Universitetet
i Stavanger

DET TEKNISK-NATURVITENSKAPELIGE FAKULTET

BACHELOROPPGAVE

Studieprogram/spesialisering: Bachelor i Datateknologi	Vårsemesteret 2021 Åpen / ~~Konfidensiell~~
Forfatter(e): Ivar Walskaar, Sebastian Sakow, Emil Pierzgalski	
Fagansvarlig: Trygve Christian Eftestøl Veileder(e): Trygve Christian Eftestøl	
Tittel på bacheloroppgaven: Analyse av data fra gjenoppliving av hjertestanspasienter -implementasjon i Python.	
Studiepoeng: 20	
Emneord: Dataanalyse Hjerteinnfarkt Biosensor Python BCG	Sidetall: 68 + vedlegg/annet: 1 fil (Resprog.exe og kildekode) Stavanger 15. mai 2021

Sammendrag

I årrekker har det blitt utført forskning på hjertestanspasienter der livreddere under vanskelige forhold prøver gjenopplivning ved hjelp av hjerte- og lungeredning (HLR), defibrillator og intravenøse legemidler. Gjenopplivningen påvirker pasientens hjerterytmer på forskjellige måter, og kan utløse positive eller negative reaksjoner [18]. Konsensus er at optimal HLR, altså 100-120 brystkompresjoner i minuttet på minst 5cm, øker oddsene for gjenoppliving av en hjertestanspasient. Dermed har det blitt utviklet hjertestartere med tilbakemelding hvor man på bakgrunn av signaler fra elektrodeputer og akselerometere [7] kan veilede brukeren.
Formålet med denne bacheloroppgaven var å utvikle et visningsprogram for å lese inn, bearbeidede, og visualisere signaler fra ulike hjertestansepisoder. Signalene blir representert i grafer med uthevede områder hvor det på forhånd har blitt utført automatisk deteksjon av hjerteslag, ventileringer osv. Programmet var opprinnelig skrevet i Matlab, men da Matlab krever lisens og har mindre utbredelse, så ble det besluttet å gå over til Python (versjon 3.8), som er et objektorientert programmeringsspråk med usedvanlig lettlest syntaks. Resultatet er et mye raskere program med fokus på robusthet og brukervennlighet. Vi håper dette forbedrer brukeropplevelsen og forenkler forskningsarbeid.

Forord

Denne oppgaven er vårt avsluttende arbeid ved Universitetet i Stavanger, Bachelor i Datateknologi, men også starten på ny og tung, men allikevel forlokkende ferd i retning av Mastergrad, blant kjente og ukjente fjes. Vi vil gjerne takke vår veileder, professor Trygve Eftestøl. Hans gode råd og kunnskap holdt håpet oppe slik at arbeidet omsider ble dratt i land. To tredjedeler av oss vil også gjerne rette en takk i retning av Ivar Walskaar, som har vist god lederskap i løpet av dette semesteret og sparket igang store deler av arbeidet. Til slutt vil vi også takke venner og familie, som viste stor forståelse for vårt økende fravær i det fristen for innlevering nærmet seg.

Stavanger, Mai 2021.
Emil Pierzgalski, Ivar Walskaar og Sebastian Sakow.

Innhold

Kapittel 1

Innledning

Denne oppgaven blir utført på bakgrunn av en tidligere studie, hvis hovedformål var å undersøke om diverse BCG (eng: ballistocardiographic) biosensorer kunne brukes til å spå kliniske forverringer (i verste fall hjerteinfarkt) hos pasienter [8][10]. Målingene fra biosensorene ble opprinnelig fremstilt via programmeringsspråket MATLAB [20]. Ettersom MATLAB krever lisens og ikke er særlig egnet til å lage visningsprogram, ble det bestemt å lage programmet med programmeringsspråket Python.
Hovedutfordingen for oss har vært å analysere koden til det opprinnelige programmet og deretter lage vårt eget grensesnitt i Python, med noe utvidet funksjonalitet.

Kapittel 2

Bakgrunn

Hjertestans og håndtering av dette er et kompleks tema og mange faktorer spiller inn. Danning av et helhetsbilde er tidkrevende ettersom rapporter av en hjertestansepisode krever tolkning av fagfolk. Gjennom tverrfaglig samarbeid og teknologisk utvikling kan dette arbeidet forenkles med bedre og billigere fastvare med mer kompleks programvare. Rapporteringer av hjerteepisoder bevares i databaser, samtidig som det pågår utvikling av felles standarder om innsamling, bearbeiding og presentasjon av disse rådataene [13].
Vi håper at visningsprogrammet vårt, på bakgrunn av innsamlede data, vil kunne brukes som et rammeverk for videre arbeid mot automatisk deteksjon av hjertekompresjoner, ventileringer osv. Vi håper også at en mer oversiktlig presentasjon av signaler vil hjelpe videre forskning om den beste fremgangsmåten ved gjenopplivning. Resultatet av disse arbeider kan forhåpentligvis være til hjelp for brukere av hjertestartere via automatisk tilbakemelding, da stressnivået under slike episoder ofte kan føre til HLR under pari.

2.1 Medisinsk bakgrunn

Følgende delkapitler vil gi en kort forklaring på de ulike signalene representert i grensesnittets grafer.

2.1.1 EKG

EKG er en forkortelse for elektrokardiogram og blir brukt for å observere hjertets tilstand. Dette gjøres ved å måle de elektriske spenningsforskjellene som oppstår når hjertet slår, og kan gi en rekke verdifulle opplysninger om blant annet; arytmier, hjerteblokk, myokarditt, hjertekrampe og mer [3]. En begrensning med EKG er at under en hjertestans kan det i blant oppstå Pulsløs Elektrisk Aktivitet (PEA). Noe som betyr at normal puls blir registrert selv om hjertet i realiteten er for svakt til å gi pulsgivende slag [9, p. 58] [8, p. 11-12].

2.1.2 $EtCO_2$

End Tidal CO_2 er ett mål på konsentrasjonen av CO_2 i slutten av ett utpust og blir målt av en gassanalysator kalt kapnograf. Om en persons lungefunksjon fungerer som normalt, vil denne målingen kunne gjenspeile mengde CO_2 i blodstrømmen gjennom hjertet. Dette blir ofte brukt i forbindelse med generell anestesi [12]. En reduksjon av $EtCO_2$ kan være en indikator på ett senket blodtrykk. Utfordringen med kapnografi er at det er sterkt avhengig av normal lungefunksjon, noe som ikke alltid er stabilt. Man kan da få ett feil inntrykk av $EtCO_2$ målingene [8, p. 14].

2.1.3 PPG

Fotopletysmografi (PPG) innebærer å plassere en lyskilde over en blodåre og måle refleksjonen med en sensitiv lysmåler kalt fotodiode. Ved å se hvor mye lys som reflekteres kan man finne blodtrykket. Det pulserende signalet vil også avsløre pulsfrekvensen. PPG signalene vi behandler i dette programmet er trolig fra ett pulsoksymeter på pasientens finger. Ett annet populært bruksområde for fotopletysmografi er i pulsklokker. [16]

2.1.4 TTI

Pasienten vil ha på seg to defibrilleringsputer på brystet og transtorakal impedans (TTI) vil bli målt ved å sende strøm mellom disse putene. Blod er en god strømleder og når hjertet pumper og hovedpulsåren fylles opp, vil denne vekslingen i motstand reflekteres i en graf av målingene. Denne humpen på grafen vil kunne representere slagvolumet, altså mengden blod hjertet pumper ut per sammentrekning. TTI kan bli brukt ved gjenoppliving for å overvåke lungefunksjon og brystkompresjon i tilfredsstillende grad [7] uten tilleggsutstyr. [8, p. 16]

2.1.5 BCG

I det hjertet slår pumpes blod nedover hovedpulsåren. Denne kraften medfører at kroppen rykker oppover[4]. I det forskjellige deler av aorta trekkes sammen eller utvides, så vil kroppen fortsette med å bevege seg opp- eller nedover. Ballistokardiografi (BCG) er en grafisk visning av disse mekaniske bevegelsene. Disse impulsene kan oppdrives via kroppens overflate, hvor det i nyere forskning ekseprimenteres med å fastslå både BCG og PPG ved hjelp av kamera, hvis oppgave det er å observere en pasients endringer i ansiktsuttrykk (BCG) og lysrefleksjoner i samme området (PPG) [11].

2.2 Teknisk bakgrunn

2.2.1 Terminologi

Høynivåspråk: Betegnelse på programmeringsspråk med syntaks som først leses av en tolk for så å oversettes (kompileres) til bytekode, altså instruksjoner for prosessoren.
Objektorientert: Objektorienterte språk baseres på klasser og deres objekter. En klasse inneholder kode om hvordan et objekt lages, dens egenskaper og oppførsel. Dermed kan objektene inneholde data/tilstand og ha metoder for å endre på disse.
Polimorfisme: Polimorfisme er en viktig byggestein i objektorientert pro-

grammering som går ut på at et barn/underklasse arver fra en forelder/overklasse. Oppbyggingen av forelderen er som regel ganske generell, hvor barnet arver disse egenskapene og i tillegg utvides med særegen funksjonalitet.
Signaler og koblingspunkter: Rammeverket for vårt program, PyQt5, inneholder såkalte signaler og koblingspunkter, som bygger videre på de objektorienterte programmeringspråkenes kommunikasjon mellom objekter. Som regel vil en handling utført av bruker utløse et signal, hvorpå dette kan sendes videre til og håndteres av et annet objekt.
Datastrukturer: En struktur hvis oppgave det er å oppbevare og gjenvinne data, som f.eks. lister, trær, hashtabeller osv. I programmet brukes ofte ordbøker, hvor hvert nøkkelord har en verdi, være det et tall, streng eller en annen datastruktur. Alle disse har fordeler og ulemper med tanke på utviding med eller henting av data.
Serialisering: Serialisering går ut på å omgjøre informasjon i en datamaskins internminne til en sekvens byter som så kan lagres i en datafil. Data i disse filene kan så oversettes og lastes tilbake inn i minnet, noe som kalles for ***parsing***.

2.2.2 Python

Python er et objektorientert programmeringspråk med lettlest syntaks og lett tilgang til biblioteker, som utvider det grunnleggende språket med en rekke ny funksjonalitet. Det faktum at Python er et høynivåspråk medfører dessverre ulemper, da det ikke er like raskt som for eksempel Java ved behandling av mer avanserte datastrukturer og iverksetting av søkealgoritmer. Dette har ikke stoppet språket i å ha steget kraftig i popularitet i løpet av de siste fem årene, der det nå er på andre plass på verdensbasis [19] og neppe vil rykke ned med det første. Flere studieprogrammer har derfor valgt å introdusere objektorientert programmering med Python istedenfor Java, deriblant vårt eget Universitet i Stavanger, noe som medfører bredere kunnskap om språket og som forhåpentligvis vil gjøre etterfølgende arbeid på vår applikasjon lettere.

2.2.3 Grafisk Brukergrensesnitt (GUI)

Et grafisk brukergrensesnitt er som regel et grensesnitt mellom bruker og programvare (eller fastvare), hvor brukeren har mulighet til å styre programmet innenfor visse grenser mens programmet da gir visuell tilbakemelding. Dette kan være alt fra et operativsystem til vår applikasjon som kan kjøres uavhengig av operativsystem, så lenge det støtter Python.

2.2.4 PyQt

For denne oppgaven har vi valgt å bruke PyQt rammeverket (versjon 5). PyQt virker å være et populært alternativ til Pythons eget GUI bibliotek Tkinter, med mer utbredt funksjonalitet - omkring 1000 klasser og mer stilrent design [21]. Det er skrevet i C++, som tillater for eksempel bedre håndtering av tråder ved hjelp av innebygde signaler og koblingspunkter.

2.2.5 GitHub

Github er et populært samarbeidsverktøy, hvor man lett kan lage et oppbevaringssted for kode på nettet for så å gi tilgang eller skriverettigheter til andre. Hovedformålet er å gi brukere et enkelt grensesnitt for endring av kode samtidig, samt oversikt over statistikker og tildeling av oppgaver. Github blir brukt av programvareutviklere over hele verden [17] og har vært et uunnværlig verktøy, spesielt i disse tider.

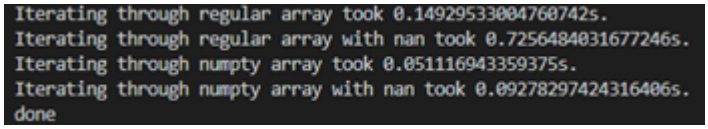
2.3 Moduler

2.3.1 OS

Dette er en standard modul i Python og blir brukt for å kunne hente innhold som pasientfiler fra ulike mapper.

2.3.2 Numpy

Numpy er et mye brukt Python-bibliotek brukt til behandling av arrays og matriser. En array er en samling elementer og i Python brukes "lists" til dette. Python sine lists anses å være trege å prossesere, men der kommer Numpy inn og tilbyr bruken av numpy-arrays som skal være opptil 50 ganger raskere enn Python sine lister. Til dette programmet ble bruken av numpy-arrays utrolig gunstig for en hurtigere plotting av grafer sammenlignet med Python-lister.



Figur 2.1: Klar indikasjon på at iterering og manipulering av Numpy lister er raskere.

En av årsakene til dette har å gjøre med hvordan array-ene lagres i minnet. Numpy klarer å lagre samme mengde elementer på en mer effektiv måte som tar mindre minne og vil dermed bli raskere prossesert. Forskjellen blir mer tydelig desto større arrayene er, og til vårt tilfelle der hver graf kan bli representert av en arary på sekshundre tusen element, vil hver numpy-array kreve 16.8Mb, mens python-arrays vil kreve 21.6Mb. Formelen brukt for å regne ut minnestørrelsen (bytes) til python arrays er: $64 + 8 * n + n * 28$, der n er antall elementer i ararayen. Formelen for numpy arrays er: $96 + n * 8$

2.3.3 Mat4Py

Dette er en pakke som tillater innlesing av matlab filene pasientdataen er lagret som.

2.3.4 Json og Pickle

JSON er en forkortelse for Javascript Object Notation og er en lettleselig tekst-basert måte å lagre informasjon. Python sin standard JSON modul

tillater utviklere å lese og lagre pythonisk datastruktur til en tekstfil i dette lettleselige formatet. Til dette programmet brukes det hovedsakelig for å lagre ulike brukerinstillinger samt filstien til hvor pasientfilene ligger.
Pickle modulen serialiserer data. Dette tillater å lagre et utbredt utvalg datastrukturer, deriblant Numpy lister, noe ikke JSON kan, men formatet er uleselig for mennesker. I dette programmet konverteres matlab filene til pickle filer for å unngå mat4py sine tidkrevende, interne rutiner for innlesing av pasientdata.

