Fremtidens fugletitter

조류
jolyu

S. H. Ali, J. Haraldstad, S. Hatlehol,
S. Mestvedthagen, L. Rønning,
Ø. Skaaden og S. Werner-Torgersen

Teknisk notat
26. april 2020

Versjon: 1.0

NTNU
elsys
ELEKTRONISK SYSTEMDESIGN & INNOVASJON

Fakultet for informasjonsteknologi og elektroteknikk

Innhold

1 Problemstilling

Behovet for energi er stadig i vekst, og er i dag større enn noen gang. Den raske veksten i levestandard og befolkning som verden opplever i dag, gjør at kraftproduksjon og infrastruktur har vanskelig for å holde tritt. Spesielt utsatt er små samfunn plassert langt unna eksisterende infrastruktur, der utvidelse eller oppgradering av eksisterende strømnett vil bli svært dyrt. En mulig løsning på dette problemet, er å gjøre det gjeldende samfunnet selvforsynt med *off-grid* fornybar energi, for eksempel fra vindturbiner.

Øysamfunnet Froan i Frøya kommune i Trøndelag, ca. 90 km nordvest for Trondheim, er et område hvor nettopp dette vurderes[1]. På 60-tallet ble det bygget en 23 km lang sjøkabel for å koble Froan til strømnettet. Denne ble så fornyet på 80-tallet, og har siden det forsynt de 115 kundene på øyene med strøm. Her ønsker TrønderEnergi å bygge ut vindturbiner, slik at øysamfunnet er selvforsynt med strøm innen kabelen må fornyes igjen. Et av problemene er at det finnes mye sårbart fugleliv i området, som hubro og havørn[2].

Selv om vindturbiner ikke slipper ut miljøgasser, utgjør de en potensiell risiko for dyrelivet i nærheten. Hvert år dør millioner av fugler som følge av kollisjon med en vindturbin[3]. Dette tallet er imidlertid lite sammenliknet med andre energikilder[3]. Dersom vindturbinene installeres i habitatet til en sjelden eller utrydningstruet fugleart, kan dette likevel skape problemer. Vindturbiner kan føre til en sterk svekking av en allerede truet art, og i verste fall bidra til utryddelse av arter. Direktoratet for naturforvaltning har dermed krevd en nøye utreding av fugleaktiviteten i området før en eventuell utbygging på Froan, slik at vindturbinene forstyrrer fuglelivet minst mulig [2]. Det er dermed et stort behov for å kartlegge fugleaktivitet i området rundt eksisterende eller planlagte vindturbiner. Dette gjøres i dag i stor grad manuelt av ornitologer, som gjør det til en svært kostbar prosess og begrenser mengden data man får samlet inn.

I samarbeid med TrønderEnergi, skal vi i Jolyu takle denne utfordringen. Ved å lage et automatisk overvåkingssystem som detekterer, teller og kartlegger fugleaktivitet i et område, muliggjøres innsamling av store mengder data fra det aktuelle området. Det vil bidra til at utbygging kan settes i gang raskere med redusert kostnad.

2 Konsept

Systemet som designes skal kartlegge og dokumentere fugleaktiviteten i et område. Det skal både kunne brukes i områder der vindturbiner allerede er installert, og i områder hvor det planlegges eller vurderes å installere vindturbiner. I slike tilfeller vil det i dag være nødvendig å ansette ornitologer, som er dyrt og tidkrevende. Systemet skal kunne kartlegge og dokumentere fugleaktivitet på en billig og effektiv måte kontinuerlig hele døgnet. Både antall fugler i området og når på døgnet det er mest aktivitet, skal kunne fastslås. Figur 2.0.1 illustrerer dette konseptet.

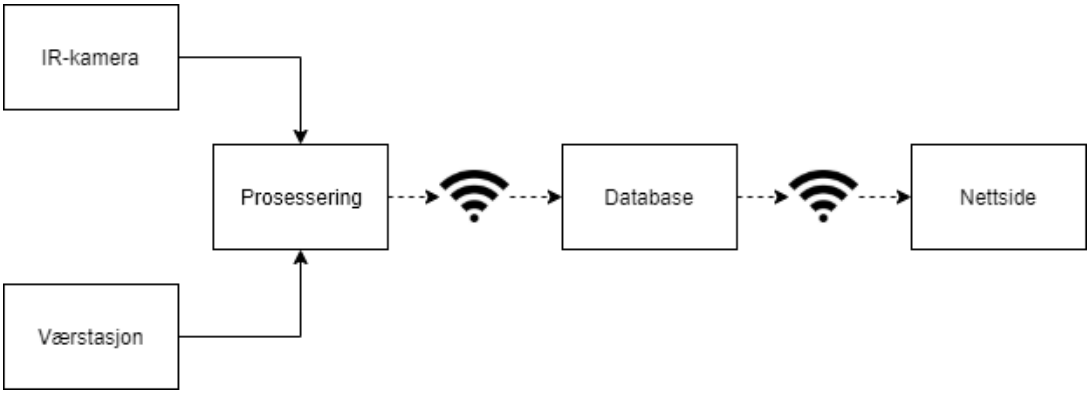
Figur 2.0.1: Illustrasjon av konseptet ved en allerede installert vindturbin.

Deteksjonen skal foregå med et infrarødt kamera som kontinuerlig tar bilder av himmelen for å se etter varmesignaturer fra fugler. Bildene prosesseres, og varmesignaturen fra hver fugl spores fra bilde til bilde og loggføres. Alle fugler i bildet spores individuelt. Systemet skal også innhente grunnleggende værdata, som nedbør, vindhastighet, vindretning, temperatur, luftfuktighet og lufttrykk. Data om fugleaktivitet og værforhold skal overføres over internett, og presenteres på en nettside.

3 Design

3.1 Systemets sammenheng

Systemets informasjonsflyt er illustrert i figur 3.1.1. Det infrarøde kameraet sender en kontinuerlig videostrøm til prosesseringsenheten, som detekterer antall fugler i bildet. Prosesseringsenheten skal videre ta inn data fra værsensorene og tolke disse, før dataene sendes videre til en database. En nettside skal så hente og vise fram dataen fra databasen, slik at brukeren får et oversiktlig bilde over fugleaktiviteten i området.



Figur 3.1.1: Blokkdiagram over systemet. Fuglene blir detektert av et infrarødt kamera og bildene prosesseres. Værdata samles inn. Data blir så sendt videre til en database og vist fram på en nettside.

3.2 Systemkrav

Tabell 1 viser systemkravene. Disse er i stor grad basert på at systemet skal stå utendørs ved vindturbinen Vestas v27-225kW [4]. Det er denne type vindturbin som står på Rye i Trondheim, og dette området blir brukt som testarena.

Tabell 1: Systemkrav.

Generelt		
Kravnavn	**Beskrivelse**	**id**
Mål	Systemet skal detektere og dokumentere fugleaktivitet i luften.	#01
Værbestandighet	Systemet skal være vanntett og tåle å stå ute i norske værforhold.	#02
Strømforsyning	Systemet vil få strøm fra strømnettet.	#03
Værdata	Systemet vil ha temperatur, lufttrykk, luftfuktighet, nedbørmåler og vindsensor.	#04
Deteksjon		
Kravnavn	**Beskrivelse**	**id**
Synsvinkel	Systemet vil bruke et infrarødt kamera med en minimum synsvinkel på 41°x31°.	#05
Rekkevidde	Systemet skal ha en rekkevidde på minimum 50 meter.	#06
Størrelse	Systemet skal minimum detektere fugler med størrelse ned til 300x100mm innenfor rekkevidden.	#07
Fysiske dimensjoner		
Kravnavn	**Beskrivelse**	**id**
Dimensjoner	Produktet vil være mindre enn 200x300x300mm ekskudert stativ.	#08
Overføring, behandling og fremstilling av data		
Kravnavn	**Beskrivelse**	**id**
Prosesseringsenhet	Dataene vil behandles av en liten ettkortsdatamaskin.	#09
Programvare	Programvaren vil være basert på åpen kildekode.	#10
Treffrate[1]	Programvaren skal være god nok til å detektere minimum 75% av fuglene innenfor synsvinkelen og rekkevidden til kameraet.	#11
Overføring internt	Systemet vil bruke USB til å overføre data mellom kamera og prosesseringsenhet.	#12
Overføring eksternt	Systemet vil bruke Wi-Fi for å overføre data fra prosesseringsenhet og databasen.	#13
Fremstilling av data	Resultatene vil fremstilles på en nettside.	#14

[1] Antall fugler som er talt og sporet korrekt

3.3 Kamera

Et infrarødt kamera (heretter kalt IR-kamera) brukes til å detektere infrarød stråling. Infrarød stråling er elektromagnetisk stråling med en bølgelengde mellom 7 µm og 1 mm [5]. Alle objekter med en temperatur mellom det absolutte nullpunkt (0K) og ca 3900K sender ut slik stråling [5], og bølgelengden og intensiteten til denne strålingen kan brukes for å avgjøre temperaturen til objektet. I dette designet brukes et IR-kamera for å detektere den infrarøde strålingen emittert fra fugler. Fugler er varmblodige og holder en kroppstemperatur på rundt 40 °C [6]. De emitterer infrarød stråling som i prinsippet kan detekteres av et infrarødt kamera. Himmelen i bakgrunnen vil oppføre seg som et tilnærmet svart legeme, og dermed emittere en relativt mindre mengde IR-stråling.

Et IR-kamera brukes i stedet for et vanlig kamera, da det gjør det enklere å detektere fuglene mot himmelen. Et godt eksempel på dette vil være når en hvit måke flyr foran hvite skyer. Fargen til måken vil da kunne gå i ett med skyene, og gjør bildebehandlingen for å detektere fuglen svært vanskelig. Med et IR-kamera vil det være mulig å se forskjell på fugler og skyer, og fuglen kan dermed detekteres. Videre vil et IR-kamera også gjøre det mulig å detektere fugler når det er mørkt, i motsetning til et konvensjonelt kamera. En av ulempene med IR-kamera er at fuglenes fysiske trekk ikke er like synlige, som for eksempel farge, som ville gjort det lettere å bestemme fuglenes art. Et IR-kamera kan også fungere dårligere i noen værtyper, som regn og snø, da nedbør vil kunne emittere relativt mye IR-stråling, og gi mange falske positiver[2].

3.4 Deteksjon

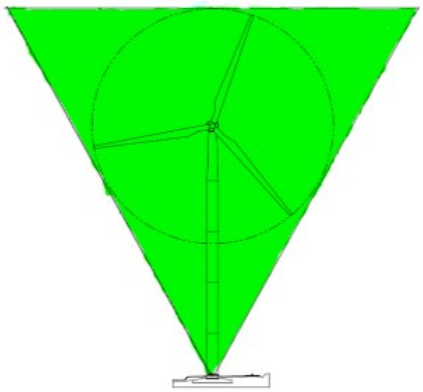
En prosesseringsenhet skal ta inn en kontinuerlig strøm av bilder fra kameraet og behandle disse bilde for bilde. Bildebehandlingen skal finne objekter i bildet, eller såkalte *blobs*[3]. Objektene som skal oppdages er fugler, og disse vil skille seg fra bakgrunnen på grunn av deres høyere temperatur, som representeres ved at de har ulik farge fra bakgrunnen på bildene. Bildebehandlingen skal kunne detektere flere objekter i samme bildet. I tillegg skal et bilde kunne sammenlignes med forrige bilde for å finne ut om et objekt har beveget seg, og dermed kunne følge dette objektet gjennom en bildeserie. Slik unngås det at hver fugl telles en gang per bilde, men blir i stedet fulgt fra den kommer inn i synsvinkelen til kameraet til den går ut, og blir registrert som én fugl.

Dersom systemet plasseres foran en allerede instalert vindturbin, er det ønskelig at deteksjonsområdet minimum dekker området som vist i figur 3.4.1 og 3.4.2[4]. Videre plasseres kameraet slik at vindturbinen ikke er i synsvinkelen til kameraet, da dette vil kunne forstyrre målingene. Trær, bygginger, kraftlinjer eller andre større objekter, bør heller ikke være i kameraets synsvinkel.

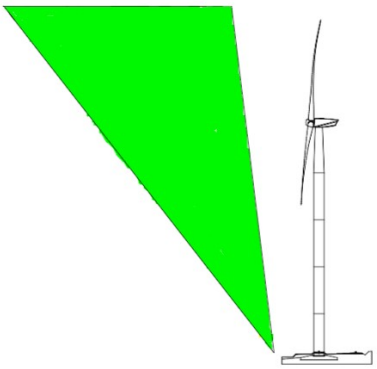
[2] Registrering av fugler som ikke er til stede

[3] Forkortelse for Binary Large OBjects

[4] Det er ikke nødvendig at kameraet står foran vindturbinen, da den vil rotere for å peke inn i vinden. Vindturbinen er tegnet inn for å vise at deteksjonsområdet ikke skal inneholde turbinen.



Figur 3.4.1: Deteksjonsområdet til systemet sett fra framsiden av vindturbinen, markert i grønt.



Figur 3.4.2: Deteksjonsområdet til systemet sett fra siden av vindturbinen, markert i grønt.

3.5 Værstasjon

Systemet skal ha egne sensorer for innsamling av værdata. Systemet vil kunne måle temperatur, trykk og fuktighet i lufta, i tillegg til vindhastighet, vindretning og nedbør. Dataene fra sensorene skal kunne knyttes opp mot fugleaktiviteten, for å finne eventuelle sammenhenger mellom vær og fugleaktivitet. Dette kan så brukes til for eksempel å estimere sannsynligheten for at en fugl kolliderer med en vindturbin i ulikt vær og ulike årstider.

3.6 Nettside og database

Systemet vil ha en nettside som skal fremstille data som samles fra systemet. Data skal vises på en slik måte at den er lett forståelig og navigerbar for en bruker uten spesiell teknisk kompetanse. Data skal kunne sorteres etter behov for å se trender i fugleaktivitet, for eksempel time for time eller dag for dag. Det vil også være mulig å eksportere rådata direkte fra nettsiden.

