Nr. _____ av _____

Sjøkrigsskolen

Bacheloroppgave

Plattform for sverm

– Fra store avanserte plattformer til mange små –

av

Andreas Handal Hellesnes & Kim André Lyssand

Levert som en del av kravet til graden:

BACHELOR I MILITÆRE STUDIER MED FORDYPNING I ELEKTRONIKK OG DATA

Innlevert: Desember 2019

Godkjent for offentlig publisering

I. Publiseringsavtale

En avtale om elektronisk publisering av bachelor/prosjektoppgave

Kadetten(ene) har opphavsrett til oppgaven, inkludert rettighetene til å publisere den.

Alle oppgaver som oppfyller kravene til publisering vil bli registrert og publisert i Bibsys Brage når kadetten(ene) har godkjent publisering.

Oppgaver som er graderte eller begrenset av en inngått avtale vil ikke bli publisert.

Vi gir herved Sjøkrigsskolen rett til å gjøre denne oppgaven tilgjengelig elektronisk, gratis og uten kostnader	[X] Ja	[ ] Nei
Finnes det en avtale om forsinket eller kun intern publisering? (Utfyllende opplysninger må fylles ut) Hvis ja: kan oppgaven publiseres elektronisk når embargoperioden utløper?	[ ] Ja [ ] Ja	[X] Nei [ ] Nei

Plagiaterklæring

Vi erklærer herved at oppgaven er mitt eget arbeid og med bruk av riktig kildehenvisning.

Vi har ikke nyttet annen hjelp enn det som er beskrevet i oppgaven.

Vi er klar over at brudd på dette vil føre til avvisning av oppgaven.

Dato: 03.12.2019

Andreas Handal Hellesnes
Kadett navn

Kadett, signatur

Kim André Lyssand
Kadett navn

Kadett, signatur

II. Forord

Denne bacheloroppgaven er skrevet av Andreas Handal Hellesnes og Kim André Lyssand i perioden mai 2019 til desember 2019. Oppgaven er skrevet som en del av Militære studier med fordypning i elektronikk og data ved Sjøkrigsskolen.

Oppgaven tar for seg vårt arbeid med utvikling av en plattform for testing og utvikling av en maritim overflate-sverm for Sjøforsvaret. Med utgangspunkt i lite, men kjent teori utvikler vi en plattform som skal drive sverm til sjøs. Da det ikke er utviklet spesielle metoder som er bedre enn andre for å utvikle en sverm, benytter vi oss av "prøve og feile"-metoden. Plattformen er ment for å tette gapet mellom den teoretiske utviklingen av svermer og den maritime krigføringsarenaen. Leseren antas å ha grunnleggende kjennskap til programmerings språket Python, herunder bruken, skrivemåten og oppbygningen av programmer, funksjoner og moduler. Videre antas en grunnleggende kjennskap til datakommunikasjon.

Vi ønsker å takke forsker ved FFI, Aleksander Simonsen for god veiledning, deling av eksempel-materiell til bruk i oppgaven og konstruktive diskusjoner over telefon som har løst mange problemer underveis. Takk til førsteamanuensis Christophe Massacand for god veiledning og mange fruktbare diskusjoner. Videre ønsker vi å takke førsteamanuensis Aleksander Sauter for hjelp med utvikling av basestasjon og gode bidrag når problemer oppsto. Til slutt vil vi også rette en takk til et utvalg av kadettene i 2.klasse og 3. klasse på sjøkrigsskolen for deltagelse i testing av plattformen. Uten dere hadde ikke plattformen blitt det den er i dag.

Bergen, Sjøkrigsskolen, 03.12.2019

_____

Andreas Handal Hellesnes

_____

Kim André Lyssand

III. Oppgaveformulering

> *"Flere oppgaver som er vanskelige eller tidkrevende for en enkelt farkost alene, kan effektivt utføres av en sverm med relativt enkle farkoster"*
>
> *(Forsvarets forskningsinstitutt, 2019).*

Overflatesvermer har potensialet til å bli en viktig del av marinens fremtidige krigføringsplattformer. Derfor tar denne oppgaven for seg **utviklingen av en plattform for testing og utvikling av en maritim overflatesverm.** Oppgaven presenterer fremstillingen av denne plattformen, hva den kan brukes til og hvilket potensial sverm kan ha.

IV. Sammendrag

Norge har en intensjon om å være ledende på teknologi til sjøs (Hareide *et.al*, 2018). Noe som kjennetegner de som er ledende er at de er nyskapende og tar initiativ til ny teknologi. I utviklingen av maritim teknologi har autonomi lenge vært et fokusområde med ubemannede farkoster i spissen, med et uttalt ønske om å fjerne mennesker fra farlige situasjoner (Hareide *et.al*, 2018). Hva om ubemannede farkoster kan løse oppdrag sammen? Vil ikke det gi et hav av muligheter til å trekke personell ut av farlige situasjoner samtidig som kampkraften styrkes? Vi har derfor utviklet **en plattform for testing og utvikling av en maritim overflatesverm** for å vurdere mulighetene det kan gi den norske marinen i framtiden.

Begrepet sverm kommer fra naturen som en beskrivelse av hvordan dyr av forskjellige arter danner grupperinger for å overleve og løse oppgaver de ikke klarer alene. Det er utviklet mange algoritmer for å etterligne naturlige svermatferder, men ikke like mange plattformer å teste dem på. Målet vårt med denne oppgaven er derfor å redusere avstanden mellom algoritmer og testplattformer. Fokuset vårt har vært å bygge en testplattform som er robust og enkel å videreutvikle. Kravene til plattformen er at den skal være *enkel, skalerbar, parallell, desentralisert, modulerbar* og kunne *kommunisere* internt og eksternt. Svermen består av tre eller flere modellbåter som aktivt deler GPS- og bevegelsesinformasjon med hverandre og tar beslutninger om videre bevegelse basert på informasjonen. For å oppfylle kravene er det utviklet en autopilot for styring, kommunikasjonsprogrammer for datadeling og en beslutningstaker med to forskjellige atferder. En basestasjon er også laget for å overvåke plattformen sin funksjonalitet.

Testing av svermen har vært høyt prioritert. Basert på resultatene kan det konstateres at plattformen er klar for videre arbeid med å utforske mulighetene til svermteknologi. Mange små enheter som samarbeider er en retning i krigføringen som kan snu fokuset fra plattformintensivt til sensorintensivt. Et slikt skifte kan gi nye strategiske muligheter til de som besitter teknologien, derfor vil det å være en ledende nasjon i utviklingen av svermteknologi kunne vise seg å gi Norge en stor fordel i framtidens maritime krigføring.

V. Innholdsfortegnelse

VI. Liste av figurer

VII. Liste av tabeller

VIII. Nomenklatur

Ord	Beskrivelse
Algoritme	Beskrivelse av hvordan operasjoner skal utføres for å løse et problem/flere problemer
Autonomi	Selvstyre/selvbestemmelse
Aktuator	Organ som styres ved hjelp av en elektrisk strøm
Ad-hoc	Desentralisert oppstående nettverk uavhengig av ekstern infrastruktur
Boid	Fiktivt navn for en fugl
Desentralisering	Utflytting av makt fra sentrale enheter
MANET	Mobile Ad-Hoc Network, distribuert lokalt nettverk av noder
Ground Control Station	Kontrollstasjon som gir bruker en oversikt over egne ubemannede kjøretøys bevegelser og posisjon
Hardware	Fysiske komponenter i et system, f.eks. Raspberry Pi eller Pixhawk
HVU	High value unit
MASC	Mission-based Architecture for Swarm Composability
Metadata	Informasjon som beskriver annen informasjon
Meta operativsystem	Operativsystem (programvare) som bruker metadata.
Modul (Python)	.py fil som inneholder koder som kan brukes i overordnede programmer
Multicast	Dataoverføringsprotokoll over UDP
Pakker	Informasjon som skal fra en avsender til en mottaker
PID-regulator	Algoritme som brukes til å regulere en utgangsverdi
PSO	Forkortelse for: Particle Swarm Optimication
ROS	Robot Operating System
RPi	Forkortelse for: Raspberry Pi
Skalerbar	Muligheten for endring av størrelse, enten i fysisk størrelse eller antall
Syntax	Definerte regler og forventet struktur på tekst i forskjellige dataspråk, F.eks Python
USV	Forkortelse for: Unmanned surface vehicle

1 Innledning

Autonomiforsker ved Forsvaret forskningsinstitutt (FFI) Jonas Moen sa i 2016 at "Teknologien [sverm] har et enormt potensial på lang sikt" (Forsvarets Forskningsinstitutt, 2016). Sverm teknologi er ikke noe nytt i konsept, men som følge av mangel på hardware, har det ikke vært mulig å realisere før i dag. Forsvaret har de senere årene økt fokuset på dette området. Et fokus som kan synes å være noe de ønsker fortsette framover. En sverm betegnes som en gruppering av enkle individer som går sammen for å løse oppdrag utenfor deres individuelle kapabilitet. Inspirasjonen til sverm kommer fra observasjoner av dyrearter som samler seg og løser oppgaver, f.eks. hvordan fugler og fisker samler seg i flokker.

Sverm betegnes som tre eller flere enheter, f.eks. droner, som opererer sammen med minimal avhengighet til ekstern styring. Hver enhet følger et sett med grunnregler. Basert på disse reglene og informasjon de får fra miljøet rundt seg tar hver enhet beslutninger. Disse prinsippene for sverm gir framveksten av to taktikker som veldig få andre plattformer i dag har muligheten til å håndtere; metning og dekning. Begge bygger på å ha et stort antall med enheter som fokuserer på en oppgave. Metning baserer seg på å sende mange nok enheter mot en fiende til at den ikke klarer å håndtere alle, mens dekning er mer rettet mot å ha nok enheter til å kunne dekke et stort område effektivt. Disse taktikkene baserer seg på billige og enkle plattformer man har råd til å tape i oppdrag, noe Norge ikke har i dagens forsvar.

Sverm er fortsatt i en utviklingsfase og overgangen fra teori til virkelighet er utfordrende. Et viktig arbeid med denne overgangen skjer nå i USA. Amerikanerne utvikler en svermdoktrine for å støtte marinen i operasjoner. Doktrinen skal være et rammeverk av algoritmer som knyttes til ulike oppdrag (Giles & Giammarco, 2017) Under utviklingen har de tatt i bruk opp til 50 droner samtidig for å løse et oppdrag. Så hva kan 3-50 ubemannede droner som opererer samtidig utgjøre? Om man vurderer konseptene dekning og metning som nevnt over er det mulig å se for seg scenarier der en sverm med droner overvåker et stort område fra luften. Eller at en større gruppe med enkle

overflatefartøyer utstyrt med sonarer jakter ubåt i sverm. Begge disse løsningene innehar store fordeler sammenlignet med måten slike oppdrag blir løst i dag, særskilt når det kommer til å trekke operatøren lengre unna farlige situasjoner.

For at Sjøforsvaret skal kunne realisere potensialet sverm innehar som teknologi og konsept må det investeres penger i utviklingen av plattformer for testing. Den norske regjeringen har uttalt at Norge skal ha ambisjoner om å være ledende innenfor teknologi på sjøen (Hareide *et.al,* 2018). Norge og FFI er godt i gang med utviklingen av sverm med droner, men siden det er ambisjoner om å være ledende på sjøen burde maritime svermer også utvikles. Derfor er målet vårt å utvikle **en plattform for testing og utvikling av en maritim overflatesverm for Sjøforsvaret.**

Oppgaven vil først ta for seg nødvendig teori for at leseren skal kunne ha et teoretisk grunnlag til å forstå videre arbeid i oppgaven. Teorien tar for seg ingeniørfaglig systemtenkning, autonomi, datakommunikasjon, robot operating system (ROS), Node-RED og sverm. Videre vil oppgaven presentere kravene som ble satt til plattformen og innledende tanker om oppbygning av strukturen. Implementering av systemets komponenter blir så gjennomgått først generelt med en oversikt over prosessflyt og filstruktur, før hver enkelt komponent blir presentert mer i dybden. Implementeringssegmentet danner et grunnlag for testingen som er siste del, før drøftingen. I drøftingen ses det tilbake på kravene som ble stilt til plattformen, der dens kvalitet og videre potensial for sverm i det maritime, vil bli diskutert. Drøftingen og det arbeidet som er gjort vil være grunnlaget for konklusjonene våre. Oppgaven avsluttes med anbefalinger til videre arbeid.

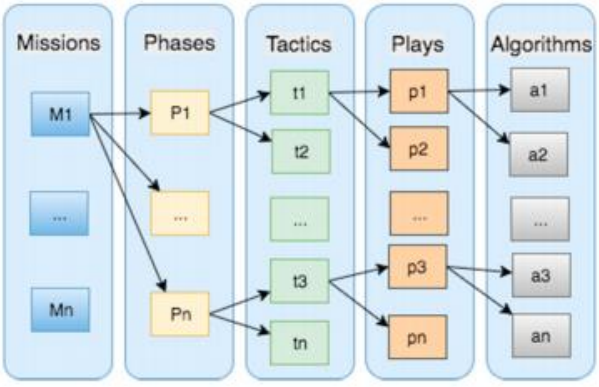
2 Teori

I den teoretiske delen av oppgaven vil det bli lagt fram det som antas å være viktig for å vite for å følge oppgavens gang. Begrepsforklaringer er del av nomenklaturen i avsnitt VIII og lagt inn som fotnoter underveis.

2.1 Ingeniørfaglig systemtenkning

"It is difficult, if at all possible, to reduce the meaning of systemic thinking to a brief definition" – Russel Ackoff, organisasjonsteoretiker (Edson, 2008). Det som er ikke er fullt så utfordrende er å si at systemtenkning har en viktig rolle i hvordan man ser på og arbeider med systemer. Et system er en ordnet sammensetning av flere deler som arbeider sammen for å løse et/flere problem. Systemtenkning blir da den kontinuerlige prosessen rundt utvikling, vedlikehold og vurdering av systemet. Det viktigste med systemtenkning for vår del er kunne se på plattformen som et enkelt system og som en del av et større system.

En Amerikansk kommandør ved navn Kathleen Giles presenterer i sin doktorgrad en svermdoktrine for å kunne støtte den amerikanske marinen med oppdrag (Giles & Giammarco, 2017). Doktrinen, som er navngitt MASC[1], skal være et rammeverk av oppdrag som blir delt opp til den minste detalj for å forklare hvordan svermen skal operere. Det utvikles kontinuerlig ulike algoritmer for sverm, men de blir ikke satt i et system. MASC skal systematisere algoritmene og anvende dem i oppdragsløsning. Som Figur 2-1 illustrerer blir doktrinene bygd ut fra oppdrag som kan være relevante. Flere algoritmer blir brukt til å løse ulike deler av oppdraget, basert på hvilken taktikk som blir brukt. Samlet gir dette et system av oppførsler som kan brukes til å løse oppdrag. (Giles & Giammarco, 2017)



Figur 2-1: MASC rammeverket illustrert (Giles & Giammarco, 2017)

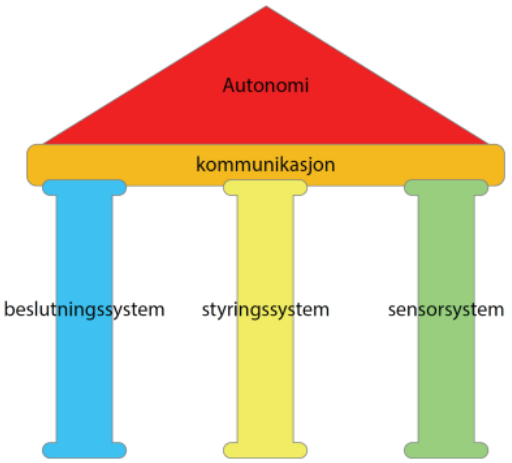
[1] MASC - Mission-based Architecture for Swarm Composability

2.2 Autonomi

"Sjøforsvaret har en lang historie med å utnytte ny teknologi for å øke den operative evne" (Hareide *et.al, 2018).* De siste årene har fokuset for denne utviklingen lagt på å redusere styring fra operatør og heller ta i bruk autonome systemer. Autonomi kommer fra det greske ordet *autos*[2] og *nomos* [3], og betyr delvis eller fullstendig selvstendighet, selvstyre eller selvbestemmelse (Hareide *et.al*, 2018). Ordet er ofte brukt i sammenheng med selvkjørende og/eller selvtenkende systemer.

Bruk og utvikling av autonome systemer er i høysetet mange steder i verden for å effektivisere prosesser og forhindre tap av menneskeliv. Et eksempel på satsning av utvikling er selvkjørende biler. Volvo planlegger å lansere sin autonome Volvo Cars i starten av 2020-årene og forventer at 1/3 av salget deres vil bestå av selvgående biler i 2025 (Raum, 2019). Det er ikke bare hos bil-produsenter autonomi står i høysetet, også i den maritime bransjen er utviklingen stor. Selv om de fleste prosjektene er i en tidlig fase, er teknologien tatt i bruk.

Ordet autonomi forbindes også ofte med fjernstyring eller automatisering. Forskjellen mellom autonomi og fjernstyring eller automatisering er evnen til å ta beslutninger. Et autonomt system vil kunne forstå situasjonen den er i og handle deretter (ofte gjennom enkle forhåndsdefinerte regler eller maskinlæring, der trening fører til erfaringsgrunnlag). Figur 2-2 illustrerer at et system må ha en sensorpakke, et styringssystem og en form for beslutningstaking som må kommunisere for at systemet skal være autonomt.



Figur 2-2: Bærebjelker for realisering av autonomi (Hareide *et.al*, 2018)

[2] Autos – Selv/egen
[3] Nomos – Lov/regel/styre

2.3 Datakommunikasjon

2.3.1 Nettverk og Ruter/IP

For at enheter skal kunne kommunisere må det finnes noe som håndterer meldingsgangen mellom dem. En god måte å håndtere meldingsgangen på er ved bruk av en ruter. Når en enhet starter opp vil den kringkaste at den trenger en statisk IP-adresse. Ruteren svarer og gir den en IP-adresse, en nettverksmaske (for å kunne vite hvilke enheter den kan kommunisere med uten å gå gjennom ruteren) og IP-adressen til ruteren. For båtene i svermen vil det kun være aktuelt å kommunisere internt i LAN[4]'et og det vil derfor kun være viktig at de får tildelt en IP som er på det samme lokale nettverket som de andre enhetene og kjenne multicast adressen til ruteren.

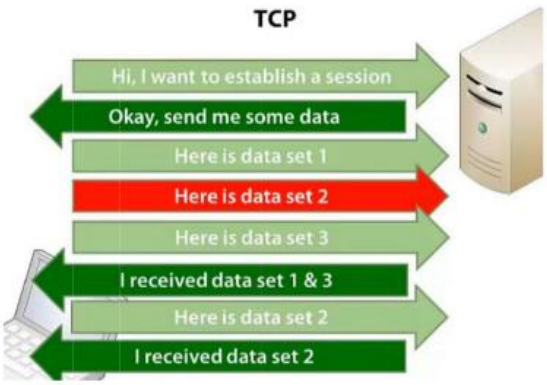
2.3.2 TCP/UDP

For at meldinger skal kunne sendes over nettverket må det opprettes en forbindelse mellom avsender og mottaker. Det finnes to typer forbindelse som tar seg av sending av pakker, TCP[5] og UDP[6]. TCP er opptatt av kontroll over det som blir sendt. Det opprettes en toveis forståelse for at det er en forbindelse (SYN->SYN+ACK->ACK) før pakken sendes. Når pakken er sendt kommer det en bekreftelse på at pakken er mottatt. TCP er altså en forbindelsesorientert, men tidkrevende prosess, som vist i Figur 2-4. Ønsker man en mer tidseffektiv prosess er UDP løsningen. Figur 2-3 viser at UDP ikke oppretter forbindelse med mottaker for å sende en pakke. UDP kan brukes når det er viktig med bekreftelse på forbindelse, men da må applikasjonen ta seg av kontrollen av dette. Det kan altså være nyttig å gjøre seg en tanke om hvilken form for transport som er fornuftig å bruke i ulike situasjoner. For å hente adresser fra internettet kan UDP være klokt siden det er mange instanser den skal innom (DNS), men hvis det skal sendes en melding over f.eks. Facebook er det ønskelig med en bekreftelse på mottatt melding som gjør TCP til den beste løsningen.

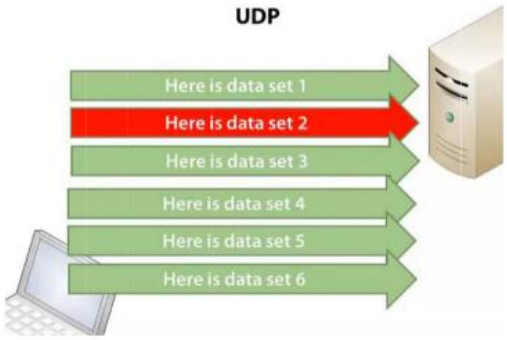
[4] LAN – Local area network

[5] TCP – Transmission control protocol

[6] UDP – User datagram protocol



Figur 2-4: TCP beskrivelse (Networking Beginners, 2014)



Figur 2-3: UDP beskrivelse (Networking Beginners, 2014)

For svermen vil ROS ta valget om UDP eller TCP skal benyttes basert på hva den selv tenker er best. Det er viktig at de små pakkene som kommer (opptil 10/sek (10Hz)) er oppdatert informasjon. Samtidig er det ikke problematisk om en pakke forsvinner, da den får en ny, millisekunder senere. På bakgrunn av dette vil det virke mest naturlig at ROS benytter seg av UDP.

2.3.2.1 MULTICAST

I prosjektet vil det være aktuelt å sende UDP/TCP pakker til flere adresser samtidig. Istedenfor å sende separate pakker til hver adresse bruker man multicasting. Multicast gir avsender mulighet til å sende ut data til flere mottakere på nettverket samtidig uten å kjenne adressen til mottakerne. Ruteren vil da fordele pakken til alle som har abonnert på multicasten på nettverket. En utfordring med denne formen for informasjonsdeling er at multicast opptar mye båndbredde når forbindelsene er trådløse. Ved f.eks. bruk av video kan det skape store problemer. For oss vil det ikke ha noe å si, da multicast er det eneste som opptar båndbredde i vårt system. (Holm, 2003)

2.4 Robot Operating System

Robot operating system (ROS) er et metaoperativsystem[7]. ROS gir blant annet verktøy og biblioteker for å skaffe seg, bygge, lese, skrive og kjøre kode internt og på tvers av flere maskiner (Open Source Robotics Foundation, 2018). I vår oppgave blir ROS hovedsakelig brukt som en informasjonskanal mellom forskjellige programmer, eller noder, for at data skal være lett tilgjengelig og at flere noder skal kunne nytte samme data samtidig. Dette er den mest grunnleggende funksjonen til ROS.

ROS-nettverket er bygd opp av en master og flere program-noder, som alle har en unik adresse i systemet. Det er masteren som deler ut disse adressene ettersom nodene oppstår, dermed har den også total oversikt over adressene til nodene i systemet. Hver enhet kjører sitt eget unike ROS-system med egen master, det er ingen form for kommunikasjon utenfor operativsystemet med ROS. Selv inneholder og kontrollerer masteren adresser og navn til alle forskjellige noder og topic'er som det blir publisert data på. Alle nyoppstartede noder må inn til masteren for å finne plasseringen til dataene. ROS kan ha flere arbeidsrom i filsystemet på hver PC, her ligger alle konfigurasjonsfiler og andre tilleggspakker til ROS, dette kalles et *workspace*. Alle arbeidsrommene har unike konfigurasjonsfiler, det er også i et av arbeidsrommene man legger filene til systemet. Det gjør mappen med filene våre til en ROS-pakke eller *package*. Pakkenavn brukes som plasseringsdefinisjon i launch-filer som beskrives senere, men det står også mer om ROS arbeidsrommet som helhet og hvordan installere ROS i Vedlegg C – ROS installasjon og bruk.

2.4.1 Noder

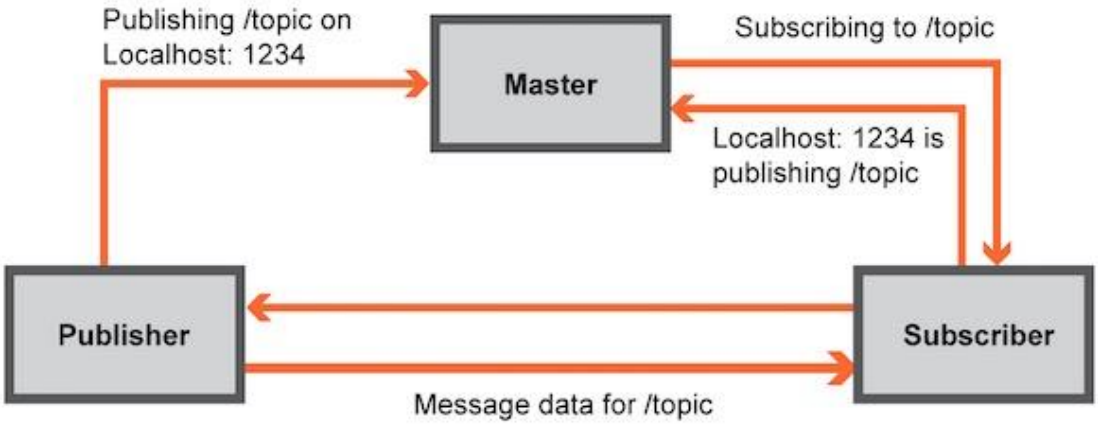
En node i ROS er et verktøy for å få kjørt en kode på ROS-nettverket. Noder kan kjøre uten tilførsel eller utlevering av data, men dette er veldig unaturlig på ROS, siden det er et operativsystem rettet mot roboter. Roboter er som regel sensortunge systemer med mye informasjon som skal inn og ut fra hjernen i systemet. Derfor er det mer naturlig at en

[7] Metaoperativsystem - Operativsystem (programvare) som bruker metadata, informasjon om informasjon.

node enten *publiserer*, *abonnerer* eller begge. Publisering og abonnering er beskrivelsen av det å enten dele - eller å hente ut data fra ROS-systemet. Dette er data som gjerne skal brukes i noden for utregninger eller sendes videre ut til en annen node. Et eksempel på dette er en generell autopilot-node, denne noden skal som regel ta inn data fra forskjellige sensorer, *abonnere*. Videre bruker den disse dataene til å finne sin egen posisjon og regne ut utslagene den må gjøre for å nå en ønsket posisjon. Disse utslagene er som regel representert med et tall, dette tallet må autopilot-noden sende videre slik at noden som er ansvarlig for å styre enheten kan styre mot målet, *publisere.*

2.4.2 Topic'er

Prosessen med å *publisere* og *abonnere* på data i ROS skjer ved hjelp av topic'er. Topic'er er et støtteverktøy for å navngi porter/busser der noder kan utveksle data. Som vist i Figur 2-5 går prosessen for å dele data over topic'er i steg. Først sier en *Publisher*-noden til

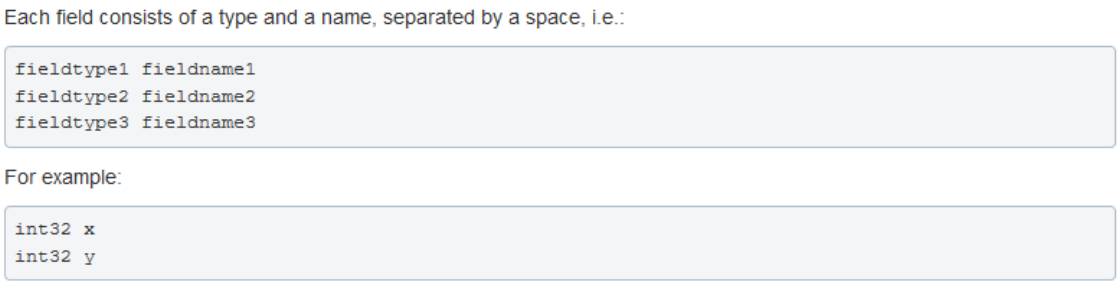


Figur 2-5: Oversikt virkemåte ROS master, noder og topic'er

masteren at den publiserer data på en port under navnet til et topic definert i noden. Dette lagrer masteren for videre bruk om en *Subscriber* skulle ønske data fra dette topic'et. Når det da startes en *Subscriber*-node som har behov for data fra dette topic'et kontakter den masteren. Masteren gir tilbake adressen til porten der data på dette topic'et publiseres fra *Publisher*-noden. Videre kan *Subscriber*-noden hente ut dataen den trenger fra det topic'et på adressen den fikk fra masteren så fort som *Publisher*-noden oppdaterer den.

2.4.3 Messages

ROS-messages, forkortet msgs, holder formatet .msg og er et meldingsformat unikt for ROS, designet for at data som blir brukt på tvers av noder over diverse topic'er blir gitt i



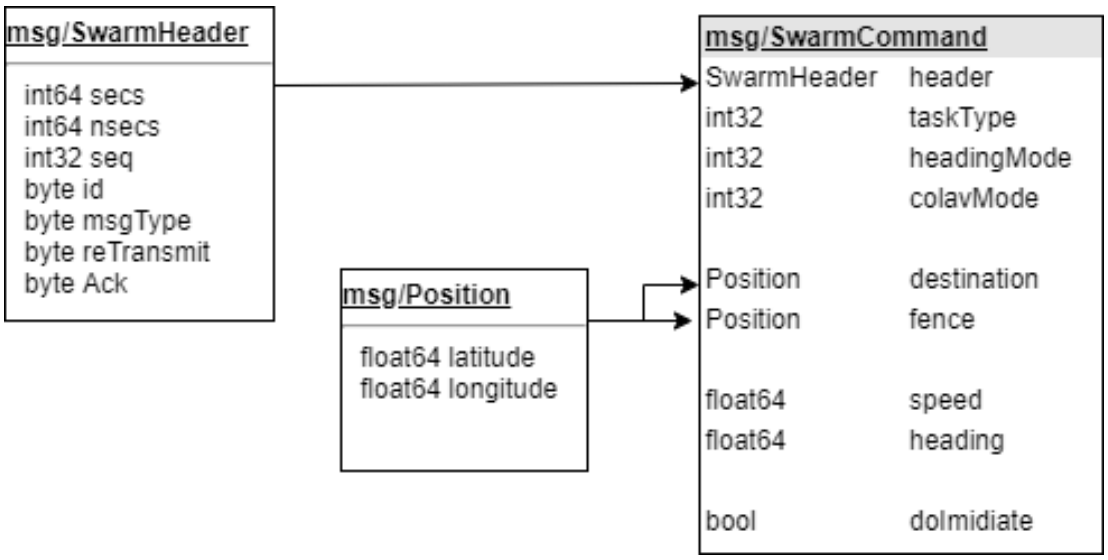
Figur 2-6: Eksempelforklaring av meldingsformatet msgs

et felles format. Dette for at data være universelt brukbart gjennom hele ROS systemet. Msgs er et forenklet beskrivende meldingsformat, som er lagd for å enkelt kunne beskrive hvilken type data som blir sendt over forskjellige topic'er. Det er bygd opp av felttyper og feltnavn, som vist i Figur 2-6. Her markerer alle *fieldnames* eller feltnavn forskjellige datavariabler som blir sendt i denne meldingen. *Fieldtype* eller felttype beskriver hvilken datatype som feltnavnet kommer til å inneholde.

Det at .msg-formatet beskriver nøyaktig hvor mye og hvilken type data som kommer i hver pakke på et topic gjør intern data-deling i ROS over topic'er enormt effektivt. Det er fordi ROS kan ta seg av formateringen av alle busser internt for deling av data og kun gi dem den plassen de har behov for. På bakgrunn av hva msg sier om hvilken data som skal sendes bygger ROS en tunnel som passer akkurat for den dataen, derfor blir det så effektivt som mulig. Det tillater også ROS å tilpasse hvordan dataen sendes, enten som UDP eller TCP. Som regel er det UDP som blir valgt fordi standarden i ROS er at det skal gå mange små pakker fram og tilbake med høy frekvens, det er ikke så viktig om en pakke blir borte, som nevnt i 2.3.2 – TCP / UDP.

Man kan definere egne meldingstyper innenfor .msg-formatet. Derfor er det siste eksempelet på .msg-typene hentet fra vårt system. Dette er meldingsoppbygningen til *SwarmCommand* eller sverm kommando, som er en melding som blir sendt fra basen til alle enhetene i svermen og inneholder forskjellige ordre som svermen kan få.

Meldingsformatet er vist i Figur 2-7, her vises hvordan hovedmeldingen *SwarmCommand* inneholder egne undermeldinger som også er unike for systemet vårt.



Figur 2-7: msg-oversikt for *SwarmCommand,* med inkluderte undermeldinger

2.4.4 Launch-filer

Launch-filer er det ROS bruker for å starte noder i systemet. I ROS bruker man en kommando kalt *roslaunch* for å si til systemet at denne filen skal åpnes med ROS. Filene man åpner med *roslaunch* er launch-filer, disse er skrevet på XML[8]-format. En launch-fil definerer variabler for nodene den skal starte og inkluderer eventuelle underfiler. Nodene den starter er filer skrevet i enten Python eller C++, de to primærspråkene ROS støtter pr nå (Open Source Robotic Foundation, 2018). En launch-fil kan starte en enkel node, med argumenter som i eksempelet i Figur 2-8, der autopiloten starter med en launch-fil.

```
You, a few seconds ago | 2 authors (Andreas Hellesnes and others)
1  <!-- Launch file for autopilot node -->
2  <launch>
3
4  <node name="autopilot_node" type="Swarmpilot.py" pkg="swarm" output="screen" respawn="true">
5
6  </node>
7  </launch>
```

Figur 2-8: Launch-fil eksempel som starter en node

[8] XML – Extensible markup language

Argumentene som blir gitt i denne launch-filen er navnet til noden, *name*, hvilken fil den skal åpne (type; den filen som inneholder programmet til noden), hvilken pakke filen ligger i, *pkg*, hvor den skal gi eventuelle outputs av advarsler eller feil, *output* og til slutt om den skal starte på nytt om noden skulle krasje, *respawn*.

Launch-filer kan også referere til andre launch-filer og på den måten starte flere noder samtidig på samme terminal. Dette er vist i Figur 2-9 som er launch-filen vår for å starte hele systemet på en båt. Her inkluderes filene på linjene 12-16, alle filer som inkluderes starter som en egen node, basert på kode som ligger i filstrukturen presentert i 4.3.

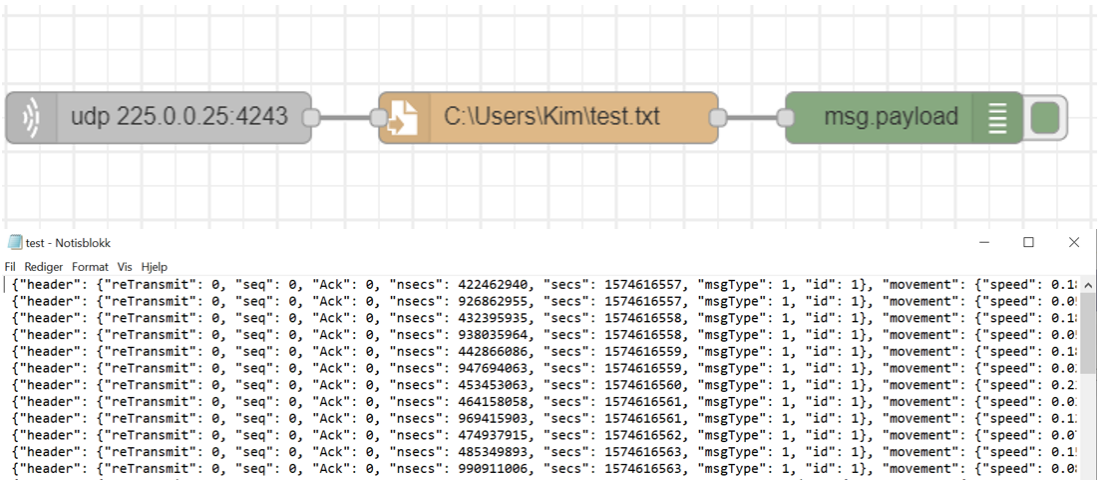
```
1  <!-- Launcher for entire swarm system -->
2  <launch>
3
4      <arg name="enable_datalink" default="true"/>
5      <arg name="enable_cmdlink" default="false"/>
6      <arg name="enable_video" default="false"/>
7      <arg name="enable_recording" default="true"/>
8      <arg name="enable_mavros" default="true"/>
9
10
11     <!-- include other .launch files -->
12     <include file="$(find swarm)/launch/autopilot.launch"/>
13     <include file="$(find swarm)/launch/swarm_control.launch"/>
14     <include file="$(find swarm)/launch/rx.launch" if="$(arg enable_datalink)"/>
15     <include file="$(find swarm)/launch/tx.launch" if="$(arg enable_datalink)"/>
16     <include file="$(find swarm)/launch/px4.launch" if="$(arg enable_mavros)"/>
17     <!--
18     <include file="$(find swarm)/launch/rosbag.launch" if="$(arg enable_recording)"/>
19     -->
20
21 </launch>
```

Figur 2-9: Launch-fil eksempel som starter flere noder samtidig

2.5 Node-RED

Node-RED er en nettbasert flyteditor som gjør at man kan bruke maskinvare via programvare. Den har et bredt utvalg av paletter, som virker som tilleggspakker for programmet. Med å laste ned paletter får man tilgang til nye noder. Alle noder er unike, det vil si at de utfører ulike oppgaver. Meldinger som blir sendt over nodene blir gitt videre med en objekt-variabel med navn "msg" fra node til node. Det er mulig å programmere noder (f.eks. funksjon-noden), da blir JavaScript brukt som programmeringsspråk.

