

Prestatieoptimalisatie en dataverwerking van de SeaCanairy: een mobiele datalogger voor luchtkwaliteitsmeting

Jonas BRAAT

Scriptie voorgedragen tot het behalen
van de graad van
Master in de Nautische Wetenschappen
aan de Hogere Zeevaartschool

Promotor: Werner JACOBS
Copromotor: Olivier SCHALM

Academiejaar 2024-2025

Voorwoord

Deze thesis is opgestart vanuit mijn interesse in techniek, programmeren en luchtkwaliteit. Hoewel de leercurve bijzonder steil was, aangezien de SeaCanairy een omvangrijk project vormt, ben ik erin geslaagd mijn doelstellingen te verwezenlijken. De ervaring die men opdoet bij een project van deze grootte is immers van onschatbare waarde.

Voor deze thesis wil ik graag een aantal mensen van harte bedanken. Allereerst mijn promotor, Dr. Werner Jacobs, voor zijn geduld en voor het steeds wijzen van de juiste richting. Daarnaast mijn copromotor, Dr. Olivier Schalm, voor zijn voortdurende steun en waardevolle raad. Verder wil ik ook Dhr. Alain Martinez van de Cubaanse universiteit *Universidad Central "Marta Abreu" de Las Villas* bedanken. Zijn vindingrijkheid heeft een zeer positieve invloed uitgeoefend op dit werk.

Tot slot gaat mijn oprechte dank uit naar mijn moeder, voor haar onvoorwaardelijke steun gedurende de voorbije jaren.

Samenvatting

Deze thesis heeft bijgedragen aan de verdere ontwikkeling van de SeaCanairy, een mobiel systeem voor luchtkwaliteitsmetingen. Het luchtcircuit werd geoptimaliseerd en het elektrisch schema herontworpen, waardoor het systeem toegankelijker werd voor gebruikers met geen ervaring met dit prototype.

Een belangrijke vooruitgang werd geboekt met de integratie van kastverwarming, die hygroscopische groei aanzienlijk verminderde en de nauwkeurigheid van de OPC-N3-sensor verbeterde. Verder werden diverse technische knelpunten aangepakt, zoals de invloed van de pompconfiguratie, de noodzaak van een extra I²C-lijn en elektromagnetische storingen. Dankzij hardware- en softwareaanpassingen daalde het aantal foutmeldingen van de OPC-N3-sensor van circa 60% naar ongeveer 3%.

Niet alle doelstellingen konden worden gerealiseerd. Zo moet de kalibratie van de Alphasense-gassensoren in de toekomst worden herhaald, en bleek het debiet van de OPC-N3-sensor instabiel, wat een merkbare invloed heeft op de meetresultaten.

Ondanks deze uitdagingen bevestigt dit onderzoek dat de SeaCanairy een veelbelovend platform vormt voor betrouwbare luchtkwaliteitsmetingen. Met verdere optimalisaties in kalibratie, debietstabilisatie en software kan het systeem uitgroeien tot een robuust instrument voor langdurige meetcampagnes en maritieme toepassingen.

Abstract

This thesis contributed to the further development of the SeaCanairy, a mobile air quality measurement system. The air circuit was optimized and the electrical diagram redesigned, making the system more accessible to users with no experience with this prototype

Significant progress was made with the integration of cabinet heating, which significantly reduced hygroscopic growth and improved the accuracy of the OPC-N3 sensor. Furthermore, several technical issues were addressed, such as the influence of the pump configuration, the need for an additional I²C line, and electromagnetic interference. Thanks to hardware and software adjustments, the error rate of the OPC-N3 sensor decreased from approximately 60% to approximately 3%.

Not all objectives were achieved. For example, the calibration of the Alphasense gas sensors will need to be repeated in the future, and the flow rate of the OPC-N3 sensor proved unstable, which noticeably affected the measurement results.

Despite these challenges, this research confirms that the SeaCanairy is a promising platform for reliable air quality measurements. With further optimizations in calibration, flow stabilization and software, the system can develop into a robust instrument for long-term measurement campaigns and maritime applications.

Inhoudsopgave

Voorwoord	i
Samenvatting	ii
Abstract	iii
Inhoudsopgave	iv
Lijst van figuren.....	viii
Lijst van tabellen	xii
Lijst van vergelijkingen.....	xiii
Verklarende woordenlijst.....	xiv
1 Inleiding.....	1
Hoofdstuk 2 Historische Context van de SeaCanairy	2
Hoofdstuk 3 Materialen en werkmethodiek.....	4
3.1 Ontwerp en constructie van de sensorbehuizingen.....	5
3.2 De bedrading en elektrische opbouw.....	6
3.3 Luchtvoorverwarming in de SeaCanairy	8
3.3.1 Afweging tussen thermostaat en hygrostaat in de regeling van temperatuur en luchtvochtigheid van de SeaCanairy	11
3.4 Software-integratie van sensoren in de SeaCanairy-architectuur.....	12
3.4.1 Problemen door meerdere pull-upweerstand op één I ² C-lijn	14
3.4.2 Aanmaken van een extra I ² C-bus.....	14
3.4.3 Integratie van een waterbestendige temperatuursensor buiten de koffer	16
3.4.4 Fout in kolomtoewijzing van gegenereerde CSV-bestanden	17
3.5 Communicatie- en timingproblemen bij de E+E Elektronik EE894 CO ₂ -sensor ...	18

3.6 Invloed van elektromagnetische interferentie van de luchtpomp op werking van OPC-N3-sensor.....	19
3.7 Analyse van de communicatieproblemen bij de OPC-N3-sensor	20
3.7.1 Opstelling en gebruik van de Saleae Logic Analyzer	20
3.7.2 Protocolconfiguratie en interpretatie van SPI-signalen	22
3.7.3 Analyse van de communicatiecyclus met de OPC-N3-sensor	24
3.7.4 Oplossingen voor communicatieproblemen: hardware en software	28
3.7.4.1 Hardwarematige oplossingen voor communicatieproblemen	28
3.7.4.2 Softwarematige oplossingen voor communicatieproblemen	28
Hoofdstuk 4 Resultaten van het herontwerp en technische optimalisaties	31
4.1 Verbeterde luchtcirculatie binnen het meetsysteem.....	32
4.2 Invloed van pompconfiguratie op gemeten fijnstofwaarden.....	34
4.2.1 Foutieve opstelling met de pomp achter de OPC-N3-sensor	34
4.2.2 Correcte opstelling met de pomp aan het begin van het luchtcircuit	35
4.2.3 Luchtlekkage ter hoogte van de OPC-N3-sensor	36
4.3 Effect van luchtverwarming op de prestaties van de OPC-N3-sensor	37
4.3.1 Invloed van luchtverwarming op fijnstofconcentraties in binnenlucht	37
4.3.2 Invloed van luchtverwarming op fijnstofconcentraties in buitenlucht.....	41
4.3.3 Invloed van ingestelde temperatuur van thermostaat op fijnstofmetingen...	44
4.3.4 Invloed van luchtverwarming op de relatieve luchtvochtigheid	45
Hoofdstuk 5 Praktische veldkalibratie van Alphasense-gassensoren	48
5.1 Filteren en interpreteren van sensorresultaten	49
5.1.1 Filtering van uitschieters met de Hampel-filter	50
5.1.2 Reductie van signaalruis met de Savitzky-Golay-filter	52
5.2 Kalibratie op basis van minimale en gemiddelde referentiepunten	55
5.2.1 Opbouw van de Alphasense-gassensoren	56

5.2.2 Kalibratie van de NO ₂ -sensor	56
5.2.3 Kalibratie van de O ₃ -sensor.....	59
5.2.4 Kalibratie van de SO ₂ -sensor.....	64
5.2.5 Kalibratie van de CO-sensor	66
5.2.6 Resultaten van de kalibratie voor de Alphasense-gassensoren	69
Hoofdstuk 6 Aanvullende analyse van de meetcampagne.....	70
6.1 De prestaties van de CO ₂ -sensor.....	70
6.2 De prestaties van de OPC-N3-sensor	71
6.2.1 Relatie tussen debiet en meetwaarden van de OPC-N3-sensor	75
Hoofdstuk 7 Suggesties voor verdere verbeteringen aan de SeaCanairy	77
7.1 Verbeteringen met betrekking tot de OPC-N3-sensor	77
7.2 Verbeteringen met betrekking tot kalibratie.....	77
7.3 Verbeteringen met betrekking tot dataverwerking.....	78
Conclusie.....	79
Bibliografie	80
Lijst van bijlagen	82
Bijlage I Foutenrapport	83
Bijlage II Elektrisch Diagram van de SeaCanairy.....	87
Bijlage III Code en toelichting voor de SHT30 buitentemperatuursensor	89
Bijlage IV Het configuratiebestand van de Raspberry Pi	92
Bijlage V 3D-ontwerp voor de ingang van de OPC-N3-sensor in OpenSCAD	95
Bijlage VI 3D-ontwerp voor de uitgang van de OPC-N3-sensor in OpenSCAD	97
Bijlage VII 3D-ontwerp voor het bovenste stuk van de CO ₂ sensor in OpenSCAD	101
Bijlage VIII 3D-ontwerp voor het middenste stuk van de CO ₂ sensor in OpenSCAD ...	104
Bijlage IX 3D-ontwerp voor het onderste stuk van de CO ₂ sensor in OpenSCAD	107

Bijlage X Aanvullend experiment rond verwarming van fijnstof en hygroscopische groei	111
Bijlage XI Python-programma om de gassensoren van Alphasense om te zetten in een waarde naar $\mu\text{g}/\text{m}^3$	113
Bijlage XII Python-programma voor het tellen en groeperen van logmeldingen	116
Bijlage XIII Overzicht van de SeaCanairy software	120
Bijlage XIV De aangepaste SeaCanairy.py-code.....	121
Bijlage XV De aangepaste OPCN3.py code	144
Bijlage XVI De aangepaste CO2.py-code	180

Lijst van figuren

Figuur 1: De eerste versie van de SeaCanairy verwezenlijkt door Lukas Van Den Borcht	2
Figuur 2: De tweede versie van de SeaCanairy verwezenlijkt door Cyril Dewez	3
Figuur 3: Illustratie van OpenSCAD	5
Figuur 4: Klemmenblok met vijf rijen waarvan de bovenste twee elektrisch met elkaar verbonden zijn	7
Figuur 5: De 4 zekeringen aanwezig in de SeaCanairy	8
Figuur 6: De luchtbuis die rust in de warmtezone van de Rittal-kastverwarming	9
Figuur 7: De aansluiting van het Rittal verwarmingssysteem	10
Figuur 8: Bevestiging van de SHT30-sensor aan de buitenkant van de meetkoffer voor omgevingsmetingen.	16
Figuur 9: Fout in de Seacanairy.py-code waarbij in de rode cirkel een komma ontbrak	18
Figuur 10: De Saleae Logic Analyzer 8 verbonden met de datakabels van de OPC-N3-sensor.....	21
Figuur 11: Configuratie voor de OPC-N3 SPI-protocol in de Logic software van Saleae	22
Figuur 12: Visualisatie in de Logic Analyzer-software van de SPI-communicatie tussen de Raspberry Pi en de OPC-N3 sensor. De signaallijnen MOSI (Master Out Slave In), MISO (Master In Slave Out) en CLOCK worden grafisch weergegeven. Uit de CLOCK-blokvorm is duidelijk af te leiden dat de databit wordt uitgelezen op het moment dat het kloksignaal een overgang maakt van hoog naar laag.....	23
Figuur 13: Spanningspiek op de MOSI-lijn door vermoedelijk elektromagnetische interferentie.....	24
Figuur 14: De foutmelding 0x62 met vervolgens de juiste werking van de sensor weergegeven op de monitor van de SeaCanairy	30
Figuur 15: Het nieuwe ontwerp van de SeaCanairy	32
Figuur 16: Het ontwerp voor de CO ₂ -sensor	33
Figuur 17: Ontwerp voor de OPC-N3-fijnstofsensor.....	33

Figuur 18: Testmeting met geleidelijke toename van PM-waarden als gevolg van fijnstofophoping in de behuizing van de OPC-N3-sensor, veroorzaakt door de positionering van de pomp achter de sensor.	35
Figuur 19: Testmetingen met normale PM-waarden waarbij de pomp voor de sensor is geplaatst waardoor er zich geen fijnstofophoping voordoet in de behuizing van de sensor.	36
Figuur 20: Vergelijking PM ₁₀ -waarden zonder en met voorverwarming in een binnenomgeving	38
Figuur 21: Vergelijking PM _{2.5} -waarden zonder en met voorverwarming in een binnenomgeving	39
Figuur 22: Vergelijking PM ₁ -waarden zonder en met voorverwarming in een binnenomgeving	40
Figuur 23: Vergelijking PM ₁₀ -waarden zonder en met voorverwarming op het dakterras van de AMA.....	41
Figuur 24: Vergelijking PM _{2.5} -waarden zonder en met voorverwarming op het dakterras van de AMA.....	42
Figuur 25: Vergelijking PM ₁ -waarden zonder en met voorverwarming op het dakterras van de AMA.....	43
Figuur 26: Invloed van de ingestelde temperatuur op metingen van fijnstofconcentratie PM _{2.5}	44
Figuur 27: Relatie tussen temperatuur en relatieve luchtvochtigheid, intern gemeten door de OPC-N3-sensor. De figuur toont hoe temperatuurschommelingen, veroorzaakt door het in- en uitschakelen van de kastverwarming, een directe invloed hebben op de stabiliteit van de relatieve vochtigheid.	46
Figuur 28: Invloed van de omgevingsluchtvochtigheid op de interne relatieve luchtvochtigheid binnen het meetsysteem.	47
Figuur 29: Uitschieters in het ruwe signaal van de werkelektrode van de NO ₂ -sensor .	49
Figuur 30: Toepassing van de Hampel-filter op de globale meetdata om de gassensoren te kalibreren.	51
Figuur 31: Gevoeligheid van de Hampel-filter voor uitschieterdetectie bij verschillende vensters	52

Figuur 32: Schermafdruck van Tabel 1 uit het werk van Abraham Savitzky en Marcel J. E. Golay. <i>Smoothing and Differentiation of Data</i>	53
Figuur 33: Toepassing van de Savitzky-Golay-filter na de Hampel-filter.....	54
Figuur 34: In het rood de locatie van het meetstation 40LD02 – Laakdal en in het blauw meetlocatie in Zandhoven	57
Figuur 35: Relatie tussen sensorspanning en gekalibreerde concentratie voor NO ₂	58
Figuur 36: Vergelijking van geconverteerde NO ₂ -concentraties, VMM-referentiedata en de werkelektrode tijdens de meetcampagne	59
Figuur 37: In het rood de locatie van het meetstation 42R811 in Schoten en in het blauw meetlocatie in Zandhoven	60
Figuur 38: Correctiefunctie voor NO ₂ -invloed op O ₃ -sensor	61
Figuur 39: Relatie tussen sensorspanning en gekalibreerde concentratie voor O ₃	62
Figuur 40: Vergelijking van geconverteerde O ₃ -concentraties, VMM-referentiedata en de werkelektrode tijdens de meetcampagne	63
Figuur 41: Evolutie van de ongecorrigeerde werkelektrode van de O ₃ -sensor tijdens de meetcampagne	63
Figuur 42: In het rood de locatie van het meetstation 42R811 in Schoten en in het blauw meetlocatie in Zandhoven	64
Figuur 43: Relatie tussen sensorspanning en gekalibreerde concentratie voor SO ₂	65
Figuur 44: Vergelijking van geconverteerde SO ₂ -concentraties, VMM-referentiedata en de werkelektrode tijdens de meetcampagne	65
Figuur 45: In het rood de locatie van het meetstation 41B006 - Bruxelles (Parlement UE) en in het blauw meetlocatie in Zandhoven	66
Figuur 46: Relatie tussen sensorspanning en gekalibreerde concentratie voor CO	67
Figuur 47: Vergelijking van geconverteerde CO-concentraties, VMM-referentiedata en de werkelektrode tijdens de meetcampagne	68
Figuur 48: Gemeten CO ₂ -waarden door de SeaCanairy tijdens de meetcampagne	71
Figuur 49: Overzicht van de foutieve antwoorden van de OPC-N3-sensor tijdens 10299 meetpunten, waarbij de checksumfouten in het geel zijn aangeduid.....	72
Figuur 50: PM ₁ -waarden door de SeaCanairy tijdens de meetcampagne.....	73
Figuur 51: PM _{2.5} -waarden door de SeaCanairy tijdens de meetcampagne	74
Figuur 52: PM ₁₀ -waarden door de SeaCanairy tijdens de meetcampagne.....	74

Figuur 53: PM ₁ -concentratie en debiet tijdens de gemeten piek tijdens de meetcampagne die een verband weergeeft tussen de waarde van het debiet en de waarde van de gemeten fijnstofconcentratie.....	76
Figuur 54: Doorgebrand elektrisch component op de PCB plaat van de OPC-N3-fijnstofsensor	85
Figuur 55: Brandplek in de rode cirkel te zien aan de terminal van de voedingskabel wat op een eerdere kortsluiting kan wijzen.	86
Figuur 56: Effect van voorverwarming op de impact van hygroscopische groei op PM-waarden waarbij het verwarmingselement een positieve invloed heeft op de gemeten fijnstofdeeltjes.....	112

Lijst van tabellen

Tabel 1: Invenvatis klemmenblok	6
Tabel 2: inventaris van de zekeringen	8
Tabel 3: Inventaris van het verwarmingssysteem	10
Tabel 4: De verschillende aansluitingen op de thermostaat	10
Tabel 5: Specificaties van de kastverwarming	10
Tabel 6: specificaties van de Adafruit/Sensirion SHT30-sensor [11].....	17
Tabel 7: pinconfiguratie voor de Adafruit/Sensirion SHT30	17
Tabel 8: SPI commando's voor de OPC-N3-sensor	25
Tabel 9: Analyse van succesvolle en mislukte datatransmissie bij SPI-communicatie met de OPC-N3-sensor.....	27
Tabel 11: Berekening van de coëfficiënten voor de Savitsky-Golay-filter met een 2 ^{de} orde polynoom en een venster van 11.	53
Tabel 12: De gebruikte Alphasense-gassensoren en hun serienummer	56
Tabel 13: Referentiepunten voor kalibratie van de NO ₂ -sensor met VMM-data	58
Tabel 14: Referentiepunten voor kalibratie van de O ₃ -sensor met VMM-data.....	62
Tabel 15: Referentiepunten voor kalibratie van de SO ₂ -sensor met VMM-data.....	64
Tabel 16: Referentiepunten voor kalibratie van de SO ₂ -sensor met VMM-data.....	67
Tabel 17: Lineaire omzettingsfuncties van mV naar concentratie voor de Alphasense-gassensoren	69

Lijst van vergelijkingen

$Z = \frac{ x_i - \tilde{x} }{\text{MAD}}$	Afwijkingsmaat binnen de Hampel-filter: Geeft aan in welke mate de waarde x_i afwijkt van de lokale mediaan $\tilde{x}$
$\text{MAD} = \text{mediaan}(x_i - \tilde{x})$	MAD (Mediaan van de Absolute Afwijkingen): Bepaalt de spreiding binnen een lokaal venster door de mediaan te nemen van de absolute afwijkingen $ x_i - \tilde{x} $. Wordt in de Hampel-filter gebruikt om uitschieters te identificeren op basis van robuuste statistiek.

