APPLICATION NOTE

INTELLIGENT CALIBRATION

Table of Contents

1 Introduction

Bed type, person and various other factors affect the BCG measurement. To ensure good measurement quality, BCG product has to be adapted, i.e. calibrated to the use case at hand.

Calibration parameter definition can be done using the embedded calibration routine, or with an off-sensor routine based on measurement output data. The latter method is called adaptive calibration. Both methods require occupied and empty bed data and the automated adaptive process requires the capability to automatically separate these two from each other.

The method described in this document consists of two functional entities: the separation of BCG data to occupied bed and empty bed data sets (see figure 1, all functions in code other than `adaptive`) and the actual adaptive calibration parameter calculation (`adaptive`). The combined functionality – data separation and adaptive calibration – is called Intelligent Calibration.

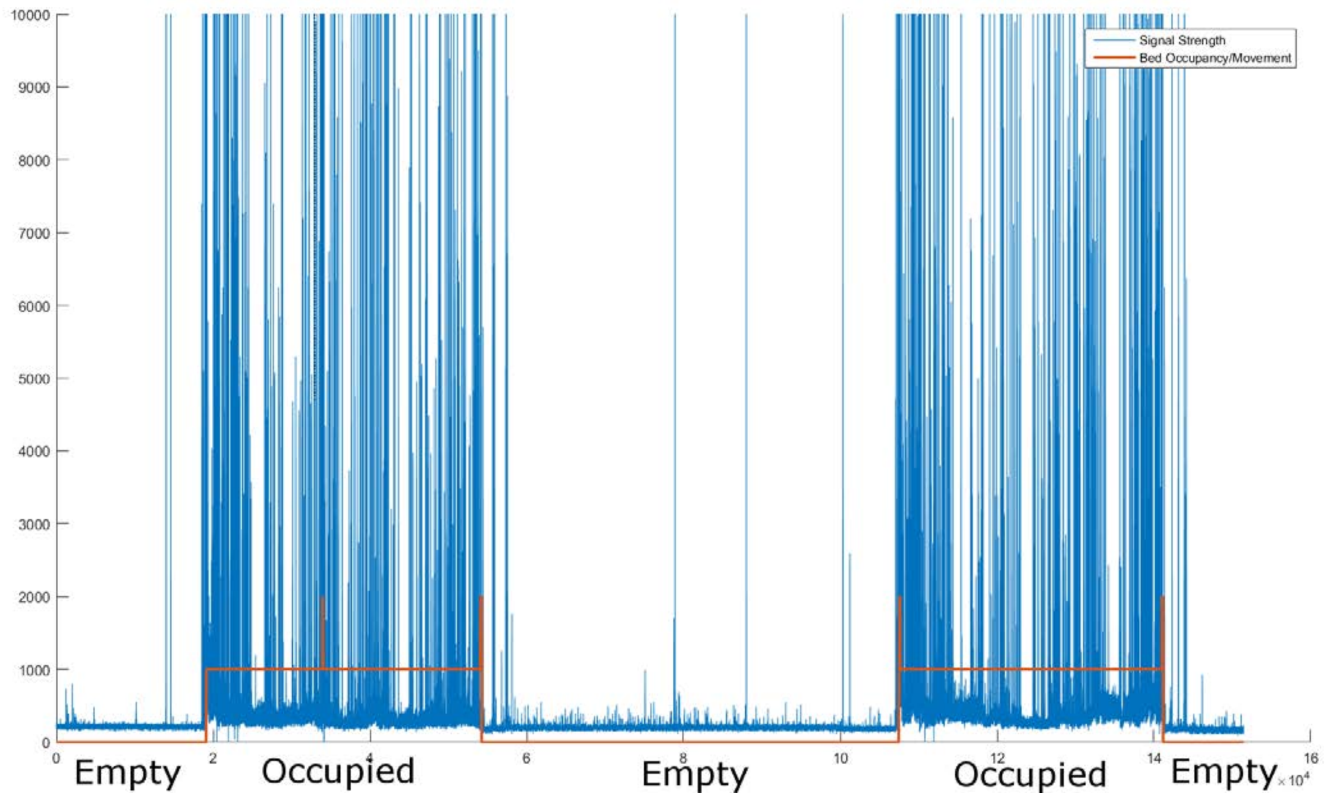

Figure 1: Data separation produced with Intelligent Calibration. Empty sections are used as empty bed data and occupied sections as occupied bed data in adaptive calibration. Peaks in the orange line are discarded as movement.

Upgrading calibration parameters naturally has an effect on the BCG output. Thus, it is recommended to repeat the calibration process ~3 times in order to obtain optimal calibration parameters.

2 Method

An example use case for the code described in this document is the following:
"A BCG sensor is installed with default calibration parameters on a bed of a long term patient in an elderly care home. Logged BCG data is processed with the Intelligent Calibration algorithm every day at 12 pm, after which the calibration parameters are upgraded to the BCG sensor using cloud OTA functionality. Calibration level is good after the first calibration interval, after which the next calibration cycles enhance the calibration quality further. Intelligent Calibration routine can be stopped typically after 3 days, although the calibrations could also be done daily indefinitely."

Requirements:

- input data includes <u>both</u> occupied bed and empty bed data <u>OR</u> input data includes <u>only</u> occupied bed data and the <u>empty bed mean signal strength</u> is used as an <u>input</u> parameter

- target minimum length for both occupied bed and empty bed data is <u>>60 minutes</u> each and it is recommended to use >6h data sets before concluding the calibration process.

Implications:

- automated calibration process control is external, i.e. the "run this code when it is known that logged data includes enough occupied bed data". Advice with use case specific automation based on BCG data can be given on request.

- the code does not analyze whether data actually includes occupied and empty bed data, so it may return poor parameters if, for example, only occupied bed data is input without defining the empty bed mean SS (signal strength) level.

