Sluttrapport Fase I

Prosjekt: Data control

Skrevet av: Andreas, Jens, Oda, Christin og Jie

Gruppe nr: 11 | Dato: 21.11.24

Innhold

Figurer

1 Problemstilling

Når vi besøkte Mausund feltstasjon tidligere i semesteret, informerte de oss om den vanskelige økonomiske situasjonen som de befinner seg i. For fire år siden mistet de den økonomiske støtten fra Miljødirektoratet, og fra neste år mister de den resterende støtten fra Handelens Miljøfond. Dette har skapt stor bekymring for framtiden til feltstasjonen. [2]

Under besøket uttrykte de et sterk ønske om å tiltrekke flere forskerer og forskningsinstitutt til Mausund. Økt forskningsaktivitet kan bidra med både inntekter og samarbeid som er viktige for å sikre feltstasjonens drift og utvikling.

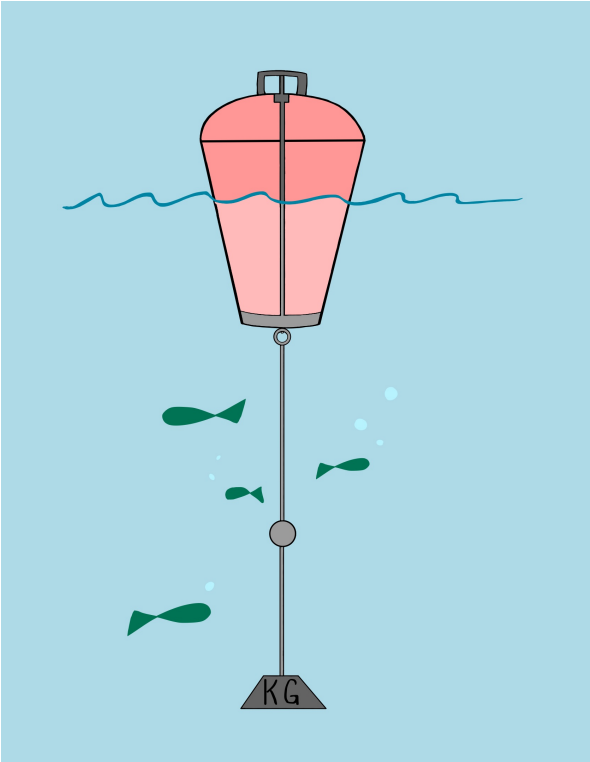
På bakgrunn av denne situasjonen valgte vi å jobbe med problemstillingen: "Hvordan kan vi tiltrekke forskerer og forskningsinstitutt til Mausund?" Hvor målet er å gjøre det mer attraktivt å drive forskning på Mausund og å gi feltstasjonen flere samarbeidspartnere, men også bidra til å styrke Mausund som en viktig aktør innen marin forskning.

2 Konsept

Vårt konsept er en flytende bøye som festes til havbunnen på et valgfritt sted innenfor naturreservatet, og som enkelt kan flyttes ved behov. Fra bøyen skal det henge en kabel med metalltråd for strekkavlastning, hvor en sensorpakke er festet. I vår prototype består sensorpakken av en PH-sensor, en temperaturmåler, en Total Dissolved Solids-sensor, og en turbiditet-sensor. Vi ønsker å få til at sensorpakkens posisjon på kabelen, skal kunne justeres. Dette er fordi det kan være interessant å undersøke om målingene varierer med ulike dybder. Videre skal det også være mulig for brukeren å bytte ut sensorpakken med andre sensorer etter behov. Det viktigste brukerkravet for oss er at bøyen skal være brukervennlig, i form av at den er lett å implementere, forstå og bruke. En brukervennlig bøye med et bredt bruksområde fører nemlig til at den kan være til nytte for en større målgruppe. Alt fra studenter som trenger data til en skoleoppgave, til forskningsinstitutter som skal lære opp ansatte eller drive diverse forskningsprosjekter.

Vi skal også utvikle en nettside som kan motta og tolke målingene fra bøyen. Målingene vil vises i grafer som oppdateres kontinuerlig, og brukerne skal enkelt kunne laste ned tidligere data. Nettsiden skal være enkel å navigere for å sikre en brukervennlig og funksjonell opplevelse.

Med hensyn til påvirkning til miljøfaktorer har vi svært lav risiko. Ettersom bøyen er mellom øyer, vil det ikke være store bølger som kan skade systemet. Det er liten risiko for skade fra dyreliv. Videre vil solceller og batteri gi god sikkerhet til 100% drift til enhver tid. En større risiko derimot er vannskade over tid, som kan mitigeres ved bruk av vanntett boks.



Figur 1: Modell av bøyen.

