Assignment 1

Simon Altschuler
s123563

Markus Færevaag
s123692

30.09.2013

Indhold

1 Introduktion

Denne opgaver omhandler processering af signaler fra et Elektrokardiogram apparat (herefter ECG). Formålet er omdanne de rå signaler til filtreret og fortolket data, som kan bruges til at måle puls og spænding, og advare om forestående problemer hos patienten.

Data fra ECG hardwaren er simuleret ved at læse linier af tal fra en tekstfil, således at rigtig data i princippet kunne bruges uden at ændre andet end funktionen der henter et nyt sample.

2 Problemstilling

Udfordringen i denne opgave er at implementere signalfiltre og detektere egenskaber effektivt og struktureret, samt at præsentere dataen for brugeren på en hensigtsmæssig og brugbar facon.

Da datasættene har op til flere millioner samples, er det vigtigt at implementere datastrukturer og algoritmer på en måde som kan håndtere arbitræt store datasæt, mens ydeevnen forbliver acceptabel.

Det er ydermere afgørende, at algoritmen giver korrekte resultater, da diagnosticeringen af en patient i modsat fald kan være forkert og lede til forkerte eller manglende beslutninger.

2.1 Funktioner

Følgende er de overordnede funktioner som programmet skal udføre. Vi kigger her på hvad de hver især skal udføre og hvad der er vigtigt at fokusere på.

2.1.1 Sensor

Sensoren kobles til patienten og laver et digitalt signal udfra den elektriske aktivitet fra hjertet. Dette simulerer vi ved at læse linje for linje af en fil indeholdende forskellige test data. Dette er simuleringen af hardwaren, og derfor bør denne funktion også have en forsinkelses mekanisme for at emulere det rigtige tidsinterval mellem samples (4ms mellemrum). Funktionen skal hente data on-the-fly og altså ikke indlæse alt i memory, da det vil være uhensigtsmæssigt for store datasæt, og det ydermere er et krav i den stillede opgave.

2.1.2 Filtre

Der er en række af filtre, som hver skal udføre én bestemt filtrering, eller transformation, af dataen. De er linært afhængige af hinanden, hvilket vil sige at de skal udføres i en bestemt rækkefølge og den næste afhænger af den forrige. Filtrene skal bruge tidligere målte samples og tidligere filtreret data, hvilket vil sige at der skal gemmes data. Da det er uhensigtsmæssigt, både hukommelses- og ydelsesvis, skal der udvikles en datastruktur som holder forbruget til et minimum.

2.1.3 Peak detektion

Detektion af peaks er en kompliceret algoritme, som bruger det filtrede data til at måle amplitude og og frekvens af patientens puls. Funktionen skal videregive dens resultater til display og output funktionerne, så det er vigtigt at denne data er tilgængelig på en brugbar facon.

2.1.4 Output

Det skal være muligt at skrive den resulterende data til en ekstern data fil, til brug for analyse, f.eks. i form af plots og grafer.

2.1.5 Display

Måden dataen bliver præsenteret er en afgørende faktor for brugervenligheden af programmet. Derfor har vi i tillæg til output til en fil, valgt at implementere en display feature som efterligner en rigtig ECG maskines output. Dataen behandles i realtid, med mulighed for at skalere tiden op og ned, og der vises essentielle data såsom puls (BPM), R-peak værdier, antal missede peaks, osv. Ydermere vil vi vise en graf af den rå data fra hardwaren i form af en simpel graf, sådan at pulsslagene kan ses visuelt.