Et eksempel på hvordan Node-RED kan brukes er vist i Figur 2-10. En UDP-input node blir brukt for å lytte etter multicast-meldinger på port 4243. Objektet som kommer inn blir lagret i en fil og så lest av i *debug*-vinduet av en *debug*-node.

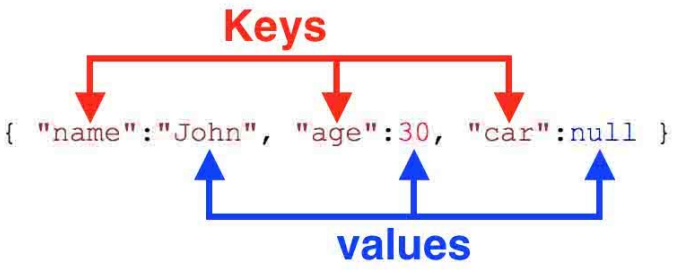


Figur 2-10: Eksempel på bruk av noder i Node-RED.

2.5.1 JSON

JSON er en forkortelse for *JavaScript Object Notation* (Crockford, 2017) og er et lettvekt språk for datautveksling originalt utviklet basert på JavaScript sine standarder, for bruk i JavaScript. Det har utviklet seg til å være universelt tilgjengelig i de fleste moderne programmeringsspråk, som C++ eller Python, gjerne under andre navn, men med tilsvarende virkemåte. Bakgrunnen for spredningen er datastrukturen til JSON, som er

lett leselig for mennesker og samtidig lett for kompilatorer å analysere. JSON-strukturen bygger på samlinger av navn og verdi-par adskilt med komma inne i et objekt, avgrenset med klammeparenteser som vist av eksempelet i Figur 2-11.



Figur 2-11: Eksempel av JSON-objekt struktur

Figuren viser et JSON-objekt som inneholder 3 verdier. Hver av verdiene er lagret med en *nøkkel* eller et navn til verdien. Navnet må være tekst, men de fleste verdityper som tekst, tall, boolsk eller tabeller kan lagres i et JSON-objekt. JSON-objekter kan også inneholde det som kalles for nøstede objekter, som i all enkelhet er et JSON-objekt inne i et JSON-objekt. JSON vil bli benyttet i prosjektet for å konverter meldinger fra ROS til systemet, samt i Node-RED for lagring av verdier fra multicast. Dette fordi det er lett å lese av verdiene direkte, samt at det er laget for JavaScript som blir brukt til koding av noder i Node-RED.

2.6 Sverm-intelligens

En gruppe autonome systemer som utfører et distribuert oppdrag uten sentralisert styring innehar sverm-intelligens (Department of Computer Engineering, 2015). Begrepet sverm er hentet fra naturen som beskrivelse av hvordan dyr av forskjellige arter danner naturlige grupperinger for å overleve og løse oppgaver de ikke klarer alene. For eksempel hvordan maur bruker feromoner til å sammen finne korteste veien til et mål, hvordan fugler flyr i flokk eller kakerlakker beveger seg. Disse dyrene er enkle skapninger i den forstand at individer i gruppen har begrenset med kognitiv kapasitet og evner til å løse oppgaver alene, men i samspill med hverandre kan disse kollektivt løse komplekse oppgaver.

Ved bruk av algoritmer kan man gjenskape dyrenes naturlige sverm-egenskaper til roboter. Algoritmene er ment til å tas i bruk på roboter for å gjenskape oppførslene for å løse ulike oppdrag. Et eksempel kan være overvåking av en militærbase med droner som

flyr som en flokk. Det er derfor viktig at svermalgoritmen samarbeider med robotene. Det blir derfor stilt fem krav til svermalgoritmen. Tabell 1 tar for seg hvert krav og hva det innebærer for algoritmen:

Krav	***Beskrivelse av krav***
Enkel	Siden evnen til hver robot er begrenset skal algoritmen være så enkel som mulig.
Skalerbar	Algoritmen må være skalerbar. Det må være mulig å legge til og trekke fra roboter uten å gå inn i koden for å endre.
Parallell	Algoritmen skal være så lik som mulig på hver robot.
Kommunikasjon	Kommunikasjon mellom robotene skal foregå lokalt og ikke globalt, f.eks. gjennom en ruter.
Desentralisert	Hver robot i svermen skal ta egne avgjørelser og unngå løsninger der avgjørelser tas sentralt utenfor svermen.

Tabell 1: Krav til svermalgoritmen (Tan, 2013).

Svermen som brukes i denne oppgaven består av småbåter som jobber sammen for å løse oppgaver utenfor enkeltindividets kapasitet. Videre vil teorien ta for seg videreføringen av konseptene funnet i naturen til anvendelse i robotikken.

2.6.1 Kollektivt handlemønster

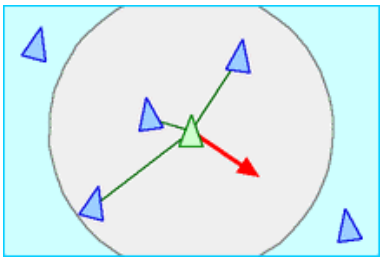
For at systemer av flere enheter skal kunne kommunisere må det ligge enkle regler til grunn for at kommunikasjonen skal gå effektivt og for at ulykker ikke skal forekomme. Gjennom observasjoner av fenomener i dyreriket er det mulig å forstå hva som gjør at enkelte arter, som for eksempel fugler eller fisker, kan kommunisere med de rundt seg.

Det er gjort mange forskningsprosjekter på svermintelligens i forbindelse med dyr, men i denne oppgaven blir det kun sett nærmere på fugler, bier og fisker.

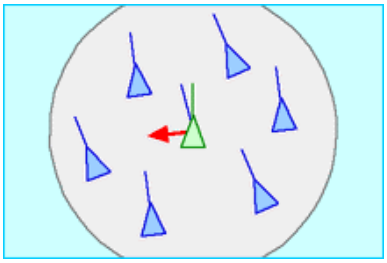
2.6.2 Flocking

"Birds of a feather flock together". I 1986 utviklet Craig Reynolds, en datagrafikk ekspert, en modell for *flocking* der han beskrev hvordan fugler grupperer seg ved hjelp av enkle regler. Han valgte å gi fuglene navnet boids. Det han oppdaget var at gruppens adferd er utelukkende avhengig av hvilke valg hver enkelt boid tar for å interagere med situasjonen den befinner seg i. Reynolds oppdaget at en boid ville handle på tre ulike måter basert på posisjonen og farten til de andre boidene rundt seg.

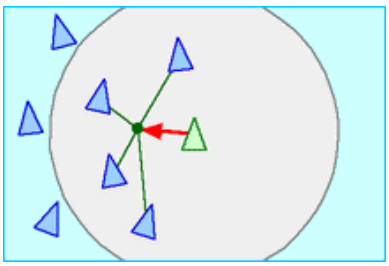
1. *Separation*[9], Figur 2-12: En boid vil styre vekk for å unngå å kollidere med andre boider.
2. *Alignment*, Figur 2-13: En boid vil styre den gjennomsnittlige retningen og farten som dem rundt seg.
3. *Cohesion*, Figur 2-14. En boid vil styre mot flokken hvis den anser seg selv som for langt unna til å danne en formasjon.



Figur 2-12: Separation



Figur 2-13: Alignment



Figur 2-14: Cohesion

Flocking krever at en boid kun tar hensyn til andre rundt seg i en liten radius. Boiden vil altså ikke bry seg om hva som ligger utenfor de nærmeste noe som gjør prosessering av informasjon mindre kompleks. Denne formen for beslutningstaking kan brukes i teknologi der enheter skal ta beslutninger helt individuelt rundt andre enheter. Så lenge den har informasjon om hva som finnes rundt seg, kan algoritmer basert på boids, bestemme hva den skal gjøre. (Reynolds, 2001)

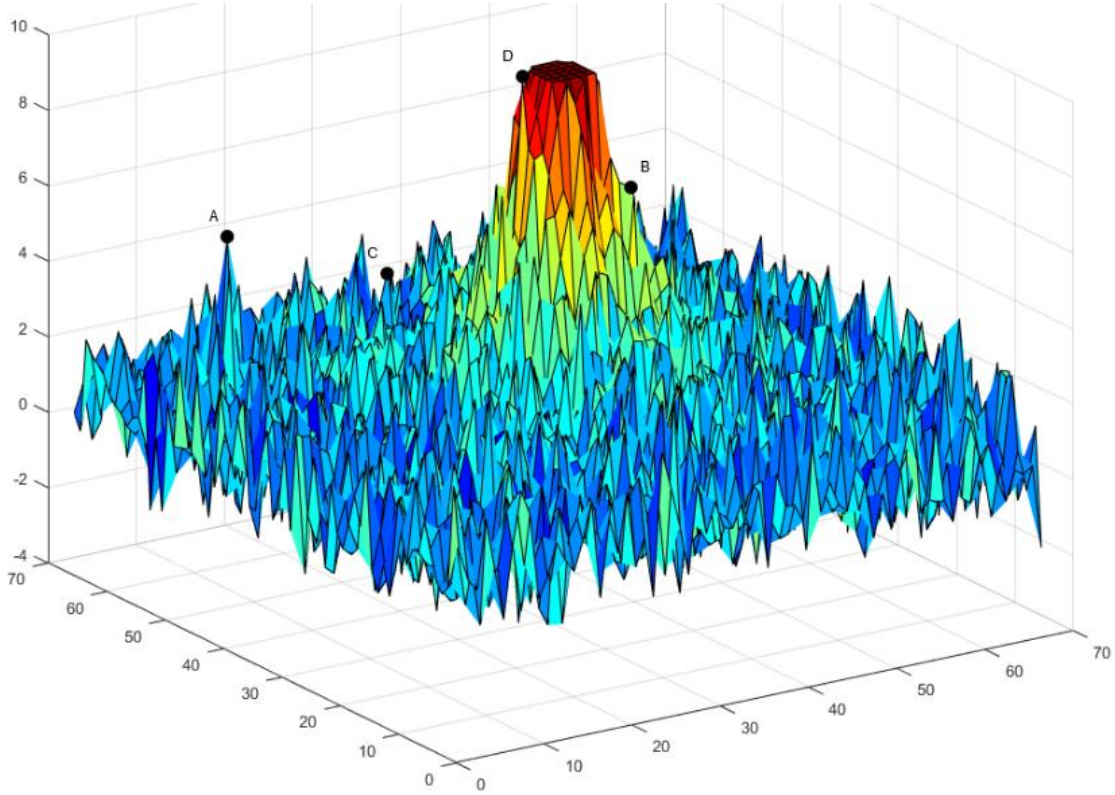
[9] Figurer fra Reynold, 2001

2.6.3 Particle swarm optimization

"Particle swarm optimization (PSO) is a robust evolutionary strategy inspired by the social behavior of animal species living in large colonies like birds, ants or fish." (ScienceDirect, 2019). I 1995 kom Dr. Kennedy og Eberhart med en algoritme som beskriver hvordan grupper med organismer bruker hverandre til å finne en optimal løsning. De observerte lignende atferd hos flere arter som gjorde dem sikker i sin konklusjon om at noen organismer klarer å dele informasjon med hverandre, selv om de er uorganisert eller spredt i sverm.

Den vanligste formen for simulert PSO er beskrevet av Xin-She Yang i boken Nature-Inspired Optimization Algorithms (Yang, 2014). PSO benytter seg av både et personlig beste (xBest)- og globalt beste (gBest) punkt for å kalkulere adferd. Dette er gjort for å gi variasjon mellom de gode løsningene og for å være robust mot støy. Hvordan det gjøres og hvordan PSO kan simuleres skal beskrives med utgangspunkt i simulering av PSO. I simuleringer av PSO er det naturlig at det settes inn et optimalt punkt som svermen skal flokke seg rundt. En partikkel vet ikke hvor den optimale lokasjonen er, men vet sin egen beste posisjon basert på hvor den var nærmest det optimale punktet. Verdien for hvor nærme den er målt som en styrke målt eller simulert. For simulering er det mulig å bruke det optimale punktet til å sende ut et signal. Signalet har en styrke som avtar basert på avstanden fra punktet, vist i Figur 2-15 med funksjonen $\frac{1}{x^2+y^2}$, med støy, der r

representerer avstanden fra sentrum av figuren. Her representerer fargen signalstyrken en partikkel vil måle, jo rødere desto sterkere signal.



Figur 2-15: 3D visning av opplevd signalstyrke i PSO

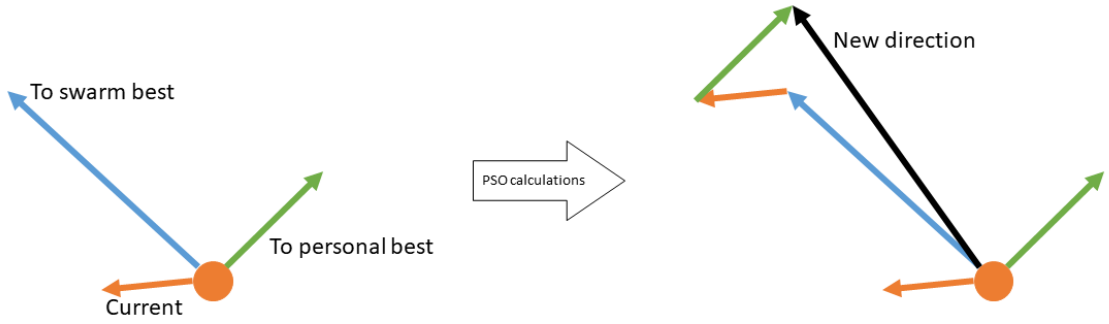
Denne signalstyrken blir bruk for å bestemme både *personlig-* og *global beste.* I Figur 2-15 er det tegnet inn noen punkter; *A, B, C og D.* Hvis dette representerer 4 partikler sine startposisjoner i PSO simuleringen blir hvert av disse punktene det personlige beste punktet til hver av partiklene. Når alle deler sin personlig beste med hverandre, altså signalstyrken de fikk og posisjonen de var i vil punktet D bli det felles globalt beste punktet, siden det er der det ble målt sterkest signalstyrke. For hver bevegelse deler de sin personlig beste med hverandre for å se om noen har målt noe sterkere enn den foreløpig globalt beste. Det blir da det nye globalt beste punkt for alle.

Punktene; *personlig-* og *global beste* blir brukt i formelen for PSO vist i Figur 2-16. Dette er en utregning som avgjør bevegelsen for hver enkel partikkel. Den bruker punktene *xBest* og *gBest* som de to beste punktene og *x* som den nåværende posisjonen til partikkelen. Ved å trekke hvor den er fra de to beste posisjonen får den to retninger mot hver av de to beste punktene. Dette er visualisert i Figur 2-18, med vektorpiler. Her er det

$$v_i^{t+1} = \omega v_i^t + c_1 r_1 \left(xBest_i^t - x_i^t\right) + c_2 r_2 \left(gBest_i^t - x_i^t\right)$$

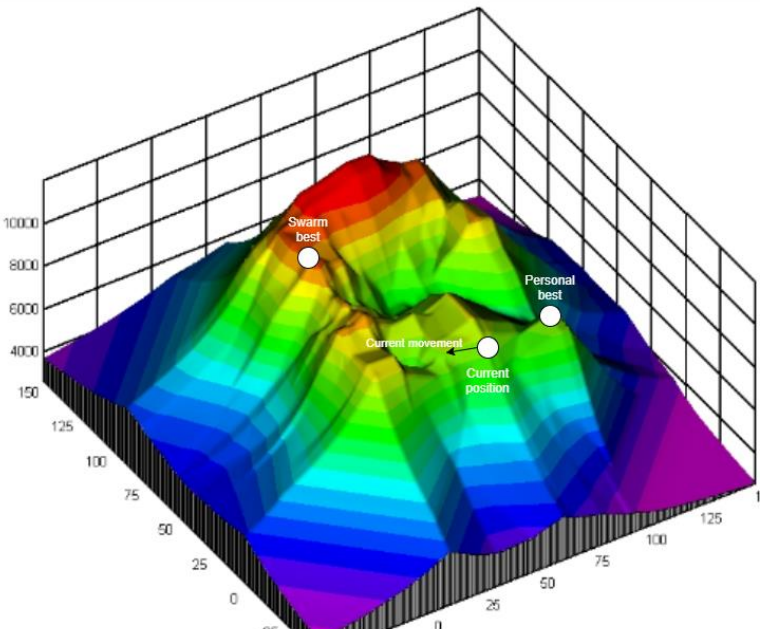
Figur 2-16: PSO-algoritmen (Martínez & Cao, 2019)

satt noen vilkårlige beste punkt for å kunne tegne retningsvektorene mot punktene. Formelen i Figur 2-16 har er tre vektorer som legges sammen til en ny vektor som er neste



Figur 2-18: Visualisering av PSO algoritmen

bevegelse. Det er nåværende bevegelse wv^t eller «*current*», $c_1 r_1 (xBest^t - x^t)$ eller «*to personal best*» og $c_2 r_2 (gBest^t - x^t)$ som er «*to swarm best*». Variabelen *r* er en tilfeldig variabel som endrer seg konstant for å sikre variasjon i løsninger og *c* konstanter for å endre hvor mye hver partikkel skal bevege seg som enten globalt eller personlig beste. Dette er avgjørende for hvordan svermen håndterer støy, konstantene kan også kalles «tillit» som er hvor mye den stoler på sine egne verdier og da spesielt hvor mye den skal stole på den globalt beste. I Figur 2-17 er et eksempel vis på hvordan

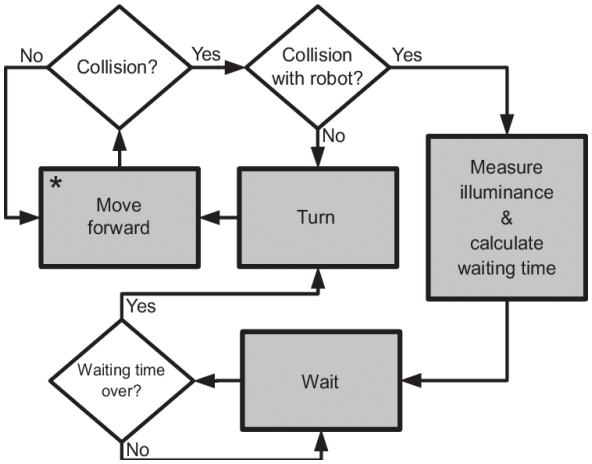


Figur 2-17: 3D visning av virkningen til støy i PSO

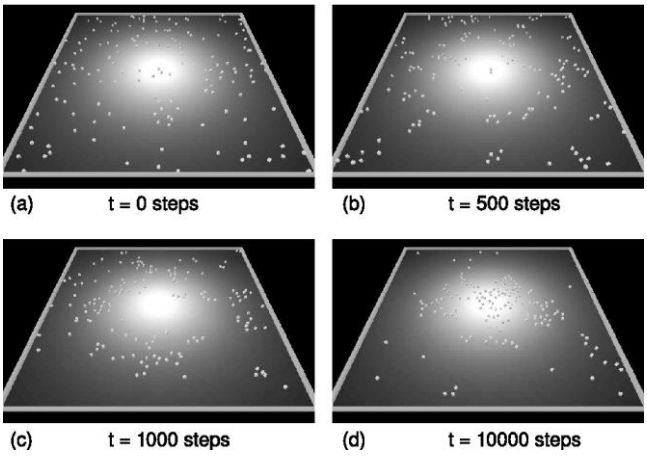
støy kan påvirke hver partikkel til å bevege seg lengre fra det optimale ved å stole for mye på seg selv. Svermens beste verdi er ganske nær den optimale verdien, mens den personlige beste er på en egenstående topp som kommer av støy i signalet. Dette punktet kommer til å «forstyrre» partikkelen i sin jakt etter det optimale punktet, men det er også en kraft i PSO at de ikke alle beveger seg etter et punkt, globale beste. Dette gjør at partiklene utforsker hele området før de med tiden samler seg rundt det optimale punktet.

2.6.4 Beeclust

Beeclust algoritmen er en sverm-algoritme som er inspirert av hvordan honningbier oppfører seg i interaksjon med andre honningbier. Som illustrert i Figur 2-20 vil hver honningbie bevege seg i området til de kolliderer med noe. Møter den på en hindring vil den bare snu og fortsette søkingen, men møter den på en annen honningbie vil den stoppe opp, sanse de lokale omgivelsene og står i ro en gitt periode. Perioden den står i ro varierer med hvor optimal temperaturen i området er, der en *bra* temperatur vil føre til at den står i ro lengre enn ved en *dårlig* temperatur. Dette gjør at honningbiene ikke trenger å kommunisere med hverandre. Etter hvert vil det danne seg store mengder honningbier i det optimale området vist i Figur 2-19. En lengre periode i ro fører til en lengre tilværelse i det optimale området for honningbiene og en større sannsynlighet for at en annen honningbie vil kollidere med dem der.



Figur 2-20: Handlemønster (Boid *et.al*, 2012)



Figur 2-19: Visualisering av en beeclust. lysmaksimum er varmeste punkt (Schmickl & Hamann, 2011)

3 Maritimt sverm-system

3.1 Krav

I denne delen av oppgaven vil sverm systemet bli presentert som helhet. For at konstruksjonen skal kunne betraktes som en sverm må det stilles noen krav til den. Kravene skal både hjelpe oss under utviklingen, men også til kvalitetssikring av arbeidet i etterkant. Kravene til systemet blir basert på kravene til svermalgoritmen som er presentert i Tabell 1, for at kravene skal gjennomsyre hele systemet og ikke bare algoritmen som utvikles. I Tabell 2 vil kravene bli presentert sammen med ønsket måloppnåelse.

Krav	***Beskrivelse av krav***	***Ønsket måloppnåelse***
Enkle enheter	Plattformen må bestå av enkle enheter som skal samarbeide.	Vi tenker å bruke enkle småbåter som er utviklet på skolen som skrog, RPi 3B+ til å håndtere programvare, Arduino Uno for å sende kommandoer til motorer og servoer og Pixhawk 4 som sensorpakke (GPS, kompass og bevegelsesdata).
Skalerbar	Det skal kunne fjernes og legges til droner uten at det skal påvirke svermen.	Vi tror den enkleste måten å gjøre svermen skalerbar på er at kommunikasjonsdelen av koden lager en liste over andre USV'er, så kan antall objekter i listen avgjøre hvor mange det er i svermen.
Parallell	Enheter skal kunne utføre handlinger uavhengig av hverandre.	Vi ønsker at koden som utvikles skal kjøre på hver USV[10]. Det gjør at alle utregninger og

[10] USV – Unmanned Surface Vessel

		beslutninger tas på sin egen RPi om bord.
Kommunikasjon	Det skal finnes en form for intern kommunikasjon mellom enhetene.	Vi ser for oss at kommunikasjonen kommer til å gå over en trådløs ruter, der alle sender til alle.
Desentralisert	Styring av droner/småbåter skal skje av seg selv og ikke av en ekstern enhet.	Tanken er at hver USV skal ha en autopilot som styrer skroget der den bestemmer seg for å gå. Et sentralisert panel (basestasjon) er ønskelig å utvikle som skal kunne overstyre autopiloten.
Modulerbar	Plattformen skal være lett å utvikle videre for andre.	Kodestrukturen blir bygget opp av hovedfiler og støttemoduler som løser funksjonaliteten.

Tabell 2: Systemkrav med ønsket måloppnåelse (basert på Tan, 2013).

3.2 Begrensninger

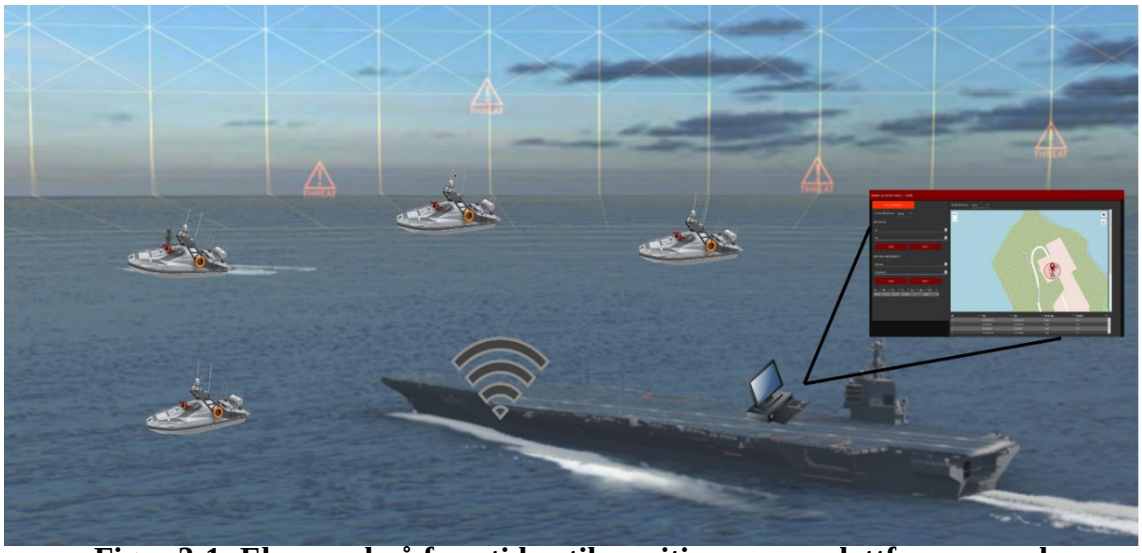
Det er mange vinklinger som kan brukes for å vise konseptet sverm, men vi bestemte oss tidlig for å begrense oss til det maritime domenet med bruk av småbåter. Vi jobber i Sjøforsvaret og ønsker derfor å se på maritime løsninger for sverm. Tid og budsjett var også en del av valget. Skulle vi utviklet en sverm av droner måtte de kjøpes inn og bygges, noe som ville kostet mye tid og store deler av budsjettet. Småbåter derimot, stod klare for bruk etter et tidligere prosjekt på skolen. Foruten sensorer, var alle båtene ferdig utstyr med nødvendige komponenter. Det er ønskelig at båtene skal kunne gjenbrukes til det formålet de ble laget til. Vi vil derfor gjøre minst mulig endringer på dem.

Småbåtene er simple i den form at det ikke er mange avanserte komponenter å lære seg. Dette gir mulighet til å sette seg bedre inn i programvare som Python, ROS, Arduino og Node-RED. Programvare vil være en stor del av oppgaven. Dette vil ikke bare bidra til økt forståelse for vår egen del, men også for kadetter som skal jobbe videre med systemet i framtiden. Vi kommer derfor kun til å bruke nødvendig programvare.

Vi kan heller ikke teste over for store distanser, siden vi benytter oss av en Wi-Fi ruter som data-hub blir vi begrenset i hvor langt unna båtene kan være ruteren før de faller ut av nettet og dermed også svermen. Ruteren vi bruker er en ice.net smartrouter som sender på 2.4GHz, hvilket gir det en maksimal rekkevidde rundt 92 meter på fri sikt siden den varierer på protokoll den bruker til å sende data. Denne rekkevidden er forventet å falle opp til 25% (LifeWire, 2019) basert på at vi ikke har en antenne på fartøyene og at mottageren som er innebygd i en RPi som sitter inne i skroget.

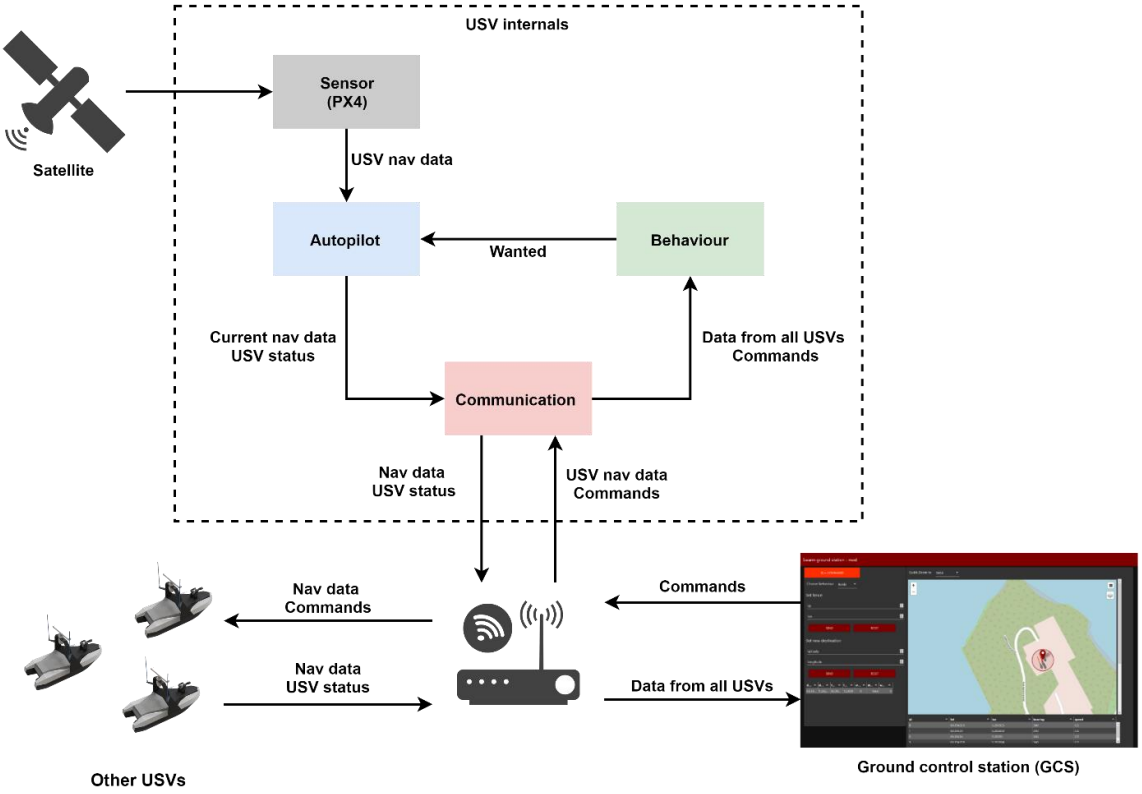
3.3 Struktur

Figur 3-1 viser et eksempel på hvordan en maritim svermplattform kan benyttes i framtiden som en sensorforlengelse til et større fartøy.



Figur 3-1: Eksempel på framtiden til maritime svermplattformer med transportfartøy (modifisert av Peck, 2016 og båter hentet fra UST, 2019)

Dette representerer overgangen fra plattformbasert krigføring til en mer sensorsentrert taktikk, der alle svermene sammen utgjør en kraftigere sensorpakke enn et fartøy kan. Dette avsnittet vil se nærmere på hvordan system må se ut og fungere for å oppfylle kravene satt i Tabell 2.



Figur 3-2: Systemstruktur av systemer i svermen (båter hentet fra Challenger Aerospace Systems, 2019)

Systemet er tre-delt; kommunikasjon, fartøysstyring og adferd. På innsiden av hver enhet går det tre parallelle prosesser. Disse er markert med farger i Figur 3-2. Prosessene går også parallelt hos alle båtene i svermen. De trenger data fra hverandre for at atferden skal fungere optimalt, men kan også operere uavhengig av andre enheter. Siden de er uavhengige blir det interne systemet fritt skalerbart fra ingen til mange enheter uten at noe må endres. Den største begrensningen på skalerbarheten til dette systemet er datatransporten mellom enheter, som gjøres over Wi-Fi via en ruter.

4 Implementering

I denne delen av oppgaven skal plattformen presenteres. Siden systemet er stort vil implementeringen starte med å presentere skroget og systemet generelt. Deretter blir hver del presentert mer i dybden.

4.1 Skroget

For vår oppgave er det av lite betydning hvilken ramme og design båten har, det viktigste for prosjektet har vært at den har plass til en RPi, et fungerende system for fremdrift og at det kan plassere nødvendige sensorer på den. Koden er det som er essensielt for oppgaven vår og virkemåten til konseptet sverm. Så fra båt til båt vil det være mindre lokale endringer, men konseptet for kommunikasjon og oppdragsløsning ville forblitt det samme. Figur 4-2 viser skroget som helhet, uten topplokk.

Figur 4-2: Skroget (Sauter, 2019

Fremst i skroget er RPi og Arduino plassert, som vist i Figur 4-1. RPi får strøm fra batteriet som er plassert midt i båten, batteriet forsyner hele fartøyet med strøm. Arduino får strøm direkte fra RPi'en gjennom en USB-kabel. Det er også en av/på bryter som styrer strømtilførselen fra batteriet. Lengre bak finner man motorer, som bruker akslinger til å styre propellene under båten. Bak batteriet blir en servo brukt til å styre tre ror, også vist i Figur 4-2.

Figur 4-1: Front skroget (Sauter, 2019)

4.1.1 Fremdrift

Fremdriftssystemet på båtene er i design helt likt. Det er tre motorer, en hovedmotor i midten og en på styrbord og en på babord side av hovedmotoren. Hver av motorene har et ror direkte akter for propellen. Dette gir et svært dynamisk styresystem, spesielt siden alle motorene er av typen som kan gå forut og akter. Alt av motorer og ror styres av et sett motor-kontrollere (OCA-150), disse får igjen

styrepulser fra en Arduino koblet til autopiloten. Denne forklares i neste avsnitt, men den sender ut styrepulser mellom 0-180 grader for å styre motorene og ror-utslaget.

4.1.2Pixhawk 4

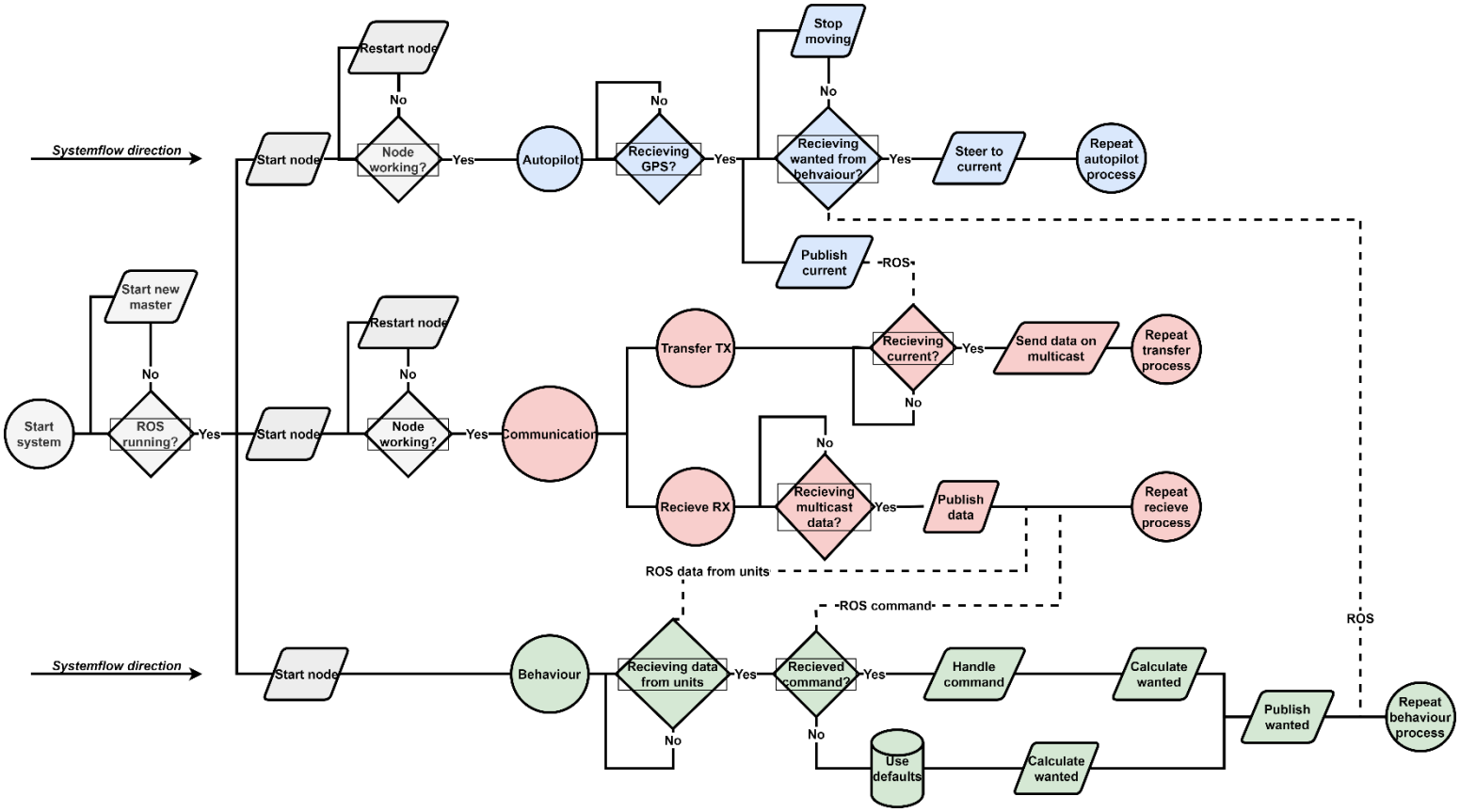
Sensorpakken som er valgt å ta i bruk er en PIXHAWK 4 (PX4). Den har en M8N GPS-modul som tillegg og er laget som en fullverdig autopilot ment til bruk på droner. Denne er vist i Figur 4-3. PX4 har mange ulike seriellporter som er tiltenkt å brukes til å kommunisere med eksterne enheter. GPS-modulen blir brukt for å hente inn GPS- og bevegelsesdata, og en telemetri radio, som også kommer med, for å opprette trådløs forbindelse med egen datamaskin. Mer informasjon om PX4 ligger i Vedlegg D – Sensorpakke Pixhawk 4.

