Universitetet i Stavanger

DET TEKNISK-NATURVITENSKAPELIGE FAKULTET

BACHELOROPPGAVE

Studieprogram/spesialisering: Automatisering og elektronikkdesign, y-vei, bachelor i ingeniørfag (B-ELE-YVEI)	Vårsemesteret, 2021 Åpen
Forfattere: Bjørnar Størksen Wiik, Jens Trydal	Bjørnar Størksen Wiik..... (signatur forfatter) ..... (signatur forfatter)
Fagansvarlig: Morten Tengesdal Veiledere: Morten Tengesdal og Karl Skretting	
Tittel på oppgaven: Bildegjenkjenning og autonom kjøring. Engelsk tittel: Image recognition and autonomous driving.	
Studiepoeng: 2 x 15	
Emneord: Bildebehandling, ROV, UiS Subsea, prosjektledelse, brukergrensesnitt, korallrev, havbunn, T-banevogn, gruppering, grafteori, malsammenligning, morfologi, perspektiv, konturer, fargemaskering, PID-regulering.	Sidetall: 192 + vedlegg/annet: 1 (GitHub) Stavanger, 15. Mai / 2021 dato/år

Sammendrag

UiS Subsea er en studentorganisasjon ved Universitetet i Stavanger. Organisasjonen har som mål å utvikle og produsere et fjernstyrt undervannsfartøy. Dette fartøyet skal delta i en internasjonal konkurranse arrangert av organisasjonen MATE. UiS Subsea er meldt opp til å delta i Explorer-klassen, som inneholder de mest utfordrende oppgavene.

Denne rapporten tar for seg MATE-oppgavene som krever bildebehandling og datasyn:

- Autonom kjøring over en trasé merket med fargede rør. Dette inkluderer måling av, og regulering etter rotasjon og sideveis forskyvning.
- Identifisere objekter på havbunnen, og tegne disse inn på et kart.
- Undersøke helsen til et korallrev basert på bilder tatt med ett års mellomrom.
- Produsere fotomosaikk av en T-banevogn ved hjelp av bilder tatt fra fem sider.

Rapporten inneholder oppgavebeskrivelser, løsningsforslag og testing av disse. I tillegg beskrives det teoretiske grunnlaget til bildebehandlingen.

Utenom oppgavene fra MATE, har vi laget et nettbasert styringsprogram til fartøyet. Dette programmet inneholder:

- Grafisk brukergrensesnitt i nettleseren.
- Styring med Xbox-kontroller via nettleser.
- Implementasjon av bildebehandlingsoppgavene.
- Sanntids videostrøm og bildetakningsfunksjon fra IP-kamera til nettleser.

Parallelt med dette har vår gruppe hatt prosjektlederansvar for hele UiS Subsea-prosjektet. Rapporten inneholder derfor et kapittel om ledelsesstrategi, utfordringer og resultat.

Testresultatene fra bildebehandlingsoppgavene bekrefter at løsningene er robuste. Løsningen for helsen til korallrev er et unntak. Den løser i noen tilfeller oppgaven, men er svært lite robust. Dette bør utbedres. En mulig løsning kan være å kombinere løsningene basert på malsammenligning, *k-means* og grafteori.

Styringsprogrammet er testet og fungerer som ønsket. Autonom kjøring skal i teorien fungere, men vi har ikke hatt anledning til å teste dette, eller noe annet med ROV-en i vann. Den største utfordringen til programmet for autonom kjøring er en lav oppdateringshastighet. Dette er forårsaket av treg kommunikasjon fra styringsprogrammet til kamera og ROV.

Det neste steget blir testing med ROV-en i et basseng. Dette muliggjør en mer realistisk validering av løsningene.

Forord

Innledningsvis ønsker vi å takke Morten Tengesdal og Karl Skretting for deres bidrag, og veiledning gjennom dette semesteret. Vi vil takke UiS Subsea for et utmerket samarbeid, og god felles arbeidsinnsats. Vi ønsker også å beklage til familie og venner som ikke har sett oss like mye i deler av dette semesteret. Takk til dere for støtte og veiledning.

Innhold

Forkortelser og uttrykk

Engelske ord i teksten vil settes i anførselstegn. Utfyllende informasjon og henvisninger til andre deler av dokumentet settes som regel i en parentes. Er teksten mer omfattende enn det som passer seg å ha inne i en parentes, brukes fotnoter. Når det kommer til kode og funksjons/klassenavn så blir det litt annerledes. Vi bruker ofte engelske ord og uttrykk som variabel og klassenavn i koden vi lager. Dette er i utgangspunktet egennavn. På grunn av syntaksen vi bruker i koden, starter ikke disse alltid med stor bokstav. Dette blir rent språklig feil, og vi markerer dem derfor i kursiv. I listen nedenfor blir noen av forkortelsene forklart.

ROI "Region Of Interest"

MST "Minimum Spanning Tree"

ROV "Remotely Operated Vehicle"

RGB "Red, Green, Blue"

BGR "Blue, Green, Red"

HSV "Hue, Saturation, Value"

RTSP "Real Time Streaming Protocol"

RTP "Real-time Transfer Protocol"

WebRTC "Web Real-Time Communication"

OpenCV "Open Source Computer Vision Libary"

TCP "Transmission Control Protocol"

GUI "Graphical User Interface"

IIR "Infinite Impuls Respons"

CSS "Cascading Style Sheets"

HTML "HyperText Markup Language"

WSGI "Web Server Gateway Interface"

IOP "Internet Offload Processor"

API "Application Programming Interface"

HTTP "Hypertext Transfer Protocol"

CSP "Content Security Policy"

JSON "JavaScript Object Notation"

1 Introduksjon til UiS Subsea og MATE ROV-konkurransen

Dette kapittelet er hovedsakelig skrevet av Daniel Vasshus fra sensorgruppen i UiS Subsea. Kapittelet er en introduksjon til UiS Subsea og MATE ROV-konkurransen og skal gi kontekst til bakgrunnen for bacheloroppgaven.

1.1 Innledning

Innledningsvis presenteres årets bacheloroppgaver og definerer hva en ROV er. Deretter introduseres studentorganisasjonen UiS Subsea og MATE ROV-konkurransen vi skal delta i. Til slutt vil det overordnede systemet for ROV-en representeres med et blokkdiagram samt kort oppsummering av ansvarsområdet til hver bachelorgruppe. Selve bacheloroppgaven innledes i kapittel 2.

1.2 Bacheloroppgaver

I år er vi totalt 16 elektro- og maskin-studenter som deltar i studentprosjektet til UiS Subsea. Målet er å bygge en velfungerende ROV som skal hevde seg i den internasjonale MATE ROV-konkurransen (beskrevet i delkapittel 1.5.1) som avholdes i USA om sommeren. Prosjektet går samtidig ut på å ha et godt tverrfaglig samarbeid, hvor man ønsker at alle får erfaring i det å delta i en større prosjektoppgave. Samtlige av årets studenter skal skrive bacheloroppgave[1] på prosjektet, derfor har vi valgt å dele oppgavene og gruppene som følgende:

- **Elektro**
  - **Bildegjenkjenning og autonom kjøring** - Bjørnar Wiik og Jens Trydal
  - **Kraftfordelingssystem** - Andrine Pedersen og Anniken Hjelm
  - **Kommunikasjonssystem** - Martin Hausken og Oliver Langvik Veland
  - **Mikro-ROV** - Geir Arne Solland Kindingstad og Mikael Rodal Helgesen
  - **Motorstyrings- og reguleringssystem** - Edmond Baloku og Markus Haldorsen
  - **Sensorsystem og elektronikkhus** - Daniel Vasshus og Espen Myrset

[1] Oppgavebeskrivelsen for hver av oppgavene finnes i kapittel: 1.6.1

- **Maskin**
  - **Design og produksjon av ROV-manipulator** - Sindre Fjermedal og Joachim Merenyi Skjervik
  - **Design og montering av ROV-ramme, og ytelsesanalyse av motorer** - Alexander Falch Voerman og Sigvart Daniel Rodriguez Høien

Til slutt vil alle oppgavene bli satt sammen til en mikro- og en vanlig ROV.

1.3 ROV

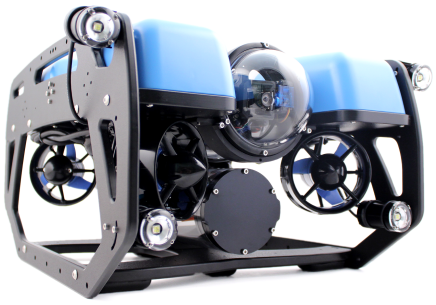
En ROV er et fjernstyrt ubemannet fartøy, som ofte omtales som ROV[2]. Disse finnes i mange størrelser og former, de minste kan være på noen få kilo, mens de største[3] kan være på størrelsen av et lite hus og veie flere tonn.

De minste ROV-ene brukes ofte til vitenskapelig forsking i form av opplæring, overvåke plante- og dyreliv, prøvetaking og gi mulighet for å inspisere plasser hvor det er uforsvarlig å sende dykkere. Større ROV-er brukes til reparasjoner, vedlikehold og inspeksjoner av konstruksjoner under vannoverflaten. Ofte vil ROV-ene være utstyrt med kamera, lys, og manipulator, men det er ikke uvanlig å utvide bruksområdet ved å installere spesialdesignet tilleggsutstyr for gitte arbeidsoppgaver.

ROV-er kan deles inn i underkategorier, og videre vil vi kort oppsummere egenskapene til disse.

1.3.1 Frittsvømmende ROV med navlestreng

Denne typen ROV er den mest vanlige og gjenkjennes med at den har en navlestreng mellom styrekonsoll og fartøy. BlueROV fra Bluerobotics som vist i figur 1 er en frittsvømmende ROV med navlestreng. Denne og tilsvarende fartøy blir ofte utstyrt med kamera, lys, manipulator eller spesialtilpassede verktøy for gitte arbeidsoppgaver.



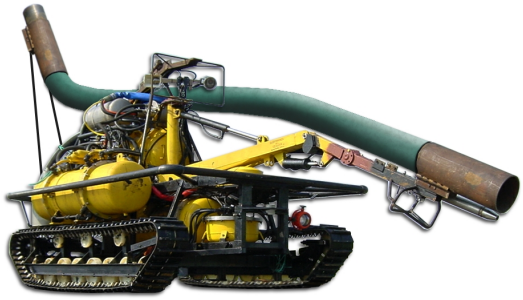
Figur 1: ROV med navlestreng av typen BlueROV2 fra BlueRobotics. Bilde hentet fra [5]

[2] **ROV**: Fjernstyrt undervannsfartøy (Remotely operated vehicle)

[3] "UT-1" en av verdens største er 7.8 m lang, 7.8 m bred, 5.6 m høy og veier 60 tonn. Denne brukes til å installere kabler på havbunnen

1.3.2 Krypende ROV

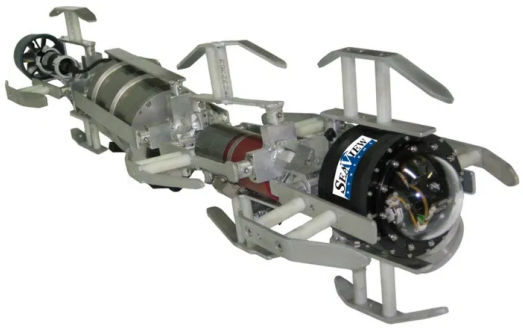
I figur 2 vises en krypende ROV[4]. Denne brukes til å krype langs sjøbunnen eller gjennom rør. Ofte graver den ned rør og kabler i sjøbunnen, men kan også utføre inspeksjoner eller brukes til gruvedrift på havbunnen.



Figur 2: Krypende ROV av typen Seabed dredger fra Seascape. Hentet fra [77]

1.3.3 Strukturelt avhengig ROV

En strukturelt avhengig ROV[5] som vist i figur 3 er primært brukt for å inspisere og vaske konstruksjonen den er festet til. For å bevege seg langs konstruksjonen den er festet til, brukes blant annet hjul, kabel, skinner, taljer eller hydrauliske løsninger.



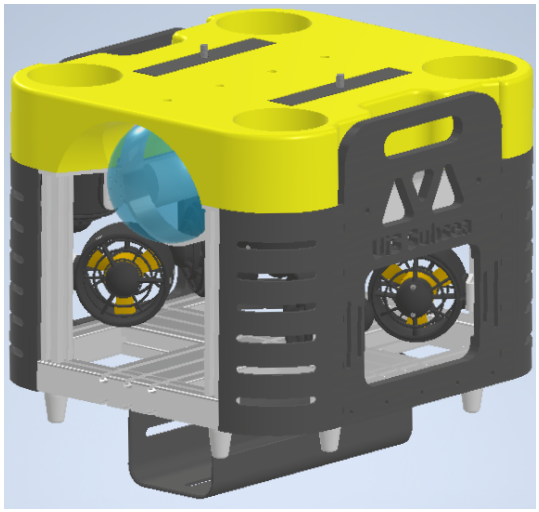
Figur 3: Strukturelt avhengig ROV av typen Serpent av Seaview. Hentet fra [78]

[4]På engelsk: Crawling ROV
[5]På engelsk: Structurally Reliant ROV

1.3.4 Hymir

I UiS Subsea i år skal vi bygge en ROV med navn Hymir, denne er frittsvømmende ROV med navlestreng som blir styrt av en ROV-pilot ved et kontrollpanel på land. Hymir skal brukes til å utføre diverse arbeidsoppgaver i basseng, men vi ønsker også å ha mulighet for å bruke den i ferskvann og sjø. I tillegg skal vi bygge en mikro-ROV som er festet til undersiden av ROV-en, denne skal kunne inspisere og hente ut objekter i rør helt ned til en størrelse på 6 tommer.

Hele kostnaden for prosjektet blir dekket av studentorganisasjonen UiS Subsea.



Figur 4: 3D modell av Hymir, med dokkingstasjon for mikro-ROV på undersiden

1.4 UiS Subsea

UiS Subsea er en studentorganisasjon ved Universitetet i Stavanger. Organisasjonen har siden 2013 hatt som mål å motivere studenter til å delta i større studentprosjekt, hvor man ønsker å skape et miljø for innovasjon, utvikle tekniske ferdigheter og vise kreativ tilnærming til problemstillinger som dukker opp. Samtidig er det svært viktig for UiS Subsea å styrke samarbeidet mellom universitetet og næringslivet. Tidligere år har det blitt utviklet både ROV og AUV-er i forbindelse med bacheloroppgaver for data-, elektro- og maskinstudenter.

Figur 5: Logo UiS Subsea

Vi har i løpet av overtakelsen av UiS Subsea gjort store endringer på organisasjonsstrukturen for å gjøre det mer bærekraftig for fremtidig rekruttering og fordeling av arbeid. I den sammenheng valgte vi å skille mellom styreoppgaver og prosjektoppgaver. Styret har ansvar for:

- Organisasjonen UiS Subsea
- Økonomi
- Sponsoravtaler
- Markedsføring
- Rekruttering av nye studenter
- HMS

Noen av oppgavene til prosjektledelsen er:

- Prosjektfremgang
- Finne en passende konkurranse laget skal delta i
- Passe på at prosjektet blir forsvarlig styrt
- Ansvar for at alle studentene er med og drar i riktig retning
- Påse at alle krav og retningslinjer følges

For å fordele ansvar og imøtekomme kravet til MATE ROV-konkurransen om entreprenørskap har vi fordelt rollene som følger:

- Styret
  - **Styreleder:** Daniel Vasshus
  - **Nestleder:** Geir Arne Solland Kindingstad
  - **Økonomiansvarlig:** Edmond Baloku
  - **Markedsføring:** Martin Hausken og Oliver Langvik Veland
  - **Sponsoransvarlig:** Sigvart Daniel Rodriguez og Bjørnar Wiik
  - **Styremedlemmer:** Espen Myrset og Mikal Rodal Helgesen
- Prosjektledelse
  - **Prosjektleder:** Jens Trydal
  - **Teknisk ansvarlig:** Sindre Fjermedal
  - **Lagleder elektro:** Markus Haldorsen
  - **Lagleder maskin:** Joachim Merenyi

1.5 MATE - Marine Advanced Technology Education

Vi vil i dette delkapittelet presentere MATE ROV-konkurransen, oppgavene som skal utføres, poengfordelingen og tekniske krav til ROV-en.

MATE Center er en nasjonal organisasjon i USA som samarbeider med skoler, forskningsinstitutt, myndigheteter, militær, og institutt for marin teknikk. Formålet til organisasjonen er å bruke marin teknologi til å utfordre og inspirere studenter ved å løse problemstillinger fra den virkelige verden. For å løse problemstillingene ønsker de at studentene har en kreativ tilnærming til forskning, teknologi og ingeniørfag.

Figur 6: MATE logo. Hentet fra [35]

For å oppnå formålet til organisasjonen blir det årlig arrangert en internasjonal ROV-konkurranse hvor samtlige lag skal løse en bestemt problemstilling. Konkurransen har fått navnet "MATE ROV Competition"

1.5.1 MATE ROV Competition

Hvert år presenterer konkurransen nye kunder med forskjellige problemstillinger som skal løses, og i årets utgave har vi "verdenssamfunnet" som kunde. Oppgaven som skal løses er å bygge en ROV som skal takle plastproblemet i sjøen, klimaendringens påvirkning på korallrev og konsekvensen med dårlig miljøpraksis på våre vannveier. I konkurransen skal vi ikke gå inn på hva vi må gjøre for å adressere den faktiske årsaken til problemstillingen, kun løse oppgavene som er gitt.

UiS Subsea stiller i utforsker klassen[6] og er den vanskeligste klassen i MATE ROV Competition.

Figur 7: Mate ROV-konkurranse logo. Hentet fra [34]

[6] I konkurransen heter klassen "Explorer"

Konkurransen gir poeng for å løse tre forskjellige praktiske oppgaver, samtidig gis det poeng for størrelse og vekt. I tilleg får man poeng for dokumentasjon, presentasjon og sikkerhet. Mer om poengfordeling i delkapittel 1.5.5. Vi skal nå ta for oss de tre praktiske oppgavene:

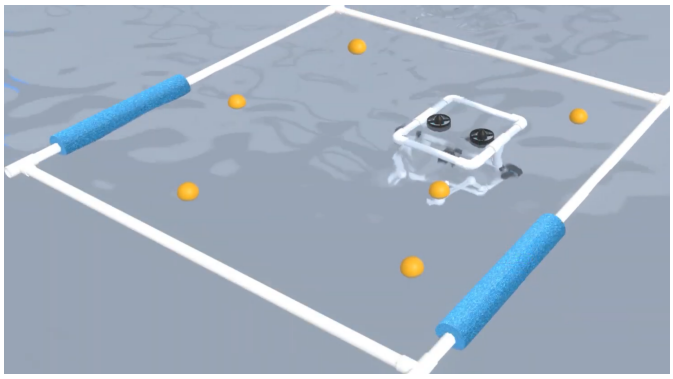
1.5.2 Oppgave 1: Problemet med plastforurensing (Totalt 90 poeng)

- **Søppelbøtte til sjøs - "Rydde opp havet, en havn om gangen"**
  - Koble fra gammel strømkontakt til en nyinstallert søppelbøtte - 5 poeng
  - Fjerne tidligere fangstpose av netting fra søppelbøtte - 10 poeng
  - Installere ny fangstpose av netting i søppelbøtta - 10 poeng
  - Koble til en ny strømkontakt[7] til den nyinstallerte søppelbøtta - 20 poeng

Figur 8: Ved at ROV-en fjerner støpselet fra søppelbøtta vil lyset slukkes. Videre i oppgaven fjerner man fangstposen med objekter i, for å så returnere med en ny fangstpose. Hentet fra [80]

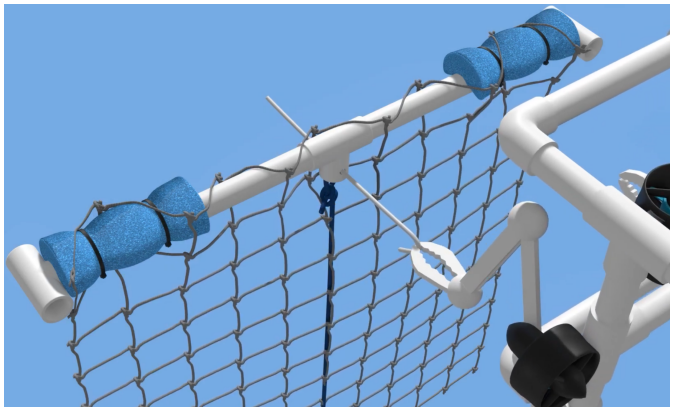
[7]Strømkontakten skal selv lages

- **Utbedring: Fjerne plastavfall fra topp til bunn**
  - Fjerne flytende plastavfall fra vannoverflaten
    - Fjerne alle 6 plastballene - 15 poeng
    - Fjerne 3 til 5 plastballer - 10 poeng
    - Fjerne 1 til 2 plastballer - 5 poeng
    - Fjerne 0 plastballer - 0 poeng



Figur 9: Fjerne flytende plastavfall fra vannoverflaten. Hentet fra [80]

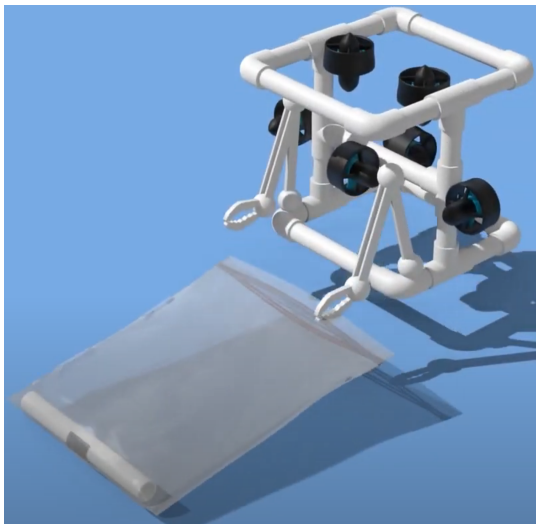
  - Fjerne et spøkelsesnett[8] fra vannet
    - Trekke ut en pinne for å simulere det å kutte spøkelsesnettet løst - 10 poeng
    - Fjerne spøkelsesnettet fra vannet - 10 poeng



Figur 10: Fjerne spøkelsesnett fra vannet. Hentet fra [80]

[8]Spøkelsesnett representerer et fiskenett som er forlatt eller mistet av fiskere.

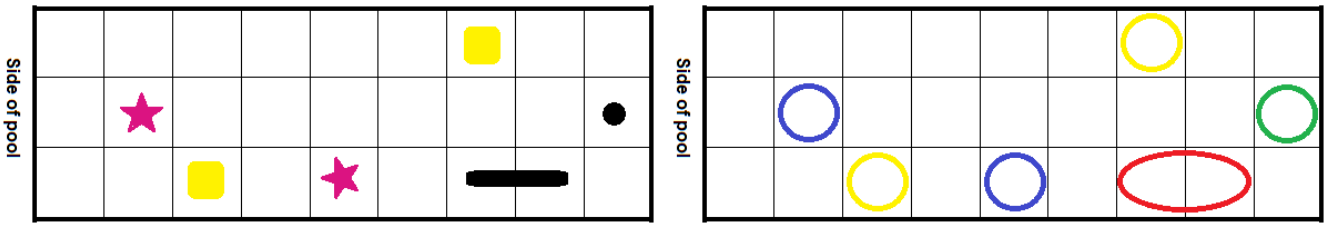
– Fjerne plastrester fra bunnen av Marianergropen[9] - 5 poeng for hver, maks 10 poeng



Figur 11: Fjerne plastrester fra havbunnen. Hentet fra [80]

1.5.3 Oppgave 2: Den katastrofale virkningen av klimaendringen på korallrev (Totalt 90 poeng)

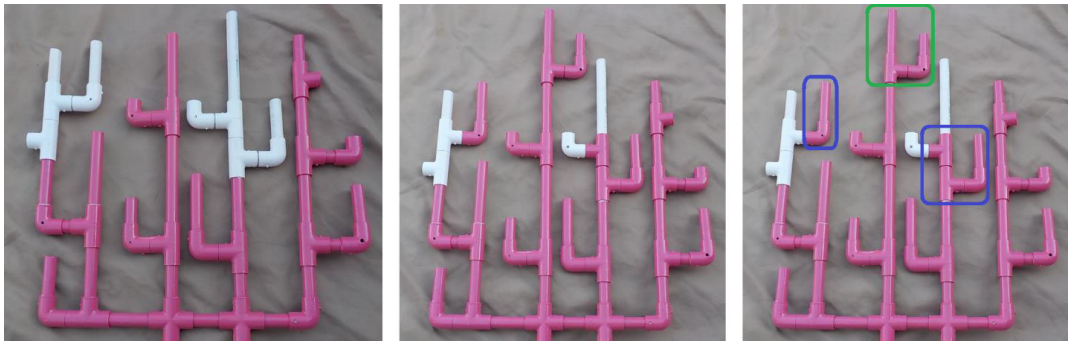
- **Kjøre i en transektlinje[10] over et korallrev og kartlegge interessepunkt**
  - Kjøre i en transektlinje over et korallrev
    - Kjøre transektlinjen autonomt - 15 poeng
    - Kjøre transektlinjen manuelt - 5 poeng
  - Kartlegge interessepunkt i korallrevet
    - Automatisk kartlegging på en dataskjerm - 10 poeng
    - Manuell kartlegging på en dataskjerm - 5 poeng



Figur 12: Kjøre over korallrevet til venstre i figuren. Her er det to lokasjoner for å plante ut korallfragmenter (gul firkant) i, to sjøstjerner (lilla stjerner), en korallkoloni (svart rektangel) en svamp (svart prikk). Høyre: Fargede sirkeloverlegg i riktige ruter for vise interessepunkt og organismene på revet. Hentet fra [15]

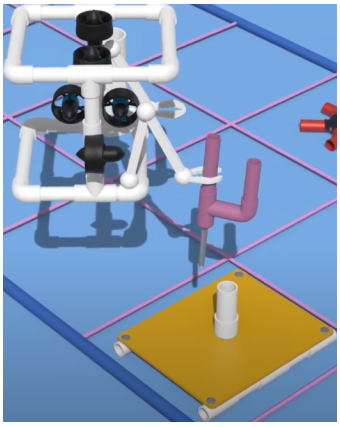
[9] Marianergropen er verdens dypeste havgrop og ligger vest i Stillehavet og øst for Marianene
[10] En transektlinje er en linje på tvers av et habitat, eller deler av et habitat

- **Bruke bildegjenkjenning for å bestemme helsen til en korallkoloni ved å sammenligne nåværende tilstand med tidligere data**
  - Bruke bildegjenkjenning for å bestemme helsen til korallkolonien
    - Alle endringer er identifisert - 20 poeng
    - Minst en, men ikke alle endringene er identifisert - 10 poeng
    - Ingen endringer er identifisert - 0 poeng
  - Bruke en håndbok for å bestemme helsen til korallkolonien - 5 poeng



Figur 13: Venstre: Korallkolonien for ett år siden. Senter: Korallkolonien slik den er i dag. Høyre: Korallkolonien med alle områder identifisert, et område med vekst og to områder med gjenoppretting fra bleking/flekking. Hentet fra [15]

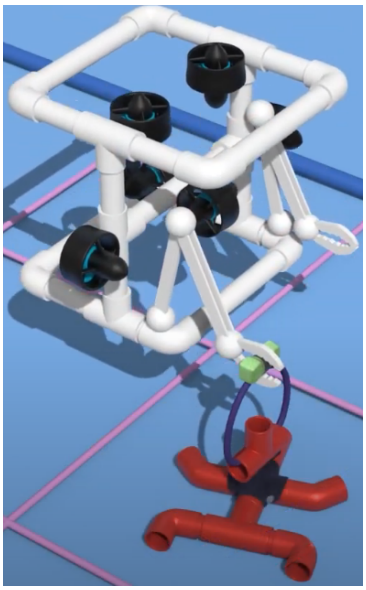
- **Gro koraller på rev**
  - Fjerne korall-fragmenter fra planteskolen[11] - 5 poeng for hver, maks 10 poeng
  - Plante ut korall-fragmenter på bestemte områder i revet - 5 poeng for hver, maks 10 poeng



Figur 14: Gro koraller på rev. Hentet fra [80]

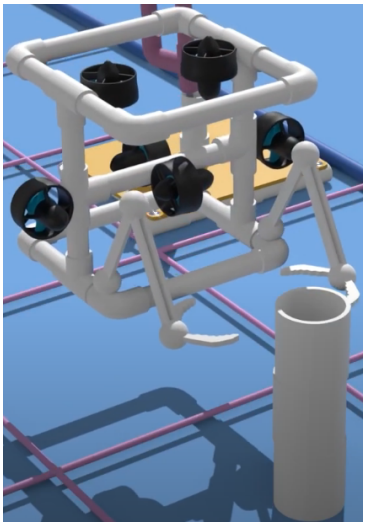
[11] Gartneri der det dyrkes planter som siden plantes ut

- **Begrense et utbrudd av tornekronesjøstjerne**[12] - 5 poeng for hver, maks 10 poeng



Figur 15: Begrense utbrudd av tornekronesjøstjerne. Hentet fra [80]

- **Samle prøver av svampearter for farmasøytisk forskning**
  - Hente en prøve fra en svamp - 10 poeng
  - Returnere prøven til overflaten - 5 poeng



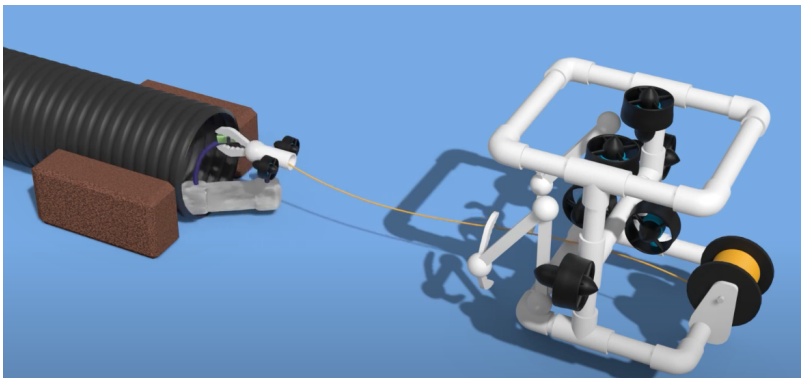
Figur 16: Hente prøve fra svampearter. Hentet fra [80]

[12]Tornekronesjøstjerne er en av verdens største sjøstjerner og er beryktet for å skade koraller i tropiske farvann

1.5.4 Oppgave 3: Vedlikeholde sunne vannveier. Del II: Delawarebukten og elva (Totalt 90 poeng)

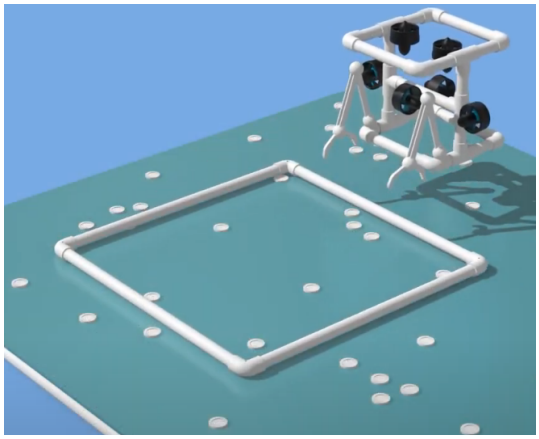
Delawarebukten er en dyp bukt og er en forlengelse av elven Delaware på USAs nordøstlige kyst. Ferskvannet renner ut i Atlanterhavet.

- **Hente ut en sedimentprøve fra innsiden av et avløpsrør for å analysere forurensing**
  - Utplassere en enhet i røret for å hente sedimentprøven - 25 poeng
  - Returnere sedimentprøven til overflaten - 10 poeng
  - Bestemme hvilken type forurensing(er) som er til stede i sedimentprøven - 5 poeng



Figur 17: Hente ut en sedimentprøve fra et 6 tommers rør. Hentet fra [80]

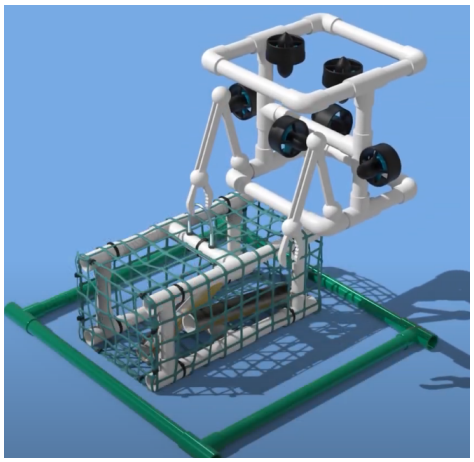
- **Estimere antall blåskjell i en blåskjell-klynge**
  - Utplassere en kvadrant og telle antall blåskjell i kvadranten - 5 poeng
  - Estimere antall blåskjell og hvor mye vann som filtreres av blåskjellene - 5 poeng



Figur 18: Estimere antall blåskjell

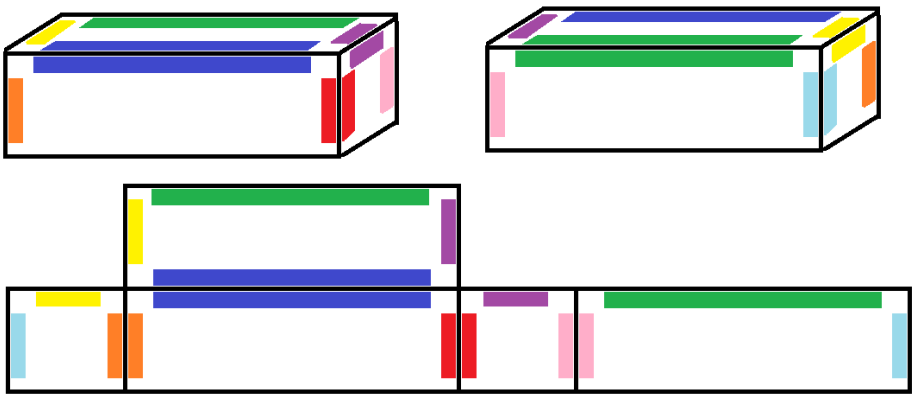
- **Restaurere ålebestanden**

- Fjerne en full åleteine fra et bestemt område - 10 poeng
- Plassere en tom åleteine i et bestemt område - 10 poeng



Figur 19: Fjerne og plassere ut åleteine. Hentet fra [80]

- **Lage fotomosaikk[13] av en nedsenket togvogn for å skape et kunstig rev**
  - Autonomt - 20 poeng
  - Manuelt - 10 poeng



Figur 20: Topp: Diagram med fargesammensetning på togvognen. Bunn: Fotomosaikk av togvognen som vist på dataskjermen. Hentet fra [15]

[13]Fotomosaikk er en samling av bilder satt sammen til ett bilde

1.5.5 Poengfordeling

For å kåre en vinner av konkurransen, vil samtlige poeng man har samlet, legges sammen, og laget med flest poeng blir kåret som vinner. I listen nedenfor har vi en oversikt på alle delene vi kan hente poeng på.

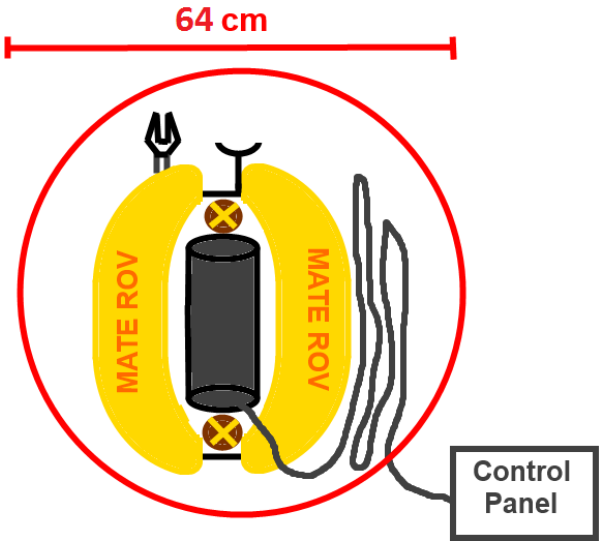
Produktdemonstrasjon	
Produktdemonstasjon (oppgaver)	270 + tidsbonus
Størrelse- og vektrestriksjoner	20
Produktdemonstrasjon av organisasjonseffektivitet	10
Prosjektering og kommunikasjon	
Teknisk dokumentasjon	100
Produktpresentasjon	100
Markedsføring	50
Selskapets spesifikasjonsark	10
Bedriftsansvar	20
Sikkerhet	
Sikkerhet og dokumentasjon gjennomgang	20
Sikkerhetsinspeskjon	30
Sikker jobb analyse (SJA)	10
Sum poeng:	**650 + tidsbonus**

1.5.6 Tekniske krav til ROV-en

Konkurransen (beskrevet i delkapittel 1.5.1) stiller strenge krav til ROV-ene som skal delta. Disse kravene setter føring på hvordan vi ønsker å løse våre prosjektoppgaver. I produktmanualen til konkurransen [15] er det en full oversikt over krav man må følge. Her vil vi gi en kort oversikt på de aller viktigste punktene. Spesifikke krav som angår enkelte grupper blir belyst i bacheloroppgavene hvor det er relevant.

Fysiske krav

I konkurransen er den fysiske størrelsesgrensa for å kunne delta, satt til å være maks 92 cm i diameter og kan ikke veie mer enn totalt 35 kg. Hele ROV-en, med tilleggsutstyr og navlestreng skal passe innenfor en ring som vist i figur 21 og veies.



Figur 21: ROV med verktøy og navlesteng kveilet opp ved siden av ROV-en. Hentet fra [15]

Videre får man ekstrapoeng for å være innenfor bestemte størrelse og vektklasser, som vist i tabellen under.

Størrelse	**Poeng**	**Vekt (på land)**	**Poeng**
<64 cm diameter	+ 10 poeng	<20 kg	+ 10 poeng
65.1 til 75 cm diameter	+ 5 poeng	20.01 kg til 28 kg	+ 5 poeng
75.1 til 92 cm diameter	0 poeng	28.01 til 35 kg	+ 0 poeng

Tekniske løsninger

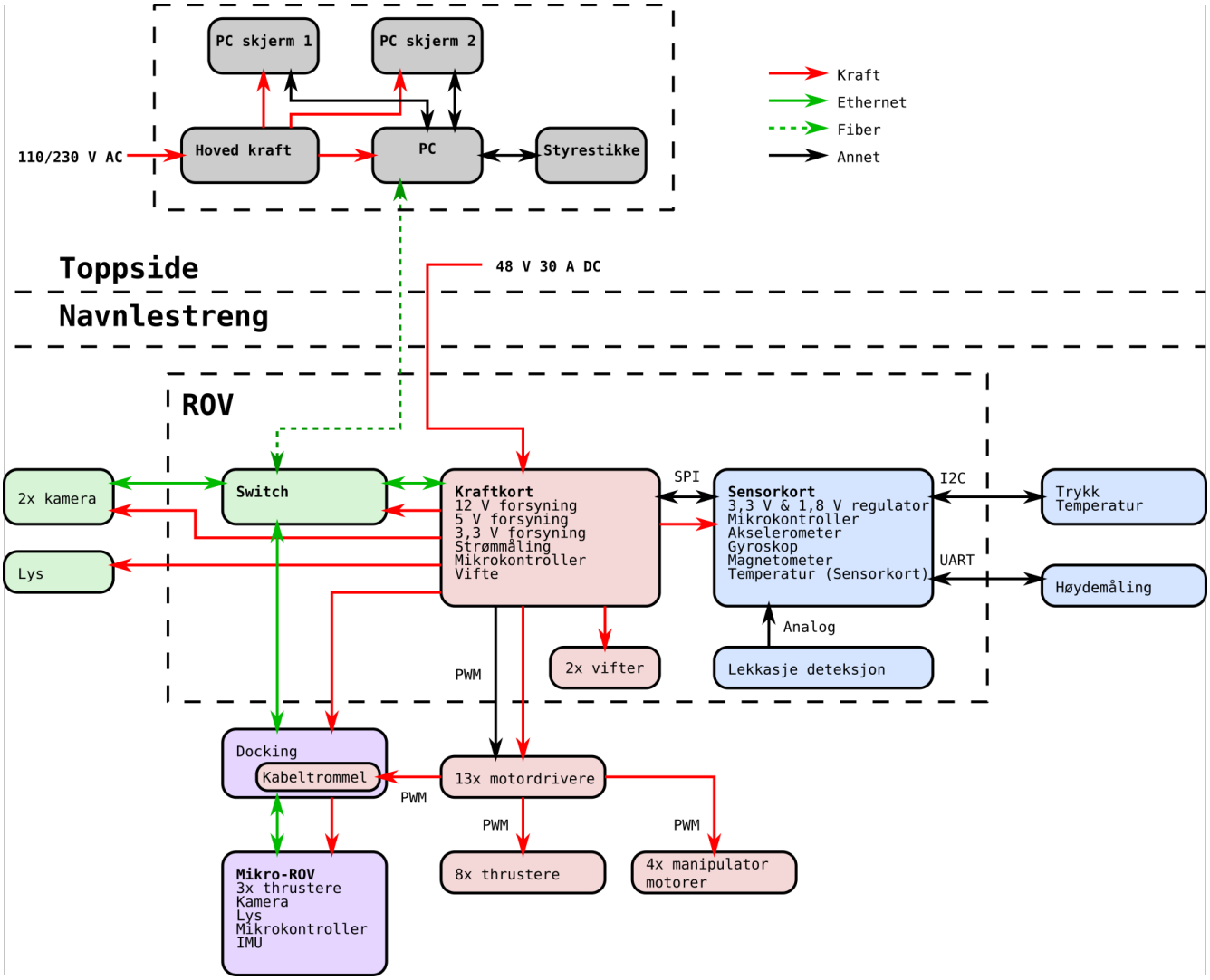
Det er kun lov å bygge én ROV til å utføre oppgavene i bassenget, men i oppgaven hvor man skal hente ut sedimentprøven (beskrevet i delkapittel 1.5.4) har man lov å bygge en mikro-ROV. Videre skal ROV-en kunne kjøre i et rent klor-basseng med en temperatur mellom 15 og 30 °C. Dybden på bassenget er maks 7 meter og samtlige oppgaver foregår innenfor 10 meter fra bassengkanten. Styrekonsollen blir plassert nær bassengkanten (maksimalt 3 meter), og navlestrengen må være lang nok til å utføre alle oppgavene.

"MATE ROV Competition" disponerer 48 V og 30 A likespenning ved styrekonsollen som skal forsyne ROV-en. Det er først lov å endre spenningsnivået på innsiden av ROV-en. Konkurransen stiller ekstra strenge krav til pneumatikk, hydraulikk og laser hvis man velger å bruke dette. Tas dette i bruk må man følge spesifikasjonene og dokumentere utstyret i henhold til konkurransemanualen [15].

1.6 Overordnet system

I dette delkapittelet vil vi presentere det overordnede systemet for ROV-en med et blokkdiagram og gi en kort oppsummering over ansvaret til hver av gruppene.

1.6.1 Blokkskjema



Figur 22: Blokkskjema av ROV

1.6.2 Bildegjenkjenning og autonom kjøring

Oppgaven består av bildebehandling, styringsprogram og prosjektledelse. Den første delen går ut på å lage algoritmer som løser oppgavene om bildebehandling- og autonom kjøring i MATE ROV-konkurransen. Den andre delen går ut på å lage et nettbasert styringsprogram / brukergrensesnitt til ROV-en. I tillegg inneholder oppgaven litt om prosjektledelse.

1.6.3 Kraftfordelingssystem

Oppgaven består av å utvikle og konstruere et kraftfordelingskort for ROV, og en navlestreng for krafttilførsel fra overflatesystemet til ROV-en. Kraftfordelingskortet skal kunne forsynes med en spenning på 48 - 56 V DC med på maksimalt 30 A strøm.

Tilførsel av kraft skjer ved hjelp av navlestrengen, som også vil bli brukt til kommunikasjonsoverføring (fiberkabel). De ulike komponentene i ROV-en forsynes med ulik spenning, derfor vil det være behov for å regulere spenningen ned til 12 V, 5 V og 3,3 V på kraftfordelingskortet. For å kjøle de varmeste delene på kretskortet blir det tatt i bruk kjøleribber og vifter.

1.6.4 Kommunikasjonssystem

Oppgaven går ut på å utvikle kommunikasjonssystemet til ROV og mikro-ROV. Her tar vi av oss all kommunikasjon som skal gå mellom ROV og toppside på land. Kommunikasjonen inneholder blant annet styringsverdier og verdier fra forskjellige sensorer på ROV-en. Samtidig inneholder oppgaven valg av kamera og belysning til ROV-en.

1.6.5 Motorstyring- og reguleringssystem

ROV-en består av 12 motorer, disse skal sørge for et robust og driftssikkert motorstyrings- og reguleringssystem for fremdrift, samt et robust og driftssikkert styringssystem for manipulator. Frihetsgradene som skal reguleres er rull, stamp og hiv.

1.6.6 Sensorsystem og elektronikkhus

Oppgaven består av å utvikle og konstruere sensorsystem og elektronikkhus til ROV-en. Sensorsystemet består av flere forskjellige sensorer som skal hjelpe til med å styre, overvåke og informere om status til ROV-en i vannet. Målingene som mottas kalibreres og brukes til utregninger før det kan brukes av de andre delsystemene. Elektronikkhuset inneholder alt av elektronikk, og har konnektorer for å koble sammen utstyr på utsiden med elektronikken på innsiden.

1.6.7 Design og kontroll av ROV manipulatoren

Oppgaven består av å utvikle og konstruere en manipulator samt klype for ROV-en. Manipulatoren skal være i stand til å utføre gitte oppgaver i konkurransen, samt være modulær. Det skal også

lages ulike simuleringer av manipulatoren for å øke forståelsen av de ønskede bevegelser og laster.

1.6.8 Design og montering av ROV-ramme, og ytelsesanalyse av motorer

Rammegruppen konstruerer et rammedesign basert på *Design for Assembly* konseptet. Det er modulært tilpasset slik at andre komponenter kan plasseres på ROV-en. ROV-en skal ha et lavt *Center of mass* og høyt *Center of buoyancy* for å ha stabil og ha høy manøverabilitet. Alle rammekomponenter styrkeanalyseres og det foretas en thrusteranalyse for plassering av thrustere for å få geometrisk optimalisert design basert på vektklasse og størrelse. Samtidig blir Autodesk benyttet til å beregne ROV-ens faktisk bevegelighet i vann og vektoranalyser av bevegelsene i vann.

1.6.9 Mikro-ROV

For å hente ut et objekt fra et 6 tommers rør utvikles det en Mikro-ROV som skal dokkes til hoved-ROV-en som igjen forsyner den med strøm og signaler via en navlestreng. Oppgaven består av maskinvare, kretskort, programvare, design, fysikk og mekanisk arbeid.

2 Om denne bacheloroppgaven

Målet med bacheloroppgaven er å løse de følgende utfordringene:

- Autonom kjøring over en trasé merket med fargede rør. Identifisering av objekter på havbunnen langs traseen, og tegning av disse inn på et kart.
- Undersøke helsen til et korallrev basert på bilder tatt med ett års mellomrom.
- Produsere fotomosaikk av en T-banevogn ved hjelp av bilder tatt fra fem sider.
- Lage et styringsprogram til ROV-ene med nettbasert brukergrensesnitt og sanntids video.
- Lede ROV-prosjektet til UiS Subsea.

Bildebehandlingsoppgavene er hentet fra MATE-konkurransen, og utgjør den største og mest omfattende delen av bacheloroppgaven. De andre utfordringene er definert ut ifra hva UiS Subsea har behov for.

Den videre rapporten er delt opp i følgende kapitler:

- **Kapittel 3 Teoridel bildebehandling:**
  Tar for seg teorigrunnlaget til bildebehandlingsoppgavene.
- **Kapittel 4 Havbunn:**
  Kapittelet beskriver løsning og testing for autonom kjøring og kartlegging av havbunnen.
- **Kapittel 5 Helsen til korallrev:**
  Kapittelet beskriver løsninger og testing av løsningene for undersøkelse av helsen til korallrev.
- **Kapittel 6 Fotomosaikk av T-banevogn:**
  Kapittelet beskriver løsning og testing av løsning for fotmosaikk av T-banevogn
- **Kapittel 7 Styringsprogrammet:**
  Denne delen inneholder krav til styringsprogrammet og hvordan vi har laget det.
- **Kapittel 8 Prosjektledelse:**
  Kapittelet inneholder litt om prosjektledelse, strategi og styringsverktøy, samt hvordan prosjektet har endt opp.
- **Kapittel 9 Refleksjon og erfaring:**
  Denne delen inneholder erfaringer fra arbeidesprosessen, forslag til forbedringer og refleksjon rundt løsningene.
- **Kapittel 10 Konklusjon:**
  Dette siste kapittelet inneholder en oppsummering av bacheloroppgaven, sammendrag av forbedringsforslag og gjenstående arbeid og beskrivelse av vedlegget.

3 Teorigrunnlag for bildehandlingsoppgavene

I dette kapittelet beskrives funksjoner og metoder vi har brukt for å løse bildebehandlingsoppgavene. Vi benytter oss hovedsakelig av biblioteket OpenCV [52], [57] i programmeringsspråket Python [24]. "Open Source Computer Vision Libary" (OpenCV) inneholder en rekke funksjoner, klasser og metoder som er beregnet for sanntidsbildebehandling. I tillegg har vi brukt Numpy [27] og Scikit-learn [76] bibliotekene. OpenCV og Python er valgt fordi vi er kjent med disse fra tidligere, og fordi de er veldokumenterte og brukt av mange.

1. **Fargemaskering**
   Benyttes hovedsakelig til å produsere svart-hvitt-bilder av områdene som inneholder bestemte farger.

2. **Terskling**
   Benyttes hovedsakelig til å lage svart-hvitt-bilder av gråskalabilder.

3. **Konturer**
   Benyttes hovedsakelig til å finne, klassifisere og gjenkjenne konturer/former/objekter i et svart-hvitt-bilde.

4. **Morfologiske operasjoner:**
   Benyttes til filtrering og forbedring av svart-hvitt-bilder. Dette er gjerne bilder vi ønsker og finne konturer i.

5. **Malsammenligning**
   Brukes til å finne bestemte deler av et bilde. Vi benytter dette i kapittelet om helsen til korallrev (5)

6. **Gruppering**
   Metoder for å samle flere punkter i et bilde rundt et felles senter. Vi fokuserer på *k-means*. Vi benytter dette i kapittelet om helsen til korallrev (5)

7. **Nærmeste nabo**
   Algoritmer for å finne nærmeste nabo til punkter. Brukes sjeldent i bildebehandling, men i vårt tilfelle har det vært nyttig i kombinasjon med gruppering og grafteori. Vi benytter dette i kapittelet om helsen til korallrev (5)

8. **Grafteori**
   Brukes for å finne stier mellom punkter. Vi benytter dette i kapittelet om helsen til korallrev (5)

9. **Annet**
   I denne seksjonen omtales OpenCV-funksjoner, som har falt under de andre kategoriene, andre selvlagede funksjoner og annet grunnlagsmateriale.

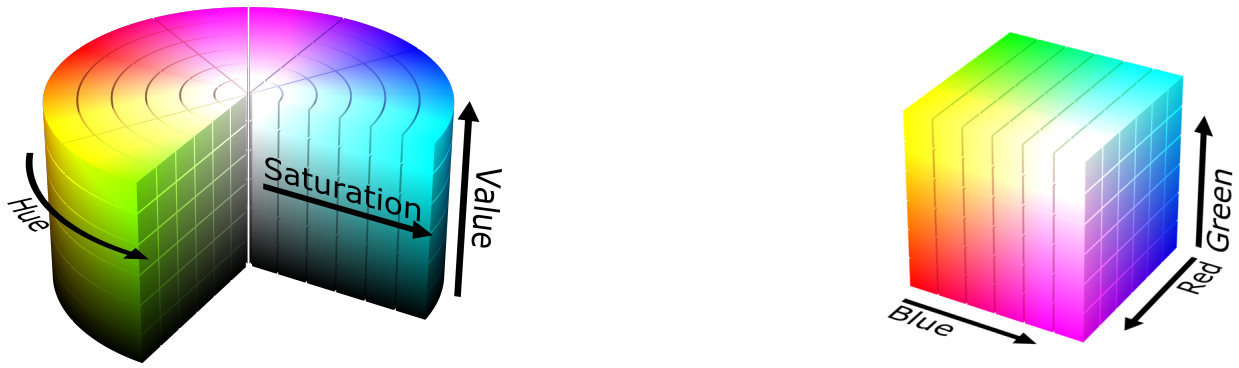
3.1 Fargemaskering

Det å klippe ut enkelte farger fra et bilde er en nyttig operasjon innen bildebehandling. Operasjonen består av tre ledd. Først benyttes HSV fargeformat (se seksjon 3.1.1) til å definere fargene/området vi skal maskere ut. Deretter brukes OpenCVs inRange-funksjon (se seksjon 3.1.2) til å lage en maskering for det valgte område. Til slutt benyttes OpenCVs bitwise_and funksjon (se seksjon 3.1.3) til å klippe ut det maskerte området fra originalbildet.

3.1.1 HSV

HSV er en alternativ fargerepresentasjon til RGB-modellen. RGB-modellen angir hvor mye av hver enkelt av grunnfargene rød, grønn og blå som er inneholdt i hvert piksel. Figur 23b illustrerer RGB-formatet hvor andelen av hver av de tre grunnfargene avgjør den endelige fargetonen. HSV angir hver enkelt piksels tilstand på en annen måte.

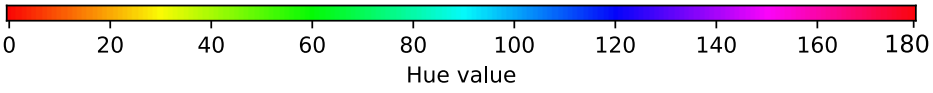
"Hue, Saturation, Value" (HSV) er en forkortelse for fargetone, metning og verdi. Det betyr at all informasjon om farge angis i én parameter, fargetone. Metning angir fargens metning fra null, som tilsvarer helt hvitt, til en som tilsvarer en helt klar og tydelig fargetone. Verdi angir lysstyrken fra null, som tilsvarer sort, til en som tilsvarer tydelig farge. Figur 23a illustrerer hvordan disse tre verdiene angir fargene. For mer detaljert forklaring se [6] og [26]. Det er også en artikkel på Wikipedia som omtaler emnet [16].
En av fordelene med HSV-formatet er at den gjør det lettere å separere ut farger. I RGB format må tre forskjellige parametre justeres for å velge en farge. I HSV-formatet er det kun en parameter (Hue, fargetone) som har med selve fargen å gjøre.



(a) HSV fargesylinder [8].

(b) RGB fargefordeling [14].

Figur 23: HSV og RGB fargeformat.



Figur 24: OpenCV sitt område for "Hue". Merk at fargen angitt med 0 er veldig lik den angitt av 180. Dette vil skape utfordringer med tanke på beregning av gjennomsnittsfarge.

OpenCV har sin egen utgave av HSV-formatet. Fargetone(H-hue) går fra 0 til 179 (vist i figur 24), metning(S-saturation) fra 0 til 255 og verdi(V-value) fra 0 til 255. I RGB formatet går alle verdiene fra 0 til 255. Årsaken til at vi holder oss mellom 0 og 255 er at hvert enkelt av de tre variablene lagres i ett 8 bits format (uint8, unsigned integer, 8 bits). Dette gir oss en laveste mulig verdi på 0 og høyeste mulige verdi på 255. Fargetonen går fra 0 til 179 da denne parameteren originalt tilsvarer gradene til en full sirkel. Siden maksverdien er 255, blir 360° halvert slik at tallgrensen blir overholdt.

cvtColor:
Dette er en enkel funksjon som brukes til å endre bildeformat. Det vi endrer er formatet til hver enkelt piksel. Vi kan for eksempel endre fra BGR (Blå, Grønn, Rød) til RGB (Rød, Grønn, Blå), eller andre veien, fra RGB til BGR. I kodesnutt 1, linje 7, brukes funksjonen til å endre fra BGR format(OpenCVs standard bildeformat) til HSV-formatet. For mer informasjon om forskjellige bildeformat og mer om selve funksjonen gå til OpenCVs dokumentasjon[38].

3.1.2 inRange:

OpenCV-funksjonen *inRange* [60] tar inn tre argumenter: Ett bilde, nedre fargegrense og øvre fargegrense. Den returnerer så ett svart-hvitt-bilde. Pikslene utenfor angitte fargegrenser blir sorte og pikslene innenfor angitte fargegrenser blir hvite. Kodesnutt 1, linje 15, viser hvordan *inRange*-funksjonen anvender fargegrensene til å lage et svart-hvitt-bilde.

Inngang:

1. **src:** Bilde.
2. **loweb:** Nedre fargegrense.
3. **upperb:** Øvre fargegrense.

Utgang:

1. **dst:** Resulterende svart-hvitt-bilde. Ofte brukt som maskering.

3.1.3 Bitwise_and:

Funksjonen *Bitwise_and* er en av flere aritmetiske funksjoner i OpenCV[37]. Slik funksjonens navn antyder utfører denne funksjonen bitvis, eller pikselvis, logiske OG-operasjoner. Enkelt forklart tar funksjonen inn en svart-hvitt maskering, som angir hvilke område operasjonen skal utføres på, og to bilder som den bitvise operasjonen utføres på. Det hvite området i maskeringen angir pikslene operasjonen skal utføres på. Det sorte område i maskeringen forblir sort i det returnerte bilde.
Hvis vi sender inn en maskering og to identiske bilder, får vi klippet ut det maskerte området fra originalbildet. I linje 21 i kodesnutt 1 benyttes *Bitwise_and* på denne måten.

Inngang:

1. **src1:** Bilde nr 1.
2. **src2:** Bilde nr 2.
3. **mask:** Svart-hvitt-bilde som er maskeringen.

Utgang:

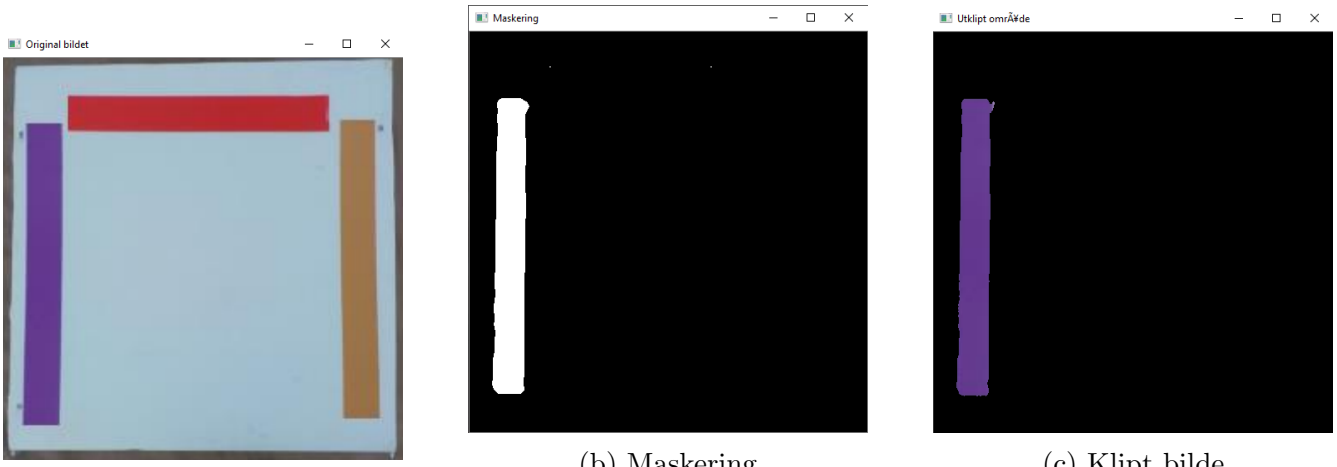
1. **dst:** Resulterende bilde.

3.1.4 Utførelse fargemaskering

I denne seksjonen har vi et praktisk eksempel (kodesnutt 1) på bruk av fargemaskering. Figur 25 presenterer resultat fra koden. Der kan vi se originalbildet(25a), maskeringen(25b) laget av *inRange* og området(25c) klippet ut med *bitwise_and*.

```python
import cv2, numpy as np

# Innlest bilde
img = cv2.imread('Github/Eksperimentelt/Bilder/Konteiner/mate_5.png')

# Formatert fra BGR format til HSV-format
hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)

# Fargegrenser: Limits = [Øvre grense, Nedre grense]
#       - Grensene angir lilla farger.
limits = [np.array([132, 50, 90], dtype='uint8')
    ,np.array([137, 255, 255], dtype='uint8')]

# Maskering av bildet med angitte fargegrenser
mask = cv2.inRange(hsv, limits[0], limits[1])
    # mask:  svart-hvitt-bilde:
    #       Hvit == innefor grensene.
    #       Sort == utenfor grensene.

# Klipper ut farget område fra original bildet
cut_img = cv2.bitwise_and(img, img, mask=mask)

# Viser bildene
cv2.imshow('Original bildet', img)
cv2.waitKey()
cv2.imshow('Maskering', mask)
cv2.waitKey()
cv2.imshow('Utklipt område', cut_img)
cv2.waitKey()
```

Kodesnutt 1: Bruk av inRange og HSV til maskering og utklipping av farger.



(a) Original bildet.

(b) Maskering

(c) Klipt bilde

Figur 25: Resultat fra kodesnutt 1. Maskering og utklipp av de lilla fargetonene fra et bilde. I kodesnutt 1 linje 11 og 12 velges fargetonene som maskeres ut. I dette tilfellet er fargetonene med en verdi mellom 132 og 137 valgt. Fargetonen til lillafargen i bildet ligger mellom 133 og 136. Rødfargen ligger i området fra 177 til og med 0. Brunfargen ligger i området fra 13 til 17. Dermed står vi kun igjen med det lilla området.

3.2 Terskling

En annen utbredt metode, som vi har benyttet oss lite av, er terskling (engelsk: "thresholding"). Ved en slik metode gjøres bildet først om til gråskala. Deretter må en terskelverdi for hva som skal bli hvitt i det resulterende svart-hvitt-bildet bestemmes.

Rundt kantene til objekter i et bilde vil det alltid være en gradvis avtagende intensitet. Figur 26a viser dette.

(a) Før terskling

(b) Etter terskling

Figur 26: Resultat av en binær terskelfunksjon. Gråtonene splittes til sorte og hvite områder. Bildet blir altså binært bilde, bestående av to farger (svart og hvitt i dette tilfellet).

Terskelfunksjonen brukes ofte til å lage et binært sort-hvitt-bilde. OpenCV har en egen funksjon, *threshold*[58], som utfører denne slags operasjoner.

Innganger

1. **src** - Inngangsbilde.
2. **thresh** - Ønsket terskelverdi.
3. **maxval** - Verdi som pikslene over terskel blir satt til (ved bruk av binærterskling).
4. **type** - Tersklingstype (se [59]).

Utganger

1. **Out** - Bilde etter terskling.
2. **Retval** - Brukt terskelverdi.

Et eksempel på bruk av funksjonen er vist i kodesnutt 2.

```
1 import cv2
2
3 img = cv2.imread('testbilde.jpg', 0)
4 ret, binar = cv2.threshold(img, 200, 255, cv2.THRESH_BINARY)
```

Kodesnutt 2: Eksempel på bruk av terskling.

Resultatet av koden i kodesnutt 2 er vist i figur 26.

En årsakt til at denne metoden blir lite brukt av oss er at fargemaskering utfører en liknenede operasjon, som i de fleste tilfeller har vist seg å være mer robust.

3.3 Konturer

I denne seksjonen skal vi se nærmere konturer i OpenCV[40]. En kontur i OpenCV er et objekt som inneholder punktene som omkranser et objekt. Dette er dermed konturen til objektet. Formatet til en kontur er et "numpy array"[27] som inneholder en liste med koordinatene til hver enkelt piksel i konturen. Når vi arbeider med konturer bruker vi som oftest bilder i svart-hvitt-format. Transformasjonen til sort-hvitt-bilde kan gjøre på forskjellige måter. Som oftest bruker vi fargemaskering (se seksjon 3.1). En annen utbredt metode, som vi benytter oss svært lite av, er terskling (se seksjon 3.2).

Konturer kan blant annet brukes til objektdeteksjon og analyse av geometriske egenskaper. OpenCV har en rekke funksjoner og metoder designet for slike oppgavene. Vi går gjennom og forklarer en rekke av dem i denne seksjonen.

1. **findContours():** Brukes til å finne konturer i gråskala/svart-hvitt-bilder.
2. **drawContours():** Brukes til å tegne konturer.
3. **contourArea():** Beregner arealet til en kontur.
4. **boundingRect():** Returnerer omkransende rektangel til en kontur.
5. **minAreaRect():** Returnerer et rortert rektangel. Dette tilsvarer det minste mulige omkransende rektanglet til en kontur.
6. **boxPoints():** Returner hjørnepunktene til et rektangel på formatet til minAreaRect().
7. **fitLine():** Returnerer en linje som representerer/følger kontures retning og posisjon.
8. **Gjennomsnittsfarge:** Hvordan finne gjennomsnittsfargen til en kontur.
9. **Størrelseforhold:** Hvordan finne størrelsesforholdet til en kontur.
10. **Utstrekning:** Hvordan finne utstrekningen til en kontur.
11. **Soliditet:** Hvordan finne soliditeten til en kontur.

3.3.1 findContours()[42]

Denne funksjonen skal finne og returnerer konturer fra et svart-hvitt- eller gråskalabilde. Funksjonen finner konturene ved å lete etter områder med lik intensitet. Benyttes et gråskalabilde behandles dette som et sort-hvitt-bilde/binært bilde hvor alle pikselverdiene over null anses som en. For at funksjonen skal finne de ønskede konturene bør inngangsbildet være et behandlet svart-hvitt-bilde.

Funksjonen returnerer også en form for familie hierarki som angir posisjonen til hver enkelt kontur relativt til de andre. Har du for eksempel en kontur inne i en annen kontur anses den første som barnet til siste. Denne posisjoneringsinformasjonen ligger i hierarki variabelen. I figur 27 illustreres deler av denne ordningen. For mer detaljert informasjon om forskjellige utgaver av hierarkiet, se nærmere i OpenCV sin dokumentasjon på dette[41]. Hierarkitype velges gjennom inngangsvariabelen "mode"/hentemodus. Med inngangsvariabelen "method"/tilnærmingsmetode kan vi blant annet velge om konturen skal inneholde alle punkter, eller kun de mest essensielle. For eksempel hvis konturen er en linje, velger vi mellom å få returnert alle punktene på linjen, eller om en kun skal ha det første og siste punktet.

Inngang:

1. **image:** Gråskalabilde, optimalt svart-hvitt-bilde.
2. **mode:** Hentemodus, valg av type hierarki/oppstilling.
3. **method:** Valg av type tilnærmingsmetode.
4. **offset=(dx,dy):** Angir forskyvning av hvert enkelt punkt i resultatet.

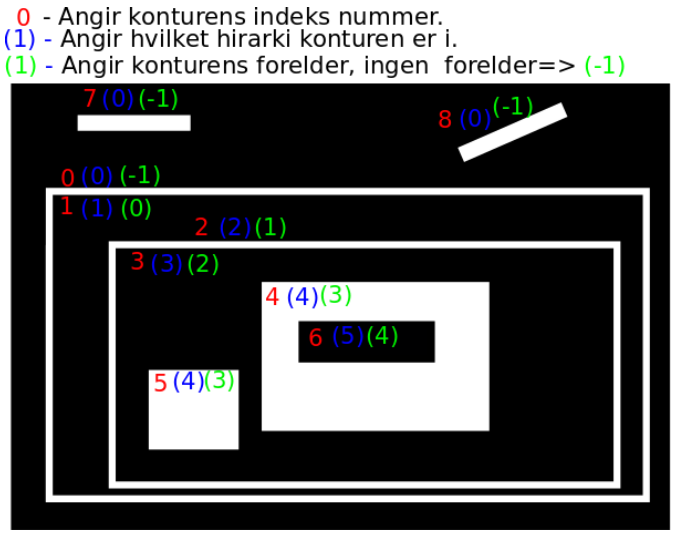
Utgang:

1. **contours:** Liste med alle konturene
2. **hierarchy:** Hierarki oversikt. Inneholder informasjon om konturenes posisjoner.

