02131 Assignment 1: Software implementation of a personal ECG scanner

Michael Reibel Boesen, Linas Kaminskas,
Paul Pop, Karsten Juul Frederiksen

September 9, 2015

1 Introduction

You work as an engineer at the fictive company ”Medembed” and are given the task of investigating whether the Pan-Thompkins QRS detection algorithm [1] might be used for the companies next product - a wearable ElectroCardioGram (ECG) scanner. Your first task will be to implement the algorithm in C. Then you have to determine whether the algorithm is suitable to be implemented in an embedded system.

For this purpose the project will be divided into the following parts:

- **A1.1:** Data acquisition.
- **A1.2:** Implement the filter part of the algorithm.
- **A1.3:** Implement the adaptable part of the algorithm.
- **A1.4:** Output relevant data to the user based on the algorithm.
- **A1.5:** Analyze your implementation and determine its critical parts.

1.1 Deadline

By **September 29 at 23:59** you have to deliver an electronic version of the report and source files as an archive uploaded on CampusNet at the location:

`CampusNet/Assignments/Assignment 1`

1.2 Notes & Hints

Althought the basic algorithm in this assignment is simple, there are many ways to implement it. As we are not enforcing a particular approach - rather we would like you to explore - it may be difficult for the TA’s to understand your implementation and thus, help debugging your code. Therefore:

- *Think before coding!*
- Make sure you understand the algorithm.

- Structure your code such that it is clear what the different parts are doing.
- Think about testing your code while planning and make sure that you test and debug the individual parts.

1.3 Expectations

The recent years we have experienced that the students have had a hard time of determining whether they are behind or on track, time-wise. Therefore this plan should guide you into checking that you're on track. You will have 3 TA supervised labs to complete this exercise. Please note that we **also** expect that you work on this as part of your preparation for the course! You will probably not be able to finish the assignment if you only use labs to work on it.

Date	**Complete**
10/9	A1.1, A1.2
17/9	A1.3
24/9	A1.4, A1.5

Table 1: Table of expectations

2 Algorithm overview

The purpose of the algorithm is to analyze an ECG signal and determine what is known as the QRS complex. The ECG is an electronic representation of the electrical activity of the heart over a period of time. The signal is usually detected using electrodes, which are attached to the persons skin around the chest region. The QRS complex is a signal feature that corresponds to the depolarization of the right and left ventricles of the heart. An example of a QRS complex and its components (P, Q, R, S and T, known as *waves*) can be seen in Figure 1. The algorithm you will implement identifies the R peak of the QRS complex in real-time. This means that it can be used to alert the patient of a potential heart problem. In our case the device will need to output:

- Intensity of the R-peak.
- Time-value of the R-peak.
- R-to-R interval (i.e. when did the last R-peak occur).
- Warn if either (1) the intensity of the R-peak is below a certain threshold, (2) if the R-to-R interval is unstable (to be explained later in section 6).

The algorithm consists of two parts, which constitutes A1.1 and A1.2, respectively.

Due to the fact that you will later need to implement the mathematical operations in hardware, we suggest that you use integers and not fixed or floating point data types, because fixed or floating point operations in hardware are out of the scope of this course.