Figur 4-3: PX4 med GPS modul (getfpv, 2019)

4.1.3 Overbygg til PX4

Den eneste modifikasjonen som er gjort på den originale båten er tillegget av et overbygg som huser PX4, vist i Figur 4-4. Grunnen til at den er laget som et overbygg og ikke direkte bygd inn i båten er for å treffe båtens rotasjonsakse. Autopiloten styrer båten på bakgrunn av de dataene den mottar fra denne sensoren, dette betyr at nøyaktigheten til autopiloten er basert på hvor gode data den får fra sensorene. De mest korrekte informasjonen om hvilken retning båten holder ligger rett på rotasjonstyngdepunktet til båten. Samtidig er sensoren gjort som et overbygg for å gi fri sikt til GPS-mottakeren som vil øke nøyaktigheten og tilgjengeligheten.

Figur 4-4: Egenprodusert overbygg til båt



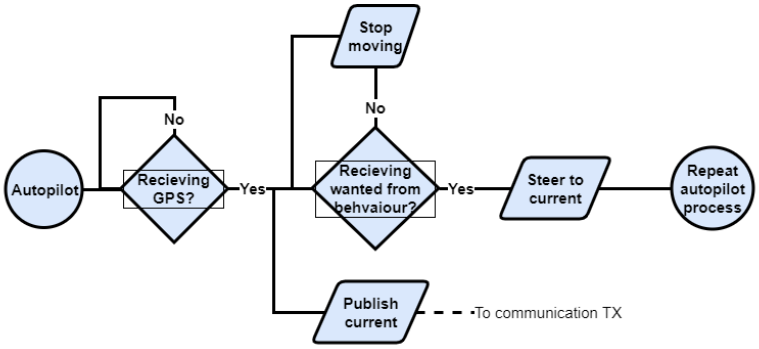
Figur 4-5: Total systemflyt for svermprogrammer i USV

4.2 Prosessflyt

Figur 4-5 viser flyten i systemet. Det er 4 prosesser som skjer parallelt og som deler data med hverandre; *Autopilot, Transfer TX, Recieve RX* og *Behaviour*. Prosessene går i en kontinuerlig loop, når de har gått gjennom blokkene og delt det de skal så begynner de på nytt til de blir stoppet eller noe uforutsett skjer, som f.eks. om sensoren skulle slutte å dele data. Prosessene drives av ROS som egne noder, om det skulle skje noe som gjør at en node krasjer uten at system er skrudd av blir disse restartet, med unntak av adferden i *Behaviour*-prosessen. Om det skulle skje en feil i behaviouren vil ikke systemet startes på nytt. Dersom behaviouren krasjer vil båten legge seg uten bevegelse siden autopiloten ikke får inn ny ønsket bevegelse fra behaviouren. Systemet må da startes på nytt og eventuelt må noe rettes i koden. Videre vil flyten i hvert enkelt program beskrives, før det blir gjort et dypdykk i implementeringskapittelet.

4.2.1 Autopilot

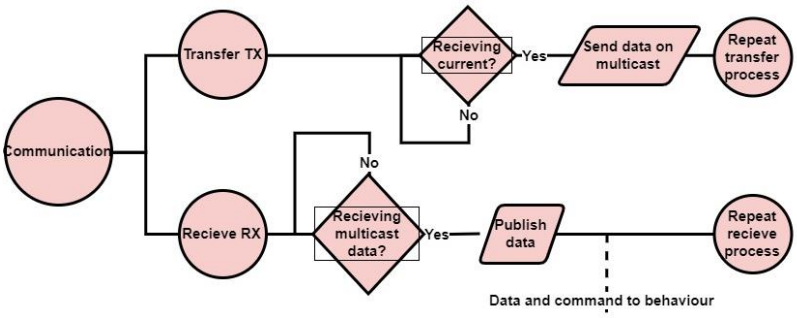
Autopiloten går gjennom en tre-delt prosess der den har behov for to typer informasjon, det er GPS data om hvor den er og hvor den beveger seg, og en ønsket bevegelse utregnet av behaviouren. Dette er vist i Figur 4-6 med de to overgangsbetingelsene der den sjekker om den har fått nødvendig data, om ikke sjekker den på nytt helt til den får dataene den trenger. Når den har fått nødvendig data gjør den kalkuleringer i PID-regulatoren for å kunne sette en ny bevegelsesvektor til aktuatorene; ror og motor. Etter at den har regnet ut nye bevegelsesvektoren begynner den hele prosessen på nytt med å sjekke om den fortsatt får dataen den trenger.



Figur 4-6: Programflyt for autopiloten

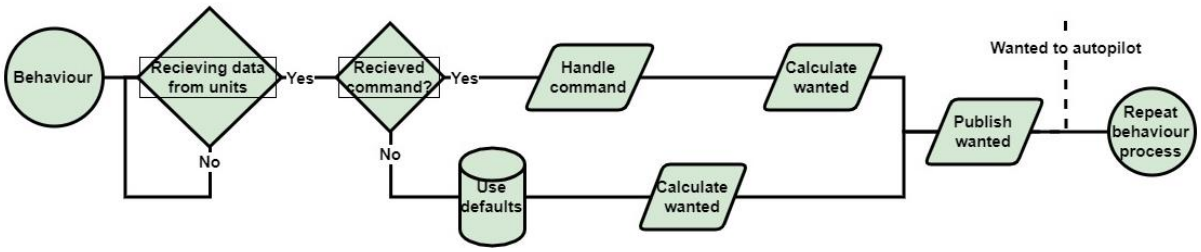
4.2.2 Kommunikasjonen

Prosessen til kommunikasjonen bygger på mye det samme som autopiloten, den har behov for data inn for å gjøre det den skal. Etter oppstart legger den seg i en loop der den kontinuerlig sjekker for ny data den kan sende, vist i Figur 4-7. TX er programmet som sender ut data til andre enheter, den venter på at data fra autopiloten skal bli publisert som den kan sende ut på multicast. Når den har sendt nyeste data publisert så går den tilbake i loopen der den venter på ny data. RX, som er programmet som leser data fra andre enheter over multicast ligger i en tilsvarende loop og venter på meldinger på nettet. Når den får en melding så konverterer den dataene og publiserer de på topic'er som behaviouren skal bruke. Når dataene fra en melding er publisert går den tilbake til å vente på ny data.



Figur 4-7: Programflyt for kommunikasjon

4.2.3 Behaviouren



Figur 4-8: Programflyt for behaviour

Behaviouren sin flyt skiller seg litt fra de andre programmene, der autopiloten og kommunikasjonen venter helt til de har fått nødvendig data, har behaviouren et sett med default-data den kan bruke om den ikke får en kommando umiddelbart, dette vises i nederste linje av Figur 4-8. Disse dataene definerer et sett med grunnverdier så enheten kan samhandle med svermen uten kommando fra en base. Ved en eventuell kommando fra basen vil den gamle behaviouren som kjører bli slettet og en ny behaviour startes basert på kommandoene gitt.

4.2.4 Eksterne programpakker

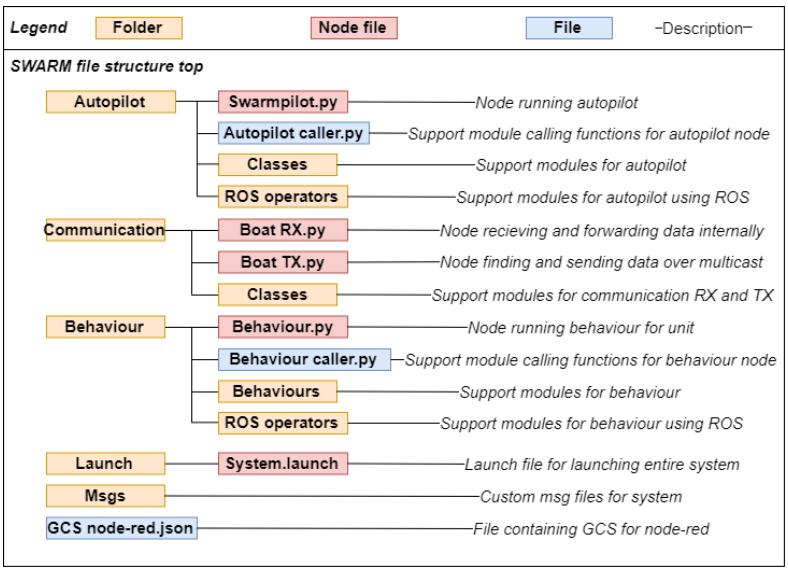
For å få systemet til å fungere slik det er laget kreves det en del eksterne programpakker og programvare. Dette inkluderer installasjonen av ROS og noen ekstra programmoduler til Python. RPi'ene som brukes kjører operativsystemet Ubuntu MATE informasjon om dette ligger i Vedlegg A - Operativsystem på RPi. Det er ellers ikke gjort noen endringer i operativsystemet. ROS installasjonen er også uten modifikasjoner, med unntak av pakkene som oppretter i ROS arbeidsrommet, informasjon om dette og hvordan laste ned og starte ROS som i systemet vårt står i Vedlegg C – ROS installasjon og bruk. Utvidelsen MAVROS blir også lastet ned for å kommunisere med sensoren vår, dette er beskrevet i samme vedlegg sammen med hvordan man kan skrive egne msg- og launch-filer.

Python 3-moduler som er brukt er: *pyfirmata*, *psutil*, *p5* og *rospy*. Med unntak av den siste modulen er disse å installere fra apt-get eller pip-bibliotekene, en prosess nøye beskrevet i Vedlegg G – Tilleggsmoduler til Python. Rospy er en modul som følger med installasjonen av ROS. Til slutt brukes QGroundControl (QGC) til kalibrering av PX4

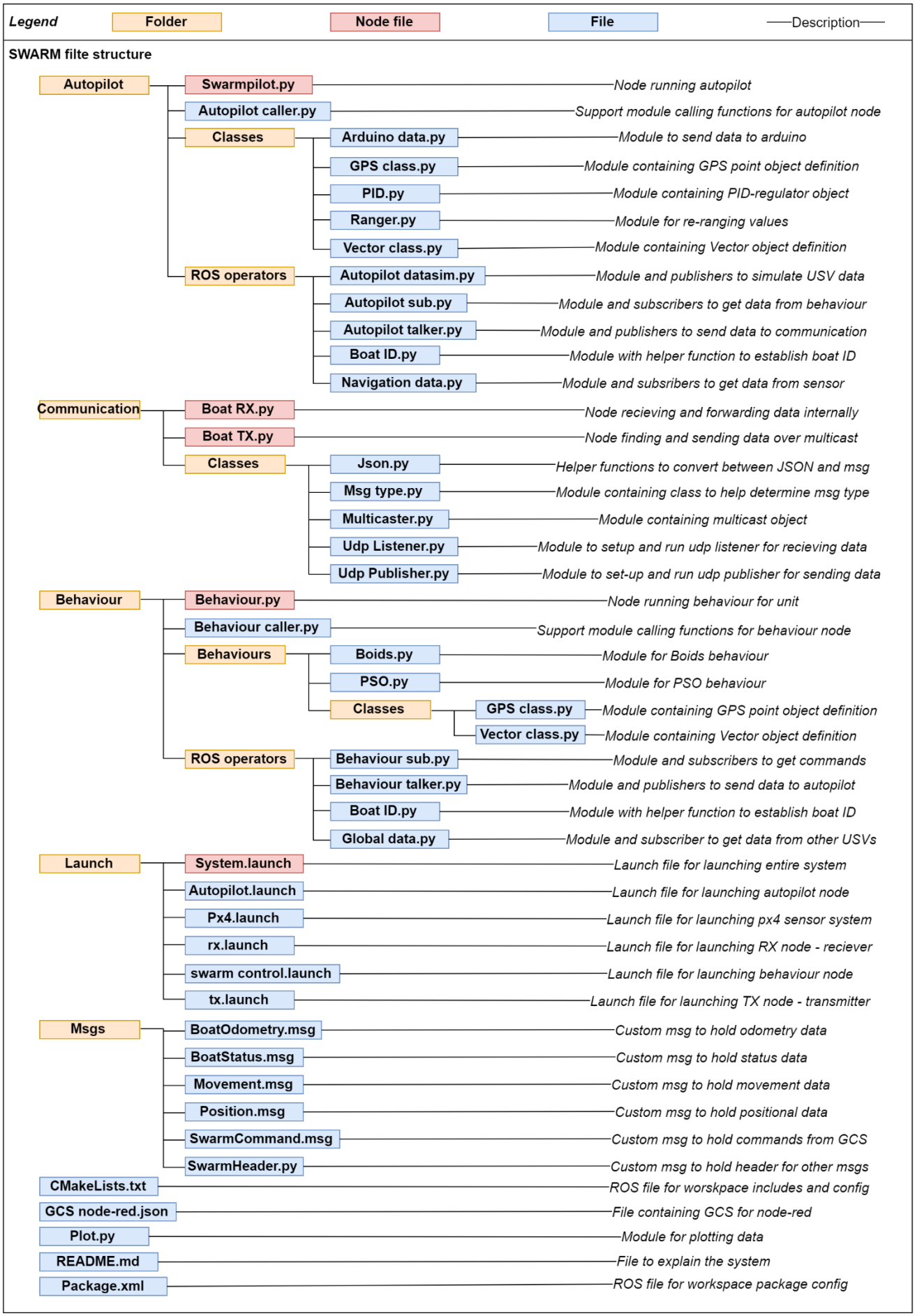
kompasset og testing av MAVROS-data. Mer informasjon om bruk av QGC er beskrevet i Vedlegg F – QGroundControl. Bruk av QGC vil bli sett nærmere på i 5.2.4.2

4.3 Filstruktur

Filstrukturen til svermimplementeringen er delt inn i 5 hoved-mapper og vist på topp-nivå i Figur 4-9 og i sin helhet i Figur 4-10. Hver av mappene inneholder forskjellige deler av systemet som i seg selv også er uavhengige. Launch- og Msg-mappen inneholder konfigurasjons-filer for ROS som er til oppstart av systemet og definering av egne meldingstyper i msg-formatet til ROS. Det er *Autopilot*, *Communication* og *Behaviour* som inneholder hoved-funksjonaliteten i systemet. Hver av disse mappene inneholder programmene og undermodulene som brukes i hvert av programmene. I kodingen er det valgt å ta en modulær løsning på bakgrunn av at det er så mange filer i systemet. I hver mappe er det kun en hovedfil, markert i rødt, som bruker deler fra de andre filene, markert i blått, som er moduler eller støttefunksjoner for at programmet skal fungere. Dette gir systemet en verdifull fleksibilitet når det kommer til å modifisere det på et senere tidspunkt eller i løpet av prosessen, som er et av kravene og målsetningene for plattformen. Eksempelet her er i behaviouren, om programmet ønsker å bytte mellom to forskjellige atferder er det bare å hente en modul for den behaviouren og slette det gamle objektet. Dette gjør også at man kan legge til så mange atferder man vil i systemet uten at programmet for å kjøre behaviour endrer seg nevneverdig.



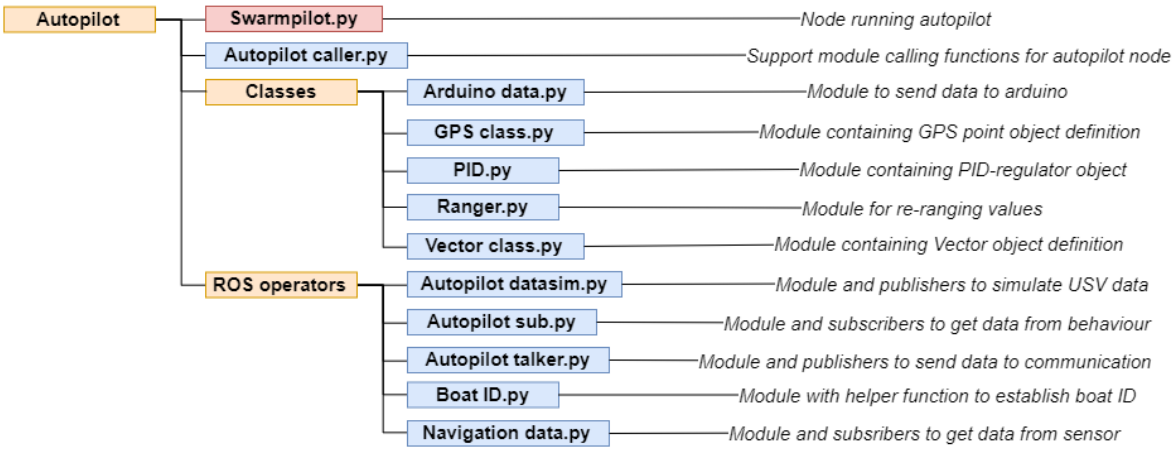
Figur 4-9: Oversiktskart over filstruktur for oppgaven



Figur 4-10: Totaloversikt over filstruktur i oppgaven

4.4 Autopilot

Autopiloten har viktig rolle i en sverm-plattform. Den er ansvarlig for at USV'ene i svermen beveger seg riktig etter det som beregnes i behaviouren. Autopiloten er vektorbasert. Vektorene blir håndtert i polarform, dvs. at vektoren inneholder en lengde og en vinkel. Alle vinkler i autopiloten og resten av systemet er gitt i forhold til nord, fordi at dataene sensorene sender inn er i dette formatet. Dette gir minst mulig omregninger og forsterker bruken av et felles koordinatsystem for hele systemet som beskrevet tidligere. Som alt annet er autopiloten modul og fasebasert; først tar den inn nødvendig data, før den så bruker dataene i en PID-regulator der den kalkulerer hvilket utslag som må gjøres. Til slutt omregner den det kalkulerte utslaget til et format som kan leses av Arduino, som gjør et motor- og ror-utslag. Funksjonene utføres av moduler og klasser som hovedprogrammet, kalt *Swarmpilot.py*, kaller på og bruker. Strukturen er vist tidligere, men Figur 4-11 viser den kun for autopilot mappen, her er noden som kjører autopilot programmet markert i rødt – *Swarmpilot.py*.



Figur 4-11: Filstruktur for autopilot

Som nevnt tidligere i 4.2.1, er styringen en tre-delt prosess som starter med data-innhenting til regulatoren. PID-regulatoren trenger 2 typer informasjon, det er den nåværende bevegelsen den har og hvor behaviouren ønsker at båten skal bevege seg.

Slik det er kodet kan den ta inn enten en ønsket vektor eller et GPS-punkt. Om det skulle være sistnevnte som sendes fra behaviouren så gjøres det om til en vektor ved hjelp av en undermodul som regner avstand og relativ vinkel mellom to GPS-punkter, basert på formelen *Haversine* formelen som vist i Figur 4-12.

$$a = \sin^2(\Delta\varphi/2) + \cos\varphi_1 \cdot \cos\varphi_2 \cdot \sin^2(\Delta\lambda/2)$$

$$c = 2 \cdot \text{atan2}(\sqrt{a}, \sqrt{(1-a)})$$

$$d = R \cdot c$$

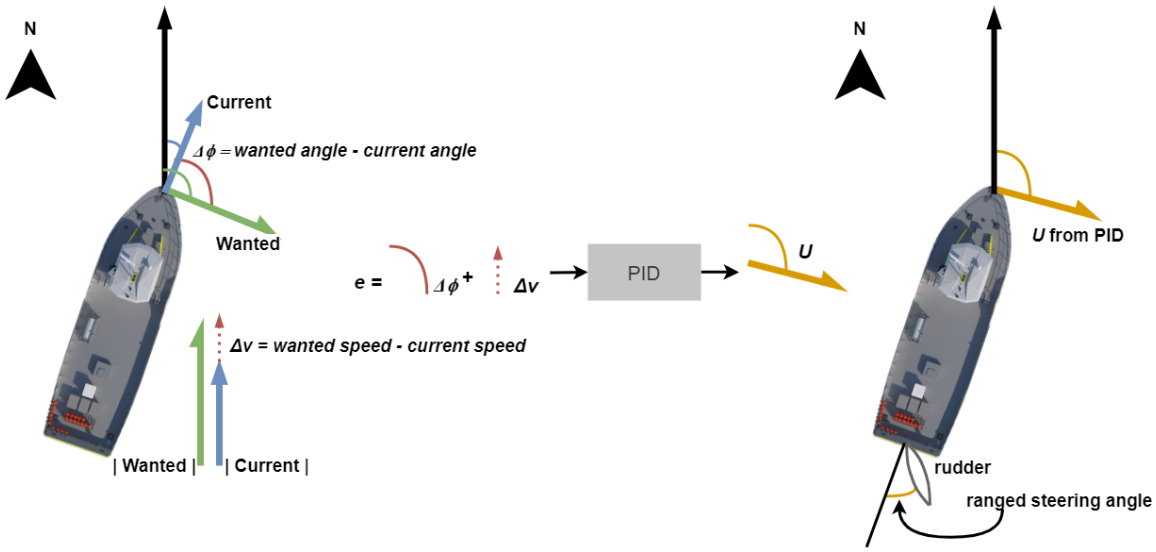
φ is latitude, λ is longitude, R is earth's radius (mean radius = 6,371km),

Figur 4-12: Haversine-formel for avstand mellom to GPS punkter (Veness, 2019)

Denne formelen tar inn to GPS-punkter på desimal form og gir avstanden. For å få vinkelen brukes formelen som på fagspråket heter *"the spherical law of cosines": d =* $\text{acos}(\sin\varphi_1 \cdot \sin\varphi_2 + \cos\varphi_1 \cdot \cos\varphi_2 \cdot \cos\Delta\lambda) \cdot R$. I koden baseres bruken av denne modulen med nåværende posisjon som startpunkt og dit den skal som sluttpunkt. Dette gir en vinkel relativ til nord som styrker det felles koordinatsystemet. Når autopiloten har de to dataene og er fornøyd med dem, brukes de som argumenter i PID-regulatoren.

4.4.1 Styringen

Styringen vår er en egen modul basert på en PID-regulator. Det er en reguleringsalgoritme som brukes for å regulere elektriske og mekaniske komponenter for å stabilisere pådraget. I vårt tilfelle brukes den til å oppnå en ønsket bevegelsesvektor som inneholder en fart og



Figur 4-13: Reguleringsprosessen som helhet for båten med data inn og ut

retning. Figur 4-13 viser prosessen i tre steg ved at regulator-modulen får inn data om

båtens bevegelse og regner det om til et vektor-utslag som realiseres av motor og ror. Regulatoren baserer seg på formelen vist i Figur 4-14, Dataene inn og ut er *e* og *U* respektive. *Kp, Td* og *Td* er konstanter for de forskjellige leddene som kan tilpasses for spesifikke applikasjoner og tregheter utslag. Dataene inn til regulatoren - nåværende og

$$U = K_p \cdot e + \frac{K_p}{T_i} \int_0^t e\,dt + K_p \cdot T_d \cdot \frac{de}{dt}$$

Figur 4-14: PID-formelen (Wikipedia, 2019)

ønsket bevegelse blir regnet om til en vektor *e* som vist i Figur 4-15. *e* er differansen mellom hvor den er nå og hvor den skal være, dvs. avstanden mellom ønsket og nåværende både for fart og vinkel, som vist tidligere i Figur 4-14 og forklart i detalj i Figur 4-15.

Figur 4-15: Visuell fremvisning av data inn og ut fra regulator

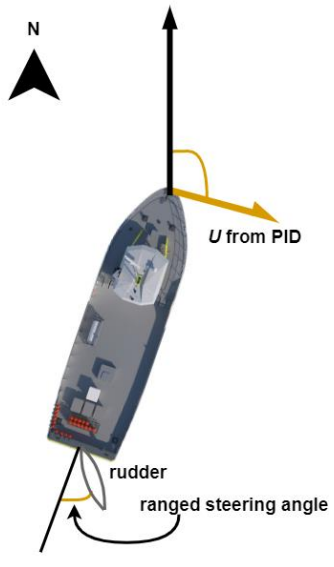
U er også gitt som vektor med fart og vinkel, og representerer er utslaget som skal settes til motor og ror. *U* regnes som summen av 3 ledd, derav PID-notasjonen, som står for *Proporsjonal-*, *Integral-* og *Derivatledd.* I vårt tilfelle er det gjort små endringer på hvordan *U* gis ut. *U* representerer forskjellen på ønsket og nåværende, fra formelen er den et utrykk for endringen som må skje på utslaget. For vinkelen er dette rett fram å sette vinkelen i *U*, men farten representerer en endring som må legges til det nåværende utslaget. Derfor er det laget et ledd i koden for å legge sammen nåværende fart og ønsket endring vist på linje 55 i Figur 4-16. I samme figur vises hvordan det er satt fart og vinkelutslagsendring for regulatoren. Dette er for at rorene ikke skal slå inntil hverandre og båten skal holde seg innenfor kontrollerbar fart.

```
52          pid = self.P_value + self.I_value + self.D_value
53
54          #linearity fix
55          pid.magnitude = current_vector.magnitude + pid.magnitude
56
57          #max speed fix
58          if pid.magnitude > self.pid_max.magnitude:
59              pid.magnitude = self.pid_max.magnitude
60
61          elif pid.magnitude < -1.0*self.pid_max.magnitude:
62              pid.magnitude = -1.0*self.pid_max.magnitude
63
64          if pid.angle > self.pid_max.angle:
65              pid.angle = self.pid_max.angle
66
67          elif pid.angle < -1.0*self.pid_max.angle:
68              pid.angle = -1.0*self.pid_max.angle
69
70          return pid
```

Figur 4-16: Kodeutklipp for linearitet og begrensning av data ut fra regulator (fra PID.py)

I en PID-regulator er det proporsjonalleddet som står for det største utslaget i regulatoren, grovstyringen. De to andre leddene er for mindre justeringer. Det er fordi regulering med kun P-leddet ofte ikke når opp til ønsket verdi eller svinger om den som en sinus-funksjon. For å løse disse to problemene og treffe ønsket verdi raskere brukes I- og D-leddet. I-leddet blir større jo lengre tid det går uten at den har nådd ønsket verdi. Med tiden vil den presse regulatoren opp på ønsket nivå om kun P-leddet ikke har klart det selv. Så med en PI-regulator er det garantert at man når ønsket verdi til slutt, da mangler det bare noe som begrenser svingninger. Det er D-leddet som er siste brikken i regulatoren vår, den sørger for å dempe svingninger siden den representerer stigningstallet. Kombinert utgjør disse leddene en fullverdig PID-regulator, men i vår implementering er det hakket mer komplisert med tilpasning og bruk av verdier over tid. Det er fordi ønsket bytter verdi omtrent like hyppig som regulatoren utregner utslag, det vil si at den som regel ikke vil få tid til å verken svinge eller nå målet helt. Det er kun gjennom testing man kan bestemme hvor mange av leddene som er nødvendige og hvilke verdier konstantene må ha. Målet vårt er å holde den så enkel som mulig, med færrest mulig variabler å justere.

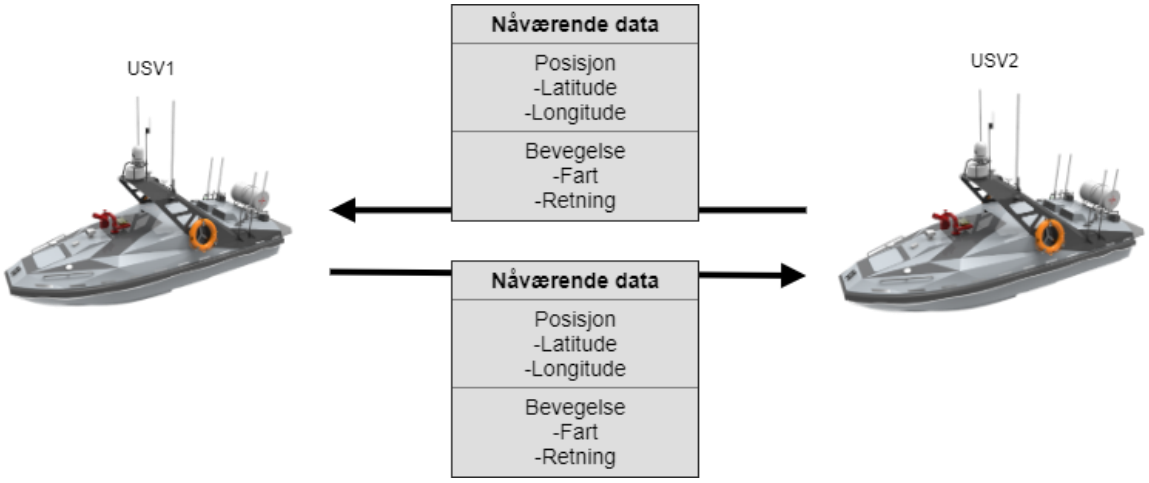
Med det fremdriftssystemet båtene er utstyrt med fra tidligere med 3 separate motorer og ror er mulighetene for en nøyaktig og hurtig autopilot stor. Det er mulig å få kraft fra motorene begge veier, på den måten er rygging av båten fult mulig med riktig design av autopiloten. Den kan også kjøre motorene med forskjellig kraftretning, babord bak og styrbord frem, dette hadde gitt fartøyet en enormt liten svingradius. Autopiloten er ikke den største prioriteringen for å utvikle en god plattform for konseptvising. Det er andre ting som også må fungere som kommunikasjonen og adferden som faktisk styrer hvor båten skal. Derfor er det ønsket å utvikle en så enkel autopilot som mulig. En viktig del av at den skal være enkel, er at den gir samme pådrag til motorene og rorene samtidig. Dette gir oss en konfigurasjon som vist i Figur 4-17, med én motor og ett ror. Dette skal i teorien være godt nok, men gir oss noe dårligere sving-radius enn andre konfigurasjoner siden rorets maksimale utslag er 45 grader begge retninger.



Figur 4-17: Ror-konfigurasjon for styring av båt fra autopilot

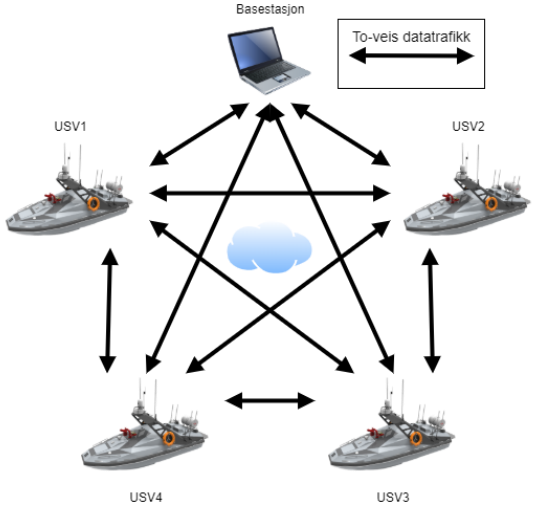
4.5 Kommunikasjon

I en plattform som baserer seg på sverm er kommunikasjon og informasjonsdeling en av de viktigste brikkene. Selv om dette ikke testes hyppig, er dette noe man må kunne stole på at fungerer uavhengig av andre faktorer hver gang systemet starter. Det er en bakgrunnsprosess, men dog ansvarlig for at alt annet fungerer. Det er mye data som skal fram og tilbake mellom båtene for at svermen skal fungere. Figur 4-18 viser den viktigste formen for kommunikasjon i svermen, deling av bevegelses data mellom båtene.



Figur 4-18: Deling av bevegelses-data mellom USV'er i svermen (USV hentet fra UST, 2019)

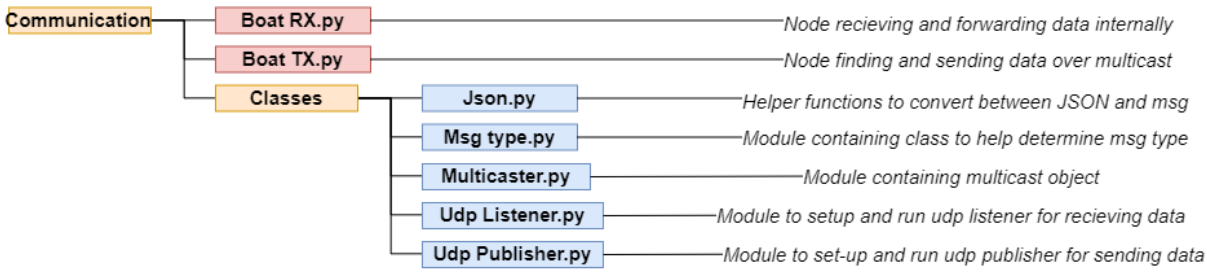
Denne kommunikasjonen skjer mellom hver USV i svermen kontinuerlig med en frekvens rundt 10 ganger i sekunder. Derfor er det viktig som beskrevet i innledningen til dette avsnittet at kommunikasjonen er effektiv og robust. Den enkleste måten for oss å implementere denne kommunikasjonen mellom båtene var å benytte oss av en ruter for all trådløs kommunikasjon. Dette er noe som kan skape utfordringer i det lange løp med tanke på rekkevidde og antall enheter det er plass til i svermen, men det er klart den raskeste løsningen. For å implementert noe som kan viser konseptet ble ruter og Wi-Fi til intern kommunikasjon, som vist i Figur 4-19, den beste løsningen.



Figur 4-19: Trådløse forbindelser i sverm med 4 USV'er og en base (USV hentet fra UST, 2019)

Figur 4-19 viser hvordan kommunikasjonen skjer mellom enheter i svermen i stor skala. Det er verdt å legge merke til at hver enkelt enhet har like mange trådløse interaksjoner gjennom ruteren som det er enheter utenom den. I et slikt oppsett representerer skyen ruteren som sprer ut alle meldingene som sendes fra hver båt over multicast. I dette eksempelet med 4 båter og en base hadde det blitt 5*4, *20* unike interaksjoner gjennom ruteren, med frekvens rundt 10Hz. I teorien skal dette ikke være for mye trafikk for verken ruteren eller hver enkel RPi, men trafikken øker med $O(N^2)$ for hver enhet som legges til i systemet.

Figur 4-20 er oppsettet for koden som løser kommunikasjonen. Det er to hovedprogrammer som tar seg av hver sin del av kommunikasjonen; Boat – *RX* og *TX*. *RX* og *TX* står for *recieve* og *transmit* henholdsvis, og er forkortelser mye brukt i RC industrien. Det er da naturlig at *RX* programmet er det som mottar meldinger fra multicast og sprer det til andre delsystemer over ROS, mens *TX* henter data fra aktuelle delsystemer og sender det ut som multicast-meldinger. Oppgaven til begge programmene er å sørge for at dataflyten inn og ut fra USV'en går i orden. Videre skal det ses nærmere på hvilke data og underfunksjoner de bruker.



Figur 4-20: Filstruktur for kommunikasjonsmappen

4.5.1TX - senderen

Senderen skal lese data ut fra topic'er i ROS, tolke og konvertere dataen til JSON, og til slutt overføre disse med multicast til de andre enhetene i svermen. Selv om det er *Boat_TX.py* som er hoved-programmet benytter den seg av undermodulene fra mappen *Classes.py*; *Json.py*, *Multicaster.py* og *Udp_Publisher.py*. Siden koden er så modulær og bygget opp av flere klasser er det eneste koden i TX gjør å definere variabler og verdier

```
29    # defining variables for udp publisher
30    mcast_grp  = rospy.get_param('~mcast_addr', "225.0.0.25")
31    mcast_port = rospy.get_param('~mcast_port', 4243)
32    compress   = rospy.get_param('~compression', False)
33    nav_hz     = rospy.get_param('~nav_hz', 10.0)
34    state_hz   = rospy.get_param('~state_hz', 1.0)
35    cpu_hz     = rospy.get_param('~cpu_hz', 1.0)
36
37    ttl = rospy.get_param('~ttl', 1) #ttl - time to live for socket
38
39    #Definition of topic adresses for data to send
40    odometry_topic    = rospy.get_param('~odometry_subscriber',      "/autopilot/current")
41    status_topic      = rospy.get_param('~status_subscriber',    "/autopilot/status")
42
43    #Initialization of publisher socket
44    publisher = PositionPublisher(mcast_grp, mcast_port,
45            ttl=ttl, compress=compress, nav_hz = nav_hz, state_hz=state_hz,
46            cpu_hz=cpu_hz)
47
48    #initialization of subscribers
49    rospy.Subscriber(odometry_topic,  BoatOdometry, publisher.handle_odometry)
50    rospy.Subscriber(status_topic,    BoatStatus,   publisher.handle_boat_status)
```

Figur 4-21: Kodeutklipp som viser definering og initiering av TX node (fra boat_TX.py)

for multicast-socketen og hvilke topic'er den skal lese data fra. Det er *Udp_Publisher.py* som igjen gjør innhentning av data fra ROS og konvertering gjennom programmet *Json.py*. *Udp_Publisher.py* lager en multicast-socket ved hjelp av støtteklassen i *multicaster.py* som da er klar for å sende data så fort det blir publisert noe på topic'ene den følger i ROS. Disse topic'ene er definert i TX som vist kodeutklippet fra Figur 4-21.