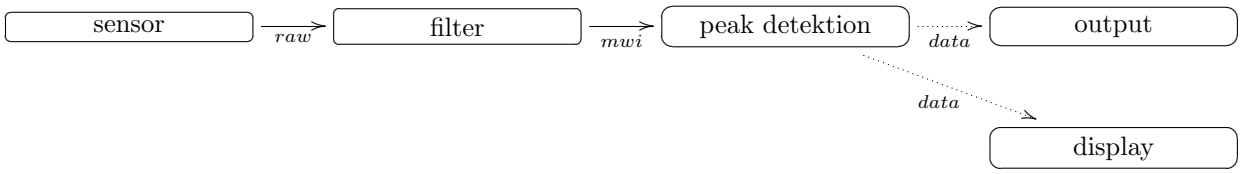
3 Design

Programmet tager en række options som styrer output, display og data. De er som følger:

`-f` *file*	Angiv hvilken fil der skal bruges som testdata input
`-o` *file*	Angiv at data ønskes gemt som csv, argumentet er filen
`-l` *uint*	Angiver antal samples der skal køres, sættes normalt til antal linier i testdata filen
`-d`	Angiver at der ønskes visuel repræsentation af programmets kørsel
`-t` *float*	Tidsskalering, 2 = dobbelt hastighed, 0.5 = halv hastighed osv.

De forskellige dele af programmet skal hver især initialiseres og destrueres ved programstart og -slut. Derfor har de fleste features en `init` og `destroy` funktion, i hvilken de udføre de relevante operationer, hvad enten det være at initialisere et array eller åbne en fil. Dette hjælper på isolering af ansvar, og er med til at holde `main` funktionen clean.

3.1 Arkitektur



Figur 1: Overordnet program flow (stiplede linjer markere valgfri eksekvering)

Programmet er hovedsageligt opdelt i filerne `sensor.c`, `filter.c`, `peak_detect.c`, `output.c` og `display.c`, som svarer direkte overens med de ovenfor beskrevne funktioner. Figur 1 viser hvordan dataen går gennem rækken af komponenter, fra det rå data til det processerede resulterende data.

Dette flow giver en overskuelig inddeling af programmet da hver fil, i henhold med navnet, er ansvarlig for en isoleret og veldefineret opgave. Det er også med til at simulere en realistisk hardware struktur hvor komponenterne ville være implementeret hver for sig og videregive information på lignende vis.

Følgende er kort om hvordan arkitekturen indenfor hvert komponent er designet.

3.2 Sensor

Innlesningen av bildet skjer paa den maate at den best skal simulere signalet fra en ECG maskin. Derfor leses test dataen inn og behandles linje for linje, i motsetning til aa lese inn hele filen. Dette ville konsumere meget hukommelse, i tillegg ikke viser en god simulering av hvordan det faktisk fungerer.

3.3 Filtre

Filtrene er implementeret hver for sig, således at hvert filter er en separat funktion. For nemt at kunne anvende disse funktioner har vi samlet dem i en funktion kaldet `apply_all_filters`. Denne tager en rå værdi fra sensoren, filtrerer denne med `low_pass`, `high_pass`, `derivate`, `squaring` og `moving_window_integration`, for så at returnere `mwi` køen. På denne måde er de specifikke filtreringsfunktioner abstraheret væk.

3.4 Peak detektion

Vores peak detektion er en forholdsvis kompleks algoritme der bruker de senest filtrerede værdier. Der eksponeres én funktion, `update_peak` som bruger MWI dataen og den nuværende tid. Algoritmen returnerer en `peak_update` struct, der beskriver algoritmens resultater, og som danner grundlag for det den endelige repræsentation til brugeren.

3.5 Output

Skrivning til en fil gøres med `update_output` funktionen, som tager en `peak_update` struct. Der skrives til den angivne fil én linie per data cyklus. På denne måde kan vi meget let inkludere eller eksludere data til outputtet. Filen skrives i formattet CSV da det er meget simpelt og let at håndtere.

3.6 Display

Ligesom output håndteres visuel præsentation af data ved kald til `update_display`, som også tager en `peak_update` struct. Her opdateres skærmen med den ny data og grafen tegnes på ny. Når visuel præsentation er slået til (med `-d`) pauser programmet og afventer et keyboard input, før det afslutter, så man kan nå at se de sidst målte data.

4 Implementering

4.1 Hjælpefunktioner

Udover de benævnte komponenter, som hver har sin egen fil, har vi skrevet nogle hjælpefunktioner til array (`array_utils`) og matematiske operationer (`math_utils`). De matematiske operationer afgrænser sig til `min` og `max` funktioner. Array funktionerne abstraherer opgaven at prepende et element til et array, samt eksponerer en `array_average`, der som navnet hentyder udregner et gennemsnit af værdierne i et array.