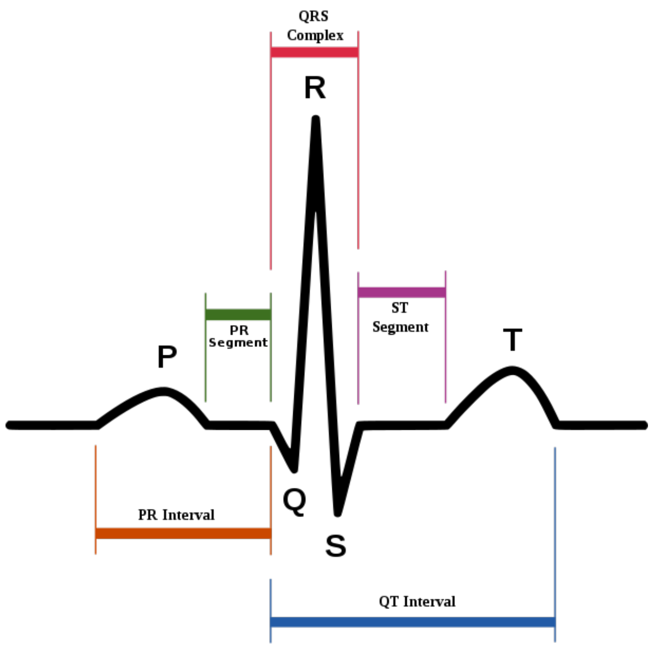


Figure 1: A QRS complex in an ECG signal

3 A1.1: Data acquisition in real-time

When you design the Medembed ECG device you will not be working with a real patient but with a dataset from a real patient. The file can be found on Campusnet under `Labs/DataFiles/ECG.txt` (sampling rate is 250 samples pr. second). But because the ECG device will have to work on data being acquired in real-time you will have to simulate this with the dataset. This means that you are not allowed to read the entire file into for instance an array and then process it. You will have to read the data points one at a time an process them as they are being read from a sensor. In order to simulate this we have provided you with a `Sensor.h` and `Sensor.c` file, which will simulate that you read the data from a sensor. Therefore, the first thing you will have to do is to implement the `int getNextData` method in the `Sensor.c` file. This method should simply return the next data point from the `ECG.txt` file.

Note, when you get to A1.3 you will need 3 data points in order to detect a peak, it is therefore accpeted to read three data points ahead the first time only.

4 A1.2: Filtering

Due to the medical nature of the algorithm, it is of utmost importance that the R-peaks are detected as accurately as possible. In order to facilitate this, the algorithm applies 5 sequential filters to the data coming in. This is done in order to enhance the R-peaks and remove as much noise as possible. There are a total of five operations that must be performed on the data. These must be performed in the following order:

1. Low-pass filter.
2. High-pass filter.
3. Derivative.
4. Squaring.
5. Moving window integration.

Figure 2 shows an overview of how the filters are connected with each other. Before we give a more detailed description of each filter, we will first give a basic introduction to what signals and filters are.

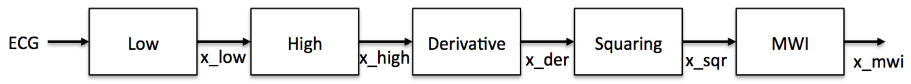


Figure 2: An overview of the filters and how they are connected

4.1 Signal and Filter introduction

This brief section will give you the bare essentials about signal processing that you will need to know. This is in order to give you a better understanding, such that the implementation of the filters becomes less problematic later on.

A *signal* represents information about features, attributes and/or behavior of some phenomenon as a function of time. It is important to distinguish between a signal sampled in continuous time and a signal sampled in discrete time. Figure 3(a) shows a signal sampled with a *sampling rate* of 5 Hz[1]. This means that a sensor reads the signal once every 0.2 seconds. In Figure 3(b) and Figure 3(c) shows the same signal, but sampled with a sampling rate of 10 Hz and 20 Hz respectively. Finally, Figure 3(d) shows the same signal sampled in continuous time. The difference between the four figures is the amount of information available. Therefore, when working with signal processing applications, choosing the sampling rate is very important. Especially in embedded systems, where there can be a large constraint on the resources.

A sample *dataset* is the individual data points, i.e. an x and a y value, where x is the sample number and y is the value of the signal. If you know the sample rate, one can compute the time value for the point y using x. For example, at $x = 5$, the value of the signal is $y = 200$. Given a sample rate of 2Hz, i.e. 2 samples pr. second, you can infer that at time $t = 2.5$ seconds the value of $y = 200$.

When you sample data from a sensor, it is quite common that the data collected is noisy. In order to be able to see the features which you are interested in, a filter can be applied. One can think of it as the filter in, for instance, the bottom of a sink. The filter is used to catch bits of food etc, which you do not want to flushed down your water pipes. In our case, a filter is capable of removing unwanted features from a signal. Unwanted features from a signal

[1]Recall that 1 Hz would equal one sample every second.