Her defineres topic'ene som skal abonneres på, konverteres og sendes. Alle variablene som brukes til denne prosessen defineres også her, som hvilke multicast port som skal brukes, hvor ofte data skal sendes ut – *x_hz*, og hvor lenge porten skal være åpen før timeout – *ttl*.

Samtidig initieres multicast-socketen på linje 41, det benyttes også funksjoner fra denne modulen for å konvertere og sende data når det kommer nytt på topic'ene. Konvertering av dataene må gjøres fra msg-formatet i ROS til noe som kan brukes på tvers av plattformer. Basestasjonen kjører ikke ROS, så dataene må derfor konverteres til et format som er leselig for den. Derfor har vi valgt å konvertere dataene fra ROS-msgs til JSON. Slik koden er bygd opp kan vi eller andre i senere tid endre hvilket format man ønsker å sende data på, om noe kan man bruke flere formater for forskjellige typer data. For å bytte format må man bare bytte filen *Json.py*, med en annen konverterings fil som er bygd opp riktig og så importere den til *Udp_Publisher.py*. For vår del var JSON et naturlig valg siden det er såpass likt i oppbygningen som ROS sitt msg-format, hvilket gjør konverteringen enkel og effektiv. De konverterte dataene blir så sendt over multicast til alle andre enheter i svermen, der de blir konvertert og spredd ut i systemet med *RX*-programmet.

4.5.2 RX – Mottakeren

Mottakeren tar inn data som blir sendt ut på multicast, konverterer det fra JSON til msg og så publiserer den dataen på topic'er som nyttes av andre programmer i systemet. Den største forskjellen på de to programmene er at senderen kan sende all data uten å bry seg om hva det er, mens mottakeren må kunne lese ut og velge riktig konvertering for riktig melding, siden strukturen i meldingene og topic'et de skal publiseres til er unike for meldingstypen. Det er i vårt system 3 forskjellige meldingstyper som enhver USV må kunne motta og tolke; status – *BoatStatus,* bevegelsesdata – *BoatOdometry og* kommandoer – *SwarmCommand.* Måten den tolker de forskjellige er at alle meldingstyper inneholder en tall-verdi som heter *msg-type*, denne blir sammenlignet med en klasse som ligger i *Msg_type.py* filen vist i Figur 4-22. Basert på hvilke verdier som er like velger programmet riktig funksjon for å konvertere og publisere dataene vist i Figur 4-23.

```
3  class MsgType(IntEnum):
4      '''Helper class method to enumerate message types to reciever'''
5      HEARTBEAT      = 0
6      ODOMETRY       = 1
7      SWARM_COMMAND  = 2
8      BOAT_STATUS    = 3
```

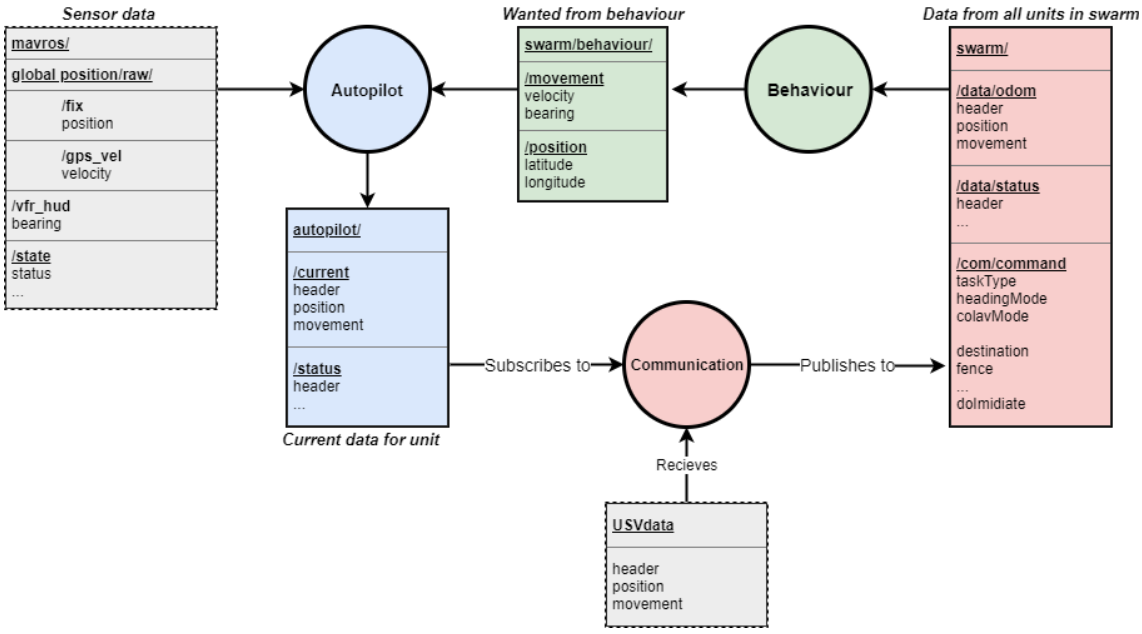
Figur 4-22: tallverdier til meldingstyper i *msg-type*

```
56          '''Continuos run process for Udp listener on multicast'''
57          while not rospy.is_shutdown():
58              try:
59                  msg = self._listener.listen()
60                  header = self._readHeader(msg)
61
62                  if header.msgType == MsgType.ODOMETRY:
63                      self._publishOdometry(msg)
64                  if header.msgType == MsgType.BOAT_STATUS:
65                      self._publishStatus(msg)
66                  if header.msgType == MsgType.SWARM_COMMAND:
67                      self._publishSwarmCommand(msg)
68              except socket.timeout:
```

Figur 4-23: Kodeutklipp for sjekk av meldingstype og valg av funksjon (fra fil Udp_Listener.py)

4.5.3 Intern kommunikasjon – datadeling

Som beskrevet i systemstrukturen er den interne programstrukturen høy-modulær. Når det kommer til kommunikasjonen mellom programmer gjør alt dette gjennom ROS. Det er tre hovedprogrammer som kjører i hver USV parallelt; *Autopilot, Communication* og *Behaviour*. Disse tre kommuniserer utelukkende over ROS topic'er, som vist under i Figur 4-25.



Figur 4-25: Dataflyt internt i USV over ROS-topic'er

Hvert av programmene har flere dedikerte topic'er de publiserer data til, disse er representert med den tilsvarende fargede blokken til programmet. Kommunikasjonen går i en retning gjennom hele systemet, det betyr at alle programmene er avhengige av hverandre for å fungere optimalt. På bakgrunn av dette er det lagt inn flere funksjoner i koden sånn at enkeltdelene ikke skal kaste feil og stoppe om de ikke får data inn i starten. I autopiloten gjøres det ved at båten gir seg selv en ønsket bevegelse lik 0 om den ikke skulle få noen ønsket bevegelse fra behaviouren, vist ved koden i Figur 4-26. Her sjekker autopiloten om den mottar data fra behaviouren i linje 60. Om den mottar data så henter autopiloten ut nyeste ønskede fart og setter den som *wanted* i linjene 61-65.

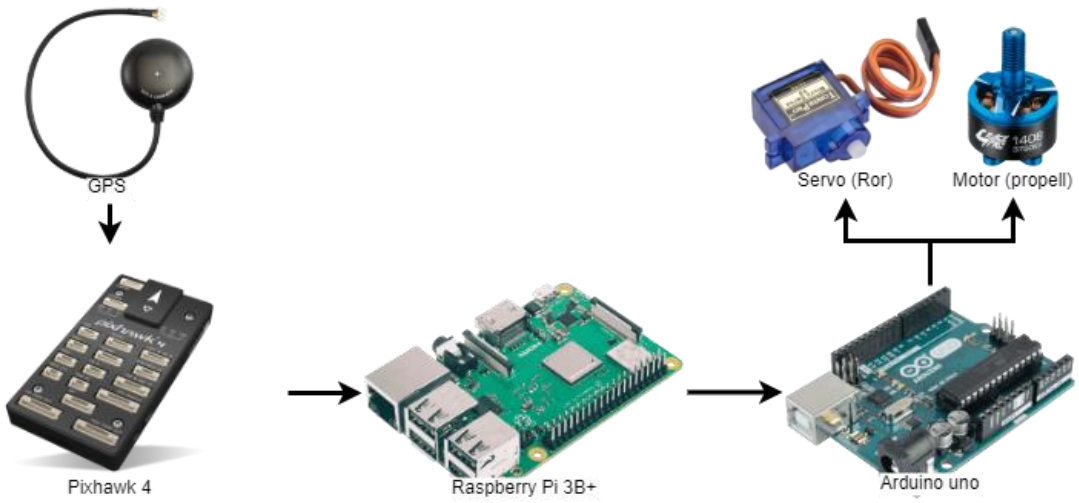
```
56          try:
57              current_GPS    = nav.get_GPS()
58              current_vector = nav.get_Vector()
59
60              if behaviour.is_recieving(): # check if
61                  wanted = behaviour()
62                  if isinstance(wanted, GPS):
63                      wanted_vector = current_GPS.calculate(wanted)
64                  else:
65                      wanted_vector = wanted
66              else: #if not recieving from behaviour stop USV
67                  print("did not recieve")
68                  wanted_vector = Vector(0.0, 0.0)
69
70              autopilot.set_wanted_vector(wanted_vector)
```

Figur 4-26: Hvordan autopiloten løser mangel på ny data

Kommunikasjons-nodene gjør ingenting med mindre de får data inn, enten gjennom topic'er eller fra multicast lytteren. Om de ikke får noe fra autopiloten så venter den bare. Behaviouren er litt spesiell fordi den har en liste med data som den fyller ettersom kommunikasjonen oppdaterer topic'er med data fra andre enheter i svermen, om det ikke skulle være noe data der har behaviouren en tom liste med data. Samtidig vil den tro at den er i posisjon lat:0 og lon:0, og derfor utenfor den geografiske begrensingen – *fence*, svermen har. Derfor vil den hvis den ikke får data inn fra kommunikasjonen sende en ønsket fart og retning mot sentrum av begrensningen. Dette er den største utfordringen med intern kommunikasjon i svermen, men den er forårsaket av at autopiloten ikke får data fra sensoren sin. Uten data inn fra sensoren vil svermen ikke fungere uansett, som er et følgeproblem som ikke løses i koden.

4.5.4 Perifer sensorkommunikasjon

Kommunikasjon med perifere sensorer gjøres på vår plattform gjennom kablet kommunikasjon. Det er fordi alle sensorene som benyttes befinner seg i eller på fartøyet. Dette minsker kablet kommunikasjon behovet for trådløs båndbredde, enten det er Wi-Fi eller Bluetooth for eksempel. Viktigheten av å minske båndbredde bruken av sensorer på Wi-Fi er stor, med tanke på at alt av data fra hvert fartøy i svermen sendes over Wi-Fi også. Dette betyr dog ikke at det er umulig å dele sensor data internt for fartøyet trådløst, i framtiden vil dette kanskje bli en realitet. Sensorene og aktuatorene som skal benyttes i hvert fartøy er sett på og konkludert med at kablet datakommunikasjon er den beste løsningen. Det er også mer stabilt enn trådløst etter våre erfaringer med dagens teknologi. I Figur 4-27 vises hvordan dataene går fra sensor til aktuatorer.

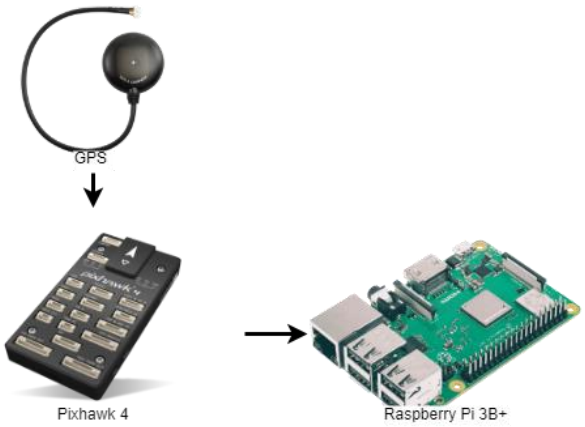


Figur 4-27: Total oppsett for perifer kommunikasjon med sensorer og aktuatorer

4.5.4.1 SENSORER

Sensoren vi har valgt å bruke i hvert fartøy er en Pixhawk 4, som beskrevet i 4.1.2. Denne har flere seriell-porter som er tiltenkt å brukes til samhandling med eksterne enheter som den inkluderte GPS-sensoren som vi benyttet oss av, men også for samhandling med en RPi som i vårt tilfelle. Vi valgte å ikke bruke seriell-portene på Pixhawk'en, men derimot ta i bruk USB kabelen som direkte kobles inn i micro-usb inngangen på Pixhawk'en. Denne er tiltenkt testing og annen bruk, og er sterkt frarådet fra produsentene å bruke aktivt, men vi valgte å benytte oss av denne til tross. Det er fordi vi kun skal lese verdier

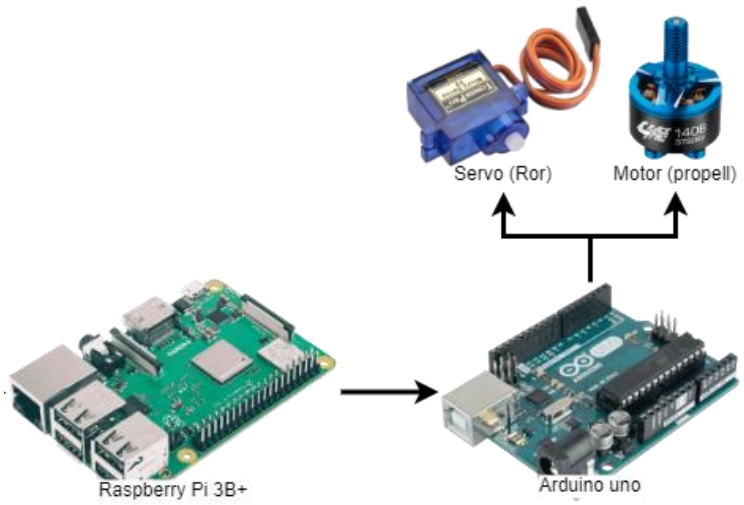
ut fra ROS som Pixhawk'en selv publiserer. Derfor er det i teorien et minimum av kommandoer som sendes på tvers av systemet. Det kan sees på som enveiskommunikasjon fra Pixhawk'en til diverse ROS-topic'er også videre fra disse til aktuelle programmer i RPi'en etter behov, vist i Figur 4-28. Alle programmer kan kjøre uavhengig av data fra Pixhawk'en, de gir feil resultat, men en time-out i noen sekunder krasjer ikke svermen. Derfor er vi godt fornøyd med denne løsningen.



Figur 4-28: Data inn fra sensor

4.5.4.2 ARDUINO

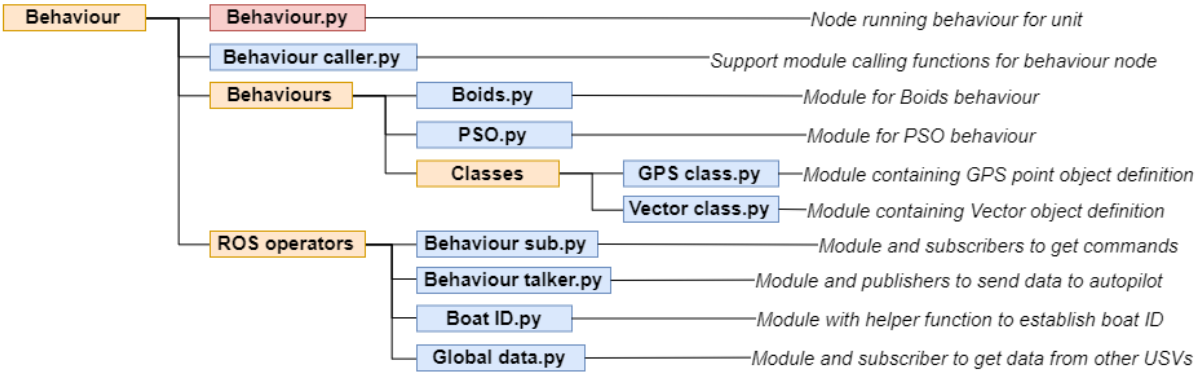
Data fra RPi til Arduino går over seriell-port med en USB-B, det er i likhet med kommunikasjonen mellom Pixhawk og RPi, stort sett en enveiskommunikasjon, visualisert i Figur 4-29. Det blir sendt data fra RPi'en til et Arduino brett som tolker denne og sender videre PWM signaler til aktuelle aktuatorer; servo for ror og motor for fremdrift. På Arduino brettet kjører et av programmene fra dets standard-bibliotek kalt StandardFirmata.



Figur 4-29: Data ut til aktuatorer

4.6 Behaviour

Hjernen i svermen er adferds-styringen, som gjøres i *behaviour* programmet. I korte trekk tar *behaviour* inn informasjon om de andre båtene i svermen, gjør utregninger på bakgrunn av avstanden til disse båtene og deres fart og retning, lager en vektor basert på utregningene og sender den til autopiloten. Autopiloten bruker så vektoren til å gjøre en kursendring. Det som er viktig er at utregningene som blir gjort er riktige, men for at det skal skje må en del ting være på plass. Figur 4-30 er en oversikt over alle filene som brukes for å kjøre hovedprogrammet *Behaviour.py*.



Figur 4-30: Filstruktur i behaviour mappen

Som nevnt i 4.5.2 sender kommunikasjonen bevegelsesdata til adferden. Denne informasjonen blir plukket opp av *Global_data.py* og lagret i en tabell. Når en tabell opprettes av dataene fra multicast-data som vist i Figur 4-32, oppfyller det samtidig kravet om at systemet skal være skalerbart, gitt i Tabell 2. Det gjør den fordi tabellen har plass til så mange båter vi definerer, men kan ta data fra færre enn det og fungere. For å bruke

```
8   class swarmData:
9       '''Helper class to continously keep an updated list of other boats'''
10      def __init__(self):
11          '''Initialises subscriber to data from multicaster'''
12          self.list_global = [BoatOdometry()] * (BOATS_IN_SWARM)
13
14          rospy.init_node('behaviour', anonymous=True)
15
16          topic_odometry = "/swarm/data/odom"
17
18          rospy.Subscriber(topic_odometry, BoatOdometry, self._update)
19
20      def _update(self, data):
21          try:
22              ID = data.header.id
23
24              self.list_global[ID] = data
25
26              self._get_time_since(data.header, ID)
27
28          except IndexError:
29              pass
```

Figur 4-32: Kodeutklipp der det lages en tabell med data fra USV'ene i svermen. (fra fil global_data.py)

```
106     def _make_list(self, dataObj):
107         clist = []
108
109         for i in range(len(dataObj)):
110             if i == self.boat_id:
111                 pass
112             elif dataObj[i].position.latitude == 0.0 and dataObj[i].position.longitude == 0.0:
113                 pass
114             else:
115                 dist = self._get_distance(dataObj[i].position)
116                 x, y = self._get_xy(dist)
117                 clist.append({"speed" : dataObj[i].movement.velocity,
118                               "bearing" : dataObj[i].movement.bearing,
119                               "distance": dist.magnitude,
120                               "relative": dist.angle,
121                               "x"       : x,
122                               "y"       : y})
123         for i in range(len(clist)):
124             print"data from boat ", i
125             print(clist[i])
126         print"elements in BOIDS list: ", len(clist)
127         return clist
```

Figur 4-31: Omregnet data satt i ny tabell og klar for bruk i ulike atferder. (fra filbehaviour_caller.py)

dataene som kommer inn blir det gjort en omregning av GPS-koordinater til x og y koordinater der båten som gjør utregningene er plassert i (0,0). Med dette regner den ut avstanden, både vinkel og størrelse. Verdiene blir så satt inn i en ny tabell, vist i Figur 4-31, som videre brukes til å gjøre utregninger i atferds-koden som er valgt.

4.6.1 Kommandobehandling

Hver USV må ha muligheten til å motta og reagere på kommandoer fra basestasjonen hurtig, spesielt med tanke på å stoppe svermen fra basestasjonen. Derfor er det laget en egen exception, et eget unntak, som kastes når enheten mottar en ny kommando. Et unntak er noe som tvinger programmet til å stoppe prosessen den utfører og håndtere unntaket. Som regel blir slike unntak laget som noe som fanges opp i *try/exception*-blokker, her kan unntak man forventer å få i koden håndteres eller hoppes over for å fortsette en loop. I vårt tilfelle er det laget en egen klassefunksjon for et unntak som kastes når det blir publisert en ny kommando, det tvinger behaviouren til å stoppe og håndtere kommandoen og endre nødvendig data, eventuelt å stoppe programmet. Exception-håndteringen er lagt som et unntak til behaviour-loopen i hovedprogrammet *behaviour.py*, selv om unntaket kastes originalt i en undermodul i *behavioor_sub.py* som lytter etter data på ROS. Figur 4-33 viser hvordan klassen er definert og Figur 4-34 viser hvordan den håndteres i hovedprogrammet.

```
 8   class NewCommand(Exception):
 9       '''New command recieved, needs to be handled'''
10       def __init__(self, msg):
11           self.msg = msg
12
13       def __str__(self):
14           return(self.msg)
```

Figur 4-33: Klassedefinisjon for unntak kalt NewCommand (fra fil

I Figur 4-34 ligger hovedprosessen til behaviouren i linjene 45-52, undermodulene i bruk der her går 2 til 3 nivåer ned i filstrukturen, men uansett hvilken modul den er i stopper den om unntaket NewCommand kastes. Da leser den ut data som definerer hvilken type kommando den har fått og hvilke endringer som må gjøres i linjene 58-67, her vises kun hvordan den stopper eller endrer geografisk begrensning, men endring av destinasjon er veldig lik som den geografiske begrensningen.

```
39  while not rospy.is_shutdown():
40      try:
41          command()
42
43          time.sleep(0.5)
44
45          #get newest table of data from units in swarm
46          data_full = data()
47
48          #calculate wanted vector based on current behaviour
49          wanted = behaviour(data_full)
50
51          #publish wanted vector to autopilot
52          behaviour_out(wanted)
53
54      except NewCommand:
55          if command.stop() == True:
56              rospy.signal_shutdown('stop command recieved')
57
58          else:
59              colav = command.get_colavMode()
60              if colav == 1: #new behaviour order
61                  new_behaviour = command.get_taskType()
62                  try:
63                      rospy.loginfo("Initiating new behaviour...")
64                      del behaviour
65                      behaviour = Behave(BOAT_ID, fence, new_behaviour)
66                  except AttributeError as e:
67                      rospy.loginfo("could not initiate new behaviour,
```

Figur 4-34: *try*-blokk for behaviour med unntakshåndtering (fra fil behaviour.py)

4.6.2 Boids

Den første atferden til svermen er basert på Flocking. Hensikten med atferden er at USV'ene skal seile i flokker. Hver USV får en radius den skal søke rundt seg. Finner den andre båter innenfor radiusen gjør den tre forskjellige utregninger. Hver utregning blir gjort for å tilfredsstille en regel som er nødvendig for at flokkene skal kunne dannes. De tre reglene er *alignment* eller samstilling, *cohesion* eller tiltrekning og *separation* eller frastøting. samstilling regner ut en vektor for å sette lik kurs som de andre USV'ene i nærområdet. Tiltrekning er den tiltrekkende kraften. Den regner ut et senter av båtene i området og setter en vektor mot den vektet etter avstanden den selv er unna dette senteret. Er den langt unna blir tiltrekningen stor og motsatt. Seperasjon sørger for den frastøtende

```
47          alignment = self._calculate_alignment(global_list)
48          cohesion = self._calculate_cohesion(global_list)
49          separation = self._calculate_separation(global_list)
50
51          wantedXY = alignment * self.Ka + cohesion * self.Kc + separation * self.Ks
52
53          return wantedXY
```

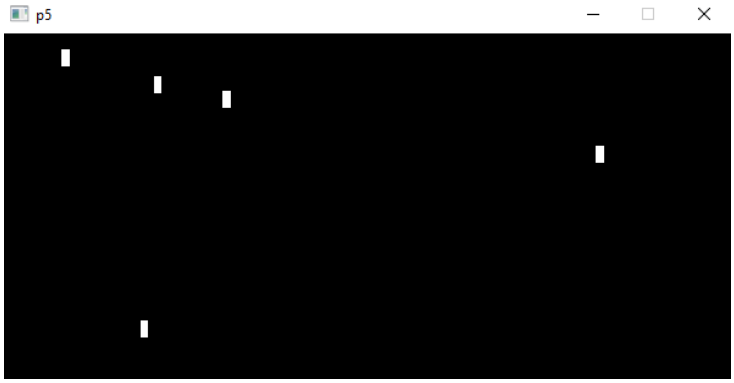
Figur 4-35: Kodeutklipp for sammenlegging av kraftvektorer i Boids adferd (fra fil Boids.py)

kraften og skal dermed virke mot tiltrekningen. Størrelsen på vektorene til tiltrekning og frastøting varierer med avstanden til de andre i svermen. For samstilling blir gjennomsnittsfarten brukt som referanse for at alle skal sette samme fart samme retning. Totalvektoren som skal brukes i autopiloten er, som vist i Figur 4-35, summen av vektorene til de tre reglene.

4.6.2.1 SIMULERING AV BOIDS

For å kunne kvalitetssikre de utregningen som blir gjort i koden er det nyttig å ta i bruk en simulator. En python-simulator ble derfor funnet på nettet og tatt i bruk i for å teste egen behaviour (Zandie, 2019). Samtidig som den lager et vindu for simulering av kode, ligger det også en egen boid-behaviour på nettsiden. Vår egen boid-behaviour er inspirert av det arbeidet de har gjort, siden koden fungerer godt i simulatoren deres. Det er gjort små endringer på simulatoren for å gjøre situasjonen i vinduet mest mulig likt vårt eget system, uten at det er viktige å presentere.

I simulatoren foregår koden ganske likt som det ville gjort i virkeligheten. Hver USV bruker informasjon om andre egne rundt seg, regner ut en total-vektor basert på de tre reglene og setter totalvektoren som fartsvektor. Ulempen med å bruke en simulator til testing er at det ikke er mulig å realisere virkeligheten og de utfordringene den medbringer. Friksjon i vannet, forsinkelse i deling av informasjon eller problematikk med fremdriften (motor og servo) er eksempler på problemstillinger som kan gjøre bevegelsene til svermen forskjellig i virkeligheten og i simulatoren. Simulatoren brukes derfor kun til å kvalitetssikre utregningene som blir gjort, den er vist Figur 4-36 med simulering av 5 båter som flokker.



Figur 4-36: Simuleringsvinduet. Hver rektangel symboliserer en USV i svermen.

4.6.3PSO

I likhet med Boids adferden benytter PSO seg av vektor beregninger for å beregne beste bevegelse. Vår implementering av PSO baserer seg på teorien og hvordan det kan simuleres, som beskrevet i 2.6.3. Det er fordi det ikke er montert noen ytterligere sensorer på båtene som kan måle signalstyrker eller lignende. Derfor simuleres en signalkilde i koden og bruker avstanden hver USV er fra den som styrke av signalet. Dette er vist i Figur 4-37 og Figur 4-38 som er utklipp av koden for PSO.

Figur 4-37 viser hvordan hver enkel USV regner ut avstanden fra sin nåværende posisjon til den simulerte signalkilden på linje 66. Denne avstanden går gjennom en funksjon for å beregne simulert styrke av signalet på den avstanden med støy. Støyen regnes ut i Figur 4-38 basert på funksjonen $Verdi = støy + \frac{persepsjon}{avstand^2}$. Persepsjon er en variabel i koden som definerer en begrensing for rekkevidden til kommunikasjonen for hver enhet. Om persepsjonen er 100m blir kun andre USV'er innenfor det tatt med i atferds-utregningen.

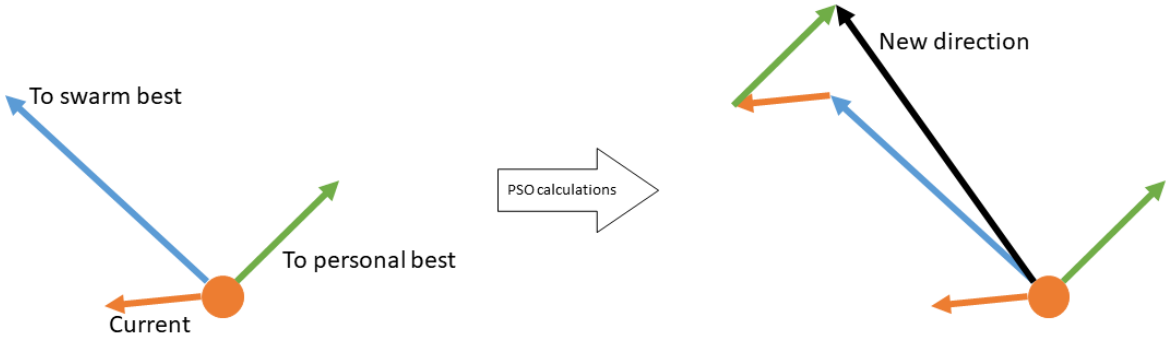
```
65      def _check_pBest(self):
66          dist = self.position.calculate(self.wanted)
67          current_self = self.noise_function(dist.magnitude)
68          if current_self > self.best_self['value']:
69              self.best_self['position'] = self.position
70              self.best_self['value']    = current_self
```

Figur 4-37: Kodeutklipp for sjekk etter ny personlig beste (fra fil PSO.py)

```
72      def noise_function(self, distance):
73          if distance != 0.0:
74              noise = random.randrange(1, 100) /100
75              value = float(noise) + (self.perception / m.pow(distance, 2.0))
76              return value
77          else:
78              return 0.0
```

Figur 4-38: kodeutklipp for støyfunksjon (fra fil PSO.py)

Som beskrevet i teorien blir de tre vektorene som peker med nåværende fart, mot personlig best og mot globalt beste lagt sammen og sendt videre til autopiloten. Også illustrert i Figur 4-39.



Figur 4-39: Illustrasjon av vektorsammenlegning i PSO

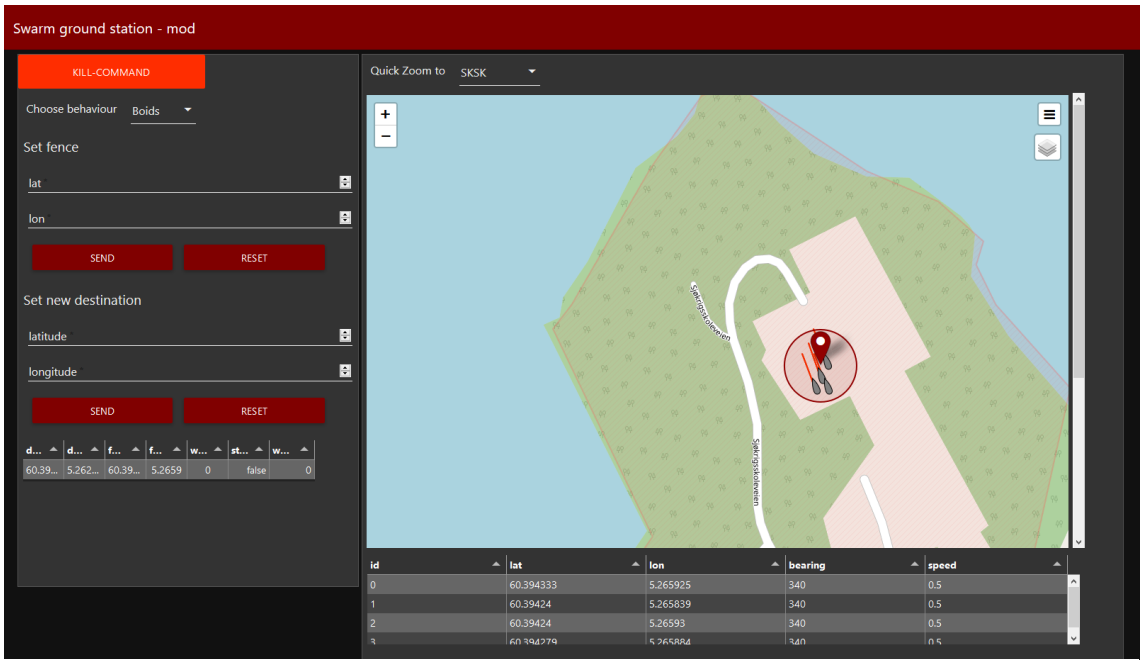
4.6.4 Geografisk avgrensning

For å ha bedre kontroll over svermen er det lagt inn en geografisk-avgrensning i koden, kalt en *fence*. Den geografiske avgrensningen er området svermen skal operere i. Den er definert som sirkelen rundt et GPS-punkt med radius lik 15 meter, avstanden kan endres i koden. Starter en USV utenfor denne avgrensningen setter de kurs mot midten av den helt til den er innenfor. Det vil si at USV'ene ikke driver en form for atferd (boids, PSO el.) når de er utenfor. I basestasjonen som presenteres i 0 er det laget en ekstra funksjon som gjør det mulig å flytte denne avgrensningen der det er ønskelig i et kart, denne funksjonen og hvordan den ser ut er vist i Figur 4-40. Ved å ha et flyttbart avgrenset område får den samtidig implementert enda en funksjon til svermen; sende svermen til det områdene som er ønskelig at den skal operere.

Figur 4-40: Flytting av geografisk avgrensning visualisert i basestasjonen.

4.7 Basestasjon

Det er vanskelig å vedlikeholde, drive og videreutvikle en sverm uten en form for visuell input for hva hver enhet driver med og hvordan de beveger seg. Basestasjonen skal gi oss mulighet til å overvåke systemet, samt overstyre den om det skulle være nødvendig eller ønskelig. I Figur 4-41 vises basestasjonen, den er primært delt i to. Venstre siden er den siden man sender kommandoer og justerer svermen fra, mens høyre siden blir brukt til overvåking av data fra båtene og deres plassering i kartet. Basestasjonen er utviklet i Node-RED ved hjelp av en ui-palette, dette er beskrevet i Vedlegg H – Node-red for basestasjon.



Figur 4-41: Utseende til basestasjon i Node-RED UI

4.7.1 Visualisering av basestasjonen

Den styrende delen av basestasjonen er utviklet ved bruk av 5 ui-noder.

1. *Kill command* legger svermen død i vannet
2. *Choose Behaviour* gjør det mulig å bytte mellom atferder.
3. *Set fence* lar oss flytte avgrensingsområde til en ny lokasjon. Fungerer også å flytte fencen ved museklikk i kartet.

4. *Set new destination* sender ut en destinasjon svermen skal gå til. Brukes hovedsakelig som *return to home* funksjon.
5. *Current command* lager en tabelloversikt over nåværende styring. Overskriftene i tabellen er ikke synlig siden størrelsen ikke er stor nok, men den tar for seg *set new destination* sine koordinater i de to første kolonnene, *fence*-koordinater i de to neste, om svermen er gitt en ønsket fart i fjerde, om *kill command* er true eller false i femte og om svermen er gitt en ønsket vinkel i siste.

4.7.2 Overvåking

Overvåkningen er ikke like omfattende, men vel så viktig. Den består av 3 ui-noder.

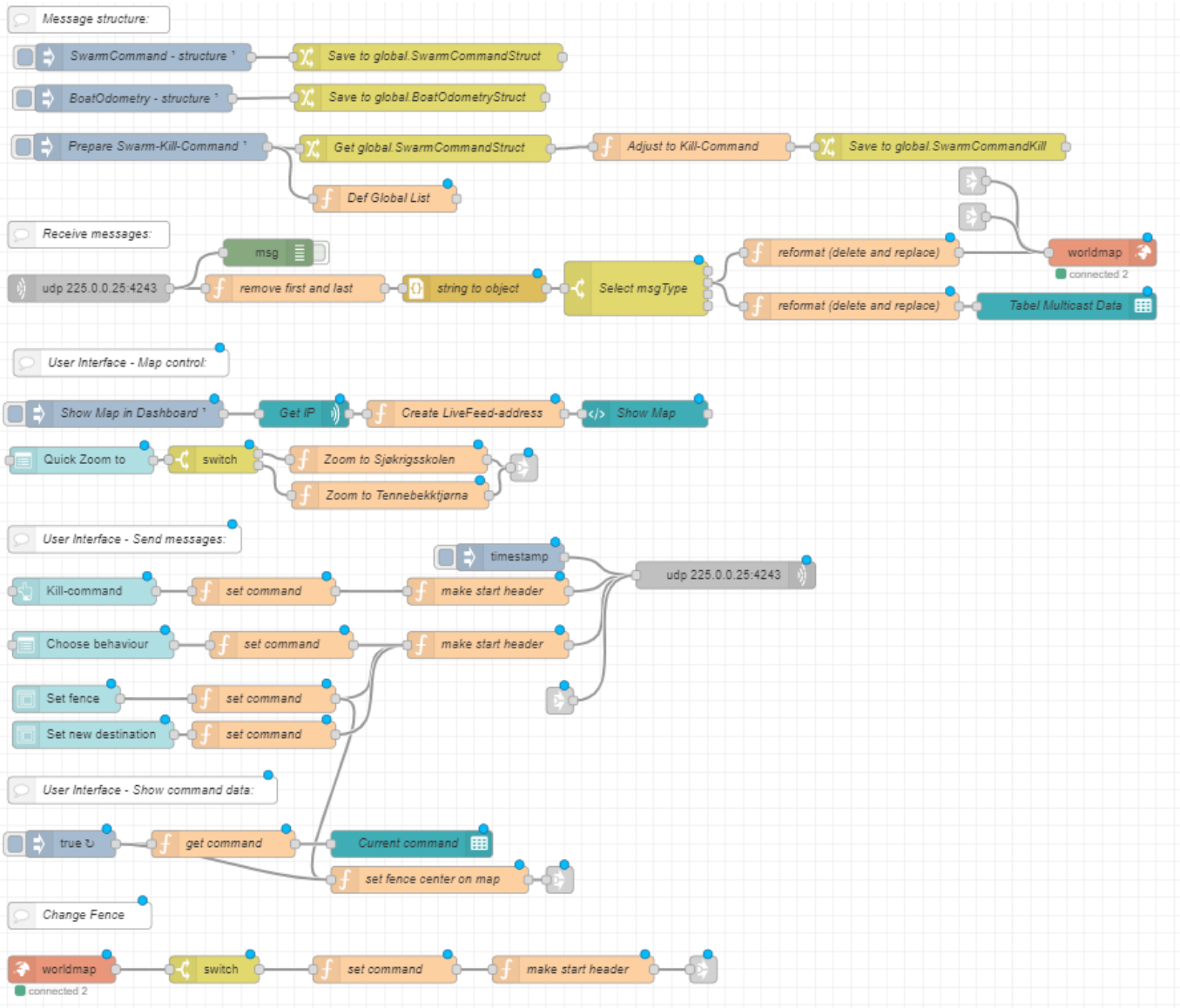
1. *Quick zoom to* brukes som hurtigzoom til sjøkrigsskolen og Tennebekktjørna, som er de områdene det testes oftest i.
2. *Map* er verdenskartet for visualisering av USV bevegelser. Som Figur 4-41 viser har hver USV også en fartsvektor som skal vise farten den holder.
3. *Informasjonstabell* henter inn de viktigste multicast dataene fra hver USV og sorterer dem i en tabell etter ID. Gir oss en enda bedre forståelse av hvordan de beveger seg og hvor mange båter som befinner seg i svermen.

