

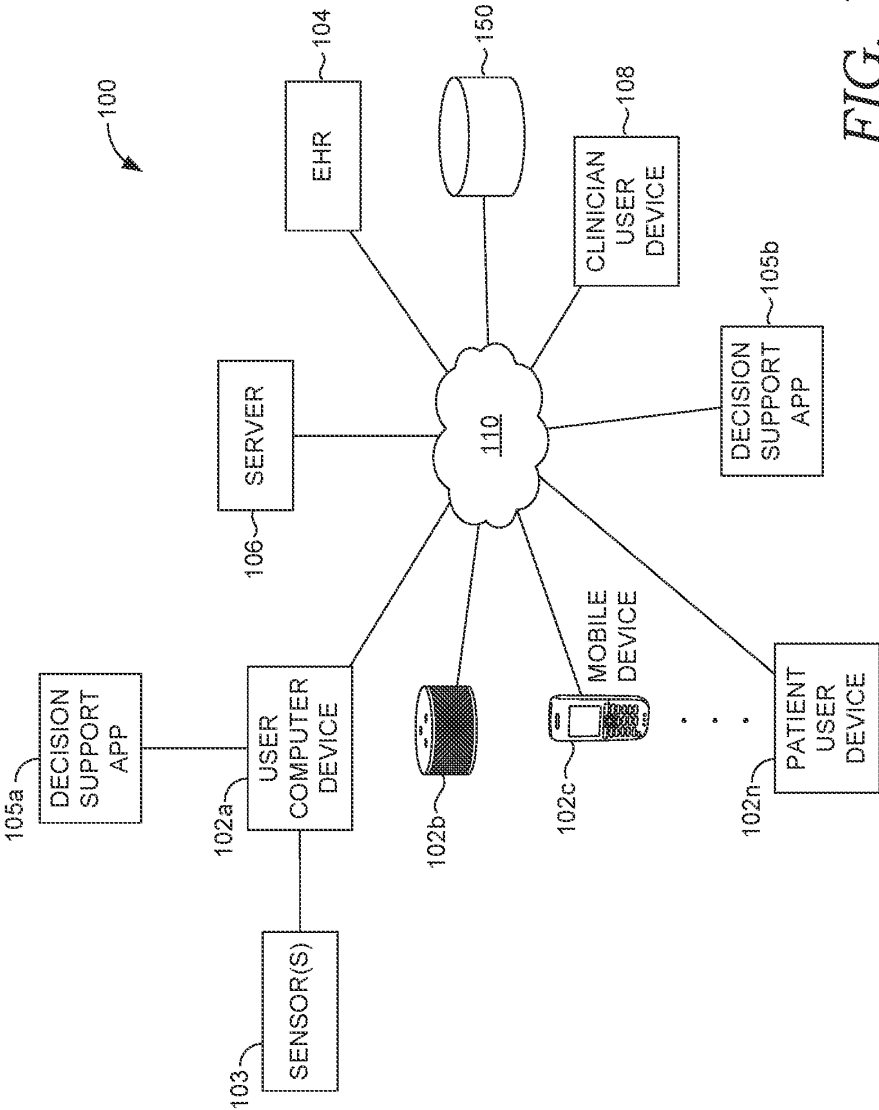
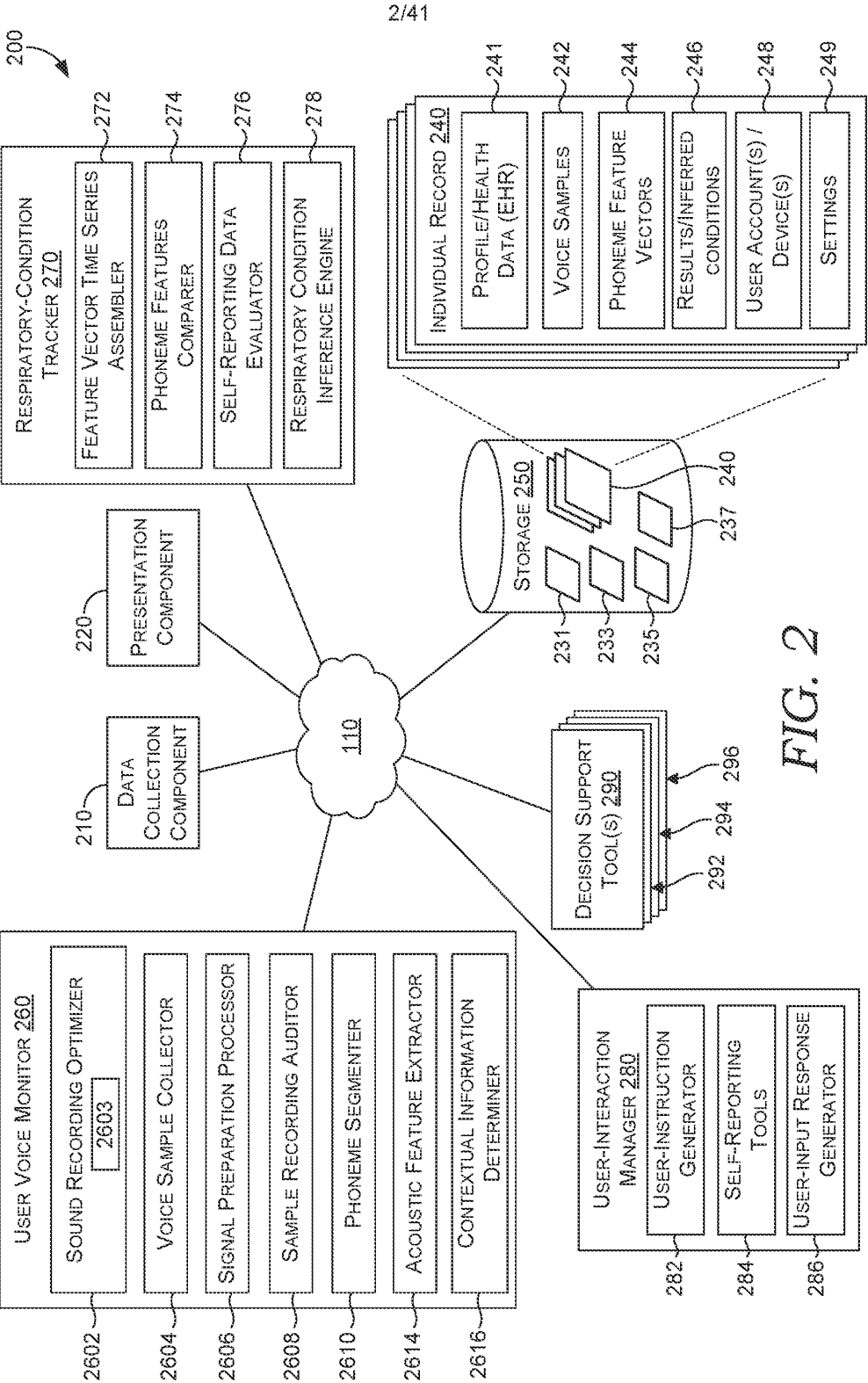