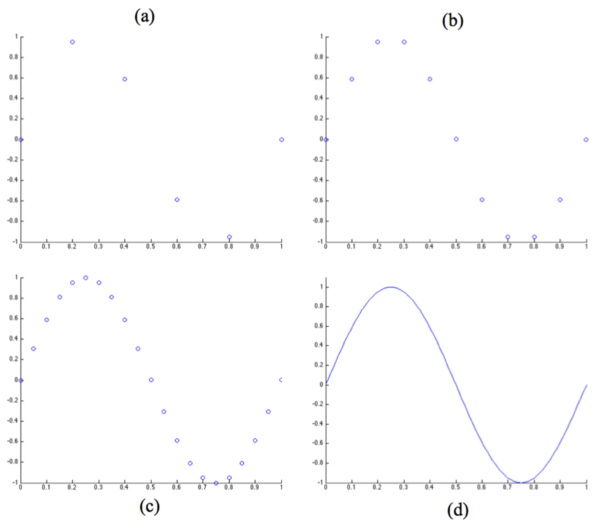


Figure 3: x-axis is time and y-axis is amplitude. (a) Sample rate: 5 Hz, (b) Sample rate: 10 Hz, (c) Sample rate: 20 Hz, (d) Continuous time.

could be noise. It could also be to attenuate parts of the signal, which are not interesting or to strengthen parts which are interesting. The problem now is, how can we distinguish between the parts of the signal we want, from the parts of the signal we don't want? For instance, how do we identify noise from the real signal?

In order to understand this, you need to know that a signal can be represented in two domains: time and frequency. When we read and plot a signal over time (as in Figure 3) the signal is in the *time domain*. However, by applying a mathematical operation known as a transformation (such as a Fourier Transform), the signal can be transfered to the *frequency domain*. A signal displayed in the frequency domain, shows how much of a signal that lies within each frequency band over a range of frequencies. That is a lot of new stuff right there - what is a frequency in relation to our signal?

Back in the late 18th century, Jean Baptiste Joseph Fourier was the first to discover that all signals can be reconstructed by adding up a number of sine waves (see Figure 4). Each sine wave will then have a frequency[2]. This implies that displaying a signal in the frequency domain, one also displays which frequencies are present in the signal and at what amplitudes. Figure 5 show the same signal from Figure 4 in both the time and frequency domain.

For example, Figure Figure 6(a) shows the signal $5 \cdot sin(2\pi \cdot 2 \cdot t) + 2 \cdot sin(2\pi \cdot 6 \cdot t)$, i.e. two sine waves; one with a frequency of 2 Hz and another with a frequency of 6 Hz. If one applies a Fourier Transform to the signal, one will get Figure Figure 6(b), i.e. the signal in the frequency domain. One can clearly see the two frequency components; one at 2 Hz and one at 6 Hz. The Matlab

[2]Recall that a sine wave can be constructed from $A \cdot sin(\omega \cdot t + \theta)$, where A is the amplitude, ω is the angular frequency and θ is the phase.

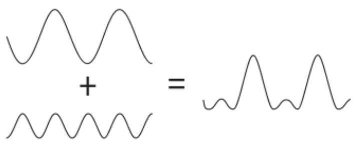


Figure 4: Adding up sine waves to produce an arbitrary signal - from [2]

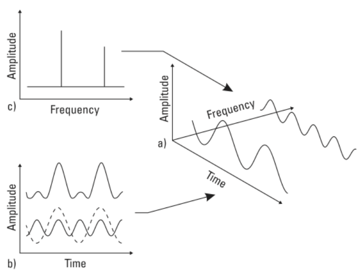


Figure 5: (a) Signal in both frequency and time domain, (b) Signal in time domain, (c) Signal in frequency domain.

program used to generate these figures can be found on CampusNet Filesharing in `Labs/Support/freqtest.m`.

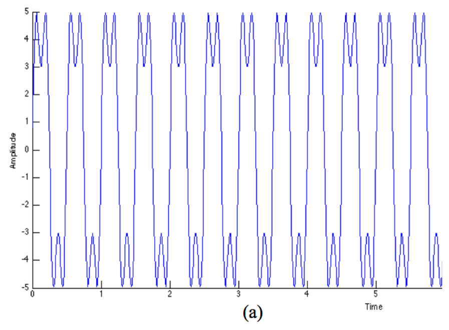
Now we can explain how to remove unwanted features from a signal. If we know the frequency of the signal that we want to read, we can filter out the frequencies that we don't want[3]. In our case, we know (from medical personnel) that the frequencies which are interesting, are in the range of 5 and 15 Hz.