Det blir brukt en database for å lagre data fra værsensorene, kameraet og prosessoren. Det blir tatt utgangspunkt i at det er Wi-Fi dekning der produktet plasseres. Databasen vil brukes som et mellomledd mellom sensorene og nettsiden. Det trengs dermed ikke å skrives en egen protokoll for overføring av data.

3.7 Strukturelt

Prosesseringsenheten og kameraet skal monteres sammen i en boks, som skal beskytte elektronikken og systemet mot vær og annen ytre påvirkning. Boksen skal være festet til en påle for å heves over bakkenivå, slik at forstyrrelser eller skader fra dyr unngås, samt for ikke å tildekkes av snø eller vegetasjon. Videre skal boksen tilfredsstille systemkrav `#02` om å være værbestandig, og dimensjonskravene `#08` fra tabell 1. Værsensorene skal være festet på samme påle som resten av systemet. Systemet vil få strøm fra det lokale strømnettet.

4 Implementering

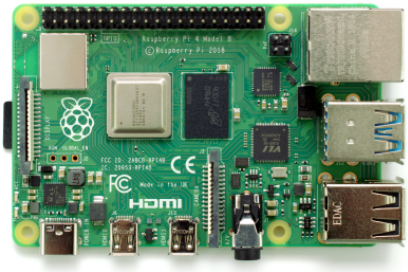
4.1 IR-kamera

Det infrarøde kameraet som benyttes er et FLIR C3-kamera. Kameraet er håndholdt, men kan også koblet til en datamaskin, her Pi-en. Kameraet har en IR-sensor med synsvinkel på 41° × 31°, bilderate på 9 Hz, og kan detektere temperaturforskjeller på under 0,1 °C. Kameraet vil kontinuerlig ta bilder med en valgbar frekvens, som optimaliseres for å gi best mulig deteksjon. Samtidig kan denne begrenses slik at bildebehandlingen ikke blir overbelastet. Bildene overføres til prosesseringsenheten via en USB-kabel som definert i systemkrav `#12`.

Figur 4.1.1: IR-kameraet som systemet bruker, en FLIR C3.

4.2 Prosesseringsenhet

Til databehandling brukes en Raspberry Pi (heretter kalt Pi-en). Dette er en liten datamaskin på ett enkelt kretskort, men som tilbyr nok minne og prosesseringskraft til systemets formål. Dette tilfredsstiller systemkrav `#09`. Pi-en utfører all bildebehandling samt prosessering av værdata og bruker trådløs kommunikasjon til overføring av data til databasen. Pi-en er modell 4B, med 4GB RAM og en 64-bits prosessor [7]. Operativsystemet som brukes er Raspbian Buster med desktop [8], som er optimalisert for bruk på Pi-en.



Figur 4.2.1: Prosesseringsenheten, en Raspberry Pi 4B.

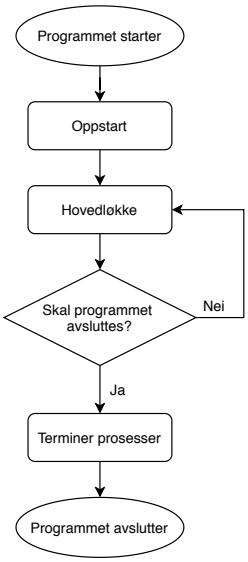
4.3 Programvare

Hovedprogrammet kjører på Pi-en og har som mål å analysere bildestrømmen fra kameraet og sende informasjon om antall fugler til en database. Til bildebehandlingen brukes Python-rammeverket *OpenCV* [9]. *OpenCV* er laget for å gi utviklere rask og enkel tilgang til avanserte algoritmer innen maskinlæring og datasyn. Dette rammeverket kan dermed brukes til bilderprosessering i form av filtrering, deteksjon av objekter og sporing, kalt *tracking*, i sanntid.

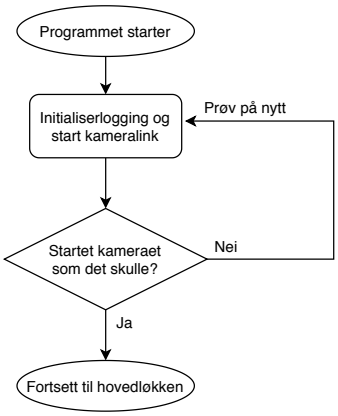
Den overordnede flyten i programmet er vist i figur 4.3.1. Først kjører programmet gjennom en oppstartsprosedyre før det går inn i en løkke med sykliske hendelser.

I oppstartsfasen initialiseres en loggfil og en link til kameraet, som vist i flytskjema i figur 4.3.2. Loggfilen er beskrevet i underavsnitt 4.3.5.

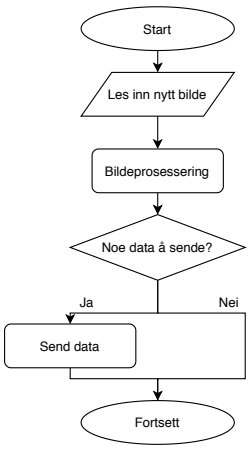
Programmet fortsetter så å kjøre gjennom en rekke sykliske hendelser for bildegjenkjenning og tracking. Dette krever at man tar i bruk flere av de avanserte algoritmene *OpenCV* tilbyr. Ytterligere informasjon om bildeprosessering i underavsnitt 4.3.1. Flyten til hovedløkken kan sees i figur 4.3.3.



Figur 4.3.1: Flytskjema for programmet.



Figur 4.3.2: Flytskjema for oppsettet til programmet.



Figur 4.3.3: Flytskjema til hovedløkken.

OpenCV har en innbygd funksjon for å lese inn videostrømmer, slik at IR-kameraet kan benyttes som om det var et vanlig webkamera. Det er altså forholdsvis enkelt å lese en bildestrøm fra kameraet med Pi-en, og kode for dette er vist i kodeeksempel 4.3.1. Problemet blir dermed i hovedsak å prosessere og analysere disse bildene for å kunne gjenkjenne blobs og spore disse mellom bildene.

```
vc = cv2.VideoCapture(0)        #starter bildestrøm fra kamera

if vc.isOpened():               #sjekker om kameraet er åpnet
    rval, frame = vc.read()     #leser bilde fra kamera
else:
    rval = False                #setter rval til False fordi kameraet ikke ble åpnet
```

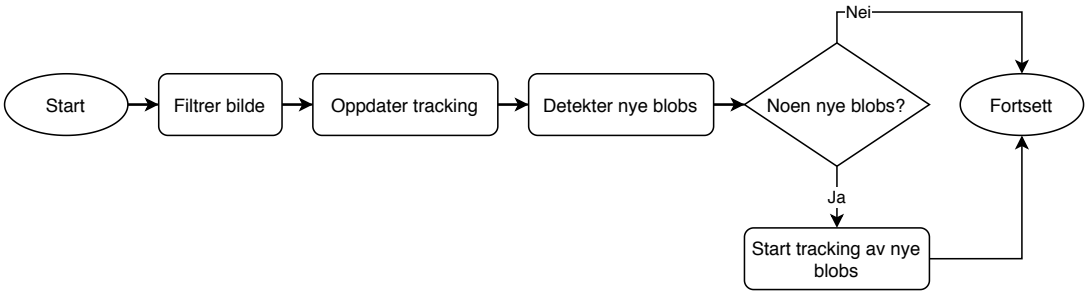
Kodeeksempel 4.3.1: Kodeeksempelet viser hvordan man kan bruke *OpenCV* for å hente inn bilder fra en videostrøm.

4.3.1 Bildeprosessering

Det å få en datamaskin til å kjenne igjen et spesifikt objekt i et bilde er en komplisert problemstilling ingeniører har jobbet med i mange år. *OpenCV* tilbyr en god del verktøy som muliggjør nettopp dette.

Flyten for bildeprosesseringen er illustrert i figur 4.3.4.

For å lettere og mer presist kunne analysere blobs må bildene filtreres. Dette er beskrevet i underavsnitt 4.3.2. Deretter kan man utføre *blob-deteksjon* (se underavsnitt 4.3.3) og tracking (se underavsnitt 4.3.4). Dette gjøres ved at man først kjører trackingalgoritmen for å spore blobs fra tidligere bilder, før man kjører blob-deteksjon og starter tracking av nye blobs.



Figur 4.3.4: Flyten til bildeprosesseringen.

4.3.2 Filtrering

For å oppnå best mulig deteksjon av fugler, filtreres bildene for å redusere støy og annen unyttig informasjon. Ideelt sett vil fulgen og bakgrunnen på bildet ha helt forskjellige farger med tydelig konstrast, da det gjør det lettest å detektere fuglene. Filtreringsalgoritmen prøver å oppnå dette idealet.

Først brukes den innebygde funksjonen i *OpenCV*, `gaussian_blur()`, for redusere støy i bildet. Algoritmen som gjør hoveddelen av filteringen, heter Otsus metode [10]. Algoritmen regner ut en terskelverdi for hvert bilde, som så kan brukes for å konvertere bildet om fra gråskala til sort-hvitt. Bilde blir gjort om til gråskala ved å bruke funksjonen `cvtColor()`, som er en innebygd funksjon i *OpenCV*. Den gjør bilde om til gråskala ved å ta et vektet gjennomsnitt av fargeverdiene [11].

Otsus metode fungerer ved å først dele bildet inn i to klasser, bakgrunn og forgrunn. Deretter utregnes variansen i fargeverdier i og mellom begge klassene. En terskelverdi velges så slik at variansen innad i klassen minimeres, og variansen mellom klassene maksimeres.

Den matematiske sammenhengen i likning (1) sier hvordan terskelverdien brukes til å forvandle bilde om til sort-hvitt. I den sammenhengen er 1 svart og 0 hvit. Funksjonen $f(x, y)$ gir fargeverdien til pikslene i gråskala og funksjonen $dst(x, y)$ gir fargeverdiene til bildet i sort-hvit. Variabelen T er terskelverdien. For mer utdypning på hvordan algoritmen fungerer, se [10]. Implementasjonen av algoritmen ligger i *filters.py* under *image_processing* på GitHuben til Jolyu [12].

$$dst(x, y) = \begin{cases} 1 \text{ hvis } f(x, y) > T \\ 0 \text{ ellers.} \end{cases} \tag{1}$$

Etter Otsus metode kan man så utføre *morphology*-transformasjoner. Dette er funksjoner som er implementert i *OpenCV*-rammeverket [13]. Her ønsker man å utføre *erosion* og *dilation*. Erosion brukes for å filtrere bort støy (white noise), men samtidig krymper det objektet innover fra kantene. Dette problemet løser *OpenCV* ved dilation, som igjen øker størrelsen til objektet. *OpenCV* har implementerte funksjoner som bruker kombinasjoner av disse til å filtrere bort støy. *Opening* brukes for å fjerne støy fra en bakgrunn, mens *closing* brukes i det motsatte tilfellet for å fjerne små hull i et objekt (blobs) i forgrunnen. Dette gjøres ved opencv funksjonen `morphologyEx()`. Disse kjøres etter Otsu-filteret for å filtrere bort eventuell støy fra de resulterende bildene.

4.3.3 Blob-deteksjon

Rammeverket *OpenCV* brukes også til blob-deteksjon. Funksjonen som brukes fra *OpenCV* til dette, er `SimpleBlobDetector()`. Den returnerer et objekt av `detector`-klassen. Klassen inneholder alle parametere til blob-deteksjonen, det vil si hvilke krav som stilles til hva som regnes som en blob. I tillegg inneholder klassen en medlemsfunksjon, `detect`, som finner alle blobene i bildet og returnerer en liste med koordinatene til disse. Kodeeksempel 4.3.2 viser hvordan disse funksjonene er brukt. Implementasjonen av funksjonene og klassen finnes i GitHuben til *OpenCV* [14].

```
1  #blob-deteksjon parametere:
2  MAX_AREA = 5000
3  MIN_AREA = 10
4
5  def init_blob_detector():
6      '''Setup SimpleBlobDetector parameters'''
7      params = cv2.SimpleBlobDetector_Params()
8
9      # Forandrer tersklene for areal
10     params.maxArea = MAX_AREA
11     params.minArea = MIN_AREA
12
13     detector = cv2.SimpleBlobDetector_create(params)    #create detector
14
15     return detector
16
17 def blob_detection(img):
18     '''Function to detect blobs. Returns list of keypoints'''
19
20     detector = init_blob_detector()   #oppretter detektor med parametere for blobs
21     keyPoints = detector.detect(img)  #analyserer og lager en liste med nøkkelpunkter
22
23     return keyPoints
```

Kodeeksempel 4.3.2: Implementasjon av blob-deteksjon. Merk at koden her er noe redigert for å være mer lesevennlig. Blant annet brukes flere ulike parametere.

Som vist i kodeeksempel 4.3.2, oppretter `init_blob_detector()`-funksjonen i linje 20 en `detector`, hvor det fritt kan endre på en rekke parametere, som for eksempel areal. Til slutt returnerer den en `detector` med parameterne satt til deres ønskede verdier. Funksjonen `blob_detection()` oppretter en slik `detector` ved å kalle på `init_blob_detector()`, og bruker den innebygde `detect`-funksjonen til *OpenCV* til å analysere bildet.

Den returnerer så en liste over *nøkkelpunkter*, eller *keypoints*. Et nøkkelpunkt er et objekt som inneholder koordinatene til en blob samt radiusen. Videre brukes `blob_detection()` hver gang nøkkelpunktene i et bilde skal bli funnet.

