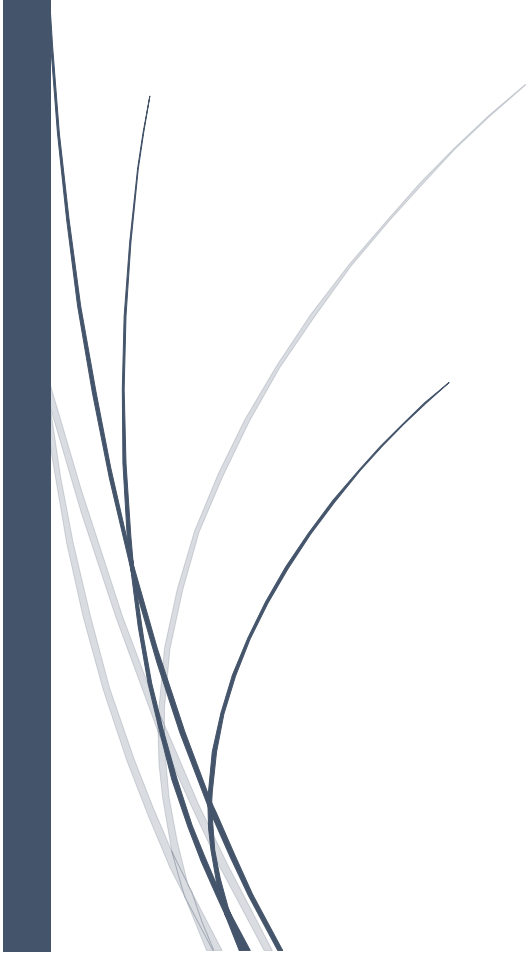
[Dato]

TMR4345 Marin Datalab

Dynamisk posisjonering



Innhold

1 Introduksjon

1.1 Bakgrunn

Denne rapporten har som formål å presentere arbeidet jeg har gjort i emnet «TMR4345 - Marin datalab», og resultatene av dette arbeidet. Emnet er designet for å styrke studentens evner til å bruke programmering effektivt senere i studiet. Dette gjelder spesielt i prosjektoppgave og masteroppgave. Introduksjon til bruk av Phidget API for input/output, referanse [1], har vært en spesielt nyttig erfaring som kan komme til nytte senere. I tillegg har bruk av C og bruk av pekere gitt en enda bredere kompetanse for programmeringsspråk. Problemer ved bruk av Mac OSX i bruk av Phidget API-et, har gjort det nødvendig å benytte en maskin som kjører Windows, og har gitt ny erfaring med bruk av PC vs. Mac i praksis. I praksis har PC visst seg mye enklere å arbeide med, når man må lete etter «paths», enn det Mac er. Til slutt i prosjektet presenteres resultatene ved plotting i «GNUPLOT» og 3D-visualisering i «GLview», som kan bli nyttige verktøy å kunne i fremtiden.

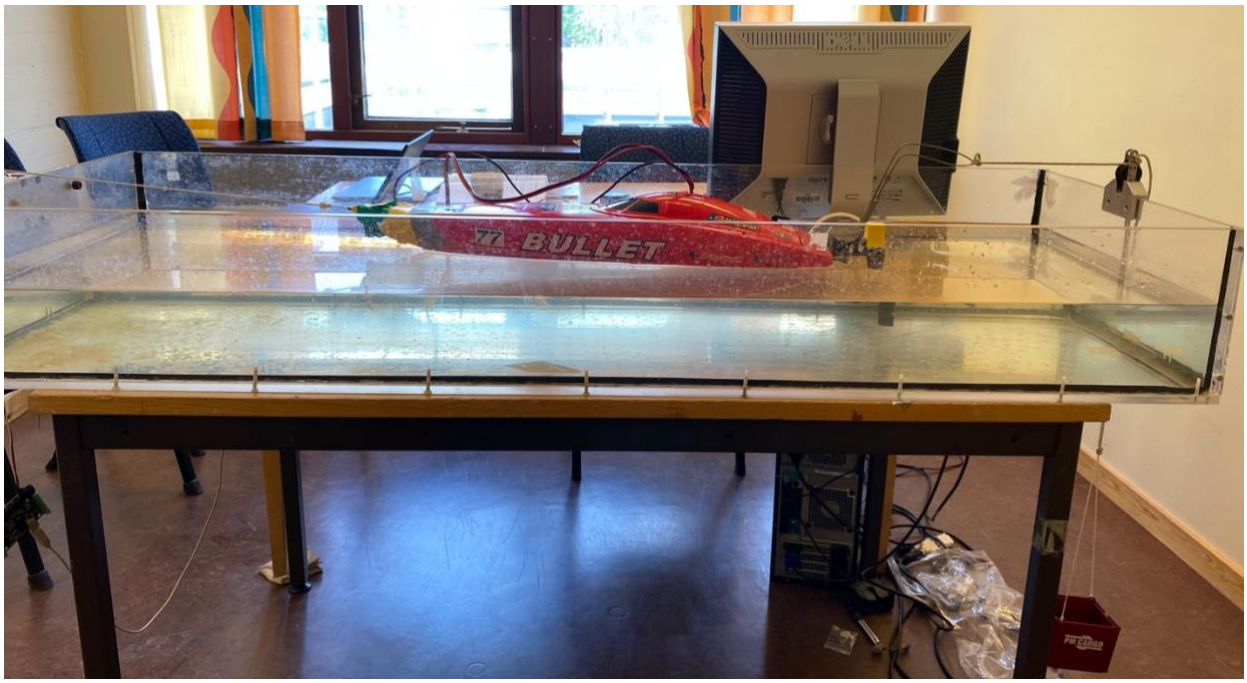
Det var et bredt spekter av oppgaver man kunne velge. Jeg valgte oppgaven hvor dynamisk posisjonering på en modellbåt skal implementeres. Denne oppgaven virket mest relevant for meg som IKT og Marin student med spesialisering innen marin kybernetikk. Prinsippene for dynamisk posisjonering er gitt i referanse [2]. I tillegg valgte jeg oppgaven p.g.a. muligheten til å se resultatet på båten i virkeligheten og for å få muligheten til å bruke teori fra reguleringsteknikk i praksis.

Denne rapporten fokusere på følgende deler:

1. **Teori** som ligger til bakgrunn for programmet
2. **Programvareutvikling**
3. **Resultatene** til programmet
4. **Konklusjon**

1.2 Oppgavebeskrivelse

Oppgaven går ut på å utvikle et enkelt dynamisk posisjoneringssystem i en frihetsgrad, som skal få en båt til å kjøre til en gitt posisjon og holde seg der ved kun bruk av motorkraft. Oppsettet består av et lite basseng, en båt som er festet i et tau som det henger vekter i for å simulere motstand, et potensiometer for å måle posisjonen i volt, og en regulerbar motor, se ***figur[1]***.