This finally brings us to the filters, which I will explain by an example. In signal processing one very common filter is the *low-pass filter*[4]. This filter is characterized by allowing numerically low frequencies to pass and blocks the high frequencies; hence the name of the filter. Exactly how the low-pass filter operates on the data is characterized by the frequencies it should let pass. The exact details are out of the scope of this course. Though it suffice to say that all filters have a *difference equation*, which relates the input signal to the output of the filter. This equation describes the mathematical operations necessary to be performed, in order to block out the unwanted frequencies. How to create the difference equation is also out of the scope of this course.

A low-pass filter has also a *cut-off frequency*, which means that all frequencies above a particular frequency are blocked out or "cut-off". The final property we need to introduce is the *group delay*. When the data passes through the filter, it will be delayed a for a small amount of time. This is represented by the group delay, and is dependent on the filter type and the sampling rate used. The group delay might be necessary to take into account, if you want to relate the data produced by the filters or succeeding algorithm back to the time domain.

[3]The easiest way to think about this is to think about listening to FM radio. You pick a frequency that you want to hear, and your radio receiver filters out the frequencies which you don't want to hear.

[4]Others are: *high-pass filter* and *band-pass filter*



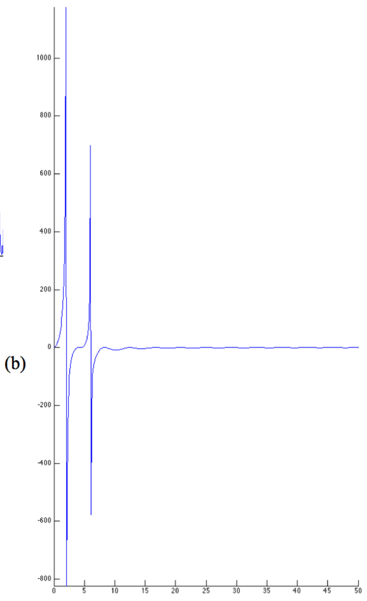


Figure 6: (a) Signal in the time domain, (b) Signal in frequency domain (x-axis is frequencies in Hz and y-axis is amplitude).

The next sections will explain the filters you need to implement according to Figure 2.

4.2 Low-pass

The difference equation for the low pass filter is seen in Equation 1:

$$y_n = 2y_{n-1} - y_{n-2} + \frac{1}{32} \cdot (x_n - 2x_{n-6} + x_{n-12}) \tag{1}$$

where x is the signal data, y is the resulting filtered signal and n refers to the n'th sample. The filter has a cut-off frequency of 12 Hz and a group delay of 25ms.

4.3 High-pass

The high-pass filter is very similar to the low-pass filter - instead of blocking out high frequencies, the high-pass filter blocks out low frequencies. The difference equation for the high pass filter can be seen in Equation 2.

$$y_n = y_{n-1} - \frac{x_n}{32} + x_{n-16} - x_{n-17} + \frac{x_{n-32}}{32} \tag{2}$$

The filter has a cut-off frequency of 6 Hz and a group delay of 80 ms.

Figure 7 shows what happens to the ECG data which you are working on after the application of the low and high-pass filters, both in the frequency and time domain. Clearly, after the application of the filters the frequencies in the ECG data has been centered around the 5-20 Hz approximately (no filter is perfect).

4.4 Derivative

After the cascaded low- and high-pass filters the signal is differentiated in order to provide the slope information of the QRS complex. This is an example of a filter that enhances parts of the signal which we are interested in analyzing.

The difference equation in Equation 3 to the data, which has been filtered by the high-pass filter.

$$y_n = \frac{1}{8}(2x_n + x_{n-1} - x_{n-3} - 2x_{n-4}) \tag{3}$$

This approximates an ideal derivative between DC and 30 Hz and it has a delay of 10 ms.

4.5 Squaring

After the derivative all data points are squared in order to emphasize the large differences resulting from the QRS complex as well as to make all points positive. Again, this is a filter which enhances parts of the signal which we are interested in.