Eksempelkode:

```
import cv2, numpy as np

img = cv2.imread('test.jpg')                        # Innlest bilde
imgray = cv2.cvtColor(im, cv.COLOR_BGR2GRAY)        # Gråskalabilde
ret, sv_hv = cv2.threshold(imgray, 127, 255, 0)     # svart-hvitt-bilde

# Finn konturene, familie hierarki, kun nødvendig punkter.
konturer, hierarki = cv.findContours(so_hv, cv.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
```

Kodesnutt 3: Eksempel på bruk av FindContours.



Figur 27: Illustrasjon av konturenes hierarki og interne forhold. Konturene er angitt ved endring mellom svart og hvitt.

3.3.2 drawContours()[42]

Denne funksjonen tegner konturer inn i et valgt bilde. Det er en rekke parameter som kan justere hvordan tegningen skal bli.
Parameter:

1. **image:** Bildet vi tegner på. Kan ses på som både inngang og utgangsvariabel.
2. **contours:** Liste med konturer.
3. **contouridx:** Indeks til kontur som skal tegnes. For å tegne alle bruk -1.
4. **color:** Farge: for eks. =(R,G,B). Gråskala: for eks =255
5. **thickness:** Tykkelse på streken som konturen tegnes med. Negativt tall her gjør at en fylt kontur tegnes.
6. **lineType:** Valg av metode for tegning av konturen.
7. **hierarchy:** Hierarki er valgfritt. Ta med hvis maxLevel skal brukes.
8. **maxLevel:** Valgfri, brukes til å angi hvor mange nivå ned i hierarkiet som skal tegnes.
9. **offset = (dx,dy):** Forskyvning av det som blir tegnet. Valgfri variabel

Eksempelkode:

```
# Tegne alle konturene fra en liste.
cv2.drawContours(bilde, contours,-1, (0,255,0), 3)
# Tegne inn kontur med indeks nr 3.
cv2.drawContours(bilde, contours, 3, (0,255,0), 3)
# Tegne inn en enkelt kontur. Denne konturen er fylt
cv2.drawContours(bilde, [bare_en_kontur], 0, (0,255,0), thickness=-1)
```

Kodesnutt 4: Eksempel på bruk av DrawContours.

3.3.3 contourArea()[39]

Denne funksjonen regner ut arealet til en kontur. Vi sikter da til arealet av området konturen omkranser. Arealet er det samme som antall piksler inneholdt i konturen.

Inngang:

1. **contour:** Konturen vi regner ut arealet til.

3. **oriented=false:** Hvis denne parameteren blir aktivert vil arealet bli positivt eller negativt avhengig av orienteringen til konturen. Alternativt returneres arealets absoluttverdi.

Utgang:

1. **retval:** Arealet til konturen

Eksempelkode:

```
# Utregning av areal for en enkelt kontur
areal = cv2.contourArea(en_kontur)
# Kontur liste sortert etter areal.
liste_med_konturer.sort(key=cv2.contourArea, reverse=True)
# Finn største kontur i en liste basert på areal
største kontur = max(contours, key=cv2.contourArea)
```

Kodesnutt 5: Eksempel på bruk av *Area*.

3.3.4 boundingRect()[39]

Denne funksjonen definerer et rektangel som omkranser en kontur. Dette rektangelet er ikke vinklet på noen måte, men følger x og y aksen til bildet. Funksjonen returnerer et startpunkt og hvor lang ut fra dette punktet rektangelet strekker seg. I figur 28 illustreres dette omkransende rektangelet. Kodesnutt 6 viser hvordan funksjonen kan brukes.

Inngang:

1. **array:** Konturen er som tidligere beskrevet er en liste med punkter.

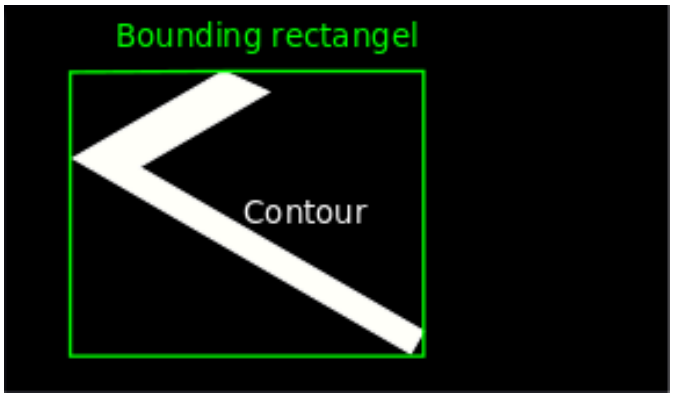
Utgang:

1. **retval:** (x-kordinat,y-kordinat,bredde,høyde)

Eksempelkode:

```
# Finner et rektangel som omkranser konturen.
x,y,bredde,høyde = cv2.boundingRect(kontur)
# Alternativt.
rektangel = cv2.boundingRect(kontur)
# Tegn rektangelet inn i et bilde.
cv2.rectangle(bilde,(x,y),(x+bredde,y+høyde),(34,255,154),2)
```

Kodesnutt 6: Eksempel på bruk av boundingRect.



Figur 28: Illustrasjon av omkransende rektangel.

3.3.5 minAreaRect()[39]

Denne funksjonen definerer det minste rektangelet som kan omkranse en gitt kontur. Rektangelet kan være vinklet. Funksjonen returnerer koordinatene til senter av rektangelet, rektangelets bredde, høyde og rotasjonen. Rotasjonsverdien kan være utfordrende å bruke da funksjonen bytter mellom å returnere rotasjonen til rektangelets kortsiden og langsiden. I figur 29 ser vi hvordan et slikt rektangel kan omkranse en kontur. Sammenlign gjerne med hvordan den andre funksjonen boundigRect() brukes på samme kontur i figur 28. Kodesnutt 7 viser hvordan denne funksjonen kan brukes.

Inngang:

1. **points:** Konturen er som tidligere beskrevet er en liste med punkter.

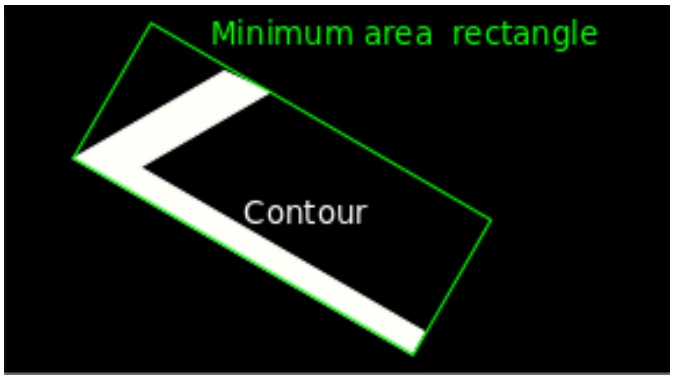
Utgang:

1. **retval:** Formatet til returverdiene, ((senter-x,senter-y), (bredde, høyde), rotasjonsvinkel)

Eksempelkode:

```
# Finner rektangelet som omkranser en kontur
rektangel = cv2.minAreaRect(kontur)
```

Kodesnutt 7: Eksempel på bruk av minAreaRect.



Figur 29: Forklarende bilde til beregning av en kontur sin utstrekning.

3.3.6 boxPoints()[39]

Denne funksjonen tar inn rektangelet som *minAreaRect*(3.3.5) returnerer. Den returnerer så en liste med hjørnekoordinatene til rektangelet. Denne listen er på samme form som en kontur, en liste med punkter.

Inngang:

1. **box:** Rektangel, ((senter-x,senter-y), (bredde, høyde), rotasjonsvinkel)

Utgang:

1. **points:** Liste med fire punkter, en kontur.

Eksempelkode:

```
boks = cv2.boxPoints(rektangel) # Returner rektangelets fire hjørner
boks = np.int0(boks)            # Konverter fra flyttall til heltall.
# Tegner rektangelet inn i et bilde.
cv2.drawContours(bilde,[boks],0,(42,66,255),2)
```

Kodesnutt 8: Eksempel på bruk av boxPoints.

3.3.7 fitLine()[43]

Funksjonen tar inn en liste med punkter (en kontur) og tilpasser en best mulig rett linje gjennom punktene(konturen). Funksjonen kan behandle både tredimensjonale og todimensjonale datasett. Kodesnutt 9 viser praktisk bruk av funksjonen. Figur 30 og 31 viser henholdsvis en kontur med tilpasset linje og konseptet bak inntegningsmetoden brukt i kodesnutt 9.

Inngang:

1. **points:** En liste med punkter (en kontur). Kan være to eller tre dimensjonale koordinater.
2. **distType:** Valg av metode for tilpasning av linje. For flere detaljer se OpenCV sin dokumentasjon[43].

3. **param:** Parameter for justering av linjetilpasningsmetodene. Hvis den settes lik null velges en optimal verdi.

Utgang:

1. **line:** Beskrivelse av linje ved hjelp av retningsvektor og ett punkt på linja. (vx,vy,x0,y0)

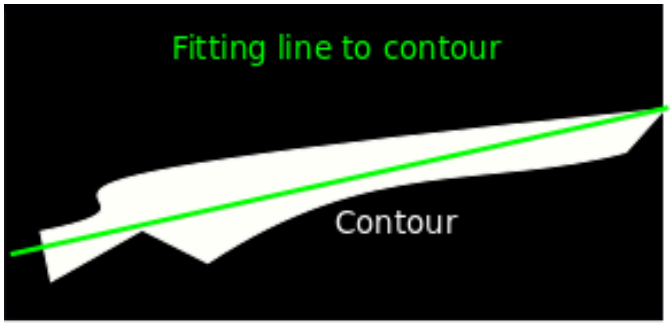
4. **reps:** Treffsikkerhetskrav - avstand mellom origo og linjen.

5. **aeps:** Treffsikkerhetskrav - linjens vinkel.

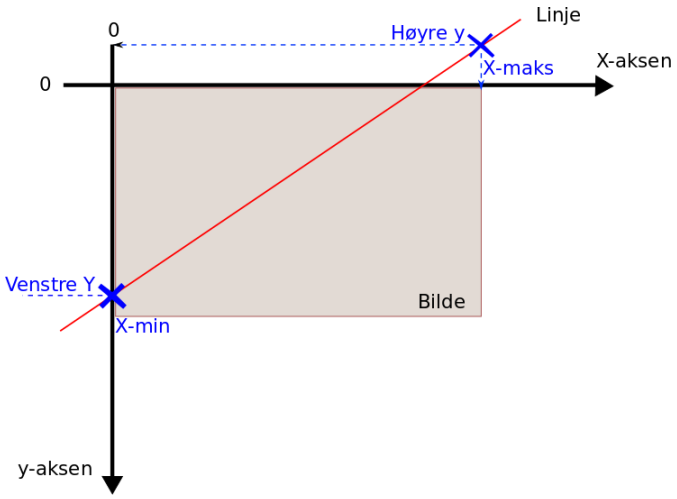
Eksempelkode:

```
# Tar i bruk fitLine funksjonen
[vx,vy,x,y] = cv.fitLine(kontur, cv2.DIST_L2,0,0.01,0.01)

# Tegne linjen inn på bildet.
rader, kolonner = bilde.shape[:2] # Henter ut størrelsen på bildet
# Punktet hvor linjen passerer null på x-aksen.
venstre_y = int((-x*vy/vx) + y)
# punktet hvor linjen passerer x-aksen ut av bildet.
høyre_y = int(((kolonner-x)*vy/vx)+y)
# Tegner inn linja ved å definere to punkter.
cv2.line(img,(kolonner-1,høyre_y),(0,venstre_y),(0,255,0),2)
```

Kodesnutt 9: Eksempel på bruk av *fitLine*. Figur 31 viser hvordan linjen tegnes inn på bildet.



Figur 30: Forklarende bilde for hvordan en konturs tilpassede linje kan se ut.



Figur 31: Illustrasjon av tegningsmetoden fra kodesnutt 9.

3.3.8 minEnclosingCircle()[39]

Denne funksjonen tar inn en liste med punkter (kontur) og finner den minste sirkelen som kan omkranse alle punktene. Kodesnutt 10 viser praktisk bruk av funksjonen. I figur 32 illustreres en kontur og dens tilhørende sirkel.

Inngang:

1. **points:** En liste med punkter (en kontur). Disse må være i to dimensjoner, x og y koordinater.

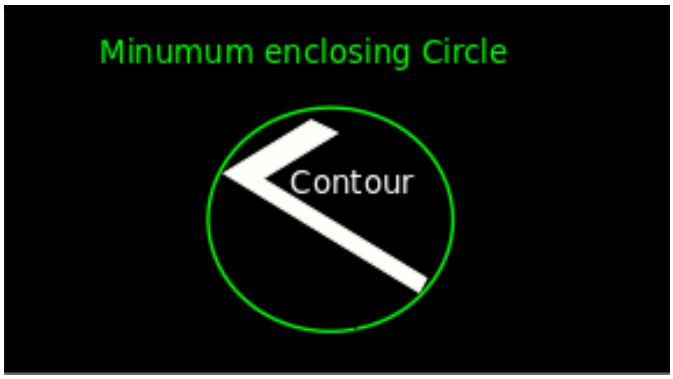
Utgang:

1. **center:** Koordinat til sirkelens senter, (x,y).
2. **radius:** Sirkelens radius.

Eksempelkode:

```
(x,y), radius = cv2.minEnclosingCircle(kontur)
senter = (int(x),int(y))
radius = int(radius)
# Tegner sirkelen inn i originalbildet
cv2.circle(bilde, senter,radius ,(0,255,0),2)
```

Kodesnutt 10: Eksempel på bruk av contourArea. Figur 31 viser hvordan linjen tegnes inn på bildet.



Figur 32: Illustrasjon av den minste omkransende sirkelen til en kontur.

3.3.9 Gjennomsnittsfarge:

For å finne gjennomsnittsfargen til en kontur benytter vi funksjonen *mean* [48] fra OpenCV. Denne funksjonen tar inn et bilde samt en maskering som definerer konturen/området vi skal undersøke. I retur får vi snittverdien av alle pikslene inneholdt i konturen.

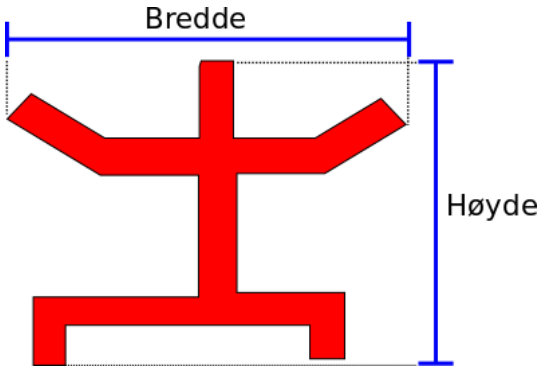
Eksempelkode:

```
# 3.4 Regner ut gjennomsnittlig farge av maskert område.
 snitt_farge = cv2.mean(bilde,mask=maskert_område)
```

Kodesnutt 11: Eksempel på bruk av OpenCV sin "mean" funksjon

3.3.10 Størrelsesforholdet:

Med størrelsesforhold sikter vi til forholdet mellom bredden og høyden til en kontur. I figur 33 illustreres det hvilke bredde og høyde det er snakk om. Likning 1 viser hvordan størrelsesforholdet skal beregnes.



Figur 33: Illustrasjon av en kontur sin bredde og høyde

$$Størrelsesforhold = \frac{Bredde}{Høyde} \tag{1}$$

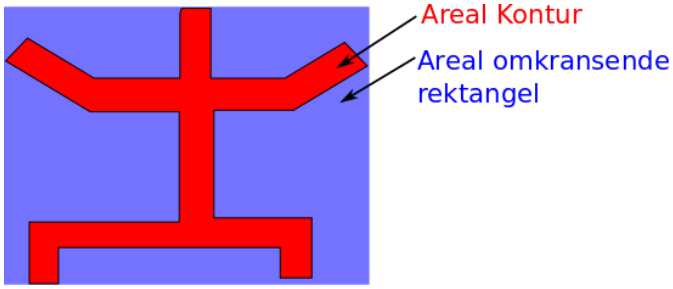
Eksempelkode:

```
x,y,bredde,hoyde = cv2.boundingRect(kontur)         # Alternativ 1
x,y,bredde,hoyde,vinkel = cv2.minAreaRect(kontur) # Alternativ 2

storreksesforhold = bredde/hoyde                    # Størrelsesforhold
```

Kodesnutt 12: Eksempel på utregning av størrelsesforholdet.

3.3.11 Utstrekning:

Med utstrekning sikter vi til andelen av konturen som fyller et omkransende rektangel. Figur 34 illustrerer hvilke areal det er snakk om. Likning 2 viser hvordan utstrekningen beregnes.



Figur 34: Forklarende bilde til beregning av en kontur sin utstrekning.

$$Utstrekning = \frac{Areal\ kontur}{Areal\ rektangel} \tag{2}$$

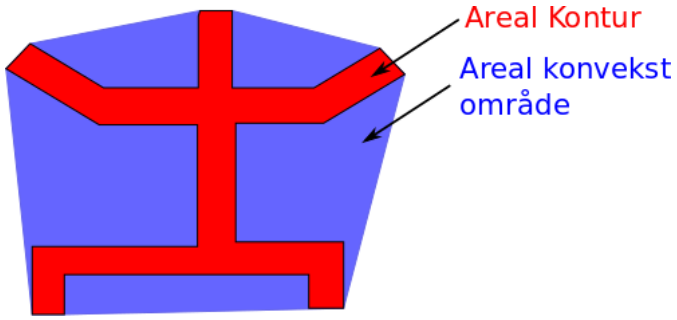
Eksempelkode:

```
x,y,bredde,hoyde = cv2.boundingRect(kontur)         # Alternativ 1
x,y,bredde,hoyde,vinkel = cv2.minAreaRect(kontur) # Alternativ 2

areal = cv2.contourArea(kontur)                     # Areal av kontur
areal_rektangel = bredde*hoyde                      # Areal rektangel
utstrekning = areal/areal_rektangel                 # Utstrekning
```

Kodesnutt 13: Eksempel på utregning av utstrekning.

3.3.12 Soliditet:

Med soliditet sikter vi til konturens areal delt på areal av det konvekse området. Figur 35 illustrerer hvilke areal det er snakk om. Likning 3 viser hvordan soliditeten beregnes.



Figur 35: Forklarende bilde til beregning av en kontur sin soliditet.

$$Soliditet = \frac{Areal\ kontur}{Areal\ konvekst\ omrade} \tag{3}$$

Eksempelkode:

```
areal = cv2.contourArea(kontur)                    # Areal av kontur
konvekst_område = cv2.convexHull(kontur)           # Konvekst område
konvekst_areal = cv2.contourArea(konvekst_område)# Areal Konvekst område
soliditet = areal/konvekst_areal                   # Soliditet
```

Kodesnutt 14: Eksempel på utregning av soliditet.

3.4 Geometrisk transformasjon

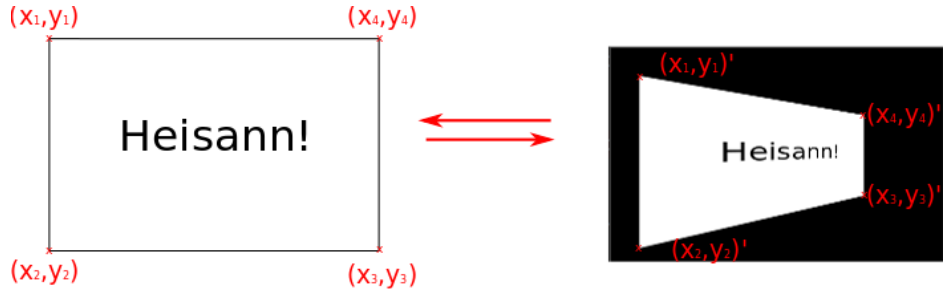
I OpenCV biblioteket er det to mye brukte metoder, en for perspektivtransformasjon og en for "affine"-transformasjon. Begge disse metodene kan brukes til mange av de samme operasjonene, men det er noe vesentlig som skiller dem. I tillegg til det vi benytter disse to metodene til, benytter vi oss av en funksjon for enkel rettvinklet rotasjon og en for skalering av bilder.

Denne seksjonen inneholder:

1. Perspektivtransformasjon
   (a) getPerspectiveTransform()
   (b) warpPerspective()
2. "Affine"-transformasjon
   (a) getAffineTransform()
   (b) warpAffine()
   (c) Rotasjon
   (d) Translasjon
3. rotate()
4. resize()

3.4.1 Perspektivtransformasjon:

Denne transformasjonen bevarer rette linjer, men de forblir ikke nødvendigvis parallelle. Som selve navnet antyder, kan denne funksjonen brukes til å endre perspektivet til bildet. Altså hvor vi tilsynelatende ser innholdet på bildet fra. For å gjennomføre en slik transformasjon trenger vi fire startpunkt og fire sluttpunkt. Hvert enkelt start- og sluttpunkt hører sammen, og sluttpunktene angir hvor startpunktene skal havn i det nye bildet. I figur 36 er en slik transformasjon illustrert. For å gjennomføre en slik operasjon bruker vi funksjonene getPerspectiveTransform() og warpPerspective().



Figur 36: Eksempel på perspektivtransformasjon. Langsidene blir forskjøvet slik at de ikke lengre er parallelle. Til gjengjeld blir bildet endret slik at det naturlige perspektivet beholdes.

getPerspectiveTransform()[45]
Fra ett sett med fire startpunkt og et annet sett med fire sluttpunkt lager denne funksjonen en 3x3 matrise. Denne matrisen brukes av warpPerspective() til å utføre selve transformasjonen. Likningen i figur 37 viser sammenhengen mellom matrisen, start- og sluttpunktene. Rekkefølgen på punktene er svært viktig da startpunkt nr. 1 er bundet til sluttpunkt nr. 1, osv. I kodesnutt 15, linje 11, er funksjonen tatt i bruk.

Inngang:

1. **src:** Fire startpunkt, altså punkter fra bildet som skal transformeres
2. **dst:** Fire sluttpunkt. Disse angir hvor de fire startpunktene skal være på sluttresultatet.

Utgang:

1. **retval:** 3x3 Matrise som kan brukes til å transformerer bildet.

$$\begin{bmatrix} t_i x'_i \\ t_i y'_i \\ t_i \end{bmatrix} = \texttt{map\_matrix} \cdot \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

$$dst(i) = (x'_i, y'_i), src(i) = (x_i, y_i), i = 0, 1, 2, 3$$

Figur 37: Map_matrix er Matrisen funksjonen produsere. Bildet av likningen er hentet fra [45].

warpPerspective()[63]

Funksjonen tar inn et bilde og en transformasjonsmatrise. Den returnerer så et transformert bilde. I likning 4 vises det hvordan verdiene fra transformasjonsmatrisen brukes til å beregne hvor hver enkelt av pikslene i inngangsbilde havner i utgangsbildet. I kodesnutt 15, linje 13, er funksjonen tatt i bruk.

Inngang:

1. **src:** Inngangsbildet. Bildet som skal transformeres.
2. **M:** 3x3 transformasjonsmatrise.
3. **dsize:** Størrelsen på utgangsbildet.
4. **flags:** Valg av interpolasjonsmetode, eller invertering av transformasjonen.
5. **borderMode:** Pikselekstrapolasjonsmetode. Ved transformasjon vil vi ofte gå utenfor det området som er definert i originalbildet. I figur 36 er dette området sort.
6. **borderValue:** Hvis borderMode er satt til en konstant angis denne konstanten her. Som standard er denne 0, altså sort.

Utgang:

1. **dst:** Utgangsbildet. Bildet som har blitt transformert.

$$(x, y) - Pikselkoordinat. \qquad dst - Utgangsbildet.$$
$$M - Tranformasjonsmatrise. \qquad src - Inngangsbildet.$$

$$src(x, y) = dst(\frac{M_{11} \cdot x + M_{12} \cdot y + M_{13}}{M_{31} \cdot x + M_{32} \cdot y + M_{33}}, \frac{M_{21} \cdot x + M_{22} \cdot y + M_{23}}{M_{31} \cdot x + M_{32} \cdot y + M_{33}}) \tag{4}$$

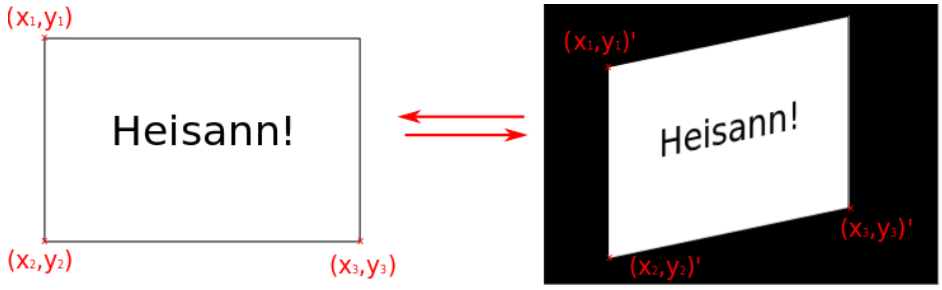
Eksempelkode perspektivtransformasjon:

```
1  bilde = cv2.imread('eks.png')
2  # Dimensjoner på inngangbildet
3  hoyde, bredde = bilde.shape[:2]
4  # Definerer fire startpunkt
5  src_points = np.array([[0, 0], [0, hoyde-1], [bredde-1, hoyde-1],
6                         [bredde-1, 0]], dtype=np.float32)
7  # Definerer fire sluttpunkt
8  dst_points = np.array([[20, 20], [20, hoyde-10], [bredde-60, hoyde-50],
9                         [bredde-60, 50]], dtype=np.float32)
10 # Lager transformasjonsmatrise
11 matrise = cv2.getPerspectiveTransform(src_points, dst_points)
12 # Bruker matrisen til og transformere bildet.
13 utgangsbilde = cv2.warpPerspective(bilde, matrise, (bredde,hoyde))
```

Kodesnutt 15: Eksempel på utførelse av perspektivtransformasjon. Bildet før transformasjon og etter transformasjon er vist i figur 36.

3.4.2 "Affinetransformasjon:

I matematikken er "affine" et adjektiv som beskriver noe som åpner for, samt bevarer parallelle forhold. Med andre ord så bevarer denne transformasjonen alle parallelle linjer i bildet. For å ta i bruk denne metoden trenges tre startpunkt og tre sluttpunkt. Metoden transformerer bildet slik at startpunktene havner på sluttpunktenes posisjon. Dette er illustrert i figur 38. Ved å sammenligne dette bildet med resultatet ved perspektiv transformasjon i figur 36, kan en se forskjellen på bevarte og ubevarte parallelle linjer.
For å gjennomføre en slik operasjon bruker vi funksjonene getAffineTransfrom()[44] og warpAffine()[62] fra OpenCV. Disse to funksjonene er relativt like de vi så ved perspektivtransformasjon, og har henholdsvis sine motstykker i getPerspectivTransform() og warpPerspective().
Denne transformasjonsmetoden har vi også benyttet oss av til å lage to enkle funksjoner. En til translasjon (forskyvning uten rotasjon) og en til rotasjon.



Figur 38: Eksempel på"affin"-tranformasjon. Alle parallelle forhold/linjer bevares.

getAffineTransform()[44]

Fra ett sett med tre startpunkt og et sett med tre sluttpunkt lager denne funksjonen en 2x3 matrise. Denne matrisen brukes av warpAffine() til å utføre selve transformasjonen. Likningen i figur 39 viser sammenhengen mellom matrisen og start- og sluttpunktene. Rekkefølgen på punktene er svært viktig da startpunkt nr. 1 er bundet til sluttpunkt nr. 1, osv. I kodesnutt 16, linje 16, er denne funksjonen tatt i bruk.

Inngang:

1. **src:** Tre startpunkt. Detter er punkter fra bildet som skal transformeres.
2. **dst:** Tre sluttpunkt, som representerer hvor de tre startpunktene skal være på sluttresultatet.

Utgang:

1. **retval:** 2x3 Matrise som kan brukes til å transformerer bildet.

$$\begin{bmatrix} x_i' \\ y_i' \end{bmatrix} = \texttt{map\_matrix} \cdot \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

$$dst(i) = (x_i', y_i'), src(i) = (x_i, y_i), i = 0, 1, 2$$

Figur 39: Map_matrix er Matrisen funksjonen produsere. Bildet av av likningen er hentet fra [44].

warpAffine()[62]

Funksjonen tar inn et bilde og en transformasjonsmatrise. Den returnerer så et transformert bilde. I likning 5 vises det hvordan verdiene fra transformasjonsmatrisen brukes til å beregne hvor hver enkelt av pikslene i inngangsbilde havner i utgangsbildet. I kodesnutt 16, linje 18-19, er funksjonen tatt i bruk.

Inngang:

1. **src:** Inngangsbildet. Bildet som skal transformeres.
2. **M:** 2x3 transformasjonsmatrise.
3. **dsize:** Størrelsen på utgangsbildet.
4. **flags:** Valg av interpolasjonsmetode, eller invertering av transformasjonen.
5. **borderMode:** Pikselekstrapolasjonsmetode. Ved transformasjon vil vi ofte gå utenfor det området som er definert i originalbildet. I figur 36 er dette området sort.
6. **borderValue:** Hvis borderMode er satt til en konstant angis denne konstanten her. Som standard er denne 0, altså sort.

Utgang:

1. **dst:** Utgangsbildet. Bildet som har blitt transformert.

$$(x, y) - Pikselkoordinat\ inngangsbildet. \qquad dst - Utgangsbildet.$$
$$M - Tranformasjonsmatrise. \qquad src - Inngangsbildet.$$

$$src(x, y) = dst(M_{11} \cdot x + M_{12} \cdot y + M_{13}, M_{21} \cdot x + M_{22} \cdot y + M_{23}) \tag{5}$$

Eksempelkode "affine"-transformasjon:

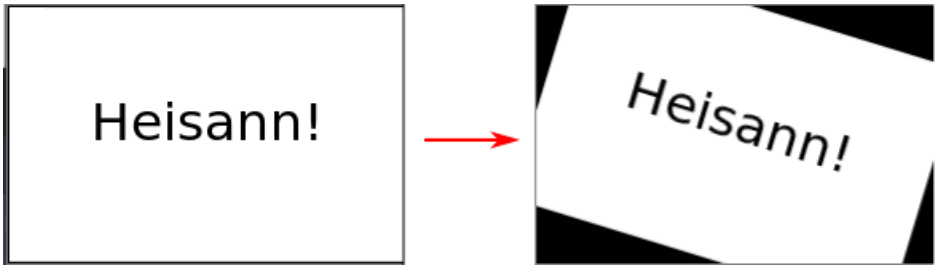
```
bilde = cv2.imread('eks.png')

#Dimensjoner på inngangbildet
hoyde, bredde = bilde.shape[:2]

# Definerer fire startpunkt
src_points = np.array([[0, 0], [0, hoyde-1], [bredde-1, hoyde-1], dtype=np.float32)

# Definerer tre sluttpunkt:
fors = 50 # Forskyver sluttpunktene/destinasjonspunktene et antall piksler ut
          # i bildet  slik at ikke det ikke havner i negative pikselkordinater.
dst_points = np.array([[fors, fors], [fors, fors + hoyde - 10],
            [fors + bredde - 60, fors + hoyde - 50]], dtype=np.float32)

# Lager transformasjonsmatrise:
matrise = cv2.getAffineTransform(src_points, dst_points)
# Bruker matrisen til og transformere bildet.
utgangsbilde = cv2.cv2.warpAffine(bilde, affine_matrix,
                                  (bredde+10+fors,hoyde+10+fors))
```

Kodesnutt 16: Eksempel på utførelse av "affine"-transformasjon. Bildet før transformasjon og etter transformasjon er vist i figur 38.

Rotasjon:
Dette er en funksjon som tar inn et bilde og en vinkel. Den returnerer så et bilde som har blitt rotert. En positiv vinkel tilsvarer en rotasjon mot klokken. Rotasjonen skjer rundt bildets senterpunkt. Funksjonen tar i bruk OpenCV sin getRotationMatrix2D[46] og warpAffine[62]. Den første av disse brukes til å lage en rotasjonsmatrise og den andre brukes til å utføre rotasjonen på bildet. I kodesnutt 17 linje 1-6 er funksjons kode, og i figur 40 et eksempel på bruk av denne.

```
1  def rotation(img,angle):
2      hoyde, bredde = img.shape[:2]
3      matrise = cv2.getRotationMatrix2D(center=(cols / 2, rows / 2),
4                                        angle=angle, scale=1)
5      rst = cv2.warpAffine(img, matrise, (hoyde, bredde))
6      return rst
7
8  # Laster inn et testbilde
9  bilde = cv2.imread('test.png')
10
11 # Tar funksjonen ovenfor i bruk
12 resultat = rotation(bilde,-17)
```

Kodesnutt 17: Koden til funksjonen som utfører rotasjon samt testing. Resultatet av testen er vist i figur 40.



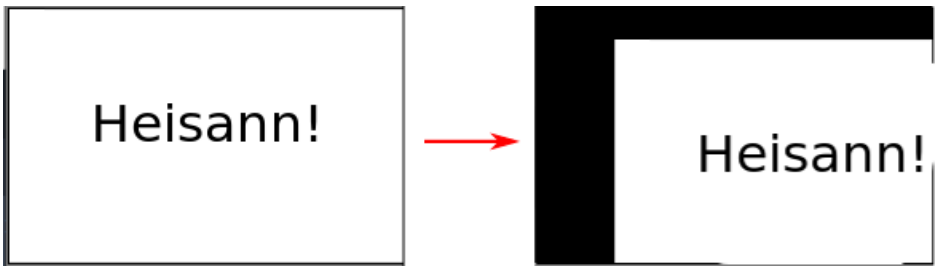
Figur 40: Resultatet av rotasjonen i kodesnutt 17
. Til venstre er bildet før rotasjonen. Til høyre er bildet etter rotasjonen. Legg merke til at noe av innholdet i originalbildet ender utenfor rammen. Tomrommet erstattes med sort.

Translasjon

Dette er en enkel funksjon som tar inn ett bilde samt ønsket forskyvning langs x- og y-aksen. Den lager så en transformasjons matrise og benytter warpAffine[62] til å utføre operasjonen på bildet. Til slutt returneres bildet som har blitt forskjøvet. I kodesnutt 18 linje 1-5 er funksjons kode, og i figur 41 et eksempel på bruk av denne.

```
1  def translation(img,x,y):
2      rows, cols = img.shape[:2]
3      matrise = np.float32([[1, 0, x], [0, 1, y]])
4      rst = cv2.warpAffine(img, matrise, (cols, rows))
5      return rst
6
7  # Laster inn et testbilde
8  bilde = cv2.imread('test.png')
9
10 # Tar funksjonen ovenfor i bruk
11 resultat = translation(bilde,50,20)
```

Kodesnutt 18: Koden til funksjonen som utfører translasjon samt testing. Resultatet av testen er vist i figur 41.



Figur 41: Resultatet av forskyvningen i kodesnutt 18
. Til venstre er bildet før forskyvningen. Til høyre er bildet etter forskyvningen. Legg merke til at noe av innholdet i originalbildet blir forskjøvet ut, og tomrommet erstattet med sort.

3.4.3 rotate()

Dette er en enkel funksjon til enkel 90° rotasjon av bilder. Funksjonen er godt dokumentert blant annet i OpenCV sin dokumentasjon [54]. I kodesnutt 19 vises tre eksempler på bruk av funksjonen. Funksjonen gjør det samme som vår egen rotasjons funksjon 3.4.2 gjør for valgfrie vinkler.

```
rotert_bilde_1 = cv2.rotate(bilde, cv2.ROTATE_90_CLOCKWISE)
rotert_bilde_2 = cv2.rotate(bilde, cv2.ROTATE_90_COUNTERCLOCKWISE)
rotert_bilde_3 = cv2.rotate(bilde, cv2.ROTATE_180)
```

Kodesnutt 19: Eksempel på bruk av OpenCV sin "rotate" funksjonen.

3.4.4 resize()

Dette er en enkel funksjon til skalering av bilder. Med skalering mener vi det å gjøre bilder større eller mindre. Funksjonen er godt dokumentert blant annet i OpenCV sin dokumentasjon [53]. I kodesnutt 20 vises to eksempler på bruk av funksjonen.

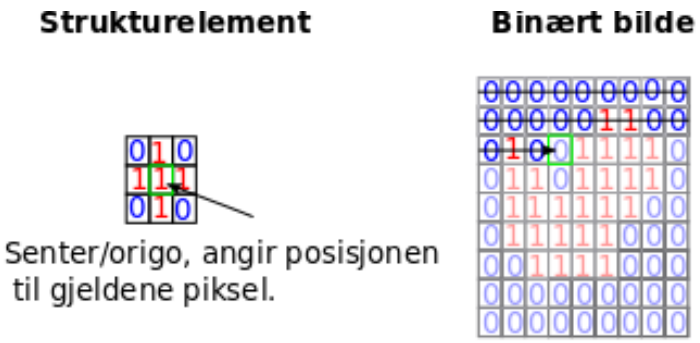
```
# INTER_LINEAR(raskest) eller INTER_CUBIC(best) når vi utvider et bilde.
img_scaled = cv2.resize(img, None, fx=1.2, fy=1.2, interpolation=cv2.INTER_LINEAR)
    # Bruker faktoren fx og til å angi mengden av reduksjon/utvidelse.

# INTER_AREA anbefales brukt når vi krymper et bilde.
redusert_bilde = cv2.resize(bilde, (90, 90), interpolation=cv2.INTER_AREA)
    # Benytter her "(90,90)" til å angi bildets ny størrelse.
```

Kodesnutt 20: Eksempel på bruk av "resize" funksjonen.

3.5 Morfologiske operasjoner

Ordet morfologi brukes i flere andre sammenhenger, men relateres ofte til beskrivelse, undersøkelse av, og læren om former. I denne sammenheng sikter vi til enkle operasjoner på former (objekter) i et bilde. Disse formene er typisk konturer i ett binært svart-hvitt-bilde. Vi bruker forskjellige morfologiske operasjoner til å undersøke, forbedre og behandle et bilde. I denne seksjonen går vi kort gjennom noen av de grunnleggende funksjonene. Dette er et emne som er nøye dokumentert og omtalt av andre. Derfor er denne gjennomgangen noe overflatisk. For mer utfyllende forklaring og dokumentasjon se OpenCVs dokumentasjon[51] og kapittel tre i læreboken "Computer Vision"[17].

I figur 42 introduseres strukturelementet. Strukturelementet er en binær matrise av valgfri størrelse og utforming. En slik matrise blir brukt i de to grunnleggende morfologiske operasjonene erosjon og dilasjon. Denne matrisen er en viktig del av den vurderingen som blir gjort for hver piksel i bildet. Utformingen av strukturelementet er avgjørende for resultatet. I bruk vil strukturelementet dras gjennom bildet for å bestemme den nye tilstanden til hver enkelt piksel. Det er kun enerene i strukturelementet som har betydning. Spørsmålet som de morfologiske operasjonene stiller er hvilke enere fra strukturelementet som treffer enere i bildet. Avhengig av hvilke morfologisk operasjon vi bruker avgjør dette tilstand til pikslene i det behandlede bildet. Pikslene "utenfor" bildet regnes alltid lik null.



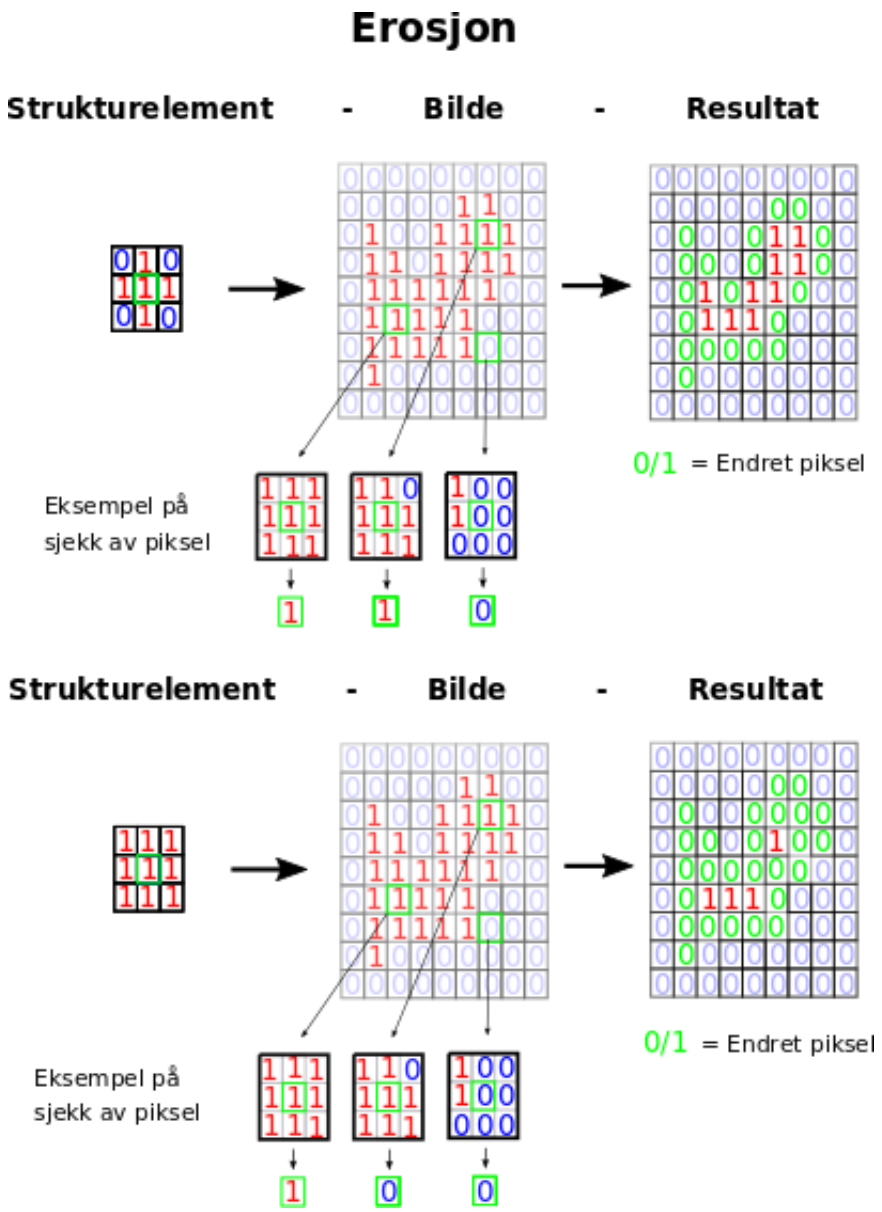
Figur 42: Illustrasjon av strukturelementet og hvordan dette dras gjennom bildet piksel for piksel.

3.5.1 Erosjon:

- Passer strukturelementet til formene/objektene i bildet?
- Overlapper alle enerne fra strukturelementet en ener for gjeldede piksel?

Hvis ja blir resultatet for gjeldene piksel 1, ellers 0.

Erosjon krever altså at hver ener fra strukturelementet overlapper en ener på bildet. Dette strenge kravet fører til at bildet blir redusert/erodert. Erosjon er på mange måter en logisk OG-operasjon. Inngangsverdiene er pikslene som blir truffet av strukturelementets enere og resultatet legges inn i gjeldene piksel. Figur 43 illustrerer dette ved og bruke to forskjellige strukturelementer til å behandle et bilde.



Figur 43: Illustrasjon av den morfologiske operasjonen erosjon.

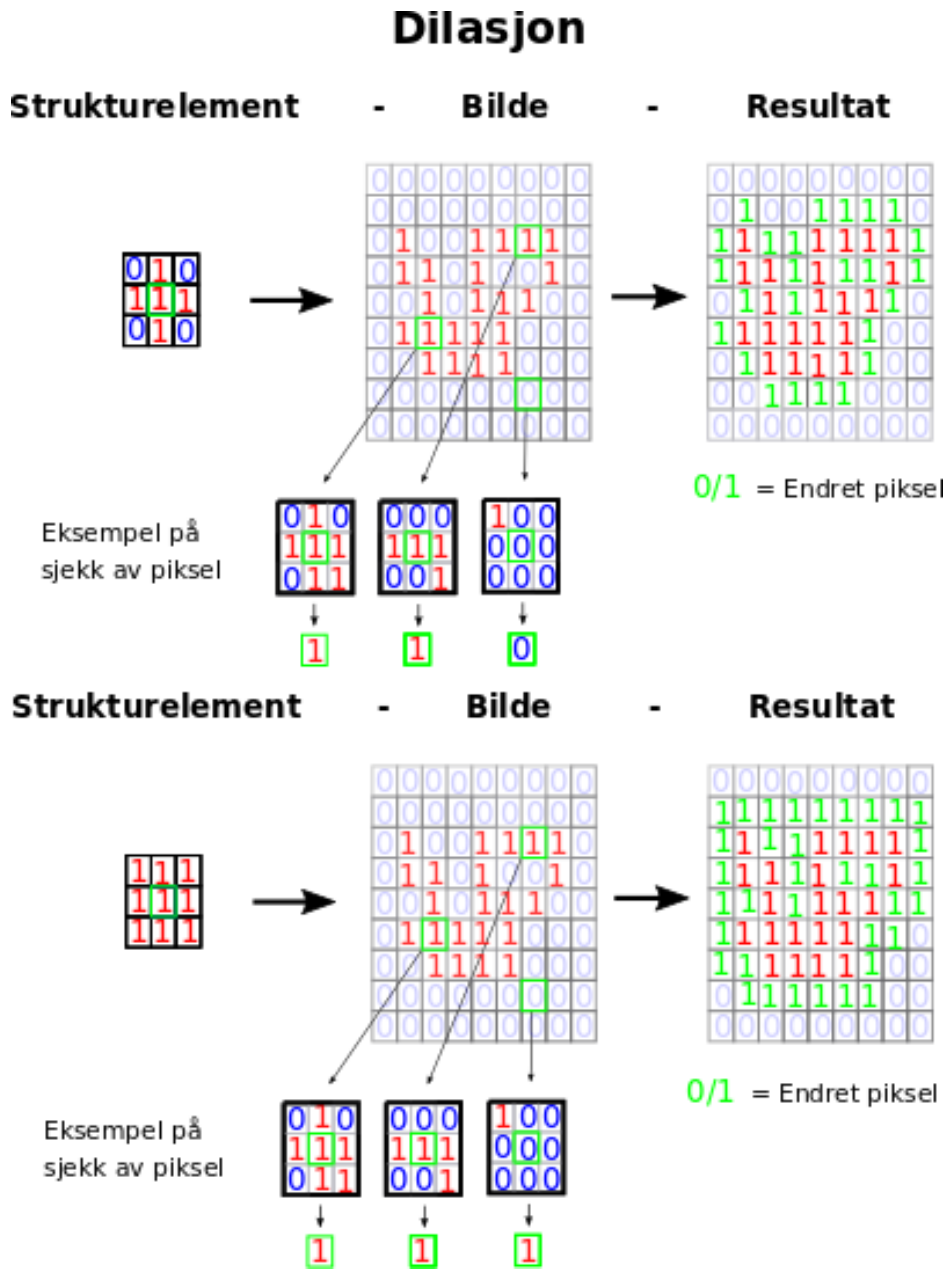
Erosjon sin effekt på et bilde:

- Krymper objekter.
- Piksler fjernes fra innsiden av hule objekt.
- Små utstikk og ujevnheter fjernes fra objektets omriss.
- Innbuktninger utvides.
- Stort strukturelement, eller flere gjentagelser gir mer erosjon.
- Kan potensielt brukes til kantdeteksjon ved å subtrahere resultatet av en erosjon med originalbildet. Da vil omrisset til eventuelle objekter stå igjen. Et slags eksempel på dette kan ses øverst til høyre i figur 43. De grønne pikslene i resultatet tilsvarer omrisset.

3.5.2 Dilasjon:

- Treffer strukturelementet formene/objektene i bildet?
- Treffer en eller flere av enerene fra strukturelementet en ener for gjeldene piksel?

Hvis ja blir resultatet for gjeldene piksel 1, ellers 0.
Dilasjon krever kun at en ener fra strukturelementet treffer en ener på bildet. Dette kravet er ikke veldig strengt, dermed utvides formene/objektene i bildet. Slik som erosjon tilsvarer en logisk OG-operasjon, tilsvarer dilasjon en ELLER-operasjon. Inngangsverdiene er pikslene som blir truffet av strukturelementets enere og resultatet legges inn i gjeldene piksel.



Figur 44: Illustrasjon av den morfologiske operasjonen erosjon.

Dilasjon sin effekt på et bilde:

- Utvider objekter
- Fyller opp små hull
- Små utstikk og ujevnheter glattes ut.
- Innbuktninger utvides.
- Stort strukturelement, eller flere gjentagelser gir mer dilasjon.

3.5.3 Åpning:

Først erosjon deretter dilasjon gir det som kalles en morfologisk åpning. Denne operasjonen fjerner utvendig støy og frakobler objekter som er svakt sammenknyttet. Dette kan være en fin metode til å separer et stort objekt fra støy og smårusk.

3.5.4 Lukking:

Først dilasjon deretter erosjon gir det som kalles en morfologisk lukking. Denne operasjonen fjerner innvendig støy og kobler sammen objekter som ligger nært til hverandre.

3.5.5 Eksempel på bruk i praksis:

Eksempelkode:

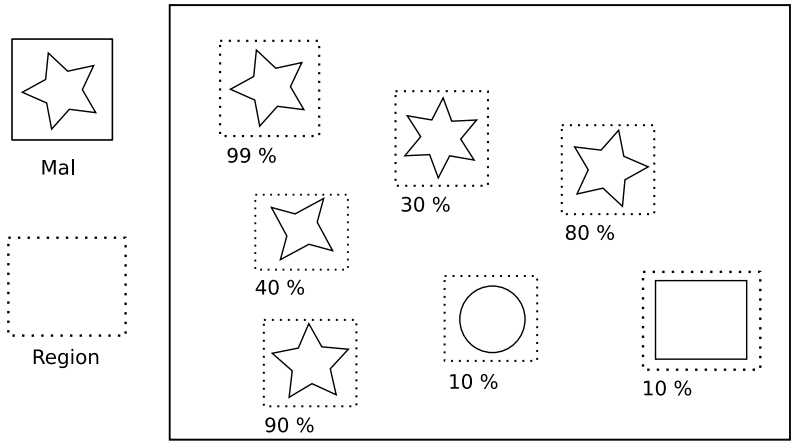
```
# Strukturelement:
#  - Vi kan designe egne strukturelement eller hente fra OpenCV
#  - Egenprodusert 5x5 matrise helt full med enere.
kernel = np.ones((5,5),np.uint8)
#  - Ferdilaget 5x5 korsformet matrise.
kernel = cv2.getStructuringElement(cv2.MORPH_CROSS,(5,5))

erosjon = cv2.erode(bilde,kernel,iterations = 1)   # Erosjon
dilasjon = cv2.dilate(bilde,kernel,iterations = 1) # Dilasjon

apning = cv2.morphologyEx(bilde, cv2.MORPH_OPEN, kernel)  # Åpning
lukking = cv2.morphologyEx(bilde, cv2.MORPH_CLOSE, kernel)# Lukking
```

Kodesnutt 21: Eksempel på kode til morfologiske operasjoner i Python.

3.6 Malsammenligning

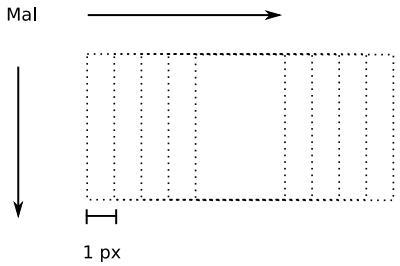
For å identifisere objekter i et bilde er det vanlig å bruke maskinlæring. Maskinlæringsalgoritmer trenger gjerne flere tusen opplæringsbilder av hvert objekt som skal gjenkjennes. Malsammenligning krever bare ett bilde (en mal) av hvert objekt vi vil identifisere. Her brukes et bilde av et kjent objekt som utgangspunkt for å finne tilsvarende objekter i et annet bildet. En viktig forutsetning for å finne igjen et objekt, er at de er like store i malbilde og bildet som det letes i. Vi benytter oss av implementasjonen i Python med funksjonen *matchTemplate* fra OpenCVs bibliotek [56].

Funksjonen *matchTemplate* virker slik at den tar inn et bilde og en mal. Malen sveipes over bildet i jakten på tilsvarende deler i dette bildet. Metoden returnerer en matriserepresentasjon av bildet der hvert element i matrisen er verdien til en piksel. Denne verdien bestemmes av likheten til malen i forhold til tilsvarende regioner i bildet. Figur 45 viser prinsippet.



Figur 45: Teoretisk treffprosent for objekter som blir oppdaget ved å bruke malsammenligning.

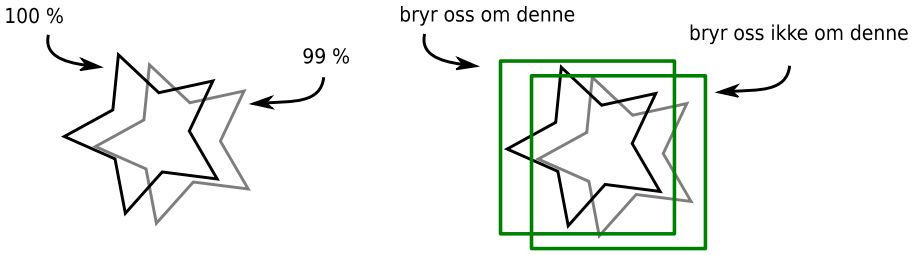
Figur 45 Viser det teoretisk prosentvise samsvaret mellom malen og noen relevante regioner i bildet. Det er bare tegnet regioner rundt objekter for lesbarhetens skyld. I praksis vil det bli mange regioner, med én piksel høyde og bredde, mellom hver. Dette er vist i figur 46.



Figur 46: Illustrasjon av at regionen som blir undersøkt flytter seg med en piksel av gangen. Malen dras gjennom bildet på en liknende måte som strukturelementet i morfologiske operasjoner i seksjon 3.5. Resultatet fra hver enkelt sammenligning legges inn i pikselen øverst til venstre i regionen som blir undersøkt.

likhetsprosenten som bestemmes legges inn i pikselen som treffes av malbildets venstre hjørne. Figur 47 viser hvordan malsammenligningsfunksjonen ville tolket piksel verdiene. Dersom malen

passer godt med området i bildet vil pikselen oppe til venstre få en høy verdi. Pikslene rundt vil få nesten like høye verdier.



Figur 47: Bildet illustrerer hvordan nærliggende piksler får nesten like høy treffprosent/intensitet som det punktet.

Området der malen passer best vil ofte ha flere piksler rundt som også har høye verdier. I hele området rundt den aktuelle regionen vil det bli gjort funn som er tilnærmet like objektet på malbildet. Den resulterende intensitet/treffprosent blir høy ,men ett punkt vil bli høyest i den aktuelle regionen.

Formel 6 viser en av de matematiske metodene (TM_SQDIFF) som OpenCV bruker for å sjekke treffprosenten mellom pikslene i malbildet og referansebildet. Den kalkulerer den kvadratiske differansen mellom malbildets og referansebildets piksler. Vi benytter oss også av metoden *TM_CCOEFF_NORMED*[56].
Kvadratbredden t bestemmes av størrelsen på malbildet. R er resultat, T er malbildets piksler, I er referansebildets piksler, x og y er start x- og y-koordinat i referansebildet. Desto mer likt malen er området i bildet, desto lavere vil differansen beregnet i likning 6 være.

$$R(x,y) = \sum_{i=x}^{x_t} \sum_{j=y}^{y_t} (T(i,j) - I(x+i, y+j))^2 \tag{6}$$

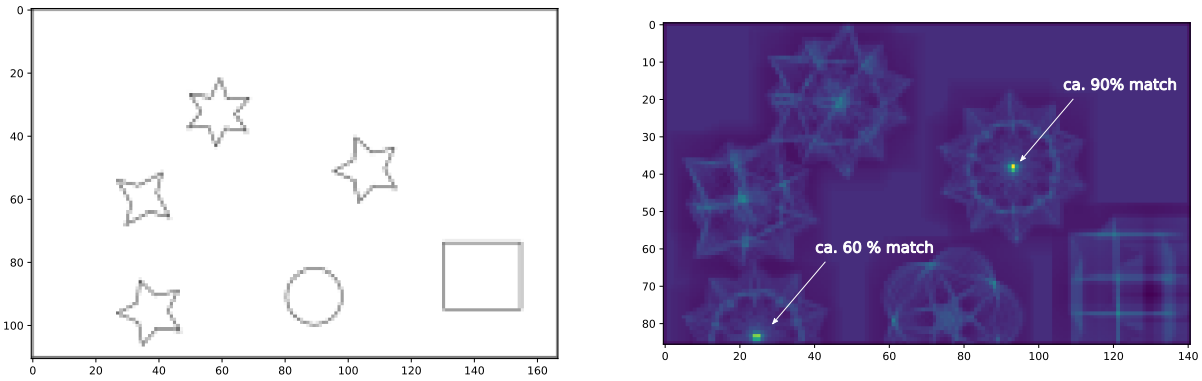
Resultatet blir forskjellen på pikselens intensitet i malbildet og referansebildet.

Kodesnutt 22 viser et minimalistisk eksempel på bruk av malsammenligningsfunksjonen. Funksjonen bryr seg ikke om farger, bare form. Den kan faktisk ikke bruke et fargebilde, kun gråskala.

```
1  import cv2, numpy as np
2
3  img = cv2.imread('test.jpg')                          # Innlest bilde
4  tmpl = cv2.imread('template.jpg')                     # Del av bildet
5  imgray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)        # Gråskalabilde
6  tmplgray = cv2.cvtColor(tmpl, cv2.COLOR_BGR2GRAY) # Gråskalatemplate
7  # Under er selve malsammenligningsfunksjonen
8  match = cv2.matchTemplate(imgray, tmplgray, cv2.TM_CCOEFF_NORMED)
9  # "match" blir da en matrise med dimensjoner lik gråskalabildet,
10 # der hvert element sier
11 # hvor mye den delen av bildet ligner på templaten fra 0 til 1.
```

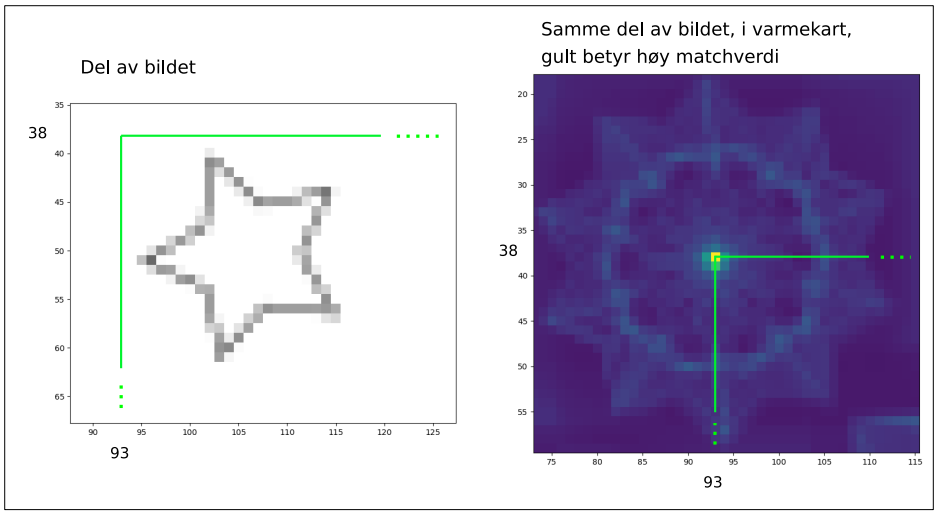
Kodesnutt 22: Eksempel på bruk av matchTemplate metoden.

Dersom vi kjører koden vist i kodesnutt 22 med bildet i figur 45 som original, og stjernen oppe i venstre hjørne klippet ut og brukt som mal, vil vi få resultatet vist i figur 48. Resultatet vises som et varmekart/intensitetskart. De mørke områdene representerer områder med dårlig treffprosent. De lyse områdene representerer områder med høy treffprosent.



Figur 48: Figur a) viser originalbildet. Bildet er likt det i figur 45, bortsett fra at en stjerne er klippet ut og brukt som mal. Figur b) viser resultatet som varmekart.

Figur 48 viser litt av svakheten til malsammenligning. Vi ser at selv om figurene har lik form, kan støy i bildet føre til at treffprosenten blir lavere enn forventet (nede til venstre i b)). Andre utfordringer ved å bruke malsammenligning er lysstyrke, blokkerende objekter, bakgrunnsstøy og endring i skalering. Figur 49 illustrere hvordan resultatet til en sammenlignet mal blir lagret i koordinaten/pikselen oppe til venstre.



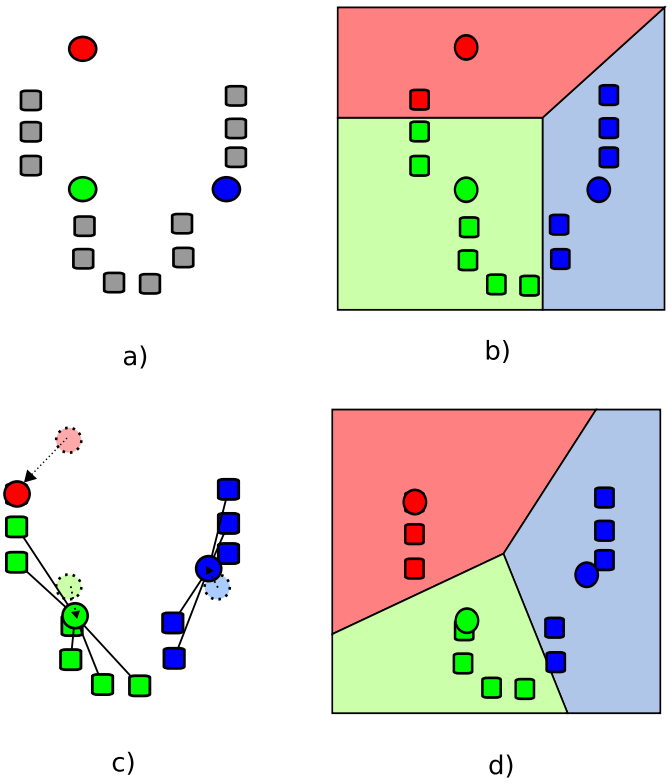
Figur 49: Figuren viser at malens treffprosent er lagres i pikselen lengst oppe til venstre.

Som beskrevet vil malsammenligning ta hensyn til alle pikselene i et bilde ved sammenlikning. Dette betyr at hvis vi har et malbilde med en bestemt bakgrunn, vil vi kun få høy treffprosent om bakgrunnen er lik i bildet som undersøkes. Derfor fjernes bakgrunnen når bildene når vi leter etter objeter. Bakgrunnen kan fjernes med metoder som for eksempel fargemaskering(se seksjon 3.1).

Malsammenligning benyttes kun i kapittelet 5, om helsen til korallrev. Når malsammenligning da brukes på korallrevet er det kun koralldelenes(malbildets) piksler som sammenliknes med korallrevets piksler. Bakgrunnen er fjernet. Dette gir et bedre resultat ved skiftende bakgrunn. For korallrevet samler vi en rekke maler for standardiserte "utvekster"/koralldeler, disse vil høy treffprosent på alle stedene de blir funnet igjen i bildet.

3.7 Gruppering

Gruppering brukes til å kategorisere punkter, og er mye brukt innenfor dataanalyse. Målet er å samle punkter og gruppere dem etter tilhørighet. På denne måten reduseres datagrunnlagets kompleksitet og en får mer håndterlige og tolkbare data. Metoden vi i hovedsak har benyttet oss av kalles *k-means*. Det er en egen klasse for dette i Scikit-learn biblioteket [74]. *K-means* algoritmen velger k antall tilfeldige punkter, og bruker dem som senter for gruppene. Punktene rundt vil bli med i den gruppen som representerer det nærmeste senteret. Hvert punkt i datasettet vil gruppere seg til det nærmeste k-punktet. Deretter vil k-punktene flytte seg til der avstanden mellom hvert punkt er gjennomsnittlig lavest. Figur 50 viser de generelle stegene som utføres i algoritmen.

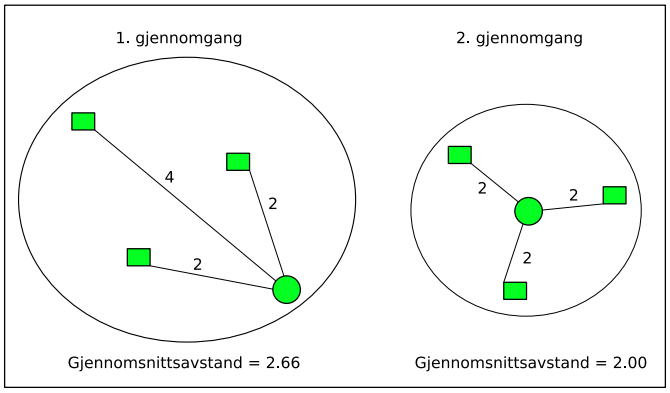


Figur 50: Fire figurer fra wikipedia ([9], [10], [11], [12]) som illustrer *k-means* algoritmen i aksjon.

Mer utdypende om figur 50.

a) Først velges k tilfeldige punkter som sentere (i dette tilfellet 3).
b) Punktene blir grupperte til nærmeste k-punkt.
c) K-punktene flyttes til senter av gjennomsnittsavstanden til punktene.
d) Punktene blir tilordnet nye grupper basert på nærmeste kvadratisk avstand til k-punkt.
Steg c) og d) gjentas til ønsket antall iterasjoner er nådd, eller til den gjennomsnittlige kvadratiske avstanden mellom punktene og hvert senter er en ønsket verdi.

En av betingelsene til hvor mange ganger koden kjører, er hvor stor avstand det er mellom punktene og senteret i gruppen. Figur 51 viser at algoritmen vil kjøre en gang til for å oppnå en veiet gjennomsnittsavstand på 2 piksler.



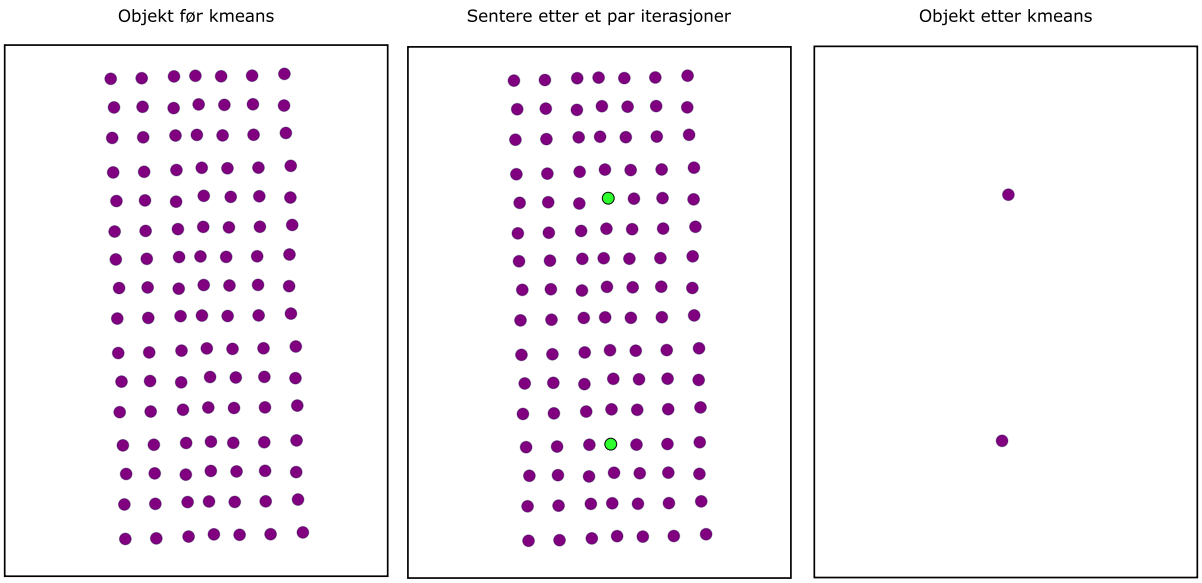
Figur 51: Viser hvordan senter kan flytte seg til den når 2 som gjennomsnittsavstand.