4.3.4 Tracking

For å unngå at programmet teller samme fugl flere ganger, brukes tracking. Måten trackingen fungerer er at en tracker konstrueres og gis koordinater til et område på skjermen der det finnes et objekt. Dette objektet vil trackingen kjenne igjen i neste bilde, og den vil klare å følge objektet gjennom videostrømmen. Siden det kan være flere fugler i bildet samtidig, må det være mulig å tracke alle uavhengig av hverandre. Dette gjøres ved å lage et *multi-tracker* objekt som samler flere trackere. Hver av disse trackerene følger forskjellige fugler. Ved bruk av flere trackere, er det viktig at hver fugl kun trackes en gang. Dette er for å unngå at den samme fuglen telles flere ganger. Dersom et objekt allerede har en tracker, vil multi-trackeren passe på at det ikke oprettes en ny tracker for samme objekt.

Når en fugl skal trackes er koordinatene til fuglen i bildet nødvendig. Dette hentes fra koden til blob-deteksjon. Blob-deteksjonskoden returnerer en liste med nøkkelpunkter. Disse inneholder senter og radius til blobene som er i bildet. *Trackerfunksjonen*, som skal tracke blobene, tar derimot inn en firkantet boks som må omslutte bloben helt. For at trackingen skal fungere optimalt må boksen være en del større enn bloben. Derfor er det behov for en funksjon som kan gjøre om nøkkelpunkter til bokser. Dette er vist i kodeeksempel 4.3.3. Funksjonen `KeypointsToBoxes(keypoint)` returnerer en liste som inneholder all informasjon om boksene som trengs til trackingen.

```
def KeypointsToBoxes(keypoints):
    boxes = []  #liste for bokser rundt alle keypoints
    for keypoint in keypoints:
    #boksene lages ved hjelp av sentrum og størrelsen på blobene
        point = keypoint.pt
        size = 10 + 3*int(keypoint.size)
        box = (int(point[0])- (size/2),int(point[1]) - (size/2), size, size)
        boxes.append(box)
    return boxes
```

Kodeeksempel 4.3.3: Kode som viser hvordan nøkkelpunkter fra blob-deteksjonen blir gjort om til bokser som kan brukes til trackingen.

Når alle nøkkelpunkter er gjort om til bokser, må det identifiseres hvilke blobs som allerede trackes. Det kan gjøres ved å sjekke om midten av noen av blobene ligger innenfor boksen til en blob som allerede trackes. Hvis den gjør det skal ikke bloben trackes på nytt. Funksjon `removeTrackedBlobs(keypoints, boxes)` i kodeeksempel 4.3.4 viser hvordan dette gjøres. I denne funksjonen blir nøkkelpunktene til blobene som allerede trackes, fjernet fra listen. Nøkkelpunktene til de resterende blobene blir returnert.

OpenCV har flere funksjoner som hjelper til med denne trackingen, og har flere forskjellige metoder for tracking som varierer i nøyaktighet og tidsbruk. For å opprette en tracker av en av disse trackertypene kalles funksjonen `createTrackerByName(trackerType)`, der `trackerType` er hvilken type tracker som blir brukt. En rask og god tracker som er valgt i dette prosjektet er *CSRT-trackeren*. Etter trackeren er konstruert, må den initialiseres. Dette gjøres ved å si hvilket bilde bloben er i og koordinater til en boks som skal omslutte bloben. Siden det kan være flere fugler i et bilde, er det nødvendig å kunne lage flere trackere, og lagre disse sammen.

OpenCV har en multitrackerklasse, men denne er ikke optimal, da det ikke kan slettes trackere

```python
def removeTrackedBlobs(keypoints, boxes):
    #newKeypoints: liste med nye blobs, diff: for allerede tracka blobs
    newKeypoints = []
    diff = []
    try:
    #Sjekker om blobs er nye. Legges til i tilhørende liste
        for points in keypoints:
            x = points.pt[0]
            y = points.pt[1]
            for box in boxes:
                xb,yb,wb,hb = box
                if xb<x and x<xb+wb and yb<y and y<yb+hb:
                    diff.append(points)
        for point in keypoints:
            if point not in diff:
                newKeypoints.append(point)
    except:
        pass
    #returnerer alle nye blobs
    return newKeypoints
```

Kodeeksempel 4.3.4: Kodeeksempelet viser hvordan det sjekkes om blobs er nye eller om de trackes fra et tidligere bilde.

fra et multitracker-objekt. Derfor er en ny klasse implementert med de ønskede egenskapene. Denne klassen er kalt `NewTracker` og en del av implementeringen av denne klassen kan sees i kodeeksempel 4.3.5. Klassen består av to lister og en rekke medlemsfunksjoner. Listen `trackers` består av enkle trackere, og listen `trackerFail` består av heltall som forteller i hvor mange bilder etter hverandre trackeren har mislyktes med å følge en blob. Når blob-deteksjons-algoritmen har kjørt, legges det til trackere til alle nye blobs med medlemsfunksjonen `add()`. I tillegg har `NewTracker`-klassen medlemsfunksjonene `pop()` og `update()` [12]. Funksjonen `pop()` fjerner trackere, og `update()` oppdaterer alle trackere. I tillegg til å oppdatere trackerene, vil `update()`-funksjonen inkrementere listeelementene i `trackerFail` som hører til trackere som mislyktes med sporingen sin. Dersom trackeren klarte å følge bloben blir tilhørende `trackerFail`-element satt til 0. Alle trackere som ikke har detektert bloben sin i fem bilder på rad, vil fjernes fra multitrackeren. Funksjonen `update()` returnerer også en liste med koordinatene til alle trackere.

4.3.5 Logging

Det er ikke alltid systemet fungerer som det skal. Derfor er det svært nyttig å ha en loggfil der all viktig informasjon om programmets aktiviteter lagres. Både operasjonene programmet gjør vellykket og det programmet ikke har fått til, blir lagt til i loggfilen. Begge heldelser er nyttig informasjon. Alt som blir lagt til i logging-fila, blir også merket med hvor viktig informasjonen er, for eksempel `INFO` eller `CRITICAL`. Denne vil også fungere som en reserveløsning dersom man mister nettverksforbindelsen.

```python
class NewTracker():
    def __init__(self):
    #objektet består av to lister
    #En med trackerne, og en som vet hvor lenge en tracker har vært ugyldig
        self.trackers = []
        self.trackerFail = []

    def add(self, trackerObj):
    #når en ny tracker legges til, legges den inn i self.trackers
    #i tillegg får den et element i trackerFail-lista,
    #som viser at den har vært ugyldig i 0 bilder
        self.trackers.append(trackerObj)
        self.trackerFail.append(0)
```

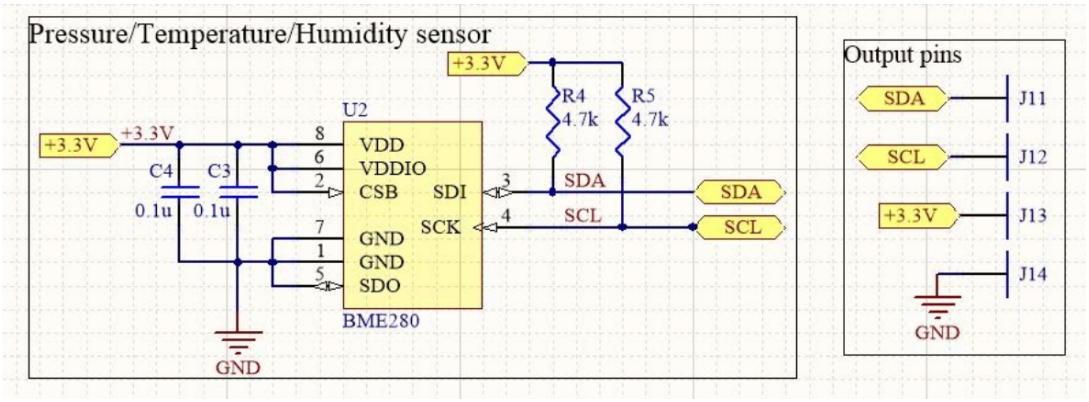
Kodeeksempel 4.3.5: Konstruktøren og add-funksjonen til multitrackeren.

4.4 Værstasjon

Værstasjonen vil måle temperatur, trykk og luftfuktighet, i tillegg til vindhastighet, vindretning og nedbørsmengde. All dataen sendes via et egendesignet kretskort til prosesseringsenheten, som prosesserer dataen og sender det videre til databasen for så å bli gjort tilgjengelig på nettsiden.

4.4.1 Trykk, temperatur, luftfuktighet

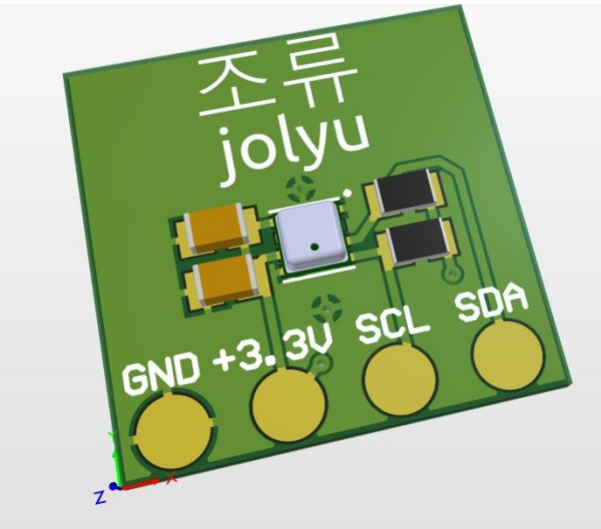
Alle luftdata hentes fra en BME280-sensor fra Bosch [15], som måler både temperatur, trykk og fuktighet i lufta. Kommunikasjonen mellom sensoren og Raspberry Pi-en foregår over en I2C-protokoll, en synkron, seriell bussprotokoll [16]. Denne bussprotokollen tillater toveiskommunikasjon, og bruker 7-bits adresser. I2C tillater dermed opp til 128 enheter ved bruk av kun to ledninger. Figur 4.4.1 under viser kretstegning av luftsensoren.



Figur 4.4.1: Kretsskjema for luftsensoren.

De eneste eksterne komponentene BME280-sensoren, $U2$, trenger rundt seg, er en kondensator ved hver av de to spenningsinngangene `VDD` og `VDDIO`, $C3$ og $C4$, og en opptrekksmotstand (eng: pull-up resistor) til databussen `SDA` og en til klokkebussen `SCL`, henholdsvis $R4$ og $R5$.

Figur 4.4.2 under viser 3D-modell av kretskortet.



Figur 4.4.2: 3D-modell av luftsensoren.

For å lese data fra sensoren finnes det et eget bibliotek som heter *bme280lib* [17], som gjør det enkelt å lese data fra sensoren med en `sample()`-funksjon. Biblioteket *smbus2* [18] brukes også for å opprette en I2C bussforbindelse. Programmet som leser dataen, ligger i *bme280.py* filen under *Weather-station* i GitHuben[12].

4.4.2 Anemometer

Anemometeret som brukes, vist i figur 4.4.3, måler vindhastigheten. Sensoren bruker en *reed-bryter*, en mekanisk bryter som lukkes av et magnetisk felt. For hver omdreining beveger en magnet seg forbi denne bryteren, slik at den lukkes i et kort øyeblikk. Ved å telle antall ganger bryteren lukkes per tidsenhet, kan rotasjonshastigheten til vindmåleren beregnes. Dette kan igjen brukes til å regne ut vindhastigheten. I følge databladet[19] vil bryteren lukkes én gang i sekundet ved en vindhastighet på 2,4 km/h, som tilsvarer 0,67 m/s.

Figur 4.4.3: Figur av anemometeret som brukes.

For å telle antall omdreininger, brukes klassen `Button` fra biblioteket *gpiozero* for å registrere når reed-bryteren lukkes. Programme til anemometeret, ligger i *WeatherStationMain.py* under *Weather-station* i GitHuben[12].

4.4.3 Vindretning

Retningsmåleren, vist i figur 4.4.4, består av 8 reed-brytere plassert i en sirkel slik at det er 45° mellom hver bryter [19]. En værhane med en påmontert magnet vil rette seg mot vinden, og vil lukke enten en bryter eller to nabobrytere om gangen, avhengig av posisjonen. Dette resulterer i totalt 16 posisjonskombinasjoner, og dermed en nøyaktighet på ±11.25°. De 16 ulike kombinasjonene vil koble signalpinen til 16 ulike motstandsverdier, som gjør at vindretingen kan leses som et analogt signal ved hjelp av en spenningsdeler for å finne retningen. Da Pi-en ikke har en innebygd analog/digital-omformer (heretter kalt ADC), sendes det analoge signalet via en MCP3221A5T ADC til Pi-en. Spenningsdelingen og ADC-en, samt programmet som henter ut data fra ADC-en, er diskutert nærmere i underavsnitt 4.4.5.

Figur 4.4.4: Figur av vindretningssensoren som brukes.

Når det analoge signalet fra sensoren er lest av prosesseringsenheten, brukes et *dictionary*[5] for å koble de ulike verdiene til ulike vinkler. Dette ble kalibrert ved manuelt å rotere sensoren, og lese av verdien ved ulike vinkler. Programmet returnerer så en retningsvinkel i grader, hvor 0°er definert som nord. Programmet ligger i *windDirection.py* under *Weather-station* i GitHuben[12].

4.4.4 Nedbør

Nedbørsmåleren består av to selvtømmende beholdere, som fylles opp og tipper over etter en gitt mengde nedbør. Når den ene beholderen er fylt opp og tipper over, renner vannet ut under sensoren, og den nye beholderen blir plassert under vanninntaket. Denne sensoren bruker også en reed-bryter, som trigges av en magnet hver gang en beholder tipper over. Ved å telle antall ganger bryteren lukkes og multiplisere dette med vannmengden før en beholder tipper, beregnes mengden nedbør som har falt i et gitt tidsrom. Mengden nedbør før en beholder tipper, er i databladet[19] definert til å være $0,2794$mm. Figur 4.4.5 viser bilde av nedbørsmåleren.