FIG. 1



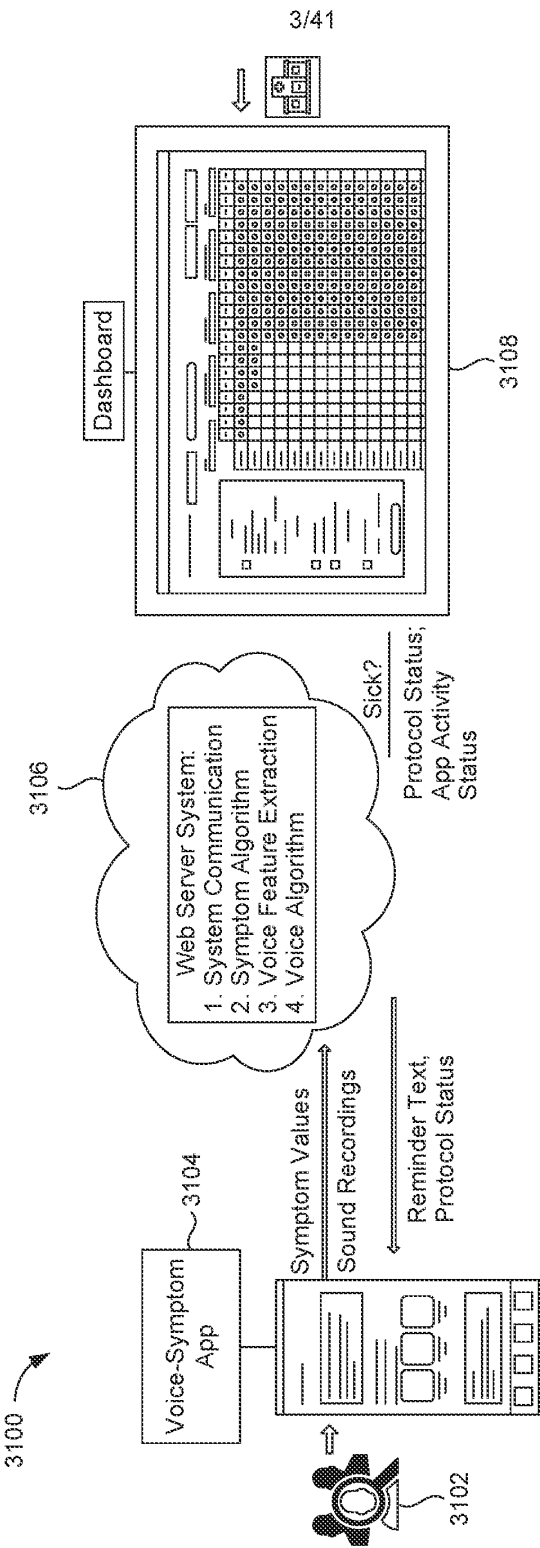


FIG. 3A

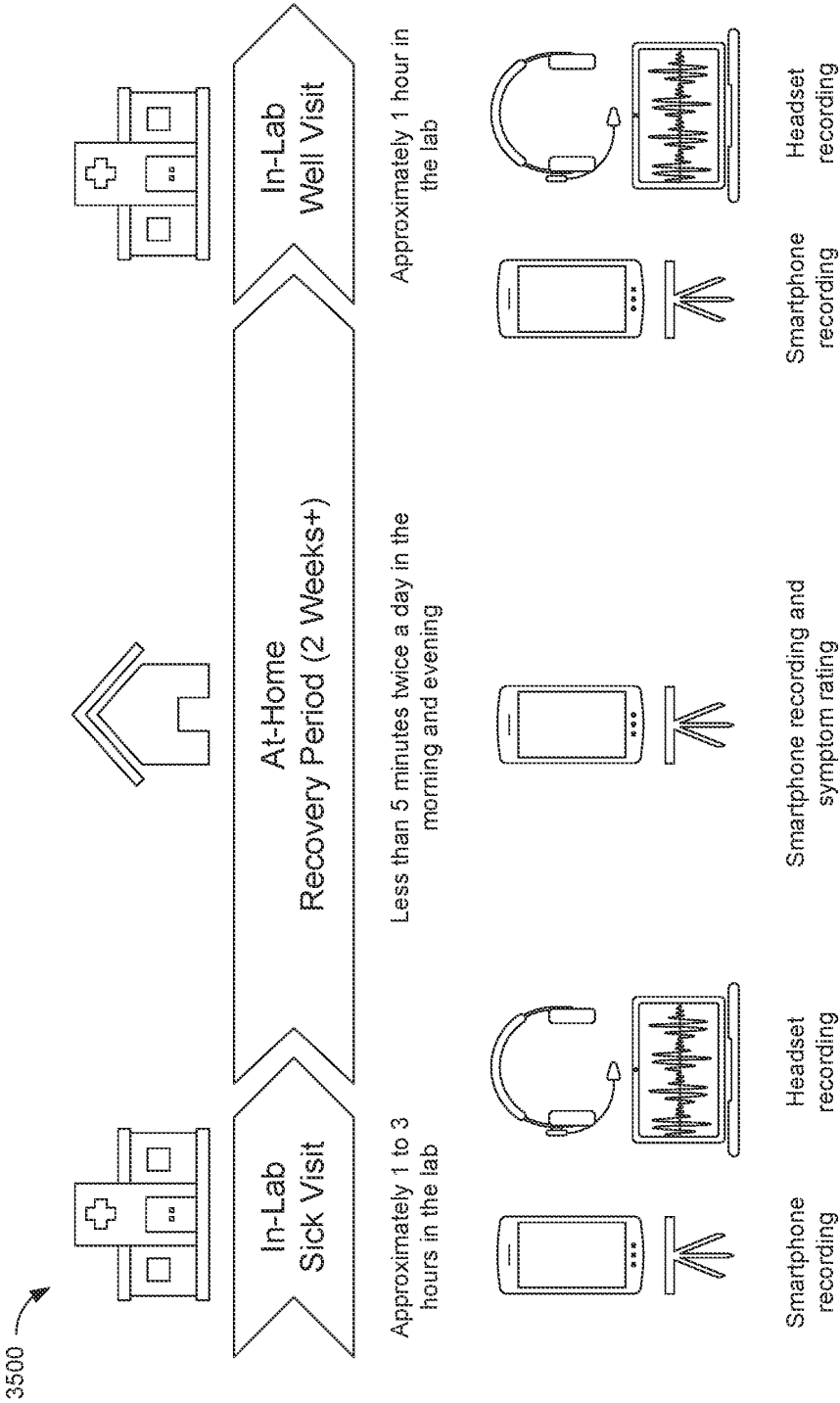


FIG. 3B

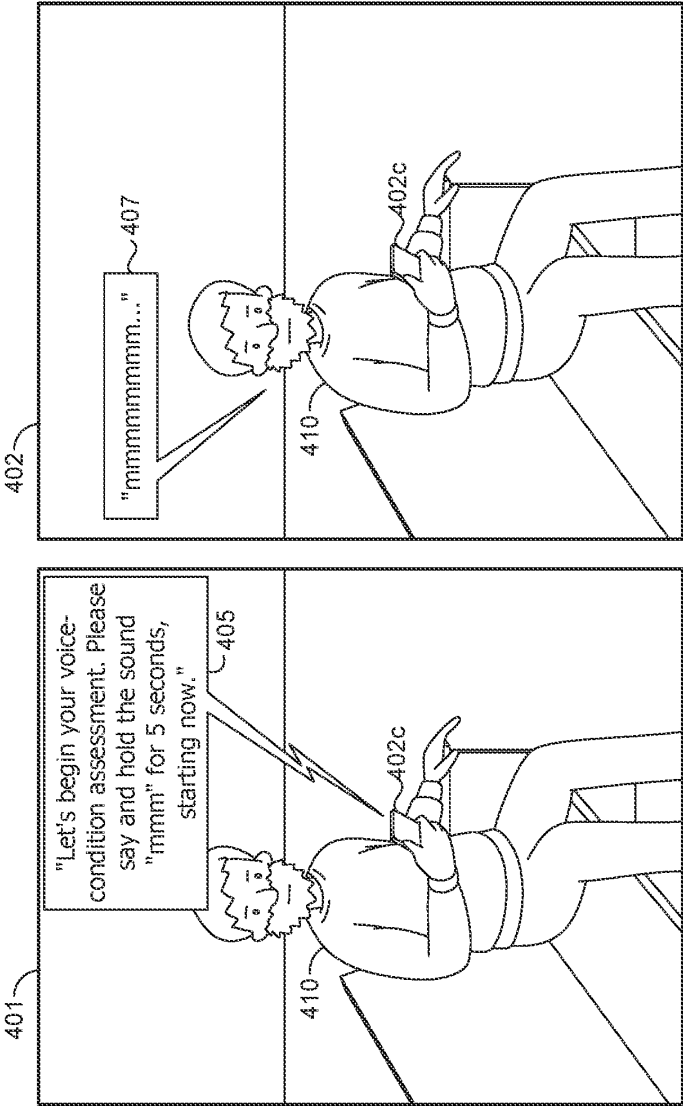


FIG. 4A

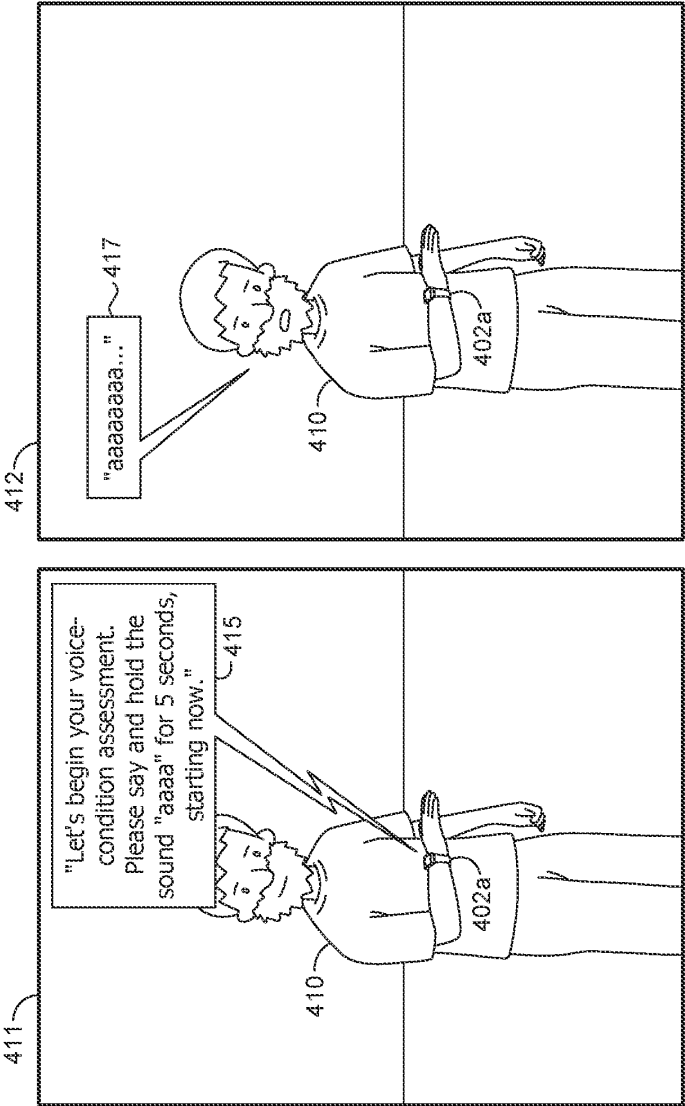


FIG. 4B

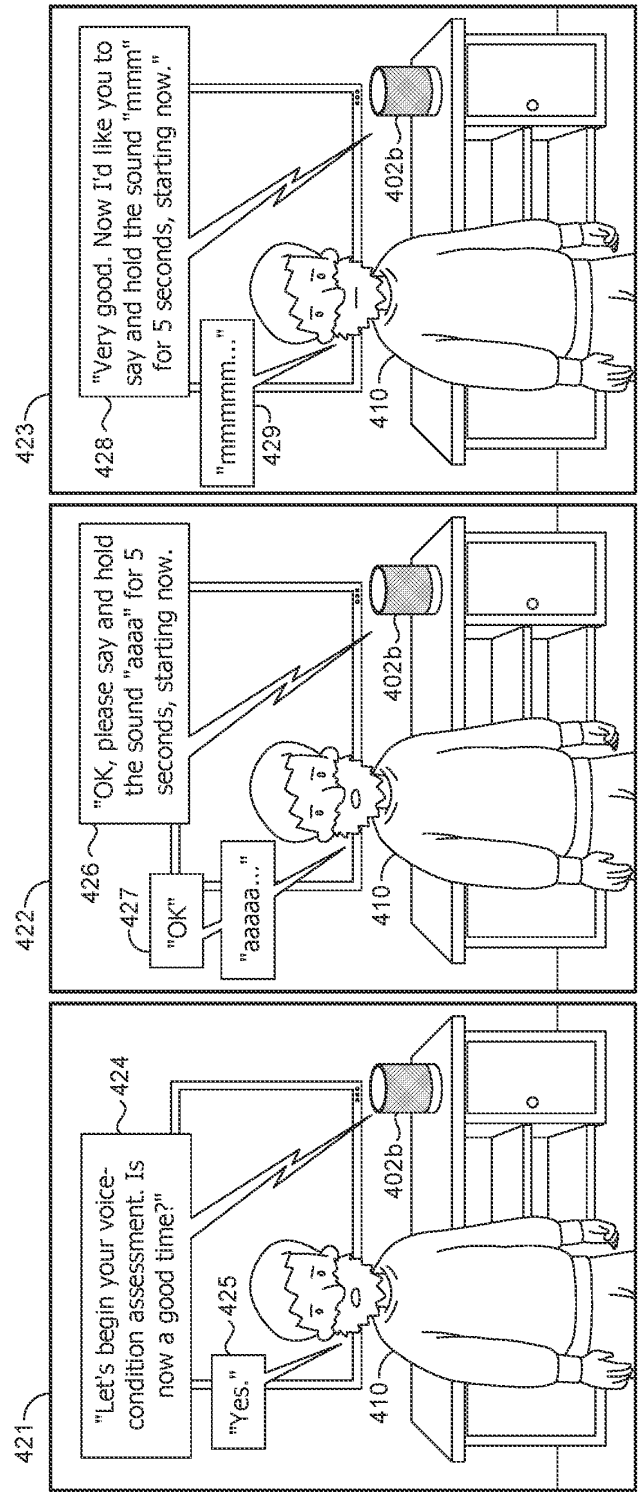