Kapittel 3

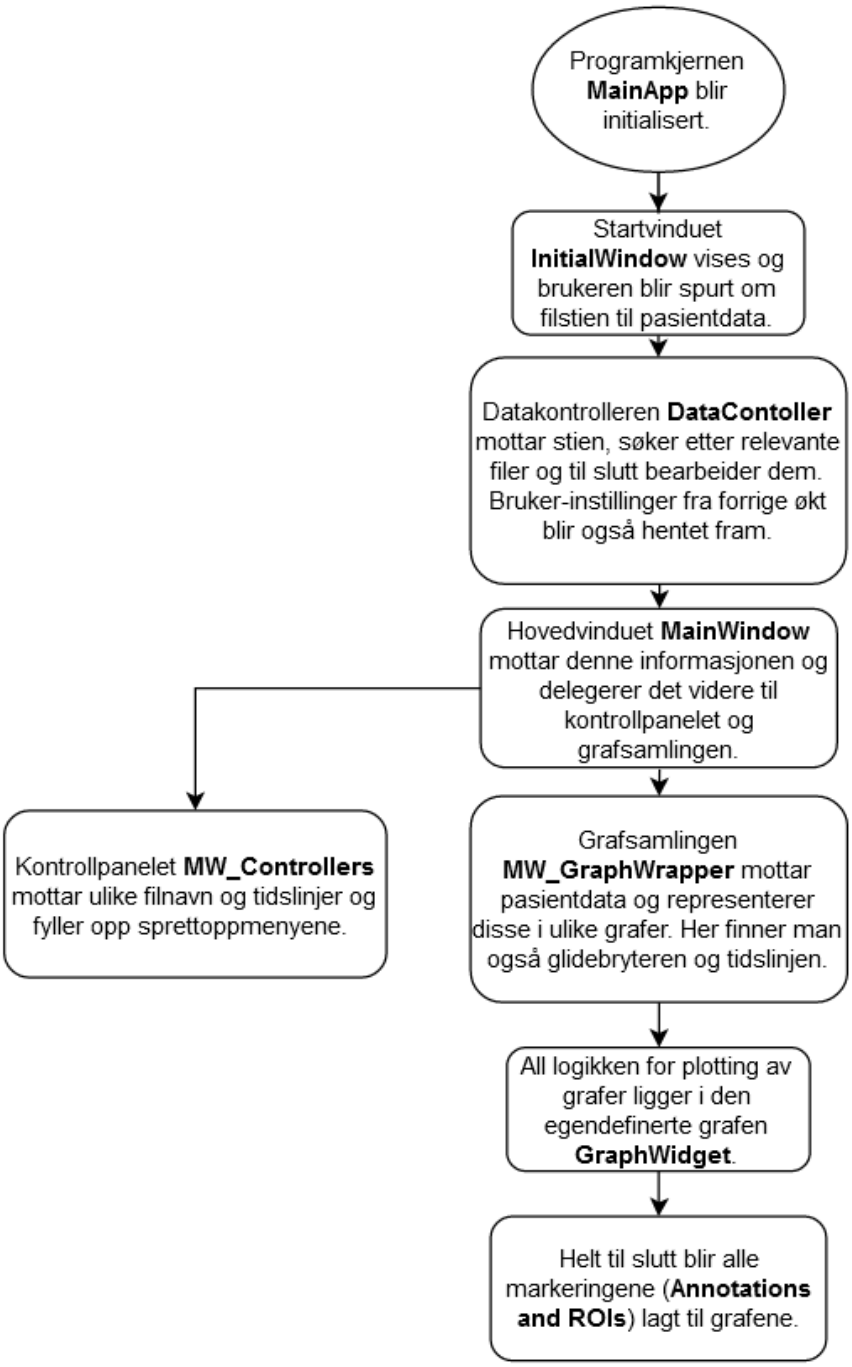
Sammenfatning

Dette kapittelet vil inneholde en kort forklaring på programmets struktur og hvordan vi deler grensesnittet inn i ulike klasser. Målet er å oppnå en lav programvareavhengighet og høy kohesjon. Kohesjon refererer til hvor fokuserte oppgaver de ulike klassene er ansvarlig for, og programvareavhengighet, også kalt kobling, omfavner hvor avhengig klassene er av hverandre. Vi vil altså at hver klasse skal ha tydelige oppgaver, og skal kunne utføre disse med liten kunnskap om tilstanden til andre klasser.

Dette målet om å oppnå lav kohesjon kan sammenlignes med hvordan hovedkortet til en pc fungerer. På hovedkortet finnes det ulike komponenter (prosessor, GPU, RAM), som kan byttes ut uten å skape problemer. Dette er fordi komponentene aldri direkte kommuniserer med andre komponenter. I stedet sender de signaler til hovedkortet og lar denne ta seg av kommunikasjonen, med et håp om at den vil sende tilbake informasjonen det ble spurt om. For å illustrere den motsatte effekten, ville et kretskort med lav kohesjon ha komponenter som prøver å direkte hente informasjon fra andre komponenter. Direkte kommunikasjon må gjøres ved å referere til de spesifikke komponentene. Om en slik komponent blir fjernet, vil referansen ligge igjen i andre komponenter, og en hel del feil vil oppstå fordi det som refereres ikke lenger er til stedet.

Til vårt formål har vi da valgt å dele inn grensesnittet til ulike klasser som hver skal ha sine spesifikke oppgaver, og kommunikasjonen mellom klassene vil foregå gjennom programkjernen kalt *MainApp*. Dette kapittelet vil inneholde en kortfattet gjennomgang av de ulike klassene og hva deres oppgaver er. I figur 3.1 vises en oversikt over hvordan de synlige klassene til hovedvinduet er plassert. I figur 3.2 ser vi flyteprosessen til alle elementene i programmet.

Figur 3.1: Hovedvinduet



Figur 3.2: Flytdiagram for hvordan klassene henger sammen.

3.1 Datamateriale

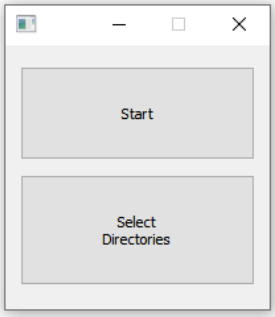
Datamaterialet brukt av dette programmet er sammensatt av to filer per tilfelle pluss en felles fil (til sammen 20 tilfeller). *CASEXX.mat* i BCG mappen, *CASEXX.mat* i LP15 mappen samt en *metadata.m* fil, hvor *XX* står for nummerering.
CASEXX.mat filen i BCG mappen inneholder vektorer med signaler for BCG1, BCG2 signalene samt en frekvensverdi.
CASEXX.mat filen i LP15 mappen inneholder vektorer signaler for CO_2, PPG, IMP, EKG, VENT samt en frekvensverdi.
Metadata.m filen inneholder en rekke informasjon og resultater av autodeteksjonsalgoritmer for alle tilfellene. De mest relevante for vårt program er dato og tid for tilfellet, tallverdier for forskyvning av CO_2 og BCG grafene, lister med annoteringsnavn og -tid og todimensjonelle tabeller med tider for uthevede områder.

3.2 Programkjernen (MainApp)

Dette er selve kjernen av programmet, hvor programmet startes og hvor alle signaler tas imot fra komponenter og sendes videre. For å referere tidligere analogi så vil dette være hovedkortet vårt.

3.3 Startvindu (InitialWindow)

Etter at programmet har startet er dette det første vinduet som åpnes, se Figur 3.3. Hovedmålet til denne klassen er å finne ut hvilken mappe pasientfiler og diverse andre innstillinger ligger for å så signalisere denne informasjonen videre til datakontrolleren. Vinduet inneholder bare to knapper, en til å starte programmet normalt (dersom pasientfilene har blitt lokalisert) før, og en til å kunne endre hvor pasientfilene ligger dersom de skal ha blitt flyttet på.



Figur 3.3: Startvindu

3.4 Datakontrolleren (DataController)

Stien til pasientfilene vil bli gitt til datakontrolleren og dens oppgave blir dermed å hente, konvertere til pickle filer om nødvendig, og bearbeide denne informasjonen. Den vil også hente og lagre diverse data om forskjellige innstillinger og markeringer. I det den er ferdig med å bearbeide pasientdata, signaliseres dette videre til Hovedvinduet.

3.5 Hovedvinduet (MainWindow)

Dette er hovedvinduet til programmet og inneholder alle kontrollene og grafene samt deres tidslinjer. Videre har det blitt gjort en beslutning om å dele vinduet opp i to klasser, en klasse for alle kontrollene, og den andre til framvisningen av alle grafene. Som resultat av oppdelingen vil programmet være mer kohesivt da alle oppgavene som tidligere ble gjort av en klasse, vil nå bli fordelt på to, og kommunikasjonen mellom disse to klassene vil bli utført fra hovedvinduet.

3.6 Kontrollpanel (MW_Controllers)

Som kontroll-seksjonen av hovedvinduet, vil denne klassen ta for seg alt fra å la brukeren velge pasientfil, markere signaler, eller å velge mellom, legge til, eller fjerne tidslinjer. Både å velge pasientfil og endre tidslinjer krever kommunikasjon med datakontrolleren og disse signalene må derfor behandles i MainApp.py. Signalet om å velge hvilke grafer som skal vises trenger derimot bare å kommunisere med den andre del-komponenten og dette signalet kan derfor behandles i MainWindow.py

3.7 Grafsamling (MW_GraphWrapper)

Denne klasse har som hovedoppgave å samle alle grafene på ett sted for å ha en mer oversiktlig måte å ta i mot signaler og bruke dette til å bestemme hva som skal vises i de ulike grafene. Den skal og kunne tillate forskjellige funksjonaliteter som endring av markeringer, muligheten til å dra og endre på rekkefølgen grafene vises, men også la denne framstillingen bli lagret til neste gang programmet startes.

3.8 Egendefinert Graf (GraphWidget)

Hver grafkomponent brukt i MW_GraphWrapper.py er sitt eget lille PyQt objekt kalt PlotWidget. Desverre har den innebygde grafen begrenset med funksjonalitet, så vi bestemte oss for å heller lage vår egen graf-komponent, basert på PlotWidget. Komponenten vil arve all funksjonalitet fra PyQt-originalen, men gir oss muligheten til å legge til eller overskrive ulike funksjoner.

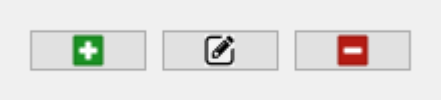
3.9 Tidslinje (AnnotationsWidget)

Når hovedvinduet åpnes, vil brukeren kunne se en standard tidslinje nederst på grensesnittet som viser hvor ulike perioder av interesser befinner seg.

Eksempler på slike perioder er Normal Respiration"og "Hyperventilation".

3.10 Konfigurasjonsvindu (TimelineSettings)

For videreutvikling kunne det være praktisk for en bruker og ha muligheten til å lage nye tidslinjer for og legge til egne meldinger om perioder av interesse. Vi har derfor i kontrollpanelet lagt til knapper, se figur 3.4, som lar brukeren lage nye, endre, eller fjerne eksisterende tidslinjer. Det gjenstår fortsatt å la brukeren fylle disse tidslinjene med egne annotasjoner. Ettersom alle signalene allerede er tilstedet, kan dette bli lagt til i løpet av mai/juni, men vil ikke være med i versjonen som blir vurdert. Vinduene som åpnes når knappene blir brukt, vises i figur 4.6.



Figur 3.4: Konfigurasjonsknapper

3.11 Hjelpefunksjoner (Utility)

Ulike funksjoner som kan tenkes gjenbrukt ofte og å ha mer alminnelig bruk og dermed ikke nødvendigvis tilhøre en spesifikk klasse plasseres her som statiske funksjoner. Statiske funksjoner er ikke avhengige av at det er laget et objekt av deres klasse og kan dermed påkalles når som helst.

3.12 Uthevede områder (PointROI, MinMaxROI, SectionROI, AddROI og CustomViewBox)

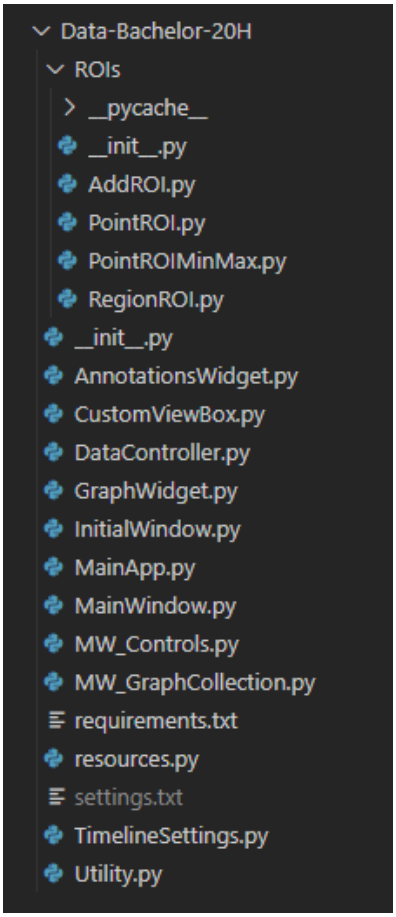
Flere av grafene har en rekke områder som krever utheving i form av et punkt eller område, som hjerteslag. Pyqtgraph har allerede implementert en del «Region of Interest» (ROI), men da disse er til mer generell bruk, så har vil valgt å lage egne underklasser med skreddersydd funksjonalitet

for endring, sletting og tilføying av disse. I tillegg måtte visningsboksene utvides med egen funksjonalitet for de relevante grafene.

Kapittel 4

Metode

Vi vil i dette kapittelet utdype mer om hvordan hver klasse er oppbygget samt fremgangsmåten for hvordan vi har valgt å utvikle selve grensesnittet. I figur 4.1 har vi en oversikt over alle filene i programmet.



Figur 4.1: Alle filene i programmet

4.1 Startvinduklassen (InitialWindow)

Etter programstart er denne klassen ansvarlig for det første vinduet som åpnes. Se figur 3.3. Vinduet har to knapper, begge med ett mål om å finne filstien til der pasientfilene ligger, og dermed sende ett signal med denne informasjonen videre til datakontrolleren.