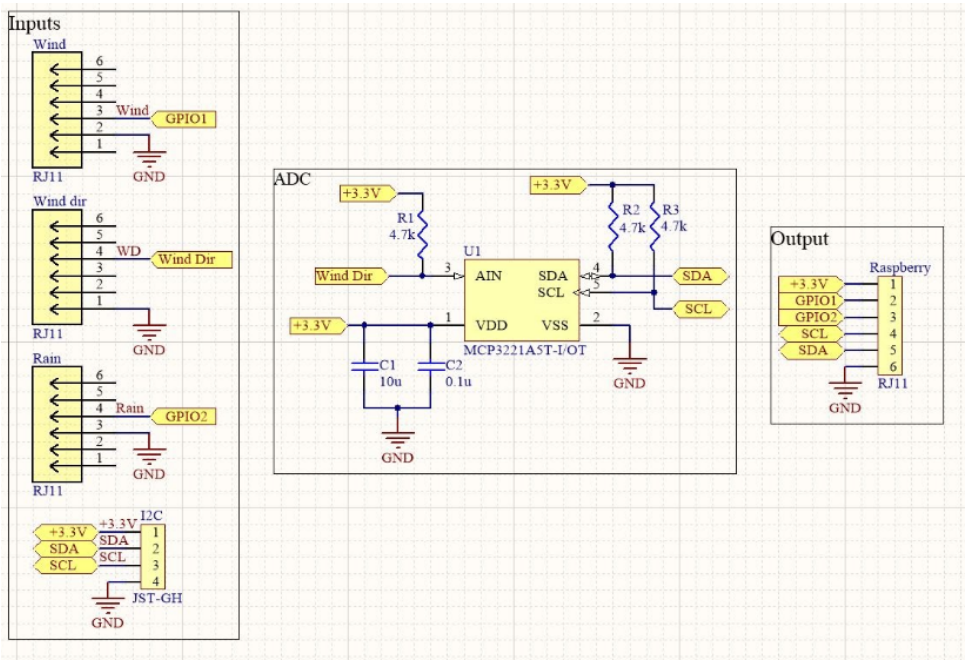
Figur 4.4.5: Figur av nedbørsmåleren som brukes.

Tilsvarende anemometeret i underavsnitt 4.4.2, brukes også klassen `Button` fra biblioteket *gpiozero* her for å detektere hver gang reed-bryteren lukkes. Programme ligger i *weatherStationMain.py* under *Weather-station* i GitHuben[12].

[5]En type uordnet liste der man lagrer data under en nøkkel(key)[20].

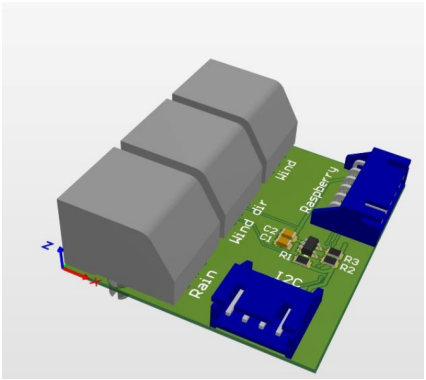
4.4.5 Kretskort til sensorer

For å enkelt kunne koble alle sensorene til Pi-en, brukes et egenutviklet kretskort. Figur 4.4.6 viser kretstegning til dette kretskortet.



Figur 4.4.6: Kretsskjema for kretskortet til sensorene.

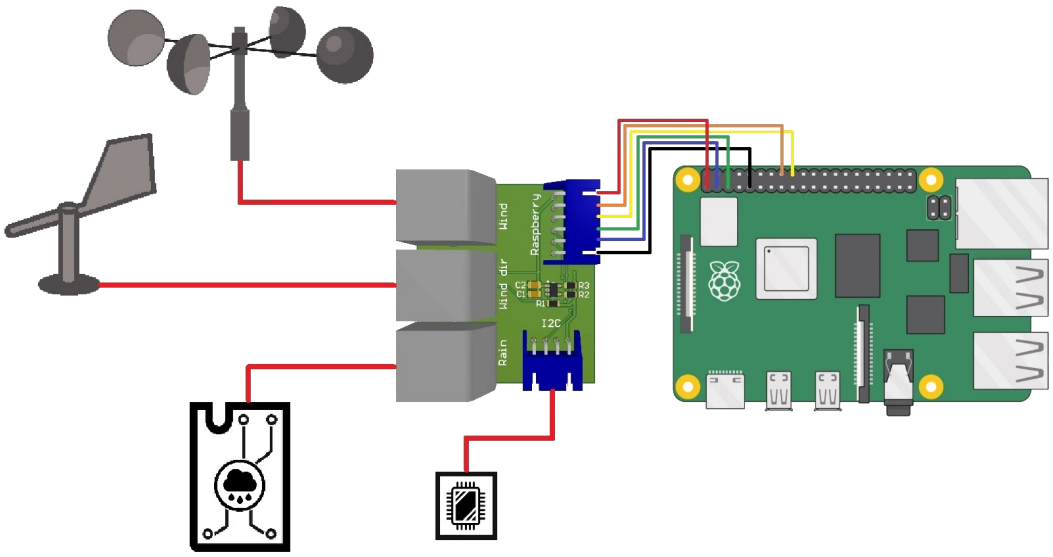
I tillegg til kontakter for å koble til sensorene, inneholder kretskortet en motstand, $R1$, til spenningsdeleren til vindretningsensoren. Videre inneholder kretskortet en ADC, $U1$, for å digitalisere signalet fra spenningsdeleren. ADC-en som brukes er en MCP3221A5T fra Microchip[21]. Omformeren tar inn et analogt signal, og kommuniserer direkte over I2C-bussprotokollen med Pi-en. Opptrekksmotstander, $R2$ og $R3$, brukes også her for å trekke `SDA` og `SCL` bussene høye, og to kondensatorer, $C1$ og $C2$, brukes for å filtrere bort ulike støyfrekvenser fra spenningsforsyningen. Figur 4.4.7 viser en 3D-modell av kretskortet.



Figur 4.4.7: 3D-modell av kretskortet til sensorene.

For å lese data fra MCP3221A5T ADC-en, brukes igjen biblioteket *smbus2*[18]. Funksjonen `read_byte_data(address, byte)` fra dette biblioteket, brukes for enkelt å lese data fra ADC-en over I2C-protokollen.

Luftsensoren og ADC-en til vindretningssensoren kommuniserer med prosesseringsenheten direkte via samme I2C-buss. GPIO pin 2 og 3 brukes til henholdsvis databussen, `SDA`, og klokkebussen, `SCL`, til kommunikasjon over I2C-protokollen. Anemometeret og nedbørssensoren kobles til henholdsvis GPIO pin 5 og 6 på Pi-en via kretskortet. Alle senorene bruker 3.3 volt spenning fra Pi-en. Oppkoblingen er vist i figur 4.4.8.



Figur 4.4.8: Figuren viser oppkoblingen av værstasjonen til prosesseringsenheten.

4.5 Nettside og database

Nettsiden har som mål å vise informasjonen som hentes inn av både fugletelleren og værstasjonen. Til dette brukes Python-rammeverket *Dash* [22]. Dash er laget spesifikt for å prosessere og vise store mengder data i enkle grafer og oppgi disse på en statisk nettside. Se figur 4.5.1 for en eksempelnettside.

Når nettsiden lastes inn, vil data fra databasen hentes fra nettet. Deretter prosesserer og filtrerer programmet dataen slik at det ikke bare er punktdata, men data basert på timer, dager, måneder og så videre. Dette krever effektive sorterings- og filtreringsalgoritmer, da dataen kan inneholde flere tusen punkter. Noen av disse algoritmene er allerede implementert i ferdige biblioteker, mens andre må implementeres med vanlige funksjoner i Python. Flyten på nettsiden kan sees i flytdiagrammet i figur 4.5.2.

Store deler av nettsiden fungerer på samme måte. Det kjøres en funksjon om det velges data i en graf, eller om nettsiden bes hente et nytt datasett. Dette registreres ved hjelp av en *callback*, se underavsnitt 4.5.2. Det vil derfor kun gis noen eksempler i avsnittene under, og koden til hele nettsiden kan leses på Jolyu sin GitHub[6].

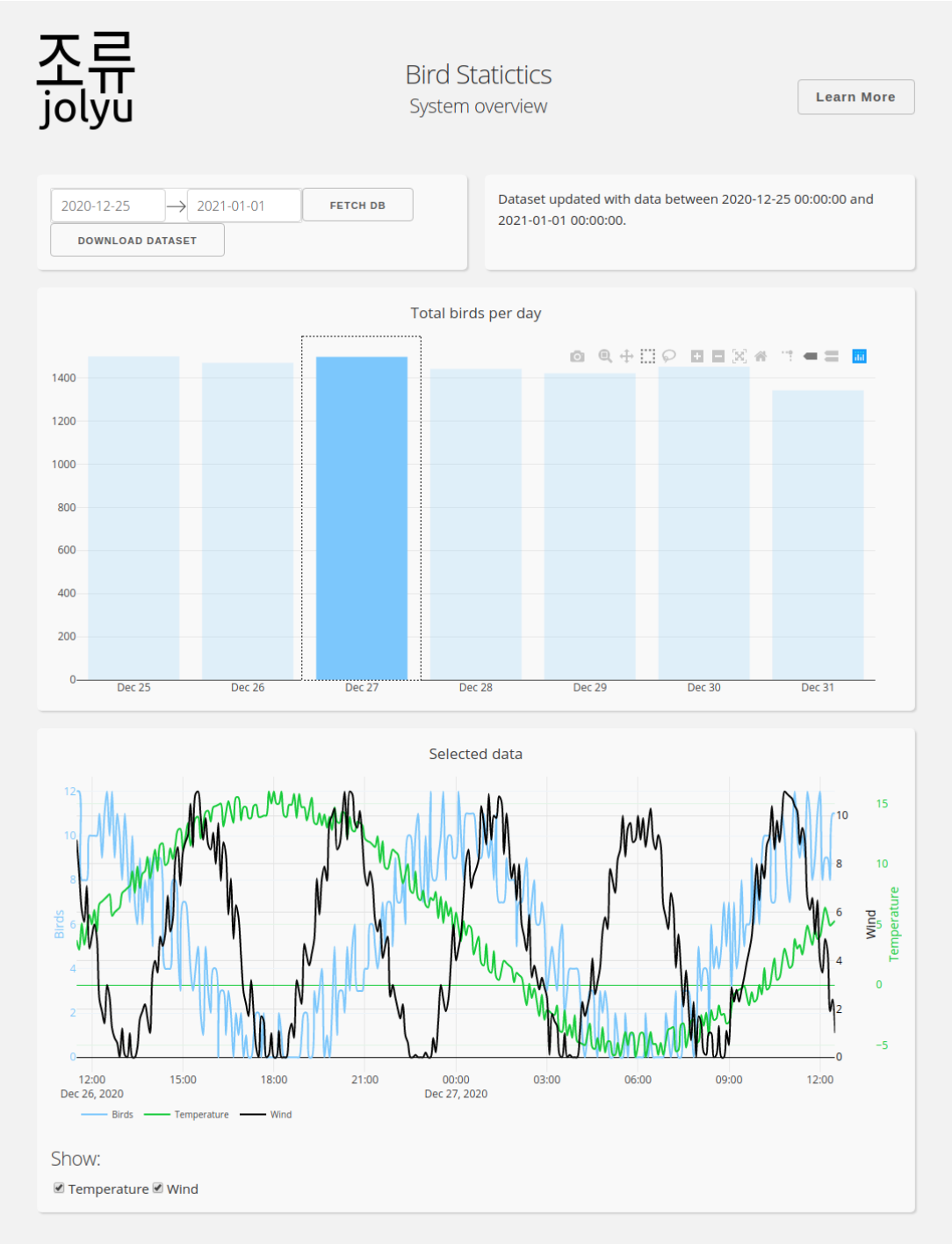
4.5.1 Rammeverket

Dash [22] er en kombinasjon og mellomledd mellom det pythonbaserte nettside-rammeverket for statiske nettsider, *flask*, [23] og javascript biblioteket *plotly* [24] for å plotte data.

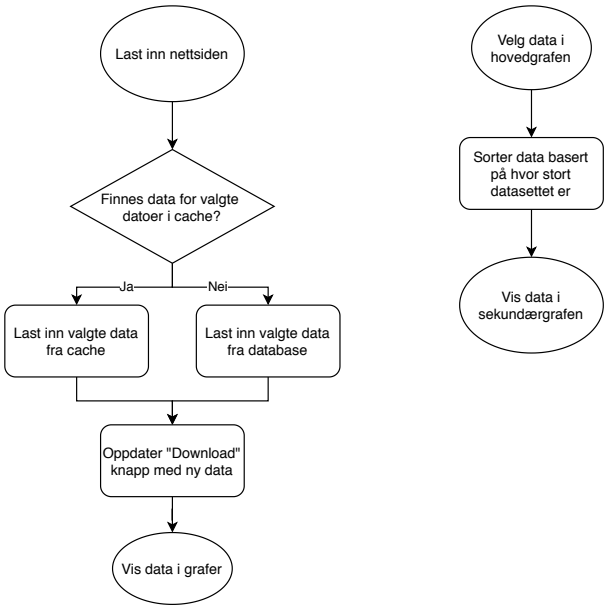
Dash fungerer ved å bruke spesialobjekter fra *plotly* som *graph* og *buttons*, og vanlige `html` objekter som `div` og `header`. Disse defineres i en liste med nøkkelverdier for variabler. Se kodeeksempel 4.5.1 som er tatt fra dokumentasjonen til *Dash*.

Flask er i utgangspunktet laget for statiske nettsider, der data er ferdig laget når nettsiden lastes inn. Det *Dash* implementerer i *flask*, er hvordan lage mer interaktive nettsider ved å legge til

[6]Direkte til nettside repository: `https://github.com/jolyu/nettside`



Figur 4.5.1: Eksempelnettside til Jolyu.