Kodesnutt 23 viser et eksempel på hvordan vi kan bruke *k-means* algoritmen fra Scikit-Learn biblioteket. Dette er en klasse (*KMeans*) hvor vi må definere en rekke parametre. Deretter kan vi benytte klassemetoder til å få ut senterpunkter.

```
from sklearn.cluster import KMeans

kmeans = KMeans(
    init="k-means++",  # hvor punktene skal plasseres
    n_clusters=nr_of_clusters,  # antall grupper
    n_init=10,  # hvor mange ganger den resetter algoritmen
    max_iter=10,  # hvor mange ganger den flytter sentrene
    random_state=13)  #
kmeans.fit(datapoints)  # Bruker kmeans på punkter
centers = kmeans.cluster_centers_  # Henter senter i hver gruppe
```

Kodesnutt 23: Eksempel på bruk av grupperingsmetoden k-means.

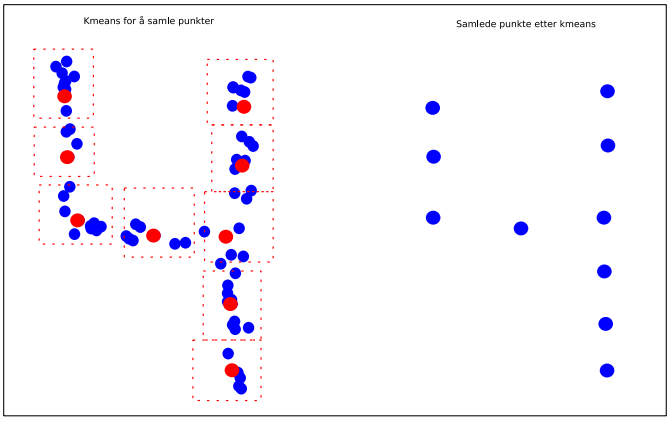
Et eksempel på bruk av *k-means* til vårt formål er vist i figur 52 og 53 . Det skal forestille en del av en gren i korallrevet.



Figur 52: Viser hvordan et objekt representert som punkter kan minskes ned til færre punkter.

På figur 52 er *KMeans* kjørt med to som antall grupper. Vi ser da at istedenfor å beskrive en form med hundre punkter, får vi redusert dette ned til to. På denne måten reduserer vi datagrunnlagets kompleksitet. Mindre kompleksitet betyr at det er lettere og hente ut den viktigste informasjonen. For korallrevets del blir det blant annet enklere å finne ut hvor forskjellige grener starter og/eller slutter.

Figur 53 viser litt mer oversiktlig hvordan en del av et korallrev deles inn i færre punkter. I et skikkelig bilde vil det være mange flere blå punkter.



Figur 53: Viser hvordan et objekt representert som punkter kan minskes ned til færre punkter. Til venstre er de røde prikkene sentrene. Til høyre er sentrene gjort om til de eneste punktene som representerer objektet.

3.8 Nærmeste nabo

Dette er en algoritme som finner nærmeste nabo til punkter. Vi benyttet oss igjen av Scikit-Learn biblioteket for denne algoritmen [75]. Kodesnutt 24 viser et eksempel på bruk i Python.

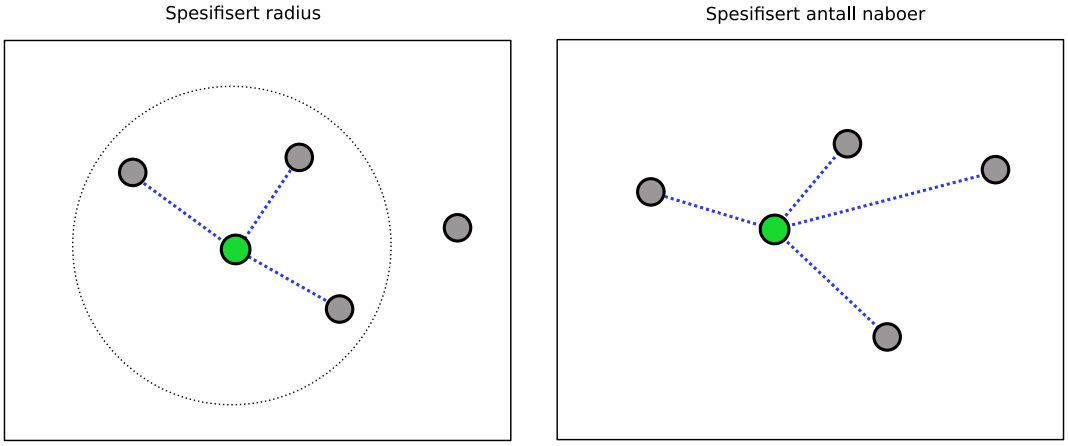
```
from sklearn.neighbors import NearestNeighbors

# lager klasse med parametere
nbrs = NearestNeighbors(n_neighbors=5, algorithm='kd_tree')
nbrs.fit(points)  # bruker klassens parametere på punkter
dist, ind = nbrs.kneighbors(points)
# dist og ind er to lister med lister.
```

Kodesnutt 24: Eksempel på bruk av nærmeste nabo.

Det som returneres av *kneighbors*-metoden er to lister. Disse listene inneholder igjen en liste for hvert enkelt punkt. Den første listen "dist" inneholder informasjon som beskriver avstanden til de nærmeste naboene. Variabelen "ind" inneholder tilsvarende oversikt, men her er innholdet ikke avstanden, men indeksnummeret til nabopikslene. Begynnelsen på ett slikt "ind" objekt er illustrert i likning 7. Nabo én er alltid punktet selv.

$$ind = \overbrace{[[\underbrace{0}_{\text{nabo 1}}, \underbrace{1}_{2}, \underbrace{2}_{3}],}^{\text{punkt 1}} \overbrace{[\underbrace{1}_{\text{nabo 1}}, \underbrace{4}_{2}, \underbrace{5}_{3}], ...]}^{\text{punkt 2}} \tag{7}$$

Med denne klassen kan vi óg bestemme om vi vil finne x antall nærmeste naboer, eller alle naboer innenfor en radius. Dette er vist i figur 54.

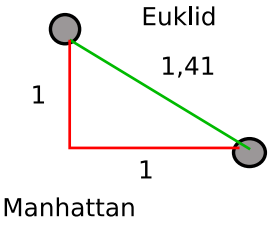


Figur 54: Andre metoder å finne naboer. Til høyre er det spesifisert 4 naboer.

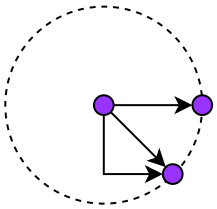
En annen parameter som kan spesifiseres er hvordan avstanden mellom et punkt og naboene beregnes. Vi har den standard euklidske avstanden, altså en rett strek direkte til naboen. Det finnes en annen metode kalt Manhattan-avstand eller drosjedistanse. For drosjedistanse adderes avstanden i vertikal- og horisontalretning. Figur 55a illustrerer dette. Utregningen til både Manhattan/drosje- og euklidsk avstand er vist i likning 8.

Vi har to punkter $P_1 = (x_1, y_1)$ og $P_2 = (x_2, y_2)$. D er avstanden mellom punktene.

$$\begin{aligned} \text{Euklid: } D &= \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \\ \text{Manhattan: } D &= |x_1 - x_2| + |y_1 - y_2| \end{aligned} \tag{8}$$



(a) Måter å regne avstand til naboer.



(b) Lengre avstand med Manhattan.

Figur 55

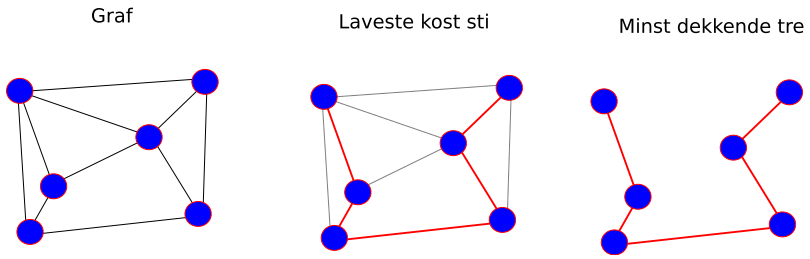
En grunn til å bruke Manhattan-avstand er at den gir lengre avstand mellom punkter som ligger på hvert sitt horisontale og vertikale plan. Dette vises i figur 55b. Når vi bruker denne metoden i kapittel 5, ønsker vi å prioritere forbindelse mellom punkter på samme vertikale/horisontal plan. Da er Manhattan-avstand metoden å velge.

Vi bruker nærmeste nabo metodene for å spesifisere vekt på linjene mellom punktene som blir funnet av *k-means*. I våre løsninger vil lav vekt tilsvare lav Manhattan distanse mellom punkter.

En utfording med nærmeste nabo er at det fort blir kaotisk med mange naboer. For å håndtere dette bruker grafteori som er omtalt i kapittel 3.9. Med grafteori vil programmet finne alle de korteste veiene mellom hvert punkt.

3.9 Grafteori

Grafteori handler om hvordan et program kan binde sammen punkter. En graf består av noder (engelsk: "verticies"), eller punkter, og kanter som binder sammen nodene. En kant kan bare binde sammen to noder, men hver node kan ha mange kanter. "Nærmeste nabo" brukes gjerne til å definere vekten/avstanden på kantene. Vi benyttet oss av en sjanger innen grafteori som kalles minst dekkende tre "Minimum Spanning Tree" (MST) Denne brukes til å finne punktene nærmest hverandre og binde dem sammen med kanter. Figur 56 viser et eksempel på hvordan algoritmen går frem for å finne MST.



Figur 56: "Grafen" til venstre illustrerer en rekke noder/punkter og kanter/streker. På figuren i midten markeres den korteste/billigste veien som binder alle nodene sammen til ett tre tre. I figuren til høyre har metoden blitt kjørt og vi har fjernet alle irrelevante kanter.

Målet er altså å velge den veien som har totalt lavest avstand mellom hvert punkt. For å oppnå dette velger algoritmen den avstanden som er kortest mellom hver node som ikke allerede er en del av treet.

For å finne det minst dekkende treet bruker vi Boruvkas algoritme. Den Kalles også for Sollins algoritme. Den blir implementert etter eksempel vist i denne artikkelen [79].

Algoritmen tar inn en gruppe noder. Først defineres alle nodene til å være trær. Deretter bindes hvert tre sammen med det nærmeste treet. På denne måten blir det totale antall trær redusert til halvparten for hver repetisjon. Etter noen repetisjoner er algoritmen ferdig, og står igjen med ett tre. Dette treet er det minst dekkende treet, MST.

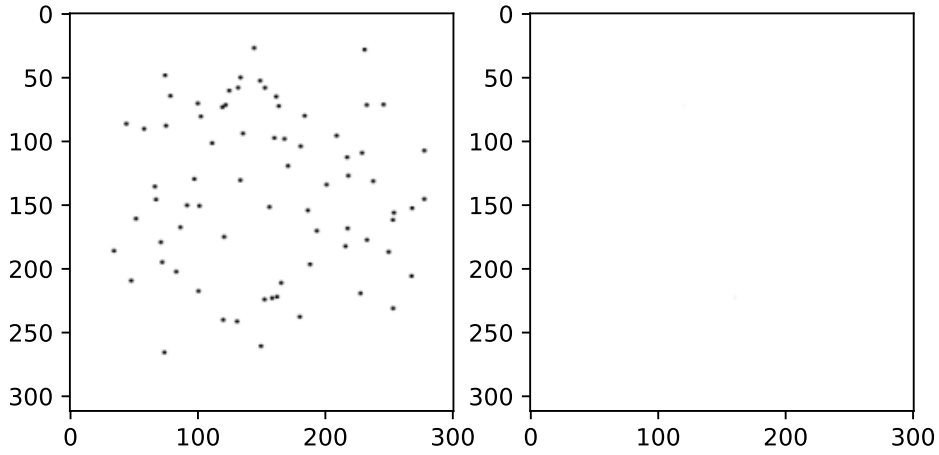
Andre alternative algoritmer er Kruskals [88] og Primms [66].

3.10 Annet:

3.10.1 medianblur() [49]

Denne funksjonen bruks oftest til å filtrere bort støy. Men den kan og brukes til å få jevnere kanter i et bilde. Funksjonen beregner gjennomsnittet av et piksel og naboene til dette pikselen. Dette gjennomsnittet blir så den nye verdien til det pikselen vi arbeidet med. Hvor nære naboer som tas med i denne beregningen avgjør styrken til filteret. Vi angir da størrelsen på en todimensjonal matrise som igjen angir hvilke av naboene til senterpikselen som skal regnes med (Likner strukturelementet vi beskrev under morfologi 3.5).

Funksjonen tar inn ett bilde og en størrelse på filtreringsmatrisen (må være oddetall). I retur sendes det ferdig filtrerte bildet. Det returnerte bildet vil ha blitt mer uklart. Kanter og skiller vil ha blitt mindre skarpe og småstrukturer helt utjevnet. Figur 57 viser hvordan *medianblur* virker på et bilde med prikker som støy.



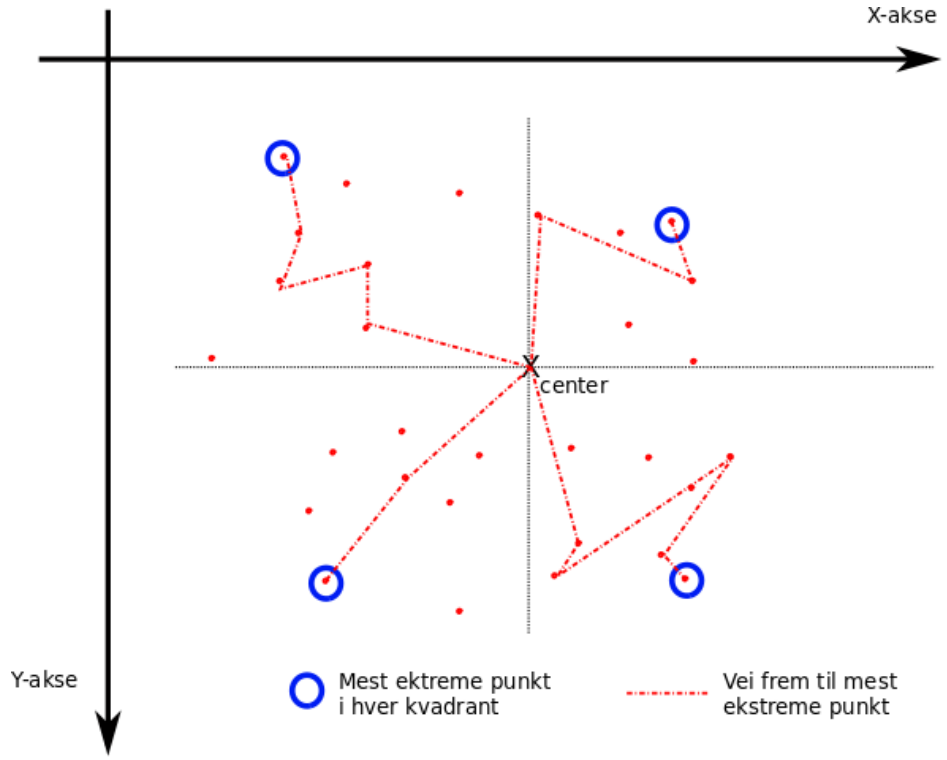
Figur 57: Resultat av *medianblur* funksjon.

```python
import cv2

im = cv2.imread('testbilde.jpg', 1)
blured = cv2.medianBlur(im, 7)  #
```

Kodesnutt 25: Eksempelkode for *medianblur*.

3.10.2 find_corner_points()

Funksjonen tar inn en liste med koordinater. Den tar også inn en valgfri senterverdi. Hvis denne ikke oppgis regnes senter ut ved hjelp av minAreaRect (se seksjon 3.3.5). Funksjonen går så gjennom alle punktene og returnerer de fire mest ekstreme hjørnepunktene. Funksjonen brukes til å finne hjørnepunkter, sortere dem i en slik rekkefølge at vi er klare til å bruke perspektiv transformasjon(seksjon 3.4.1). Hvis det ikke eksisterer noen punkter i en kvadrant vil senterpunktet bli gjeldende som den mest ekstreme verdien for denne kvadranten.



Figur 58

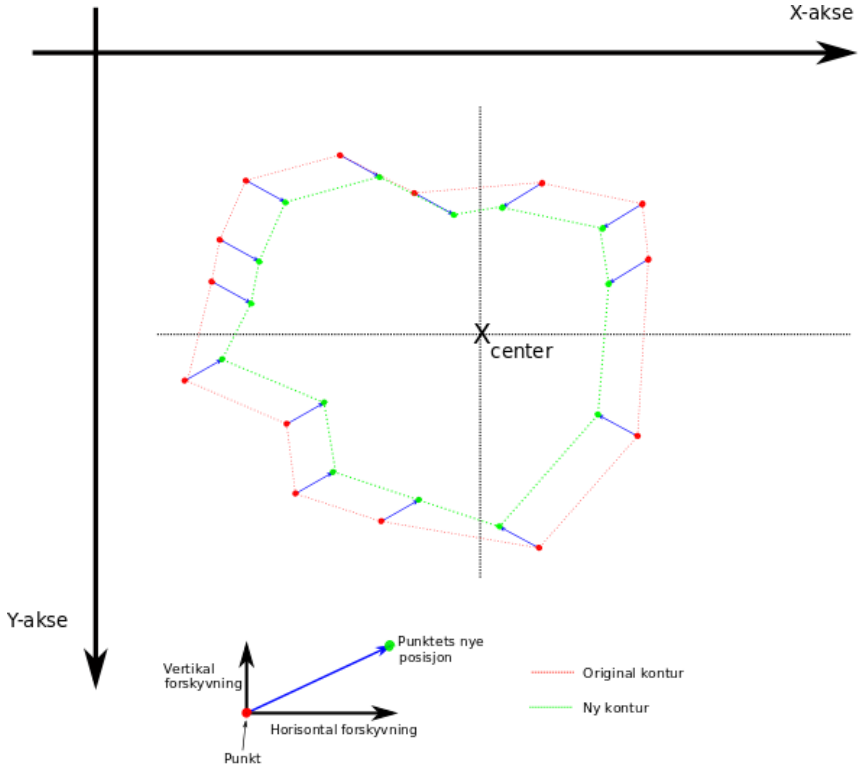
```
def find_corner_points(points, center=0, thresh=5):
    if center == 0:
        rect = cv2.minAreaRect(points)
        center = rect[0]
    th = thresh
    left_high = center
    left_low = center
    right_high = center
    right_low = center

    for p in points:
        if p[0] < center[0] and p[1] < center[1] and ((left_high[0] - p[0]) +
                                                     (left_high[1] - p[1]) > th):
            left_high = [p[0], p[1]]
        if p[0] < center[0] and p[1] > center[1] and ((left_low[0] - p[0]) +
                                                     (p[1] - left_low[1]) > th):
            left_low = [p[0], p[1]]
        if p[0] > center[0] and p[1] < center[1] and ((right_high[1] - p[1]) +
                                                     (p[0] - right_high[0]) > th):
            right_high = [p[0], p[1]]
        if p[0] > center[0] and p[1] > center[1] and ((p[0] - right_low[0]) +
                                                     (p[1] - right_low[1]) > th):
            right_low = [p[0], p[1]]

```

```
25      new_points = np.array([left_high, left_low,
26                             right_low, right_high], dtype = np.float32)
27      return new_points
```

Kodesnutt 26: Koden til find_corner_points funksjonen.

3.10.3 shrink_contour()

Denne funksjonen kan brukes til å trekke alle punktene i en kontur mot fra konturens senter, eller bort fra konturens senter. Hvor stor sammentrekningen skal være bestemmes av to parameter. En for horisontalretning og en for vertikalretning. Disse angir sammentrekningen i antall piksler. Er inngangsverdiene negative utvides konturen. Da forskyves punktene bort fra senter. Senter blir funnet ved hjelp av *moments*-funksjonen[50].



Figur 59

```
1   def shrink_contour(points, y_axis=0, x_axis=0):
2       # x-aksen avgjør forflytning horisontal.
3       # Y-aksen avgjør forflytning vertikalt.
4       # Positive verdier av x_axis/y_axis krymper konturen.
5       # Negative verdier utvider konturen.
6       # p[1]=y p[0]=x
7       moment = cv2.moments(points)
```

```
    center = [int(moment['m10'] / (moment['m00'] + 1e-6)),
                int(moment['m01'] / (moment['m00'] + 1e-6))]
    for p in points:
        if p[1] < center[1] and p[0] < center[0]:   #Upper Left
            p[1] += y_axis
            p[0] += x_axis
        elif p[1] > center[1] and p[0] > center[0]: #Lower Right
            p[1] -= y_axis
            p[0] -= x_axis
        elif p[1] < center[1] and p[0] > center[0]: # Upper Right
            p[1] += y_axis
            p[0] -= x_axis
        else:                                       # Lower left
            p[1] -= y_axis
            p[0] += x_axis
        np.float32(p[1])
        np.float32(p[0])
    return points
```

Kodesnutt 27: Koden til shrink_contour funksjonen

3.10.4 mean_HSV_color_of_masked_area

Dette er en spesiallaget funksjon som beregnes gjennomsnittsfargen til et maskert området. Funksjonen tar inn ett bilde i HSV-format, og en maskering. Maskeringen angir pikslene vi er interessert i med hvit farge. Funksjonen regner fargetone verdien i alle pikslene om fra 0-179, til 0-360. Denne nye verdien for fargetoner ser vi på som punkter i enhetssirkelen. Videre regner vi fargetoneverdiene om til kartesiske koordinater. Så regnes den gjennomsnittlige vinkelen ut med tangens invers. Til slutt halveres denne gjennomsnittlige vinkelen slik at verdiområdet igjen blir 0-179. Koden til funksjonen tatt med i kodesnutt 28. Funksjonen er spesiallagt til fotomosaikk oppgave i seksjon 6.

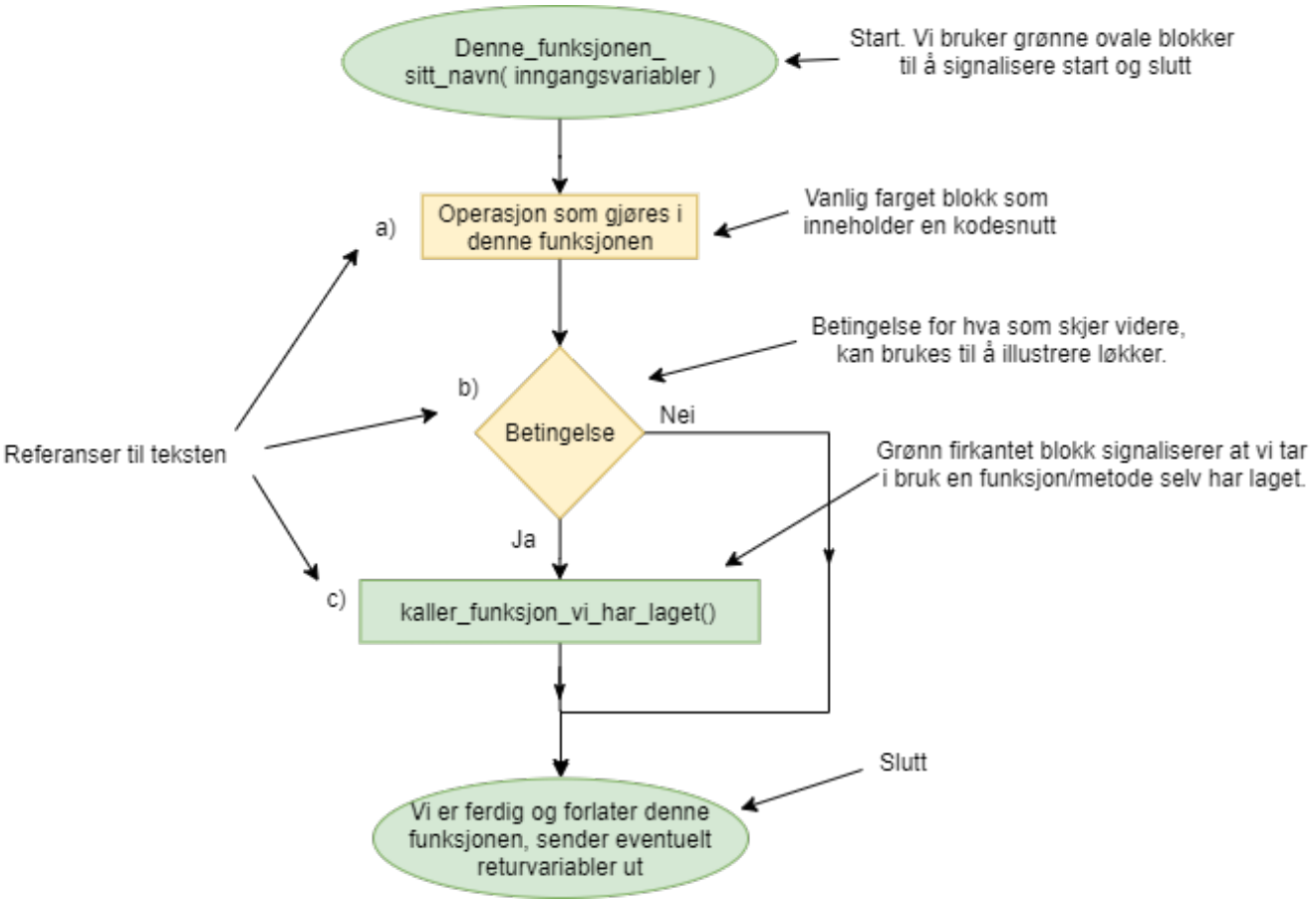
```
def mean_HSV_color_of_masked_area(img, mask):
    import numpy as np
    pks = []
    hgt, wdt = mask.shape
    # Gjør om hver fiksels farge til HSVs hue verdi fra 0-360
    # Deretter beregne vi denne vinkelen om til det kartesiske kordinater.

    for i in range(hgt - 1):  # X koordinat
        for j in range(wdt - 1):  # Y koordinat
            if mask[i][j] == 255:
                pks.append(img[i][j][0])
    X = sum(np.cos(np.deg2rad(2 * pks)))/len(pks)
    Y = sum(np.sin(np.deg2rad(2 * pks)))/len(pks)
```

```
    mean = np.rad2deg(np.arctan2(Y, X))  # Gjennomsnittlig vinkel i grader

    # Til slutt returneres vinkelen i OpenCV HSV, hue format, 0-179
    return mean / 2
```

Kodesnutt 28: Koden til mean_HSV_color_of_masked_area

3.10.5 Bruk av flytskjema:

Vi har i denne rapporten brukt en rekke flytskjema til å forklare løsningene våre. I figur 60 har vi et eksempel på et slikt flytskjema som beskriver hva de forskjellige boksene og fargene betyr Disse blokkskjemaene reflektere hvordan den bakenforliggende koden fungere. De er derimot ikke direkte kopier av koden, men forenklinger som skal illustrere funksjonen. For eksempel så er ikke håndtering og rapportering av feil under gjennomføringen ikke tatt med i skjemane.



Figur 60: Dette er et eksempel på et flytskjema. Teksten i bildet forklarer hva de forskjellige boksene og fargen betyr.

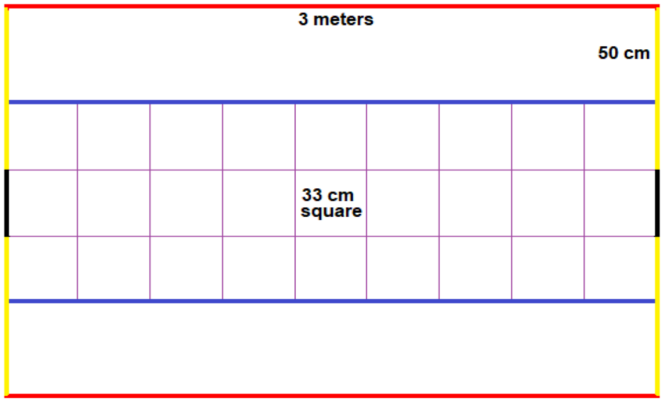
4 Havbunn

4.1 Oppgaven

Som beskrevet i innledningen (seksjon 1.5.3) inneholder årets MATE ROV-konkurranse en oppgave med autonom/automatisk kjøring samt kartlegging av havbunnen. Oppgaven har naturlig blitt delt opp i to deloppgaver. Den ene deloppgaven innebærer å kjøre ROV-en i en rett linje over havbunnen. Den andre deloppgaven innebærer å kartlegge objekter på havbunnen i området som ble passert i deloppgave 1. Med havbunn sikter vi til en spesiallaget modell i lakkerte PVC-rør som MATE har beskrevet i sin oppgabeskrivelse [15]. Modellen er skissert i figur 61.

I MATE ROV-oppgavebeskrivelsen blir det spesifisert at deltagerne kan velge mellom manuelle eller autonome/automatiske løsninger. Velges de manuelle løsningene får man færre poeng enn om man klarer de automatiske. Vi har derfor valgt å lage løsninger som utfører disse oppgavene på en autonom/automatisk måte i henhold til oppgavebeskrivelsen.

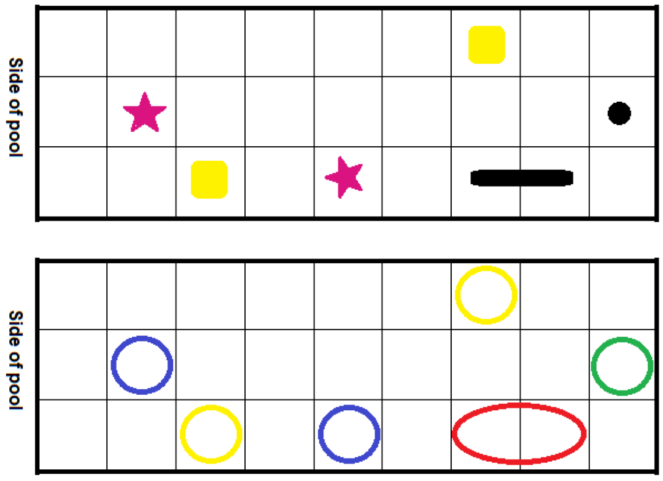
Deloppgave 1:
Undervannsfartøyet skal starte sentrert over den svarte streken (figur 61) i en av traseens ender. Turen regnes som ferdig når den svarte streken på den andre siden av traseen er passert. Under kjøring skal de blå linjene alltid være synlige, mens de røde linjene ikke skal vise i det hele tatt. Dette betyr at ROV-en må holde en konstant høyde og retning gjennom hele forsøket. Mens oppgaven gjennomføres skal det vises sanntidsbilder fra et kamera som er rettet direkte ned mot havbunnen. Dommeren skal slik bekrefte en vellykket gjennomføring.



Figur 61: Bildet viser området som ROV-en skal traversere. ROV-en skal gå fra sort strek i den ene enden til sort strek i den andre enden. Samtidig må den holde en stabil høyde og retning gjennom hele gjennomkjøringen. De røde rørene kan ikke være synlig på sanntidsbildene fra ROV-en. Bilde hentet fra referanse [15].

Deloppgave 2:
Målet med denne oppgaven er å oppdage og klassifisere seks objekter på havbunnen. Deltagerne skal lage et program som automatisk gjør dette. Programmet skal så produsere et kart lignende det nederste bildet i figur 62. Det øverste bildet i figur 62 viser objektene som skal identifiseres. Det nederste bildet viser hvordan objektene skal markeres. Blå ring for sjøstjerner, gul ring for områder

der nye korallrev kan plantes, rød ring for korallrev, grønn ring for svamp. Det må også markeres hvilken kortside som er nærmest bassengkanten. Det er valgfritt om dette kartet over havbunnen produseres når vi kjører linjen i deloppgave 1, eller fra et oversiktsbilde hvor vi manuelt kjører ROV-en i rett posisjon. Det er ingen poengforskjell på disse løsningene.

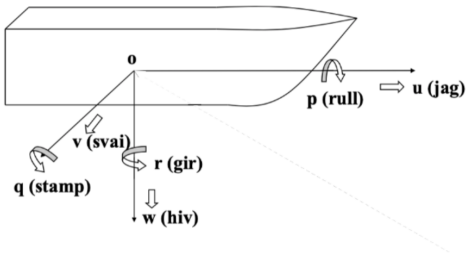


Figur 62: Det øverste bildet illustrerer havbunnen med objektene som skal kartlegges. Nederst i bildet har vi ett kart hvor de forskjellige objektene har blitt markert. De blå ringene markerer sjøstjernene. Sjøstjernene er de lilla (på testbanen er de røde) stjernene. De gule ringene markerer planteplasser for korallrev. Planteplassene likner gule bokser. Den røde ellipsen markerer et område med korallrev. Korallrevet er illustrert med et svart rektangel som strekker seg over to ruter. Den grønne ringen markerer en rute som inneholder svamp. Svampen er illustrert med en sort ring. Kortsiden nærmest bassengveggen skal markeres med "Side of pool". Bilde er hentet fra referanse [15].

Vår løsning:
Vi har valgt å dele oppgaven opp i to delløsninger. Det blir et program til å kartlegge havbunnen fra et oversiktsbilde, og et program til å kjøre autonomt over havbunnen. Dette gjør løsningene mer uavhengige og robuste. Skulle den ene løsningen feile kan vi fortsatt klare den andre oppgaven. Ved å dele opp oppgaven styrker vi robustheten til gjennomføringen vår i MATE-konkurransen.

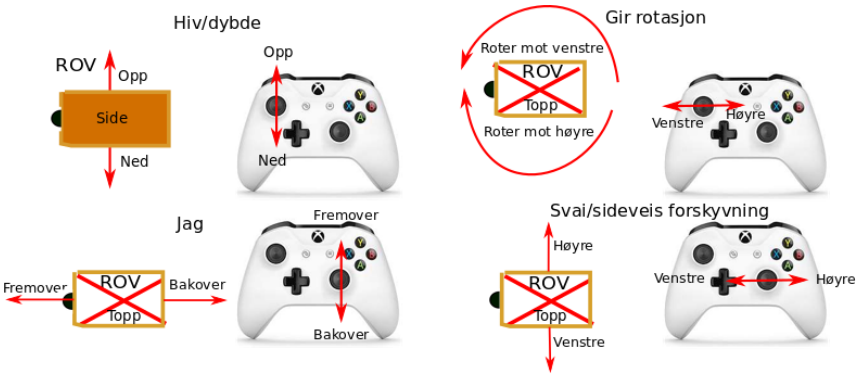
4.2 Autonom kjøring

Ved manuell kjøring vil ROV-en bli styrt fra en Xbox-kontroller. Intern regulering av dybde, rull- og stampvinkel (se figur 63a) gjør at vi enkelt kan styre ROV-en med Xbox-kontrolleren. Dette blir illustrert i figur 63b. Når autonom/automatisk kjøring blir aktivert, tar programmet kontrollen over variablene som tidligere ble styrt fra Xbox-kontrolleren. Programmet vil bli tatt i bruk fra brukergrensesnittet, som er omtalt i kapittel 7.



(a) Bevegelses- og posisjonsvariabler for et sjøgående fartøy. Vi benytter oss av terminologien definert av bildet. Selve bildet er hentet fra [86].

.



(b) Styring av ROV med Xbox kontroller.

Figur 63

Løsningen legger til grunn noen forutsetninger:

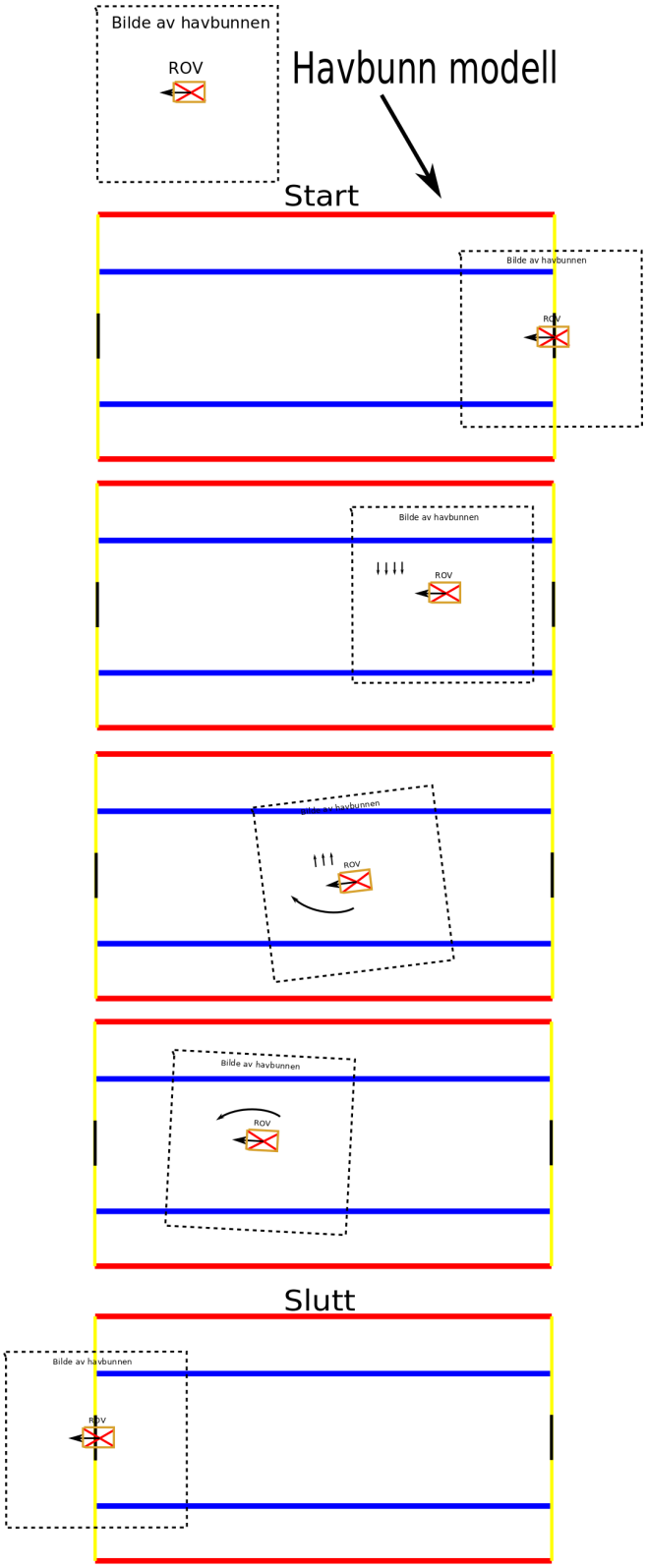
- Intern regulering av dybde samt stamp og rull vinklene.
- ROV-en kan styres fra Xbox-kontrolleren slik det blir illustrert i figur 63b. Det som er viktig er at pådrag på gir eller svai ikke påvirker hverandre i nevneverdig grad.
- Programmet må først startes når ROV-en har blitt plassert i en korrekt posisjon i forhold til banen som skal traverseres.
- ROV-en beholder dybden som den hadde ved programmets start.
- Kameraet plassert under ROV-en har fri utsikt ned mot havbunnen.
- Rull-, stamp- og dybderegulatorene fungerer slik at ROV-en ligger stabilt i vannet.
- Hastighet på bildeanalyse samt sending av nye kommandoer til ROV-en må være tilfredstillende.

Videre i teksten kommer det en overordnet beskrivelse, etterfulgt at av en mer detaljert del om bildebehandling, testing og regulering. Til slutt oppsummeres resultatene for regulerings- og bildebehandlingsdelen

4.2.1 Overordnet løsning

ROV-en skal kjøre selstendig over havbunnmodellen. Dette vil med vår løsning foregå på denne måten:

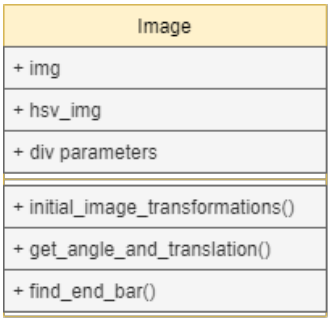
- ROV-en starter sentrert over det sorte feltet på den ene kortsiden.
- Dybden reguleres med en regulator nede på selve ROV-en. Denne regulatoren benytter en avstandsmåler (mot havbunn) og trykksensor (mot overflaten) til å låse dybden.
- Variabelen som bestemmer fremdriften i ROV-ens frontalretning (jag), settes slik at ROV-en kjører frem mot den andre siden av banen. Dette vil gi ROV-en en konstant hastighet mens den kjører gjennom banen. Denne hastigheten kan bli justert etter hva operatøren ønsker, men et forslag er 0.1-0.2 [m/s]
- Videre benyttes to PID-regulatorer til å regulere rotasjonen(girvinkelen) og sideveis forskyvning(svai).
- Regulatorene henter sine sensordata fra en bildebehandlingsalgoritme. Algoritmen tar utgangspunkt i de to blå rørene. Fra disse beregnes ROV-ens girvinkel og svaiforskyvning. Svaiforskyvning måles som ROV-ens avvik fra en posisjon midt mellom de blå rørene, og girvinkelen som avviket mellom ROV-ens girvinkel og banens lengderetning.
- Etter et gitt tidsintervall begynner algoritmen også å kikke etter den gule kortsiden. Idet ROV-en passerer over denne kortsiden er programmet fullført og kontrollen gis tilbake til operatøren.



Figur 64

4.2.2 Detaljert om løsningen

Løsningen består av en rekke komponenter. For bildebehandlingsdelen er det laget en egen klasse, *Image*, til å håndtere bildene av havbunnen. Denne brukes til å beregne ROV-ens girvinkel og svaiforskyvning, samt finne sluttstreken. I figur 65 er *Image* skissert på formatet til et klassediagram. Videre har vi laget et program som benytter seg av denne klassen til å utføre automatisk kjøring av ROV-en. Flytskjemaet i figur 66 beskriver dette programmet. Skjemaet er ikke en helt korrekt gjengivelse av hvordan automatisk kjøring har blitt implementert i styringsprogrammet, men prinsippet er det samme. Videre i dette underkapittelet går vi gjennom de viktigste delene av *Image*, og *automatisk_kjoring*



Figur 65: *Image* klassen som er konstruert for å hente ut data til automatisk kjøring.

automatisk_kjoring:

a) Setter opp to PID-regulatorer. Koden til regulatorene er hentet fra pakken Simple PID [32]. Mer om dette i seksjon 4.2.8.

b) Deretter initialiseres kameraet. Til dette brukes OpenCV sin VideoCapture[61] funksjon.

For hver runde i regulatorsløyfen utføres det en rekke operasjoner:

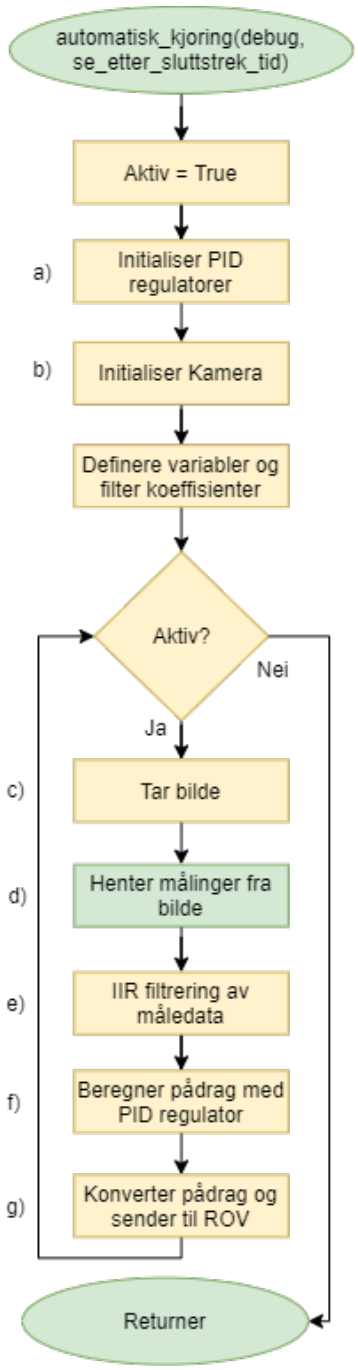
c) Først hentes det et bilde med VideoCapture objektets "read" metode[61].

d) Så beregnes det avvik på svaiforskyvning og girvinkel. Dette omtales nærmere i de påfølgende flytdiagrammene.

e) Målingene fra bildet filtreres med et IIR-filter. Dette omtales nærmere i seksjon 4.2.8.

f) Deretter beregner regulatorene nytt pådrag. Mer om dette i seksjon 4.2.8

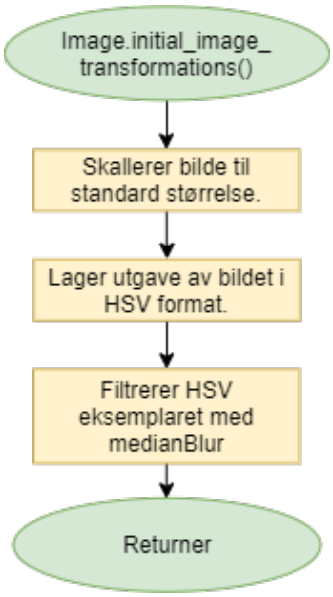
g) Pådraget konverteres til rett format, før det sendes til ROV-en.



Figur 66: Automatisk kjøring hovedprogram.

Initial_image_transformations:

- Først skaleres bildene ned til en standardisert størrelse. Til dette benyttes OpenCV sin resize funksjon (se seksjon 3.4.4).
- Deretter transformeres bildeformatet om til HSV ved hjelp av cvtColor(se seksjon 3.1.1).
- Til slutt filtreres HSV-bildet med medianBlur (se seksjon 3.10.1). Denne Filtreringen er en av de mest tidkrevende delene av programmet. Dette gjøres for å forbedre resultatet av fargemaskeringen i påfølgende metode.



Figur 67

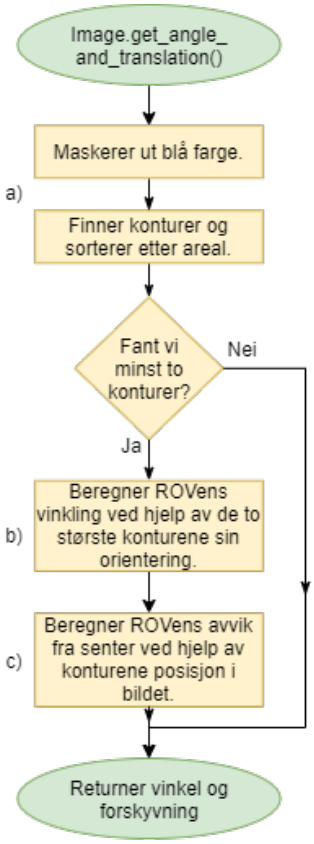
get_angle_and_translation:

a) De blå fargene maskeres ut ved hjelp av fargemaskering (se seksjon 3.1). Deretter plukkes de blå rørene ut med *findContours* (se seksjon 3) og *contourArea* (se seksjon 3.3.3).

b) Gitt at det ble funnet to store konturer(de to rørene), beregner vi rørenes vinkel på bilde. Vinkelen finnes ved å bruke *fitLine* (se seksjon 3.3.7) og tangens invers. Vinkelen som returneres er et gjennomsnitt av de to konturenes vinkler. I seksjon 4.2.4 beskrives utregningen i detalj.

c) Gitt at de to blå rørene ble funnet, beregnes ROV-ens svai (sideveis forskyvning). Med utgangspunkt i konturenes senter, kjent bildestørrelse og den kjente avstanden mellom rørene(1 meter), fastslås ROV-ens avvik fra senter i meter. Konturenes senter finnes ved å bruke *minAreaRect* som er nærmere beskrevet i seksjon 3.3.5. I seksjon 4.2.5 beskrives utregningen i detalj.

   Finner vi ikke de nødvendige konturene returneres det avvik lik null.



Figur 68

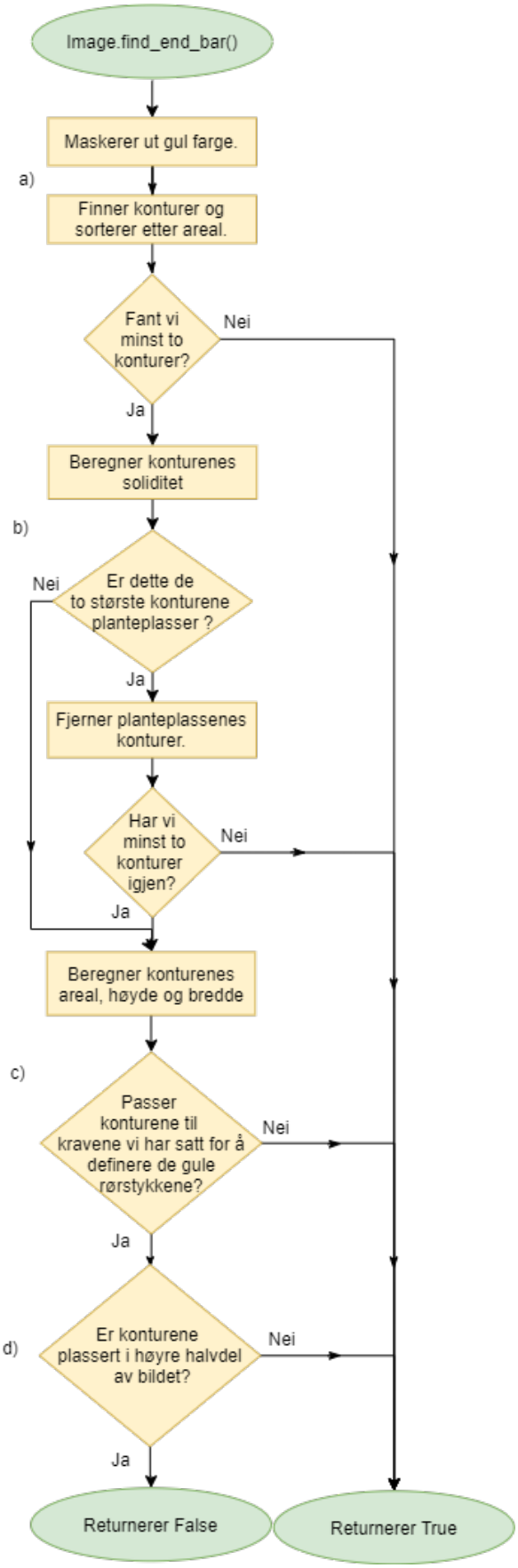
find_end_bar:

a) De gule fargene maskeres ut ved hjelp av fargemaskering (se seksjon3.1). Deretter plukkes de gule rørdelene ut med *findContours* (se seksjon 3) og *contourArea* (se seksjon 3.3.3).

b) Hvis vi har mer enn to konturer kikker vi nærmere på disse. Aller først beregnes konturenes soliditet(seksjon 3.3.12) og det omkransende rektangelet (seksjon 3.3.4). Har konturene egenskaper som tilsier at vi har funnet en planteplass, slettes disse fra listen. Et eksempel på dette kan ses i figur 73.

c) Så beregnes noen flere egenskaper for konturene. *BoundingRect* (se seksjon 3.3.4) anvendes til å finne konturenes lengder og bredder, og *contourArea* (seksjon 3.3.3) til å finne arealet. Ved hjelp av prøving mot testbilder har det blitt definert grenser for lengde, bredde og areal. Oppfyller de to største konturene disse kravene har vi funnet rørdelene som markerer slutt/startstreken.

d) Til slutt sjekkes det om rørdelene er plassert i den øvre eller nedre halvdelen av bildet. På denne måten bekrefter vi om ROV-en har passert sluttstreken. Om dette er tilfellet vil programmet for automatisk kjøring avsluttes og kontrollen gis tilbake til operatøren.



Figur 69

4.2.3 Bildet og posisjon

For å definere høyden ROV-en bør kjøre i, har vi kikket nærmere på kameraet[84]. Dette kameraet har en oppløsning på 1920x1080 piksler, og ved bruk av linsen[85], en justerbar synsvinkel mellom 28° og 90°. På et generelt grunnlag ønsker vi at ROV-en skal se mest mulig, og dermed bruke en størst mulig synsvinkel.
På vår egen testbane er det høyeste objektet korallrevet. Dette er 1.1 meter høyt. Vi kan ikke kollidere med korallrevet og setter derfor minimumshøyden til 1.2 meter. MATE sitt korallrev er en god del mindre på grunn av andre rørdimensjoner. I dette tilfellet vil vi måtte ta utgangspunkt i en annen minimumshøyde.
I figur 70b er et optimalt bilde av havbunnen illustrert. Dette eksemplet veier ønsket om stor avstand til de røde rørene opp mot ønsket om at begge de blå rørene skal være i bildet. Begge tilfeller vil føre til en ugyldig gjennomføring. Derfor er grensen plassert midt mellom de røde og blå rørene.
Med minimumshøyden og ønsket bildebredde, samt kravene til et vellykket forsøk i bakhand, finner vi kameraets optimale synsvinkel. Utregningen er utført i likning 9. I figur 70a illustreres ROV-ens optimale posisjon i forhold til havbunnen.

$$tangens(\theta) = \frac{motstaende\ side}{hosliggende\ side}$$

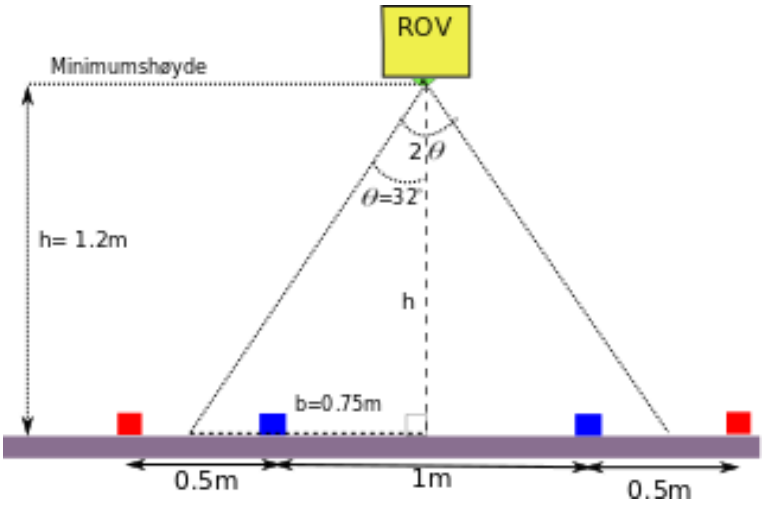
b – Havbunnbildets bredde delt på to.

h – ROV-ens høyde over havbunn.

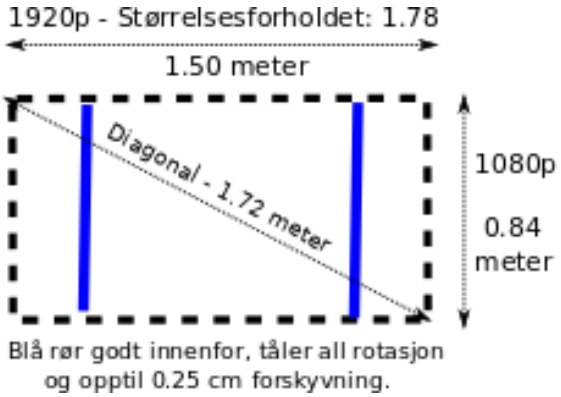
θ – Kameraets synsvinkel delt på to.

$$tangens^{-1}(\frac{b}{h}) = tangens^{-1}(\frac{(1.5/2)}{1.2}) = \underline{\underline{32°}} \tag{9}$$

Fra likning 9 ser vi at kameralinsen må settes til 64° åpning. Den optimale gjennomkjøringshøyden blir da minimumshøyden på 1.2 meter.



(a) ROV plassert i optimal posisjon for automatisk kjøring hvor kameraets synsvinkel er illustrert.

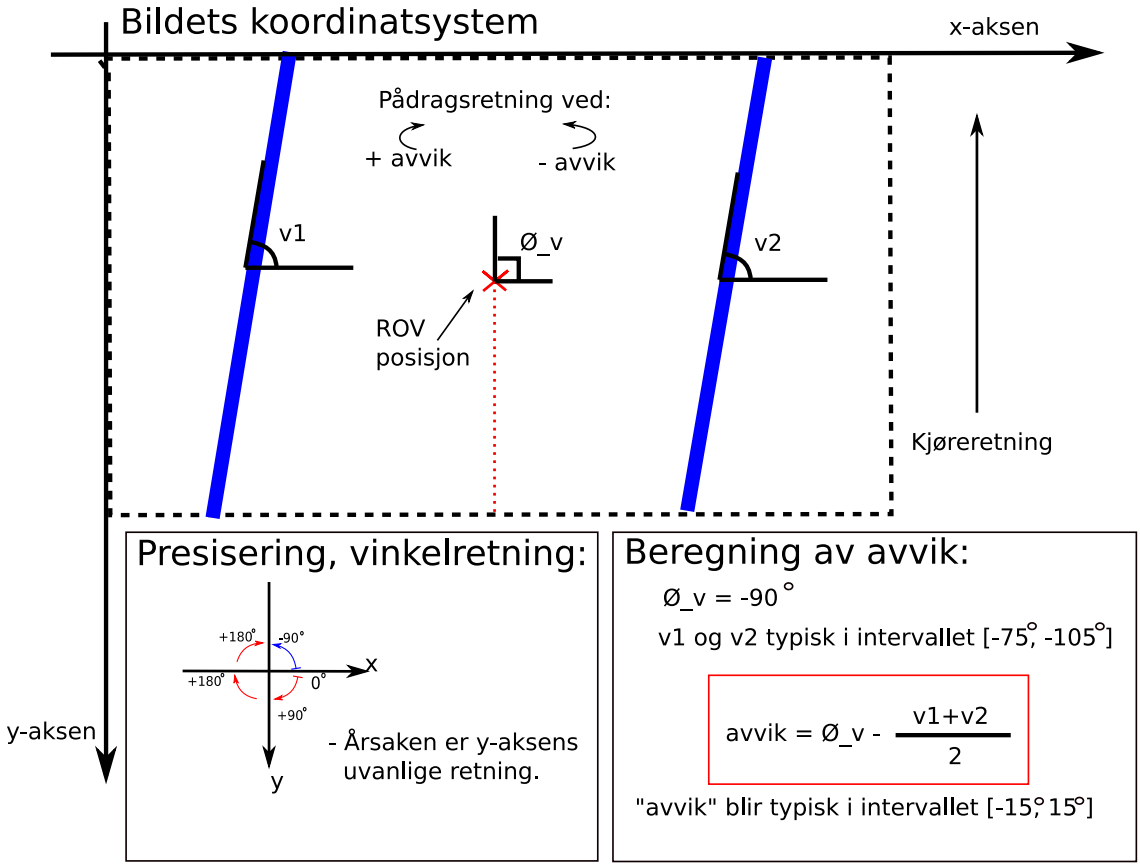


(b) Eksempel på et optimalt bilde av havbunnen ved automatisk kjøring. Det er her 0.25 meter til de røde rørene fra bildets kortsider.

Figur 70

4.2.4 Beregning av girvinkel

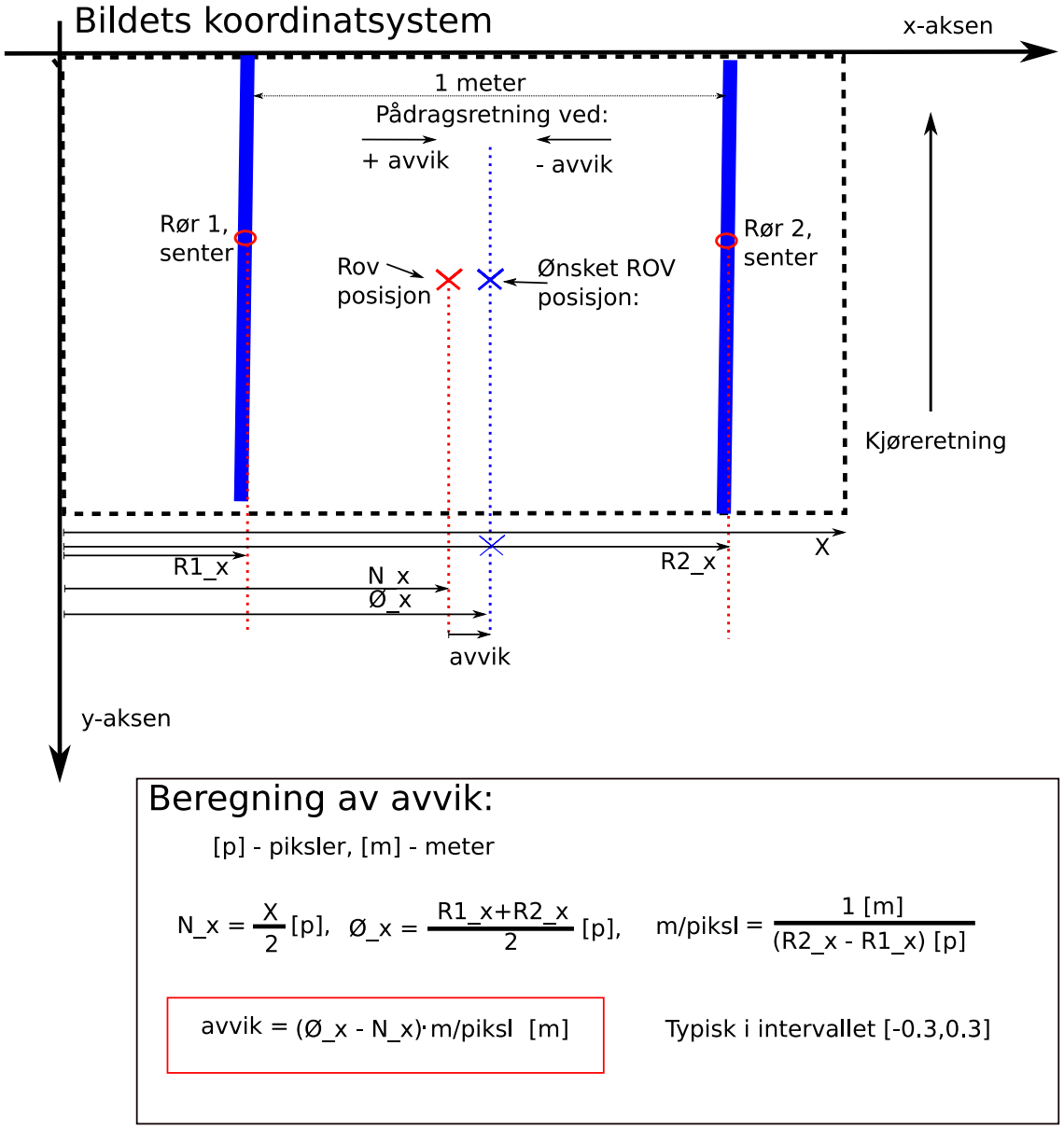
Figur 71 beskriver hvordan ROV-ens girvinkel beregnes ut ifra de blå rørene på havbunnen. I boksen om beregning av avvik fastslår vi hva slags utslag vi forventer å kunne finne.



Figur 71: Bildet beskriver hvordan ROV-en bestemmer girvinkelen i forhold til de blå rørene. Øverst har vi en grafisk illustrasjon av målingene. Nederst til venstre presiseres vinkelretning og aksesystem. Nederst til høyre kan vi se beregningen av vinkelavviket.

4.2.5 Beregning av svaiforskyvning

Figur 71 beskriver hvordan ROV-ens forskyvning, i svai retning, beregnes ved hjelp av de blå rørene på havbunnen. Helt nederst i figuren finner vi intervallet vi forventer at avviket holder seg innenfor.



Figur 72: Bildet beskriver hvordan svaiforskyvning beregnes ut ifra de blå rørene på havbunnen. Øverst defineres variablene i beregningen ved hjelp av en grafisk illustrasjon. Nederst benyttes disse definisjonene til å beregne avviket i meter

4.2.6 Test av bildebehandlingsdelen

For å teste bildebehandlingsdelen av programmet har vi samlet inn mange testbilder. Fotografen har holdt kameraet i korrekt høyde. Han har så gått sakte gjennom testbanen slik som ROV-en skal kjøre. Gjennom hele turen tar han en mengde med bilder. Ikke bare gode bilder, men et utvalg av rotasjoner og forskyvninger. Testbildene simulerer tre gjennomkjøringer med forskjellig fokus. Den første er en gjenomkjøring, den andre et forsøk ned mest mulig forskyvning og den tredje med mest mulig rotasjon. Bildene ligger i vedlegget under ../Bildebehandling/Bilder/Auto_kjoring.
Forklaring til tabellene:

- Nr - Bildets nummer fra start til slutt. Bildene kommer i en naturlig rekkefølge lik den ved en faktisk gjennomkjøring.
- Rotasjon - Rapportert rotasjon i grader.
- Forskyvning - Rapportert forskyvning i meter.
- Tid - Programmets tidsforbruk i millisekunder. Dette er selvfølgelig avhengig av datamaskinen programmet blir kjørt på. Tiden er allikevel med for å gi en indikasjon på hvor lang tid det kan ta. Tiden blir sterkt påvirket av filtreringen med *medianBlur*(se seksjon 3.10.1). Vi unngår derfor å bruke et større filtreringselement enn en 5x5-matrise. Ved å bruke et større element enn dette ble programmets tidsforbruk firedoblet.
- Resultat - Er resultatet slik det skal være?

Potensielle svakheter:

- Utfordringer med å gjenkjenne gule rørstykker når bildet er kraftig rotert. Utslag i gjennomkjøring 3, nr 1,2,23 og 24. Dette anses ikke som et problem da denne slags rotasjon av ROV-en er usannsynlig.
- Når korallrevet skygger over de blå rørene, og bildet er kraftig rotert vil ikke rørene bli rett sammenkoblet. Ingen tilfeller oppdaget i testeksemplarene.

Tidsbruk:
Gjennomsnittstiden testprogrammet brukte per bilde var 13 ms.

Kommentar til testing av gjennomkjøring:
Testen simulerer tenkte gjennomkjøringer med ROV-en. Vi undersøker om det er samsvar mellom bilde og resultat. Det har ikke vært anledning til å kontrollere gir- og svaimålingene mot faktiske verdier fra bildetakningstidspunkt. Testene blir derfor omtrentlige mål på om resultatene samsvarer med bildene.

Testresultatene samsvarer med bildene og ønsket resultat.