FIG. 4C

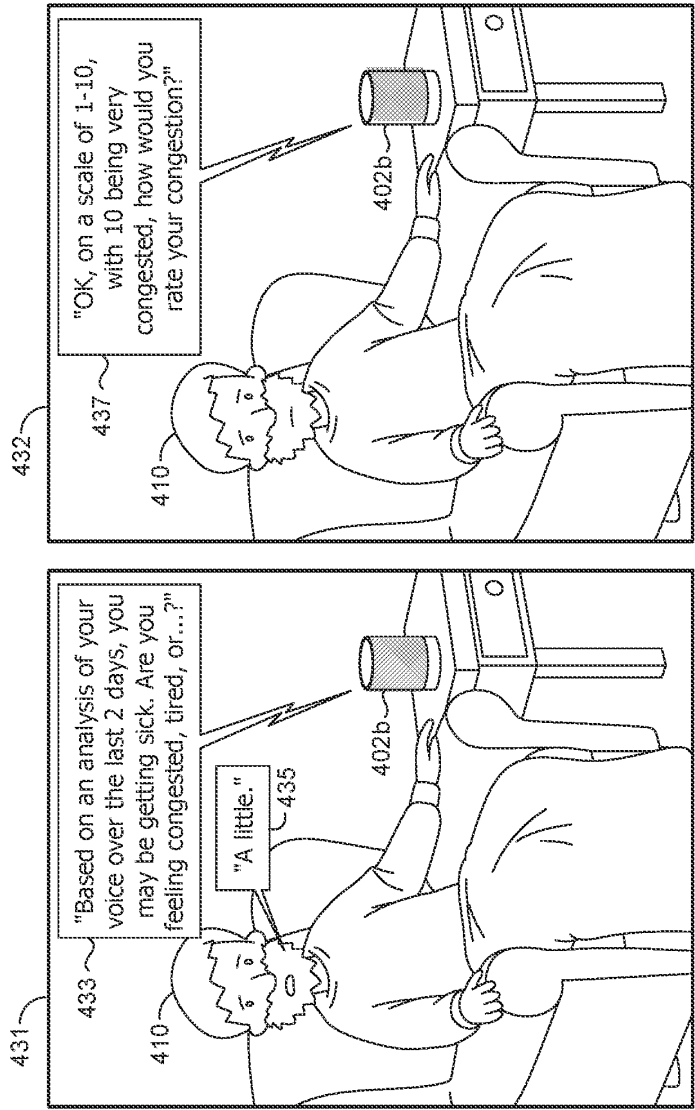


FIG. 4D

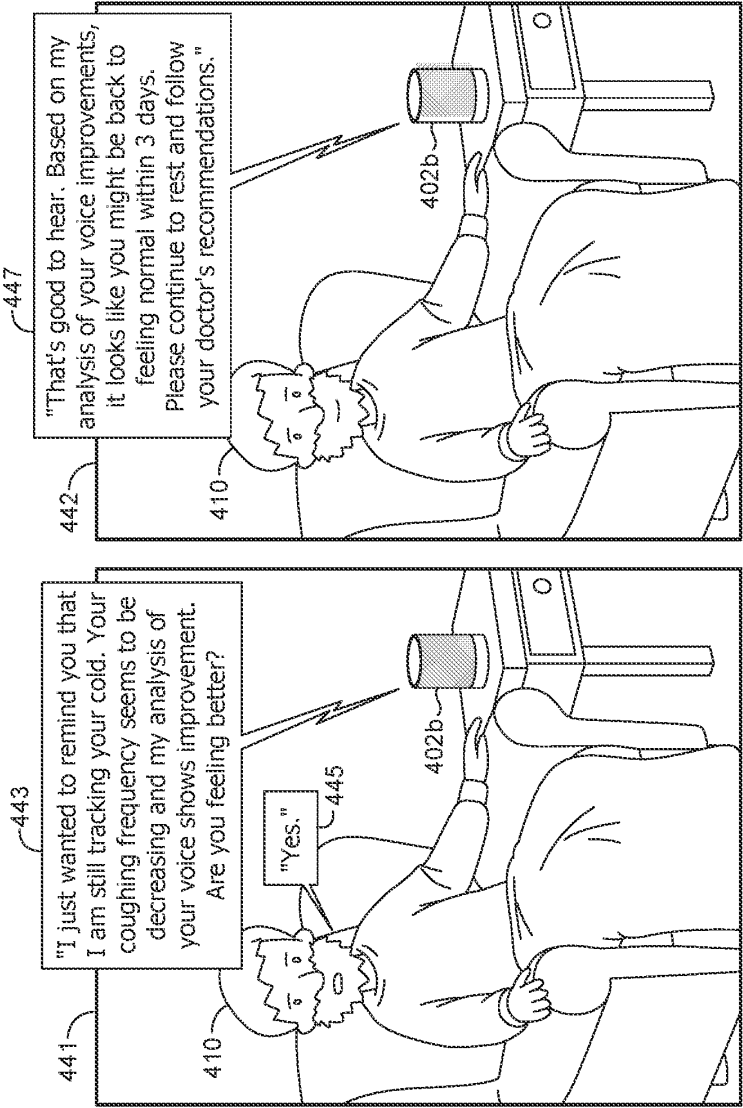


FIG. 4E

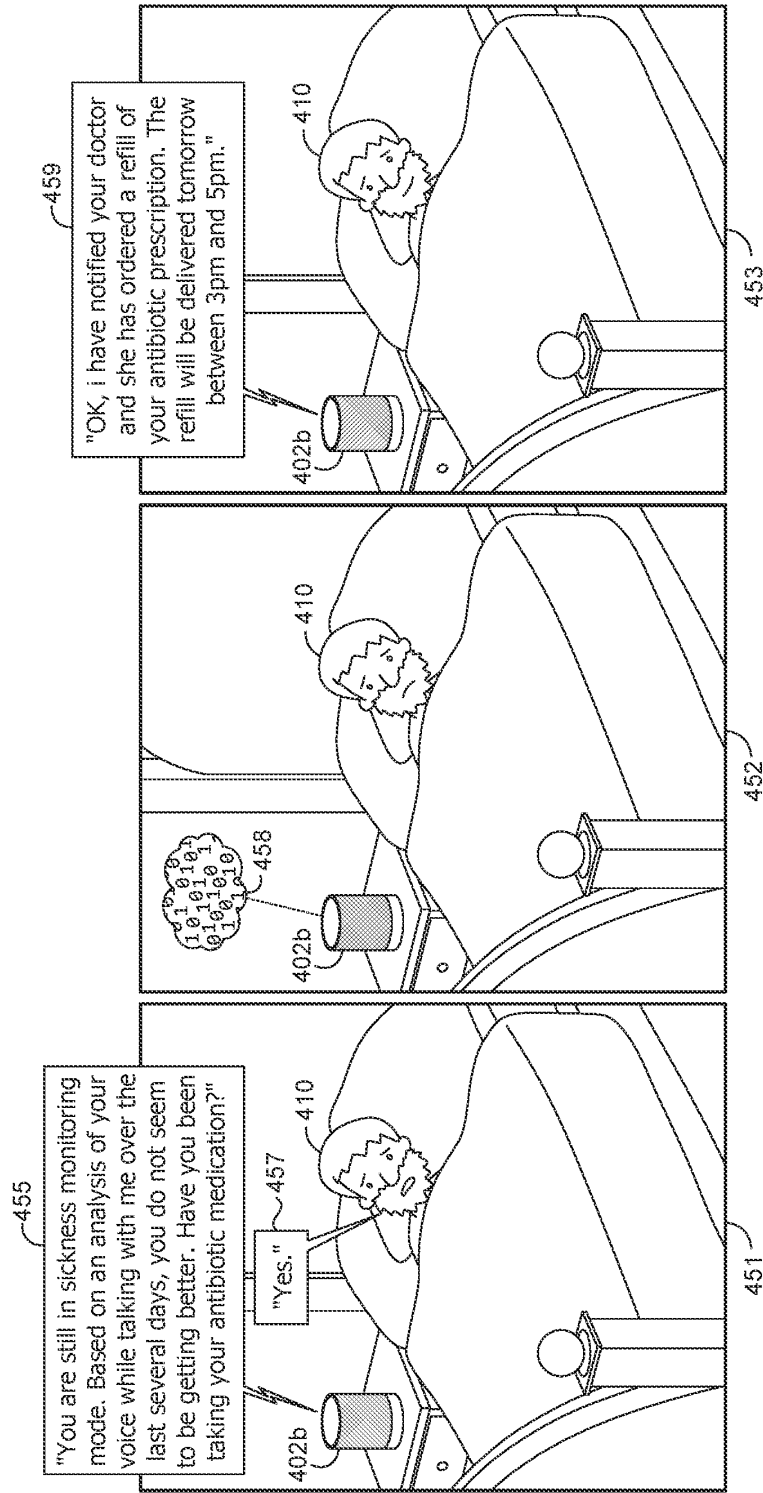


FIG. 4F

11/41

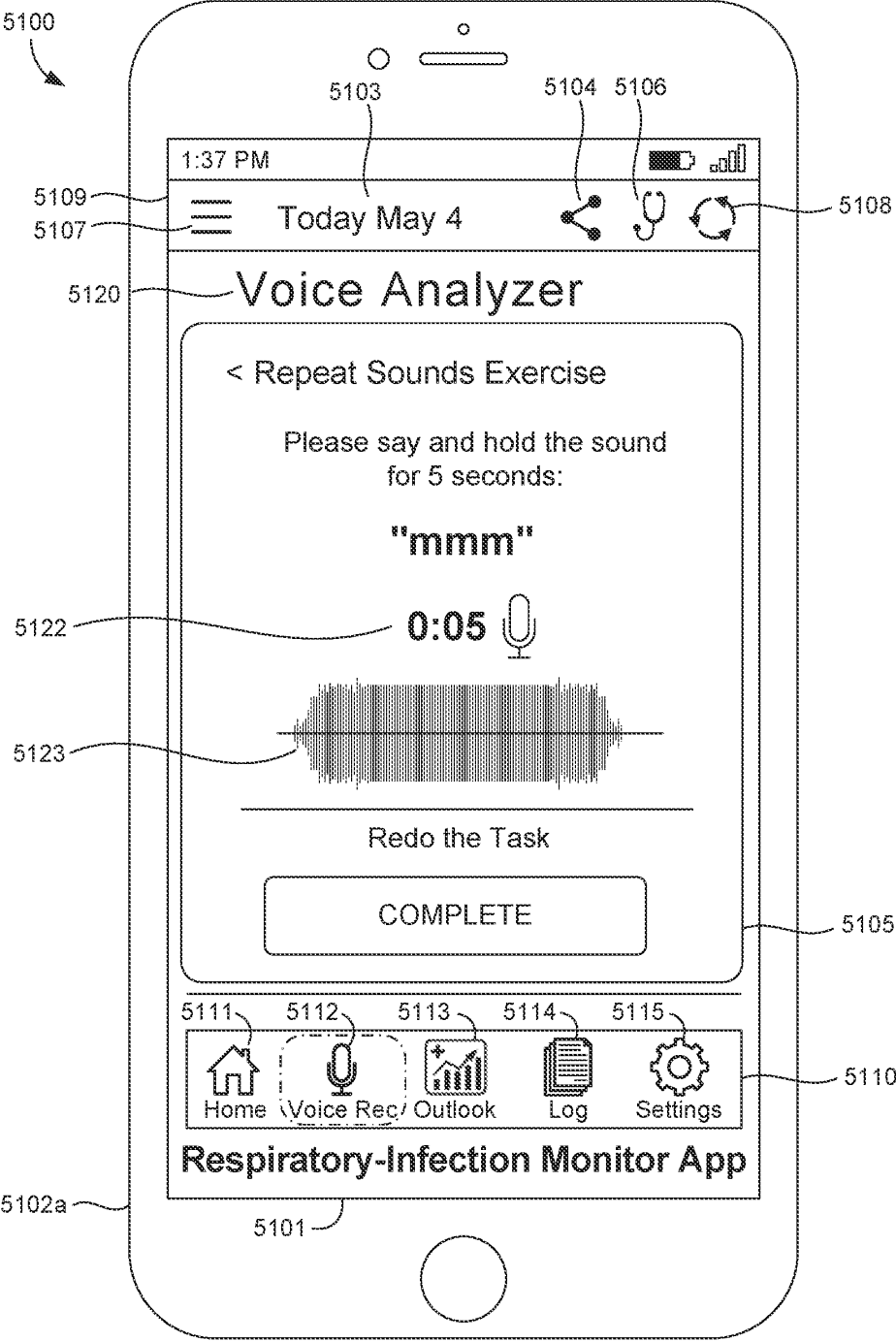
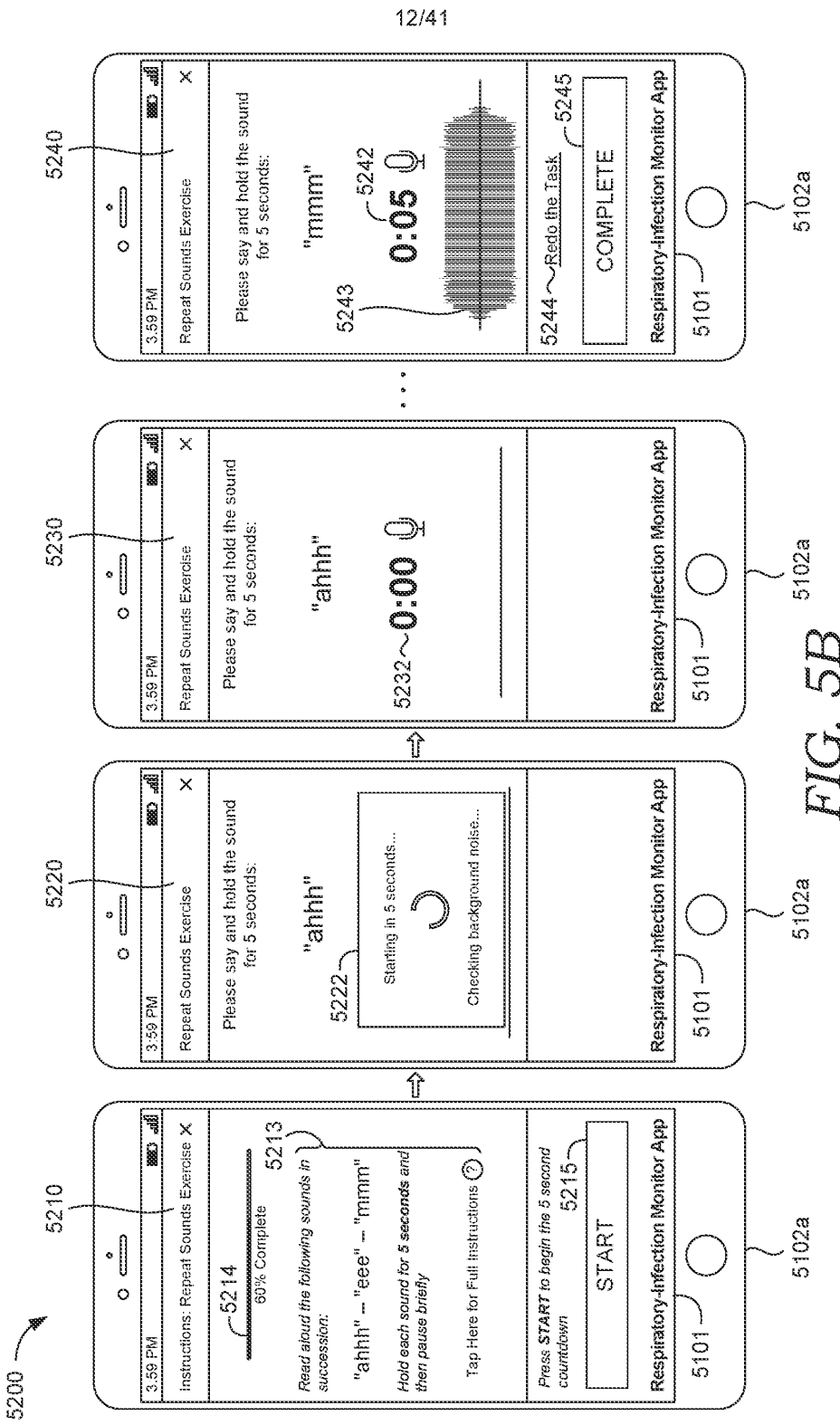