4.1.1 Be om filsti (requestDirectory())

Første gang programmet startes vil begge knappene føre til at følgende funksjon vist nedenfor blir kjørt. Den vil åpne filutforskeren og spørre brukeren om å velge hvor pasientfilene ligger. For å forhindre brukerfeil ved valg av mapper, har vi laget to statiske variabler øverst i klassen, og forventer at brukeren velger mapper med følgende navn. Når programmet senere avsluttes vil mappenes filsti, samt andre instillinger, bli lagret som json format i en tekstfil fil med navn settings.txt

```
DIRECTORY_NAME_DATASET = "2 DATASET"
DIRECTORY_NAME_ANNOTATIONS = "5 ANNOTATIONS"
```

```
def requestDirectory(self):
datasetPath = qtw.QFileDialog.getExistingDirectory(self, ...
    "Select 2 DATASET folder.")
datasetName = datasetPath.split("/")[-1]
annotationsPath = ...
    qtw.QFileDialog.getExistingDirectory(self, "Select 5 ...
    ANNOTATIONS folder.")
annotationsName = annotationsPath.split("/")[-1]
if (datasetName == self.DIRECTORY_NAME_DATASET and
    annotationsName == self.DIRECTORY_NAME_ANNOTATIONS):
    settings = {}
    settings["dataset"] = datasetPath
    settings["annotations"] = annotationsPath
    settings["checkboxes"] = {}
    #Send directory path to the datacontroller
    self.settings_submitted.emit(settings)
else:
    print("Incorrect folder chosen")
self.close()
```

4.1.2 Vis brukergrensesnitt (launchGUI())

Nedenfor vises funksjonen som kjøres dersom Start knappen blir trykket på. Om programmet har blitt kjørt før, vil den lete etter en settings.txt fil som bl.a. inneholder stien til pasientfilene. Funksjonen vil dermed sende innholdet i denne filen videre til datakontrolleren for opphenting og bearbeiding

av diverse data. Dersom det derimot er første gang programmet kjøres, vil startknappen istedenfor kjøre den øverstenevnte funksjonen, requestDirectory(). Begge knappene vil sende ut ett signal med ønsket informasjon. Det lille utsnittet nedenfor viser hvordan signalet blir tatt opp i MainApp og deretter sendt videre til riktig koblingspunkt.

```
self.initial_window.settings_submitted[dict].connect(
    self.data_controller.receiveSettings)
```

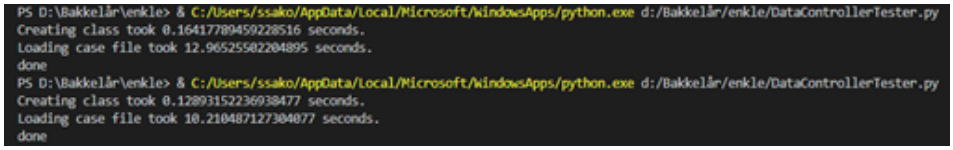
```
def launchGUI(self):
    if "settings.txt" in next(os.walk(os.getcwd()))[2]:
        with open("settings.txt", "r") as json_file:
            #If the file is empty, something is wrong -> ...
                default back to asking for a directory
            print(os.path.getsize("settings.txt"))
            if os.path.getsize("settings.txt") ≤ 2:
                self.requestDirectory()
                return
            #JSON
            settings = json.load(json_file)
            #Send settings containing the directory path ...
                to the datacontroller
            self.settings_submitted.emit(settings)
    else:
        self.requestDirectory()
    self.close()
```

4.2 Datakontrollerklassen (DataController)

Datakontrolleren har som oppgave å hente inn og tolke de nødvendige .mat filene. To av disse filene inneholder vektorer med verdier for EKG, TTI, CO_2, PPG og BCG grafene, mens den siste filen, metadata.mat, inneholder en del informasjon om alle episodene samt automatiske deteksjoner som skal markeres på grafene.

4.2.1 Henting av nytt tilfelle (getCase())

I et forsøk på å gjøre programmet raskest mulig oppdaget vi fort at innlasting av data krevde lengst tid. Mat4py biblioteket brukes for å lese inn .mat filene og hente ut deres strukturer med nødvendig data. Et forsøk ble gjort med bruk av flere tråder, uten hell.



Figur 4.2: Øverste resultat med flere tråder, mens nederste resultat er med kun én tråd.

Dette skyldes Pythons GIL (Global Interpreter Lock). Et vanlig problem med flere tråder er tilgang til samme objekt i minnet; hvor man helst vil at en tråd ser seg ferdig før neste setter i gang med å manipulere objektet. Førstnevnte tråd låser dermed objektet til den er ferdig, før så neste tråd kan be om låsen. Dette i seg selv kan føre til andre problemer, som deadlock og Python har dermed implementert GIL med én lås, som i korte trekk medfører at kun én tråd kan kjøre Python kode om gangen. Av figuren ovenfor ser man at flere tråder resulterer i lengre innlastingstid, som igjen skyldes at filene leses inn fra samme disk der man innfører mer overhead. Pyqt i seg selv er skrevet i C++ og har egne metoder for å iverksette flere tråder ved bruk av koblingspunkt og signaler, som kan brukes til å lage et mer responsivt brukergrensesnitt.
Innhenting av data må dessverre heller ses på som sekvensielt i dette tilfellet, da man trenger alle tre filer i deres helhet før man kan bearbeide dataene videre og sende dette av gårde til resten av programmet. Da det ikke ga mening at henting og innlasting av omlag 20MB fra nettdisk skulle ta opp til 11 sekunder, så konkluderte vi med at det sannsynligvis var mat4py sine interne rutiner som var flaskehalsen. Dermed ble det forsøkt å konvertere .mat filene til Pickle filer, som var en forbedring selv om Pickle filene er like store som .mat filene. Da det antas at programmet vil lese inn filer på lokaldisk, så ble samme eksperiment utført på en SSD disk med lesehastighet på 500MB/s og resultatet overgikk forventningene våre; fra en gjennomsnitts ventetid per tilfelle på 3,88 sekunder med mat4py gikk vi til 0,0259 sekunder med pickle. Dette tillater en mye raskere overgang

mellom hvert tilfelle.

4.2.2 Omgjøring av .mat filer til Pickle filer (Utility.convertMatToPickle())

Derfor konverterer datakontrolleren .mat filene ved førstegangsinnlesing til Pickle filer. Best praksis dikterer at gjenbrukt kode settes i metoder, og derfor er hjelpefunksjoner klassen utstyrt med metoder for konvertering av filformatet samt lagring av Pickle filer.

```
@staticmethod
def convertMatToPickle(datasetFilepath, subsetFilepath, ...
    fileName):
    try:
        with open(datasetFilepath + subsetFilepath + ...
            "Pickle/" + fileName + ".p", "rb") as fp:
            dataset = pickle.load(fp)
            if("fullFile" in dataset and ...
                dataset["fullFile"] == True):
                dataset.pop("fullFile")
                return dataset
            else:
                raise Exception("Pickle file corrupted.")
```

Koden ovenfor er et utsnitt hvor en Pickle fil forsøkes å lastes inn. Filen må også gjennomgå en enkel test hvor vi forsikrer oss om at denne faktisk ble lagret fullstendig tidligere. Finnes ikke filen eller svikter testen, så leses .mat filen inn i neste del av funksjonen og disse må så bearbeides en del, da Matlab foretrekker å lagre vektorer i kolonner. Dette resulterer i ei todimensjonal liste som ikke er intuitiv å jobbe med i de fleste andre språk inkludert større ressursbruk. I figur 4.3 kan man allerede se at konvertering av kolonnevektorer til Numpy lister tar mye lengre tid og at rekkefølgen her er viktig. Datasettene for grafene lagres så ved hjelp av *Utility.savePickleFile*(), mens samme metode brukes ved programavslutning for å lagre metadata filen, da verdiene for uthevede områder og annoteringer kan endres på av bruker. I likhet med matlab programmet lagres sistnevnte i følgende format: $< Dato\ og\ tid > metadata.p$.

```
114     def _prepDataset(self, dataset):
115         Utility.flattenVector(dataset)
116         Utility.array2NumpyArray(dataset)
Adding metadata to local variable, prepping all datasets and creating case file took 0.9037940502166748 seconds.
114     def _prepDataset(self, dataset):
115         Utility.array2NumpyArray(dataset)
116         Utility.flattenVector(dataset)
Adding metadata to local variable, prepping all datasets and creating case file took 2.1981866359710693 seconds.
```

Figur 4.3: Bearbeiding av data krever riktig rekkefølge.

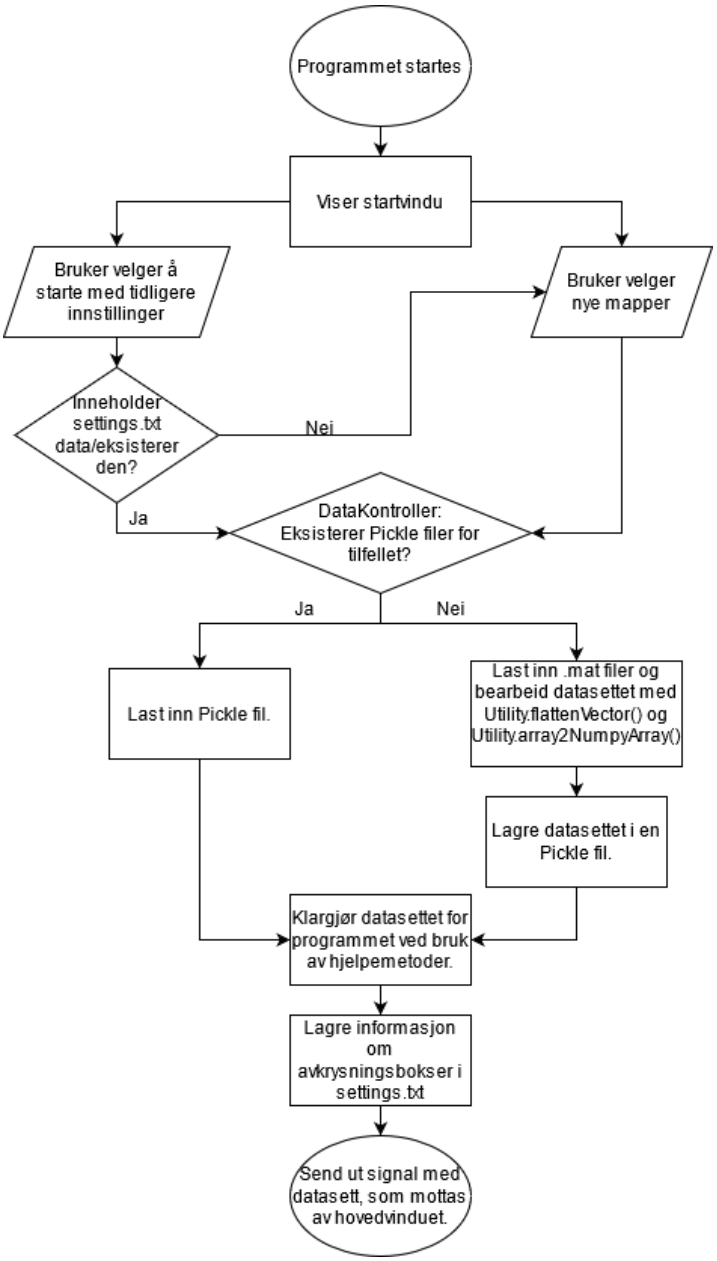
4.2.3 Forskyvning av datasett (Utility.displaceSignal())

Da visningsprogrammet skal tillate forskyvning av CO_2 og BCG grafene, så gjøres dette ved hjelp av en hjelpefunksjon. Brukeren har også anledning til å legge inn egne verdier for dette i selve brukergrensesnittet, så denne er lagt opp slik at den er allsidig nok til å tilpasse grafenes datasett ved forandring i begge retninger. Forflytting til høyre håndteres ved å legge til Not a Number (NaN) i starten av vektoren mens en endring i motsatt retning krever at vektoren trimmes.

```
@staticmethod
def displaceSignal(aList, seconds, frequency):
    finalDisplacement = int(seconds*frequency)
    resultList = None
    if finalDisplacement ≥ 0:
        resultList = np.pad(aList, (finalDisplacement, 0), ...
            "constant", constant_values=(np.nan, 0))
    else:
        resultList = aList[abs(finalDisplacement):]
    return [resultList, finalDisplacement]
```

Resten av programmet er avhengig av at alle datasett for grafene er av lik lengde, noe *Utility.equalizeLengthLists*() funksjonen gjør ved å legge til en erstatningsverdi på slutten av kortere lister. Da det er lite sannsynlig at man vil oppleve et signal likt +uendelig, så ble np.inf brukt til dette. For å unngå et tilfelle der allerede utvidete lister førte til kunstig økning av størst lengde, så beregnes dette ved å finne første reelle tall, fra enden. Videre bearbeides inndata for de relevante grafene med *Utility.normalizeSignals*() før man til slutt omgjør erstatningsverdiene fra listeutjevningen til NaN ved hjelp av *Utility.clearPadding*(), som vil vises som tomrom i slutten av grafene. Før datasettet sendes videre til hovedvinduet, så lagres også informasjon om

avkrysningsbokser i settings.txt. I figur 4.4 kan man se hendelsesforløpet til nå.



Figur 4.4: Flytskjema for programoppstart.

4.3 Hovedvinduklassen (MainWindow)

Denne klassen fungerer som programmets hovedvindu, og er ansvarlig for å samle alle de visuelle komponentene til ett sentralt sted. De visuelle komponentene blir laget i klassen's __init__ funksjon, som kjøres når klassen først blir tilkalt fra MainApp ved programstart. Komponentene er for det meste delt inn i to mindre klasser og dette er for å gjøre koden mer oversiktlig og mer kohesivt. Utsnittet nedenfor er fra hovedvinduet's __init__ funksjon og viser de to mindre klassene bli tilkalt. Disse vil da automatisk kjøre sine egne __init__ funksjoner for å lage deres respektive komponenter.

```
# -------- Main Window Sections --------
self.MW_Controls = MW_Controls()
self.MW_GraphCollection = MW_GraphCollection()
```

4.3.1 Signaler og koblingspunkt

Alle visuelle komponenter blir altså laget ved programstart, men menyer og grafer i hovedvinduet vil være tomme fram til datakontrolleren har sendt signaler med informasjon til MainApp, som vil videreføre det til denne klassen. Utsnittet nedenfor viser hvordan hovedklassen MainApp mottar ulike signaler fra datakontrolleren, og sender det så videre til tre forskjellige koblingspunkt i hovedvinduet. Disse tre koblingspunktene kan ses under sistnevnte utsnitt. Koblingspunktene har som formål å videre fordele dataen til ulike funksjoner som skal bl.a. fylle grafer og sprettoppmenyer med ny informasjon.

```
# -------- Signaler & Koblingspunkt --------
self.data_controller.filenames_submitted.connect(
self.main_window.receiveFilenames)
self.data_controller.case_submitted.connect(
self.main_window.receiveNewCase)
self.data_controller.case_submitted.connect(
self.main_window.show)
```

```
    def receiveFilenames(self, filenames):
        self.MW_Controls.showCases(filenames)

    def receiveTimelines(self, timelines):
        self.MW_Controls.showTimelines(timelines)

    def receiveNewCase(self, case):
        self.initializeWindowSize(case["settings"])
        self.MW_Controls.createCheckboxes(case["settings"])
        self.MW_Controls.receiveInputValues(
        co2=case["metadata"]["t_CO2"], ...
            bcg=case["metadata"]["t_bcg"])
        self.MW_GraphCollection.setDataLength(
        len(case["data"]["s_ecg"])) #Can use any signal (all ...
            same length)
        self.MW_GraphCollection.plotGraphs(case)
```