4.2 Datastrukture

De fleste af programmets interne dele benytter fixed-size FIFO[1] køer. Vi prepender, altså sætter ind i arrayets første position, da dette gør det meget nemmere at bruge tidligere målt data. Hvis man eksempelvis skal bruge X_{n-7} fra MWI skrives `mwi[7]`, hvilket er pænere, nemmere og mere overskueligt end at skulle holde styr på arrayets længde.

Køerne er implementeret med arrays, og når der prependes shiftes alle elementer i arrayet én plads til højre, og efterfølgende sættes element på position 0 til den nye værdi. På den måde bliver sidste element overskrevet af det næstsidste og vi undgår hukommelses leaks.

4.3 Sensor

Sensoren implementeres i tre dele. Første del initialiserer sensoren, `init_sensor`. Der gives et filnavn som argument, og denne fil åbnes, med read permissions.

Derefter kan man kalde funktionen `get_next_data`, som læser én linie i filen som en integer og returnere denne. Sensoren gemmer intet data, det overlades til de andre komponenter som skal bruge det.

Ved programmets exit kaldes `destroy_sensor`, som lukker input filen.

4.4 Filtre

Hver af de fem filtre er implementeret i deres egen funktion, som tager data fra den forrige filterfunktion og returnerer ny filtreret data. Selve filter algoritmene er implementert i følge formlene givet opgaven.

Figur 2: Respektive filterfunktioner

Ved filtrering af dataen skal der bruges et array for hvert filter i hvilket resultaterne af det respektive filter bliver gemt. Ydermere skal man være omhyggelig med i hvilken rækkefølge de eksekveres, da de afhænger af hinanden. Dette bliver hurtigt kompliceret, og derfor har vi ligesom i de andre komponenter implementeret en `init_filters` metode som initialisere de forskellige arrays. Vi har `typedef`'et et array med buffer størrelsen (til navnet `list`) for at gøre det mere overskueligt hvad de forskellige arrays er.

Længden på disse lister er 33, da visse filtre skal bruge data fra 32 samples tilbage, plus de skal holde det nuværende sample.

Selve filtreringen foregår ved at kalde `apply_all_filters` med den rå data som argument, og denne funktion sørger for at kalde filtrene i korrekt rækkefølge og returnerer `mwi` listen.

De specifikke filterfunktioner er deklareret `static` da de ikke har nogen værdi udenfor `filters.c` filen.

4.5 Peak detektion

Peak detektion er implemeneret efter de givne instruktioner. Vi har brugt følgende start værdier for variabler nødvendigvis må initialiseres før algoritmen kan fungere:

threshold1 = 2500 Den omtrentlige grænse for hvad der kan klasificeres som en R-peak

[1] First In - First Out

rr_high = max	Max (sat til 99999), da første peak skal falde inden for high og low
spkf = 5000	Den omtrentlige R-peak værdi
npkf = 1000	Den omtrentlige gennemsnitlige peak værdi

For at fundne peaks og anden relevant data nemt kan videregives til næste del af programmet, har vi lavet en struct `peak_update`, som indeholder det relevante data, såsom den seneste R-peak værdi, puls (`average1`), tid, seneste MWI værdi, etc. Denne struct videregives til både output, og display delene af programmet, som så kan bruge det data de har brug for.

4.6 Display

Det visuelle interface er udviklet ved brug af biblioteket `ncurses`[2]. Det tager en `peak_update` og viser essentielt set blot dens indhold. For at tegne puls grafen har `display` funktionen sin egen kø, for at den kan gemme de sidste 100 data punkter. Disse tegnes med en simpel algoritme der udregner række og kolonne position for et data punkt. Den er meget simpel og efterlader mulighed for megen forbedring, men den er stadig et fint proof of concept.