Figur[1]: Oppsett med båten, bassenget og vektmotstand (nederst til høyre)

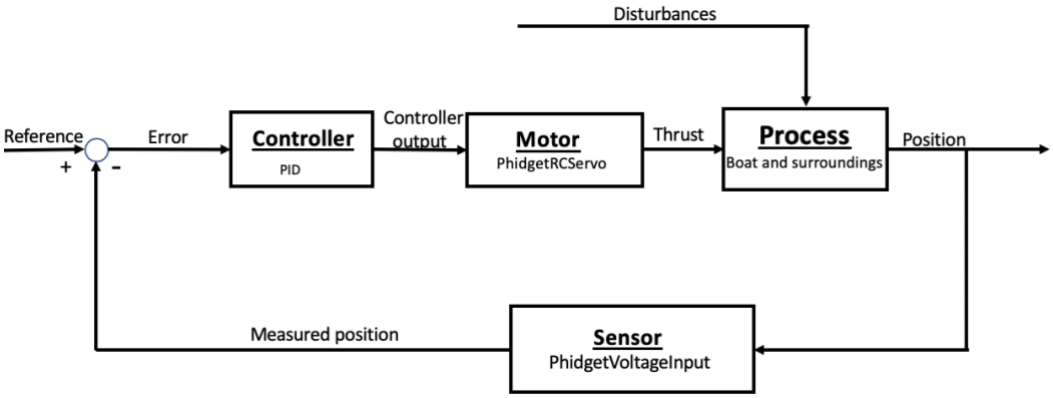
Potensiometeret skal leses av for å hente posisjonen til båten. Basert på posisjonen skal et C-program beregne pådraget som motoren skal utføre for å komme til ønsket posisjon. Deretter skal programmet sette pådraget i motoren. For å kunne lese data og sette pådrag kreves to essensielle hardwaredeler, et PhidgetInterfaceKit 8/8/8 og en PhidgetServo 1-Motor. Den første benyttes til å lese data fra potensiometeret og den siste brukes til å sette pådrag i motoren. En såkalt PID-regulatoralgoritme skal implementeres for så å benyttes til å beregne pådraget som skal settes.

Programmet skal også være i stand til å skrive data til fil, slik at resultatene kan plottes i «GNUPLOT» og visualiseres i «GLview». Det er viktig å kunne plotte dataene og visualisere de for å kunne vurdere ytelsen til programmet.

2 Teori

2.1 Modellbeskrivelse

For å være i stand til å kunne regulere båtens posisjon, er det behov for å kunne å beskrive systemet som båt, dens omgivelser og regulator utgjør. Et blokkdiagram av reguleringssystemet for DP er vist i ***figur[2]***:



Figur[2]: Blokkdiagram for reguleringssystemet i sin helhet.

Blokkdiagrammet beskriver et dynamisk system, som endrer seg ved tiden. **Prosessen** som skal reguleres er båtens jag-bevegelse i bassenget. Mange faktorer virker inn når denne prosessen skal reguleres. Veldig forenklet virker kun den konstante motstanden fra snoren og motstanden som kommer fra dynamisk bevegelse i jag (eng.: surge). Forenklet kan det fysiske systemets dynamiske likevekt beskrives ved

$$Treghetskrefter: \ F_I = (A_{11} + \ M)\eta_1$$
$$Dempingskrefter: F_D = B_{11}\eta_1$$
$$Stivhetskrefter: F_S = C_{11}\eta_1$$
$$Ytre\ krefter: P(t)$$

I surge er det ikke noe bidrag fra stivhetskrefter, derfor vil den dynamiske likevekten til systemet bli

$$F_I + F_D + F_S = P(t)$$
$$(A_{11} + \ M)\ddot{\eta}_1 + B_{11}\dot{\eta}_1 = P(t)$$

De ytre kreftene i den dynamiske likevektslikningen kommer i den virkelige verden fra bølgekrefter, viskøs trykkmotstand, friksjonsmotstand, luftmotstand osv. I bassenget kan vi se

bort i fra bølger og vind. I dette systemet vil derimot strømningsbildet rundt båten være veldig komplisert å beskrive p.g.a. størrelsen på bassenget og interaksjon med propellstrålen og veggene. Det vil også være forstyrrelse fra en vekt som er koblet via en trinse og snordrag på båten. Snodraget blir som en forstyrrelse fra konstant strøm. Prosessen (dynamikken) i dette reguleringssystemet bli sett på som en «black box», der det utelukkende vil bli regulert med hensyn på måling av båtens **posisjon**. Altså er bruk av foroverkopling i reguleringen av systemets prosess utelukket. Faktorene som forstyrrer båten kan deles inn i to kategorier, de vi kan regulere (f. eks. motorkraft) og de vi ikke kan regulere- forstyrrelsene. I dette tilfellet vil forstyrrelsene være alle krefter som virker på båten, utenom motorens pådrag. Siden prosessen i dette tilfellet blir sett på som en «black box», er det nødvendig med en **sensor** for å lese av posisjonsverdien i stedet for å beregne den. Gitt den målte posisjonen beregnes avviket fra referanseposisjonen, som deretter blir sendt som input til regulatoren. **Regulatoren** har som oppgave å beregne hvor stort pådraget til motoren skal være. Regulatoren sender deretter denne verdien til **motoren** som setter pådraget og påvirker prosessen, slik at båten endrer posisjon (eller holder seg der den er).

2.2 Regulator

Regulatoren er den delen av systemet som beregner pådraget. Dette er en algoritme som implementeres i datamaskinen. Regulatorens ytelse bestemmes av regulatorens evne til å:

1. Motvirke uønskede forstyrrelser
2. Fungere ved flere referansepunkter
3. Ikke være sensitiv for endring i parameterne i prosessen
4. Stabilisere en i utgangspunktet ustabil prosess

Systemet som skal reguleres er en lukket sløyfe system med tilbakekobling. Dette betyr prosessens output måles og brukes til å beregne hvor mye systemets input skal manipuleres. I dette tilfellet leses av kun en output, posisjon, og kun en input, motorkraft, manipuleres. Altså er dette den enkleste formen for reguleringssystem, single-input/single-output (SISO).

2.2.1 PID-regulering

Regulatoren implementerer en standardregulator, PID-regulator, som benyttes mye i reguleringsteknikken, referanse [3]. PID-regulatoren er en standard tilbakekoblingsalgoritme som tar inn en referanseverdi man ønsker å oppnå og den faktiske verdien, og beregner hvor mye

pådrag som skal settes basert på disse to verdiene. PID-regulator er en forkortelse for proporsjonalitet- , integral- og derivatregulator. Regulatoren bruker avviket fra referansen,

$$e(t) = r(t) - y(t),$$

i beregningen av pådraget. Dette kommer også frem i ***figur[2]***. Feilen beregnes fra $r(t)$, som er ønsket verdi og $\mathrm{y}(t)$, som er nåverdien. Pådraget beregnes deretter ved følgende formel:

$$u(t) = K_p \cdot e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{d}{dt} e(t)$$

Virkningen av hvert ledd kan justeres ved å endre på verdiene til parameterne, $K_p, K_i \; og \; K_d$. Senere i rapporten kommer det fram at størrelsene for disse parameterne har mye å si for systemets respons. Vi kaller dette for tuning av regulator.

2.2.2 P-regulering

P-regulering er en veldig enkel form for regulering. Det går ut på at pådraget blir beregnet til å være proporsjonalt med avviket fra referansen. Vi kan se på P-regulering som en proporsjonalfjær. Dette vil si at dersom avviket er stort, blir pådraget/kraft stor, og dersom avviket er lite blir pådraget lite. K_p er proporsjonalitetsparameteren. Problemet er at dersom K_p blir for stor, vil responsen bli for stor, og responsen vil svinge med store utslag rundt referanseverdien. Med godt valg av K_p- verdi, vil responsen nærme seg referanseverdien i en kontrollert hastighet og ikke svinge for mye rundt den. I dette tilfellet er ikke en P-regulator tilstrekkelig, fordi at båten får et stasjonært avvik på grunn av den statiske lasten fra vektene som henger i snoren. I praksis betyr dette at når båten kommer til ønsket posisjon, vil pådraget bli null, men det virker en konstant statisk last på båten som drar den tilbake. Derfor vil båten legge seg i ro ved et punkt før referansepunktet der motorkraften er like stor som den konstante statiske kraften. Selv om at P-regulering i seg selv ikke er godt nok i dette tilfellet til å kompensere den statiske lasten fra loddet/snordraget, er P-leddet det dynamiske leddet som bør være størst og dominere i reguleringen.