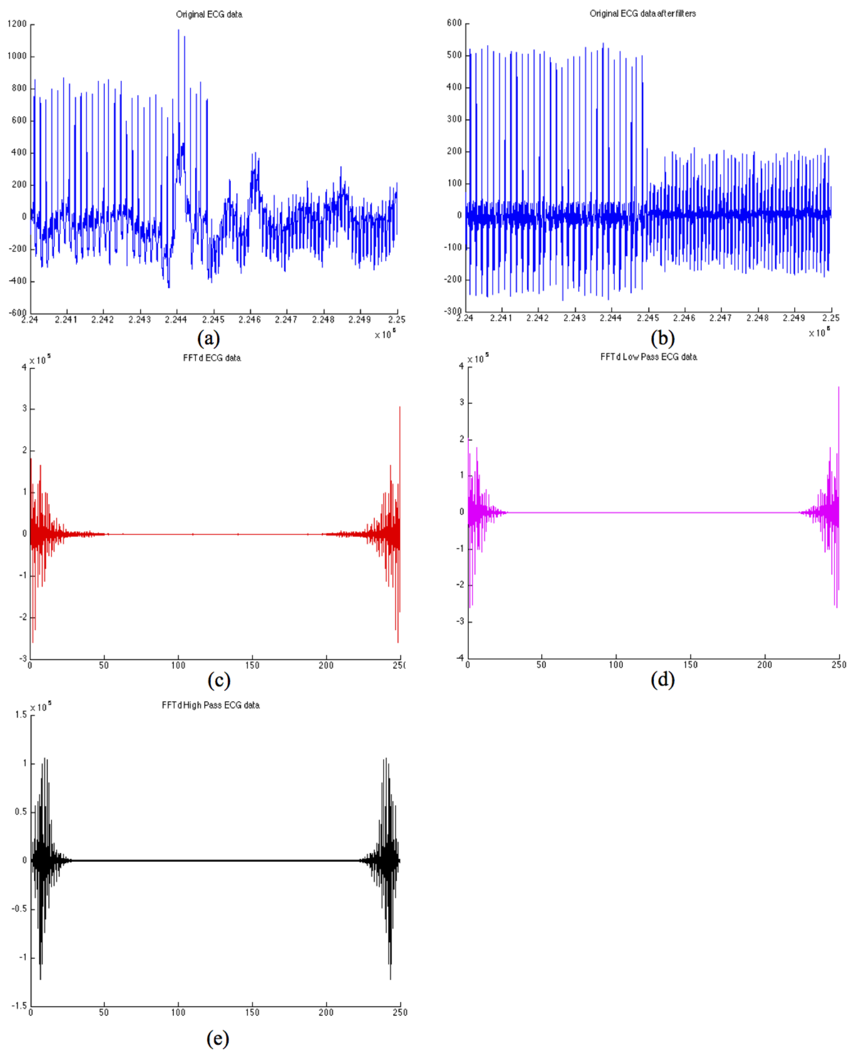


Figure 7: (a) Original ECG data in time domain, (b) ECG data after low and high-pass filters in time domain, (c) Original ECG data in frequency domain, (d) ECG data in frequency domain after low-pass filter, (e) ECG data in frequency domain after low-pass and high-pass filter. Note that the Fourier Transform used to get the signal into the frequency domain, always creates a mirror image of itself.

4.6 Moving window integration

Finally, a moving-window integration is applied in order to obtain a smoothed signal. This is done because the output of the differentiated signal might exhibit several peaks from the other waves associated with the QRS complex. The moving-window integration is defined as seen by the difference equation seen in Equation 4:

$$y_n = \frac{1}{N}(x_{n-(N-1)} + x_{n-(N-2)} + \cdots + x_n) \tag{4}$$

where N (the size of the moving window integration equals 30). The filter has a delay of 72.5ms.

4.7 Tips

- Design and implement one filter at a time and test the output of it using the files available on CampusNet Filesharing in `/Labs/DataFiles`. Note that the names of the filters correspond to output of the filter from Figure 2.
- When picking up data, make sure to use and implement the methods as specified in the `Sensor.h` file as this will simulate reading data in real time via the sensor.
- Note that you cannot simply read all data points from the sensor into an array and then operate on them afterwards. The sensor needs to operate on the data in real-time with the data available.
- When you have implemented the filters (and if you have time) try and see if you can optimize them in terms of speed.

