DET TEKNISK-NATURVITENSKAPELIGE FAKULTET

BACHELOROPPGAVE

Studieprogram/spesialisering: Bachelor i ingeniørfag / Automatisering og elektronikkdesign	Vårsemesteret 2022 Åpen
Forfatter: Håvard Sørbotten	
Fagansvarlig: Trygve Christian Eftestøl Veileder: Trygve Christian Eftestøl	
Tittel på bacheloroppgaven: Python implementering av filter for fjerning av kompresjonsstøy fra behandling av hjertestanspasienter	
Studiepoeng: 20	
Emneord: Python, MATLAB, Hjerteinnfarkt	Sidetall: 36 + vedlegg A Stavanger 15. mai 2021

Sammendrag

Denne oppgaven tar for seg å implementere ett LMS filter basert på eksisterende MATLAB kode til Python basert kode. Python er ett objektorientert språk som ligger åpent, i motsetning til MATLAB som krever en lisens. Ved å implementere programmet i Python vil dette kunne gjøre det enklere å utvikle programmet videre. Under implementeringen er det bevisst valgt å holde strukturen i den nye Python koden så lik som originalen, slik at den delen som filtrerer vil ha tilnærmet lik oppbygging som resten av hovedprogrammet.

Resultatet etter denne implementeringen er et Python program som tar inn data fra MATLAB og returnerer et ferdig filtrert elektrokardiogram(EKG) samt de estimerte artifaktene. Python programmet testes ved å bruke samme inngangssignalene som i MATLAB og deretter sammenligne de returnerte utgangssignalene. Denne testen viser at de to ulike programmene gav like utgangssignaler.

Forord

Denne oppgaven markerer slutten på min 5 års periode som student og arbeidstaker. Det viste seg å være betydelig verre å kombinere fulltidsjobb og deltidsstudier enn først antatt. Hvem hadde trodd at å ha en 166% stilling skulle være så tidkrevende. Tiden min på Universitetet i Stavanger har vært svært lærerik, men også utfordrende. Jeg ønsker å takke veileder Professor Trygve Eftestøl, for sitt engasjement og gode råd underveis. Selv om våre ukentlige møter gjennom denne perioden var satt opp til å vare en time tror jeg ikke vi en eneste gang har hatt et så kort møte. Jeg må også få takke kolleger som har vist forståelse for at jeg til tider har vært ganske stresset.

Stavanger, Mai 2022
Håvard Sørbotten

Innhold

Sammendrag i

Forord ii

1 Innledning 1

2 Bakgrunn 2

2.1 Hjerte- og lungeredning 3

2.1.1 Utføre HLR 3

2.1.2 Hjertestarter 3

2.1.3 EKG 3

2.2 Adaptivt filter for demping av artifakter 4

2.2.1 Signalmodell av artifaktene 4

2.2.2 Filteret for fjerning av artifakter 7

2.3 MATLAB-implementering av filteret 9

2.3.1 Flytskjema med MATLAB funksjoner 9

2.3.2 Hovedfunksjoner 11

2.3.3 Hjelpefunksjoner 17

3 Metode 21

3.1 Teoretisk bakgrunn for filteret 21

3.2 Python implementering 22

3.2.1 Isolere filteret fra hovedprogrammet 22

3.2.2 Isolering av enkeltfunksjoner 23

3.2.3 Fremgangsmåte 24

4 Resultater og konklusjon 27

4.1 Sammenligning MATLAB og Python 27

4.2 Resultater 30

4.3 Diskusjon 33

4.4 Konklusjon 34

Bibliografi 36

Vedlegg 36

A Matlab og Python kode 37

A.1 Tilleggspakker 37

A.2 Smoothstep 38
A.2.1 Smoothstep i Matlab 38
A.2.2 Smoothstep i Python 38
A.3 mCC ProcessPeaks 39
A.3.1 Process Peaks i Matlab 39
A.3.2 Process Peaks i Python 39
A.4 eComp 40
A.4.1 eComp i Matlab 40
A.4.2 eComp i Python 41
A.5 aRef 42
A.5.1 aRef i Matlab 42
A.5.2 aRef i Python 43
A.6 pRef 44
A.6.1 pRef i Matlab 44
A.6.2 pRef i Python 45
A.7 gRef Signal 47
A.7.1 gRef Signal i Matlab 47
A.7.2 gRef Signal i Python 47
A.8 Algorithm LMS Vectoriced 48
A.8.1 Algorithm LMS Vectoriced i Matlab 48

A.8.2 Algorithm LMS Vectoriced i Python 49

A.9 Artifact Removal 50

A.9.1 Artifact Removal i Matlab 50

A.9.2 Artifact removal i Python 50

Kapittel 1

Innledning

Det skjer omtrent 3000 hjertestanser i Norge utenfor sykehuset der cirka to av tre tilfeller skjer i hjemmet[6]. Når uhellet først er ute er det viktig å sette igang livreddende tiltak så rask som mulig, altså hjerte- og lungeredning(HLR). Hjernecellene tåler bare få minutter uten oksygentilførsel før de begynner å dø, men ved å utføre god HLR gir en pasienten mer tid. Det er avgjørende at pasienten har en uavbrutt behandlingskjede fra hjertestansen til ankomst sykehus. Her er tid den viktigste begrensningen[5]. Når det er koblet opp en hjertestarter til pasienten er man i dag avhengig av avbrudd for å gjennomføre analyse av hjertet for å sjekke om det er sjokkbar rytme. Det er dette oppholdet denne oppgaven dreier seg om, målet er at man skal kunne utføre analyse samtidig som man driver med HLR, på denne måten vil det føre til mindre avbrudd som vil gi pasienten en økt sannsynlighet for å overleve uten varige men. For å klare dette er hjertestarteren avhengig av å ha ett elektrokardiogram(EKG) den kan analysere som er fritt for artifakter. Til dette er det fra tidligere utviklet ett filter som kan fjerne artifakter generert av kompresjoner [3]. Denne oppgaven tar for seg å oversette dette filteret fra MATLAB kode til Python kode.

Rapporten er bygd opp på en måte som tar for seg bakgrunnen til hvordan filteret er designet og hvordan oppgaven er løst med å implementere dette i Python. Deretter avsluttes det med en gjennomgang av resultater og en konklusjon.

Kapittel 2

Bakgrunn

Dette kapitellet tar for seg bakgrunnstoffet for å få en bedre forståelse av koden. Det blir først gitt en generell gjennomgang av hjerte- og lungeredning, før teorien som ligger til grunn for design av filteret blir gjennomgått. Kapitellet avsluttes med hvordan dette er implementert i MATLAB.

Det har fra tidligere blitt utviklet en programvare i MATLAB som leser inn data fra en hjertestarter, detekterer kompresjoner og utfører filtrering av kompresjonene ved hjelp av et ”Least Mean Square” filter [3], heretter kalt LMS filter. Grunnen til at en ønsker å kunne filtrere bort kompresjoner er å kunne utføre livreddende førstehjelp HLR kontinuerlig. Selve programmet er stort og komplekst, det inneholder mange funksjonaliteter som denne oppgaven ikke dreier seg om. Denne oppgaven dreier seg om den delen som tar for seg artifaktfjerningen, koden som realiserer LMS-filteret. Hvordan dette er løst blir beskrevet i avsnitt 2.2.2

Hovedprogrammet er ikke offentlig publisert, dersom en ønsker mer informasjon om det må Trygve Eftestøl kontaktes direkte.

2.1 Hjerte- og lungeredning

Her vil det bli gitt en kort innføring i HLR samt hjertestarteren og EKG.

2.1.1 Utføre HLR

Det første man gjør før man starter HLR er å se etter livstegn, om man ikke ser livstegn skal man rope på hjelp og varsle 113. Ved en hjertestans er det viktig å starte hjerte-lungeredning så raskt som mulig. Sjansen for å overleve synker for hvert minutt som går uten god HLR. Ved hjerte og -lungeredning til voksne skal en gi 30 brystkompresjoner og 2 innblåsinger, kompresjonene skal være omtrent 5-6cm dype og gjennomføres med en takt på 100-120 ganger i minuttet. Etter de første 30 kompresjonene skal en gi to innblåsinger med munn-til-munn metoden[2].

2.1.2 Hjertestarter

Hjertestarter også kjent som defibrillator brukes ved hjelp av to elektroder klistret på brystet til å gi ett, eller flere strømstøt gjennom hjertet ved en hjertestans. Vanligvis er hjertestartere halv-automatiske, med det menes at de leser av rytmen til hjertet og ved hjelp av innebygde algoritmer gir beskjed når det skal gis støt. På engelsk brukes ofte betegnelsen AED(Automatic External Defibrillator)[4], det er denne typen som er vanligst og som i dagligtale blir omtalt som hjertestarter. Det finnes også manuelle hvor brukeren selv bestemmer når det skal gis støt, men disse brukes kun av profesjonelle. For at algoritmen i hjertestarteren skal kunne avgjøre om det skal gis støt eller ikke er den avhengig av at det ikke utføres kompresjoner samtidig. Moderne hjertestartere gir beskjed "Ikke berør pasienten, analyse pågår"

2.1.3 EKG

Et EKG registrerer elektriske spenningsforskjeller i hjertemuskulaturen når hjertet slår. Dette kan gi verdifull informasjon om hjertets funksjon, som

forskjellige uregelmessigheter(arytmier), hjerteblokk, endringer i hjertemuskulaturen forårsaket av betennelse og mer[1]. Hjertestarteren bruker EKG for å bestemme om det skal gis sjokk eller ikke. Dersom det utføres kompresjoner samtidig som hjerterytmen analyseres vil dette påvirke algoritmen som bestemmer om det skal gis sjokk, eller ikke.

2.2 Adaptivt filter for demping av artifakter

Det er ønskelig å kunne utføre artifaktfjerningen med så minimale fysiske endringer på hjertestarteren som mulig. Filteret baserer seg på frekvensen av kompresjonene som dagens hjertestartere allerede detekterer. Basert på denne frekvensen blir det laget en tidsvarierende modell av HLR-artifakten. Ved hjelp av et LMS filter kan man estimere de dynamiske parameterne til artifaktene fra kompresjonene. Dette filteret vil også være nyttig i bruk ved analyse av data fra behandlingen av en hjertestans.

2.2.1 Signalmodell av artifaktene

Koden for filtrering av kompresjoner er basert på artikkelen ”A Least Mean Square Filter for the Estimation of the Cardiopulmonary Resuscitation Artifact Based on the Frequency of the Compressions” [3]. Formler nevnt i dette kapitellet er hentet fra artikkelen om ikke annet er opplyst. Dette gjelder formlene: (2.1) (2.2) (2.3) (2.4) (2.5) (2.6) (2.7) (2.8) (2.9) (2.10). Kapittelet tar for seg teorien bak oppbyggingen av filteret.