2.2.3 I-regulering

Det andre leddet i PID-regulatoren er integralleddet. Integralleddet integrerer opp feilen fra programmet starter og til nåtidspunktet. Integralet er altså akkumulert avvik. Integralets viktigste funksjon er å eliminere stasjonært avvik. Som nevnt under P-regulering, er det en del stasjonært

avvik i prosessen i dette tilfellet pga. loddet, altså er det strengt nødvendig med et I-ledd i regulatoren. Dersom det er stasjonært avvik, vil integralet gradvis vokse over tid helt til det stasjonære avviket blir eliminert ved at pådraget vokser. En utfordring med I-leddet er å sørge for at verdien blir stor nok, da vil responsen fra I-ledet bli treg. På den andre siden, dersom I-leddet blir for stort, vil det responsen bli for stor og begynne å oscillere rundt referansepunktet. Derfor velges ofte en noe lavere verdi for I-forstrekningen K_i.

2.2.4 D-regulering

Derivatleddet er feilen i hastighet – dvs. posisjon derivert med hensyn på tiden. Derivat fungerer som en demper. Hensikten med derivatleddet er å forutse når feilen vokser raskt. Når feilen vokser raskt, vil derivatleddet respondere med å vokse raskt selv, og når feilen går raskt mot null, vil derivatleddet bli negativt og bremse responsen. Ved raske små endringer i feilen og ved målestøy, blir derivatleddet tilnærmet ubrukelig. Da kan det være nødvendig å sende inputen (målingen) gjennom et Butterworth filter, eller lavpassfilter, som filtrerer bort høye frekvenser.

2.3 Diskretisering

Prosessen vi skal regulere er en kontinuerlig prosess, men posisjonen kan ikke måles kontinuerlig, og derfor blir resultatet av målingene en mengde med diskrete data som samples av datamaskinen. For at PID-algoritmen skal være i stand til å håndtere disse dataene, må den tilpasses å fungere for diskrete data. Dette betyr at integralet må byttes ut med en sum

$$\int_0^{t_n} e(\tau)d\tau \approx \sum_{n=0}^{N} e(t_n)\Delta t,$$

der $e(t_n)$ er feilen ved tidspunkt t_n og Δt er tiden siden forrige måling, og derivasjonsleddet må byttes ut med en lineær approksimasjon

$$\frac{d}{dt}e(t) \approx \frac{e(t_i)-e(t_{i-1})}{\Delta}$$

Både uttrykket for integralet og derivasjonsleddet er en numerisk tilnærming, og ikke gir ikke det eksakte svaret. Ved bruk av tilbakekopling er det viktigste at reguleringen gjør at responsen beveger seg i riktig retning og at samplingstiden er rask i forhold til båtens dynamikk.

3 Programvareutvikling

I denne delen av rapporten vil arbeidsmetodikken i utviklingsprosessen og programmets struktur presenteres.

3.1 Utviklingsprosessen

I utviklingen av programmet har prinsipper hentet fra fagområdet programvareutvikling, referanse [4] blitt anvendt for å oppnå en så smidig utviklingsprosess som mulig. Smidig, eller agile, utvikling er et mye brukt uttrykk i programvareutvikling, og brukes for å beskrive effektiv og enkel utvikling. Vanligvis gjelder prinsippene i programvareutvikling for teamarbeid, men det er enkelte prinsipper som også kan anvendes alene i et prosjekt som dette. De to viktigste prinsippene for agile utvikling, som har blitt anvendt i dette prosjektet, er iterativ utvikling og testing.

Iterativ utvikling går generelt ut på å utvikle programmet i iterasjoner. De første iterasjonene handler om å få til et enkelt kjørende program. For hver iterasjon forbedres programmet og funksjonaliteter legges til, slik at det blir mer og mer brukervennlig og sofistikert. Siden fokuset har vært på iterativ utvikling, ble det ikke laget noen skisse av programmet i oppstarten, med utgangspunkt i at ny funksjonalitet ville bli lagt til underveis ved behov. I dette prosjektet har programmet vært gjennom følgende iterativ utvikling:

1. Main-metode som skriver ut «Hello world!», for å teste at C kjører som det skal i Visual Studio Code.
2. Main-metode som kobler seg til Phidget-enhetene, leser av verdien til potensiometeret og setter en konstant verdi for propellthrust.
3. Main-metode der verdi for propellthrust kan settes, og motoren kan skrus av og på.
4. Main-metode med enkel PID-algoritme implementert
5. I denne iterasjonen ble flere deler av main-metoden ved forrige iterasjon, delt opp i hjelpefunksjoner, slik at koden ble mer oversiktlig. Hjelpefunksjonene som ble opprettet var «get_error», for å finne feilen, «regulator», for å beregne ny thrustverdi og «set_thrust», for å sette den nye thrustverdien.
6. I denne iterasjonen ble følgende funksjoner lagt til; «manage_time», som brukes for å ha kontroll på tiden som skal skrives til fil, «write_result», som skriver resultatene til fil, «write_vtf()», som skriver om .dat-filen til .vtf-fil for bruk i GLView.
7. Denne iterasjonen ble brukt for å gjøre programmet mer brukervennlig. I stedet for at parameterverdiene blir satt i main-metoden, blir bruker bedt om å sette parameterverdiene i terminalen.

8. I den siste iterasjonen ble funksjonen «setReference» lagt til for å gjøre det enklere å sette referanseverdi. Et problem med oppsettet var at posisjonen ble lest forskjellig ut ifra hvordan potensiometeret var posisjonert når båten ble lagt opp i bassenget. Dette problemet ble løst ved å sette origo og maksimalverdi for posisjon før kjøring av algoritmen, og referanseverdien blir satt til å være en prosentandel av differansen mellom de to posisjonene. Denne funksjonen ble lagd brukervennlig ved at bruker kan presse hvilken som helst tast for å markere at båten er i origo eller maksimal posisjon.

Dersom flere iterasjoner skulle blitt gjennomført, kunne det vært aktuelt å dele opp enda en del av main-metoden i flere hjelpefunksjoner, for å få et mer oversiktlig program. Følgende funksjoner hadde vært aktuelle å legge til, «Setup()», der Phidghet-oppsettet gjøres klart, «get_user_input()», for å hente brukerinput, «close_phidgets()», for å koble fra Phidget-enhetene. Samtidig er ikke programmet veldig langt og komplisert, og det er i utgangspunktet oversiktlig nok til å jobbe med. Derfor har det vært et bevisst valg å ikke lage flere hjelpefunksjoner. Derimot hadde det vært aktuelt å gjøre dette dersom flere og mer kompliserte funksjonaliteter skulle blitt lagt til.

I utviklingsprosessen ble den enkleste formen for testing benyttet. Hver gang en ny linje kode som kunne føre til krasj ble lagt til, ble en linje med «printf(«Beskrivelse av linjen over»)» lagt til. Ved bruk av denne type testing er det veldig lett å finne ut hvor koden krasjer, siden man kan lese av konsollen hvilken «printf()», som ikke blir skrevet ut. I tillegg ble koden testet på det fysiske systemet for å se om den fungerte i praksis.

I tillegg til å ha fokus på smidig utvikling, har det i dette prosjektet også blitt lagt stor vekt på å dokumentere koden. Generelt er dokumentasjon av kode viktig for vedlikehold, kommunikasjon innad i prosjektgruppe, læring og deling av kunnskap. I utgangspunktet skal koden bygges struktureres og utvikles på en slik måte at den er selvforklarende, slik at dokumentasjon er unødvendig, men samtidig er det bedre å dokumentere for mye, enn for lite. Derfor kan det virke som deler av koden i dette prosjektet er overdokumentert, men i dette tilfellet ble det besluttet at det er best å være på den sikre siden.