5 A1.3: Detecting

The data from the moving window integration filter is finally being passed to the part of the algorithm which implements the actual detection of the R-peaks. The main idea of the algorithm is to:

1. Search for peaks.
2. Determine if a peak is higher than a certain threshold (which indicates an R-peak) and if not classify it as noise.
3. If no peak has been detected for a certain period of time searchback in order to find another significant peak.

Figure Figure 8 shows a flowchart that describes how these three steps are performed. The steps are further elaborated in the following.

5.1 Searching for peaks

This part of the algorithm searches for peaks in the dataset i.e. a point, that is a local maxima. One way of performing this, is to use the following:

$$x_{n-1} < x_n \, , \, x_n > x_{n+1} \tag{5}$$

where n is the index, and x_n is the value of the local maximum in the dataset.

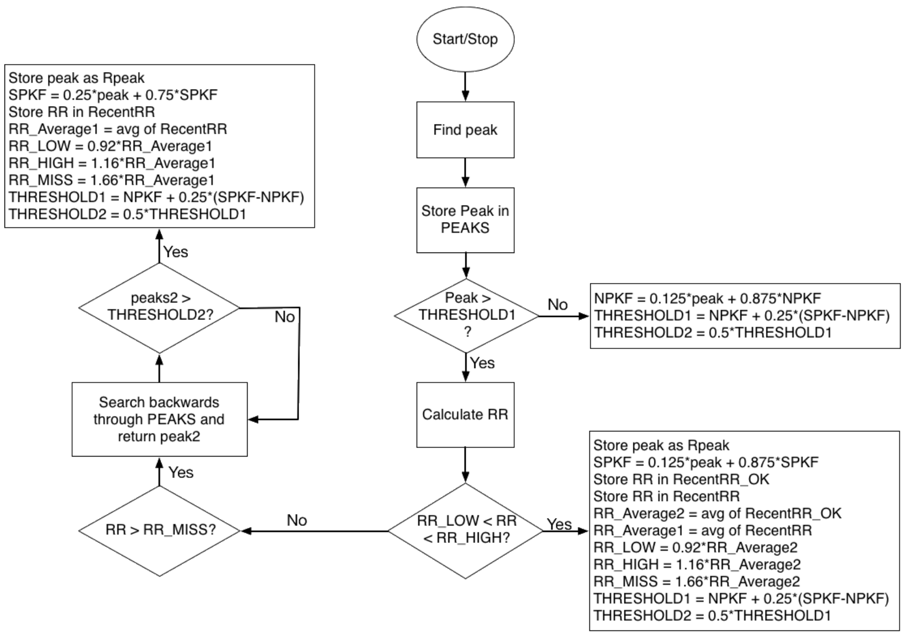

Start/Stop
Find peak
Store Peak in PEAKS
Peak > THRESHOLD1 ?
No
NPKF = 0.125*peak + 0.875*NPKF
THRESHOLD1 = NPKF + 0.25*(SPKF-NPKF)
THRESHOLD2 = 0.5*THRESHOLD1
Yes
Calculate RR
RR_LOW < RR < RR_HIGH?
Yes
Store peak as Rpeak
SPKF = 0.125*peak + 0.875*SPKF
Store RR in RecentRR_OK
Store RR in RecentRR
RR_Average2 = avg of RecentRR_OK
RR_Average1 = avg of RecentRR
RR_LOW = 0.92*RR_Average2
RR_HIGH = 1.16*RR_Average2
RR_MISS = 1.66*RR_Average2
THRESHOLD1 = NPKF + 0.25*(SPKF-NPKF)
THRESHOLD2 = 0.5*THRESHOLD1
No
RR > RR_MISS?
Yes
Search backwards through PEAKS and return peak2
peaks2 > THRESHOLD2?
No
Yes
Store peak as Rpeak
SPKF = 0.25*peak + 0.75*SPKF
Store RR in RecentRR
RR_Average1 = avg of RecentRR
RR_LOW = 0.92*RR_Average1
RR_HIGH = 1.16*RR_Average1
RR_MISS = 1.66*RR_Average1
THRESHOLD1 = NPKF + 0.25*(SPKF-NPKF)
THRESHOLD2 = 0.5*THRESHOLD1