Frekvens og fase til kompresjonene

Basert på dybden til kompresjonene kan frekvensen estimeres for kompresjonene. For hver kompresjon vil det være en negativ topp på signalet når kompresjonen er på det dypeste, for dette filteret er det antatt at frekvensen i hver kompresjonsperiode er lik, altså for de 30 kompresjonene mellom hver innblåsning er frekvensen den samme, men kan være ulik etter en stopp i kompresjoner. Ved et plutselig hjertestans kan de piezoelektriske trykksensorene i elektrodene til hjertestarteren brukes for å måle frekvensen. For en samplingsperiode lik T_s og en rekke med maksimale kompresjoner $(t_i = n_i T_s)$ kan instantanfrekvensen(f_i) og fasen$(\phi(n))$ skrives som:

$$f_i = \frac{1}{T_s(n_i + 1 - n_i)} = \frac{f_s}{\Delta n_i} \qquad n_i \leq n < n_i + 1 \tag{2.1}$$

$$\phi(n) = \frac{2\pi}{\Delta n_i}(n - n_i) + i2\pi \qquad n_i \leq n < n_i + 1 \tag{2.2}$$

Modellen for artifakter fra HLR

Filteret baserer seg på en Fourier rekke representasjon av artifaktene fra kompresjonene, men bruker tidsvarierende amplitude og fase, som endres for hver kompresjonssyklus. Under pågående kompresjoner blir artifaktene modellert som vist i formel 2.3.

$$\begin{aligned} \hat{S}_{comp}(n) &= \sum_{k=1}^{N} c_k(n) cos(k\phi(n) + \phi k(n)) \\ &= \sum_{k=1}^{N} a_k(n) cos(k\phi(n) + b_k(n) sin(k\phi(n)) \end{aligned} \tag{2.3}$$

Hvor den tidsvarierende amplituden og fasen $(c_k(n)$ og $\phi_k(n))$ er estimert gjennom et adaptivt filter som følger endringene i artifakten. Dette tilsvarer å følge variasjonen i amplituden og kvadraturkomponentene $(a_k(n)$ og $b_k(n))$.

Intervaller uten kompresjoner

De periodene hvor det ikke utføres kompresjoner, såkalt "hands-off' trengs det ingen adaptiv filtrering av signalet. For at filteret ikke skal være aktivt under hands-off tiden er det lagt inn en faktor $A(n)$, denne vil være 1 under kompresjoner og null ellers. Det gjør at formelen for artifaktene skrives som formel 2.4, hvor $\hat{S}_{cpr}$ er de estimerte artifaktene som skal sendes til filteret og $\hat{S}_{comp}$ er de estimerte kompresjonene fra artifaktene.

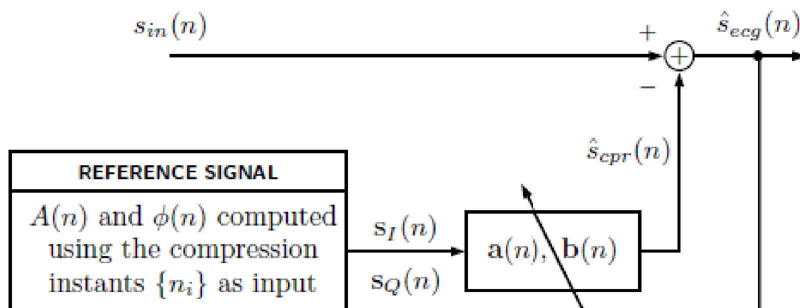
$$\hat{S}_{cpr}(n) = A(n) \cdot \hat{S}_{comp}(n) \tag{2.4}$$

For å unngå brå overganger er det definert to perioder hvor overgangen skal gjøres mykere. Den første er definert som overgangen fra ventilering til komprimering, denne er satt til å være en fjerdedel av den første syklusen, da det ikke er mekanisk aktivitet i denne perioden. Den neste perioden er når man går fra komprimering til ventilering, denne er satt til å være tre fjerdedeler av tiden på forrige syklus da dette er en bedre måte å representere tregheten i den påførte mekaniske aktiviteten. Den mekaniske aktiviteten er et resultat av kompresjonene som påføres brystet, det vil være mindre mekanisk aktivitet å ta hensyn til når vi går fra innblåsinger til kompresjon enn for kompresjon til innblåsning. For å få til disse mye overgangene er det laget en funksjon som kalles `Smoothstep` denne blir forklart mer i detalj i avsnitt 2.3.3.

Perioden mellom to etterfølgende kompresjoner er brukt for å bestemme kompresjonsintervallet. Det er laget en funksjon for å bestemme det maksimale intervallet mellom to påfølgende kompresjoner, denne tillater 50% avvik fra gjennomsnittskompresjonen i syklusen. Alt over dette er å anse som hands-off tid, denne funksjonen kalles ProcessPeaks og blir beskrevet mer i avsnitt 2.3.3. Artikkelen om LMS filteret [3] bruker et fast intervall på 1s for å definere hands-off tid.

2.2.2 Filteret for fjerning av artifakter

Strukturen til filteret som brukes til fjerning av artifakter er vist i figur 2.1.



Figur 2.1: Filter struktur. $S_{in}(n)$ er EKG signalet fra hjertestarteren, med artifakter, $\hat{S}_{cpr}$ er det estimert signalet av artifaktene fra HLR. $S_I(n)$ = fase komponenten, $S_Q(n)$ = kvadratur komponenten, $a(n)$ og $b(n)$ = Filterkoeffisientene. Figuren er hentet fra [3]

Ved hjelp av formel 2.4 estimeres artifaktene og trekkes fra inngangsignalet til hjertestarteren som gjør at utgangssignalet er fritt for artifakter. Det som trengs for å få til dette utenom EKG fra hjertestarteren er $A(n)$ og $\phi(n)$ som beregnes ut fra kompresjonene. Disse komponentene regnes ut ved hjelp av hver sin funksjon, i programvaren kalles $A(n)$ for `aRef` og $\phi(n)$ for `pRef`. Filterkoeffisientene oppdateres ved hjelp av LMS metoden, dette er en anerkjent metode for for å filtrere sinus komponenter med en kjent frekvens.

Filterkoeffisientene kan skrives som to vektorer ved tidspunktet n, $a(n)$ for fase, og $b(n)$ for kvadraturkomponenten. Hvor N er antallet harmoniske i $\hat{S}_{cpr}(n)$.

$$\begin{aligned} a(n) &= [a_1(n), \cdots, a_N(n)]^T \\ b(n) &= [b_1(n), \cdots, b_N(n)]^T \end{aligned} \tag{2.5}$$

For vårt tilfelle er komponentene for fase og kvadratur gitt av $A(n)$ og $\phi(n)$, referansesignalet skrives som:

$$\begin{aligned} S_I(n) &= A(n)[cos(\phi(n)), \cdots, cos(k\phi(n))] \\ S_Q(n) &= A(n)[sin(\phi(n)), \cdots, sin(k\phi(n))] \end{aligned} \tag{2.6}$$

Formel 2.3 for estimering av artifakter blir da som følger:

$$\hat{S}_{cpr}(n) = S_I(n)a(n) + S_Q(n)b(n) \tag{2.7}$$

Denne brukes videre til å finne det filtrerte EKG signalet:

$$\hat{S}_{ecg}(n) = S_{in}(n) - \hat{S}_{cpr}(n) \tag{2.8}$$

Filterkoeffisientene oppdateres kontinuerlig ved bruk av LMS metoden. Formlene for de oppdaterte koeffisientene blir som følger:

$$a(n+1) = a(n) + 2\hat{S}_{ecg}(n)MS_I^T(n) \tag{2.9}$$

$$b(n+1) = b(n) + 2\hat{S}_{ecg}(n)MS_Q^T(n) \tag{2.10}$$

Som vi ser av figur 2.1 er det ferdig filtrert EKG vi bruker som avviket når vi oppdaterer koeffisientene. Her er $M = diag(\mu_1, \cdots, \mu_N)$ hvor μ er steglengden til filteret. Steglengden kan ses på som grovheten til filteret. Filterkoeffisientene oppdateres jevnlig for å minimere de sinusformede forstyrrelsene.

2.3 MATLAB-implementering av filteret

Dette kapittelet inneholder en gjennomgang av MATLAB implementeringen av filteret. Filteret er delt opp i flere funksjoner som utfører beregningene som forklart i avsnitt 2.2.2. Komplett kode i MATLAB og Python oversettingen finner man i vedlegg A.

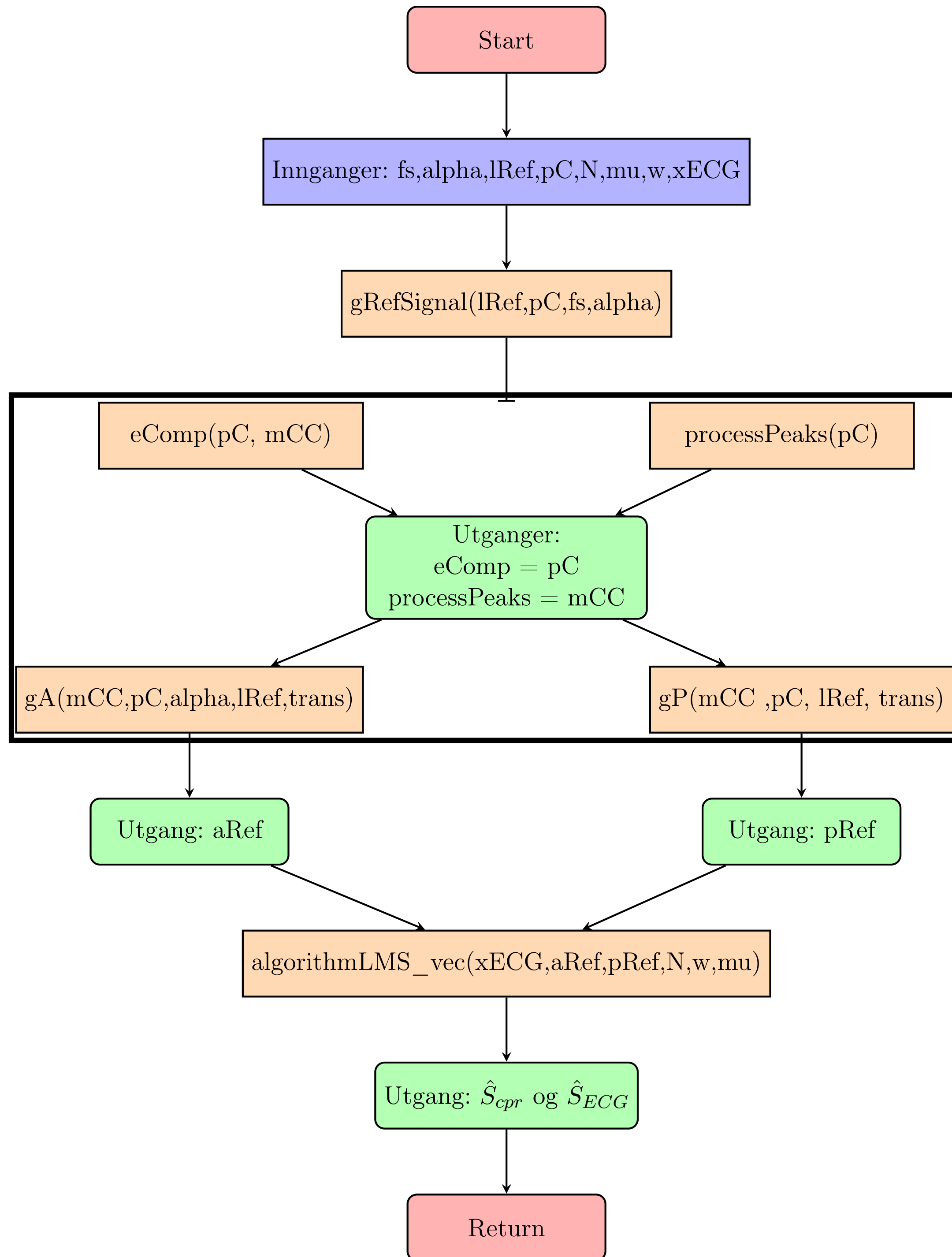
Under forklaringen av funksjonene er det brukt flere variabelnavn, de forskjellige variablene listet opp med en kort forklaring.

Ordliste:

- mCC = Maksimalt tillat tid mellom påfølgende kompresjoner
- pC = Tidspunktene for kompresjonene
- Alpha = Grunnverdi til aRef
- lA og lP = Lengden av EKG signalet også kalt lRef
- fT = Frekvensen til overgangen som en brøkdel av tidligere frekvens
- dT = Varigheten til overgangen
- fs = Samplingsraten
- N = Antallet harmoniske
- w = Veiefaktor for harmoniske komponenter
- mu = μ fra kap:2.2.2 grovheten til filteret
- hI = Filterkoeffisient formel:2.9
- hQ = Filterkoeffisient formel:2.10

2.3.1 Flytskjema med MATLAB funksjoner

Figur 2.2 viser en oversikt over funksjonskallene i MATLAB og hvilken rekkefølge de kjøres i programmet. Alt som er inne i boksen utføres ved å kalle funksjonen `gRefSignal`.