FIG. 5A



13/41

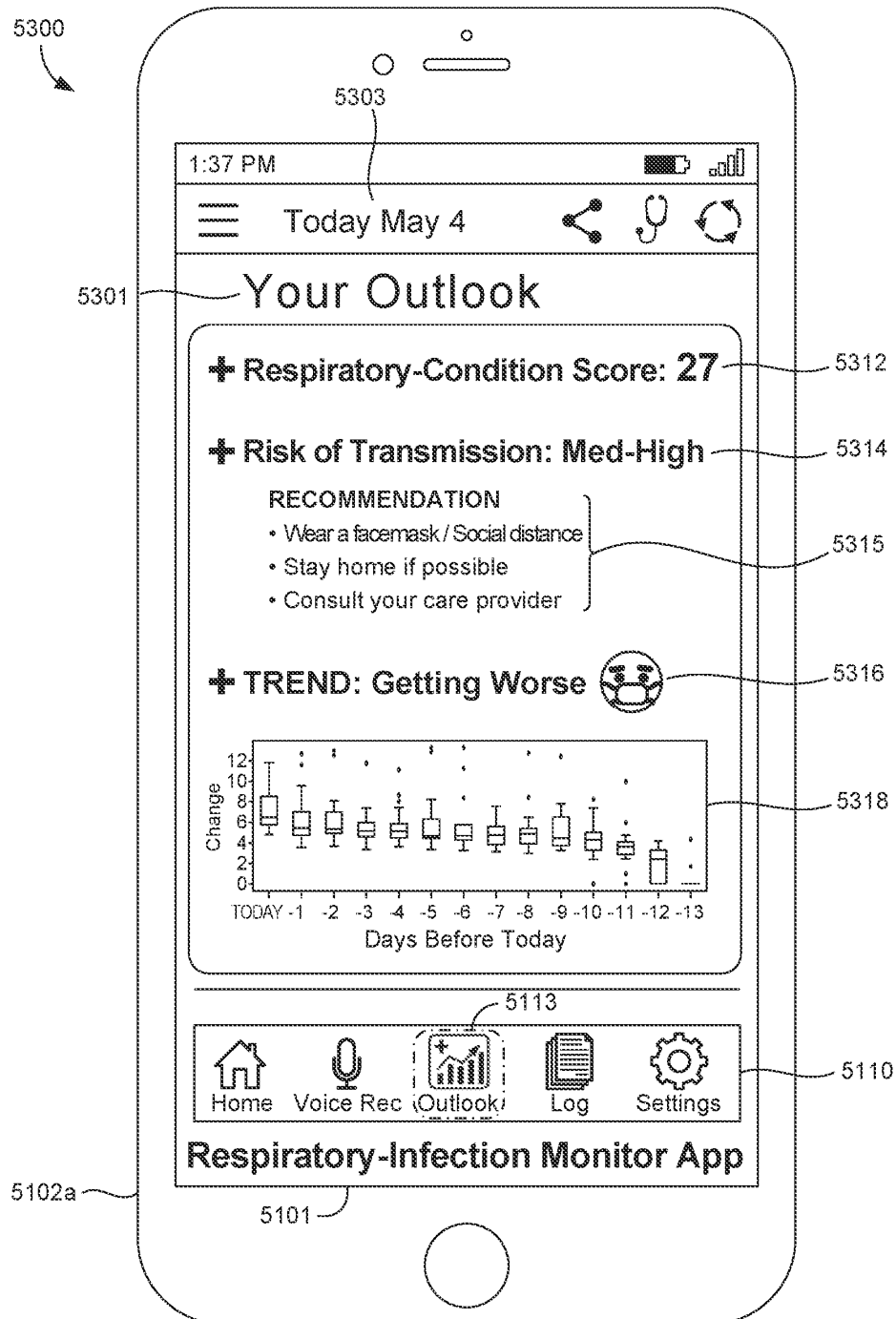


FIG. 5C

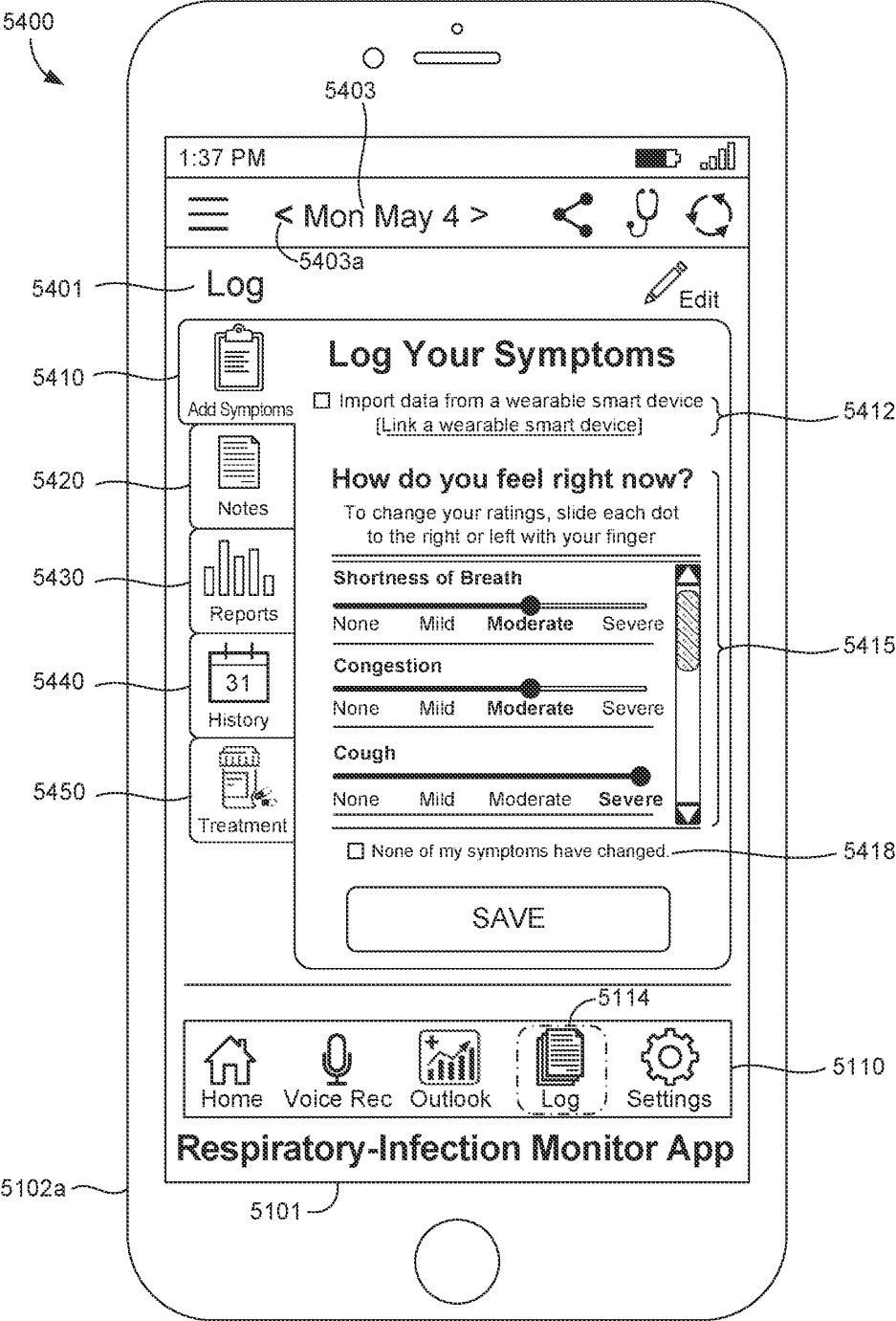
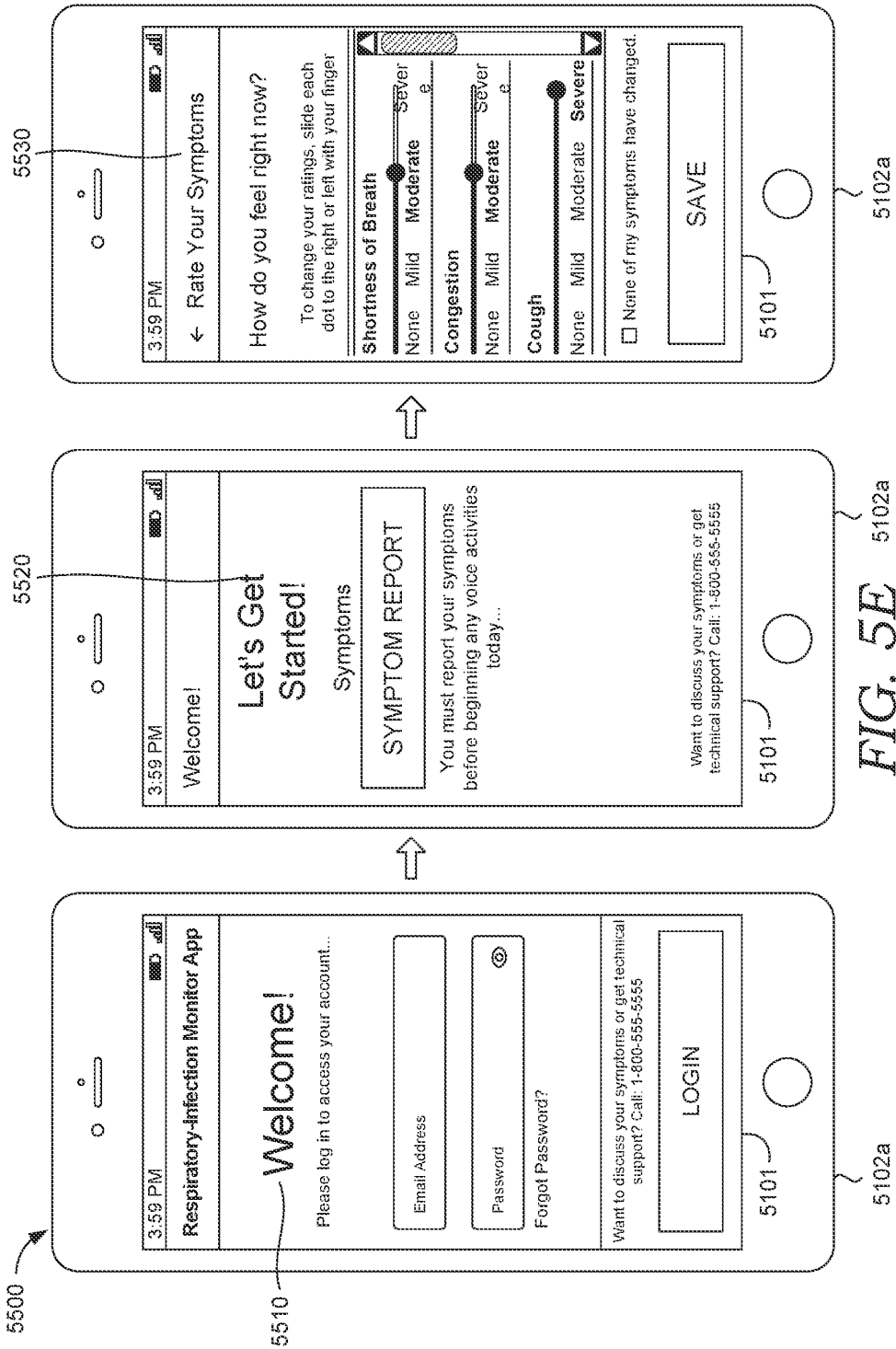
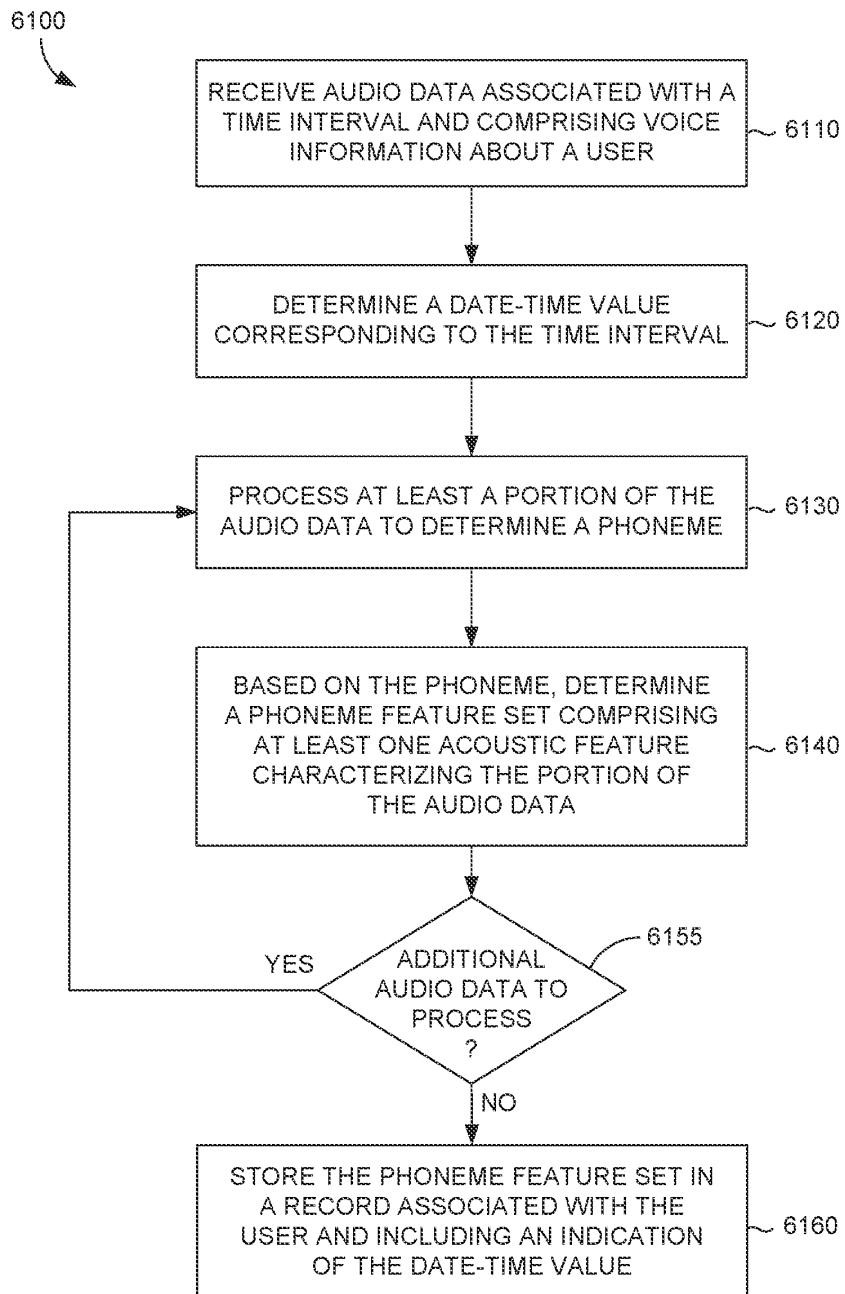