Figure 8: QRS algorithm flow chart

5.2 Finding R-peaks

The algorithm maintains a number of variables in order to adaptively adjust itself to the monitored person's heartbeat. These variables are:

- `SPKF`: An estimate of the value of an R-peak.
- `NPKF`: An estimate of the value of a noise peak.
- `THRESHOLD1`: All peaks higher than this value is classified as an R-peak.
- `THRESHOLD2`: Used during the searchback procedure (see subsection 5.3).

All peaks that are found, are stored in a list named `PEAKS`. The algorithm then checks if they exceed `THRESHOLD1`. If they do, they are classified as an R-peak. Once we have identified an R-peak we can calculate what is known as the RR-interval (denoted "Calculate RR" in Figure 8). The RR-interval is the time interval between each R-peak, i.e. the time difference between each heartbeat. We can use this to say something about the stability of the monitored person's heartbeat. If this value varies significantly there might be something wrong. But in order to account for a regular drop in pulse (due to for instance rest) the algorithm keep track of a HIGH and a LOW limit for, when the R-peak should occur. This means we need to introduce 5 additional variables:

- `RR_AVERAGE1`: The average of the 8 most recent RR-intervals **regardless** of their value (used during the "searchback" see subsection 5.3).

- RR_AVERAGE2: The average of the 8 most recent RR-intervals, for which the corresponding R-peak is higher than THRESHOLD1.
- RR_LOW: Defined as 92% of RR_AVERAGE2.
- RR_HIGH: Defined as 116% of RR_AVERAGE2.
- RR_MISS: Defined as 166% of RR_AVERAGE2 (used during the "searchback", see subsection 5.3).

This means that once an R-peak has been identified the algorithm checks if the current RR interval is between RR_LOW and RR_HIGH. If this is the case the R-peak found is categorized as a regular R-peak. This means that the variables above should be updated accordingly as well as the SPKF, THRESHOLD1 and THRESHOLD2 according to Equation 6 - Equation 8.

$$SPKF = 0.125 \cdot x_n + 0.875 \cdot SPKF \tag{6}$$

$$THRESHOLD1 = NPKF + 0.25 \cdot (SPKF - NPKF) \tag{7}$$

$$THRESHOLD2 = 0.5 \cdot THRESHOLD1 \tag{8}$$

If the peak found earlier is lower than THRESHOLD1 it is classfied as a noise peak and THRESHOLD1 and THRESHOLD2 is updated according to Equation 7 and Equation 8 and NPKF is updated according to Equation 9.

$$NPKF = 0.125 \cdot x_n + 0.875 \cdot NPKF \tag{9}$$

5.3 Searchback

If the current RR interval is not within the RR_LOW and RR_HIGH (as described in the previous subsection 5.2), and below the RR_MISS, nothing happens. If it is above the RR_MISS, a procedure known as *searchback* is initialized (denoted in Figure 8 as starting from "Search backwards through PEAKS..."). The searchback procedure searches backwards through all peaks detected so far and the first peak p_n which exceeds THRESHOLD2 is then classified as an R-peak. The RR_LOW, RR_HIGH, RR_MISS and RR_AVERAGE1 is updated, as well as THRESHOLD1 and THRESHOLD2 according to Equation 7 and Equation 8 and SPKF according to Equation 10.

$$SPKF = 0.25 \cdot p_n + 0.75 \cdot SPKF \tag{10}$$

6 A1.4: Outputting data to the user

In order to provide the patient with an early warning of a pending heart problem, we want to augment the algorithm to display certain data to the patient and warn if certain conditions occur. The algorithm should display (1) the value of the latest R-peak detected, (2) the time-value at which it occurred and (3) the patients pulse. In addition it should warn the patient if:

- The R-peak value is less than 2000
- If 5 successive RR-intervals has missed the RR_LOW and RR_HIGH

7 A1.5: Program analysis

In order for your company to decide how to implement the ECG device it is necessary for you to analyze the QRS algorithm you have implemented to determine the performance of various parts of your implementation.

