second=0, microsecond=0)
        )
        idx = pd.date_range(start=idx_start, periods=1440, freq="60s")
        raw = raw.reindex(idx, fill_value=np.nan)

```

CONTINUES IN FIG. 11H

*FIG. 11G*

53/59

: CONTINUES FROM FIG. 11G

```

# ACTIVITY INDEX
act = self.activity_index_days[day]
act = act.resample("60s").max()
act = act.reindex(idx, fill_value=np.nan)

# ARM ANGLE
ang = self.arm_angle_days[day]
ang = ang.resample("60s").max()
ang = ang.reindex(idx, fill_value=np.nan)

# SLEEP WAKE PREDICTIONS
swp = self.sleep_wake_prediction_days[day]
swp[swp == 0] = np.nan
swp = swp.resample("60s").max()
swp = swp.reindex(idx, fill_value=np.nan)

# REST PERIODS
rps = self.rest_period_days[day]
rps[rps == 1] = np.nan
rps[rps == 0] = 1
rps = rps.resample("60s").max()
rps = rps.reindex(idx, fill_value=np.nan)

# DATAFRAME FOR STRAIGHT LINES
df = swp.copy()
df.columns = ["wake predictions"]
df[["wake predictions"]] -= 0.25
df["rest periods"] = rps.values - 0.75

# SET FIRST AND LAST VALUES TO PREVENT NAN PLOT ISSUES
df.loc[:1, "wake predictions"] = 0.75
df.loc[-1:, "wake predictions"] = 0.75
df.loc[:1, "rest periods"] = 0.25
df.loc[-1:, "rest periods"] = 0.25

# ENDPOINTS FOR TABLE
t_labels = (
    "Total Sleep Time(min)",
    "Percent Time Asleep",
    "Wake After Sleep Onset(min)",
    "Sleep Onset Latency(min)",
    "Number of Wake Bouts",
    "Total Sleep Opportunity(min)",
)
t_vals = [self.endpoints[day + 1]]

# PLOTTING
fig, (axt, ax0, ax1, ax2, ax3, ax4, ax5) = plt.subplots(
    7, 1, figsize=(30, 15), sharex=True
)
plt.suptitle(
    "Visual Report for Source: {}\nDay: {}\nDate: {}".format(
        self.src_name, day + 1, idx[0].date()
    ),
    fontsize=25,
)
hours = mdates.HourLocator(interval=1)
h_fmt = mdates.DateFormatter("%H:%M")
all_axes = (ax0, ax1, ax2, ax3, ax4, ax5)

```

*FIG. 11H*

: CONTINUES IN FIG. 11I

54/59

CONTINUES FROM FIG. 11H :

```

# PLOT TABLE
tbl = axt.table(
    cellText=t_vals,
    colLabels=t_labels,
    cellLoc="center",
    rowLoc="center",
    loc="center",
    fontsize=20,
)
tbl.auto_set_font_size(False)
tbl.set_fontsize(24)
tbl.scale(1.15, 2.4)
axt.axis("off")

# RAW DATA
raw.rename(columns={"T": "Temperature", "LUX": "Light"}, inplace=True)
raw[["X", "Y", "Z"]].plot(ax=ax0, lw=1, x_compat=True).legend(
    bbox_to_anchor=(0, 1), fontsize=20
)

# TEMPERATURE
raw[["Temperature"]].plot(
    ax=ax1, lw=1, x_compat=True, color=sns.xkcd_rgb["pale red"]
).legend(bbox_to_anchor=(0, 1), fontsize=20)
idx_first = raw.Temperature.first_valid_index()
idx_last = raw.Temperature.last_valid_index()
ax1.hlines(
    y=self.min_t,
    xmin=idx_first,
    xmax=idx_last,
    color="r",
    linestyle="--",
    lw=2,
)
props = dict(boxstyle="round", facecolor="lavender", alpha=0.35)
textstr = "max: {} \xb0C \nmin: {} \xb0C \nthresh: {} \xb0C".format(
    np.round(raw[["Temperature"]].max().values[0], decimals=2),
    np.round(raw[["Temperature"]].min().values[0], decimals=2),
    self.min_t,
)
ax1.text(
    1.01,
    0.95,
    textstr,
    transform=ax1.transAxes,
    fontsize=14,
    verticalalignment="top",
    bbox=props,
)