Figur 2.2: Flytskjema over MATLAB funksjonene

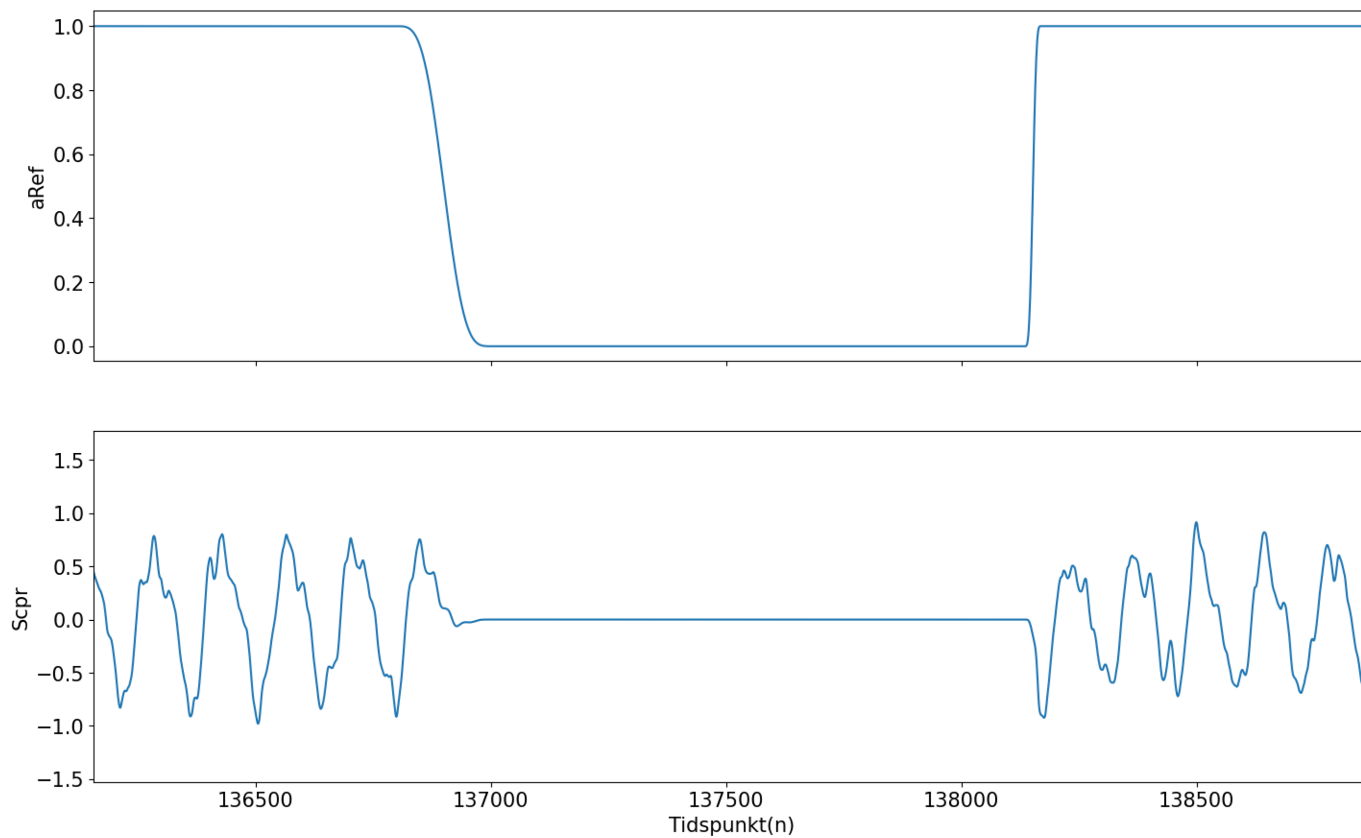
2.3.2 Hovedfunksjoner

Hovedfunksjonene er de som lager signalene og utfører selve filtreringen.

Amplituden til artifaktene (aRef)

For å definere intervallene hvor det utføres kompresjoner er det laget en egen funksjon. Funksjonen kalles for `gA` og tar inn seks variabler: `mCC, pC, alpha, lA, fT` og `dT`. Den returnerer en vektor `aRef` som gir 1 for når det er kompresjoner og 0 ellers, som forklart i avsnitt 2.2.1. Om vi ser på ligning 2.4 er `aRef` $= A(n)$, der er det også tydelig at dersom det ikke er pågående kompresjoner altså $A(n)$=0 vil artifaktene også være 0. Programmet sjekker om tiden mellom påfølgende kompresjoner er mindre enn det maksimalt tillate avviket, dersom den er det settes `aRef`$= 1$ for de tidspunktene hvor disse kompresjonene er. Her er også de myke overgangene konfigurert.

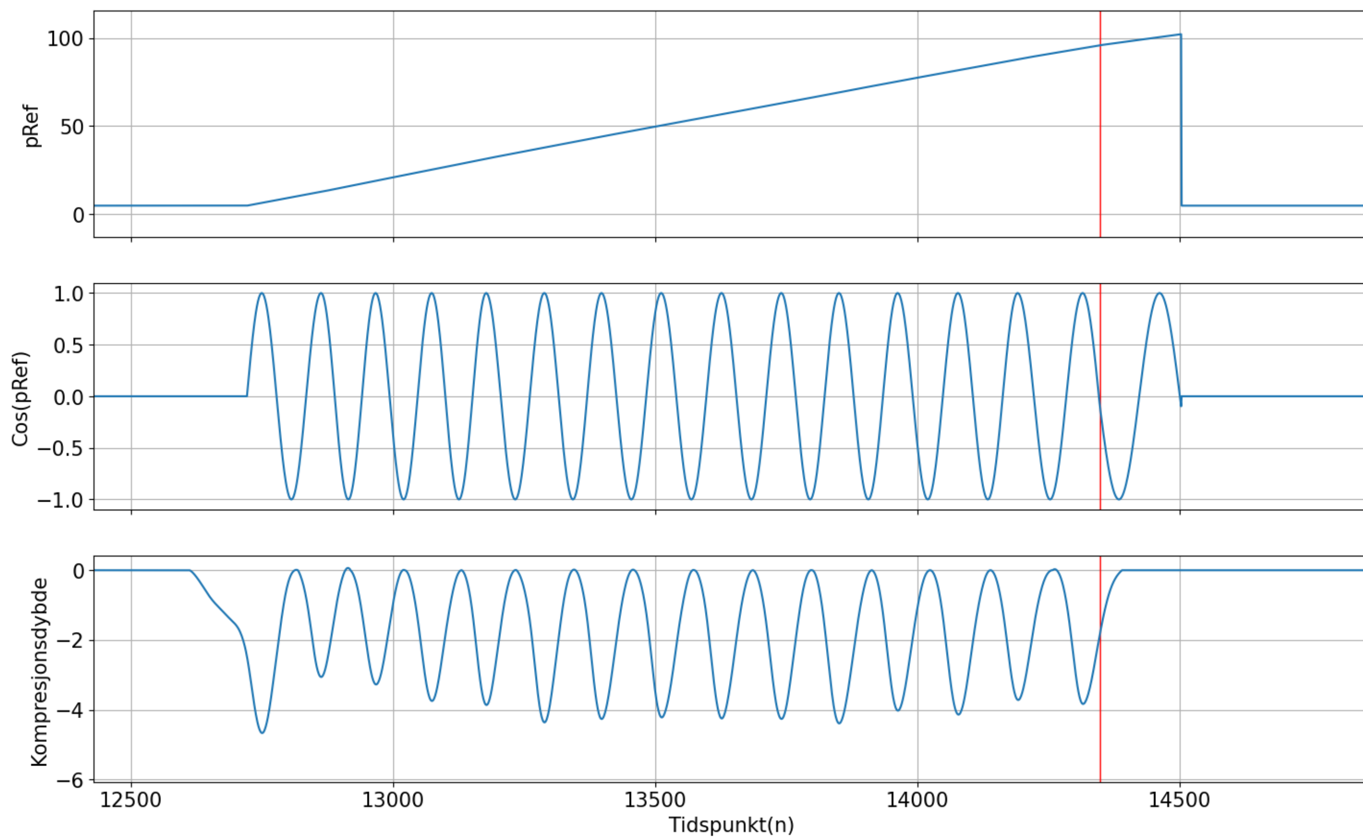
Figur 2.3 viser de myke overgangene som er lagt inn. Som forklart i avsnitt 2.2.1 er overgangen når vi går fra kompresjoner til opphold lengre enn for opphold til kompresjoner, det ser vi tydelig på figuren at kurven fra 1 til null er mykere enn for null til 1. Her vises også effekten av faktoren $A(n)$ på de utregnede artifaktene, når `aRef=1` har vi artifakter, ellers er de null.



Figur 2.3: Plot av aRef, sammen med $\hat{S}_{cpr}$

Fasen til artifaktene (pRef)