7.1 Performance

For the performance analysis you will have to use code instrumentation; one has two options:

Manual instrumentation:

You will have to develop a module for source code instrumentation. Source code instrumentation consists of additional instructions that are added to the original source code, which, in our case are used to measure the execution time of the code. To get a process' CPU time, you can use the `clock` function declared in the header file `time.h`:

```c
#include <time.h>

int main(int argc, char *argv[])
{
    clock_t start, end;
    double cpu_time_used;

    start = clock();
    /* The code that has to be measured. */
    ...
    end = clock();

    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;

    return 0;
}
```

Consider using macros to annotate parts of the code that you want to measure. The macros can be implemented in such a way, that they are easy to turn on/off using conditional compilation. Read more about the "C preprocessor" in the "C Primer Plus" book.

`gprof`-based instrumentation:

The second option is to use an existing source code profiler to determine details about how often the functions are called, and how much time is spent executing them. One must compile and link your program with profiling enabled.

1. In Eclipse, right-click a project and select **Properties**.
2. Navigate to **C/C++ Build** → **Tool Settings** → **Debugging**.
3. Mark **Generate gprof information (-pg)**. Click **Apply** and **OK**.

Before executing the profiler in Eclipse, one needs to select the right profiler for the binary. To do this, perform the following:

1. Navigate to a project and expand **Binaries**.
2. Right-click a binary, and navigate to **Profile As** → **Profile Configurations...**.
3. Select the **Profiler** tab. In the dropdown menu, select **Gprof**.
4. Next, navigate to **C/C++ General** → **Profiling Categories**. Select the **Timing** tab.
5. Mark **Enable project-specific settings**, and select **Gprof** [**Function Execution Timing**].
6. Click **Apply** and **OK**.

In order to profile in Eclipse, perform the following:

1. Navigate to your project and expand **Binaries**.
2. Right-click the chosen binary, and navigate to **Profile As** → **Profile C/C++ Application**.
3. After execution of the program is complete, next to the **Console** window, a **gprof** tab will appear.
4. Inspect the **gprof** tab.

When a program has completed execution, a `gmon.out` file will be created. This file contains the information about the profiled functions. In case the **gprof** tab does not contain meaningful information, then the program is executing to fast. There are several ways to deal with this:

1. Use a larger data set in order to see impact on the number of function calls and timing.
2. Execute the same program on the same data set several times, i.e. a `for` or `while` loop in the `main` function, with a reasonable amount of repetition.
3. If the **gprof** does not display any functions, make sure that these functions are not `inline` functions or preprocessor macros.

In case the **gprof** does not display properly the data, one can export the profiled data as a CSV file via Eclipse, and one can import CSV data in any spreadsheet program. Last measure is to use the terminal emulator and execute `gprof` from the terminal. To execute from a terminal emulator, perform the following:

1. Navigate to **Applications** → **Terminal Emulator**.
2. Change the directory to the main project folder where the `gmon.out` file is available. Execute:

   ```
   cd path-to-project-folder
   ```

3. Execute:

   ```
   gprof path-to-the-executable ./gmon.out
   ```

   Example:

   ```
   gprof ./Debug/hello_world ./gmon.out
   ```

This will display the timing interpretation of each column, and display the measurements performed on each function by `gprof`. To suppress the measurement interpretation text blob, execute with `gprof -b` instead. See the lecture slides and the GNU `gprof` Manual on CampusNet or execute `man gprof` in a terminal emulator for more information.

Optional: If some parts of the code stands out due to a long execution time, see if you can improve it. Document the changes of the implementation, if the change results in significant improvement.

7.2 Energy consumption and code size

How would you estimate the energy consumption of your program functions, given the numbers your have obtained during the performance analysis and the power consumption data for your processor (search for the relevant data on the Internet)?

Also:

- How would you determine the code size of your program?
- Can you say something about the code size of individual functions?
- Is the code size influenced by compiler optimization flags?

References

[1] Jiapu Pan and Willis J. Tompkins. A real-time qrs detection algorithm. *IEEE Transactions on Biomedical Engineering*, BME-32(3):230–236, 1985.

[2] Agilent Technologies. The fundamentals of signal analysis. *Application note*, page 68.