Kjøring 1 - Normal:

Nr	Rotasjon [°]	Forskyvning [m]	Stopp	Tid[ms]	Resultat
1	3.88	-0.0	Nei	12	Ja
2	2.29	0.02	Nei	13	Ja
3	0.02	0.02	Ja	11	Ja
4	-0.86	-0.0	Ja	11	Ja
5	0.07	0.01	Ja	11	Ja
6	0.86	0.02	Ja	12	Ja
7	2.97	0.02	Nei	13	Ja
8	3.16	-0.04	Nei	15	Ja
9	5.69	-0.01	Nei	12	Ja
10	0.5	-0.03	Nei	12	Ja
11	-0.39	-0.02	Nei	12	Ja
12	0.8	0.01	Nei	11	Ja
13	1.33	0.01	Nei	12	Ja
14	2.07	0.01	Nei	12	Ja
15	5.15	0.04	Nei	12	Ja
16	2.04	-0.0	Nei	11	Ja
17	0.47	-0.01	Nei	11	Ja
18	0.05	-0.01	Nei	11	Ja
19	0.22	-0.02	Nei	11	Ja
20	3.12	-0.0	Nei	12	Ja
21	1.04	-0.04	Nei	11	Ja
22	1.91	-0.01	Nei	12	Ja
23	-0.91	-0.01	Nei	15	Ja
24	2.93	0.01	Nei	11	Ja
25	9.28	-0.03	Ja	11	Ja

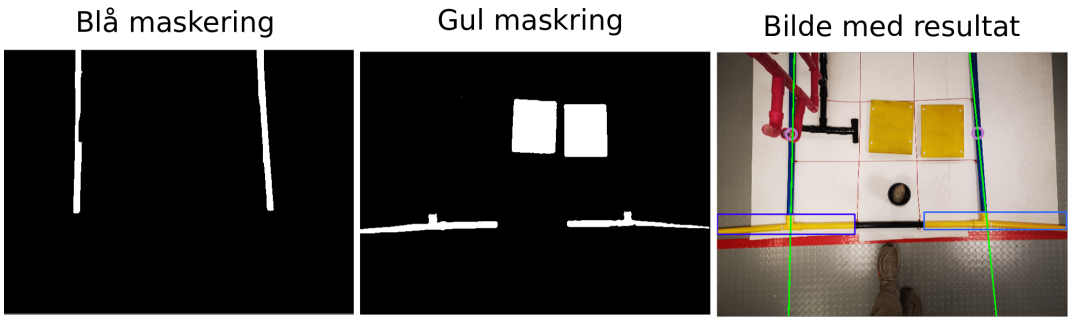
Kjøring 2 - Forskyvning:

Nr	Rotasjon [°]	Forskyvning [m]	Stopp	Tid[ms]	Resultat
1	1.04	-0.17	Nei	12	Ja
2	-1.61	-0.15	Nei	13	Ja
3	-0.29	-0.15	Nei	13	Ja
4	-0.68	-0.19	Nei	12	Ja
5	-3.39	-0.22	Nei	16	Ja
6	-3.16	-0.21	Nei	12	Ja
7	2.8	0.18	Nei	12	Ja
8	1.97	0.22	Nei	12	Ja
9	-0.25	0.2	Nei	12	Ja
10	-13.8	0.17	Nei	14	Ja
11	-5.47	0.21	Nei	12	Ja
12	-1.71	0.19	Nei	14	Ja
13	-4.39	0.03	Nei	12	Ja
14	-1.27	0.08	Nei	16	Ja
15	-6.49	0.05	Nei	17	Ja
16	4.11	0.08	Nei	11	Ja
17	1.55	0.08	Nei	12	Ja
18	4.53	0.05	Nei	12	Ja
19	4.42	-0.02	Nei	12	Ja
20	2.6	-0.02	Nei	12	Ja
21	3.73	-0.16	Nei	11	Ja
22	1.61	-0.08	Ja	11	Ja
23	8.98	-0.05	Ja	11	Ja
24	-2.62	-0.18	Ja	13	Ja
25	1.0	-0.19	Ja	12	Ja

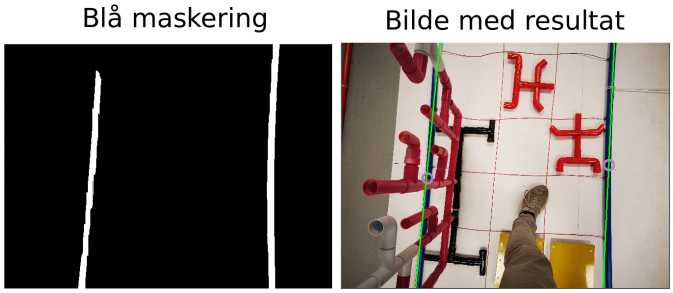
Kjøring 3 - Rotasjon:

Nr	Rotasjon [°]	Forskyvning [m]	Stopp	Tid[ms]	Resultat
1	-20.82	-0.12	Nei	14	Nei
2	-22.07	-0.06	Nei	15	Nei
3	-21.8	-0.05	Nei	15	Ja
4	-33.19	-0.09	Nei	15	Ja
5	-28.09	-0.15	Nei	14	Ja
6	-35.69	-0.23	Nei	13	Ja
7	-17.68	-0.13	Nei	14	Ja
8	-12.82	-0.13	Nei	13	Ja
9	-14.61	-0.1	Nei	13	Ja
10	-4.16	-0.09	Nei	11	Ja
11	15.36	-0.06	Nei	16	Ja
12	25.53	0.02	Nei	18	Ja
13	18.41	0.08	Nei	15	Ja
14	11.26	0.07	Nei	13	Ja
15	-14.22	0.11	Nei	14	Ja
16	-27.78	0.05	Nei	16	Ja
17	-30.37	0.04	Nei	17	Ja
18	-33.7	0.05	Nei	14	Ja
19	-45.47	0.07	Nei	11	Ja
20	-37.88	0.05	Nei	13	Ja
21	-6.5	-0.04	Nei	12	Ja
22	20.29	-0.07	Nei	15	Ja
23	28.59	-0.07	Nei	15	Ja
24	7.86	-0.04	Nei	11	Nei
25	5.14	-0.03	Nei	11	Nei

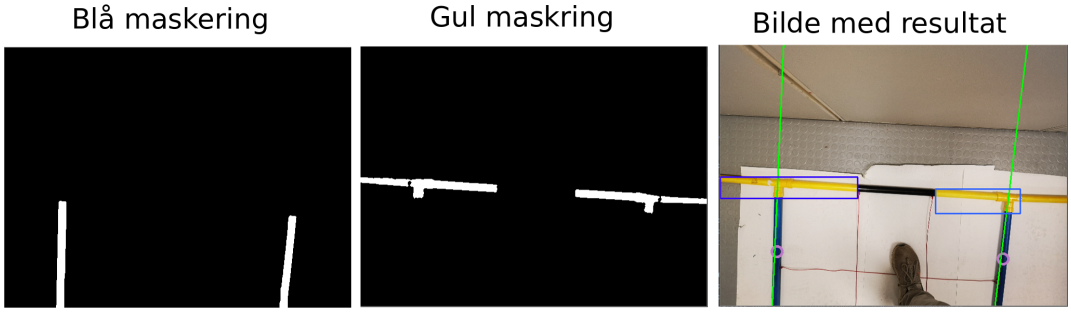
Utvalgte eksempler:
Eksemplene er ikke hentet ut fra testbildene i tabellen, men fra en annen serie testbilder. Disse ligger også vedlagt i vedlegget under ../Bildebehandling/Bilder/Auto_kjoring. Blå maskering benyttes av *get_angle_and_translation()* til å finne rotasjon og forskyvning. Gul maskering brukes av *find_end_bar()* til å finne enderørene.



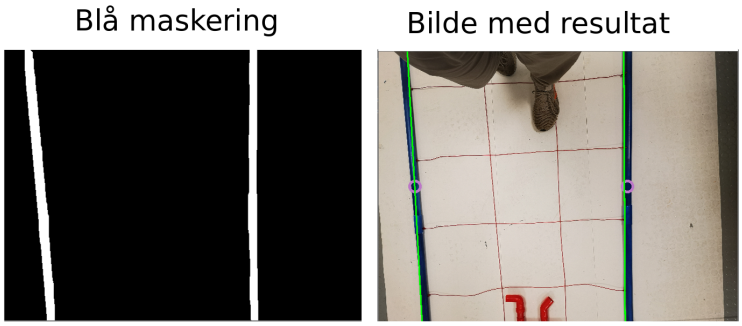
Figur 73: Eksempel 1. Bildet inneholder de gule enderørene samtidig som det inneholder to gule planteplasser for korallrev. Disse blir sett bort ifra som beskrevet i *find_end_bar()*.
Rotasjon: 2.3° Forskyvning: 0.04[m] Stoppsignal: Ja



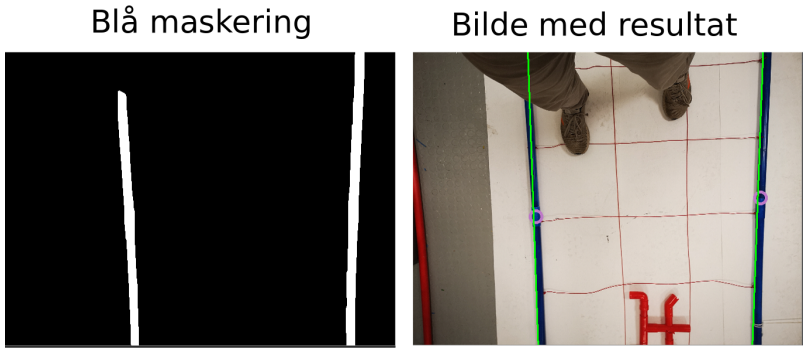
Figur 74: Eksempel 2. Utfordring med at korallrevet skygger over det ene blå røret. Dette blir ikke et problem da vi bruker morfologisk lukking(seksjon 3.5.4) til å koble sammen rørdelene. Dette er besrevet i metoden *get_angle_and_translation()*.
Rotasjon: -4.1° Forskyvning: -0.07[m] Stoppsignal: Nei



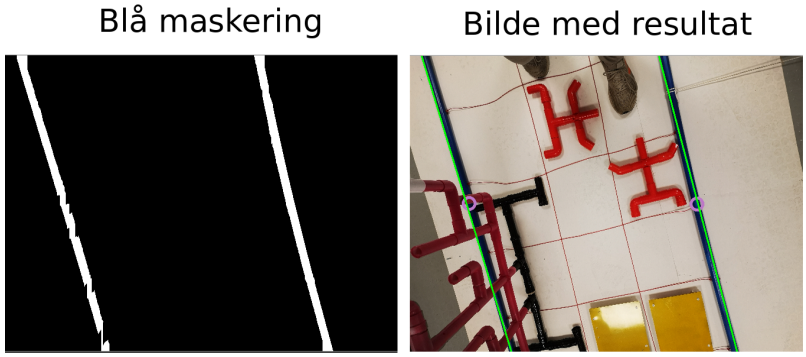
Figur 75: Eksempel 3. Dette bildet illustrerer situasjonen hvor ROV-en har kjørt gjennom hele traseen og har passert målstreken. De gule rørene er i den nedre delen av bildet. Dette oppdages, og kontrollen gis tilbake til operatøren.
Rotasjon: -4.8° Forskyvning: 0.02[m] Stoppsignal: Ja



Figur 76: Eksempel 4. Dette er et eksempel på et bilde med mye forskyvning. Rotasjon: 2.8° Forskyvning: 0.17[m] Stoppsignal: Nei



Figur 77: Eksempel 5. Dette er et eksempel på et bilde med så mye forskyvning at et rødt rør syntes. Gjennomføringen er dermed ugyldig og kan avbrytes manuelt.
Rotasjon: -0.4° Forskyvning: -0.19[m] Stoppsignal: Nei



Figur 78: Eksempel 6. Dette er et eksempel på en situasjon hvor ROV-en har rotert mye. Det er også mulig å observere hvordan korallrevet kutter det venstre blå røret opp. Røret blir koblet sammen ved hjelp av morfologisk lukking (seksjon 3.5.4).
Rotasjon: 15.9° Forskyvning: 0.10[m] Stoppsignal: Nei

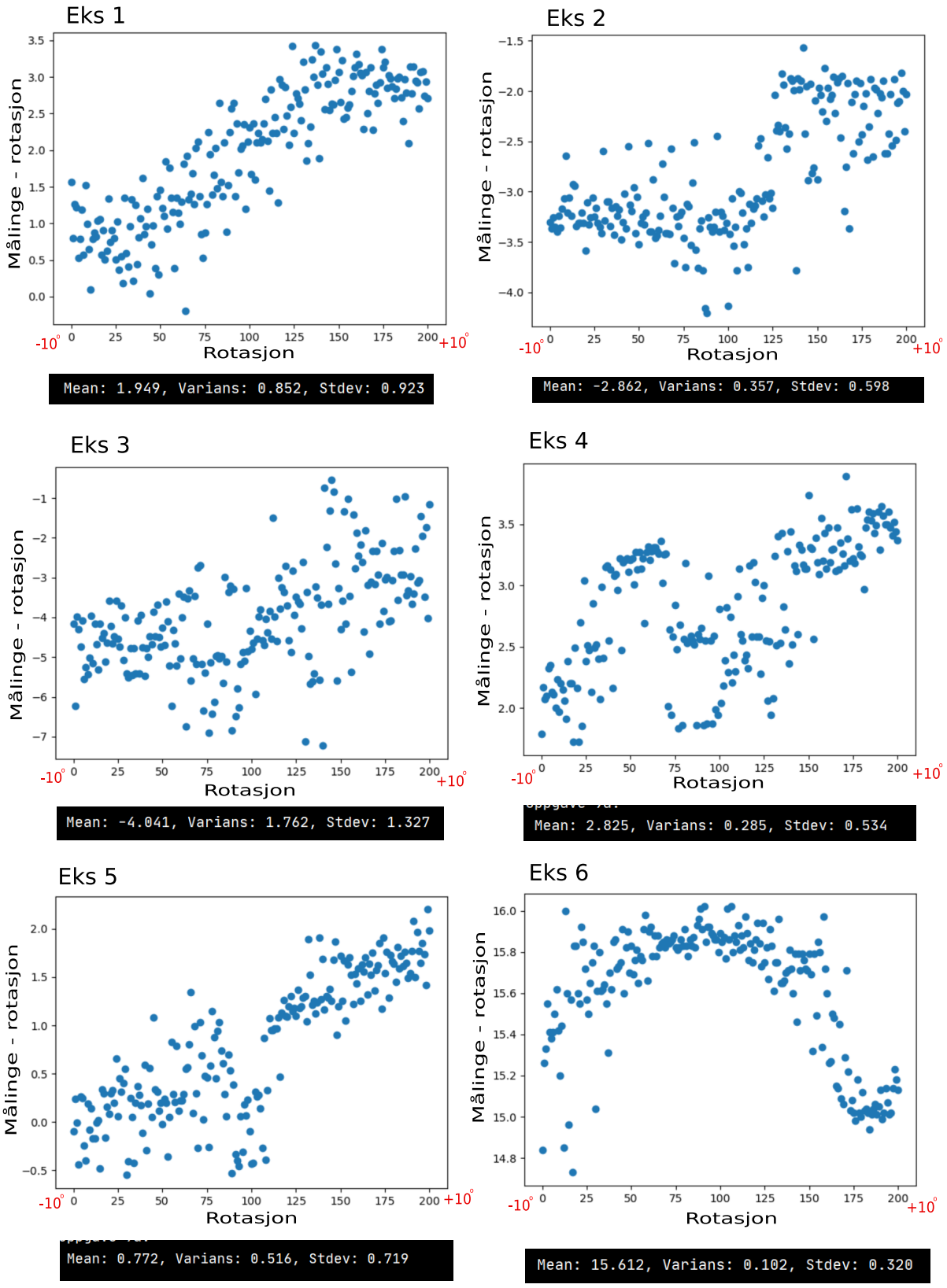
Test av treffsikkerhet ved rotasjon av bildene:
Denne testingen tar utgangspunkt i de seks eksempelbildene fra forrige avsnitt. Vi benytter rotasjonsfunksjonen fra seksjon 3.4.2 til å rotere bildene. Vi roterer så bildene $\pm 10°$ med $0.1°$ inkrement. For rotasjon måler vi den målte girvinkelen minus vinkelen som bildet har blitt rotert. Vi får på denne måten frem hvordan målingene varrierer ved rotasjon. For svaiforskyvning måler vi direkte uten justeringer.
Vi beregner så gjennomsnitt, varians og standardavvik, og tegner spredningsplott. Resultatene for girrotasjon er vist i figur 79 og i figur 80 for svaiforskyvning. I figur 82 og 81 sammenlignes resultatene i samme figur.

Tolkning av resultatene:

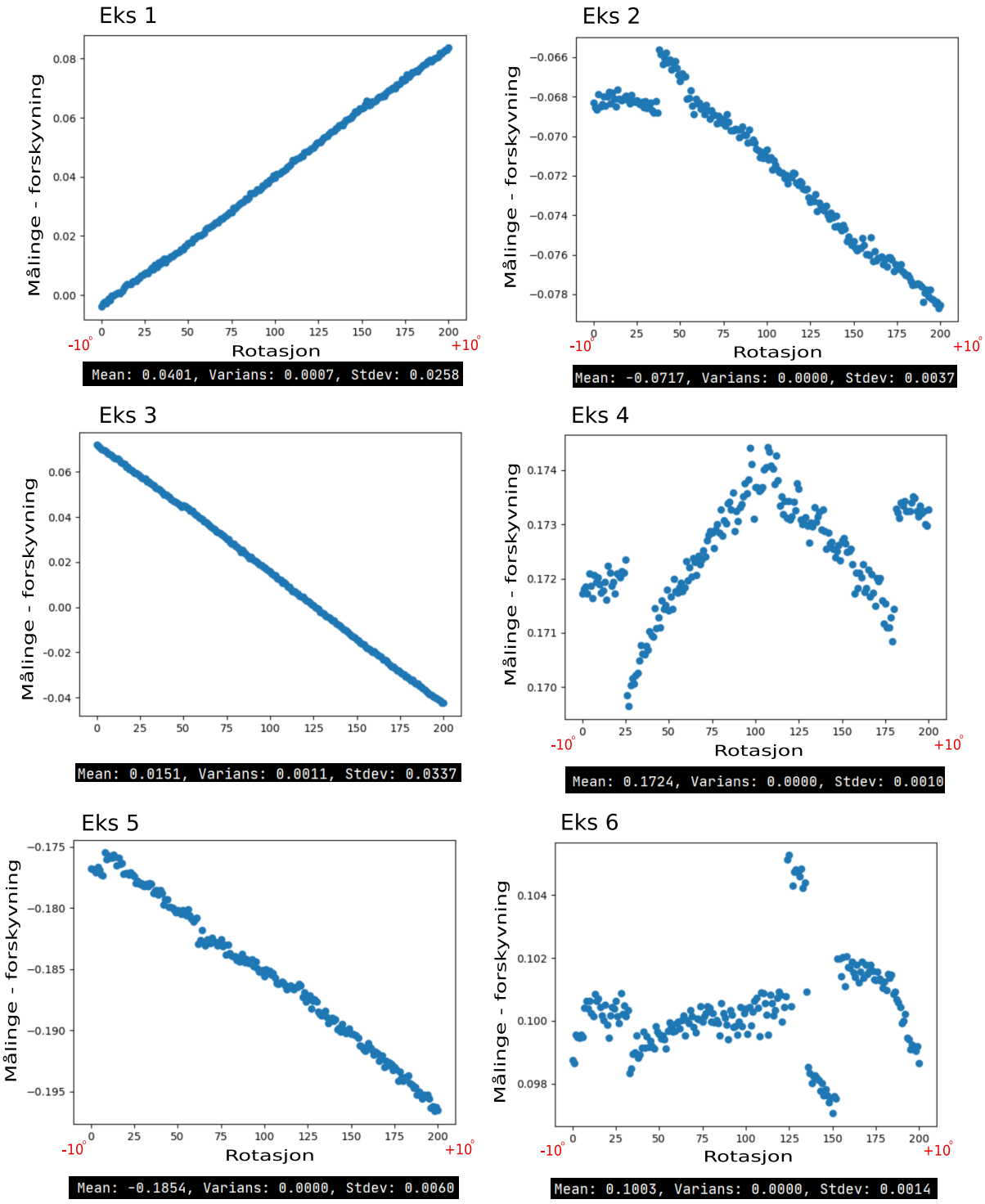
- Svai:
  Resultatene viser at det i de fleste tilfellene er en tydelig sammenheng mellom rotasjon og endring av girmålingen. Det fremstår ikke tydelig hva som er årsaken til denne sammenhengen. En mulig bidragsyter er at når vi roterer bildene, forskyves mer og mer av de blå rørene ut av bildet. De blå rørene er utgangspunktet for målingene, og om disse blir endret/redusert gir det dårligere målinger. Den sammenhengen kan ses i resultatene fra eks 1 og 3. Disse eksemplene har de høyeste variansene og den minste andelen av de blå rørene i bildet. En annen bidragsyter kan være metoden som *fitLine*(seksjon 3.3.7) bruker til å beregne vinklene. Testingen gir et godt grunnlag for analyse og viderutvikling av målemetoden. Det samlede resultatet for de seks testbildene gav en varians på 0.643 og standardavvik på 0.802. Med dette som grunnlag får vi resultatet at 95% av alle målingene ligger innenfor $\pm 1.6°$ av faktisk vinkel. Om vi ser bort ifra eksempel 1 og 3 blir intervallet $\pm 1.1°$.

- Gir:
  Resultatene viser en klar og tydelig systemisk sammenheng mellom rotasjon og endring av forskyvningsmålingen. Noe av dette kan antagelig forklares av målemetoden som har blitt brukt(se figur72). Metoden har ikke tatt hensyn til hva eventuelle rotasjoner vil ha å si for målingene. Testresultatene gir et godt utgangspunkt for utbedring og analyse av målemetoden. Eksempel 1 og 3 er bildene med minst andel av blå rør i bildet. Disse eksemplene har også den markant høyeste variansen. Det betyr at målingene er svært avhengig av de blå rørene. Det samlede resultatet for de seks testbildene gav en varians på 0.0003 og standardavvik på 0.0175. Med dette som grunnlag får vi resultatet at 95% av alle målingene ligger innenfor $\pm 3.5cm$ av faktisk faktisk vinkel. Om vi ser bort ifra eksempel 1 og 3 blir intervallet $\pm 0.7cm$.

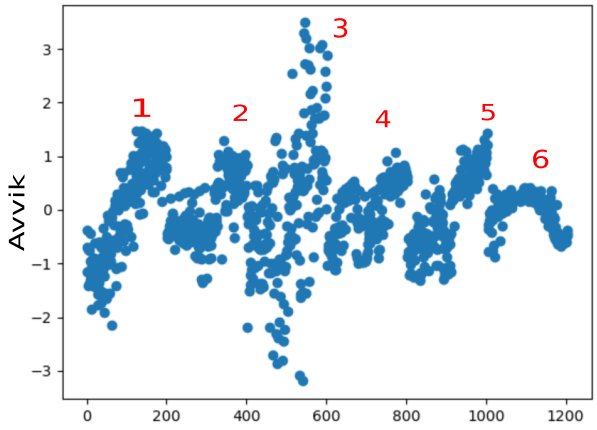
Merk at målemetodene er svært avhengig av kvaliteten på bildene. Merk også at vi ikke har kontrollert de målte vinklene mot faktisk vinkel. Vi kan derfor ikke si noe om metodenes riktighet, kun om treffsikkerheten. Et viktig moment fra denne testingen er at målingenes treffsikkerhet blir redusert når de blå rørene fyller en liten andel av bildet. Dette er tilstanden ved start og slutt av kjøringen. Dette er altså programmets mest sårbare faser. Testene viser også også en sammenheng mellom rotasjon av bildet og en forsterkning av feilbidraget til målingene. Rotasjon av bildet ser ut til å forskyve målingenes riktighet.



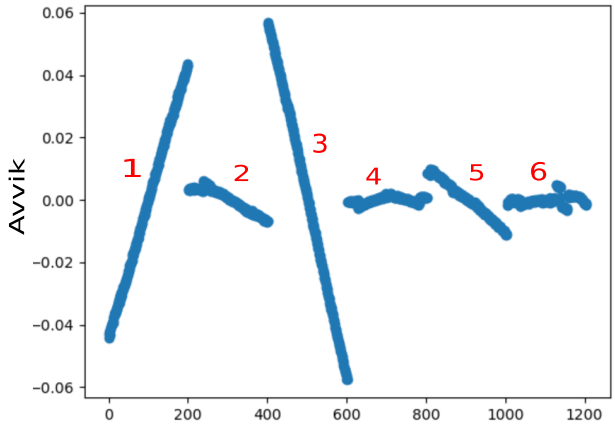
Figur 79: Resultatet av måling av girrotasjon mot rotasjon av bildet.



Figur 80: Resultatet av måling av svaiforskyvning mot rotasjon av bildet.



Figur 81: Sammenligning av testresultat fra de forskjellige eksempelbildene. Eksemplene er nummerert med røde tall.



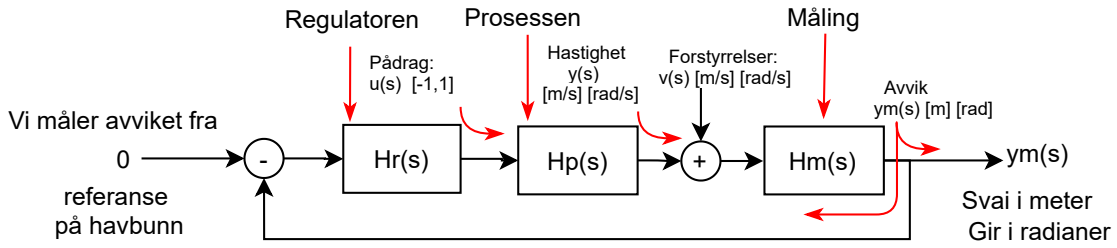
Figur 82: Testresultatene til svaimålingene sammenlignet. Eksemplene er nummerert med røde tall.

4.2.7 Resultat bildebehandlingsdelen

Det har blitt testet at programmet håndterer normale bilder, bilder med rotasjon og bilder med forskyvning. Programmet har håndtert dette og returnert fornuftige resultater. I tillegg har vi testet hvordan målingene blir påvirket av forskjellige grader av rotasjon. Vi har her funnet en klar sammenheng mellom lav andel blå rør i bildet og resultater med stor usikkerhet. Dette antyder at programmet er spesielt utsatt for feilmålinger i start- og sluttfasen av kjøringen, hvor en liten andel av rørene er synlig. I tillegg har det blitt identifisert et forbedringspotensial på målemetoden for svaiforskyvning. Denne har ikke tatt hensyn til hvordan rotasjon påvirker beregningen. Resultatene fra de mest optimale bildene anga et 95% konfidensintervall på $\pm 0.7cm$. og$\pm 1.1°$ ved rotasjon innenfor $\pm 10°$. En stor andel av feilbidragene forbundet med rotasjon virker å være forutsigbare. Det betyr at det skal være mulig og kompensere for dette feilbidraget.

4.2.8 PID-regulering

PID-regulatorenes hovedoppgave er å kompensere for forstyrrelser slik at avvikene forblir lik null. PID-regulatorenens hovedprioritet blir dermed god forstyrrelseskompensering. Hadde banen vært mer avansert og inneholdt svinger ville det blitt lagt større tyngde på reguleringens følgeegenskaper. I figur 83 beskrives de overordnede reguleringssystemene. Som tidligere beskrevet, måler vi avviket i meter og radianer.



Figur 83: Overordnet beskrivelse av reguleringssystemene for svai og gir. Variablene beskrevet i figuren benyttes gjennom resten av dette delkapittelet.

Likning 10, 11 og 12, vil bli brukt til å beskrive egenskaper ved reguleringssystemene. Definisjonene er hentet fra Finn Haugens bok om praktisk reguleringsteknikk[29](kapittel 7). Denne boken er utgangspunktet for hele dette delkapittelet.

$$\text{Sløyfetransferfunksjonen}: \quad H_0(s) = H_r(s) \cdot H_p(s) \cdot H_m(s) \tag{10}$$

$$\text{Sensitivitetsfunksjonen}: \quad N(s) = \frac{1}{1 + H_0(s)} \tag{11}$$

$$\text{Systemet/Følgeforholdet}: \quad M(s) = \frac{H_0(s)}{1 + H_0(s)} \tag{12}$$

H_p(s):
Til å beskrive denne delen av systemet har vi fått hjelp av motor- og reguleringsgruppen. De har laget en grov oversikt over ROV-ens overflateareal og motorens posisjoner og vinkling. De har så beregnet dynamikken til ROV-en i vannet. Deres utgangspunkt har vært regulering av skyvekraft, som igjen tilsvarer en hastighet. Vi har slik fått en 1.ordens respons som beskriver sammenhengen mellom regulatorens utgangsverdi og ROV-ens hastighet(likning 13 og 14). Tidskonstantene representerer ROV-ens dynamikk i vannet, mens forsterkningskoeffisentene representerer motorenes pådrag.

$$H_{p,svai}(s) = \frac{y(s)}{u(s)} = \frac{K_{svai}}{(T_{svai} \cdot s + 1)} = \frac{6.5}{(0.9 \cdot s + 1)} \tag{13}$$

$$H_{p,gir}(s) = \frac{y(s)}{u(s)} = \frac{K_{gir}}{(T_{gir} \cdot s + 1)} = \frac{40}{(0.2 \cdot s + 1)} \tag{14}$$

$y(s)$−Pådrag fra ROV, endringshastighet
$H_{p,svai}(s)$−ROV-ens pådragsdynamikk, svai
$H_{p,gir}(s)$−ROV-ens pådragsdynamikk, gir
$u(s)$ − Pådrag fra regulator
T − Tidskonstant
K − Pådragskoeffisient

Det er et ulineært forhold mellom reguleringspådrag og endringshastighet. Vannmotstanden ROV-en møter er ulik ved varierende hastighet. Mer bakgrunnstoff om dette kan finnes her [33]. Verdiene satt inn i likning 13 er funnet ved et arbeidspunkt med en svai hastighet på 0.2 [m/s]. Dette er noe høyt da det kun skal kompenseres for svært små avvik. Verdiene i likning 14 er funnet når ROV-en står i ro. Dette representerer heller ikke virkeligheten optimalt da ROV-en vil kjøre fremover, som igjen vil påvirke girprosessens dynamikk. Uavhengig av dette tar vi utgangspunkt i verdiene fra motor- og reguleringsgruppen i den videre utredningen av reguleringsprosessen. Idet ROV-en blir operasjonell vil vi ved hjelp testing fastsette disse verdiene mer presist.

$\mathbf{H}_m$(s):
Målefunksjonen består av to faktorer: en tidsforsinkelse og en integrator. Tidsforsinkelsen inneholder tiden fra måleøyeblikket til faktisk motorpådrag. Integratoren kommer av at vi måler avvik fra en referanse på havbunnen. Målemetodene blir illustrert i kapittel 4.2.4 og 4.2.5. Avviket vi måler blir distansen/vinkelen ROV-en må forflytte seg for å oppnå null avvik. Sammenhengen mellom prosessen og målingen blir at målingen integrerer alle endringer av svaiforskyvning og girrotasjon. Målefunksjonen er vist i likning 15.

$$H_m(s) = \frac{y_m(s)}{y(s)} = \frac{1}{s} \cdot e^{-\tau s} = \frac{1}{s} \cdot e^{-0.153s} \tag{15}$$

$H_m(s)$−Målefunksjonen
τ−Tidsforsinkelsen
$y_m(s)$ − Avvik
$y(s)$ − Pådrag fra ROV, endringshastighet

Tidsforsinkelsen består av bildetagningstid, bildebehandlings- og beregningstid, kommunuikasjonstid, og tiden frem til motorene responderer på pådragskommandoen. De tre første har vi gode overslag på. Det siste punktet antar vi at er null. I likning 16 har vi regnet ut den totale tidsforsinkelsen. Bildetagningstid er hentet fra seksjon 7.3.1 og ble funnet å være 70 ms. I delen om testing av bildebehandlingsalgoritmen blir tidsbruken anslått til å være 13 ms. Skrivehastigheten til ROV-en er anslått til å være 70 ms. Dette overslaget er hentet fra testing i samarbeid med kommunikasjonsgruppen. I likning 16 har disse blitt summert til den totale forsinkelsen. Siste faktor i likning 15, viser forsinkelsens bidrag til systemene.

$$Tidsforsinkelse = 70 + 13 + 70 = 153ms,\ Oppdateringsfrekvens = \frac{1}{0.153} = 6.5Hz \tag{16}$$

Filtrering:
Vi ønsker å benytte oss av et 1.ordens lavpassfilter. Filterets hovedoppgave er å begrense skaden eventuelle feilmålinger får for reguleringssystemet. Likning 17 viser overføringsfunksjonen til et slikt filter. T_f er filterets tidskonstant.

$$H_f(s) = \frac{1}{T_f \cdot s + 1} \tag{17}$$

I praksis benyttes dette filteret i et tidsdiskret system. I [28] utleder Finn Haugen et enkelt IIR-filter fra transferfunksjonen til et 1.ordens lavpassfilter. IIR står for uendelig impulsrespons. Filteret benytter seg kun av nåværende måling og den forrige filtrerte verdien. Styrken eller størrelsen til filteret avgjøres av hvor mye nåværende måling og den forrige filtrerte verdien skal ha for denne utgangsverdien. I likning 18 har vi IIR-filterts praktiske implementasjon. Symbolet α angir filterets styrke.

$$\text{Ny utgangsverdi} = \alpha \cdot \text{Måling} + (1 - \alpha) \cdot \text{Forrige utgangsverdi} \tag{18}$$

Sammenhengen mellom α, T_f og τ(sampleintervallet er likt som tidsforsinkelsen) er gitt i likning 19. Utledningen til dette kan ses i [28]. IIR-filteret bidrar til reguleringssystemet med en tilnærmet 1.ordens respons. For å bevare egenskapene til det tidsdiskrete filteret likt egenskapene til et tidskontunerlige 1.ordens filter $H_f(s)$, bør kravet i likning 20 tilfredsstilles.

$$\alpha = \frac{\tau}{T_f + \tau} \tag{19}$$

$$\tau \leq \frac{T_f}{5} \tag{20}$$

- Lav oppdateringshastighet og filtrering:

I utgangspunktet vil filteret tilføre systemet en 1.ordens respons hvor tidskonstant er avhengig av filterets styrke og systemets oppdateringsfrekvens. Hva filterets styrke blir er også avhengig av hvordan det påvirker systemets stabilitet. Optimalt sett ønsker vi oss en oppdateringshastighet som er så stor at det nødvendige filteret kan bli sett bort ifra i den store sammenhengen. Dette er ikke tilfellet her.

Nå når oppdateringsfrekvensen er på kun 6.5Hz får filteret straks en stor negativ betydning for systemets stabilitetsegenskaper. Derfor blir ikke dette filteret brukt. Det vil eventuelt kunne tas i bruk om høyere oppdateringshastighet oppnås.

Som en kompensasjon for manglende filtrering legger vi inn en logikk som ikke tillater større endringer fra måling til måling enn 0.02 meter for svai og 2 ° for gir. Denne logikken vil redusere konsekvensene en eventuell feilmåling får for prosessen. Det er svært usannsynlig at dette vil påvirke prosessene ved andre tilfeller enn eventuelle feilmålinger. Med en oppdateringsfrekvens på 6.5 Hz skal det ha gått veldig galt for at ROV-en faktisk har fått et avvik større enn de definerte grensene. For eksempel vil ROV-en med en hastighet på 0.2 [m/s], kun ha beveget seg 0.03 [m] i løpet av et sampleintervall.

$\mathbf{H_r(s)}$:
Som nevnt tidligere blir systemet ustabilt ved normal filtrering av måleverdiene. Som et kompromiss mellom god stabilitet og trygge målinger tar vi filteret med på regulatorens derivasjonsdel. Vi benytter oss av regulatoren i likning 21. Denne er på såkalt parallellform og har en filtrering av derivatdelen.

$$H_r(s) = K_p + \frac{K_p}{T_i \cdot s} + \frac{K_p \cdot T_d \cdot s}{T_f \cdot s + 1} \tag{21}$$

T_d−Derivasjonstid.
T_f−Tidskonstanten til filteret
på derivasjonsdelen. For eksempel $10 \cdot T_d$
T_i−Integrasjonstid.
K_p−Forsterkningskoeffisient.

Det er en god regel å alltid filtrere inngangsverdiene til derivasjonsdelen av PID-regulatoren. For oss vil det bidra til ustabilitet i systemet. Men det å ikke ha med en filtrering på derivasjonsdelen utgjør en ekstra risiko for pådragsendringer forårsaket av høyfrekvent støy på målingene. Utslagene på derivasjonsdelen er også begrenset av logikken beskrevet i siste del av seksjonen om filtrering. Dette tas det ikke høyde for i simuleringene.

Regulatorparametrene:

Som nevnt innledningsvis er det posisjons-/rotasjonsavviket fra havbunnens referanse som skal reguleres. For å finne regulatorenes parametre tar vi utgangspunkt i likning 22 og 23. Dette er $H_p(s)$ og $H_m(s)$ fra tidligere. For definisjon av variable se likning 13, 14 og 15.

$$H_{pm,svai}(s) = H_{p,svai}(s) \cdot H_m(s) = \frac{y_m(s)}{u(s)} = \frac{K_{svai}}{(T_{svai} \cdot s + 1)} \cdot \frac{1}{s} \cdot e^{-\tau s}\ [m] \tag{22}$$

$$H_{pm,gir}(s) = H_{p,gir}(s) \cdot H_m(s) = \frac{y_m(s)}{u(s)} = \frac{K_{gir}}{(T_{gir} \cdot s + 1)} \cdot \frac{1}{s} \cdot e^{-\tau s}\ [rad] \tag{23}$$

Dette er 1.ordens prosesser med integrator og tidsforsinkelse. For å finne regulatorparameterne benytter vi oss av Skogestads metode for et slik system. Denne er beskrevet i linje 4 av tabellen i figur 84.

$H_p(s)$ (process)	K_p	T_i	T_d
$\frac{K}{s}e^{-\tau s}$	$\frac{1}{K(T_C+\tau)}$	$k_1\,(T_C+\tau)$	0
$\frac{K}{Ts+1}e^{-\tau s}$	$\frac{T}{K(T_C+\tau)}$	$\min\left[T,\, k_1\,(T_C+\tau)\right]$	0
$\frac{K}{(Ts+1)s}e^{-\tau s}$	$\frac{1}{K(T_C+\tau)}$	$k_1\,(T_C+\tau)$	T
$\frac{K}{(T_1s+1)(T_2s+1)}e^{-\tau s}$	$\frac{T_1}{K(T_C+\tau)}$	$\min\left[T_1,\, k_1\,(T_C+\tau)\right]$	T_2
$\frac{K}{s^2}e^{-\tau s}$	$\frac{1}{4K(T_C+\tau)^2}$	$4\,(T_C+\tau)$	$4\,(T_C+\tau)$

Figur 84: Formler for beregning av regulatorparametre etter Skogestads metode. Innholdet er hentet fra Finn Haugens bok om praktisk reguleringsteknikk[29] side 221. Parameterne definert her forutsetter en regulator på serieform. Vi benytter derfor likning 2.45-2.47 i [29] til å transformere parameterne fra serie- til parallellform.

$k1$ – Velg 4 for gode følgeegenskaper og 1.44 for gode kompenseringsegenskaper([29] side 221).

τ – Dødtiden/tidsforsinkelsen.

T_C – Systemets tidskonstant definert av oss. Settes ofte lik tidsforsinkelsen ([29] side 221).

- Valg av k1:
Reguleringssystemene skal prioritere gode kompenseringsegenskaper. K1 settes derfor til 1.44 slik det blir anbefalt i Praktisk Reguleringsteknikk[29]. Dette vil gå utover prosessens stabilitet og oversving.

- Tidskonstanten:
Tidskonstanten til et system, T_c ,velges ofte til å være lik tidsforsinkelsen. I vårt tilfelle blir dette svært lavt. For å bevare systemnes stabilitet, og ikke lage en overivrig regulering, har vi testet system der denne er noe høyere. Dette skal bli den tiden systemet bruker på å hente inn 63% av avviket ved et sprang. Det blir ikke alltid resultatet, men ofte nært.

- Oversving:
Både for svai- og girprosessene går det greit med høyt oversving i sprangresponsen. Prioriteten er at prosessene har gode kompenseringsegenskaper. Årsaken til at oversving ikke anses som et problem er at vi forventer små avvik i størrelsesorden [-3,3]° og [-0.05,0.05][m]. Det viktigste er at avviket reduseres, samtidig som prosessens stabilitet ivaretas. Utenom dette er maksimalt tillatt oversving det samme som maksimalt avvik. Maksimalt avvik er begrenset av hva som definerer en godkjent gjennomkjøring.

- Stabilitetsegenskaper:
Prosessens stabilitetsegenskaper måles i fasemargin og forsterkningsmargin. Stabilitetsegenskapene tar utgangspunkt i frekvensresponsen til sløyfetransferfunksjonen H_0. I figur 85 og 86 markeres disse verdiene på H_0 sin frekvensrespons.
Vi gjør stabilitetsegenskapene om til mer konkrete verdier som beskriver systemets robusthet. I likning 24 benyttes fasemarginen til å beregne hvor mye større systemets tidsforsinkelse kan bli. Og i likning 25 regnes forsterkningsmarginen om fra dB, til en faktor.

For mer informasjon om stabilitetsegenskaper se [29] kapittel 7.6.

$$\Delta\tau_{maks} = \frac{F_m}{\omega_{avlesing}} \frac{\pi}{180^\circ} \quad (24)$$

$$\Delta K = 10^{\frac{G_m}{20}} \quad (25)$$

$\Delta\tau_{maks}$−Den maksimale forlengelsen av prosessens tidsforsinkelse, før systemet blir ustabilt.
F_m−Fasemarginen i grader.
$\omega_{avlesing}$−Frekvensen i [rad/s] ved avlesningspunkt for fasemargin.
G_m−Forsterkningsmargin i dB.
ΔK−H_0 sin maksimale forsterkning før systemet blir ustabilt.

- Sensitivitetsbåndbredden:
Sensitivitetsbåndbredden angir den øvre frekvensen av reguleringsavvik som prosessen kompenserer godt for. Sensitivitetsbåndbredden er definert til å være forholdet mellom avviket til regulatoren med tilbakekobling og avviket til regulatoren uten tilbakekobling. Båndbredden blir definert av en øvre frekvensgrense angitt av frekvensen hvor absoluttverdien av N(s)(sensitivitetsfunksjonen) er lik -11dB. Sensitivitetsbåndbredden brukes som et mål på hvor gode kompenseringsegenskaper systemet har. For mer informasjon om dette se [29] side 162-171. Her beskriver Finn Haugen sensitivitetsbåndbredden som det beste målet på regulatorens båndbredde.

- Forklaring til tabellen :

- Nr - Test nummer.
- Pr - Prosess svai/gir.
- T_c - Valgt tidskonstant for systemet.[s]
- O_s - Andel oversving.[%]
- F_m - Fasemargin i grader.
- T_r - Faktisk responstid[s]. Tiden fra et sprang til 63% av avviket er hentet inn. Illustrert i 85.
- G_m - Forsterkningsmargin i dB.
- T_f - Tidskonstant filtrering av derivasjonsdel til PID-regulatoren.
- Kmp - Sensitivitetsbåndbredden i radianer.

Vi kunne gjerne sett på avviksresponsen, men vi har valgt å bruke responsen til et referansesprang ved sammenligning og klassifisering av prosessene.

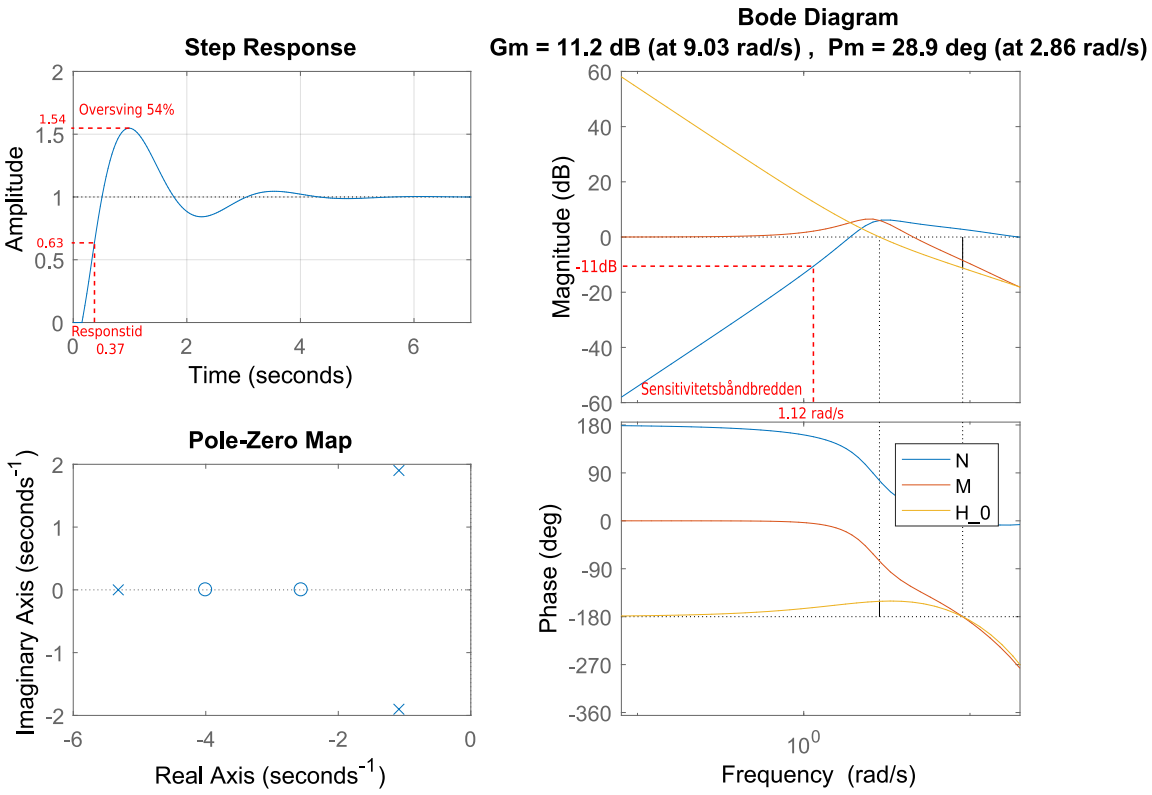
Nr	Pr	T_f	T_c	T_r	O_s	G_m	F_m	kmp
1	gir	0	τ	0.37	54%	11.2	28.9	1.12
2	gir	0	0.5	0.61	37%	17.4	42.2	0.59
3	gir	0	1	0.97	31%	22	48.9	0.35
4	gir	0.3	τ	0.43	93%	3.66	16.3	1.12
5	gir	0.3	0.5	0.62	38%	11.2	41.9	0.60
6	gir	0.3	1	0.93	30%	16.4	49.7	0.35
7	gir	0.75	τ	0.45	107%	2.31	8.05	1.15
8	gir	0.75	0.5	0.66	46%	10.7	36.5	0.60
9	gir	0.75	1	0.96	30%	16.3	48.6	0.35
10	gir	1.5	τ	0.46	112%	1.87	5.65	1.18
11	gir	1.5	0.5	0.68	52%	10.8	31.6	0.60
12	gir	1.5	1	0.98	33%	16.5	46	0.35
13	svai	0	τ	0.46	34%	14	46.3	0.72
14	svai	0	0.5	0.78	35%	10.7	45.4	0.44
15	svai	0	1	1.18	33%	23.9	47.4	0.29
16	svai	0.3	τ	0.58	72%	3.6	13.5	0.75
17	svai	0.3	0.5	0.84	41%	12.8	40.9	0.45
18	svai	0.3	1	1.17	31%	17.5	47.9	0.29
19	svai	0.75	τ	0.65	101%	2.41	6.27	0.80
20	svai	0.75	0.5	0.93	61%	9.89	27.4	0.46
21	svai	0.75	1	1.26	39%	15.3	42.4	0.29
22	svai	1.5	τ	-	-	-	-	-
23	svai	1.5	0.5	0.99	75%	8.09	17.9	0.47
24	svai	1.5	1	1.34	49%	14.1	33.6	0.30

- Valg av prosessegenskaper

Girprosessen har den raskeste dynamikken. Dette er også den viktigste regulatoren. Vi prioriterer som nevnt kompenseringsegenskaper. Om det blir oversving i girsystemet er det positivt. Et oversving vil bidra til at forskyvningen som oppsto ved giravviket blir rettet opp. Dette vil igjen påvirke svaiprosessen. Vi prioriterer derfor en rask prosess for gir, og lar svaiprosessen være tregere. Oversving i svaiprosessen har ingen ekstra positivt effekt, men vil heller ikke bidra negativt. Desto kraftigere filtreringen er, desto dårligere blir stabilitetsegenskapene. Det viser seg utfordrende å oppfylle kravet gitt i likning 20, samtidig som best mulig stabilitetsegenskaper skal bevares. Vi velger derfor å ikke benytte oss av filtreringen. Ikke engang på derivasjonsleddet til regulatoren.

Gir:

For gir velger vi egenskapene beskrevet i linje 1. I figur 85 beskrives systemet av sprangrespons, forsterkningsmargin, fasemargin, polplot og sensitivitetsbåndbredden.

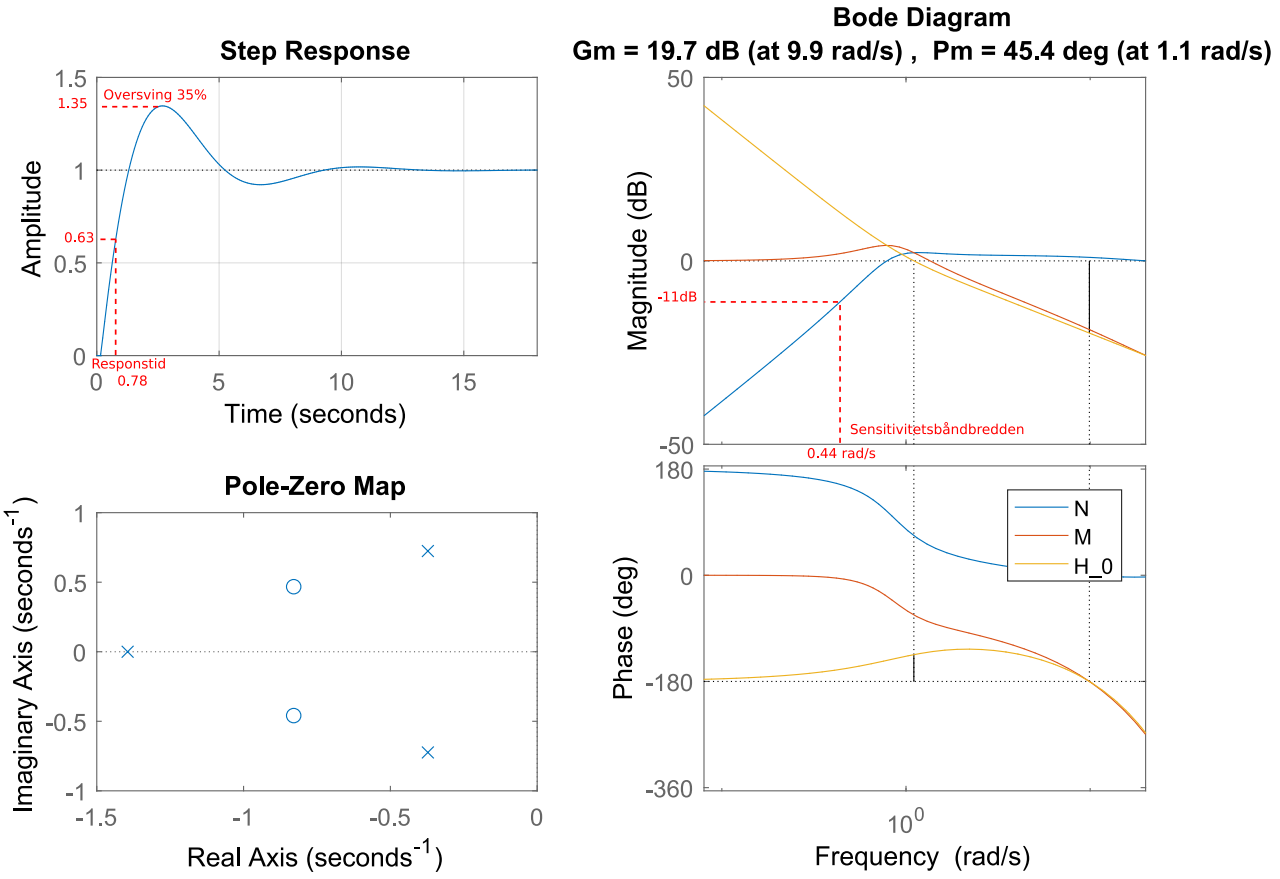


Figur 85: Fra polplottet ser vi at alle polene har en negativ realdel. Dette betyr at systemet er stabilt. Fra sprangresponsen finner vi at systemet har en oversvingning på 54%, og at systemet er stabilt. Fra bodediagrammet finner vi sensitivitetsbåndbredden, fase- og forsterkningsmargin.

Dette tilfellet har den beste kombinasjonen av kompenseringsegenskaper og stabilitet. Det er et stort oversving og ingen filtrering av derivatdelen. Fra testingen med filter fant vi ut at det var vanskelig å oppfylle kravet om størrelsen på filteret samtidig som vi beholdt de gode kompenserings- og stabilitetsegenskaper.

Ved å ta utgangspunkt i fasemarginen på 28.9° ved 2.86 [rad/s], og likning 24, finner vi den maksimale tilleggstidsforsinkelsen til å være 176[ms]. Blir systemets totale forsinkelse større enn $\tau + 176$ =329[ms] blir det ustabilt. En forsterkningmargin på 11.2 dB betyr at forsterkningen kan multipliseres med inntil 3.63 før systemet blir ustabilt(se likning 25).

Svai:
For svai velger vi egenskapene beskrevet i linje 14. I figur 86 beskrives systemet av sprangrespons, forsterkningsmargin, fasemargin, polplot og sensitivitetsbåndbredden.



Figur 86: Fra polplottet ser vi at alle polene har en negativ realdel. Dette betyr at systemet er stabilt. Fra sprangresponsen finner vi at systemet har en oversvingning på 35%, og at systemet er stabilt. Fra bodediagrammet finner vi sensitivitetsbåndbredden, fase- og forsterkningsmargin.
Dette systemet har en dobbelt så stor responstid, bedre stabilitet og dårligere kompenseringsegenskaper enn girsystemet. Systemet prioriter beste mulige kompenseringsegenskaper, uten å inkludere noen filtrering.
Ved å ta utgangspunkt i fasemarginen på 45.4° ved 1.1[rad/s], finner vi den maksimale tilleggstidsforsinkelsen til å være 720[ms]. Blir systemets totale forsinkelse større enn $\tau + 720$ =873[ms], blir det ustabilt.En forsterkningmargin på 19.7 dB betyr at forsterkningen kan multipliseres med inntil 9.66 før systemet blir ustabilt. Disse tallene uttrykker systemets robusthet(se likning 25).

4.2.9 Resultat regulering:

Reguleringsprosessene er stabile med gode marginer samt gode kompenseringsegenskaper. Ved å ha så robuste system øker vi sannsynligheten for at reguleringssystemet fungere godt på tross av unøyaktige forutsetninger. Viser det seg at målingene av svaiforskyvning og girrotasjon inneholder støykomponenter som forstyrrer regulatorene, bør vi benytte oss av filtreringen. Oppnår vi en høyere avlesningshastighet i systemet, kan egenskapene forbedres, og filter lettere implementeres.

4.3 Kartlegging av havbunn fra oversiktsbilde

Programmet er delt opp i en rekke funksjoner samt to klasser, Seabed og Square(se figur 88a). Figur 88 skisserer strukturen til løsningen. Innholdet i figur 88 er selvforklarende, der skisseres hovedtrekkene til løsningen. Etter dette kommer en detaljert gjennomgang av løsningen med blokkskjema og forklaring av de forskjellige stegene. Videre blir det litt om testing, og til slutt en vurdering av resultatene.

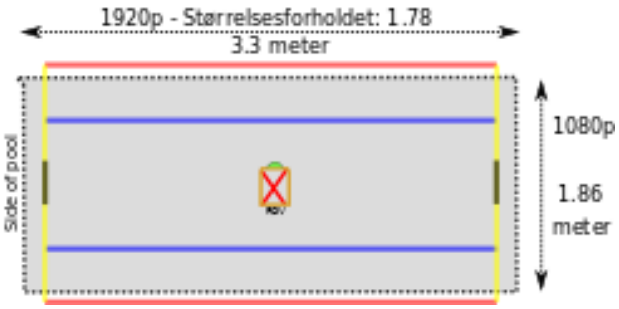
Antagelser og forutsetninger for løsningen:

- Programmet skal iverksettes etter at operatør manuelt har tatt et oversiktsbilde.
- Ved iverksetting lagres dette bildet på en definert plass hvorfra programmet henter det inn.
- Kartet hvor objektene skal tegnes inn ligger allerede lagret på definert plass.
- Oversiktsbildet bør optimalt sett tas fra en vinkel hvor ingen av objektene på havbunnen hindrer for utsikten til de blå rørene. Objektene må være plassert innenfor rutene i rutenettet.
- Venstresiden til ROV-en skal være rettet mot nærmeste bassengvegg når oversiktsbildet tas. Venstreside i forhold til siden med frontkamera og manipulator.
- Oversiktsbildet må inneholde hele området som skal kartlegges. Havbunnmodellen må altså fylle tilnærmet hele bildet. Den minste lengden disse blå rørene kan ha er satt til 75% av bildebredden. Laveste mulige bildebredde blir dermed 3 meter, etter modellens bredde, og maksimalt 4 meter på grunn av kravet til de blå rørene.

  For å sikre oss at alt kommer med kan vi for eksempel ta et bilde av havbunnen slik som illustrert i figur 87. Ved å ta et slikt bilde får vi med oss hele det interessante området. I likning 26 beregnes høyden over havbunnen som ROV-en må ha for å kunne ta et slikt bilde. Kameraets synsvinkel er hentet fra seksjon 4.2.3. I likning 27 beregnes høydens grenseverdier.

b – Havbunnbildets bredde delt på to.
h – ROV-ens høyde over havbunn.
θ – Kameraets synsvinkel delt på to.

$$h = \frac{b}{tangens(\theta)} = \frac{3.3[m]/2}{tangens(64°/2)} = \underline{\underline{2.64[m]}} \quad (26)$$

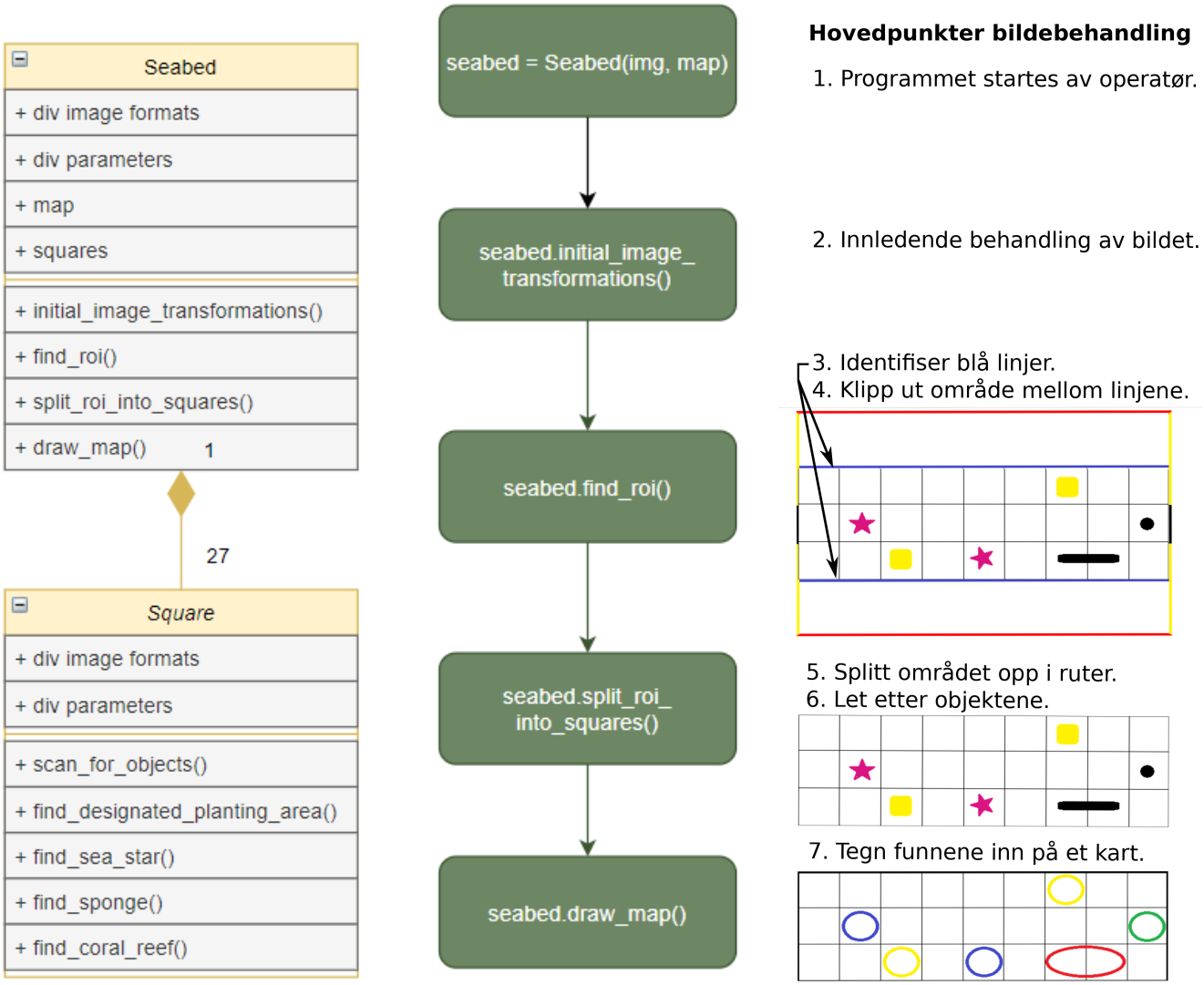


Figur 87: Illustrasjon av hvordan et bilde som dekker hele det aktuelle området kan tas.

$$\text{Laveste mulige høyde} = \frac{3[m]/2}{tangens(64°/2)} = \underline{\underline{2.4}}[m] \quad \text{Høyeste mulige høyde} = \frac{\frac{3[m]}{0.75}/2}{64°/2} = \underline{\underline{3.2}}[m] \quad (27)$$

Oversiktsbildet må altså tas fra en høyde over havbunnen på mellom 2.4 og 3.2 meter, gitt at kameraets synsvinkel forblir 64°.

4.3.1 Oversikt og blokkskjema



(a) Klassediagram (b) Flytdiagram (c) Hovedpunkter bildebehandling

Figur 88: Skjema som beskriver hovedlinjene i programmet som skal kartlegge havbunnen. Området mellom de blå linjene i delfigur (c) punkt 3/4, omtales heretter som ROI("region of interest").

4.3.2 Detaljert gjennomgang av løsningen:

1. **Oppstart fra brukergrensesnitt "Graphical User Interface" (GUI):**
   (a) Oppstart av program ved å trykke på knapp for kartlegging av havbunnen.
   (b) Dette henter ett oversiktsbilde(manuelt godkjent/bare ett bilde) som mates inn i programmet.

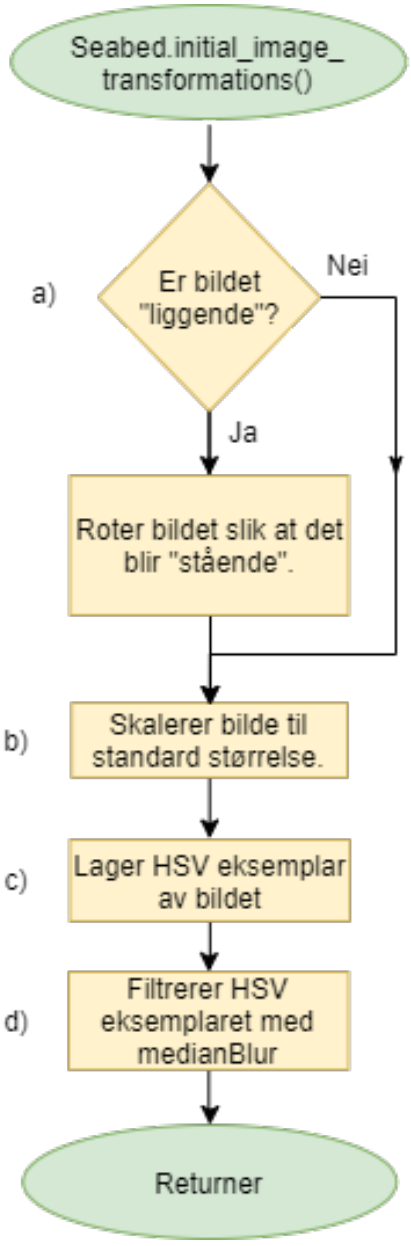
2. **Innledende bildebehandling:**

   (a) Roterer bildet slik at havbunnen som skal kartlegges har en vertikal langside. Til dette brukes OpenCv sin *rotate*

   (b) Skalerer ned bildet til en standardisert størrelse. Til dette brukes OpenCV sin *resize* funksjon[53]. Denne er omtalt i seksjon 3.4.4. Vi forventer å redusere bildestørrelsen og benytter dermed *INTER_AREA* metoden. funksjon[55].

   (c) Lager kopi av bildet i HSV-format.(Bruker *cvtColor* til dette, se seksjon 3.1.1).

   (d) Filtrere HSV-bildet med *medianBlur* funksjonen(se seksjon 3.10.1). Dette gjøres for å forbedre grunnlaget for fargemaskering. Maskeringsoperasjonen krever at bildet har en jevn fargefordeling.



Figur 89

3. **Identifikasjon av blå linjer:**

   (a) Her benytter vi oss av fargemaskering omtalt i seksjon 3.1.

   (b) Videre henter vi ut de to største konturene (seksjon 3.3.3). For å finne rotasjonen til bildet brukes *fitLine* (seksjon 3.3.7) på de to største konturene. Etter dette justerer vi inn rotasjonsavviket med rotasjonsfunksjonen fra seksjon 3.4.2.

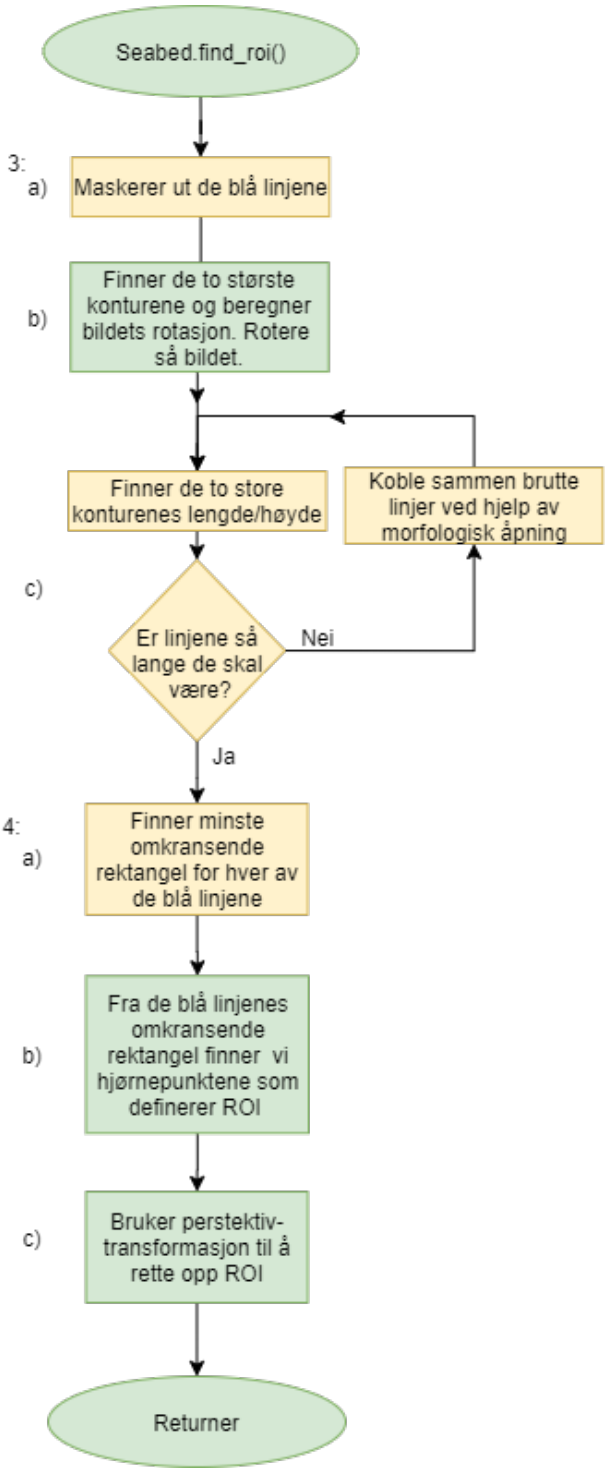
   (c) Det er satt et krav til hvor lange disse sammenhengende blå linjene skal være. Er disse under denne terskelverdien betyr dette at noe skygger for at vi ser hele linjen. Hvis dette er tilfellet benytter vi oss av morfologisk lukking til å sammenføye den brutte linjen(se seksjon 3.5.4). Dette gjøres helt til vi har sammenkoblet linjene. Dette er et kritisk punkt for programmet. Linjene må bli funnet og de må oppfylle kravene for å gi et tilfredsstillende resultat. Hovedgrunnen til rotasjonen i forrige punkt var nettopp å muliggjøre dette.

4. **Isoler det interessante område(ROI)**:

   (a) Minste omkransende rektangel (se seksjon 3.3.5) brukes til å hente hver av linjenes hjørnepunkter.

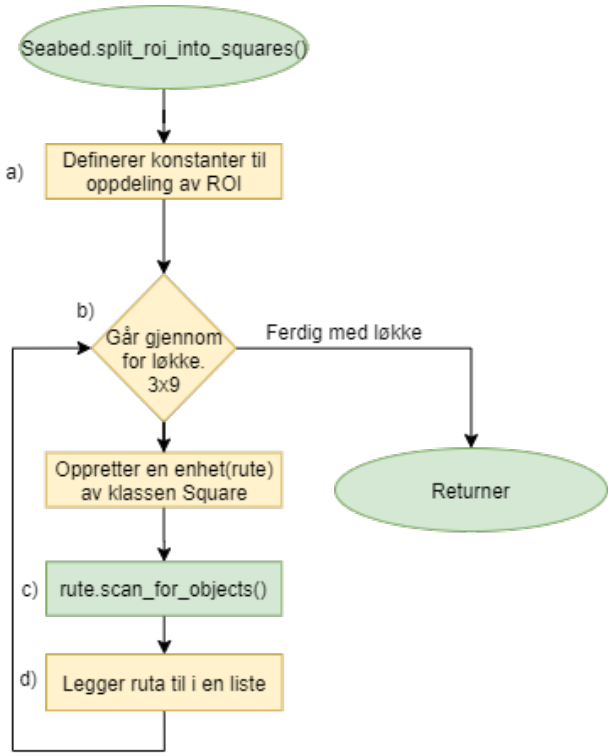
   (b) Disse åtte hjørnepunktene samles i en felles liste. Deretter brukes den spesialdesignede funksjonen *find_corner_points* 3.10.2, til å finne de fire ytterste hjørnepunktene. Dette gir oss en boks som inneholder ROI. Til slutt bruker vi funksjonen *shrink_contour* til å reduseres bredden på ROI slik at vi ikke tar med de blå linjene, men kun det mellom de blå linjene.