3 Design

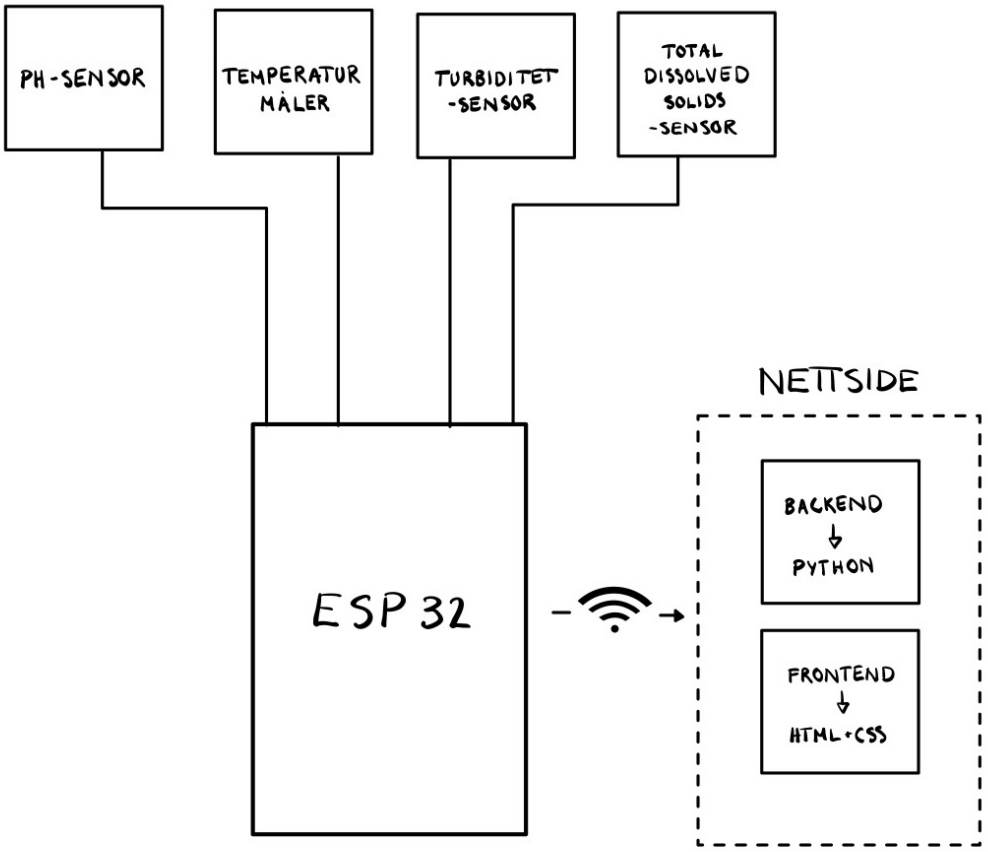
3.1 Generelt

For prototypen ønsker vi å lage en bøye som har 4 sensorer med ulike bruksområder koblet til seg. Dette er fordi vi i prototypen heller ønsker å demonstrere med et eksempel hvordan en modulbasert bøye med flere sensorer koblet til seg kunne virket. Dette er fordi i et ferdig produkt så ville det ha vært ideelt å ha flere sensorer koblet til bøyen, slik at man kan ta flere målinger, samtidig som at man har muligheten til å bytte ut noen av sensorene om man så ønsker. De sensorene vi ønsker å benytte i prototypen er følgende:

- PH sensor: Måler pH-verdien i havet
- Temp sensor: Måler temperaturen i havet
- TDS sensor: Måler konsentrasjonen av ulike partikler i vannet
- Turbiditet sensor: Måler hvor skittent/grumsete vannet er

Valget bak sensorene er at funksjonen deres var det vi følte de fleste forskerene ville ha ønsker målinger av, dersom de kom til Mausund. Disse sensorene skal styres/drives med hjelp av en ESP32, og de skal være koblet til et tilkoblingskort (breakout board), slik at

kan koble flere sensorer til ESP32 samtidig, og at man lett skal kunne fjerne og sette på ønsket modularitet/sensor om man skulle ønske. Valget bak å bruke ESP32 framfor en annen mikrokontroller som arduino, er at den har en wifi funksjon som gjør det mulig for oss å sende målingene tatt av ESP32, via wifi nettverket. Dette betyr at alle enheter koblet til dette nettverket vil kunne motta målingene som sendes av ESP32, for eksempel den eksterne nettsiden som vi har satt opp.



Figur 2: Systemoversikt

3.2 Tilkoblingsbrett

Vi skal utvikle et tilkoblingskort for ESP32 slik at sensorer kan kobles modulært til mikrokontrolleren. Det er viktig at denne god støykompatibilitet for å mitigere feil-lesing av data fra sensorene. Dette kan vi oppnå med et lowpass filter.