Verklarende woordenlijst

SPI (Serial Peripheral Interface)	Een synchrone seriële communicatie-interface die typisch gebruikt wordt voor snelle gegevensoverdracht tussen microcontrollers en randapparatuur (zoals sensoren of geheugenmodules). SPI maakt gebruik van vier signaallijnen: MISO (Master In Slave Out), MOSI (Master Out Slave In), SCLK (kloklijn) en CS (chip select). In tegenstelling tot I ² C is SPI full-duplex en sneller, maar vereist meer bedrading en ondersteunt geen meerdere slaves zonder extra chip-selectlijnen.
Kruissensitiviteit	Het effect waarbij een sensor ook reageert op gassen of stoffen waarvoor hij niet bedoeld is. Dit kan leiden tot verstoorde of foutieve metingen als de kruissensitiviteit niet wordt gecorrigeerd.
.stl	STereoLithography is een bestandsformaat voor 3D-printing
Heximale waarde	Een getal weergegeven in het zestientallig stelsel (hexadecimaal), waarbij de cijfers 0–9 en de letters A–F gebruikt worden om waarden van 0 tot 15 voor te stellen. Dit stelsel wordt vaak gebruikt in digitale elektronica en programmeren omdat het compact aansluit bij binaire getallen. (voorbeeld 0x30 = 48 in decimale waarde)
Hulpelektrode (AE)	Een referentie-elektrode die niet rechtstreeks met het doelgas reageert, maar wel dezelfde omgevingsinvloeden ondergaat. Het signaal van de hulpelektrode wordt gebruikt om storingen uit de werkelektrode te corrigeren.

Hygroscopische groei	Het verschijnsel waarbij fijnstofdeeltjes waterdamp uit de lucht opnemen, waardoor hun massa en afmetingen toenemen. Dit beïnvloedt de meting van deeltjesconcentraties, vooral bij optische sensoren.
I²C (Inter-Integrated Circuit)	Een seriële communicatiestandaard waarmee meerdere digitale componenten met elkaar kunnen communiceren via slechts twee draden: SDA (data) en SCL (klok). I ² C maakt het mogelijk om meerdere apparaten op één bus aan te sluiten door gebruik te maken van unieke adressen, en vereist doorgaans pull-upweerstand om correcte signaalniveaus te garanderen.
klokfase	De relatieve verschuiving in tijd tussen het kloksignaal en het datasignaal in digitale communicatie, die bepaalt op welk moment de data correct wordt ingelezen.
Klokstatus	De actuele toestand van een kloksignaal (hoog of laag niveau), die bepaalt of het systeem zich in de actieve of inactieve fase van de klokcyclus bevindt.
MISO (Master In Slave Out)	Datapin in een SPI-bus waarop de slave data terugstuurt naar de master.
MOSI (Master Out Slave In)	Datapin in een SPI-bus waarop de master data naar de slave verstuurt.
PM₁	Ultrafijnstof kleiner dan 1 µm
PM₁₀	Fijnstof met een diameter kleiner dan 10 µm
PM_{2.5}	Fijnstof kleiner dan 2,5 µm
PTFE-buizen (polytetrafluorethyleen)	Transparante leidingen die fysiek ondoorlaatbaar zijn en chemisch inert blijven ten opzichte van de gemeten gassen en deeltjes

Pull-upweerstand	Weerstanden die tussen een signaallijn en de voedingsspanning worden geplaatst om ervoor te zorgen dat de lijn een gedefinieerde logische 'hoog' waarde heeft wanneer er geen actief signaal is. Ze worden vaak gebruikt in digitale communicatieprotocollen zoals I ² C om stabiele signaalniveaus te garanderen en storingen te voorkomen.
Werkelektrode (WE)	De elektrode in een elektrochemische sensor waarop de gewenste chemische reactie plaatsvindt. De spanning of stroom die hier gemeten wordt, is rechtstreekse maat voor de concentratie van het doelgas.
OPC (Optical Particle Counter)	Een sensor die de concentratie en grootteverdeling van deeltjes in de lucht bepaalt door lichtverstrooiing. Deeltjes passeren een laserstraal en de hoeveelheid verstrooid licht wordt gemeten om hun aanwezigheid en grootte te berekenen.

1 Inleiding

De SeaCanairy is een mobiel meettoestel dat de luchtkwaliteit meet en een project dat is opgestart door Lukas Van den Borgh [11] aan de Antwerp Maritime Academy en vervolgens heeft Cyril Dewez [8] de SeaCanairy omgevormd van een concept naar een echt meettoestel.

Het doel van de SeaCanairy is om op een autonome en gebruiksvriendelijke manier voor langere periodes de luchtkwaliteit aan boord van schepen te meten. De gemeten parameters omvatten fijnstof (PM), koolstofdioxide (CO₂), Ozon (O₃), koolstofmonoxide (CO), stikstofoxiden (NO_x), zwaveloxiden (SO_x), temperatuur en relatieve luchtvochtigheid.

Hoewel de voorgaande personen heel wat belangrijke stappen hebben gezet in de ontwikkeling van de SeaCanairy bleek dat er nog tal van tekortkomingen waren. Deze thesis zal dan ook in eerste instantie gaan over het oplossen van deze tekortkomingen alsook het herwerken van een aantal onderdelen. Verder in de thesis gebeuren er grondige testen wat de betrouwbaarheid van de SeaCanairy zal aantonen.

Hoofdstuk 2 Historische Context van de SeaCanairy

Om de context van deze thesis te begrijpen, is het essentieel inzicht te hebben in de evolutie van de SeaCanairy. Het oorspronkelijke ontwerp van Lukas Van den Borghet huisvestte sensoren voor ozon (O_3), koolstofmonoxide (CO), stikstofoxiden (NO_x) en zwaveloxiden (SO_x) in een metalen behuizing die rechtstreeks aan de buitenlucht werd blootgesteld.

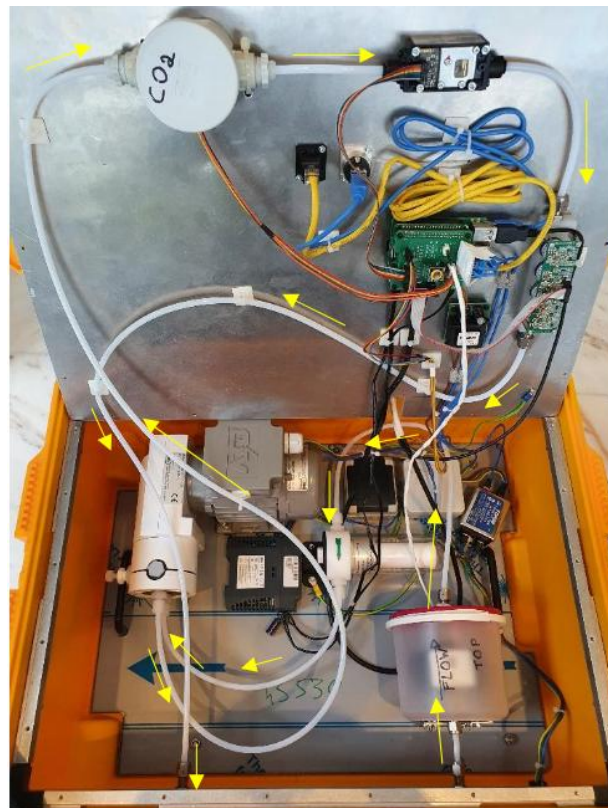


Figuur 1: De eerste versie van de SeaCanairy verwezenlijkt door Lukas Van Den Borghet

Bron: Foto uit thesis Lukas Van Den Borghet [11]

Cyril Dewez breidde het systeem vervolgens uit met een fijnstofsensor en een koolstofdioxidesensor (CO_2), en plaatste alle sensoren in de huidige iconische gele koffer. Daarnaast werden de sensoren onderling verbonden via PTFE-buizen

(polytetrafluorethyleen), transparante leidingen die fysiek ondoorlaatbaar zijn en chemisch inert blijven ten opzichte van de gemeten gassen en deeltjes



Figuur 2: De tweede versie van de SeaCanairy verwezenlijkt door Cyril Dewez

Bron: Foto uit de thesis van Cyril Dewez [8]

Het resultaat was een systeem waarin veel ontwikkelingstijd en technische vindingrijkheid was gestoken, maar dat desondanks nog verdere optimalisatie vereiste. Op hardwarevlak moest de bekabeling herwerkt worden om een duidelijk modulair elektrisch circuit te bekomen en ook de configuratie van de PTFE-slangen diende herzien te worden. Bovendien was de toegepaste pomp te krachtig en veroorzaakte ze elektromagnetische interferentie, wat de communicatie tussen de Raspberry Pi 3B en de OPC-N3-fijnstofsensoren verstoorte. Dit leidde tot diverse softwarematige communicatieproblemen. Bij ontvangst van de SeaCanairy functioneerden de koolstofdioxidesensor en de fijnstofsensoren niet altijd correct. Daarnaast was er onvoldoende tijd beschikbaar om het systeem onderworpen aan degelijke testen, die nochtans noodzakelijk zijn om de betrouwbaarheid en nauwkeurigheid van het meetsysteem te waarborgen.

Hoofdstuk 3 Materialen en werkmethodiek

Om de tekortkomingen aan de SeaCanairy tegemoet te komen werd het bestaande ontwerp via drie complementaire methoden geëvalueerd en geoptimaliseerd:

1. Kritische analyse bij ontvangst van het prototype.

Bij de eerste evaluatiefase werd het oorspronkelijke ontwerp systematisch geanalyseerd. Op basis van deze beoordeling werden gerichte verbeteringen aangebracht aan drie essentiële subsystemen:

- de fysieke behuizing,
- het luchtcircuit, waarbij de luchtstroom zodanig werd geoptimaliseerd dat deze in serie in nauw contact komt met de verschillende sensoren,
- de elektrische bekabeling, met als doel een hogere betrouwbaarheid en veiligheid.

2. Iteratief oplossen van fouten tijdens testfases.

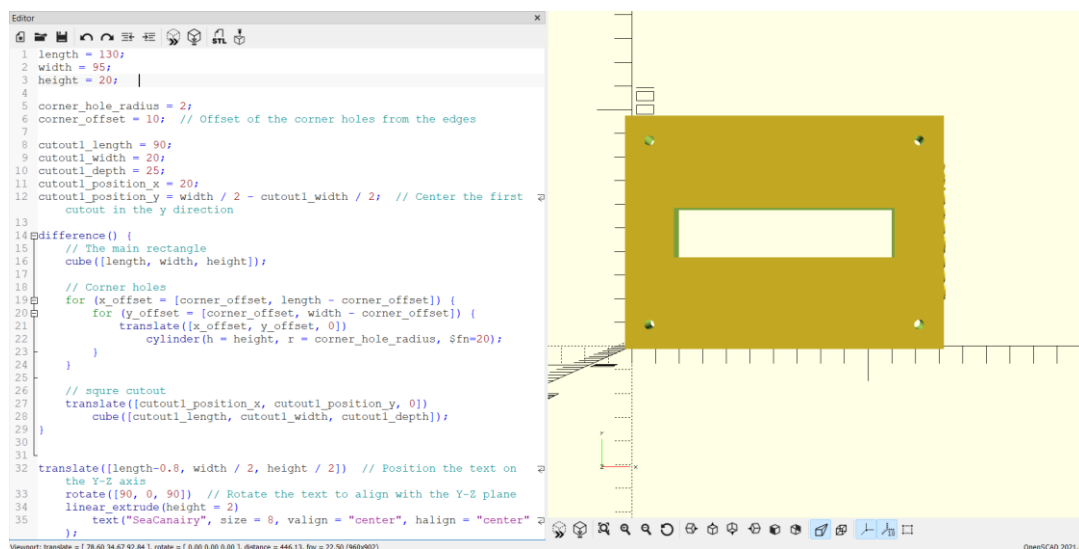
Tijdens de operationele testperiodes werden alle vastgestelde defecten of functionele tekortkomingen nauwgezet geregistreerd en opgelost. Deze iteratieve aanpak draagt bij tot een structurele verhoging van de betrouwbaarheid en stabiliteit van het systeem.

3. Gerichte experimentele configuratietesten.

In de derde benadering werd het systeem tijdelijk aangepast op componentniveau, waarbij de effecten van deze wijzigingen tijdens kortdurende tests werden geëvalueerd. Deze experimentele aanpak liet toe om de impact van specifieke ontwerpkeuzes op de globale werking van de SeaCanairy objectief te beoordelen.

3.1 Ontwerp en constructie van de sensorbehuizingen

Niet elke sensor is voorzien om een luchtslang aan te bevestigen, dus hierin dient naar een ontwerp gezocht te worden die zo weinig mogelijk plaats inneemt in de koffer voor de OPC-N3-fijnstofsensoren en de EE894 CO₂-sensor. Deze ontwerpen kunnen met software van OpenSCAD ontworpen worden. OpenSCAD is een programmeerbare 3D-modelleringssoftware waarmee objecten worden ontworpen via code (zie Figuur 3) in plaats van met een muis of grafische interface. Elk object wordt opgebouwd uit basisvormen zoals blokken, cilinders en bollen, die met wiskundige precisie kunnen worden gecombineerd, verplaatst, geroteerd of uitgesneden. Dit gebeurt aan de hand van zogenaamde constructieve ruimtemeetkunde (*constructive solid geometry*).



Figuur 3: Illustratie van OpenSCAD

Bron: eigen werk

Dankzij deze codegebaseerde aanpak is OpenSCAD uitermate geschikt voor technische toepassingen waarbij exacte afmetingen belangrijk zijn. Ontwerpen kunnen bovendien gemakkelijk worden aangepast door het wijzigen van parameters in de code, wat het hergebruiken en aanpassen van modellen zeer efficiënt maakt. Vervolgens kan de gebruiker een .stl-bestand aanmaken, dat uitgelezen wordt door de software van een 3D-

printer. De .stl-extensie staat voor STereoLithography en is een veelgebruikt bestandsformaat in de wereld van 3D-printing.

De ontwerpen die in deze thesis zijn gemaakt, zijn geprint in PLA (*Polylactic Acid*). Het probleem met 3D-geprinte ontwerpen is dat er zich *off-gassing* kan voordoen. *Off-gassing* is het vrijkomen van vluchtige organische stoffen (VOS) uit materialen na productie of gebruik. Bij 3D-geprinte onderdelen in PLA kunnen kleine hoeveelheden stoffen zoals lactide of acetaldehyde vrijkomen, wat de luchtkwaliteit licht kan beïnvloeden.[5]

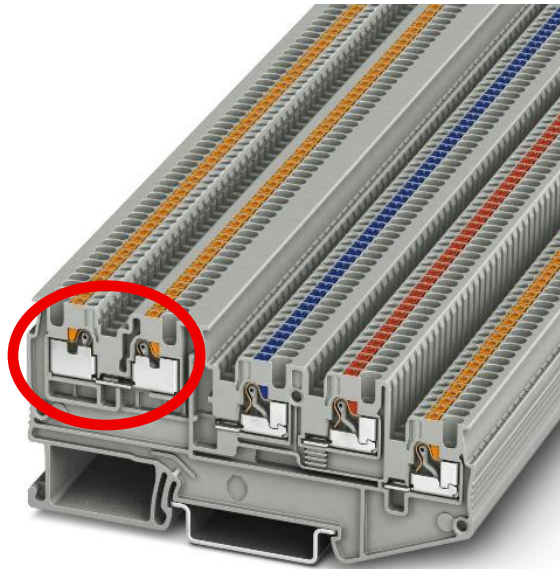
3.2 De bedrading en elektrische opbouw

Om de betrouwbaarheid en het onderhoudsgemak van het systeem te verbeteren, werd de volledige bedrading herzien door het gebruik van klemmenblokken in plaats van hefboomdraadconnectoren, zodat componenten eenvoudig kunnen worden losgekoppeld of vervangen, en door het installeren van een kabelgoot om de bedrading overzichtelijk en ordelijk te organiseren.

