02131
Indlejrede Systemer

DTU Compute
Department of Applied Mathematics and Computer Science

Aflevering 1

Alexander Kristian Armstrong (s154302)
Emilie Isabella Dahl (s153762)

29. september 2016

Indhold

1 Introduktion

Formålet er at implementere en bærbar ElectroCardioGram (ECG). Dette gøres ved at antage at der modtages data, som skal analyseres ved at implementere en algorithme i C. Dataen antages at blive målt løbende, men i denne opgave vil dataen blive givet i form af en txt-fil. Vi har udarbejdet designet, implementationen og rapporten sammen som et hold.

2 Design

2.1 Data

Den føste del af opgaven går ud på at opsamle og gemme data, som derved kan filtreres til et output. Dataen, der er vores ECG signal, er i dette tilfælde allerede blevet gemt i en tekst fil. Der skal derfor findes en metode til at læse inputtet fra filen enkeltvis, for at skabe en troværdig repræsentation af virkeligheden, hvor dataen vil blive indsamlet løbende over tid. Dette gøres ved at lave en source file som scanner txt-filen for det næste input, som derefter bliver retuneret til main filen. Det er derved main-filen, som opdaterer data punkterne ved hjælp af et while loop.

2.2 Filtrene

Filtrerne er konstrueret i en source file for sig selv. Denne modtager det nuværende datapunkt og filtrerer det. Hvert datapunkt skal igennem 5 filtre. Disse filtrer transformerer bl.a. de rå data over i frekvens domænet, hvilket gør det lettere at adskille støj fra. De to første filtre fjerner høje og lave frekvenser. Herefter sendes dataen igennem et differentierende filter for at få oplysninger om hældningen af QRS komplpekset. Sidst sættes alle tallene i anden, for at undgå negative tal, og det sendes igennem et såkaldt Moving Window Integration filter, der glatter daataen ud.

Alle filtrerne ligger i filter.c filen. Hvert filter tager blot imod et datapunkt. Ved at benytte statiske integer arrays gemmes de forrige tal i de specifikke filtres individuelle funktioner. I main.c filen kaldes blot én funktion: `filterData(int data)`. Denne funktion kører alle filtrerne på det enkelte datapunkt, hvorefter det filtrerede datapunkt returneres.

2.3 QRS

Formålet med QRS-algoritmen er at detekterer de interessante peaks i den givne data. Disse kaldes R-peaks. Selve algoritmen er en række if-else sætninger der udfører nogle udregninger alt efter hvordan peaken der er givet ser ud. I korte træk går algoritmen ud på først at finde en peak, derefter at se om den er over en bestemt grænse, i så fald er det en R-peak. Hvis der ikke findes en R-peak i et stykke tid, ses der tilbage på nogle forrige peaks, for at se om disse kunne være gyldige. Denne process kaldes for searchback.

Inden QRS-algoritmen køres, så konstrueres en structure, som indeholder væsentlig data der benyttes og opdateres løbende igennem algoritmen. Dette er smart da der så ikke skal holdes styr på variable imellem to source filer. Structuren indeholder bla.a. følgende variable:

- `SPKF` - En estimeret værdi for en R-peak

- `NPKF` - En estimeret værdi for en støj peak
- `THRESHOLD1` - Minimumsgrænsen for en R-peak
- `THRESHOLD2` - Grænse der benyttes under searchback
- `Rpeak` - Holder værdien for den seneste R-peak
- `RpeakTime` - Holder tiden for den seneste R-peak
- `RR` - Holder tiden imellem de to seneste R-peaks
- `seconds` - Holder tiden programmet har kørt i sekunder

Denne structure oprettes i main filen og initialiseres med nogle startværdier. De ovenstående variable er alle relevante for at udskrive og gemme resultatet af analysen. Der er 14 variabel mere der kun bliver brugt i qrs.c til at holde styr på det førhenværende data til når en nyt datapunkt skal analyseres.

Det første der undersøges er om datapunktet er en peak. Dette gøres ved at sammenligne det med 4 andre hosliggende datapunkter. Hvis det givne datapunkt er det største af de 5, så antages det, at der er tale om en peak. Denne peak indsættes i et cirkulært array. Det samme gøres for tidspunktet som peaken blev fundet i. Der benyttes cirkulære arrays for at sænke køretiden, da dette betyder at der kun er et tal der skal ændres hver gang der findes en ny peak. Herefter undersøges det om peaken er over den første grænse: Threshold1. Er den ikke det så skal NPKF opdateres samt begge Thresholds. Er den større end Threshold1, så skal dens RR værdi udregnes og gemmes.