The separation of the logged BCG data to occupied & empty bed datasets is based on signal strength and signal strength variability. The input data <u>must</u> include some occupied bed and some empty bed data. After separation, data is run through the adaptive calibration function in order to obtain calibration parameters (`adaptive`).

The data separation is based on analyzing how the level of "mean" (actually 10% percentile) signal strength and the standard deviation of signal strength varies in the data set.

The example code can also be used without data separation by inputting a value for mean empty bed signal strength (in the `init_config`). In this case, the program assumes there's only occupied bed data in the input file and runs `adaptive` based on the occupied bed data input, and the given empty bed mean SS.

The method is documented in a MATLAB code. This code is available as a MATLAB Coder-made port to C-language. The MATLAB code is commented in order to help understand the principles and the functionality of the machine ported C-code. The MATLAB code is included at the end of this application note and it can be supplied also as a .m file library at request. The C-code can as well be supplied on request. More details on the C port are given in chapter "C-implementation". Test cases and test summary can be supplied at request as well.

3 Basic functionality of the MATLAB code

The MATLAB code is usable as it is, as long as the surrounding main program is properly implemented and the configuration parameters are correctly set. This section gives a basic overview of the provided functions, and more detailed descriptions are given in the next section.

The core of the code is the `intelligentCalibration` function. It takes a set of BCG data, number of lines in this BCG data set, configuration parameters and the initialized output struct. The program outputs an output struct that contains the following parameters:

- Calibration parameters in format [var_level_1, var_level_2, stroke_vol, tentative_stroke_vol, signal_range, 7]
- Bed occupancy indices with 0 for empty, 1 for occupied and 2 for movement (for each line of input data).
- Empty bed mean signal strength that is calculated by `adaptive`
- Occupied bed 25% percentile of signal strength calculated by `adaptive`
- Occupied bed 25% percentile of stroke volume calculated by `adaptive`
- Error flags (calibration parameters are returned as 0,0,0,0,0,0 for any other error flag than 0)
    - 0 for no errors
    - 1 if length of supplied data is less than the configured interval length
    - 2 if no empty/occupied bed separation is found
    - 3 if configured empty mean level (.emptyLevel) is higher than found occupied level

In addition to these, the function outputs its input configs and input data.

The library consists of the following MATLAB functions (core function bolded):

```
runIntelligentCalibration (example main program)
algorithm_version
init_config
init_output
intelligentCalibration
findLimits
calculateRate
findValleys
analyzeBedOccupancy
percentile
probabilityCounter
interpolate_bedOccupancy
adaptive
```

Two debug plotting MATLAB scripts are as well available as extras (at the bottom of the supplied MATLAB code) that can be used to study the functionality of the code.

The example main program can be run with any BCG data file with the following format (no xml messages/excess line changes etc.):

```
1092,0,0,0,0,3932,2,0,0,0
1093,0,0,0,0,5477,2,0,0,0
1094,86,9,46,178,3040,1,3000,0,0
1095,86,9,46,178,3060,1,3000,0,0
1096,86,9,46,178,3935,2,2286,0,0
```

Data content:
(timestamp, HR, RR, SV, HRV, SS, status, b2b, b2b', b2b")

The basic functional flow of the code is the following: (the functions are described further below and the hierarchy of the program is presented in figure 1)

1. `runIntelligentCalibration:` The example main program: Read input file, set configurations and call the main function, `intelligentCalibration.` The main function handles the actual functionality described earlier and outputs the desired outputs of the function.

2. Signal strength (SS) is input into `findLimits` with both mean and std (standard deviation) options enabled. `findLimits` outputs std/meanProbabilities and std/meanSignal. `calculateRate` uses std/meanProbabilities and SS as inputs and outputs std/meanRate. `findValleys` then uses std/meanRate and SS inputs to calculate std/meanBestValley.

3. `analyzeBedOccupancy` then takes std/meanSignal, std/meanProbabilities and std/meanBestValley to calculate bedOccupancy indexes.

4. `interpolate_bedOccupancy` and code in the main function `intelligentCalibration` splits data into empty bed SS, occupied bed SS and occupied bed SV data arrays according to the bedOccupancy indexes.

5. Empty bed SS and occupied bed SS and SV data are input into `adaptive` that outputs the previously mentioned set of outputs. `intelligentCalibration` then returns these outputs, an error_flag and config and data arrays.

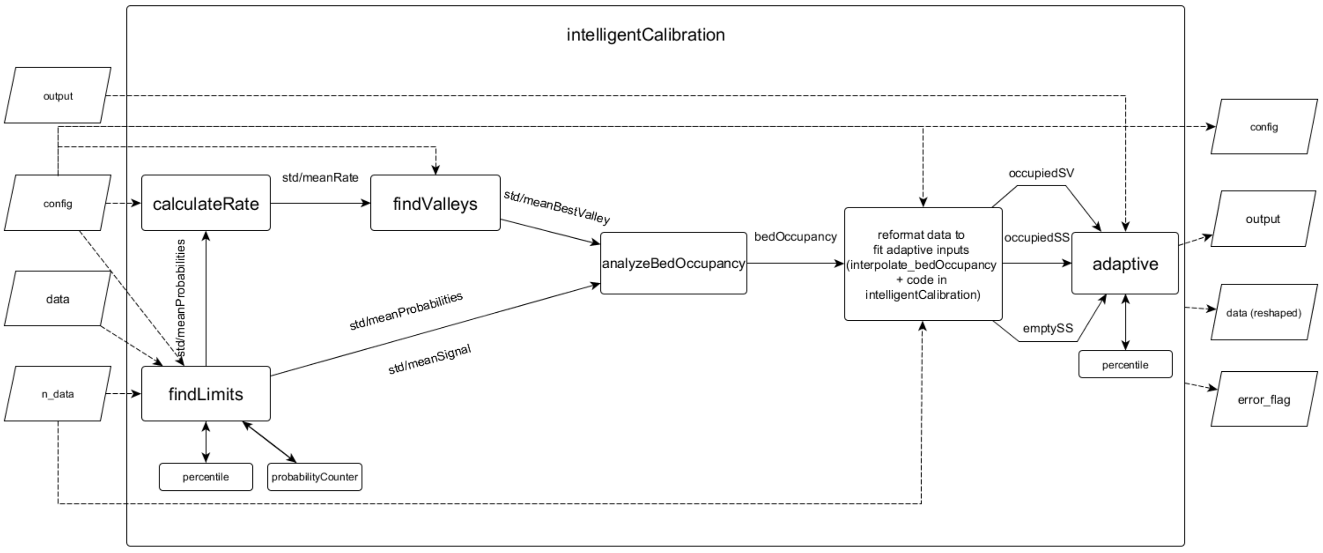

Figure 2: Flow chart visualizing the functionality of the Intelligent Calibration code