4.3.2 Gjenopprett vindustørrelse (initializeWindowSize())

Hensikten til denne funksjonen er å skape en mer flytende brukeropplevelse ved gjenopprette grensesnittets størrelse og posisjon fra brukerens forrige økt. Om det er første gang grensesnittet startes, vil vinduet bli sentrert og oppta 75% av skjermens høyde og bredde.

```
        def initializeWindowSize(self, saved_settings):
            self.settings = saved_settings
            if "gui_box" in self.settings.keys():
                saved_box = self.settings["gui_box"]
                current_box = self.geometry()
                current_box.setRect(saved_box[0], ...
                    saved_box[1], saved_box[2], saved_box[3])
                self.setGeometry(current_box)
            else:
                #Get the QRect dimensions of the user's screen
                desktop_screen = ...
                    qtw.QDesktopWidget().screenGeometry(-1)
                screen_width = desktop_screen.width()
                screen_heigth = desktop_screen.height()
                gui_width = screen_width*0.75
                gui_height = screen_heigth*0.75

                #Enter new QRect values
                desktop_screen.setWidth(gui_width)
```

```
            desktop_screen.setHeight(gui_height)
            desktop_screen.moveLeft(math.floor((
            screen_width-gui_width)/2))
            desktop_screen.moveTop(math.floor((
            screen_heigth-gui_height)/2))
            self.setGeometry(desktop_screen)
```

4.3.3 Direkte tilbakemeldinger (showStatus(), hideStatus())

Helt nederst i hovedvinduet har vi lagt til en statuslinje, som er en pyqt komponent med mål om å kunne oppdatere en bruker med ulike meldinger. Utsnittet nedenfor viser koblingspunktet *showStatus*(), som kan ta i mot signaler fra alle plasser i programmet der man kunne tenke seg å signalisere en melding til brukeren. Statusbaren tar opp en liten plass nederst i grensesnittet, men for at den ikke skal ta opp tomrom, vil den bli gjemt dersom ingen meldinger blir vist. Funksjonen vil også hindre at resten av grensesnittet flytter og tilpasser grafene hver gang komponenten nederst blir vist eller skjult. Det blir brukt en QTimer for å, etter ønsket tid, automatisk kjøre funksjonen *hideStatus*() til å gjemme komponenten.

```
    @qtc.pyqtSlot(str, int)
    def showStatus(self, message, timer):
        if self.statusBar.currentMessage() == "":
            self.statusBar.show()
            self.statusBar.showMessage(message, timer)

            geometry = self.geometry()
            geometry.setHeight(geometry.height()+26) ...
                #statusbar pixelheight = 26
            self.setGeometry(geometry)

            self.timer = qtc.QTimer()
            self.timer.timeout.connect(self.hideStatus)
            self.timer.start(timer)
```

```
    def hideStatus(self):
        self.statusBar.hide()
        geometry = self.geometry()
        geometry.setHeight(geometry.height()-26) #statusbar ...
            pixelheight = 26
```

```
        self.setGeometry(geometry)
        self.timer.stop()
```

4.3.4 Avslutningshendelse (closeEvent())

Dersom brukeren avslutter programmet vil programmet først aktivere en "avslutnings-hendelse". Hendelser er innebygde funksjoner som har egne navn, og de brukes til å kjøre kode ettersom ulike hendelser oppstår. For å kjøre sin egen kode etter slike handlinger, må man overskrive de innebygde funksjonene ved å lage sin egen funksjon med samme navn. Ved å overskrive slike funksjoner er det fare for at man fjerner innebygd funksjonalitet, men for å forhindre dette kan man tilkalle *super.hendelsenavn*(). Dette sørger for at originalkoden blir kjørt før man legger til egen funksjonalitet. I kodeutsnittet nedenfor kan man se ett slikt eksempel, hvor vi overskriver funksjonen for avslutningshendelsen med mål om å lagre ulike brukerinnstillinger som har blitt endret underveis.

```
    def closeEvent(self, argv):
        super().closeEvent(argv)
        self.saveMetadata()
        self.MW_Controls.saveCheckboxStates()
        self.MW_GraphCollection.saveDockState()
        self.saveWindowSize()
```

4.4 Kontrollpanelklassen (MW_Controls)

Denne klassen fungerer som en spesifikk seksjon av hovedvinduet med hovedmål om å samle alle de visuelle komponenter relatert til brukerstyrte innstillinger. Eksempler på slike interaktive komponenter er de ulike menyene for å velge pasientfil og tidslinjer, eller avkrysningsbokser for å bestemme hvilke signaler som skal vises eller skjules. For at disse komponentene skal kunne ha en påvirkning må deres innebygde hendelser knyttes til egendefinerte funksjoner. Utsnittene nedenfor er tatt fra klassens __init__ funksjon, og viser to eksempler på dette. Det ene utsnittet viser hvordan hendelsen til to menyer blir koblet til hver sin funksjon, med mål om å sende

ut ett signal for at grensesnittet skal oppdateres basert på brukerens valg. Det andre utsnittet viser hvordan klikkhendelsen til ulike avkrysningsbokser blir knyttet opp for at ulike grafer skal vises eller skjules.

```
self.dropdown_cases = ...
    qtw.QComboBox(currentIndexChanged=self.emitNewCaseIndex)
self.dropdown_timelines = ...
    qtw.QComboBox(currentTextChanged=self.changeTimeline)
```

```
self.checkboxes["s_ecg"] = qtw.QCheckBox("ECG (mV)", ...
    clicked=lambda:self.emitCheckboxState("s_ecg"))
self.checkboxes["s_CO2"] = qtw.QCheckBox("CO2 (mmHg)", ...
    clicked=lambda:self.emitCheckboxState("s_CO2"))
self.checkboxes["s_ppg"] = qtw.QCheckBox("PPG (SpO2)", ...
    clicked=lambda:self.emitCheckboxState("s_ppg"))
```

4.4.1 Fylling av sprettoppmenyer (show...())

De to funksjonene i utsnittet nedenfor har som oppgave å fylle sprettoppmenyene med data som har blitt sendt fra MainWindow (se Figur 4.3.1). Bruken av blockSignal() er for å forhindre at menyene sin currentIndexChanged hendelse skal aktiveres hver gang ett nytt element blir lagt til i menyen.

```
def showCases(self, filenames):
    self.dropdown_cases.blockSignals(True)
    self.dropdown_cases.insertItems(0, filenames)
    self.dropdown_cases.blockSignals(False)

def showTimelines(self, timelines):
    self.dropdown_timelines.blockSignals(True)
    self.dropdown_timelines.insertItems(0, timelines)
    self.dropdown_timelines.blockSignals(False)
```

4.4.2 Lese inndata (co2InputEntered(), bcgInputEntered())

Jobben til disse funksjonene er å lese brukerens inndata fra to interaktive skrivefelt og bestemme forskyvning av CO_2 og BCG grafene. Det skal så sjekkes at brukeren har tastet inn en lovlig verdi, før informasjonen signaliseres videre og blir motatt av koblingspunktet i $MW\_GraphCollection$, der forskyvningen justeres i henholdsvis $displaceCO2()$ og $displaceBCG()$. Da forskjellige lokaliteter bruker ulike tallformat, så brukes PyQt sine egne $QLocale()$ metoder for håndtering av dette. Nedenfor vises det hvordan inndata blir godkjent og signalisert. Brukeren vil få en visuell oppdatering på dette i statuslinjen.

```
def co2InputEntered(self):
    number = qtc.QLocale().toDouble(self.co2_input.text())[0]
    self.co2_input.setText(qtc.QLocale().toString(number))
    if 0 ≤ number ≤ 2:
        self.co2 = number
        self.co2_input_submitted.emit(number)
        self.console_msg_submitted.emit("New CO2 value ...
            successfully submitted", 3000)
    else:
        self.co2_input.setText(f"{self.co2}")
        self.console_msg_submitted.emit("Invalid CO2 value ...
            submitted. Allowed values are between 0 and ...
            2.", 5000)
```

4.4.3 Lagring av grafers tilstand (saveCheckboxStates())

Denne funksjonen har som oppgave å lagre hvilke grafer brukeren har valgt å skjule. Dette er for å skape bedre flyt og en mer kontinuerlig opplevelse mellom ulike økter. Funksjonen blir tilkalt når programmet avsluttes, ved å ta i bruk avslutnings-hendelsen til hovedvinduet, som selvinnlysende nok, aktiveres når programmet avsluttes (se figur 4.3.2)

```
with open("settings.txt", "w") as f:
    json.dump(self.settings, f)
```

4.5 Grafsamlingsklassen (MW_GraphCollection)

Dette er klassen til den andre av hovedvinduets to seksjoner. Denne har som formål å samle alle komponenter som har oppgaven om å fremstille pasientdata. Som følger vil da alle graf-komponentene samt tidslinjen havne i denne klassen, noe som fører til en enklere fordeling av instruksjoner. Klassen inneholder også en glidebryter komponent, herved referert til som en slider. Denne ligger nederst i hovedvinduet. Grunnen til at komponenten ikke ble lagt til i seksjonen med de andre kontrollkomponentene, var av pragmatiske grunner for å avverge layout problemer, men også for å slippe å bruke unødvendige signaler som kan forsinke den raske kommunikasjonen som kreves for å automatisk oppdatere grafer samtidig som slideren blir dratt.

4.5.1 Opprettelse av grafkomponenter (plotGraphs())

Nedenfor vises ett utsnitt fra den første funksjonen som blir tilkalt i klassen, og kjøres etter MainWindow sender inn data fra en ny pasientfil. Koden i utsnittet er ansvarlig for å skape tomme graf-komponenter til hvert unike signal motatt. Ettersom de fleste pasientfilene har de samme signalene vil utsnittet bare utføre arbeid ved programstart, etter innsending av første pasientfil. Graf-komponentene blir skapt ved å tilkalle $GraphWidget()$, som er vår modifiserte versjon av pyqt sin PlotWidget komponent. Grafene blir også plassert inni hver sin $Dock()$, noe som tillater brukeren å kunne dra og flytte på rekkefølgen grafene representeres.

For å kunne oppdatere grafene ettersom de blir dratt eller skalert, kreves det aktivering av mus-hendelser som kan brukes for å kjøre ønsket kode. Ved hjelp av StackOverflow fant vi heldigvis ut at dette kunne gjøres ved å legge til ett hendelses-filter, $installEventFilter()$, på en udokoumentert del-komponent av grafen som heter $viewport()$.

```
    for signal in case["data"].keys():
        if signal not in self.graphs.keys() and signal != "fs" ...
            and signal != "s_imp":
            #Prepare a dock for the graphs and add it to the ...
                dock wrapper
```

```
            dock_name = signal.split("_")[-1].upper()
            self.docks[signal] = Dock(f"{dock_name}")
            self.dock_area.addDock(self.docks[signal], "bottom")
            #Move the ecg dock to the top
            if signal == "s_ecg":
                self.dock_area.addDock(self.docks[signal], "top")
            else:
                self.dock_area.addDock(self.docks[signal], ...
                    "bottom")
            #Make a graph for every signal and assign them to ...
                their own docks
            self.graphs[signal] = GraphWidget(signal)
            self.graphs[signal].setLimits(xMin=-500, ...
                minXRange=self.MIN_X_RANGE, ...
                maxXRange=self.MAX_X_RANGE)
            self.graphs[signal].stopPlotting.connect(
            self.blockPlotting)
            self.graphs[signal].viewport().installEventFilter(self)
            self.graphs[signal].getAxis("left").setWidth(w=25)
            self.graphs[signal].setMouseEnabled(x=True, y=False)
            self.docks[signal].addWidget(self.graphs[signal])
```

Ettersom utsnittet ovenfor bare er ansvarlig for å lage og plassere de ulike komponentene, vil fortsatt grafene være tomme, men det skal koden i utsnittet nedenfor fikse. Hver gang en ny pasientfil blir sendt inn, vil koden kjøre gjennom de ulike graf-komponentene og sende inn fersk data som vil oppdatere og tegne inn de nye grafene.

```
    for signal, graphObj in self.graphs.items():
        graphObj.setStartTime(date, time)
        graphObj.setFrequency(sample_rate)
        graphObj.storeData(case)
```

Ettersom brukeren kan bevege en individuell graf ved å dra med musen, vil $setXLink()$ i utsnittet nedenfor synkronisere x-aksen til alle grafene slik at om en graf beveger seg, vil de andre følge etter.

```
    for signal, graphObj in self.graphs.items():
        graphObj.setXLink(self.graphs["s_vent"]) #Arbitrary Choice
```

4.5.2 Utregning av glidebryter verdier (computeIncrements())

Ved programstart vil graf-komponentene vise ett vindu som inneholder en mengde datapunkter som til sammen skal vise en periode på 60 sekunder. Ved hjelp av vår zoom funksjonalitet skal vinduet kunne skaleres dynamisk til mellom alt fra 0.5s og 90s. Uansett ny størrelse på vinduet, vil det alltid kreve at man bruker den nye vindustørrelsen til å deretter regne ut hvor mange hakk slideren skal bestå av. Denne utregningen utføres av *computeIncrements*() og ett eksempel blir gitt etter utsnittet.

```
def computeIncrements(self, window_length=0):
    data_length = len(self.case["data"][self.name])
    #Calculate window_length based on frequency (250) and ...
        timeframe (60)
    if window_length == 0:
        self.window_length = self.frequency*self.span
    else:
        self.window_length = window_length
    complete_sections = ...
        math.floor(data_length/self.window_length) - 1
    self.total_increments = complete_sections*5 ...
        #Increments will slide graph by 20%

    incomplete_section = ((data_length/self.window_length)-(
    complete_sections+1))*10
    self.total_increments += math.ceil(incomplete_section/2)
```

Forklaring på utregning: Som nevnt ovenfor vil hvert vindu bare være på noen få sekund, mens hele signalet lett kan bli 40 minutter. Man må først finne ut hvor mange antall hele vinduer signalet kan bli delt opp i. Dette gjøres ved å dele lengden av hele signalet med lengden av ett vindu. Hvert hakk i slideren vil da forflytte grafen med ett helt vindu, men vi har valgt å gange antallet med 5, slik at hvert hakk bare forflytter grafen med 20% om gangen, relativt til størrelsen på vinduet.

Ved å finne antall hele vinduer vil man også sitte igjen med en "rest-sone" på størrelse mellom 0.0 og 1.0 av ett helt vindu. Hvis denne resten for eksempel er 0.68 må man først gange det opp til 6.8 og dermed dele på 2 for å få 3.4. Dette betyr at en graf som forflytter seg med 20% om gangen, må flyttes 3.4 ganger inn i "rest-sonen" for hele signalet skal bli vist. Ettersom slideren ikke kan flyttes en halv gang, blir dette tallet rundet opp til 4 for å kunne

få med seg den siste lille biten av signalet.