Tabel 1: inventaris klemmenblok

Item	Aantal	Leverancier
Phoenix Contact PTIO 1.5/S/4 Series Grey DIN Rail Terminal Block, 1.5mm ² , Quadruple- Level, Push In Termination	10	RS Components
Phoenix Contact FBS 10-3.5 Series Plug-in Bridge, 12A	5	RS Components

Extra aandacht dient besteed te worden aan de volgende Figuur 4: Klemmenblok met vijf rijen waarvan de bovenste twee elektrisch met elkaar verbonden zijn waar te zien is dat de twee bovenste oranje rijen elektrisch met elkaar verbonden zijn. Als men de kabels opnieuw in de klemmenblokken steekt dient men op te letten dat er niet per ongeluk een kortsluiting veroorzaakt wordt.



Figuur 4: Klemmenblok met vijf rijen waarvan de bovenste twee elektrisch met elkaar verbonden zijn

Bron: phoenixcontact [13]

Verder zijn er 4 zekeringen toegevoegd aan het systeem die apart te bedienen zijn en de veiligheid verhoogt. De datafiche geeft aan dat deze zekeringen een werkingstemperatuur hebben van -30° tot $+70^{\circ}\text{C}$ en voor een belasting van twee ampère bij 25°C geschikt zijn. [3,12] Daarnaast beschikken de zekeringen over een belastingscurve. Een B-curve betekent dat de zekering uitschakelt bij een stroom tussen 3 en 5 keer de nominale belastingsstroom. Deze curve wordt voornamelijk gebruikt in residentiële toepassingen met resistieve belastingen, zoals verlichtingsapparatuur en huishoudelijke apparaten met een lage piekstroom. Een C-curve betekent dat de zekering uitschakelt bij een stroom tussen 5 en 10 keer de nominale belastingsstroom. Deze curve komt vaker voor in commerciële of industriële toepassingen, waar hogere inschakelstromen mogelijk zijn, bijvoorbeeld bij inductieve belastingen of pompen, zoals de pomp die in de SeaCanairy aanwezig is. [16]



Figuur 5: De 4 zekeringen aanwezig in de SeaCanairy

Bron: Eigen werk

Tabel 2: inventaris van de zekeringen

Item	Aantal	Leverancier
Schneider Multi 9 C60SP 1P, 2A, Curve B	2	RS Components
Schneider Electric Acti 9 iC60N, 2P, 2A Curve C	2	RS Components

Het volledige elektrische schema is opgenomen in Bijlage II

Elektrisch Diagram van de SeaCanairy. Dit geeft weer dat de twee B-curve zekeringen gebruikt worden voor de primaire en secundaire voeding van de Raspberry Pi, terwijl de twee C-curve zekeringen de stroomvoorziening van de pomp en de verwarming verzorgen.

3.3 Luchtvoorverwarming in de SeaCanairy

De kastverwarming in de SeaCanairy vervult een dubbele functie. Enerzijds voorkomt ze dat de temperatuur in de meetkast onder het dauwpunt daalt, waardoor condensvorming en de daaruit voortvloeiende risico's op corrosie of elektrische kortsluiting worden vermeden. Anderzijds verhoogt de verwarming de meetnauwkeurigheid door de aangezogen lucht vóór de meting te conditioneren. Uit wetenschappelijk onderzoek blijkt dat het verlagen van de relatieve luchtvochtigheid door middel van luchtverwarming de hygroscopische groei van fijnstofdeeltjes tegengaat. Hierdoor blijft de aerodynamische diameter van de deeltjes stabiel, wat

cruciaal is voor de nauwkeurigheid van optische fijnstofmetingen. Dit effect is vooral uitgesproken bij PM_{10} -metingen in vochtige omgevingen. [10]

In de SeaCanairy wordt dit gerealiseerd door de luchtbuis fysiek te positioneren binnen de directe warmtezone van de Rittal-kastverwarming, die op een doeltemperatuur van $60\text{ }^{\circ}\text{C}$ is ingesteld met behulp van een thermostaat. Met een kwikthermometer werd vastgesteld dat het metaaloppervlak van de verwarmingselementen lokaal een temperatuur van $105\text{ }^{\circ}\text{C}$ bereikt. Bij het ontwerp moet men rekening houden met de maximale werkt temperatuur van de toegepaste zekeringen, die beperkt is tot $70\text{ }^{\circ}\text{C}$. Tijdens de testfase van de SeaCanairy trad sporadisch automatische uitschakeling van zekeringen op, vermoedelijk veroorzaakt door een te korte afstand tussen de zekering en de warmtebron. Verder om de warmte te verdelen is er een fan geplaatst die recht op de verwarming blaast om de luchtverwarming te verdelen over de gehele ruimte van de koffer. Vervolgens bleek dat er tijdens de testfase met een ingestelde temperatuur van $50\text{ }^{\circ}\text{C}$ het ingebouwde scherm van Waveshare is stukgegaan terwijl deze een specificatie heeft van een maximale temperatuur van $60\text{ }^{\circ}\text{C}$. Als er in de toekomst een nieuw scherm wordt toegevoegd dient men hierbij rekening te houden dat de ingestelde temperatuur lager dan $50\text{ }^{\circ}\text{C}$ dient te zijn.



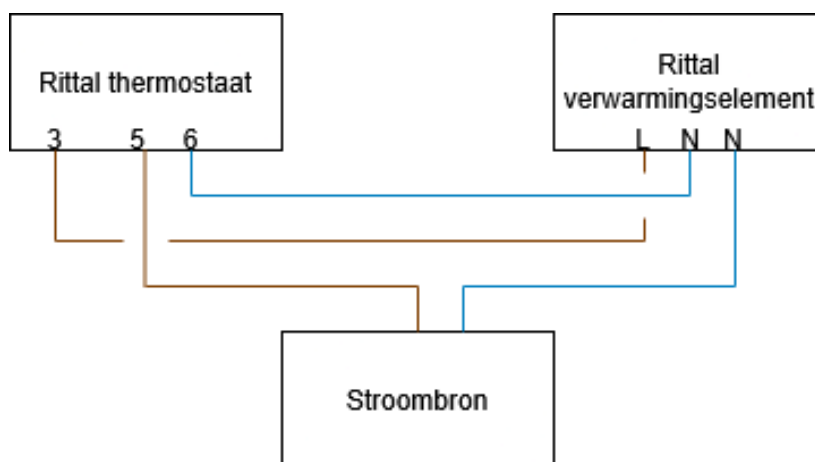
Figuur 6: De luchtbuis die rust in de warmtezone van de Rittal-kastverwarming

Bron: Eigen werk

De installatie van het verwarmingssysteem gebeurt via een thermostaat en wordt aangesloten op het elektrisch wisselstroomnetwerk. De Rittal-verwarming is ontworpen voor rechtstreekse aansluiting op een netspanning van 230 V AC . De L-geleider (bruine draad) vertegenwoordigt de fasedraad, terwijl de N-geleider (blauwe draad) de neutrale

geleider aanduidt. Figuur 7 toont een schema van de aansluiting van de verwarming en thermostaat op de stroomvoorziening. Voor een volledig overzicht van het elektrisch circuit wordt verwezen naar Bijlage II

Elektrisch Diagram van de SeaCanairy. De bijhorende tabellen geven een inventaris en beschrijving van de gebruikte componenten en hun eigenschappen weer.



Figuur 7: De aansluiting van het Rittal verwarmingssysteem

Bron: Eigen werk gemaakt met draw.io

Tabel 3: Inventaris van het verwarmingssysteem

Artikelnummer	Fabrikant	Naam
SK 3105.360	Rittal	Kastverwarming zonder ventilator
SK 3110.000	Rittal	Thermostaat Kastthermostaat

Tabel 4: De verschillende aansluitingen op de thermostaat

Bron: productpagina Rittal thermostaat [2]

Nummer	1	2	3	4	5	6	7
Beschrijving	48V, 60V, N	24V, N	Verwarming	Koeling	L	250V, N	115V, N

Tabel 5: Specificaties van de kastverwarming

Bron: Rittal productpagina [1]

Omschrijving	Bereik
Continu vermogen	10 tot 150 Watt
Nominale spanning	110 V - 240 V, 50 Hz/60 Hz

Bedrijfstemperatuur	-33°C tot 65°C
Voorzekering	4 Ampère

3.3.1 Afweging tussen thermostaat en hygrostaat in de regeling van temperatuur en luchtvochtigheid van de SeaCanairy

Een thermostaat regelt de temperatuur in de SeaCanairy door de verwarming in of uit te schakelen wanneer een ingestelde temperatuur wordt bereikt. Een hygrostaat daarentegen stuurt de verwarming aan op basis van de gemeten relatieve luchtvochtigheid.

Hoewel een hygrostaat in theorie een stabielere luchtvochtigheid biedt, wat gunstig is voor de werking van bepaalde sensoren, blijkt deze regeling in de praktijk minder betrouwbaar. Omdat de luchtvochtigheid doorgaans indirect wordt beïnvloed via verwarming, kan de temperatuur oplopen tot boven de maximaal toegestane waarden voor specifieke componenten. Dit vormt een risico voor de langdurige werking van het systeem. Daarnaast is het essentieel dat de aangezogen lucht voldoende wordt verwarmd om hygroscopische groei van fijnstofdeeltjes te vermijden.

Verder zijn de elektrochemische gassensoren van Alphasense gevoelig voor temperatuurschommelingen, wat de nauwkeurigheid van de meetwaarden beïnvloedt. Om deze redenen is gekozen voor de toepassing van een thermostaat in plaats van een hygrostaat. Deze keuze beperkt weliswaar de controle over de luchtvochtigheid, maar waarborgt een veilige temperatuur voor de interne elektronica. Een nadeel van deze aanpak is dat de verwarming, zoals de Rittal-kastverwarming, automatisch uitschakelt zodra de ingestelde temperatuur bereikt is. Hierdoor ontstaan lichte temperatuurschommelingen, wat leidt tot cyclische meetfluctuaties.

3.4 Software-integratie van sensoren in de SeaCanairy-architectuur

De integratie of vervanging van een sensor binnen de SeaCanairy-architectuur verloopt volgens een gestandaardiseerd proces bestaande uit vier stappen. Deze aanpak garandeert een consistente werking, modulaire opbouw en eenvoudige uitbreidbaarheid. Om de hoofdcode overzichtelijk en onderhoudbaar te houden, is voor elke sensor een aparte Python-bibliotheek voorzien. Deze bibliotheken worden samen met het hoofdsript `seacanairy.py` en het configuratiebestand `seacanairy_settings.yaml` opgeslagen in de hoofdmap van het project:

`/home/pi/seacanairy_project/`.

De vier stappen zijn als volgt:

1. Configuratie via YAML-bestand

In het bestand `seacanairy_settings.yaml` wordt gespecificeerd welke sensoren actief zijn. Hierdoor kunnen sensoren in- of uitgeschakeld worden zonder de hoofdcode aan te passen.

2. Import van de bijhorende Python-bibliotheek

In het hoofdsript `seacanairy.py` wordt de relevante bibliotheek ingeladen op basis van de YAML-configuratie.

3. Dataverwerking binnen het hoofdsript

De sensorfunctie wordt aangeroepen binnen de centrale verwerkingsstructuur van `seacanairy.py`, zodat meetdata correct wordt geïnterpreteerd en verwerkt.

4. Automatische logging van meetwaarden

De output van elke sensor wordt automatisch opgeslagen in een CSV-bestand. Deze logging maakt verdere analyse of export van de meetdata mogelijk.

De sensoren binnen de SeaCanairy maken gebruik van vier verschillende communicatieprotocollen, afhankelijk van hun technische vereisten en functies: SPI,

I²C, UART en analoog. Elk protocol biedt specifieke voordelen inzake snelheid, betrouwbaarheid en aansluitvereisten.

- **SPI** (Serial Peripheral Interface) is een snel en robuust protocol dat vier signaallijnen vereist per sensor (MISO, MOSI, SCK, CS). Binnen het SeaCanairy-systeem gebruikt de OPC-N3 fijnstofsensor dit protocol.
- **UART** (Universal Asynchronous Receiver/Transmitter) is een punt-tot-puntprotocol, voornamelijk toegepast voor externe modules of debugging. De Velleman VMA430 GNSS-module voor positiebepaling maakt gebruik van UART, maar werd niet actief gebruikt in deze thesis (zie thesis van C. Dewez voor meer informatie).
- **Analoog** wordt toegepast bij sensoren zonder digitale interface. Deze sturen een spanningssignaal in het millivoltbereik naar een ADC (Analog-to-Digital Converter). De elektrochemische gassensoren van Alphasense functioneren op deze manier. Via kalibratie worden de analoge spanningswaarden omgerekend naar betekenisvolle concentraties.
- **I²C** (*Inter-Integrated Circuit*) is een seriële bus die toelaat meerdere apparaten aan te sluiten via slechts twee lijnen (SDA en SCL). Elke I²C-bus vereist twee pull-upweerstand om correcte signaalniveaus te garanderen. Binnen het SeaCanairy-systeem werken meerdere sensoren via I²C, waaronder:

de EE894 CO₂-sensor,

de SFM4100 debietsensor,

de *Real Time Clock* (RTC) voor tijdsbewaring bij spanningsverlies,

en de SHT30 temperaturen luchtvochtigheidssensor, die verderop in dit hoofdstuk wordt besproken.

3.4.1 Problemen door meerdere pull-upweerstand op één I²C-lijn

indien meerdere modules elk hun eigen pull-ups hebben, worden deze weerstanden effectief parallel geschakeld. Dit verlaagt de totale weerstand en verhoogt de stroom op de lijnen. Een te sterke pull-up kan ervoor zorgen dat *slave*-apparaten de lijn onvoldoende laag kunnen trekken, waardoor dataverlies, foutmeldingen of complete communicatie-uitval kunnen optreden. Het gebruik van twee sets pull-ups impliceert dat deze weerstanden effectief parallel geschakeld zijn. Wanneer beide weerstanden dezelfde waarde hebben, wordt de equivalente weerstand gehalveerd. Dit betekent dat de stroom die door het uitgangselement van elk aangesloten apparaat moet worden afgevoerd, verdubbelt zoals blijkt uit het volgende voorbeeld:

$$\frac{1}{R_{\text{totaal}}} = \frac{1}{10 \text{ k}\Omega} + \frac{1}{10 \text{ k}\Omega} \Rightarrow R_{\text{totaal}} = 5 \text{ k}\Omega$$

Indien dit verder doorgaat, kan de equivalente weerstand onder de aanbevolen minimumwaarde van circa 2,2 k Ω komen, wat de betrouwbaarheid van de I²C-communicatie in gevaar brengt.

3.4.2 Aanmaken van een extra I²C-bus

Bij het toevoegen van een nieuwe I²C-sensor moet men opletten voor het bovengemelde probleem als er zich teveel weerstanden op 1 I²C-bus bevinden. Door de SHT30-sensor toe te voegen viel de communicatie met de CO₂-sensor volledig weg. Opzoekwerk over de interne werking van de Raspberry Pi met betrekking tot I²C bracht aan het licht dat men ook via software een I²C-bus kan nabootsen. Hiervoor dient met een paar stappen te verrichten in de Linux terminal zoals samengevat hieronder.

Om een extra I²C-bus op de Raspberry Pi te activeren via software, dient het configuratiebestand van het besturingssysteem aangepast te worden, het volledig bestand is terug te vinden in Bijlage IV

Het configuratiebestand van de Raspberry Pi. De procedure is als volgt:

1. Open het configuratiebestand `/boot/config.txt` met rootrechten via het commando:

```
sudo nano /boot/config.txt
```

2. Voeg onderaan het bestand de volgende regel toe:

```
dtoverlay=i2c-  
gpio,bus=3,i2c_gpio_sda=6,i2c_gpio_scl=5,i2c_gpio_pull=none
```

Deze regel definieert de parameters voor het activeren van een extra I²C-bus via GPIO-pinnen, met de volgende betekenis:

- `dtoverlay=i2c-gpio`: activeert de softwarematige I²C-ondersteuning op opgegeven GPIO-pinnen.
- `bus=3`: specificeert het busnummer. Indien deze al in gebruik is, kan het systeem automatisch een ander nummer toewijzen (bv. 11 in de gebruikte Raspberry Pi).
- `i2c_gpio_sda=6` en `i2c_gpio_scl=5`: definiëren respectievelijk GPIO 6 als SDA-lijn en GPIO 5 als SCL-lijn.
- `i2c_gpio_pull=none`: schakelt de interne pull-upweerstand van de Raspberry Pi uit. Dit is noodzakelijk wanneer externe weerstanden (bv. 10 kΩ) worden voorzien voor een stabiele communicatie.