4 C-implementation

The MATLAB code described in this document is available ported to C using MATLAB Coder (all functions except `runIntelligentCalibration` and debug plot functions). An example main function and demo .exe (intelligent_calibration.exe) with reading capability of a randomly sized data file are provided for testing purposes.

5 Detailed description of MATLAB functions

The following sections intend to clarify the functionality of the MATLAB program by giving a detailed verbal description of the functionality of the components of the functions.

5.1 `algorithm_version`

A function for version control.

Input:
versionArray: array of size 3 for output initialization

Output:
versionArray: returns the current version in format [MAJOR,MINOR,PATCH] (see http://semver.org/)

5.2 init_output

Configures the struct required for outputs of the program. See `intelligentCalibration` for definitions of the outputs.

The output struct contains the following:
output.parameters = calibration parameter output, size 6
output.bedOccupancy = an array of size n_data for bedOccupancy indexes, can be used to plot & debug occupancy separation quality
output.emptySS = empty bed mean signal strength that is calculated by `adaptive`
output.occupiedSS = occupied bed 25% percentile of signal strength calculated by `adaptive`
output.occupiedSV = occupied bed 25% percentile of stroke volume calculated by `adaptive`

5.3 init_config

The configuration parameters used by the data separation. Default values can be used in most cases. The following configurable parameters are available:

The configurations inputs are the following:
config.emptyLevel = empty bed mean Signal Strength level. If this is input, the intelligentCalibration function assumes only occupied data is input in the function and calibrates accordingly.
config.interval = decides the intervals for which STD and percentile are calculated (default and recommended minimum 60 seconds). The shorter the interval, the more sensitive the separation is to signal strength variation.
config.meanLimits = the "resolution" of the mean level analysis, mean limit levels that are used to analyze the separation quality (i.e. "how well the data separates at a level of mean signal strength = 500 or 510 etc.?") Default 51 steps between 0 and 1000.
config.nMeanLimits = number of mean limit levels;
config.stdLimits = the "resolution" of the std level analysis, limit levels that are used to analyze the separation quality (i.e. "how well the data separates at level of std of signal strength = 50 or 55 etc.?"). Default 41 steps between 0 and 200.
config.nStdLimits = number of standard deviation limit levels

5.4 intelligentCalibration

The main function of the program. Calls the other functions in the library in order to calculate best occupied/empty bed separation and runs adaptive calibration with the separated data.

Inputs:
data: the BCG data in a [n_data, 10] matrix (file reading must be done outside of the function)
n_data: number of lines of BCG data in the input matrix
config: function configuration parameters, initialized with `init_config`
parameters: an empty array of size 6 for output parameter memory handling

Outputs:
output.parameters = calibration parameter output, format [var_level_1, var_level_2, stroke_vol, tentative_stroke_vol, signal_range, 7]
output.bedOccupancy = bedOccupancy indexes, 1 for occupied, 0 for empty and 2 for movement. Can be used to plot & debug occupancy separation quality
output.emptySS = empty bed mean signal strength that is calculated by `adaptive`
output.occupiedSS = occupied bed 25% percentile of signal strength calculated by `adaptive`
output.occupiedSV = occupied bed 25% percentile of stroke volume calculated by `adaptive`
error_flag: error flag that is returned on various occasions, where operation of the program is incorrect. See error flags in *Basic Functionality of the MATLAB code chapter* for detailed descriptions.
data: outputs input data reformatted using MATLAB reshape()
config: the configs that are described in `init_config`

5.5 findLimits

Calculates the probability of each point of data for being "over" the value of configured limits at that certain point. Outputs the probabilities at different limits in addition to the actual std's and means (10% percentiles) for each interval of data.

Inputs:
data: see `intelligentCalibration`
n_data: see `intelligentCalibration`
limits: limits chosen in config. Defines the "levels" at which separation is tested and evaluated (lines on the plots). size nLimits
nLimits: number of limits
interval: defines the length of the interval from which std and mean are calculated (60 seconds interval = std and mean are calculated for every 60 lines of data)
option: std or mean
signal: formatting input for output array, needs to be of size n_data/interval
probabilities: formatting input for output matrix, needs to be of size [n_data/interval, nLimits]

Outputs:
std/meanSignal: outputs mean (10% percentile) or std for every "interval" of data (size n_data/interval)
std/meanProbabilities: outputs probability of being "occupied" or "empty" for each interval at each level of "limits"

5.6 `calculateRate`

This function calculates a sum of probabilities for each "limit" level of data. This sum describes the amount of separation at certain limit and is further evaluated in `findValleys`.

Inputs:
probabilities: output of `findLimits`
limits: limits array
rate: output initialization matrix (should be size [2, nLimits]

Outputs:
std/meanRate: matrix of [2, nLimits] size, 1st row is input limits and 2nd row is according "rate" (sum of squares of probabilities at that limit)

5.7 `findValleys`

This function analyzes the values of rate at different limits and finds the limit at which the separation is best; i.e. the rate has the biggest local minimum.