Figur 4.5.2: Flytdiagrammet til nettsiden.

```
import dash_core_components as dcc
dcc.Dropdown(
    options=[
        {'label': 'New York City', 'value': 'NYC'},
        {'label': 'Montréal', 'value': 'MTL'}
    ],
    value='MTL'
)
```

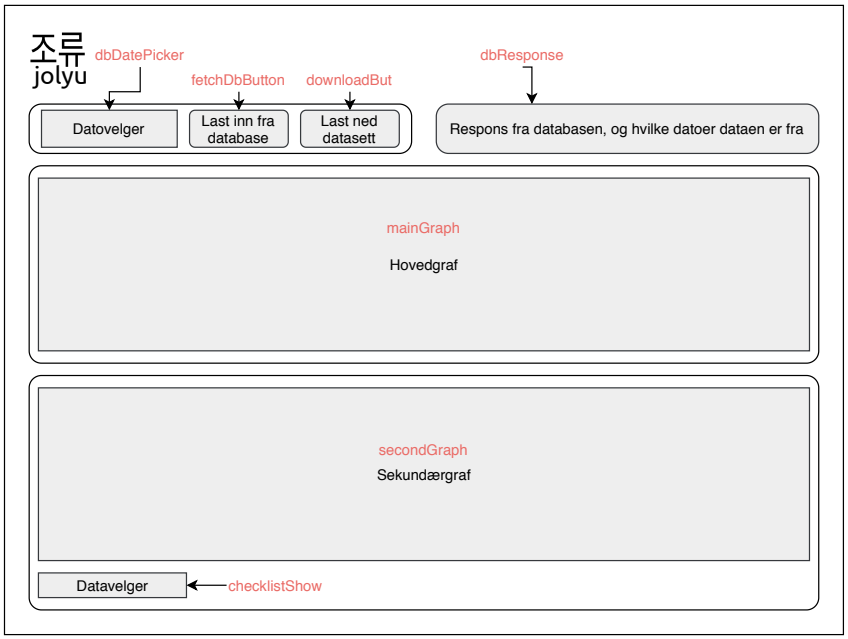
Kodeeksempel 4.5.1: Eksempel på hvordan implementere en enkel *dropdown* (valgliste) i *Dash*.

callbacks, funksjoner som kjører når du for eksempel trykker på en knapp eller endrer på zoom i en graf. Det er slik store deler av nettsiden er bygget opp, ved at brukeren endrer på hvilke objekter som er valgt eller drar i en slider. Da vil det skje en callback som vil kunne endre på data og deretter oppdatere nettsiden.

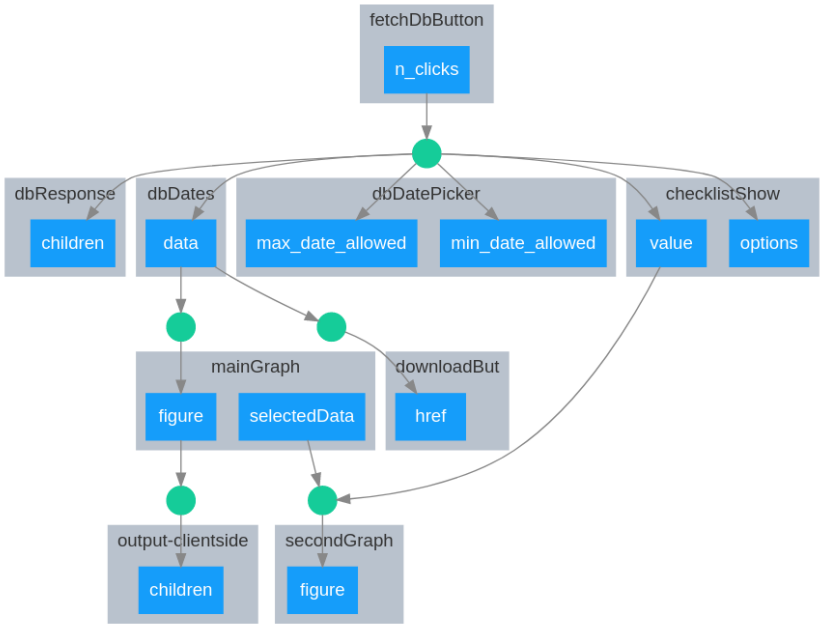
4.5.2 Callbacks

Nettsiden bruker stort sett callbacks for å endre på hvilke data som skal vises i de forskjellige grafene. En callback er en type funksjon som reagerer når noe endres på eller en hendelse, som et knappetrykk. Deretter kjøres funksjonen for så å returnere data til et annet objekt.

Alle de grå elementene i figur 4.5.3 brukes til å gjøre en callback for ulike ting. De røde navnene er navnene på elementene som vil kjøre en callback. Figur 4.5.4 viser hvordan callbacks flytter data rundt om på siden. Ta `fetchDbButton` som eksempel. Når knappen klikkes på, vil data skrives til alle stedene pilene peker til. Der de blå boksene med tekst er forskjellige atributter i hvert av elementene.



Figur 4.5.3: Forenklet figur av nettsiden med hva elementer i grå bokser og navnet til elementet i rød teks, eventuelt med pil til elementet.



Figur 4.5.4: Callback-diagram for nettsiden. De grønne nodene er et callback som utløses av det som kommer inn, og skriver til der pilene peker.

Når `dbDates` oppdateres, trigger det to forskjellige callbacks. Den ene vil oppdatere `href` i `downloadBut` med nedlastning av det aktive datasettet, og den andre vil oppdatere hovedgrafen. I kodeeksempel 4.5.2 vises det hvordan callback-en for oppdatering av nedlastning fungerer.

```
@app.callback(
    Output("downloadBut", "href"),
    [Input("dbDates", "data"),]
)
def UpdateDownloadButton(dates):
    """ Oppdaterer nedlasningsknappen med den nyeste dataen """
    # Sjekk om variabelen dates ikke er tom
    if dates == None:
        dates = GetInitialDates(ref, initialDays)
    else:
        dates = pd.to_datetime(dates)
    # Hent det aktive datasettet
    df = QueryDF(ref, dates)
    # Gjør om datasettet til data som kan lagres i en fil
    # uten ulovelige bokstaver
    csvString = df.to_csv(encoding="utf-8")
    csvString = "data:text/csv;charset=utf-8," + urlParse.quote(csvString)
    return csvString
```

Kodeeksempel 4.5.2: Callback som oppdaterer nedlastningen av det aktive datasettet, basert på en oppdatering av aktive datoer i datasettet.

En callback initialiseres ved å si hva som skal være `Input` (en type *trigger*) og hvor data skal plasseres når callback er ferdig med en `Output`. I dette tilfellet vil callback-en kjøres når `dbDates` endres. Denne inneholder første og siste dato i det aktive datasettet. Denne oppdateres av `fetchDbButton`, som henter datasettet fra databasen, som vist i figur 4.5.4. Den vil deretter lage en `string` med alle aktive datasett og plassere det som en link i atributten `href`.

Dette er bare en av callback-ene på nettsiden, for å se hele nettsiden kan en gå til Jolyu sin GitHub [12].

4.5.3 Sortering og filtrering

Sortering og filtrering kan gjøres på mange måter. På nettsiden brukes en avansert liste, en `dataframe` fra biblioteket *pandas* [25]. Den har sine egne sorteringsalgoritmer som brukes aktivt på nettsiden.

Den mest intense sorteringen er å samle all punktdataen til måneder (eller uker/dager). Da det kan være flere hundretusen datapunkter, og sorteringsalgoritmen må derfor være effektiv. Derfor brukes de innebygde funksjonene for akkurat dette formålet. I kodeeksempel 4.5.3 kan det ses hvordan funksjonen `dataframe.resample(<timegroup>).sum()` i `DataToMonths(df)` brukes for å summere alle punkter i en måned, sammen til et enkelt punkt.

For å hente ut en liten del av datasettet brukes noen selvlagde funksjoner. I `FilterData(df, startDate, endDate)` sendes en dataframe, startdato og sluttdato inn. Funk-

```python
def DataToMonths(df):
    """ Sorter data til måneder """
    df = df.resample('M').sum()
    return df
```

Kodeeksempel 4.5.3: Funksjoner som sorterer punkter på måneder og uker.

sjonen vil finne de aktuelle datapunktene som er innenfor dette tidsintervallet og returnere en ny dataframe.

```python
def FilterData(df, startDate, endDate):
    """ Filtrer ut en del av datasettet """
    dff = df.loc[(df.index > startDate) & (df.index < endDate)]
    return dff
```

Kodeeksempel 4.5.4: Funksjon som henter ut et område innenfor to gitte datoer.

4.5.4 Database

Kommunikasjon mellom Pi-en og nettsiden går via Google sin *Realtime Firebase*-database. Denne kommunikasjonen foregår over Wi-Fi. Dataene Pi-en samler inn blir sendt til databasen som *dictionaries* og lagret der på et NoSQL-format. En NoSQL-database er ikke like streng med struktur i forhold til en SQL-database, så den takler bedre endringer i strukturen til dataene. Firebase har Python-bibliotek for overføring mellom prosesseringsenhet og databasen, og fra databasen til nettsiden. Måten dette fungerer på blir abstrahert bort av ferdigutviklet software, som ligger i *firebase_admin* biblioteket.

All data i databasen har en nøkkel som er et tidsstempel[7] som angir når det ble lastet opp i databasen. Når data blir hentet fra databasen kan det sorteres etter nøklene eller de forskjellige dataene som ligger innenfor hovednøkkelen. Et eksempel på en slik database er en enkel `json`[8]-liste, som sett i kodeeksempel 4.5.5.

```json
{
    "1609455600": {
        "time": 1609455600.0,
        "birds": 8,
        "Temperature": 7.5,
        "Wind": 0.1
    }
}
```

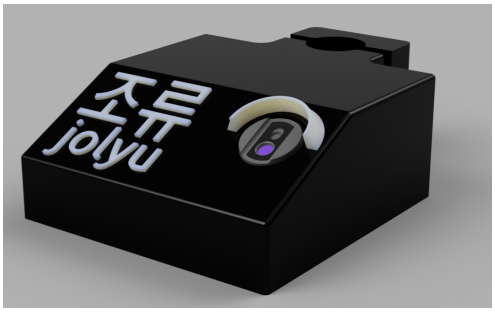
Kodeeksempel 4.5.5: Enkel data i json. Dette er også en bra måte å visualisere en *dictionary*.

[7] Unix tidsstempel (eng: Unix Timestamp) - Tid i sekunder fra 1. januar 1970. Brukes mye i datateknologi.
[8] Et filformat bassert på *dictionaries*.

4.6 Strukturelt

Systemet vil monteres til en aluminiumsstolpe, som vist i figur 4.6.2. Her er boksen festet til stangen 1 m over bakken med modulene til værstasjonen festet øverst, 1,5 m over bakken. Stativet vil plasseres på et flatt underlag, og vil forankres i bakken med vaier eller veies ned for å forhindre velting som følge av for eksempel sterk vind. Kameraet festes inne i boksen som vist i figur 4.6.1. Boksen er tiltenkt å printes i PETG-plast på en 3D-printer, med relativt tykke vegger for å være vanntett.

En 3D-modell av boksen er vist i figur 4.6.1, og en 3D-modell av strukturen til hele systemet er vist i figur 4.6.2.



Figur 4.6.1: 3D-modell av boksen som skal holde kamera og prosesseringsenhet.

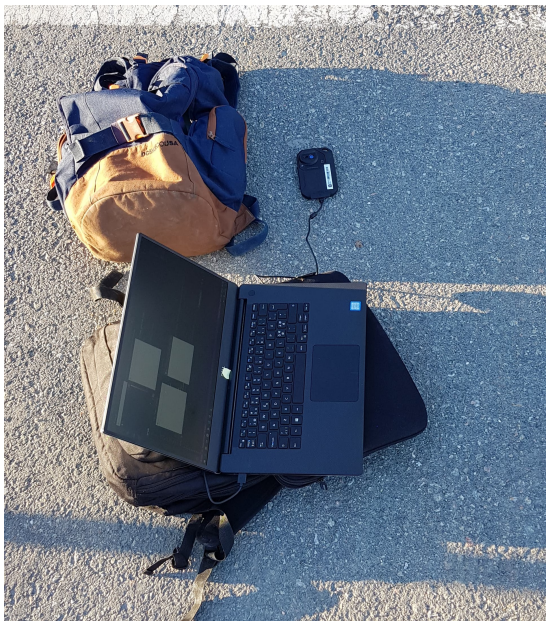
Figur 4.6.2: 3D-modell av hele systemets fysiske realisering.

5 Verifikasjon og test

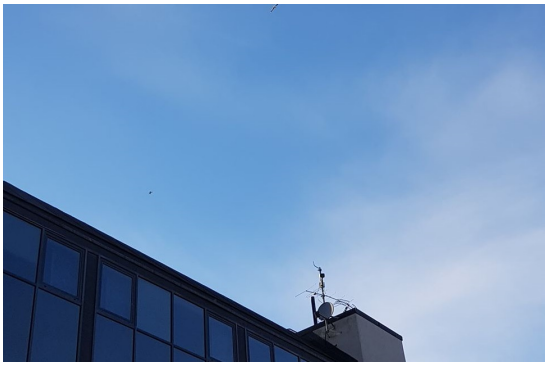
5.1 Testoppsett

Det har i hovedsak blitt gjennomført to tester.

Under test 1 var det oppholdsvær med nesten skyfri himmel. Kameraet pekte vertikalt og lå i skyggen, og var koblet til en bærbar pc for å kunne overvåke datastrømmen under forsøket. Denne testen varte i omtrent to timer. Fokuset for denne testen var å teste trackingen av fugler i tilnærmet idéelle værforhold (skyfritt).



Figur 5.1.1: Utendørs testoppsett. Kamera var koblet til en bærbar datamaskin som kjørte programvaren.



Figur 5.1.2: Værforholdene under testing med testoppsettet i figur 5.1.1.