# LIGHT
raw[["Light"]].plot(
    ax=ax2, lw=1, x_compat=True, color=sns.xkcd_rgb["pale orange"]
).legend(bbox_to_anchor=(0, 1), fontsize=20)

```

: CONTINUES IN FIG. 11J

*FIG. 11I*

55/59

CONTINUES FROM FIG. 11I :

```

# ACTIVITY INDEX
act.plot(ax=ax3, lw=1, x_compat=True, color="#6fc276").legend(
    labels=["activity"], bbox_to_anchor=(0, 0.75), fontsize=20
)

# ARM ANGLE
ang.plot(ax=ax4, lw=1, x_compat=True, color="#b36ff6").legend(
    labels=["arm angle"], bbox_to_anchor=(0, 0.75), fontsize=20
)
textstr = "thresh: {}".format(
    np.round(self.arm_angle_thresholds[day], decimals=2)
)
ax4.text(
    1.01,
    0.95,
    textstr,
    transform=ax4.transAxes,
    fontsize=14,
    verticalalignment="top",
    bbox=props,
)

# SLEEP WAKE/REST PERIODS
df.plot(ax=ax5, lw=10, x_compat=True, ylim=(0, 1)).legend(
    bbox_to_anchor=(0, 0.85), fontsize=20
)

# FORMAT
plt.draw()
count = 0
for ax in all_axes:
    count += 1
    ax.set_xlabel("")
    ax.set_ylabel("")
    ax.spines["top"].set_visible(False)
    ax.spines["right"].set_visible(False)
    ax.spines["bottom"].set_visible(False)
    ax.spines["left"].set_visible(False)
    ax.grid(False)
    if count < 6:
        ax.get_xaxis().set_ticks([])
        ax.get_yaxis().set_ticks([])
    ax5.xaxis.set_major_locator(hours)
    ax5.xaxis.set_major_formatter(h_fmt)
    plt.subplots_adjust(wspace=0, hspace=0)
    fig.autofmt_xdate()
    for tick in ax5.xaxis.get_major_ticks():
        tick.label1.set_fontsize(16)
    pdf.savefig(fig.number)
pdf.close()
plt.close("all")

# SUMMARY REPORT
fig, (ax0, ax1, ax2, ax3, ax4, ax5) = plt.subplots(6, 1, figsize=(12, 12))
plt.suptitle("Summary Report for Source: {}".format(self.src_name), fontsize=16)
all_axes = (ax0, ax1, ax2, ax3, ax4, ax5)

```

*FIG. 11J*

: CONTINUES IN FIG. 11K

56/59

CONTINUES FROM FIG. 11J

```

# ENDPOINTS DATAFRAME
endpoints = pd.DataFrame(self.endpoints[1:])
endpoints.columns = self.endpoints[0]
if endpoints.isna().values.all():
    self.logger.error(
        "No valid endpoints detected across all days, aborting summary report generation..."
    )
    return
ylabels = [
    "Total Sleep\nTime(min)\nMean: {}".format(
        int(np.round(endpoints.total_sleep_time.mean()))
    ),
    "Percent Time\nAsleep\nMean: {}".format(
        int(np.round(endpoints.percent_time_asleep.mean()))
    ),
    "Wake After\nSleep Onset(min)\nMean: {}".format(
        int(np.round(endpoints.waso.mean()))
    ),
    "Sleep Onset\nLatency(min)\nMean: {}".format(
        int(np.round(endpoints.sleep_onset_latency.mean()))
    ),
    "Number of\nWake Bouts\nMean: {}".format(
        int(np.round(endpoints.number_wake_bouts.mean()))
    ),
    "Total Sleep\nOpportunity(min)\nMean: {}".format(
        int(np.round(endpoints.total_sleep_opportunity.mean()))
    ),
]