Inputs:
rate: output of `calculateRate`
limits: limits array of size nLimits
bestValley: initialization for output, size 1

Outputs:
std/meanbestValley: the index of the rate at which best separation is achieved (practically the index of the limit level that achieves the best separation)

5.8 `analyzeBedOccupancy`

The function analyzes its Signal and Probabilities inputs at the limit level of BestValley in order to define bed status for each index of Signal (each interval of data). It outputs a set of indices for the Signal array with 0 meaning empty bed, 1 occupied bed and 2 movement bed status.

Inputs:
std&meanSignal: output of `findLimits`
std&meanProbabilities (at level of std/meanBestValley): output of `findLimits`

nProbs/nSignal: number of elements in the std&meanSignal arrays, see output of `findLimits`
bedOccupancy: initialization array for bedOccupancy, size nSignal

Outputs:
bedOccupancy: output that has 1 on indexes that are deemed occupied bed,0 for empty bed and 2 for "movement" data which is discarded from adaptive calibration. These match the Signal indexes (there's an index for each interval of data).

5.9 interpolate_bedOccupancy (& lines of code after this in intelligentCalibration)

This function extrapolates the indices of bedOccupancy array to each point in the data matrix (not just for each interval). Indexes in this format can then be used to separate occupied and empty bed datas to arrays in the main function for use in adaptive calibration calculation (occupiedBedSS, occupiedBedSV, emptyBedSS).

Inputs:
output.bedOccupancy: initialization of output bed indices (size n_data)
n_data: number of data lines in total
bedOccupancy: bed occupancy indices in the output format of bedOccupancy
config: config struct

Outputs:
output.bedOccupancy: bedOccupancy indicators 0, 1 or 2 for all lines of input data

5.10 percentile

Calculates the percentile from given data. Sorts data from smallest to biggest, and takes the value that is at "percentile" index in the array, i.e. 50% percentile would return the mean of the given data set. (see https://en.wikipedia.org/wiki/Percentile)

Inputs:
data: set of data from which percentile is to be returned
percentile: decides which index of the input array is returned (0.5 for mean value)

Outputs:
percentile: the percentile value of the given data set (a single value)

5.11 probabilityCounter

Calculates the probability of a given value for being occupied or empty bed. If value between lower and upperLimit, probability is simply how far the value is from lowerLimit.

Inputs:
value: value for which probability is to be returned
lowerLimit: lower boundary ("empty bed")
upperLimit: upper boundary ("occupied bed")

Outputs:
probability: 1 for occupied bed, 0 for empty bed, something in between for values between the given limits.

5.12 adaptive

The function that calculates the calibration parameters based on the input data.

Inputs:
occupiedBedSS: Signal strengths of input data that are deemed occupied bed by the separation (size [n_data x 1])
occupiedBedSV: Stroke volumes of input data that are deemed occupied bed by the separation (size [n_data x 1])
emptyBedSS: Signal strengths of input data that are deemed empty bed by the separation (size [n_data x 1])
parameters: initialization array for output calibration parameters (size 6)

Outputs:
parameters = calibration parameter output, format [var_level_1, var_level_2, stroke_vol, tentative_stroke_vol, signal_range, 7]
emptyLevel = empty bed mean signal strength that is calculated by `adaptive`
calcSS = occupied bed 25% percentile of signal strength calculated by `adaptive`
calcSV = occupied bed 25% percentile of stroke volume calculated by `adaptive`

5.13 DEBUG plot scripts (extras)

The MATLAB code has plot functionality for visualizing the data separation. Plots can be used by uncommenting all DEBUG variables from main function, and running the two scripts that are listed at the end of the example code in this document (debug_plotLimitAndRate.m and debug_plotOccupancyResults.m).

6 Example MATLAB code (version 1.0.0)

SEPARATE EACH FUNCTION TO ITS OWN .m FILE
(functions split by %%%% in this document, name files as (<function_name>.m) in a single folder and run runIntelligentCalibration.m)

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


% THIS SOFTWARE IS PROVIDED BY MURATA "AS IS" AND
% ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
% LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
% FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
% MURATA BE LIABLE FOR ANY DIRECT, INDIRECT,
% INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
% (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
% SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
% HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
% STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING
% IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
% POSSIBILITY OF SUCH DAMAGE.



%%%%runIntelligentCalibration (example main)
% clear; clc;

filename = 'sample.txt';

data = load(filename);

n_data = size(data,1);

EMPTYLEVEL = 0;
INTERVAL = 60;
STD_START = 0;
STD_STOP = 200;
STD_LENGTH = 41;
MEAN_START = 0;
MEAN_STOP = 1000;
MEAN_LENGTH = 51;

config = init_config(EMPTYLEVEL, INTERVAL,...
    STD_START, STD_STOP, STD_LENGTH,...
    MEAN_START, MEAN_STOP, MEAN_LENGTH);

[ output ] = init_output( n_data );

[error_flag, output, data, config] = intelligentCalibration(data, n_data , config, output);

fprintf('Algorithm Version: %u.%u.%u\n', algorithm_version( [0,0,0 ] ))
fprintf('Parameters: %u, %u, %u, %u, %u, %u\n',output.parameters)
fprintf('Error flag: %u\n', error_flag);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


function [ versionArray ] = algorithm_version( versionArray )
% ALGORITHM_VERSION returns the algorithm version.
%   The ALGORITHM_VERSION return algorithm version in array.
%               The version array has the following structure: MAJOR.MINOR.PATCH. See
%               more information at http://semver.org/


MAJOR = 1;
MINOR = 0;
PATCH = 0;

versionArray(:) = [MAJOR, MINOR, PATCH];