4.5.3 Oppdatering av glidebryterposisjon (updateSliderPosition())

Ettersom man kan vise andre deler av signalene uten å bruke slideren (dra selve grafen eller trykke på tidslinjen), vil funksjonen under passe på at dersom dette skjer, vil slideren bli flyttet til hakket som ligger nærmest det som vises på grafen. Dette gjør at brukeren lett kan fortsette å bruke slideren fra grafens nye posisjon. Bruken av *blockSignals*() er for å forhindre at *plotSlider*() funksjonen skal plotte grafen på nytt når slideren følger etter.

```
def updateSliderPosition(self, xPos):
    #Set slider to nearest tick from the current position
    inc_step = self.window_length/5
    nearest_inc = math.floor(xPos/inc_step)
    self.slider.blockSignals(True)
    self.slider.setValue(nearest_inc)
    self.slider.blockSignals(False)
```

4.5.4 Handlingsfilter (eventFilter())

Som nevnt tidligere kreves det anvendelse av mus-hendelser for å oppdatere grafen dersom musen blir brukt til å dra (venstre museklikk) eller forstørre (høyre museklikk) grafen. Utsnittet nedenfor viser *eventFilter*() funksjonen som kjøres automatisk når alle mulige hendelser blir aktivert. Ettersom det bare er to hendelser vi er interessert i, brukes en if-setning for å bare kjøre egen kode dersom hendelse Nr.3 eller Nr.82 har blitt aktivert. Hendelse nr. 3 aktiveres når en av musetastene blir sluppet, mens hendelse 82 aktiveres etter en av grafene har blitt forstørret eller forminsket. På slutten av funksjonen er det viktig at man skriver *return False* for at den innebygde funksjonaliteten til alle de andre hendelsene ikke blir fjernet.

```
def eventFilter(self, o, e):
    if not self.stopPlot:
        if e.type() == 3: #3 -> MouseRelease
```

```
                x_range = self.graphs["s_ecg"].viewRange()[0]
                window_length = math.floor(x_range[1] - ...
                    x_range[0])
                increments = self.computeIncrements(window_length)

                for signal, graphObj in self.graphs.items():
                    graphObj.computeIncrements(window_length)
                    if x_range[0] < 0:
                        x_range[0] = 0
                    graphObj.plotPosition(math.floor(x_range[0]))

                self.updateSliderPosition(x_range[0])
            elif e.type() == 82: #82 -> ZoomRelease
                for signal, graphObj in self.graphs.items():
                    graphObj.updateAxis()
            return False
        return False
```

Dersom hendelse Nr.3 blir aktivert, betyr det at brukeren har dratt grafen til en ny posisjon, og ved slepp av mustetasten vil den nye posisjonen sendes til alle grafene slik at *plotPosition*() kan fylle inn grafene på nytt. Om brukeren forstørrer eller forminsker grafen, vil tiden på x-aksen klumpes eller bli for spredt, og krever at aksen blir oppdatert ved å bruke den nye størrelsen på vinduet. Dette blir utført av graf-komponenten sin *updateAxis*() funksjon.

4.5.5 Lagring av grafers rekkefølge (saveDockState())

Som nevnt tidligere vil hver graf-komponent bli pakket inn i sin egen Dock(), noe som tillater brukeren å dra og flytte på rekkefølgen de vises i. Koden i figur 4.5.5 sørger for at denne rekkefølgen lagres i en lokal fil for ulike instillinger, slik at den kan hentes igjen til neste gang programmet brukes.

```
    def saveDockState(self):
        state = self.dock_area.saveState()
        self.settings["dockstate"] = state
        with open("settings.txt", "w") as f:
            json.dump(self.settings, f)
```

4.6 Grafkomponentklassen (GraphWidget)

Følgende klasse er vår egen modifiserte versjon av pyqt sin graf-komponent PlotWidget. Klassen inneholder for det meste bare en hel mengde funksjoner og kalkuleringer for å bestemme hvor mye, og hvordan dataen skal plottes. Den inneholder også en funksjon for å overskrive komponenten sin x-akse til å kunne vise tekst (tiden målingen ble tatt), istedenfor bare tall. Det vi til nå har kalt for det synlige vinduet til grafen, vil herved bli referert til som grafen's viewbox. Skillet blir gjort fordi selv om brukeren bare ser 60s i viewboxen, vil grafen totalt ha plottet 180s. Dette vil fungere som en slags buffer slik at brukeren fortsatt skal kunne se signalet dersom musetasten blir brukt til å flytte viewboxen sidelengs.

4.6.1 Plotte grafer basert på x-posisjon (plotPosition())

Denne funksjonen kjøres bare etter brukeren har skalert eller forflyttet en graf. Variabelen som blir sendt inn til funksjonen er x-verdien til starten på viewboxen. Funksjonen bruker så denne til å regne ut start og slutt posisjon for hvor mye av grafen som skal plottes. For øyeblikket er grafene programmert til å plotte ett helt vindu ekstra på hver side av viewboxen. Dette er for å gi brukeren en følelse av bedre flyt når viewboxen blir dratt sidelengs. Funksjonen blir vist i utsnittet nedenfor, og passer også på at x-verdien holder seg mellom null og lengden på signalene.

```
def plotPosition(self, viewbox_start):
    self.clear()
    data_length = len(self.case["data"][self.name])
    self.x_start = viewbox_start - self.window_length
    self.x_end = viewbox_start + self.window_length*2

    if self.x_start < 0:
        self.x_start = 0
    elif self.x_end ≥ data_length:
        self.x_end = data_length

    self.time = np.array(list(range(self.x_start, ...
        self.x_end)))
    self.plotSection()
```

4.6.2 Plotte basert på glidebryterposisjon (plotSlider())

Denne funksjonen fungerer på samme prinsipp som plotPosition() når det gjelder å plotte litt ekstra på hver side av viewboxen, men forskjellen ligger i at istedenfor å få gitt x-verdien, vil funksjonen kalkulere seg fram ved å bruke posisjonen til slideren. Ettersom hakkene til slideren er beregnet for å forflytte grafen med 20% om gangen, vil man kunne finne x-verdien å gange sliderposisjonen med 20% av nåværende vindu-størrelse.
Som nevnt tidligere må det programmeres inn at slideren skal følge etter grafen dersom brukeren drar i viewboxen. På lignende måte må man i dette tilfellet programmere inn at viewboxen skal følge etter grafen mens brukeren drar i slideren. Dette gjøres ved bruk av graf-komponenten sin $setXRange(start, slutt)$ funksjon. Uten denne vil viewboxen stå stille mens grafen flyttes ut av synlighet. Utsnittet nedenfor viser hvordan intervallet til både grafen og viewboxen blir kalkulert basert på slideren sin posisjon og nåværende vindustørrelse.

```
    def plotSlider(self, slider_value=0):
            self.clear()
            data_length = len(self.case["data"][self.name])
            low = math.floor(slider_value/5)
            remainder = slider_value%5

            #Set the range of the visible part of the graph
            viewbox_start = low*self.window_length + ...
                math.floor(remainder*(self.window_length/5))
            viewbox_end = viewbox_start + self.window_length

            if slider_value == self.total_increments:
                viewbox_end = ...
                    data_length+math.floor(self.window_length/10)
                viewbox_start = viewbox_end - self.window_length
            elif slider_value == 0:
                viewbox_start = -math.floor(self.window_length/10)
                viewbox_end = viewbox_start + self.window_length
            self.setXRange(viewbox_start, viewbox_end, 0)

            #Set the range of the plotted part of the graph
            self.x_start = viewbox_start - self.window_length
            self.x_end = viewbox_start + self.window_length*2
            if viewbox_start - self.window_length < 0:
                self.x_start = 0
            elif viewbox_start + self.window_length*2 ≥ ...
```

```
                data_length:
                self.x_end = data_length

            self.time = list(range(self.x_start, self.x_end))

            hms = self.start_time[1:8]
            miliseconds = self.start_time[9:11]
            h, m, s = hms.split(":")
            self.plotSection()
```

4.6.3 Oppdatere tid på x-aksen (updateAxis())

En graf-komponent består av flere akser og hver av dem er sin egen PyQt komponent ved navn AxisItem. I utgangspunktet kan disse aksene bare vise tallverdier, men vi vil ha dem til å vise klokkeslett, altså tekst. For å løse dette finnes det to alternativer. Den vanskelige framgangsmåten vil være å lage sin egen modifiserte versjon av AxisItem og overskrive en rekke funksjoner for å få til riktig funksjonalitet. På grunn av stor mangel på dokumentasjon, viste dette seg å være vanskeligere enn først forventet. Vi fikk klokkeslett til å vises, men etter mye problemer med oppdatering av aksen i sanntid bestemte vi oss for å gå for en enklere løsning. Den eneste visuelle forskjellen er at aksen til grafene oppdateres først etter enn skalering istedenfor i sanntid. Dersom noen skulle ha en ambisjon om å få til det første alternativer vil releavente ressurser bli referert under videreutvikling i Kapittel 6.

Ettersom det ble for krevende å lage sin helt egen akse-komponent, tar vi heller i bruk en funksjon til den opprinnelige akse-komponenten. Funksjonen heter *setTicks*(), og tillater utvikleren å manuelt knytte sammen x-verdier og tekst for å kunne vise det på aksen. *setTicks*() tar inn en liste med ett spesielt format som vises i utsnittet nedenfor.

```
[[(x_verdi_1, "klokkeslett_1"), (x_verdi_2, ...
    "klokkeslett_2"), (x_verdi_3, "klokkeslett_3"), ... ]]
```

Som sagt, krever *setTicks*() en liste med x-verdier og dets respektive klokkeslett, og hvordan vi utførte dette vises is utsnittet nedenfor. Ettersom vi bare ser litt av grafen til en hver tid, har vi valgt å lage 15 ticks om gangen. 5 til

viewboxen, og 5 utover hver side som en buffer. For å finne ut hvor mange x-verdier det er mellom hvert tick, deles lengden på viewboxen med 5, og blir lagret som $tick\_interval$. Dette intervallet blir deretter brukt sammen med numpy modulen for å generere en liste med 15 x-verdier. Ved bruk av litt enkel mattematikk vil koden regne ut klokkeslettet til hver x-verdi, basert på klokkeslettet gitt ved $x = 0$. Til slutt blir verdiene plassert i riktig format og deretter sendt inn til $setTicks()$ funksjonen.

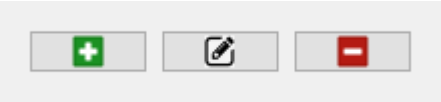
```
def updateAxis(self):
    #Find time given at x=0
    h, m, s = [int(string) for string in ...
        self.start_time[1:8].split(":")]
    ms = int(self.start_time[9:11])

    tick_interval = math.floor(self.window_length/5)
    tick_start = ...
        math.floor(self.x_start/tick_interval)*tick_interval
    tick_values = np.arange(tick_start, ...
        tick_start+15*tick_interval, tick_interval)

    tick_times = []
    for x in tick_values:
        ms_total = x*4 + ms
        ms_added = ms_total%1000
        s_total = math.floor(ms_total/1000) + s
        s_added = s_total%60
        m_total = math.floor(s_total/60) + m
        m_added = m_total%60
        h_total = math.floor(m_total/60) + h
        h_added = h_total%24

        if h_added < 10:
            h_added = f"0{h_added}"
        if m_added < 10:
            m_added = f"0{m_added}"
        if s_added < 10:
            s_added = f"0{s_added}"
        if ms_added < 10:
            ms_added = f"00{ms_added}"
        elif ms_added < 100:
            ms_added = f"0{ms_added}"

        if tick_interval < 100:
            #Show milliseconds when viewbox shows < 2s
            tick_times.append(
            f"{h_added}:{m_added}:{s_added}.{ms_added}")
```

```
            else:
                tick_times.append(
                f"{h_added}:{m_added}:{s_added}")

        #Format the values
        ticks = [list(zip(tick_values, tick_times))]
        axis = self.getAxis("bottom")
        axis.setTicks(ticks)
```

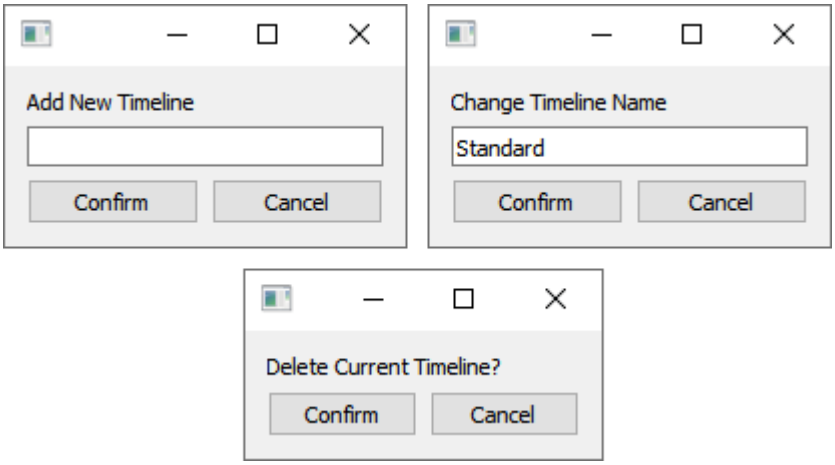
4.7 Tidslinjekonfigureringsklassen (TimelineSettings)

Denne klassen representerer instillings-vinduet som dukker opp dersom brukeren ønsker å legge til, fjerne eller endre navn på en tidslinje. Vinduet som dukker opp vil være ulikt basert på hvilken knapp brukeren trykker på se figur 4.5 og figur 4.6.

Ikonene på konfigurasjonsknappene er vanlig bildefiler som har blitt omdannet til binærtall og lagt til i en egen fil slik at grensesnittet kan så importere denne filen og vise bildene. Dette regnes som bestepraksis for å forhindre problemer med lokalisering av bilder på ulike operativsystem.



Figur 4.5: Konfigurasjonsknapper



Figur 4.6: Konfigurasjonsvinduer

4.7.1 Endring av tidslinjer (Submit())

```
def submit(self):
    if len(self.input.text()) == 0:
        self.label.setText("This field can't be left empty!")
    else:
        self.timeline_submitted.emit(self.option, ...
            self.input.text())
        self.close()
```

Oppgaven til denne funksjonen er å verifisere brukerens valg og signalisere videre informasjonen.

Dette signalet blir både tatt opp av datakontrolleren for å oppdatere filene etter behovet, men også av kontrollpanelet for å kunne oppdatere menyen for tidslinjene. Hvordan signalene blir fanget og delegert videre fra MainApp vises is utsnittet under.

```
self.timeline_window.timeline_submitted.connect(
    self.main_window.MW_Controls.updateTimelines)
```

4.8 Uthevet Sirkelområdeklassen (PointROI)

Formålet til denne klassen er å utheve punkter på EKG, CO_2 og TTI grafene. Dette er som regel et toppunkt i et hjerteslag. Klassen baseres på pyqtgraphs egne *CircleROI*() klasse, som er en underklasse av *EllipseROI*(), som igjen arver fra *ROI*(). Disse klassene uthever en region på grafen og har egne "håndtak", eller handles, som kan brukes til å dreie om, forstørre, flytte på området osv.