   (c) Videre benytter vi oss av perspektivtransformasjon(se seksjon 4) til å rette opp ROI. Disse skjevhetene kan for eksempel stamme fra at bildet ikke har blitt tatt fra en sentrert posisjon. Dette forbedrer treffsikkerheten til algoritmen og er essensielt for de videre stegene.



Figur 90

5. **Splitt området opp i ruter:**



Figur 91

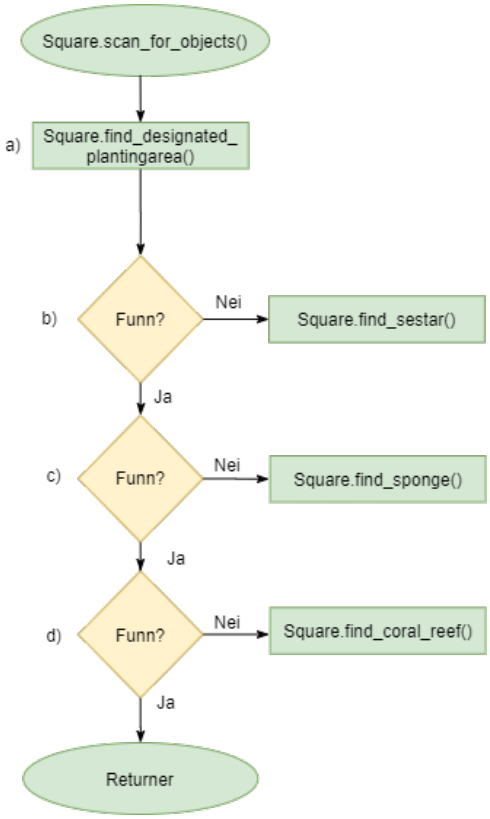
   (a) Først defineres noen konstanter som skal brukes videre. Bland annet en rutes lengde og bredde i piksler.

   (b) Bruker så en dobbel for-løkke til å gå gjennom og opprette de 27 rutene. Hver rute blir tildelt hvert sitt lille bilde av tilhørende område. Dette bilde er satt til å være 90x90 piksler. Ruta tildeles en utgave av bildet i RGB og ett i HSV-format.

   (c) For hver rute som blir opprettet kjøres klassemetoden *scan_for_objects()* på rutene.

   (d) Alle rutene blir samlet i en liste.

6. **Let etter objektene på havbunnen:**
Klassemetoden kaller andre klassemetoder til å lete etter objekter i ruta. Denne rekkefølgen er ikke tilfeldig, men er ment å optimere prosessens nøyaktighet. Vi gjennomfører derfor de klassifiseringene som er mest treffsikre først. Dermed blir sjansen for feil redusert ved at færre ruter blir sjekket med de minst nøyaktige metodene. At objektene som sjekkes først ikke er i ruten anses som et premiss for metodene som kommer senere i prosessen.



Figur 92

   (a) Leter først etter planteområdet for korallrev. Dette er de gule firkantene. Objektene er beskrevet i innledningen til oppgaven.

   (b) Så sant det ikke ble funnet noe leter vi etter sjøstjerner.

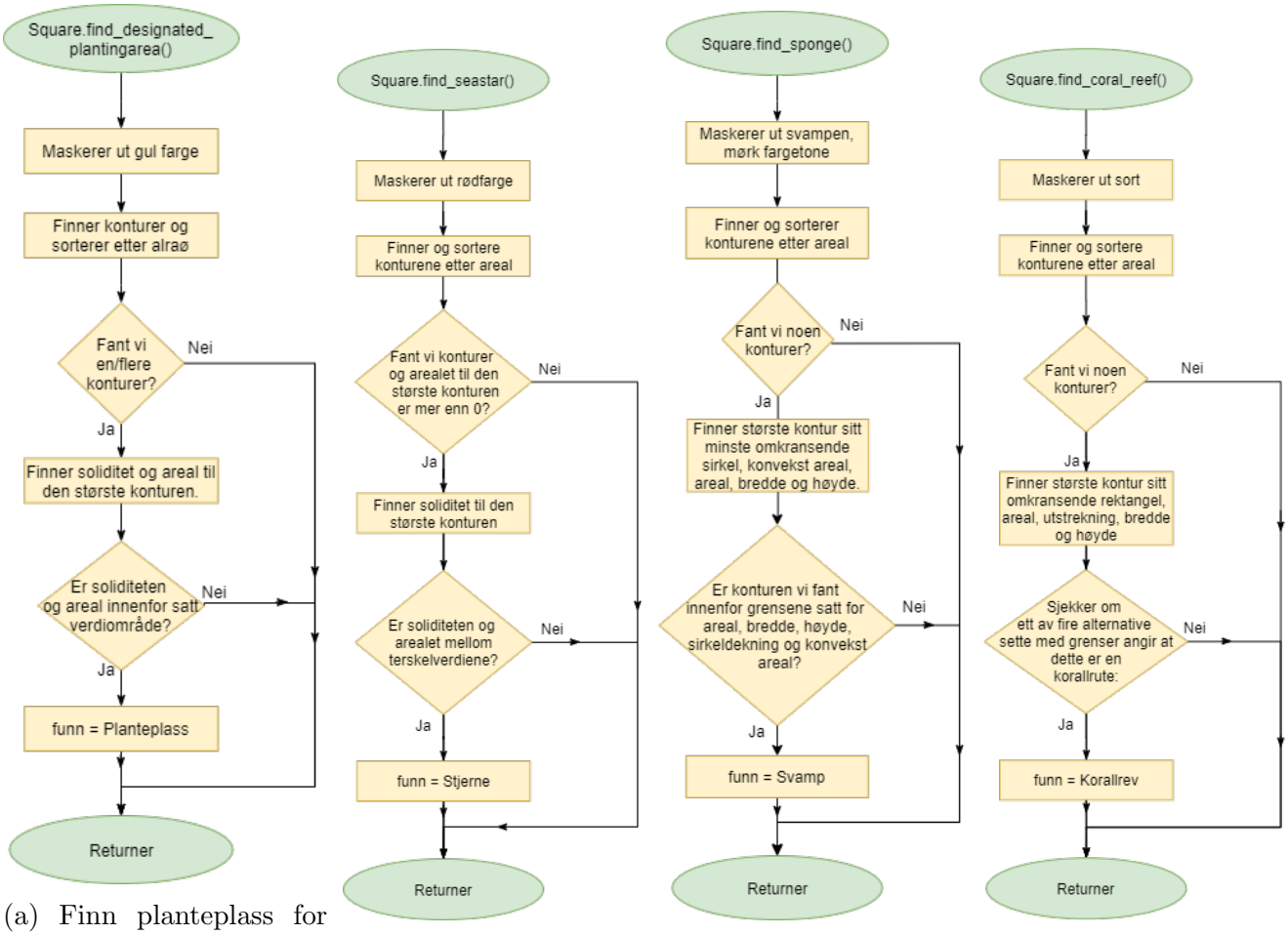
   (c) Deretter svamp.

   (d) Og til slutt koraller.

Letemetodene:
I figur 93 er metodene brukt av *scan_for_objekts* illustrert med flytskjema. Dette er altså metodene som brukes til å gjenkjenne de enkelte objektene på havbunnen. Disse objektene ble introdusert av figur 61.

Alle metodene benytter seg av fargemaskering, som er omtalt i seksjon 3.1, og konturer, som er omtalt i seksjon 3.3. Til å gjenkjenne objektene bruker metodene en rekke fastsatte grenser og terskelverdier. Disse konstantene beskriver typisk konturenes areal, soliditet, utstrekning, lengde eller bredde.

Tallverdiene til disse konstantene har blitt fastsatt ved hjelp av omfattende testing. Verdiene har gjennom hele utviklingen vært i kontinuerlig endring, og har blitt optimalisert til å gi en mest mulig robust gjenkjenning. De konkrete verdiene som oppgis her er ikke nødvendigvis de endelige tallene.



(a) Finn planteplass for korallrev. (b) Finn sjøstjerne. (c) Finn svamp. (d) Finn korallrev.

Figur 93: Flytskjema som beskriver hovedlinjene i metodene som skal kartlegge havbunnen.

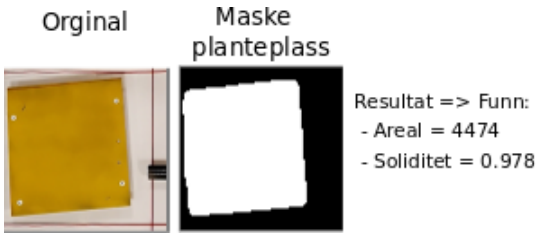
Planteplass:

Figur 93a viser strukturen til *find_designated_planting_area*-metoden. Her benyttes fargemaskering til å klippe ut gul farge. Deretter sjekkes det om eventuelle konturer har en soliditet(se seksjon 3) og areal som passer med objektet vi leter etter. For å gjenkjenne planteplassen har vi altså satt tre kriterier. Den skal være gul, større enn et gitt areal og være mer solid enn en satt grense. Dette gir en robust identifikasjon, spesielt da det ikke er noe som kan forveksles med dette objektet.

Den største feilkilden her er antagelig endrede lysforhold som kan kreve justering av valgte fargetoner som skal maskeres ut. I figur 94 er det et eksempel på en planteplass som blir funnet av metoden.

Kriterier for funn:

- Farge: Gul, (20, 20, 20)->(30, 255, 255) HSV OpenCV-format.
- Areal $>$ 3000.
- Soliditet $>$ 0.9.



Figur 94: Funn av planteplass.

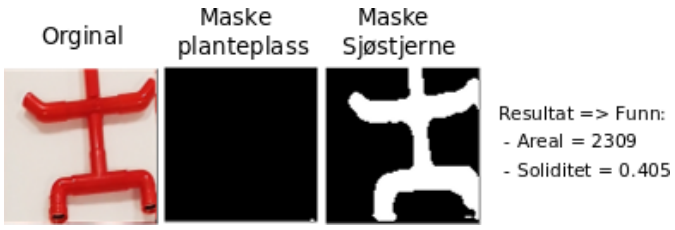
Sjøstjerne:

Figur 93b viser strukturen til *find_seastar*-metoden. Også her benyttes fargemaskering til å isolere ut rødefargene. Deretter sjekkes det om eventuelle konturer har en soliditet og areal som passer med objektet vi leter etter, en sjøstjerne. For å gjenkjenne sjøstjerne finner vi en rød kontur, denne må være innenfor øvre og nedre grenser for areal og soliditet. Dette gir en relativt robust identifikasjon.

En eventuell feilkilde er rett justering av fargetone og lysforhold. Under de rette lysforholdene er det potensielt mulig med en sammenblanding av de lilla delene av korallrevet og sjøstjernen. Denne risikoen reduseres ved hjelp av strenge krav til areal og soliditet. Figur 95 viser et eksempel på en sjøstjerne som blir funnet av metoden.

Kriterier for funn:

- Farge: Rød, (0, 50, 80)->(5, 255, 255) HSV OpenCV-format.
- $1500 < \text{Areal} < 3500$
- $0.3 < \text{Soliditet} < 0.6$



Figur 95

Svamp:

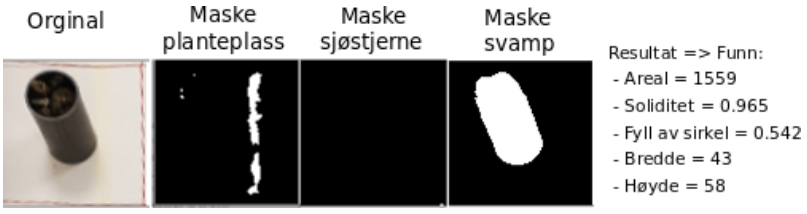
Figur 93c viser flyten til *find_sponge*-metoden. Vi bruker her fargemaskering å hente ut fargene som svampen har. Vi plukker så ut den største konturen, så sant den eksisterer. Deretter finner vi denne konturens konvekse areal(seksjon 3.3.12), minste omkransende sirkel(seksjon 3.3.8) og arealet av selve konturen(seksjon 3.3.3). Vi regner litt om og bruker så disse verdiene til å avgjøre om vi har funnet en svamp. For det første må konturen være innenfor øvre og nedre arealgrense. Deretter er det to alternativ.

Alternativ nummer en, hovedalternativet, bruker grenser for soliditet, høyde og bredde. Alternativ nummer to skal være et supplerende alternativ til de tilfellene hvor fargemaskeringen er dårlig. Her kontrolleres det hvor stort arealet innenfor den omkransende sirkelen er i forhold til kontures areal. Altså hvor mye av sirkelen konturen fyller opp. Dette alternativet er nesten overflødig, og vil sjeldent bli brukt av programmet. Er disse kriteriene oppfylt, defineres ruten til å inneholde en svamp.

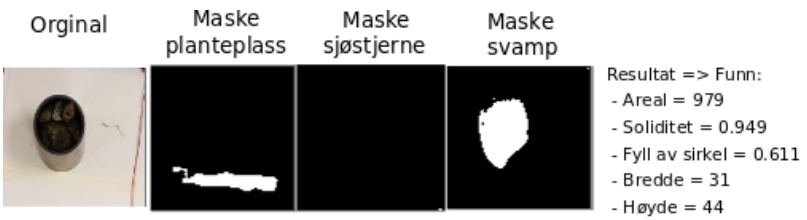
Figur 96 viser et eksempel hvor svampen blir funnet og alternativ én oppfylles. Videre i figur 97 har vi et eksempel hvor en svamp blir funnet og både alternativ en og to oppfylles.

Kriterier for funn:

- Farge: Uspesifisert mørk, (12, 30, 0)->(20, 170, 150) HSV OpenCV-format.
- $400 <$ Areal < 2500
- Alternativ 1:
  - Soliditet > 0.8
  - $15 <$ Bredde < 60
  - $15 <$ Høyde < 85
- Alternativ 2:
  - Andel av omkransende sirkel som er fylt: > 0.6



Figur 96: Funn av svamp ved hjelp av alternativ 1



Figur 97: Funn ved hjelp av alternativ 1 og 2.

Noe som gjør svampene vanskeligere å gjenkjenne er det at de strekker seg nokså høyt over havbunnen. Når bildet blir tatt fra forskjellige vinkler så endrer også svampens kontur seg betraktelig. I disse tilfellene er gjenkjenning av svampen et moment som gjør at det stilles strengere krav til hva slags bilder som kan brukes. Et annet risikomoment er at svampens fargetone er relativt lik den som brukes til å maskere ut korallrev.

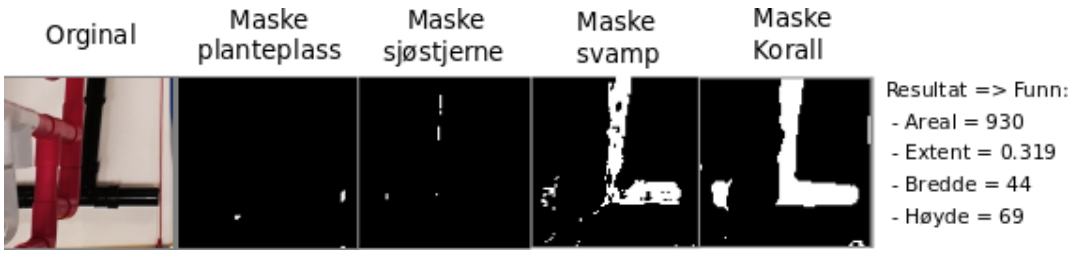
Korall:

Figur 93d viser strukturen til *find_coral_reef*-metoden. Først bruker vi fargemaskering til å sortere ut de mørke fargene. Deretter sorteres eventuelle konturer etter areal. Hvis vi da fant noen konturer sjekker vi om den største passer med kriteriene våre. Disse kriteriene angir grenser for areal, bredde, høyde og utstrekning(se seksjon 3.3.11).

Vi har laget fire alternativ som kan definere et korallrev. En metode for å finne et enkelt vertikalt rørstykke, enkelt horisontalt rørstykke, avkappet støtteben eller et komplett støtteben. Hvis kriteriene til et av disse fire alternativene oppfylles, så har vi antageligvis funnet et korallrev. Figur 98, 99, 100 og 101 viser eksempler hvor de fire alternativene oppdager ett korallrev hver.

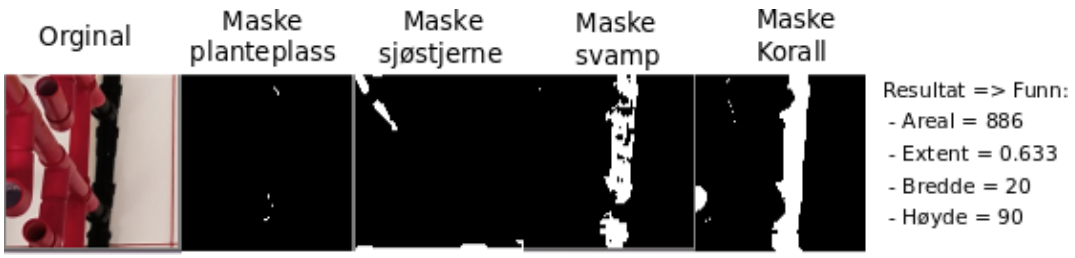
Kriterier for funn:

- Farge: Sort, (0, 0, 0)->(179, 130, 130) HSV OpenCV-format.
- Alternativ 1, avkappet støtteben:
  - 0.2 < Utstrekning < 0.7
  - 25 < Bredde < 80
  - 25 < Høyde < 80



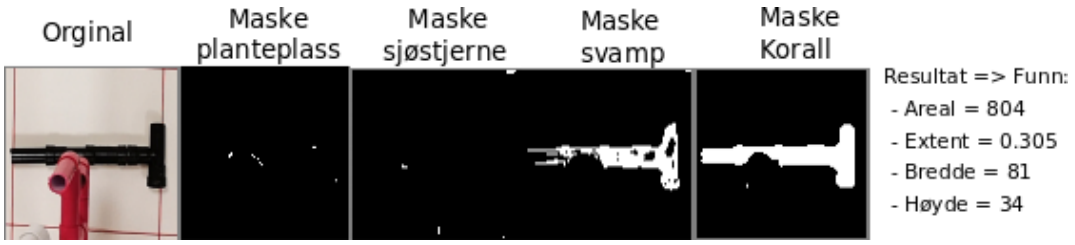
Figur 98: Eksempel på funn med alternativ 1, avkappet støtteben.

- Alternativ 2, vertikalt rørstykke:
  - Utstrekning > 0.5
  - 5 < Bredde < 25
  - 40 < Høyde < 91



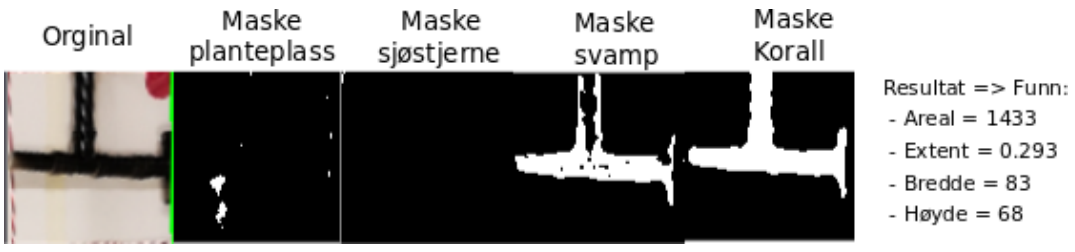
Figur 99: Eksempel på funn med alternativ 2, vertikalt rørstykke.

- Alternativ 3, horisontalt rørstykke:
  - $0.15 <$ Utstrekning < 0.75
  - $70 <$ Bredde < 91
  - $5 <$ Høyde < 40



Figur 100: Eksempel på funn med alternativ 3, horisontal rørstykk/støtteben som blir skygget for men ikke delt i to.

- Alternativ 4, komplett støtteben:
  - $1000 <$ Areal < 3000
  - $0.2 <$ Utstrekning < 0.4
  - Bredde > 40
  - Høyde > 40



Figur 101: Eksempel på funn med alternativ 4, komplett støtteben.

Som vi kan se fra disse bildene så utgjør metoden for å finne svamp en risiko. Både svamp og korallrev har mørke fargetoner og objektene fanges dermed delvis opp av begge metodene. I tillegg er også denne metoden sårbar for endringer i lyssettingen.

7. **Tegn funnene inn på et kart:**

   (a) Først defineres en rekke konstanter og verdier som muliggjør korrekt inntegning på kartet.

   (b) Deretter går vi gjennom listen hvor alle rutene er definerte. Fra hver rute sjekker vi om det er registrert ett funn. Hvis det er ett funn tegner vi inn tilhørende symbol.

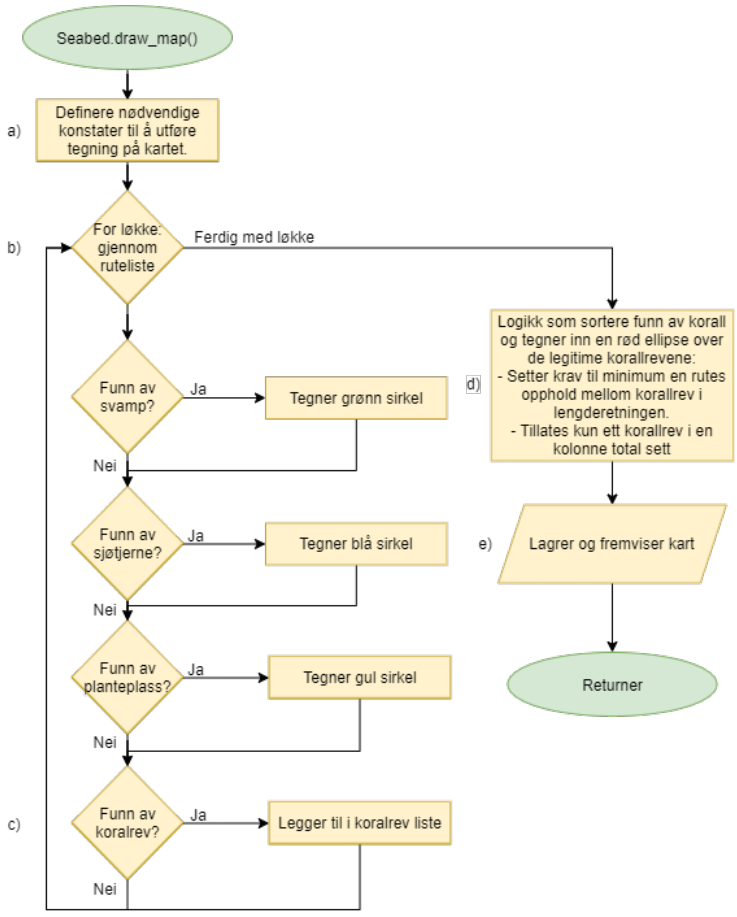
   (c) Det blir litt annerledes når vi finner en rute med korallrev. Dette symbolet skal dekke flere ruter. MATE har en bane hvor denne dekker maksimalt to ruter. Banen som vi har laget, med andre rørdimensjoner, har et korallrev som dekker tre ruter. Vi samler derfor opp en liste med alle rutene hvor det har blitt funnet korall og behandler disse til slutt.

   (d) Først sorteres korallene etter hvilke kolonne i rutenettet de hører til. Den kolonnen med flest koraller blir den vi tegner inn. Derreter beholdes kun de korallene som har en nær korallnabo i kolonnen.

   Vi tillater ikke større avstand enn en tom rute. Er avstanden større regnes ikke korallfunnet med i det endelige resultatet. I de tilfellene hvor korallrevet er plassert rett nedenunder kameraet, så skygger korallrevets øvrige struktur for den sorte delen. Vi finner i de tilfellene ikke korallrevrutene som er definert i Alternativ 2 for korallmetoden. Dette er årsalen til at vi tillater en rutes mellomrom.

   Det at vi kun tegner inn det største korallrevet filtrerer effektivt ut falske positive korallrev.

   (e) Etter at tegningen er fullført presenteres kartet til ROV-ens operatør.



Figur 102

4.3.3 Testing

For å teste programmet har vi tatt en rekke oversiktsbilder av testbanen. Disse har vi brukt til å teste om programmet fungerer som forventet. Oversiktsbildene er tatt i litt forskjellige vinkler, fra forskjellige posisjoner og med objektene plassert på forskjellige steder. Mange av testbildene er av langt dårligere enn den kvaliteten som ble spesifisert i forutsetningene for løsningen. Dette er meningen slik at vi får identifisert programmets svakheter og grenser. Vi har ikke tatt dette helt ut, men prøvd å holde oss innenfor de situasjonene som faktisk kan oppstå. Målet med testingen er å belyse programmets svakheter slik at vi enten kan forbedre det, eller motvirke og ta hensyn til svakhetene ved gjennomføringen i MATE-konkurransen.

Test av hovedprogram med oversiktsbilder:
Vi har i tabellen på neste side satt opp resultatene fra noen utvalgte testbilder. Etter denne tabellen analyseres feilene som ble funnet. På denne måten får vi frem programmets svakheter og begrensninger.

Forklaring til tabellen:

- Nr - Hvilke nummer det gjeldende testbildet har.
- Bilde - (x, y), H, K, T.
  - x - Omtrentlig forskyvning langs bildets x-akse. [m]
  - y - Omtrentlig forskyvning langs bildets y-akse. [m].
  - H - Et hjørne av ROI har blitt kuttet ut av bildet.
  - K - Korallrevet skygger over noe av de blå rørene.
  - T - Bildet har blitt tiltet kraftig. ROV-en skal ikke oppleve dette da den har regulering som holder den i vater.
- Rot - Angir hvor mye bilde blir rotert i forbindelse med å finne ROI..
- ROI - Funn av ROI. Andel virkelig rute som ligger i tilsvarende rute funnet av programmet.
- St - Antall Sjøstjerner funnet.
- P - Antall planteplasser funnet.
- Sv - Antall svamper funnet.
- K - Antall korallrev funnet.
- -/8 - Antall objekter funnet.

Nr	Bilde	Rot [°]	ROI	S	P	Sv	K	-/8
1	(1.2, 1), K, H	4.8	99%	2	2	1	3	8/8
2	(1.2, 1), K	2	98%	2	2	1	3	8/8
3	(1.2, 1), K	-3.7	98%	2	2	1	3	8/8
4	(1.2, 1) , K	-10.1	94%	2	2	1	3	8/8
5	(1.5, 1), T	-1.2	90%	2	2	1	3	8/8
6	(0.2, 0.3)	-1.3	100%	2	2	1	3	8/8
7	(0, 0.5)	-0.8	99%	2	2	1	3	8/8
8	(0, 0.6)	-0.7	100%	2	2	1	3	8/8
9	(0, 0.7), H	-12.4	95%	2	2	1	3	8/8
10	(0, 0.7)	8.5	100%	2	2	1	3	8/8
11	(0, 0.7), K	0.9	98%	2	2	1	3	8/8
12	(0, 0.3), K	3.8	97%	2	2	1	3	8/8
13	(0.5, 0.5), K	4.9	93%	2	2	1	3	8/8
14	(0.5, 0.5), K	5.8	89%	2	2	1	3	8/8
15	(0, 2)	0.5	92%	2	2	1	3	8/8
16	(0, 2)	2	94%	2	2	1	3	8/8
17	(0, 2)	1.4	95%	2	2	1	3	8/8
18	(0, 2)	0.7	95%	2	2	1	2	7/8
19	(0, 2)	1.0	92%	2	2	1	3	8/8
20	(0.5, 2), K, T	2	93%	2	2	1	3	8/8
21	(0.5, 2)	3	93%	2	2	1	3	8/8
22	(1, 2)	-5.7	90%	2	2	1	2	7/8
23	(1, 2)	5.1	91%	2	2	1	3	8/8
24	(1, 2), K	-5.6	93%	2	2	1	3	8/8
25	(1, 2), K, T	14.3	94%	2	2	1	3	8/8
26	(1, 2), K, T	10.6	0%	0	0	0	0	0/8

Nr	Bilde	Rot [°]	ROI	S	P	Sv	K	-/8
27	(0.3, 1.5),T	21.3	87%	2	2	1	3	8/8
28	(0.5, 1.5) H,T	-24.3	82%	2	2	1	3	8/8
29	(0.3, 1.5),K,T,H	-23.5	85%	2	2	1	3	8/8
30	(0, 1.5), K, T	-27.8	89%	2	2	1	3	8/8
31	(0, 0)	-3.5	97%	2	2	1	3	8/8
32	(1.5,0), K	5.3	98%	2	2	1	2	7/8
33	(1.5, 0), K	5.8	97%	2	2	1	3	8/8
34	(0.5, 1), K	0.3	68%	2	2	0	2	6/8
35	(0.5, 0.5), K	1.8	95%	2	2	1	3	8/8
36	(0.5, 0.7), H, K	2.7	93%	2	2	1	3	8/8
37	(0.5, 0.5), K	6.2	82%	2	2	1	3	8/8
38	(0.4, 0), K	5.8	89%	2	2	1	2	8/8
39	(1, 0.5), H, K	1.3	92%	2	2	1	2	7/8
40	(0,1.5), K	-16.9	88%	2	2	1	3	8/8

Nærmere kikk på feilene:

Nr	Bilde	Rot [°]	ROI	S	P	Sv	K	-/8
18	(0, 2)	0.7	95%	2	2	1	2	7/8
22	(1, 2)	-5.7	90%	2	2	1	3	7/8
32	(1.5, 0), K	5.3	98%	2	2	1	3	7/8

Disse tre forsøkene er egentlig vellykket. De endte opp med å ikke finne den midterste av de tre korallrevrutene. Dette kan skje i de tilfellene hvor kameraet står rett overfor korallrevet slik at korallrevet skygger for seg selv. I disse tilfellen finner vi kun endestykkene som har støtteben. Vi har altså kalkulert med at dette kan skje. Ellipsen som tegnes inn er mindre, men den berører fortsatt alle de tre rutene som korallrevet dekker. Vi får altså tegnet hele korallrevet inn på kartet. Forsøket kan dermed regnes som vellykket.

Nr	Bilde	Rot [°]	ROI	S	P	Sv	K	-/8
39	(1, 0.5), H, K	1.3	92%	2	2	1	2	7/8

I dette tilfellet blir ikke den ene korallrevruten funnet. Årsaken er at korallrevet er plassert skjevt fordelt mellom de tre rutene det fyller. Dermed oppstår det et spesialtilfelle hvor korallrevet i ruten ikke oppfyller noen av alternativene som gjelder for funn av korallrev.

Nr	Bilde	Rot [°]	ROI	S	P	Sv	K	-/8
34	(0.5, 1), K	0.3	68%	2	2	0	2	6/8

Denne testen feilet fordi korallrevet skygget fullstendig over enden til ett av de blå rørene. Dette førte til et forvridd ROI, som igjen førte til at ikke alle objektene ble funnet.

Nr	Bilde	Rot [°]	ROI	S	P	Sv	K	-/8
26	(1, 2), K, T	10.6	0%	0	0	0	0	0/8

Ved nærmere undersøkelse fant vi ut at programmet feilet på grunn av at de blå rørene ble for korte i forhold til størrelsen på bildet. Det er satt en minimumsgrense for hvor lange rørene må være for å kunne regnes som rør. I dette tilfellet ble bildet tatt slik at ROI fylte en for liten del av bildet og dermed ble rørene så korte at de begge kom under minimumsgrensen. Dette ledet programmet inn i en loop hvor det ble forsøkt å koble sammen avkuttede rørdeler som ikke eksisterer.

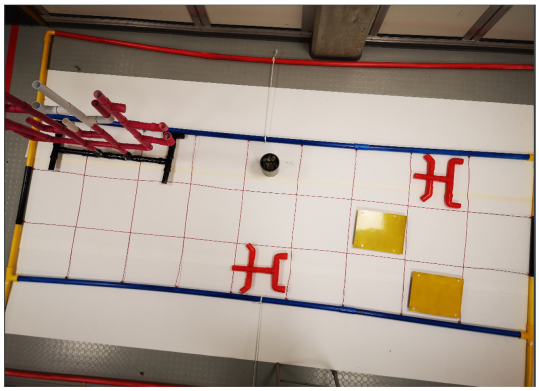
Tidsbruk:

Gjennomsnittstiden for behandling av de 40 testbildene er målt til 154 ms.

Eksempel på gjennomføring:

Denne seksjonen presenterer de stegvise resultatene fra en kartlegging av havbunnen. Nummereringen på stegene er den samme som ble brukt innledningsvis, og gjennom hele kapittelet.

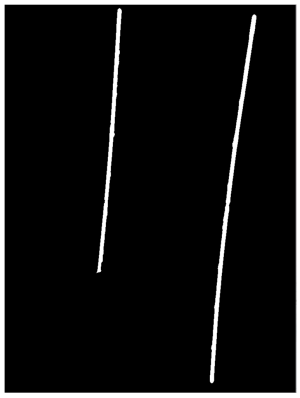
1. Bildet sendes fra operatør.
2. BIldet roteres og skaleres.



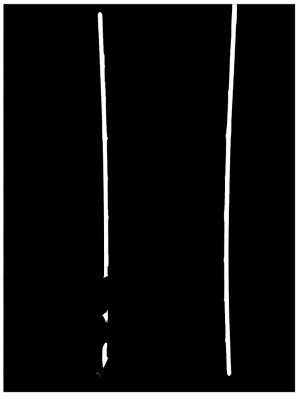
Figur 103: Havbunn steg 1 og 2. Testbilde nr 37

3. Inspeksjon av blå rør:

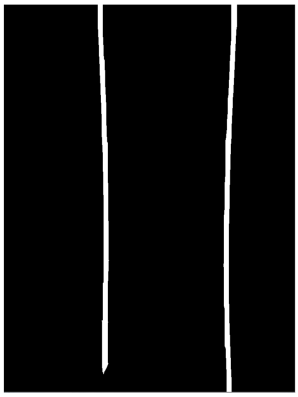
- Finjusterer rotasjon.



- Kobler sammen rørene.

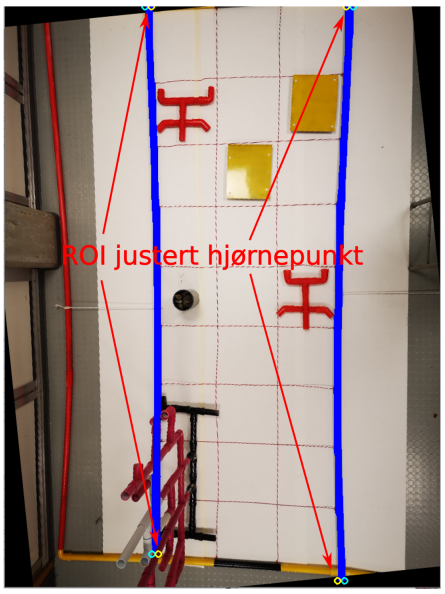


- Rørene ferdig sammekoblet.

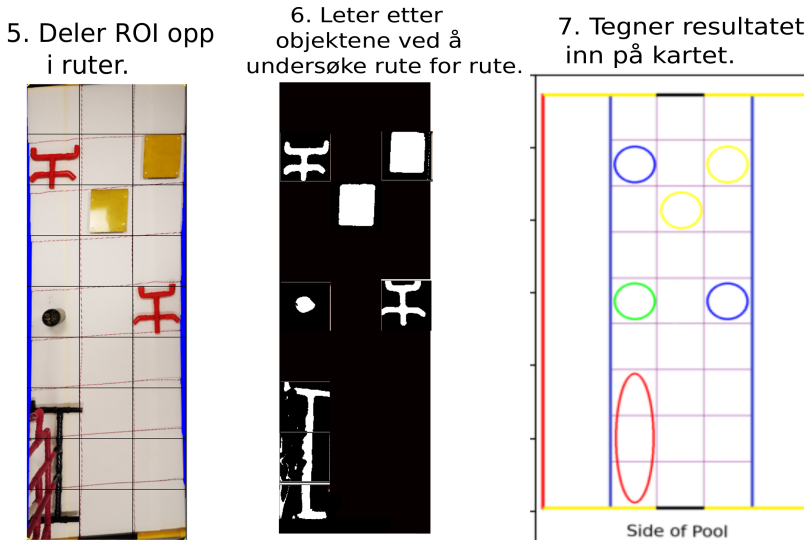


Figur 104: Havbunn steg 3.

4. Definerer ROI ved hjelp av hjørnepunkter fra de blå rørene.
   - Disse punktene justeres noe.
   - De små ringene illustrere hjørnepunktene,
     de gule ringene er de ferdig justerte punktene.
   - ROI plukkes ut med perspektivtransformasjon.



Figur 105: Havbunn steg 4.



Figur 106: Havbunn steg 5,6 og 7. Objektene har nå blitt detektert og markert i henhold til oppgavebeskrivelsen og illustrasjonen i figur 61.

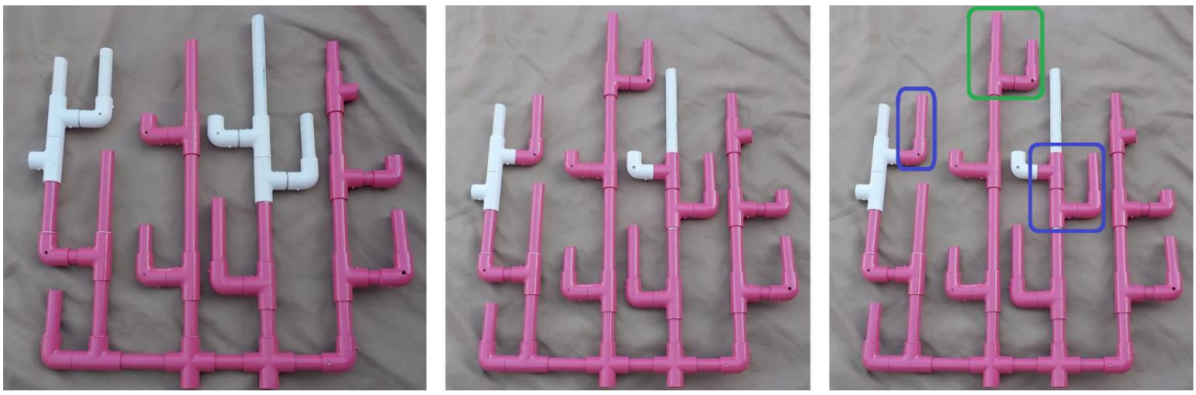
4.3.4 Resultat

Algoritmen håndterer det vi kan forvente at den bør håndtere. Den håndterer rotasjon og forskyvning i alle retninger. Den håndterer også at korallrevet skygger over andre deler av banen. I enkelte spesialtilfeller oppstår det situasjoner hvor algoritmen ikke klarer å fullføre, eller får ett feil resultat. Skulle dette skje under MATE-konkurransen kan vi kjapt ta et bedre oversiktsbilde å gjøre ett nytt forsøk. Løsningen tilfredsstiller kravene/forutsetningen definert av oppgaven og oss. Den fremstår som robust er derfor egnet til bruk i MATE-konkurransen.

5 Helsen til korallrev

5.1 Oppgaven

Den andre oppgaven fra MATE-konkurransen går ut på å undersøke helsen til et korallrev. Som et utgangspunkt for undersøkelsen får vi utdelt ett bilde av korallrevet tatt for ett år siden. Formålet med oppgaven er å markere hvordan korallrevet har endret seg det siste året. For å simulere de forskjellige helsetilstandene brukes fargede PVC rør. Hvitt rør signaliserer at området er bleket og rosa/lilla angir et friskt område. I tillegg til å endre farge kan korallrevet vokse eller dø. Denne endringen representeres ved at deler korallrevet har blitt lagt til, utvidet, fjernet eller redusert. Figur 107 viser et eksempel med gammelt og nytt bilde av korallrevet, hvor endringene er markert.



Figur 107: Bildet til venstre viser korallrev gamle tilstand. Bildet i midten viser korallrevets nåværende tilstand. På bildet til høyre er endringene markert. Bilde hentet fra [15].

MATE-oppgaven beskriver hvordan endring skal markeres. Grønn ring markerer områder som har fått ny utvekst. Blå ring markerer områder som har helbredet bleking. Skadde eller døde områder merkes med gult og blekte områder merkes med rødt.

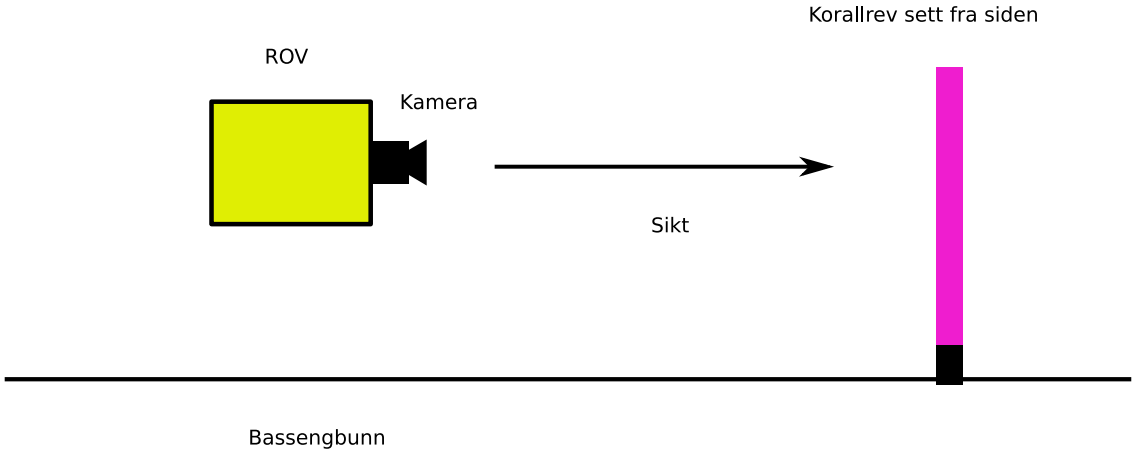
5.2 Generelle forutsetninger og antagelser

Som forutsetning for løsningen legger vi noen antagelser:

- korallrevet kan kun få/miste utvekster, stammen mellom grenene vil ikke vokse.
- Vi behøver ikke markere hvite korallrev-deler som forsvinner/dør.
  Se kommentar på dette punktet i kapittel 9.2.1.

I tillegg til antagelsene rundt selve oppgaven, spesifiseres noen antagelser som ligger til grunn for løsningene.

- En operatør tar bildet manuelt med ROV-en. Dette bildet sendes så inn til et bildebehandlingsprogram som returnerer resultatet og publiserer dette i brukergrensesnittet.
- Hele korallrevet må være på bildet.
- Ingen objekter skal komme foran kamera og blokkerer deler av korallrevet.
- Bildet skal tas rettet normalt mot korallrevet. Det skal være minimalt med forskyvninger og rotasjoner. Hvordan bildet bør tas er illustrert i figur 108.
- Bildet av korallrevet før og malbilder må være tilgjengelige for programmet.

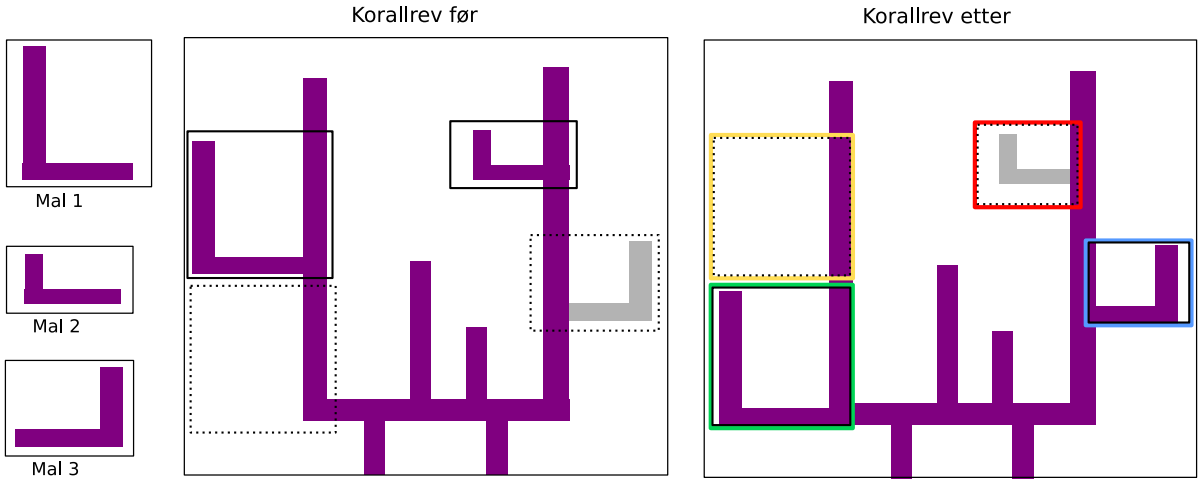


Figur 108: Illustrasjonen viser hvordan bildet av korallrevet bør tas. Korallrevet er plassert stående opp fra havbunnen.

Videre i kapittelet tar vi for oss to av løsningene vi forsøkte å implementere. Først beskrives og testes løsningen basert på malsammenligning. Deretter gjøres det samme med løsningen som er basert på *k-means* og grafteori.

5.3 Løsningsforslag: Malsammenligning

Denne løsningen er basert på malsammenligning. Malsammenligning er beskrevet i kapittel 3.6. Oppgaven går ut på å oppdage endringer i korallrevet. Løsningen benytter sentrale deler av korallrevet som malbilder. Programmet fokuserer deretter på områdene hvor funnene fra malsammenligningen er unike. Med unike menes det at funnet kun eksisterer i ett av bildene. Disse tilfellene undersøkes videre for å klassifisere hva slags endring som har skjedd. Dette blir illustrert i figur 109. I den videre teksten kalles området rundt malene som bokser, eller områder av interesse, "Region Of Interest" (ROI).



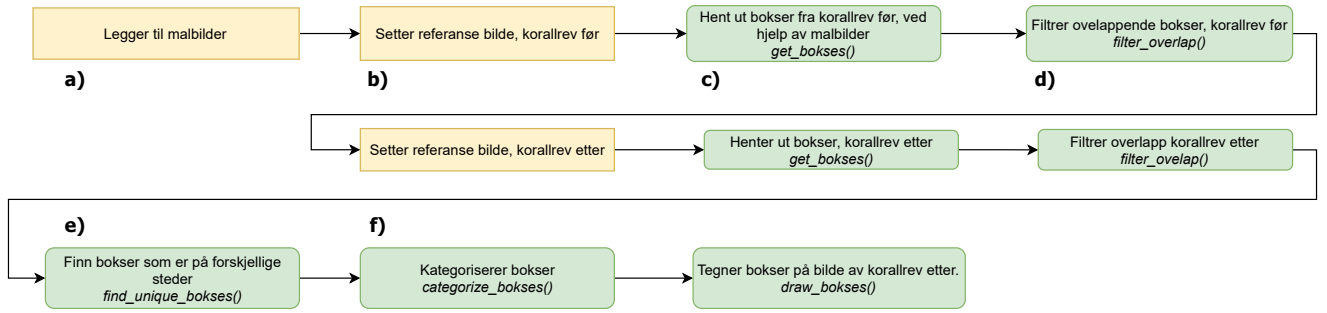
Figur 109: Illustrasjon av løsningen basert på malsammenligning.

Til høyre i figur 109 illustreres eksempler på malbilder ."Korallrev før" viser hvordan korallrevet så ut før, for ett år siden. "Korallrev etter" viser hvordan korallrevet er nå. Ofte referert til som bare "før" og "etter". Det er på dette bilde vi skal markere endringene fra hvordan korallrevet var for ett år siden. Solide bokser betyr mal funnet i gjeldende bilde, stiplede betyr funnet i motpart. De fargede boksene markerer de forskjellige endringene som kan forekomme. Disse ble beskrevet i innledningen av kapittel 5 I resten av kapittelet går vi detaljert gjennom løsningen, før vi tester og dokumenterer løsningens begrensninger.

5.3.1 Detaljert om programmet

Før programmet begynner å analysere bildet, må det behandles. Først finnes konturen av korallrevet ved hjelp av fargemaskering (kapittel 3.1). Dette klippes ut, og skaleres til et 400 x 400 bilde. Dette gjøres for at malbildene skal passe til referansebildet. Metodene brukt til denne typen operasjoner er nevnt i de andre bildebehandlings-kapitlene. Dette er ikke fokus for denne løsningen. En viktig forutsetning for løsningen er at malbildene er klippet ut av tilsvarende bilde med 400 x 400 oppløsning.

I figur 110 er det en detaljert stegvis beskrivelse av hvordan den videre løsningen er bygd opp. Resten av delkapittelet går dypere inn i viktige momenter ved løsningen.



Figur 110: Flytdiagram for malsammenligning hovedprogram.

a) Legger til malbilder. Her tar programmet inn sti til malbildene og bruker klassen "Template" til å lage et malbildeobjekt av et malbilde. En av parameterne i denne er terskelverdi (samme som treffprosent i kapittel 3.6).

b) Setter referansebilde. I dette steget blir referansebildet "før" behandlet og lagt inn som klassen "ObjectTotal".

c) Henter ut bokser ved hjelp av *get_boxes* metoden til "ObjectTotal" klassen. Denne bruker malsammenligning (se kapittel 3.6), og lagrer hvert område som er over ønsket terskelverdi til en klasse "ObjectPart". Finner deretter de forskjellige delene i korallrevet (gren høyre, gren venstre, osv.) og lager bokser rundt dem.

d) Filtrerer ut overlappende bokser med *filter_overlap*. Vi sitter da igjen med et lite sett med bokser per korallrev som beskriver alle delene i sitt respektive korallrev. Dette skjer samtidig som programmet henter ut bokser, slik at *get_boxes* bare returnerer relevante bokser uten overlapp. b), c), og d) gjentas for korallrev "etter"(nåværende tilstand), slik at vi har to sett med bokser som beskriver korallrevet. Ett for referansenbildet, og ett for nåværende tilstand.

e) Boksene fra de to bildene sammenlignes med *find_unique_boxes*. Vi får da returnert de alle bokser som er unike. Altså at de kun eksistere i det ene bildet.

f) Videre sammenliknes de unike boksene fra de to bildene. Vi finner da de eventuelle endringene i korallrevet. Boksene blir kategorisert etter følgende kriterier.
Er det en boks etter, men ikke før, vil det bety at det område er nytt, eller helbredet fra bleking.
Er det en boks i bildet før, men ikke etter, betyr det at den delen er ødelagt eller bleket. Ser på hvor mye hvitt som er i boksen for å finne ut om det er bleking eller ikke.

Når vi finner et område som har høy treffprosent, vil også pikslene som er rett ved siden av ha nesten like høy treffprosent. Dette fører til at vi får mange bokser som tilsynelatende er rett oppå hverandre. Algoritmen vil altså ende opp med mange bokser på samme sted. Dette står det mer om i kapitlet om malsammenligning 3.6. For å fikse dette problemet brukes funksjonen "filter_overlap" som er beskrevet mer på neste side. Denne funksjonen er laget selv, men har mye til felles med den som er beskrevet i artikkelen til Sicara [30].

Filtrer ut overlappende bokser

Det denne funksjonen gjør er å finne ut hvor mye en boks overlapper med en annen. Den gjør det ved å ta utgangspunkt i en boks og sammenligne denne med alle andre bokser. For hver boks vil det bli gitt en verdi som forteller hvor mye overlapp det er, dersom boksen som blir sammenlignet overlapper mer enn en gitt terskel vil den bli filtrert ut. Dette gjentas til programmet står igjen med én boks (den beste) per område.

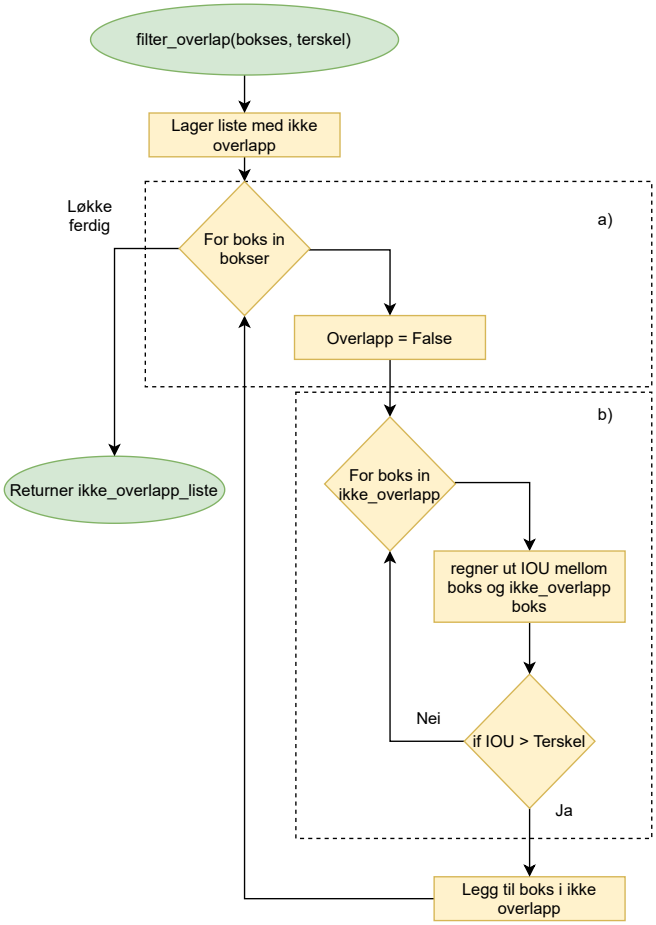
I figur 111 ser vi hvordan funksjonen *filter_overlap* filtrerer vekk overlappende bokser. Først lages en tom liste der programmet samler boksene som står igjen etter filtrering.

Del a) "For boks in bokser" betyr at programmet tar én og én boks fra listen "bokser" som er en inngangsvariabel. Setter flagg at overlapp er usann, grunnen blir forklart senere.

Del b) av figuren viser valget om hvilke bokser som beholdes ut fra overlapp, "Intersection Over Union" (IOU). Denne kan også kalles Jaccard indeksen [72]. Formelen 28 er den normale måten å beregne denne.

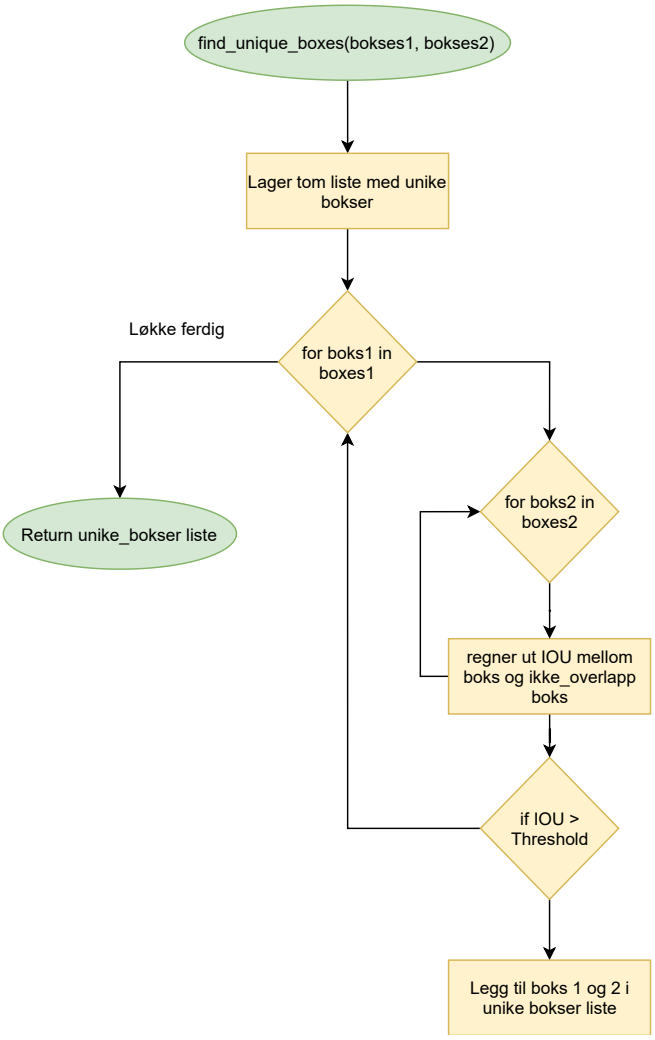
$$J(A,B) = \frac{|A \bigcap B|}{|A \bigcup B|} \tag{28}$$

Når vi beregner IOU brukes en funksjon som er implementert ved hjelp av eksempelet i [71]. Det denne funksjonen gjør er å ta boksenes fire hjørnepunkter og kontrollerer hvor mye de overlapper hverandre. Her gjorde vi litt modifikasjoner slik at den kan ta inn objekter av klassen "ObjectPart".



Figur 111: Flytdiagram av funksjonen for å filtrere overlappende bokser.

Etter at vi har filtrert ut bokser for hver del av korallrevet, sammenligner vi boksene fra bildet før og bildet etter. Her bruker vi prinsippet med overlappende bokser igjen. Denne gangen vil vi sammenligne hver boks før med hver boks etter. Dersom boksene overlapper mindre enn en terskel (ikke overlapper med en annen boks) vil programmet tolke denne boksen som en differanse mellom bildene. Bokser som overlapper før og etter har ingen forandring, og er derfor ikke interessante. De vil bli filtrert vekk ved hjelp av *find_unique_bokses* funksjonen. Flyten til denne er vist i figur 112.

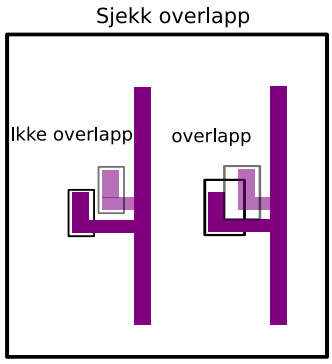


Figur 112: Flytdiagram av finn unike bokser funksjonen.

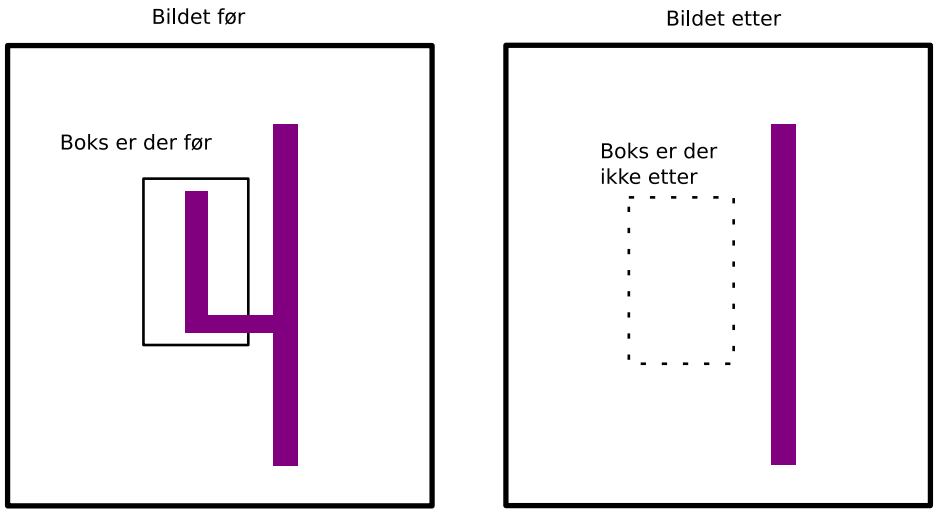
Finn unike bokser

For å finne unike bokser bruker vi nesten samme prinsipp som for å finne overlappende bokser. Forskjellen er at når vi ville finne unike bokser, vil vi ta vare på de boksene som *ikke* overlapper. På figur 114 vil boksen i det venstre bildet bli en unik boks, fordi den er der før, og det er ingen bokser som overlapper etter. Dersom det ikke hadde vært noen forandring, ville vi fått to bokser oppå hverandre her. Det er derfor viktig å ha store nok malbilder.

Dersom malbildene er for små, kan forskyvninger på korallrevets posisjon føre til at bokser som skulle ha vært på samme sted, ikke er det. Dette fenomenet er vist i figur 113.

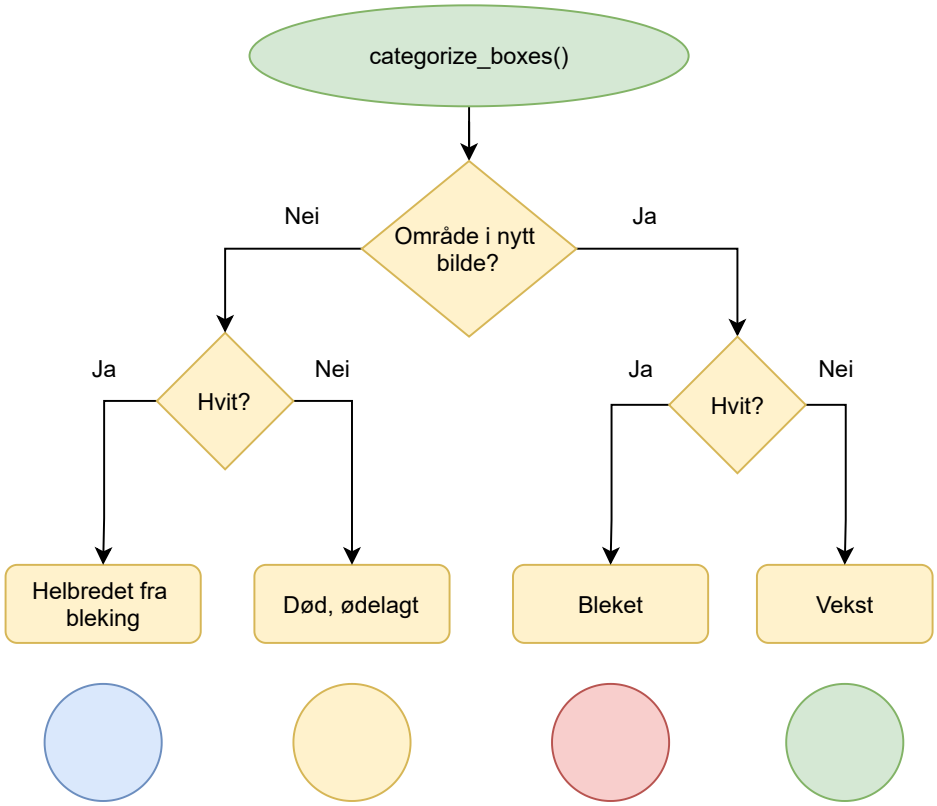


Figur 113: Hva som kan skje hvis malbilder er for små.



Figur 114: Viser et eksempel på en unik boks. En boks er unik om den kun blir funnet i det ene bildet.

Kategoriser bokser
Når programmet har alle områder som inneholder en forandring, kan det sjekk den regionen i bildet der forandringen er oppdaget. Deretter kategoriseres forandringen etter hvordan fargen i området er. Område rundt forandringen vil få en boks rundt seg som forteller hva som har hendt. Figur 115 viser prosessen bak hvilken farge boksen får.



Figur 115: Hvordan programmet ser at en del av korallrevet er forandret ved hjelp av bokser.

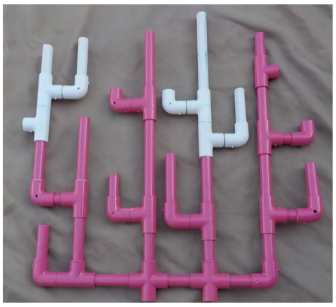
Figur 115 er litt forenklet. Programmet Sjekker først området hvor boksen er (ROI der malsammenligningen oppdaget en forskjell) i bildet før. Deretter sjekkes det samme området i bildet etter. Dersom den gjennomsnittlige intensiteten (hvithet) til pikslene i området før, men ikke etter, er høyere enn en terskelverdi, vil det området bli tolket som helbredet. Er den gjennomsnittlige intensiteten høyere før enn etter, tolkes det som bleket. Finnes området før, men ikke etter, og begge intensitetene er under terskel, regnes det som dødt. Finnes området etter, men ikke før, og begge intensitetene er under terskel, regnes det som utvekst.

Resultatet av hele programmet er bildet av korallrevet etter, med korrekt fargede bokser rundt forandringer.

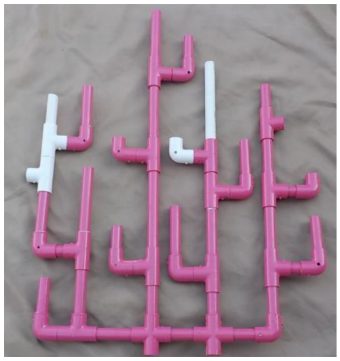
5.3.2 Testing

Her beskrives testing av malsammenligningsalgoritmen. For å teste algoritmen til malsammenligning har vi brukt bildene vist i figur 116. Vi brukte og malbildene vist i figurene 117 og 118.

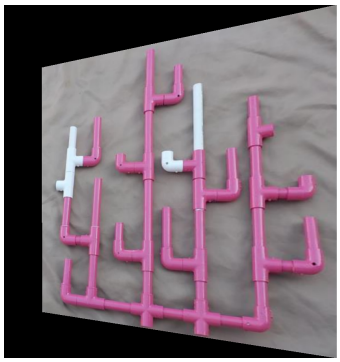
Testbilder



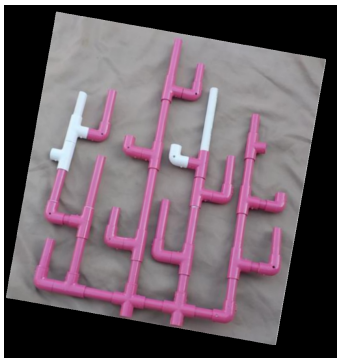
(a) Korallrev før, fra MATE [15]



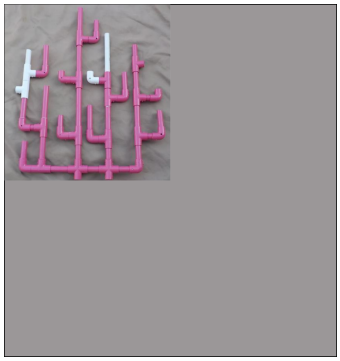
(b) Korallrev etter, fra MATE. Filnavn: korallrev_etter.PNG [15]



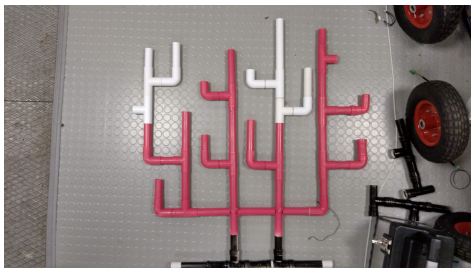
(c) Korallrev etter, MATE, innsynsvinkel.



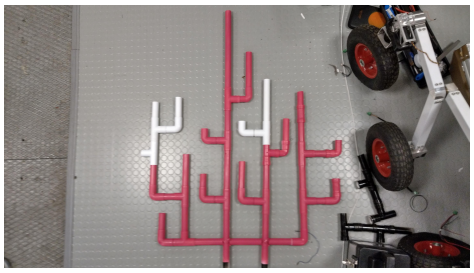
(d) Korallrev etter, MATE, rotert 10°



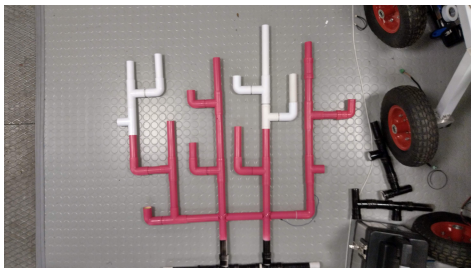
(e) Korallrev etter, MATE, Tatt fra større avstand.



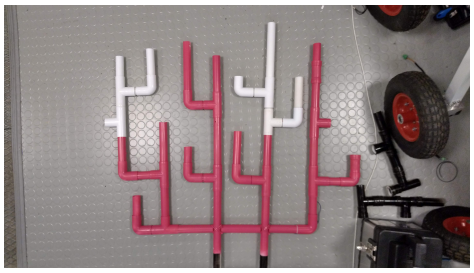
(f) Eget, før, liggende.