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
function [config] = init_config(empty_level, interval,...
    std_start, std_stop, std_length,...
    mean_start, mean_stop, mean_length)

config.emptyLevel = empty_level;
config.interval = interval; % Splits the data by interval and calculates "mean"/"std" from those
config.nStdLimits = std_length;
config.stdLimits = linspace(std_start, std_stop, std_length);
config.nMeanLimits = mean_length;
config.meanLimits = linspace(mean_start, mean_stop, mean_length);
end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


function [ output ] = init_output( n_data )

output.parameters = zeros(6,1);
output.bedOccupancy = ones(n_data,1);
output.emptySS = 0;
output.occupiedSS = 0;
output.occupiedSV = 0;
end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


function [error_flag, output, data, config] = intelligentCalibration(data, n_data, config, output)
%% findBedOccupancyLevels Distinguis empty/occupied bed and calibrate
% Analyzes given data for empty/occupied bed data and adaptively calibrates
% it. Data must have both empty bed and occupied bed data as well as there
% must be a signal, not only noise. Does not expect input data to be
% calibrated, but for best calibration result some beats should be found in
% the data (i.e. use low enough parameters when recording data to ensure
% beats will be found, no matter if beats are "found" in empty bed also).
% This example reads input data from a file but only signal strength and stroke
% volume data is required.
%

%% UNCOMMENT BELOW FOR DEBUG PLOTS
% global DEBUG

if n_data < config.interval
    error_flag = 1;
    output.parameters(:) = 0;
    return;
end

error_flag = 0;

if config.emptyLevel == 0
    %% Init
    nSignal = floor(n_data / config.interval);
    stdSignal = zeros(nSignal,1);
    stdProbabilities = zeros(nSignal,config.nStdLimits);
    stdRate = zeros(config.nStdLimits,2);
    stdBestValley = zeros(1,3);
    meanSignal = zeros(nSignal,1);
    meanProbabilities = zeros(nSignal,config.nMeanLimits);
    meanRate = zeros(config.nMeanLimits,2);
    meanBestValley = zeros(1,3);
    bedOccupancy = zeros(nSignal,1);

    %% Analyze signal strength
    % Searches for best separation between empty and occupied beds from "mean"
    % and "std" of signal strength. Note "mean" is actually 10% percentile (50%
    % percentile = median) and "std" is simplified calculation of rms.

    [stdSignal, stdProbabilities] = findLimits(data(:,6), n_data, config.stdLimits, config.nStdLimits,...
        config.interval, 'std', stdSignal, stdProbabilities);
    [stdRate] = calculateRate(stdProbabilities, config.stdLimits, stdRate);
    [stdBestValley] = findValleys(stdRate,config.nStdLimits,stdBestValley);
    [meanSignal, meanProbabilities] = findLimits(data(:,6), n_data, config.meanLimits, config.nMeanLimits,...
        config.interval, 'mean', meanSignal, meanProbabilities);
    [meanRate] = calculateRate(meanProbabilities, config.meanLimits, meanRate);
    [meanBestValley] = findValleys(meanRate, config.nMeanLimits, meanBestValley);

    %% Analyze bed occupancy and gather occupied/empty data
    % Analyze bed occupancy based on best empty and occupied bed separation
    [bedOccupancy] = analyzeBedOccupancy(stdSignal,...
        stdProbabilities(:,stdBestValley(1,3)), ...
        meanSignal,...
        meanProbabilities(:,meanBestValley(1,3)),...
        nSignal,...
        bedOccupancy);

    output.bedOccupancy = interpolate_bedOccupancy(bedOccupancy, output.bedOccupancy, n_data, config);

    occupiedBedSS = data(output.bedOccupancy == 1,6);     % gather signal strength for occupied bed.
    occupiedBedSV = data(output.bedOccupancy == 1,4);     % gather stroke volume for occupied bed.
    emptyBedSS = data(output.bedOccupancy == 0,6);       % gather signal strength for empty bed.

    % Note: Movement is not used here (bedOccupancy == 2)
    if isempty(emptyBedSS) || isempty(occupiedBedSS) || isempty(occupiedBedSV)
        error_flag = 2;
        output.parameters(:) = 0;
        return;
    end

    %% Adaptively calibrate for both empty and occupied bed
   [ output.parameters, output.occupiedSS, output.occupiedSV, output.emptySS ] =...
       adaptive(occupiedBedSS, occupiedBedSV, emptyBedSS, output.parameters);
        %% DEBUG, UNCOMMENT BELOW FOR DEBUG PLOTTING
%         DEBUG.stdSignal = stdSignal;
%         DEBUG.stdProbabilities = stdProbabilities;
%         DEBUG.stdRate = stdRate;
%         DEBUG.stdBestValley = stdBestValley;
%         DEBUG.meanSignal = meanSignal;
%         DEBUG.meanProbabilities = meanProbabilities;
%         DEBUG.meanRate = meanRate;
%         DEBUG.meanBestValley = meanBestValley;
%         DEBUG.bedOccupancy = output.bedOccupancy;
%         DEBUG.stdLimits = config.stdLimits;
%         DEBUG.meanLimits = config.meanLimits;
else
    %% Adaptive calibration if data given is only for occupied bed
   [ output.parameters, output.occupiedSS, output.occupiedSV, output.emptySS ] =...
       adaptive(data(:,6),data(:,4),config.emptyLevel, output.parameters);
end
if output.emptySS >= output.occupiedSS
    output.parameters(:) = 0;
    error_flag = 3;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [signal, probabilities] = findLimits(data, n_data, limits, nLimits,...
    interval, option, signal, probabilities)
switch option
    case 'mean'
    case 'std'
    otherwise
        Error('Invalid option in findLimits.');
end

%% Init
iSignal = 1;
probAlpha = 1/2;                % alpha of exponential low pass filter used to filter probability changes
buffer = zeros(interval,1);     % circular buffer
bufferIndex = 1;                % index for circular buffer

%% Analyze
for iData = 1: n_data
    % circular buffer
    buffer(bufferIndex,1) = data(iData);
    bufferIndex = bufferIndex + 1;
    if bufferIndex > interval
        bufferIndex = 1;
    end
    % check if interval amount of data gathered (remainder)
    if rem(iData,interval) == 0
        switch option
            case 'mean'
                value = percentile(buffer,0.1);
                limitDistance = 10;     % Used to calculate probability,
                % above limit + limitDistance probability is 1, below 0
            case 'std'
```