Under test 2 var det overskyet, og kameraet var vinklet omtrent 45° fra horisontalen. Denne testen gikk i omtrent 6 timer. I tillegg til IR-kamera ble det brukt et konvensjonelt kamera for å kunne sammenlikne infrarøde og synlige bilder. Her ble ikke alle bilder analysert, men kun de bildene som ble oppdaget av programvaren til å inneholde blobs ble lagret grunnet den store mengden data. Under denne testen ble det testet hvor stort utslag overskyet vær har på resultater og tracking.

Figur 5.1.3: Utendørs testoppsett. IR-kameraet er lagt i en vinduskarm ved siden av et kamera. Det var så koblet til en bærbar datamaskin som kjørte en modifisert programvare som lagret de bildene hvor det var detektert blobs.

Figur 5.1.4: Værforholdene under testing med testoppsettet i figur 5.1.3 var mer varierende enn under test 1, men det var for det meste skyet. Dette bildet er tatt med et kamera med mye bredere synsvinkel enn FLIR C3.

5.2 Kamera

5.2.1 Rekkevidde

Kameraet har under testing hatt en rekkevidde opp mot 30 meter, og kan klare å se fugler i høyder over dette. I figur 5.2.1 kan vi se et eksempel på et bilde av en fugl mot skyfri himmel. Det var hovedsaklig måker i området testingen ble utført, og disse kan man tydelig se med kameraet i høyder estimert opp til 30 m. Dette er estimert ved å sammenligne med høyden til bygninger i området. Det er stor usikkerhet i høyden fuglene ble observert, men fra testingen er det ikke urealistisk at man kan observere større fugler i høyder rundt 40-50 meter. Dette betyr at systemet potensielt kan oppfylle systemkrav `#06` gitt gode forhold. Et av problemene ved testing av rekkevidde, er mangel på måleutstyr. Det er derfor ikke mulig å fastslå nøyaktig høyde for fuglene som var synlig, eller ikke synlig på bildene. Dette gjør at systemkrav `#06` ikke er fullstendig testet og verifisert, og fører videre til at systemkrav `#07` ikke er verifisert, grunnet vanskelig verifikasjon av høyde og størrelse.

Figur 5.2.1: Her kan vi se et bilde av en måke tatt med kamera. Denne fuglen var ca. 10-20 m oppe i lufta. Bildet er slått sammen av et vanlig bilde og et termisk bilde med FLIR sin programvare.

Rekkevidde i andre værforhold ble ikke testet like godt, men det virker som om rekkevidden er betydelig lengre når det det er skyfritt.

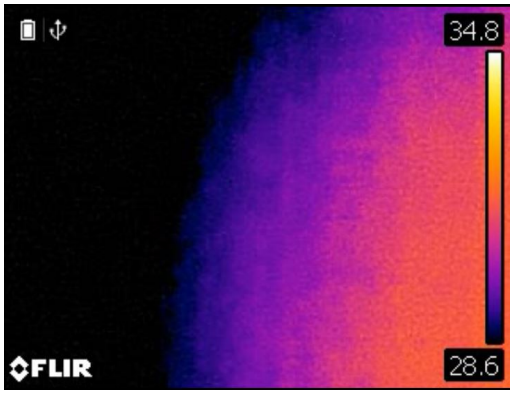
5.2.2 Værforhold

Kameraet ble testet mot åpen himmel og mot skyer. I figur 5.2.2 kan vi se et eksempel på hvordan et bilde ser ut med kameraet rettet mot åpen himmel uten fugler i bildet. Dette gir lite støy å filtrere bort, som man tydelig kan se fra bildet. Dermed blir det relativt enkle bilder for filtrering, som vi for eksempel kan se i figur 5.2.1. Dersom man rettet kamera mot solen fremsto den som en tydelig blob, men som vi kommer til å se i underavsnitt 5.4.1 er det en mulighet for at fugler vil være for kalde i forhold til solen, slik at de kan bli filtrert bort. Det er ikke utført målrettet testing for å bekrefte dette.

Figur 5.2.2: Her kan vi se et bilde tatt med kameraet rettet mot en skyfri himmel. Vi kan se at det ikke reflekteres noe IR-stråling fra himmelen.

Filtreringen av bildet blir mer krevende dersom man har skyer på himmelen. I figur 5.2.3 ser vi at kameraet har kalibrert seg slik at skyen blir godt synlig på bildet.

Vi ser på temperaturskalaen til høyre at temperaturskalaen går fra 28,6 °C til 34,8 °C i figur 5.2.3. Denne skalaen inngår i FLIR sin software til kameraet. Dersom man stiller inn kameraet til en annen skala ble dette betydelig forbedret, men det er usikkert hva dette gjør med rekkevidden til kamera, da det ikke ble mulig å teste dette i tilsvarende forhold som de i underavsnitt 5.2.1.



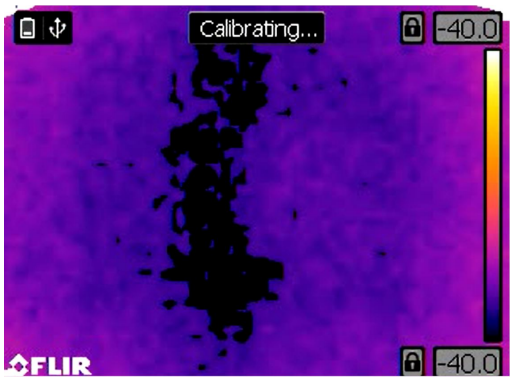
Figur 5.2.3: Her kan vi se et bilde tatt med kameraet rettet mot kanten av en sky. Vi kan se at skyen, altså områdene med farge, gir en betydelig større utfordring når bildene skal filtreres.

Det er også usikkert hvordan systemet fungerer i andre værforhold som snø, regn og tåke, siden disse værforholdene ikke ble testet. Derimot har FLIR funnet ut at ved sikt på under 300 meter (synlig lys) er det neglisjerbar forskjell i synsrekkevidde mellom infrarøde instrumenter og det blotte øye [26]. Utfordringen ligger i refraksjon av elektromagnetiske bølger når de kommer i kontakt med regndråper eller snø [27]. Dette kan føre til en redusert sannsynlighet for at varmestråling fra fugler når fram til sensoren. I tillegg er det mulig at regn og snø kan feilaktig bli detektert av systemet som fugler.

5.2.3 Kalibrering

Med jevne mellomrom kalibrerer kameraet seg. Kameraet er et kommersielt produkt, og den forhåndsinstallerte programvaren vil justere kameraet for å gi større kontrast mellom de varme områdene i et bilde. Dette skjer oftere dersom varme objekter raskt forsvinner ut av bildet, eller dersom det er mulig å finne store kontraster som for eksempel når det er skyer. Dette er ugunstig ettersom vi ikke ønsker at kameraet skal fremkalle varmesignaturer på denne måten. Når kameraet kalibreres oppstår det mønstre på skjermen, som vist i figur 5.2.4, og dette er utvilsomt noe av det som ga store feil i form av falske positiver. Dette kunne resultere i alt fra null til flere hundre feil (falsk positiv) i timen, avhengig av værforhold og innstillinger på kameraet. Få til ingen feil oppsto ved skyfri himmel og godt vær, flest feil oppsto når det var skyer eller hyppige endringer i værforhold. Det er vanskelig å gi konkrete tall på feil siden det var mye menneskelig aktivitet i og rundt kameraet som også kunne påvirket dette.

En test som ble utført var å peke kameraet mot en fugl, slik at man lot kameraet kalibrere seg. Bildene ville ofte gi tydelige kontraster mellom fugler og bakgrunn uavhengig om det var skyer eller ikke siden fuglene var mye varmere enn bakrunnen. Utfordringer oppsto når fulgen forsvant ut av synsfeltet til kamera, og kameraet ville forsøke å kalibrere seg mot det nye motivet. Dersom det var skyer ville dette ofte gi output som i figur 5.2.4 etterfulgt av støyfulle bilder som i figur 5.2.3 som vi undersøkte i underavsnitt 5.2.2. Dette var et mer ubetydelig problem ved skyfri himmel, ettersom at den nye outputen ville bli som i figur 5.2.2.



Figur 5.2.4: Et eksempel på hva som skjer når kamera kalibreres. Bildet inneholder mye støy.

Når dette skjer vil mange blobs bli detektert i bildet, og det vil opprettes trackere for disse. Det vil føre til at programmet teller veldig mange fugler som ikke er der, i tillegg til at prosessoren ikke har nok prosesseringskraft til å oppdatere alle trackerene.

Bildene tatt med kameraet har også visse markeringer i kantene, slik som FLIR-logoen, temperaturskalaen, og så videre. Disse kan sees i for eksempel figur 5.2.4. Under bildebehandlingen ble disse delene av bildene klippet vekk, som førte til en reduksjon på 30 % i bildenes størrelse.

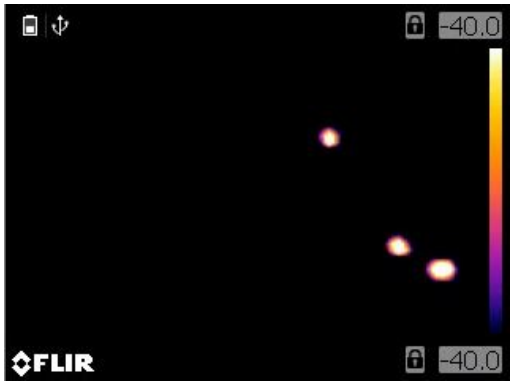
5.3 Prosesseringsenheten

Utendørs testing ble i hovedsak utført på en bærbar datamaskin med 16GB RAM og i7-prosessor, men innendørs testing ble gjort med Pi-en. Denne testingen med en last på 0-5 blobs per bilde var det uproblematisk å oppnå et bildefrekvens på 30Hz ved blob-deteksjon. Siden kameraet vi bruker har en maksimal bildefrekvens på 9Hz, er prosesseringsenheten kraftig nok til å håndtere prosesseringen. Dersom man fikk store mengder data som skulle prosesseres (mer enn 10 objekter), ga det betydelig større prosesseringstid for tracking. Dette var heller ikke en bærbar data i stand til å håndtere, og det var ikke forventet at prosesseringsenheten skulle bli utsatt for slike mengder data.

5.4 Programvare

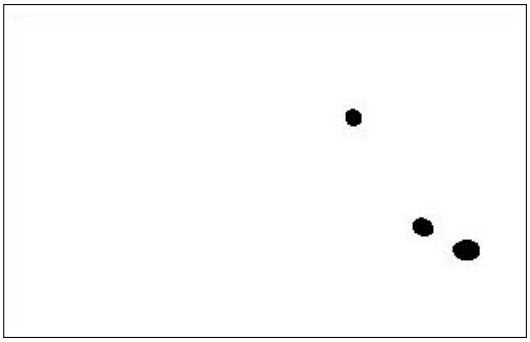
5.4.1 Filtrering

For skyfri himmel gir filtreringen forventet resultat. Vi kan for eksempel se på figur 5.4.1 som et eksempel. Bildet viser 3 måker innenfor synsfeltet, alle i en estimert høyde rundt 15-20 meter.



Figur 5.4.1: Bilde av 3 måker mot skyfri himmel.

Vi kan så bruke filtreringsfunksjonen på dette bildet, og får et resultat som vist i figur 5.4.2. Dette resultatet er godt egnet for blob-deteksjon med klare blobs på en hvit bakgrunn. I dette eksempelet hadde de tre fuglene omtrent lik varmesignatur. I ett tilfelle under testingen hadde en fugl lavere varmesignatur som følge av at den fløy høyere oppe enn to andre fugler. Fordi varmesignaturen var betydelig lavere for fuglen som fløy høyere oppe enn fuglene som fløy lavere, ble denne tolket som bakgrunn og filtrert bort. Likevel klarer programvaren å detektere denne etter fuglene i forgrunnen har forsvunnet, men dette kun ut fra tilfeldighetene til hvilke fugler som kom inn og ut av bildet først.



Figur 5.4.2: Filtrering av bildet i figur 5.4.1.

Eksempelet i figur 5.4.2 var tilnærmet ideelle forhold for systemet. Dersom man for eksemel har delvis skyet vær, som vi kunne se et eksempel på i figur 5.2.3, blir resultatet mindre egnet for blob-deteksjon og tracking. En filtrert utgave av figur 5.2.3 ser vi i figur 5.4.3. Resultatet i seg selv er ikke overraskende, ettersom at Otsus metode, beskevet i underavsnitt 4.3.2, nettopp skal skille forgrunn fra bakgrunn. Likevel gir dette store mengder støy, slik at filtreringen ikke klarer å filtrere bort en sky. Riktignok ville ikke dette eksempelet gitt noen falske positiver, siden denne bloben blir for stor i forhold til parameterene som ble brukt for blob-deteksjon under testing. En mindre sky, eller en sky med større kontraster og lokale maksimum og minimum, kunne derimot gitt falske positiver.

Figur 5.4.3: Skyen i figur 5.2.3 blir ikke filtrert bort.

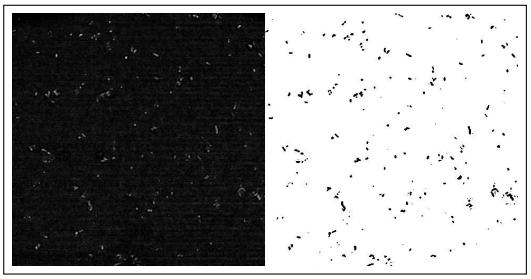
For å skaffe en bedre forståelse av hvordan filteret oppfører seg ble det også testet med falsk data i form av bilder med blobs og støy. I figur 5.4.4 ser vi et mørkt bilde, noen hvite objekter, samt en del bakgrunnstøy (enkeltpiksler / små grupper piksler som er lys). Resultatet etter Otsus metode blir tydelige blobs på en hvit bakgrunn. Det ser ikke ut til at antallet og størrelsene til blobbene blir fullstendig bevart, men vi ser at filtreret som en helhet utfører oppgaven med å fremheve blobs.