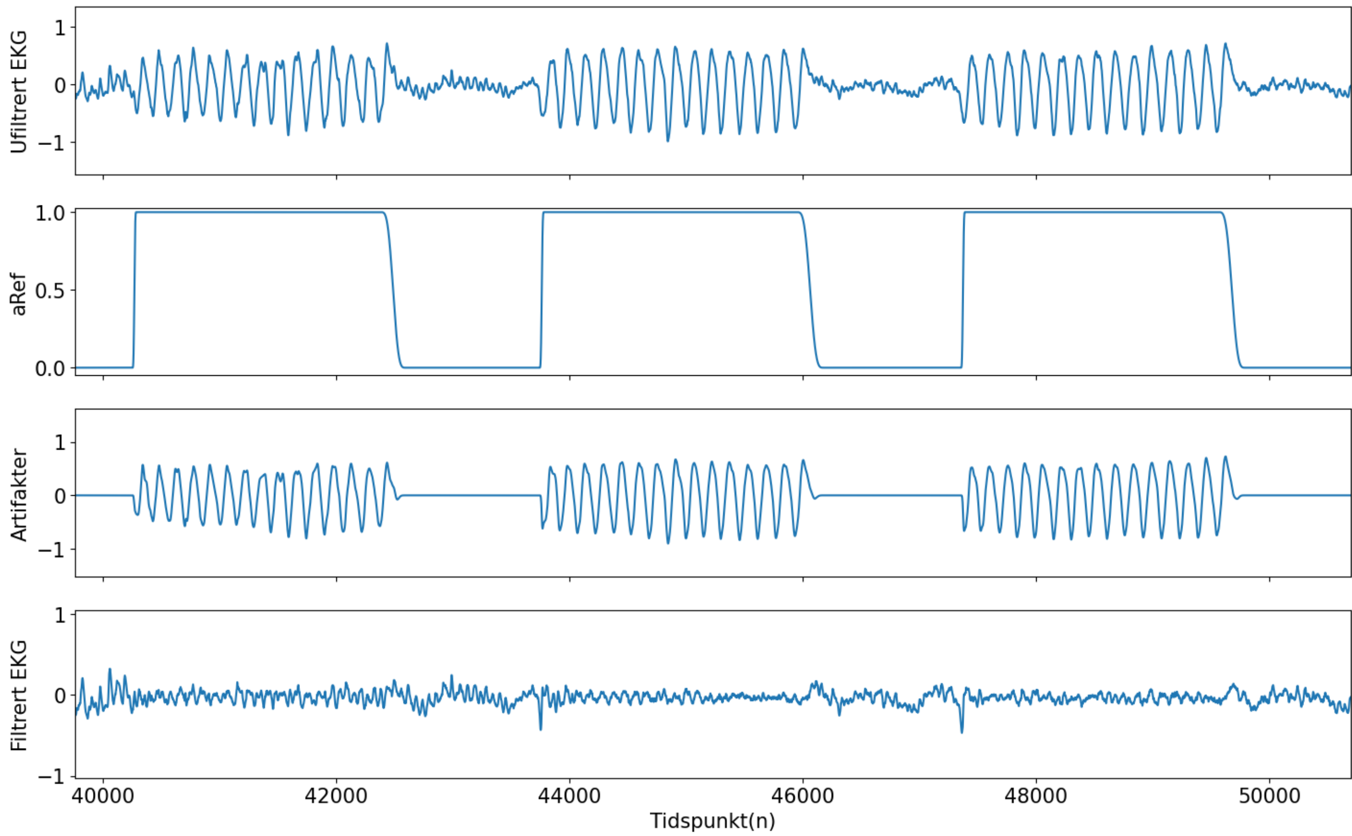
Fasen til artifaktene regnes ut med formel 2.2, denne realiseres i programmet ved hjelp av en funksjon som kalles `gP`. For å kunne regne ut fasen trengs: `mCC, pC, lp, fT` og `dT`. Det første funksjonen gjør er å definere en referanse fase, denne settes til å være $\frac{3\pi}{2}$. Deretter går funksjonen gjennom alle kompresjonstidspunktene, og legger inn fasen til signalet for påfølgende kompresjoner. Dette signalet stiger i multipler av 2π. Figur 2.4 illustrerer dette. Dersom man ser på `pRef` på figuren ser man at denne stiger jevnt fra referansefasen som ble definert tidligere og opp til rett under 90, der endres stigningsgraden noe. Punktet hvor stigningsgraden endrer seg er markert med en rød strek på grafene. Fasen endrer stigning når instantanfrekvensen endres. Dette er enklere å se dersom man ser på Cos(`pRef`) hvor man ser en forskyvning i den siste halve perioden på signalet. Ved å se på Cos(`pRef`) ser vi også på en god måte at signalet stiger jevnt i multipler av 2π da vi har en jevn sinuskurve. Steglengden for disse multiplene blir regnet ut for hver kompresjonssløyfe. Grunnen til at vi har denne endringen i fasen skyldes at i utregningen av `pRef` er det en annen behandling for den siste kompresjonen. Det er lagt inn en transient(`fT`) slik at overgangen ikke blir så brå, denne er lagt inn slik at `pRef` fortsetter etter at siste kompresjonen er utført. Effekten vises godt når man ser på kompresjonsdybden i figur 2.4 hvor pRef fortsetter selv etter siste kompresjon.



Figur 2.4: Plot av pRef og Cos(pRef) sammen med kompresjonsdybden

Filterfunksjonen

Selve filteret har også en egen funksjon `algorithm_lms_vec`. Det er her filtreringen samt oppdatering av koeffisientene skjer. Denne funksjonen tar inn det ufiltrerte EKG signalet, `aRef, pRef, N, w, mu` og kan ta inn en startverdi for `hI` og `hQ`. Dersom `hI` og `hQ` ikke er fylt ut på forhånd settes de til null. Det er denne funksjonen som realiserer formel 2.7 og 2.8. Funksjonen returnerer det ferdige filtrerte EKG signalet og de estimerte artifaktene fra HLR. Hvordan filteret påvirket det ufiltrerte EKG signalet vises godt på figur 2.5. Der har man det ufiltrerte EKG(S_{in}) signalet på toppen, `aRef`($A(n)$) er tatt med for å vise når filteret er aktivt. De estimerte artifaktene($\hat{S}_{cpr}$) og til slutt ferdig filtrert EKG($\hat{S}_{ecg}$). Der kan man tydelig se på det ufiltrerte EKG signalet hvordan det blir påvirket av kompresjonene og hvordan det filtrerte signalet ikke blir påvirket i like stor grad. På x-aksen har vi tidspunktet for samplingen fra hjertestarteren.



Figur 2.5: Plot av ufiltrert EKG, aRef, artifaktene og det filtrerte EKG signalet

2.3.3 Hjelpefunksjoner

For å kunne generere de dataene som trengs til hovedfunksjonene er det laget et par hjelpefunksjoner, disse brukes aktivt i hovedfunksjonene og er helt essensielle for at filteret skal fungere.

Generering av variabler

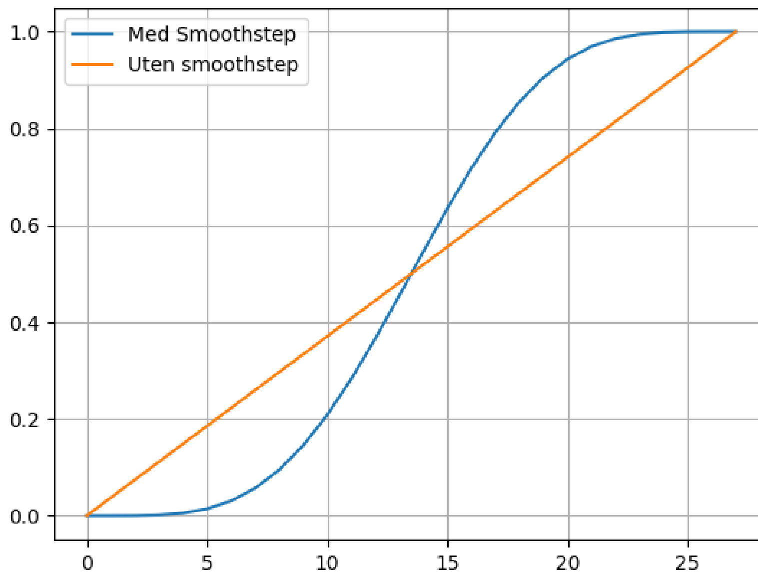
Det er laget en funksjon `gRefSignal` som ikke utfører beregninger og realiserer formler slik som de andre. Denne er laget da filtreringen av EKG signalet er en del av et større program, ved å lage funksjonen blir programmet ryddigere. Denne tar inn de fire inngangsvariablene: `lRef, pC, fs` og `alpha` og returnerer `aRef` og `pRef`.

Funksjon for å lage myke overganger

For å lage myke overganger som beskrevet i avsnitt 2.3.2 er det laget en funksjon kalt Smoothstep. Denne baserer seg på polynominterpolasjon og bruker ett 9.grad polynom(2.11) for å få ønsket nøyaktighet på punktene mellom null og 1. Det er vanlig på smoothstep funksjoner å ha tre inngangsvariabler, en inngang som er variabelen man ønsker smoothstep på, en høyre og en venstre flanke. Hvor den venstre flanken er antatt å være lavere enn høyre. Funksjonen tar inn dataene og returnerer null dersom de er mindre, eller lik venstre flanke og 1 dersom de er høyere eller lik høyre. Smoothstep funksjonen som er implementert i MATLAB tar inn en variabel `t`, men ingen høyre og venstre flanke. Måten dette er løst på er at funksjonen lager en variabel `s` som inneholder den minste verdien av `(t>0)·t` og 1, på den måten har vi ingen verdier lavere eller lik null, og heller ingen høyere enn 1.

$$70s^9 - 315s^8 + 540s^7 - 420s^6 + 126s^5 \tag{2.11}$$

Hva funksjonen gjør illustreres godt på figur 2.6 hvor smoothstep er brukt for å gå fra null til 1. Man ser også effekten av funksjonen på figur 2.3. Dersom man ser på `aRef`er funksjonen brukt ved overgangen fra 1 til null og opp igjen til 1. Det er i disse intervallene funksjonen er i bruk.



Figur 2.6: Graf for å illustrere de myke overgangene som genereres med smoothstep funksjonen

Bestemme maksimalt intervall mellom kompresjoner(mCC)

Funksjonen som er laget for å definere maksimalt tillat avstand mellom påfølgende kompresjoner heter `Processpeaks`, denne tar inn tidspunktene for kompresjonene(`pC`). Tidspunktene brukes for å regne ut intervallet mellom kompresjoner, dette gjøres ved å ta tiden for en kompresjon minus tiden for den forrige. Intervallene lagres som variabelen `it`, denne brukes videre for å definere "hands-off" intervaller, altså tid hvor man ikke driver med kompresjoner. Funksjonen sjekker så om tiden mellom påfølgende kompresjoner er større enn to ganger gjennomsnittstiden mellom to kompresjoner. Etter dette defineres variabelen `mCC` som $1.5 \times$ gjennomsnittsverdien av `it`. Gjennomsnittsverdien multipliseres med 1.5 da det tillates 50% avvik fra snittiden mellom påfølgende kompresjoner. Funksjonen returnerer variabelen `mCC` som da er maksimalt tillat tid mellom to kompresjoner, er tiden større er dette å regne som et avbrudd i kompresjoner.

Denne funksjonen kommer kun til sin rett ved analyse av historiske hjerteinfarkt. Da denne funksjonen er avhengig å vite om alle kompresjonene først for å bestemme gjennomsnittstiden mellom hver kompresjon. Det ligger derfor i hjelpefunksjonen `gRef_signal` en mulighet for å sette `mCC` manuelt til ønsket tidsintervall mellom kompresjonene. Som nevnt i avnsitt 2.2.1 beskriver [3] at de bruker 1 sekund for å definere maksimalt tilatt tid uten kompresjoner.

Fjerning av enkeltstående topper på kompresjonstidspunktene (eComp)

`eComp` er en funksjon som tar inn tidspunktene for kompresjoner(pC) og mCC. Det denne funksjonen gjør er å fjerne enkeltstående topper på variabelen pC. Det er annerledes behandling for første og siste topp av en ganske naturlig årsak. Det funksjonen gjør er å gå gjennom alle dataene i pC og sjekker om `pC(n)-pC(n-1) > mCC` og `pC(n+1)-pC(n) > mCC`, hvor n er et heltall fra en til lengden av `pC`. Dersom begge er større enn `mCC` fjernes denne fra `pC`, grunnen til at første og siste topp behandles ulikt er at for den første variabelen kan vi ikke sammenligne med variabelen før, og for den siste kan vi ikke sjekke for variabelen etter. Dette er løst ved at for den første sjekkes det kun mot den etter, og på den siste kun mot den før. Dette gjøres for å fjerne enkeltstående topper på signalet, disse blir ikke behandlet som om det var en kompresjon.

Kapittel 3

Metode

Under dette kapittelet blir metoden for å oversette programmet fra MATLAB til Python gjennomgått. Filosofien har hele tiden vært å forsøke å få Python koden så lik MATLAB varianten som mulig. Hovedårsaken til dette er at filtreringsfunksjonen er en liten del i et større program, det vil derfor være en ulempe om denne delen ikke følger strukturen som allerede er etablert.

3.1 Teoretisk bakgrunn for filteret

For å kunne gjennomføre Python-implementeringen viste det seg tidlig at en skikkelig gjennomgang av MATLAB koden og det å kunne knytte opp det som skjedde i koden mot beskrivelsen av filteret som er gitt i artikkel [3] var helt avgjørende for å kunne gjennomføre implementeringen. Det ble derfor brukt mye tid på å forstå hva koden faktisk gjorde, på den måten var det enklere å oversette koden da man skjønte hvorfor ting ble gjort.

3.2 Python implementering

Dette kapittelet dekker selve Python implementeringen.