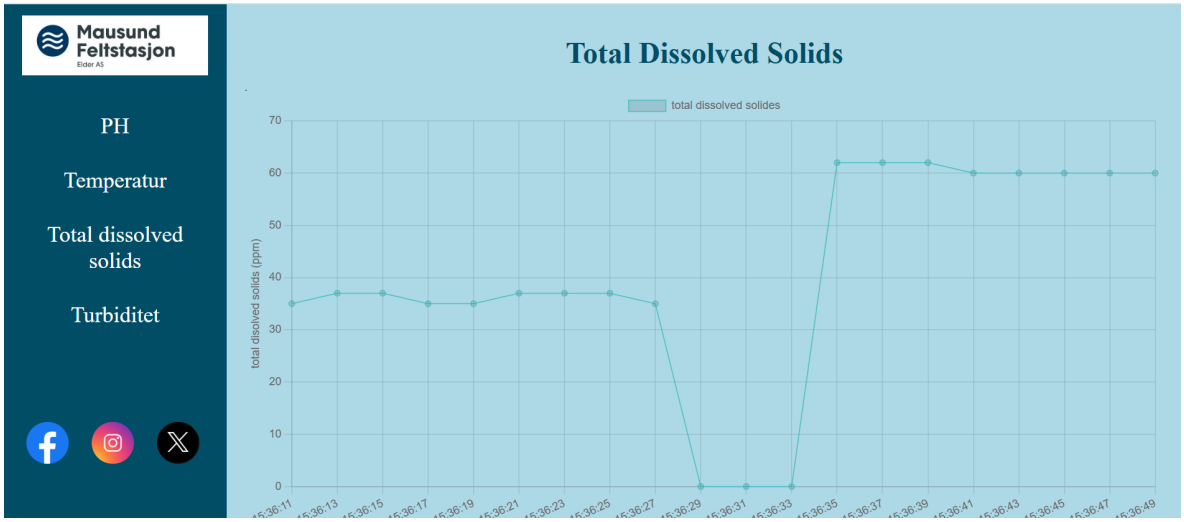
3.3 Nettside

Vi startet først med å opprette en Github som versjonskontrollsystem slik at vi kunne samarbeide og samtidig sikre at nettsiden ble utviklet strukturert og systematisk. Ved å bruke Github kunne alle jobbe med koden samtidig, som har vært essensielt for å holde produktiviteten oppe og organisert. I tillegg har det gjort det enkelt for oss å håndtere oppdateringer og feilrettinger uten å miste tidligere arbeid.

For å gjøre målingene til sensorbøyen lett tilgjengelige og brukervennlige, utviklet vi nettsiden slik at den mottar og illustrerer målinger fra ESP32-enheten. Vi utviklet nettsiden med en kombinasjon av HTML, CSS og Python, som vi organiserte i separate vinduer i VS Code for å holde kode og design adskilt, og gjøre det lettere å navigere i kodearbeidet og gjøre endringer i ett element uten at det påvirker de andre.

For å strukturere innholdet til nettsiden brukte vi HTML. Vi har fokusert på at nettsiden skal være ryddig, oversiktlig og at målingene skal vises på en intuitiv måte ved å trykke på ulike lenker knyttet til de ulike type målingene. I tillegg har vi gjort det enkelt for brukeren å laste ned dataene fremstilt i grafer på nettsiden ved å trykke på en lenke for nedlastning.

Figur 3: Nettsidens fremside



Figur 4: Graf som viser data fra TDS-sensor

Når vi har laget layout for nettsiden, har brukervennlighet vært hovedfokuset. Vi har brukt CSS for å lage et simpelt design med minimale distraksjoner slik at det er lett for brukeren å navigere inne på nettsiden. Informasjonen vises i tydelige grafer knyttet til de ulike type målingene, med kontinuerlige oppdateringer. I Figur 4 ser vi grafen over data fra Total Dissolved Solids-sensoren. Vi har implementert et responsivt design, slik at nettsiden skalerer godt på både datamaskin og mobile enheter.

HTML kan ikke ta imot målinger trådløst, derfor bruker vi Python som backend. Python tar imot og håndterer data fra ESP32-enheten, via Wifi, ved at det sendes HTTP-forespørsler til nettsiden. Målingene oppdateres kontinuerlig, og gjør det mulig for brukeren å alltid se den nyeste informasjonen fra sensorbøyen.

3.4 Alternativ design

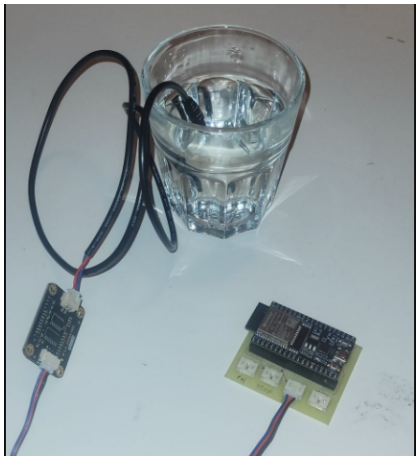
Kule på bunn av havet

Det ble vurdert en sensorenhet som slippes og synker til bunn av havet for å få data fra havbunnen kontra en midtseksjon mellom bunn og overflate. Dette ble valgt bort pga utfordringer med å hente enheten senere, samt sende data-en gjennom vannet.

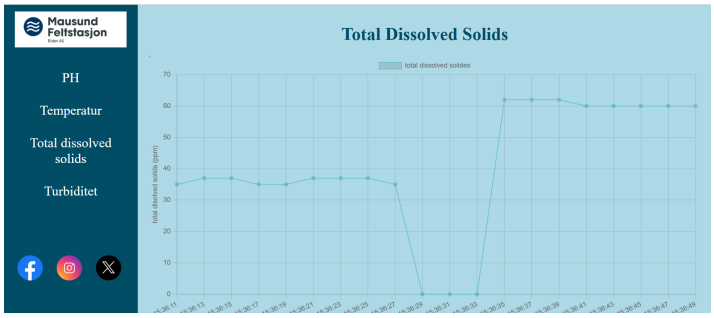
4 Implementering

4.1 prototype

Prototypen vi har laga har 4 sensorar (TDS, pH, temperatur og turbiditet) som tar ulike vannmålingar. Med moglegheit til å lett skifte ut med andre sensorar. Desse målingane blir så sendt til ei nettside som viser dataen. Som ein ser på Figur 5, består prototypen av ulike delar som gjer den i stand til å klare dette. Vi ser her bilete av den eine sensoren som er kobla til tilkoblingskortet som er festa til esp 32. Denne esp 32 har opplasta kode som gjer den istand til å sende data over wifi til en pc. Den neste delen av prototypen er koden som sørger for overføringa av data til nettsida (Figur 6)