```
                value = sum(abs(diff(buffer)))/interval; % simple "std" / "power"
                limitDistance = 2;
        end
        for iLimits = 1: nLimits
            lowerLimit = limits(iLimits) - limitDistance; % below probability 0
            upperLimit = limits(iLimits) + limitDistance; % above probability 1
            probTmp = probabilityCounter(value,lowerLimit,upperLimit);
            % exponential low pass filter, for first value use probability
            % directly
            if iSignal == 1
                [probabilities(iSignal,iLimits)] = probTmp;
            else
                [probabilities(iSignal,iLimits)] = probAlpha .* probTmp + (1 - probAlpha) .* probabilities(iSignal-
1,iLimits);
            end
        end
        signal(iSignal,1) = value;
        iSignal = iSignal + 1;
    end
end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


function [rate] = calculateRate(probabilities, limits, rate)
% calculateGoodness Calculates rate from probabilities
% Calculates rate from probabilities. Rate indicates how good empty bed /
% occupied bed is distinguished. BUT for example, only empty bed would
% result rate of best possible. Used together with findValleys.m to analyze
% best possible separating values for empty/occupied bed.

% In probabilities [0...1], signal is good when signal is close to either 0 or 1
% In probs [-0.5...0.5], signal is best at 0 and worse the further away it is
% --> rate is sum of probs squared

probs = probabilities;
probs(probabilities > 0.5) = probs(probabilities > 0.5) - 1;
rate(:,1) = limits;
rate(:,2) = sum(probs.^2);
end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


function [bestValley] = findValleys(rate, nRate, bestValley)

minValue = 9999;    % Initialize as higher than possible min
maxValue = -9999;   % Initialize as lower than possible max
indexValley = 1;
indexPeak = 1;
valleys = zeros(nRate,2); % Cannot be larger in any case
peaks = zeros(nRate,2);   % Cannot be larger in any case

for i=2:nRate-1
    % find peaks
    if rate(i,2) > maxValue
        peaks(indexPeak,1) = rate(i,1);
        peaks(indexPeak,2) = rate(i,2);
        maxValue = rate(i,2);
    elseif (rate(i,2) > rate(i-1,2)) || (rate(i,2) < rate(i+1,2))
        indexPeak = indexPeak + 1;
        maxValue = -9999;
    end
    % find valleys
    if rate(i,2) < minValue
        valleys(indexValley,1) = rate(i,1);
        valleys(indexValley,2) = rate(i,2);
        minValue = rate(i,2);
    elseif (rate(i,2) < rate(i-1,2)) || (rate(i,2) > rate(i+1,2))
        indexValley = indexValley + 1;
        minValue = 9999;
    end
end

% Combine and sort peaks and valleys based on rate(:,1) i.e. sort by
% limits
nPeaksAndValleys = indexPeak + indexValley;
peaksAndValleys = zeros(nPeaksAndValleys,3);
```

```
peaksAndValleys(:,1:2) = [peaks(1:indexPeak,:);valleys(1:indexValley,:)];
peaksAndValleys = sortrows(peaksAndValleys,1);
nPeaksAndValleys = size(peaksAndValleys,1);

if indexValley < 2
    % If there is no valleys
%     warning(['findValleys: no valleys found, make sure ',...
%             '1) Limits configuration are sensible. ',...
%             '2) there is both empty and occupied bed in the input data. ',...
%             '3) the signal is good (i.e. location).']);
   bestValley(1:2) = peaksAndValleys(1,1:2);
elseif nPeaksAndValleys < 3
    % If there is no valleys
%     warning(['findValleys: no good valleys found, make sure ',...
%             '1) Limits configuration are sensible. ',...
%             '2) there is both empty and occupied bed in the input data. ',...
%             '3) the signal is good (i.e. location).']);
    iBestValley = nPeaksAndValleys;
    for iBestValley = nPeaksAndValleys:-1:1
        if peaksAndValleys(iBestValley,1) ~= 0 && peaksAndValleys(iBestValley,2)
            bestValley(1:2) = peaksAndValleys(iBestValley,1:2);
        end
    end
else
    % Calculate value for how "large" a valley is (and peaks). "Better" the
    % valley is the larger the calculated value is. Peaks should be
    % negative.
    % Adds (subtracts) difference between peak/valley n-1 & peak/valley n,
    % and n & n+1 --> valleys are positive, peaks negative, larger the
    % differences "better" the valley is (more positive)
    peaksAndValleys(2:end-1,3) = diff(peaksAndValleys(2:end,2)) -...
        2.*diff(peaksAndValleys(1:end-1,2));
    peaksAndValleys(:,3) = peaksAndValleys(:,3) .* ((rate(end,1)-peaksAndValleys(:,1))./100);
    peaksAndValleys = sortrows(peaksAndValleys,3);

    iBestValley = nPeaksAndValleys;
    % Find first non-zero valley calculated backwards (i.e. largest number)
    for iBestValley = nPeaksAndValleys:-1:1
        if peaksAndValleys(iBestValley,1) ~= 0 && peaksAndValleys(iBestValley,2)
            break;
        end
    end

    % If we don't find and peaks/valleys something is probably wrong with
    % the data.
    if peaksAndValleys(iBestValley,3) <= 0
        bestValley(1:2) = peaksAndValleys(1,1:2);
%         warning(['findValleys: no good valleys found, make sure ',...
%             '1) Limits configuration are sensible ',...
%             '2) there is both empty and occupied bed in the input data. ',...
%             '3) the signal is good (i.e. location).']);
    else
        bestValley(1:2) = peaksAndValleys(iBestValley,1:2);
    end
end