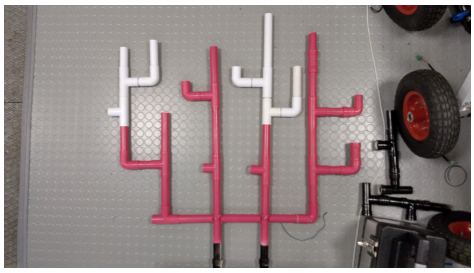
(g) Eget, etter, vekst og helb.



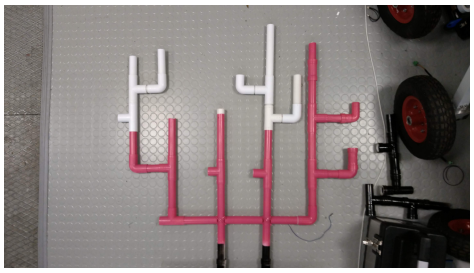
(h) Eget, etter, to feil, en kategori.



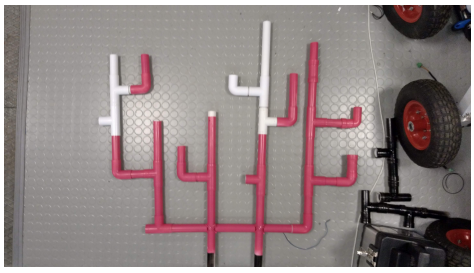
(i) Eget etter, to feil, to kategorier



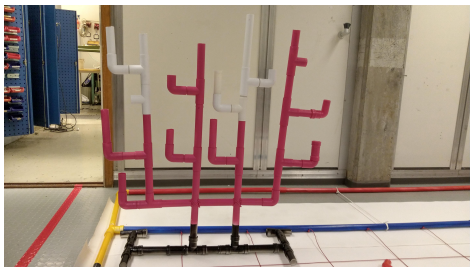
(j) Eget etter, tre feil, en kategori.



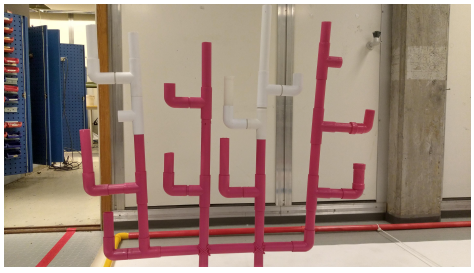
(k) Eget etter, fire feil, en kategori.



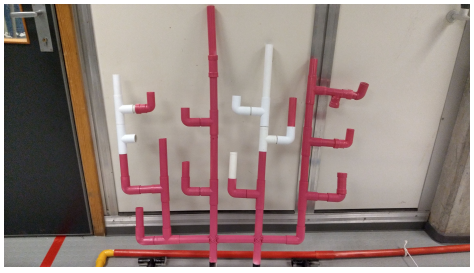
(l) Eget, fire feil, to kategorier



(m) Eget, stående



(n) Eget, tatt fra kortere avstand.



(o) Eget, etter, bakgrunn og vekst.

Figur 116: Testbilder for korallrevets helse.

(a) (b) (c)

Figur 117: Malbilder brukt i test nr. 1 til og med 4.

(a) (b)

Figur 118: Malbilder brukt i test nr. 5 til og med 14.

Forklaring til tabell 1:

- Før - Bilde av korallrevet før.
- Etter - Nåværende situasjon til korallrevet. Bildet som sammenlignes med bildet før.
- Resultat - Hvilken figur som viser testresultatet.

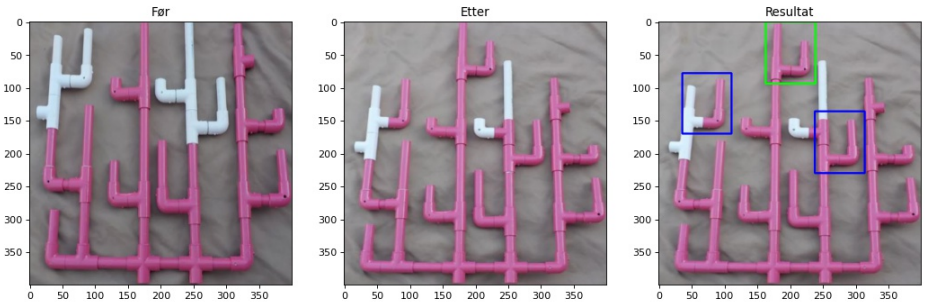
Ønsket treffprosent (se kap. 3.6), test en til fire: [0.95, 0.7, 0.7], for maler i figur 117
Ønsket treffprosent, test fem til fjorten: [0.8, 0.8], for maler i figur 118

Tabell 1 viser hvilke bilder som ble sammenlignet, og hvilken figur resultatbildet er. Neste del av testing brukes bilder av korallrevet som er laget av oss, og tatt med et mobilkamera. Forskjellene med et mobilkamera i forhold til bilder som er klippet ut fra ett dokument er at oppløsningen er mye høyere. Dette trenger ikke være et problem, vi kan klippe ut malbilder av et bilde med samme oppløsning. Tykkelsen på linjene til boksene vil derimot bli mye mindre, siden de er en fast pikselbredde. Vi har derfor valgt å redusere oppløsningen til bilder som skal sammenlignes til en fast verdi på 400 x 400. Dette skjer internt i programmet.

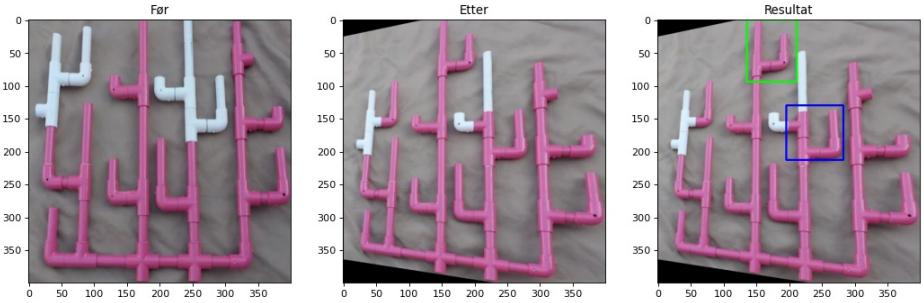
Figurene 119 til 132 viser resultatene.

Test nr.	Før	Etter	Resultat figur
1	Figur 116a	Figur 116b	Testbilde119
2	Figur 116a	Figur 116c	Testbilde120
3	Figur 116a	Figur 116d	Testbilde121
4	Figur 116a	Figur 116e	Testbilde122
5	Figur 116f	Figur 116g	Figur 123
6	Figur 116f	Figur 116h	Figur 124
7	Figur 116f	Figur 116i	Figur 125
8	Figur 116f	Figur 116j	Figur 126
9	Figur 116f	Figur 116k	Figur 127
10	Figur 116f	Figur 116l	Figur 128
11	Figur 116f	Figur 116m	Figur 129
12	Figur 116f	Figur 116n	Figur 130
13	Figur 116f	Figur 116o	Figur 131
14	Figur 116f	Figur 116o	Figur 132

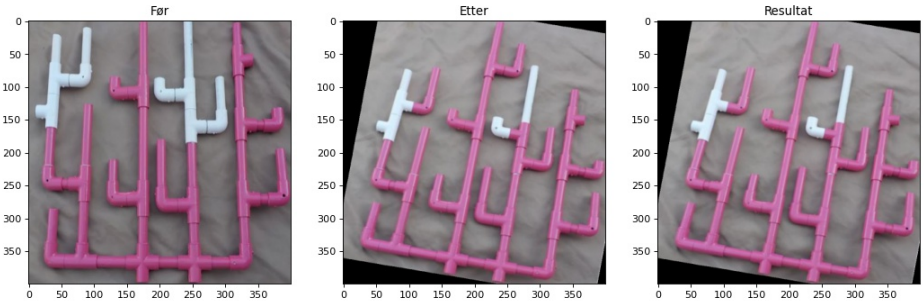
Tabell 1: Testresultater av malsammenligning.



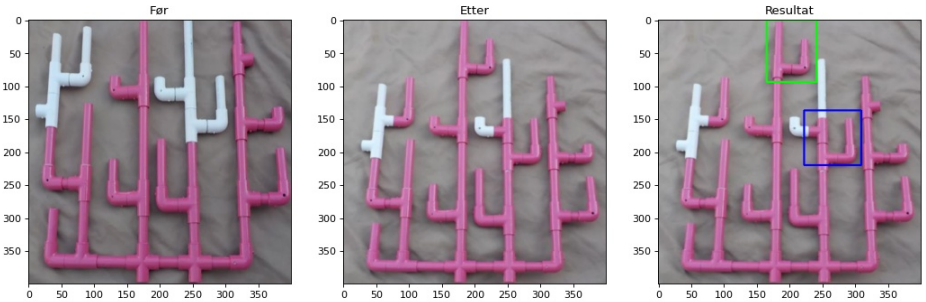
Figur 119: Resultat test 1. Basis fra MATE. Ingen modifikasjoner på bildet, og den gir helt riktig resultat.



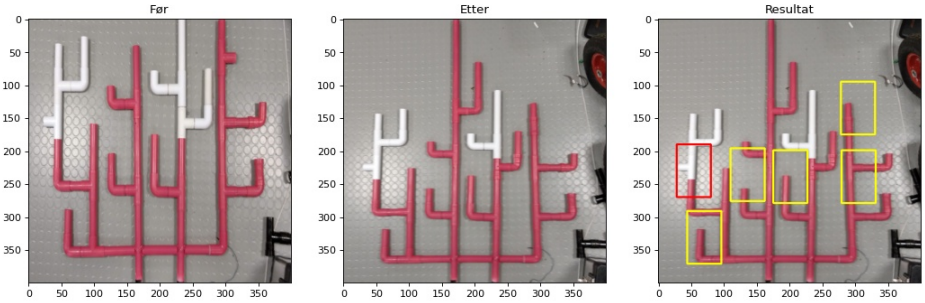
Figur 120: Resultat test 2. Her er innsynsvinkel endret. Ser at programmet ikke finner forskjellen der hvor bildet er skvist, til venstre i etter.



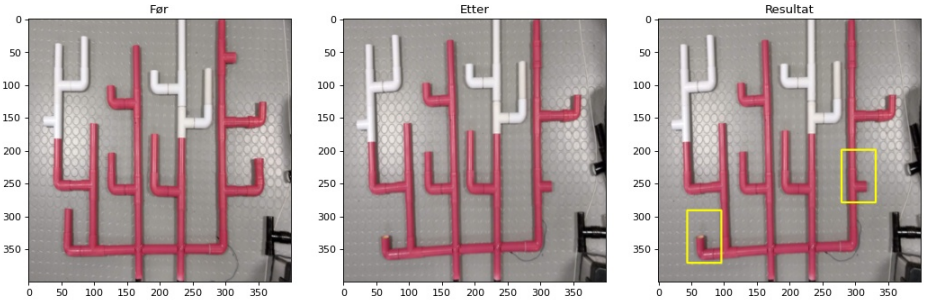
Figur 121: Resultat test 3. Her finner malsammenligning ingenting, på grunn av at ingen av malbildene er roterte.



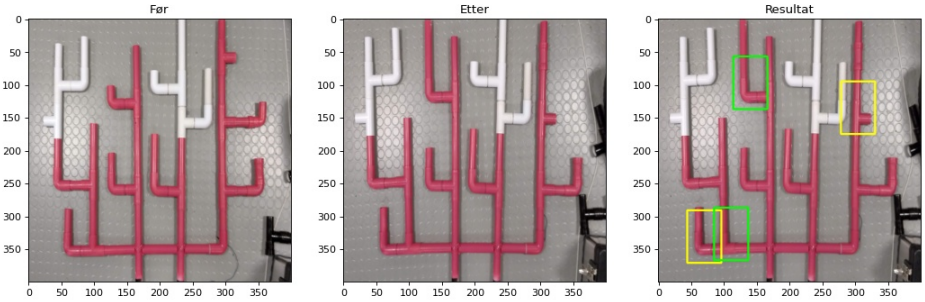
Figur 122: Resultat test 5. Her har vi brukt et zoomet inn bilde etter. Fordi programmet klipper korallrevet ut, ser vi ikke dette. Grunnen til at den ikke fant den siste forskjellen er at ønsket treffprosent var for høy.



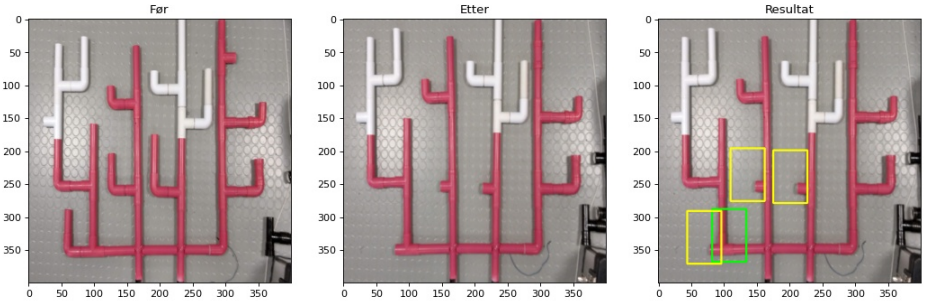
Figur 123: Resultat test 6. Her er alle boksene falske positiver, og programmet fant ikke utveksten på den midterste grenen. Dette på grunn av at sideforholdet ble for forskjellig.



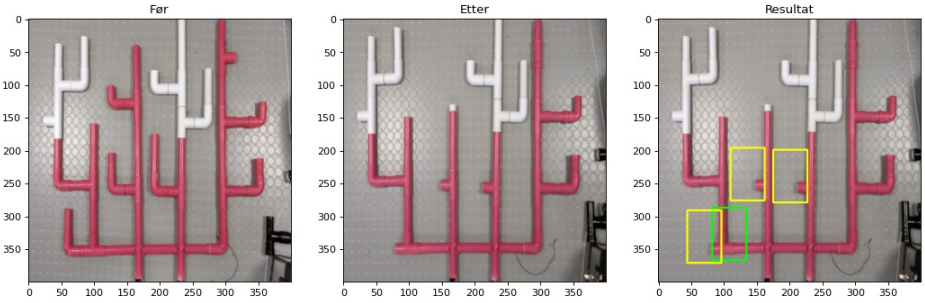
Figur 124: Resultat test 7. Her finner programmet korrekte forskjeller, nemlig at to grener har dødd.



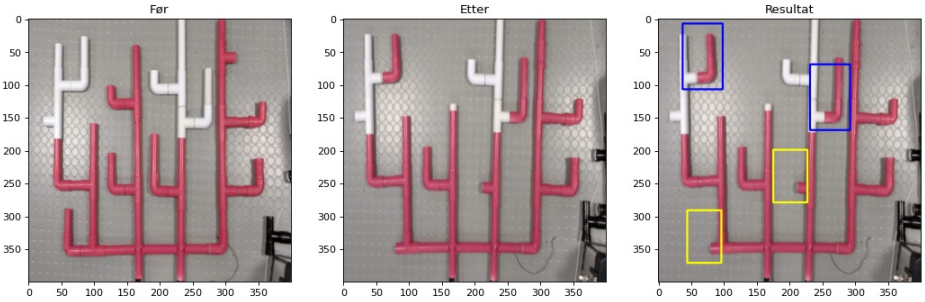
Figur 125: Resultat test 8. Her finner programmet to falske positiver nede til venstre. Ønsket treffprosenten er for lav. Det finner og to korrekte forskjeller oppe og til høyre, en utvekst og en død gren.



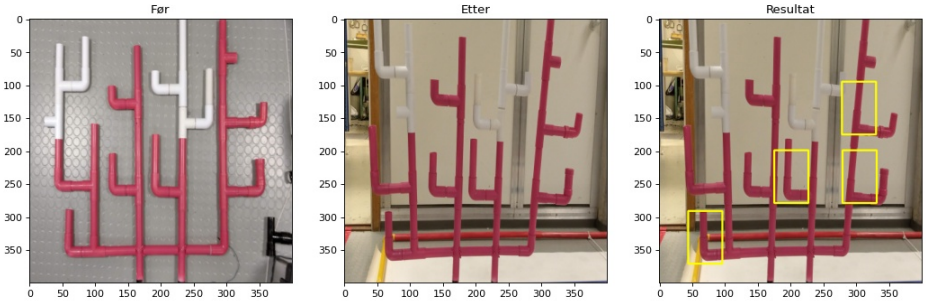
Figur 126: Resultat test 9. Her finner programmet alle de riktige forskjellene, men og en falsk positiv. Den falske positive kommer av at ønsket treffprosenten er for lav.



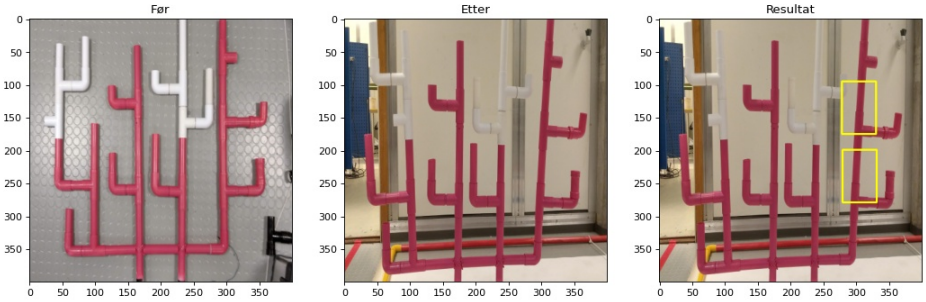
Figur 127: Resultat test 10. Samme som test 9, men finner ikke den store grenen oppe i midten. Hadde man hatt en annen mal, eller lavere ønsket treffprosent, ville denne blitt funnet.



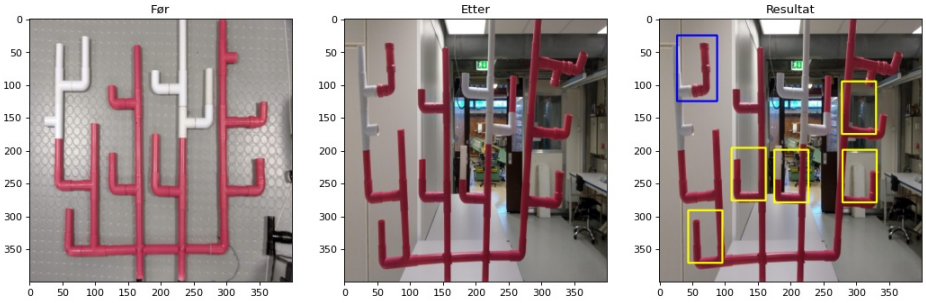
Figur 128: Resultat test 11. Her finner programmet fire forskjeller og markerer dem som korrekte kategorier. Her finner den heller ikke den store grenen oppe i midten.



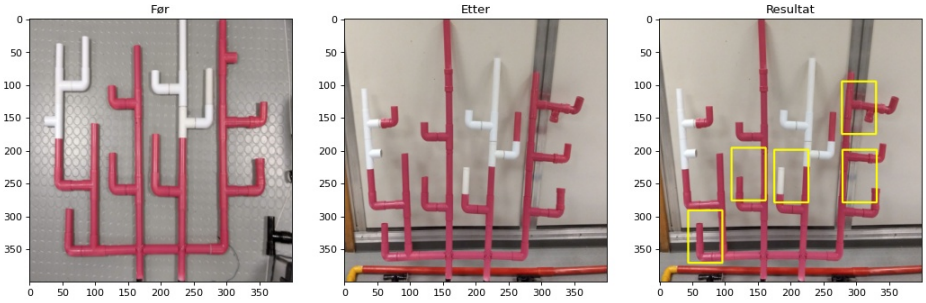
Figur 129: Resultat test 12. Her finner programmet bare falske positiver. Korallrevet før og etter er for ulike i form for at det skal bli noe bra resultat. Vi kan se at grenene (etter) på sidene er bøyde til hver sin kant.



Figur 130: Resultat test 13. Samme som test 12.



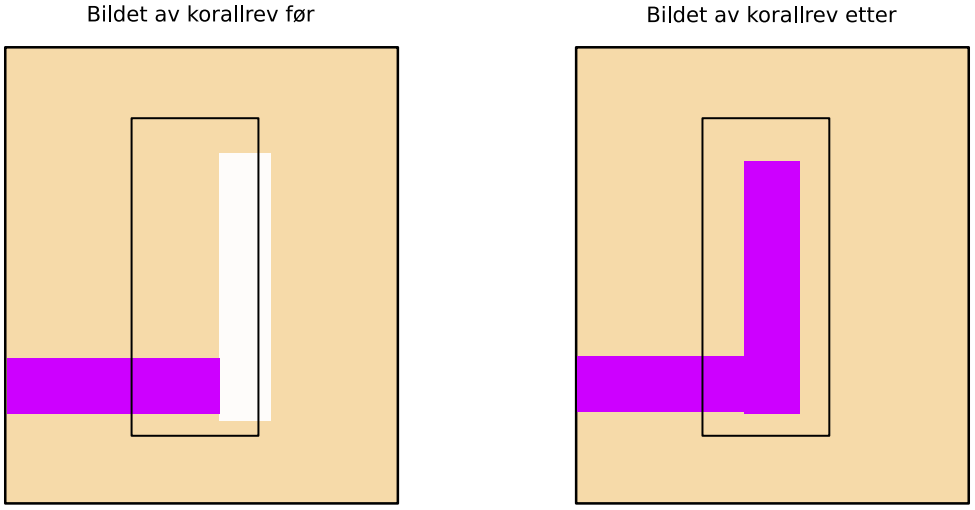
Figur 131: Resultat test 14. Samme som 12. Her fant programmet faktisk en korrekt forskjell oppe til venstre.



Figur 132: Resultat test 15. Samme som test seks, forskjellige sideforhold på korallrevene.

På figur 122 ser vi bildet etter at det er zoomet inn. På resultat 6 til 15 brukes malbildene vist i figur 118. Dette er på grunn av at vi brukte tynnere rør. Det hadde fungert å bruke de samme malbildene for begge korallrevene, men da trengs en lavere treffprosent terskel, og vi risikerer flere ukorrekt merkede områder.

For å få bedre resultater kan det hjelpe å ha lavere terskel verdi. Dette kan derimot og føre til falske positiver (endringer der det ikke er noen). Dette er en av ulempene med malsammenligning. Parametere som fungerer på noen bilder, og noen typer feil, vil ikke nødvendigvis virke like bra på andre.



(a) Her er den samme boksen som på bildet etter.

(b) Her finner programmet en boks.

Figur 133: Figur som viser hva som kan skje dersom det ene bildet av korallrevet har et annet perspektiv.

Kommentarer til testresultatene:
Vi kan se ut fra figur 133 at det er en utfordring å ikke vite hvor grenene er plassert i bildet. Dersom vi hadde visst om et fellespunkt i begge bildene, og laget en boks rundt det, hadde programmet taklet slike utfordringer.

Fra test 15 ser vi at det å ha gode malbilder er viktig. For eksempel kan vi ha en mal av det lilla hjørnet oppe til venstre i bildet etter. Men det er ulempen med malsammenligning, vi vet ikke hvilke malbilder som trengs for å oppdage de rette forandringene, derfor er det lurt å ha mange forskjellige malbilder.

5.3.3 Resultat

Fra testresultatene seks til elleve, ser vi at malsammenligning fungerer bra i tilfeller hvor korallrevet ikke får nye utvekster, eller mister store grener. Det vil føre til at bildet får et annet sideforhold (blir bredere/smalere). Vi kan og si at denne løsningen ikke er veldig robust, det skal lite forandring til i bildet før programmet gir helt gale resultater.

Vi ser at forvrengninger (innsynsvinkel, test to) og rotasjon (test tre) av bildet fører til problemer. En måte å fikse rotering på er å ha malbilder som også er roterte. Vi valgte å ikke ta med dette fordi vi forventer at ROV-en klarer å holde seg i rett vinkel.

For å få god gjenkjenning av hvite områder er det viktig å ha bilder som er lite forvrengt i forhold til hverandre. Årsaken er at et område som er av interesse i det en bildet, kan overlappe dårlig i det andre bildets hvite område. Eksempel på dette er vist i figur 133.

En svakhet i hvitgjennkjenningsfunksjonen er at det er lite som skal til for at et område regnes som hvit.

Forbedringspotensialer:

- Bedre fjerning av støy for å få bedre malbilder. For eksempel 117c er dårlig mens 118b er bra.

For å forbedre testresultatene kunne vi ha funnet flere og mer representative malbilder. Samt finjustert treffprosentene til malbildene ved hjelp av mer omfattende testing.

Konklusjon

Det var kun i eksempel 1 og 7 at programmet klarte å identifisere alle endringene korrekt. I alle de andre testene klarte ikke programmet å identifisere alle endringene, eller det fant falske endringer.

Programmet løser altså oppgavene når alle forutsetningene ligger på plass. Dette innebærer gode malbilder med korrekt innstilt treffprosent samt perfekt tatt bilde. Under slike omstendigheter kan sammenligningen bli vellykket. Testingen mangler godt optimaliserte malbilder.

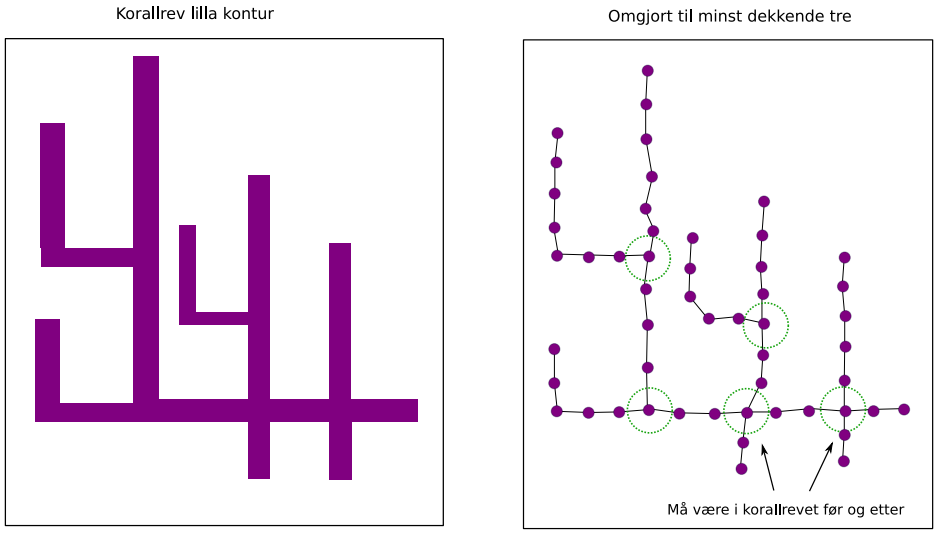
Vi konkluderer dermed med at programmet løser oppgaven, men er svært lite robust og krever mye finjustering. Programmet vil i sin nåværende tilstand ikke garantere full uttelling i MATE-konkurransen. Til dette er den altfor lite robust og treffsikker.

5.4 Løsningsforslag: *k-means* og grafteori

Denne metoden baserer seg på å dele hvert korallrev (før og etter) inn i nøkkelpunkter og grener. Et nøkkelpunkt er definert som et punkt som har enten én, eller flere enn to nabo(er). En gren er definert som et sett med punkter og et nøkkelpunkt i hver ende. Dette gjør at vi kan sammenligne hver gren før, med den samme grenen etter. For å kunne definere posisjonene i korallrevet setter vi et krav: Hvert korallrev må ha to nøkkelpunkter med fire naboer. Disse punktene er illustrert i figur 134. Programmet vil da ha referansepunkter til å navngi grenene ut fra. Dette spesialiserer programmet til akkurat denne typen korallrev.

For å dele korallrevet inn i punkter har vi benyttet oss av gruppering beskrevet i kapittel 3.7. Deretter, for å binde disse punktene sammen, brukte vi nærmeste nabo og grafteori som er beskrevet i henholdsvis kapittel 3.8 og 3.9.

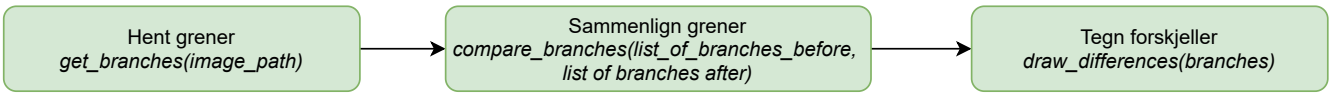
Figur 134 illustrerer fremgangsmåten. Her vises den lilla konturen av korallrevet omgjort til punkter, og bundet sammen ved å finne korteste vei mellom dem.



Figur 134: Figuren illustrerer bruken av *k-means* og grafteori til å definere grener og nøkkelpunkter. Bare nøkkelpunkter med minst tre naboer er merket.

Som nevnt kategoriserer programmet hver gren ut fra de to fireveis kryssene (nøkkelpunktene) (se figur 134). Disse referansene gjør det mulig å sammenligne grenene i korallrevet fra referansebildet og den nåværende situasjonen.

Figur 135 viser flytdiagrammet til løsningen med *k-means* og grafteori.



Figur 135: Flytdiagram til "main" programmet.

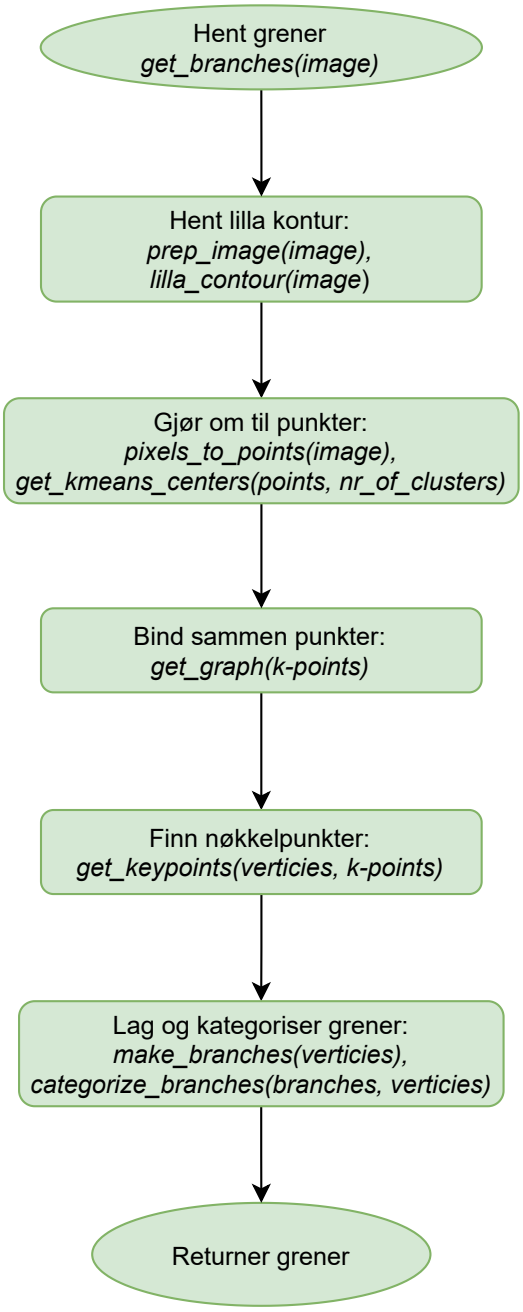
En litt mer detaljert beskrivelse av figur 135:

- Det første programmet gjør er å hente ut grener fra bildet av korallrevet før og etter. For at programmet skal klare det brukes en funksjon som ved hjelp av maskering (kap. 3.1) og kontur (kap. 3.3) forenkler bildet av korallrevet til en lilla kontur.
- Etter at korallrevene er delt inn i grener kan programmet sammenligne lengdene til grenene, og oppdage om noen er vekke.
- Til slutt vil programmet tegne bokser med korrekte farger rundt grener som er forskjellige.

I den videre teksten kikker vi nærmere på funksjonene nevnt i figur 135.

Kommentar:
Funksjonen for å tegne inn endringene har ikke blitt fullført. Denne blir derfor ikke beskrevet.

5.4.1 Detaljert om programmet



Figur 136: Flyt av "hent grenerfunksjon.

Hente grener:
Figur 136 viser grovt hvordan programmet benytter seg av *k-means* og finner grener i hvert korallrev.

1. Her hentes selve korallrevet ut av bildet. Til denne oppgaven har vi laget en funksjon som benytter seg av fargemaskering av lillafargen. Funksjonen *prep_image* tar inn et bilde av et korallrev, klipper ut korallrevet og skalerer ned oppløsningen på bildet.

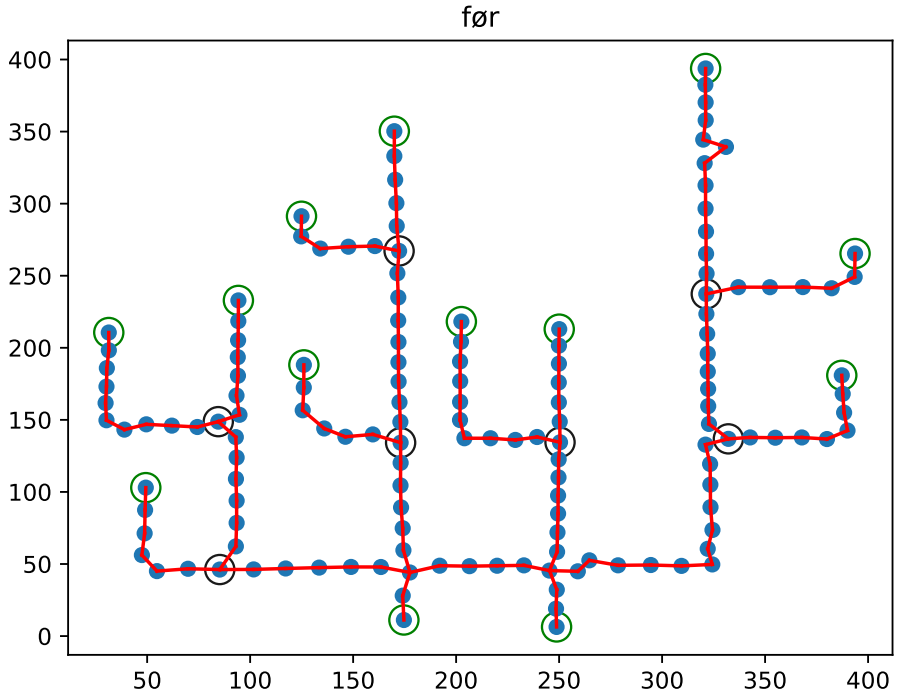
2. Bruker så *k-means* beskrevet i kapittel 3.7 til å redusere antall punkter. Korallrevet får da ikke mer enn ett punkt i bredden. Dette gir et godt grunnlag til å finne nøkkelpunktene (kryss-/forgrenings-/endepunkt). Funksjonen *get_kmeans_centers* returnerer punktene som brukes videre i programmet.

3. Bruker grafteori for å binde sammen punktene (se kapittel 3.9). Vekten mellom punktene bestemmes av nærmeste nabo beskrevet i kapittel 3.8. Manhattan-metoden benyttes. I grafteori kalles punkter for noder, men i programmet er det to forskjellige ting. Punkter er inngang og utgang til *k-means*, mens noder returneres av *get_graph* funksjonen. En node (i programmet) er definert som en egen klasse, som inneholder punktet og dets naboer.

4. Deretter brukes nodene til å finne nøkkelpunkter. Nøkkelpunktene er nodene med en, tre eller fire naboer. En nabo betyr et endepunkt, tre et forgreiningspunkt, og fire et referansepunkt.

5. Til slutt defineres og navngis grenene ved hjelp av nøkkelpunktene definert i steg 4.

Etter at programmet har brukt *get_branches* vist i figur 136 vil det ende opp med grener som er bundet sammen via nøkkelpunkter. Et eksempel på dette er vist i figur 137.



Figur 137: Figuren viser resultat av hent grener funksjonen. Grønne ringer er punkter med bare én nabo, svarte ringer har tre naboer. Nøkkelpunktene med fire naboer (referansepunktene) er ikke markert.

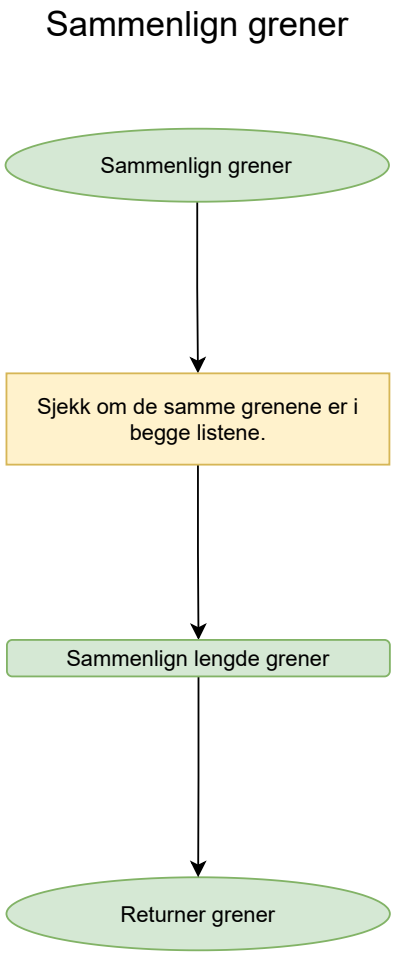
Kommentar:
Ved å gjøre dette reduserer vi datagrunnlagets kompleksitet. Dette muliggjør enklere behandling og undersøkelse av korallrevet

Sammenligne grener:
I denne funksjonen sammenligner programmet grener med likt navn i referansebildet og bildet med nåværende tilstand(før og etter). Navnene ble bestemt ut fra nøkkelpunktene i *get_branches*-funksjonen.

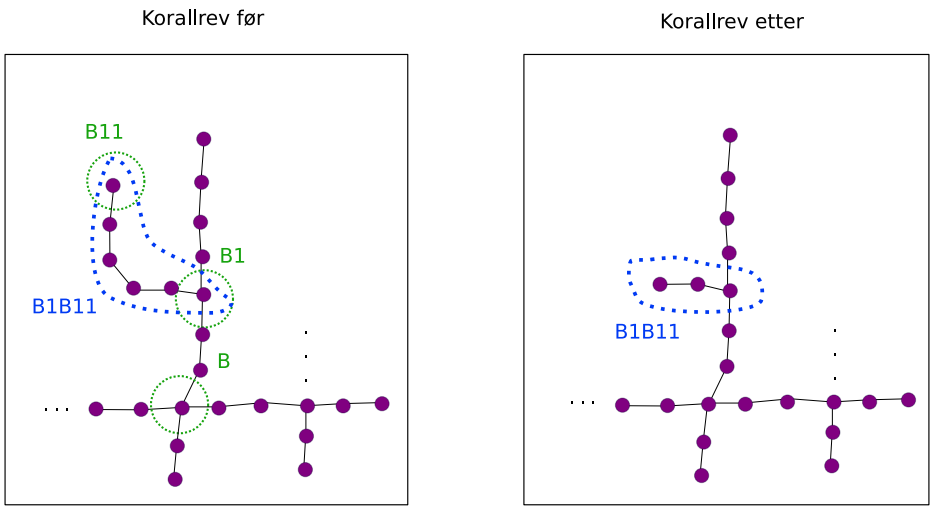
For å sjekke at begge korallrevene har grener med like navn, opprettes det to lister. Én med grener fra korallrevet før, og én med grener fra korallrevet etter. Dersom en gren er i en liste, men ikke i den andre, vil den bli lagt til som en endring (endring i korallrevet).

Lengden til hver gren beregnes ved å finne summen av avstanden mellom hvert punkt. Deretter sammenlignes lengdene til alle grener som har samme navn. På denne måten finner vi grener som har krympet eller vokst. Disse tilfellene samles også opp i listen med grener som beskriver endringer. En liste med disse grenene returneres.

Et eksempel på gren som endres og sammenlignes er vist i figur 139. De grønne sirklene er nøkkelpunkter. Det er flere nøkkelpunkter, men de er ikke tatt med i dette eksempelet. Den blå sirkelen viser hvilken gren som blir sjekket. Vi ser at grenen får et navn ut fra hvilke nøkkelpunkter som er i hver ende.



Figur 138: Grovt flytskjema av "sammenlign grener" funksjonen.



Figur 139: Figuren viser eksempel på når en del av en gren blir ødelagt eller bleket. Grønne sirkler er nøkkelpunkter, blå er gren.

5.4.2 Testing

For å teste denne metoden brukte vi de samme bildene som ble brukt ved malsammenligning, vist i figur 116. I henhold til MATE sin oppgavebeskrivelsen kan korallrevet under konkurransen ha fra to til fire forskjeller, men ikke mer enn to kategorier. Vi har derfor bygget vårt eget korallrev og tatt bilde av dette med diverse forskjeller og kategorier.

Resultatene er vist i tabell 2.

Bilde før	Bilde etter	Antall forskjeller funnet av program
EG_mate_for 116f	EG_mate_etter_uten_v_feil 116g	13
EG_mate_for 116f	EG_to_en 116h	6
korallrev_for 116a	korallrev_etter_mate 116b	15

Tabell 2: Tabell av resultat av k-means testing.

Funn av nøkkelpunkt:
På grunn av at programmet ikke fant nøkkelpunktene i bunnen på en del av bildene, fikk vi ikke testet alle. Grunnen til at programmet ikke fant dem kan være at vi valgte for mange, eller for få, grupper (altså parameter til *k-means* funksjonen), eller at filtrering av uønskede punkter tok vekk for mange.

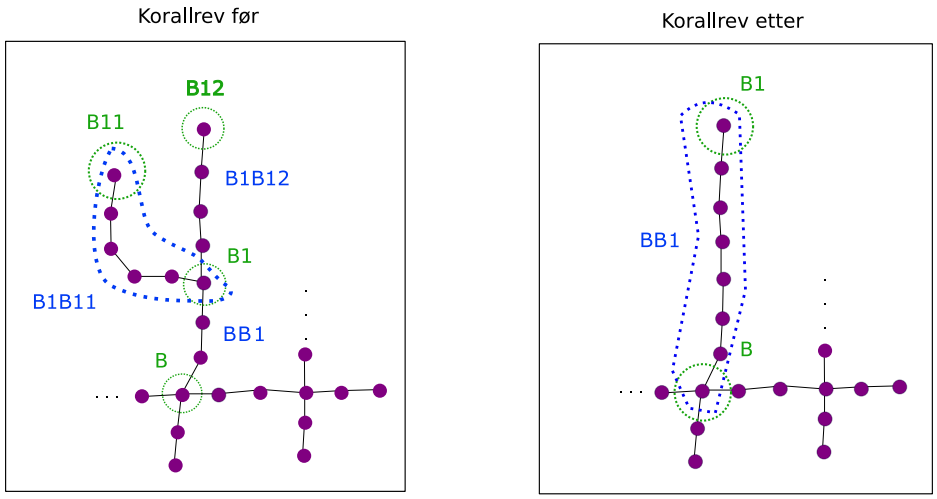
En annen årsak til at vi ikke finner nøkkelpunktene(referansepunktene) er vist i figur 140b. Dette kan hende ved punkter som skal ha fire naboer, men ender opp med at to punkter ved siden av hverandre har tre naboer hver. Oppe til høyre i figur 145 illustreres dette igjen.

(a) (b)

Figur 140: Problem som kan oppstå ved dårlig plasserte punkter. a) viser ett treerpunkt som er klumsete sammenkoblet. Dette kan stor følger for de videre grenlengdene. b) viser ett firepunkt som har blitt til to treerpunkt.

Den nåværende algoritmen vil filtrere ut ett av treerpunktene figur 140b. Egentlig skulle algoritmen gjort to nærliggende treerpunkter om til et nøkkelpunkt med fire naboer(referansepunkt).

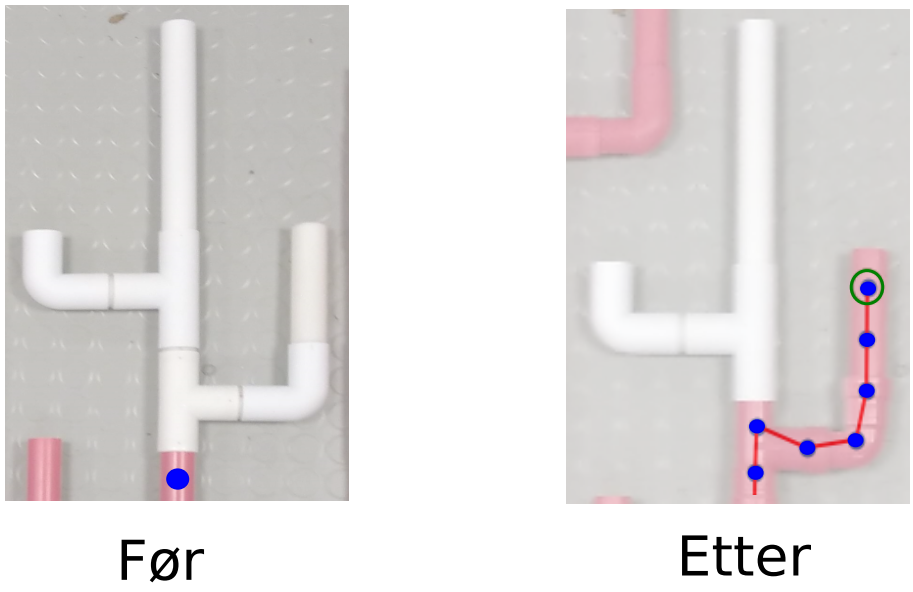
Nye/manglende grener:
En Hovedgrunn til at programmet under testing finner mange ekstra forskjeller, er problemet vist i figur 141. Dersom én gren forsvinner, vil det påvirke alle grener den var festet til. De grenene den var festet til vil også forsvinne, eller blir tolket som andre grener.



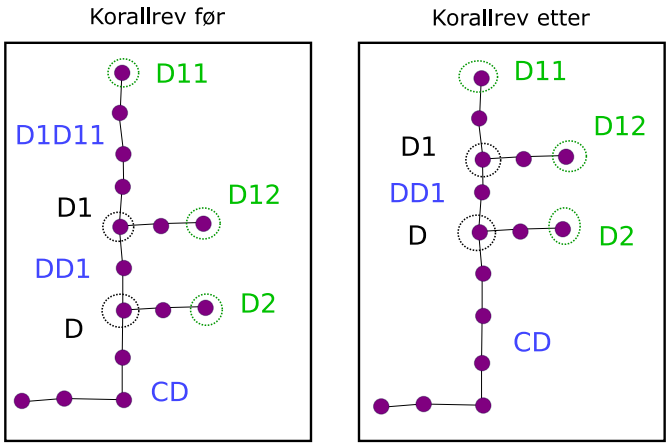
Figur 141: Figuren viser eksempel på når en hel gren blir ødelagt eller bleket. Grønne sirkler er nøkkelpunkter, blå er gren. Ikke alle grener har blå ring for klarhet.

Man ser ut fra figur 141 at dersom en gren forsvinner, vil programmet tolke det som at to grener forsvinner (B1B11 og B1B12), og én gren blir mye lengre enn før (BB1). En annen måte å holde styr på grenen hadde vært å kategorisere de grenene med grener på, som hele grener. For eksempel: B til B12 til venstre i figur 141, hadde blitt én gren. Programmet vil da få et mer korrekt resultat dersom mindre grener forsvinner.

Figur 142 viser at en ny, eller helbredet, gren tolkes som en forlengelse av en eksisterende gren. Dette er ikke gunstig. Vi vil at slike utstikkere skal tolkes som nye grener. Men slik programmet virket under testing ville denne grenen allikevel blitt tolket som en feil, på grunn av at den er lengre enn den var før.



Figur 142: Figuren viser eksempel på når en del av en gren blir helbredet fra blekning.

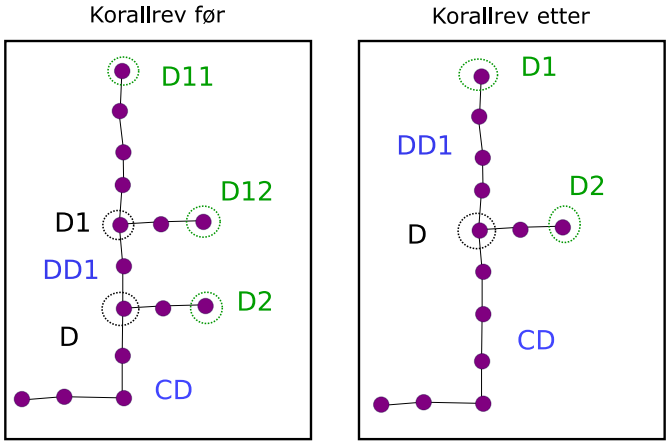


Figur 143: Figuren viser hva som skjer hvis en gren vokser ut over en annen gren, mens den grenen som var under forsvinner. Med andre ord: Gren over og under bytter plass.

En måte å løse dette på er å sammenligne alle like grener sin størrelse. Da Kan programmet peke ut de grenen som er avviket. Men bare å gjøre det er ikke nok til å beskrive *hva* som faktisk har skjedd.

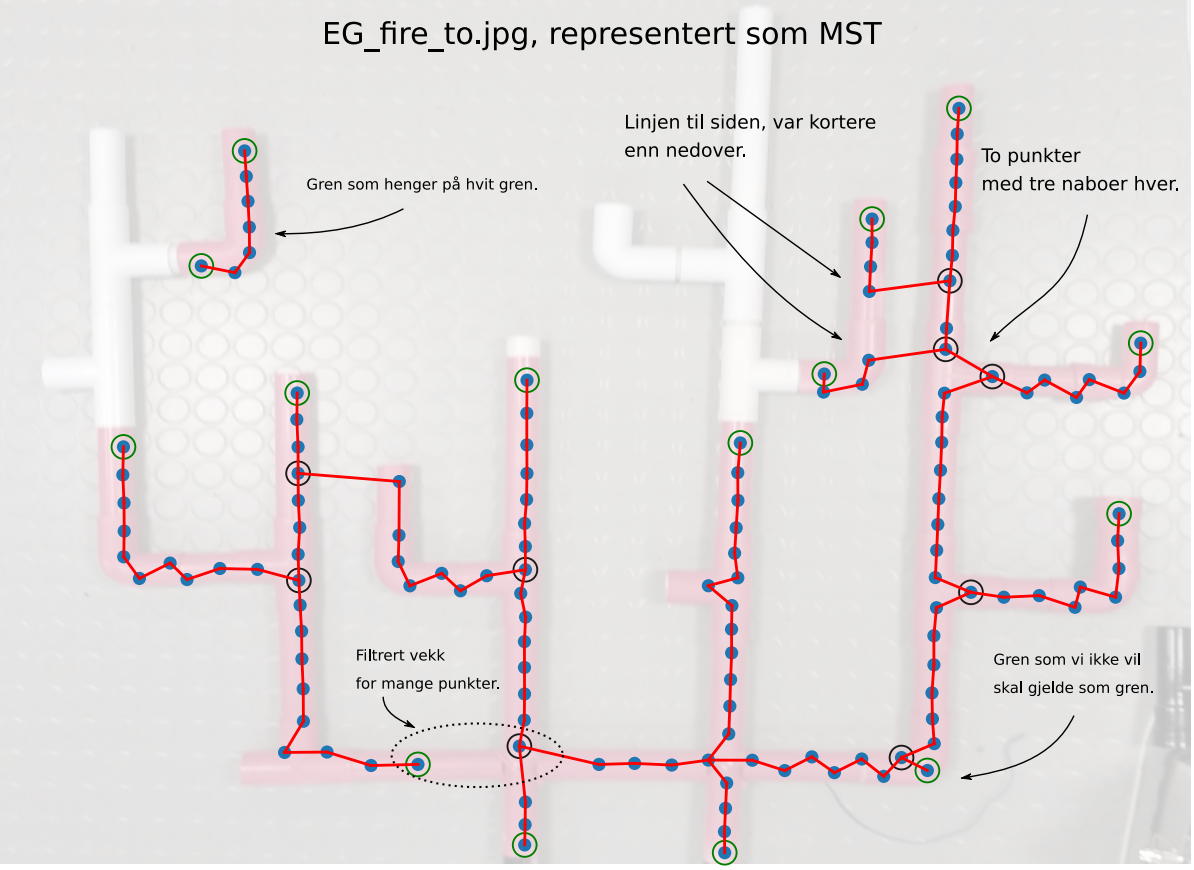
Figur 144 viser hvordan en gren som forsvinner vil føre til at programmet bytter om på hvilke grener som er hvilke. Gren DD1 går fra å være mellom to horisontale grener til å være en gren på topp.

Fordelen med å merke hver hovedgren istedenfor, er at vi løser problemet vist i figur 144. Dersom hver gren i stammen også er merket, vil vi ihvertfall være istand til å finne ut at det er noe galt på stammen. Da blir den nederste grenen i stammen lengre.



Figur 144: Figuren viser hva som skjer hvis en gren vokser ut over en annen gren, mens den grenen som var under forsvinner. Med andre ord: Gren over og under bytter plass.

Typiske feil/utfordringer med løsningen:
Figur 145 viser en del av de feilene som kan skje ved bruk av *k-means*. Bildet fra figur 116l er utgangspunktet til forsøket.



Figur 145: Figur av mesteparten av det som kan gå galt ved bruk av *k-means*.

Det var mange feil som kunne oppstå ved bruk av *k-means*. De fleste av feilene er vist i figur 145. For å forklare feilene er det laget en liste.

- Oppe til venstre, lilla gren på hvit gren. Slik Boruvkas algoritme virker (se kapittel 3.9 skulle ikke dette fenomenet vært mulig. Det som har skjedd er at Boruvkas algoritme har låst seg på de to siste trærne. Det som skulle ha hendt, var at det gikk en strek fra denne grenen ned til korallrevet. Dette ville ikke gitt riktig representasjon av korallrevet.
- Nede til venstre, filtrert vekk for mange punkter. Her har fenomenet vist i figur 140 skjedd. Programmet har deretter filtrert vekk for mange punkter. Dette har og ført til brofenomenet rett ovenfor i figuren.
- Oppe til høyre, Linjer til siden. Her har programmet og filtrert vekk for mange punkter, som har ført til at det var kortere til punktet på en annen gren, enn det som hadde vært naturlig i forhold til korallrevet.
- Nede til høyre, en utstikker som ikke skulle vært der.

Mange av feilene kommer av at *k-means* ikke nødvendigvis legger punkter på rette linjer, og at det er for mange punkter noen steder.

5.4.3 Resultat

I dette kapittelet har vi studert en potensielt god løsning til å oppdage endringer i et korallrev. Vi erfarte at det var krevende å lage en algoritme som tok hensyn til alle type endringer.

Løsningen hadde større utfordringer enn vi hadde håpet. Kapittelet beskriver essensen i, de sterke sidene ved og utfordringene med løsningen. Vi klarer enkelt å identifisere grener som har blitt kortere. Grener som forsvinner vil skape litt mer problemer, på grunn av måten vi gir navn til grenene.

En av grunnene til at vi valgte å ikke ferdigstille algoritmen var utfordringen med friske grener festet til bleknede grener (se oppe til venstre i figur 145). Vi så ingen åpenbar løsning på denne utfordringen.

Forslag til forbedringer:

1) Dele inn i stammer, slik at for eksempel fra B til toppen av B teller som én gren, eller stamme. Da vil programmet bedre identifisere når grener som er festet til denne forsvinner.

2) Konsekvent definisjon av start og endepunkter til grener: Dersom et endepunkt har tre eller flere naboer, vil det alltid være startpunkt. Dersom forslag 1) blir utført, vil alle grener ha ett endepunkt med en nabo, og ett med tre eller flere.

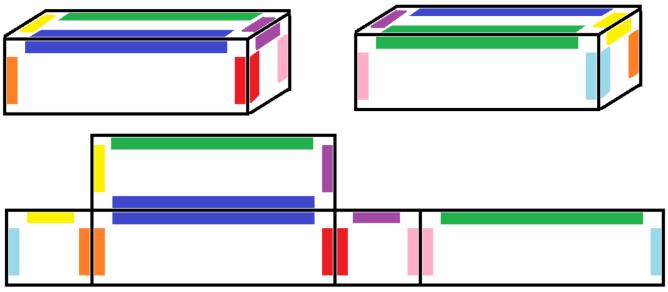
3) Grener med likt navn beskriver ikke nødvendigvis den samme grenen, noe figur 143 viser. For å løse dette kan programmet ta hensyn grenens posisjon i bildet.

4) Det å benytte en skjeletteringsalgoritme [73] kan føre til at punktene gitt av *k-means* legger seg på en rettere linjer og ikke klumper seg sammen.

6 Fotomosaikk av T-banevogn

6.1 Oppgaven

I den siste bildebehandlingsoppgaven fra MATE tas det bilder av en nedsunket T-banevogn. Disse bildene skal så limes sammen til en fotomosaikk. ROV-en kan styres manuelt for å ta bildene. Inngangsverdiene til algoritmen blir dermed fem bilder, en for hver side av T-banevognen. Algoritmen skal så behandle disse bildene å produsere en bildemosaikk sammensatt av topp- og sideflatene til T-banevognen. Nederst i figur 146 har vi et eksempel på hvordan en slik bildemosaikk kan se ut.



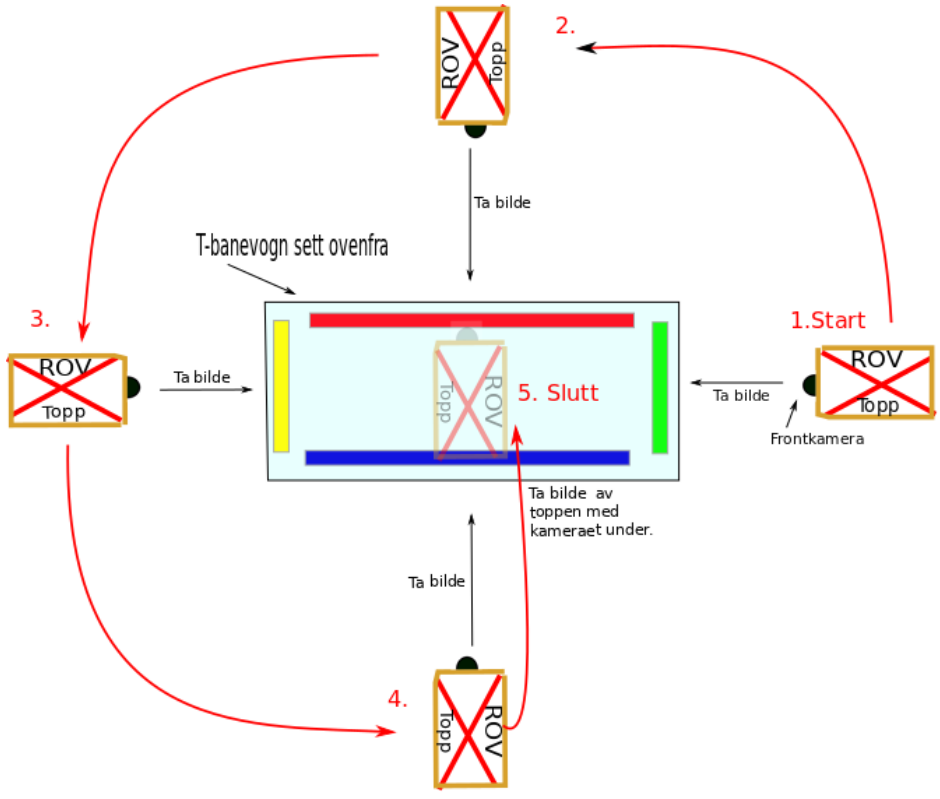
Figur 146: Den øverste delen av bildet viser hvordan T-banevognen blir illustrert av MATE. Den nedereste delen viser bildemosaikken. Altså det bildebehandlingsprogrammet skal produsere. Bildet er hentet fra MATE sin konkurransemanual [15]. Legg merke til hvordan hver enkelt flates kanter er markert med en farge, og hvordan denne fargen er identisk med nabokantens fargemarkering. Dette er en viktig egenskap ved T-banevognen som potensielt kan bli brukt til å løse oppgaven.

6.2 Løsningsforslag

Løsning tar direkte utgangspunkt i oppgaven slik den er beskrevet. Først benyttes ROV-en til å ta fem bilder. Ett bilde av hver av T-banevognens fem synlige sider. To langsider, to kortsider og en toppflate. Disse sendes så inn for behandling og til slutt presenteres resultatet via ROV-ens brukergrensesnitt.

Antagelser og forutsetninger for løsningen:

- Programmet iverksettes av en operatør etter at det har blitt tatt fem bilder, et bilde for hver av T-banevognens sider. Resultatet returneres og publiseres via brukergrensesnittet.
- Disse bildene må tas på en best mulig måte. Gode bildetakningsposisjoner er illustrert i figur 147. ROV-ens rull- og stampregulatorer bør være aktivert slik at ROV ligger rett i vannet.
- Bildene må tas slik at kanten som er nærmest bunnen på hver sideflate er orientert mot den nedre delen av bildet. Alle bildene som sendes bør for ordens skyld være i et liggende format.
- Bildene bør tas slik at hver enkelt side-/toppflate blir likest mulig et rektangel/kvadrat og fyller en størst mulig andel av bildet. Hele flaten må være inneholdt i bildet.



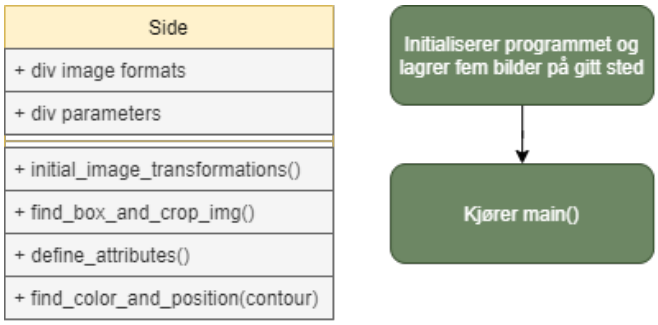
Figur 147: Mulig prosedyre for å ta bilder til fotomosaikk av T-banevogn. Rekkefølgen bildene tas i er uviktig. Det viktigste er at ROV-en ligger horisontalt i vannet og tar bildene fra en normal vinkel i forhold til sideflatene.

6.2.1 Overordnet løsning

Programmet vi har laget består av en rekke funksjoner og en klasse. Klassen har vi kalt *Side* da vi oppretter ett objekt av denne klassen for hver av sideflatene. I figur 148 er det et klassediagram til denne klassen. Vi har også laget en funksjon for tegning av selve fotomosaikken, og en hovedfunksjon vi kaller *main*. Hovedfunksjonen setter sammen puslespillet av klasser og funksjoner og produserer et ferdig resultat.

Programets gang er at operatøren lagrer fem bilder, for så å iverksette *main*-funksjonen, som henter disse bildene og lager en fotomosaikk.

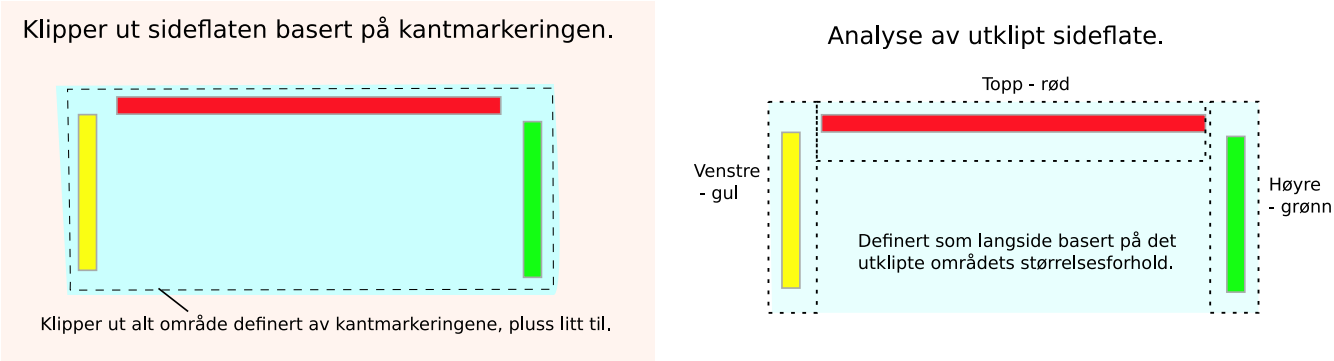
Bildebehandlingsdelen av programmet ligger i klassemetodene til klassen Side. De viktigste punktene i denne prosedyren er som følger:



Figur 148: Klassen *Side* og fremgangsmåte for å kjøre programmet.

- Forberedende behandling av originalbildet i *initial_image_transformations()*.
- Deretter klippes selve side-/toppflaten ut av bildet i *find_box_and_crop_img()*. Vi benytter oss her av markeringen med farger langs flatenes kanter. Dette er illustrert i venstre del av figur 149.
- Deretter defineres det hva slags flate som behandles, kortside, langside eller toppflaten. Det defineres også hvilke farge hver av sideflatens markerte kanter har. I høyre del av figur 149 illustreres dette. Disse operasjonene gjøres med *define_attributes()* og *find_color_and_position()*.

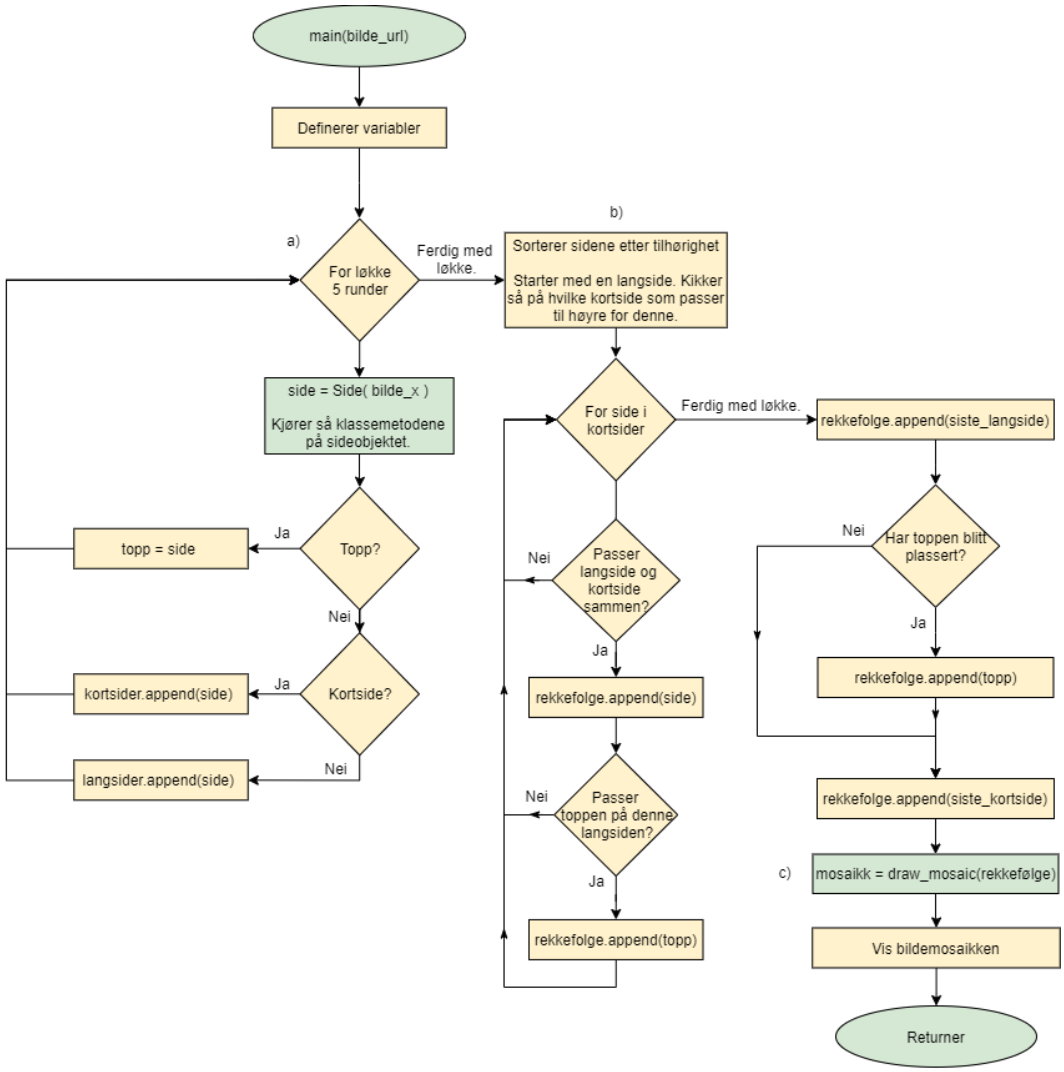
Etter at alle de fem bildene har blitt behandlet av bildebehandlingsmetodene, sorteres de og limes sammen til en fotomosaikk tilsvarende den vi så nederst i figur 146.