4.8.1 Graphwidget._plotQRS()

GraphWidget klassen bestemmer selv om ROI objekter opprettes via signaler fra de respektive avkrysningsboksene. Dette håndteres av *plotSection*() metoden, som kaller sine private metoder for visning av de uthevede områdene. Under vises hvordan disse områdene legges til EKG grafen, som består av både PointROI og uthevet seksjonsområde (RegionROI). Tabellen med relevent data er av størrelsesorden nx3, hvor den har n antall rader og tre kolonner. Den første og siste kolonnen indikerer når hjerteslaget startet og sluttet, mens den andre kolonnen forteller hvor hjerteslagets toppunkt befinner seg, hvis y-verdi finnes i vektoren med EKG signalet.

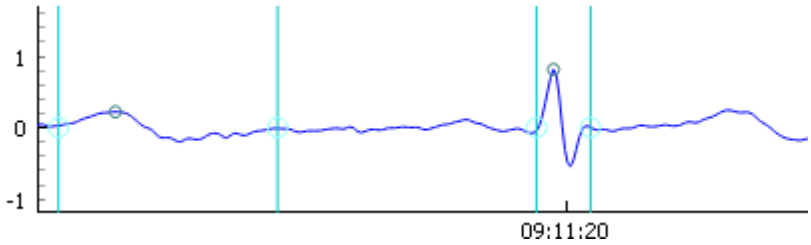
```
    def _plotQRS(self):
        t_qrs = self.case["metadata"]["t_qrs"]
        s_ecg = self.case["data"]["s_ecg"]

        xyRatios = Utility.getRangeRatio(self.getViewBox())
        sizeX = xyRatios["sizeX"]
        sizeY = xyRatios["sizeY"]

        xStart = self.getViewBox().state["viewRange"][0][0]
        xEnd = self.getViewBox().state["viewRange"][0][1]

        for i in range(len(t_qrs)):
            x0 = t_qrs[i][0]*self.frequency
            x1 = t_qrs[i][2]*self.frequency
            xPoint = t_qrs[i][1]*self.frequency
            xPointIndeks = int(np.round(xPoint))
        #If Entry ends after viewRange start and begins before ...
            viewRange end
            if x1 ≥ xStart and x0 ≤ xEnd:
```

```
            self._addRegion(x0, x1, i, "t_qrs", sizeX)
        if xPoint ≥ xStart and xPoint ≤ xEnd:
            self._addPoint("s_ecg", "t_qrs", i, xPoint, ...
                sizeX, xPointIndeks, sizeY, False, (64, ...
                128, 128))
        if x0 > xEnd:
            break
```

Et viktig aspekt ved visning av sirkelområdene er deres størrelse, som avhenger av både vindusstørrelsen samt størrelsen på x- og y-aksene. Regnestykket under viser hvordan $Utility.getRangeRatio()$ beregner dette utifra en fastsatt størrelse på 10 piksler og forholdet mellom antall variabler på aksen og dens lengde i piksler. Da vi ikke ønsket at sirklene skulle være påtrengende idet brukeren forminsket grafen, så ble denne verdien delt på logaritmen av en brøkdel av aksestørrelse. Logaritmen ble valgt da denne stiger sakte.

$$\frac{\frac{aksestørrelse}{vindustørrelse\ piksler} \times fast\ størrelse}{\log_{10}(aksestørrelse \times 0,05)}$$

Etterpå lages PointROI og RegionROI objektene av deres respektive $\_add()$ metoder, som igjen kaller $\_connectSignals()$ metoden for å koble deres signaler til koblingspunkter i det man fokuserer på, endrer eller fjerner de uthevede områdene. Se figur 4.7 for resultatet.



Figur 4.7: EKG graf med uthevede områder.

PointROI klassen bruker det meste av sin overklasses funksjonalitet, men en del særegen oppførsel krever at noen metoder og signaler overskrives, deriblant $hoverEvent()$, som sender ut et signal idet brukeren fokuserer musepekeren på et uthevet område. Signalet forteller $MW\_GraphCollection$ sin $eventFilter()$ om å ikke tegne grafene på nytt, da brukeren samhandler med ROI objektene.

4.8.2 Parallellforskyving (translate())

En måte brukeren kan interagere med et uthevet området er å flytte på det, men da det ikke gir mening at et toppunkt skal kunne dras utfor grafvinduet eller et hjerteslag, så settes allerede grensene i konstruktøren, i form av en *maxBounds* variabel. Den overskrevne *translate*() funksjonen begrenser så områdets bevegelser i sanntid via kodeutsnittet nedenfor, hvor man i tillegg krever at hele sirkelen er innenfor grensene for at ikke grafiske elementer skal overlappes.

```
...
if self.maxBounds is not None:
    r = self.stateRect(newState)
    d = pg.Point(0,0)
    offsetX = 0.5*self.getState()["size"].x()
    offsetY = 0.5*self.getState()["size"].y()
if self.maxBounds["left"] > r.left() + offsetX:
    d[0] = self.maxBounds["left"] - r.left() + offsetX
elif self.maxBounds["right"] < r.right() - offsetX:
    d[0] = self.maxBounds["right"] - r.right() - offsetX
if self.maxBounds["top"] < r.top() - offsetY:
    d[1] = self.maxBounds["top"] - r.top() - offsetY
elif self.maxBounds["bottom"] > r.bottom() + offsetY:
    d[1] = self.maxBounds["bottom"] - r.bottom() + offsetY
newState['pos'] += d
...
```

4.8.3 Flytting av området (GraphWidget._ROImoved())

I det brukeren er ferdig med å flytte det uthevede området så sendes et signal tilbake til *GraphWidget*, som endrer på tabellen i metadata for å reflektere denne endringen. Ny posisjon må derimot dobbeltsjekkes, da det oppstår en del grensetilfeller ved flytting av alle uthevede områder. I tilfelle vi drar en PointROI, så kan det ved en kraftig forminsket graf oppstå en situasjon der sirkelen, selv om den er ganske liten, fortsatt er bredere enn den uthevede seksjonen for hjerteslaget. Dette medfører at sirkelen kan tvinges utfor sin region. Selv om formålet er at bruker skal finjustere deres posisjon ved forstørret graf for økt presisjon, så kan man ikke utelukke den slags oppførsel og metoden sørger for å forhindre dette, for så å informere bruker

om flyttingen av området var vellykket eller ei.

4.8.4 Eget visningsvinduklasse (CustomViewBox)

Fjerning av et uthevet område gjøres i overklassens innebygde metoder. I EKG tilfelle resulterer det i at hele det uthevede området for hjerteslaget fjernes, altså sirkelen og seksjonen. Tilføying av uthevede områder krevde at grafvinduets innebygde menyer ble utvidet og at ønsket oppførsel ble spesifisert tidlig. Da for eksempel et hjerteslag var oppført med tre verdier i metadata, så ble det bestemt at en ny oppføring skulle ha samme struktur. Et eget visningsvindu blir kun gitt til EKG, CO_2 og TTI grafene og når avkrysningsboksene for uthevede områder er huket av.

4.8.5 Tilføying av nye områder (addROI())

I det brukeren høyreklikker på et område i de overnevnte grafene som ikke er okkupert av en eksisterende uthevet region, så vil menyen tillate å legge til et relevant uthevet område, såfremt vilkårene i *addRoi*() sine statiske metoder oppfylles. Her mottas det et signal fra visningsvinduet om musepekerposisjonen, som må forskyves en del, da selve y-aksen bruker en del plass og at musepekeren mister en del nøyaktighet desto større denne er i piksler i forhold til verdiene langs x-aksen. Etter at den faktiske posisjonen langs x-aksen er beregnet, så kreves det at området ikke legges til utfor grafen og at det ser ryddig ut. Sistnevnte betyr at man overholder tidligere regler om hvor nærme hverandre disse områdene kan befinne seg. Dermed trenger man plass på grafen og brukeren oppfordres til å utføre disse handlingene i en forstørret graf, se figur 4.8.

Figur 4.8: Forminsket graf tillater ikke presisjon.

Metoden nedenfor viser en del av fremgangsmåten for dette.

```
@staticmethod
def _addRangeCircleROIs(rawSignal, metaSignal, case, pos, ...
    sizeX):

    rawSignalList = case["data"][rawSignal]
    metaSignalList = case["metadata"][metaSignal]*250

    if (
        (pos - 2*sizeX ≥ 0.0 and not ...
            np.isnan(rawSignalList[int(np.round(pos - ...
            2*sizeX))])) and
        (pos + 2*sizeX ≤ len(rawSignalList) and not ...
            np.isnan(rawSignalList[int(np.round(pos + ...
            2*sizeX))]))
        ):
        #Check for empty list since this requires special ...
            attention, i.e. initiating list.
        if len(metaSignalList) == 0:
            case["metadata"][metaSignal] = np.empty([1, 3])
            case["metadata"][metaSignal][0] = (pos-sizeX)/250
            case["metadata"][metaSignal][1] = pos/250
            case["metadata"][metaSignal][-1] = (pos+sizeX)/250
            return "Roi Successfully added!"
        else:
            result = ...
                AddROI._calculatePosition(rawSignalList, ...
                metaSignalList, pos, sizeX)
            if "pos" in result and "myIndex" in result:
                pos = result["pos"]
                myIndex = result["myIndex"]
                case["metadata"][metaSignal] = ...
                    np.insert(case["metadata"][metaSignal], ...
                    myIndex, [(pos - sizeX)/250, pos/250, ...
                    (pos + sizeX)/250], axis=0)
            return result["msg"]
    else:
        return "Can't add ROIs outside of graph."
```

Tilbakemelding gis til brukeren om hvorvidt et nytt område ble lagt til og i tilfelle det ikke var plass, så gis det tips om å fjerne eksisterende områder eller å forstørre grafen. Til slutt lagres den nye oppføringen.

4.8.6 Uthevet bunn- og toppunktklassen (PointROIMinMax)

Denne klassen arver fra Sirkelområdeklassen, slik at den fungerer ganske likt, men i motsetning til EKG og TTI grafene, så vises det kun uthevede sirkelområder i CO_2 grafen som indikerer lungefunksjon. Dermed endres grenser for flytting av disse objektene til å ikke gå forbi nabosirkler. I tilfelle av et bunnpunkt er venstre grense satt til grafens start eller det forrige parets toppunkt og høyre grense er satt til eget toppunkt. Tilnærmet logikk vil gjelde for et toppunkt. Ved fjerning av et bunn- eller toppunkt fjernes hele paret og tilføying av nye punkter føyer seg etter de generelle reglene for den direkte overklassen, med det unntak at et nytt par ikke kan settes mellom et eksisterende par.

4.9 Uthevet seksjonsklasse (SectionROI)

Implementasjonen av denne klassen er ganske lik de forrige uthevede områdene, men har krevd en annen tilnærming ved endringer. Resultatet av denne klassen er stolpene med håndtak i figur 4.7 på side 43, som finnes i EKG og TTI grafene. Stolpen til venstre i figuren, som indikerer start på hjerteslag, kan ikke flyttes forbi det tidligere hjerteslagets sluttlinje, ei heller forbi eget toppunkt, men holdes i rimelig avstand fra disse, slik at ingen overlapping skjer. Den høyre stolpen, som indikerer slutt på hjerteslag, følger samme logikk.

4.9.1 Flytting av stolpene (movePoint())

Da denne klassen arver fra *ROI*, som er en firkant, så er det en del uønsket oppførsel som måtte fjernes, deriblant rotering og flytting av hele seksjonen, da håndtakene brukes til dette.

```
#Positions cannot overlap maxpoints nor neighboring regions.
if index == 0: #Left handle
    horizontalBounds = self.leftBounds
    if newState["pos"].x() < horizontalBounds["left"]:
        diff = newState["pos"].x() - horizontalBounds["left"]
```

```
            newState["size"].setX(newState["size"].x() + diff)
            newState["pos"].setX(newState["pos"].x() - diff)
        if newState["pos"].x() > horizontalBounds["right"]:
            diff = newState["pos"].x() - horizontalBounds["right"]
            newState["size"].setX(newState["size"].x() + diff)
            newState["pos"].setX(newState["pos"].x() - diff)
    else:
        horizontalBounds = self.rightBounds
        if newState["pos"].x() + newState["size"].x() < ...
            horizontalBounds["left"]:
            diff = newState["pos"].x() + newState["size"].x() ...
                - horizontalBounds["left"]
            newState["size"].setX(newState["size"].x() - diff)
        if newState["pos"].x() + newState["size"].x() > ...
            horizontalBounds["right"]:
            diff = newState["pos"].x() + newState["size"].x() ...
                - horizontalBounds["right"]
            newState["size"].setX(newState["size"].x() - diff)
```

I kodeutsnittet ovenfor vises det hvordan man implementerer flytting av stolpene i metoden ansvarlig for dette, på bakgrunn av reglene nevnt tidligere.

4.10 Tidslinjeklassen AnnotationsWidget

Oppgaven til denne klassen er å lage et vindu hvor de forskjellige annotasjonene kan vises. Når klassen blir initialisert, så lagres det en important_annotations-katalog. Denne katalogen bruker annotasjonene fra metadata-filene som nøkler og 'tags' (eller merkelapper) som verdier. Det er disse merkelappene som vises på tidslinja. Hvis en bruker vil forandre på navnene til merkelappene eller legge til flere merkelapper, så kan det gjøres enkelt ved å redigere på important_annotations-katalogen.

```
    self.important_annotations = {'Power On': 'S',
                                  'Generic':' NR',
                                  'Oxygen': 'HV',
                                  'Nitroglycerin': 'LU',
                                  'Morphine': 'TA',
                                  'IV Access': 'HB',
                                  'Intubation': 'IA',
                                  'CPR': 'FA',
```

```
                                'Epinephrine': 'IV',
                                'Atropine': 'IVS',
                                'Lidocaine': 'OB',
                                'Power Off': 'E'
                                }
```

4.10.1 Plassering av merkelapper (_setTags())

Metoden _setTags() er den eneste metoden i AnnotationsWidget og har som oppgave å lese inn annotasjonene og vise dem inn i plott-vinduet. For selve visningen brukes det Pyqtgraph's såkalte 'Infinitylines'. Disse er egentlig bare linjer som går fra minus til pluss uendelig i vertikal retning. Til disse linjene er det lagt til en merkelapp som tilsvarer taggene i tidligere nevnte important_annotations-katalogen.

```
    def _setTags(self, case):
        self.clear()
        self.setLimits(xMax=len(case["data"]["s_ecg"])-1, xMin=0)
        for i in range(len(case["metadata"]["ann"])):
            if case["metadata"]["ann"][i] in ...
                self.important_annotations:
                tag = case["metadata"]["ann"][i]
                if i == 0:
                    line = pg.InfiniteLine(
                    pos=case["metadata"]["t_ann"][i]*250, ...
                        label=self.important_annotations[tag],
                            pen=pg.mkPen('g', width=2), ...
                                labelOpts={'color': 'w', ...
                                'position': 0.6, 'fill': 'g'})
                    self.addItem(line)
                elif i == len(case["metadata"]["ann"])-1:
                    line = pg.InfiniteLine(
                    pos=case["metadata"]["t_ann"][i]*250, ...
                        label=self.important_annotations[tag],
                            pen=pg.mkPen('r', width=2), ...
                                labelOpts={'color': 'w', ...
                                'position': 0.6, 'fill': 'r'})
                    self.addItem(line)
                else:
                    line = pg.InfiniteLine(
                    pos=case["metadata"]["t_ann"][i]*250, ...
                        label=self.important_annotations[tag],
                            pen=pg.mkPen('b', width=2), ...
```

```
                                labelOpts={'color': 'w', ...
                                'position': 0.6, 'fill': 'b'})
                    self.addItem(line)
```