Figur 5: Prototype

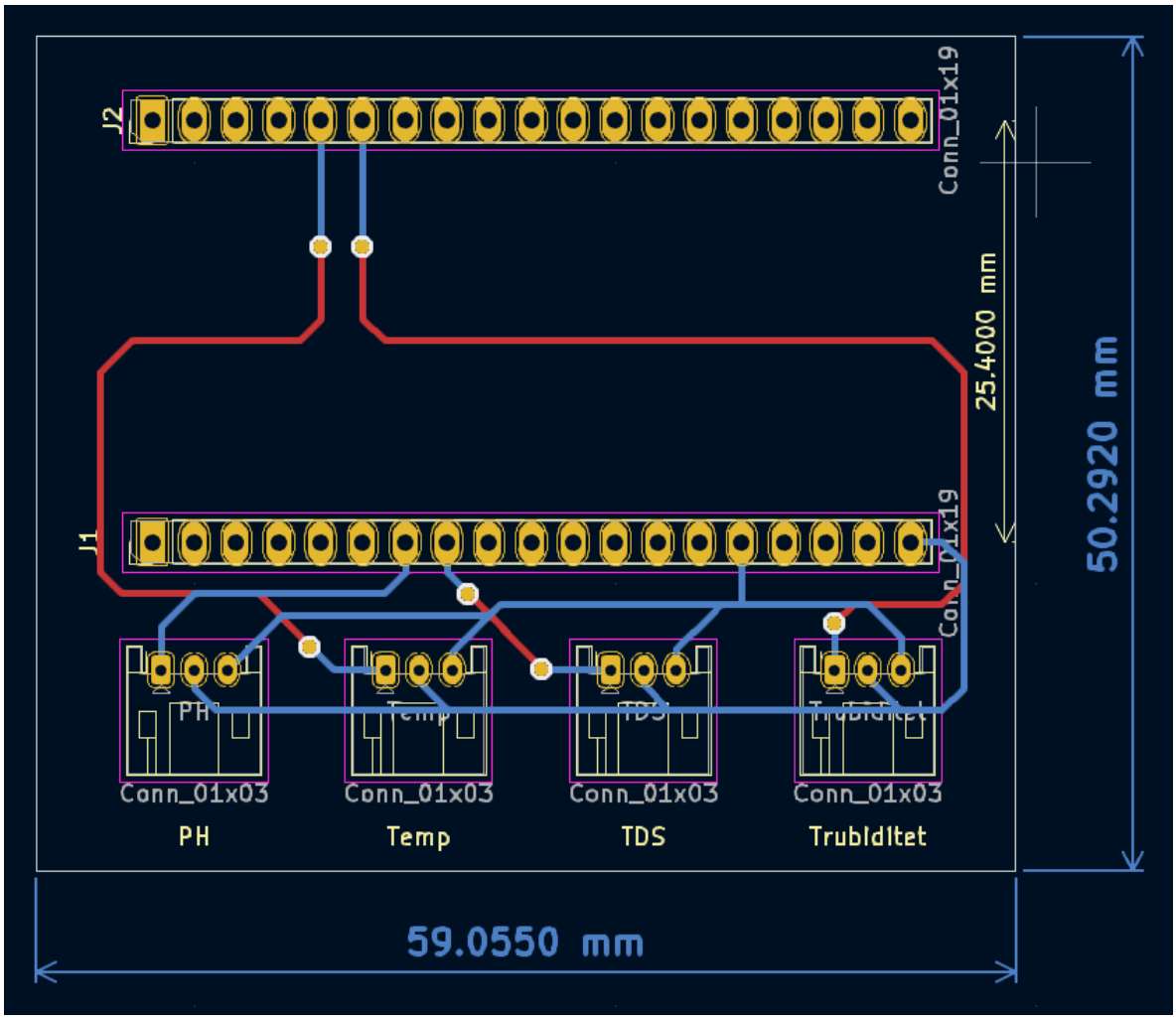


Figur 6: nettside

4.2 Tilkoblingskort

Spesifikasjoner:

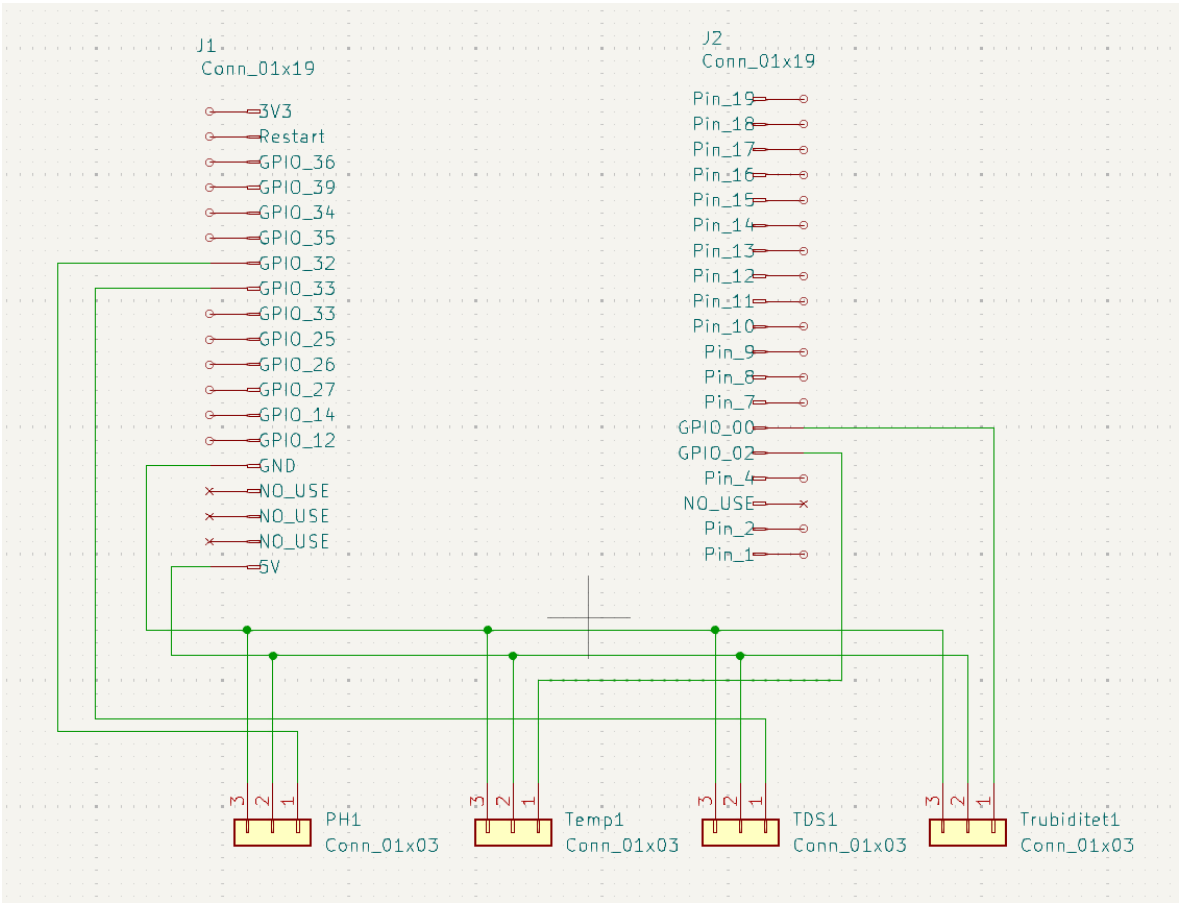
- Designet i Kicad 8
- Trace: 0.4 mm
- Via diameter: 1.4 mm
- Via drill hole: 0.9 mm
- Board size: $50mm \cdot 50mm$



Figur 7: PCB editor: Oversikt over avstand og traces

I bilde Figur 7 er alle loddepunkter ført på undersiden av kretskortet grunnet EL-pro labben sine begrensninger. Videre er det valgt å bruke 45% vinkler for impedansdiskontinuiteter, altså lage god signal flyt. Vi endret loddelandet til nålehodene fra $1.7mm^2$ til $1.5mm \cdot 2.5mm$ som gir bedre kontakt mellom tin og nålholde

Schematic diagrammet gir en oversikt over hvordan koblinger mellom komponenter er gjort. Dette er også i Schemmatic editoren vi velger fotavtrykk til komponentene i PCB editoren



Figur 8: Schematic editor

4.3 ESP 32 / Arduino IDE

Vi bruker Arduino IDE kodene fra produsenten og kombinerer den til en kode som beregner verdien til de 4 sensorene sømløst.

I Figur 9 sikrer vi at vi fortsatt er koblet til nett før vi sender over data i form av en Jsondata string. Denne dataen sendes til en nettside som opprettholdes av et Python skript.

```
// Sjekk om ESP32 fortsatt er koblet til Wi-Fi
if (WiFi.status() == WL_CONNECTED) {
  HTTPClient http;  // Opprett en HTTPClient-objekt for å lage forespørselen
  http.begin(serverUrl);  // Sett opp URL-en for forespørselen
  http.addHeader("Content-Type", "application/json");  // Angi at vi sender JSON-data

  // Eksempel på data som skal sendes (her temperatur og fuktighet)
  String jsonData = "{\"temperature\": " + String(temperature, 0) + ", \"tdsverdi\": " + String(tdsValue, 0) + "}";

  // Send en POST-forespørsel med JSON-dataene
  int httpResponseCode = http.POST(jsonData);

// Hvis forespørselen var vellykket (responskode > 0)
if (httpResponseCode > 0) {
  String response = http.getString();  // Få responsen fra serveren
  Serial.println(httpResponseCode);    // Skriv ut HTTP-responskode
  Serial.println(response);            // Skriv ut responsinnholdet (hvis noe)
  }
```