Figur 149: Venstre del: Utklipp av området som inneholder sideflaten. Klippingen basert på kantmarkeringene. Høyre del: Analyse av kantmarkeringene og sidetype. Fire kantmarkeringer antyder toppflaten. Når det kun er tre kantmarkeringer er det flatens størrelsesforhold(seksjon 3.3.10) som avgjør sidetypen, lang- eller kortside.

6.2.2 Detaljert gjennomgang av løsningen

main:



Figur 150: Blokkskjema til hovedfunksjonen *main*.

a) Ved initialisering opprettes en rekke variabler og tomme lister. Deretter kjører programmet fem runder i en løkke hvor fem bilder blir hentet fra en plass som blir definert i inngangsverdien til funksjonen. For hver av bildene opprettes det et objekt av klassen Side. Klassemetodene *initial_image_transformations()*, *find_box_and_crop_img()* og *define_attributes()* kalles så på objektet. Ut fra hvilke definisjon flaten fikk, kortside, langside eller toppflate, blir den lagret i tilhørende variabel/liste.

b) Programmet sortere så flatene slik at de ligger i en fast rekkefølge basert på hvor i fotomosaikken de skal tegnes inn.

c) Tilsutt sendes denne listen til *draw_mosaic* som returnerer den ferdige fotomosaikken av T-banevognen. Dette bildet lagres og publiseres i brukergrensesnittet slik at dommeren kan se resultatet.

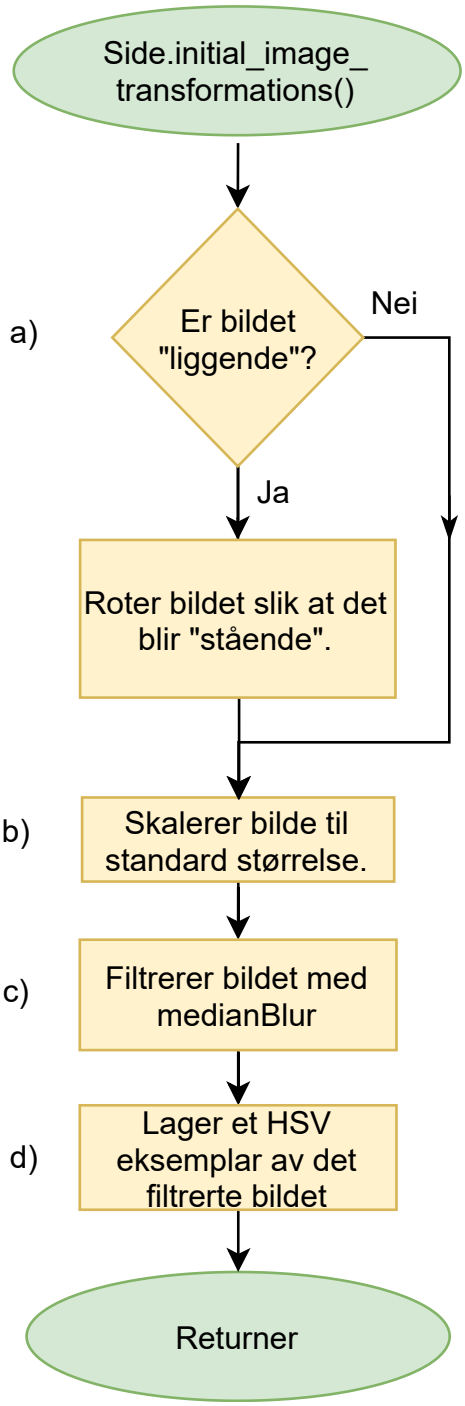
initial_image_transformations:

a) Bildets størrelsesforhold sjekkes, og roteres eventuelt slik at det blir liggende. Med liggende mener vi at bildet er bredere en det er høyt. Til å utføre denne rotasjonen bruker vi OpenCV sin *rotate* funksjon (se seksjon 3.4.3).

b) Deretter skaleres bildet ned til en standardisert størrelse basert på bildehøyden. Til å utføre den operasjonen benytter vi oss av OpenCV sin *resize* funksjon (se seksjon 3.4.4). Vi benytter *INTER_AREA* metoden som er best egnet til reduksjon av bildestørrelsen.

c) Videre filtreres bildet med *medianBlur* funksjonen omtalt i seksjon 3.10.1.

d) Så lagres et eksemplar av det filtrerte bilde i HSV-format. Vi gjør da om fra BGR- til HSV-format med cvtColor som ble omtalt i seksjon 3.1.1.



Figur 151

find_box_and_crop_image:

a) Først maskeres alle tydelige/markerte farger ut fra bildet. Vi benytter oss da av fargemaskering omtalt i seksjon 3.1.

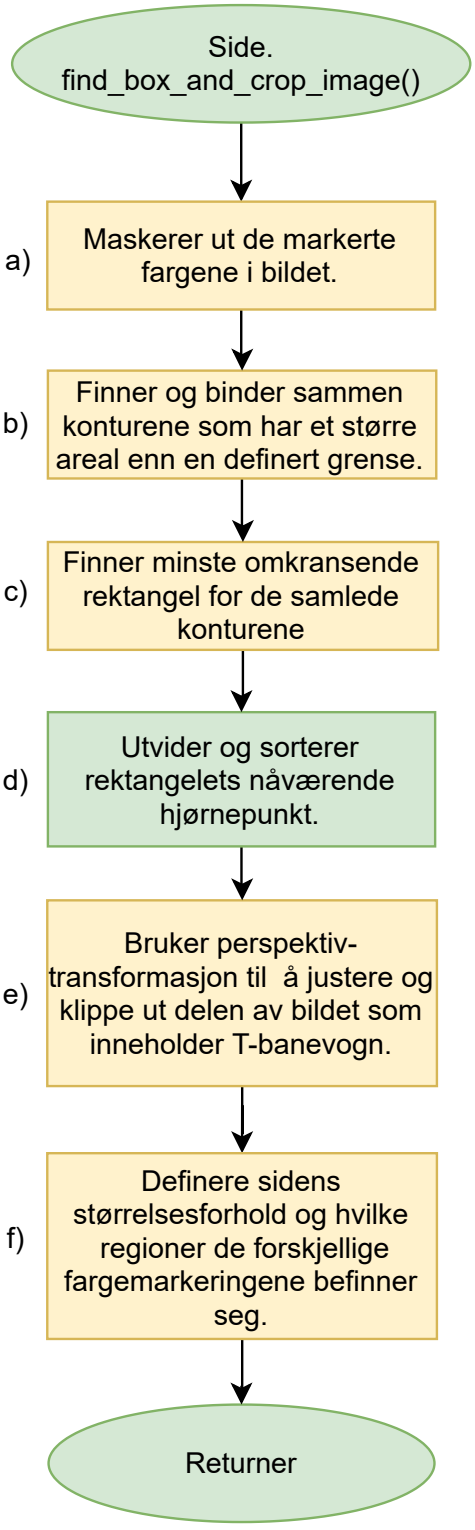
b) Så finner vi konturene i denne masken som er større enn en gitt grense. Til dette bruker vi *findContours* og *contourArea* funksjonene omtalt i seksjon 3.3. Vi sitter da igjen med alle områdene som markerer en kant. Vi binder så alle disse konturene sammen til en stor kontur ved å tegne en strek fra kontursenter til kontursenter.

c) Med *minAreaRect*(seksjon 3.3.5) finner vi minste omkransende rektangel til denne store konturen.

d) Deretter bruker vi *boxPoints*(seksjon 3.3.6) til å finne rektangelets hjørnepunkter, *shrink_contour*(seksjon 3.10.3) til og utvide rektangelet noe og *find_corner_points* til å sortere hjørnepunktene slik at de er klare for neste steg.

e) Vi bruker så de fire hjørnepunktene fra forrige punkt til å klippe ut selve flaten med perspektiv transformasjon (se seksjon 3.4). I tillegg til å klippe ut flaten rettes noe av eventuelle rotasjoner og forskyvninger i bildet opp. Den utklipte flaten lagres i en standardisert størrelse.

f) Til slutt defineres størrelsesforholdet som brukes til å fastslå sideflatens type, altså om det er en kortside eller langside. Basert på størrelsesforhold defineres det også hvilke områder av sideflaten som bør inneholde kantmarkering for høyre, venstre, topp og bunn.



Figur 152

define_attributes:

a) Først maskeres alle tydelige/markerte farger ut fra bildet. Vi benytter oss da av fargemaskering omtalt i seksjon 3.1. På denne måten henter vi ut fargemarkeringen av sideflatens kanter.

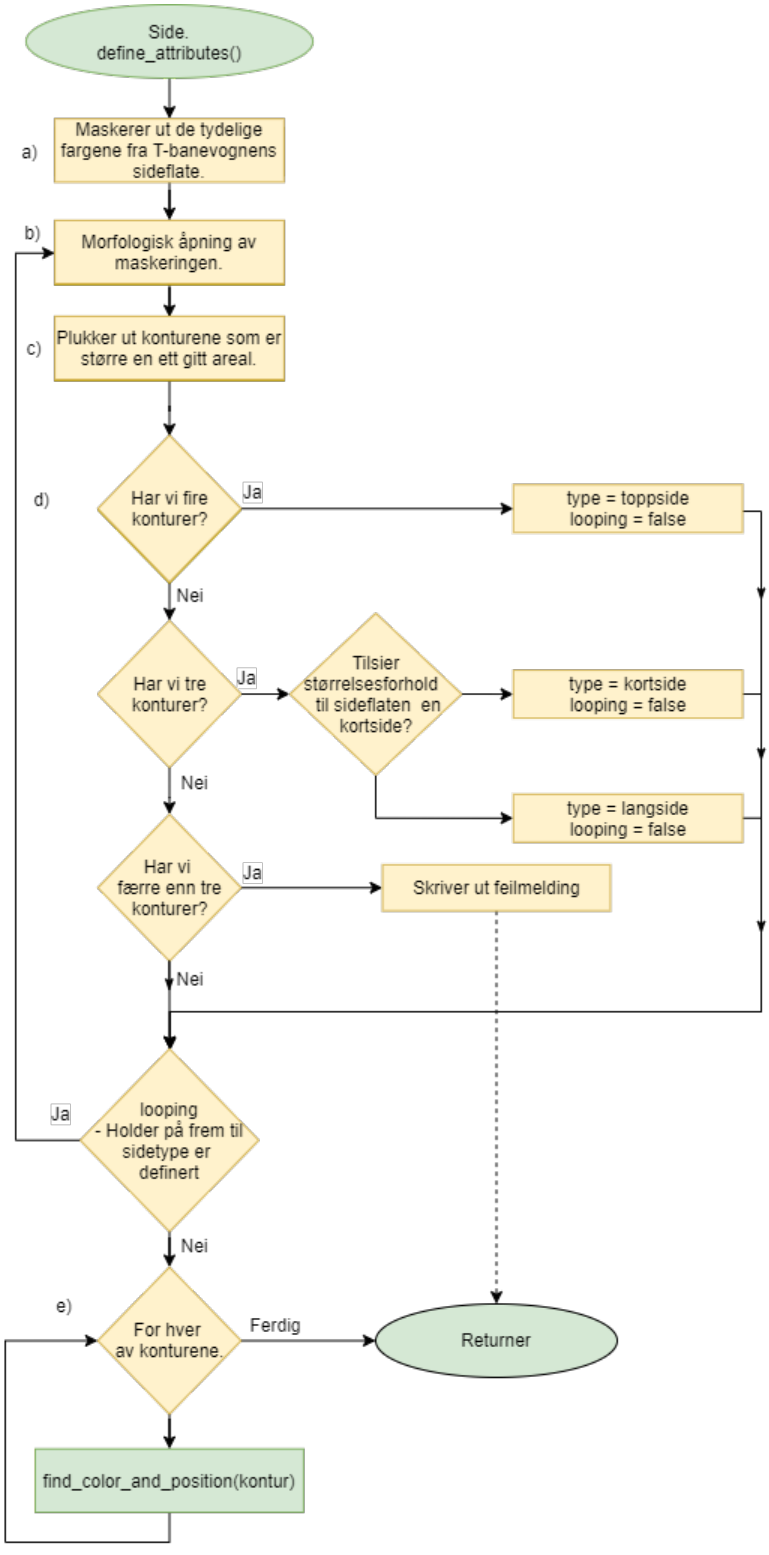
Deretter går programmet inn i en løkke hvor det blir værende helt til ett av tre mulige utfall er oppnådd. At vi finner fire, tre eller færre konturer.

b) Det første som skjer i denne løkka er at vi utfører morfologisk åpning på det maskerte området (se seksjon 3.5.3). For hver iterasjon av løkka blir denne kraftigere og kraftigere. Målet er med dette trinnet er å filtrere bort konturer som ikke er en kantmarkering, samt jevne ut konturene som er kantmarkeringer.

c) Videre finner vi konturene i masken som er større enn en gitt grense. Til dette bruker vi *findContours* og *contourArea* funksjonene omtalt i seksjon 3.3. Vi sitter forhåpentligvis igjen med alle områdene som markerer en kant.

d) I dette nest trinnet avgjør vi hva slags sidetype dette er. Har vi funnet fire konturer er det en toppflate. Har vi funnet tre så er det enten en kort- eller langside. Kort- og langsidene skilles ved å kikke på flates størrelsesforhold. Kortsidene er tilnærmede kvadrater, mens langsidene er brede rektangler.

e) Programmet har nå fastslått hva slags sidetype den aktuelle siden er. Det går så gjennom hver av konturene som den forrige løkka produserte. Dette er kantmarkeringene. Disse sendes til *find_color_and_positon* metoden. Her fastslås fargen til hver av kantmarkeringene og hvilke regioner/kanter de er representerer.



Figur 153

find_color_and_position::

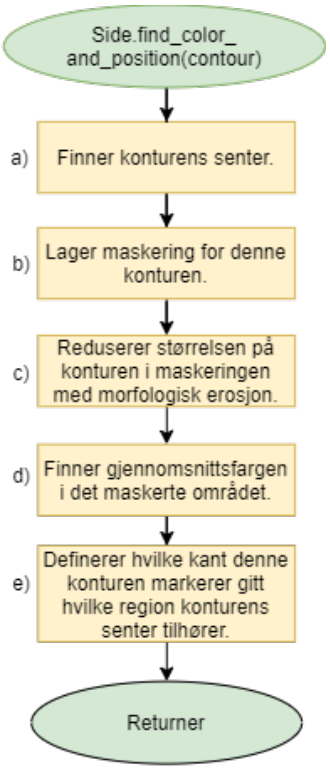
a) Først finner vi konturens senter ved hjelp av OpenCV sin *moments* funksjon[50].

b) Deretter oppretter vi en maskering ved å lage et tomt bilde hvor konturen tegnes inn med *drawContours* (se seksjon 3.3.2).

c) Videre reduseres størrelsen på konturen med morfologisk erosjon (se seksjon 3.5.1).

d) Deretter benyttes denne masken til å regne ut gjennomsnittsfargen på til alle pikslene inneholdt i konturen. Til denne operasjonen bruker vi funksjonen *mean* fra seksjon 3.3.9, eller *mean_HSV_color_of_masked_area* fra seksjon 3.10.4. Årsaken til at vi reduserte konturen i forrige punkt var at overgangen mellom fargelagt og hvitt område har blitt noe påvirket av tidligere filtrering. Vi trekker oss derfor bort fra ytterkanten for å unngå at disse pikslene påvirker hvilke farge kantmarkeringen får.

e) Til slutt defineres det hvilke kant/region kantmarkeringen tilhører ved hjelp av konturens senter. Det lagres så i Side-objektet hvilke farge denne kantmarkeringen har.



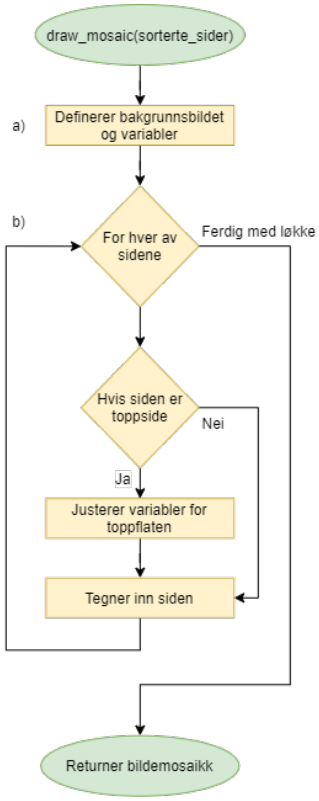
Figur 154

draw_mosaic:

a) Først defineres variablene som brukes til å tegne inn sideflatene på rett plass. Deretter opprettes et tomt bilde

b) Deretter går vi gjennom alle sidene og tegner inn bildene deres. Idet vi kommer til toppflaten justeres tegningsvariablene noe slik at også den tegnes på rett plass.

c) Til slutt returneres den ferdige fotomosaikken.



Figur 155

6.2.3 Testing

Vi har ikke hatt anledning til å teste programmet med ROV-en i vann. Istedenfor har vi manuelt tatt bilder av modellen som skal tilsvare en T-banevogn. Disse bildene er typisk innenfor det formatet vi kan forventet blir sendt inn fra ROV-en. Noen av bildene er tatt optimalt, slik det ble beskrevet innledningsvis under antagelser og forutsetninger, andre er noe forskjøvet og rotert. Testingen er ikke ment å vise programmets grenser, men skal bekrefte at programmet klarer å løse MATE-oppgaven under de gitte forutsetningene. Bildene ligger i vedlegget under /Bildebehandling/Bilder/Fotomosaikk.

Forklaring til tabellen:

- Nr - Bildeseriens nummer. Det er total 20 serier med testbilder. Med 5 bilder per serie blir dette 100 bilder totalt.
- Ulempe - Ukurant situasjon ved denne bildeserien.
- L-1 - Er første langside plassert?
- K-1 - Er første kortside plassert?
- L-2 - Er andre langside plassert?
- K-2 - Er andre kortside plassert?
- T - Er toppflaten plassert?
- Resultat - Er plasseringen rett eller feil?

Nr	Ulempe	L-1	K-1	L-2	K-2	T	Resultat
1	Hvit bakgrunn	Ja	Ja	Ja	Ja	Ja	Rett
2	Hvit bakgrunn	Ja	Ja	Ja	Ja	Ja	Rett
3	Hvit bakgrunn	Ja	Ja	Ja	Ja	Ja	Rett
4	Hvit bakgrunn	Ja	Ja	Ja	Ja	Ja	Rett
5	Hvit bakgrunn	Ja	Ja	Ja	Ja	Ja	Rett
6	Hvit bakgrunn	Ja	Ja	Ja	Ja	Ja	Rett
7	Variert bakgrunn	Ja	Ja	Ja	Ja	Ja	Rett
8	Variert bakgrunn	Ja	Ja	Ja	Ja	Ja	Rett
9	Variert bakgrunn, rotasjon	Ja	Ja	Ja	Ja	Ja	Rett
10	Variert bakgrunn, rotasjon	Ja	Ja	Ja	Ja	Ja	Rett
11	Variert bakgrunn, rotasjon	Ja	Ja	Ja	Ja	Ja	Rett
12	Variert bakgrunn, rotasjon	Ja	Ja	Ja	Ja	Ja	Rett
13	Variert bakgrunn	Ja	Ja	Ja	Ja	Ja	Rett

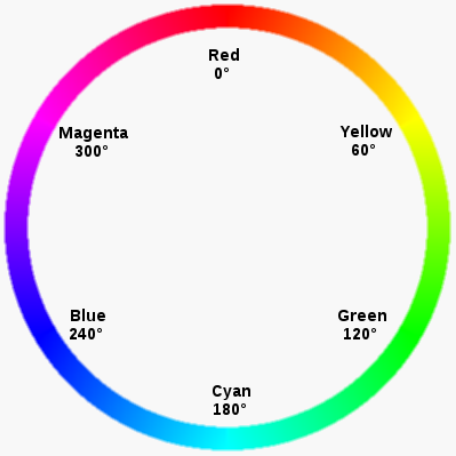
Nr	Ulempe	L-1	K-1	L-2	K-2	T	Resultat
14	Variert bakgrunn	Ja	Ja	Ja	Ja	Ja	Rett
15	Variert bakgrunn, stor rotasjon	Ja	Ja	Ja	Ja	Ja	Rett
16	Variert bakgrunn, stor rotasjon	Ja	Ja	Ja	Ja	Ja	Rett
17	Variert bakgrunn, stor rotasjon	Ja	Ja	Ja	Ja	Ja	Rett
18	stor rotasjon	Ja	Ja	Ja	Ja	Ja	Rett
19	stor rotasjon	Ja	Nei	Ja	Ja	Ja	Feil
20	stor rotasjon	Ja	Nei	Ja	Ja	Ja	Feil

Tidsbruk: I snitt bruker programmet 0.9 sekund på å lese inn fem bilder, behandle disse, tegne og lagre fotomosaikken.

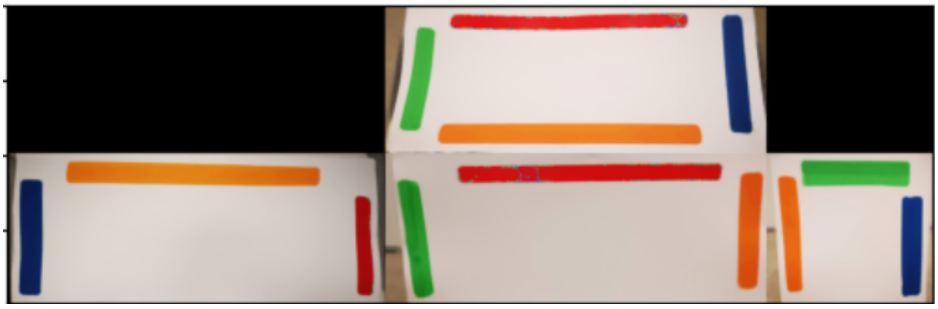
Nærmere kikk på feilene:

Forsøk nummer 19 og 20 manglet en kortside i fotomosaikken. I figur 157 er resultatet fra forsøk nummer 19 vist. Årsaken til svikten var feil beregning av gjennomsnittsfargen for en av de røde kantmarkeringene. Som nevnt i seksjon 3.1 om fargemaskering, representeres fargetonene i HSV-formatet av en enkelt verdi. Denne verdien tilordner fargene som gradene i en sirkel. Dette er illustrert i figur 156. Utfordringen oppstår rundt 0 og 360 grader, altså de røde fargetonene. Disse vinklene/fargene er tilnærmet like, men idet vi forsøker å beregne gjennomsnittsverdien blir svaret helt feil. Fargene/vinklene må derfor regnes om til det kartesiske koordinatsystemet for at vi kan beregne det korrekte gjennomsnittet.

Problemet har en enkel løsning. Hvis vi bytter ut OpenCV sin *mean* funksjon med den spesiallagde funksjonen *mean_HSV_color_of_masked_area* (seksjon 3.10.4) har vi løst problemet. Denne funksjonen er dessverre svært mye tyngre enn OpenCV sin utgave. Ved å bruke vår egen spesiallagde funksjon blir tidsbruken nesten firdoblet fra 0.9 til 3.5 s. Vi velger derfor å fortsette å bruke OpenCV sin funksjon. Vi tillater oss dette fordi modellen som skal brukes i konkurransen ikke har en rødfarge i grenseområdet mellom 0/360°. Vi prioritere altså hastighet da dette kun er et problem ved røde kantmarkeringer.

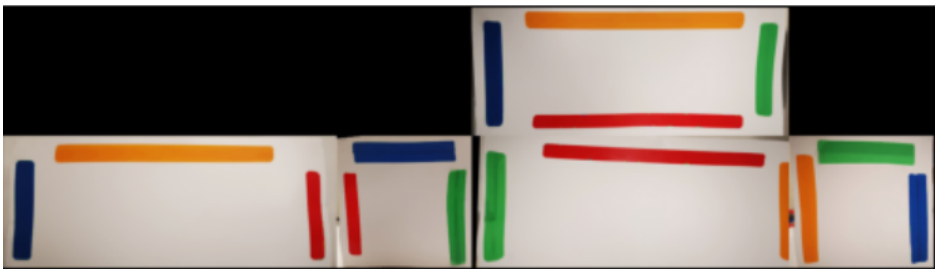


Figur 156: Fargetone sirkelen for HSV-formatet. Bildet er hentet fra [13]



Figur 157: Resultatet fra test nr 19 som feilet. Legg merke til den kortsiden som mangler. Det er denne kortsiden hvor gjennomsnittsfargen til den røde kantmarkeringen ikke ble korrekt beregnet.

Eksempeler på produsert fotomosaikk:
I figur 158, 159 og 160 har vi eksempler fotomosaikk som programmet har laget. Alle eksemplarene har skjevheter som kan spores tilbake til at bildene ble tatt fra ugunstige vinkler. Disse vinklene er ikke i henhold til den innledende beskrivelsen av hvordan bildene skal tas. Dette er ikke mer enn kosmetiske detaljer, og har ingenting og si for bedømmelsen i MATE-konkurransen.

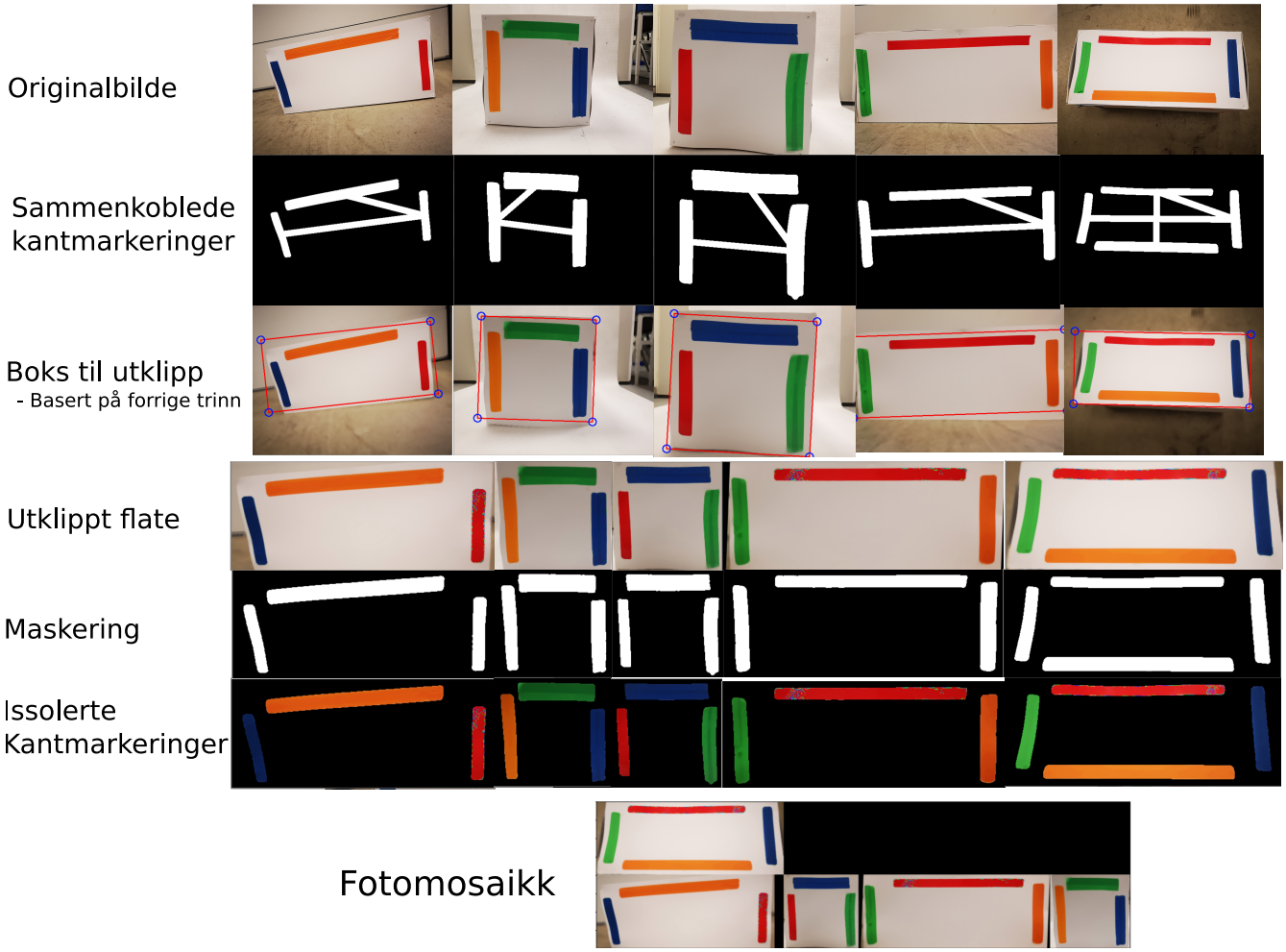


Figur 158

Figur 159

Figur 160

Eksempel på produksjon av bildemosaikk:
Figur 161 viser de viktigste stegene hvert enkelt bilde går gjennom i prosessen for å lage en fotomosaikk.



Figur 161: Bildet viser hovedtrinnene fra produksjon av en fotomosaikk. Eksemplet er fra bildeserie nummer 15.

6.2.4 Resultat

Vi fant to tilfeller hvor fotomosaikken ikke ble fullført. Disse tilfellene ser vi bort ifra da vi allerede har definert en løsning på problemet. Programmet lyktes altså i alle tilfeller å produsere fotomosaikken. Programmet har blitt testet med stort et utvalg av bilder. Disse bildene har forskjellige bakgrunner, lysforhold, rotasjoner og forskyvninger. Alt dette har programmet håndtert. Det har ikke blitt utført testing av ekstremtilfeller, men programmet håndterer tilfeller som er langt utenfor forutsetningene definert i innledningen. Dårlig kvalitet på inngangsbildene kan kjennes igjen i den resulterende fotomosaikken. Forskyvinger og skjevheter i inngangsbildene blir altså delvis beholdt i fotomosaikken.

Totalt sett oppfyller programmet kravene vi har satt og fungerer som ønsket. Programmet er egnet til bruk i MATE-konkurransen.

7 Styringsprogrammet

Denne delen inneholder en beskrivelse med krav til hvordan brukergrensesnittet og programstrukturen til ROV-ens styreprogram skal være. Deretter hvordan vi har løst oppgaven.

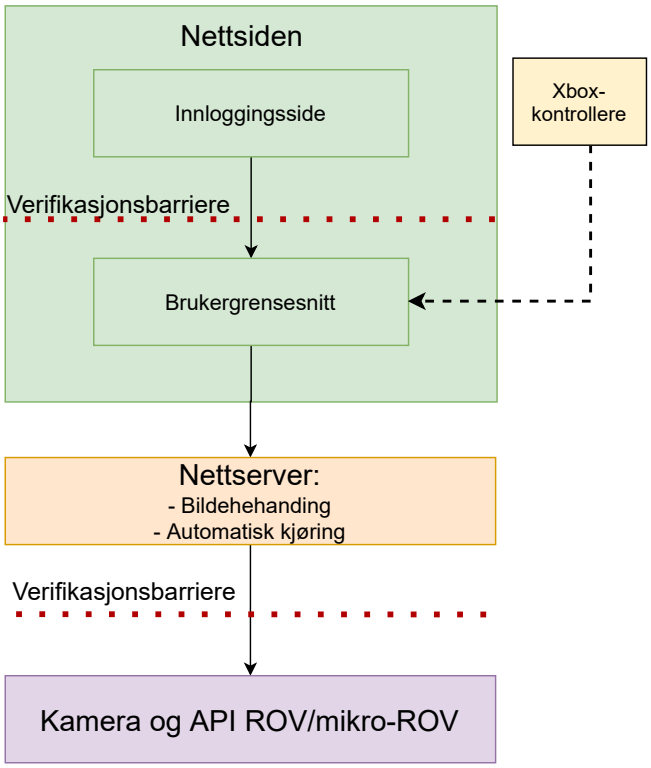
7.1 Krav og forutsentniger

UiS Subsea ønsker seg et nettbasert brukergrensesnitt. Denne løsningen skal være slik at ROV-en potensielt kan bli styrt over internett fra en nettleser. For å kunne fullføre dette prosjektet er vi avhengig av at alle av UiS Subsea sine involverte grupper lykkes i sitt arbeid. Vår oppgave går ut på å lage brukergrensesnittet og strukturen som ligger bakenfor denne. Til grunn for løsningen legger vi derfor at de andre gruppene lyktes i sitt arbeid.

I figur 162 ser vi en enkel skisse av programstrukturen slik vi ønsker den.

Løsningen skal oppfylle en rekke krav:

- Adgangskontroll til nettsiden/ brukergrensesnittet.
- Lese og skrive data til ROV.
- Detektere og benytte styreinstrukser fra Xbox-kontrollere.
- Initialisere program for bildebehandling samt fremvise resultatet i nettleser.
- Ta bilder til bildebehandlingen.
- Vise sanntidsvideo fra ROV-ens kamera.
- Bytte mellom automatisk og manuell styremodus.
- Aktivere/deaktivere bruk av manipulator og mikro-ROV.
- Koble fra eller til ROV.



Figur 162: Forenklet prinsippskisse av ønsket løsning. API står for "Application Programming Interface". På norsk blir dette programmeringsgrensesnitt. API beskriver grensesnittet for kommunikasjon mellom program.

Bidrag fra Kommunikasjonsgruppen:

Kommunikasjonsgruppen har ansvar for å lage en API-løsning til ROV-en og mikro-ROV-en. Dette er grensesnittet for tilkobling til ROV-ene fra styringsprogrammet. Vi legger noen antagelser til grunn for løsningen som blir produsert:

- Sikker og rask kommunikasjonsløsning til ROV.
- Denne kommunikasjonsløsningen inneholder mulighet for adgangskontroll og verifikasjon.

I løpet av utviklingsprosessen har det blitt klart at disse antagelsene ikke ville bli oppfylt. Forutsetningen til har derfor blitt endret til løsningen beskrevet nedenfor.

- "Hypertext Transfer Protocol" (HTTP) basert API. Metoden benytter "GET"- og "POST"-metoder til og å lese og skrive data til ROV-ene. I samarbeid med kommunikasjonsgruppen har vi målt en oppdateringsfrekvens på omtrent 10Hz. Denne hastigheten er avhengig av at det kun er én klient som etterspør informasjon. Dette resultatet er vesentlig lavere enn hva vi hadde ønsket oss. Ved å benytte en metode som holder tilkoblingen åpen, kunne resultatet antageligvis blitt mye bedre. HTTP-metoden oppretter en tilkobling, og lukker denne for hver melding som blir sendt. Tilsynelatende er det opprettelse av denne tilkoblingen som tar lengst tid.
- Tilkoblingen har ikke blitt sikret og har ingen autentiseringsmulighet. Så lenge dette er status bør vi ikke koble ROV-en til internett.

I tillegg til kommunikasjonsløsningen har kommunikasjonsgruppen hatt ansvaret for å velge ut et kamera til bruk på ROV-ene. De har valgt å bruke et IP-kameraet[84] kjøpt hos AliExpress.

Bidrag fra Motorstyrings- og reguleringsystemgruppen:

Denne gruppen har hatt ansvar for å bestemme en løsning for manuell styring av ROV-en. De har i den forbindelse valgt å bruke Xbox-kontrollere. De har bekreftet at JavaScript kan håndtere tilkobling av Xbox-kontrollere til nettleseren. Videre har de laget en funksjon for å omgjøre pådrag fra Xbox-kontrolleres styrestikker til korrekte pådragsverdier til ROV-en. De har også deltatt i arbeidet med å utvikle ett felles sett med variabler som transporteres mellom nettsiden og ROV-en. Disse settene med variabler lagres i JSON-filer[31] som sendes mellom nettsiden, serveren og ROV-ene. Denne prosessen beskrives detaljert i kapittelet om løsningen for styringsprogrammet.

7.2 Komponenter brukt i løsningen:

7.2.1 Python

Vi har valgt å bruke programmeringsspråket Python [24] av to hovedgrunner.

- Vi har noe erfaring med programmering i Python.
- All bildebehandlingen foregår i Python. Hvis vi bruker Python blir det enklere å implementere disse løsningen.
- Python er et av de mest populære programmeringsspråkene. Dette gjenspeiles i mye god dokumentasjon og eksempelkode.

7.2.2 Flask

Vi har valgt å bruke nettrammeverket Flask[68] til å bygge opp "baksiden" av nettsiden. Et nettrammeverk er et rammeverk som forenkler konstruksjonen av nettsider og applikasjoner. Den legger grunnlaget/strukturen for hvordan programmene kan bli bygd opp. Flask har blitt valgt av følgende grunner:

- Dette er et såkalt mikrorammeverk for WSGI nettprogram. WSGI står for "Web Server Gateway Interface" og er en metode for kommunikasjon mellom en nettserver og ett nettrammeverk skrevet i Python. Den inneholder i seg selv kun det mest nødvendige til et nettrammeverk, derav mikronrammeverk. All ekstra funksjonalitet må altså implementeres ved hjelp av tredjepartsløsninger eller egen kode. For eksempel benyttes tredjepartsløsninger til kryptering av passord, datalagring, datahåndtering og innloggingløsning. Det at Flask er et mikrorammeverk gjør det til en utmerket platform til å konstruere spesiallagde løsninger.
- Vi er noe kjent med dette rammeverket fra tidligere.
- Flask er også et av de mest brukte netterammeverkene i Python. Dette betyr at det finnes mye god dokumentasjon.

7.2.3 Gunicorn og Nginx

I utviklingsperioden blir Flask sin innebygde utviklingsserver(nettserver) brukt. Det ferdige produktet skal settes opp på en egen datamaskin. Da anbefales det å bruke en kraftigere motor enn utviklingsserveren[67]. Til dette trengs en såkalt WSGI-server(nettserveren). WSGI står for "Web Server Gateway Interface" og er en metode for kommunikasjon mellom nettserveren og nettrammeverk skrevet i Python. Vi har valgt[67] en metode som benyter Gunicorn og Nginx[7]. Gunicorn er en WSGI HTTP server for Python(nettserver). Nginx er en nettserver som anbefales[7] brukt som et mellomledd mellom Gunicorn og internett. Dette oppsettet svært utbredt for nettsider basert på Python og Flask. Nginx brukes da som et lastbalanseringsledd som fordeler trafikk/forspørsler fra internett på flere identiske prosesser/tråder med Gunicorn-servere. Vi prioriter å ikke utrede dette konseptet videre, men lener oss på kildene som beskriver dette som vanlige løsninger.

7.2.4 JavaScript, HTML og CSS

JavaScript, HTML og CSS er grunnsteinene i den moderne nettutviklingen. Vi benytter oss av disse til nettsiden/brukergrensesnittet.

HTML:
HTML står for "HyperText Markup Language", på norsk blir dette hypertekstmarkeringsspråk. Dette språket brukes til å bygge opp strukturen i nettsider. Dette er på mange måter det visuelle rammeverket for nettsider.

CSS:
CSS står for "Cascading Style Sheets", på norsk blir dette gjennomgående, eller fallende, stilark. CSS er et språk som brukes til å definere hvordan HTML filene skal se ut. I en tegning kan HTML sammenlignes med alle streker og kanter, mens CSS er fargeleggingen. Mer informasjon om HTML og CSS er å finne i artikkelen "HTML For Beginners"[36].

JavaScript:
JavaScript er et høynivåspråk som alle moderne nettlesere kan kjøre. JavaScript benyttes til å lage avansert dynamisk funksjonalitet på nettsidene. Med JavaScript kan tegningen nevnt i forrige avsnitt, bli til en film. I tillegg til standard JavaScript benytter vi oss av det populære biblioteket Jquery.js. For mer dokumentasjon og informasjon om JavaScript, se MDM Web Docs sin guide[20].

Go
Go er et programmeringsspråk først påbegynt av en gruppe hos Google. Det blir ofte kalt Golang, etter hjemmesidens navn, men det offisielle navnet er Go. Språket er åpen kildekode, som har ført til at flere frivillige har bidratt til utviklingen. Go er statisk, kompilert og syntaksmessig ganske likt C. Versjon 1 ble utgitt mars 2012. Det ble utviklet med vekt på:

- Kjøreeffektivitet (som C).
- Lesbarhet og brukervennlighet (som Python).
- Høy ytelse, nettverksmuligheter og multiprosessering.

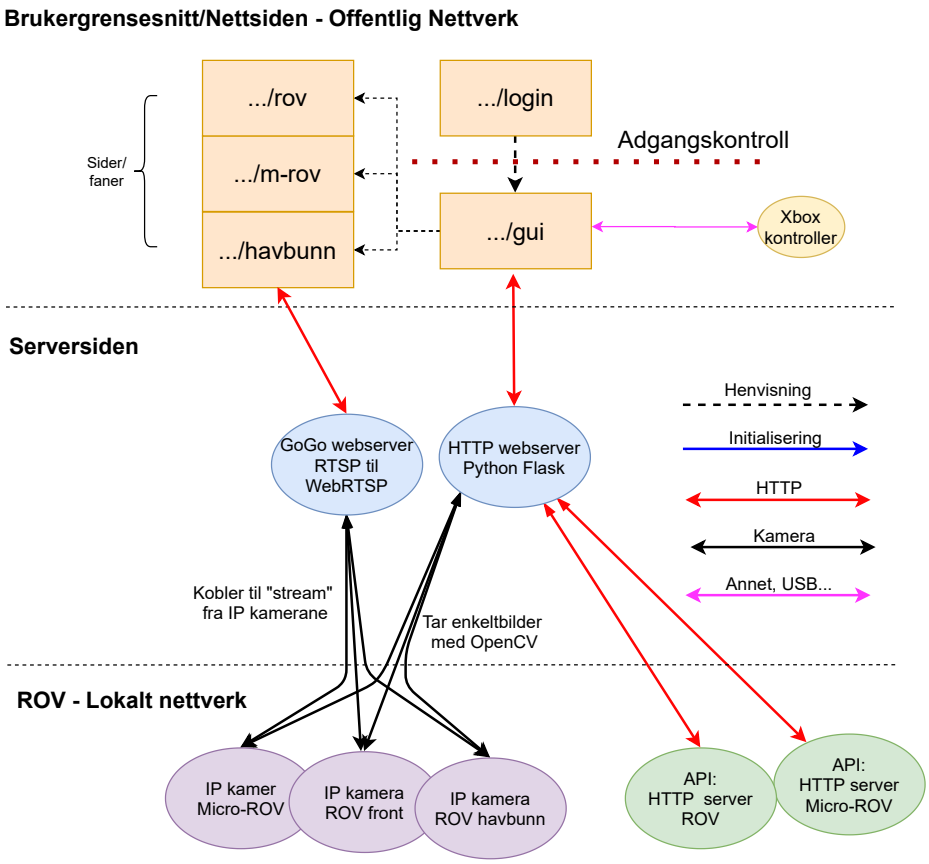
Go er brukt mye internt hos Google og i mange prosjekter basert på åpen kildekode. For mer informasjon om Go, se deres nettside [4].

7.3 Løsningen

Hele systemet er delt i tre nivå:

- **Brukergrensesnittet:**
  Nettsiden som operatør/bruker kobler seg til. Denne skal være tilgjengelig over internett.
- **Serversiden:**
  Løsningen benytter seg av to nettservere, Go-server til sanntids video og en Flask basert Python-server til de andre funksjonalitetene. Serverene er tilkoblet internett samtidig som de har tilgang til ett lokalt nettverket med ROV-ene. Python-serveren blir bindeleddet mellom brukergrensesnittet i nettleseren, ROV-ene og bildebehandlingen. Arbeidsflyten går slik at det sendes instrukser fra brukergrensesnittet/nettsiden som utføres i Python. Dette kan være beskjed om å hente data fra ROV, skrive data til ROV, ta bilder, slette bilder eller analysere bilder.
- **ROV-ene:**
  Her inngår alt fra ROV-ene. IP-kamera og kommunikasjon til kretskortene på ROV-ene. Alle komponentene er samlet i et lokalt nettverk.

Løsningen blir skissert i figur 163.

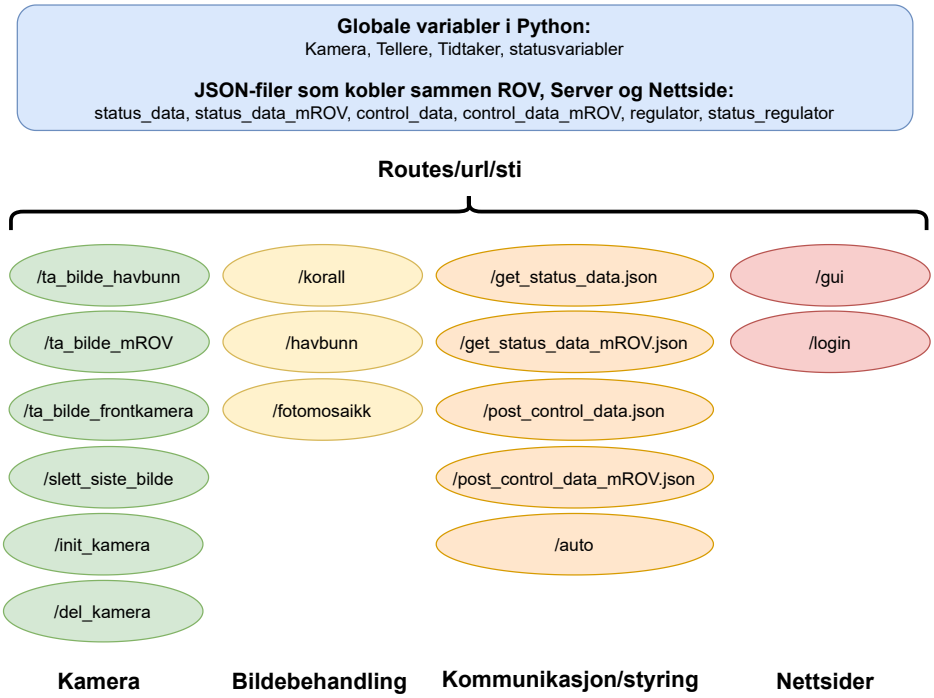


Figur 163: Beskrivelse av programstrukturen.

I resten av dette kapittelet går vi detaljert gjennom de ulike komponentene i, og momenter ved løsningen.

7.3.1 Flask

Flask er som tidligere beskrevet et nettrammeverk. I dette rammeverket har vi definert en rekke nettadresser/stier som utfører forskjellige oppgaver når de kalles. Denne strukturen er beskrevet i figur 164.



Figur 164: Innholdet i figuren er å finne i den vedlagt filstrukturen under /Brukergrensesnitt/routes.py

Innholdet i den blå boksen beskriver globale variable definert i Python-koden. JSON-filene[31] eksisterer både på ROV-en, i Python- og JavaScript-miljøene. Filene inneholder informasjon om alt fra sensordata, statusvariabler og pådragskoffisienter til motorene. De brukes til å synkronisere informasjon mellom lagene i programstrukturen.

De grønne ellipsene representerer stier som, når kalt, utfører operasjoner forbundet med kameraene. Ta bilde, og slett bilde funksjonene returnerer det siste bildet i en egen fane.

De gule ellipsene representerer stier som kaller bildebehandlingprogrammene. Disse henter bilder fra en definert lagringsplass og utfører den aktuelle operasjonen på dem. Resultatene presenteres som et bilde i en egen fane som åpnes i nettleseren.

De oransje ellipsene representerer metodene for kommunikasjon og styring. Alle metodene returnerer JSON-filer med status fra ROV/mikro-ROV. */auto* og POST-metodene skriver data ned til ROV-en. Informasjonen som sendes til ROV-/mikro-ROV-en er variabler som beskriver motorpådrag, posisjon styrestikker, lys av eller på, osv.

De røde ellipsene inneholder stiene som returnerer komplette enkeltsideapplikasjoner bestående av HTML, JavaScript og CSS.

Kamera og bilde:
Når operatøren tar et bilde lagres dette på en forhåndsdefinert plassering. En global variabel benyttes til å nummerere bildene. *../init_kamera* og *../del_kamera* metodene, oppretter og sletter tilkoblingene til IP-kameraene *../ta_bilde* metodene kobler seg til det aktuelle kameraet på nytt om tilkoblingen faller ut. *../slett_bilde* metoden sletter det siste bildet som ble tatt.

Bildetakningstid med OpenCV:
Den vedlagte filen /Kodesnutter/test_kamera.py inneholder kode for testing av bildetakningshastigheten til ett av IP-kameraene ved å bruke OpenCV. Ved å kjøre 1000 testbilder har vi funnet ut at det tar omtrent 67 ms å ta et bilde. Resultatet er basert på målinger gjort med med Python sitt innebygde bibliotek *time* og funksjonen *perf_counter* [25].

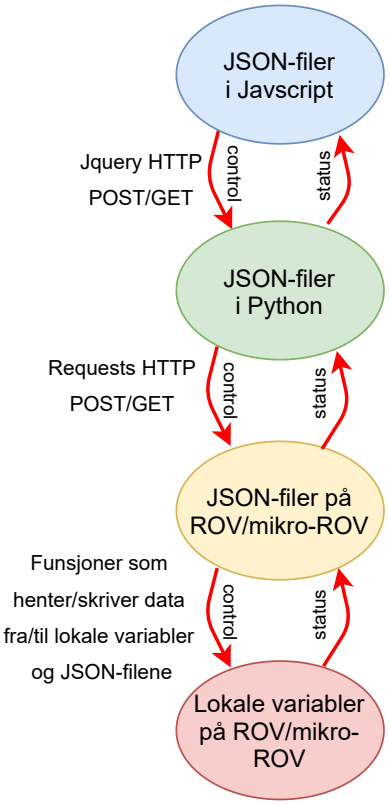
Bildebehandling:
Når en bildebehandlingsmetode kalles, behandler metoden det/de siste bildet/bildene som ble tatt. Disse bildene hentes fra en forhåndsdefinert plass på serveren. Resultatet lagres på en annen forhåndsdefinert plass. Deretter returneres en HTML-side som henter inn resultatet. Denne siden åpnes i en ny fane.

Lesing og skriving til ROV:

For lesing og skriving til ROV/mikro-ROV fra Python, benyttes biblioteket Requests [70]. All kommunikasjon gjøres ved hjelp av de standardiserte JSON-filene nevnt i figur 164. På ROV-siden er det spesiallagde funksjoner som håndterer de forskjellige JSON-filene og lagrer/henter data fra de lokale variablene. Disse funksjonene er det kommunikasjonsgruppen som har laget. Eksempel på bruk av Requests for kommunikasjon med JSON-filer kan ses i kodesnutt 29. Figur 165 illustrere kommunikasjonen mellom programmets ulike deler.

```
1  import  requests, json
2
3  url = 'http://192.168.3.101'
4
5  # JSON filer
6  control = {"control" : {...}}
7  status = {"status" : {...}}
8
9  # POST - Skriver og leser til ROV
10 r1 = requests.post(url+'/control.json',control)
11 status['status'].update(r1.json()['status'])
12
13 # GET - Leser fra ROV
14 r2 = requests.get(url+'/status.json')
15 status['status'].update(r2.json()['status'])
```

Kodesnutt 29: Eksempel på bruk av Requests til å lese og skrive JSON filer.



Figur 165: Enkel illustrasjon av hvordan JSON-filene *control_data* og *status_data* fraktes fra brukergrensesnittet i nettleseren, til Python-serveren og videre til ROV-en.

Automatisk kjøring:
Programmet for automatisk kjøring er i utgangspunktet slik som illustrert i figur 66. Løkkefunksjonen er flyttet ut til JavaScript funksjonen *update_status()* beskrevet i figur 171. Første gang *../auto* kalles, opprettes en rekke globale variabler og objekter i en oppstartsekvens. Deretter, for hver gang *../auto* kalles hentes ett bilde, dette behandles, pådrag beregnes med PID-regulatorene, pådraget sendes ned til ROV-en, og oppdatert status returneres til *update_status()* funksjonen i brukergrensesnittet.

7.3.2 Sikkerhet:

Nettsiden til brukergrensesnittet er beskyttet ved hjelp av verifikasjon av brukerne. ROV-ene er ikke tilkoblet internett, men er kun åpent for det lokale nettverket. All kommunikasjon til ROV-ene går gjennom Python-serveren som driver nettsiden. Videostrømmene er åpne for alle som kjenner nettadressen.

Tabellen nedenfor går gjennom viktige aspekter ved håndteringen av sikkerhet. Den oppsummerer sentrale svakheter og risiko momenter, samt grepene/løsningene vi har brukt til å bygge en sikrest mulig nettside. Tiltakene som blir nevnt i tabellen skal sikre oss mot noen av de vanligste angrepsformene definert av OWASP topp ti (2017)[64]. Det gås ikke dypere inn i denne tematikken enn det som blir nevnt i denne tabellen. For forklaring av forkortelser og faguttrykk gå direkte til kilden som det blir referert til.

Emner	**Sak**	**Tiltak:**
Database/lagring	Passord/brukernavn	Lagres i en SQLite[83] database
	Injeksjon av SQL kode	Kommunikasjon til database vha. ORM (sqlAlchemy[82])
	Passordlagring	Passord lagres i kryptert form (bcrypt[69])
	Feile sikkert	Operatør observerer/sikrer ROV-ene.
		Eneste bindeledd mellom brukergrensesnitt og ROV er JSON-filene
	Logging	Lagrer datastrøm fra Flask i tekst fil
	Hemmeligheter	Secret key og SQlite3 URI - Lagret i egen json fil eller systemvariabler
Inndata	Brukerinnput	WTForms[91] til datahåndtering
		Spesifikke valideringskrav til hvert dataelement
		POST- metoden for inndata
	XSS	Jinja2(Flask) - Autoescaping
		Jinja2(Flask) - Skjule koden for nettleser
		Talisman[65]/"security headers"[1] - Kun HTTPS tillatt +++
	CSP	White list over godtatte kilder Content-security-policy[19]
	CSRF	WTForms[91]: CSRF-token

Emner	Sak	Tiltak:
Adgangskontrol	Uvedkomne får tilgang	Flask-Login[22] sikrer adgangskontrollen til sidene
	Session/cookies	Sikkerhetskonfigurasjon: kun over HTTPS[81]
		Husk meg funksjonalitet utløper etter 30 dager
Kommunikasjon	HTTPS Serifikat	Levert av Certbot[23]
	Konfigurasjon	Manuelt og ved hjelp av Flask-Talisman[65]
		=> HTTPS påtvunget og eneste mulig tilkobling
		=> diverse andre svakeheter tettet med "security headers"[1]
	Brannmur	Konfigurasjon av brannmur med UFW [89]

Overordnet analyse av sikkerheten:

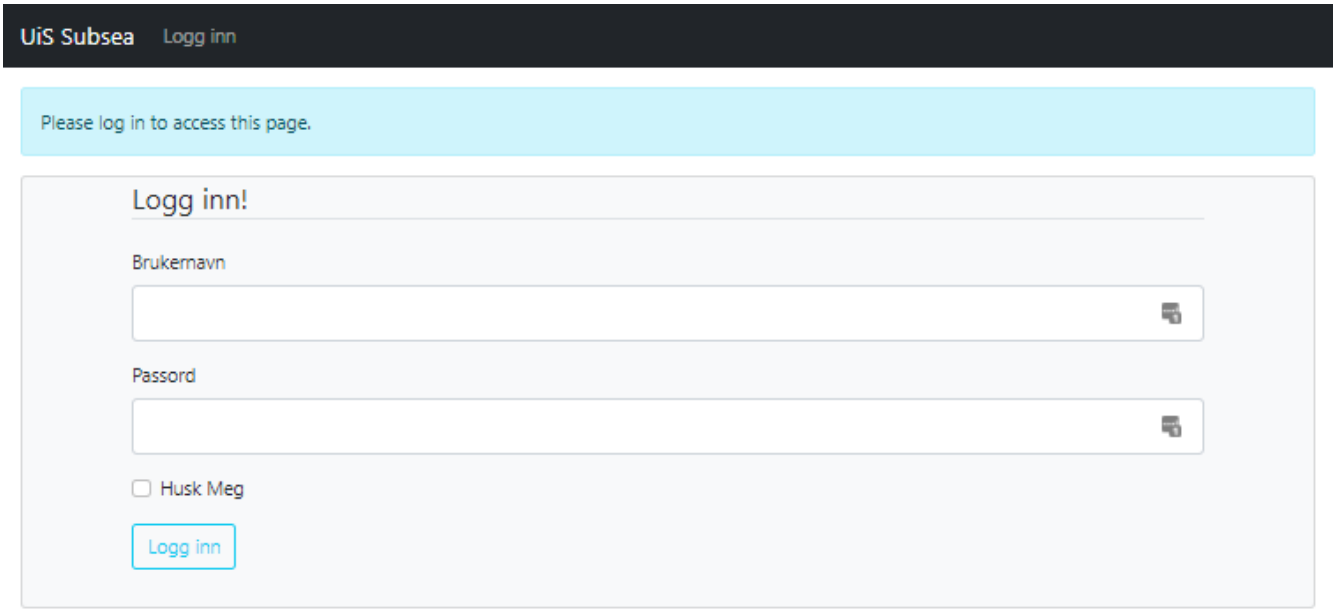
Adgang til brukergrensesnitt er godt sikret med et sterkt verifikasjonsledd. Hvis dette leddet svikter er systemet svært sårbart. Brukeren får da en relativt åpen tilgang til systemet og kontroll over ROV-en. En ondsinnet bruker kan for eksempel manipulere innholdet i JSON-filene som blir sendt til ROV-en. Dette kan åpne for uante konsekvenser både på ROV og Python-server. Når systemet er operativt må ROV-en overvåkes av en operatør. Denne operatøren har da anledning til å kutte strømmen til systemet, og hente ROV-en inn ved hjelp av navlestrengen.

Systemet er altså godt sikret ved hjelp av en solid verifikasjonsbarriere, men mangler tilstrekkelig sikkerhetstiltak som kan kompensere for et eventuelt brudd av denne barrieren. Denne svakheten kompenseres for med manuell overvåking ved bruk.

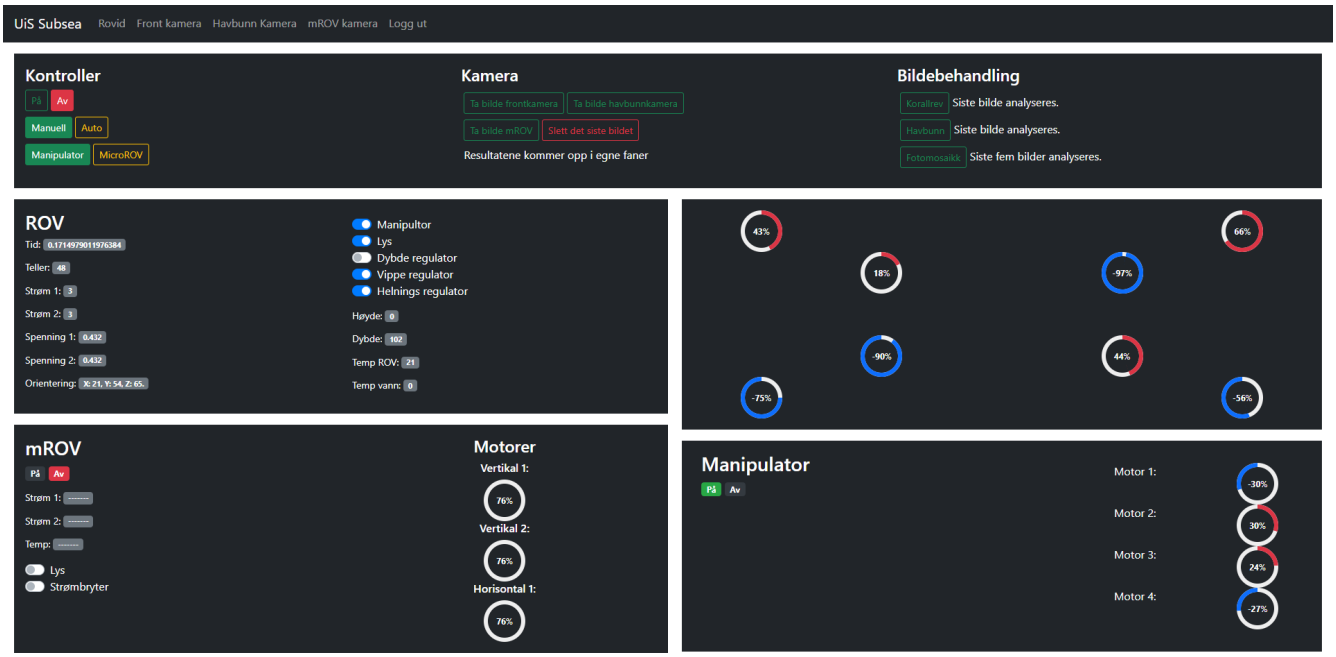
7.3.3 JavaScript, HTML og CSS

HTML

Innloggingsiden blir vist i figur 166. Brukergrensesnittsiden blir vist i figur 167. All HTML-koden ligger i den vedlagte filstrukturen under /Brukergrensesnitt/templates.

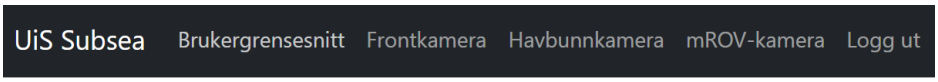


Figur 166: Siden for innlogging.



Figur 167: Brukergrensesnittet.

Når en bruker har logget seg inn får han tilgang til menylinjen vist i 168 og alle stiene vist i 164.

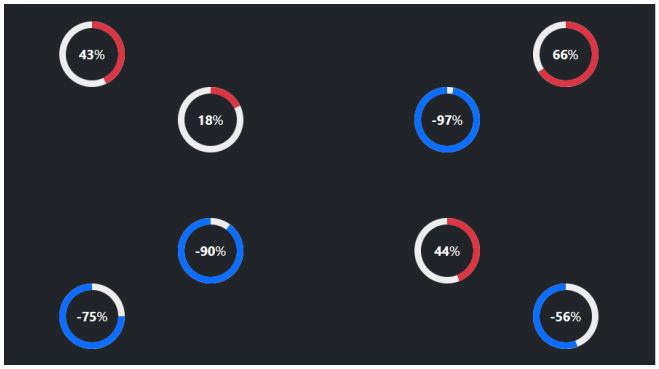


Figur 168

- UiS Subsea - Leder til UiS Subsea sin hjemmeside.
- Brukergrensesnitt - Leder til siden for brukergrensesnittet.
- Frontkamera - Leder til Go-serveren sin side for sanntidsvideo fra frontkameraet.
- Havbunn - Leder til Go-serveren sin side for sanntidsvideo fra havbunnkameraet.
- mROV-kamera - Leder til Go-serveren sin side for sanntidsvideo fra mikro-ROV-en
- Logg ut - Logger brukeren ut av nettsiden.

CSS

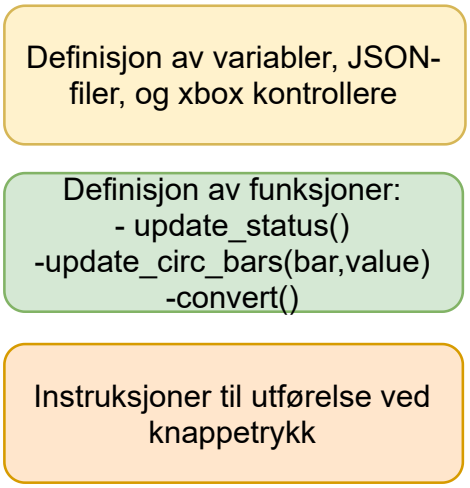
På grafisk design fronten har vi tatt i bruk et polulært bibliotek som heter Bootstrap [2]. I tillegg har vi laget noe spesialdesignet CSS-kode. Denne koden er fokusert på konstruksjonen av elementene som illustrerer motorenes pådragsverdier. Bruken av dette er vist i figur 169. Koden ligger i den vedlagte filstrukturen under /Brukergrensesnitt/static/css.



Figur 169: Illustrasjon av ROV-ens motorpådrag. De fire innerste sirklene representerer de fire vertikalt rettede motorene. De fire ytterste sirklene representerer motorene i det horisontale planet.

JavaScript
JavaScript brukes kun til hovedsiden for brukergrensesnittet */gui* (vist i figur 167). Koden brukes til å gjøre innholdet på nettsiden dynamisk. Den lytter etter Xbox-kontrollere, definerer hva som skal skje når de forskjellige knappene trykkes på og fyller HTML-objektene opp med den mest oppdaterte informasjonen fra ROV/mikro-ROV. All JavaScript-koden har blitt samlet i én fil. Denne ligger i den vedlagte filstrukturen under /Brukergrensesnitt/static/JavaScript.

1. Inneholder definisjon av diverse variabler, Xbox-kontrollere og de felles JSON-filen som brukes til kommunikasjon mellom nettside, server og ROV/mikro-ROV.
2. Den neste delen inneholder definisjoner av noen viktige funksjoner. *update_status()* er hovedfunksjonen som kommuniser med Python, Xbox-kontrollerene og fyller siden med innhold. *update_circ_bars()* brukes til å legge motorpådragsverdiene inn i objektene vist i figur 169. *convert()* er levert av motorgruppen og konvertere verdier fra Xbox-kontrollerens styrestikker til korrekt format før de sendes til ROV-en.
3. Den siste delen av filen inneholder definisjoner av hva som skal skje når de forskjellige knappene blir tatt i bruk.



Figur 170: Vi har samlet all JavScriptkoden i én fil. Denne filen har vi kalt uis_subsea2021.js og består av tre deler.

update_status() er hovedfunksjonen som brukes til å opprettholde en kontinuerlig kommunikasjon ned til ROV/mikro-ROV-en. Den er så viktig for styringsprogrammet av vi går detaljert inn i denne på neste side. For å kommunisere mellom Nettsiden og Python bruker vi Jquery sine *POST* og *GET* metoder til å kalle metodene/stiene i Flask rammeverket.

update_status():

a) Ved oppstart av brukergrensesnittet i nettleseren kalles *../init_kamera* og *../status_data*. Når *../init_kamera* kalles opprettes det en tilkobling til kameraene fra Python. *../status_data* returnerer en pakke med informasjon om ROV-ens nåværende tilstand.

b) Hvis startknappen har blitt trykt på, går vi videre. Hvis ikke går vi til punkt i).

c) Hvis modus for automatisk kjøring er aktivert, kalles *../auto* metoden.