4.7.3 Flyt

Figur 4-42 viser flyten til basestasjonen. Flyten presenteres etter brukergrensesnittet for å gi en bedre forståelse for hvordan brukergrensesnitt nodene fungerer. Flytfiguren er all programmeringen som gjort i Node-RED for å realisere basestasjonen. Figuren vil ikke bli beskrevet i detalj, men hver bolk vil heller bli presentert. Flyten er delt opp etter kommentering i Node-RED (hvite noder) og har dermed tilhørende kode under seg.

1. *Message structure* definerer strukturen de forskjellige dataene fra USV'en kommer i, til bruk i videre funksjoner. Direkte oversetning fra ROS-msg til JSON-objekt
2. *Recieve Messages* lytter etter multicast meldinger på nettet og formaterer dataene før de publiseres i informasjonstabellen og viser båtene i kartet.
3. *User Interface – Map Control* gir oss tilgang til å vise kartet i basestasjonen og oppretter samtidig en node for hurtigzoom til sjøkrigsskolen og Tennebekktjørna.

4. *User Interface – Send messages* oppretter 4 noder for den styrende delen av brukergrensesnittet. Enkle funksjoner med kommandoer blir multicastet til hver USV i svermen om de endres.
5. *User Interface – Show command data* lager tabelloversikten over hvilke styringskommandoer som er aktive og oppretter muligheten til å flytte fencen med et museklikk.
6. *Change Fence* sender melding til multicast hvor fencen i kartet er definert. Så om den skulle endres med et museklikk eller m nye koordinater skulle bli skrevet inn blir fencen hentet i kartet og sendt til multicasten som omdefinerer fencen hos hver USV.



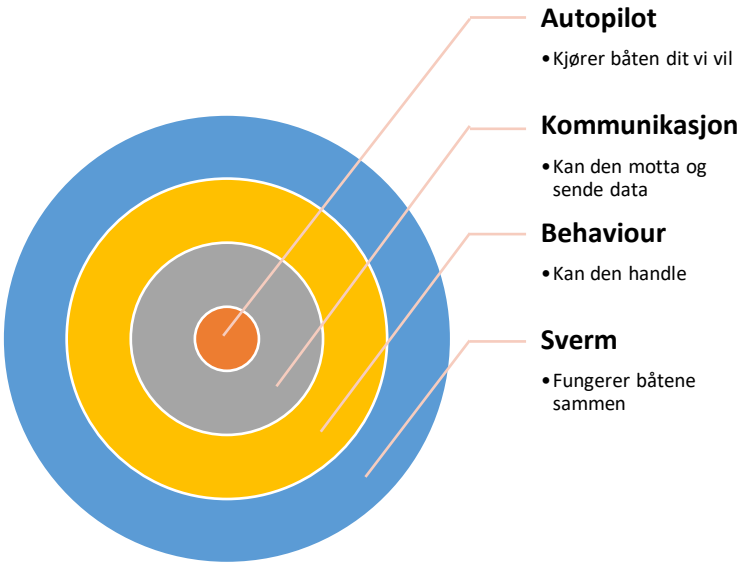
Figur 4-42: Node-red flyten som visualiserer og driver basestasjonen.

5 Tester og resultater

I denne delen vil vi se nærmere på testene vi har gjort i prosjektet. Testene skal vise hvordan vi har gått fram for å teste plattformen mens den har vært under utvikling. Kapittelet vil starte med en beskrivelse av den metoden vi har brukt, før vi ser nærmere på hver del av testene. For å forhindre usikkerhet rundt hvorfor tester har blitt utført vil vi presentere testene kronologisk.

5.1 Metode

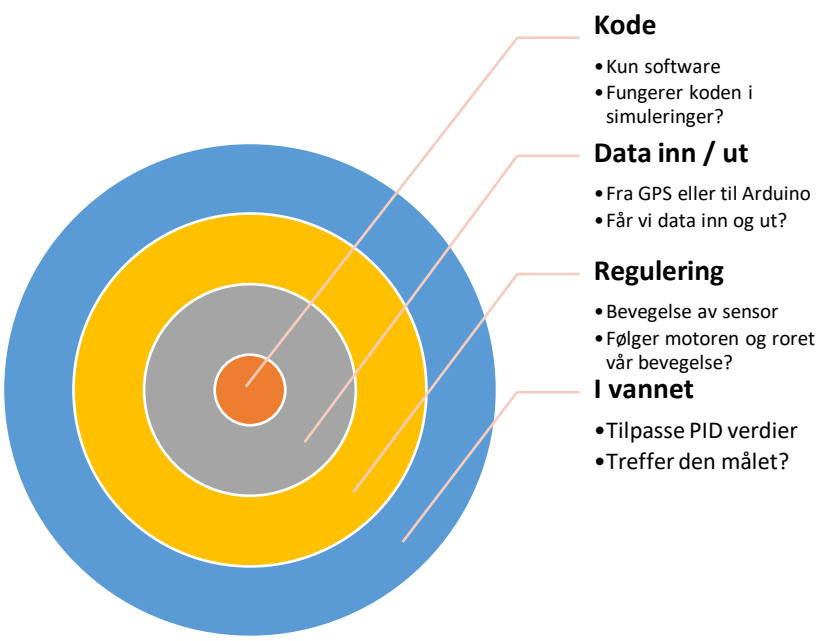
Det er mye kode som skal kvalitetssikres når vi går fram for å teste en sverm. Derfor har vi valgt å dele testene opp i fire hovedbolker (autopilot, kommunikasjon, behaviour og sverm), disse vises stegvis i Figur 5-1. For å kunne kalle plattformen for en sverm må autopilot, kommunikasjon og behaviour først fungere for seg selv, for så å fungere sammen. Hensikten er å få resultater på land som viser at hver del av svermen fungerer som den skal før den testes på vannet. På denne måten er det enklere å avdekke feil hos de ulike hovedprogrammene enn hvis vi skulle teste alt samtidig fra starten. Denne metodikken for testing passer godt for utvikling av sverm konsepter fordi hver av blokkene bygger utenpå den tidligere testet. Hver båt må kunne styre seg selv først, så derfor starter vi med å teste og ferdigstille autopiloten. Kommunikasjonen blir så testet for å forsikre oss at båtene får inn informasjonen de trenger til adferden Når alle delene fungerer for seg selv setter vi de sammen og tester plattformens evne til å drive sverm.



Figur 5-1: Testmetodikk for systemet

5.2 Autopilot

Prosessen for å teste autopiloten er en av de lengste og mest krevende i dette prosjektet, på bakgrunn av hvor ødeleggende dårlig styring av båtene vil være for resten av sverm-prosessen. Derfor delte vi testingen av autopiloten inn i flere faser, med tanke på hvor mange komponenter/utstyr vi involverte i testene. For hver komponent vi legger til i testen, mot det endelige systemet, øker mulighetene for feil. Samtidig øker også verdien av testene og resultatene vi får, de blir mer treffsikre mot det endelige systemet for hver komponent vi legger til i testene. Til slutt må vi ende opp med å teste systemet med alle komponentene for å vite at alt fungerer, men konklusjonen vår var da at en stegvis tilnærming gir bedre oversikt og kontroll over feilmarginene i hver enkel test. Samtidig gjør det feilsøking av testene lettere ved at vi vet hva som fungerer for hver test som går. Vi illustrerer det nok en gang som en målskive i Figur 5-2. Det innerste laget er det første vi tester, koden.



Figur 5-2: Metodikk for testing av autopilot

Når vi er fornøyd med koden alene så legger vi til data fra sensorer til koden for å se hvordan det går. I alle videre tester bruker / tilpasser vi koden vi har testet. På den måten tester vi kjernen av autopiloten først også bygger videre oppå den derfra til vi har et endelig produkt med alle komponenter. Målsetningen er å få en regulator som treffer godt nok uten å bruke mye tid på komplisert regulering.

Først ser vi om elementer fra autopiloten gir de resultatene vi ønsker med simulerte verdier. Så tester vi om dataflyten inn og ut fra autopiloten fungerer. Neste test er å bevege sensorene for å se om koden håndtere bevegelsesdata og endringer i omgivelsene. Når

autopiloten viser til robuste nok resultater i de foregående testene tar vi den med ut på vannet. Ser vi så at koden ikke fungerer som den skal på vannet gjør vi de tilpasningene vi anser som nødvendige og prøver på nytt.

5.2.1 Kode

Ved de første testene av kun kode lager vi flere skript for å teste de forskjellige komponentene av autopiloten. Den viktigste komponenten å teste for autopiloten er PID-regulatoren. PID-regulatoren er hjertet av autopiloten, alle andre funksjoner er for å hjelpe, få data inn eller sende data ut fra regulatoren. Utfordringen var at vi ikke kunne teste regulatoren før vi fikk båten på vannet. Så første steg ble å teste noen av utregningene innad i regulatoren, viktigst av alt den for å finne nærmeste vinkel.

5.2.1.1 UTREGNING NÆRMESTE VINKEL

Denne testen er ganske rett fram, men fortsatt viktig å få med. Siden sensoren gir ut retningsdata i forhold til nord må vi regne ut minste vinkel mellom ønsket og nåværende retning. F.eks. om båten holder en retning på 320, men det er ønsket at den legger om til 10 grader, vil vi med å bare finne differansen få at båten må legge om 10-320=-310 grader. Et slikt utslag vil gjøre at den tar en enorm sving mot venstre da den egentlig bare 50 grader fra ønsket retning om den svinger til høyre. Derfor må vi ha en funksjon som for alle 2 vinkler mellom 0-360 grader gir den minste forskjellen mellom disse med riktig fortegn. For å gjøre dette bruker vi en funksjon som heter atan2, den returnerer vinkler mellom -180 og 180 istedenfor -90 til 90, derfor gir den hele 360 ved en enkel omregning for vinkler som er negative. Ved å kjøre et kjapt test-skript for funksjonen som går gjennom alle vinkler mellom 0-360 ser vi at funksjonen gir ut minste vinkelverdier for alle tilfeller. Figur 5-3 er det vi får av resultater fra testen, viser kun økninger med 45 grader for å synliggjøre alle muligheter.

```
current angle:  225  | wanted angle:  225
closest angle:  0.0

current angle:  270  | wanted angle:  225
closest angle:  -45.0

current angle:  315  | wanted angle:  225
closest angle:  -90.0

current angle:  45  | wanted angle:  270
closest angle:  -135.0

current angle:  90  | wanted angle:  270
closest angle:  180.0

current angle:  135  | wanted angle:  270
closest angle:  135.0

current angle:  180  | wanted angle:  270
closest angle:  90.0

current angle:  225  | wanted angle:  270
closest angle:  45.0

current angle:  270  | wanted angle:  270
closest angle:  0.0

current angle:  315  | wanted angle:  270
closest angle:  -45.0
```

Figur 5-3: Simulert test av funksjon for nærmeste vinkel

5.2.2 Data inn / ut

Denne testen er kun gjort for å sjekke at vi får data inn og ut i autopiloten. For å teste dette ser vi om autopiloten henter inn GPS data og ser om autopiloten bruker det til å gjøre ror-utslag. Målet med testen er å forsikre oss om at vi får data inn i autopiloten og ut til rorene. Samtidig vil vi også bruke testene til å kvalitetssikre arbeidet ved å diskutere kvaliteten på dataene.

5.2.2.1 DATA INN

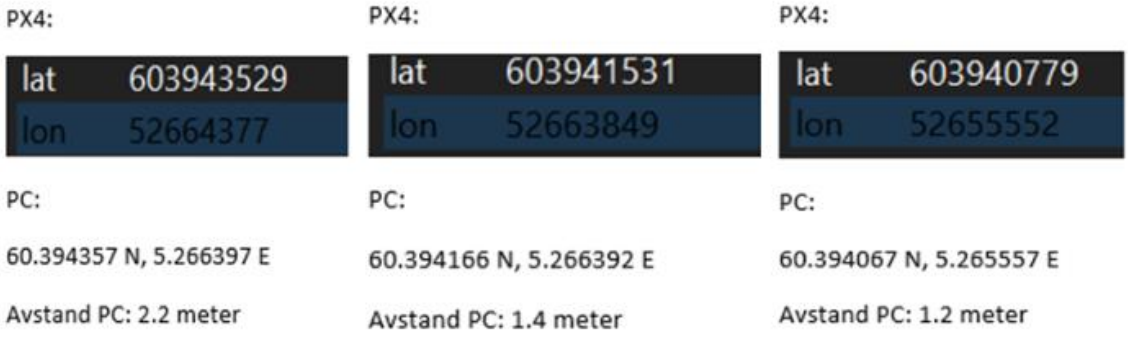
Det første vi oppdaget når vi startet med testene var at vi slet med å få data inn fra sensoren vår, PX4. Den er laget til å kunne fly droner som en egen autopilot, men vi bruker den kun til innsamling av sensor data. Dette medfører noen mindre utfordringer for datauthentingen. Siden sensoren er ment for å kjøre droner i luften utelukkende på GPS og innsamlet data fra interne sensorer, lar den ikke dronen ta av før den er fornøyd med dataene den har. Her kommer begrepet *GPS-fix*, med mindre Pixhawk'en har fått gode nok GPS-data over en lang nok periode så publiserer den ikke data på en del topic'er. Det er nettopp disse topic'ene vi prøvde å lese data fra. Vi valgte derfor å gå over til å bruke nye topic'er som blir publisert av PX4 for å gjøre autopiloten mindre avhengig av *GPS-fix*. De nye topic'ene kan brukes uansett om PX4 har *GPS-fix* eller ikke, noe som gjør dem mindre nøyaktige, men de vil samtidig publiserer hele tiden. I Tabell 3er det en oversikt over de gamle topic'ene og hvilke vi byttet dem ut med.

Gamle topic'er	Nye topic'er
/mavros/global_position/global	/mavros/global_postion/fix/raw
/mavros/global_position/compass_hdg	/mavros/vfr_hud
/mavros/global_position/gp_vel	/mavros/global_position/raw/gp_vel

Tabell 3: Oversikt endring av topic'er for å ikke være avhengig av *GPS-fix*

For å kunne tilpasse oss den økende unøyaktigheten til sensordataene, gjør vi en rask test for å se hvor stor feilmargin det er snakk om. Mobile GPS mottakere er kjent for å ha en markant feilmargin, noe vi også målte på PX4 når vi henter du data fra de nye topic'ene. Med å sammenligne GPS koordinatene PX4 publiserer og Google Maps på ulike steder,

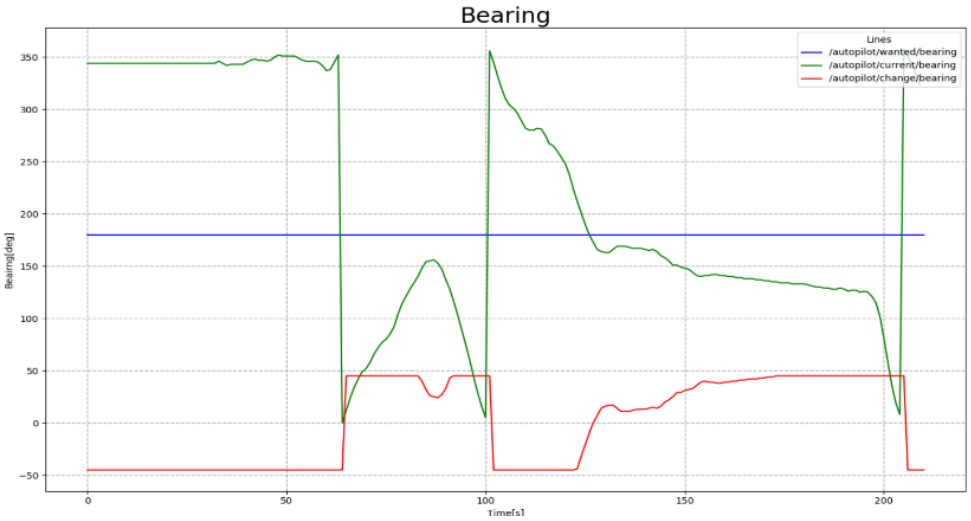
kom vi fram til en unøyaktighet på 1.6 meter. Testene vist i Figur 5-4 gir oss ikke bevis på at feilmarginen til Pixhawk'en er 1.6 meter, men det forteller oss at dataene den får inn sjeldent er 100% korrekte og må derfor tas høyde for i koden og resultater fra videre tester. Da dette ikke er den eneste feilmarginen, vil vi forsikre oss ved å ha en sikkerhetsradius rundt hver USV når vi tester svermen senere.



Figur 5-4: Brukt mobil og PC til å teste GPS-nøyaktighet til PX4.

5.2.2.2 *DATA UT*

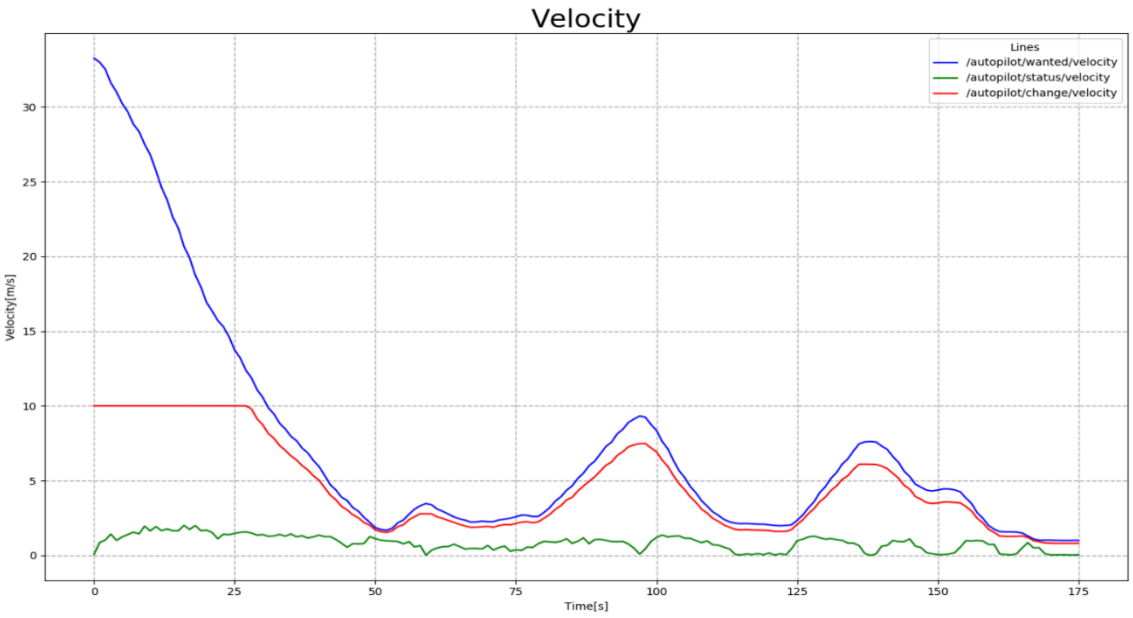
Når problemet med å få sendt inn data er løst ser vi om Arduino får data tilsendt og sender data til roret. Figur 5-5 viser at roret gjør en endring i retningen når den får tilsendt data fra PX4. Selv om kvaliteten på dataene ikke skulle være i fokus ser vi fortsatt at den setter en kurs for å treffe ønsket vektor. I starten av testen er den nåværende vinkelen mye større enn ønsket vinkel. Roret gir da ut en negativ vinkel for å få nåværende vinkel ned til ønsket. Midt i figuren er nåværende vinkel lavere enn ønsket vinkel, roret gir derfor et positivt utslag for å øke nåværende vinkel til ønsket. Det viser seg altså at rorene får data, men også gjør de riktige avgjørelsene.



Figur 5-5: data sendt til roret ved test av data inn

5.2.3 Regulering ved fysisk bevegelse

I forrige test fikk vi bekreftet at PID-regulatoren gjør regner ut noe og sender det til motor og ror. Samtidig er det ikke gitt at den innretter seg like godt når vi tilfører en ny faktor; bevegelse. For å teste dette tok vi båten i hendene og gikk rundt med den. Målet med testen er å se om PID-regulatoren innretter seg etter ønsket fart. Vi satt derfor et GPS-punkt som autopiloten skulle navigere til. Farten den ønsker øker med avstanden fra målet. I Figur 5-6 ser man at endringen motoren gir i fart endrer seg med ønsket fart. Autopiloten er klar for tester på vann.

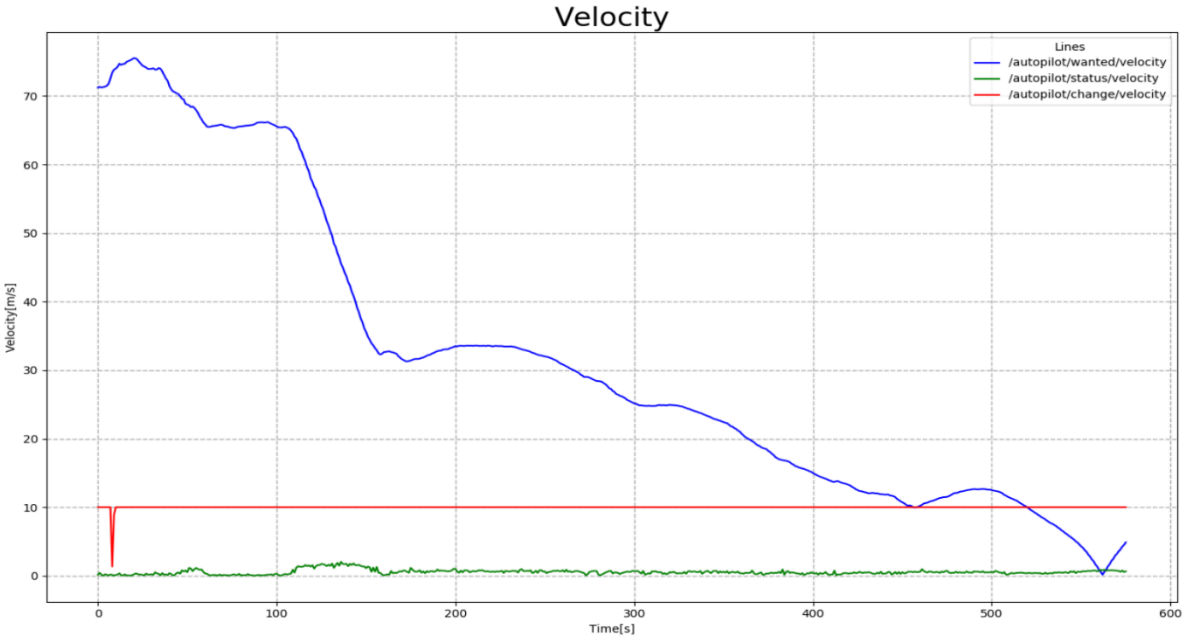


Figur 5-6: Bevegelse med båten på land. Ønsket fart øker med avstand fra GPS punktet som den er satt til å kjøre til.

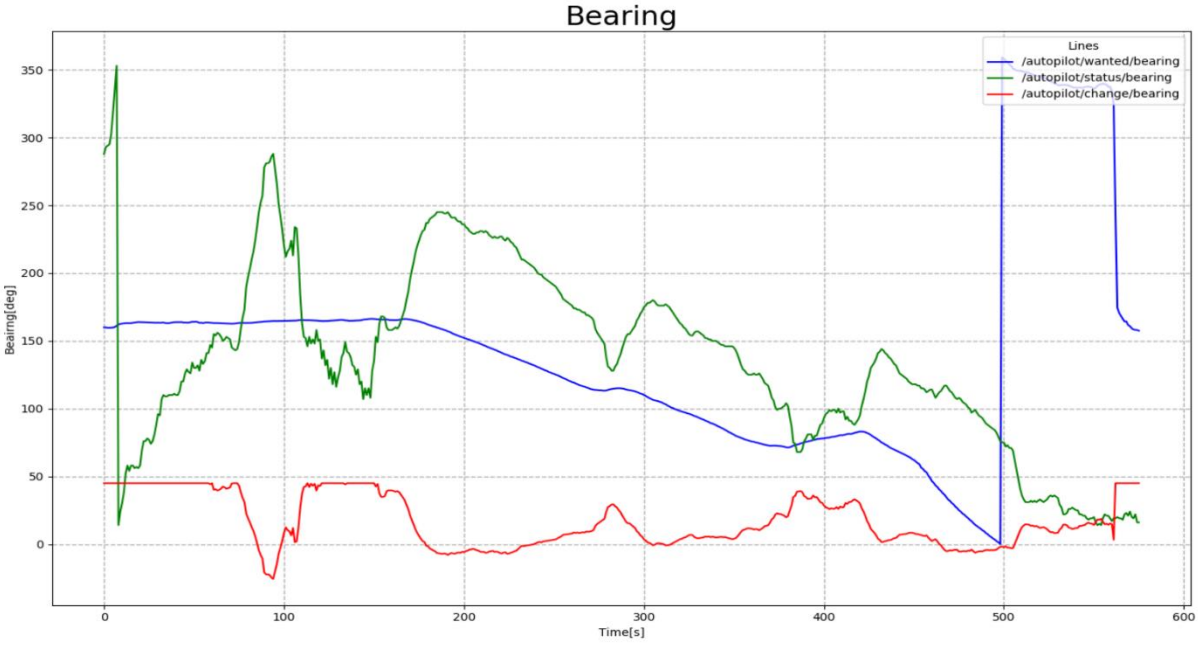
5.2.4 Tester på vannet

Når båtene skal på vannet, blir vær og vind en faktor. Testene vi har gjort til nå sier mer om hvorvidt koden fungerer enn noe om hvilken grad den kommer til å fungere i reelle omgivelser. Derfor er målet med disse testen å forsikre oss om at autopiloten fungerer som den skal i de omgivelsene den skal operere.

5.2.4.1 FØRSTE TEST AV AUTOPILOT PÅ VANNET



Figur 5-7: Fartsdata fra første test på vannet



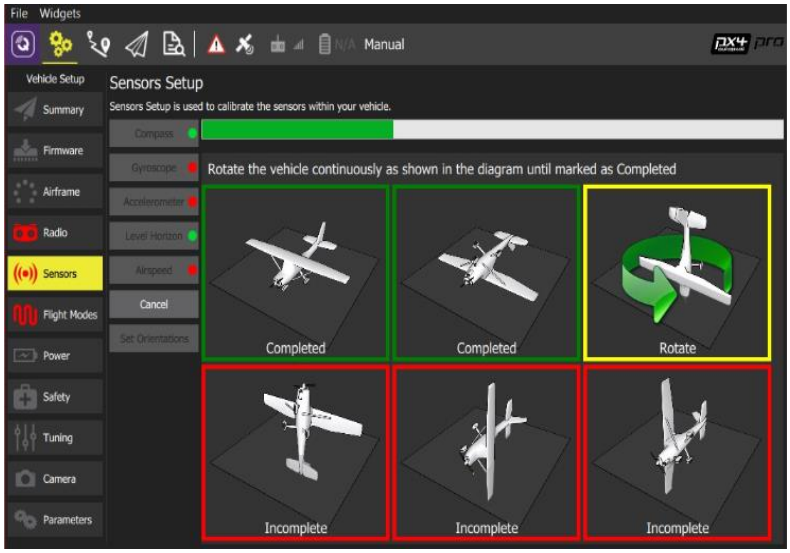
Figur 5-8: Retningsdata fra første test på vannet

Testene gir oss ikke så mye mer enn at vi får klarhet i to problemer. Det første er at I-leddet bygde seg for stort når avviket var stort over tid. Vi tok i bruk P og I leddet under testene, og det virker som at I-leddet overstyrer pådraget. Avviket mellom ønsket fart og nåværende fart er stort over lang tid, som gjør at I-leddet til bygger seg opp. Når ønsket

fart nærmer seg nåværende, etter 450 sekunder i Figur 5-7, burde endringen i fart synke, noe den ikke gjør på grunn av I-leddet. Regulatoren må derfor reguleres forsiktig i vann til vi får noe som fungerer godt nok, først som en P-regulator, før vi eventuelt ser på PI- og PID-regulator. Det andre er at USV'en ikke håndterer dårlig vær spesielt godt. Retning grafen i Figur 5-8 viser hvor mye endringer det er i nåværende vinkel selv om endring i vinkel fra autopiloten ikke varierer stort. Samtidig prøver USV'en å stabilisere vinkelen bedre enn farten (mye fordi I-leddet ikke får bygd seg opp), men det er fortsatt ikke presist.

5.2.4.2 KALIBRERING AV KOMPASS I QGC

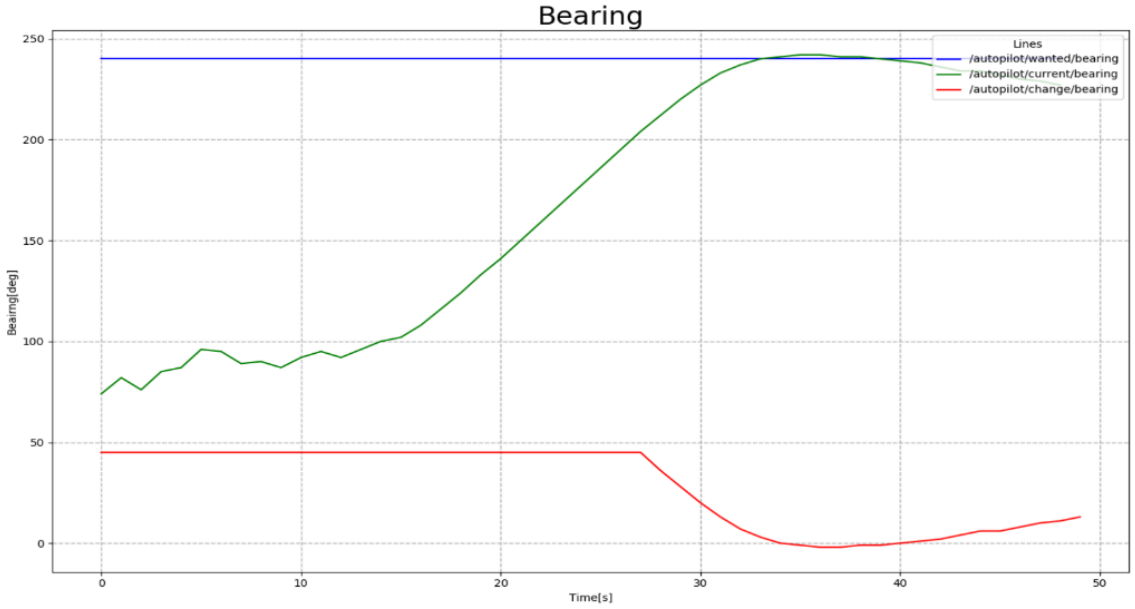
Første test viste at kompassdataene fra Pixhawk'en var unøyaktig. Siden kompass reagerer på magnetisk nord, kan magnetiske felt rundt kompasset føre til forstyrrelser. Vi testet derfor først om det kunne være motoren som førte til feilen, men det viste seg at feilen ikke kom derfra. Tidligere hadde vi brukt et program kalt QGC (Q Ground Control Station) for å koble til en radiokontroller og bruke programmet som en kontrollstasjon for småbåtene. I programmet er det også mulig å kalibrere kjøretøyet for å optimalisere PX4 sin forståelse for kjøretøyet den er tiltenkt. Kompass kalibrering ble derfor utført som vist i Figur 5-9 og kompasset var ikke lengre unøyaktig.



Figur 5-9: Kalibrering av kompass i QGC

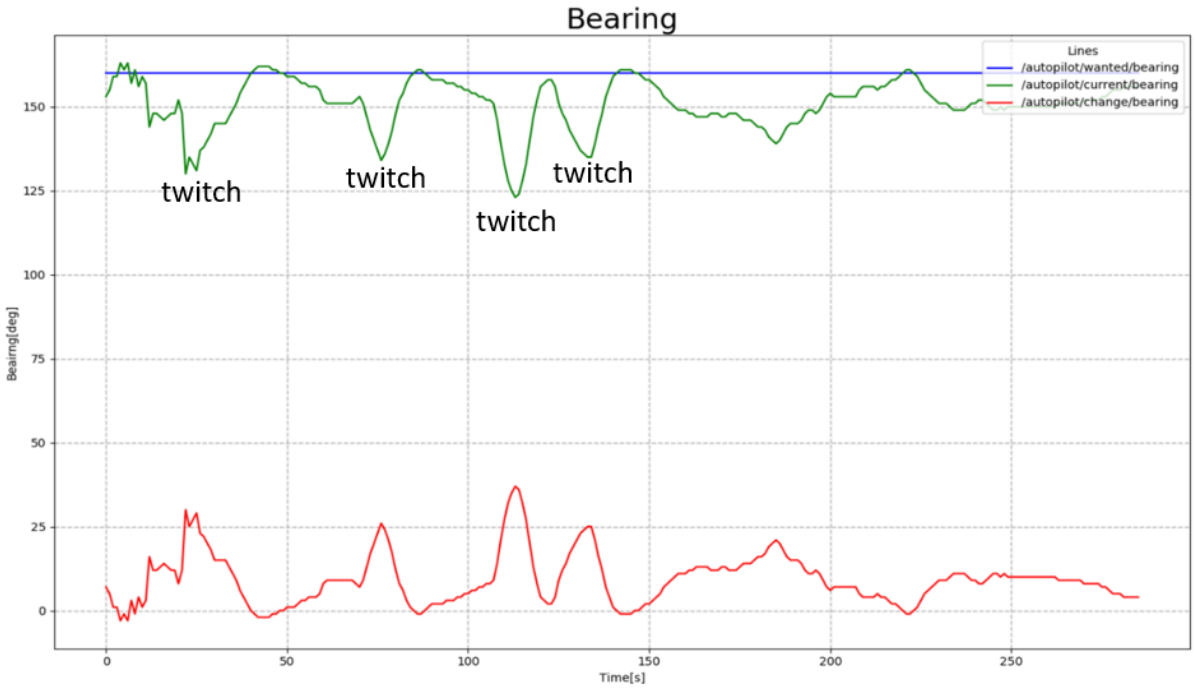
5.2.4.3 ANDRE TEST AV AUTOPILOT PÅ VANNET

Den første testen som ble gjort denne dagen var å gi båten en ønsket fart og vinkel den skulle holde, kun med bruk av P-leddet i regulatoren. Figur 5-10 viser retningen båten har i løpet av testen. Endringen i vinkel avtar forsiktig og holder seg på null når nåværende vinkel blir lik ønsket vinkel, som kan ses fra 30 til 50 sekunder i Figur 5-10. P-regulatoren klarer dermed å stabilisere seg på ønsket vinkel ganske fort, og ligge på ønsket vinkel over lengre tid.