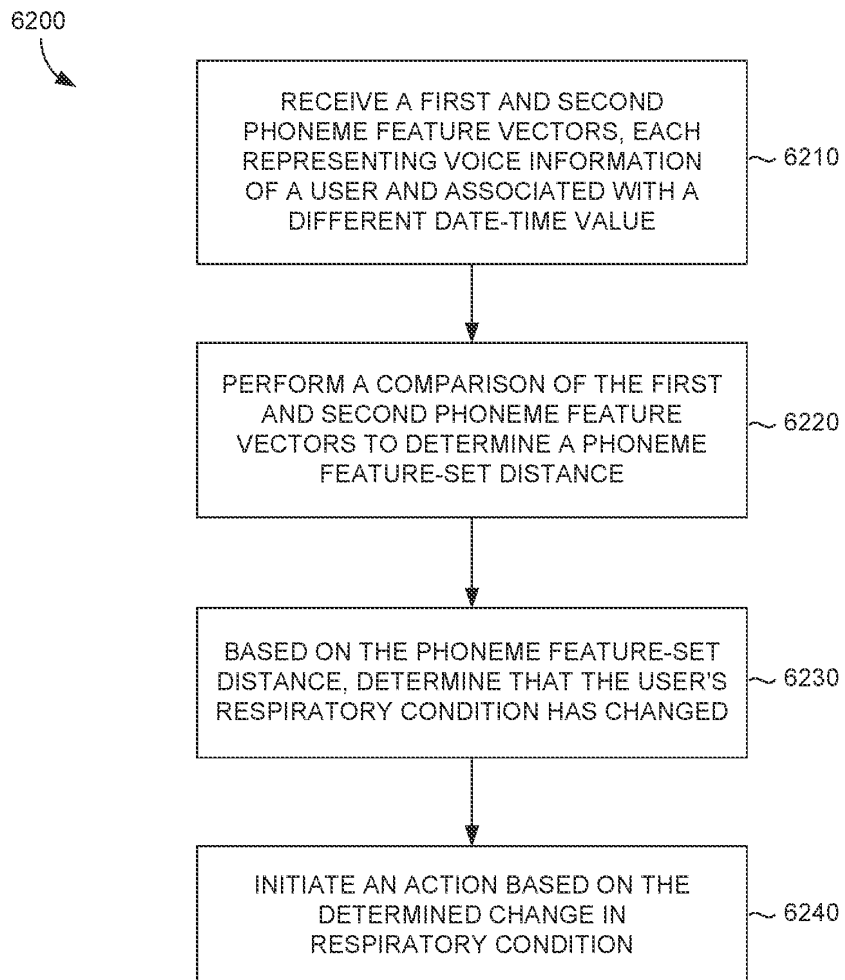
FIG. 5D



16/41

*FIG. 6A*

17/41

*FIG. 6B*

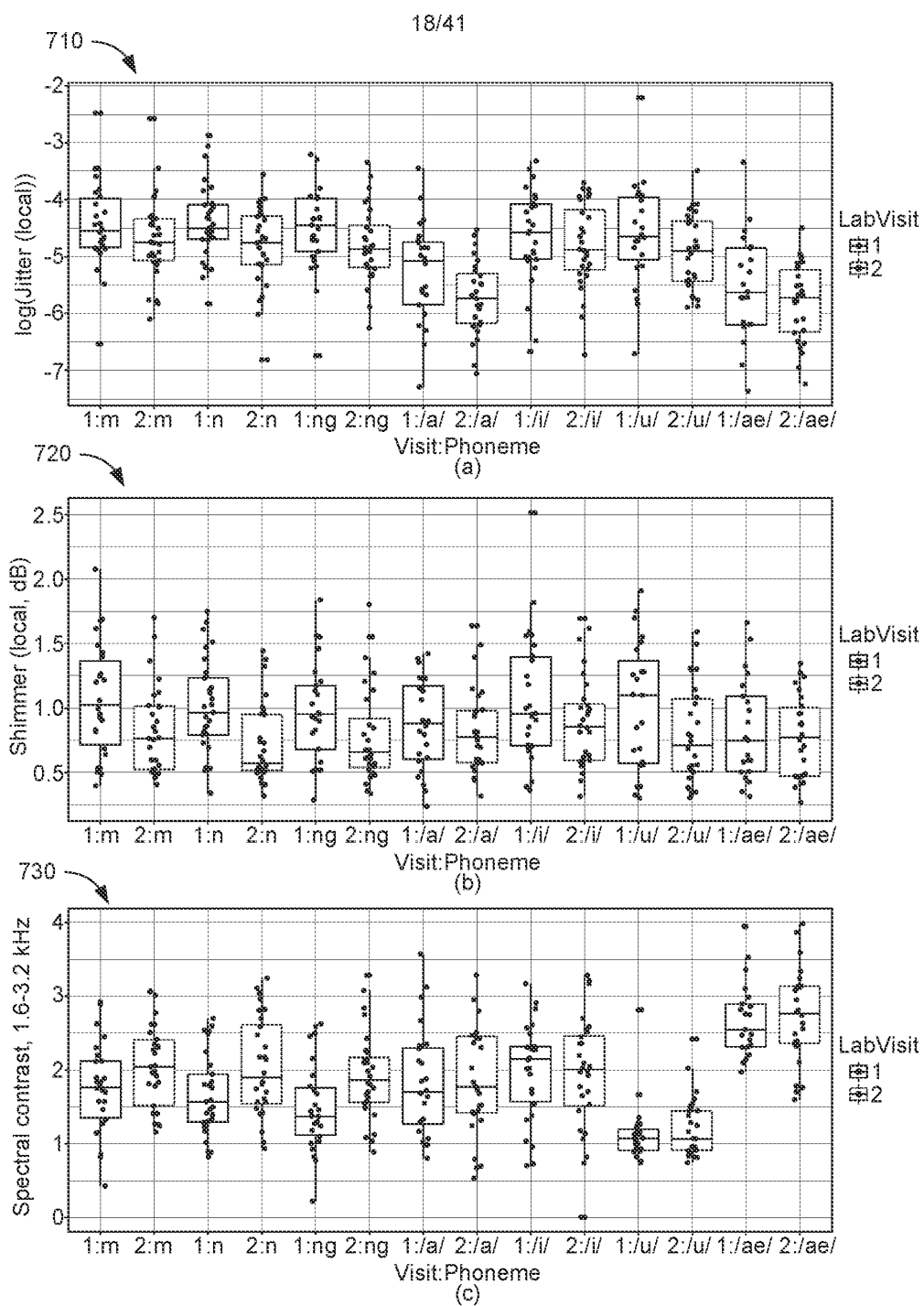


FIG. 7

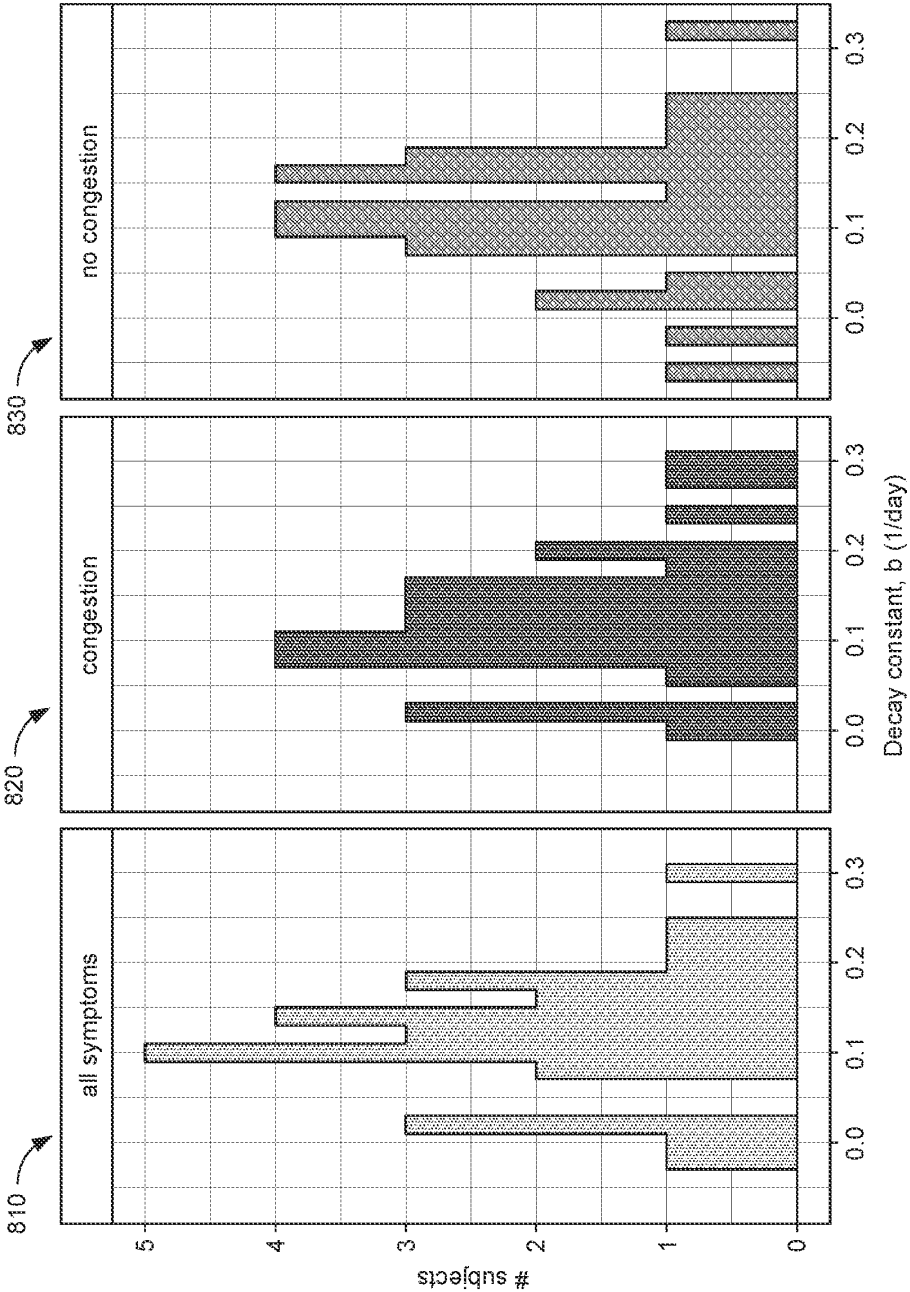


FIG. 8

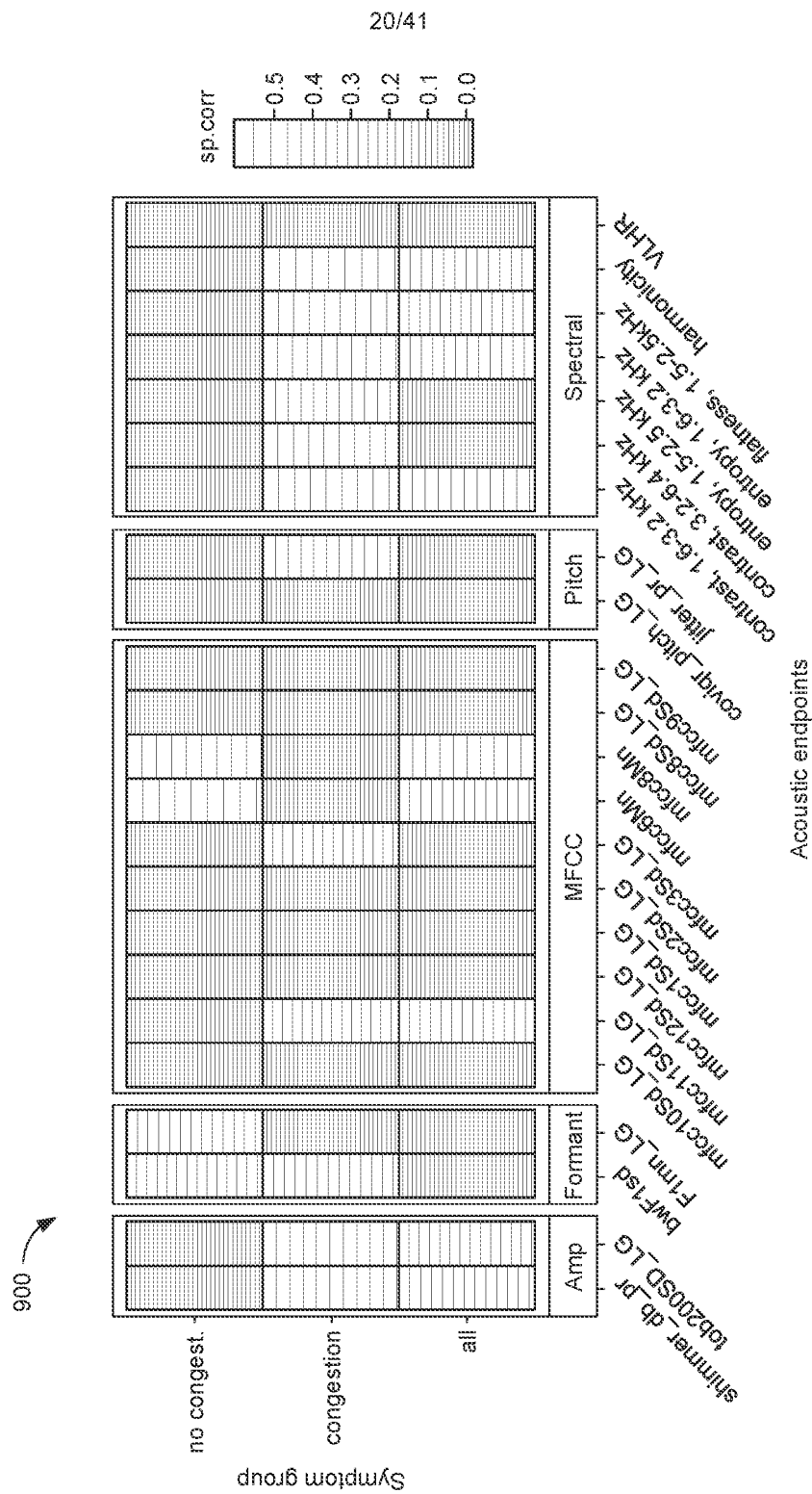
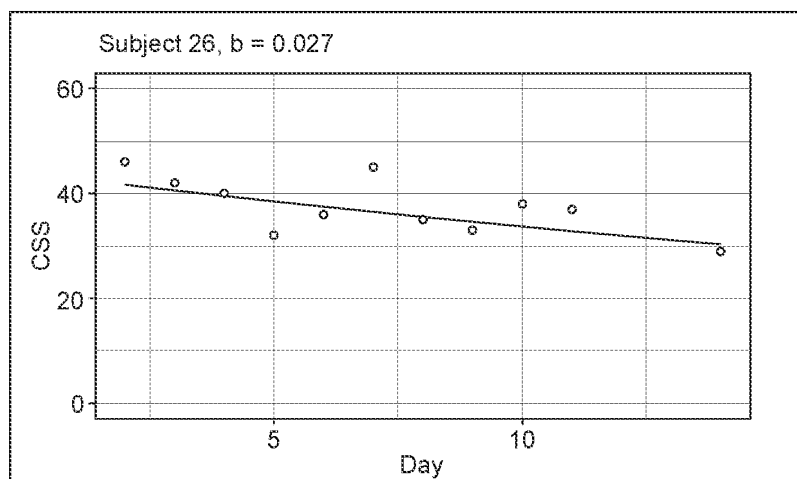