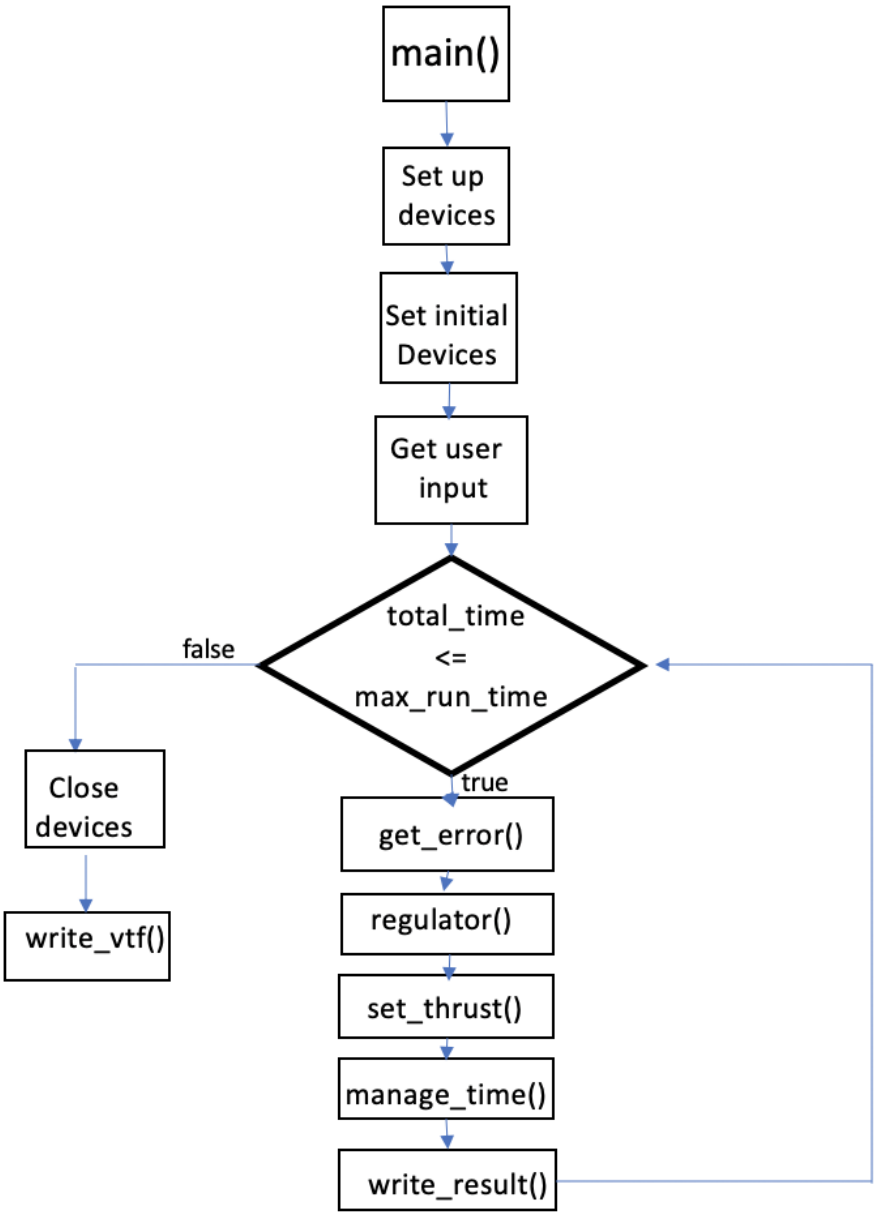
3.2 Programmets struktur

Hele programmet er skrevet i en og samme fil, slik at alle hjelpefunksjonene ligger i samme fil som main-metoden. Dette gjorde at det ikke var nødvendig å lage en egen header-fil for å

inkludere alle kodebibliotekene som ble benyttet. I stedet kunne alle kodebibliotekene inkluderes i toppen av «main.c» fila. Siden programmet er så kort, 343 linjer, ble det ikke ansett som nødvendig å fordele koden utover flere filer, siden det er tilstrekkelig oversiktlig i en fil.

Programmet kjøres i main-metoden, og følgende skjer i følgende rekkefølge, se ***Figur[3]*** for flytdiagram:

1. Phidget-enhetene klargjøres for bruk:
    - «Handles» for motoren og potensiometeret deklareres og lages.
    - Serienummeret til enhetene må settes for at Phidget-API-et skal koble seg opp mot enhetene.
    - Hvilken kanal som skal brukes må settes.
    - Koblingen til enhetene åpnes.
2. Gamle resultat-filer slettes.
3. Verdier som skal brukes i programmet deklareres og initialiseres.
4. Referanseverdi, PID-parameterne og kjøretid settes av bruker.
5. While-løkka kjøres for bestemt kjøretid:
    - Starttid settes
    - Feil beregnes ved «get_error()»
    - Regulatoren, «regulator()», beregner ny thrust basert på feilen og parameterverdiene satt av bruker
    - Den nye thrusten settes i «set_thrust()»
    - Tiden oppdateres i «manage_time()»
    - Resultatet skrives til .dat-fil i «write_result()»
6. Alle Phidget-kanalene lukkes og slettes.
7. Resultatfilen skrives til .vtf-fil.



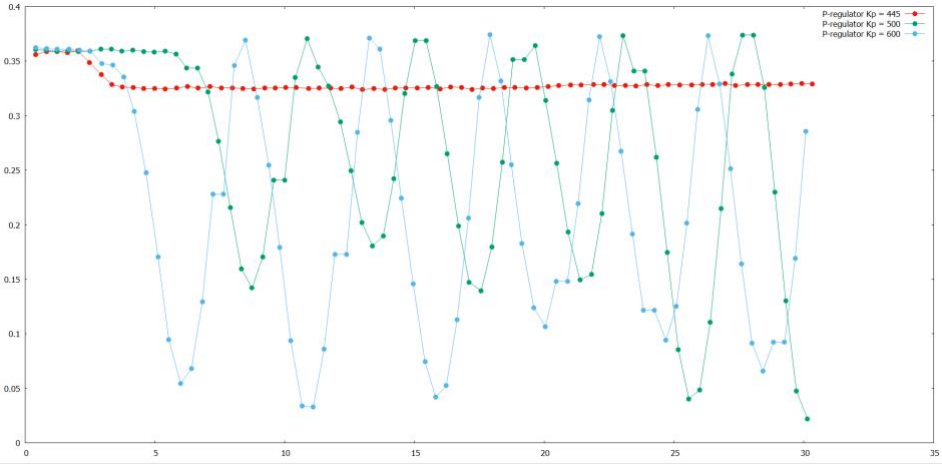
Figur[3]: flytdiagram for hovedprogrammet

4 Resultater

Under følger visualisering av programmet. Målet er satt til å være 50% av distansen mellom origo og maksimal posisjon. I plottene er kun feilen visualisert, siden referansen varierte fra gang til gang, og formålet med programmet er i utgangspunktet å oppnå null feil.