3.2.1 Isolere filteret fra hovedprogrammet

Da funksjonene som utfører filtreringen er en del av ett større program som er tidkrevende å kjøre ble det tidlig besluttet at det mest hensiktsmessige var å få isolert den delen som tar for seg filtrering fra det store programmet. For å kunne gjøre dette på en måte som ikke påvirker det originale programmets funksjon ble det laget en ”if-setning” hvor man setter inn 1 for å kjøre originalkoden og 0 for å kjøre ny del. Det ble i den forbindelse også laget en funksjon `artifact_removal`. Hvordan koden ble isolert er vist i kode 3.1. If-setningen som velger originalkode eller ny kode ser man på linje nummer 140. Linje 155 viser bruken av denne, dersom man ikke skulle gått over til ett Python basert program hadde denne ikke hatt noen hensikt, da den kun kaller allerede genererte funksjoner, for mer detaljer se vedlegg A.9. Denne er laget for at man kun skal trenger å kalle en Python fil for å utføre filtreringen. For å endre om man skal kjøre i MATLAB eller i Python er det lagt inn en velger på linje 148, der kan man endre `env` fra ’matlab’ til ’python’ alt etter hvilket program man ønsker å kjøre. Hensikten med å kjøre Python fra MATLAB på denne måten har begrenset nytteverdi da det jobbes med å overføre hele programmet til Python, funksjonaliteten er lagt inn for å kunne verifisere samtidig som en oversetter programmet.

```
140 if 0 % Select replacement/original code with 0/1 switch
141         fs = 250; alpha = 0; lRef = length(xECG);
142         [aRef,pRef]=gRefSignal(lRef,PC,fs,alpha);
143         N = 5; w = ones(1,N); mu = 0.0178;
144         nanidx=find(isnan(xECG));
145         xECG(nanidx)=0;
146         [cpr_hat,
       xECG_hat]=algorithmLMS_vec(xECG,aRef,pRef,N,w,mu);
147     else
148         env = 'matlab';
149         fs = 250; alpha = 0;
150         N = 5; w = ones(1,N); mu = 0.0178;
151         nanidx=find(isnan(xECG));
152         xECG(nanidx)=0;
153         switch env
154             case 'matlab'
```

```
155                 [cpr_hat, xECG_hat] =
        artifact_removal(xECG,N,w,mu,PC,fs,alpha);
156             case 'python'
157                 % Code for interfacing python and
158                 % calling the python version, artifact_removal.py
159                 pyrunfile("artifacl_removal.py")
160         end
161     end
```

Kode 3.1: Utdrag av hovedprogrammet

3.2.2 Isolering av enkeltfunksjoner

Som beskrevet i avsnitt 2.3 er programmet delt opp i funksjoner. Under Python-implementeringen av de enkelte funksjonene var det veldig nyttig å kunne kjøre hver enkelt funksjon i MATLAB isolert fra hovedprogrammet. For å kunne gjøre dette måtte de nødvendige dataene fra hovedprogrammet lagres.

Deretter ble funksjonen satt opp i MATLAB helt isolert og kjørt for å sjekke at den var lik som i det store programmet. Ved å gjøre det på denne måten kunne en ta for seg en og en funksjon hvor hjelpefunksjonene fra avsnitt 2.3.3 var de som måtte tas først. Dette ble gjort ved å kopiere hver enkelt funksjon over til egne MATLAB filer. Hovedprogrammet ble så kjørt frem til linjen hvor denne funksjonen skal kjøres, der ble det satt ett stopp-punkt og inngangsvariablene ble lagret som en ".mat" fil. Ved å laste inn denne filen har man alle inngangsdataene som er nødvendige for at funksjonen skal fungere på egenhånd. Da ble det lettere og raskere å kjøre funksjonen for å sjekke utgangsvariablene. En kunne også lett legge inn linjer for å få funksjonen til å skrive ut mellomregninger der det var behov for dette.

De samme dataene som ble brukt for å kunne kjøre MATLAB funksjonen isolert sett ble lastet opp i Python for å kunne teste oversettingen samt at variablene ble importert fra MATLAB på en korrekt måte. På denne måten får man verifisert at Python funksjonen fungerer med de samme inngangene som gis fra MATLAB. Filene ble importert til Python ved å bruke en tilleggspakke som heter `SciPy.io`[7] som gjør det mulig å lese data lagret i en ".mat" fil.

3.2.3 Fremgangsmåte

Selve implementeringen i Python ble gjort linje for linje med ett mål om å beholde oppbyggingen og strukturen av koden så lik som mulig MATLAB versjonen. Under implementeringen viste det seg fort at det var hensiktsmessig å kunne teste mindre deler av hver funksjon. For å kunne gjøre dette ble deler av koden kommentert vekk, slik at det ble lettere å isolere hvor eventuelle ulikheter var. Kode 3.2 viser ett eksempel hvor endringene som gjøres dersom dette er definert som "hands-off" intervaller er kommentert vekk, altså det som skjer etter linje 15. Koden ble så kjørt med disse endringene og variablene ble lagret og eksportert til Python, da kunne en ganske ryddig gå gjennom funksjonen steg for steg og plukke opp eventuelle ulikheter så tidlig som mulig.

```
1  %% Initialize amplitude, initiate 1 to peaks
2      aRef(1:pC(1))=zeros(1,pC(1));
3          % Cuarter period rise time
4          tI=(pC(1)-ceil((pC(2)-(pC(1)))/4):pC(1));
5          aRef(tI)=(1-alpha)*smoothstep(0:1/(length(tI)-1):1)+alpha;
6      else
7          aRef(1:pC(1))=ones(1,pC(1));
8      end
9
10     for i=1:length(pC)-1
11         % Consecutive compressions
12         if( pC(i+1)-pC(i) < mCC )
13             aRef(pC(i)+1:pC(i+1))=1;
14         % Hands-off intervals
15         else
16             %% decay transition plus cuarter period
17
       %lT=floor(dT*fT*(pC(i)-pC(i-1)))+floor((pC(i)-(pC(i-1)))/4);
18
       %aRef(pC(i):pC(i)+lT-1)=fliplr((1-alpha)*smoothstep(0:1/(lT-1):1)+alpha);
```

Kode 3.2: Utdrag av aRef i MATLAB fra vedlegg A.5

Da flere av variablene brukt i programmet inneholder store mengder data var det ikke hensiktsmessig å manuelt verifisere punktene. For å illustrere dette så kunne `aRef` og `pRef` bestå av 468750 punkter, dette gjorde at all sammenligning måtte gjøres i programvaren. Hvordan dette ble gjort blir forklart mer i metode for sammenligning av resultater.

Falsk presisjon

Med falsk presisjon menes at dataene presenteres med en høyere nøyaktighet enn de faktisk har, dette er ett kjent problem innen programmering når en bruker desimaltall, det kommer av måten datamaskinen representerer desimaltall på. Ved tilfeller hvor koden ga ulikheter etter 13 desimaler måtte koden og teorien bak den aktuelle utregningen sjekkes. På den måten var det mulig å verifisere om dette var ett tilfelle av falsk presisjon, eller en feil i Python-implementeringen. Det kunne dreie seg om tilfeller hvor det skulle genereres en liste fra 0 til 1 med steglengde: $\frac{1}{34}$. Ved å gjøre en elementvis sammenligning vil listene i MATLAB og Python bli ulike, grunnet denne falske presisjonen. Slike tilfeller med ulikheter ble originalkoden sjekket opp mot den teoretiske bakgrunnen og datasettene hvor høy presisjon det var. Det ble etterhvert gått over til å sjekke for om utgangen til koden var lik opp til 13 desimaler. Dette er høyere presisjon enn det datasettene som brukes til å gjennomføre utregningene har, og gir en høy nok nøyaktighet.

Metode for sammenligning av resultater

De tilfellene hvor sammenligningen ikke var lik ble det nødvendig å kunne visualisere feilen og manuelt sjekke hva som var galt. Kode 3.3 viser hvordan dataene i kode: 3.2 ble verifisert underveis. Som forklart i avsnitt 3.2.3 ble deler av koden kommentert vekk og koden kjørt. Dersom man ser på linje 8 er det utført en sammenligning av `aRef` og `aRef_for_u_else` hvor `aRef` er fra Python og `aRef_for_u_else` er resultatet av å kjøre kode 3.2. Linje nummer 8 vil returnere "True" dersom disse to er like opp til 10 desimaler, eller "False" dersom den ikke er det. Det er også lagt inn på linje nummer 11 en direkte sammenligning, denne vil skrive ut posisjonene hvor de to variablene ikke er like. Dersom det ble oppdaget ulikheter kunne disse verifiseres ved å sette inn 1 i linje nummer 12, en vil da printe ut variablene i Python og MATLAB versjonen av `aRef`. For å bestemme intervallet man ønsker å printe ut variablene må disse velges manuelt i linje nummer 4 hvor man bestemmer midtpunktet, linje nummer 5 er hvor mange før og etter en ønsker og linje 6 genererer dette intervallet. Grunnen til at det ble gjort på denne måten er at en kan da velge hvor mye som skal skrives ut.

```
1
```

```
2 # Her ligger verifisering av resultatet etter for-lokken uten
      else:
3 if 1:
4     i0 = 74879
5     b = 100
6     inter = range((i0-b), (i0+b))
7     print('Er python utgangen samme som for matlab? Opp til 10
       desimaler')
8     print(np.allclose(aRef, aRef_for_u_else, rtol=1e-10,
       atol=1e-10))
9
10    print('Her er "ulikhetene"')
11    print(np.where(aRef != aRef_for_u_else))  # Gir ut
       posisjonene hvor de ikke er like, vil ha avrundingsfeil
12    if 1:
13        print('enkeltverdier python i', inter)
14        print(aRef[inter])
15        print('enkeltverdier matlab i', inter)
16        print(aRef_for_u_else[inter])
```

Kode 3.3: Utdrag av verifisering i Python

Kapittel 4

Resultater og konklusjon

Dette kapitellet inneholder presentering av resultatene, litt diskusjon rundt overgangen fra MATLAB til Python og en konklusjon.

4.1 Sammenligning MATLAB og Python

Som nevnt i avsnitt 3.2.3 har målet hele tiden vert å beholde strukturen så lik som mulig etter overgangen til Python. Dette har til stor grad latt seg gjennomføre. Dette gjenspeiler seg om man sammenligner det enkelte funksjonene i vedlegg A. Det er likevell enkelte forskjeller det kan være greit å utheve.