# TOTAL SLEEP TIME
endpoints.total_sleep_time.plot.bar(
    ax=ax0, title="", color=["C" + str(i) for i in range(len(endpoints))]
)
# PERCENT TIME ASLEEP
endpoints.percent_time_asleep.plot.bar(
    ax=ax1, title="", color=["C" + str(i) for i in range(len(endpoints))]
)
# WAKE AFTER SLEEP ONSET
endpoints.waso.plot.bar(
    ax=ax2, title="", color=["C" + str(i) for i in range(len(endpoints))]
)
# SLEEP ONSET LATENCY
endpoints.sleep_onset_latency.plot.bar(
    ax=ax3, title="", color=["C" + str(i) for i in range(len(endpoints))]
)
# NUMBER OF WAKE BOUTS
endpoints.number_wake_bouts.plot.bar(
    ax=ax4, title="", color=["C" + str(i) for i in range(len(endpoints))]
)
# TOTAL SLEEP OPPORTUNITY
endpoints.total_sleep_opportunity.plot.bar(
    ax=ax5, title="", color=["C" + str(i) for i in range(len(endpoints))]
)

```

CONTINUES IN FIG. 11L

*FIG. 11K*

57/59

CONTINUES FROM FIG. 11K :

```

# FORMAT
count = 0
for ax in all_axes:
    count += 1
    ax.spines["top"].set_visible(False)
    ax.spines["right"].set_visible(False)
    ax.spines["bottom"].set_visible(False)
    ax.spines["left"].set_visible(False)
    ax.grid(False)
    ax.set_ylabel(ylabels[count - 1], rotation=0, fontsize=12, labelpad=50)
    if count < 5:
        ax.set_xlabel("")
        ax.get_xaxis().set_ticks([])
        ax.get_yaxis().set_ticks([])
    else:
        ax.set_xlabel("Day", fontsize=20)
        ax.get_yaxis().set_ticks([])
    plt.setp(ax.xaxis.get_majorticklabels(), rotation=0)
    for p in ax.patches:
        ax.annotate(
            np.round(p.get_height(), decimals=2),
            (p.get_x() + p.get_width() / 2.0, 0),
            ha="center",
            va="center",
            xytext=(0, 10),
            textcoords="offset points",
            fontweight="bold",
        )
    plt.subplots_adjust(wspace=0, hspace=0.01)
    plt.xticks(fontsize=20)
    plt.draw()
    plt.savefig(self.dst + "/" + self._summary_report.pdf".format(self.src_name))
    plt.close("all")
    return

def export(self):
    """
    Exports a results table to csv.
    """
    header = ["date"] + self.endpoints[0] + ["sleep_start", "sleep_end"]
    days = list(range(0, len(self.raw_days_to_plot)))
    export_df = []
    for day in days:
        idx_start = self.raw_days_to_plot[day].index[0]
        idx_start = (
            idx_start.replace(hour=12, minute=0, second=0, microsecond=0)
            - pd.to_timedelta("24h")
        )
        if idx_start
            < idx_start.replace(hour=12, minute=0, second=0, microsecond=0)
        else idx_start.replace(hour=12, minute=0, second=0, microsecond=0)
    )
    if self.major_rest_periods[day]:
        export_df.append(
            [str(idx_start.date())
             + self.endpoints[day + 1]
             + [str(i) for i in self.major_rest_periods[day]]
            ]
        )

```

*FIG. 11L*

CONTINUES IN FIG. 11M

58/59

CONTINUES FROM FIG. 11L .