Figur 9: Kode: Overføring av data

4.4 Overføring av data med Flask

For å vise dataen på nettsiden bruker vi Python og Flask. Flask har operasjoner som *post* og *get request*. *Post request* bruker vi til å sende ett og ett datapunkt fra ESP32 til Python som tar imot dataen. *http.POST* sender dataen fra ESP32 over mobilnett til Python.

```
# Rute for å motta data fra ESP32
@app.route('/data', methods=['POST'])
def receive_data():
    global esp32_data  # Bruk global variabel for å lagre data
    data = request.json  # Hent JSON-data fra forespørselen
    if data:  # Hvis det er data i forespørselen
        esp32_data['tdsverdi'] = data.get('tdsverdi')  # Oppdater fuktighet
        esp32_data['temperature'] = data.get('temperature')
        return jsonify({"message": "Data received successfully"}), 200  # Send suksessmelding
    return jsonify({"message": "No data received"}), 400  # Send feilmelding hvis ingen data
```

Figur 10: Kode som mottar data

I Python tar vi imot dataen ved å lage en route ("/data") som kan ta imot data med en *post request*. Deretter lagrer vi dataen i to variabler ESP32['tdsverdi'] og ESP32['temperatur'].

```python
def generate_data_temperatur():
        global data_graf_temperatur
        while True:
            temperatur_verdi_graf =  esp32_data['temperature']
            temperatur_liste_synkron.append(temperatur_verdi_graf)
            timestamp_temperatur.append(datetime.datetime.now().strftime("%H:%M"))
            if len(temperatur_liste_synkron)>20:
                temperatur_liste_synkron.pop(0)
                timestamp_temperatur.pop(0)

            data_graf_temperatur = {
                "temperatur": temperatur_liste_synkron,
                "timestamp": timestamp_temperatur
                }
            time.sleep(2)
thread = Thread(target=generate_data_temperatur)
thread.daemon = True  # Tråden stopper når Flask stopper
thread.start()
```

Figur 11: lagring av dataen i lister

For hver av sensorene har vi en funksjon som tar verdien og lagrer det i en liste. Vi har begrenset listen til 20 verdier. Vi har lagt in *time.sleep* slik at funksjonene kjører på nytt etter en viss tid og henter en ny verdi. På dette tidspunktet bruker vi *datetime* biblioteket til å lagre tiden der verdien blir målt. Siden vi kun har fått temperatur- og TDS-sensoren til å virke, så har vi lagd kunstige verdier for pH- og turbiditet-sensorene.

```python
@app.route('/temperatur')
def chello_func():
    return render_template("temperatur.html")
@app.route('/download_temperatur_csv')
def download_temperatur_csv():
    # Generer DataFrame
    temperatur_data_fil = pd.DataFrame(data_graf_temperatur)
    # Lagre DataFrame som CSV i minnet (StringIO)
    csv_data_temperatur = BytesIO()
    temperatur_data_fil.to_csv(csv_data_temperatur, index=False)
    csv_data_temperatur.seek(0)  # Sett tilbake filpekeren til starten

    # Send CSV-filen som et vedlegg
    return send_file(csv_data_temperatur, mimetype='text/csv', as_attachment=True, download_name="data.csv")

@app.route('/load_temperatur')
def load_temperatur():
    return jsonify(data_graf_temperatur)
@app.route('/update_temperatur')
def update_temperatur():
    return jsonify(data_graf_temperatur)
```

Figur 12: Sending av data fra python til htmlfil

For å oppdatere filen som en skal kunne laste ned, har vi en funksjon som lagrer den nye dataen som *pd.DataFrame*. Da kan vi gjøre dataen om til en CSV-fil.

```
// Funksjon som starter oppdatering av grafen med én ny verdi om gangen
function start_updating_chart() {
    // Sjekk om vi er ferdig med å laste initial data
    if (isInitialLoadComplete) {
        // Start å oppdatere grafen med nye data hvert 2. sekund (for eksempel)
        setInterval(update_chart, 2000);
    }
}

// Funksjon for å oppdatere grafen med ny temperaturverdi
function update_chart() {
    fetch("/update_temperatur")  // Henter den siste temperaturverdien
    .then(response => {
        if (!response.ok) {
            throw new Error('Network response was not ok ' + response.statusText);
        }
        return response.json();
    })
    .then(data_temperatur_graf => {
        console.log('Ny data mottatt:', data_temperatur_graf);
```

Figur 13: html fil som tar imot dataen

I hver av HTML-filene har vi funkjsoner som *fetcher routs* fra *main-filen*. I HTML-filen har vi en funksjon *startupdatingchart* som oppdaterer grafen og CSV-filen etter et gitt intervall. Siden det er *main-filen* som oppdaterer listene med verdier, så vil alle grafene og filene oppdatere seg så lenge brukeren er på nettsiden. Når brukeren går inn på fanen til en sensor så kjører først funksjonen *load initial data*, som laster inn hele datasettet. Deretter vil grafen og filen oppdateres med én verdi om gangen ved bruk av HTML/CSS som *frontend* til nettsiden.

5 Verifikasjon og test

5.1 ESP / Python

Vi utførte dette ved å koble opp sensorene til ESP32 og brødfjølen, for å teste funksjonen deres i ulike vesker. Noe annet som var viktig for oss å teste, var om systemet oppfylte brukerkravene, blant annet at nettsiden vår skal motta kontinuerlig og oversiktlig data. Dette testet vi ved å koble opp ESP32 til wifi, og satt opp en terminal som kunne motta data fra sensorene, slik at vi kunne sende det videre til nettsiden ved et senere tidspunkt. Her er hensikten at vi benytter oss av funksjonene til Flask, et bibliotek i Python, til å sende og motta data fra sensorene.