Na het herstarten van het systeem kan worden gecontroleerd welke I²C-bussen actief zijn met het commando:

```
ls /dev/i2c*
```

In een standaardconfiguratie verwijst `/dev/i2c-1` naar de hardwarematige I²C-bus. De nieuw toegevoegde softwarematige bus zal typisch aangeduid worden als `/dev/i2c-11` (of het door het systeem toegekende busnummer).

De aanwezigheid van de aangesloten sensor op deze bus kan geverifieerd worden met:

```
i2cdetect -y 11
```

Indien correct aangesloten, verschijnt het adres van de sensor in de uitvoer (bv. 0x44 voor de SHT30).

3.4.3 Integratie van een waterbestendige temperatuursensor buiten de koffer

Aangezien de interne temperatuur van de koffer aanzienlijk hoger ligt dan de omgevingstemperatuur, is het noodzakelijk om temperatuur en relatieve luchtvochtigheid extern te meten. De gekozen sensor moet compact, waterbestendig en eenvoudig te monteren zijn op de buitenzijde van de behuizing. Daarom werd gekozen voor de SHT30, een temperatuur- en luchtvochtigheidssensor van Sensirion, aangeboden in een waterbestendige uitvoering door Adafruit. De sensor werd via een wartel aan de koffer bevestigd, wat een waterdichte doorgang voor de bedrading garandeert. Ze is strategisch geplaatst aan de tegenovergestelde zijde van het interne verwarmingselement, om meetverstoringen als gevolg van warmte-invloed te vermijden. De bijhorende Python-code voor communicatie met de sensor is opgenomen in Bijlage III

Code en toelichting voor de SHT30 buitentemperatuursensor, samen met een toelichting op de werking van het script.



Figuur 8: Bevestiging van de SHT30-sensor aan de buitenkant van de meetkoffer voor omgevingsmetingen.

Bron: Eigen werk

Tabel 6: specificaties van de Adafruit/Sensirion SHT30-sensor [7]

Parameter	Waarden
Optimaal bereik temperatuur	5°C - 60°C
Volledig temperatuur bereik	-40°C - 125°C
Lange termijn drift	<0.03°C/jaar
Optimaal bereik luchtvochtigheid	20% en 80%

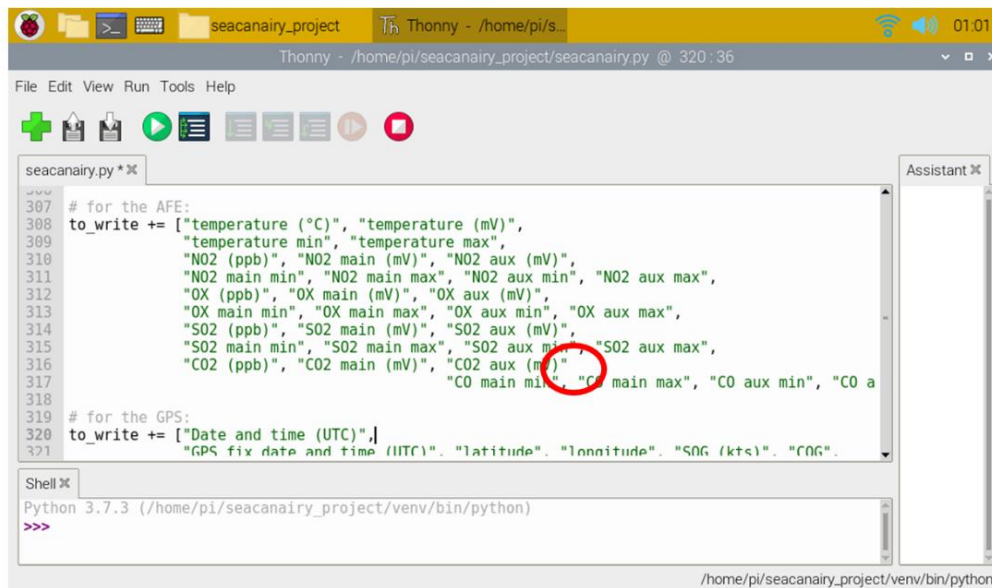
Tabel 7: pinconfiguratie voor de Adafruit/Sensirion SHT30

Sensorpin (SHT30)	Raspberry Pi GPIO
VCC	3.3V
SDA	GPIO5
SCL	GPIO6
GND	GND

3.4.4 Fout in kolomtoewijzing van gegenereerde CSV-bestanden

Tijdens het verwerken van testresultaten uit een gegenereerd .csv-bestand werd vastgesteld dat de kolomnamen niet overeenkwamen met de bijhorende sensorwaarden. De oorzaak bleek een ontbrekende komma in de Python-code van seacanairy.py, waardoor waarden in de verkeerde volgorde aan het bestand werden toegevoegd.

Deze fout toont aan dat een Python-script zonder syntactische fouten kan worden uitgevoerd, maar desondanks functioneel incorrect kan zijn. In dit geval werden meetwaarden foutief gekoppeld aan kolomnamen, wat kan leiden tot foutieve interpretaties bij data-analyse. Dit benadrukt het belang van grondige functionele validatie, waarbij niet alleen geverifieerd wordt of een script zonder foutmeldingen draait, maar ook of de output inhoudelijk klopt met de bedoeling van de programmeur.



Figuur 9: Fout in de Seacanairy.py-code waarbij in de rode cirkel een komma ontbrak

Bron: Eigen werk

3.5 Communicatie- en timingproblemen bij de E+E Elektronik EE894 CO₂-sensor

De CO₂-sensor vertoonde aanvankelijk problemen met betrekking tot checksumfouten. Een *checksum* is een controlewaarde die wordt berekend op basis van een gegevensreeks om fouten tijdens de gegevensoverdracht of -opslag te detecteren. Indien de *checksum* niet overeenkomt, beschouwt de software de meting als ongeldig. In dergelijke gevallen probeert het systeem maximaal zes keer om correcte meetwaarden van de sensor te verkrijgen. Indien dit niet lukt, wordt er een foutmelding gelogd en gaat het systeem verder met de volgende meting. Deze fout deed zich bij aanvang voor in ongeveer 50% van de iteraties. Om dit probleem te verhelpen, zijn er twee belangrijke wijzigingen in het systeem doorgevoerd.

Ten eerste werd de timing van de communicatie aangepast. Na analyse van de software bleek dat het commando dat de Raspberry Pi naar de CO₂-sensor verstuurt, onmiddellijk werd gevolgd door het uitleescommando, zonder enige pauze. Dit gaf de sensor onvoldoende tijd om het initiële commando correct te verwerken. Door een korte pauze

van 0,1 seconden in te bouwen in de functie `getCO2P()`, werd de communicatie betrouwbaarder. De aangepaste code ziet er als volgt uit:

```
write = i2c_msg.write(CO2_address, [0xE0, 0x27])  
  
time.sleep(0.1)  
  
read = i2c_msg.read(CO2_address, 9)
```

Hoewel deze wijziging de prestaties aanzienlijk verbeterde, loste dit het probleem niet volledig op. Daarom werd als tweede maatregel de I²C-communicatiesnelheid verlaagd van 100 kHz naar 50 kHz. Dit werd gerealiseerd door in het configuratiebestand `/boot/config.txt` de volgende parameter toe te voegen:

```
dtoverlay=i2c-arms-baudrate=50000
```

Sinds het doorvoeren van deze optimalisaties werkt de CO₂-sensor aanzienlijk stabiel. Af en toe treedt nog een checksumfout op, maar deze wordt door de software opgevangen, waardoor uiteindelijk bij elke iteratie een geldige CO₂-waarde beschikbaar komt.

3.6 Invloed van elektromagnetische interferentie van de luchtpomp op werking van OPC-N3-sensor

Om een correcte werking van de OPC-N3-sensor te garanderen, is het essentieel dat de luchtstroom in het systeem zo dicht mogelijk aansluit bij het aanbevolen debiet van 5,5 L/min.[4] Voor het genereren van de benodigde onderdruk wordt gebruikgemaakt van een mini-vacuümpomp met een werkspanning van 12V DC. Deze pomp beschikt over een maximaal debiet van 5 L/min en een maximale drukcapaciteit van 120 kPa. Dankzij het lage stroomverbruik (ca. 500 mA) is dit type pomp bijzonder geschikt voor mobiele en geïntegreerde toepassingen zoals de SeaCanairy.

Door het toevoegen van een luchtpomp aan het systeem kan elektromagnetische interferentie (EMI) ontstaan. Dit fenomeen treedt op in twee vormen: *radiated EMI*, waarbij elektromagnetische golven via straling door de lucht worden verspreid, en *conducted EMI*, waarbij storingen via de elektrische bekabeling binnen het circuit

worden doorgegeven [6]. In het geval van de SeaCanairy veroorzaakt deze interferentie voornamelijk problemen met de optische fijnstofsensor OPC-N3 van Alphasense. De communicatie tussen de sensor en de Raspberry Pi wordt verstoord, wat leidt tot mislukte meetpogingen en dataverlies dat kan oplopen tot meer dan vijf minuten.

Om dit te verifiëren, werd een eenvoudige test uitgevoerd. Het meetprogramma van de SeaCanairy werd automatisch uitgevoerd onder twee testcondities:

- Test 1: de zekering van de luchtpomp stond ingeschakeld.
- Test 2: de zekering van de luchtpomp stond uitgeschakeld.

Uit test 1, uitgevoerd over een iteratie van 42 metingen, bleek dat 60,02% van de metingen resulteerde in een communicatiefout. Bij test 2, uitgevoerd over een langere reeks van 76 metingen, traden daarentegen geen enkele foutmeldingen op. Deze resultaten bevestigen dat de EMI afkomstig van de luchtpomp een significante negatieve invloed heeft op de betrouwbaarheid van de datacommunicatie met de OPC-N3-sensor.

3.7 Analyse van de communicatieproblemen bij de OPC-N3-sensor

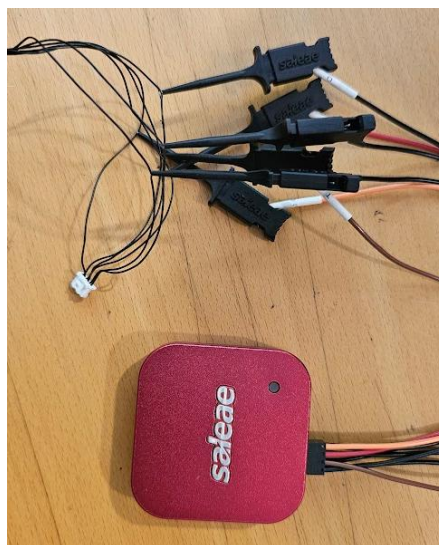
Om de werking en mogelijke foutbronnen van de OPC-N3-fijnstofsensor te analyseren, werd gebruikgemaakt van een *logic analyzer*. Dit instrument maakt het mogelijk om de digitale communicatie tussen de sensor en de Raspberry Pi in real-time te registreren, waardoor gecontroleerd kan worden of de juiste datastructuren worden uitgewisseld en of er fouten optreden in de timing of de inhoud van de signalen.

3.7.1 Opstelling en gebruik van de Saleae Logic Analyzer

In het kader van deze thesis werd gebruikgemaakt van de *Saleae Logic Analyzer 8*, een professioneel meetinstrument dat in staat is om zowel digitale als analoge signalen op te vangen. Dankzij de hoge bemonsteringssnelheid, wat in het Engels wordt aangeduid als "*sample rate*", kunnen zelfs zeer korte pulsen en signaalverstoringen nauwkeurig worden vastgelegd. Naast de bemonsteringssnelheid zijn er nog andere belangrijke

technische parameters die de geschiktheid van een *logic analyzer* bepalen. Zo beschikt het toestel over acht afzonderlijke kanalen, waardoor meerdere signaallijnen gelijktijdig kunnen worden geregistreerd. Het voltagebereik is compatibel met typische logicaniveaus zoals 3,3 V en 5 V, wat essentieel is voor het correct interpreteren van signalen afkomstig van verschillende hardwarecomponenten. Bovendien biedt de Saleae-software uitgebreide ondersteuning voor protocoldecodering, waardoor protocollen zoals I²C, SPI en UART automatisch worden geïnterpreteerd. Deze eigenschappen maken de *analyzer* uitermate geschikt voor het oplossen van foutmeldingen in complexe systemen zoals de SeaCanairy, waarin de communicatie tussen sensoren en de verwerkingsunit uiterst kritisch is.

De opstelling van de *Saleae Logic Analyzer* is eenvoudig en gebruiksvriendelijk. In eerste instantie moet de bijbehorende software worden gedownload en geïnstalleerd via de officiële website van de fabrikant. Om de gegevenslijnen te kunnen uitlezen, is het noodzakelijk dat fysieke toegang tot de afzonderlijke geleiders van de datakabel wordt verkregen. In het voorbeeld weergegeven op Figuur 10 werd dit gerealiseerd door de kabel zorgvuldig door te knippen en nadien opnieuw aan elkaar te solderen, met tussenkomst van meetpunten. Vervolgens worden de relevante signaaldraden, evenals de gemeenschappelijke massa (GND), verbonden met de corresponderende ingangen van de logic analyzer.

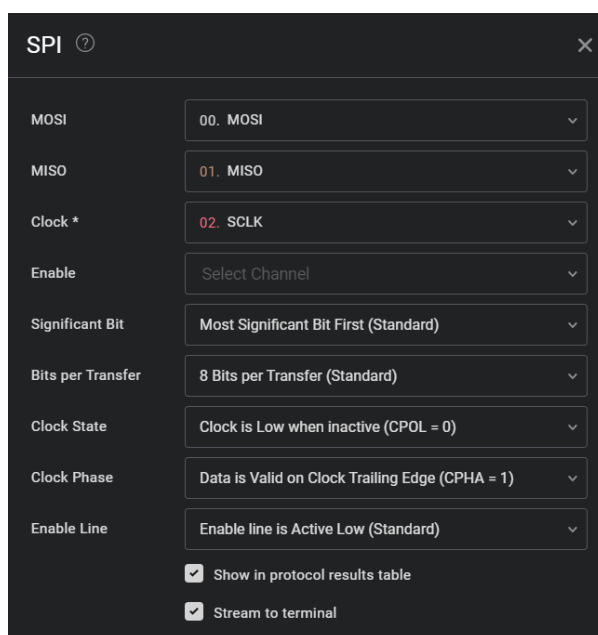


Figuur 10: De Saleae Logic Analyzer 8 verbonden met de datakabels van de OPC-N3-sensor

Bron: Eigen werk

3.7.2 Protocolconfiguratie en interpretatie van SPI-signalen

Daarnaast moet de softwareconfiguratie voor de OPC-N3-sensor correct ingesteld worden om een goede communicatie via het SPI-protocol te garanderen. In het bijzonder zijn twee parameters van cruciaal belang: de klokstatus (*Clock Polarity* of *Clock State*) en de klokfase (*Clock Phase*). Deze instellingen bepalen hoe en wanneer de gegevens van de sensor gelezen worden. Voor de OPC-N3 geldt dat de kloklijn zich in een lage toestand bevindt wanneer er geen communicatie plaatsvindt. Met andere woorden: zolang de sensor niet actief gegevens verzendt, blijft het kloksignaal op een laag niveau. Zodra de communicatie begint, worden de databits uitgelezen op het moment dat het kloksignaal van hoog naar laag gaat. Dit wordt in technische termen aangeduid als "*data capture on the falling edge of the clock*". Omdat deze instellingen per sensor kunnen verschillen, is het van belang dat ze nauwkeurig afgestemd worden op de specificaties van de OPC-N3. Een foutieve configuratie kan leiden tot foute interpretatie van de getoonde gegevens.