En av forskjellene som skiller seg ut er i koden for selve filteret i vedlegg A.9. Her har MATLAB en metode for å sjekke om inngangsvariablene er mindre enn seks, dersom den ikke er det settes de to siste til å være null. Måten dette løses på i Python er at ved definering av funksjonen settes en startverdi for de inngangsvariablene. Python metoden for å løse dette oppleves som ryddigere og gir en raskere oversikt. Kode 4.1 viser hvordan dette er løst i Python, hvor initialiseringen av filteret skjer allerede i funksjonskallet, MATLAB har flere linjer med kode for å utføre samme opperasjon se kode 4.2

```
1 def algorithm_lms_vec(xECG, aRef, pRef, N, w, mu,
      hI=np.zeros(N)[np.newaxis], hQ=np.zeros(N)[np.newaxis]):
```

Kode 4.1: Utdrag av algorithm_lms_vec i Python

```
1 function [y, e]=algorithmLMS_vec(s1,aRef,pRef,w,mu,hI,hQ)
2
3 %% Initialization
4 %  Filter coefficients
5 if nargin > 6
6     hI=hI(:);
7     hQ=hQ(:);
8 else
9     hI=zeros(1,N);
10    hQ=zeros(1,N);
11 end
```

Kode 4.2: Utdrag av algorithmLMS_vec i MATLAB

Den andre er i funksjonen `g_ref_signal`(A.7), her blir overgangsperiodene definert forskjellig i MATLAB og Python. MATLAB bruker en "struct" for å gruppere disse overgangene, i Python implementeringen er det valgt å definere disse direkte og heller legge til en ekstra inngangsvariabel i funksjonskallet til `aRef`(A.5) og `pRef`(A.6). For Python implementeringen er det valgt å kun definere `fT` og `dT`, da variablene `fTB` og `dTB` ikke er i i bruk for MATLAB. Hvordan dette gjøres er vist i utdrag av kodene, se kode 4.3 og 4.4 i linje 4 og 5 i kode 4.3 settes `fT` og `dT`. For MATLAB, kode 4.4 er dette også løst i linje 4 og 5, hvor det også vises hvordan de grupperes som `trans`.

```
1 def g_ref_signal(lRef, pC, fs, alpha):
2
3     # Define transition periods
4     fT = 1.25
5     dT = 1
```

Kode 4.3: Utdrag av g_ref_signal i Python

```
1 function [aRef,pRef]=gRefSignal(lRef,pC,fs,alpha);
2
3     %% Define transition periods
4     trans.fTA=1.25;
5     trans.dTA=1;
6     trans.fTB=1.25;
7     trans.dTB=0;
```

Kode 4.4: Utdrag av gRefSignal i MATLAB

En av de andre forskjellene som går igjen gjennom hele koden er at Python teller fra null, mens MATLAB teller fra 1. Dette er det flere eksempler på gjennom koden hvor lister starter fra 0 i Python og 1 i MATLAB. Det viser også når de myke overgangene defineres i `aRef`(A.5) for første og siste topp. For kodene 4.5 og 4.6 kan vi se allerede på linje 1 hvor Python sjekker mot `pC[0]` mens MATLAB sjekker `pC(1)` hvor de begge peker på første verdien i `pC`. Det vises også i linje nummer 3 i defineringen av `tI` at Python ligger ett nummer bak for hver verdi.

```
1 if pC[0] > mCC:
2         # Cuarter period rise time
3         tI = np.arange(pC[0]-np.ceil((pC[1]-pC[0])/4), pC[0]+1,
      dtype=int)
4         aRef[tI-1] = (1-alpha)*smoothstep(np.arange(0,
      1+1/(len(tI)-1), 1/(len(tI)-1)))+alpha
```

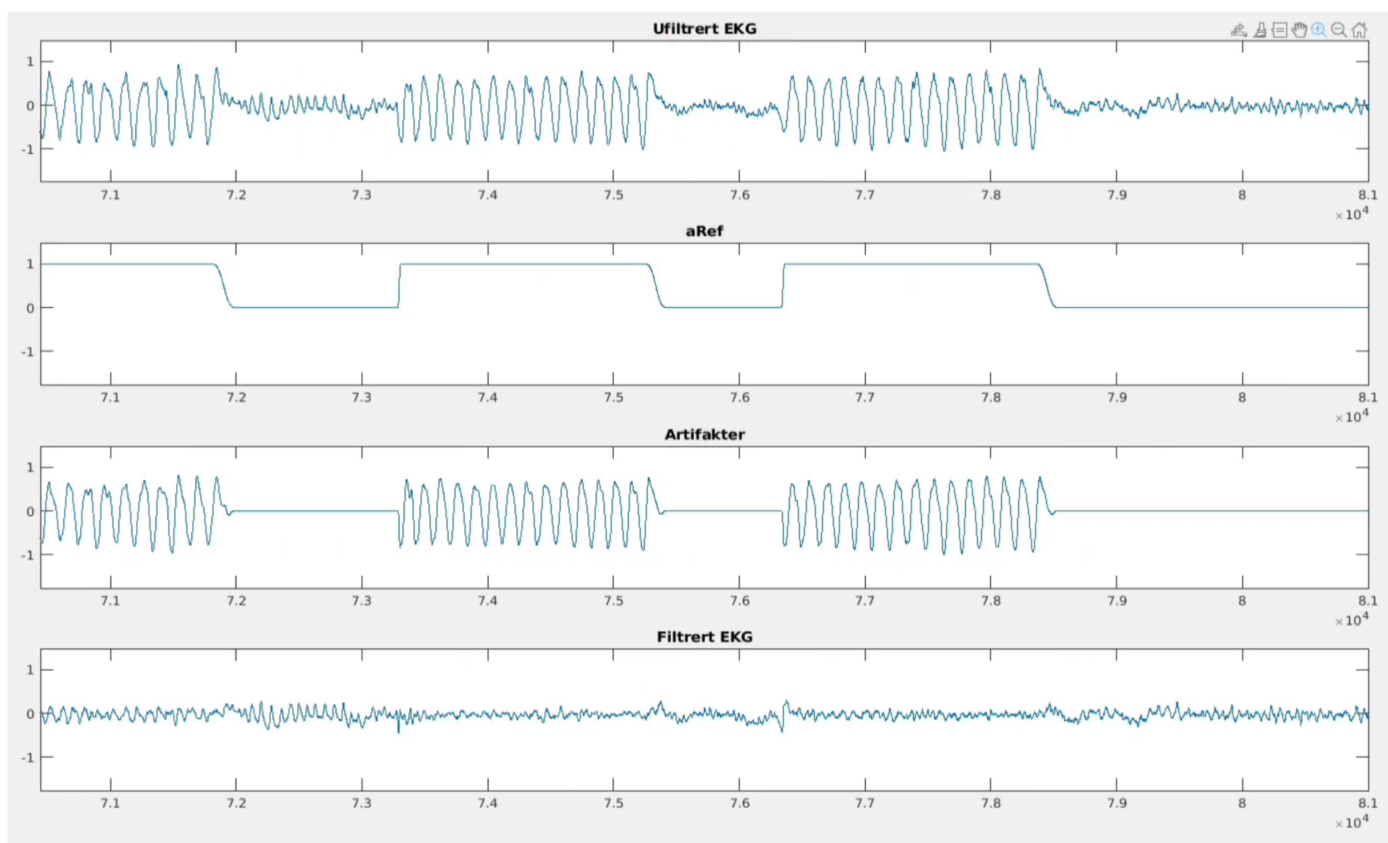
Kode 4.5: Utdrag av aRef i Python

```
1 if(pC(1)>mCC)
2         % Cuarter period rise time
3         tI=(pC(1)-ceil((pC(2)-(pC(1)))/4):pC(1));
4         aRef(tI)=(1-alpha)*smoothstep(0:1/(length(tI)-1):1)+alpha;
```

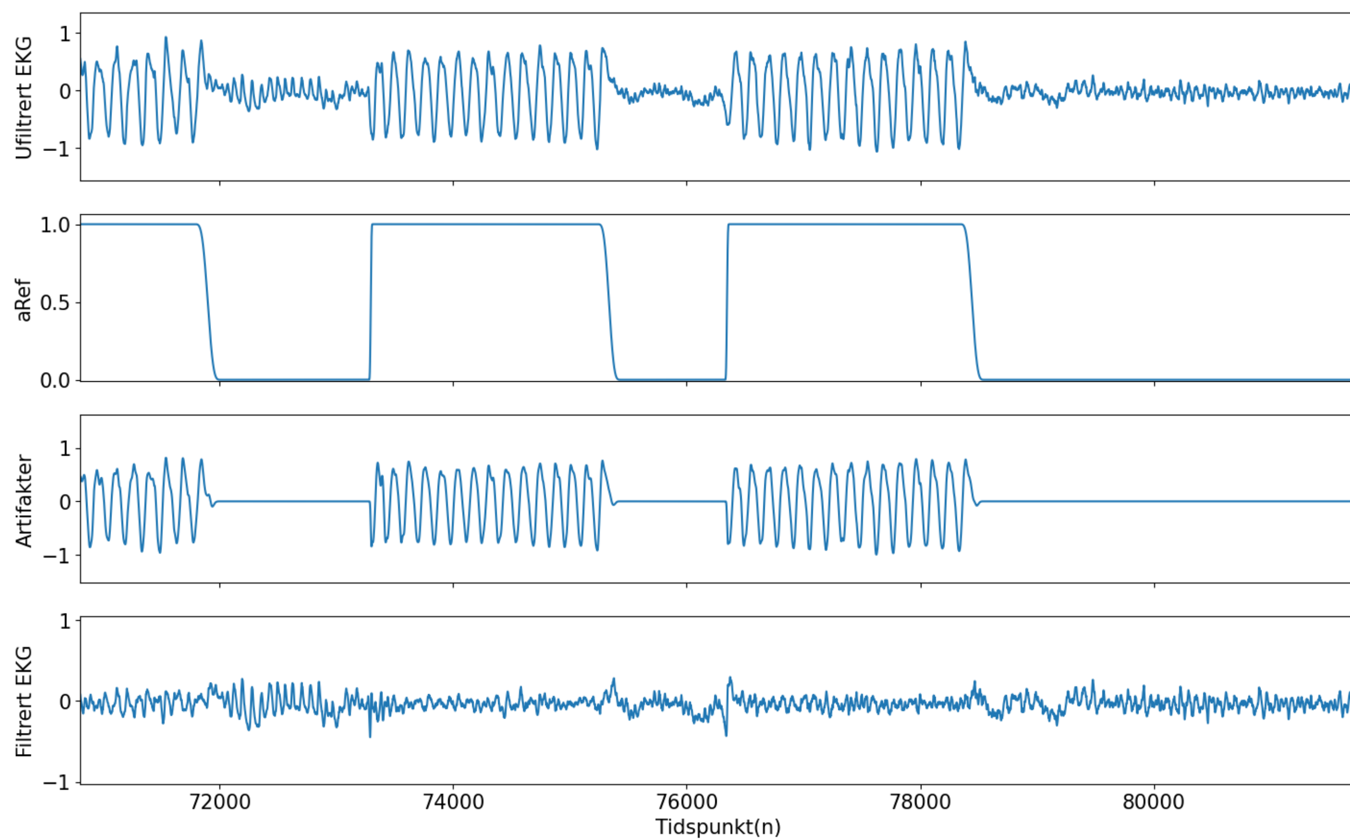
Kode 4.6: Utdrag av aRef i MATLAB

4.2 Resultater

Den mest effektive måten å presentere resultatene for denne oppgaven blir å plotte de forskjellige funksjonene mot hverandre. Under er det lagt ved plot fra MATLAB se figur 4.1 og plot fra Python, figur 4.2.



Figur 4.1: Plot av ufiltrert EKG, aRef, artifaktene og det filtrerte EKG signalet. Fra MATLAB



Figur 4.2: Plot av ufiltrert EKG, aRef, artifaktene og det filtrerte EKG signalet. Fra Python

Ved å sammenligne de to figurene over ser man at det er tilnærmet identiske resultater. Dette stemmer bra med resultatene fra testing av enkeltfunksjoner underveis. Alle variablene som plottes i figurene er sjekket ved å sammenligne utgangen fra MATLAB og Python, disse viser at de er helt like opp til 13 desimaler. Dette er en ubetydelig ulikhet som ikke har noen praktisk betydning da dataene ikke inneholder så nøyaktige inngangsvariabler. Ulikheten skyldes en håndtering i Python hvor en i kode 4.5 i linje 4 skal utføre `smoothstep` på en rekke som går fra 0 til `1+1/len(tI)-1` med steglengde lik: `1/len(tI)`. Denne har blitt manuelt sjekket og for det ene punktet er `len(tI)`=27, Python returnerer da `1/len(tI) = 0.037037037037037035` og MATLAB: `1/length(tI)=0.037037037037037`. For denne aktuelle utregningen har vi på inngangsverdien 27 kun to signifikante nummer. Presisjonen vår er derfor begrenset til to signifikante nummer altså 0.037.

Denne ulikheten ligger i utregningen av `aRef` som gjør at funksjonene som bruker denne i utregningen får en følgefeil, som også ligger etter 13 desimaler og blir sett på som ubetydelig. De funksjonene som ikke bruker `aRef` leverer identiske utgangsvariabler, dette gjelder `gP`, `smoothstep`, `eComp` og `process_peaks`.