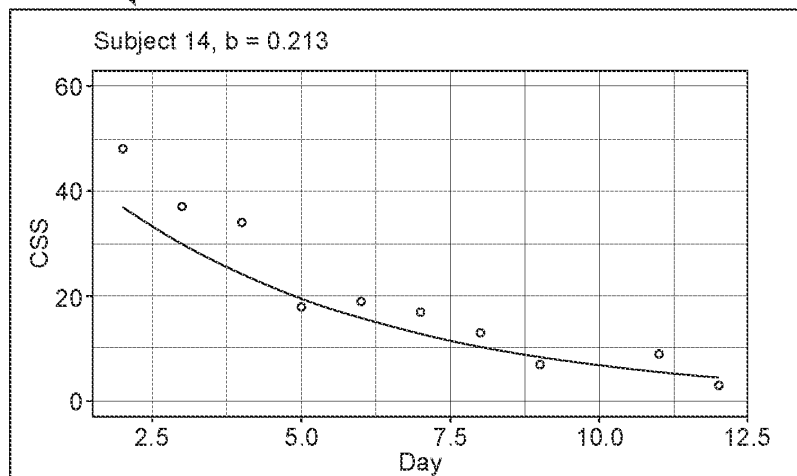
FIG. 9

21/41

1010



1020

*FIG. 10*

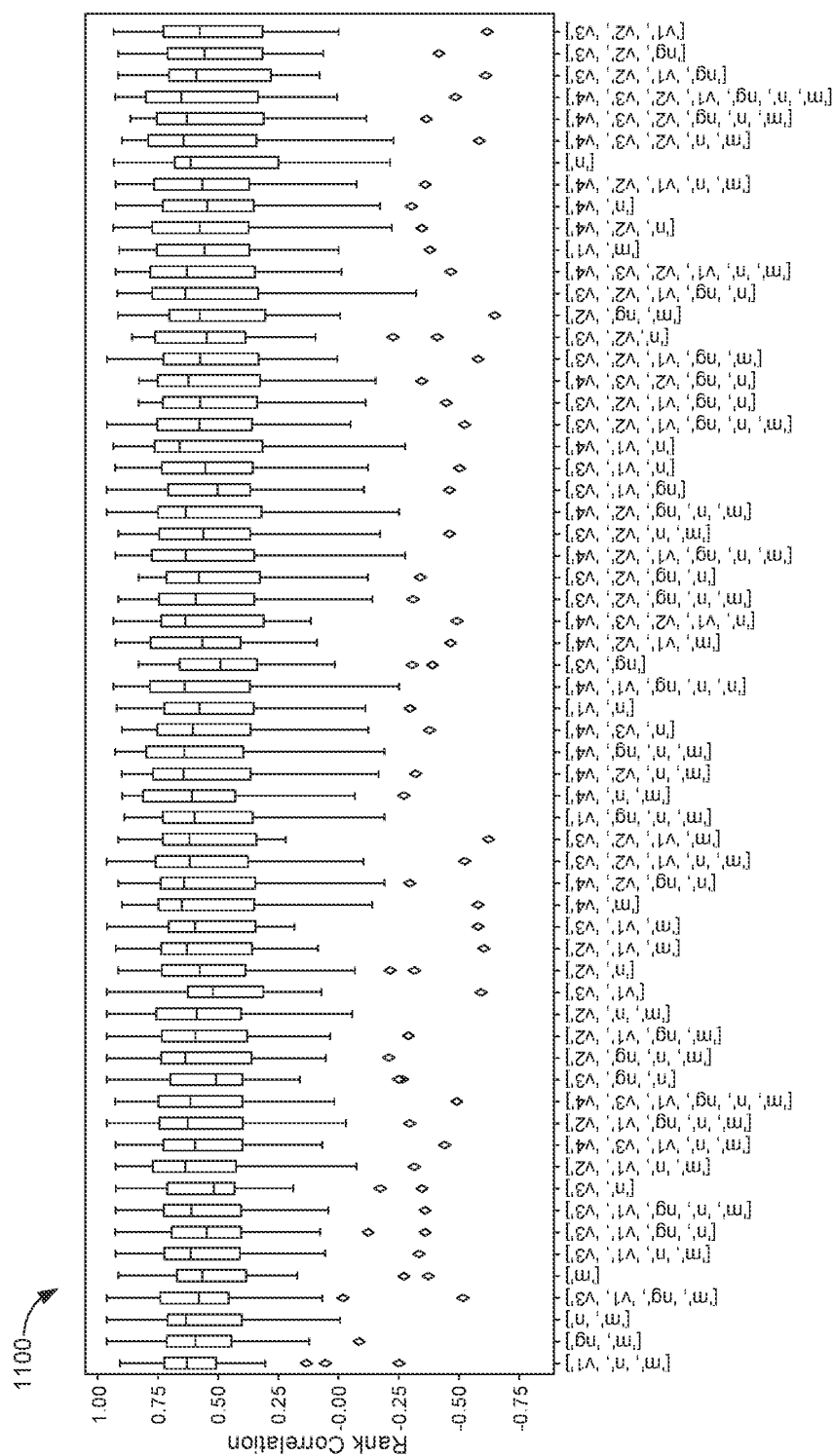


FIG. 11A

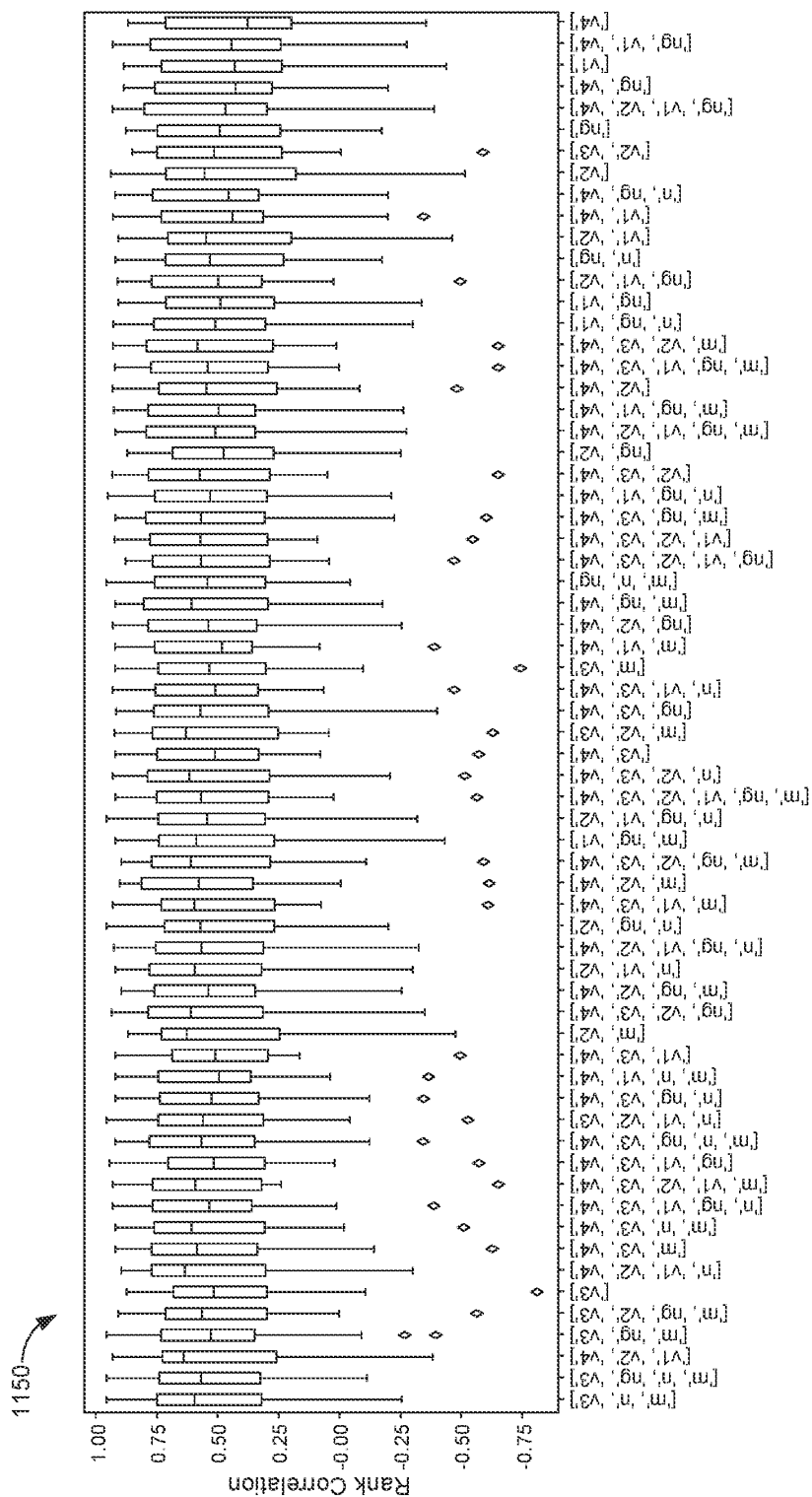


FIG. 11B

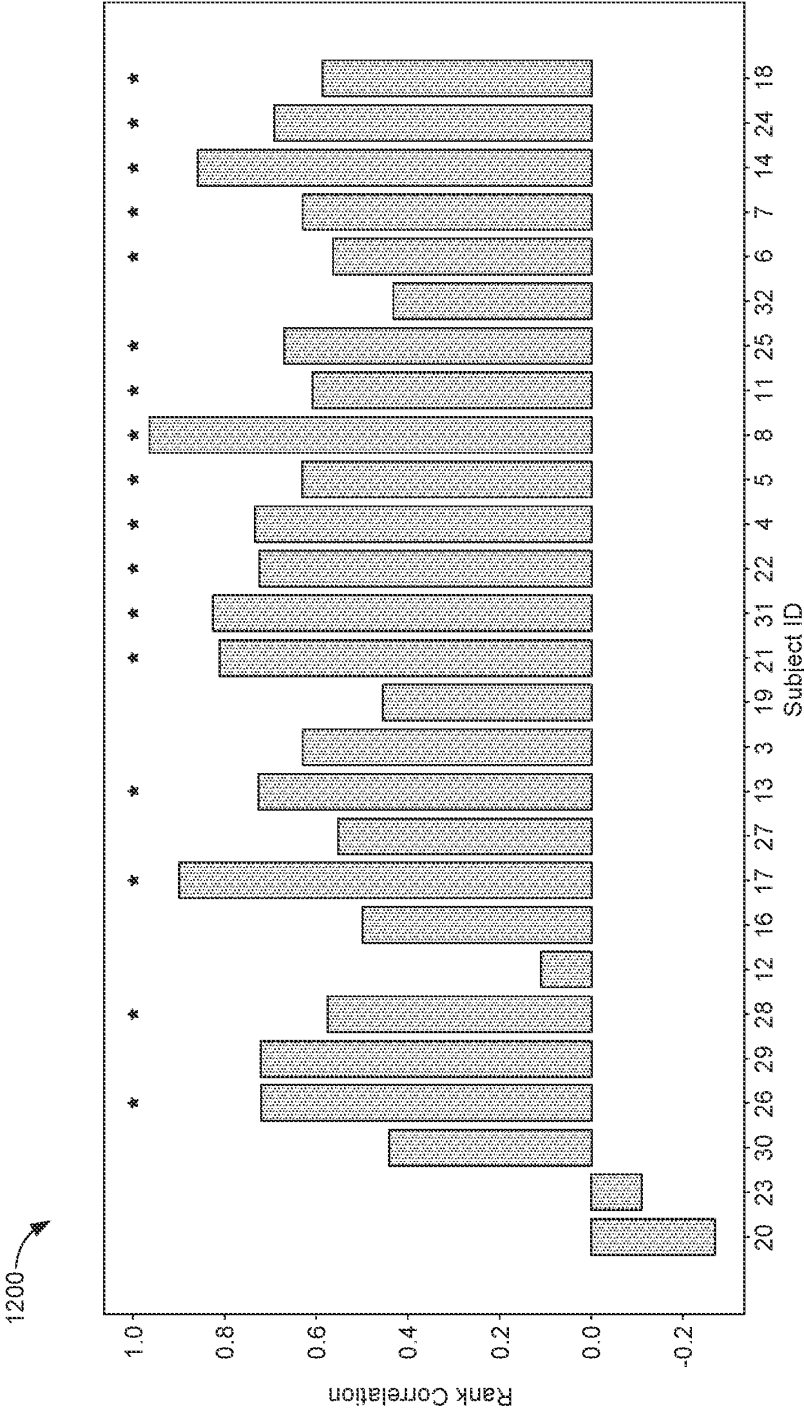
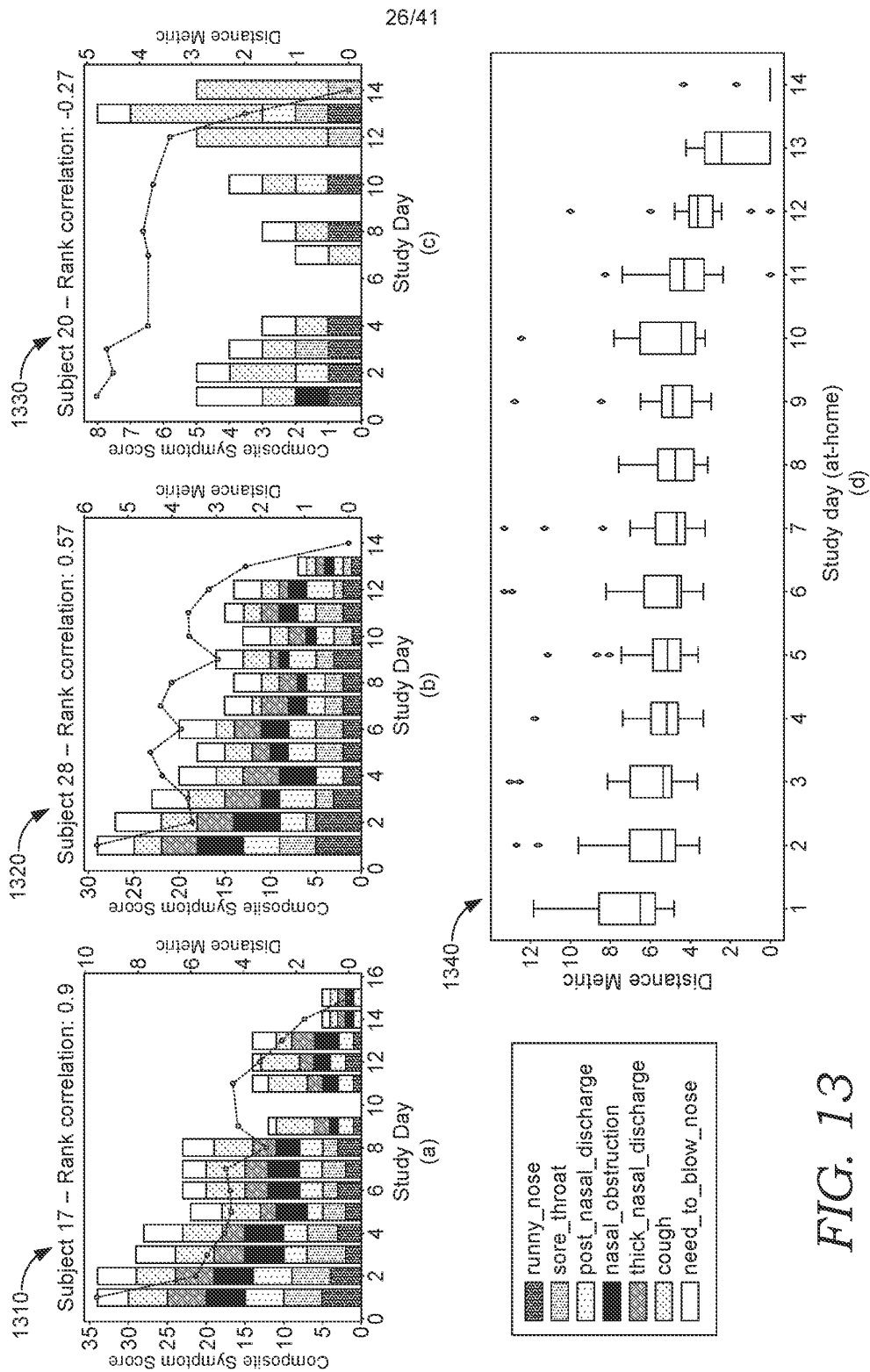


FIG. 12A

Category	Feature	m	N	ng	/a/	/i/	/u/	/ae/	Subject group
Pitch-related	covqr_pitch, i.e. IQR/median (LG) (LG)						0.046		all
	covqr_pitch			0.004					high-recovery
	Jitter, local (LG)		0.0024		4.47E-2				all
Amplitude-related	Jitter, local (LG)		4.4E-2						high-recovery
	RMS (LG) (*)	2.20E-3	2.00E-4	9.7E-3					all
	RMS (LG) (*)	1.21E-2	5.4E-3	2.06E-2					high-recovery
	Shimmer, dB	9.5E-3	3.4E-3						all
	Shimmer, dB	3.07E-2							high-recovery
Formants	200 Hz 1/3 octave band, sd (LG)	1.41E-2		2.18E-2		3.12E-2		1.01E-2	all
	200 Hz 1/3 octave band, sd (LG)					4.72E-2		4.95E-2	high-recovery
	F1 mn (LG)		2.958E-2		4.76E-2				all
	F1 (LG)		1.07E-2			3.91E-2			high-recovery
	F1 bandwidth sd	4.77E-2							all
Spectral structure	F1 bandwidth sd								high-recovery
	Entropy, 1.5-2.5 kHz	6.00E-3		1.13E-2					all
	Entropy, 1.5-2.5 kHz	1.42E-2		6.3E-3					high-recovery
	Entropy, 1.6-3.2 kHz			1E-3					all
	Entropy, 1.6-3.2 kHz			2.70E-3					high-recovery
	Flatness, 1.5-2.5 kHz	1.38E-2							all
	Flatness, 1.5-2.5 kHz			4.26E-2					high-recovery
	Harmonicity	0.01	7.0E-4	2.15E-2					all
	Harmonicity	6.70E-3	6.4E-3	1.63E-2					high-recovery
	Contrast, 1.6-3.2 kHz (LG)	4.08E-2	1.13E-2	1.5E-3					all
	Contrast, 1.6-3.2 kHz (LG)			3.4E-3					high-recovery
	Contrast, 3.2-6.4 kHz (LG)		2.90E-2						all
	Contrast, 3.2-6.4 kHz (LG)		2.92E-2						high-recovery
	VLHR (*)		1.86E-2		4.4E-3				all
	VLHR (*)								high-recovery
MFCC	MFCC 6				1.48E-2				all
	MFCC 6				4.46E-2				high-recovery
	MFCC 8						7.70E-3		all
	MFCC 8						1.08E-2		high-recovery
	MFCC 1 sd (LG)				4.49E-2			9.5E-3	all
	MFCC 1 sd (LG)				1.8E-2	4.37E-2			high-recovery
	MFCC 2 sd (LG)							1.17E-2	all
	MFCC 2 sd (LG)		2.27E-2		1.13E-2			3.38E-2	high-recovery
	MFCC 3 sd (LG)							4.11E-2	all
	MFCC 3 sd (LG)							3.97E-2	high-recovery
	MFCC 8 sd (LG)							2.12E-2	all
	MFCC 8 sd (LG)								high-recovery
	MFCC 9 sd (LG)			2.93E-2					all
	MFCC 9 sd (LG)		4.18E-2	2.71E-2					high-recovery
	MFCC 10 sd (LG)								all
	MFCC 10 sd (LG)			1.3E-3					high-recovery
	MFCC 11 sd (LG)						4.97E-2		all
	MFCC 11 sd (LG)								high-recovery
	MFCC 12 sd (LG)								all
	MFCC 12 sd (LG)		1.59E-2						high-recovery

FIG. 12B

SUBSTITUTE SHEET (RULE 26)



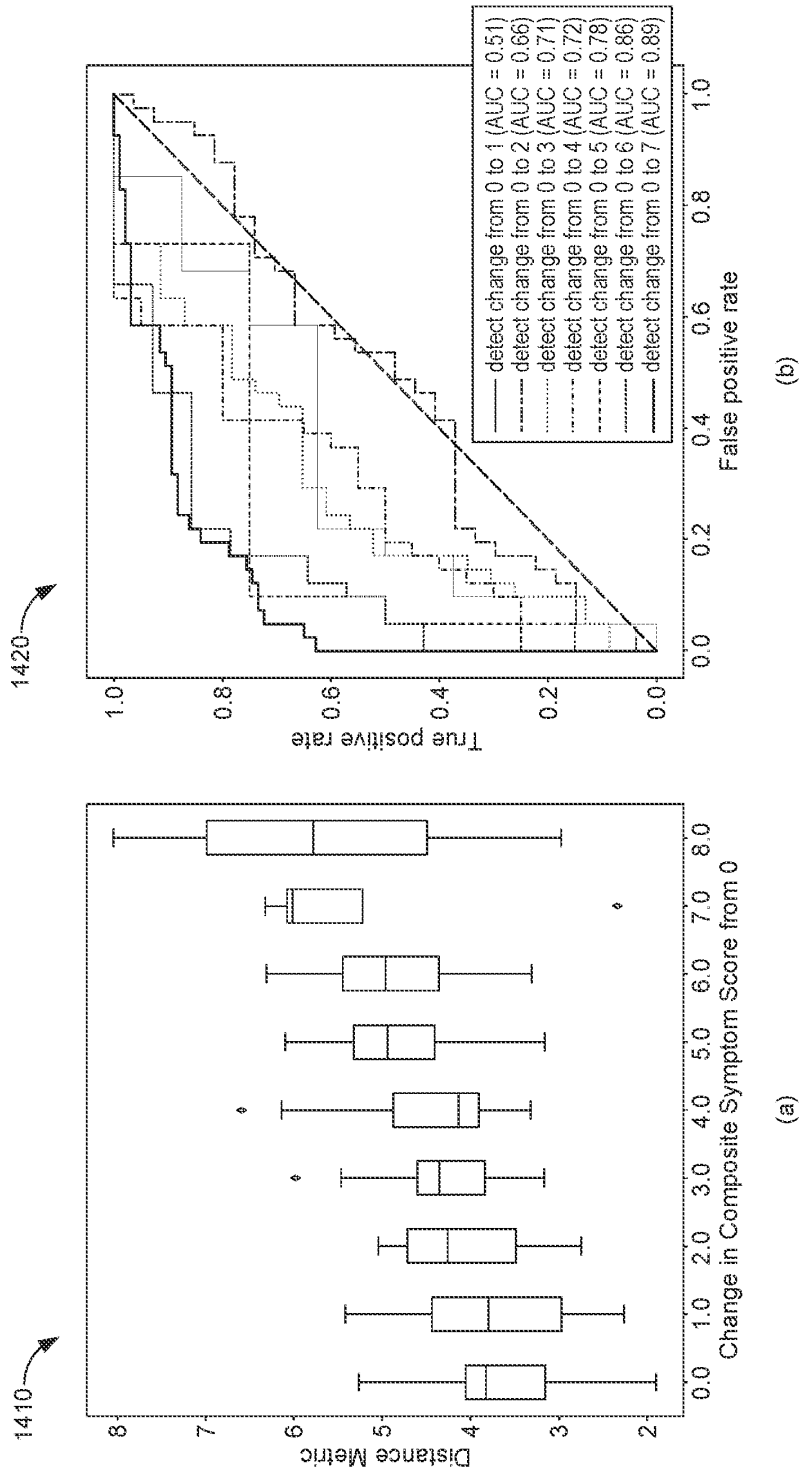


FIG. 14

28/41

```
#####
#
# PHONEME_VOCAL FEATURES (pre-alpha prototype)
#
#####

#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Prototype Only – Not For Distribution
"""
import os
import numpy as np
import scipy.signal
import librosa
import parselmouth # praat-parselmouth API (https://github.com/YannickJadoul/Parselmouth)
from parselmouth.praat import call # to make calls directly to Praat engine

def compute_power_vector(y, sampling_rate, window_size=0.25):
    """
    Compute the power time series of a signal.

    :param y: Audio data. (np.array)
    :param sampling_rate: Sampling rate of audio data. (int)
    :param window_size: Window size in seconds to compute mean over.
    :return: Power vector time series of audio data. (np.array)
    """
    y_duration = librosa.core.get_duration(y, sr=sampling_rate)
    time_indices = np.arange(0, y_duration, window_size)
    sample_indices = librosa.core.time_to_samples(time_indices, sr=sampling_rate)
    # Remove last point from start indices and then make copy of indices with lag of 1
    start_indices = sample_indices[:-1]
    end_indices = sample_indices[1:]
    # Zip together indices and take mean of each window
    power_vector = [np.mean(y[start_ix:end_ix]) for start_ix, end_ix in zip(start_indices, end_indices)]

    return power_vector

def band_pass_filter(y, filtOrder=6, fLo=100, fHi=4000, fSamp=44100):
    """
    Design a Butterworth bandpass filter and filter input signal y

    :param y: Audio data. (np array)
    :param filtOrder: Order of Butterworth filter. (int)
    :param fLo: low edge of passband, Hz. (int)
    :param fHi: high edge of passband, Hz. (int)
    :param fSamp: sampling rate, Hz. (int)
    :return: Filtered audio data. (np.array)
    """
    # use butterworth bandpass
    W1 = fLo / (fSamp / 2)
    W2 = fHi / (fSamp / 2)
    Wn = [W1, W2]
    b, a = scipy.signal.butter(filtOrder, Wn, btype='bandpass')

    filtered_y = scipy.signal.lfilter(b, a, y)

    return filtered_y
```

FIG. 15A.