% Find the index of best valley
if bestValley(1,1) ~= 0
    tmp = find(rate(:,1) == bestValley(1,1));
    bestValley(1,3) = tmp(1);
else
    bestValley(1,3) = 1;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [bedOccupancy] = analyzeBedOccupancy(stdSignal, stdProbabilities, meanSignal, meanProbabilities, nProbs,
bedOccupancy)
% analyzeBedOccupancy Analyzes bed occupancy
% Analyzes bed occupancy from given std and mean of signal strength and
% probabilities

% Average probability
sumProbabilities = (stdProbabilities + meanProbabilities) / 2;

% Estimate movement level and set movement to probability 2
IndexSimpleOccupancy = sumProbabilities > 0.5;
meanStd = mean(stdSignal(IndexSimpleOccupancy,1));
fourthMean = percentile(meanSignal(IndexSimpleOccupancy,1),0.25);
```

```
movementLimit = floor(fourthMean + 2 * meanStd);
sumProbabilities(meanSignal(:,1) > movementLimit) = 2;

% Analyze first interval
if sumProbabilities(1) ~= 2
    bedOccupancy(1) = sumProbabilities(1) > 0.5;
else
    bedOccupancy(1) = sumProbabilities(1);
end

% Analyze bed occupancy
iProbs = 2;
for iProbs = 2 : nProbs - 1
    if sumProbabilities(iProbs) == 2
%         % If current is movement, use it as occupancy
        bedOccupancy(iProbs) = sumProbabilities(iProbs);
    else
        if stdProbabilities(iProbs) > 0.5 && meanProbabilities(iProbs) > 0.5
            bedOccupancy(iProbs) = 1;
        elseif stdProbabilities(iProbs) < 0.5 && meanProbabilities(iProbs) < 0.5
            bedOccupancy(iProbs) = 0;
        else
            bedOccupancy(iProbs) = bedOccupancy(iProbs-1);
        end
    end
end

% Handle rest of data
bedOccupancy(iProbs+1:end) = bedOccupancy(iProbs);


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


function [ percentileValue ] = percentile(data, percent)
%PERCENTILE Calculates percentile from input array
%   Calculates percentile from input array at specific percentile. Input
%   parameters: array, percent (0...1). Floors to closest percentile value
%   (e.q. row 11.74 --> get value in row 11)

percentileValue = zeros(1,1);

if isempty(data)
%     warning('Percentile: no data')
    percentileValue = 0;
else
    dataSorted = sort(data);
    percentIndex = floor(length(dataSorted)*percent);
    if percentIndex == 0
        percentIndex = 1;
    end
    percentileValue = dataSorted(percentIndex);
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [probability] = probabilityCounter(value,lowerLimit,upperLimit)
% probabilityCounter Calculates probability of empty...occupied bed [0...1]
% Calculates probability of empty...occupied bed [0...1]. Values above
% upperLimit will have probability of 1 and below lowerLimit probability of
% 0.

probability = zeros(1,1);

if value >= upperLimit
    probability = 1;
elseif value <= lowerLimit
    probability = 0;
else
    probability = (value-lowerLimit) * (1/(upperLimit-lowerLimit)); % y = bx
end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function bedOccupancy_full = interpolate_bedOccupancy(bedOccupancy, bedOccupancy_full, n_data, config)
% "interpolate" occupancy from [1,61...,n] to [1,2,3...,n]
x_interp = [1:n_data]';
x_tmp = [1:config.interval:n_data-config.interval+1]';
bedOccupancy_full(:) = interp1(x_tmp, bedOccupancy, x_interp,'previous','extrap');
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [ parameters, calcSS, calcSV, emptyLevel ] = adaptive(occupiedSS,occupiedSV,emptySS, parameters)
%ADAPTIVE Optimizes BCG parameters based on occupied Signal Strength,
%stroke volume and empty bed level
%   Input parameters: array of occupied signal strengths, array of occupied
%   stroke volumes, empty bed signal strength level.
%   Output: array of BCG calibration parameters
%   [var_level_1,var_level_2, stroke_volume, tentative_stroke_volume,
%   signal_range,to_micro_g]. tentative_stroke_volume = 0, to_micro_g = 7

%% Check which adaptive calibration method to use
% Ratio of non-zero stroke volume readings
svNonZeroRatio = nnz(occupiedSV) / length(occupiedSV);
% Stroke volume variation
svDiff = sum(abs(diff(nonzeros(occupiedSV)))) / length(nonzeros(occupiedSV));
var1_min = 4000;
svDiff_min = 0.25;
svNonZeroRatio_min = 0.5;
empty_scaler = 0.25;
occupiedPercentile = 0.25;  % 0.5 = median
emptyPercentile = 0.5;

occupiedSS = nonzeros(occupiedSS);
occupiedSV = nonzeros(occupiedSV);
calcSS = percentile(occupiedSS, occupiedPercentile);
calcSV = percentile(occupiedSV, occupiedPercentile);
if svDiff > svDiff_min && svNonZeroRatio > svNonZeroRatio_min
    %% Adaptive calibration based on stroke_volume and signal strength
    fprintf(1,'Running adaptive calibration based on stroke volumes and signal strengths.\n');
    var1_multiplier = 7;            % from signal strength
    signal_range_multiplier = 45;   % from stroke volume
    stroke_vol_multiplier = 2.6;    % from signal range