4.3 Diskusjon

Store deler av arbeidet i starten gikk til å få en forståelse av hva koden faktisk utførte. Det ble forsøkt en litt naiv inngang på løsing av oppgaven med en tanke om at koden kunne oversettes uten å ta hensyn til hva den faktisk utførte. Når en fikk forståelsen for hva de forskjellige funksjonene faktisk gjorde viste det seg at oversettingen gikk mye raskere og ble bedre. Dette gjorde at teorien bak oppbyggingen av filteret som beskrevet i [3] ble gått gjennom ganske grundig. Det er også grunnen til at det er lagt vekt på bakgrunnen til filteret i avsnitt 2.2. Som det ble forklart i avsnitt 3.2.3 er det en svakhet i koden hvor det blir fryktelig mange desimaler som gir en falsk presisjon. Dette kunne med fordel vert løst ved at en hadde sett på signifikante nummer i inngangsvariabelen og definert at utgangsvariabelen skal ha samme antall.

Valget rundt det å beholde strukturen i koden falt ganske naturlig, på denne måten er det enkelt å sammenligne resultatene for hver funksjon. Det å kunne kjøre hver funksjon helt isolert sett var også en stor fordel, det gjorde at en stegvis kunne gå gjennom koden og sammenligne med teorien i avsnitt 2.2.2. Dette viste seg spesielt nyttig for funksjonen som utfører selve filteret se vedlegg A.8. Koden skiller seg litt ut fra teorien ved at det er lagt inn en veiefaktor for de harmoniske komponentene, denne er nok lagt til i etterkant for å kunne forsøke forskjellige veiefaktorer. Som standard blir denne satt til å være 1.

Det har blitt brukt mye tid på å finne de optimale løsningene i overgangen MATLAB - Python. Dette kommer av at det finnes flere forskjellige måter å løse samme problemet på. Kode 4.7 viser ett eksempel på to forskjellige løsninger som gir samme resultat. Hvor linje 1 er den løsningen som ble gått bort fra, forskjellen her er at i de innledende fasene ble brukt en funksjon fra Numpy biblioteket for å utføre matrisemultiplikasjon. Funksjonen `np.matmul(xI, hI.transpose())` kunne erstattes med `xI @ hI.transpose()`, dette gjorde koden ryddigere og mer oversiktlig. Den endelige koden hvor linje nummer 2 ble løsningen ligger i vedlegg A.8. Dette har vert en aktiv prosess gjennom hele implementeringen å finne de ryddigste løsningene for å utføre de operasjonene som har vert nødvendig.

```
1 y[i] = y[i] + np.matmul(xI, hI.transpose()) + np.matmul(xQ,
      hQ.transpose())
2 y[i] = y[i] + xI @ hI.transpose() + xQ @ hQ.transpose()
```

Kode 4.7: Utdrag av allgorithm_lms_vec i Python

4.4 Konklusjon

Hovedmålet med denne oppgaven var å få oversatt filterfunksjonen fra MATLAB kode til Python. Overgangen til Python har ikke krevd endringer i funksjoner for å kunne gjennomføre dette. Koden har blitt oversatt på en måte som beholder original struktur som forhåpentligvis gjør den lettere å lese for de som jobber med hovedprogrammet. Hele funksjonaliteten til koden fra MATLAB er overført til Python. Hver og en funksjon er kjørt en sammenligning på i Python for å verifisere at resultatene er de samme som for MATLAB. Python versjonen er testet og returnerer de samme utgangsverdiene som MATLAB opp til 13 desimaler, gitt at de får samme inngangsverdier.

Bibliografi

[1] Harald Arnesen. Ekg. Hentet 27. April 2022 fra: https://sml.snl.no/EKG.

[2] Helsedirektoratet. Hjerte- og lungeredning(hlr). Hentet 27.04.22 fra: https://www.helsenorge.no/sykdom/forstehjelp-og-skader/hjerte-og-lungeredning/.

[3] Unai Irusta, JesÚs Ruiz, SofÍa Ruiz de Gauna, Trygve Eftestøl, and Jo Kramer-Johansen. A least mean-square filter for the estimation of the cardiopulmonary resuscitation artifact based on the frequency of the compressions. *IEEE Transactions on Biomedical Engineering*, 56(4):1052–1062, 2009.

[4] Trond Nordseth. Hjertestarter. Hentet 27.04.22 fra: https://sml.snl.no/hjertestarter.

[5] Jo Kramer-Johansen og Kristin Alm-Kruse. Finnes det en tidsmaskin for pasienter med hjertestans utenfor sykehus. Hentet 20. April 2022 fra: https://oslo-universitetssykehus.no/avdelinger/prehospital-klinikk/finnes-det-en-tidsmaskin-for-pasienter-med-hjertestans-utenfor-sykehus.

[6] Oslo universitetssykehus. Hjertestans. Hentet 22. April 2022 fra: https://www.helsenorge.no/sykdom/hjerte-og-kar/hjertestans/.

[7] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat,

Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

Vedlegg A

Matlab og Python kode

A.1 Tilleggspakker

Dette vedlegget inneholder de originale MATLAB funksjonene, og Python oversettingen til hver av funksjonene. For Python koden er det en forutsetning at en importerer noen tilleggspakker. Hvordan er vist i kodeutdraget under.

```
1 import numpy as np
2 import scipy.io as scio
```

Kode A.1: Tillegsmoduler i Python

A.2 Smoothstep

A.2.1 Smoothstep i Matlab

```
1 function y = smoothstep(t)
2     s = min((t>0).*t,1);
3     y = (126+(-420+(540+(-315+70*s).*s).*s).*s).*s.^5;
4 return;
```

A.2.2 Smoothstep i Python

```
1 def smoothstep(t):
2     s = np.minimum(np.multiply((t > 0), t), 1)
3     y = (126+(-420+(540+(-315+70*s)*s)*s)*s)*np.power(s, 5)
4     return y
```

A.3 mCC ProcessPeaks

A.3.1 Process Peaks i Matlab

```
1 % Calculates the maximum interval between consecutive
2 % compressions
3 % allows 50% over the mean time between compressions.
4 % Consecutive peaks separated by more than 2*mean interval
5 % are considered hands-off intervals.
6
7 function mCC=processPeaks(pC)
8
9     iT  = pC(2:end)-pC(1:end-1); % calculate intervals between
       compressions
10    iT(find(iT > 2*mean(iT)))=[]; % these are hands off intervals
11
12    mCC=1.5*mean(iT);
13
14 return;
```

A.3.2 Process Peaks i Python

```
1 def process_peaks(pC):
2     # Calculates the maximum interval between consecutive
3     # compressions.
4     # allows 50% over the mean time between compressions.
5     # Consecutive peaks seperated by more than 2*mean interval
6     # are considered hands-off intervals
7     iT = pC[1:]-pC[0:-1]  # intervals between compressions
8     iT = iT[(np.logical_not(iT > 2*np.mean(iT)))]  # hands-of
       intervals
9     mCC = 1.5*np.mean(iT)
10    return mCC
```

A.4 eComp

A.4.1 eComp i Matlab

```
% Eliminate lonely peaks, 1st and last peaks
% have different treatment
% pC = array with position of the peaks
% mCC = maximum allowable separation between peaks
function pC=eComp(pC,mCC)

    eP=[];    % Possition of the peaks to suppress

    %%% First peak
    if(pC(2)-pC(1)> mCC)
        eP=1;
    end

    for i=2:length(pC)-1
        if( pC(i)-pC(i-1) > mCC && pC(i+1)-pC(i) > mCC)
            eP(end+1)=i;
        end
    end

    %%% Last peak
    if(pC(end)-pC(end-1) > mCC)
        eP(end+1)=length(pC);
    end

    pC(eP)=[];

return;
```

A.4.2 eComp i Python

```
1  # Eliminate lonely peaks, 1st and last peaks
2  # have different treatment
3  # pC = array with position of the peaks
4  # mCC = maximum allowable separation between peaks
5  def eComp(pC, mcc):
6      eP = []  # Possition of the peaks to suppress
7  # first peak
8      if pC[1]-pC[0] > mcc:
9          eP.append(0)
10     for i in range(1, len(pC)-1):
11         if pC[i]-pC[i-1] > mcc and pC[i+1]-pC[i] > mcc:
12             eP.append(i)
13 # last peak
14     if pC[-1]-pC[-2] > mcc:
15         eP.append(len(pC)-1)
16     pC = np.delete(pC, eP, 0)
17     return pC
```

A.5 aRef

A.5.1 aRef i Matlab

```
1 function aRef = gA(mCC,pC,alpha,lA,trans)
2     % Frequency of the transient as a fraction of previous
       frequency
3     fT=trans.fTA;
4     % durarion of the transient in periods
5     dT=trans.dTA;
6     aRef=alpha*ones(1,lA);
7
8     %% Initialize amplitude, initiate 1 to peaks
9     aRef(1:pC(1))=zeros(1,pC(1));
10        % Cuarter period rise time
11        tI=(pC(1)-ceil((pC(2)-(pC(1)))/4):pC(1));
12        aRef(tI)=(1-alpha)*smoothstep(0:1/(length(tI)-1):1)+alpha;
13    else
14        aRef(1:pC(1))=ones(1,pC(1));   # UtfÃ¸res med
       aRef[:int(pC[0])] = 1
15    end
16
17    for i=1:length(pC)-1
18        % Consecutive compressions
19        if( pC(i+1)-pC(i) < mCC )
20            aRef(pC(i)+1:pC(i+1))=1;
21        % Hands-off intervals
22        else
23            %% decay transition plus cuarter period
24
       lT=floor(dT*fT*(pC(i)-pC(i-1)))+floor((pC(i)-(pC(i-1)))/4);
25
       aRef(pC(i):pC(i)+lT-1)=fliplr((1-alpha)*smoothstep(0:1/(lT-1):1)+alpha);
26
27            %% elevation
28            if(i<length(pC)-1)
29                tI=(pC(i+1)-ceil((pC(i+2)-(pC(i+1)))/4):pC(i+1));
30
       aRef(tI)=(1-alpha)*smoothstep(0:1/(length(tI)-1):1)+alpha;
31            end
32        end
33    end
34    %% Last peak
35    if( pC(end)-pC(end-1) < mCC )
36        %% decay transition plus cuarter period
37
       lT=ceil(dT*fT*(pC(end)-pC(end-1)))+floor((pC(end)-(pC(end-1)))/4);
38
       aRef(pC(end):pC(end)+lT-1)=fliplr((1-alpha)*smoothstep(0:1/(lT-1):1)+alpha);
39        aRef=aRef(1:lA);
40    end
41
```

```
42 return;
```

A.5.2 aRef i Python

```
1 def gA(mCC, pC, alpha, lA, fT, dT):
2     # Frequency of the transient as a fraction of previous
      frequency = fT
3     # Duration of the transient in periods = dT
4
5     aRef = np.ones(int(lA))*int(alpha)
6     # Initialize amplitude, initiate 1 to peaks
7     aRef[:int(pC[0])] = 0
8     if pC[0] > mCC:
9         # Cuarter period rise time
10        tI = np.arange(pC[0]-np.ceil((pC[1]-pC[0])/4), pC[0]+1,
      dtype=int)
11        aRef[tI-1] = (1-alpha)*smoothstep(np.arange(0,
      1+1/(len(tI)-1), 1/(len(tI)-1)))+alpha
12    else:
13        aRef[:int(pC[0])+1] = 1
14    for i in range(len(pC)-1):
15            # Consecutive compressions
16        if pC[i+1]-pC[i] < mCC:
17            aRef[int(pC[i]):int(pC[i+1])] = 1
18            # Hands of intervals
19        else:
20            # Decay transition plus cuarter period
21            lT =
      np.floor(dT*fT*(pC[i]-pC[i-1]))+np.floor((pC[i]-pC[i-1])/4)
22            aRef[int(pC[i])-1:int(pC[i]+lT)-1] =
      np.flip((1-alpha)*smoothstep(np.arange(0, 1.001,
      1/(lT-1))))+alpha)
23            # elevation
24            if i < len(pC)-1:
25                x = i+1
26                tI = np.arange(pC[x]-np.ceil((pC[x+1]-pC[x])/4),
      pC[x]+1, dtype=int)
27                step = int(np.ceil(1/len(tI)))
28                num = int(np.ceil(len(tI)/step))  # number of
      steps
29                aRef[tI-1] = (1-alpha)*smoothstep(np.linspace(0,
      1, num))+alpha
30    # Last peak
31    if pC[-1] - pC[-2] < mCC:
32        lT =
      np.ceil(dT*fT*(pC[-1]-pC[-2]))+np.floor((pC[-1]-pC[-2])/4)
33        aRef[int(pC[-1]-1):int(pC[-1]+lT-1)] =
      np.flip((1-alpha)*smoothstep(np.arange(0, 1,
      1/(lT-1))))+alpha)
34        aRef = aRef[0:int(lA)]
35    return aRef
```