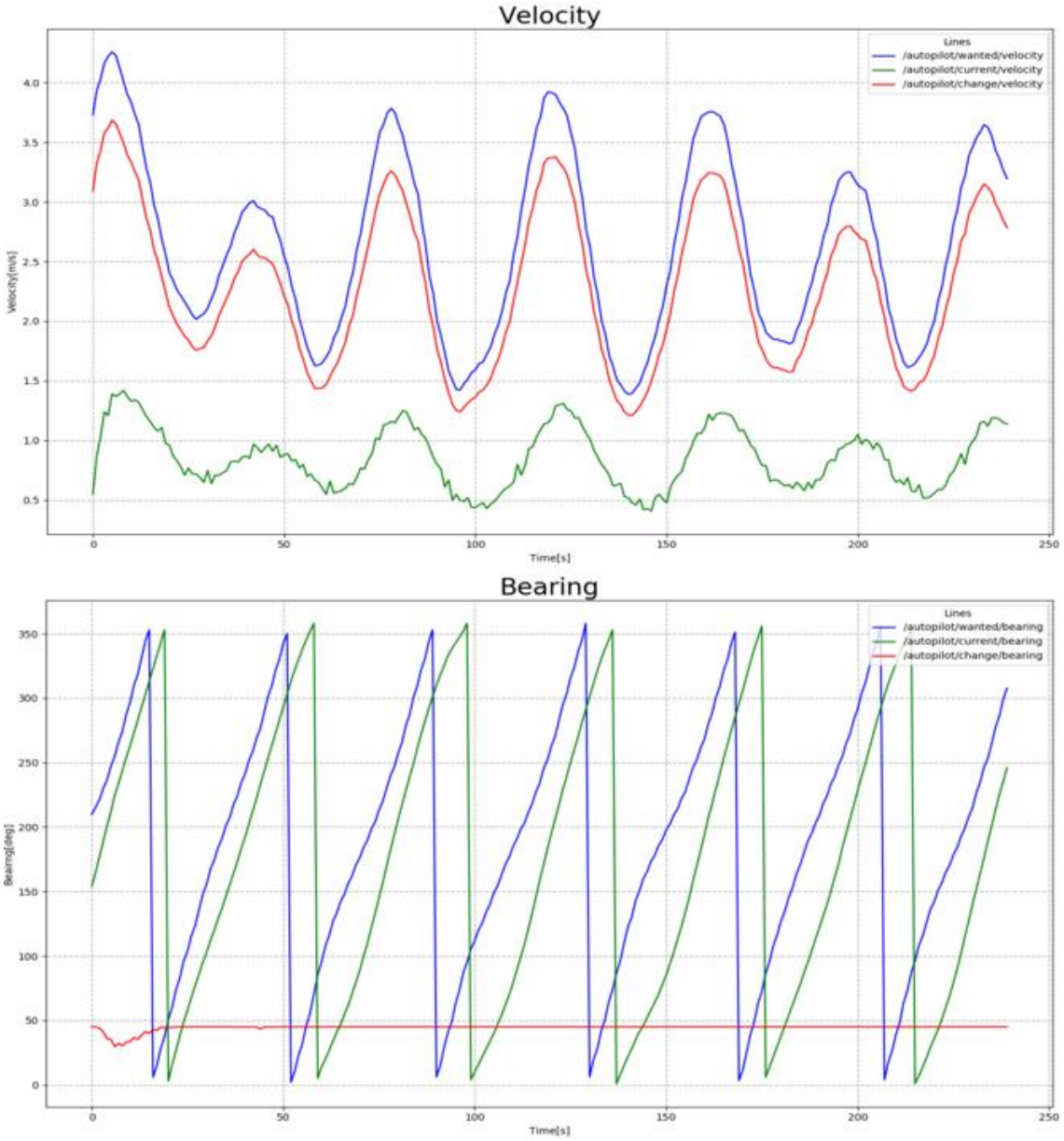
Figur 5-10: Tester å sette en fast ønsket vinkel båten skal stabilisere seg på.

Videre testet vi om autopiloten evnet å komme seg tilbake på ønsket kurs selv om den ble utsatt for krefter som satt den ut av kurs. Kreftene som ble påført var et rykk i sikkerhetssnoren som er festet på båten. Figur 5-11 viser at det blir gjort fire rykk i snoren, representert som *twitch* i figuren. Etter hvert rykk gir autopiloten ut en endring i vinkel som skal kompensere for feilen rykket påfører. Figur 5-11 viser at autopiloten klarer å finne tilbake til riktig kurs i løpet av noen sekunder. Endringen i vinkel går tilbake til null når ønsket vinkel er lik nåværende vinkel, som at den ikke gjør et utslag på roret og holder riktig kurs.



Figur 5-11: Test med rykk i snor (twitch) der den innretter seg fint tilbake til ønsket vinkel.

En siste test ble gjort for å sjekke autopiloten klarer å treffe et GPS-punkt. Det ble derfor satt et ønsket GPS-punkt. Målet med testen er å se om autopiloten klarer å finne GPS-punktet og holde seg der. Siden vi har et håp om å få tid til å utvikle en PSO-behaviour i tillegg til boids, er denne testen å bli ansett som et minstekriterium for autopiloten. Figur 5-12 viser fart og retning til båten når den er ved GPS-punktet. Retningen til båten viser at den går i sirkler rundt punktet. Den gir et jevnt pådrag (endring i vinkel) kontinuerlig, noe som er naturlig da den ikke klarer å stå stille på GPS-punktet (siden båten ikke kan gå pådrag akterut). Siden båten klarer å sirkle et GPS-punkt, som var forventet, fungerer autopiloten godt.



Figur 5-12: Data for fart og vinkel ved sirkling av GPS-punkt.

5.3 Kommunikasjon

Når vi sa oss fornøyd med hvordan autopiloten fungerte, var kommunikasjonen neste steg. En velfungerende kommunikasjon mellom enhetene er en viktig brikke for å kunne drive sverm. Vi har skrevet to hovedprogrammer som blir kjørt som ROS-noder som beskrevet i 4.5. For å teste dette delte vi inn i tre faser og brukte vi samme metodikk som tidligere med å teste det mest grunnleggende først og så bygge videre på den til vi endte opp med en velfungerende kommunikasjon.

5.3.1 Kode

For å teste koden kjørte vi programmene på en RPi og observerte meldingene vi fikk opp på terminalen. Om programmene fungerer skal de skrive ut hvilken multicast-adresse og -port de bruker. I vårt tilfelle er dette 225.0.0.25 og 4243 henholdsvis som er satt i koden. Som vist i Figur 5-13 og Figur 5-14 var dette det samme som det vi så i terminalen.

```
NODES
  /
    boat_rx_node (swarm/Boat_RX.py)

auto-starting new master
process[master]: started with pid [696]
ROS_MASTER_URI=http://localhost:11311

setting /run_id to dadbf972-1397-11ea-9844-b827eb2c4f5c
process[rosout-1]: started with pid [707]
started core service [/rosout]
process[boat_rx_node-2]: started with pid [710]
[INFO] [1575135357.383241]: Using multicast group: 225.0.0.25:4243
[INFO] [1575135357.406464]: Starting run
```

Figur 5-13: Terminal meldinger ved kjøring av fil Boat_RX.py som node

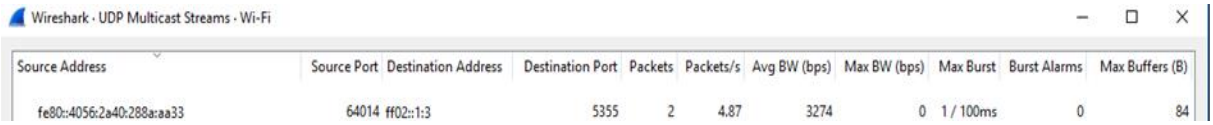
Denne testen sier egentlig veldig lite om hvorvidt programmene fungerer, det eneste vi vet etter å ha testet uten data er at programmene ikke inneholder skrivefeil. Begge programmene må få data for å kunne testes deres funksjonalitet, hvilket er neste testfase.

```
    boat_tx_node (swarm/Boat_TX.py)

auto-starting new master
process[master]: started with pid [767]
ROS_MASTER_URI=http://localhost:11311

setting /run_id to f5a40182-1397-11ea-9844-b827eb2c4f5c
process[rosout-1]: started with pid [778]
started core service [/rosout]
process[boat_tx_node-2]: started with pid [781]
[INFO] [1575135402.137688]: Using multicast group: 225.0.0.25:4243
[INFO] [1575135402.142080]: Odometry subscription: /autopilot/current
[INFO] [1575135402.146235]: Should output be compressed: False
[INFO] [1575135402.150227]: Starting to publish
```

Figur 5-14: Terminal meldinger ved kjøring av fil Boat_Tx.py som node

5.3.2 Data inn/ut

Siden vi allerede hadde en autopilot som fungerte på dette stadiet i testingen, som vi visste publiserte data til flere ROS-topic'er. Vi brukte disse dataene til å teste først å sende data også det å motta. Det var naturlig å gjøre samtidig fordi vi kunne starte de to på hver sin



Figur 5-15: UDP Multicast strømmer i Wireshark. En USV sender til alle aktive adresser på ruteren.

båt og se om de klarer å sende data en vei. Dette tillater oss til å teste programmene sin evne til å opprettholde en multicast-link med et eksternt program kalt Wireshark. Figur 5-15 viser hvordan vi ser en slik multicast-strøm i Wireshark. Figur 5-15 viser en båt som sender UDP multicast-pakker over ruteren. Pakkene blir sendt ved bruk av en protokoll, *Link-local Multicast Name Resolution*, som gjør at pakkene kun blir sendt til de aktive adresse på nettet (Venaas, 2019). Med andre ord, sender og mottaker fungerer.

5.3.3 Samhandling

Siste test for kommunikasjonsbiten av svermen er å starte kommunikasjonen på alle båtene for så å se på en båt hvilke verdier den får inn. Dataene den båten får inn er ikke spesielt viktig for oss, det viktigste er at den får like mange elementer inn som det er andre båter koblet opp. For å teste startet vi systemet som det var og simulerte data fra 4 båter samtidig en på hver båt, det er da forventet at hver båt skal motta data fra 3 andre. Resultatet er vist i Figur 5-16, det er første linjen av terminalen som er interessant for å konkludere denne testen. Her vises det at båten mottar data fra 3 andre båter, hvilket er ønsket resultat. Dette resultatet gjør at testingen kan fortsette et steg videre til å prøve ut forskjellige atferder til systemet og hvordan de fungerer sammen.

```
('elements in list: ', 3)
('clist: ', [{'bearing': 153.0, 'distance': 6.0828142577114 98, 'relative': 200.01533535769, 'y': 3.0445835190941932, 'x'
266036506622253, 'speed': 0.05}, {'bearing': 155.0, 'distance': 5.316527315449961, 'relative': 196.26095123815043, 'y':
3732664884277, 'x': 5.295678585988634, 'speed': 0.08}, {'bearing': 109.0, 'distance': 5.912677828483066, 'relative': 184
6229852153, 'y': -4.454123799834041, 'x': 3.888513890831199, 'speed': 0.1}])
```

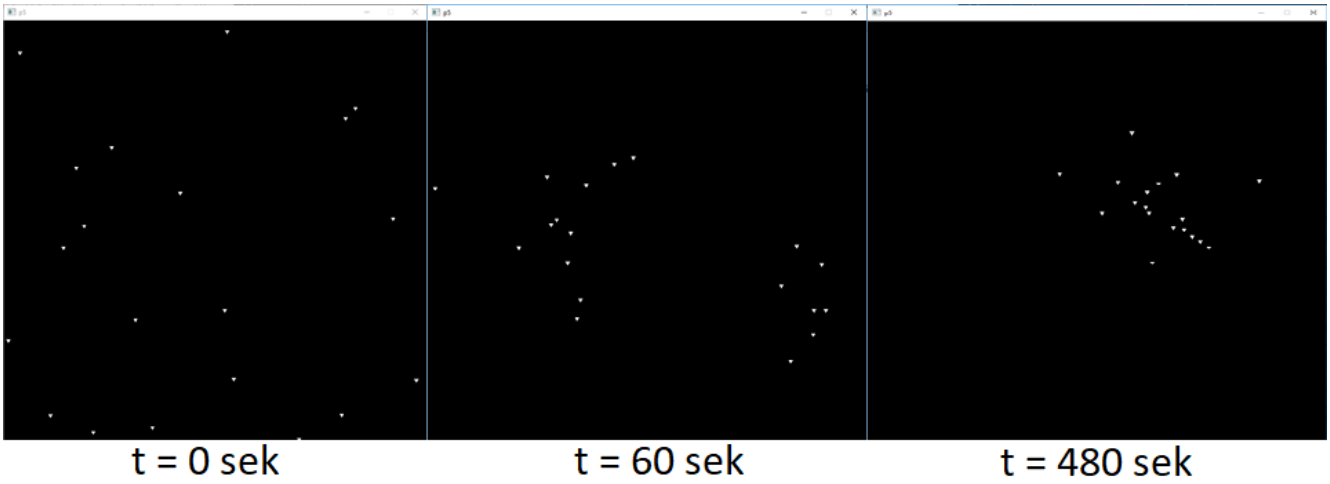
Figur 5-16: Terminal informasjon fra kommunikasjonstest med 4 båter

5.4 Behaviour

Behaviour-koden er hjertet i svermen. For at båtene skal drive sverm må vi forsikre oss om at koden henter inn informasjon fra seg selv og de andre enhetene på en god måte, gjør de nødvendige utregningene i behaviour-koden og sender videre en ønsket vektor som autopiloten kan benytte for å sette en ny kurs.

5.4.1 Simulering av boids

For å bruke minst mulig tid på å sjekke om behaviour-koden gjør de riktige utregningene tar vi i bruk simulatoren som ble presentert i 4.6.2.1. for å se at fungerer som den skal har vi lagt inn 20 båter i simulatoren. Figur 5-17 viser tre stillbilder fra simuleringen. Båtene starter med tilfeldige posisjoner spredt i vinduet. Da vi har sagt at båtene ikke skal ta hensyn til annet enn det som er innenfor en radius på 100 pixler vil det bli naturlig at de først samler seg i mindre flokker. Denne samlingen vil gå relativt fort da det er mange båter i dette eksempelet. Når vi lar tiden gå ser vi at de finner hverandre og danner en flokk, noe som vi introduserte i avsnitt 2.6.2 om *Flocking*.



Figur 5-17: Tre simulatorvinduer ved ulike tidspunkt.

5.4.2 Data inn/ut

Selv om koden fungerer i simulatoren er det ikke gitt at koden fungerer som ønsket for den plattformen vi har utviklet. En test der vi ser på dataene behaviour-koden tar i bruk og sender videre, er derfor nødvendig. Vi trikset litt med koden for å vise den viktigste informasjonen fra behaviour-koden i terminalvinduet. Som Figur 5-18 viser henter den inn informasjon fra to andre USV'er, gjør utregninger for atferden boids; samstilling, tiltrekning og seperasjon. Hvorvidt utregningene er riktige er ikke relevant for denne testen, det viktige er at den sender inn og ut data i riktig format.

```
data from boat  0
{'bearing': 340.0, 'distance': 4.998952961012484, 'relative': 89.99996044178022, 'y': 0.0, 'x': 4.99895, 'speed': 0.5}
data from boat  1
{'bearing': 340.0, 'distance': 4.99168736736, 'relative': 29.68456662966446, 'y': 4.3366, 'x': 2.47201, 'speed': 0.5}
elements in BOIDS list:  2
alignment x:  -0.246254503194  y :  0.676578686966
cohesion x:  0.648766410929  y :  0.376583520409
separation x:  -0.864645945566  y :  -0.502381716244
```

Figur 5-18: Printing i terminalvinduet av viktig informasjon fra behaviour-koden.

5.5 Sverm

Når alle hoveddelene av svermen; Autopilot, Kommunikasjon og Behaviour fungerer for seg selv, kan de testes sammen i sverm. Det er utviklet 2 forskjellige atferder som kunne bli brukt til testing; Boids og PSO. Av de to benyttet vi oss av Boids i innledende testing da den virket mer naturlig å implementere i starten av plattformen.

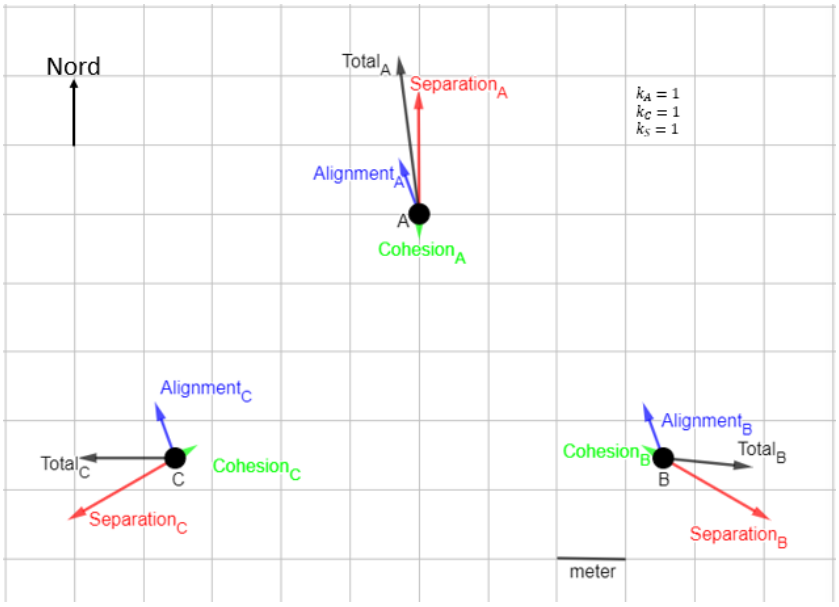
5.5.1 Første test av svermen på vannet

Første test av hele systemet på vannet ga oss ikke de resultatene vi ønsket å få, men den ga viste noen utregningsfeil i koden. Den første feilen var en vinkelregningsfeil. Utregningene av vinkelen på vektoren fra atferd som skulle til autopiloten ble gjort med sinus og cosinus, noe som kun gir oss vinkler i første og fjerde kvadrant (-90 til 90). Løsningen på dette var å gå over til å bruke funksjonen atan2(), en funksjon som returnerer det vi ønsker; en vinkel mellom -180 og 180 grader. Det er samme funksjon som vi brukte til å finne nærmeste vinkel i autopiloten i 5.2.1.1.Den andre feilen oppdaget vi da vinklene vi fikk ikke stemte overens med det vi trodde den skulle få. Det viste seg at vi vinklene var i radianer og ikke grader. En enkel konvertering løste problemet.

Siste feilen var at båtene hadde en tendens til å svinge til høyre. Det var fordi en funksjon i autopiloten som skulle forhindre ror-utslag langt over det rorene håndterer ga feil verdier over -45. Ror-utslag er satt av oss til $\lfloor 45, -45 \rfloor$. Alle vinkler over eller under disse verdiene blir satt til nærmeste verdi i intervallet. Problemet med koden vår var at når verdien til vinkelen var under intervallet ble den satt til 45, noe som gjorde at vinkelen hoppet ofte og ga konstant høyresving ved vinkler lavere enn -45 grader. Dette var også enkelt å løse så fort det ble oppdaget, ved å legge til en ekstra funksjon for returnere -45 for verdier under intervallet.

5.5.2 Kontrolltester

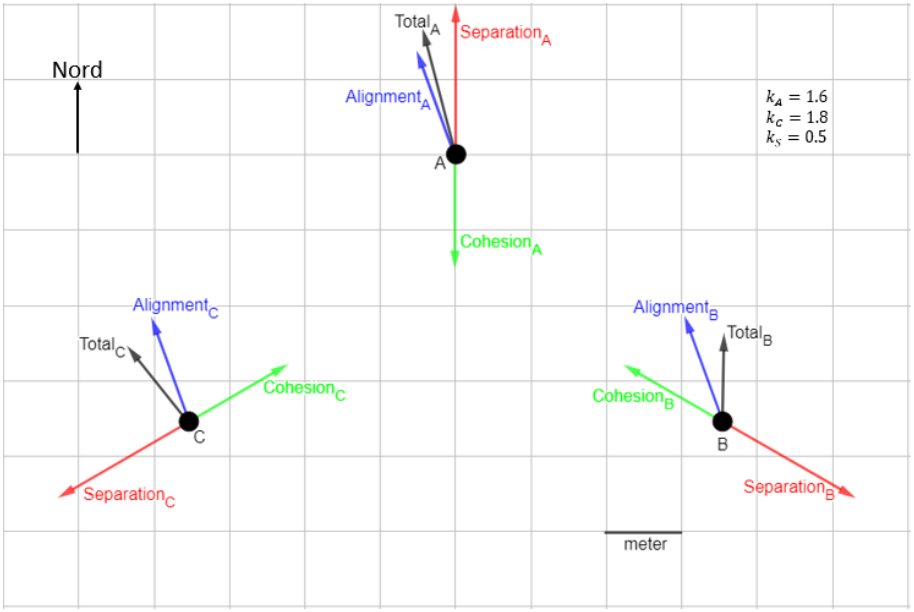
Når kodefeilene var fikset tok vi noen enkle tester på land for å forsikre oss om at atferden *Boids* returnerte det som var forventet. Vi simulerte 3 båter med statiske GPS-koordinater, fart og bevegelsesretning. Det var fordi testen skulle gjøres innendørs og det der vanskelig å få stabile GPS-data. Båtene ble simulert i en perfekt trekant, 5 meter unna hverandre, med en bevegelsesretning på 340 grader. For å illustrere hvordan de ønsker å bevege seg har vi brukt vektorene atferden bruker til å kalkulere ønsket bevegelse og plottet dem i Geogebra i Figur 5-19. Målet med testen var ikke kun å forsikre oss om at feilene fra første test var løst, men også å se hvordan båtene ønsker å bevege seg.



Figur 5-19: Krefter som påvirker ønsket bevegelse for hver båt med avstand 5 meter.

På denne måten kunne vi finstille konstanter for å få vektorer som passet med avstanden vi ønsket å teste med. For å hindre kollisjoner måtte det være en nøytralsone på minst 5 meter rundt farkosten. Med nøytralsone menes forskjellen på tiltrekning og frastøting skal være tilnærmet lik null ved 5 meters avstand.

Testen viser at frastøting er den dominerende vektoren, som gjør at de to andre får minimal påvirkning på totalvektoren. Vinkelen på vektorene viser at problemene fra tidligere test ble løst av endringene i koden. For å oppnå en nøytralsone på 5 meterer senket vi frastøting og økte samstilling og tiltrekning fra henholdsvis 1 på alle til 0.5, 1.6 og 1.8 til neste test.

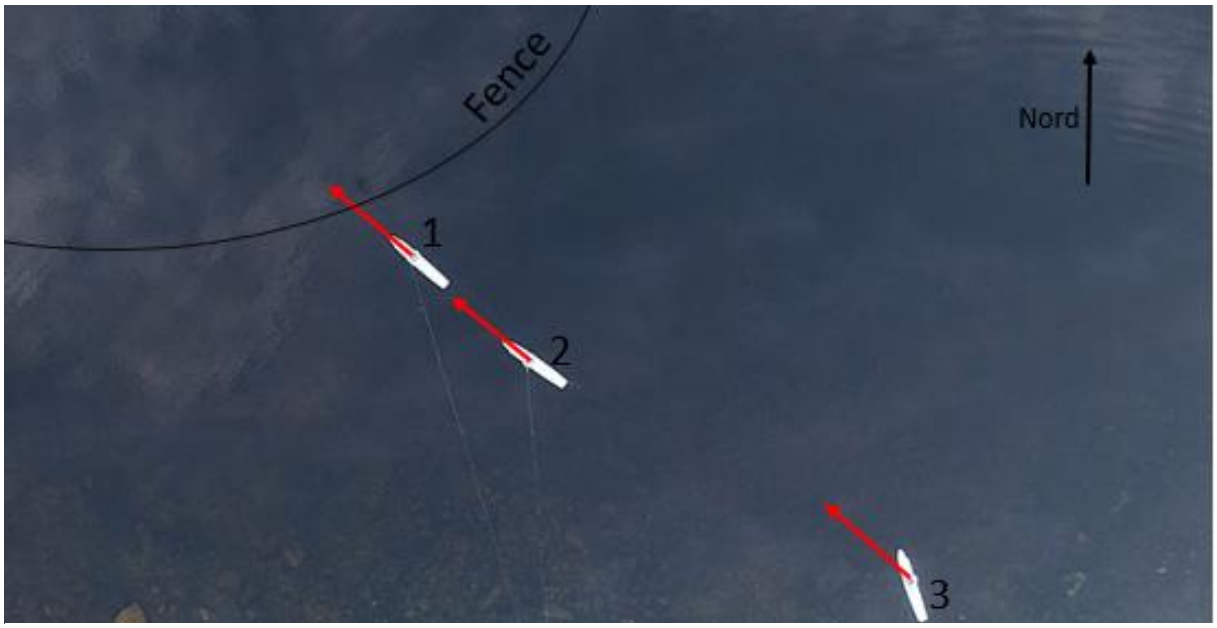


Figur 5-20: Ny test av kreftene som virker på ønsket bevegelse for hver båt med avstand 5 meter.

Med nye verdier $k_A = 1.6,\ k_C = 1.8$ og $k_S = 0.5$ har vi fortsatt en sterk frastøting, som vist i Figur 5-20. Siden verdiene ellers ser veldig bra ut, velger vi å si oss fornøyd med konstantene til neste test på vannet.

5.5.3 Andre test av svermen på vann

Målet med denne testen var nok en gang å teste plattformen sin evne til å operere sammen som en sverm, testen ble gjort med boids som atferd. Alle USV'ene ble satt ut samtidig med 2 meters avstand langs elvebredden. Når de starter er de utenfor den innlagte geografiske avgrensningen. Figur 5-21 illustrerer at båtene beveger seg mot avgrensingen, som er tegnet inn med beskrivelsen *Fence*. Reglene de skal operere etter i behaviouren er ikke i spill enda, det skjer første når det er innenfor avgrensingen.

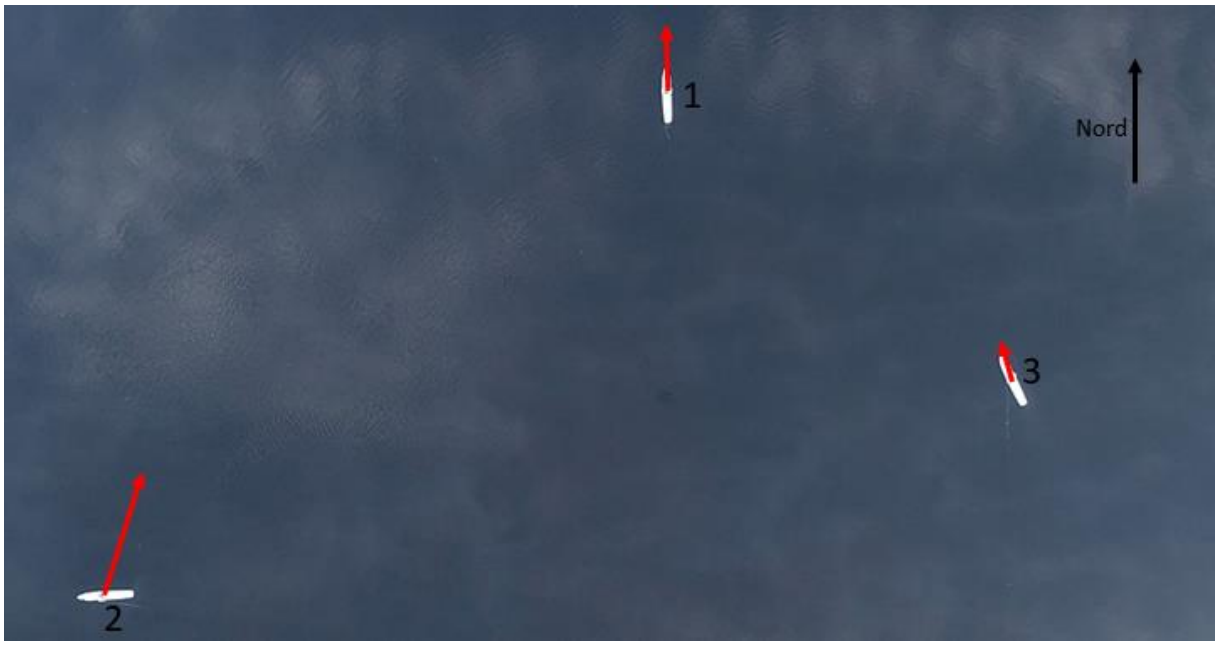


Figur 5-21: Drone bilde av starten på test 2, sverm på vei inn i avgrensning

Figur 5-22 viser når båtene kommer inn i avgrensninger starter behaviour-koden å gjøre atferds-utregningene. Siden USV1 og USV2 er innenfor noen meters avstand av hverandre når de kommer inn blir de kraftig frastøtt hverandre og setter vektorer vekk fra hverandre. Hver båt har en sikkerhetsline som håndteres av noen på land, USV-3 får et problem med sin snor, så den blir stående på samme lokasjon resten av testen. Det vi derimot ser er et forsøk på å sette en kurs rett nord, hvilket er riktig retning gitt de andre USV1 og -2 sin plassering.

Figur 5-22: Illustrasjon hvordan USV'ene reagerer når de kommer inn i avgrensningen. Båt 3 får problem med snor

I Figur 5-23 vises posisjonene etter 30 sekunder, USV1 og USV2 har fått litt avstand fra hverandre begynner samstilling til USV2 å virke inn sammen med tiltrekning. USV2 setter derfor samme kurs som USV1 og USV3. USV3 prøver fortsatt på å sette en kurs nordover, men blir holdt tilbake av snoren.



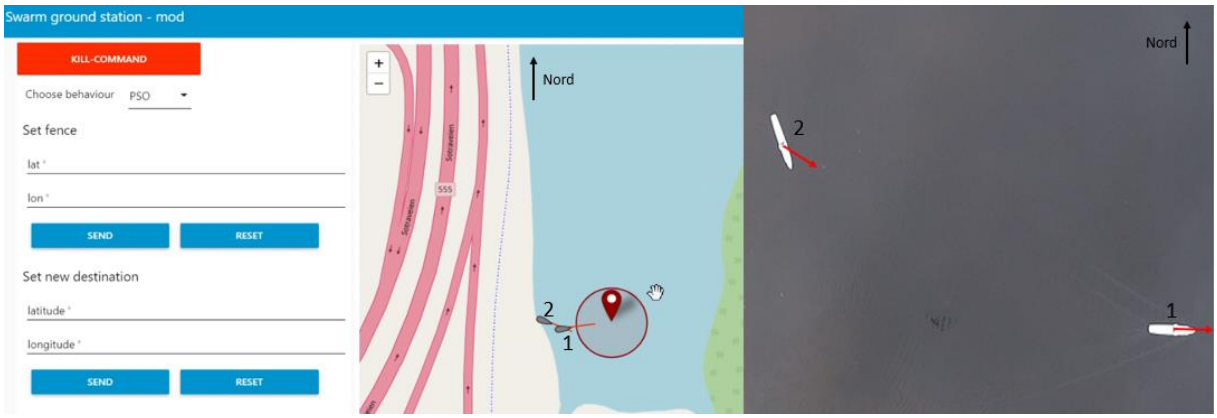
Figur 5-23: t=30 sek: Båt 2 fortsetter med å tilpasse avstanden til de andre USV'ene.

Som illustrert i Figur 5-24, har USV'ene funnet en slags formasjon og beveger seg alle i samme retning, men med større avstand enn nøytralsonen med avstanden på 5 meter. Det viser seg, med de avstandene vi får fra denne testen, at den er større i vann. En ny justering av konstantene kan derfor være aktuelt, men resulatene gir oss det vi ønsker; en sverm av 3 USV'er som fungerer.

Figur 5-24: t=60 sek: USV'ene finner en likevekt i svermen

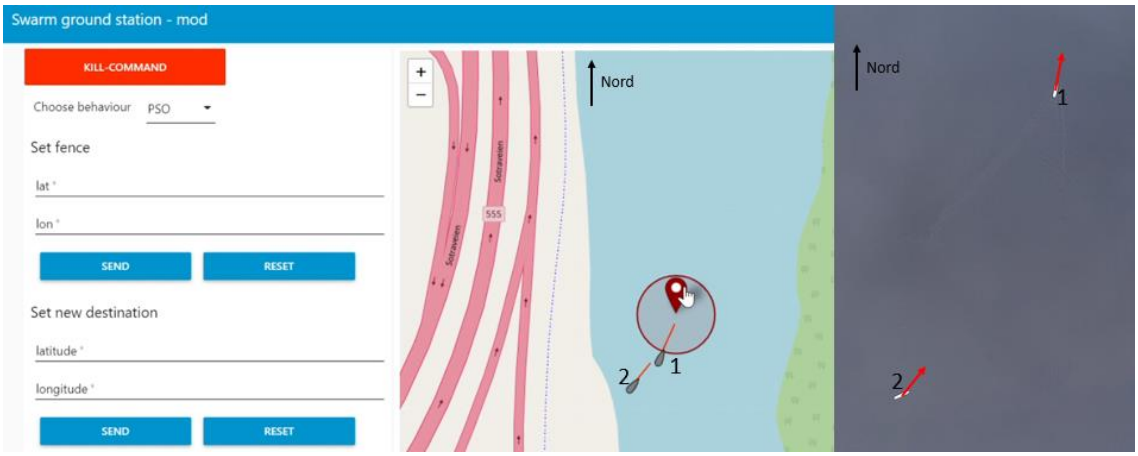
5.5.4 Tredje test på vannet – med basestasjon

Når plattformen er vist å fungere for sverm var neste å implementere basestasjonen som ble utviklet for overvåkning og overordnet kontroll av USV'ene. Basestasjonen gjør at svermen kan flyttes dit vi ønsker uten å måtte stoppe systemet og hard-kode det det inn. Testene går ut på å observere posisjoner i basen opp mot virkeligheten samt å flytte den geografiske-avgrensningen. Målet er å se om svermen evner å flytte seg til et nytt område og da kjøre atferd i den nye avgrensingen. Etter noen lengre mislykkede tester av funksjoner, som å bytte atferd fra basen opplevde vi problemer med vanninntrenging på noen av USV'ene. Derfor ble siste test utført med kun to båter.



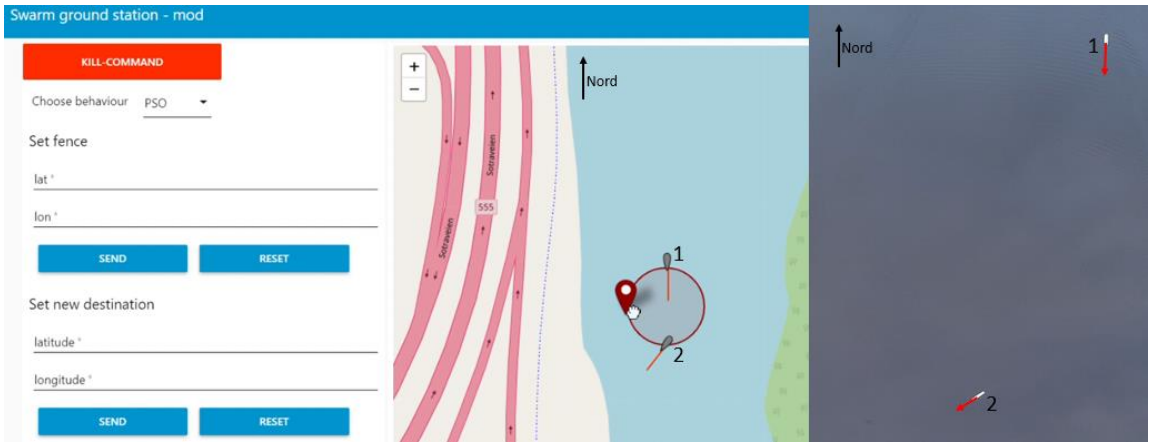
Figur 5-25: Start av siste test – USV'er beveger seg mot avgrensingen

Venstre side av Figur 5-25 som er laget for denne testen er stillbilde av basestasjonen, mens høyre side er dronebilde fra samme tidspunkt. I starten av testen er avgrensingen satt rett utenfor der båtene settes ut. Som Figur 5-25 viser beveger begge USV'ene seg mot avgrensingen umiddelbart. Når USV1 kom innenfor avgrensingen flyttet vi den lengre nord for å se om de endret retning mot det nye området. Som Figur 5-26 viser, satt de umiddelbart retning mot det nye området. Dette viser at de evner å motta kommandoer med endringer til atferden og tilpasse seg disse.



Figur 5-26: Flytter avgrensingen, USV'ene følger etter.

Siste del av testen, vist i Figur 5-27, viser at USV2 kommer inn i avgrensingen samtidig som USV1 beveger seg utenfor den på motsatt side. USV2 er på den avstanden at tiltrekning og seperasjon er tilnærmet like store. Det er da primært samstilling som styrer kursen til USV2. Det vises tydelig ved at USV2 snur rundt og setter tilnærmet lik kurs som USV1 holder på vei inn igjen i avgrensingen. Det fungerer med basestasjon og atferd for 2 båter i sverm, men ser fortsatt at seperasjon er for kraftig selv etter justeringene i 5.5.2. Den er for stor fordi de søker fra hverandre på tilnærmet lik avstand som avgrensingens diameter, det betyr at det ikke vil være mulig å observere en god atferd i et så lite avgrenset område. En økning av den geografiske avgrensingens radius blir gjort i etterkant for å gi de mer rom til å operer; ideelt sett skulle det vært gjort flere tester med mer variasjon i konstanter *Ka*, *Ks*, *Kc*, men antall båter og tiden var ikke med oss. Det vi trekker med oss fra siste test er at plattformen fungerer for flere båter i sverm, med basestasjon. Neste steg hadde vært en finjustering av variablene til atferdene.



Figur 5-27: Begge USV'ene inne i avgrensning. Setter kurs samme vei, men frastøtes også ganske kraftig

6 Drøfting

I dette kapittelet skal den produserte plattformen diskuteres og drøftes opp mot kravene som ble satt. Svakheter og styrker som ble oppdaget underveis skal også diskuteres, før så potensialet og brukbarheten av lignende systemer skal drøftes i forhold til en generell maritim kontekst.