4.1 P-regulator

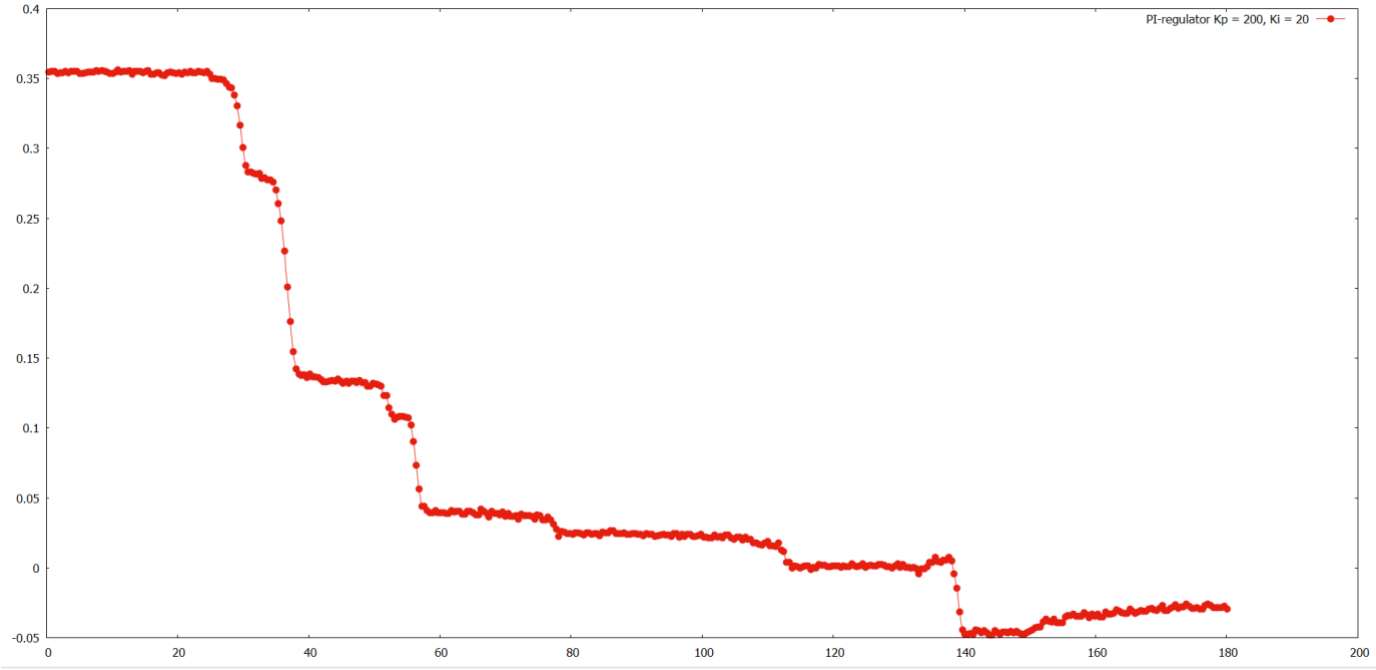
I ***Figur[4]*** følger resultatene til tre forsøk på å tune p-regulatoren. Av plottet kan det leses av at P-regulatoren klarte ikke å stabilisere seg på null feil ved noen av forsøkene. Ved $K_P = 445$ stabiliserer båten seg på rundt 0.35 i feil, mens ved $K_P = 500\ og\ K_P = 600$ kjører båten opp mot null feil, og blir dratt tilbake med engang. Hvorfor dette skjer vil bli diskutert i neste kapittel, diskusjon.



Figur[4]: Plott av feilen ved bruk av tre forskjellige Kp-verdier i P-regulatoren

4.2 PI-regulator

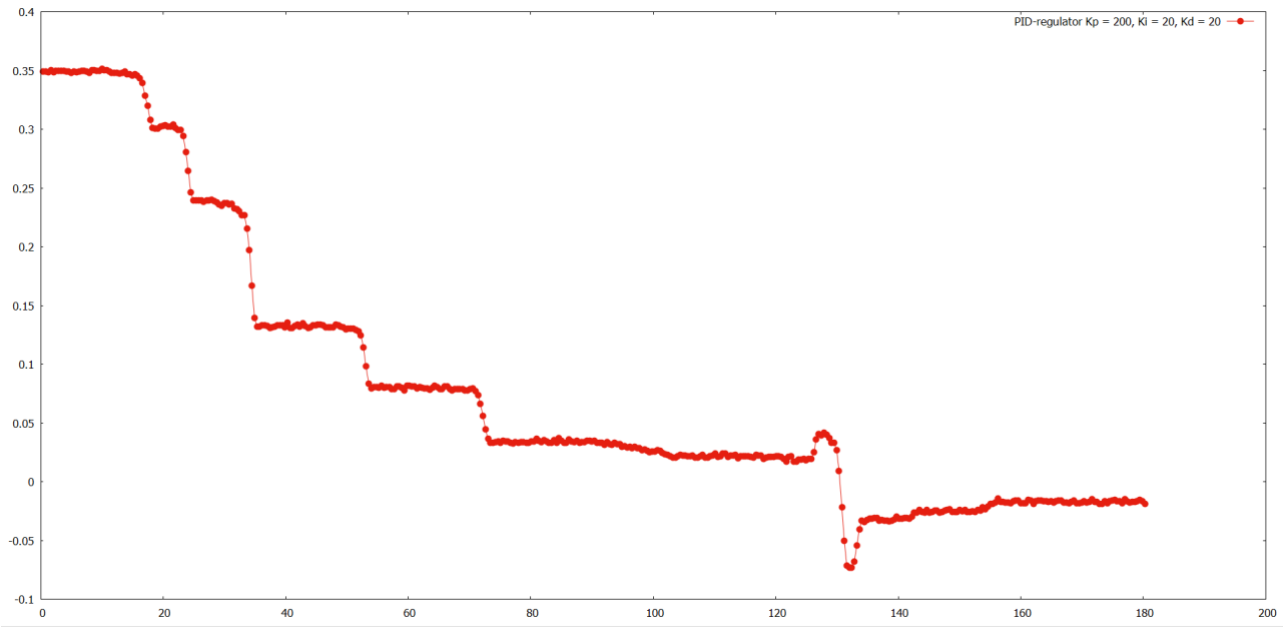
Under følger resultatene, ***Figur[5],*** for PI-regulatoren, for $K_P = 200\ og\ K_i = 20$. PI-regulatoren stabiliserer seg rundt null feil etter 110 sek, og holder seg i området rundt null feil for resten av kjøretiden. Altså er PI-regulatoren i stand til å oppnå tilnærmet null feil og holde seg der, mens P-regulatoren ikke er i stand til det.