A.6 pRef

A.6.1 pRef i Matlab

```
1 function pRef = gP(mCC,pC,lP,trans)
2     zP=3*pi/2; %%% Referece phase value;
3     pRef=zP*ones(1,lP);
4     % Frequency of the transient as a fraction of previous
      frequency
5     fT=trans.fTA;
6     % durarion of the transient in periods
7     dT=trans.dTA;
8     %% Initialize phase, first peak at 0 phase
9     dP = 2*pi/(pC(2)-pC(1));
10    pRef(1:pC(1))=dP*(1-pC(1):0);
11
12    for i=1:length(pC)-1
13        % Consecutive compressions
14        if( pC(i+1)-pC(i) < mCC )
15            dP=2*pi/(pC(i+1)-pC(i));
16
      pRef(pC(i)+1:pC(i+1))=pRef(pC(i))+dP*(1:pC(i+1)-pC(i));
17        % Hands off intervals
18        else
19            %% Set a quarter period at f(i-1)
20            dP=2*pi/(pC(i)-pC(i-1)); % previous compression
21            tI=(pC(i)+1:pC(i)+floor((pC(i)-(pC(i-1)))/4)); % 1/4
      period
22            pRef(tI)=pRef(pC(i))+dP*(1:length(tI));
23
24            %% dT period at fT*f(i-1)
25            dP=2*pi/(fT*(pC(i)-pC(i-1)));
26            tIT=(tI(end)+1:tI(end)+floor(dT*fT*(pC(i)-pC(i-1))));
27            pRef(tIT)=pRef(tI(end))+dP*(1:length(tIT));
28
29            %% quarter period at fi
30            if(i<length(pC)-1)
31                dP=2*pi/(pC(i+2)-pC(i+1));
32                tI=(pC(i+1)-ceil((pC(i+2)-(pC(i+1)))/4):pC(i+1));
33                pRef(tI)=zP+dP*(1:length(tI));
34            end
35        end
36    end
37    %% Last peak
38    if( pC(end)-pC(end-1) < mCC )
39        %% cuarter period at f(i-1)
40        dP=2*pi/(pC(end)-pC(end-1));
41        tI=(1:floor((pC(end)-(pC(end-1)))/4));
42        pRefL(tI)=pRef(pC(end))+dP*(tI);
43        %% Transient
44        dP=2*pi/(fT*(pC(end)-pC(end-1)));
45        tIT=(tI(end)+1:tI(end)+ceil(dT*fT*(pC(end)-pC(end-1))));
```

```
46         pRefL(tIT)=pRefL(end)+dP*(1:length(tIT));
47         pRef(pC(end):pC(end)+length(pRefL)-1)=pRefL;
48     end
49
50     pRef=pRef(1:lP);
51 return;
```

A.6.2 pRef i Python

```
1  def gP(mCC, pC, lP, fT, dT):
2      zP = 3*np.pi/2  # Referece phase value
3      pRef = np.ones(lP)*zP
4      # initialize phase, first peak at 0 phase
5      dP = 2*np.pi/(pC[1]-pC[0])
6      pRef[:int(pC[0])] = dP[0]*np.linspace(1-int(pC[0]), 0,
        int(pC[0]))
7      for i in range(len(pC)-1):
8          # Consecutive compressions
9          if pC[i+1]-pC[i] < mCC:
10             dP = 2*np.pi/(pC[i+1]-pC[i])
11             pRef[int(pC[i]):int(pC[i+1])] =
        pRef[int(pC[i]-1)]+(dP*(np.linspace(1,
        int(pC[i+1])-int(pC[i]), int(pC[i+1])-int(pC[i]))))
12         # Hands of intervals
13         else:
14             # Set a quarter period at f(i-1)
15             dP = 2*np.pi/(pC[i]-pC[i-1])
16             start = int(pC[i]-1)
17             stop =
        int(pC[i])+int(np.floor((int(pC[i])-int(pC[i-1]))/4))
18             tI = np.arange(start, stop, 1)
19             pRef[tI] = pRef[int(pC[i]-1)]+dP*np.arange(0, len(tI))
20
21             # dT period at fT*f(i-1)
22             dP = 2*np.pi/(fT*(pC[i]-pC[i-1]))
23             tIT = np.arange(int(tI[-1]+2),
        int(tI[-1]+2)+int(np.floor(dT*fT*(int(pC[i]-pC[i-1])))))
24             pRef[tIT-1] = pRef[tI[-1]]+dP*np.arange(1, len(tIT)+1)
25
26             # Quarter period at fi
27             if i < len(pC)-1:
28                 dP = 2*np.pi/(pC[i+2]-pC[i+1])
29                 tI =
        np.arange(int(pC[i+1]-np.ceil((pC[i+2]-pC[i+1])/4)),
        int(pC[i+1]+1))
30                 pRef[tI-1] = zP+dP*np.arange(1, len(tI)+1)
31
32     # Last peak
33     if pC[-1]-pC[-2] < mCC:
34         # Quarter period at f(i-1)
35         dP = 2*np.pi/(pC[-1]-pC[-2])
```

A.6 pRef

```
36        tI = np.arange(1, int(np.floor((pC[-1]-pC[-2])/4))+1)
37        pRefL = np.empty(len(tI))
38        pRefL[tI-1] = pRef[pC[-1]-1]+dP*tI
39
40        # Transient
41        dP = 2*np.pi/(fT*(pC[-1]-pC[-2]))
42        tIT = np.arange(tI[-1]+1,
     tI[-1]+int(np.ceil(dT*fT*(pC[-1]-pC[-2]))+1))
43        value = pRefL[-1]+dP*np.arange(1, len(tIT)+1)
44        pRefL = np.append(pRefL, value)
45        pRef[np.arange(pC[-1], pC[-1]+len(pRefL))-1] = pRefL
46        pRef = pRef[0:lP]
47    return pRef
```

A.7 gRef Signal

A.7.1 gRef Signal i Matlab

```
1 function [aRef,pRef]=gRefSignal(lRef,pC,fs,alpha);
2
3     %% Define transition periods
4     trans.fTA=1.25;
5     trans.dTA=1;
6     trans.fTB=1.25;
7     trans.dTB=0;
8
9     % Maximum interval between consecutive compressions = 0.8 sec
10    % mCC = 0.8*fs;
11    mCC=processPeaks(pC);
12
13    % Eliminate lonely peaks, maximum separation mCC
14    pC   = eComp(pC,mCC);
15    aRef = gA(mCC,pC,alpha,lRef,trans);
16    pRef = gP(mCC,pC,lRef,trans);
17 return;
```

A.7.2 gRef Signal i Python

```
1 def g_ref_signal(lRef, pC, fs, alpha):
2     # Define transition periods
3     fT = 1.25
4     dT = 1
5     # Maximum interval between consecutive compressions = 0.8sec
6     man_mCC = 0.8*fs
7     mCC = process_peaks(pC)
8
9     # Eliminate lonely peaks, maximum separation mCC
10    pC = eComp(pC, mCC)
11    aRef = gA(mCC, pC, alpha, lRef, fT, dT)
12    pRef = gP(mCC, pC, lRef, fT, dT)
13
14    return [aRef, pRef]
```

A.8 Algorithm LMS Vectoriced

A.8.1 Algorithm LMS Vectoriced i Matlab

```
1  % Vectorized version of the LMS algorithm, improve performance
2  % s1:  ECG corrupted by CPR
3  % aRef and pRef, reference signals derived from compression
       depth: A(n) eta Phi(n)
4  % N :  Number of harmonics
5  % mu:  mu
6  function [y, e]=algorithmLMS_vec(s1,aRef,pRef,N,w,mu,hI,hQ)
7
8  %% Initialization
9  %  Filter coefficients
10 if nargin > 6
11     hI=hI(:);
12     hQ=hQ(:);
13 else
14     hI=zeros(1,N);
15     hQ=zeros(1,N);
16 end
17
18 %  Estimated CPR and ECG
19 y=zeros(1,length(s1));
20 e=zeros(1,length(s1));
21
22 K = 1:N;
23
24 for i=1:length(s1)
25
26     % estimated CPR
27     xI = aRef(i)*cos(K*pRef(i));
28     xQ = aRef(i)*sin(K*pRef(i));
29     y(i) = y(i) + xI*hI' + xQ*hQ';
30
31     e(i)=s1(i)-y(i);
32
33     % update the filter coefficients
34     hI = hI + 2*mu*e(i)*w.*xI;
35     hQ = hQ + 2*mu*e(i)*w.*xQ;
36 end
```

A.8.2 Algorithm LMS Vectoriced i Python

```
1  # Vectorized version of the LMS algorithm, improve performance
2  # xECG = ECG corrupted by CPR
3  # aRef and pRef, reference signals derived from compression
       depth: A(n) eta Phi(n)
4  # N = Number of harmonics
5  # mu = mu
6
7  def algorithm_lms_vec(xECG, aRef, pRef, N, w, mu, hI=0, hQ=0):
8
9      if hI == 0 and hQ == 0:
10         hI = np.zeros(N)[np.newaxis]
11         hQ = np.zeros(N)[np.newaxis]
12
13     # Estimate CPR and ECG
14     y = np.zeros(len(xECG))  # cpr_hat
15     e = np.zeros(len(xECG))  # xECG_hat
16     k = np.arange(1, N+1)
17
18     for i in range(0, len(xECG)):
19         xI = aRef[i]*np.cos(k*pRef[i])
20         xQ = aRef[i]*np.sin(k*pRef[i])
21
22         y[i] = y[i] + xI @ hI.transpose() + xQ @ hQ.transpose()
23         e[i] = xECG[i]-y[i]
24
25     # Update the filter coefficients
26         hI = hI + np.multiply((2*mu*e[i]*w), xI)
27         hQ = hQ + np.multiply((2*mu*e[i]*w), xQ)
28     return [y, e]
```

A.9 Artifact Removal

A.9.1 Artifact Removal i Matlab

```
1 function [cpr_hat, xECG_hat] =
      artifact_removal(xECG,N,w,mu,PC,fs,alpha)
2  lRef = length(xECG);
3  [aRef,pRef]=gRefSignal(lRef,PC,fs,alpha);
4  [cpr_hat, xECG_hat]=algorithmLMS_vec(xECG,aRef,pRef,N,w,mu);
5 end
```

A.9.2 Artifact removal i Python

```
1 def artifact_removal(xECG, N, w, mu, PC, fs, alpha):
2     l_ref = len(xECG)
3     # Define transition periods
4     fT = 1.25
5     dT = 1
6     #
7     [aRef, pRef] = g_ref_signal(l_ref, PC, fs, alpha)
8     [cpr_hat, xECG_hat] = algorithm_lms_vec(xECG, aRef, pRef, N,
       w, mu)
9     return [cpr_hat, xECG_hat]
```