Parameter	Value
MOSI	00. MOSI
MISO	01. MISO
Clock *	02. SCLK
Enable	Select Channel
Significant Bit	Most Significant Bit First (Standard)
Bits per Transfer	8 Bits per Transfer (Standard)
Clock State	Clock is Low when inactive (CPOL = 0)
Clock Phase	Data is Valid on Clock Trailing Edge (CPHA = 1)
Enable Line	Enable line is Active Low (Standard)

☒ Show in protocol results table

☒ Stream to terminal

Figuur 11: Configuratie voor de OPC-N3 SPI-protocol in de Logic software van Saleae

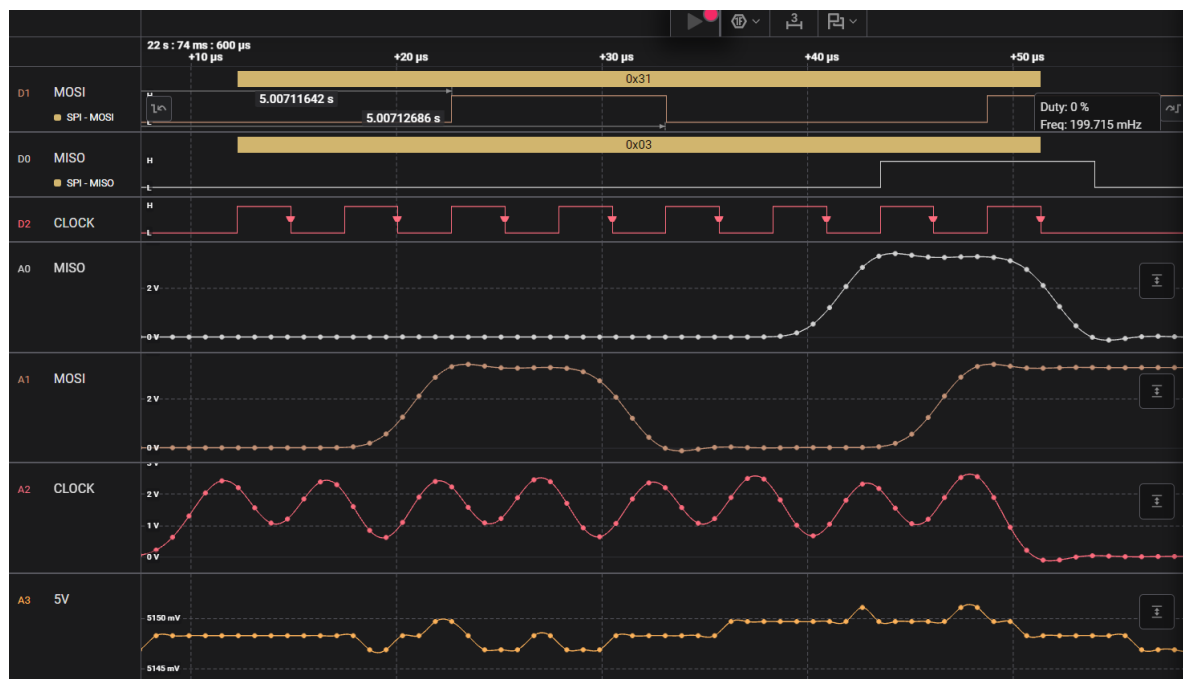
Bron: Eigen werk

In Figuur 12 is te zien dat elke keer wanneer het kloksignaal een overgang maakt van hoog naar laag, dit in de logic analyzer-software aangeduid wordt met een neerwaarts gericht

pijltje. Dit moment markeert het tijdstip waarop de gegevens op de signaallijnen MOSI (*Master Out Slave In*) en MISO (*Master In Slave Out*) geldig zijn en kunnen worden uitgelezen. In dit voorbeeld worden er acht opeenvolgende bits overgedragen, wat overeenkomt met één byte. De MOSI-lijn toont hierbij de binaire reeks 0011 0001, wat vertaald kan worden naar de hexadecimale waarde 0x31. Het omzetten van een binair getal naar een hexadecimale waarde gebeurt door de binaire reeks op te splitsen in groepjes van vier bits, beginnend van rechts. Elke groep van vier bits wordt vervolgens vertaald naar zijn overeenkomstige hexadecimale cijfer. In dit geval:

- 0011 komt overeen met het hexadecimale cijfer 3.
- 0001 komt overeen met het hexadecimale cijfer 1.

Samen vormen deze dus 0x31, waarbij de prefix 0x aangeeft dat het om een hexadecimale waarde gaat.



Figuur 12: Visualisatie in de Logic Analyzer-software van de SPI-communicatie tussen de Raspberry Pi en de OPC-N3 sensor. De signaallijnen MOSI (Master Out Slave In), MISO (Master In Slave Out) en CLOCK worden grafisch weergegeven. Uit de CLOCK-blokvorm is duidelijk af te leiden dat de databit wordt uitgelezen op het moment dat het kloksignaal een overgang maakt van hoog naar laag.

Bron: Eigen werk

3.7.3 Analyse van de communicatiecyclus met de OPC-N3-sensor

Allereerst wordt telkens een verhoogde spanning waargenomen op de analoge MOSI-lijn. Dit probleem treedt systematisch op wanneer de pomp geactiveerd wordt. De gemeten spanningsverhoging varieert van enkele millivolts tot pieken tot wel 2 volt. Opvallend is dat dit fenomeen enkel optreedt op de MOSI-lijn, en niet op andere signaallijnen zoals de kloklijn. Hierdoor is het aannemelijk dat de sensor deze verhoogde logische waarde niet interpreteert als geldig signaal. De software die wordt gebruikt om de communicatielijnen te analyseren houdt echter wel rekening met deze spanningspieken. Wanneer de spanning boven de logische drempel uitkomt, wordt de lijn foutief als ‘hoog’ geïnterpreteerd. Dit is duidelijk zichtbaar in Figuur 13, waar de spanningspiek ten onrechte leidt tot een logische ‘1’ op de MOSI-lijn, ondanks het ontbreken van een geldig datasignaal.



Figuur 13: Spanningspiek op de MOSI-lijn door vermoedelijk elektromagnetische interferentie

Bron: Eigen werk

Aangezien de OPC-N3-sensor regelmatig foutmeldingen genereerde tijdens iteraties van luchtmetingen, en deze storingen zichzelf niet automatisch corrigeerden, is het zinvol om een correcte iteratie te vergelijken met een foutieve. Op die manier kan worden nagegaan of de communicatiefout zich steeds op een identiek moment in het communicatieproces voordoet, of eerder willekeurig van aard is. Door de verschillen

tussen een succesvolle en een mislukte meetcyclus nauwkeurig te analyseren, kunnen potentiële oorzaken van de instabiliteit worden geïdentificeerd, zoals, signaalvervormingen of verstoringen op de SPI-bus door elektromagnetische interferentie.

Er kunnen verschillende commando's worden verstuurd en deze zijn opgenomen in de volgende tabel:

Tabel 8: SPI commando's voor de OPC-N3-sensor

Bron: Supplemental SPI information for the OPC-N3 door Alphasense [4]

SPI-Commando	Lijn	Functie
0x03 + 0x02	MOSI	Eerste byte 0x03: Power control, tweede byte 0x02: ventilator UIT
0x03 + 0x03	MOSI	Eerste byte 0x03: Power control, tweede byte 0x03: ventilator AAN
0x03 + 0x06	MOSI	Eerste byte 0x03: Power control, tweede byte 0x06: Laser AAN
0x03 + 0x07	MOSI	Eerste byte 0x03: Power control, tweede byte 0x07: Laser UIT
0x30	MOSI	Commando om histogram data op te vragen
0xF3	MISO	Sensor is klaar voor SPI-communicatie
0x31	MISO	Sensor is nog bezig, communicatie uitstellen
0x3F	MOSI	Opvragen sensorinformatie

De onderstaande tabel toont links de normale werking van de OPC-N3-sensor zoals geregistreerd met behulp van de Saleae Logic Analyzer. Zoals blijkt uit de communicatiereeks, stuurt de sensor bij elk commando dat betrekking heeft op de interne componenten (zoals de ventilator of de laser) eerst de statusmelding 'Bezig'

terug. Dit gedrag is conform de specificaties zoals beschreven in de technische documentatie van de OPC-N3-sensor, uitgegeven door Alphasense. Daarnaast werd waargenomen dat de sensor in sommige gevallen het histogram tweemaal terugstuurt. Dit fenomeen treedt op wanneer de checksumcontrole een fout detecteert bij het eerste datablok. In dat geval initieert de software automatisch een herhaalde aanvraag van het histogram, zodat de gegevens alsnog correct worden ontvangen.

Aan de rechterzijde wordt beschreven hoe de OPC-N3-sensor zich gedraagt bij een miscommunicatie tussen de Raspberry Pi en de sensor. Zoals gebruikelijk stuurt de software het hexadecimale commando 0x03 uit om de status van de sensor op te vragen. In een normale situatie zou de sensor hierop antwoorden met 0x31 (bezig) of 0xF3 (klaar voor communicatie). In dit geval ontvangt de Raspberry Pi echter het onjuiste antwoord 0x30, wat geen geldige statuswaarde is volgens de protocoldefinitie van de OPC-N3. Deze afwijkende respons veroorzaakt verwarring in het verdere verloop van de communicatie, waardoor het uitlezen van de luchtkwaliteitsdata tijdens die iteratie mislukt. De fout leidt ertoe dat de software de meting als ongeldig beschouwt en geen bruikbare data registreert voor deze cyclus. Deze fout doet zich op consequente wijze voor.

Hoewel de datasheet momenteel niet publiek beschikbaar is via de officiële website van de fabrikant, kan deze nog steeds op verzoek verkregen worden via Alphasense, of eventueel via archiefbronnen online.

Tabel 9: Analyse van succesvolle en mislukte datatransmissie bij SPI-communicatie met de OPC-N3-sensor

Succesvolle datatransmissie				Mislukte datatransmissie		
	MOSI	MISO	Beschrijving	MOSI	MISO	Beschrijving
1	0x03	0x31	Bezig	0x03	0x30 0x(xx)	Fout antwoord van de sensor
2	0x03	0xF3	Klaar voor communicatie	0x03	0x31	Bezig
3	0x03	0x03	Ventilator ingeschakeld	0x03	0x03	De sensor geeft aan dat het een commando heeft uitgevoerd wat in deze stap overeen zou komen met het aanzetten van de ventilator. De sensor heeft echter nooit gecommuniceerd dat hij klaar voor communicatie was
4	0x03	0x31	Bezig	-	-	
5	0x03	0xF3	Klaar voor communicatie	-	-	
6	0x07	0x03	Laser ingeschakeld	-	-	
7	0x30	0x31	Bezig	-	-	
8	0x30	0xF3	Start histogramoverdracht	-	-	
9	0x00	86 bytes	Histogramgegevens	-	-	
10	0x03	0x31	Bezig	-	-	
11	0x03	0xF3	Klaar voor communicatie	-	-	
12	0x06	0x03	Laser uitgeschakeld	-	-	
13	0x03	0x31	Bezig	-	-	
14	0x03	0xF3	Klaar voor communicatie	-	-	
15	0x02	0x03	Ventilator uitgeschakeld	-	-	

3.7.4 Oplossingen voor communicatieproblemen: hardware en software

3.7.4.1 Hardwarematige oplossingen voor communicatieproblemen

Om de invloed van elektromagnetische interferentie (EMI), afkomstig van de luchtpomp, op de werking van de OPC-N3-sensor te beperken, werden drie hardwarematige maatregelen toegepast. Ten eerste werd een keramische ontkoppelcondensator van 100 μF geplaatst nabij de spanningsinvoer van de sensor, met als doel hogefrequente ruis lokaal te onderdrukken en spanningsfluctuaties te stabiliseren. Daarnaast werd er een ferrietkern aangebracht rond de voedingslijn van de luchtpomp, ter demping van hoogfrequente stoorsignalen die via de bekabeling konden worden geïnjecteerd. Tot slot werden de positieve en negatieve voedingsdraden van de luchtpomp in elkaar gedraaid (getwist) zodat de emissie van elektromagnetische straling in de omgeving grotendeels wordt opgeheven door de tegenfase van beide geleiders. Deze combinatie van passieve maatregelen resulteerde in een lichtmerkbare verbetering van de signaalkwaliteit en een afname van communicatieverstoringen bij het gebruik van de OPC-N3-sensor.

3.7.4.2 Softwarematige oplossingen voor communicatieproblemen

Ondanks deze maatregelen, werd de spanningspiek nooit volledig geëlimineerd. Daarom werd aanvullend een softwarematige oplossing geïmplementeerd: een wachttijd van vijf seconden na het inschakelen van de pomp vóór het eerste SPI-commando naar de sensor wordt verzonden. Hierdoor krijgt de sensor voldoende tijd om eventuele foutieve signalen in zijn buffer te negeren of te wissen, nog voor de Raspberry Pi een geldige communicatie opstart. Zelfs als de sensor de spanningspiek zou registreren zoals in Figuur 13: Spanningspiek op de MOSI-lijn door vermoedelijk elektromagnetische interferentie, is de kans groot dat deze tegen dan al geneutraliseerd is.

Naast deze spanningsgerelateerde maatregelen werd ook aandacht besteed aan structurele foutmeldingen van de OPC-N3. Doordat de foutmeldingen van de deze sensor zich op een vrij consistent patroon voordoen, bleek het oplossen van deze problemen via software eenvoudiger dan aanvankelijk gedacht. Zo werd vastgesteld dat de sensor regelmatig foutcodes zoals 0x30, 0x03 en sporadisch ook 0x62 terugstuurt

tijdens de initialisatie. Hoewel de foutmelding 0x62 nooit werd opgevangen met de *logic analyzer*, werd deze wel herkend in de software en meegenomen in de foutafhandeling.

In de functie `def initiate_transmission(command_byte)` werden verscheidene strategieën getest om deze foutmeldingen op te vangen. Een eerste benadering was het inlassen van een vertraging van vijf seconden in de hoop dat de interne buffer van de sensor zichzelf zou legen. Deze aanpak bleek echter slechts zelden effectief.

Een veel betrouwbaardere oplossing werd gevonden door een commando naar de sensor te sturen na het ontvangen van een foutcode. Aanvankelijk werd geprobeerd het standaard commando 0x03 te gebruiken, maar dit had geen merkbaar effect; mogelijk omdat dit commando slechts deel uitmaakt van een tweedelige sequentie die bedoeld is om de laser of ventilator aan of uit te zetten.

Uit experimenteel testen bleek echter dat het commando 0x3F, bedoeld om de infostring van de sensor uit te lezen (zoals firmwareversie en serienummer), veilig kan worden gebruikt als "*flush*"-mechanisme. Dit commando veroorzaakt geen wijziging in de instellingen of werking van de sensor, maar dwingt de sensor wel om 60 opeenvolgende bytes terug te sturen. Hierdoor wordt de SPI-buffer effectief leeggemaakt en raakt de sensor opnieuw gesynchroniseerd met het besturingssysteem. Het gebruik van dit commando na het detecteren van foutcodes bleek in de praktijk betrouwbaar.

Onderstaande Python-code toont hoe deze oplossing is geïmplementeerd is door onderaan in de "else" te zorgen dat alle afwijkende antwoorden van de sensor worden opgevangen. Uit ervaring stuurt de sensor heel vaak de tweede keer de heximale waarde 0x3 terug. Dit is dan opgevangen in de "elif" code als de sensor voor de tweede maal de functie doorloopt.

```
elif reading == [3]: #returned 0x3. this is always the second
wrong byte returned, see else statement below in the function
```

```
    print("Returned 0x3, wait for buffer and send command 0xF3
to force a reaction")
```

```
    time.sleep(1)
```

```
    spi.xfer([0xF3])
```


Hoofdstuk 4 Resultaten van het herontwerp en technische optimalisaties

Om de effectiviteit van het nieuwe systeem objectief te beoordelen, wordt nagegaan in hoeverre het systeem geschikt is voor langdurige, autonome meetcampagnes. De evaluatie richt zich op drie cruciale aspecten die rechtstreeks verband houden met de structurele kwaliteit, technische betrouwbaarheid en stabiliteit van het systeem:

1. Bekende tekortkomingen en doorgevoerde verbeteracties.

Hierbij wordt onderzocht welke gebreken of beperkingen van het oorspronkelijke ontwerp vooraf geïdentificeerd werden, en hoe deze structureel werden aangepakt. De focus ligt onder andere op de luchtcirculatie, positionering van sensoren en de praktische opbouw van het systeem.

2. Nieuw vastgestelde problemen tijdens testcampagnes.

Tijdens praktijktests kwamen nieuwe knelpunten aan het licht, zoals verstoringen door interne verwarming of onverwachte sensorreacties. Deze sectie bespreekt de invloed van dergelijke factoren op de meetkwaliteit en operationele werking van het systeem.

3. Het resultaat van de kastverwarming om hygroscopische groei tegen te gaan

Hierbij wordt nagegaan in welke mate de kastverwarming de relatieve luchtvochtigheid effectief verlaagt en daarmee de hygroscopische groei van deeltjes beperkt. De nadruk ligt op de stabiliteit van de metingen vóór en na verwarming, de invloed op de nauwkeurigheid van de fijnstofsensoren en de praktische toepasbaarheid in uiteenlopende omgevingen (binnen en buiten). Ook wordt besproken of er neveneffecten optreden, zoals een fluctuerende waarde voor luchtvochtigheid.