```
else:
    export_df.append(
        [str(idx_start.date())]
        + [np.nan for i in self.endpoints[day + 1]]
        + [np.nan, np.nan]
    )

export_df = pd.DataFrame(export_df)
export_df.columns = header
export_df.to_csv(
    self.dst + "/" + self.output_table.csv.format(self.src_name), index=False
)
return

if __name__ == "__main__":
    infile = ""
    outdir = ""
    SleepPy(
        input_file=infile,
        results_directory=outdir,
        sampling_frequency=None,
        aws_object=None,
        start_buffer="0s",
        stop_buffer="0s",
        start_time="",
        stop_time="",
        temperature_threshold=25.0,
        minimum_rest_block=30,
        allowed_rest_break=150,
        minimum_rest_threshold=0.1,
        maximum_rest_threshold=1.0,
        minimum_rest_period=None,
        movement_based_nonwear=0.001,
        minimum_hours=12,
        downsample=True,
        plot=True,
        fast_load=True,
        logger=None,
    ).run()
```

*FIG. 11M*

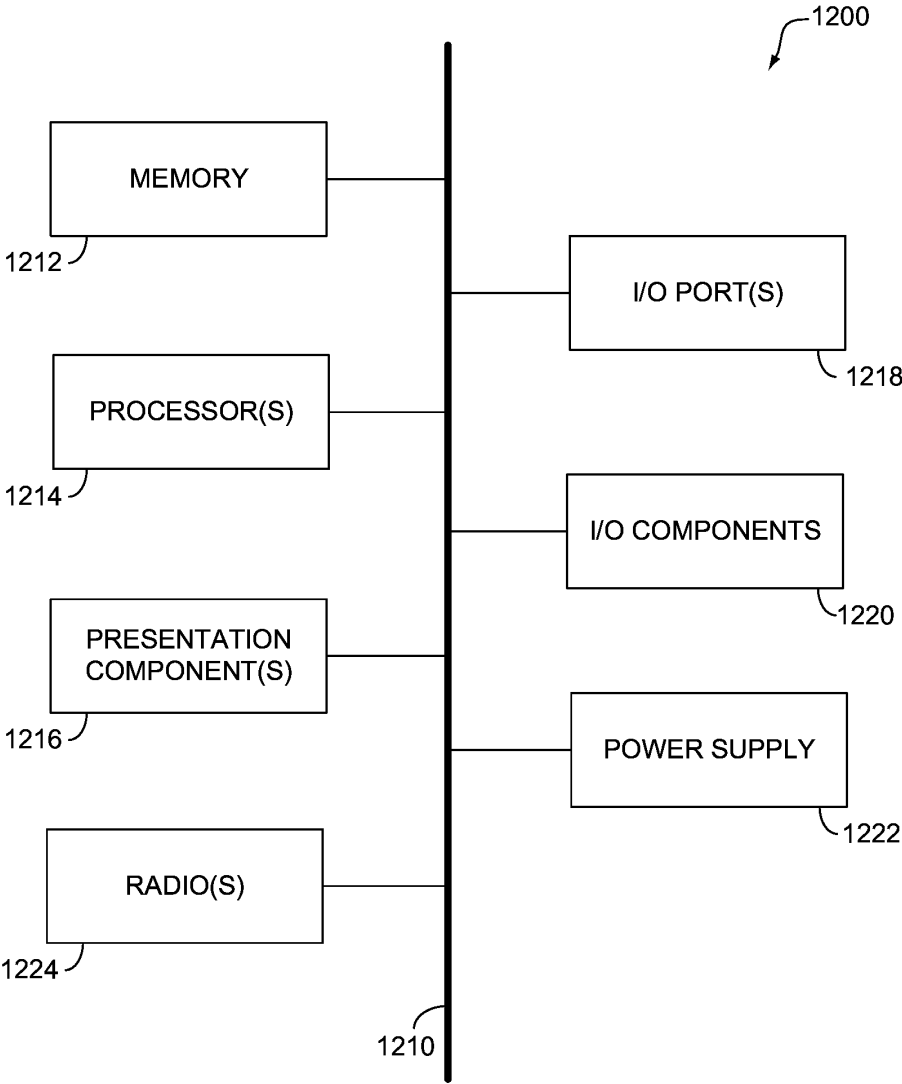


FIG. 12