RR værdien gemmes her i to arrays: Et kun for Threshold1 peaks, og et for alle R-peaks. Dette gøres ved at trække den nuværende tid fra den forrige R-peaks tid. Herefter sammenlignes RR værdien med to andre grænseværdier: `RR_low` og `RR_high`. Disse grænser bestemmer tidsintervallet for hvornår der er tale om en R-peak. De bestemmes ud fra en procentdel af gennemsnittet af de 8 nyeste RR værdier. Dette gennemsnit beregnes ved hjælp af arrayet af RR værdier. Hvis peaken er under RR_low så opdateres en variabel der holder styr på ustabil hjerterytme og derefter kører algoritmen blot igen. Derimod hvis RR værdien er over RR_high sammenlignes den med endnu en grænse: `RR_miss`. Er den over denne grænse initialiseres der en searchback, da der herved er en god chance for at et peak et blevet overset.

I en searchback undersøges de forrige peaks, som er blevet gemt i et array, indtil der findes en der er over Threshold2. Denne peak ses så som en R-peak. Her skal RR værdierne dog kun gemmes i et enkelt array, det der er for alle R-peaks. Derudover opdateres grænserne og RR værdierne igen.

Funktionen ender nu med at returnere en variabel exit som specificerer at der er fundet en Rpeak over threshold1 (1), en via en searchback (2), eller der slet ikke blev fundet en (0).

2.4 Output

Programmet skal i sidste ende kunne fortælle brugeren om Rpeak-værdien, tidpunktet for denne værdi, pulsen samt give en advarsel hvis hjerterytmen bliver ustabil. R-peak værdien uskrives først efter funktionerne fra filen qrs.c er blevet kaldt, så det er nu nemt at finde den i den konstrueret structure. Tiden i sekunder bliver fundet ved en integer counter, som sættes op hver gang 250 datapunkter er blevet indlæst. Dette

fungerer da der i beskrivelsen forklares at der indsættes 250 datapunkter pr. sekund til *ECG.txt* filen.

For så at beregne pulsen kan RR værdien som gemmes i QRS_params bruges. Pulsen er beskrevet som hjerteslag pr. minut, og ved hjælp ef RR beregnes ved følgende formel:

$$\text{Pulse} = \frac{60 \cdot 100}{\frac{RR \cdot 100}{250}}$$

Da RR værdien er en integer og formlen kræver division, ganges RR værdien i første omgang op med 100, således at divisionen ikke bliver afrundet upræcist. Pulsen opdateres derved efter hvert peak, da denne konstruerer en ny RR værdi, og outputtet derved er præcis som mulig i øjeblikket.

Det sidste output er advarelser, som bliver printet i både main.c og qrs.c. I main.c undersøges om der er en for høj Rpeak-værdi (> 2000) via en if-sætning, mens qrs.c undersøger om der er 5 eller over ustabile peaks i træk.

Al dataen bliver lige nu præsenteret i form af udskrivelse til konsollen. Denne bliver printet hver gang koden registrerer et nyt peak og er derfor altid opdateret ift. de seneste oplysninger, kun forsinket af programmets egen kørertid.

3 Implementering

3.1 Struktur

Programmet består overordnet set af syv filer; main source filen, 3 source filer og deres tilhørende headers; `sensor`, `filters` og `qrs`. Disse bliver alle kaldt i filen main.c, som betytter source filerne individuelt til at finde, filtrere og analysere dataen. Først skal dataen læses fra txt-filen. Dette gør sensor.c, som indholder 3 metoder der, læser og konstruerer filer, samt finder det næste datapunkt fra en given fil.

Main filen opdaterer derefter dataen ved at benytte metoden `filterData(int data)` fra filters.c. Herinde vil dataen blive filtreret i de 5 filtre og returneret til main filen. Denne source file består derved af 6 metoder, en til hvert filter samt en der sender dataen igennem alle filtrene.

Da dataen nu er blevet filtreret skal det analyseres således at den detekterer hvornår der er peaks, hvor store disse er og hvor høj pulsen er. filen, qrs.c, bliver kaldt i main ved metoden `peakDetektion(QRS\_params *qrs\_params)`. Denne starter med at bestemme om der reelt set er et peak og derefter om dette er fundet inden for en stabil tid. Hvis hjernerytmen er ustabil skal brugeren have det at vide hvilket detekteres ved hjælp af en integer, exit, som returnerer 0 hvis der ikke er fundet et peak, 1, hvis peaket er fundet under stabile omstændigheder og 2, hvis det er fundet igennem en searchBack.

Main filen gemmer nu oplysningerne fra qrs.c i filer alt efter hvilken værdi exit har, således at værdierne kan blive sorteret og man nemmere kan danne sig overblik over de dataen. Der bliver konstrueret i alt 7 filer der beskriver Rpeaks værdier og tid for almindelige Rpeaks og dem fundet ved searchBack, en fil til hver af thresholdsene samt en til de filtreret data.