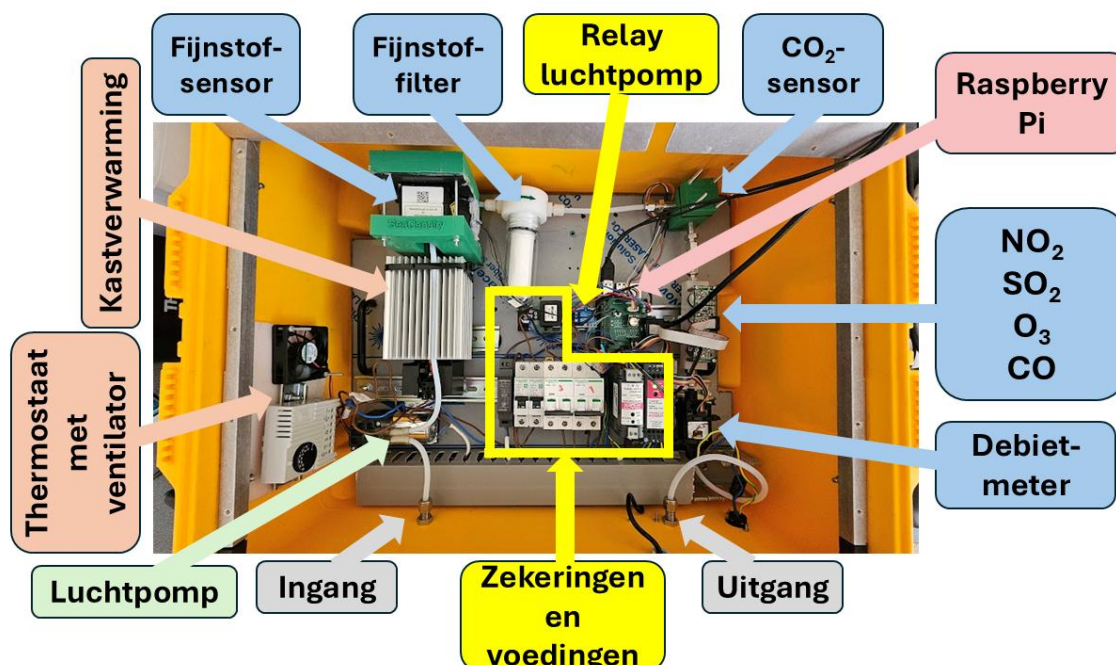
4.1 Verbeterde luchtcirculatie binnen het meetsysteem

Om de luchtcirculatie te optimaliseren, werd het systeem herwerkt naar een eenvoudigere configuratie waarbij de volledige opstelling zich op één oppervlak in de koffer bevindt. Voor de sensoren zijn op maat gemaakte 3D-geprinte behuizingen in PLA ontwikkeld: één voor de OPC-N3-fijnstofsensor en één voor de E+E Elektronik EE894 CO₂-sensor. De ontwerpen van de OPC-N3-behuizing zijn weergegeven in Bijlage V 3D-ontwerp voor de ingang van de OPC-N3-sensor in OpenSCAD en Bijlage VI 3D-ontwerp voor de uitgang van de OPC-N3-sensor in OpenSCAD, en deze voor de CO₂-sensor in Bijlage VII

3D-ontwerp voor het bovenste stuk van de CO₂ sensor in OpenSCAD, Bijlage VIII

3D-ontwerp voor het middenste stuk van de CO₂ sensor in OpenSCAD en Bijlage IX

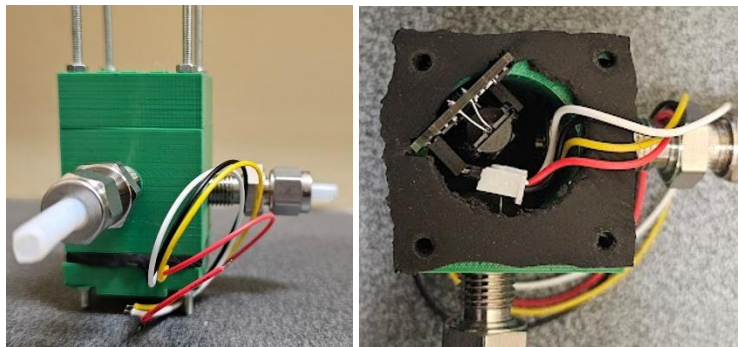
3D-ontwerp voor het onderste stuk van de CO₂ sensor in OpenSCAD. De onderstaande figuren illustreren deze ontwerpen. Figuur 15 toont het eindresultaat van de opstelling.



Figuur 15: Het nieuwe ontwerp van de SeaCanairy

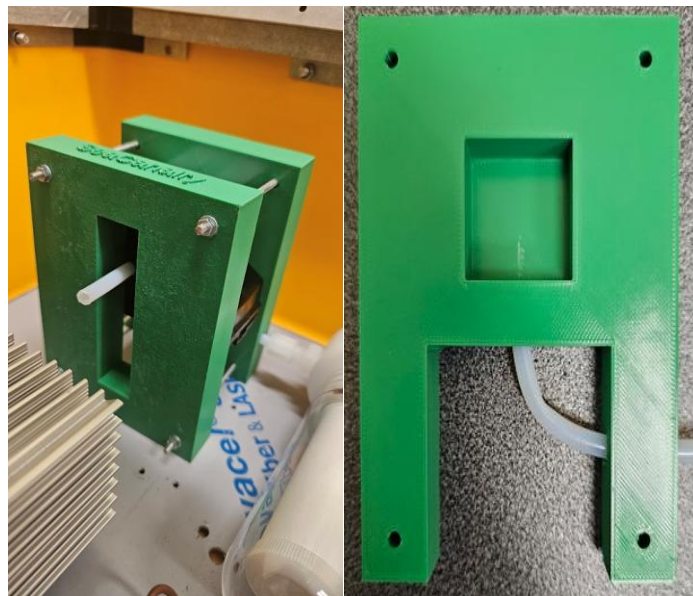
Bron: Eigen werk

De 3D-prints bleken een geschikte methode om de sensoren te integreren met de PTFE-luchtleidingen. De behuizing van de CO₂-sensor functioneerde naar behoren, maar bij de OPC-N3 bleek de sensorkamer niet volledig luchtdicht te zijn. Dit probleem werd tijdelijk opgelost door de behuizing af te plakken. In een latere fase zal het ontwerp van deze behuizing moeten worden herzien, mede omdat de plaatsing van de pomp aan het begin van het luchtcircuit duidelijk maakte dat er een luchtlek was. De onderstaande figuren tonen de op maat gemaakte ontwerpen.



Figuur 16: Het ontwerp voor de CO₂-sensor

Bron: Eigen werk



Figuur 17: Ontwerp voor de OPC-N3-fijnstofsensor

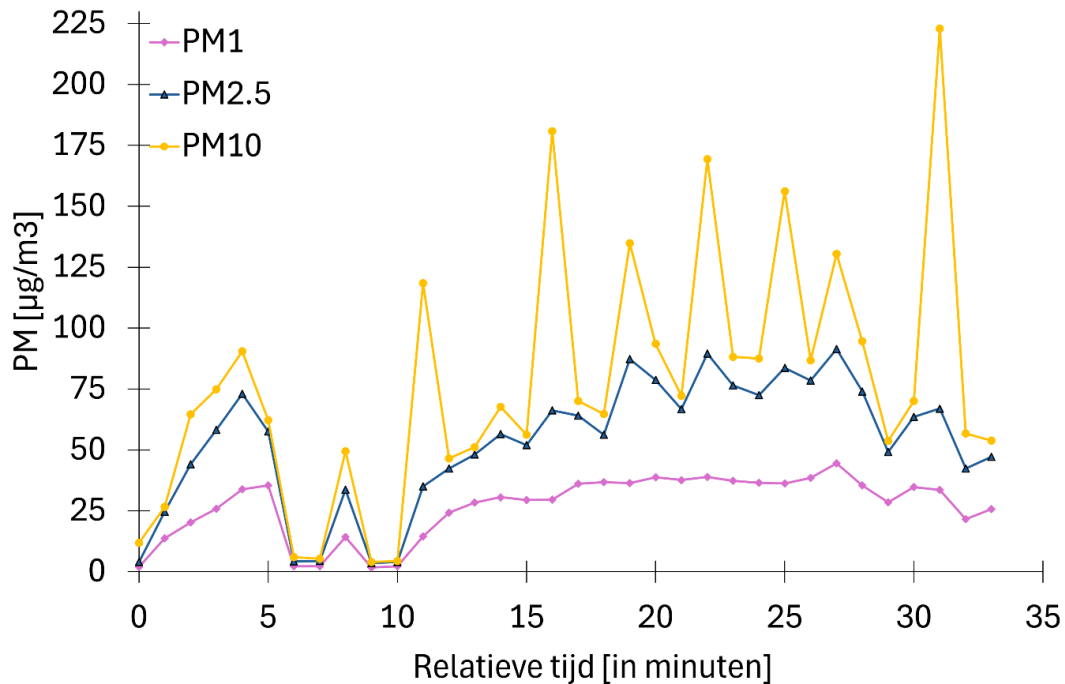
Bron: Eigen werk

4.2 Invloed van pompconfiguratie op gemeten fijnstofwaarden

Tijdens testmetingen werd vastgesteld dat de gemeten fijnstofconcentraties (PM₁, PM_{2.5} en PM₁₀) bleven stijgen tot onrealistische niveaus. Dit vormde aanleiding om de opstelling kritisch te evalueren en de werking van de OPC-N3-sensor nader te onderzoeken. Uit analyse bleek dat de oorspronkelijke configuratie, waarbij de pomp zich achter de sensor bevond, ongewenste stromingspatronen veroorzaakte met als gevolg de ophoping van deeltjes in de meetkamer. In de onderstaande tekst wordt beschreven hoe het verplaatsen van de pomp naar de aanzuigzijde, waardoor alle sensoren zich aan de perszijde bevinden, leidde tot realistischere en stabielere meetresultaten.

4.2.1 Foutieve opstelling met de pomp achter de OPC-N3-sensor

Uit testmetingen ($n = 34$) werden statistische parameters bepaald voor de concentraties van PM₁, PM_{2.5} en PM₁₀. De gemiddelde waarden bedroegen respectievelijk 26,70 $\mu\text{g}/\text{m}^3$ (PM₁), 52,93 $\mu\text{g}/\text{m}^3$ (PM_{2.5}) en 77,23 $\mu\text{g}/\text{m}^3$ (PM₁₀). Deze waarden liggen aanzienlijk boven de door de Wereldgezondheidsorganisatie (WHO) aanbevolen jaarlijkse grenswaarde van 5 $\mu\text{g}/\text{m}^3$ voor PM_{2.5}. [15] De gemeten maximumwaarden bedroegen zelfs 91,39 $\mu\text{g}/\text{m}^3$ voor PM_{2.5} en 222,92 $\mu\text{g}/\text{m}^3$ voor PM₁₀, wat wijst op zeer hoge pieken in de concentratie. De standaardafwijking voor PM_{2.5} bedroeg 26,25 $\mu\text{g}/\text{m}^3$, wat duidt op een aanzienlijke spreiding rond het gemiddelde. De mediaanwaarde van 57,06 $\mu\text{g}/\text{m}^3$ voor PM_{2.5} toont aan dat de helft van de meetwaarden hoger lag dan de WHO-jaargrens, wat structureel verhoogde concentraties suggereert. Gezien deze waarden en spreiding lijkt het waarschijnlijk dat er sprake is van verstoorde stroming of interne ophoping van deeltjes.



Figuur 18: Testmeting met geleidelijke toename van PM-waarden als gevolg van fijnstofophoping in de behuizing van de OPC-N3-sensor, veroorzaakt door de positionering van de pomp achter de sensor.

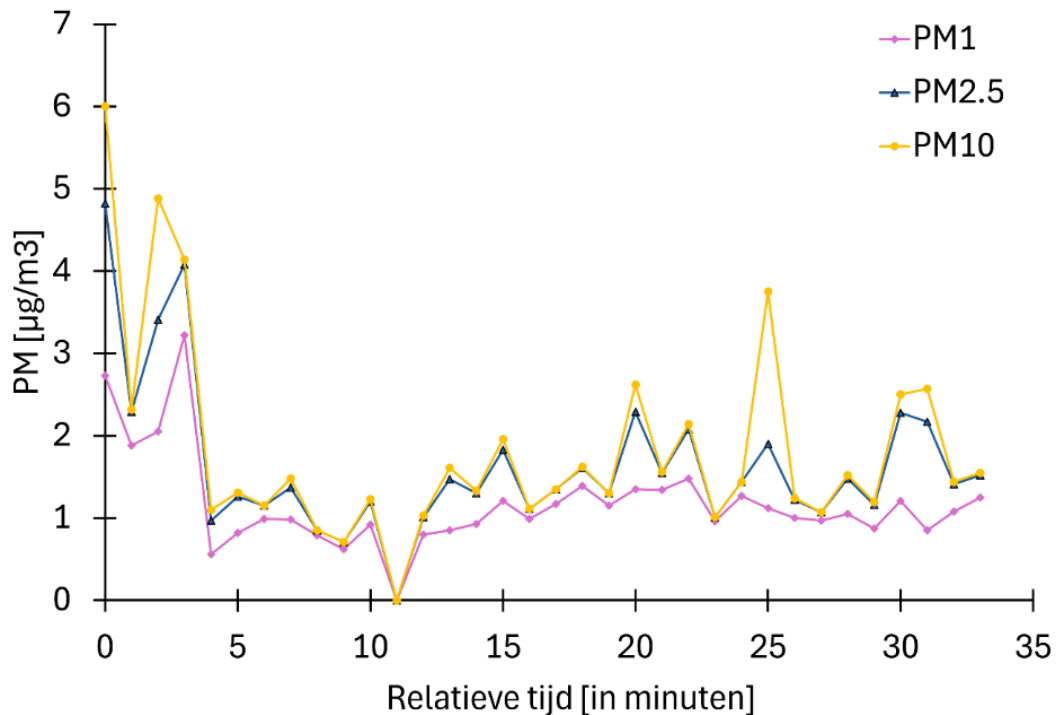
Bron: Eigen werk

4.2.2 Correcte opstelling met de pomp aan het begin van het luchtcircuit

Om dit probleem op te lossen werd de pomp aan het begin van het luchtcircuit geplaatst. Hierdoor bevinden alle sensoren zich aan de perszijde van de pomp, wat voorkomt dat er een vacuüm wordt getrokken in het systeem. Deze configuratie werd aanvankelijk vermeden om mogelijke verstoringen op de PM-sensor te vermijden. Uit recente testmetingen ($n = 34$) bleek echter dat de gemeten fijnstofconcentraties stabiel en realistisch blijven binnen deze opstelling.

De gemiddelde concentraties bedroegen $1,21 \mu\text{g}/\text{m}^3$ voor PM_1 , $1,69 \mu\text{g}/\text{m}^3$ voor $\text{PM}_{2.5}$ en $1,88 \mu\text{g}/\text{m}^3$ voor PM_{10} , met maximale waarden van respectievelijk 3,22, 4,82 en $6,00 \mu\text{g}/\text{m}^3$. Deze waarden blijven ruim onder de door de Wereldgezondheidsorganisatie (WHO) gehanteerde grenswaarden voor langdurige blootstelling. De relatief lage standaardafwijkingen (bijvoorbeeld $0,90 \mu\text{g}/\text{m}^3$ voor $\text{PM}_{2.5}$) wijzen bovendien op een consistente luchtkwaliteit tijdens de metingen. De mediaanwaarden van 1,05 (PM_1), 1,41 ($\text{PM}_{2.5}$) en 1,44 (PM_{10}) bevestigen dit beeld.

De licht verhoogde waarden die tijdens de eerste drie metingen werden waargenomen, kunnen waarschijnlijk worden toegeschreven aan fijnstofdeeltjes die tijdens de veranderingen van het systeem in de luchtbuis terechtkwamen. Nadien stabiliseerden de meetwaarden zich rond een normaal achtergrondniveau.



Figuur 19: Testmetingen met normale PM-waarden waarbij de pomp voor de sensor is geplaatst waardoor er zich geen fijnstofophoping voordoet in de behuizing van de sensor.

Bron: Eigen werk

4.2.3 Luchtlekkage ter hoogte van de OPC-N3-sensor

Tijdens de metingen in de aangepaste correcte configuratie werd vastgesteld dat de meetkamer van de OPC-N3-sensor niet volledig luchtdicht is. Er was voelbare luchtstroming waarneembaar uit de sensorbehuizing, wat erop wijst dat de interne meetkamer onder lichte overdruk komt te staan wanneer de pomp aan de aanzuigzijde van het luchtcircuit is geplaatst. Hoewel dit geen merkbare invloed leek te hebben op de werking van de OPC-N3-sensor zelf, vormt het een potentieel lekpunt binnen het totale luchtsysteem. Om dit te verhelpen zijn de openingen en kieren in de sensorbehuizing afgedicht met smeltlijm, waardoor het luchtcircuit weer hermetisch gesloten werd. Dit is echter maar een tijdelijke oplossing en het geprinte ontwerp zal herzien moeten worden voor verder langdurig gebruik van de SeaCanairy.

4.3 Effect van luchtverwarming op de prestaties van de OPC-N3-sensor

Wanneer fijnstofdeeltjes in de lucht gemeten worden, kunnen deze deels bestaan uit of omgeven zijn door waterdamp. Dit fenomeen staat bekend als hygroscopische groei, waarbij deeltjes water aantrekken en daardoor in volume en massa toenemen. Dit kan leiden tot een overschatting van de werkelijke fijnstofconcentratie. Om dit effect te minimaliseren, kan de lucht worden opgewarmd voordat deze in de fijnstoffilter komt, zodat het gebonden vocht verdampt en de meting zich beperkt tot het droge deeltje wat een accurater resultaat oplevert met betrekking tot fijnstofmetingen. In deze proeven werd onderzocht wat het effect is van verwarming op de gemeten fijnstofconcentraties.