5.2 Problem med tilkoblingskort

Etter å ha loddet og omhet tilkoblingsbretter for å teste etter kortsluttninger. Derimot ved kobling av esp 32 og sensorer til tilkoblingsbret ble det feil i den leste dataen. Dette skyldes at kobberet til elpro-labben sit brett har åpent kobber, og dermed mer mottakelig til støy. Problemet kan løses ved å enten bruke en fabrikant med bedre produksjonskapasitet eller evt legge til at passthorugh filter i tilkoblingsbrett. Disse to tiltakene øker EMC og signal stabilitet til PCBen. Videre er kortet testet for kortsluttninger ved hjelp av et multimeter

6 Konklusjon

Denne rapporten har beskrevet utviklingen av en sensorbøye og en tilhørende nettside med mål om å gjøre Mausund feltstasjon mer attraktiv for forskere og forskningsinstitutter. Gjennom prosjektet har vi utforsket hvordan teknologi kan brukes til å forbedre datainnsamling og tilgjengeliggjøring av informasjon for en bred målgruppe. Selv om prosjektet har hatt noen utfordringer, har vi oppnådd viktige resultater og lært verdifulle metoder for å overkomme tekniske og organisatoriske hindre.

Til tross for at vi ikke rakk å implementere alle sensorene fullt ut, klarte vi å utvikle en prototype som demonstrerer hovedfunksjonaliteten. ESP32-enheten har vist seg å være et godt valg for trådløs overføring av måledata, og nettsiden vi utviklet, viser at det er mulig å visualisere og tilgjengeliggjøre data på en brukervennlig måte.

Vi opplevde utfordringer med produksjonen av PCB-en, noe som resulterte i støy og ustabil signalflyt. Dette gir oss viktige innspill for fremtidige forbedringer, inkludert bruk av mer avanserte produksjonsteknikker og implementering av filtre for bedre signalkvalitet.

Prosjektet har også gitt oss erfaring med viktige samarbeidsverktøy som GitHub, noe som var avgjørende for å holde progresjonen strukturert til tross for begrenset tid og forsinkelser med komponenter.

For videre arbeid anbefaler vi følgende tiltak:

- Full integrasjon og testing av alle sensorer for å sikre et modulbasert og fleksibelt system.
- Forbedring av PCB-design og produksjonsmetode for å sikre stabilitet og redusere støy.
- Videre utvikling av nettsiden for å håndtere større mengder data og forbedre visualiseringene.

Med de erfaringene vi har gjort oss, er vi trygge på at denne løsningen har potensial til å styrke Mausund feltstasjon som et attraktivt sted for marin forskning. Løsningen kan videreutvikles til et ferdig produkt som imøtekommer både forskeres og studenters behov.

Referanser

[1] Gruppe 11, *Gruppe11*, GitHub, 2024, https://github.com/odabre/Gruppe11

[2] NRK Trøndelag, *Rydder søppel i havet ved Mausund på Frøya i Trøndelag – nå mister de støtten*, NRK, 2023, https://www.nrk.no/trondelag/rydder-soppel-i-havet-ved-mausund-pa-froya-i-trondelag-_-na-mister-de-stotten-1.16974922.

Appendix A Kildekode og GitHub-repositorium

A.1 Git repositorium

I git repositorumet har vi koden for html og python scriptet. Repositoriumet er tiljengelig med følgende lenke.