CONTINUES IN FIG. 15B

29/41

CONTINUES FROM FIG. 15A

```

def tobSD200(y, sampling_rate):
    """
    Computes standard deviation and mean of power in 200 Hz third octave band.

    :param y: Audio data. (np.array)
    :param sampling_rate: Sampling rate of audio data. (int)
    :return: 200 Hz standard deviation and mean power. (float, float)
    """

    # filter coeffs for 200 Hz TOB at fSamp = 20000. Designed in Matlab code quickTOB200.m
    FS_RESAMPLED = 20000.0
    B_VEC = [0.150463235850018e-3, 0.0, -0.451389707550054e-3, 0.0, 0.451389707550054e-3, 0, 0, -
0.150463235850018e-3]
    A_VEC = [1.000, -5.075334750820790, 11.371686568135365, -14.269615060367908,
10.565209386202410, -4.381014625166177,
0.802032128639927]
    WINLEN_SEC = 0.06

    # make y a floating point column vector
    y = np.double(y)
    sz = np.shape(y) # col vector....
    if sz[0] == 1:
        y = y.T
        print("flipped y: new size")
        print(np.shape(y))

    # resample to 20,000 Hz
    # convert sampling frequencies to integer multiples of 100 Hz for resampling
    upFact = np rint(FS_RESAMPLED / 100)
    dnFact = np rint(sampling_rate / 100) # round to nearest integer
    y = scipy.signal.resample_poly(y, upFact, dnFact)

    b = np.asarray(B_VEC)
    a = np.asarray(A_VEC)

    # normalize power
    y = y - np.mean(y)
    y = y / max(np.abs(y))

    # step size for windows
    stp = np.int(WINLEN_SEC * FS_RESAMPLED)

    nsamp = np.size(y)
    nwin = np.int(nsamp / stp)

    window = scipy.signal.hamming(stp)

    p = np.zeros(nwin) # power in window for each time window

```

CONTINUES IN FIG. 15C

FIG. 15B.

30/41

CONTINUES FROM FIG. 15B

```

for k in range(nwin):
    i1 = k * stp
    i2 = i1 + stp
    i2 = min(i2, nsamp)

    y2 = y[np.arange(i1, i2)] * window

    # down-sample by 8 following definition
    y2ds = scipy.signal.decimate(y2, 2)
    y2ds = scipy.signal.decimate(y2ds, 2)
    y2ds = scipy.signal.decimate(y2ds, 2)

    y3 = scipy.signal.lfilter(b, a, y2ds)
    tmp = np.sum(y3**2) / len(y3)

    p[k] = 10 * np.log10(tmp)

# note: converting to dB *before* computing stats
sd200 = np.std(p)
mn200 = np.mean(p)

return (sd200, mn200)

def compute_powerspectrum(y, sampling_rate, window_length=0.1, time_step=0.05):
    """
    For purposes of prototyping: this function computes a power spectrum using Parselmouth's spectrogram
    method. It includes commented out code for multitaper spectrum estimation, which reduces noise but is
    very slow... this code may be called elsewhere in the code (e.g., outside of phoneme_vocal_features.py)
    but it may be cleaner to call it from inside this code.

    :param y: Audio data. (np.array)
    :param sampling_rate: Sampling rate of audio data. (int)
    :param window_length: length of FFT window in seconds. (float)
    :param time_step: Step between FFT windows, in seconds. Default value gives 50% overlap (float)
    :return:
        specDB: time-averaged spectrum in decibels
        fvec: frequency vector at for spectrum values
        specLin: time-averaged spectrum, linear scale
    """
    if np.all(np.isnan(y)) | (len(y)==0):
        return np.nan, np.nan, np.nan
    else:
        snd = parselmouth.Sound(y, sampling_frequency=sampling_rate)
        # praat default window lenght is 0.005 sec; 200 Hz bin width
        # VLHR paper says to use a 10 Hz bandwidth - OR ANYWAY LESS THAN 100
        spectrogram = snd.io_spectrogram(maximum_frequency=8e3, window_length=window_length,
        time_step=time_step)
        tvec, fvec = spectrogram.x_grid(), spectrogram.y_grid() # Y is freq vector
        sg_lin = np.abs(spectrogram.values)
        specLin = np.mean(sg_lin, axis=1)
        specDB = 10*np.log10(specLin+0.0000001)

    return(specDB, fvec, specLin)

```

FIG. 15C.

CONTINUES IN FIG. 15D

31/41

CONTINUES FROM FIG. 15C

```

def compute_rms(y):
    """
    Computes RMS of a waveform

    :param y: Audio data. (np.array)
    :return rms: Root mean square of audio data. (float)
    """
    rms = np.sqrt(np.mean(y**2))
    return rms

def compute_spectrum_subband_entropy(fvec, flo, fhi, spectrum):
    """
    Computes entropy of spectrum 'spect' between frequencies flo and fhi.

    :param fvec: frequency vector for spectrum estimate, in Hz
    :param flo: low end of frequency band of interest
    :param fhi: hi end of frequency band of interest
    :param spectrum: linear spectrum estimate, length of fvec
    :return: Computed entropy in band
    """
    if np.all(np.isnan(spectrum)):
        return np.nan
    else:
        ifrq = np.where( (fvec >= flo) & (fvec <= fhi))
        specNorm = spectrum[ifrq]/np.sum(spectrum[ifrq])
        entrop = -np.mean( specNorm * np.log2(specNorm+0.0000001) )

    return entrop

def compute_spectrum_subband_flatness(fvec, flo, fhi, spectrum):
    """
    Computes spectral flatness of spectrum 'spect' between frequencies flo and fhi

    :param fvec: frequency vector for spectrum estimate, in Hz (np.array)
    :param flo: low end of frequency band of interest. (int)
    :param fhi: hi end of frequency band of interest. (int)
    :param spectrum: linear spectrum estimate, length of fvec. (np.array)
    :return: Computed spectral flatness in band. (float)
    """
    if np.all(np.isnan(spectrum)):
        return np.nan
    else:
        ifrq = np.where( (fvec >= flo) & (fvec <= fhi))
        arithMean = np.mean(spectrum[ifrq])
        geomMean = np.exp(np.mean(np.log2(spectrum[ifrq])))

        spectralFlatness = geomMean / arithMean

    return spectralFlatness

```

CONTINUES IN FIG. 15E

FIG. 15D.

32/41

CONTINUES FROM FIG. 15D

```

def compute_harmonicity(y, sampling_rate):
    """
    Computes harmonicity (harmonics - to - noise ratio) using default Praat settings (this is wrapper for Praat).

    :param y: Audio data. (np.array)
    :param sampling_rate: Sampling rate of audio data. (int)
    :return: Average harmonicity over time. (float)
    """
    if np.all(np.isnan(y)):
        return np.nan
    else:
        snd = parselmouth.Sound(y, sampling_frequency=sampling_rate)
        hh = snd.to_harmonicity()
        A_mean = hh.values.T[hh.values.T>0].mean()
        return A_mean

def compute_pitch_stats(y, sampling_rate, gender):
    """
    Calculate pitch stats, including jitter.

    :param y: Audio data. (np.array)
    :param sampling_rate: Sampling rate of audio data. (int)
    :param gender: Sex of this subject (str)
    :return:
        mn: Mean pitch in Hz, cross-correlation method. (float)
        mnAlt: Mean pitch in Hz, auto-correlation method (if they match, higher confidence (float)
        sd: Standard deviation of pitch in Hz. (float)
        iqr: Interquartile range of pitch (more robust than st dev). (float)
        pr_localJitter: Praat local jitter estimate. (float)
        pr_localabsoluteJitter: Praat local abs jitter estimate. (float)
        pr_localShimmer: Praat local shimmer estimate. (float)
        pr_localdbShimmer: Praat local shimmer estimate, dB. (float)
        meanGlottalPP: Praat mean glottal pulse period. (float)
    """
    if np.all(np.isnan(y)):
        mn, mnAlt, sd, iqr, n = np.nan, np.nan, np.nan, np.nan, np.nan
        # TODO BT:
        jitter_abs, jitter_rel = np.nan, np.nan
        pr_localJitter, pr_localabsoluteJitter, pr_localShimmer, pr_localdbShimmer, meanGlottalPulsePeriod = \
            np.nan, np.nan, np.nan, np.nan, np.nan
    else:
        snd = parselmouth.Sound(y, sampling_frequency=sampling_rate)
        # AR: extract these attributes to class
        # BT: background noise is an issue in Samsung lab data, so limit the lower bound of pitch estimate (fl)
        # to avoid locking on to fan noise. Not an issue for Audacity, so use lower bound.

        if gender == 'Female':
            fl = 100
            fh = 500
        else: # 'Male'
            fl = 80
            fh = 500

```

FIG. 15E.

CONTINUES IN FIG. 15F

33/41

CONTINUES FROM FIG. 15E

```

try:
    # auto-correlation method (default)
    hh = snd.to_pitch(pitch_floor=floor, pitch_ceiling=fh)
    pitch_vec = hh.selected_array['frequency']
    # note, check below helps to ensure that pitch is not found exactly at floor... which would indicate
    # that pitch is likely actually below the specified floor
    mn = pitch_vec[pitch_vec > floor].mean()
    sd = pitch_vec[pitch_vec > floor].std()
    n = len(pitch_vec[pitch_vec > floor])
    iqr = np.diff(np.percentile(pitch_vec, [25, 75]))[0]

    # get jitter while we have pitch! Call praat directly!
    pointProcess = call(snd, "To PointProcess (periodic, cc)", floor, fh)
    pr_localJitter = call(pointProcess, "Get jitter (local)", 0, 0, 0.0001, 0.02, 1.3)
    pr_localabsoluteJitter = call(pointProcess, "Get jitter (local, absolute)", 0, 0, 0.0001, 0.02, 1.3)
    # could consider adding these features in the future...
    #rapJitter = call(pointProcess, "Get jitter (rap)", 0, 0, 0.0001, 0.02, 1.3)
    #ppq5Jitter = call(pointProcess, "Get jitter (ppq5)", 0, 0, 0.0001, 0.02, 1.3)
    #ddpJitter = call(pointProcess, "Get jitter (ddp)", 0, 0, 0.0001, 0.02, 1.3)
    pr_localShimmer = call([snd, pointProcess], "Get shimmer (local)", 0, 0, 0.0001, 0.02, 1.3, 1.6)
    pr_localdbShimmer = call([snd, pointProcess], "Get shimmer (local_dB)", 0, 0, 0.0001, 0.02, 1.3, 1.6)

    # mean glottal pulse period
    tmp = call(pointProcess, "To Matrix")
    meanGlottalPP = np.nanmean(np.diff(tmp.values))

    # collect some alternate pitch values; can check to see if estimates match
    hh2 = snd.to_pitch_cc(pitch_floor=floor, pitch_ceiling=fh)
    pitch_vec2 = hh2.selected_array['frequency'] # autocorrelation method (default)
    mnAlt = pitch_vec2.mean()

except:
    mn = mnAlt = sd = iqr = n = np.nan
    pr_localJitter, pr_localabsoluteJitter, pr_localShimmer, pr_localdbShimmer, meanGlottalPP = \
        np.nan, np.nan, np.nan, np.nan, np.nan

return mn, mnAlt, sd, iqr, pr_localJitter, pr_localabsoluteJitter, pr_localShimmer, pr_localdbShimmer,
meanGlottalPP

```

CONTINUES IN FIG. 15G

FIG. 15F.

34/41

CONTINUES FROM FIG. 15F

```

def compute_vlhr(y, sampling_rate, fcut=1000, time_step=0.05, window_length=0.1,
               frequency_step=5, minimum_frequency=65, maximum_frequency=8e3):
    """
    Computes VLHR (voiced low to high ratio)

    :param y: Audio data. (np.array)
    :param sampling_rate: Sampling rate of audio data. (int)
    :param fcut: Frequency that divides 'low' from 'high' freq range. (int)
    :param time_step: Time in seconds used to compute the underlying spectrogram. (float)
    :param window_length: Time in seconds used to compute the underlying spectrogram. (float)
    :param frequency_step: Time in seconds used to compute the underlying spectrogram. (int)
    :param minimum_frequency: Lower bound of low range, Hz defaults to 65 Hz, but can be set higher if fan
    noise is issue. (int)
    :param maximum_frequency: Upper bound of high range, Hz. (int)
    :return: Voiced low to high ratio. (float)
    """
    if np.all(np.isnan(y)):
        return np.nan
    else:
        snd = parselmouth.Sound(y, sampling_frequency=sampling_rate)

        try:
            spectrogram =
snd.to_spectrogram(time_step=time_step,window_length=window_length,frequency_step=frequency_step,
maximum_frequency=maximum_frequency)
        except parselmouth.PraatError as e: # Praat API Error
            # waveform is too short
            # raise Exception(e)
            VLHR_out = np.nan
        else:
            _, Y = spectrogram.x_grid(), spectrogram.y_grid()
            Y=Y[1:]

            indsLFP = np.logical_and( Y > minimum_frequency, Y < fcut)
            indsHFP = Y > fcut
            LFP = np.sum(spectrogram.values[indsLFP,:],axis=0)
            HFP = np.sum(spectrogram.values[indsHFP,:],axis=0)

            # compute ratio
            VLHR =[x/y for x, y in zip(LFP,HFP)]

            # average over time & log transform
            VLHR_out = 10*np.log10(np.array( VLHR).mean())

    return VLHR_out

```

CONTINUES IN FIG. 15H

FIG. 15G.

35/41

CONTINUES FROM FIG. 15G

```

def compute_spec_contrast(y, sampling_rate):
    """
    Computes spectral contrast metric for spectral complexity. Higher contrast is a likely cue that subject
    is congested. This example prototype function uses the librosa implementation.

    :param y: Audio data vector. (np.array)
    :param sampling_rate: Sampling rate of y. (int)
    :return: spectral contrast values in 3 freq bins (see librosa documentation link below)
    """
    if not sampling_rate:
        sampling_rate = 22050
    S = np.abs(librosa.stft(y))
    contrast = librosa.feature.spectral_contrast(S=S, sr=sampling_rate)
    # extract relevant octave bands - see https://musicinformationretrieval.com/spectral_features.html
    mnBin3 = np.mean(contrast[3, :])
    mnBin4 = np.mean(contrast[4, :])
    mnBin5 = np.mean(contrast[5, :])

    return mnBin3, mnBin4, mnBin5

def compute_mfcc(y, n_mfcc, sampling_rate, n_fft=2048, hop_length=512):
    """
    Extract Mel-frequency cepstral coefficient series on audio signal y through Librosa's implementation.
    librosa documentation: https://librosa.github.io/librosa/generated/librosa.feature.mfcc.html

    :param y: audio data vector. (np.array)
    :param n_mfcc: # coefficients to compute (int)
    :param n_fft: FFT length in samples (int)
    :param hop_length: FFT overlap in samples (float)
    :return: MFCC sequence (dictionary)
    """

    if np.all(np.isnan(y)):
        return np.nan
    mfcc_series = librosa.feature.mfcc(y, sr=sampling_rate, n_mfcc=n_mfcc, n_fft=n_fft, hop_length=
    hop_length)

    # compute mean and standard deviations for each coefficient
    mfcc_mean = np.mean(mfcc_series, axis=1)
    mfcc_sd = np.std(mfcc_series, axis=1)
    mfcc_dict = {'mfcc{}'.format(idx+1): val for idx, val in enumerate(mfcc_mean)}
    mfcc_dict.update({'mfcc{}'.format(idx+1): val for idx, val in enumerate(mfcc_sd)})

    return mfcc_dict

```

CONTINUES IN FIG. 15I

FIG. 15H.