Ser vi derimot på figur 5.4.5 har vi et lignende bilde, men med mer støy i forhold til eksempelet i figur 5.4.4. Filteret klarer ikke å filtrere dette bildet, slik at det filtrerte bildet også bare er støy uten at vi klarer å kjenne igjen noen mønster fra det originale bildet slik som i det forrige eksempelet. Med denne mengden støy har vi gått forbi grensen over hva som er mulig med denne implementasjonen, ettersom at filteret er implementert til å analysere hele bildet som en helhet for å bestemme en terskelverdi, fremfor å finne lokale maksimum og minimum.

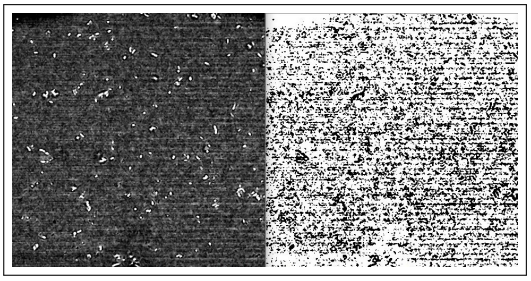
Vi kan så se på figur 5.4.6, som tydelig får fram hvordan funksjonen til *morphology* operasjonene implementert i underavsnitt 4.3.2. Til venstre i figuren har vi har et filtrert bilde med støy, og kan se veldig mange små blobs. Filtrering ved *Morphology*-operasjoner fjerner disse, som resulterer i et mye klarere bilde til høyre hvor man tydelig kan se mindre støy på de svarte og hvite områdene. Merk at operasjonen vist i dette eksempelet er modifisert for testingen slik at den utfører sterkere og dermed mer synlig filtrering.

5.4.2 Blob-deteksjon og tracking

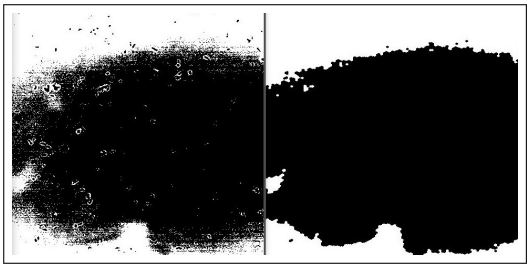
Resultatene i dette kapittelet kommer fra test 1 beskrevet i underavsnitt 5.1. I en testvideo med 20 fugler og skyfri himmel, klarte systemet å detektere 18 av de 20 fuglene. Det vil si at blob-deteksjon oppdaget 90% av alle fuglene i videostrømmen. Av disse 18 fuglene som ble detektert klarte trackingen å følge 15 av fuglene helt til de forsvant ut av skjermen. På tre av fuglene mistet trackeren fuglen en gang, og startet en ny tracker. De tre fuglene ble altså talt to ganger hver. Trackingen fungerte i 83% av de tilfellene den fikk kjøre. Resultater er vist i tabell 2.



Figur 5.4.4: Testing av filter med falsk data. Inputen til venstre har støy, men filteret leverer fremdeles tydelige blobs som vi ser til høyre. Ser vi nøye kan man også kjenne igjen noen av de originale objektene.



Figur 5.4.5: Testing av filter med falsk data. Inputen til venstre har betydelig mengder støy, og vi kan se til høyre at filtret ikke klarer å behandle all denne støyen. Ser man nøye klarer man skimte noen av de originale objektene, men disse er i stor grad druknet i støy.



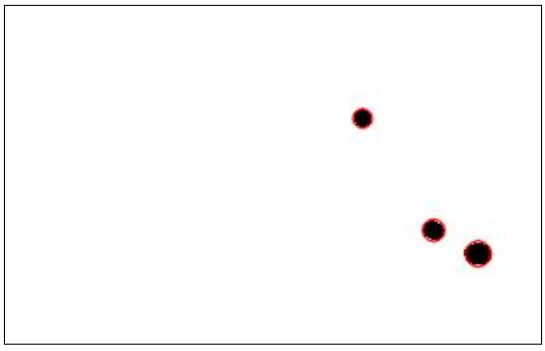
Figur 5.4.6: Testing av filter med falsk data. *Morphology*-operasjoner reduserer støy betraktelig. Mesteparten av små prikker fjernes etter at filtreringen er gjennomført.

Tabell 2: Delresultater og totalresultater fra testing av blob-deteksjon og tracking.

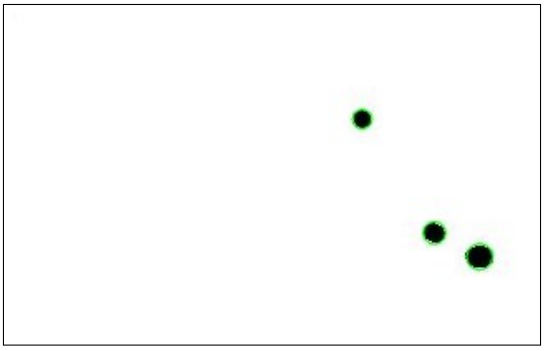
	Fugler inn	**Fugler oppdaget**	**Prosentandel korrekt**	**Prosentandel totalt**
Blob-deteksjon	20	18	90%	90%
Tracking	18	15	83%	75%
Hele systemet	20	15	75%	75%

Systemet talte totalt 15 av 20 fugler, altså 75% av fuglene, helt uten feil. Dette ligger innenfor systemkrav #11. I to tilfeller talte den ikke fuglen som fløy i bildet og i tre tilfeller talte den fuglen to ganger. Det betyr at etter de 20 fuglene hadde flydd forbi, hadde 21 fugler blitt talt av systemet. Testdataen gav ikke mulighet for å teste hvordan systemet oppfører seg dersom to fugler passerer hverandre slik at blobene deres overlapper fullstendig. Dette vil sannsynligvis være en utfordring for trackingen slik den fungerer nå. Figur 5.4.7 viser systemet som korrekt identifiserer 3 blobs. Figur 5.4.8 viser at systemet korrekt tracker 3 blobs.

Trackingen har ikke blitt testet godt nok når det har vært overskyet, men dersom filtreringen klarer å fjerne bakgrunnen helt, vil trackingen fungere som på skyfri himmel. Om det derimot vil bli med forstyrrelser fra skyene i den filtrerte videostrømmen, vil trackingen få større problemer med å følge fuglene.

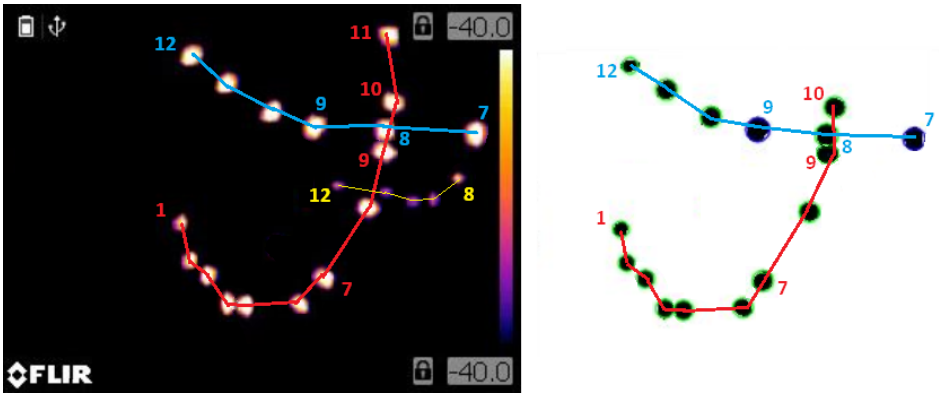


Figur 5.4.7: Programvaren identifiserer 3 blobs og markerer disse mer rød ring.



Figur 5.4.8: Systemet tracker 3 blobs og markerer disse med en grønn ring.

Et lengre utdrag av testvideoen er vist i figur 5.4.9, med IR-bildene til venstre og den filtrerte dataen til høyre. Figuren er en sammensetning av 12 etterfølgende bilder fra testvideoen. En fugl (rød) som allerede var oppdaget før bilde nummer 1 i serien trackes feilfritt fram til bilde 10. Fuglen er synlig i bilde 11 også, men er for nærme kanten av bildet til å trackes, før den forsvinner ut av kameraets synsfelt i bilde 12. Dette kan være en indikasjon på at blob-deteksjon ikke fungerer optimalt med blober på kanten av et bilde, men det var få tilfeller under testing for å kunne analysere dette nærmere. En annen fugl (blå) blir synlig i bilde 7 og trackes til bilde 8. Mellom bilde 8 og 9 brytes trackingen, og fuglen tolkes i bilde 9 som en ny fugl. Den trackes så fram til bilde 12, hvor bildeserien slutter. Det at fuglen oppdages på nytt vises i den filtrerte dataen som en blå kant rundt objektet, der objekter som trackes fra forrige bilde har grønn kant. En fugl som enten er mindre eller lengre unna enn de to andre (gul) er også synlig i kameraet fra bilde 8 til 12, men den har blitt filtrert bort. Denne oppdages først etter at bildeseiren er over, altså i bilde 13 når de to andre fuglene er ute av synsfeltet til kameraet.



Figur 5.4.9: Et lengre eksempel på tracking av fugler fra IR-bilder.

5.5 Værstasjon

Kommunikasjon mellom Pi-en og de ulike sensorene er oppnådd, og sensorene sender ut fornuftig data. Kommunikasjonen med databasen er også testet og fungerer. Da det ikke har vært mulig å 3D-printe boksen til systemet etter at NTNU ble stengt, har værstasjonen kun blitt testet inne.

Temperatursensoren er verifisert ved å sammenligne med en kommersiell temperaturmåler, og lufttrykket ser realistisk ut sammenlignet med værvarsel fra `yr.no` på det aktuelle tidspunktet fra teststedet. Temperatursensoren viste 24,0 °C inne, mens den kommersielle viste 24,4 °C. Trykket var 1003,9 hPa, mens værmeldinga vist 1007,0 hPa. Grunnet mangel på en kommersiell luftfuktighetsensor, har denne dataen ikke blitt verifisert mot et troverdig resultat. Det er alikvel naturlig å anta at sensoren fungerer som spesifisert i databladet[15], da sensoren sender ut realistisk data og den registrerte luftfuktigheten øker når man blåser på sensoren. Til vanlig lå luftfuktigheten på underkant av 30%, mens den økte mot nærmere 90% når man blåste på den. Da luftfuktigheten registreres av samme komponent som temperaturen og lufttrykket som er verifisert, underbygger dette ytterligere at luftfuktighetssensoren gir riktig data.

Nedbørsmåler er verifisert ved å helle vann over den, mens vindretnings- og vindhastighetsmåleren er testet ved å manuelt snu på dem.

5.6 Nettside og database

Fordi systemet ikke er testet i sin helhet over lengre tid ble databasen fylt med testdata. Den er generert slik at en kan se forskjell på forskjellige data. Programmet som genererer testdata ligger også i jolyu sin GitHub [12], under `nettside/data`. Nettsiden leser denne dataen og produserer grafer som visualiserer dataen på en fornuftig måte, og oppfyller derfor systemkrav `#14`.

Overføring av data fra prosessorenhet er også testet ved å skrive værdata og telemetri til databasen. Dette fungerer også slik som det skal, og dermed oppfyller systemkrav `#13`.

Systemkrav `#01` og `#10` er oppnådd, siden systemet utfører oppgavene sine. Igjen så er deler av dette gjort med testdata, siden systemet ikke er testet med ekte data som helhet.

5.7 Systemet som en helhet

Grunnet koronapandemien våren 2020 var det enkelte ting som var planlagt å testes, men som ikke ble gjennomført. Mangel på 3D-printer førte til at produktets ytre struktur ikke ble konstruert, slik at det ikke lot seg teste i hvilken grad denne er værbestandig, altså systemkrav `#02`. Strukturens design er laget for å være mest mulig vanntett og værbestandig, men uten testing er dette vanskelig å fastslå. Systemkrav `#08` er gjennomført i den grad at boksen er designet, men ikke 3D-printet.

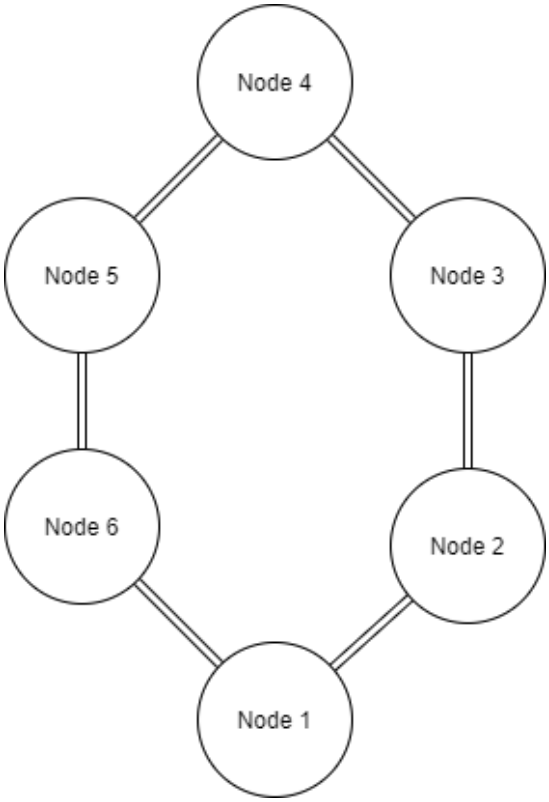
Det var heller ikke mulighet å sette alle delesystemene sammen, slik som å feste boks og sensorer til stativ og sette det utendørs til testing. Det største problemet var at testing av et komplett system ble umulig, da delsystemene ble spredt mellom gruppemedlemmer for å kunne fortsette arbeid hver for seg rundt om i Norge.