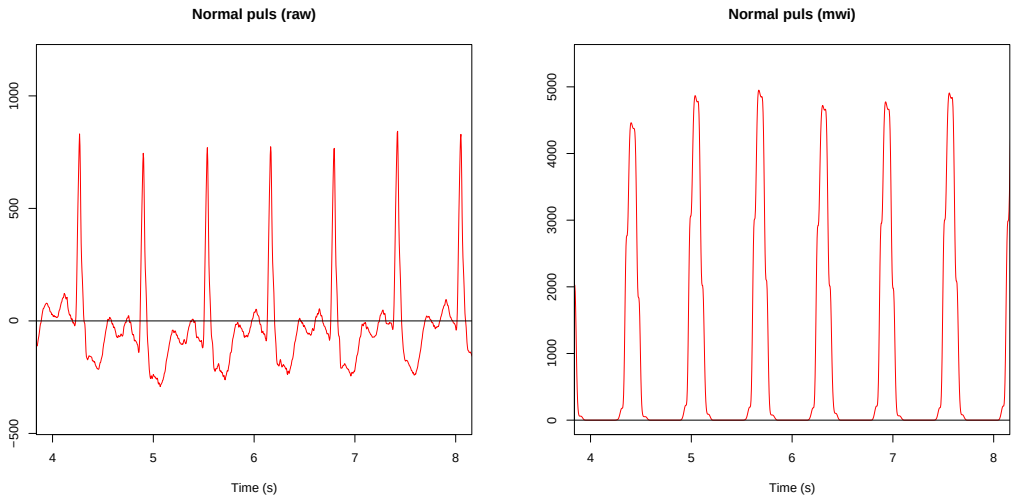
4.7 Output

Programmet kan skrive data til en given fil i CSV formattet. Dette kan så kan plottes til en graf med for eksempel programmeringssproget `R` (hvilket vi har gjort). Ligesom `sensor` funktionen, åbner `output` filen, der skal skrives til i `init_output` funktionen. I `destroy_output` lukkes filen igen.

5 Resultater

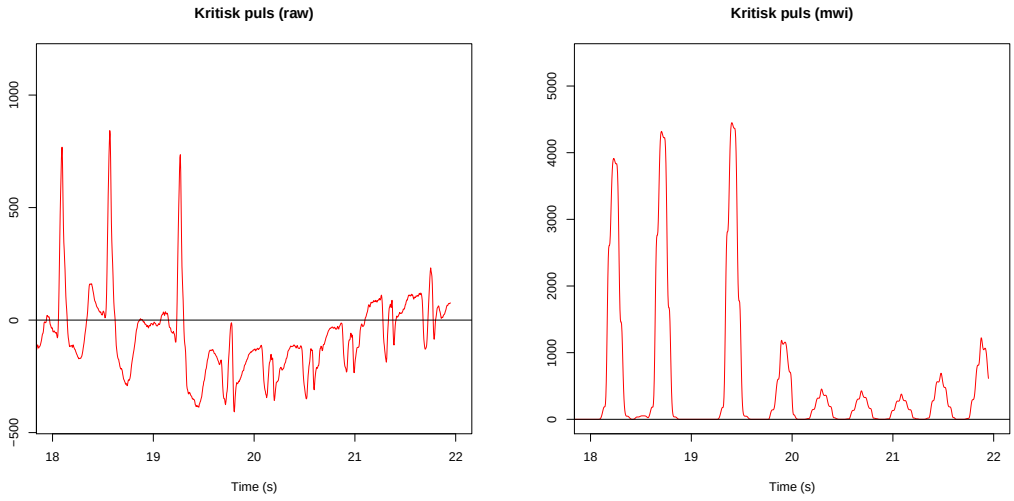
Nedenfor i figur 3 vises den rå data og den relaterede filtrede (mwi) data. Det ses tydeligt hvordan algoritmen gør det utroligt meget nemmere at detektere amplitude og frekvens, da kurverne er blødere og næsten intet "flimmer" har.