36/41

CONTINUES FROM FIG. 15H.

```

def compute_formants(y, sampling_rate):
    """
    - Computes first three formants (prototype example uses parselmouth). These can be used to distinguish
    different vowels.

    :param y: Audio data. (np.array)
    :param sampling_rate: Sampling rate of audio data. (int)
    :return:
        F1mn, F2mn, F3mn: mean formant frequency in Hz for formants 1,2 and 3
        F1sd, F2sd, F3sd: standard deviation of formant frequency in Hz for formants 1,2 and 3
        F1bw, F2bw, F3bw: formant bandwidth in Hz for formants 1,2 and 3
    """
    snd = parselmouth.Sound(y, sampling_rate)
    frmt = snd.io_formant_burg() # reverting back to defaults time_step=0.0025)

    frmt_frq = np.zeros( (frmt.nt, 3) )
    frmt_bw = np.zeros( (frmt.nt, 3) )

    pptimes = frmt.xs()
    num_points = len(pptimes)

    for itm in range(num_points):
        for ifm in range(3):
            fn = ifm + 1 # formant numbers are ones-based
            frmt_frq[itm, ifm] = frmt.get_value_at_time(formant_number=fn, time=pptimes[itm])
            frmt_bw[itm, ifm] = frmt.get_bandwidth_at_time(formant_number=fn, time=pptimes[itm])

    mnFreq = np.nanmean(frmt_frq,axis=0)
    sdFreq = np.nanstd(frmt_frq, axis=0)
    mnBW = np.nanmean(frmt_bw,axis=0)
    sdBW = np.nanstd(frmt_bw, axis=0)

    tt = np.hstack([mnFreq, sdFreq, mnBW, sdBW])
    colnames = ['F1mn','F2mn','F3mn','F1sd','F2sd','F3sd','bwF1mn','bwF2mn','bwF3mn','bwF1sd','bwF2sd',
    'bwF3sd']
    formant_dict = dict(zip(colnames, tt))

    return formant_dict

```

CONTINUES IN FIG. 15J

FIG. 15I.

37/41

CONTINUES FROM FIG. 15I

```

def segment_audio_data(path_to_phoneme_recording, recording_duration, sampling_rate, offset_time,
transient_time, signal_duration):
    """
    :param path_to_phoneme_recording: File path to phoneme recording. (str)
    :param recording_duration: Length of phoneme recording. (float)
    :param sampling_rate: Sampling rate of recording. (int)
    :param offset_time: Time in seconds to truncate from each end to eliminate any non-speech. (float)
    :param transient_time: Additional time to remove from start of cleaned recording to remove transient
sounds. (float)
    :param signal_duration: Duration of phoneme recording to capture after removing transient sound from
the start. (float)
    :return: Segmented recording of length signal_duration which contains clean phoneme recording.
(np.array)
    """
    # further remove transient sound from start of recording and take min(len(trimmed_audio), 2.5s) of data
    segmented_recording_duration = recording_duration - 2 * offset_time
    phoneme_audio_data, _ = librosa.core.load(
        path=path_to_phoneme_recording,
        sr=sampling_rate,
        offset=offset_time,
        duration=segmented_recording_duration
    )
    filtered_phoneme_data = band_pass_filter(phoneme_audio_data)
    phoneme_duration = librosa.get_duration(filtered_phoneme_data, sr=sampling_rate)
    min_phoneme_duration = transient_time + signal_duration
    segment_start_idx = librosa.time_to_samples(transient_time, sr=sampling_rate)

    if phoneme_duration < min_phoneme_duration:
        segmented_phoneme_data = filtered_phoneme_data[segment_start_idx:]
    else:
        segment_end_idx = librosa.time_to_samples(min_phoneme_duration, sr=sampling_rate)
        segmented_phoneme_data = filtered_phoneme_data[segment_start_idx:segment_end_idx]

    return segmented_phoneme_data

def extract_vocal_features(audio_data, sampling_rate, gender, n_mfcc=20):
    """
    Computes vocal features to be extracted from audio recordings and merge into a single dictionary.

    :param audio_data: Segmented audio data. (np.array)
    :param sampling_rate: Sampling rate of recordings. (int)
    :param gender: Gender of subject - male or female. Used in calculating pitch features. (str).
    :param n_mfcc: Number of MFCC's to extract. (int)
    :param window_length: Window length for VLHR periodogram analysis, sec. (float)
    :return: A dictionary storing each voice feature.
    """
    features_dict = {}
    # truncated_audio_data = truncate_audio_data(audio_data, sampling_rate, buffer_time, offset, signal_
duration)

```

CONTINUES IN FIG. 15K

FIG. 15J.

38/41

CONTINUES FROM FIG. 15J

```

try:
    # first output was formerly labeled specLin1
    __, freq_vec, specLinL1 = compute_powerspectrum(audio_data, sampling_rate)
except parselmouth.PraatError as e:
    # waveform is too short
    print(e)
    print('Storing specLinL1 and freq_vec values as Nan.')
    freq_vec = np.nan
    specLinL1 = np.nan

# use rms to normalize wave forms
features_dict['rms'] = compute_rms(audio_data)
norm_audio_data = audio_data/features_dict['rms']
features_dict['tob200SD'], features_dict['tob200MB'] = tobSD200(norm_audio_data, sampling_rate)
features_dict['harm'] = compute_harmonicity(norm_audio_data, sampling_rate)
features_dict['pitchMn'], features_dict['cov_pitch'], features_dict['coviqr_pitch'], features_dict['pitchMnALT'], \
features_dict['jitter_pr'], features_dict['abs_jitter_pr'], \
features_dict['shimmer_pr'], features_dict['shimmer_db_pr'], features_dict['mean_glottal_pulse_period'] = \
    compute_pitch_stats(norm_audio_data, sampling_rate, gender)

# see DOI: 10.19080/GJO.2017.08.555737 for references to different definitions.
# Lee 2003 uses 4.47*pitch; LeeYankKuo 2003 uses 600 Hz cutoff
# factor of sqrt(5) = 4.47 is from Lee et al, 2003 VLHR paper
freq_cut = features_dict['pitchMn']*4.47
features_dict['VLHR'] = compute_vlhr(norm_audio_data, sampling_rate, fcut=freq_cut)
features_dict['VLHR600'] = compute_vlhr(norm_audio_data, sampling_rate, fcut=600.0)

specContrastMnBin3, specContrastMnBin4, specContrastMnBin5 = compute_spec_contrast(
    norm_audio_data, sampling_rate)
features_dict['specContrast800to1600'] = specContrastMnBin3
features_dict['specContrast1600to3200'] = specContrastMnBin4
features_dict['specContrast3200to6400'] = specContrastMnBin5

# lab visit data suggests 1500-2500 is best freq range for entropy; could be revisited
features_dict['entropy1500to2500'] = compute_spectrum_subband_entropy(freq_vec, 1500, 2500, specLinL1)
features_dict['entropy1600to3200'] = compute_spectrum_subband_entropy(freq_vec, 1600, 3200, specLinL1)
features_dict['flatness1500to2500'] = compute_spectrum_subband_flatness(freq_vec, 1500, 2500, specLinL1)

# Compute formant feature
formants_dict = compute_formants(norm_audio_data, sampling_rate)
features_dict.update(formants_dict)

mfcc_dict = compute_mfcc(norm_audio_data, n_mfcc, sampling_rate)
features_dict.update(mfcc_dict)

return features_dict

def check_formant_bounds(f1mn, phoneme_label):
    if phoneme_label == 'm':
        f1_is_out_of_bounds = f1mn < 0 or f1mn > 5000
    elif phoneme_label == 'o':
        f1_is_out_of_bounds = f1mn < 400 or f1mn > 950
    elif phoneme_label == 'e':
        f1_is_out_of_bounds = f1mn < 200 or f1mn > 400

    return f1_is_out_of_bounds

```

CONTINUES IN FIG. 15L

FIG. 15K.

39/41

CONTINUES FROM FIG. 15K

```

def check_phoneme_quality(phoneme_data, background_data, sampling_rate, f1mn,
                          phoneme_label, threshold_length=2, threshold_snr_linear=5):
    """
    :param phoneme_data: Segmented audio data. (np.array)
    :param background_data: Background sound sample data. (np.array)
    :param f1mn: mean frequency of first formant, Hz. (int)
    :param phoneme_label: Ground truth label of phoneme. (str)
    :param sampling_rate: Sampling rate of audio and background data. (int)
    :param threshold_length: Minimum desired signal length, sec. (int)
    :param threshold_snr_linear: minimum desired SNR, linear scale - not dB. (int)
    :return: Dictionary with booleans for short signal, low SNR, and formant 1 out of bounds.
    """
    audio_duration = librosa.get_duration(phoneme_data, sr=sampling_rate)
    speech_power_vector = compute_power_vector(phoneme_data, sampling_rate)
    background_power_vector = compute_power_vector(background_data, sampling_rate)
    est_snr = np.median(speech_power_vector) / np.median(background_power_vector)
    f1_is_out_bounds = check_formant_bounds(f1mn, phoneme_label)

    qc_dict = {
        'short_signal_length': audio_duration < threshold_length,
        'low_snr': est_snr < threshold_snr_linear,
        'f1_out_of_bounds': f1_is_out_bounds
    }

    return qc_dict

def segment_and_extract_features(path_to_phoneme_recording, path_to_background_sample,
                                phoneme_label, gender, sampling_rate=44100,
                                offset_time=0.25, transient_time=0.75, signal_duration=2.5, n_mfcc=13, **qc_kwargs):
    """
    :param path_to_phoneme_recording: File path to phoneme recording. (str)
    :param path_to_background_sample: File path to background noise segment. (str)
    :param phoneme_label: Ground truth label of phoneme. (str)
    :param gender: Gender of subject - 'Female' or 'Male'. (str)
    :param sampling_rate: Sampling rate of audio data. (int)
    :param offset_time: Time in seconds to truncate from each end to eliminate any non-speech. (float)
    :param transient_time: Additional time to remove from start of cleaned recording to remove transient
    sounds. (float)
    :param signal_duration: Duration of phoneme recording to capture after removing transient sound from
    the start. (float)
    :param n_mfcc: Number of MFCC coefficients to extract. (int)
    :param qc_kwargs: Additional key word arguments for check_quality_control()
        f1mn: mean frequency of first formant, Hz. (int)
        threshold_length: Minimum desired signal length, sec. (int)
        threshold_snr_linear: minimum desired SNR, linear scale - not dB. (int)
        bounds_formant: 2-element vector with upper, lower range of believable formant frequencies, Hz [int, int]
    :return: A dictionary with vocal features and a dictionary with QC booleans.
    """
    # minimum speech time needed in order to segment features and run pipeline
    min_recording_duration = offset_time*2 + transient_time
    audio_duration = librosa.get_duration(filename=path_to_phoneme_recording)
    assert audio_duration > min_recording_duration

```

FIG. 15L.

CONTINUES IN FIG. 15M

40/41

CONTINUES FROM FIG. 15L

```

phoneme_data = segment_audio_data(path_to_phoneme_recording, audio_duration, sampling_rate,
                                  offset_time, transient_time, signal_duration)
vocal_features = extract_vocal_features(phoneme_data, sampling_rate, gender, n_mfcc)
background_data, _ = librosa.core.load(path_to_background_sample, sr=sampling_rate)
qc_flags = check_phoneme_quality(phoneme_data, background_data, sampling_rate,
                                 vocal_features['F1mn'], phoneme_label, **qc_kwargs)

return vocal_features, qc_flags

# Entry point for running prototype code is below
if __name__ == '__main__':
    segmented_phonemes_dir = os.path.expanduser('~/.Downloads/example_voice_algorithm_files')

    # meta-data and extraction parameters
    CONFIG = {
        'path_to_segmented_phoneme': os.path.join(segmented_phonemes_dir, 'phoneme_example.wav'),
        'path_to_background_sample': os.path.join(segmented_phonemes_dir, 'phoneme_background_
example.wav'),
        'phoneme_label': 'm',
        'gender': 'Male',
    }

    features_dictionary, qc_dictionary = segment_and_extract_features(
        CONFIG['path_to_segmented_phoneme'],
        CONFIG['path_to_background_sample'],
        CONFIG['phoneme_label'],
        CONFIG['gender']
    )

    for key, value in features_dictionary.items():
        print(key, value)

    print(qc_dictionary)

```

FIG. 15M.

41/41

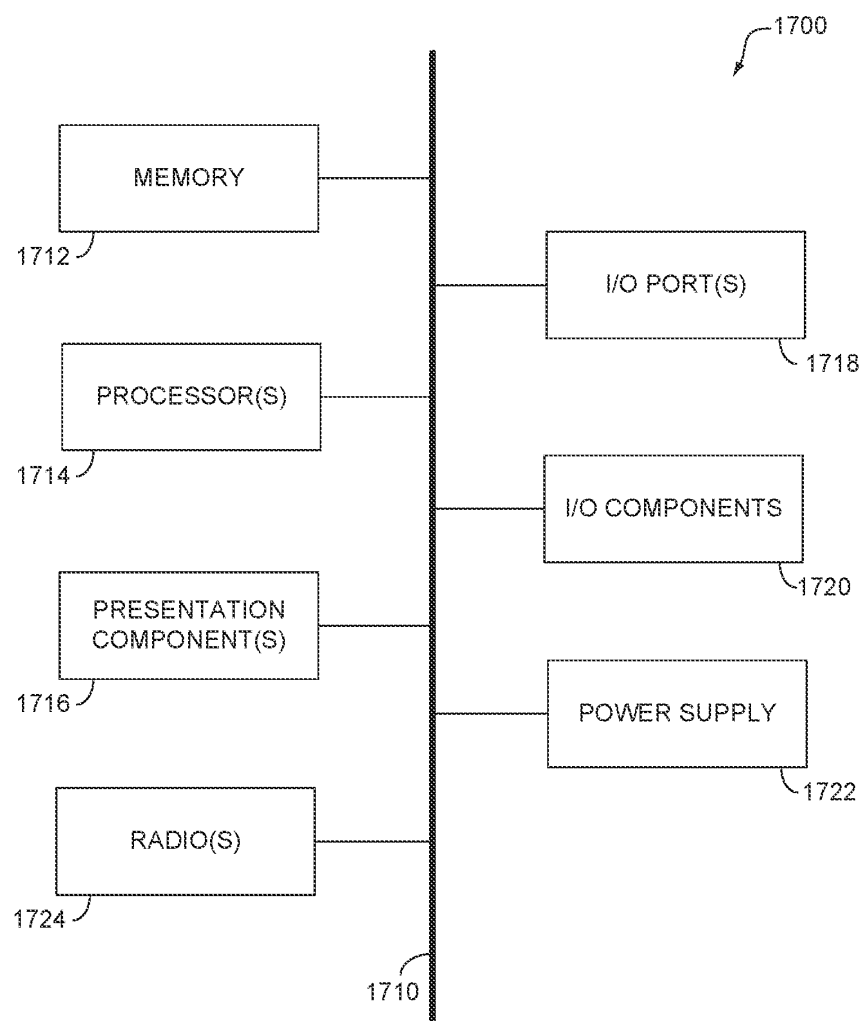


FIG. 16