6 Konklusjon og anbefalinger

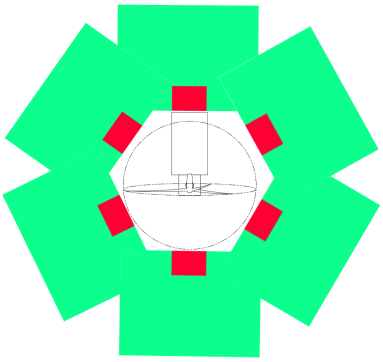
6.1 Videreutvikling

6.1.1 Maskenettverk

Systemet har slik det er nå en stor svakhet: aktivitet detekteres kun fra en side av en eventuell turbin, som gjør at mange fugler ikke blir detektert. Det vil heller ikke oppdages om en fugl faktisk kolliderer med turbinen. En tenkt videreutvikling av systemet er dermed å utvide til et maskenettverk, bestående av flere noder med hvert sitt kamera og prosesseringsenhet, som illustrert i figur 6.1.1 og 6.1.2.



Figur 6.1.1: Et tenkt maskenettverk for å dekke en hel vindturbin med seks noder.



Figur 6.1.2: Konsept for hvordan seks kameraer kan detektere fugleaktivitet inn mot en vindturbin fra alle retninger. Det grønne området er deteksjonsområdet og rødt er boksenes plassering.

En hovedprosesseringsenhet kan ta inn data fra alle nodene, og bruke dette for å lage et 2D-kart over flymønstre. Til dette må det brukes sannsynlighet for å forsøke å modellere banene til fuglene som er detektert, sånn at man klarer å tracke en fugl fra en node til en annen. I informasjonsinnhentingen til dette prosjektet, ble det sett eksempler på hvor kalman-filter ble tatt i bruk for et slikt formål [28]. Et maskenettverk kan potensieltsett også brukes for å detektere om en fugl har kollidert med turbinene. Dersom en fugl flyr inn mot en turbinen, uten å fly ut igjen, vil det være en reell sannsynlighet for at fuglen har kollidert.

Dette kan også hjelpe i områder uten vindkraftutbygging der man skal telle fugler på et større område. Dette 2D-kartet over flymønstre vil for eksempel kunne brukes for å optimalisere plasseringen av vindturbinen, dersom fuglene har en tendens til å fly over spesifikke områder innenfor deteksjonsområdet.

6.1.2 Oppgradering av IR-sensoren

IR-kameraet som ble brukt er et kommersielt kamera designet for å være brukervennlig. Dette skaper en del begrensninger rundt dataen vi får ut fra kameraet. Spesielt har den automatiske kalibreringen, diskutert i underavsnitt 5.2.3, skapt problemer. I en videreutvikling burde dermed dette kommersielle kameraet byttes ut med kun en IR-sensor. Dette vil gi systemet større kontroll over rådataen fra kameraet. En IR-sensor med lengre rekkevidde og bedre oppløsning vil også gjøre det mulig å kartlegge arter og størrelser på fugler [29]. Dette minsker behovet for et konvensjonelt kamera, som også ville vært avhengig av å ha god sikt. Kunnskap om art kan også i teorien kombineres med fuglens størrelse på skjermen for å slå fast avstand, gitt at fugler av den aktuelle arten har en kjent størrelse. Dette vil sannsynligvis være komplisert og kreve mye testing.

6.1.3 Ordinært bilde

For enkelt å kunne sjekke om dataen fra fugletelleren er rimelig, kunne det vært gunstig å supplere IR-kameraet med et ordinært kamera som kun tar bilder når IR-kameraet detekterer en fugl. Dette bilde kan så bli sendt til databasen, og gjort tilgjengelig på nettsiden. Dette gjør det mulig for brukeren å enkelt dobbeltsjekke dataen, i tillegg til at det gjør det mulig å kartlegge art, enten automatisk ved hjelp av maskinlæring eller av et menneske. Dette vil kun fungere på dagtid og med god sikt.

6.1.4 Uavhengighet fra infrastruktur

For at brukeren skal ha større frihet ved plasseringen av produktet, er det viktig at det har sin egen strømforsyner og ikke er avhengig av å kobles til strømnettet. Dette er spesielt viktig dersom man ønsker å overvåke et sted uten foreløpig installerte vindturbiner, da dette kan være langt unna eksisterende strømnett. I en videreutvikling av produktet, vil det dermed være lurt å se på mulighetene for å koble produktet til et batteri, eventuelt supplert fra et lite solcellepanel eller liten vindturbin.

Det vil også være aktuelt å utforske muligheter for å ikke være avhengig av Wi-Fi for å overføre data, slik at systemet kan brukes fritt flere steder. Dette kan for eksempel erstattets ved å bruke mobilnettverk.

6.1.5 Programvare

Filtreringen av bildene hadde sine begrensninger i dette prosjektet, som diskutert i underavsnitt 5.4.1. Ettersom det også hadde vært ønskelig å ta i bruk en annen IR-sensor som nevnt i underavsnitt 6.1.2 ville filtreringsoppgaven blitt mer utfordrende og komplisert. Flere typer filter måtte blitt tatt i bruk og kombinert for å pålitelig kunne filtrere bilder i forskjellige typer værforhold. Dette vil kunne brukes sammen med værdata fra værstasjonen slik at systemet automatisk bestemmer type filter avhengig av værforholdene.

6.1.6 Nettside og database

Videreutvikling av nettsiden og databasen vil kunne være å legge til flere spesialiserte grafer for forskjellige data. Dersom det blir behov for et maskenettverk vil det også måtte være nødvendig

å legge til funksjonalitet for å kunne støtte dette. Det kan være et kart der du kan se de forskjellige systemene og statistikk for hver enkelt node.

Databasen har en svakhet, den er relativt treg. Samtidig er den meget fleksibel. Det finnes flere løsninger som håndterer dette, men å gå over til en database som ikke er så fleksibel mot at den blir raskere er en mulighet.

6.2 Bruk

Etter at systemet er satt ut på ønsket område, vil produktet automatisk registrere fugler og værdata uten ytterligere menneskelig interaksjon. Produktet er svært enkelt å sette opp, og krever kun tilgang til en vanlig stikkontakt og internett.

Produktet er i utgangspunktet vedlikeholdsfritt. Det vil allikevel kunne oppstå situasjoner der menneskelig interaksjon er nødvendig. Dette kan for eksempel være dersom kameralinsen blir blokkert, eller pålen systemet står på velter.

6.3 Konklusjon

Systemet bruker et IR-kamera, en prosesseringsenhet og bildebehandling for å detektere, telle og spore fugler i kameraets synsfelt. Systemet sender antall registrerte fugler sammen med værdata fra systemets egne sensorer til en database. En nettside henter data fra databasen, og sorterer disse etter brukerens ønske. Under testing oppnåde systemet en treffrate, antall fugler korrekt registrert, på 75% i klart vær. Systemet har lavere treffrate når det er skyer. Dette kommer i hovedsak av den automatiske kalibreringen av IR-kameraet. Alle delsystemene fungerer, fra overvåking av fugleaktivitet og innsamling av værdata til presentering på en nettside.

6.4 Takk

Takk til vår faglige veileder Bjørn B. Larsen for god veiledning.

Takk til Institutt for Elkraftteknikk ved NTNU for lån av IR-kameraet, FLIR C3.

Takk til tålmodige foreldre og gode venner for å lese gjennom og sjekke dokumentet.

Takk til Institutt for Elektroniske Systemer for lån av en Raspberry Pi 3B til testing.

Takk til Torjus Bakkene ved Ascend NTNU for teknisk veiledning.

Takk til Omega verksted for teknisk veiledning og lokaler.

Referanser

[1] L. Jørgensen. (10. mai 2016). Vil gjøre Froan selvforsynt med strøm, Hentet fra: https://www.froya.no/nyheter/vil-gj%C3%B8re-froan-selvforsynt-med-str%C3%B8m. Lastet ned: 16.03.2020.

[2] NTB, *Fraråder vindpark på Frøya*, 4. feb. 2015. Hentet fra: https://www.nrk.no/trondelag/frarader-vindpark-pa-froya-1.133198. Lastet ned: 23.04.2020.

[3] B. Sovacool, «Contextualizing avian mortality: A preliminary appraisal of bird and bat fatalities from wind, fossil-fuel, and nuclear electricity», *Energy Policy*, årg. 37, s. 2242–2248, nr: 6. Jun. 2009. Hentet fra: https://www.researchgate.net/publication/46496253_Contextualizing_avian_mortality_A_preliminary_appraisal_of_bird_and_bat_fatalities_from_wind_fossil-fuel_and_nuclear_electricity. Lastet ned: 17.03.2020.

[4] *Vestas V27-225kW, 50 Hz Windturbine with tubular/lattice tower*, versjon 1.2.0. Vestas, 13. jan. 1994. Hentet fra: http://www.husdesign.no/lars/V27-Teknisk%5C%20spesifikasjon/gen%5C%20specification%5C%20v27.pdf. Lastet ned: 10.02.2020.

[5] T. Holtebekk. (21. apr. 2020). Infrarød stråling, Store norske leksikon, Hentet fra: https://snl.no/infrar%C3%B8d_str%C3%A5ling. Lastet ned: 21.04.2020.

[6] J. H. H. o. J. o. G. Edvard K. Barth. (des. 2019). Fugler, Hentet fra: https://snl.no/fugler. Lastet ned: 24.04.2020.

[7] (24. jun. 2019). Raspberry Pi 4, The Raspberry Pi Foundation, Hentet fra: https://www.raspberrypi.org/products/raspberry-pi-4-model-b/. Lastet ned: 24.04.2020.

[8] (13. feb. 2020). Raspbian Buster Lite, Raspberry Pi Foundation, Hentet fra: https://www.raspberrypi.org/downloads/raspbian/. Lastet ned: 23.04.2020.

[9] *OpenCV*, versjon 4.3.0, OpenCV Team, 6. apr. 2020. Hentet fra: https://opencv.org/. Lastet ned: 23.04.2020.

[10] J. Yousefi, «Image Binarization using Otsu Thresholding Algorithm», 18. apr. 2011. Hentet fra: https://www.academia.edu/36112769/Image_Binarization_using_Otsu_Thresholding_Algorithm. Lastet ned: 24.04.2020.

[11] (4. des. 2019). Miscellaneous Image Transformations, OpenCv, Hentet fra: https://docs.opencv.org/2.4/modules/imgproc/doc/miscellaneous_transformations.html. Lastet ned: 25.04.2020.

[12] (2020). Jolyu | GitHub, Jolyu, Hentet fra: https://github.com/jolyu. Lastet ned: 23.04.2020.

[13] *OpenCV: Eroding and Dilating*, versjon 4.3.0, OpenCV, 4. apr. 2020. Hentet fra: https://docs.opencv.org/4.3.0/db/df6/tutorial_erosion_dilatation.html. Lastet ned: 26.04.2020.

[14] (25. apr. 2020). OpenCv | GitHub, OpenCV, Hentet fra: https://github.com/opencv/opencv/blob/master/modules/features2d/src/blobdetector.cpp. Lastet ned: 25.04.2020.

[15] *BME280 Datasheet*, versjon 1.6, Bosh, sep. 2018. Hentet fra: https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-bme280-ds002.pdf. Lastet ned: 23.04.2020.

[16] R. electronics, *Using the I2C bus*. Hentet fra: https://www.robot-electronics.co.uk/i2c-tutorial. Lastet ned: 23.04.2020.

[17] R. Hull, *RPi.bme280*, versjon 0.2.3, Pypi, 20. mai 2019. Hentet fra: https://pypi.org/project/RPi.bme280/. Lastet ned: 23.04.2020.

[18] K.-P. Lindegaard, *Smbus2*, versjon 0.3.0, Pypi, 7. sep. 2019. Hentet fra: https://pypi.org/project/smbus2/. Lastet ned: 23.04.2020.

[19] *Weather Sensor Assembly p/n 80422*, Argent Data Systems, 2020. Hentet fra: https://www.argentdata.com/files/80422_datasheet.pdf. Lastet ned: 23.04.2020.

[20] *Datastructures in Python, 5.5. Dictionaries*, versjon 3.8.2, Python Software Foundation, 24. feb. 2020, kap. 5.5. Hentet fra: https://docs.python.org/3/tutorial/datastructures.html#dictionaries. Lastet ned: 25.04.2020.

[21] *MCP3221, Low-Power 12-Bit A/D Converter with I2C Interface*, versjon DS20001732E, Microchip, 7. nov. 2016. Hentet fra: http://ww1.microchip.com/downloads/en/devicedoc/20001732e.pdf. Lastet ned: 23.04.2020.

[22] *Dash Documentation*, versjon 1.11.0, Ploty, 2020. Hentet fra: https://dash.plotly.com. Lastet ned: 23.04.2020.

[23] *flask*, versjon 1.1.x, Pallets, 3. apr. 2020. Hentet fra: https://palletsprojects.com/p/flask/. Lastet ned: 23.04.2020.

[24] *Plotly*, Plotly, 2020. Hentet fra: https://plotly.com/. Lastet ned: 23.04.2020.

[25] *DataFrame documentation*, versjon 1.0.3, Pandas, 17. mar. 2020. Hentet fra: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html. Lastet ned: 23.04.2020.

[26] FLIR, *Seeing through fog and rain with a thermal imaging camera*. Hentet fra: https://www.flirmedia.com/MMC/CVS/Tech_Notes/TN_0001_EN.pdf. Lastet ned: 23.04.2020.

[27] T. Holtsmark og J. Skaar. (jun. 2018). Brytning - optikk i Store Norske Leksikon, Hentet fra: https://snl.no/brytning_-_optikk. Lastet ned: 24.04.2020.

[28] (26. apr. 2020). Kalman filter, Wikipedia, Hentet fra: https://en.wikipedia.org/wiki/Kalman_filter. Lastet ned: 26.04.2020.

[29] D. J. McCafferty, «Applications of thermal imaging in avian science», 21. des. 2012. Hentet fra: https://core.ac.uk/download/pdf/9650686.pdf. Lastet ned: 25.04.2020.