[2] `http://www.gnu.org/software/ncurses`



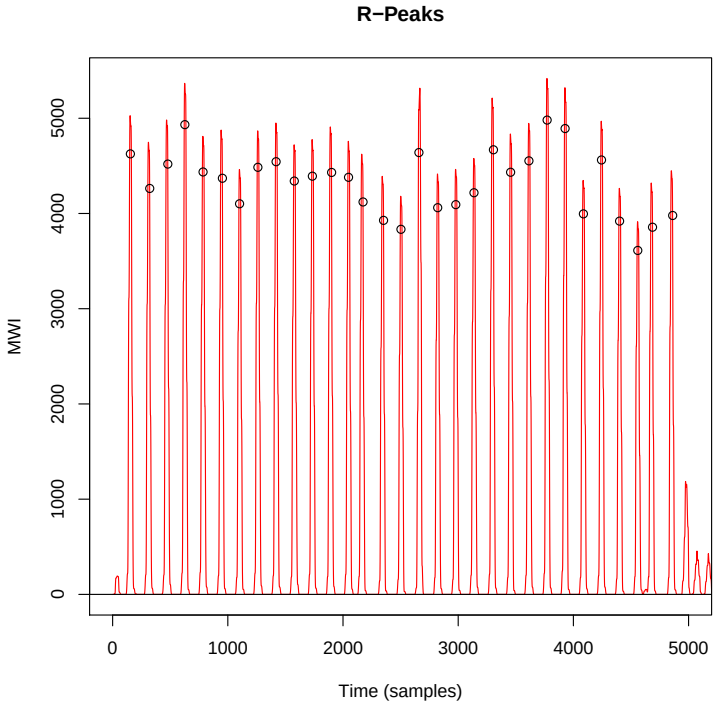
Figur 3: Normal puls, raw data og det filtrerede MWI

I figur **??** ses hvad der sker når patientens tilstand bliver kritisk. Pulsen stiger og slagkraften falder drastisk. Dette detekteres omgående i `peak_detect` og der kan vises advarsler.



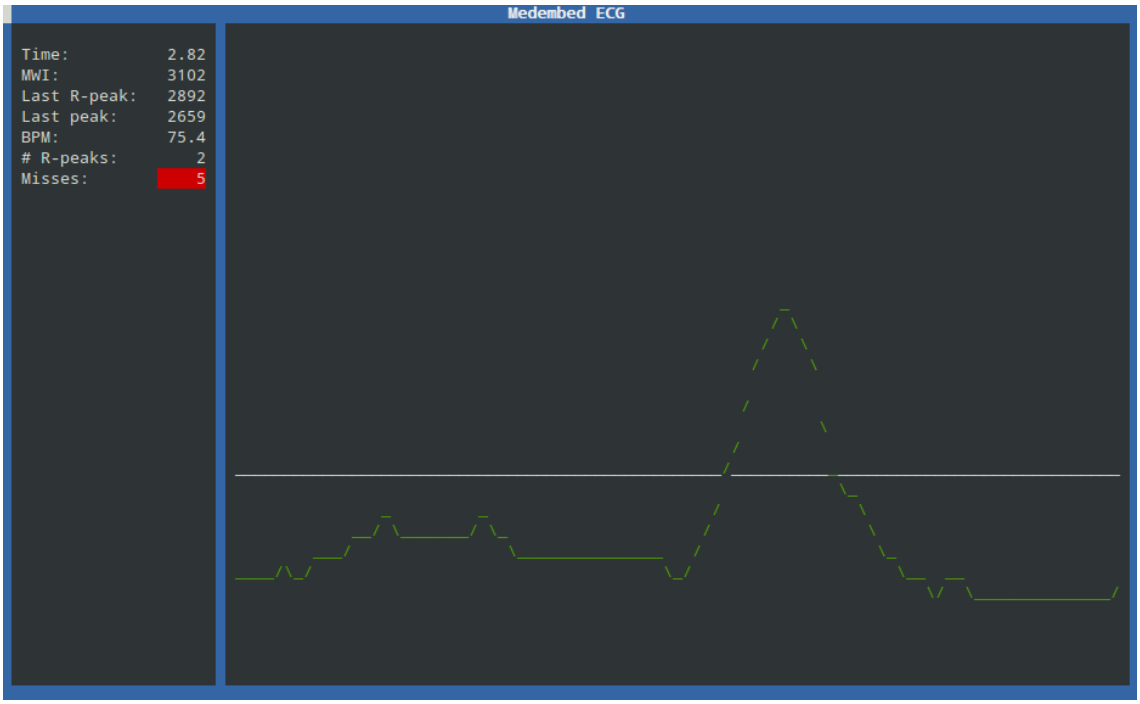
Figur 4: Begyndende kritisk puls, raw data og det filtrerede MWI