    % calculate parameters
    var_level_1 = max(var1_min, var1_multiplier * calcSS);
    if var_level_1 > 15000  % limits var_level_1 of going too high
        var_level_1 = 15000 + (var_level_1-15000) / 2;
    end
    signal_range = calcSV * signal_range_multiplier;
    stroke_vol = round(stroke_vol_multiplier * signal_range);
else
    %% Adaptive calibration based on only signal strength
    fprintf(1,'Running adaptive calibration based only on signal strengths.\n');
    var1_multiplier = 7;            % from signal strength
    signal_range_multiplier = 2.7;  % from signal strength
    stroke_vol_multiplier = 2.6;    % from signal range
    % calculate parameters
    calcSS = percentile(occupiedSS, occupiedPercentile);
    var_level_1 = max(var1_min, var1_multiplier * calcSS);
    if var_level_1 > 15000  % limits var_level_1 of going too high
        var_level_1 = 15000 + (var_level_1-15000) / 2;
    end
    signal_range = calcSS * signal_range_multiplier;
    stroke_vol = round(stroke_vol_multiplier * signal_range);
end
emptyLevel = percentile(emptySS, emptyPercentile);
var_level_2 = emptyLevel + empty_scaler*(calcSS-emptyLevel);    % var_level_2 is between emptyLevel and percentile of
occupied Signal Strengths
parameters(:) = round([var_level_1, var_level_2, stroke_vol, stroke_vol, signal_range, 7]);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%END OF REQUIRED CODE%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% FOLLOWING SCRIPTS CAN BE USED FOR DATA PLOTTING, UNCOMMENT DEBUGS FROM intelligentCalibration & RUN BELOW SCRIPTS
FOR PLOTS



% debug_plotLimitAndRate.m

global DEBUG

if config.emptyLevel
    fprintf('Using only adaptive calibration, nothing to plot.\n');
    return;
end
nStdLimits = length(DEBUG.stdLimits);
nMeanLimits = length(DEBUG.meanLimits);
nSignalStd = size(DEBUG.stdSignal,1);
nSignalMean = size(DEBUG.meanSignal,1);
diffStdLimits = diff(DEBUG.stdLimits);
diffMeanLimits = diff(DEBUG.meanLimits);

stdTime = [config.interval:config.interval:n_data]';
meanTime = [config.interval:config.interval:n_data]';
time = [1:n_data]';

f1 = figure;
pos = get(f1,'Position');
set(f1,'Position',[pos(1) pos(2) pos(3)*1.5 pos(4)]);

% Plot std limits
subplot(2,4,1:3);
str = strsplit(filename,'\\');
title(str(end),'Interpreter','none');
h1 =
plot(repmat(stdTime./3600,[1,nStdLimits]),DEBUG.stdProbabilities.*diffStdLimits(1)*0.8+repmat(DEBUG.stdLimits,[nSignal
Std 1]));
set(h1(DEBUG.stdBestValley(3)),'LineWidth',2,'Color',[0.2 0.2 0.2]);
axis tight
ylabel('STD Limit Sweep');
xlabel('Time / [h]');

% Plot std rate
subplot(2,4,4);
plot(DEBUG.stdRate(:,1),DEBUG.stdRate(:,2));
hold on;
plot(DEBUG.stdBestValley(:,1),DEBUG.stdBestValley(:,2),'ko','LineWidth',5);
text(DEBUG.stdBestValley(:,1)+10, DEBUG.stdBestValley(:,2), num2str(DEBUG.stdBestValley(:,1),'%.0f'));
hold off;
axis tight
ylabel('Rate');
xlabel('STD Limit Sweep');
legend('rate','best valley');

% Plot mean limits
subplot(2,4,5:7);
str = strsplit(filename,'\\');
title(str(end),'Interpreter','none');
h2 =
plot(repmat(meanTime./3600,[1,nMeanLimits]),DEBUG.meanProbabilities.*diffMeanLimits(1)*0.8+repmat(DEBUG.meanLimits,[nS
ignalMean 1]));
set(h2(DEBUG.meanBestValley(3)),'LineWidth',2,'Color',[0.2 0.2 0.2]);
axis tight
ylabel('Percentile Limit Sweep');
xlabel('Time / [h]');

% Plot mean rate
subplot(2,4,8);
plot(DEBUG.meanRate(:,1),DEBUG.meanRate(:,2));
hold on;
plot(DEBUG.meanBestValley(:,1),DEBUG.meanBestValley(:,2),'ko','LineWidth',5);
text(DEBUG.meanBestValley(:,1)+40, DEBUG.meanBestValley(:,2), num2str(DEBUG.meanBestValley(:,1),'%.0f'));
hold off;
axis tight
ylabel('Rate');
xlabel('Percentile Limit Sweep');
legend('rate','best valley');

drawnow
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% debug_plotOccupancyResults.m
```

```
f2 = figure;
pos = get(f2,'Position');
set(f2,'Position',[pos(1) pos(2) pos(3)*1.5 pos(4)]);

global DEBUG

stdTime = [config.interval:config.interval:n_data]';
meanTime = [config.interval:config.interval:n_data]';
time = [1:n_data]';

% Plot signal strength and bed occupancy
ax1 = subplot(3,4,1:8);
str = strsplit(filename,'\\');
title(str(end),'Interpreter','none');
hold on;
plot(time./3600,data(:,6));
plot(time./3600,DEBUG.bedOccupancy.*1000,'LineWidth',2);
hold off;
axis tight;
ylim([0 15000]);
legend('Signal Strength','Bed Occupancy/Movement');
xlabel('Time / [h]');
ylabel('Signal Strength / [a.u.]');


% Show results in figure
str0 = sprintf('Interval (s): %.0f\n',config.interval);
str1 = sprintf('Adaptive Parameters: %.0f,%.0f,%.0f,%.0f,%.0f,%.0f\n',output.parameters);
str2 = sprintf('Best Mean separation level: %.0f',DEBUG.meanBestValley(1,1));
str3 = sprintf('Best STD separation level: %.0f',DEBUG.stdBestValley(1,1));
uitext = uicontrol('Style','text','String',{str0;str1},'HorizontalAlignment','left','Position',[100 30 200 100]);
set(uitext,'FontSize',10);
set(uitext,'Unit', 'normalized','FontUnits','normalized')

drawnow;
```

Rev.	Date	Change Description
1	29-August-17	First version (version 1.0.0 of the code)
2	7-September-17	Corrected soma false instances of word struct to array/matrix