6.1 Drøfting av plattform

I Tabell 2 ble det presentert krav til det som kreves av et system for å bli ansett som en maritim sverm. Tabell 4 viser måloppnåelse på hvert krav som ble stilt.

Krav	***Måloppnåelse***
Enkle enheter	Det er brukt enkle småbåter med en sensor for posisjon og bevegelse (PX4), en lettere PC(RPi) og en Arduino for å hente inn data, gjøre utregninger og styre etter valgt adferd.
Skalerbar	Hver USV er lik alle andre og tar kun stilling til andre USV'er eller basestasjoner som sender den data, på den måten kan vi fjerne og legge til båter i svermen etter ønske.
Parallell	Alle båtene kjører samme kode og baserer utregninger på lokal data fra systemet. Dette gjør at alle tar sine egne valg parallelt og skaper en felles adferd siden valgene baserer seg på like regler.
Kommunikasjon	Kommunikasjonen er enkel både eksternt og internt. Internt deles all data over ROS[11] topic'er som beskrevet i 4.5.3. Eksternt går kommunikasjonen med multicast over Wi-Fi som beskrevet i 4.5. Alle båter og baser i svermen kommuniserer med hverandre og deler kontinuerlig data i vårt oppsett.

[11] ROS – Robot Operating System

Desentralisert	Hver USV har en egen autopilot som blir styrt av en adferd, basert på en felles kode. Ingen form for ekstern styring er nødvendig, men svermen er mulig å kommandere til ønsket lokasjon gjennom basestasjonen.
Modularitet	Som presentert i 4.3 er kodestrukturen bygd opp av hovedfiler og støttemoduler som løser funksjonaliteten. Dette betyr at koden er svært modulære.

Tabell 4: Måloppnåelse på de 5 kravene som blir stilt til en sverm i 3.1.

Testing av svermplattformen viser at den er *enkel*, *skalerbar*, *parallell*, evner å *kommunisere* og er *desentralisert*. Den konklusjonen baserer seg på de korte forklaringene i Tabell 4 som skal drøftes nærmere nå og videre diskutere eventuelle svakheter og styrker med plattformen og oppsettet.

Båtene er basert på et enkelt modellbåtskrog med på motor og ror påmontert, som beskrevet i 4.1. Et overhus er bygd for å romme båtens eneste sensorpakke for miljøoppfatning. Dette gjør hver båt enkel å produsere, men medfører også begrensede kapasiteter når det kommer til miljøoppfatning. Legger man på flere sensorer på båtene vil evnen til å skape et korrekt bilde av virkeligheten øke. Våre fartøy henter kun inn GPS og bevegelsesdata. Dette er gjort for å gjøre utgangspunktet for vår maritime sverm minst mulig kompleks, men også fordi det ikke var mulig å gjøre permanente endringer på båtene da de skal brukes til andre formål på skolen. Dette begrenset egenskapene vi kunne gi svermen, men bidro til å holde hver småbåt enkel. Hadde enheten vært mer kompleks med mer avanserte sensorer hadde det økt kapabiliteten, men også økt forbeholdene som måtte tas for å teste dem.

Jevnlig testing av plattformen har vært en stor suksess. Det har gitt oss muligheten til å justere koden underveis. Noe som ikke hadde vært mulig med de økte kravene til sikkerhet under testing av mer komplekse båter. Sikkerheten og effektiviseringen

testingen kunne vært å økt ved å implementere en bedre miljøoppfatning med en hinder-gjenkjenning. En slik hinder-gjenkjenning kunne enkelt bli implementert med en sirkulær distansesensor, men hadde gått ut over enkelheten ønsket for hver USV.

Enkle båter medfører også enkle skrog. En enkel løsning er brukt for å verne om elektronikken inne i skroget. Det er skrudd på et lokk, noe som fører til glipper langs kantene. Glippene gjør at vann lett trenger inn. Når skroget ikke er tett reduserer det samtidig båtens evne til å operere over lengre tid. Dermed vil det være en viktig prioritering å tette skroget hvis plattformen skal brukes i det operative. Gitt de rammebetingelsene vi hadde i forhold til båtene og var det i dette tilfellet lite vi kunne gjøre for å tette skroget. Derfor burde det heller anskaffes noe som kan slippes i vann og være der over lengre tid uten å ta inn vann.

Kommunikasjonsmetoden mellom fartøyer vi har valgt å benytte oss av i denne svermplattformen er trådløst Wi-Fi. Det er primært fordi det beste utgangspunktet å starte med for et prosjekt som dette, samtidig som det ikke krever installasjon av noe ekstra utstyr. For større tester blir rekkevidden til trådløst nett for svak. Når basestasjonen ble implementert for oppfølging av småbåtene ble rekkevidden et tydelig problem. Når fartøyene er på vannet må ruteren bli værende på land. Etter observasjoner på land ser vi at ruteren mister forbindelse med båtene når avstanden er på rundt 50 meter. Uten forbindelse med basen stopper fartøyene som en sikkerhetsmekanisme. For å øke rekkevidden kan en ruter med bedre rekkevidde brukes, eventuelt en radiosender hvis svermen skal testes over store avstander. Å gå over til en kraftigere ruter er en løsning som vil la svermen skaleres ytterligere, men det vil fortsatt være en begrensning når svermen skal dekke store arealer innad. Derfor kan en mer permanent løsning være å endre kilden til den interne kommunikasjonen. Bytte fra å ha en sentral enhet som alle kommuniserer med til at alle enhetene er en node i et felles nettverk, en form for mesh-nettverk. Typen MANET[12]. I tillegg til svak rekkevidde var en antatt utfordring med UDP multicast at pakker forsvinner, noe som kunne skapt utfordringer for systemet. Under

[12] Mobile Ad-Hoc Network

testene har vi ikke opplevd noen problemer med dette, noe som er en merknad for robustheten til svermen.

Forbindelsen over større avstander er ikke den eneste begrensningen til nettverket. Hvis svermen skal ta inn flere båter, noe som kan være nødvendig for å realisere et konsept kan det, som presentert i 4.5, ha en betydning for prosesseringskapasiteten til ruteren og enhetene. Flere forbindelser fører til en stor økning i data som skal sendes gjennom systemet. Dette kan fikses med å kun hente inn data fra de nærmeste naboene. Dette kan bli implementert med å legge inn en oppfatningsradius i koden, som gjort med boids i 4.6.2. En annen løsning kan være en skalering av systemet med sterkere maskiner og bedre kommunikasjonsløsninger. En slik løsning vil ikke nødvendigvis føre til en økning av størrelse på båtene, men vil betraktelig øke kompleksiteten til hver enhet, spesielt med en endring av kommunikasjon.

En annen faktor som sier noe om kvaliteten på plattformen er autopiloten. Når avgjørelser blir tatt er det viktig at svermen innretter seg nøyaktig. Vårt styringssystem er utviklet som en PID-regulator, men brukes kun som en P-regulator. Som nevnt i 4.4.1 ønsket vi en regulator som er så enkel som mulig med minst mulige variabler å moderere. Etter testene der P-regulatoren fungerer og er enkel å moderere, kan man si at den er ideell som desentralisert styring for systemet vårt. Basestasjonen virker som et sentralisert alternativ til styring av svermen. Den er i seg selv ikke en del av svermen, men hjelper med forståelse av systemets bevegelser og sikkerhet under testing. For videre arbeid med plattformen vil basestasjonen kunne ha en sentral rolle.

Basestasjonen er den eneste delen av svermen som ikke har felles distribuert kode. Dette kan bidra til å enkelt skalere systemet, siden det eneste som må gjøres for å øke antall enheter i svermen er å kopiere over koden til ny enhet. Samtidig er det også likheten som gjør plattformen parallell og robust for feil i enheter. Alle i svermen har sin egen data og gjør beslutninger basert på disse, men alle beslutninger baserer seg på de samme grunnreglene lagt inn i atferdene. Sett i lys av kravene satt til plattformen er produktet parallelt i aller høyeste grad, men det er ikke nødvendigvis slik at dette er eneste måten å

implementere en form for parallellitet i systemet. Alle enhetene i svermen må ikke jobbe parallelt med samme oppgave for å være en sverm. Svermen kan gjøres multifunksjonell ved å ha flere forskjellige atferder på forskjellige enheter samtidig. Svermer kan bestå av flere under-svermer; grupper av enheter som driver samme adferd. PSO og beeclust, som beskrevet i 2.6.3 og 2.6.4, Disse undersvermene kan dele informasjon og på den måten samarbeide parallelt med forskjellige deloppgaver.

Eksempelvis kan man utstyre et utvalg av USV'er med bedre sensorer for miljøoppfatning på lengre avstander, mens et annet utvalg blir utstyrt med kameraer for bildebygging. På den måten kan man bygge opp en dynamisk sverm for å finne, markere og beskrive større arealer i vannoverflaten. Utvalget med god miljøoppfatning kan markere abnormaliteter i overflaten på et felles kart for svermen, for så at de nærmeste med kamera installert kan nærme seg det punktet for å få et visualisere situasjonen. Denne informasjonen kan sendes til en basestasjon der en operatør kan navngi bildene og markere hva det er. Alle elementene i et slikt eksempel er til stede i vår plattform, med unntaket av sensor kapasitetene og andre mindre justeringer for flere funksjoner samtidig. Ved å dele opp sensorene på forskjellige grupper i svermen kan vi holde hver enhet enkel, mens gruppen blir kompleks og kan løse større oppgaver.

Målet med plattformen er at den skal kunne muliggjøre utviklingen av ulike atferder for maritim sverm. For at det skal være mulig å utvikle forskjellige atferder må plattformen og systemet bak være robust og lagd for å bygges på. Siden testresultatene viser at systemet sjeldent stopper på grunn av en ukjent feil kan vi si at systemet er robust. For at plattformen skal være lett å utvikle videre er vi tilbake på poengene fra kapittel 4.3 om modularitet. Det som er skrevet av programmer til systemet er godt dokumentert og bygd fra bunnen opp med moduler som byggeklosser. Systemet kan tilpasses til å kjøre flere forskjellige atferder, eksempelvis PSO som ble utviklet, men ikke testet. Dette er en stor styrke for plattformens utvikling, at man kan ta atferder fra simulatoren direkte inn i filstrukturen for å teste i reelle omgivelser med reelle data.

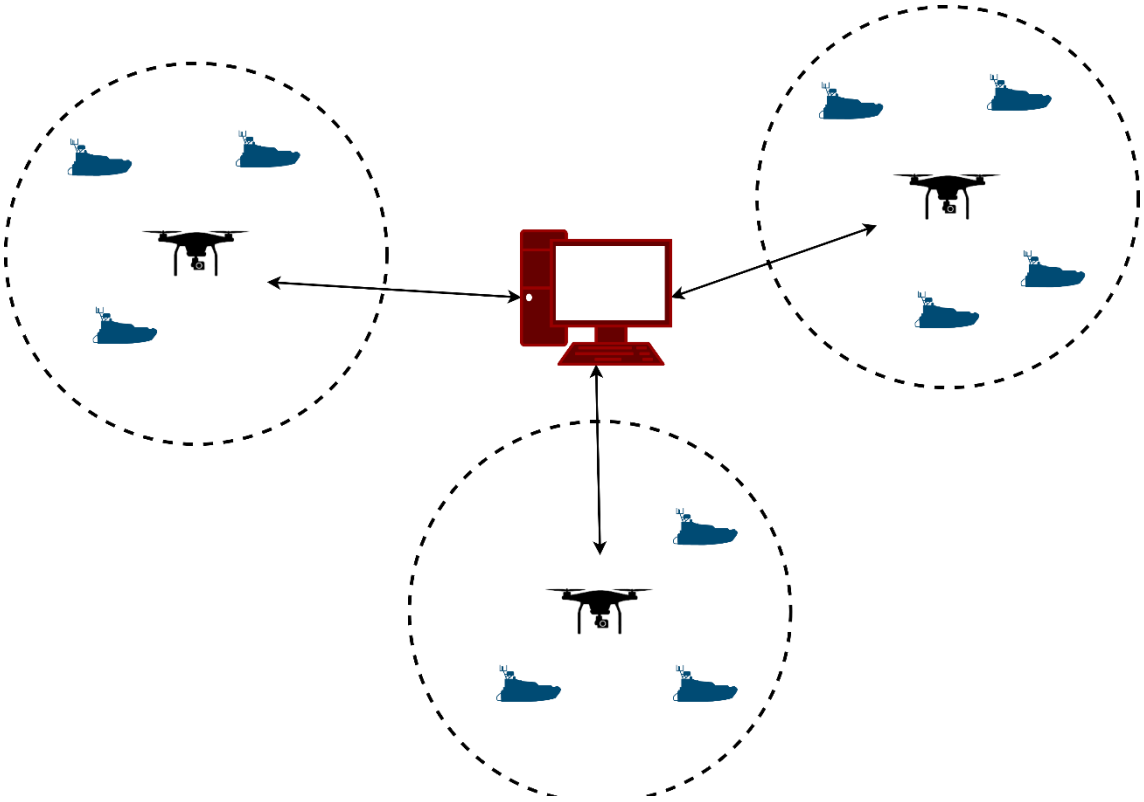
Uten tester kan ikke kravene vurderes. I løpet av perioden har derfor testing blitt en sentral del av oppgaven. For å forhindre tap av båtene ble det blitt gjort sikkerhetsmessige tiltak som har gjort testingen utfordrende. En sikkerhetssnor ble festet i hver båt før basestasjonen var ferdig utviklet. Dette førte til at det ble et behov for bemanning av sikkerhetssnorene, en bemanning som krevde langt mer enn to personer. Tidlig utvikling av en basestasjon med tilhørende returfunksjon kunne gjort testingen lettere fra start enklere og redusert behovet for økt bemanning.

Etter drøftingen som er gjort et er det helt klart at plattformen fungerer godt som en testplattform, men hva skal til for at den kan anvendes i det operative? Da skroget ikke er vannsikkert, vil det redusere plattformens evne til å løse oppdrag over lengre tid. Skroget burde være så tett at er mulig å kaste båtene i vannet. Videre må en bedre miljøoppfatning i form av hindergjenkjenning og sensorkapasitet som kan bidra til visualisering av området rundt svermen vil implementeres. Det er også et behov for å øke forbindelsesrekkevidden med eksempelvis bruk av en radiosender. Med modifikasjonene beskrevet ovenfor er det kun framtidens krigføring som setter grenser for potensialet til plattformen!

6.2 Potensialet til sverm

Konseptet sverm ble presentert allerede på 1980-tallet (Reynolds, 2001), men har ikke vært aktuelt før i nyere tid hovedsakelig på grunn av tre ting; Små men kraftige datamaskiner har ikke eksistert, kommunikasjonshastighet har vært for dårlig og små billige sensorpakker har vært vanskelig å oppdrive. Selv om det er mulig å realisere med dagens teknologi, er det stadig nye utfordringer som må løses for at sverm skal bli et solid og robust konsept. En av de første utfordringene oppstår når svermen skal skaleres med opp mot 100 enheter. Tanken som styrer skalering og utvikling av sverm er at det skal bestå av mindre enheter som droner, småbåter eller en kombinasjon, noe som fører til at de ikke kan bære store datamaskiner med mye datakraft. Mindre maskiner, som for eksempel RPi, blir da ofte brukt lokalt for å håndtere farkosten. De har gjerne begrenset kapasitet til å behandle data, noe som begrenser hver enhet sin evne til å håndtere input

fra mange andre farkoster samtidig. Det å kombinere domener kan være en løsning. Svermen kan bli delt opp i mindre deler og bruke droner til å danne et nettverk av forbindelser. Småbåtene vil dermed kun forholde seg til de andre båtene som er forbundet til samme drone. Figur 6-1 er ment til å vise hvordan et slikt system ville fungert.



Figur 6-1: Sverm av enheter i forskjellige domener (drone hentet fra (CleanPNG.com, uå) og båt hentet fra (L3HARRIS, 2019).

Basert på utviklingen i teknologien som ser ut til å fortsette og nasjoners fortsatte militære satsning på større og mer avanserte enkeltplattformer er det ikke umulig å se for seg at sverm kan ha en plass i framtidens maritime krigføring. Både som en naturlig motpart, men også som en forlengelse av sensorer, til de store og avanserte enkeltplattformene. "Flere oppgaver som er vanskelige eller tidkrevende for en enkelt farkost alene, kan effektivt utføres av en sverm med relativt enkle farkoster (FFI, 2019)". FFI har forsket på sverm i flere år og ser for seg at rollene sverm kan fylle er metning og dekning. Metning handler om korte og intensive operasjoner der man sender ut en sverm i håp om at deler av svermen skal løse oppgaven. Metning kan eksempelvis være nytt for oppdrag som innebærer informasjonsinnsanking i et fiendtlig område. Et stort antall enheter blir sendt

inn fiendtlig området i håp om at noen av de skal kunne hente inn informasjon i området og returnere. Ved å sende et stort antall individer mot fienden er målet å skape kaos og usikkerhet hos fienden om de ikke kan håndtere alle samtidig, og potensielt klarer noen enheter å hente ut viktig informasjon. Det sendes derfor inn mer enn det man føler er nødvendig i den hensikt at fienden skal få problemer med å ta ut alle samme, slik at noen klarer å utføre oppdraget. For metning forventes det store tap av den totale befolkningen i svermen, derfor krever dette av hver enhet er billig å bygge.

Dekning bygger på at en stor oppgave blir fordelt mellom mange farkoster. Søk etter ubåt er et eksempel på en stor oppgave som kan strekke seg over store områder. Ser vi omgivelsene fra ubåtens perspektiv er det mye som tyder for at den har god kontroll på situasjonen hvis det kun er én farkost med én sonar som søker etter den. Derimot vil situasjonen bli mye mer kompleks for ubåten om det er 50 farkoster med 50 sonarer som søker. Svermen vil gi en langt større sensorkapasitet i området og HVU[13] kan ligge utenfor og overvåke situasjonen. Slik oppdragsløsning krever god samhandling og velutviklet atferd i svermen. Det bygger på evnen til å kommunisere data og miljøforståelse til resten av svermen og HVU utenfor området. Dekning med bruk av småbåter vil øke deteksjonsmulighetene og redusere risiko for tap av menneskeliv.

"Sjøforsvarets intensjon med autonomi bør i hovedsak baseres på å redusere risiko for tap av menneskeliv, samt effektivisering av operasjoner der mennesket er en restriksjon" (Hareide *et.al,* 2018). Det er ønskelig fra Sjøforsvarets side at satsningen skal få mennesket ut av situasjonen. Dette er ikke kun en satsning for å forhindre tap av menneskeliv, men også et viktig virkemiddel politisk for å beholde støtten hos befolkningen. Da mye av potensialet til bruk av sverm er som metning og dekning, ligger det også et stort potensial for å fjerne mennesket fra oppdrag av samme kategori. Figur 6-2 er ment å illustrere at sverm vil kunne ta piloter ut av krigssonen og erstatte dem med en sverm kommandør som står på trygg grunn. Problemet med en slik løsning er at mer av kontrollen overlates til teknologien og koden som er skrevet. Å overlate mye eller all kontroll til teknologi er, med hensyn til sikkerhet og dagens teknologi, problematisk.

[13] HVU - High Value Unit

Svermen må kunne håndtere alle situasjoner den skulle støte på uavhengig av operatør, noe som er fult mulig om det utvikles en samling av løsninger på situasjoner svermen kan bli stilt ovenfor.

Figur 6-2: Fra flere piloter i krigssonen til én sverm kommandør (basert på Giles,

Implementering av én svermalgoritme gjør at svermen kan drive én form for adferd. Problemet som da oppstår er at den kun er satt til å løse én form for oppdrag. Et rammeverk av flere algoritmer der man knytter hver adferd opp mot oppdrag, slik amerikanerne har gjort med MASC (skrevet om i 2.1), kan være en løsning. Siden taktikker ofte er standardiserte, er det mulig å utvikle taktikkalgoritmer for svermen. En samling av disse algoritme gjør at svermen kan løse flere oppdrag og dermed et mer allsidig konsept. For å kunne utvikle et slikt rammeverk må man ha muligheten til å teste taktikker og svermoppbygninger. Derfor er det viktig med plattformer, som vår, å kunne benytte seg av i utvikling av både oppdragsløsning og algoritmer.

Svermplattformer er allerede et satsningsområde med bruk av hovedsakelig droner. Det er nå sett på potensialet svermplattformen kan ha i det maritime domenet. Ubåtkrigføring, søk av områder og angrip av mål er bare noen av oppdragene en HVU[14] vanligvis må løse, men som i framtiden kan løses sammen med sverm av småbåter med mindre risiko for personell. Store krigsskip er lett å senke og koster mye, med mange småbåter er det vanskelig å ta ut, samtidig som at kostnaden per enhet er mindre. Utvikles det gode nok algoritmer som kan settes i system vil vi kunne ta mennesket ut av risikofylte situasjoner,

[14] HVU – High Value Unit

potensielt spare sjøforsvaret for milliarder og effektivisere oppdragsløsningen. Eksempler på algoritmer som viser stort potensial er PSO og beeclust, som også er beskrevet i 2.6.3 og 2.6.4. PSO bruker svermen til å finne optimale løsninger og kan dermed anvendes til ubåtkrigføring der svermen tolker data i et område til å finne en lokasjon ubåten mest sannsynlig befinner seg i. Beeclust gir svermen mulighet til å operere uten å kommunisere, noe som kan være aktuelt i mer sensitive oppdrag der det ikke er ønskelig at svermen oppdages. Sverm kan altså bidra til å løse mange av framtidens militære utfordringer.

7 Konklusjon

I denne oppgaven har vi arbeidet med å lage **en plattform for testing og utvikling av en maritim overflatesverm for Sjøforsvaret**. Plattformen oppfyller de målene som ble satt for ønsket funksjonalitet da den er *enkel*, *skalerbar*, *parallell*, *desentralisert* og *kommuniserer* internt og eksternt. Vi har i tillegg laget en basestasjon for overvåking og styring av systemet, som ikke var et mål fra starten, men hjalp stort med utvikling og kvalitetssikring av svermen.

Plattformen er testet med egenutviklet svermalgoritme som fungerer, og den er dermed klar for videre utvikling og testing med andre algoritmer. De største utfordringene knyttet til oppgaven har vært tilrettelegging av tester og utvikling av atferder som gir gode testresultater. Testing av svermen har vært en essensiell del av oppgaven, da dette er eneste måten vi får vurdert måloppnåelsen til plattformen. Praktisk utførelse av tester har krevd mye personell for å verne om sikkerheten til og forhindre tap av USV'ene. Mot slutten av utviklingsperioden da en basestasjon var utviklet og plattformen fungerte som ønsket var dette ikke lengre et behov. Det kan konstateres basert på resultatene fra tester at plattformen er klar for videre arbeid med å utforske mulighetene til sverm-intelligens.

Mulighetene for at sverm er en del av framtidens krigføring er stor om konseptet utvikling videre. Potensialet styrkes av mangelen på plattformer med samme strategiske muligheter som sverm viser til i testing og teori. Metning og dekning med mange små enheter som samarbeider er en retning i krigføringen som snur fokuset fra plattformintensivt til sensorintensivt. Et slikt skifte kan gi nye strategiske muligheter til de som besitter teknologien, derfor vil det å være en ledende nasjon i utviklingen av svermteknologi kunne vise seg å gi Norge en stor fordel i framtidens maritime krigføring.

7.1 Anbefalinger og videreutvikling

Framtidig utvikling av denne plattformen vil hjelpe kadettene til å bli kjent med framtidens teknologi samtidig som det gir Sjøforsvaret verdifulle data og ideer for utviklingen av en teknologi som viser stort potensiale. Derfor er plattformen bygget for å tilrettelegge for utvidelser og videre arbeid. Gjennom hele produksjonsprosessen har modularitet og robusthet vært et fokus for at det skal være mulig å legge til eller fjerne elementer fra systemet. Noen av disse mulige endringer inkluderer økt evne til miljøoppfattelse, skalering av kommunikasjon eller å kombinere atferder for å løse ulike oppdrag.

7.1.1 Økt miljøoppfatning

En utvikling av plattformen kan være å utvide dens sensorkapasiteter. Dette vil gi økte muligheter til hvilke atferder den kan håndtere og deretter åpne muligheter for å utvikle en enda bedre sverm. En slik utvidelse må nødvendigvis ikke utvide alle enhetene likt. En sverm bestående av to ulike typer enheter, som beskrevet tidligere i drøftingen, hadde også økt plattformens miljøoppfatning om de alle deler data med hverandre. En slik sverm bestående av grupper med enheter som har forskjellige sensorer og kapasiteter kan øke svermens operative potensial betraktelig.

7.1.2 Skalering av kommunikasjon

Noe av det som gjør sverm effektivt er muligheten til å avsette mange, små, mobile enheter i et operasjonsområde. Flere enheter gir større avstander, det krever en skalering av kommunikasjonen mellom enhetene. En slik endring kan oppnås ved å implementere en type mesh eller ad-hoc nettverk. Blant disse virker MANET[15] å være oppsettet med størst potensial for sverm. MANET bygger videre på funksjonaliteten til mesh-type nettverk. En slik skalering hadde økt antallet mulige enheter i svermen før båndbredde eller store avstander internt blir en begrensende faktor. Det er også mulig å skalere opp den eksterne kommunikasjonen mellom USV'er og basestasjonen, den kan eksempelvis

[15] MANET – Mobile Ad-Hoc Network

gjøres over radio. Overgangen fra Wi-Fi til radio kunne økt rekkevidden til svermen ved at den kunne operert med større avstand til basestasjonen. En økning i rekkevidde øker funksjonaliteten til svermen drastisk.

7.1.3 Kombinasjoner av atferder

Siden de to foregående utvidelsene presenter muligheter for å utvikle plattformen, vil en kombinasjon av atferder være en utvidelse av svermens kapabiliteter. Bakgrunnen for det er at oppdrag i det maritime domenet er komplekse og krever derfor en atferd som kan håndtere ulike situasjoner hurtig og effektivt. En slik utvidelse skaper en mer oppdragsorientert plattform. Det kan gjøres ved trekke ut ulike deler av atferder som eksisterer for å kombinere de til en som rettes mot et spesifikt oppdrag eller ved å lage en atferd basert på ulike faser i et oppdrag.

Bibliografi

1. Hareide, O. S. Relling, T. Pettersen, A. Sauter, A. Voll Mjelde, F & Ostnes, R. (2018). Fremtidens autonome ubemannede kapasiteter i Sjøforsvaret. *Necesse 2018.* https://www.semanticscholar.org/paper/Fremtidens-autonome-ubemannede-kapasiteter-i-Hareide-Relling/5c8150a75aad425f0aed355e02f2cf98c5a60937
2. Forvarets Forskningsinstitutt. (2016). FFI tar et teknologisk ansvar. Kjeller: FFI
3. Forsvarets forskningsinstitutt. (2016, 17.mars 2019). Sverm. Hentet fra http://2016.ffi.no/sverm
4. Department of Computer Engineering. (2015). Neurocomputing. *Ataturk University.*
5. Reynolds, C. (2001, 10.juni.2019). Boids. Hentet fra https://www.red3d.com/cwr/boids/
6. Python Software Foundation. (2019, 17.juni 2019). Python tutorial. Hentet fra https://docs.python.org/3/tutorial/classes.html
7. Open Source Robotic Foundation. (2017, 17.juni 2019). Writing a simple publisher and subscriber (Python). Hentet fra http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28c%2B%2B%29
8. Open Source Robotic Foundation. (2018, 17.juni 2019). ROS. Hentet fra http://wiki.ros.org/ROS
9. Networking Beginners. (2014, 19.juni 2019). TCP and UDP. Hentet fra https://www.youtube.com/watch?v=TKrTnPz7gvk
10. Raum, P. (2019, 15. august 2019). Selvkjørende Uber-Volvo klar for produksjon. Hentet fra https://www.motor.no/artikler/2019/juni/volvo-og-uber-med-ny-selvkjorende-modell/
11. ScienceDirect.com. (2019, 25.september 2019). Particle Swarm Optimication. Hentet fra https://www.sciencedirect.com/topics/engineering/particle-swarm-optimization
12. Preben S. Ottesen. (2019, 26.september 2019). Honningbier. Hentet fra https://snl.no/honningbie
13. QGroundControl.com. (2019, 01.desember 2019). QGroundControl User Guide. Hentet fra https://docs.qgroundcontrol.com/en/

14. Giles, K & Giammarco, K. (2017). Mission-based Architecture for Swarm Composability (MASC). *Procedia Computer Science 114, 2017.* https://www.sciencedirect.com/science/article/pii/S1877050917317994
15. Holm, K. (November 2003). *Multicast i nettverk med støtte for mobil IP* (Masteroppgave). Universitetet i Oslo Institutt for informatikk, Oslo.
16. Crockford, D. (2017, 24.november 2019). Introducing JSON. Hentet fra http://www.json.org/
17. Wikipedia.org. (2019, 03.desember 2019). PID-regulator. Hentet fra https://www.google.com/search?q=PID&rlz=1C1GCEV_enNO852NO852&oq=PID&aqs=chrome..69i57j0l2j69i59j69i60l2.622j0j7&sourceid=chrome&ie=UTF-8
18. Forsvarets forskningsinstitutt. (2019). Den autonome framtid. *Viten, 2019(1)*. Hentet fra https://www.ffi.no/publikasjoner/arkiv/den-autonome-framtid
19. Venaas, S. (2019, 25.november 2019). IPv6 Multicast Address Space Registry. Hentet fra https://www.iana.org/assignments/ipv6-multicast-addresses/ipv6-multicast-addresses.xhtml
20. Veness, C. (2019, 16.august 2019). Calculate distance, bearing and more between Latitude/Longitude points. Hentet fra https://www.movable-type.co.uk/scripts/latlong.html
21. Tan, Y. & Zheng, Z. (2013). *Research Advance in Swarm Robotics.* Key Laboratory of Machine Percetion and Intelligence. https://www.sciencedirect.com/science/article/pii/S221491471300024X
22. Zandie, R. (2019, 01.desember 2019). Simulating Bird Flock Behaviour in Python Using Boids. Hentet fra https://medium.com/better-programming/boids-simulating-birds-flock-behavior-in-python-9fff99375118
23. OpenWire. (2019). How far will your Wifi signal reach? Hentet fra https://openweb.co.za/how-far-will-your-wifi-signal-reach/
24. Martínez, C & Cao, Dongpu. (2019, 02.desember 2019). Integrated energy management for electrified vehicle. Hentet fra https://www.sciencedirect.com/topics/engineering/particle-swarm-optimization
25. Unmanned Systems Technology (UST). (2019). *L30 Fire Control and Rescue USV* [figur]. Hentet fra

https://www.unmannedsystemstechnology.com/company/oceanalpha/

26. CleanPNG.com. (u.å). *Airplane Silhoutte* [figur]. Hentet fra https://www.cleanpng.com/png-general-atomics-mq-1-predator-aircraft-unmanned-ae-765933/?fbclid=IwAR0sgbopbkFsvb3BN00ElsGeGgh--tzUMRPvO7IL-eFrbS71jY90OK5xa6w
27. L3HARRIS. (2019). *STAT-3-1-1* [figur]. Hentet fra https://www.asvglobal.com/home/stat-3-1-1/?fbclid=IwAR0PiCeEmZVmZpaBMpVdg63mNhMuX5EuTmVkZXgb7Z8ToCFkS_ySUDbFiFo
28. Sauter, Aleksander. (2019). *Skrog* [fotografi]. Sjøkrigsskolen. Upublisert.
29. Giles, K. (2017). *Unknown title* [figur]. Hentet fra https://sercuarc.org/wp-content/uploads/2018/08/SDSF2017_P5_Giles-SDSF-2017.pdf?fbclid=IwAR29jGTsGbOFE5Zt2fHALtOwtONdXF7BiSUAWAW-TuMCg09chk7RiDm2LOA
30. Challenger Aerospace Systems. (2019). *UNMANNED SURFACE VEHICLE (USV)* [figur]. Hentet fra http://kildekompasset.no/referansestiler/apa-6th.aspx
31. Bodi, M. Thenius, R. Szopek, M. Schmickl, T & Crailsheim, K. (2012). *Finite-state machine of the BEECLUST controlle* [figur]. Hentet fra https://www.researchgate.net/figure/Finite-state-machine-of-the-BEECLUST-controller-Boxes-represent-the-different_fig3_233895701
32. Schmickl, T & Hamann, H. (2011). *Unknown title* [figur]. Hentet fra https://www.semanticscholar.org/paper/Beeclust%3A-A-Swarm-Algorithm-Derived-from-Honeybees-Schmickl-Hamann/502a4880e050e4229b269ca42188bd8c7af15ff8
33. getfpv.com. (2019). *Holybro Pixhawk 4 Autopilot + Neo-M8N GPS + PM07 Combo* [bilde]. Hentet fra: https://www.getfpv.com/pixhawk-4-autopilot-and-neo-m8n-gps-pm07-combo.html
34. Peck, M. (2016). *Navy awards BAE next-gen electronic warfare contract* [figur]. Hentet fra https://www.c4isrnet.com/intel-geoint/sensors/2016/03/03/navy-awards-bae-next-gen-electronic-warfare-contract/

Vedlegg

Kommentar: Kodevedlegg er unnlatt i papirformat, ligger på vedlagt på minnepenn og på https://github.com/AnKIbach/swarm

På bakgrunn av at det er 3000+ linjer fordelt på 46 .py filer vil kun hovedfiler med tilhørende støttemodul bli lagt som vedlegg.

Vedlegg A - Operativsystem på RPi

Vi har benyttet oss av en lett-versjon av Ubuntu kalt Ubuntu MATE 18.04.3:

https://ubuntu-mate.org/raspberry-pi/

Operativsystemet uendret fra det som er i installasjonen, men vi har skrevet noen få systemregler for å løse diverse tekniske utfordringer.

Udev regler – disse er skrevet for at systemet skal kjenne igjen sensoren og arduinoen vi kobler til over USB og tilegne de samme symlink til seriellporten de får hver gang. Dette er viktig fordi koden avhenger av å kontakte samme seriellport hver gang.

Det gjøres etter standard: http://www.reactivated.net/writing_udev_rules.html

Vår regel i /etc/udev/rules.d/98-local.rules:
SUBSYSTEM==”tty”, SUBSYSTEMS==”usb”, ATTRS{manufacturer}==”3D Robotics”, SYMLINK+=“PX4”
SUBSYSTEM==”tty”, SUBSYSTEMS==”usb”, ATTRS{manufacturer}==”Arduino (www.arduino.cc)”, SYMLINK+=“Arduino”

Passord innlogging for brukeren av systemet er skrudd av, for at systemet skal gå rett tilskrivebordet etter oppstart.

SSH – er brukt for å fjernstyre båtene og starte systemet fra PCen som driver basestasjonen.

Installasjon gjøres på: https://help.ubuntu.com/lts/serverguide/openssh-server.html

Kommandoer:

Installer server på RPien som er på båten:
sudo apt install openssh-server

Sjekk status på SSH:

sudo service ssh status

Installer PuTTy på link – på Windows PC som skal logge på RPi eksternt:

https://www.putty.org/

Videre kan man logge inn på fartøyet over samme nett med IP adressen til fartøyet.

GIT - er brukt for å hente ut nyeste versjon av koden vår fra github, se github vedlegg.

Git installeres med kommando:

Sudo apt install git

Link: https://help.ubuntu.com/lts/serverguide/git.html

Kan være til hjelp å introdusere seg på nye systemer – se link over.

Viser til bildet under for hjelp til terminal kommandoer og lignende

CREATE

Clone an existing repository
```
$ git clone ssh://user@domain.com/repo.git
```
Create a new local repository
```
$ git init
```

LOCAL CHANGES

Changed files in your working directory
```
$ git status
```
Changes to tracked files
```
$ git diff
```
Add all current changes to the next commit
```
$ git add .
```
Add some changes in <file> to the next commit
```
$ git add -p <file>
```
Commit all local changes in tracked files
```
$ git commit -a
```
Commit previously staged changes
```
$ git commit
```
Change the last commit

Don't amend published commits!
```
$ git commit --amend
```

COMMIT HISTORY

Show all commits, starting with newest
```
$ git log
```
Show changes over time for a specific file
```
$ git log -p <file>
```
Who changed what and when in <file>
```
$ git blame <file>
```

BRANCHES & TAGS

List all existing branches
```
$ git branch -av
```
Switch HEAD branch
```
$ git checkout <branch>
```
Create a new branch based on your current HEAD
```
$ git branch <new-branch>
```
Create a new tracking branch based on a remote branch
```
$ git checkout --track <remote/branch>
```
Delete a local branch
```
$ git branch -d <branch>
```
Mark the current commit with a tag
```
$ git tag <tag-name>
```

UPDATE & PUBLISH

List all currently configured remotes
```
$ git remote -v
```
Show information about a remote
```
$ git remote show <remote>
```
Add new remote repository, named <remote>
```
$ git remote add <shortname> <url>
```
Download all changes from <remote>, but don't integrate into HEAD
```
$ git fetch <remote>
```
Download changes and directly merge/integrate into HEAD
```
$ git pull <remote> <branch>
```
Publish local changes on a remote
```
$ git push <remote> <branch>
```
Delete a branch on the remote
```
$ git branch -dr <remote/branch>
```
Publish your tags
```
$ git push --tags
```

MERGE & REBASE

Merge <branch> into your current HEAD
```
$ git merge <branch>
```
Rebase your current HEAD onto <branch>

Don't rebase published commits!
```
$ git rebase <branch>
```
Abort a rebase
```
$ git rebase --abort
```
Continue a rebase after resolving conflicts
```
$ git rebase --continue
```
Use your configured merge tool to solve conflicts
```
$ git mergetool
```
Use your editor to manually solve conflicts and (after resolving) mark file as resolved
```
$ git add <resolved-file>
$ git rm <resolved-file>
```

UNDO

Discard all local changes in your working directory
```
$ git reset --hard HEAD
```
Discard local changes in a specific file
```
$ git checkout HEAD <file>
```
Revert a commit (by producing a new commit with contrary changes)
```
$ git revert <commit>
```
Reset your HEAD pointer to a previous commit …and discard all changes since then
```
$ git reset --hard <commit>
```
…and preserve all changes as unstaged changes
```
$ git reset <commit>
```
…and preserve uncommitted local changes
```
$ git reset --keep <commit>
```

Vedlegg B – Github for kildekode lagring

For å kunne kjapt hente nyeste kode til alle båter samtidig fra en ekstern enhet har vi benyttet oss av github.com

Dette er et online repository lagringsside spesifikt designet for å samarbeide på kode på tvers av utviklere.