Vi har implementeret algoritmen korrekt, i den forstand at den finder alle R-peaks korrekt. Den egentlige værdi af de funde peaks ligger en smule, dog konsistent, under toppunkterne på `mwi` dataen. Dette er vist i figur 5.



Figur 5: Fundne R-Peaks vist som cirkler på MWI data

Det visuelle output som vises når `-d` bruges, er vist i figur 6. Det ses hvordan rød farve er brugt til at markere alarmerende omstændigheder.



Figur 6: Visuelt output fra programmet, hvor det ses ud for "Misses" hvordan advarsler markeres

6 Profiling

I det følgende er der lagt vægt på selve QRS algoritmen og ikke output. Derfor er programmet kørt uden nogen form for output, hverken til fil eller visuelt, da vi mener at det er irrelevant for måling ydeevnen.

6.1 Flaskehalse

Vi har profilet programmet med `gprof` og har observeret at det er array operationerne som udgør den klart dominerende del af køretiden. I et uoptimeret build udgør funktionen `prepend_array_int` næsten 60.5% af CPU tiden, hvor den i et optimeret build udgør omkring 30.5%. Med denne information kunne man skære utrolig meget af køretiden ved at omskrive array operationerne til en hurtigere metode.

Vi fandt også at `apply_mwi` filteret tager 33.5% af køretiden i det uoptimerede build, og `apply_all_filters` hele 66% i det optimerede. Det betyder at compileren har inlinet funktionen `apply_mwi` (og formodentlig også alle de andre filter funktioner) i dens optimeringsfase således at filter instruktionerne bliver kørt direkte i `apply_all_filters`. Det er smart da der elimineres et funktions kald, men det kan i visse situationer øge kodestørrelsen. Det bør nævnes at vi kunne have gjort dette eksplicit ved at benytte keywordet `inline` på filter funktionerne, da det i så fald også ville være inlined i det uoptimerede build.

Nedenfor ses det dominerende `gprof` output af det uoptimerede build (kørt med datasættet med $10800K$ samples)

% time	cumulative seconds	self seconds	calls	self ns/call	total ns/call	name
60.55	5.37	5.37	64800000	82.89	82.89	prepend_array_int
33.52	8.34	2.97	10800000	275.31	275.31	apply_mwi

```
  1.58        8.48       0.14 10800000      12.98      13.90   update_peak
```

Nedenfor ses det dominerende `gprof` output af det `O3` optimerede build (også kørt med datasættet med $10800K$ samples)