`https://github.com/odabre/Gruppe11`

A.2 Kode fra Arduino IDE

Under følger et en listing av hele koden

```
#include <EEPROM.h>
#include <OneWire.h>
#include <WiFi.h>
#include <HTTPClient.h>

#define EEPROM_write(address, p) {int i = 0; byte *pp = (byte*)&(p); for
    (; i < sizeof(p); i++) EEPROM.write(address+i, pp[i]);}
#define EEPROM_read(address, p) {int i = 0; byte *pp = (byte*)&(p); for(;
     i < sizeof(p); i++) pp[i] = EEPROM.read(address+i);}

#define SCOUNT 30
#define PH_SENSOR_PIN 33
#define TDS_SENSOR_PIN 32
#define DS18S20_Pin 2 // Temperature sensor pin
#define VREF 5.0 // ADC reference voltage (assuming 5V ADC)

// Definer SSID og passord til WiFi-nettverket du  nsker      koble til
const char* ssid = "Andreas";
const char* password = "12345678";

// Definer URL til Flask-serveren som ESP32 skal sende data til
const char* serverUrl = "http://192.168.229.19:5000/data";

// EEPROM addresses for pH calibration
#define SlopeValueAddress 0
#define InterceptValueAddress 4

OneWire ds(DS18S20_Pin);

int phAnalogBuffer[SCOUNT];
int tdsAnalogBuffer[SCOUNT];
int phBufferIndex = 0;
```

```
int tdsBufferIndex = 0;
float slopeValue, interceptValue, averagePhVoltage, tdsValue, temperature
    = 25;
boolean enterCalibrationFlag = false;

// Sampling time trackers for each sensor
unsigned long phSampleTimepoint = millis();
unsigned long tdsSampleTimepoint = millis();

void setup() {
    Serial.begin(115200);
    pinMode(PH_SENSOR_PIN, INPUT);
    pinMode(TDS_SENSOR_PIN, INPUT);
    readCharacteristicValues();

    // Koble til Wi-Fi
    WiFi.begin(ssid, password);

    // Vent til enheten er koblet til Wi-Fi-nettverket
    while (WiFi.status() != WL_CONNECTED) {
      delay(1000);
      Serial.println("Connecting to WiFi...");
      }
    Serial.println("Connected to WiFi");
}

void loop() {
    // pH sensor reading every 40 ms
    if (millis() - phSampleTimepoint > 40U) {
        phSampleTimepoint = millis();
        phAnalogBuffer[phBufferIndex] = analogRead(PH_SENSOR_PIN);
        phBufferIndex = (phBufferIndex + 1) % SCOUNT;
        averagePhVoltage = getMedianNum(phAnalogBuffer, SCOUNT) * VREF /
            1024.0;
    }

    // TDS sensor reading every 40 ms
    if (millis() - tdsSampleTimepoint > 40U) {
        tdsSampleTimepoint = millis();
        tdsAnalogBuffer[tdsBufferIndex] = analogRead(TDS_SENSOR_PIN);
        tdsBufferIndex = (tdsBufferIndex + 1) % SCOUNT;

        // TDS median calculation and compensation
        float tdsAverageVoltage = getMedianNum(tdsAnalogBuffer, SCOUNT) *
            VREF / 1024.0;
        float compensationCoefficient = 1.0 + 0.02 * (25 - 25.0); // Temp
            compensation formula
        float compensationVoltage = tdsAverageVoltage /
            compensationCoefficient; // Compensated voltage
        tdsValue = (133.42 * pow(compensationVoltage, 3) - 255.86 * pow(
            compensationVoltage, 2) + 857.39 * compensationVoltage) * 0.5;
    }
```

```
    // Temperature reading from DS18S20 sensor
    temperature = getTemp();

    // Print data every second
    static unsigned long printTimepoint = millis();
    if (millis() - printTimepoint > 1000U) {
        printTimepoint = millis();

        Serial.print("pH: ");
        Serial.print(averagePhVoltage / 1000.0 * slopeValue +
            interceptValue);
        Serial.print(" | Temp: ");
        Serial.print(temperature);
        Serial.print(" C | TDS: ");
        Serial.print(tdsValue, 0);
        Serial.println(" ppm");


        // Sjekk om ESP32 fortsatt er koblet til Wi-Fi
        if (WiFi.status() == WL_CONNECTED) {
          HTTPClient http;  // Opprett en HTTPClient-objekt for    lage
              foresp rselen
          http.begin(serverUrl);  // Sett opp URL-en for foresp rselen
          http.addHeader("Content-Type", "application/json");  // Angi at
               vi sender JSON-data

          // Eksempel p  data som skal sendes (her temperatur og
              fuktighet)
          String jsonData = "{\"temperature\": " + String(temperature, 0)
               + ", \"tdsverdi\": " + String(tdsValue, 0) + "}";

          // Send en POST-foresp rsel med JSON-dataene
          int httpResponseCode = http.POST(jsonData);

        // Hvis foresp rselen var vellykket (responskode > 0)
        if (httpResponseCode > 0) {
          String response = http.getString();  // F  responsen fra
              serveren
          Serial.println(httpResponseCode);     // Skriv ut HTTP-
              responskode
          Serial.println(response);             // Skriv ut
              responsinnholdet (hvis noe)
          }
        else {
          Serial.println("Error sending request");
          }
          http.end();  // Lukk HTTP-forbindelsen
          }
    }
}

// Get temperature from DS18S20 sensor
float getTemp() {
```

```
    byte data[12], addr[8];
    if (!ds.search(addr)) {
        ds.reset_search();
        return -1000;
    }
    if (OneWire::crc8(addr, 7) != addr[7] || (addr[0] != 0x10 && addr[0]
        != 0x28)) return -1000;
    ds.reset();
    ds.select(addr);
    ds.write(0x44, 1);
    ds.reset();
    ds.select(addr);
    ds.write(0xBE);
    for (int i = 0; i < 9; i++) data[i] = ds.read();
    ds.reset_search();
    float tempRead = ((data[1] << 8) | data[0]) / 16.0;
    return tempRead;
}

// Median filter for stabilizing sensor readings
int getMedianNum(int bArray[], int iFilterLen) {
    int bTab[iFilterLen];
    memcpy(bTab, bArray, sizeof(bTab));
    for (int i = 0; i < iFilterLen - 1; i++) {
        for (int j = 0; j < iFilterLen - i - 1; j++) {
            if (bTab[j] > bTab[j + 1]) swap(bTab[j], bTab[j + 1]);
        }
    }
    return (iFilterLen % 2) ? bTab[iFilterLen / 2] : (bTab[iFilterLen /
        2] + bTab[iFilterLen / 2 - 1]) / 2;
}

// Load pH slope and intercept values from EEPROM
void readCharacteristicValues() {
    EEPROM_read(SlopeValueAddress, slopeValue);
    EEPROM_read(InterceptValueAddress, interceptValue);
    if (isnan(slopeValue) || isnan(interceptValue)) {
        slopeValue = 3.5;
        interceptValue = 0.0;
    }
}

// Utility swap function
void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}
```