Vi har gjort bacheloren vår tilgjengelig for kloning offentlig på link: https://github.com/AnKIbach/swarm

Er det noen spørsmål til pakken – ta kontakt med forfatteren direkte.

Github benytter seg av git som språk for å interagere med deres servere fra ulike enheter, derfor er det viktig at hver enhet som skal bruke github har git eller github desktop installert.

Vedlegg C – ROS installasjon og bruk

ROS er brukt som intern-kommunikasjons plattform og system for utvikling og testing av svermen. Vi har brukt ROS melodic på både båtene og på utviklingsmaskinen vi har brukt, Vedlegg E – Virituell PC for utvikling av kode. Systemet skal nok også fungere med kinetic versjonen av ROS siden vi ikke har brukt mange versjonsspesifikke løsninger.

Ros melodic nedlasting :
http://wiki.ros.org/melodic/Installation

Vi har brukt melodic – siden det er den nye langtidsstøttede versjonen av ROS.

Mavros og PX4
MAVROS er en forlengelse til ROS som brukes for å dele data med sensoren Pixhawk 4.

Kan installeres med hjelp fra:
https://github.com/mavlink/mavros/tree/master/mavros#installation

Bortsett fra dette har vi opprettet et workspace kalt catkin_ws som standard i:

/home/<user>/catkin_ws

og kalt pakken med innholdet swarm i senere tid – denne kan alltids endres navn på. Ligger på:

/home/<user>/catkin_ws/src/swarm

Bruksanvisning og tutorial for ROS finnes på: http://wiki.ros.org/

Se bildet for hjelp med ROS kommandoer

https://twitter.com/intellrobotlabs/status/971006670531973121

ROS COMMANDS CHEAT SHEET v1.0

ROS.org

MASTER

Run a Master
$ roscore

NODES

Get list of nodes
$ rosnode list

Get info of a node
Given a node name, it return its subscriptions, publications and services
$ rosnode info [node]

Terminate a node
$ rosnode kill [node]

Test if a node is alive
Get latency and if node is alive given its name
$ rosnode ping [node]

TOPICS

Get list of topics
$ rostopic list

Get info of topics
Get publishers and subscribers, as well as the message type
$ rostopic info [topic]

Manual publish
Manually publication to a topic
$ rostopic pub [-r freq] [topic] [msg_type] [msg]

Visualize topic publications
$ rostopic echo [topic]

MESSAGES

Get list of available messages
$ rosmsg list

Show message structure
$ rosmsg show [msg_type]

SERVICES

Get list of available service types
$ rossrv list

Show service type structure
$ rossrv show [srv_type]

Show available running services
$ rosservice list

Show service type of a running service
$ rosservice info [srv]

Call a running service
$ rosservice call [srv] [srv_type] [request]

FILESYSTEM

change working package dir
Change to active workspace dir
$ roscd
Change to a dir starting from a package
$ roscd [package]
List package content
$ rosls [package]

Running a node
Requires a running Master
$ rosrun [package] [node]

Launch an application
$ roslaunch [package] [launcher]

CONFIGURE DISTRIBUTED SYSTEM

***Host**: machine that contains the master*
***Remote**: machine that connects to Host*
Both machines have to ping each other by name

In Remote:
$ export ROS_IP=[REMOTE_IP]
$ export ROS_HOSTNAME=[REMOTE_HOSTNAME]
$ export ROS_MASTER_URI=http://[HOST_HOSTNAME]:11311

Check for problems (even in package):
$ roswtf

DEBUGGING

Show nodes and topics connections
$ rosrun rqt_graph rqt_graph

Show TF transforms tree
$ rosrun rqt_tf_tree rqt_tf_tree

Show TF frames info
$ rosrun tf tf_monitor

Show TF transformation
$ rosrun tf tf_echo [frame_src] [frame_dst]

COMPILATION

Compile a workspace

Compile all the packages in a workspace with num_threads threads
$ catkin_make [-j num_threads]
Compile a specified pkg
$ catkin_make [-j num_threads] [--pkg package]

Intelligent Robotics *Lab*

www.inrobots.es
www.intelligentroboticslab.robotica.gsyc.es

Vedlegg D – Sensorpakke Pixhawk 4

Sensorpakken vi har brukt heter pixhawk 4 og er designet som en fullverdig autopilot og kontrollmodul for droner, vi har brukt den for å hente data ut utelukkende. Dette gjøres med Mavros i ROS Vedlegg C – ROS installasjon og bruk. For å kalibrere de bruker vi QGC som beskrevet i Vedlegg F – QGroundControl og i teksten Kalibrering av kompass i qgc, dette gjøres ved start av bruk og så ofte det er behov. Det er lett å sjekke hvor godt de treffer ved å sjekke opp mot en telefon eller lignende med hva man får ut fra basestasjonen. Vår kobling var med USB til hver RPi – det fungerer.

Link til brukerhåndbok:

https://docs.px4.io/v1.9.0/en/flight_controller/pixhawk4.html

Oversikt pinouts:

http://www.holybro.com/manual/Pixhawk4-Pinouts.pdf

Pixhawk 4

Pixhawk 4® is an advanced autopilot designed and made in collaboration with Holybro® and the PX4 team. It is optimized to run PX4 version 1.7, suitable for academic and commercial developers.

It is based on the Pixhawk-project **FMUv5** open hardware design and runs PX4 on the NuttX OS.

Quick Summary

- Main FMU Processor: STM32F765
  - 32 Bit Arm® Cortex®-M7, 216MHz, 2MB memory, 512KB RAM
- IO Processor: STM32F100
  - 32 Bit Arm® Cortex®-M3, 24MHz, 8KB SRAM
- On-board sensors:
  - Accel/Gyro: ICM-20689
  - Accel/Gyro: BMI055
  - Magnetometer: IST8310
  - Barometer: MS5611
- GPS: ublox Neo-M8N GPS/GLONASS receiver; integrated magnetometer IST8310
- Interfaces:
  - 8-16 PWM outputs (8 from IO, 8 from FMU)
  - 3 dedicated PWM/Capture inputs on FMU
  - Dedicated R/C input for CPPM
  - Dedicated R/C input for Spektrum / DSM and S.Bus with analog / PWM RSSI input
  - Dedicated S.Bus servo output
  - 5 general purpose serial ports
  - 3 I2C ports
  - 4 SPI buses
  - Up to 2 CANBuses for dual CAN with serial ESC
  - Analog inputs for voltage / current of 2 batteries
- Power System:
  - Power module output: 4.9~5.5V
  - USB Power Input: 4.75~5.25V
  - Servo Rail Input: 0~36V
- Weight and Dimensions:
  - Weight: 15.8g
  - Dimensions: 44x84x12mm
- Other Characteristics:
  - Operating temperature: -40 ~ 85°c

Additional information can be found in the Pixhawk 4 Technical Data Sheet.

Vedlegg E – Virituell PC for utvikling av kode

ROS og Windows spiller dårlig på lag sammen derfor kunne vi ikke teste kode skrevet på en vanlig Windows PC. RPi og koding dirkete spiller også dårlig sammen, på bakgrunn av RPi sin manglende CPU og dårlig evne til å vise grafikk bra. Derfor startet vi tidlig med en virtuell PC med Lubuntu (samme som var på RPi'ene før vi gikk over til MATE) for å teste og utvikle kode på. Dette er en veldig god løsning vi anbefaler til alle som skal jobbe med ROS, om ikke de bare bytter operativsystem på sin egen PC.

Vi brukte Oracle VM virtualbox for å drive det virtuelle systemet (vår eksakte VM er kopiert over og gitt til en lærer på skolen – du får den om du tar over oppgaven).

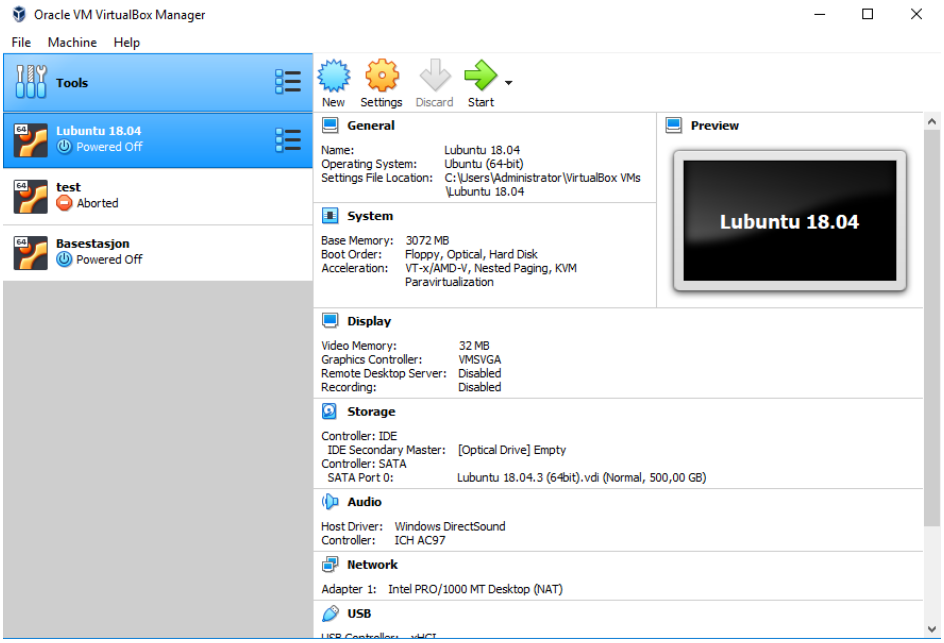
Virtualbox installeres fra: https://www.virtualbox.org/wiki/Downloads

Vi hentet et ferdig operativsystem Lubuntu fra Osboxes på link: https://www.osboxes.org/lubuntu/#lubuntu-18-10-info – vår installasjon var 18.04 Bionic Beaver

Videre installerte vi ROS og annen nødvendig programvare på lik linje som på hver båt.

I tillegg benyttet vi oss av VSCode fra windows på link: https://code.visualstudio.com/

Det er fordi det tilbyr god debug og utvidelser for samhandling med github – som kan lastes ned direkte i VScode.Under er et utklipp av innstillingene på vår virtuelle PC



Vedlegg F – QGroundControl

Dette er primært brukt til å installere ny firmware til Pixhawk'ene når de er rett ut av boksen og kalibrere de. Ellers ikke brukt, men fungerer godt og har flere funksjonaliteter som kan benyttes fra brukerhåndboken. Bildet under er hentet rett fra dronecode sin brukerhåndbok (link nederst på siden).

QGroundControl provides full flight control and vehicle setup for PX4 or ArduPilot powered vehicles. It provides easy and straightforward usage for beginners, while still delivering high end feature support for experienced users.

Key Features:

- Full setup/configuration of ArduPilot and PX4 Pro powered vehicles.
- Flight support for vehicles running PX4 and ArduPilot (or any other autopilot that communicates using the MAVLink protocol).
- Mission planning for autonomous flight.
- Flight map display showing vehicle position, flight track, waypoints and vehicle instruments.
- Video streaming with instrument display overlays.
- Support for managing multiple vehicles.
- QGC runs on Windows, OS X, Linux platforms, iOS and Android devices.

Nedlastning:

https://docs.qgroundcontrol.com/en/getting_started/download_and_install.html

Oppstart:

https://docs.qgroundcontrol.com/en/getting_started/quick_start.html

Link til brukerhåndbok:

https://docs.qgroundcontrol.com/en/

Vedlegg G – Tilleggsmoduler til Python

Tilleggsmoduler som bygger på Python sitt standard bibliotek og installasjonslinker, rekkefølgen er etter når man har behov for de – med unntak av PIP. PIP må installeres først da det er gjennom pip man installerer alle de andre tilleggene. I tabellen under er et forslag til rekkefølge å installere pakker i.

Navn	**Kommando**	**Link**	**Kommentar**
Pip	https://pip.pypa.io/en/stable/installing/	https://pip.pypa.io/en/stable/#	Repository for inneholdende pakker til Linux
Pyfirmata	Sudo pip install pyfirmata	https://pypi.org/project/pyFirmata/	Brukt til å sende data til Arduino
Psutil	Sudo pip install psutil	https://pypi.org/project/psutil/	Versjon 5.6.7, brukt til å lese ut CPU data fra RPi
P5	Sudo pip install p5	https://pypi.org/project/p5/	Versjon 0.6.0, brukt til grafisk display av simulator for adferd

Alle pakkene er hentet og installert gjennom pip installeren fra pip sitt repository med link: https://pypi.org/

Installasjoner gjøres gjennom Linux-terminalen i operativsystemet som beskrevet i operativsystem vedlegget - Vedlegg A - Operativsystem på RPi

NB – ROS må installeres og fungere på systemet før noe av dette går, les Vedlegg C – ROS installasjon og bruk

Pyfirmata brukes til å snakke med arduinoen, den inneholder en fil fra standardbiblioteket kalt StandardFirmata – den løser alt av styring og konvertering for oss

Vedlegg H – Node-red for basestasjon

Alt arbeid med Node-RED er gjort på en windows pc.

Node-RED nedlastning windows pc:

https://nodered.org/docs/getting-started/windows

Kjør Node-RED på windows pc:

https://nodered.org/docs/getting-started/windows#running-on-windows

I web-browser:

(egen IPv4 adresse):1880 eksempel: 127.0.0.1:1880 eller 192.168.136.67:1880

For å åpne user interface (som er brukt som addon for å visualisere basestasjon):

(egen IPv4 adresse:1880/ui eksempel: 127.0.0.1:1880/ui eller 192.168.136.67:1880/ui

For å importere arbeidet vårt:

Valgmeny øverst høyre -> import -> lim inn flow i clipboard

Flow ligger som fil i Gitbub strukturen som GCS_node-red.json – se Vedlegg B – Github for kildekode lagring og Figur 4-11.

Vedlegg I – Oversikt over båter brukt

NAVN (OG)	IP	BÅT	OS	NYESTE IMAGE?	ARDUINO	PX4	KOMMENTAR
B01 (05)	192.168.136.61	B01	MATE	JA	A01	P01	
B02 (09)	192.168.136.62	B02	MATE	Kopi	A02	P02	
B03	192.168.136.63	B03	MATE	Kopi	A03	P03	Vært under vann
B04 (06)	192.168.136.64	B04	MATE	Kopi	A04	P04	Vært under vann

Oversikten er ment for å gjøre det enklere å tolke hvilke komponenter som er brukt sammen og hvilke utfordringer som fører med. Alle komponentene er merket.

Oversikt programvare for RPi på båter

NAVN	KAN BRUKE GITHUB	ROS MELODIC	MAVROS	HAR ARDUINO KODE	SSH	INTERNETT	KONTAKT MED RUTER?	FAST SERIELL PORT?
B01	X	X	X	X	X	X	X	X
B02	X	X	X	X	X	X	X	X
B03	X	X	X	X	X	X	X	X
B04	X	X	X	X	X	X	X	X
FUNGERER VED			Launch		port 22	ice-net		udev-regel

Oversikten loggfører hva som fungerer av programvare på hver båt. "Fungerer ved" er en forklaring på hva programvaren fungerer gjennom. F.eks. Mavros fungerer gjennom roslaunch swarm system.launch eller roslaunch mavros px4.launch mer info om launch filer på http://wiki.ros.org/launch.

Vedlegg J – Oppstart og sekvensiell sjekkliste for test

OPPSTART AV BÅT FOR TEST

HVA	**Kommando**	**Annet**
LOGG INN		**brukernavn** b05 **passord** Bachelor
HENT NYESTE KODE:	git pull origin master	
OPPDATER ARBIEDSROMMET MED NYE FILER	catkin_make	innenfor mappen for arbeidsrommet
SOURCE SETUP FILEN	source /devel/setup.bash	gjøres automatisk i .bashrc
LUKK BÅTEN OG GJØR KLAR		
START SYSTEMET	roslaunch <pkg> <launchfil>	For dette systemet: roslaunch swarm system.launch
START ROSBAG	rosbag record -a	Tar opp data på topics på hele ROS
STOPP SYSTEMET	ctrl + c	Stopper hele systemet - kan gjøres fra basen
LASTE NED FILER FRA GITHUB		
HVA	**Kommando**	**Annet**
GÅ INN I ARBEIDSROMMET	cd <arbeidsrom>/src/	For dette er arbeidsrommet: catkin_ws
KLON FILENE FRA GITHUB	git clone https://github.com/AnKiBach/swarm.git	**Brukernavn** AnKiBach **Passord** 1Blirstor
OPPDATER ARBEIDSROMMET	catkin_make	

Stegbasert sjekkliste før, under og etter en test

FØR		UNDER		ETTER		UTSTYR	
1	Lade batterier	1	Skru på alt	1	Tøm båter for vann	1	Båt
2	Likt Image alle båt(ene)	2	Kalibrer PX4	2	Se gjennom ROSBAGs		- RPI
3	Hent nyeste kode til alle og test at den starter	3	Start MAVROS (skal skje automatisk)	3	Ta utklipp av nødvendig data		- Arduino
4	Båt(ene) og utstyr i bilen	4	Kjør kode med roslaunch	4	Data til rapport med kommentar		- PX4
5	Skru sammen båt(ene)		- roslaunch swarm system.launch				- Batteri
6		5	Start ROSBAG				- Lokk
							- Fiskesnor
						2	Ruter
						3	Fulladet PC

Vedlegg K – Test av ror-utslag

Hentet fra tidligere tester – beskriver maksimalt og anbefalt utslag på ror

Hva ble gjort:

1. Tester videre for å få en oversikt på vinkel satt og utslag for rorene. Se tabell for resultater.

Resultater og erfaringer:

1. Motor i midten er ikke i midten av kablene, men på en av sidene. Pin 10 er venstre motor.
2. Løst saken med servo og motorkontroll – må ha felles 0. pkt for jord i arduinoen.
3. Videre full-rekke test bekrefter tabellen under

Vinkel ut (fra kode)	Utslag (negativ er utslag mot babord)
10	Over max
20	Over max
30	Fysisk Max
40	Over max
50	Max
60	Anbefalt max
70	-20*
80	-10*
90	0*
100	10*
110	20*
120	Anbefalt max 30*90
130	Max
140	Fysisk max
150	Over max
160	Over max
170	Over max
180	Over max

Vedlegg L – Kildekode Autopilot og –caller

```
#!/usr/bin/env python
'''''
Swarmpilot.py
This is the main program swarmpilot,
it gets data from ROS with navData and swarmWanted
then calculates new movement with the caller module
finally publishes to ROS with talker

Run this as a node either standalone or as part of system.launch

Questions: anhellesnes@oslo.mil.no
'''

import time
import sys
import rospy

from Classes.GPS_class import GPS
from Classes.Vector_class import Vector
from Classes.Arduino_data import Arduino
from Autopilot_caller import Autopilot

from ROS_operators.Autopilot_sub import swarmWanted
from ROS_operators.Navigation_data import navData
from ROS_operators.Autopilot_talker import Talker
from ROS_operators.Autopilot_datasim import Sim # for simulation


def main():
    #definitin of status dictionary
    status = {'pixhawk': False,
                'arduino': False,
                'fix': False,
                'wifi': False}

    #initiating objects for autopilot
    nav = navData()
    autopilot = Autopilot()
    time.sleep(0.2)
    talker = Talker() # change between for sim or real
    # sim = Sim()
    time.sleep(0.2)
    arduino = Arduino('/dev/Arduino', speedLimit = 0.8) #speed limiter for test
ing
    time.sleep(0.2)
    behaviour = swarmWanted()
    time.sleep(0.2)

    wait_time, clicks = 0.0, 0

    #waits for both systems to connect
    while not nav.is_ready() and not arduino.is_ready():
        wait_time += 0.1
        time.sleep(0.1)
        if wait_time > 10.0: #exit if timeout is over 10s
            sys.exit(0)

    rospy.loginfo("Pixhawk is connected and ready: {}".format(nav.mode))
    rospy.loginfo("Arduino is connected and started at: {}".format(arduino.port
))
```

```
59.     rospy.loginfo("Behaviour is publishing data at: {}".format(behaviour.topic_
    main))
60.
61.     status = {'pixhawk': True,
62.               'arduino': True,
63.               'fix': False,
64.               'wifi': False}
65.
66.     print("entering loop...")
67.
68.     while not rospy.is_shutdown():
69.         try:
70.             current_GPS    = nav.get_GPS() #gets newest
71.             current_vector = nav.get_Vector()
72.
73.             if behaviour.is_recieving(): # check if behaviour is sending wanted

74.                 wanted = behaviour()
75.                 if isinstance(wanted, GPS):
76.                     wanted_vector = current_GPS.calculate(wanted)
77.                 else:
78.                     wanted_vector = wanted
79.             else: #if not recieving from behaviour stop USV
80.                 print("did not recieve")
81.                 wanted_vector = Vector(0.0, 0.0)
82.
83.             #sets wanted value for regulator
84.             autopilot.set_wanted_vector(wanted_vector)
85.
86.             #calculates vector for new mvoement
87.             change_vector = autopilot(current_vector)
88.
89.             #sends vector with new movmement to arduino
90.             arduino(change_vector.magnitude, change_vector.angle) #possible add
    ition
91.
92.             #publishes current data to ROS
93.             talker(current_vector, current_GPS, wanted_vector, change_vector) #
    change between for sim or real
94.             # sim()
95.
96.             #publishes status of USV every 20 clicks
97.             if clicks >= 20:
98.                 talker.publish_status(status)
99.                 clicks = 0
100.            else:
101.                clicks += 1
102.            #comment out for full test
103.            time.sleep(0.5)
104.
105.        except rospy.ROSInterruptException():
106.            arduino()
107.            sys.exit()
108.        finally:
109.            pass
110.    arduino() # to reset boat when exiting node
111.
112.if __name__ == "__main__":
113.    main()
```

```
1.  #!/usr/bin/env python
2.  """
3.  Autopilot_caller.py
4.  This class is responsible for formating vector for PID-regulator in autopilot
5.
6.  Questions: anhellesnes@oslo.mil.no
7.  '''
8.
9.  import time
10.
11.
12. from Classes.PID import PID
13. from Classes.GPS_class import GPS
14. from Classes.Vector_class import Vector
15.
16. class Autopilot:
17.     def __init__(self, use_guidance = True):
18.         self.use_guidance = use_guidance
19.
20.         self.guided_velocity = 0.0
21.         self.guided_angle = 0.0
22.
23.         self.wanted_x = 0.0
24.         self.wanted_y = 0.0
25.
26.         self.controller = PID()
27.
28.     def set_wanted_xy(self, velocity_east, velocity_north):
29.         """Set wanted values if vector is x, y based
30.
31.         Args:
32.             velocity_east: float for x value of velocity vector
33.             velocity_north: float for y value of velocity vector
34.         '''
35.         self.wanted_x = velocity_east
36.         self.wanted_y = velocity_north
37.         wanted_list = [self.wanted_x, self.wanted_y]
38.
39.         self.controller.set_wanted(wanted_list)
40.
41.     def set_wanted_vector(self, wanted):
42.         """Set wanted values if vector is Vector format
43.
44.         Args:
45.           wanted: Vector object containing wanted movement
46.         '''
47.         self.guided_magnitude = wanted.magnitude
48.         self.guided_angle = wanted.angle
49.
50.         self.controller.set_wanted(wanted)
51.
52.     def __call__(self, current):
53.         if isinstance(current, Vector):
54.             pass
55.
56.         if self.use_guidance:
57.             update = self.controller.update(current)
58.             return update
59.
60.         else:
61.             #change vector type magnitude/angle to XY
62.             update = self.controller.update(current)
63.             return update
```

Vedlegg M – Kildekode Kommunikasjon - TX og RX

```
#!/usr/bin/env python
"""
Boat_TX.py
ROS node responsible for encoding multicast UDP messages and distributing
them over multicast

Questions: anhellesnes@fhs.mil.no
"'

import sys
import os
import rospy

from Classes.Udp_Publisher import PositionPublisher

from swarm.msg import BoatOdometry
from swarm.msg import BoatStatus
from swarm.msg import SwarmCommand

HEADER_FMT = 'b'

def main():
    rospy.init_node('uav_tx_node', anonymous=True)
    rospy.logdebug("Started pos_udp node")

    # defining variables for udp publisher
    mcast_grp  = rospy.get_param('~mcast_addr', "225.0.0.25")
    mcast_port = rospy.get_param('~mcast_port', 4243)
    compress   = rospy.get_param('~compression', False)
    nav_hz     = rospy.get_param('~nav_hz', 10.0)
    state_hz   = rospy.get_param('~state_hz', 1.0)
    cpu_hz     = rospy.get_param('~cpu_hz', 1.0)

    ttl = rospy.get_param('~ttl', 1) #ttl - time to live for socket

    #Definition of topic adresses for data to send
    odometry_topic    = rospy.get_param('~odometry_subscriber',      "/autopilot/current")
    status_topic      = rospy.get_param('~status_subscriber',    "/autopilot/status")

    #Initialization of publisher socket
    publisher = PositionPublisher(mcast_grp, mcast_port,
            ttl=ttl, compress=compress, nav_hz = nav_hz, state_hz=state_hz,
            cpu_hz=cpu_hz)

    #initialization of subscribers
    rospy.Subscriber(odometry_topic,  BoatOdometry, publisher.handle_odometry)
    rospy.Subscriber(status_topic,    BoatStatus,   publisher.handle_boat_status)

    rospy.loginfo("Using multicast group: {}:{}".format(mcast_grp, mcast_port))
    rospy.loginfo("Odometry subscription: {!s}".format(odometry_topic))

    rospy.loginfo("Should output be compressed: {!s}".format(compress))
    rospy.logdebug("Time to live for UDP: {!s}".format(ttl))

    #Give control over to ROS so that Python doesn't exit
    rospy.loginfo("Starting to publish")
    rospy.spin()
    publisher.shutdown()
    rospy.loginfo("Shutting down")

if __name__ == '__main__':
    main()
```

```
#!/usr/bin/env python
"
Boat_RX.py
ROS node responsible for decoding multicast UDP messages and distributing
them internally to ROS

Questions: anhellesnes@fhs.mil.no
'''

import sys
import os

import rospy

from Classes.Udp_Listener import Listener

def main():
    #initialize ROS node
    rospy.init_node('uav_rx_node', anonymous=True)
    rospy.logdebug("Started udp_pos node")

    mcast_grp  = rospy.get_param('~mcast_addr', "225.0.0.25")
    mcast_port = rospy.get_param('~mcast_port', 4243)

    rospy.loginfo("Using multicast group: {}:{}".format(mcast_grp, mcast_port))
    #initiate listener socket
    pub = Listener(mcast_grp, mcast_port)

    #Start listening and handling data from multicast
    rospy.loginfo("Starting run")
    pub.run()
    rospy.loginfo("Shutting down")

if __name__ == '__main__':
    main()
```

Vedlegg N – Kildekode Behaviour og -caller

```
#!/usr/bin/env python
'''''
Behaviour.py
This is the main program behaviour,
it gets data from ROS with swarmData and Subscriber
then calculates new movement from behave class
finally publishes to ROS with talker

Questions: anhellesnes@fhs.mil.no
'''

import time
import rospy

from Behaviour_caller import Behave
from ROS_operators.Boat_ID import get_ID
from ROS_operators.Global_data import swarmData
from ROS_operators.Behaviour_sub import Subscriber, NewCommand
from ROS_operators.Behaviour_talker import Talker

def main():
    wait_time = 0.0

    BOAT_ID = get_ID()

    #initialise objects for loop
    data = swarmData()
    command = Subscriber()
    fence = command.get_static_fence()

    behaviour_out = Talker()

    rospy.loginfo("INITIALIZING BEHAVIOUR")
    rospy.loginfo("Waiting for data...")

    while not data.has_recieved():
    #waits to recieve data
        wait_time += 0.1
        time.sleep(0.1)
        if wait_time in (10.0, 20.0, 30.0, 40.0):
            rospy.loginfo("Time waited: {}".format(wait_time))
        elif wait_time > 60.0:
            rospy.loginfo("No data recieved in 60 seconds, behaviour timed out"
)
            rospy.signal_shutdown('Behaviour timed out')

    rospy.loginfo("Data recieved after time: {}, starting".format(wait_time))

    behaviour = Behave(BOAT_ID, fence, use_behaviour=0) # BOIDS, PSO

    while not rospy.is_shutdown():
        try:
            command()

            time.sleep(0.5)

            #get newest table of data from units in swarm
            data_full = data()

            #calculate wanted vector based on current behaviour
            wanted = behaviour(data_full)
```

```
61.
62.             #publish wanted vector to autopilot
63.             behaviour_out(wanted)
64.
65.         except NewCommand:
66.             if command.stop() == True:
67.                 rospy.signal_shutdown('stop command recieved')
68.
69.             else:
70.                 colav = command.get_colavMode()
71.                 if colav == 1: #new behaviour order
72.                     new_behaviour = command.get_taskType()
73.                     try:
74.                         rospy.loginfo("Initiating new behaviour...")
75.                         del behaviour
76.                         behaviour = Behave(BOAT_ID, fence, new_behaviour)
77.                     except AttributeError as e:
78.                         rospy.loginfo("could not initiate new behaviour, with e
    rror: {} ", format(e))
79.
80.                 if colav == 2: #new fence
81.                     print("trying to set new fence")
82.                     new_fence = command.get_fence()
83.
84.                     behaviour.change_fence(new_fence)
85.
86.                 if colav == 3: #new destination
87.                     print("trying to set new dest")
88.                     destination = command.get_wantedPos()
89.
90.                     behaviour.set_destination(destination)
91.
92.                 if colav == 4: #new wanted movement
93.                     pass #not in use for any beahviour pr now
94.
95.         except rospy.ROSInterruptException():
96.             pass
97.
98. if __name__=="__main__":
99.     main()
```

```
1.  #!/usr/bin/env python
2.  '''''
3.  Behaviour_caller.py
4.  This class is responsible for calling chosen behaviour and formatting data to b
    ehaviour
5.
6.  Questions: anhellesnes@fhs.mil.no
7.  '''
8.
9.  import time
10. import rospy
11. import math as m
12.
13. from enum import IntEnum
14.
15. from Behaviours.Classes.GPS_class import GPS
16. from Behaviours.Classes.Vector_class import Vector
17.
18. from Behaviours.Boids import boidBehavior
19. from Behaviours.PSO import psoBehaviour
20.
```

```
21. from swarm.msg import BoatOdometry
22. 
23. class BehaviourType(IntEnum):
24.     BOID     = 0
25.     PSO      = 1
26.     OTHER    = 2
27.     SPECIALE = 3
28. 
29. class Behave:
30.     '''''Class to run chosen behaviour'''
31.     def __init__(self, ID, fencePOS, use_behaviour = 0):
32.         '''''Initializes data holders and chosen behaviour object'''
33. 
34.         self.current_position = GPS()
35.         self.current_movement = Vector()
36.         self.boat_id = ID
37.         self.fence_center = GPS(fencePOS.latitude, fencePOS.longitude) #could i
    nsert GPS point here for test
38.         self.fence_radius = 15.0
39. 
40.         self._handle_behaviour(use_behaviour)
41. 
42.         self.has_newSelf = False
43. 
44.     def __call__(self, global_list):
45.         '''''Caller function to check if unit is inside swarm and run chosen be
    haviour
46. 
47.         Args:
48.             global_list: list of data from boats from ROS
49. 
50.         Returns:
51.             Vector to wanted movement
52.         '''
53.         self._update_current(global_list)
54.         toFence = self._check_fence()
55. 
56.         if toFence.magnitude <= self.fence_radius:
57.             if self.behaviour_chosen == "BOIDS" and self.has_newSelf == True:
58.                 boid_data = self._make_list(global_list)
59. 
60.                 self.has_newSelf = False
61.                 behaviourXY = self.boids(self.current_position, self.current_mo
    vement, boid_data)
62. 
63.                 return self._get_vec(behaviourXY)
64. 
65.             if self.behaviour_chosen == "PSO" and self.has_newSelf == True:
66.                 pso_data = self._make_PSO_list(global_list)
67. 
68.                 self.has_newSelf = False
69. 
70.                 vectorPSO = self.pso(self.current_position, self.current_moveme
    nt, pso_data)
71. 
72.                 return self._get_vec(vectorPSO)
73.         else:
74.             print"outside fence by ", toFence.magnitude, "m, returning to cente
    r "
75.             toFence.set(1.0, toFence.angle)
76.             return toFence
77. 
78.     def change_fence(self, new_fence):
79.         '''''Helper function to set new fence
```

```
80.
81.         Args:
82.             GPS point containing center of new fence
83.         '''
84.
85.         fence = GPS(new_fence.latitude, new_fence.longitude)
86.         self.fence_center = fence
87.
88.     def set_destination(self, destination):
89.         '''''Helper function to set a destination for behaviour
90.
91.         Args:
92.             GPS point containing new destination
93.         '''
94.         destGPS = GPS(destination.latitude, destination.longitude)
95.         try:
96.             del self.pso
97.             self.pso = psoBehaviour(self.fence_center, destGPS)
98.
99.         except AttributeError as e:
100.             print"could not set destination, with error: ", e
101.
102.     def _handle_behaviour(self, behaviour):
103.         if behaviour == BehaviourType.BOID:
104.             self.behaviour_chosen = "BOIDS"
105.             self.boids = boidBehavior() #add borders
106.
107.         if behaviour == BehaviourType.PSO:
108.             self.behaviour_chosen = "PSO"
109.             self.pso = psoBehaviour(self.fence_center, self.fence_center)
110.
111.     def _update_current(self, data):
112.         try:
113.             self.current_position.set(data[self.boat_id].position.latitude, da
    ta[self.boat_id].position.longitude)
114.             self.current_movement.set(data[self.boat_id].movement.velocity, da
    ta[self.boat_id].movement.bearing)
115.
116.             self.has_newSelf = True
117.
118.         except IndexError as e:
119.             self.has_newSelf = False
120.             print("could not update current position with erorr: {s}", format(
    e))
121.
122.     def _make_list(self, dataObj):
123.         clist = []
124.
125.         for i in range(len(dataObj)):
126.             if i == self.boat_id: #removes unwanted elements from list - i.e o
    wn boat and empty elements
127.                 pass
128.             elif dataObj[i].position.latitude == 0.0 and dataObj[i].position.l
    ongitude == 0.0:
129.                 pass
130.             else:
131.                 dist = self._get_distance(dataObj[i].position)
132.                 x, y = self._get_xy(dist)
133.                 clist.append({"speed" : dataObj[i].movement.velocity,
134.                               "bearing" : dataObj[i].movement.bearing,
135.                               "distance": dist.magnitude,
136.                               "relative": dist.angle,
137.                               "x"       : x,
138.                               "y"       : y})
```

```
        for i in range(len(clist)):
            print"data from boat ", i
            print(clist[i])
        print"elements in BOIDS list: ", len(clist)
        return clist

    def _make_PSO_list(self, dataObj):
        clist = []

        for i in range(len(dataObj)):
            if i == self.boat_id: #removes unwanted elements from list - i.e o
    wn boat and empty elements
                pass
            elif dataObj[i].position.latitude == 0.0 and dataObj[i].position.l
    ongitude == 0.0:
                pass
            else:
                # dist = self._get_distance(dataObj[i].position)
                clist.append({'lat' : dataObj[i].position.latitude,
                              'lon' : dataObj[i].position.longitude})

        print"elements in PSO list: ", len(clist)
        return clist

    def _get_distance(self, pos):
        other = GPS()

        other.set(pos.latitude, pos.longitude)
        distance = self.current_position.calculate(other)

        return distance

    def _check_fence(self):
        try:
            distFence = self.current_position.calculate(self.fence_center)

            return distFence

        except AttributeError as e:
            print(e)
            return Vector(0.0, 0.0)

    def _get_xy(self, vector):
        dx = round(vector.magnitude * m.sin(m.radians(vector.angle)), 5)
        dy = round(vector.magnitude * m.cos(m.radians(vector.angle)), 5)

        return dx, dy

    def _get_vec(self, XY):
        vec = Vector()
        if XY.magnitude != 0.0 and XY.angle != 0.0:
            magnitude = m.sqrt(m.pow(XY.magnitude, 2.0) + m.pow(XY.angle, 2.0)
    )
            angle = m.degrees(m.atan2(XY.magnitude, XY.angle))

            if XY.magnitude < 0.0:
                angle = angle + 360

            vec.set(magnitude, angle)
        else:
            vec = self.current_movement

        return vec
```