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 66.03      2.38     2.38                             apply_all_filters
 30.66      3.49     1.11 64800000     17.08    17.08  prepend_array_int
  1.66      3.55     0.06                             update_peak
```

6.2 Kodestørrelse

Kodestørrelsen varierer afhængig af hvad man undersøger. Først og fremmest ignorerer vi ting som antal linier og karakterer i kildekoden. Det giver ikke mening at se på disse ud fra et hardware mæssigt perspektiv, da stil og præferencer spiller en stor rolle, samt at kildekoden i sig selv er mere eller mindre irrelevant for det endelige resultat efter kompilering.

	-O0	-O1	-O2	-O3	-Os
bin	20380	19799 (97.15%)	19799 (97.15%)	19847 (97.38%)	19794 (97.12%)
.text	6072	4552 (74.97%)	4776 (78.66%)	4808 (79.18%)	3976 (65.48%)
update_peak	1669	1296 (77.65%)	1258 (75.37%)	1258 (75.37%)	1130 (67.71%)

Tabel 1: Kodestørrelse (bytes) af forskellige aspekter og optimeringer

Vi har målt kodestørrelse på 5 forskellige builds, med compiler flags O0-O3 og Os. O0-O3 optimerer for køretid og vi forventer dermed ikke at kodestørrelsen nødvendigvis er lavere. Derimod optimerer Os netop for størrelse så her burde vi kunne se tydelig forskel.

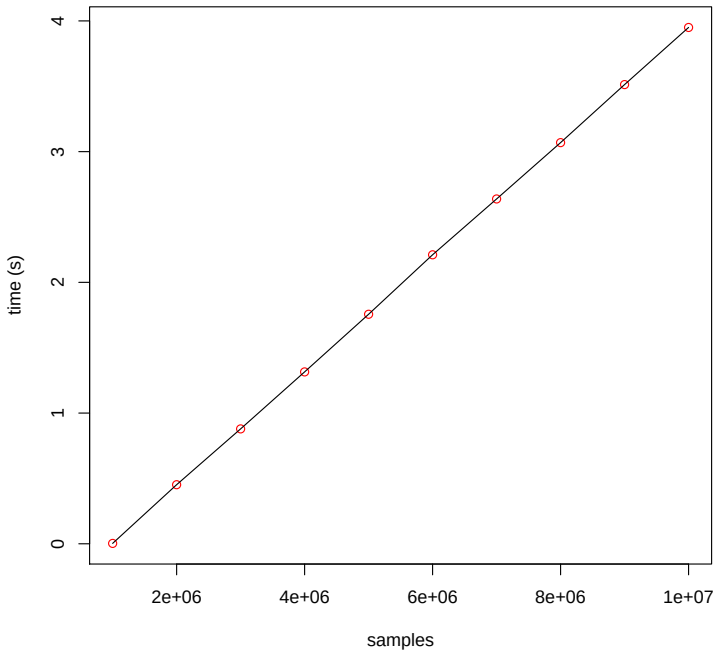
Tabel 1 viser de målinger vi er kommet frem til, og hvor stort det er i forhold til O0 buildet. "bin" er simpelthen størrelsen på den binary der bliver kompileret. ".text" er målt vha. unix programmet `size`[3] og måler størrelsen af instruktionssektoren af en binary, som er der hvor de egentlige instruktioner som CPU'en vil udføre ligger. "update_peak" er størrelsen af `update_peak` funktionen og er målt vha. unix programmet `nm`[4], som viser information om forskellige symboler i en object fil (binary). Vi målte størrelsen af `update_peak` funktionen fordi det er den største og derfor kan give den bedste sammenligning.

Det er interessant at den samlede binarys størrelse kun variere med $\tilde{5}00$ bytes, selv i Os buildet. Det skyldes formodentlig at der ikke er specielt meget kode samlet set, hvilket bekræftes af at instruktionssættet udgør $\tilde{2}0\%$ af de totale bytes. I O3 buildet er koden generelt større end i O1 og O2, hvilket ikke er bemærkelsesværdigt, men det er interessant at `update_peak` funktionen fylder mindre, og et tegn på at compileren er i stand til optimere kompakt og indviklet kode til noget langt mere optimalt.

6.3 Køretid

Algoritmen er af natur linær, den foretager de nøjagtig samme instruktioner (bortset fra init og destroy funktionerne), blot flere gange ved større input. Derfor forventer vi en køretid på $O(n)$. Vi har dog alligevel foretaget en måling og resultatet ses i figur 7. Det ses at køretiden meget præcist er linær.

[3] `http://unixhelp.ed.ac.uk/CGI/man-cgi?size`
[4] `http://unixhelp.ed.ac.uk/CGI/man-cgi?nm`



Figur 7: Fundne R-Peaks vist som cirkler på MWI data

6.4 Metode til målinger

Vi har skrevet to `zsh`[5] scripts til at hjælpe med målingen af performance. Disse er inkluderet her, men af relevanshensyn ikke forklaret nærmere

Til måling af køretider