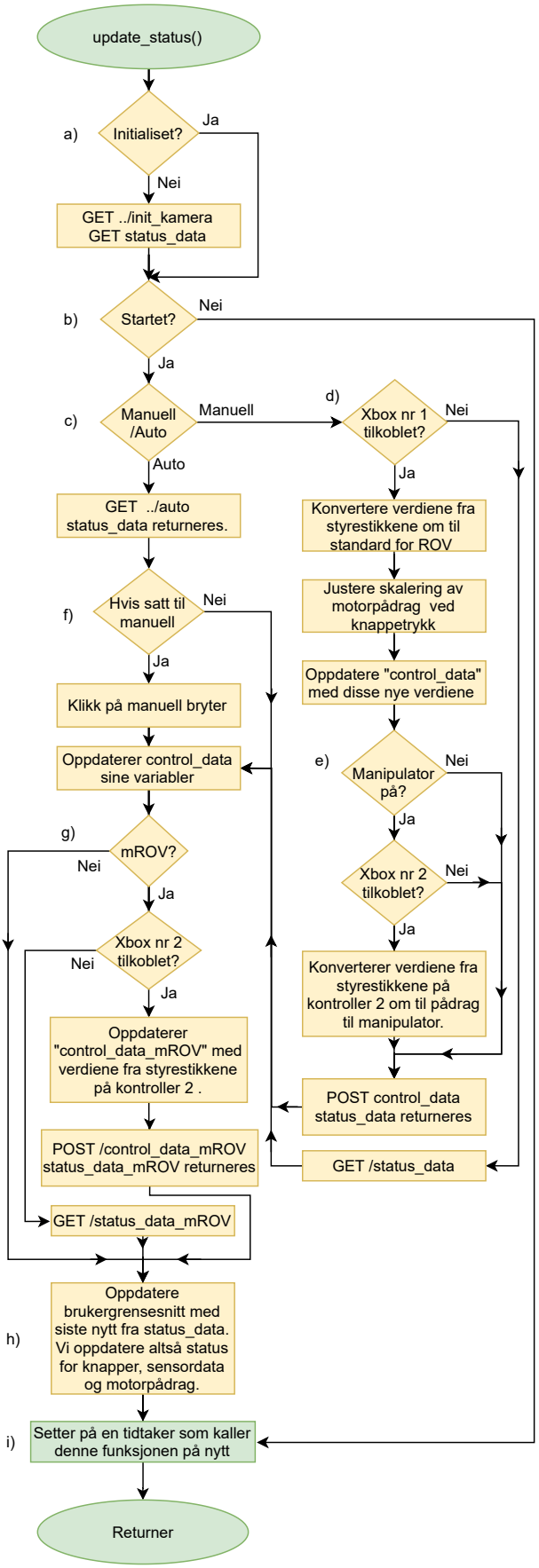
d) Hvis ROV-en er i manuell modus sjekkes det først om det eksisterer en Xbox-kontroller. Er dette tilfellet, hentes verdiene fra styrestikkene. De konverteres videre til korrekt format. Samtidig hentes status for eventuelle inntrykte knapper på kontrolleren. Har det blitt tilordnet en funksjon/variabel til knappene, utføres/endres disse. Den lokale kopien av *control_data* oppdateres med de nye verdiene

e) Er manipulatoren aktivert, sjekkes det om vi har en annen Xbox-kontroller tilgjengelig. Om vi har det, hentes data fra kontrollerens styrestikker og gjøres om til pådragsverdier til manipulatorens motorer.
Deretter sendes *control_data* til Python og videre til ROV-en. Alternativt, om ingen andre Xbox-kontrollere var tilkoblet, hentes *status_data*.

f) Hvis mikro-ROV-en er tilkoblet sjekker vi om en annen Xbox-kontroller er tilkoblet. Om dette er tilfellet, hentes styrestikkenes posisjon og *control_data_mROV* oppdateres med pådragsverdiene til mikro-ROV-en. Videre sendes dette til Python, og derfra til mikro-ROV-en.
Om Xbox-kontroller nummer to ikke var tilkoblet hentes *status_data_mROV*.

g) Alt innholdet på nettsiden oppdateres med data hentet fra ROV/mikro-ROV.

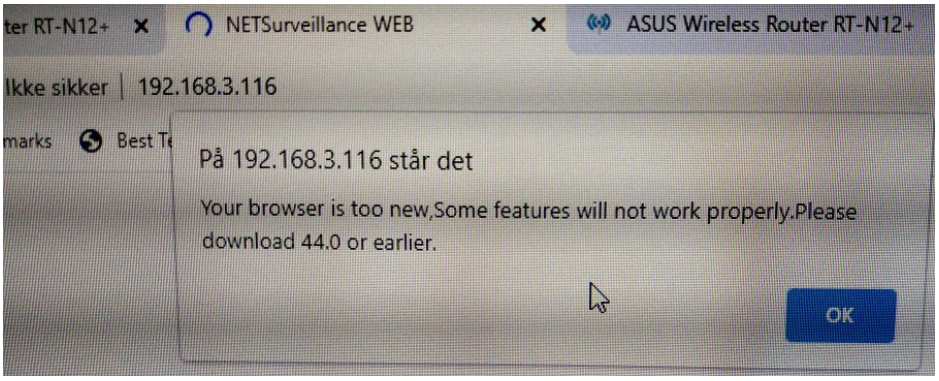
h) *setTimeout*-funksjonen benyttes til å kalle denne funksjonen på nytt etter et valgfritt tidsintervall.



Figur 171

7.3.4 Sanntids video

RTSP:
De fleste IP-kameraer i dag har innebygd mulighet for å se videostrømmen direkte i en nettleser. Det kameraet som har blitt valgt, har i utgangspunktet denne muligheten. Videostrømmen går via "Real-time Transfer Protocol" (RTP) og bruker "Real Time Streaming Protocol" (RTSP) for styring. RTP er protokollen for å overføre selve videostrømmen, mens RTSP gir mulighet for styring som: start, stopp, spill av og pause. Dessverre så fungerer ikke løsningen, som produsenten av kameraet har laget, noe særlig bra. Kameraet krever installasjon av en utdatert utvidelse, som ikke passer med moderne nettlesere, for å kunne strømme til nettleseren. Figur 172 viser resultatet av et forsøk på å koble seg direkte til kameraene via nettleseren.



Figur 172: Viser resultatet av forsøk på å se videostrøm direkte fra kameraet til nettleseren. Den innebygde funksjonen er så gammel at den ikke er kompatibel med moderne nettlesere.

Istedenfor å teste forsinkelsen via nettleseren har vi benyttet oss av VLC Media Player[90]. Testresultatene ligger i tabell 3.

protokoll	Før [s]	Etter [s]	Forsinkelse [ms]
RTSP	02.360	03.090	730
RTSP	06.760	07.560	800
RTSP	09.950	10.680	730

Tabell 3: Resultat fra testing av forsinkelsen til RTSP.

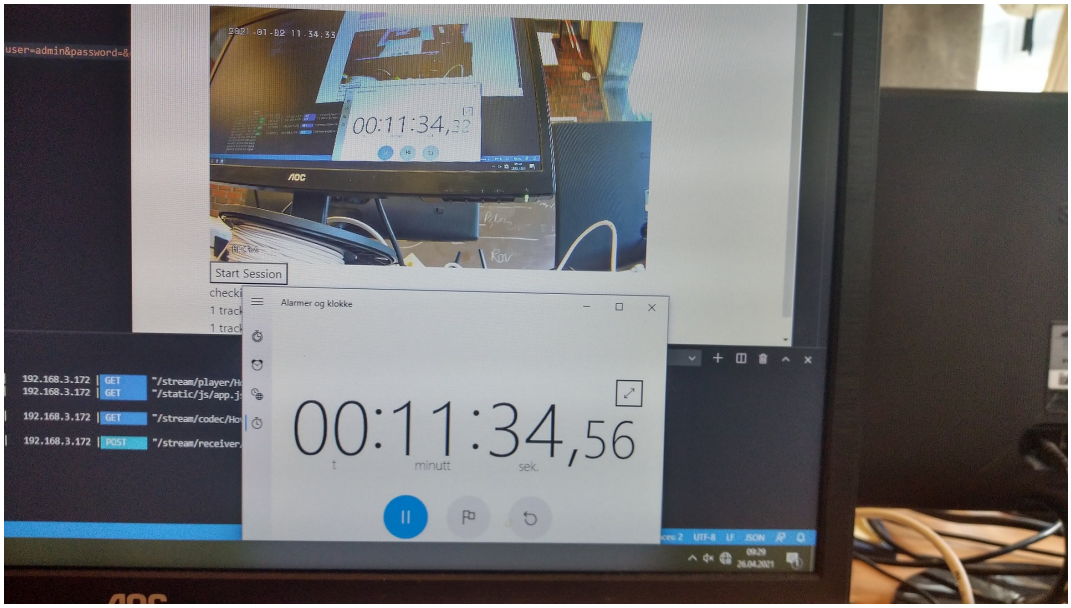
For RTSP vil det normalt være en forsinkelse på mellom 400 og 800 ms. Dette er avhengig av hva slags program som brukes til fremvisning av videostrømmen og bildets oppløsning. Resultatene i tabellen ovenfor indikerer en forsinkelse på oppimot 800 ms.

WebRTC

For å oppnå lavere forsinkelse har vi testet ut en annen protokoll som heter "Web Real-Time Communication" (WebRTC). Med denne metoden går videostrømmen via en egen server, før den sendes ut i nettleseren. Figur 173 viser ett av bildene vi tok mens kameraet filmet en skjerm med stoppeklokke. Tabell 4 viser resultatene fra tester med WebRTC direkte i nettleseren.

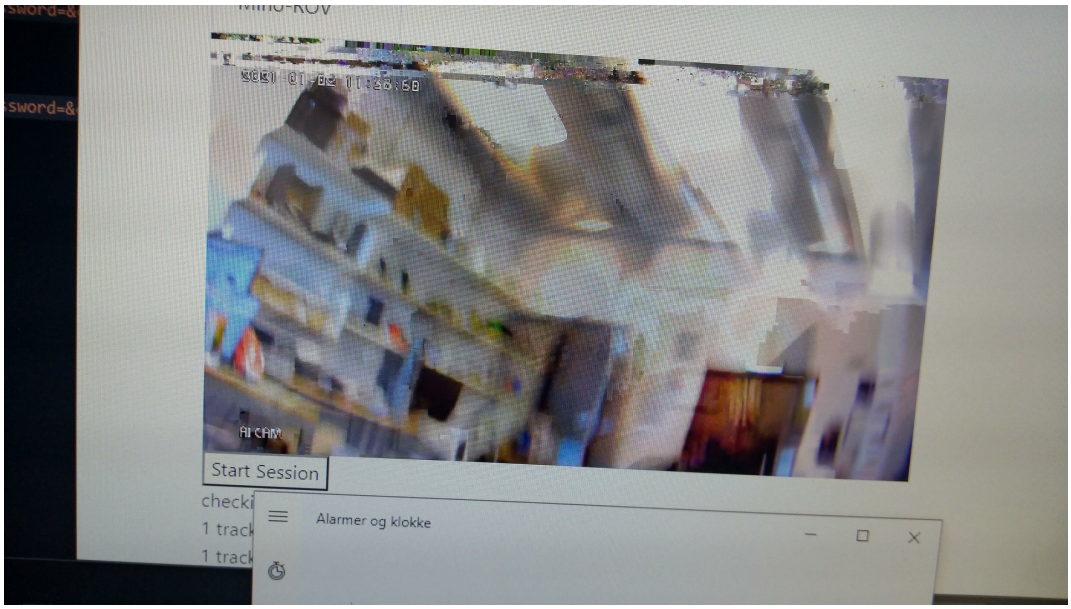
protokoll	Før [s]	Etter [s]	Forsinkelse [ms]
WebRTC	34.320	34.560	240
WebRTC	05.970	06.200	230
WebRTC	10.710	10.970	260

Tabell 4: Resultat fra testing av forsinkelsen til WebRTC.



Figur 173: Test av WebRTC forsinkelse 1.

En av ulempene med WebRTC er at det er komplisert å sette opp. I tillegg vil bildet vil noen ganger stoppe opp og gi en spøkelsesaktig effekt. Dette er et fenomen som er vanskelig å få på bilde, men figur 174 prøver å vise dette. Dette er antageligvis forårsaket av at WebRTC ikke kontrollere at hvert bilde er komplett. Dette er også en av grunnene til at denne protokollen/metoden er raskere enn RTSP. RTSP kontrollerer at alt innhold i hvert enkelt bilde er med. [21]



Figur 174: Figuren viser hvordan bildet kan bli revet opp ved bruk av WebRTC.

Valgt løsning
I de foregående punktene fant vi ut at WebRTC gir en lavere forsinkelse på videostrømmen enn RTSP. På dette grunnlaget velger vi en løsning som benytter seg av WebRTC formatet. Dette blir en relativt komplisert operasjon, altså og transformere videostrømmen fra RTSP- til WebRTC-format. Vi benytter derfor en eksisterende løsning[18]. Denne løsningen er basert på programmeringsspråkene Go (beskrevet i kapittel 7.2.4) og JavaScript. Den tar inn IP-kameraene sin RTSP videostrøm og gjør denne om til WebRTC før den vises direkte i nettleseren.

7.4 Testing og resultat

Styringsprogrammet har ikke blitt testet på en operativ ROV. Gjennom utviklingsprosessen har hver enkelt av komponentene blitt testet. På denne måten vet vi at alle komponentene fungere. Som en kompensasjon for testing mot ROV-en har vi simulert ROV-ens respons ved å la Python-serveren lese og skrive til seg selv istedenfor til ROV-en. Denne testingen har gitt gode resultat. Vi har ikke priortert og dokumentere dette. Styringsprogrammet som ligger vedlagt er satt opp på denne test måten. I vedlegget ligger det en instruksjon for hvordan programmet kan tas i bruk. Denne instruksjonen tar ikke hensyn til alle potensielle problemer, som kan oppstå ved installasjon av Python og andre avhengigheter. For å se hvordan styringsprogrammet presterer kan det vedlagte programmet testes. Programmet er satt opp for testing uten ROV-en tilkoblet.

8 Prosjektledelse

I kapittel 1.4 introduseres organisasjonsstrukturen. Organisasjonen er delt opp i et styre og en prosjektledelse. Kort fortalt har styret ansvar for det utadvendte arbeidet, mens prosjektledelsen har ansvar for den interne driften av det tekniske prosjektet. Detaljert informasjon om denne strukturen ligger i UiS Subsea sin styreinstruks (under Andre_doukumenter i vedlegget). Denne er hovedsaklig utarbeidet av Geir-Arne Solland Kindingstad og Daniel Vasshus.

8.1 Prosjektledelsen:

Prosjektledelsen består av fem personer:

- **Prosjektleder:** Jens Trydal
- **Teknisk ansvarlig:** Sindre Fjermedal
- **Lagleder elektro:** Markus Haldorsen
- **Lagleder maskin:** Joachim Merenyi
- **Hjelper:** Alexander Falch Voerman

I den vedlagte styreinstruksen beskrives rollene mer detaljert.

Hovedoppgavene til prosjektledelsen er:

- Prosjektfremgang.
- Finne en passende konkurranse laget skal delta i.
- Passe på at prosjektet blir forsvarlig styrt.
- Ansvar for at alle studentene er med og drar i riktig retning.
- Påse at alle krav og retningslinjer følges.

8.2 Styringsverktøy

For å følge opp prosjektet har vi tatt i bruk følgende styringsverktøy:

- Ukentlige statusmøter med representanter fra alle bachelorgruppene.
- Planleggingsmøter med prosjektledelsen.
- Styringsdokument som beskriver planlagt og faktisk fremdrift.
- Individuell oppfølging av bachelorgruppene.

8.2.1 Ukentlige statusmøter:

De ukentlige statusmøtene har en fast struktur. Først informeres det om viktige hendelser som alle må kjenne til. Derretter skal hver enkelt gruppe svare på de tre følgende spørsmålene:

- Hva har blitt gjort denne uken?
- Har det oppstått noen hindringer for arbeidet?
- Hva skal bli gjort neste uke?
  - Åpning for spørsmål til gruppen som har ordet.

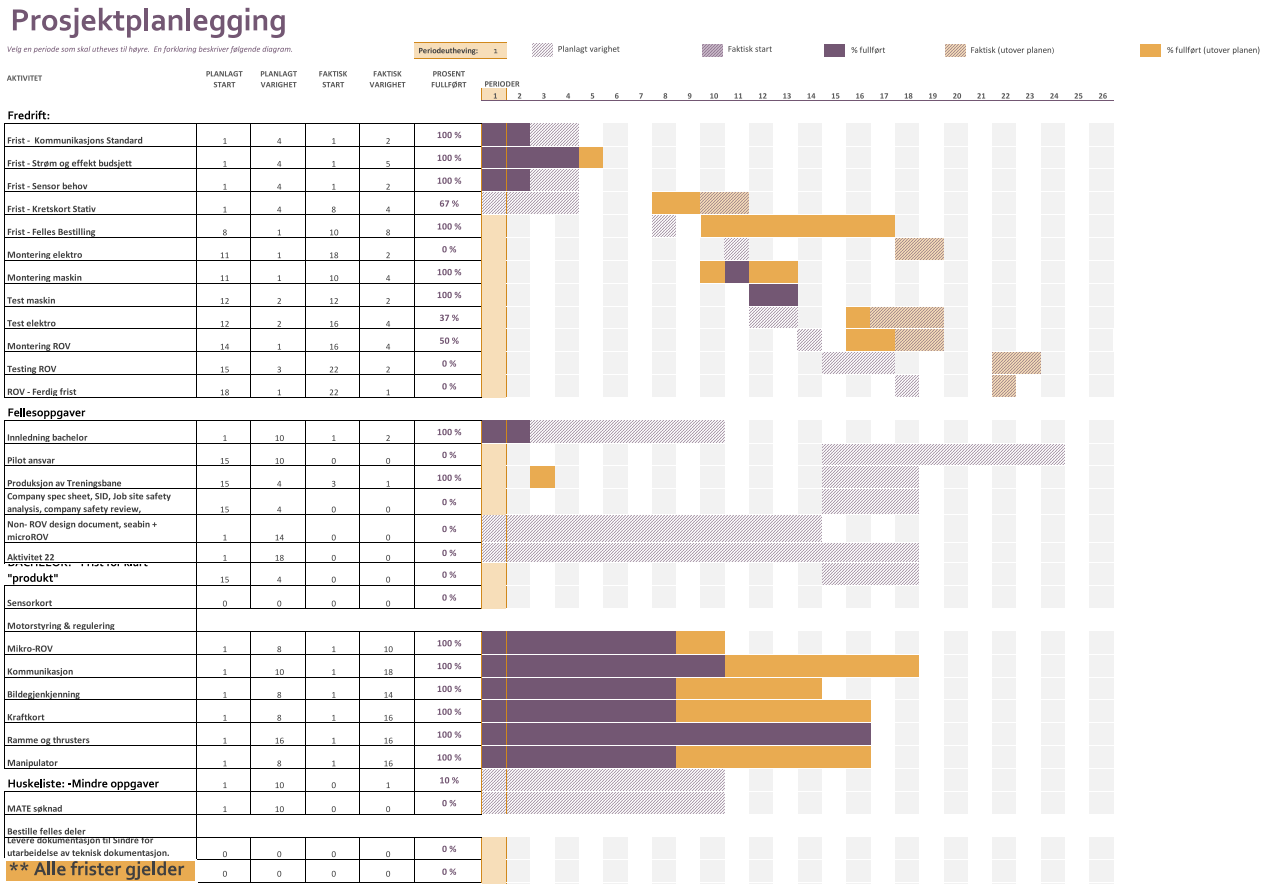
Etter at hver gruppe har avlagt rapport åpnes det for diskusjon og spørsmål om større saker. Møtet avrundes med en oppsummering av de viktigste punktene som har blitt avgjort. Etter møtet publiseres et notat som inneholder hva som ble sagt og vedtatt i løpet av møtet. Denne statusinformasjonen blir brukt til å oppdatere Gantt-diagrammet som beskriver prosjektets fremdrift. Dette møtet er det viktigste styringverktøyet og samlingspunktet for prosjektet.

8.2.2 Planleggingsmøter med prosjektledelsen

Disse møtene arrangeres når det oppstår behov for å fordele oppgaver, justere fremdriftplanen eller andre hendelser som krever behandling i prosjektledelsen før det tas videre med hele gruppen.

8.2.3 Fremdriftsplan:

Som styringsverktøy for fremdriften har vi laget et overordnet Gantt-diagram. Dette dokumentet beskriver den planlagte og faktiske fremdriften til prosjektet. Dokumentet holdes kontinuerlig oppdatert med status fra prosjektet. I de tilfellene hvor det oppstår store avvik fra original fremdriftsplan, utarbeides det en ny revidert fremdriftsplan. Et av Gantt-diagrammene fra sluttfasen i prosjektet, kan ses i figur 175.



Figur 175: Bildet viser et Gantt-diagram fra underveis i prosjektet. Diagrammet ligger også i vedlegget under Andre_dokumenter.

8.2.4 Individuell oppfølging

I tillegg til styringsverktøyene har prosjektleder jevnlig kontaktet hver enkelt gruppe for å prate om hvordan det går. Hvordan arbeidet går, motivasjonen er, samarbeid går, bekymringer osv. Dette har vært en god strategi for å ha en oversikt over hvordan det virkelig går ute i prosjektet. Denne proaktive oppfølgingen har gjort at vi tidlig har kunnet identifisere utfordringer og styrke samarbeidet.

8.3 Erfaringer

8.3.1 Prosjektledelsesstrukturen

På tross av en struktur med flere underledere har prosjektlederrollen endt opp med å bli den som har koordinert og ledet arbeidet mellom gruppene. Årsaken til at det har vært vanskelig å delegere ned til underlederne er sammensatt.
Medvirkende årsaker:

- Relativt liten gruppe i forhold størrelsen på ledergruppen.
- Prosjektleder har arbeidet nært med mange av gruppene. Og på denne måten hatt god oversikt.
- Prosjektleder har vært proaktiv og involvert seg direkte når behovet har vært til stede.
- Koronasituasjonen har ført til mindre naturlig nærkontakt med gruppene.
- Uerfarne medlemmer i prosjektledelsen som ikke har vært tilstrekkelig proaktive med tanke på rollene sine.

Prosjektledelsesstrukturen har altså ikke fungert helt som ønsket. Målet har vært at prosjektlederen skal få rapporter fra underlederne og disse skal ha kontroll på koordineringsarbeidet. Prosjektlederes hovedoppgave skulle da være å ta beslutninger, delegere oppgaver og rapportere til styret. I praksis har leddet med underleder ofte blitt hoppet over, og kommet inn i etterkant. Dette er et læringspunkt for alle involverte parter.

8.3.2 Kommentar til samarbeidsutviklingen og oppfølgingsbehov

I startfasen av prosjektet var det nødvendig med mye oppfølging og svært aktiv koordinering. Etter at startfasen til prosjektet var fullført, kom gruppene inn i arbeidsrytmen, kjente sine arbeidsoppgaver og samarbeidspartnere. Dermed gikk prosjektet mye lettere og det ble ikke like stort behov for oppfølging og kontroll fra prosjektledelsen.

8.3.3 Prosjektet fremdrift og resultat

Prosjektet har i det store og hele gått over all forventning. Men det har oppstått noen stor utfordringer underveis.

- **Koronasituasjonen:**
  Koronasituasjonen har gjort det umulig for hele gruppen å kunne samles. Dette har vanskeliggjort samarbeidet mellom gruppene. I tillegg har det vært stor usikkerhet rundt gjennomføringen av, og viktige tidsfrister til, MATE-konkurransen.

- **Anskaffelse og produksjon av elektronikk:**
  Elektronikken som har blitt laget til ROV-en har blitt produsert i Kina. Den pågående handelskrigen mellom USA og Kina har vanskeliggjort leveransen av viktige deler til produsenter i Kina. Dette førte til store forsinkelser i produksjon og leveranse av elektronikk til oss i UiS Subsea.
  Dette er hovedårsaken til av ROV-en ikke har blitt operasjonell til den planlagte tidsfristen. Dette har gjort det vanskelig å få med testresultater i flere av bacheloroppgavene.

- **Utfordringer med valgt kommunikasjonsløsning:**
  Det ble tidlig avgjort at vi skulle gå for en løsning som benyttet en såkalt "Internet Offload Processor" (IOP)til kommunikasjon mellom kretskortene og brukergrensesnittet. Kommunikasjonsgruppen skulle fokusere på utvikling og testing av denne løsningen. Løsningen som ble laget oppfylte ikke de forventningen vi hadde til kommunikasjonshastighet. Dermed ble det stilt strengere krav til løsningen for brukergrensesnittet.

Overordnet vurdering av prosjektet:

Årets UiS Subsea prosjekt har levert en komplett ROV designet fra bunn og opp. Prosjektmedlemmene har på det tekniske og faglige plan levert over det jeg som prosjektleder har forventet. Prosjektet har blitt forsinket, men dette har hovedsakelig vært på grunn av eksterne faktorer.

De ukentlige møtene har fungert svært godt, og gjort det mulig med godt samarbeid på tross av koronasituasjonen. Gantt-diagrammet ble brukt som et utgangspunkt i starten av prosjektet, men da de store eksterne forsinkelsene inntraff, falt den opprinnelige planen bort. Etter dette var det ikke mulig å fastsette en nye plan da leveringen stadig ble utsatt.

I startfasen av prosjektet ble det mye ekstraarbeid for prosjektledelsen, men dette bedret seg og gruppen har samarbeidet godt.

9 Refleksjon og erfaring

9.1 Havbunn

9.1.1 Identifisere objektene fra hele ROI, eller fra hver enkelt rute

Tidlig i utviklingen av benyttet vi en metode som fant objektene direkte i ROI, uten å dele opp i ruter. Denne metode fungerte godt. Deretter endret vi programmet til slik det er nå. Altså at ROI splittes opp i ruter, og hver enkelt rute undersøkes hver for seg selv.

Dette gjorde vi for å forberede en eventuell kombinasjon av automatisk kjøring og kartlegging av havbunnen. Tanken var at mens ROV-en kjørte over havbunnen kunne programmet lagre rute for rute. For så å til slutt analysere alle rutene og produsere ett kart.

Denne tenkte kombinasjon av automatisk kjøring og kartlegging av havbunnen utforsket vi ikke videre. Hovedgrunnen var et ønske om ikke å gjøre oppdateringshastigheten til automatisk kjøring enda dårligere. I tillegg anså vi løsningene som mer robuste om vi splittet dem opp i to deler til slik de er nå. En ulempe med å dele opp i ruter er at vi kun detekterer objekter som er plassert korrekt inne i rutenettet.

9.1.2 Metode for identifikasjon av svamp

For å finne objektene på havbunnen testet vi flere fremgangsmåter. Blant annet testet vi ut HoughCircles[47] til å lete etter den sirkelformede svampen. Vi erfarte her at metoden ble lite robust. Blant annet klarte den ikke å håndtere bilder hvor svampen var plassert langt fra bildetakningsposisjonen. I disse tilfellene så ikke svampen ut som en sirkel , men heller en søyle. I tillegg krevde HoughCircles mye finjustering av parameterne. Vi gikk derfor bort fra denne metoden. Vi valgte istedenfor en enklere tilnærming basert på konvekst areal, soliditet, lengde og bredde.

9.1.3 Justering av grenseverdier for objektgjenkjenning på havbunnen

For å bestemme parameterene til de forskjellige deteksjonsmetodene brukte vi et sett med prøvebilder. Vi kjørte disse gjennom en testalgoritme og justerte så parameterne manuelt. Etter manuell justering fikk alle testbildene fikk korrekte resultat. Alternativt til manuell gjennomgang, kunne vi ha brukt statistiske metoder til å justere parameterne.

En annen videreutvikling eller forbedring av dette kunne vært å bruke maskinlæring til å gjenkjenne objektene. Vi ville da ha laget en liste med egenskaper eller punkter for hvert bilde. Vi kunne så ha brukt testbilder til å lære programmet å gjenkjenne objektene. Årsaken til at vi ikke har kikket nærmere på dette er at vi ikke har hatt nok tid eller kjennskap til konseptet.

9.1.4 Forbedring av automatisk kjøring

Ved testingen av målemetodene til automatisk kjøring (se konklusjon i kapittel 4.2.7) undersøker vi hva som skjer med målingene når bildet roteres. Vi fant da et det var en systemisk sammenheng mellom bildets rotasjon og målingene. Dette er dermed et punk som bør utforskes videre. Programmet for automatisk kjøring er spesielt avhengig av gode og stabile målinger da vi ikke får anledning til å filtrere målingen (se kapittel 4.2.9).

Enda bedre testing av metodenes totale treffsikkerhet og riktighet hadde vært ønskelig. Hadde vi kjent disse verdiene hadde vi hatt et bedre grunnlag til å vurdere det faktiske behovet for filtrering.

9.2 Helsen til korallrev

9.2.1 Feilaktig antagelse til grunn for hele løsningen

Tidlig i prosjektet ble det lagt til grunn en antagelse om at vi ikke behøvde å markere hvite korallrevdeler som døde. Denne antagelsen var basert på en informasjonsvideo fra MATE-ROV. Denne antagelsen har vist seg å være basert på en alternativ tolkning av oppgavebeskrivelsen. Antagelsen er altså ikke i henhold til MATE-ROV sin oppgavebeskrivelse. Det er naturlig å tolke det slik at koralldeler vil dø etter et år med bleking. Og dette bør selvfølgelig registreres. Feilen ble oppdaget så sent i prosjektet at vi ikke fikk anledning til å utbedre løsningen. Derfor ligger antagelsen fortsatt til grunn for løsningen.

Dersom en hvit del av korallrevet forsvinner/dør, vil ingen av løsningene merke dette. Alle løsningene baserer seg på å fargemaskere(se seksjon 3.1) ut de lilla delene av korallrevet. Med denne metoden oppdager vi alle endringer, bortsett fra hvitt korallrev som forsvinner.

Det å finne hvitt korallrev på en robust måte kan være utfordrende. Gitt at bakgrunnen i bassenget er tilstrekkelig ulik korallrevets hvite deler, kan vi fargemaskere ut de hvite rørene. Alternativt er det mulig å bruke *k-means* til splitte bildet opp i tre fargegrupper: lilla, hvit og bakgrunn. Med fargene separert er den en enkel sak å modifisere resten av algoritmene.

9.2.2 Flere alternative løsninger på korallrevoppgaven

Det er lurt å planlegge hele løsningen, og se etter eventuelt ødeleggende svakheter, før man graver seg for dypt ned i utviklingen. Korallrev programmene er et eksempel på en slik tabbe. Vi hadde til å begynne med en brukbar løsning med malsammenligning. Men vi følte at den ikke var tilstrekkelig til å løse oppgaven. Vi startet derfor på *k-means* løsningen. Desverre gjorde vi ikke en godt nok forhåndsvurdering på om dette ville bli en bedre løsning. Vi burde altså ha gjort bedre forarbeid å slik spart tid.

9.2.3 Litt om maskinlæring

Den beste løsningen kunne vært å basere seg på maskinlæring. Vi valgte tidlig en annen retning for oppgaveløsningene. Noe av årsken var at maskinlæring mye dataressurser og det kan være arbeidskrevende å lære opp algoritmen. For å løse oppgaven med maskinlæring måtte vi hatt flere hundre bilder av forskjellige deler av korallrevet. Selv om programmet hadde klart å finne objektene/koralldelene, trenger det fortsatt å definere endring. Maskinlæringen kunne da erstattet malsammenligningsmetoden i løsningen.

9.2.4 Kombinere løsninger

Gjennom bacheloren har vi laget to måter for å undersøke helsen til korallrev. Dersom vi hadde hatt mer tid kunne vi prøvd å kombinere malsammenligning, *k-means* og grafteori. Vi kunne brukt grener gitt av *k-means*-metoden og sett etter forskjeller med malsammenligning. Dette vil gi en mer robust metode ved at man med sikkerhet kjenner endringens posisjon ved hjelp av k-means. Og vi med sikkerhet finner endringstypen ved hjelp av malsammenlikningsmetoden. På denne måten kombinere vi det beste fra hver av løsningene.

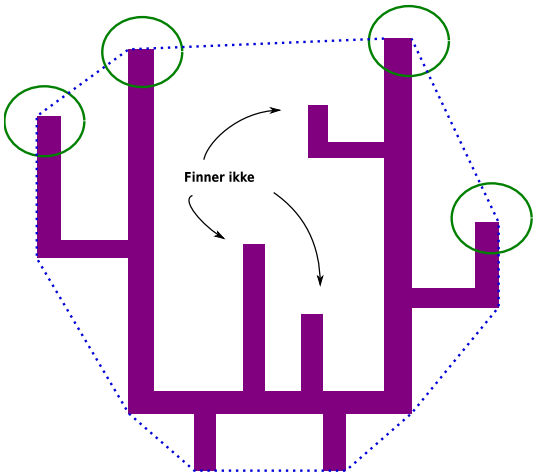
9.2.5 *K-means* hastighet

En av ulempene med å bruke *K-means* algoritmen er at den bruker en del tid på bli ferdig. Tiden avhenger av antall punkter og gjennomkjøringer. En måte å redusere tiden på er å redusere oppløsningen på bildet som skal konverteres til punkter.

9.2.6 Forkastet løsning, ytterpunkter

Denne metoden baserer seg på å bruke konvekst område "convex hull", som er nevnt i kapittel 3.3.12. Vi ønsket bruke denne metoden til å finne tuppene på korallrevet. Dette ville gitt oss referansepunkter på korallrevet. Vi kunne deretter brukt referansepunktene til å sjekke om tuppene var plassert på samme plass i det neste bildet av korallrevet.

Fordelen med denne metoden er den er treffsikker og forutsigbar på måten den finner referansepunktene. Ulempe er at den kun finner utvendige punkter, og ingen innvendige punkter. Prøver man denne metoden på korallrevet vil den finne de ytterste tuppene, men ikke de innerste. I figur 176 illustreres dette.



Figur 176: Figur som viser hvilke deler som blir oppdaget av konvekst område funksjonen. Altså hvilke ytterpunkter som definerer området rundt konturen. Den blå sirkelen viser det konvekse området.

En måte å løse dette på kan være å bruke lokale konvekse områder. Vi bestemte oss for å ikke grave oss for dypt ned i denne metoden da den ikke håndere mange av endringene som er spesifisert i MATE-oppgaven. Derimot så kan den potensielt bli brukt som en delløsning. Den er derfor nevnt kort her.

9.3 Fotomosaikk av T-banevogn

9.3.1 Utklipp av T-banevogn fra omgivelsene:

Den største utfordringen i fotomosaikk oppgaven, var å isolere den hvite T-banevognen fra de lyse omgivelsene. I tillegg til løsningen vi har falt ned på, har vi utforsket og testet andre løsninger.

Fargemaskering
Med denne metoden forsøker vi å isolere ut flaten ved hjelp av fargemaskering(3.1). Fremgangsmåten er som følger:

1. Definerer en maskering som beskriver alt annet enn den hvite sideflaten.
2. Isolerer så ut dette området.

"Terskling"
I denne metoden forsøker vi å isolere ut flaten ved å sette en grenseverdi på hva som skal bli sort, og hva som skal bli hvitt i et sort-hvitt-bilde. Til dette benyttes funksjonen "terskling"(3.2).

1. Gjør om til gråskalabilde.
2. Benytter "terskling" til å isolere ut sideflaten.

Fargemaskering og Kmeans
Dette er et forsøk på å forbedre fargemaskeringsmetoden ved hjelp av grupperingsmetoden Kmeans (kapittel 3.7).

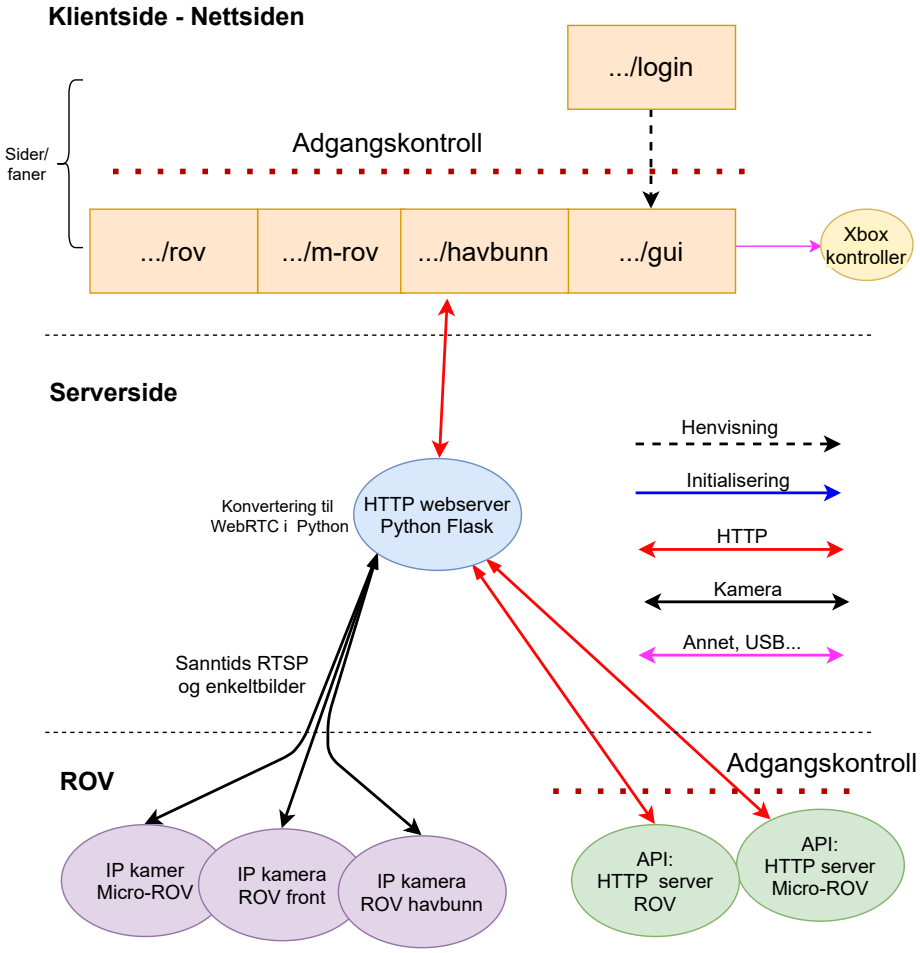
1. Fjerner alle farger unntatt den hvite boksen og den lyse bakgrunnsfargen.
2. Benytter *Kmeans* til å skille de to fargene fra hverandre.
3. Isolerer ut det området som tilhører T-banevognen.

Svakhet ved disse løsningsmetodene
Den felles svakheten og årsaken til at ingen av disse løsningen har blitt brukt, er utfordringer med varierende belysning og lys bakgrunnsfarge. I våre tester har vi benyttet hvit bakgrunn. Antageligvis er bakgrunnsfargen i basseng/vann mørkere og letter å arbeide med en det vi har brukt i testene.
Gitt at en av metodene hadde fungert, ville fordelen vært bedre håndtering av dårlige bilder. Programmene kunne gitt et estetisk vakrere resultat enn den nåværende løsningen. Vi valgte å gå bort fra dem fordi de var svært sårbare for endring i lysforhold og bakgrunnstype. Vår prioritet ble at løsningen skulle være mest mulig robust.

9.4 Styringsprogrammet

9.4.1 Ønsket løsning/forbedring av styreprogrammet

Styreprogrammet slik det er skissert i kapittel 7.3, er ikke helt slik vi ønsker å ha det. Figur 177 illustrer hvordan vi optimalt sett ville hatt programstrukturen.



Figur 177: Bildet viser programstrukturen vi ønsker oss. ROV-ene implementerer adgangskontroll og kan kobles til internett. Programmet som håndtere transformasjon fra RTSP til WebRTC er flyttet til Python. I tillegg har ROV-ene verifikasjon av data og adgangskontroll, slik at den trygt kan kobles direkte til internett.

Vi ønsker altså å integrere løsningen for transformasjon fra RTSP til WebRTC i Python. På denne måten går alt kommunikasjon ut til bruker gjennom samme server. Som et utgangspunkt kan vi hente inspirasjon til dette fra en løsning basert på JavaScript og C++[3] og den løsningen vi faktisk benytter som er basert på Go og JavaScript[18].

I tillegg ønsker vi at ROV-ene har en egen sikker løsning for adgangskontroll slik at den kan kobles direkte til internett. Med denne løsning trenger ikke Python-serveren å være på det samme lokale nettverket som ROV-en, men kan bli flyttet hvor som helst. Det blir dermed lettere å bruke ROV-en da en trenger mindre utstyr lokalt på arbeidsplassen.

9.4.2 Valg av rammerverk til brukergrensesnittet

Til å begynne med kikket vi på å bruke FastApi[87] som nettrammeverk i Python. Hovedårsaken til dette var at denne kunne gjøre samme jobb som Flask[68], men kjappere. Da vi begynte på utviklingen av grensesnittet fant vi ut at vi ikke egentlig hadde et stort behov for denne hastigheten. Flask på sin side var et eldre og mer modent rammeverk. Her fantes det store mengder dokumentasjon og eksempler. I tillegg hadde vi noe erfaring med Flask fra tidligere prosjekter. Det viste seg også at overgangen fra Flask til FastApi kunne gjøres relativt enkelt. Vi valgte derfor å ta utgangspunkt i Flask.

9.5 Generelt

9.5.1 Arbeidsprosessen oppsummert

Arbeidet med oppgaven har foregått i følgende faser:

1. Planlegging og skissering av løsninger.
2. Konstruksjon av testbane for testing av bildebehandlingsdelen.
3. Dypdykk i løsningene, testing og rapportskriving.
4. Ferdigstilling av styringsprogram og rapport.

I vedlegget ligger timelister og statusrapporter fra underveis i prosjektet.

9.5.2 Testing med ROV i basseng

Vi hadde opprinnelig planlagt å teste alle løsningene på ROV-en i et basseng. Hele UiS Subsea prosjektet ble så forsinket at vi aldri fikk anledning til dette. Ved testing i vann er det mulig vi hadde oppdaget nye utfordringer vi i utgangspunktet ikke har tatt hensyn til. For eksempel forventer vi at alle bildene tatt i basseng vil ha et blåskjær. Vi kjenner heller ikke til hvordan bakgrunnen som da er bassengbunn og vegger, vil kunne påvirke løsningene våre. Mest sannsynlig kan eventuelle avvik kompanseres for ved å justere på definisjonene av farger i HSV-formatet. Programmene skal derfor enkelt kunne tilpasses til andre lysforhold og bakgrunner. HSV-fargene som blir brukt av de forskjellige bildebehandlingsprogrammene er definert innledningsvis i hver enkelt fil.

9.5.3 Kompleksitet og robusthet

Gjennom utviklingen av de forskjellige bildegjennkjenningsprogrammene har vi gjort en rekke erfaringer. Den kanskje viktigste erfaringen er at det enkle ofte er det beste. Vi har utforsket og testet en rekke avanserte bildebhandlingsmetoder. Det vi har erfart er at løsningene ikke blir så robuste som vi ønsker. For eksempel så har vi erfart at enkel fargemaskering er et svært robust verktøy som krever lite finjustering. Da vi utviklet algoritmen til å lage en fotomosaikk av en T-banevogn ble dette tydelig. Vi utforsket en rekke metoder for å skille den hvite vognen fra en nokså lys og hvit bakgrunn. Disse metode funket vel og bra, men var sårbare for små endringer. Vi endte opp med å ta tak i det tydeligste ved T-banevognen og bruke dette som et referansepunkt. Vi kikket da etter de fargelagte områdene med fargemaskering. Dette gav oss lærdommen at en enklest mulig løsning er ønskelig.

9.5.4 Avgrensning av oppgaven og stor arbeidsmengde

Denne bacheloroppgaven har blitt svært omfattende. I etterpåklokskapens navn skulle vi vært flinkere til å avgrense oppgavene. For eksempel så burde vi tidligere ha valgt en hovedretning for korallrevoppgaven. Dette ville ha redusert arbeidsmengden betraktelig.

Hele delen om styringsprogrammet var opprinnelig tenkt å være en egen bacheloroppgave. Fordi UiS Subsea hadde behov, tok vi ansvaret for denne delen. I tillegg har vi hatt UiS Subsea sin prosjektleder på gruppen. Dette vervet har krevet sin del i løpet av semesteret.

Vi hadde i utgangspunktet nok arbeid med delene for bildebehandling og autonom kjøring. Hadde vi avgrenset oppgaven til dette hadde vi fått ett mer avslappende semester. Vi kunne også fått anledning til å undersøke flere metoder og teste programmene på mer kvantitative måter.

10 Konklusjon

10.1 Oppsummering

Oppgaven med autonom kjøring og kartlegging av havbunnen ble delt i to oppgaver. Én for autonom kjøring, og én for kartlegging av havbunnen. Begrunnelsen for dette var at vi ønsket en mest mulig robust løsning. Programmet for kartlegging av havbunnen løser den tiltenkte oppgaven når de definerte forutsetningene er tilstede. Når det kommer til bildebehandlingsdelen av autonom kjøring fungerer denne slik den skal ved de gitte forutsetningene. Testresultatene på målingene i bildet viser at treffsikkerheten endres med rotasjon av bildet. Når det kommer til PID-regulatorene som skal holde ROV-en på rett kurs, fikk vi utfordringer med lav oppdateringshastighet på målingene. Dette førte til at vi valgte å ikke benytte filtrering av måleverdiene. Filtreringen førte til at systemene ble ustabile ved denne oppdateringshastigheten. Dette er en svakhet som vil kunne utbedres om oppdateringshastigheten forbedres. Den lave oppdateringshastigheten er forårsaket av treg kommunikasjon mellom styringsprogram, ROV og kamera.

Det å undersøke helsen til korallrev har vært en krevende oppgave. Vi har forsøkt flere fremgangsmåter og metoder. Den beste løsningen er basert på malsammenligning. Denne løsningen er avhengig av gode malbilder, korrekt tatt sammenligningsbilde og korrekt valgt treffprosent. Er disse kravene oppfylt vil programmet antagelig løse oppgaven. Denne optimaliseringen er ikke utført og løsningen vil derfor ikke garantere full uttelling i konkurransen. På grunn av en misforståelse av MATE-oppgaven, legger løsning en feil antagelse til grunn. Løsningen må derfor utbedres for at oppgaven skal bli fulstendig løst. En annen metode er basert på *k-means* og grafteori. Denne løsningen har ikke blitt fullført. Metoden har et stort potensiale, spesielt hvis den kombineres med element fra malsammenligningsmetoden. Det er denne kombinasjonen vi anser som det beste utgangspunktet for en videreutvikling av løsningen.

Fotomosaikkprogrammet løser utfordringen definert av MATE. Det mest utfordrende med oppgaven var å isolere ut en hvit T-banevogn fra en lys bakgrunn. Her undersøkte vi flere alternative løsninger, før vi valgte en metode som benyttet de fargede kantmarkeringer på T-banevognen. Testing viste at løsningen fungerte i henhold til oppgaven, under de gitte forutsetningene.

Vi har ikke fått anledning til å teste løsningene med ROV-en i vann. All testingen har foregått på land. Vi forventer derfor at bildebehandlingsoppgavene behøver justering av noen parametre med tanke på lysforhold og farger.

Styringsprogrammet benytter seg av en Python-server til bildebehandling og kommunikasjon, samt Go-server til sanntids video i WebRTC-format. Vi ønsket opprinnelig å integrere løsningen for sanntids video i Python-serveren, men dette viste seg å være en for omfattende oppgave. Styringsprogrammet har fungert godt under testing, men begrenses av en lav kommunikasjonshastighet ned til ROV-en. Når det kommer til sikkerhet har vi en sterk verifikasjonsbarriere av systemets brukere. Svikter denne barrieren er det lite som beskytter systemet fra ondsinnede aktører.

På prosjektledelsessiden har det blitt benyttet enkle verktøy som ukentlige statusmøter og individuell oppfølging. Disse verktøyene har fungert godt. Prosjektet har møtt på eksterne utfordringer som Koronasituasjonen og handelskrig mellom Kina og USA. Det har ført til et mer krevende samarbeidsklima samt forsinkelser av elektronikkproduksjon. Totalt sett har prosjektet blitt forsinket, men har allikevel kommet i mål og produsert ROV-ene.

10.2 Forbedringsforslag og gjenstående arbeid

Denne seksjonen går gjennom de viktigste forbedringsforslagene og punktene med gjenstående arbeid.

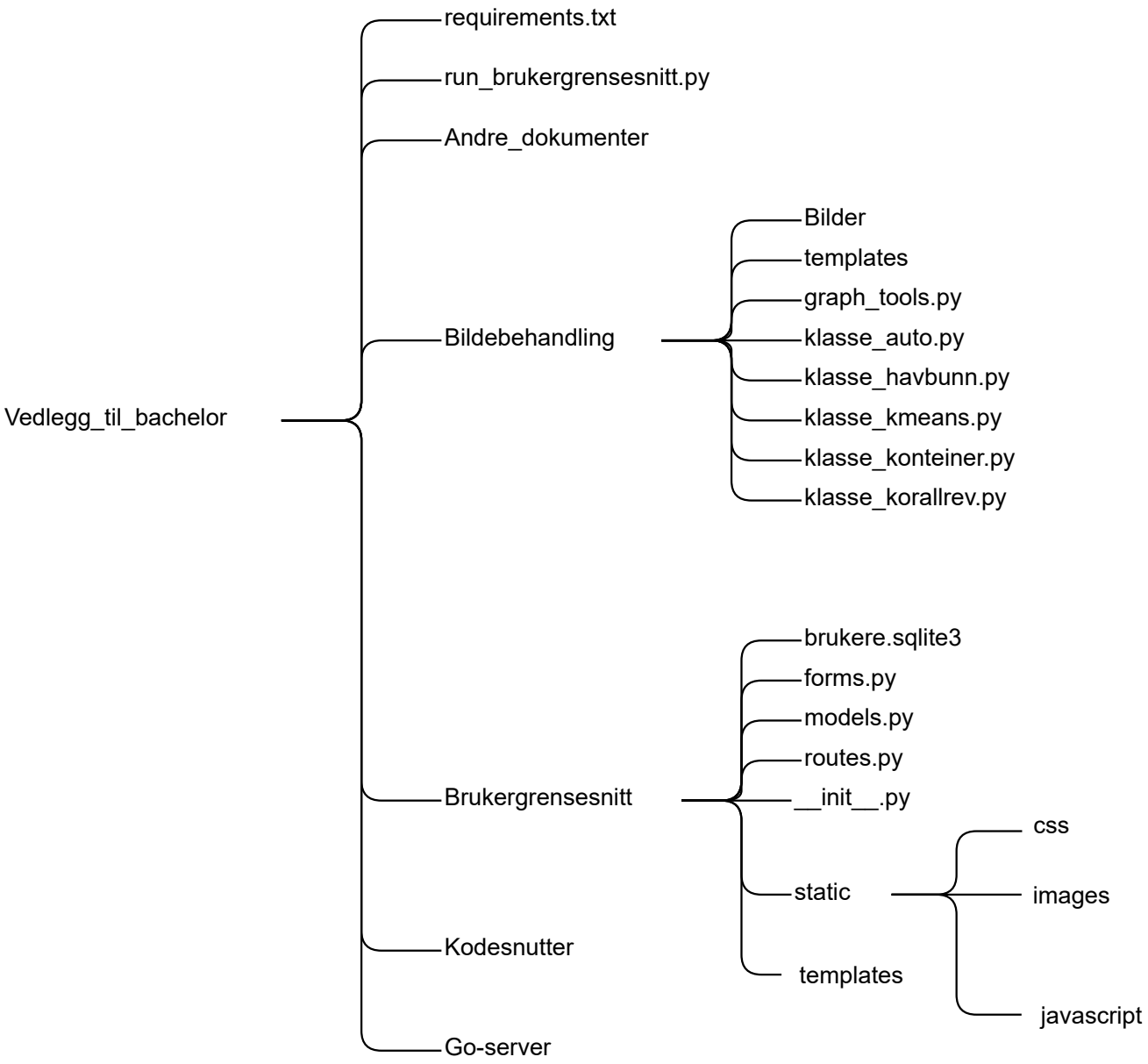
Forbedringsforslag:

- Utbedre kommunikasjonsløsningen mellom brukergrensesnitt og ROV. Flaskehalsen er serveren på ROV-en.
- Forbedre bildetakningshastigheten mellom Python og IP-kameraene. Mulig løsning kan være skifte fra enkeltbilder til å plukke bilder fra en videostrøm. (se kapittel 7.3.1).
- Løse deler av bildebehandlingsoppgavene ved hjelp av maskinlæringsalgoritmer.
- Integrere transformasjon fra RTSP til WebRTC i Python-serveren (se kapittel 9.4.1).
- Mulighet for å tilkoble ROV-ene trygt til internett (se kapittel 9.4.1).
- Forbedre styringsprogrammet slik at det blir mindre sårbart ved sikkerhetsbrudd (se kapittel 7.3.2).
- Forbedrede målemetoder for automatisk kjøring. (se kapittel 9.1.4)
- Utvikle en ny metode for helsen til korallrev basert på grafteori, *k-means* og malsammenligning (se kapittel 9.2.4).

Gjenstående arbeid:

- Modifisere programmet for helse korallrev til å være i henhold til MATE-oppgaven (se kapittel 9.2.1).
- Optimalisere malbilder og treffprosent for helsen til korallrev.
- Teste styringsprogrammet mot ferdig ROV.
- Teste bildebehandlingsprogrammene på ROV-en i vann (se kapittel 9.5.2).
- Teste dynamikken til ROV-en i vann. Dette gir oss korrekte verdier for ROV-ens dynamikk i kapittelet om PID-regulering (kapittel 4.2.8).
- Gjøre brukergrensesnittet mer brukervennlig ved å tilordne flere funksjoner til knappene på Xbox-kontrolleren.
- Fullføre detaljert bruksanvisning og installasjonsprosedyre til styringsprogrammet.
- Utplassere styringsprogrammet på en datamaskin som kan følge med ROV-en.

10.3 Vedlegg



Figur 178: Figuren viser mappestrukturen til vedlegget. De viktigste filene er tatt med i strukturen.

Hele mappen kan lastes ned fra:
`https://github.com/ExtremeOptimist/Vedlegg_til_bachelor`

Referanseliste

[1] Ziyahan Albeniz. *HTTP Security Headers and How They Work*. https://www.netsparker.com/whitepaper-http-security-headers/, [Besøkt 10.05.21].

[2] The Bootstrap Authors. *Bootstrap*. https://getbootstrap.com, [Besøkt 07.05.21]. 2020.

[3] Bhlowe. *streamer*. https://github.com/Bhlowe/streamer, [Besøkt 25.03.21]. 2020.

[4] Google og frivillige bidragsytere. *Programmeringsspråket Go*. https://golang.org/, [Besøkt 07.05.21].

[5] *BlueROV2*. https://bluerobotics.com/store/rov/bluerov2/, [Besøkt 06.01.21].

[6] H.D. Cheng mfl. «Color image segmentation: advances and prospects». I: *Pattern Recognition* 34.12 (2001), s. 2259–2281. ISSN: 0031-3203. DOI: https://doi.org/10.1016/S0031-3203(00)00149-7. URL: https://www.sciencedirect.com/science/article/pii/S0031320300001497.

[7] Benoit Chesneau. *Deploying Gunicorn*. https://docs.gunicorn.org/en/stable/deploy.html, [Besøkt 10.05.21].

[8] Wikimedia Commons. *File:HSV color solid cylinder saturation gray.png — Wikimedia Commons, the free media repository*. https://commons.wikimedia.org/w/index.php?title=File:HSV_color_solid_cylinder_saturation_gray.png&oldid=486761992, [Besøkt 15.02.21].

[9] Wikimedia Commons. *File:K Means Example Step 1.svg — Wikimedia Commons, the free media repository*. https://commons.wikimedia.org/w/index.php?title=File:K_Means_Example_Step_1.svg, [Besøkt 15.02.21]. 2020.

[10] Wikimedia Commons. *File:K Means Example Step 2.svg — Wikimedia Commons, the free media repository*. https://commons.wikimedia.org/w/index.php?title=File:K_Means_Example_Step_2.svg, [Besøkt 15.02.21]. 2020.

[11] Wikimedia Commons. *File:K Means Example Step 3.svg — Wikimedia Commons, the free media repository*. https://commons.wikimedia.org/w/index.php?title=File:K_Means_Example_Step_3.svg, [Besøkt 15.02.21]. 2020.

[12] Wikimedia Commons. *File:K Means Example Step 4.svg — Wikimedia Commons, the free media repository*. https://commons.wikimedia.org/w/index.php?title=File:K_Means_Example_Step_4.svg, [Besøkt 15.02.21]. 2020.

[13] Wikimedia Commons. *File:RGB color circle.png — Wikimedia Commons, the free media repository*. [Online; accessed 8-May-2021]. 2020. URL: %5Curl%7Bhttps://commons.wikimedia.org/w/index.php?title=File:RGB_color_circle.png&oldid=491780966%7D.

[14] Wikimedia Commons. *File:RGB Cube Show lowgamma cutout a.png — Wikimedia Commons, the free media repository*. https://commons.wikimedia.org/w/index.php?title=File:RGB_Cube_Show_lowgamma_cutout_a.png&oldid=472938367, [Besøkt 15.02.21]. 2020.

[15] The MATE ROV Competition. *Competition manual 2020 - Explorer*. http://files.materovcompetition.org/2021/2021_EXPLORER_Manual_8Mar2021.pdf, [Besøkt 15.04.21].

[16] Wikipedia contributors. *HSL and HSV — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=HSL_and_HSV&oldid=1001803663, [Besøkt 15.02.21].

[17] E.R. Davies. «Chapter 3 - Image filtering and morphology». I: *Computer Vision (Fifth Edition)*. Red. av E.R. Davies. Fifth Edition. Academic Press, 2018, s. 39–92. ISBN: 978-0-12-809284-2. DOI: https://doi.org/10.1016/B978-0-12-809284-2.00003-4. URL: https://www.sciencedirect.com/science/article/pii/B9780128092842000034.

[18] deepch. *RTSP to WebRTC*. https://github.com/deepch/RTSPtoWebRTC, [Besøkt 25.03.21]. 2020.

[19] MDM web docs. *Content Security Policy*. https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP, [Besøkt 10.05.21].

[20] MDM Web Docs. *JavaScript*. https://developer.mozilla.org/en-US/docs/Web/JavaScript, [Besøkt 15.04.21].

[21] Flashphoner. *Browser-based WebRTC stream from RTSP*. https://flashphoner.com/browser-based-webrtc-stream-from-rtsp-ip-camera-with-low-latency/, [Besøkt 10.05.21]. 2020.

[22] Flask-Login. *Flask-Login*. https://flask-login.readthedocs.io/en/latest/, [Besøkt 10.05.21].

[23] Electronic Frontier Foundation. *Certbot*. https://certbot.eff.org/, [Besøkt 10.05.21].

[24] Python Software Foundation. *Programmeringsspråket Python*. https://www.python.org/, [Besøkt 18.04.21].

[25] Python Software Foundation. *Python perfcounter*. https://docs.python.org/3/library/time.html#time.perf_counter, [Besøkt 19.04.21].

[26] Allan Hanbury. «Constructing cylindrical coordinate colour spaces». I: *Pattern Recognition Letters* 29.4 (2008), s. 494–500. ISSN: 0167-8655. DOI: https://doi.org/10.1016/j.patrec.2007.11.002. URL: https://www.sciencedirect.com/science/article/pii/S0167865507003601.

[27] Charles R. Harris mfl. «Array programming with NumPy». I: *Nature* 585.7825 (sep. 2020), s. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[28] Finn Haugen. *Derivation of a Discrete-Time Lowpass Filter*. http://techteach.no/simview/lowpass_filter/doc/filter_algorithm.pdf, [Besøkt 01.05.21].

[29] Finn Haugen. *Praktisk Reguleringsteknikk*. 2. utg. Fagbokforlaget, 2009.

[30] Jean. *End-to-end Object Detection with Template Matching using Python*. https://www.sicara.ai/blog/object-detection-template-matching, [Besøkt 07.01.2021]. Jul. 2020.

[31] JSON.org. *Introducing JSON*. https://www.json.org/json-en.html, [Besøkt 12.05.21].

[32] Martin Lundberg. *Simple PID*. https://pypi.org/project/simple-pid/, [Besøkt 05.04.21].

[33] NTNU Marin. *Motstand på skip*. http://www.marin.ntnu.no/havromsteknologi/depot/tema-hefter/motstand.pdf, [Besøkt 01.05.21].

[34] *MATE Competition Logo*. https://files.materovcompetition.org/images/logos/MATEROVCompetition5683x2662.png, [Besøkt 06.01.21].

[35] *MATE logo*. https://www.facebook.com/materovcompetition/photos/1250385541654400, [Besøkt 06.01.21].

[36] Frank Moraes. *HTML For Beginners*. https://html.com/, [Besøkt 15.04.21].

[37] OpenCV. *Arithmetic Operations on Images*. https://docs.opencv.org/master/d0/d86/tutorial_py_image_arithmetics.html, [Besøkt 17.02.21].

[38] OpenCV. *Changing Colorspaces*. https://docs.opencv.org/master/db/d64/tutorial_js_colorspaces.html, [Besøkt 17.02.21].

[39] OpenCV. *Contour Features*. https://docs.opencv.org/4.4.0/dd/d49/tutorial_py_contour_features.html, [Besøkt 18.02.21].

[40] OpenCV. *Contours*. https://docs.opencv.org/master/d3/d05/tutorial_py_table_of_contents_contours.html, [Besøkt 17.02.21].

[41] OpenCV. *Countour Hierarchy*. https://docs.opencv.org/master/d9/d8b/tutorial_py_contours_hierarchy.html, [Besøkt 18.02.21].

[42] OpenCV. *findContours*. https://docs.opencv.org/4.4.0/d4/d73/tutorial_py_contours_begin.html, [Besøkt 18.02.21].

[43] OpenCV. *FitLine Function*. https://docs.opencv.org/4.4.0/d3/dc0/group__imgproc_shape.html#gaf849da1fdafa67ee84b1e9a23b93f91f, [Besøkt 18.02.21].

[44] OpenCV. *Get Affine Transform*. https://docs.opencv.org/master/da/d54/group_imgproc__transform.html#ga47069038267385913c61334e3d6af2e0, [Besøkt 14.04.21].

[45] OpenCV. *Get Perspective Transform*. https://docs.opencv.org/master/da/d54/group_imgproc__transform.html#gae66ba39ba2e47dd0750555c7e986ab85, [Besøkt 24.02.21].

[46] OpenCV. *getRotationMatrix2d*. https://docs.opencv.org/3.4/da/d54/group_imgproc__transform.html#gafbbc470ce83812914a70abfb604f4326, [Besøkt 20.03.21].

[47] OpenCV. *Hough Circles*. https://docs.opencv.org/master/da/d53/tutorial_py_houghcircles.html, [Besøkt 10.03.21].

[48] OpenCV. *Mean*. https://docs.opencv.org/master/d2/de8/group__core__array.html#ga191389f8a0e58180bb13a727782cd461, [Besøkt 22.02.21].

[49] OpenCV. *medianBlur*. https://docs.opencv.org/master/d4/d86/group__imgproc_filter.html#ga564869aa33e58769b4469101aac458f9, [Besøkt 10.03.21].

[50] OpenCV. *Moments*. https://docs.opencv.org/master/d8/d23/classcv_1_1Moments.html#ab8972f76cccd51af351cbda199fb4a0d, [Besøkt 12.04.21].

[51] OpenCV. *Morpholocical Transforamations*. https://docs.opencv.org/master/d9/d61/tutorial_py_morphological_ops.html, [Besøkt 24.02.21].

[52] OpenCV. *OpenCV - Hjemmeside med filer og dokumentasjon*. https://opencv.org/, [Besøkt 15.02.21].

[53] OpenCV. *Resize*. https://docs.opencv.org/master/da/d54/group__imgproc_transform.html#ga47a974309e9102f5f08231edc7e7529d, [Besøkt 10.03.21].

[54] OpenCV. *Rotate*. https://docs.opencv.org/master/d2/de8/group__core__array.html#ga4ad01c0978b0ce64baa246811deeac24, [Besøkt 06.04.21].

[55] OpenCV. *Rotate*. https://docs.opencv.org/3.4/d2/de8/group__core__array.html#ga4ad01c0978b0ce64baa246811deeac24, [Besøkt 10.03.21].

[56] OpenCV. *Template Matching*. https://docs.opencv.org/master/df/dfb/group__imgproc__object.html#ga586ebfb0a7fb604b35a23d85391329be, [Besøkt 17.02.21].

[57] OpenCV. *The OpenCV-Python Tutorials*. https://docs.opencv.org/master/d6/d00/tutorial_py_root.html, [Besøkt 15.02.21].

[58] OpenCV. *Threshold*. https://docs.opencv.org/master/d7/d1b/group__imgproc__misc.html#gae8a4a146d1ca78c626a53577199e9c57, [Besøkt 15.04.21].

[59] OpenCV. *Threshold types*. https://docs.opencv.org/master/d7/d1b/group__imgproc__misc.html#gaa9e58d2860d4afa658ef70a9b1115576, [Besøkt 15.04.21].

[60] OpenCV. *Thresholding Operations uisng InRange*. https://docs.opencv.org/master/da/d97/tutorial_threshold_inRange.html, [Besøkt 15.02.21].

[61] OpenCV. *Video Capture*. https://docs.opencv.org/master/d8/dfe/classcv_1_1VideoCapture.html#a57c0e81e83e60f36c83027dc2a188e80, [Besøkt 07.04.21].

[62] OpenCV. *Warp Affine*. https://docs.opencv.org/master/da/d54/group__imgproc__transform.html#ga0203d9ee5fcd28d40dbc4a1ea4451983, [Besøkt 24.04.21].

[63] OpenCV. *Warp Perspective*. https://docs.opencv.org/master/da/d54/group__imgproc__transform.html#gaf73673a7e8e18ec6963e3774e6a94b87, [Besøkt 24.02.21].

[64] OWASP. *OWASP Top Ten*. https://owasp.org/www-project-top-ten/, [Besøkt 10.05.21].

[65] Google Cloud Platform. *Flask-Talisman*. https://github.com/GoogleCloudPlatform/flask-talisman, [Besøkt 10.05.21].

[66] Programiz. *Prim Algorithm*. https://www.programiz.com/dsa/prim-algorithm, [Besøkt 18.02.21].

[67] The Pallets Project. *Deployment Options*. https://flask.palletsprojects.com/en/2.0.x/deploying/, [Besøkt 10.05.21].

[68] The Pallets Projects. *Flask, web developmet a drop at a time*. https://flask.palletsprojects.com/en/1.1.x/, [Besøkt 18.04.21].

[69] PyPI. *Bcrypt*. https://pypi.org/project/bcrypt/, [Besøkt 10.05.21].

[70] Kenneth Reitz. *Requests: HTTP for Humans*. https://docs.python-requests.org/en/master/, [Besøkt 10.05.21].

[71] Adrian Rosebrock. *Intersection over Union (IoU) for object detection*. https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/, [Besøkt 07.012021]. Nov. 2016.

[72] Adrian Rosebrock. *Jaccard index*. https://deepai.org/machine-learning-glossary-and-terms/jaccard-index, [Besøkt 20.04.21].

[73] Scikit-image. *Skeletonize*. https://scikit-image.org/docs/dev/auto_examples/edges/plot_skeleton.html, [Besøkt 15.05.21]. 2020.

[74] Scikit-learn. *Kmeans clustering*. https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html?highlight=kmeans#sklearn.cluster.KMeans, [Besøkt 17.02.21].

[75] Scikit-learn. *Nearest Neighbors*. https://scikit-learn.org/stable/modules/neighbors.html, [Besøkt 09.03.21].

[76] scikit-learn. *Scikit-learn - Hjemmeside med filer og dokumentasjon*. https://scikit-learn.org/stable/, [Besøkt 15.02.21].

[77] *Seabed Dredger*. https://www.novasub.com/project-details/seabed-dredger/#1532340251241-e0d5a862-6c83, [Besøkt 06.01.21].

[78] *Serpent Seaview*. https://www.seaviewsystems.com/toolbox/serpent-2/, [Besøkt 06.01.21].

[79] Jaspreet Singh. *Boruvka's algorithm for Minimum Spanning Tree in Python*. https://www.codespeedy.com/boruvkas-algorithm-for-minimum-spanning-tree-in-python/, [Besøkt 25.03.21]. 2020.

[80] *Skjermdump av konkurranseoppgaver*. https://www.youtube.com/watch?v=KWNBOUqVIPQ&t=6s&ab_channel=MATECenter, [Besøkt 15.01.21].

[81] SNL. *HTTPS*. https://snl.no/HTTPS, [Besøkt 10.05.21].

[82] SQLAlchemy. *Hjemmeside, SQLAlchemy*. https://www.sqlalchemy.org/, [Besøkt 10.05.21].

[83] SQLite. *Hjemmeside, SQLite*. https://www.sqlite.org/index.html, [Besøkt 10.05.21].

[84] VOLDERLI Security Market Stor. *IP camera, XM535AI and SC3235*. https://www.aliexpress.com/item/4000062646398.html, [Besøkt 07.04.21].

[85] AHWVSE Security Store. *Lens 90-28 degree view angle*. https://www.aliexpress.com/item/32856254519.html?spm=a2g0s.9042311.0.0.a90e4c4dlPP87m, [Besøkt 07.04.21].

[86] Morten Tengesdal. *Kalman-filteret: Prosessinnsyn i praksis*. 2017.

[87] Tiangolo. *FastAPI*. https://fastapi.tiangolo.com/, [Besøkt 18.04.21].

[88] Tutorialspoint. *Kruskals Spanning Tree Algorithm*. https://www.tutorialspoint.com/data_structures_algorithms/kruskals_spanning_tree_algorithm.htm, [Besøkt 18.02.21].

[89] Ubuntu. *UncomplicatedFirewall - UFW*. https://wiki.ubuntu.com/UncomplicatedFirewall, [Besøkt 10.05.21].

[90] VideoLanOrganization. *VLC*. https://www.videolan.org/vlc/, [Besøkt 07.05.21].

[91] WTForms. *WTForms*. https://wtforms.readthedocs.io/en/2.3.x/, [Besøkt 10.05.21].