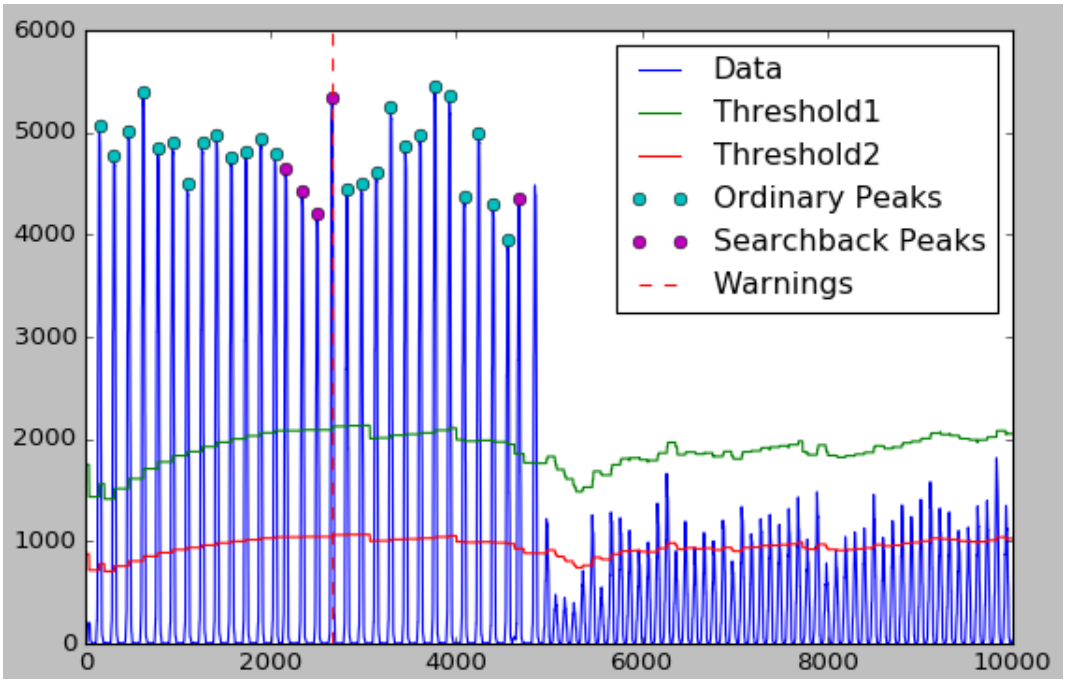
3.2 Centrale datatyper

Igennem hele implementeringen er der kun benyttet integer værdier, da dette gør det nemmere at implementere det i hardwaren, samt de fylder mindre end doubles, floats eller long. Det bruges i form af arrays og pointers til at få programmet til at

kører og gemme de nødvendige oplysninger. Der bliver også brugt filer, men disse er udlykkende til at læse og skrive koden til ekstrene filer således at nødvendigt data vil blive gemt.

4 Resultater

4.1 Output



Figur 1: Plottet resultat af programmet

Plottet i figur 1 viser resultaterne af den implementerede QRS algoritme. Det ses heraf at der har været relativ succes med implemteringen. De noteres dog også, at det sidste punkt inden der sker et hjertesvigt ikke registreres. Dette skyldes at der aldrig kommer en peak over Threshold1 efter hjertesvigtet, derved initialiseres der aldrig en searchback der finder den sidste peak.

4.2 Performance analyse

Programmets tidsforbrug i forhold til de forskellige funktioner er fordelt procentvist som ses i figur 4 i *Bilag 6.1*. Denne er baseret på, at programmet har kørt 1.000 gange, hvilket svarer til datamængde på 100.000.000 datapunkter. Her ses at det uden tvivl er filtrene, specielt `movingWindowIntegrationFilter`, `highPassFilter` og `lowPassFilter`, som optager den største del af køretiden. Dette hænder pga. at disse klasser skal gennem kører større loops og derved også vil komme til at optage mere af tiden. De næste metoder som bruger en del tid er `peakDetection` og `peakDetermination`. Dette er også logisk, da disse er funktioner som skal kører til alle datapunkterne. Ligesom med filterene gennemløber `peakDetermination` et loop for at opfylde et array af integers.

Koden er compiled uden nogen optimereingsflag, hvilket betyder at compileren ikke har forsøgt at optimere koden overhovedet. Derfor ville det nok være bedre

at benytte sig af -O2 flaget, idet compileren så vil forsøge at producere mindre og hurtigere kode. -O3 flaget kunne også benyttes, dog giver dette ofte større binary uden at garantere hurtigere kode. -O3 benytter også mere humkommelse til at compile, hvorimod -O2 kun benytter en anelse mere hukommelse, men samtidig ofte producerer bedre kode.