Kapittel 5

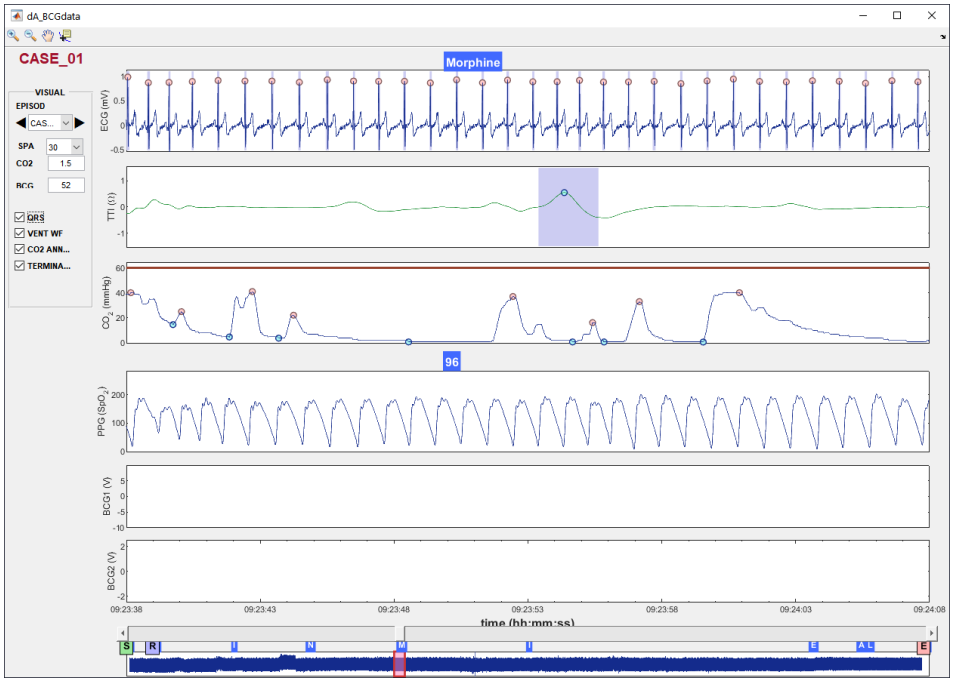
Resultater

I dette kapitlet presenteres vårt ferdige produkt, altså vårt Python-baserte grensesnitt.
Programmet vårt baserer seg på et annet brukergrensesnitt laget i MATLAB, der oppgaven har vært å implementere dette og i tillegg optimalisere og utvide en del funksjonalitet. Følgende delkapitler blir benyttet til å sammenligne programmene.

5.1 Utseende og funksjonalitet

Som nevnt tidligere, så baserer programmet vårt seg på et annet program. I figur 5.1 ser vi et utklipp av MATLAB-programmet.

5.1 Utseende og funksjonalitet



Figur 5.1: Programmet i MATLAB som vårt program baserer seg på. [20]

Fra figuren ovenfor ser vi at vinduet har seks grafer totalt (EKG, TTI, CO_2, PPG, BCG1 og BCG2). Tre av dem bruker spesielle merkeringer for viktige punkter på grafene. På venstre siden av vinduet er det en boks hvor man kan velge pasientfil (case), omfang av grafene (span). Litt lenger ned finner man CO2 og BCG inndataboksene. Ved å justere disse kan brukeren forskyve grafene, dersom ønskelig.
Videre ser man avkrysningsboksene. Disse brukes for å vise eller skjule uthevede områder på grafene.
Helt nederst i vinduet er det en glidebryter som tillater brukeren å bevege seg til angitt sted i grafene og en tidslinje som viser hvor de ulike taggene ligger.

Dette har vi forsøkt å emulere i vårt eget program, se figur 3.1 på side 10. Helt øverst til venstre har vi to nedtrekkslister hvor vi kan velge pasientfil og omfang av grafene. Justeringen kan gjøres vet hjelp av nedtrekkslisten, men vi har også lagt til en mer dynamisk løsning og omfanget kan justeres

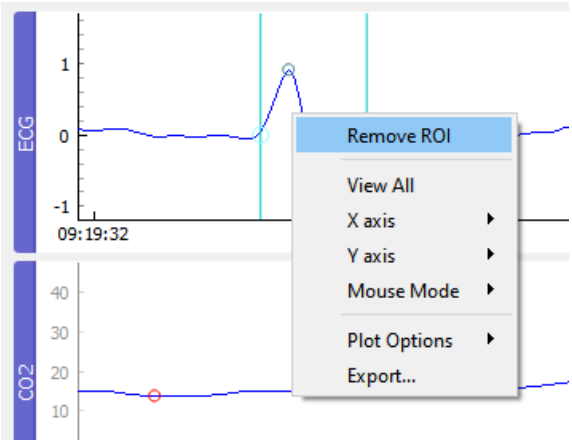
ved å rulle på mushjulet.
Litt lenger til høyre finnes tre avkrysningsbokser. Disse brukes til å vise eller å skjule merkeringene (på lik linje med MATLAB programmet). I vår utgave av programmet har vi derimot fjernet boksen for terminasjon da denne linjen ikke viste seg å tjene noe formål.
Enda lenger til høyre er det to bokser for justering av forskyvning av grafene.
Deretter har det blitt lagt til tre knapper for å legge til, endre eller slette tidslinjer. En annen funksjonaliet som vi har lagt til er at grafene kan kan vises og skjules etter behov. Dette gjøres ved å trykke på avkrysningsboksene helt øverst til høyre.

5.1.1 Omposisjonering av uthevede områder.

Hovedidéen har vært å implementere mye strengere krav til disse områdene med tanke på utseende og grensetilfeller. I korte trekk betyr dette at om man for eksempel har et uthevet område og toppunkt i EKG grafen, så tillater ikke *ROI*-enes *translate* eller *movePoint* funksjoner å flytte på disse slik at toppunktet havner utfor start og slutt for hjerteslag, ei heller tillater disse at start eller sluttpunktet er henholdsvis etter/før toppunktet. Det samme skjer i TTI grafen. I CO_2 grafen er det bunn- og toppunkt. Disse finnes i par i deres respektive lister og brukeren får ikke lov til å dra et bunnpunkt slik at det ender opp etter toppunktet og motsatt.

5.1.2 Sletting av uthevede områder.

I tilfelle man har *PointROI* og *SectionROI* i samme graf, så vil sletting av toppunktet eller området resultere i at både toppunktet og området slettes. I CO_2 sitt tilfelle betyr sletting av et punkt at det tilhørende bunn- og toppunktet også slettes. Se figur 5.2.



Figur 5.2: Eksempel på hvordan vi kan slette merkeringer.

5.1.3 Tilføying av uthevede områder.

Her vil da et toppunkt legges til samt et område rundt, med en forhåndsbestemt bredde så disse ikke overlapper hverandre eller andre områder. I tilfelle CO_2 så legges bunn- og toppunkt ved siden av hverandre på grafen. Se figur 5.3

Figur 5.3: Eksempel på hvordan vi kan legge til nye merkeringer.

5.2 Ekstra funksjonalitet

I tillegg til å implementere hovedfunksjonaliteten, så har vi hatt fokus på å legge til annen funksjonalitet hvis formål har vært å forbedre brukeropplevelsen og forenkle forskningsarbeid.

5.2.1 Justere omfanget på grafer.

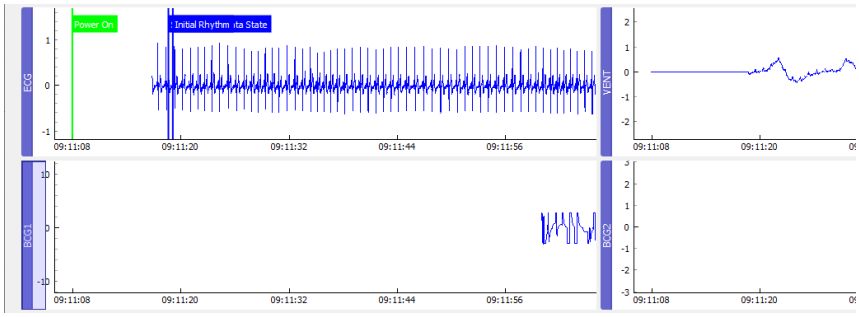
Dette kan gjøres med mushjulet, som er intuitivt, eller grafen sin visningsboks egen kontekstuelle meny, som tillater bruker å spesifisere omfanget til begge akser.

5.2.2 Navigasjon langs grafen

Til navigasjon kan brukeren trykke på ønsket punkt i tidslinjen, men også flytte på glidebryteren eller dra grafene direkte med musen, noe som vil plotte dem i rimelig jevnt.

5.2.3 Omposisjonering av grafene

Grafene kan omposisjoneres fritt i hovedvinduet i ønsket rekkefølge, eller legges side om side. Se figur 5.4



Figur 5.4: Eksempel på flytting av grafer.

5.2.4 Eksportering av grafer.

Synlig del av en graf kan eksporteres i form av et bilde eller tallverdier til en .csv fil, som vil forenkle presentasjon samt forskningsarbeid.

5.3 Hastighet og minnebruk

Etter å ha løst flere utfordringer med tanke på optimalisering så er programmet blitt mye raskere enn det opprinnelige Matlab programmet. Pasienttilfeller etter førstegangsinnlesing kan skiftes mellom nesten sømløst og navigasjon langs grafene skjer nå tilnærmet umiddelbart, avhengig av hvor mange verdier som vises. I tillegg har minnebruken gått ned fra omkring 1500MB ved bruk av Matlab til omlag 109MB ved bruk av Python.

Kapittel 6

Diskusjon

6.1 Valg av rammeverk

Programmeringspråket vi ble bedt om å bruke var Python, men til å lage grafiske grensesnitt er det flere forskjellige rammeverk å velge mellom. Ett populært rammeverk er TKinter, som ligger innebygget i Python, og er kanskje pga. denne grunnen det mest brukte rammeverket. Vi valgte derimot å ikke bruke denne ettersom den til vårt behov, verken er fleksibel, eller hurtig nok til vårt formål. Det hjelper heller ikke at TKinter har en veldig begrenset mulighet for å kunne endre utseende, noe PyQt har muligheten til dersom ønsket skulle finnes.

Når det gjelder de ulike rammeverkene vi kunne importere, var det flere å velge mellom, hver med sine unike styrker og svakheter. Potensielle kandidater vi leste om var blant annet: PySide2, wxPython og Kivy, men vi endte til sist opp med PyQt. Det som skiller PyQt fra de andre kandidatene, er dens ekstreme objektiv orienterte natur som gjør dette til ett stort, robust og fleksibelt rammeverk.

6.2 Fordeler og vansker med PyQt

Fordelene med PyQt er mange. Noe av det viktigste er dens evne til å kjøre på ulike operativsystem. Rammeverket tillater også utvikleren stor fleksibilitet ved at alle komponenter og funksjonaliteter kan endres etter eget behov, i mye større grad enn andre tilgjengelige rammeverk. Disse fordelene kan være en av grunnene til hvorfor PyQt har blitt ett populært valg blant Python utviklere til storskalert applikasjonsutvikling. Disse fordelene kommer derimot ikke uten ulemper ettersom jo mer tilpasselig ett rammeverk er, desto brattere er læringskurven for å kunne forstå og implementere det til eget forbruk.

Selv med populariteten til PyQt tatt i betrakning, er det fortsatt en betydelig mangel på offisiell dokumentering av komponenter og deres funksjoner. Dette gjør det i blant vanskelig, og til tids frustrerende, å finne de riktige funksjonene som må tas i bruk for å implementere en ønsket egenskap.

PyQt er basert på, og er strukturelt likt, programmeringsspråket C++ sitt rammeverk Qt. Qt er derimot er ekstremt godt dokumentert, og ettersom vi var litt kjent med C++ fra før, har dette vist seg å være en svært god kilde til å forstå rammeverkets ulike komponenter [5]. Selv om rammeverkenes struktur er nokså like, gjelder dette desverre ikke navn på funksjoner. Her har heldigvis nettsiden 'Stack Overflow' ofte vært til hjelp [6]. I etterkant fant vi ut at rammeverket PySide, som er nærmest identisk med PyQt, har publisert en relativt god oversikt over navn til rammeverkets ulike funksjoner [15]. Denne kilden ville ha spart oss mye unødig googling om vi fant den tidligere.

6.3 QtDesigner

Dette er et mye brukt visuelt verktøy for å gi utvikleren en lettere måte å bygge opp grensesnittet ved å la brukeren dra inn og plassere ulike komponenter fra en meny. Grunnen til at vi valgte å ikke bruke dette verktøyet er tildels pga. hvordan strukturen til programmet endres dersom verktøyet blir brukt. Når man bruker dette verktøyet, vil mye av koden bli klemt inn i lite oversiktlige UI filer. Vi valgte derfor å programmere grensesnittet manuelt

for å beholde både kontroll og oversikt. Samtidig vil det forhåpentligvis gjøre det lettere for andre å kunne forstå og videreutvikle grensesnittet. En annen ulempe med QtDesigner er at den ikke er særlig egnet for bruk av egen-modifiserte komponenter, noe vi har hatt behov for, og tatt i bruk.

6.4 Uthevede områder

Det ble nevnt tidligere at kravene til de uthevede områdene er mye strengere i dette programmet, da vi ønsket et robust program med stilren utforming. Håndtering av grensetilfeller var et følsomt tema, da man tillater for eksempel programmet å selv finne riktig posisjon til en *SectionROI* om et håndtak dras inn i et NaN område, som regel utenfor grafen, men hva med områder som allerede er plottet utenfor grafen ved programoppstart? En løsning ville vært å endre på eller slette disse ved oppstart, men da formålet med programmet er forskningsarbeid, så ville dette resultert i et forpurret datasett istedenfor å gi brukeren en grafisk fremstilling av feilaktige deteksjoner. Derfor bestemte vi oss for å vise de feilaktige deteksjonene, men ikke tillate bruker å flytte på dem, da det ville kreve enda mer grensehåndtering og funksjoner med lang og uforståelig logikk på grunn av sjeldne problemer.