```
for ((i = 0; i < 10000000; i = i + 1000000)); do
    time bin/ecgo3 -f test_data/ECG10800K.txt -l $i;
done;
```

Til måling af kodestørrelser

```
for x (0 1 2 3 s); do
    size -A bin/ecgo$x | grep .text;
done;
```

6.5 Energiforbrug

Vi vil prøve at måle hvor meget energi der forbruges under kørsel af algoritmen. Vi har brugt en Intel i7 3537U CPU, og den har et energiforbrug på $17W$[6]. Den samlede køretid af $10800K$ samples var $4.734sek$. Dermed får vi følgende forbrug:

[5] `http://www.zsh.org`
[6] `http://www.intel.com/content/www/us/en/processors/core/core-i7-processor/Corei7Specifications.html`

$$Energiforbrug_{cpu} = 17W \tag{1}$$
$$Samples = 10800000 \tag{2}$$
$$Tid_{total} = 4.734s \tag{3}$$
$$Tid_{100k} = \frac{Tid_{total}}{Samples} \cdot 100000 = 0.04383s \tag{4}$$
$$Energi_{100K} = Energiforbrug_{cpu} \cdot Tid_{100k} = 0.7452 \; watt \; sekunder \; (Joule) \tag{5}$$
$$\tag{6}$$

Der bliver altså brugt 0.7452 Ws per $100K$ processerede samples. Hvis vi omregner til pris i Danmark og sætter prisen til $2.1\frac{kr}{kWh}$[7] fås:

$$Pris_{kWh} = 2.1\frac{kr}{kWh} \tag{7}$$
$$Energi_{100K} = 0.7452Ws = 2.07 \cdot 10^{-7}kWh \tag{8}$$
$$Pris_{100K} = Energi_{100K} * Pris_{kWh} = 4 \cdot 10^{-5}øre \tag{9}$$

Det koster altså 0.00004 øre at processere 100.000 samples.

7 Konklusion

I dette projekt har vi arbejdet med et indlejret system i form af en ECG maskine. Vi har simuleret input fra en sensor, filtreret og behandlet denne data, for så at præsentere det til brugeren eller gemt den sådan at den kan blive videre analyseret.

Vi har bl.a. lagt vægt på presentationen af dataen. En faktisk ECG maskine har en skærm, hvorpå den præsenterer den resulterende data. Hvis den eneste mulighed for at vise informationen som bliver behandlet er at skrive den til konsollen, giver det et meget ubrugervenligt interface. Informationen ændrer sig meget hurtigt, så for at skabe et komplet og brugbart system har vi gjort et forsøg på at lave en mere realistisk løsning, vha. bibloteket `ncurses`. Med dette har vi været i stand til at udvikle et overskuelig og informationsrigt interface.

Vi har fået god træning i at strukturere `C` kode, og har i den forbindelse udviklet systemet modulært og enkapsuleret. Vi har lært hvordan forskellige scoping teknikker (såsom static, extern) kan udnyttes til dette formål.

Vi har set hvordan kompilere kan udnyttes til at optimere til specifikke formål. Yderligere har vi lavet uddybende analyse af programmets ydeevne. Køretiden er acceptabel og programmet bruger forholdsvist få ressoucer, hvilket betyder at det er egnet til en eventuel microprocessor løsning eller decideret hardware implementation. Programmet kører fejlfrit og giver de korrekte resultater i følge de givne facit. Med det ser vi os godt tilfredse med resultatet.

[7] `http://www.elpristavlen.dk/Elpristavlen/Soegeresultat.aspx?kwh=2000&postnr=2500&netcompany=DONGnet&customer-group=private&ratetype=FlatRate`

8 Kørsel af programmet

En optimeret binary kan bygges ved at køre `make ecgo3` og derefter eksekvere programmet med de ønskede argumenter. F.eks. for at bruge "ECG.txt" og se den visuelle repræsentation af programmet i normal tid, kan det køres således: `ecgo3 -f ECG.txt -d -t 1`. Se 3 for detaljer vedr. argumenter.