Selve koden fylder omkring 450 linjer. Dette er dog inklusiv whitespace imellem linjer. Idet der også ofte indgår loops i koden ville linje tal ikke give et særlig godt overblik over hvor meget koden egentlig fylder. Kodens .exe fil fylder 55 kb efter compilation. Denne størrelse ændrer sig ikke medmindre den compiles med et -Os flag der compiler med henblik på at formindske fil størrelse. Det viser sig faktisk, ifølge gprof, at movingWindowIntegration ikke længere er et problem når programmet compiles med -O2 flaget. Nu er det filter der har længst køretid highPassFilter. Hvorfor dette er tilfældet vides ikke.

5 Konklusion

Alt i alt er programmet implementeret tilfredstillende. Det finder de peaks der er i testdataene, og searchbacker korrekt. Den kunne dog nok have været implementeret bedre for at mindske køretiden og tage højde for enkelte edgecases. Specielt kunne det nok betale sig at implementere nogle af filterene som hardware løsninger. Endvidere kunne filterene nok implementeres bedre idet deres opbevaring af data sker med statiske tal i stedet for pointers og de ikke benytter sig af cirkulære arrays.

Referencer

[1] Profiling with gprof *http://yzhong.co/profiling-with-gprof-under-64-bit-window-7/*

[2] Options That Control Optimization *https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html*

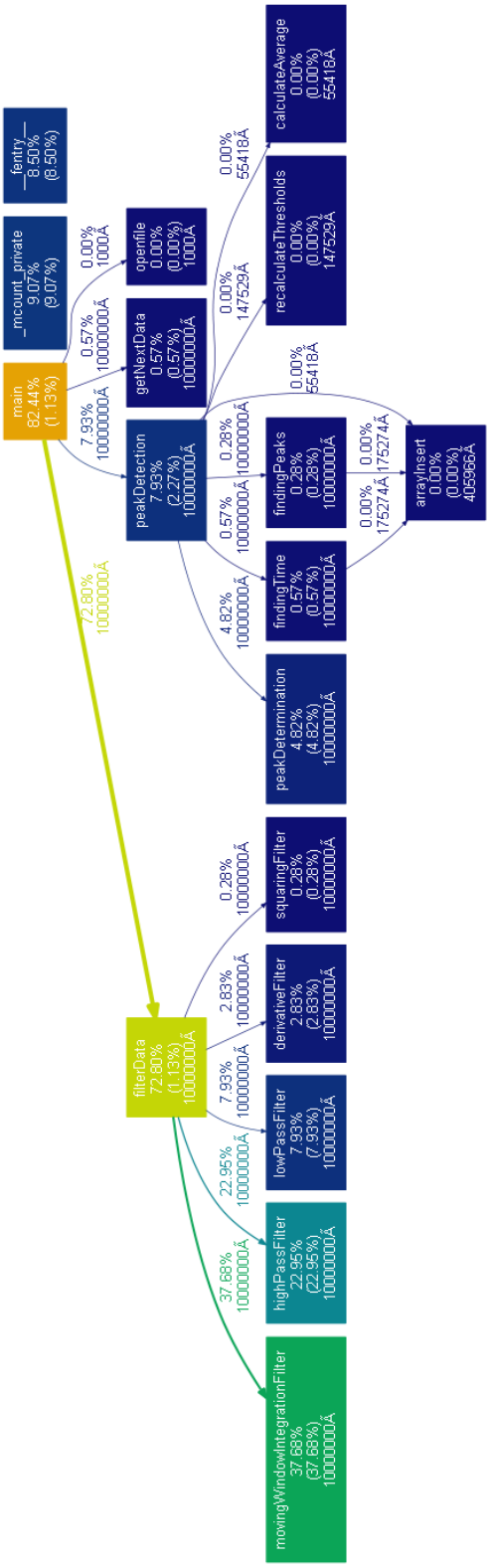
[3] GitHub *https://github.com/s154302/Assignment1*

Emilie Isabella Dahl (s153762)

Alexander Kristian Armstrong (s154302)

6 Bilag

6.1 Bilag 1



Køretiden af programmet. Figuren viser funktionens navn, dens totale procentvise køretid, dens egentlige procentvise køretid og antal kørsler.

6.2 Bilag 2

Svar til Exercise 1:

```
#include <stdio.h>

int main(){
        static const char filename[] = "ECG.txt";
        FILE *file = fopen(filename, "r");
        int value = 1, max_value;
        fscanf(file, "%d", &max_value);

        while(!feof(file)){
                fscanf(file, "%d", &value);
                if(value > max_value){
                        max_value = value;

                }
        }
        printf("%d", max_value);


        return 0;
}
```