6.5 Videreutvkling

Da vi har utviklet selve brukergrensesnittet ved hjelp av Python og rammeverket PyQt5, så vil det sannsynligvis være aktuelt å utvide funksjonaliteten i lang tid fremover.
Applikasjonens utseende er ikke blitt vektlagt i stor grad og ser derfor kanskje litt simpelt ut. Dette kan endres på, da PyQt har tilgang til noe som ligner på CSS stilark og PyQtgraph tillater endring av grafenes utseende.
Algoritmer for automatisk deteksjon har blitt diskutert tidligere i oppgaven og er en klar kandidat for fremtidig utvidelse av programmet.
I tillegg kan det tenkes å utvide Datakontrolleren med flere tråder til å konvertere .mat filer til Pickle filer i bakgrunnen. Dette kan gjøres med PyQt sine signaler og koblingspunkt, men en utfordring her vil vært å konvertere relevante datafiler i det brukeren skifter fra et tilfelle til et annet, da det nye tilfellet må fremskyndes i køen, altså håndteres rett etter at datafiler

for nåværende tilfelle i køen er ferdighåndtert.
Vi ser også for oss at det kan være mulig å utvide startvinduet til å holde selve kjerneprogrammet oppdatert via forespørsler til en nettserver, samt skape en aktiv dialog mellom bruker og utvikler om status, feil som må rettes på og ønsket fremtidig funksjonalitet.
Om grafene, i dagens tilstand, blir forstørret eller forminsket vil dens x-akse bare oppdateres etter musetasten slippes. En liten, men krevende, visuell forbedring ville vært å utvikle sin egen akse-komponent, slik at tiden på x-aksen kan oppdateres i sanntid mens skaleringen av grafene pågår. For å få dette til må man først og fremst ha god kunnskap om kildekoden til PyQt sin akse-komponent *AxisItem* [14]. Man må deretter redigere og overskrive viktige funksjoner uten å ødelegge for annen funksjonalitet. Noen forsøk på dette har allerede blitt utført av andre, og kan hjelpe å sette en i riktig spor [1] [2].
I gjeldende versjon vil brukeren ikke ha muligheten til å skrive inn egne meldinger på de ulike tidslinjene. Med tanke på at dette er ett grensesnitt vi ønsker skal være til hjelp, er dette en ekstra funksjonalitet vi skal legge til snarest mulig.
Når det gjelder andre måter programmet kan forbedres, kan en prøve å implementere OpenGL, som vil ta i bruk datamaskinens skjermkort for en enda hurtigere plotting av grafer.

Kapittel 7

Konklusjon

Figur 7.1: I grove trekk. [22]

Oppgaven gikk ut på å utvikle et grensesnitt i Python for analysering av biomedisinske signaler fra hjertestanspasienter. Grensesnittet er basert på ett MATLAB program som tidligere har blitt brukt til dette formålet. En årsak til behovet for ett nytt grensesnitt kommer fra at det tidligere grense-

nittet krevde at brukeren måtte kjøpe ett kostbart MATLAB lisens. Vårt Python grensesnitt krever ingen lisens, og det eneste som kreves fra brukeren er nedlastingen av en liten kjørbar fil. Denne filen kan bli plassert hvor som helst på brukerens datamaskin, men første gang det startes vil brukeren bli spurt om å peke til mappen hvor pasientfilene ligger. Visuelt ligner grensesnittet på det gamle programmet, men med utvidet funksjonalitet og en forhåpentligvis bedre brukeropplevelse. Grensesnittet er i stor grad kontrollert for ulike handlinger, slik at en bruker ikke skal klare å utføre uønskede handlinger eller krasje programmet. Kildekoden ligger i ett offentlig kodelager på nettsiden GitHub (se vedlegg A).

Mye av arbeidet i starten gikk til å bygge opp selve grensesnittet og få representert de ulike signalene. Etter hvert helte arbeidet seg mot å optimalisere, legge til mer funksjonalitet, forbedre brukeropplevelsen og å forhindre uønskede oppførsel. Det ble lagt mye arbeid i å gi brukeren muligheten til å legge til, fjerne, eller dra/flytte på grafenes ulike markeringer. Vi håper at dette er et grensesnitt som kan være til hjelp. Det ville vært fint å videreutvikle applikasjonen, kanskje som en eventuell masteroppgave.

Bibliografi

[1] C. Pascual (cpascual). 2020. URL: https://gist.github.com/cpascual/cdcead6c166e63de2981bc23f5840a98.

[2] I. Egge (iverasp). 2017. URL: https://gist.github.com/iverasp/9349dffa42aeffb32e48a0868edfa32d.

[3] Haralds Arnesen. "EKG". I: *Store medisinske leksikon* (2018). URL: https://sml.snl.no/EKG.

[4] *Ballistocardiography*. URL: https://en.wikipedia.org/wiki/Ballistocardiography. (accessed: 03.05.2021).

[5] The Qt Company. *Qt*. URL: https://doc.qt.io/qt-5/qtwidgets-module.html.

[6] Stack Exchange Inc. *Stack Overflow*. URL: https://stackoverflow.com/.

[7] U.Ayala T.Eftestøl E.Alonso U.Irusta E.Aramendi S.Wali J.Kramer-Johansen. "Automatic detection of chest compressions for the assessment of CPR-quality parameters". I: *Resuscitation, vol. 85* (2014), s. 957–963.

[8] L.Wik J.Nordsteien R.Beck H.Kongsgaard U.Irusta E.Aramendi T.Eftestøl H.Stave R.Bakke J.Ræder. "Prehospital use of biosensors to non- and invasively monitor hemodynamics and blood flow during emergency medicine cases, a prehospital observational feasibility study". I: (2020). URL: https://nextcloud.ux.uis.no/s/JGeyTDzzg6q8Gad.

[9] M. Bjaanes K. Gjesdal. "EKG for sykepleiere: Vurdering av EKG / Skop Systematisk tilnærming". I: (2020). URL: https://oslo-universitetssykehus.no/Documents/Hjerte-lunge-karklinikken/AAA%5C%20-%5C%20Sykepleierkurs%5C%20EKG%5C%20(50ab).pdf.

[10] L.Wik E.Doppler H.Kongsgård T.Eftestøl E.Aramendi U.Irusta M.Dreggevik. "Ballistocardiographic biosensor technology used to monitor blood flow generated by a beating heart and during cardiopulmonary resuscitation." I: (2020). URL: https://nextcloud.ux.uis.no/s/x72qGJkdLACL53b.

[11] D.Shao F.Tsow C.Liu Y.Yang N.Taoi. "Simultaneous Monitoring of Ballistocardiogram and Photoplethysmogram Using Camera". I: *HHS Author Manuscripts* (2016). DOI: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5523454/.

[12] Helge Opdahl. "Kapnografi". I: *Store medisinske leksikon* (2019). URL: https://sml.snl.no/kapnografi.

[13] T.Eftestøl K.A.Håland-Thorsen E.Tøssebrø C.Rong P.A.Steen. "Representing resuscitation data—Considerations on efficient analysis of quality of cardiopulmonary resuscitation". I: *Resuscitation, vol. 80* (2009), s. 311–317.

[14] *Pyqtgraph documentation*. URL: https://pyqtgraph.readthedocs.io/en/latest/_modules/pyqtgraph/graphicsItems/AxisItem.html#AxisItem.

[15] *PySide v1.0.7 documentation*. URL: https://srinikom.github.io/pyside-docs/.

[16] Bjørn-Jostein Singstad. "Fotopletysmografi". I: *Store norske leksikon* (2020). URL: https://snl.no/fotopletysmografi.

[17] *State of the Octoverse*. URL: https://octoverse.github.com/. (accessed: 03.05.2021).

[18] E.Alonso T.Eftestøl E.Aramendi J.Kramer-Johansen E.Skogvoll T.Nordseth. "Beyond ventricular fibrillation analysis: Comprehensive waveform analysis for all cardiac rhythms occurring during resuscitation". I: *Resuscitation, vol. 85* (2014), s. 1541–1548.

[19] *TIOBE Index for May 2021*. URL: https://www.tiobe.com. (accessed: 05.05.2021).

[20] U.Irusta Universitetet i Baskerland. "db_BCGdata v2.5 18-Dec-2020".

[21] *What is PyQt?* URL: https://riverbankcomputing.com/software/pyqt/.

[22] F. Øverli. "Pondus - Hat Trick". I: *Bladkompaniet AS* (2003), s. 139.

Vedlegg A

Installering av program

For å kjøre programmet er det eneste som kreves fra brukeren, å laste ned en kjørbar fil med en størrelse på 47.8MB. Versjon 1.0 av grensesnittet er lagt til som vedlegg i denne oppgaven, men etter eventuelle oppdateringer vil brukeren kunne finne den nyeste versjonen på vårt offentlige kodelager på nettsiden GitHub: `https://github.com/MetisPrometheus/Data-Bachelor-20H`

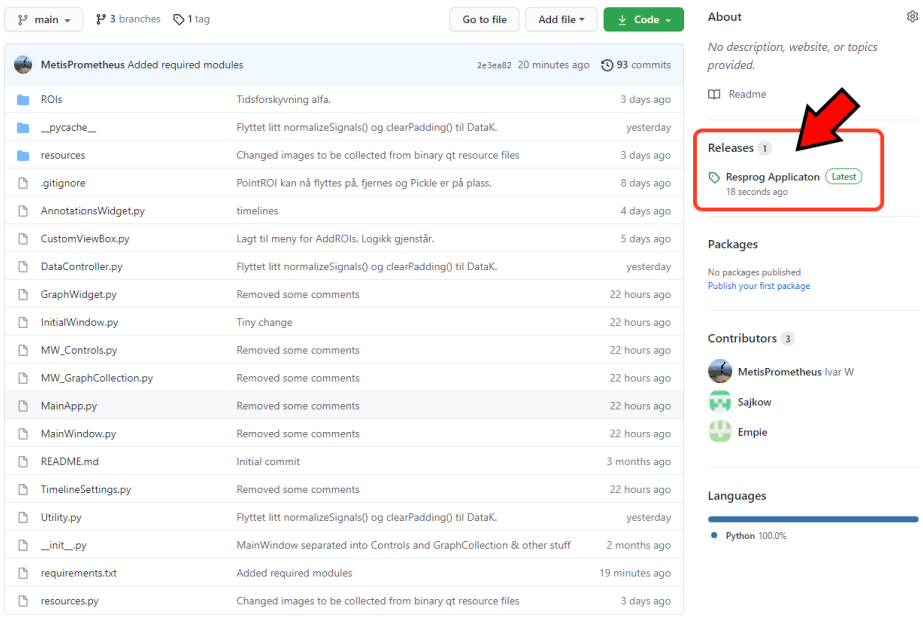
Kodelageret inneholder enkle instruksjoner på hvordan man kan laste ned og kjøre programmet ved bare 2 museklikk, men vi vil også her gå gjennom instruksjonene. Det finnes to måter å kjøre programmet på, enten via en lett tilgjengelig kjørbar fil, eller via å importere kildekoden og kjøre programkjernen fra terminalen.

A.1 Nedlasting av kjørbar fil

1) Besøk vårt offentlige kodelager på `https://github.com/MetisPrometheus/Data-Bachelor-20H`

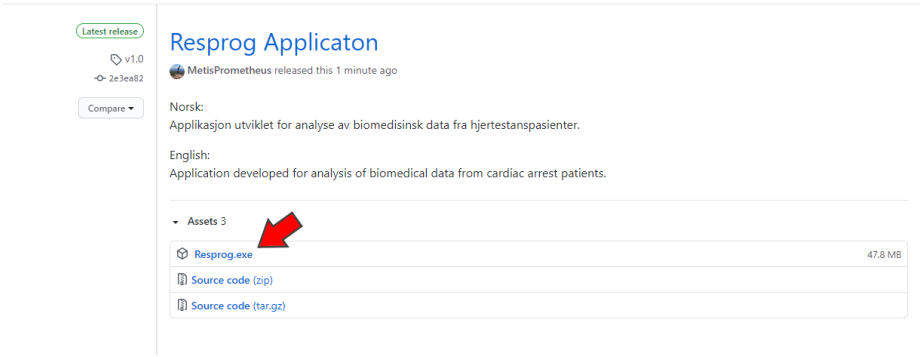
2) Trykk så på linken som viser alle utgitte versjoner av grensesnittet. Se Figur A.1

A.1 Nedlasting av kjørbar fil



Figur A.1: Link til en historikk av grensesnittets utgitte versjoner

3) Trykk til sist på Resprog.exe, se Figur A.2. Dette vil laste ned en kjørbar fil som du kan starte fra hvor som helst på datamaskinen, men for en bedre oversikt anbefales å ha sin egen mappe ettersom programmet vil skape en settings.txt fil i samme lokalisjon som programmet kjøres.



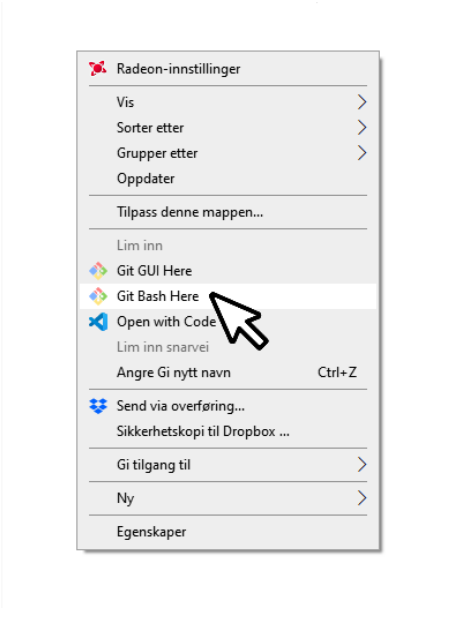
Figur A.2: Bildet for nedlasting til den første utgaven av grensesnittet

4) Når programmet startes vil brukeren bli spurt om å lokalisere to ulike mapper. Den første er mappen som inneholder alle de ulike signalene fra pasienter, og det antas at mappen heter "2 DATASETS". Den andre mappen heter "5 ANNOTATIONS"og inneholder selvinnlysende nok, alle annotasjonene. For den beste brukeropplevelsen bør mappene befinne seg lokalt på samme datamaskin som den kjørbare filen, ettersom dette vil føre til en mye hurtigere plotting av de ulike grafene.

A.2 Importering av kildekode

Dersom det skulle være noen spesielt interesserte i å importere kildekoden til gjennomgang eller manuell oppstart av programmet, vil dette delkapittelet kort forklare hvordan dette kan gjøres på ett enkelt vis.

1) Last ned Python versjon 3.8 eller nyere: `https://www.python.org/downloads/`
2) Last ned nyeste versjon av Git: `https://git-scm.com/downloads`
3) Høyreklikk i en tom mappe for å åpne en terminal, se Figur A.3



Figur A.3: Åpne git terminalen i filutforskeren

4) Skriv følgende kommando for å importere filer fra kodelageret til din lokale mappe:
git clone https://github.com/MetisPrometheus/Data-Bachelor-20H.git

5) Skriv følgende kommando for å laste ned alle python moduler nødvendig for å kunne kjøre programmet: ***pip install -r requirements.txt***

6) Skriv følgende kommando for å starte programmet:
python MainApp.py