Figur[5]: PI-regulatoren optimalt tunet

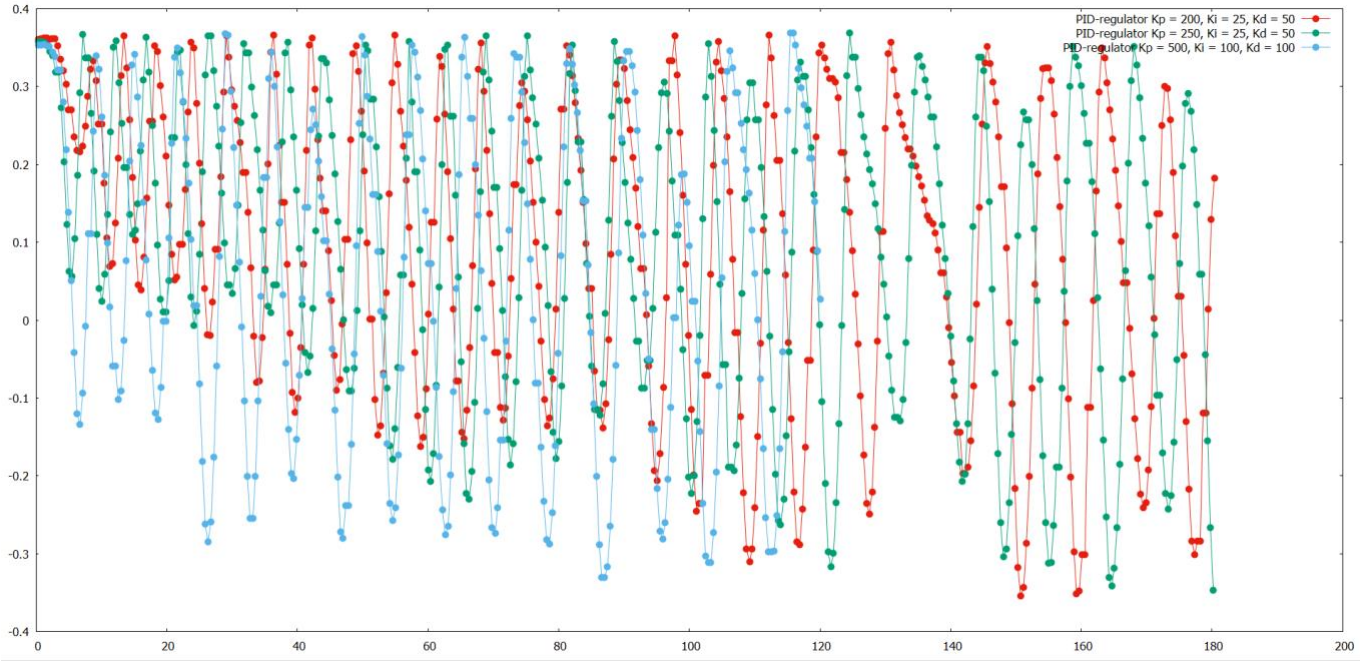
4.3 PID-regulator

Under er resultatene fra PID-regulator med $K_P = 200, K_i = 20\ og\ K_d = 20$ plottet i ***Figur[6]***. Ved å studere plottet til PI-regulatoren for $K_P = 200\ og\ K_i = 20$, er det tydelig at ytelsen til disse to regulatorene er veldig like. Altså ser det ut som at PID-regulatoren oppfører seg som en PI-regulator.



Figur[6]: PID-regulator, optimalt tunet

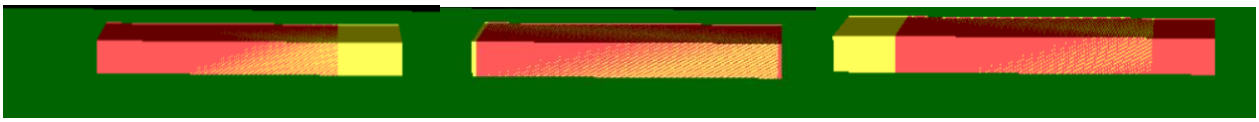
Figur[7] viser tre andre tilfeller der PID-regulatoren en tunet annerledes. Bildet er en ganske kaotisk framstilling av dataene, men det viktigste å ta ut ifra plottet er at i ingen av tilfellene stabiliserer båten seg rundt referansepunktet. Derimot oscillerer båten med store utslag rundt referansepunktet. Illustrasjonen er tatt med for å gi et visuelt inntrykk av den eksperimentelle fremgangsmåten for å komme fram til optimale parameterverdier, som ***Figur[6]*** illustrerer.



Figur[7]: PID-regulator, dårlig tunet

4.4 Animasjon

Figur[8] viser bilder av animasjon i GLview, se vedlagt fil for selve animasjonen. Referanseverdien er det røde området. Altså svinger båten i denne visualiseringen rundt referansepunktet.



Figur[8]: visualisering av båten i GLview

4.5 Regulatorparametere

Tuning av PID-regulator er gjort gjennom uttesting i bassenget og inspirert av tunemetoder fra reguleringsteknikk, referanse [3]. I Tabell [1] er oversikt over valgte forsterkninger som fungerte gitt i ***Figur[4,5]***.

Parameter	PI	PID
K_P	200	200
K_i	20	20
K_d	0	20

Tabell[1]: Verdiene bruk for de ulike regulatorparameterne, der programmet utførte sin hensikt

5 Diskusjon

5.1 Resultater

I ***Figur[3]*** kommer det tydelig fram at P-regulator ikke er tilstrekkelig i dette tilfellet, noe som samsvarer med diskusjonen under teori. Ved lavt/moderat pådrag stabiliserer båten seg på en feil på 0.35. Grunnen til at feilen stabiliserer seg rundt denne verdien og ikke referansepunktet, er at det virker en konstant statisk kraft fra snoren som henger bak på båten. For at båten skal ligge i ro, må motorkraften være like stor som den statiske kraften. Dvs. at når båten kommer til referansepunktet og pådraget fra motoren blir null, siden feilen er null, vil motoren skru seg av, og båten vil bli dratt bakover av den statiske kraften til motoren slår seg på igjen. Altså er det i dette tilfellet ikke fysisk mulig for en P-regulator å gi ønsket resultat, noe som allerede er diskutert under teori.

Derimot ved å studere plottet til PI-regulatoren, se ***Figur[4],*** kan man se at båten klarer å nærme seg feil på null, og stabilisere seg der. Dette kommer av at integralleddet bygger seg opp til å eliminere det stasjonære avviket som kommer av den statiske kraften, og båten stabiliserer seg på en tilnærmet konstant motorkraft i referansepunktet. Selv om at båten ved bruk av PI-regulatoren klarer å stabilisere seg rundt referansepunktet, bruker den relativt lang tid før den kommer dit. Altså kunne det vært en ide å teste større verdi på regulatorparameterne

Ved sammenligning er det liten forskjell på oppførselen til PI-regulatoren og PID-regulatoren, se henholdsvis ***Figur[4]*** og ***Figur[5]***. Dette tyder på at derivatleddet til regulatoren har et svært lite bidrag. Det kan være flere grunner til dette. Den ene grunnen kan være at siden båten beveger seg relativt tregt mot referansepunktet, slår ikke derivatleddet inn, siden hastighetsvariasjonene ikke er veldig stor. Den andre grunnen kan være hakkete målinger som gir veldig små svingninger. Dette kan evt. fikses ved å sende feilen gjennom et Butterworthfilter, lavpassfilter, som glatter ut kurven.

5.2 Tuning

I Tuning av PID-regulatorer kan man bruke en metode, som kalles Ziegler-Nichols. I dette tilfellet var det dessverre ikke mulig å bruke denne metoden. P-regulatoren ikke hadde tilfredsstillende ytelse, og derfor ville det bli vanskelig å si hvilken verdi av K_P, som ville bringe P-regulatoren til stabilitetsgrensa. Dette skyldes nok også at motoren hadde en nedre og øvre

begrensning for motorkraft. Derfor ble eksperimentell tuning av regulatoren benyttet. Altså ble parameterverdiene bestemt etter en del prøving og feiling, se ***Figur[6]*** for et godt eksempel på feiling og ***Figur[5]*** for et eksempel på at ting fungerer. Det er nok en god mulighet for at regulatoren kunne blitt tunet enda bedre, om det hadde blitt brukt mer tid på å teste ulike parameterverdier, men siden enkelte av verdiene ga ganske tilfredsstillende resultater, ble ikke dette gjort.

5.3 Forbedringer av programmet

Som forklart under programvareutvikling, kapittel 3.1, kunne flere iterasjoner i utviklingsprosessen blitt gjennomført om det var mer tid, der en større del av main-metoden kunne blitt fordelt utover på flere hjelpefunksjoner, se avsnitt 3.1 for mer forklaring. Når det kommer til selve ytelsen av algoritmen, kunne et lavpassfilter blitt lagt til, som diskutert under diskusjonen av derivatleddets virkning. I tillegg kunne plott av resultatene blitt gjort direkte i Visual Studio Code, i stedet for i GNUPLOT, slik at en evt. bruker uten kjennskap til GNUPLOT, kunne fått en illustrasjon av responsen til regulatoren uten å måtte sette seg inn i GNUPLOT.