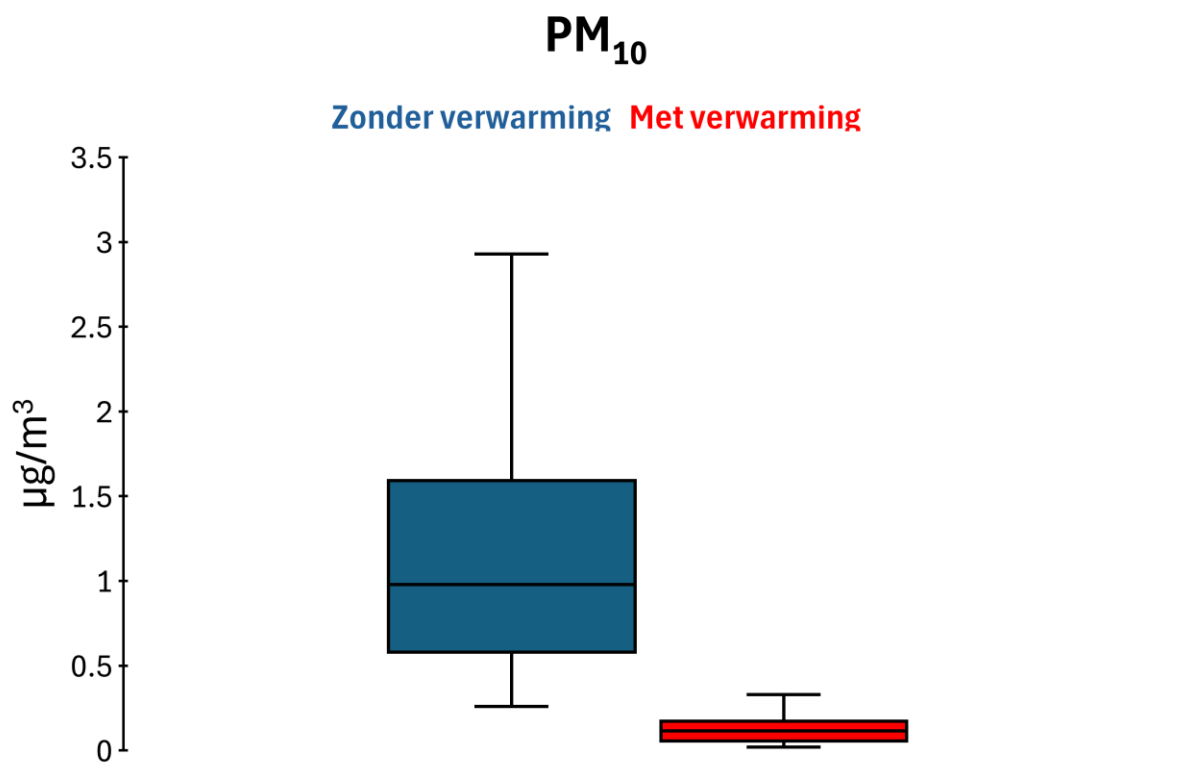
In dit hoofdstuk zijn twee proeven beschreven, met name, luchtverwarming tegen hygroscopische groei in zowel binnen- als buitenlucht, om te onderzoeken of beide omgevingen hetzelfde resultaat opleveren. Voor elke test werd eerst gemeten zonder voorverwarming, waarna de kastverwarming werd ingeschakeld om de invloed van verwarmde lucht te bestuderen. Het doel van de verwarming is om de relatieve luchtvochtigheid zo laag mogelijk te brengen, zodat aanwezige waterdeeltjes verdampen en de fijnstofmetingen minder verstoord worden. Omdat een stabiel niveau van relatieve luchtvochtigheid essentieel is voor betrouwbare conclusies, moet na het inschakelen van de kastverwarming eerst een wachttijd in acht worden genomen voordat de metingen in rekening gebracht kunnen worden.

4.3.1 Invloed van luchtverwarming op fijnstofconcentraties in binnenlucht

De eerste test is binnen uitgevoerd met telkens 2 reeksen van 30 fijnstofmetingen waarbij eerst metingen werden uitgevoerd zonder het verwarmingselement en vervolgens de kastverwarming werd aangezet. De ingestelde temperatuur bedroeg 50 °C. De relatieve luchtvochtigheid is met het aanzetten van de kastverwarming gezakt van gemiddeld

46,79% tot 30,39%. Deze daling is een positieve indicatie dat luchtvoorverwarming een positieve invloed heeft om hygroscopische groei tegen te gaan.

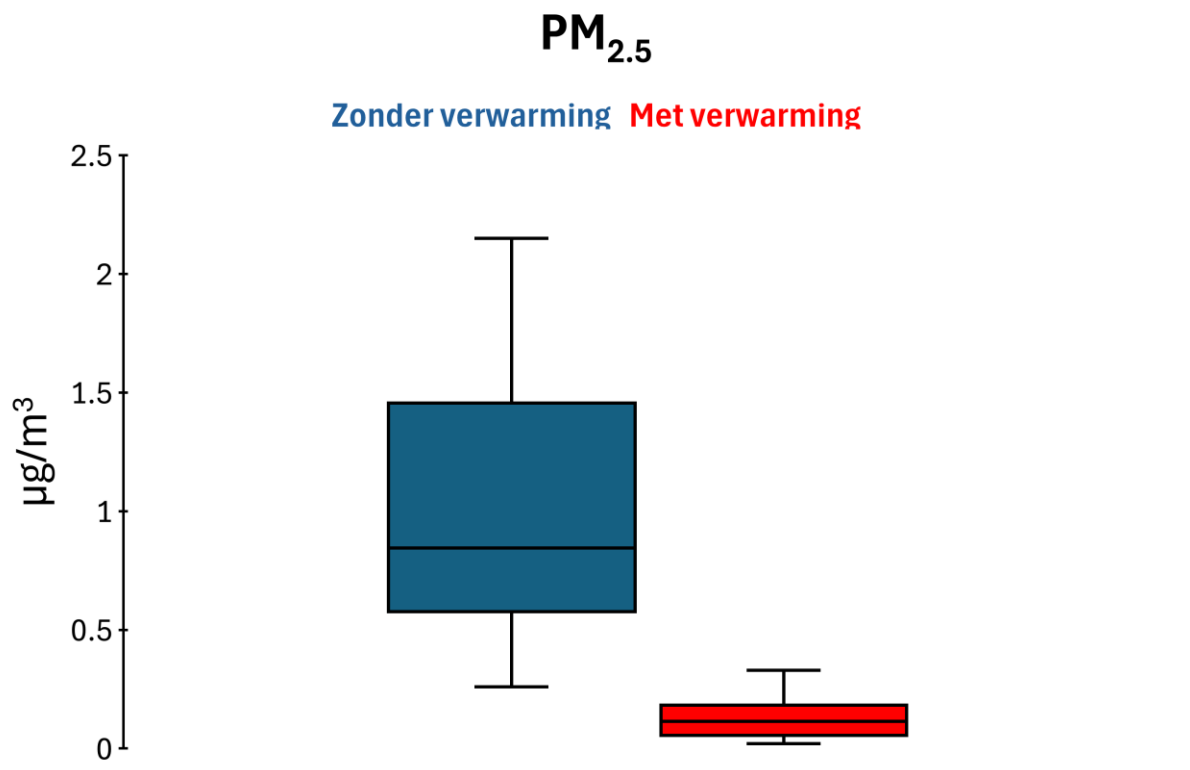
De eerste meetreeks behandelt de concentratie van PM₁₀. Na het inschakelen van de verwarming daalde de mediaanwaarde van 0,98 $\mu\text{g}/\text{m}^3$ naar 0,12 $\mu\text{g}/\text{m}^3$, terwijl ook het gemiddelde terugviel van 1,37 naar 0,26 $\mu\text{g}/\text{m}^3$. Daarnaast verminderde de standaardafwijking van 1,42 naar 0,58 $\mu\text{g}/\text{m}^3$, wat wijst op een lagere variabiliteit binnen de data. De daling in zowel centrale tendens als spreiding geeft aan dat PM₁₀ beïnvloed wordt door hygroscopische groei en dus duidelijk afhankelijk is van de luchtvochtigheid.



Figuur 20: Vergelijking PM₁₀-waarden zonder en met voorverwarming in een binnenomgeving

Bron: Eigen werk

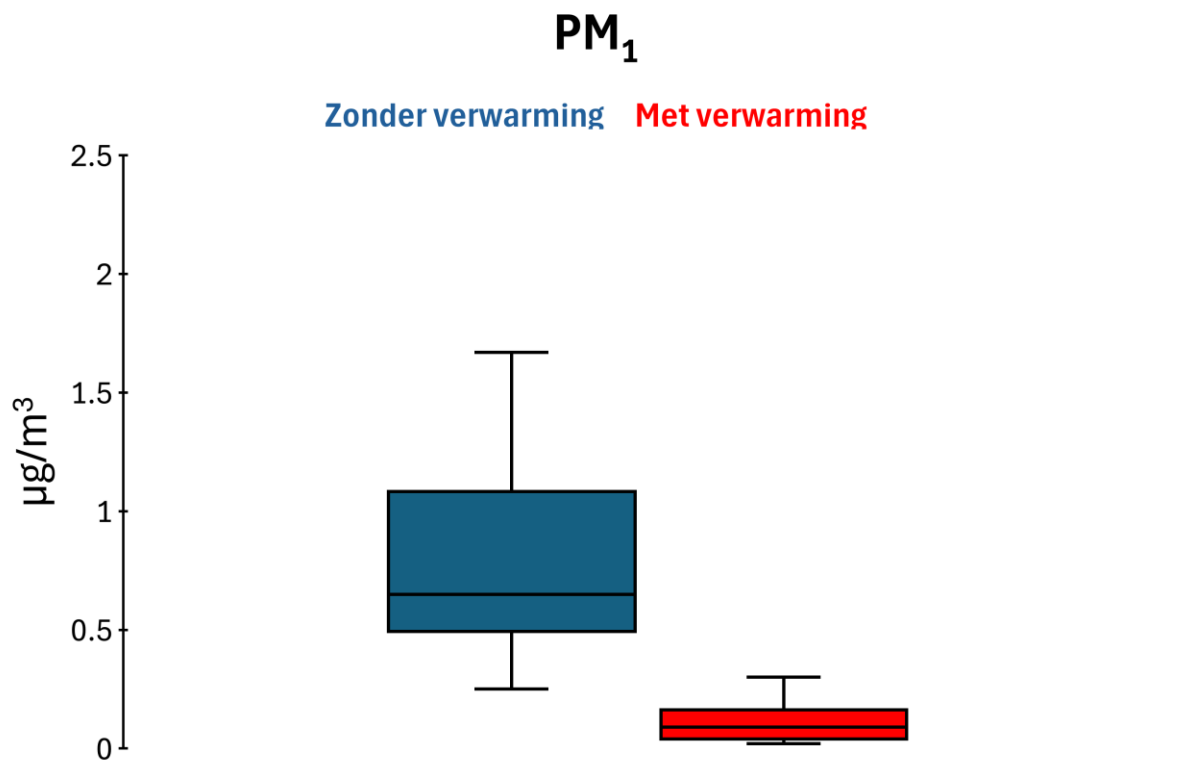
Voor $PM_{2.5}$ werd een soortgelijke trend vastgesteld. De mediaan daalde van $0,85 \mu\text{g}/\text{m}^3$ naar $0,12 \mu\text{g}/\text{m}^3$, en het gemiddelde van $1,03$ naar $0,24 \mu\text{g}/\text{m}^3$. De standaardafwijking nam af van $0,57$ naar $0,47 \mu\text{g}/\text{m}^3$, wat duidt op een homogenisering van de meetwaarden. $PM_{2.5}$ blijkt dus eveneens gevoelig voor vochtgebonden deeltjesgroei.



Figuur 21: Vergelijking $PM_{2.5}$ -waarden zonder en met voorverwarming in een binnenomgeving

Bron: Eigen werk

Ook bij PM_1 is de invloed van verwarming merkbaar, zij het subtieler. De mediaan daalde van $0,65 \mu\text{g}/\text{m}^3$ naar $0,09 \mu\text{g}/\text{m}^3$, en het gemiddelde van $0,77$ naar $0,16 \mu\text{g}/\text{m}^3$. De standaardafwijking verminderde van $0,38$ naar $0,19 \mu\text{g}/\text{m}^3$. Wat wil zeggen dat ook bij de kleinste fijnstofdeeltjes het verwarmen van lucht een invloed heeft op de metingen in binnenlucht.



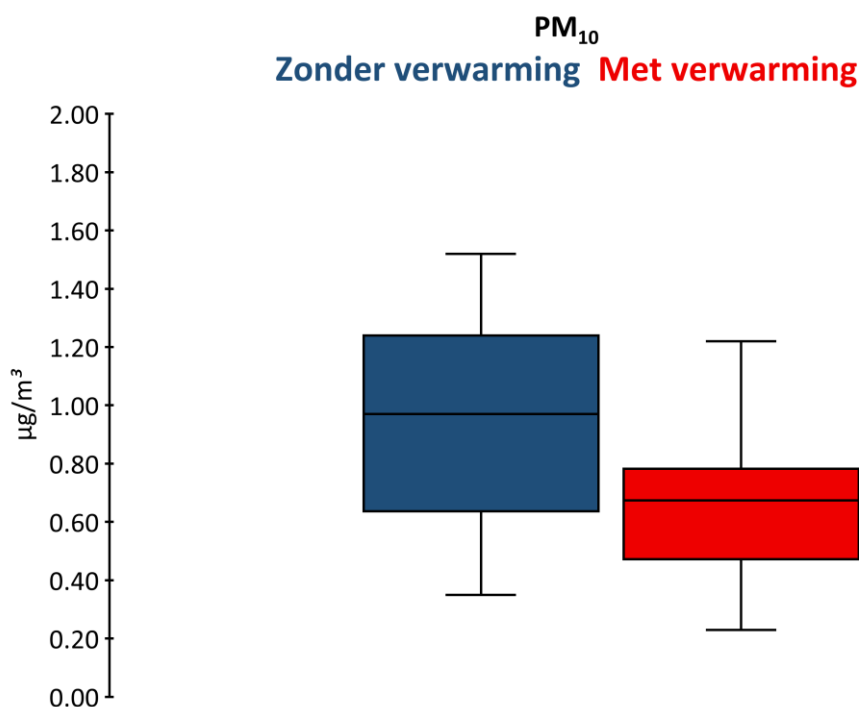
Figuur 22: Vergelijking PM_1 -waarden zonder en met voorverwarming in een binnenomgeving

Bron: Eigen werk

4.3.2 Invloed van luchtverwarming op fijnstofconcentraties in buitenlucht

Een tweede soortgelijke test werd buiten herhaalt op het dakterras van de Antwerp Maritime Academy. Tijdens twee opeenvolgende reeksen van telkens 36 metingen werden eerst fijnstofwaarden verzameld zonder actieve verwarming, gevolgd door een tweede meetreeks de verwarming ingeschakeld. De relatieve luchtvochtigheid daalde daarbij van gemiddeld 34,25% naar 22,51% met een ingestelde thermostaat op 50 °C, wat opnieuw bevestigt dat de verwarming effectief verdamping van waterdamp induceerde.

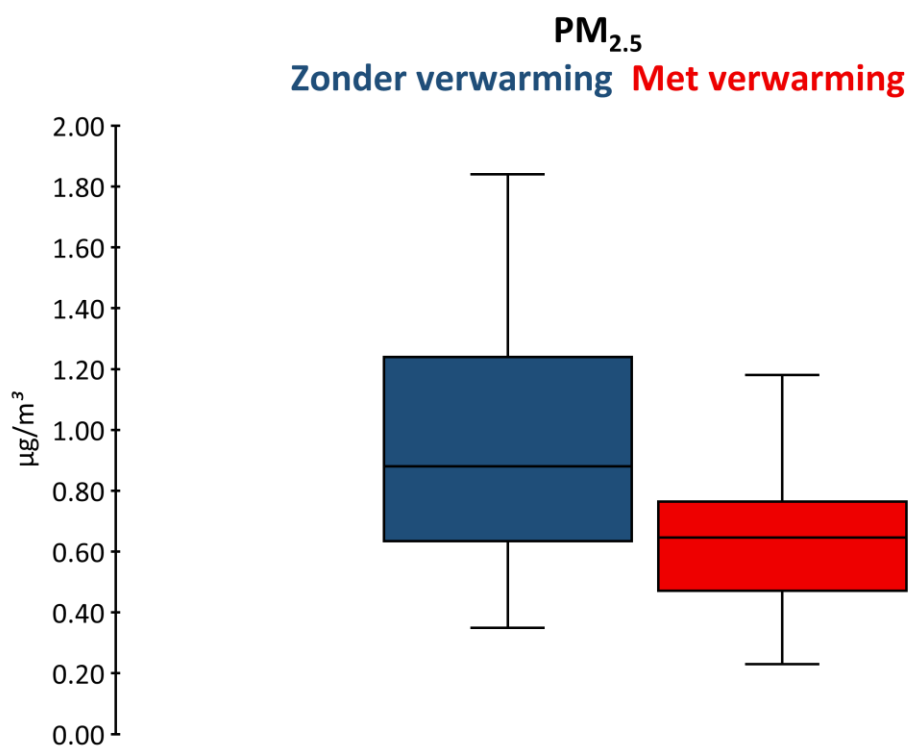
Na het inschakelen van de verwarming daalde de mediaanwaarde van PM_{10} van $0,97 \mu\text{g}/\text{m}^3$ naar $0,67 \mu\text{g}/\text{m}^3$, terwijl ook het gemiddelde afnam van $0,94$ naar $0,67 \mu\text{g}/\text{m}^3$. De standaardafwijking verminderde van $0,35$ naar $0,28 \mu\text{g}/\text{m}^3$, wat wijst op een lagere variabiliteit binnen de meetreeks. De afname in zowel centrale tendens als spreiding suggereert dat PM_{10} , als grootste fractie, het sterkst onderhevig is aan hygroscopische groei, en dus aanzienlijk beïnvloed wordt door luchtvochtigheid.