5.4 Annet

I testingen ble det observert at det var litt slark i motoren til båten, og en viss treghet i endring av thrust. Dette er en ulinearitet som ikke er uvanlig i pådragsorganer. Dette kan ha påvirket resultatene noe, men har nok ikke vært av veldig stor betydning, da programmet ser ut til å fungere tilnærmet som forventet på forhånd.

6 Konklusjon

Dette prosjektet har vært en nyttig erfaring. Det har vært en del utfordringer rundt å få det tekniske til å fungere. Spesielt var det en del utfordringer med å få Visual Studio Code til å finne de ulike kodebibliotekene som ble brukt underveis, som virkelig har satt tålmodigheten på prøve, men også gitt vesentlig mer erfaring med det praktiske rundt IKT. I tillegg har det vært en god erfaring å kunne se at koden man skriver, faktisk kan få ting til å skje i virkeligheten. Derfor passet oppgaven om dynamisk posisjonering veldig bra, siden det var mulig å teste koden på båten i praksis. Å skrive prosjektet i C har både vært en utfordring og en nyttig erfaring. Som student ved I og IKT har jeg blitt eksponert for en del forskjellige programmeringsspråk, men har aldri jobbet med pekere. Derfor har C-programmering vært spesielt nyttig for meg, siden det kunne bidra til å utvide min kompetanse om programmeringsspråk.

Oppgaven visste seg å være en gjennomførbar oppgave. Programmet fungerer stort sett som det skal. Programmet klarer å styre båten til en satt posisjon, og holde seg tilnærmet i ro ved den posisjonen. Det produserer også resultatfiler som kan benyttes i visualisering av båten. Dermed kan det konkluderes med at oppgaven er utført, selv om at det kunne blitt utført noen forbedringer når det kommer til ytelse, brukervennlighet og vedlikehold.

7 Referanseliste

[1] Phidgets user manuals, https://www.phidgets.com/?tier=3&prodid=0

[2] Holm, H., A. J. Sørensen. Manøvrering, styring og regulering. Kap. 8. i Lundby, L. Havromsteknologier, 2015, NTNU Institutt for marin teknikk, Fagbokforlaget Vigmostad & Bjørke

[3] Andresen, T., J.G. Balchen, B. Foss. Reguleringsteknikk. 2016, NTNU Institutt for teknisk kybernetikk

[4] Andresen, T., J.G. Balchen, B. Foss. Reguleringsteknikk. 2016, NTNU Institutt for teknisk kybernetikk

Appendix

```c
#include <phidget22.h>
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <time.h>
#include <math.h>
#include <string.h>
#include <unistd.h>


/*
@function get_error()
@param PhidgetVoltageInputHandle: position channel
@param int: reference
@return double: error
Calculates the error based on reference value and measured position
*/
double get_error(PhidgetVoltageInputHandle position_channel, double reference){

    double current_pos;
    double error;

    // Get current measured position
    PhidgetVoltageInput_getVoltage(position_channel, &current_pos);
    // Calculate the error
    error = -((double)reference - (double)current_pos);

    printf("Error: %lf\n", error);

    return error;
}


/*
@function regulator()
@param double: Kp --- Gain
@param double: Ti --- Integrator parameter
@param double: Td --- Derivative parameter
@param double: error --- deviance from reference position
@param double: dt --- timestep
@param double: *integral_pointer --- Value of the integral
@param double: *pre_error_pointer --- previous error value
@return double: controller --- controller output
The function calculates the new thrust, but if the controller output exceeds the limits of the motor,
the thrust is set to the limits.
*/
double regulator(double Kp, double Ki, double Kd, double error, double dt,  double *integral_pointer, double *pre_error_pointer){
    double derivative;
    double thrust;
    double integral = *integral_pointer;
    double pre_error = *pre_error_pointer;
    // Calculate the integral term
```

```
    integral = integral + error*dt;
    printf("integral: %lf \n dt = %f\n", integral, dt);
    // Calculate the derivative term
    derivative = (error - pre_error)/dt;
    printf("Derivative: %lf \n", derivative);
    // Calculate new thrust as controller output
    thrust = Kp*error + Ki*integral + Kd*derivative;
    printf("Controller output = %f \n", thrust);
    // Store the error for caluculation of the derivative in the next iteration
    pre_error = error;
    *integral_pointer = integral;

    return thrust;
}

/*
@function set_thrust
@param PhidgetRCServoHandle: rcServo --- Servo channel handle
@param double: thrust --- Calculated new thrust value from the controller
@param double: dt --- timestep
The function sets the new thrust and engages, and lets the thrust be activated
for the chosen time.
*/
void set_thrust(PhidgetRCServoHandle rcServo, double thrust, double dt){

    // The timestep has to be in micro seconds
    int sleep = 1000000*dt;

    // Limits the thrust(based on the note in the lab)
    if(thrust > 169){
        thrust = 169;
        printf("Controller output exceeded the limits and is set to %f \n", thrust);
    }
    else if(thrust < 110){
        thrust = 110;
        printf("Controller output exceeded the limits and is set to %f \n", thrust);
    }

    PhidgetRCServo_setTargetPosition(rcServo, thrust);
    PhidgetRCServo_setEngaged(rcServo, 1);

    usleep(sleep);
}


/*
@function
@param double: *elapsed_time_pointer --- The pointer to the elapsed time variable
@param double: *total_time_pointer --- The pointer to the total time variable
@param double: start_time --- the time of when the program starts running
@param double: sleeptime --- seconds slept while activating the phidget devices
The function updates the total time, for use in analysis
*/
```

```
106 void manage_time(double *elapsed_time_pointer, double *total_time_pointer, double
    start_time, double sleeptime){
107
108     *elapsed_time_pointer = ((double) (clock()- start_time)/CLOCKS_PER_SEC) +
    sleeptime;
109     *total_time_pointer = *total_time_pointer + *elapsed_time_pointer;
110
111     printf("Elapsed time: %f \n", *elapsed_time_pointer);
112     printf("Total time: %f \n", *total_time_pointer);
113 }
114
115
116 /*
117 @function
118 @param double: total_time --- Total run time
119 @param double: error --- deviance from reference position
120 @param PhidgetVoltageInputHandle: position --- voltage input handle for measuring of
    position
121 Writes time, position and error to the  results.dat-file
122 */
123 void write_result(double total_time, double error, PhidgetVoltageInputHandle
    position){
124     double current_position;
125     PhidgetVoltageInput_getVoltage(position, &current_position);
126     FILE *f = fopen("results.dat", "a");
127     fprintf(f, "%lf %lf %lf \n", total_time, error, current_position);
128     fclose(f);
129 }
130
131
132 /*
133 @function
134 writes the input file to GLview in window format by reading from the result file
    results.dat
135 */
136 void write_vtf(){
137
138     //Open files
139     FILE *f = fopen("dynpos.vtf", "a");
140     FILE *fr = fopen("result.dat", "r");
141     int i=1, j, c, error, new_pos;
142     double t;
143
144
145
146     //Set up vessel
147     fprintf(f,"*VTF-1.00\r\n\r\n\r\n");
148     fprintf(f,"!vessel:\r\n");
149     fprintf(f,"*NODES 1\r\n");
150     fprintf(f,"0. 12. 0.\r\n");
151     fprintf(f,"150. 12. 0.\r\n");
152     fprintf(f,"150. -12. 0.\r\n");
153     fprintf(f,"0. -12. 0.\r\n");
154     fprintf(f,"0. 12. 12.\r\n");
```

```
155     fprintf(f,"150. 12. 12.\r\n");
156     fprintf(f,"150. -12. 12.\r\n");
157     fprintf(f,"0. -12. 12.\r\n\r\n\r\n");
158 
159     fprintf(f,"!pool:\r\n");
160     fprintf(f,"*NODES 2\r\n");
161     fprintf(f,"0. 50. 0.\r\n");
162     fprintf(f,"350. 50. 0.\r\n");
163     fprintf(f,"350. -50. 0.\r\n");
164     fprintf(f,"0. -50. 0.\r\n\r\n\r\n");
165 
166     fprintf(f,"*ELEMENTS 1\r\n");
167     fprintf(f,"%%NODES #1\r\n");
168     fprintf(f,"%%HEXAHEDRONS\r\n");
169     fprintf(f,"1 2 3 4 5 6 7 8\r\n\r\n\r\n");
170 
171     fprintf(f,"*ELEMENTS 2\r\n");
172     fprintf(f,"%%NODES #2\r\n");
173     fprintf(f,"%%QUADS\r\n");
174     fprintf(f,"1 2 3 4\r\n\r\n\r\n");
175 
176     fprintf(f,"*GLVIEWGEOMETRY 1\r\n");
177     fprintf(f,"%%ELEMENTS\r\n");
178     fprintf(f,"1, 2\r\n\r\n\r\n");
179 
180     while((c = fgetc(fr)) != EOF) {
181         //Read new pos
182         fscanf(fr,"%lf %lf %lf", &t, &error, &new_pos);
183 
184         //Write to vtf file
185         fprintf(f,"*RESULTS %lf\r\n",i);
186 
187         //Sjekk formateringen endret fra d til lf
188         fprintf(f,"%%DIMENSION 3\r\n");
189         fprintf(f,"%%PER NODE #1\r\n");
190         fprintf(f,"%lf 0 0\r\n",new_pos);
191         fprintf(f,"%lf 0 0\r\n",new_pos);
192         fprintf(f,"%lf 0 0\r\n",new_pos);
193         fprintf(f,"%lf 0 0\r\n",new_pos);
194         fprintf(f,"%lf 0 0\r\n",new_pos);
195         fprintf(f,"%lf 0 0\r\n",new_pos);
196         fprintf(f,"%lf 0 0\r\n",new_pos);
197         fprintf(f,"%lf 0 0\r\n\r\n\r\n",new_pos);
198 
199         ++i;
200         }
201 
202     fprintf(f,"*GLVIEWVECTOR 1\r\n");
203     fprintf(f,"%%NAME DISPLACEMENT\r\n");
204 
205     for(j=1;j<i;++j){
206         fprintf(f,"%%STEP %d\r\n",j);
207         fprintf(f,"%d\r\n",j);
208         }
```

```
209
210
211     fclose(f);
212     fclose(fr);
213     }
214
215 /*
216 @function
217 @param PhidgetVoltageInputHandle: pos --- The voltage input handle for position
218 @param double: percentage --- The percentage of the differanse between the origin and
    the maximum position, we want
219                               the boat to travel.
220 @return double: reference --- The reference value calculated
221 The function setReference calculates the reference position based on maximum value
    for position, the origin position
222 and a percentage of the differanse between them.
223 */
224 double setReference(PhidgetVoltageInputHandle pos, double percentage){
225     // Set origin position
226     double origin_pos;
227     printf("\nWhen in origin position: ");
228     system("pause");
229     PhidgetVoltageInput_getVoltage(pos, &origin_pos);
230     printf("%lf", origin_pos);
231     // Set maximum position
232     double max_pos;
233     printf("\nWhen in maximum position: ");
234     system("pause");
235     PhidgetVoltageInput_getVoltage(pos, &max_pos);
236     printf("%lf", max_pos);
237     // Calculates reference position
238     double reference = origin_pos + percentage*(max_pos - origin_pos);
239     return reference;
240 }
241
242 /*
243 @function - main()
244 The main function sets intitial value, the channels needed for I/O, runs the program
    and writes the results
245 to the vtf-file.
246 */
247 int main(){
248     printf("Wait for set up.............\n");
249     // Create servo channel
250     PhidgetRCServoHandle rcServo;
251     PhidgetRCServo_create(&rcServo);
252
253     // Create position channel
254     PhidgetVoltageInputHandle pos;
255     PhidgetVoltageInput_create(&pos);
256
257
258     Phidget_setDeviceSerialNumber((PhidgetHandle) pos, 85753); // Set the device
    serialnumber
```

```
259     Phidget_setChannel((PhidgetHandle)pos, 1); // Choose which channel to use
260     Phidget_openWaitForAttachment((PhidgetHandle)pos, 5000); // Open and wait for
attachment(usb)
261
262
263     Phidget_setDeviceSerialNumber((PhidgetHandle)rcServo, 42685); // Set the device
serialnumber
264     Phidget_openWaitForAttachment((PhidgetHandle)rcServo, 5000); // Open and wait for
attachment(usb)
265
266     // remove previous result files
267     remove("results.dat");
268     remove("dynpos.vtf");
269
270     // set intial values:
271     double dt = 0.2;
272     double total_time = 0;
273     double elapsed_time;
274     double integral = 0;
275     double pre_error = 0;
276
277     printf("set up is done!\n");
278
279     // Get reference from user
280     double percentage;
281     double reference;
282     printf("\nSet percentage of distance as referance(number between 0-1): \n");
283     scanf("%lf", &percentage);
284     printf("\nYou entered: %lf\n", percentage);
285     reference = setReference(pos, percentage);
286     printf("\nThe reference is %lf \n", reference);
287
288     // Get gain(Kp) from user
289     double Kp;
290     printf( "\nSet gain(Kp): \n");
291     scanf("%lf", &Kp);
292     printf( "\nYou entered: %lf \n", Kp);
293
294     // Get integral gain(Ki) from user
295     double Ki;
296     printf( "\nSet integral parameter(Ki): \n");
297     scanf("%lf", &Ki);
298     printf( "\nYou entered: %lf\n", Ki);
299
300     // Get derivative gain(Td) from user
301     double Kd;
302     printf( "\nSet derivate parameter(Kd): \n");
303     scanf("%lf", &Kd);
304     printf( "\nYou entered: %lf \n", Kd);
305
306     // Get the maximum run time from user
307     double max_run_time;
308     printf( "\nset maximum run time: \n");
309     scanf("%lf", &max_run_time);
```

```
310     printf( "\nYou entered: %lf \n", max_run_time);
311
312     // Run the program
313     while(total_time <= max_run_time){
314         // Set start time
315         int start_time = clock();
316
317         // Calculate error
318         double error = get_error(pos, reference);
319
320         // Calculate new thrust value
321         double new_thrust = regulator(Kp, Ki, Kd, error, dt, &integral, &pre_error);
322         printf("The new thrust is: %lf \n", new_thrust);
323
324         // set new thrust value
325         set_thrust(rcServo, new_thrust, dt);
326
327         // manage time
328         manage_time(&elapsed_time, &total_time, start_time, dt);
329
330         // write result to file
331         write_result(total_time, error, pos);
332     }
333
334     // Close and delete all channels
335     Phidget_close((PhidgetHandle) rcServo);
336     PhidgetRCServo_delete(&rcServo);
337     Phidget_close((PhidgetHandle) pos);
338     PhidgetVoltageInput_delete(&pos);
339
340     // write .vtf file
341     write_vtf();
342     return 0;
343 }
```