Figuur 23: Vergelijking PM_{10} -waarden zonder en met voorverwarming op het dakterras van de AMA

Bron: Eigen werk

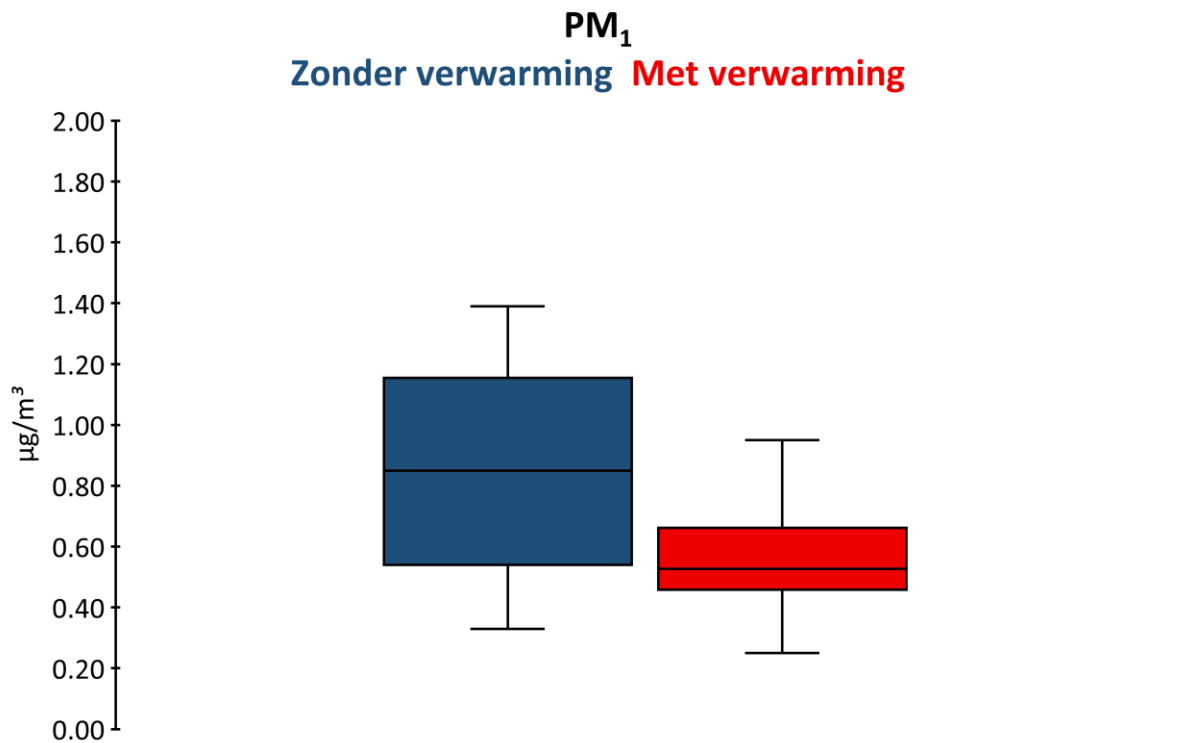
Voor $PM_{2.5}$ werd een soortgelijke trend vastgesteld. De mediaan daalde van $0,88 \mu\text{g}/\text{m}^3$ naar $0,65 \mu\text{g}/\text{m}^3$, en het gemiddelde van $0,96$ naar $0,65 \mu\text{g}/\text{m}^3$. De standaardafwijking nam af van $0,41$ naar $0,27 \mu\text{g}/\text{m}^3$, wat duidt op een homogenisering van de meetwaarden. $PM_{2.5}$ blijkt dus eveneens gevoelig voor vochtgebonden deeltjesgroei, zij het in iets mindere mate dan PM_{10} .



Figuur 24: Vergelijking $PM_{2.5}$ -waarden zonder en met voorverwarming op het dakterras van de AMA

Bron: Eigen werk

Ook bij PM_1 is de invloed van verwarming merkbaar, zij het subtieler. De mediaan daalde van $0,85 \mu\text{g}/\text{m}^3$ naar $0,53 \mu\text{g}/\text{m}^3$, en het gemiddelde van $0,84$ naar $0,58 \mu\text{g}/\text{m}^3$. De standaardafwijking verminderde van $0,34$ naar $0,21 \mu\text{g}/\text{m}^3$. Wat wil zeggen dat ook bij de kleinste fijnstofdeeltjes het verwarmen van lucht een invloed heeft op de metingen.



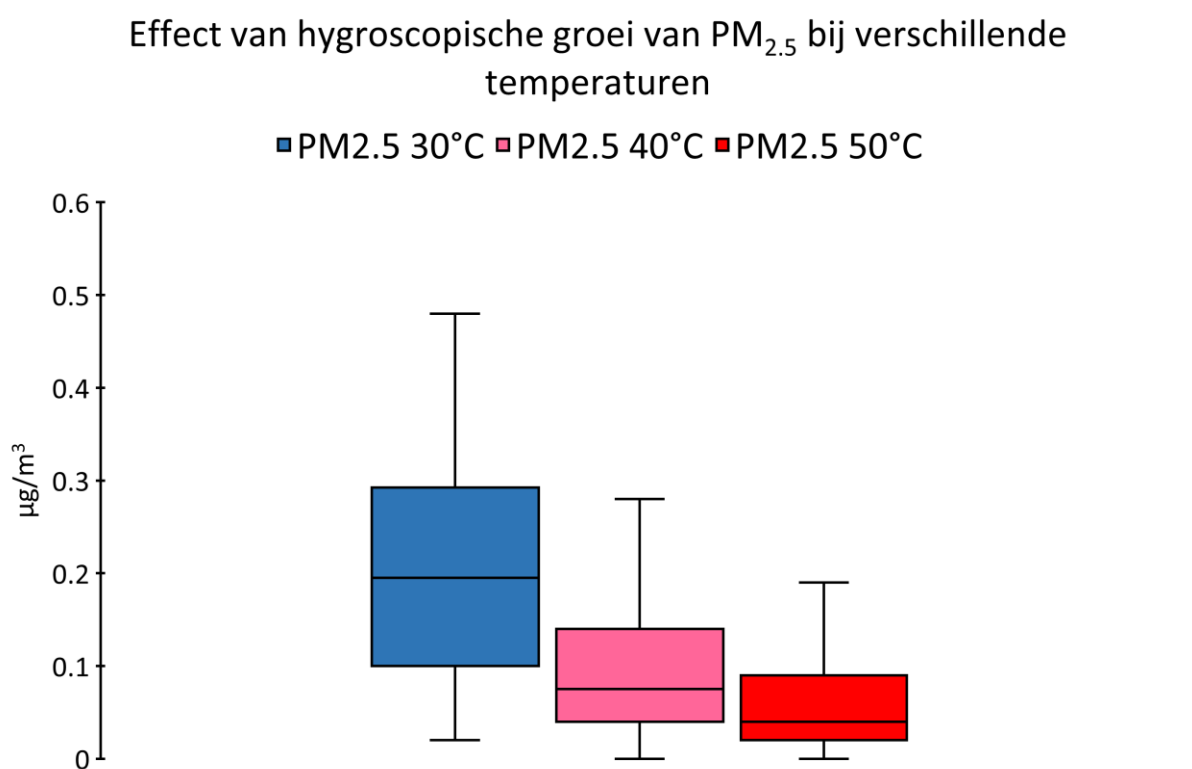
Figuur 25: Vergelijking PM_1 -waarden zonder en met voorverwarming op het dakterras van de AMA

Bron: Eigen werk

Het valt op dat de OPC-N3-sensor vaker fouten geeft bij het aanzetten van de verwarming en dat de foutieve antwoorden die de sensor terugstuurt gevarieerder zijn dan zonder verwarming. Deze fouten worden opgevangen door de software om alsnog een resultaat te bekomen van de sensor. Samenvattend suggereren de resultaten dat verwarming van de luchtinlaat effectief kan zijn in het reduceren van overschatting veroorzaakt door hygroscopische groei.

4.3.3 Invloed van ingestelde temperatuur van thermostaat op fijnstofmetingen

De ingestelde temperatuur is een grote parameter om hygroscopische groei tegen te gaan. Onderstaande grafiek toont het effect van de ingestelde temperatuur op de fijnstofmetingen aan 30 °C, 40 °C en 50 °C met telkens reeksen van 60 meetpunten. Deze test is in een binnenomgeving uitgevoerd. Hierbij daalde de relatieve luchtvochtigheid respectievelijk van gemiddeld van 52,23% naar 37,59% naar 27,53%.



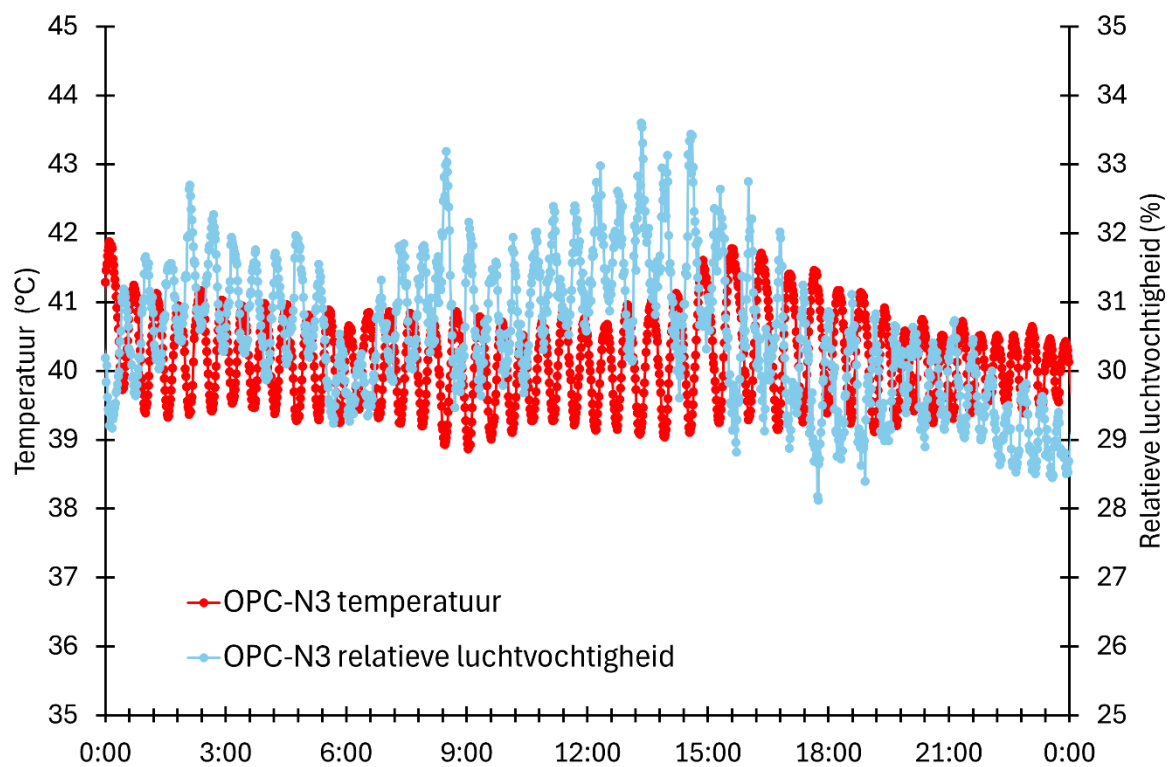
Figuur 26: Invloed van de ingestelde temperatuur op metingen van fijnstofconcentratie PM_{2.5}

Bron: Eigen werk

Het is echter af te raden om binnen de SeaCanairy een temperatuur hoger dan 50°C in te stellen. Dit zou er voor kunnen zorgen dat de zekeringen uitslaan. Ook heeft de OPC-N3 sensor volgens de specificaties maar een operationeel temperatuursbereik van -10°C tot 50°C.

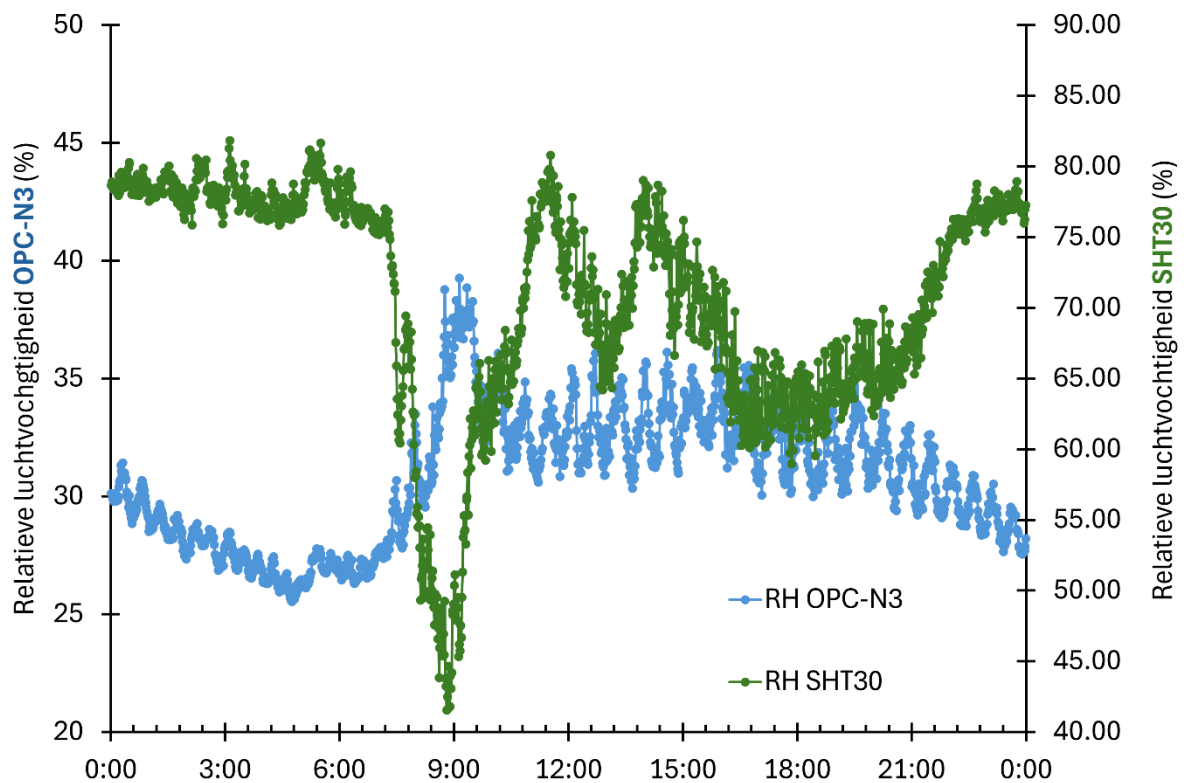
4.3.4 Invloed van luchtverwarming op de relatieve luchtvochtigheid

Verder is er het aspect dat de verwarming wordt geregeld in door een thermostaat. Dit wil zeggen eenmaal de ingestelde temperatuur bereikt is, zal deze vanzelf uitgaan tot de temperatuur weer zakt tot onder het ingestelde niveau. Dit heeft als gevolg dat er interne fluctuaties optreden met betrekking tot de gemeten temperatuur en relatieve vochtigheid door de OPC-N3-sensor. Figuur 27 geeft weer hoe deze twee parameters fluctueren over een periode van 24 uur in buitenlucht. Hieruit blijkt duidelijk dat de relatieve vochtigheid een golfpatroon vertoont die een hoogte heeft van ongeveer 4%. Het op en neer gaan van deze golfbeweging kan verklaard worden door de verandering in luchtvochtigheid tijdens de dag zoals weergegeven in Figuur 28: Invloed van de omgevingsluchtvochtigheid op de interne relatieve luchtvochtigheid binnen het meetsysteem.



Figuur 27: Relatie tussen temperatuur en relatieve luchtvochtigheid, intern gemeten door de OPC-N3-sensor. De figuur toont hoe temperatuurschommelingen, veroorzaakt door het in- en uitschakelen van de kastverwarming, een directe invloed hebben op de stabiliteit van de relatieve vochtigheid.

Bron: Eigen werk



Figuur 28: Invloed van de omgevingsluchtvochtigheid op de interne relatieve luchtvochtigheid binnen het meetsysteem.

Bron: Eigen werk

De conclusie van dit hoofdstuk is dat de relatieve luchtvochtigheid nooit constant zal zijn op langere termijn en zelfs op korte termijn met 4 á 5 % schommelt. Wat wel echter vaststaat is dat de opwarming van lucht de relatieve luchtvochtigheid effectief verlaagd en hygroscopische groei tegengaat.

Hoofdstuk 5 Praktische veldkalibratie van Alphasense-gassensoren

Voor de kalibratie van de sensoren werd een meetcampagne uitgevoerd in Zandhoven, waarbij 10229 meetpunten werden verzameld. Hoewel een ideale kalibratie uitgevoerd wordt naast een officieel meetstation dat ook het doelgas meet, kan dit hoofdstuk beschouwd worden als een voorstel tot veldkalibratie van de vier elektrochemische gassensoren van Alphasense voor toekomstig gebruik. De sensoren zijn ontworpen voor de detectie van NO_2 , O_3 , CO en SO_2 en leveren een uitgangssignaal in millivolt (mV). Dit signaal kan worden omgerekend naar een concentratie in $\mu\text{g}/\text{m}^3$. De berekeningen hiervoor zijn in een apart bestand gemaakt met Excel, dit bestand is samen met deze thesis apart ingediend.

De kalibratie werd in twee stappen gerealiseerd, namelijk:

1. Filtering van uitschieters en ruis op het elektrisch signaal

Het ruwe sensorsignaal wordt in een eerste stap gefilterd met statistische technieken zoals de Hampel-filter en de Savitzky-Golay-filter. Deze filters zorgen voor de onderdrukking van uitschieters en elektrische ruis zonder de onderliggende signaalstructuur te verstoren.

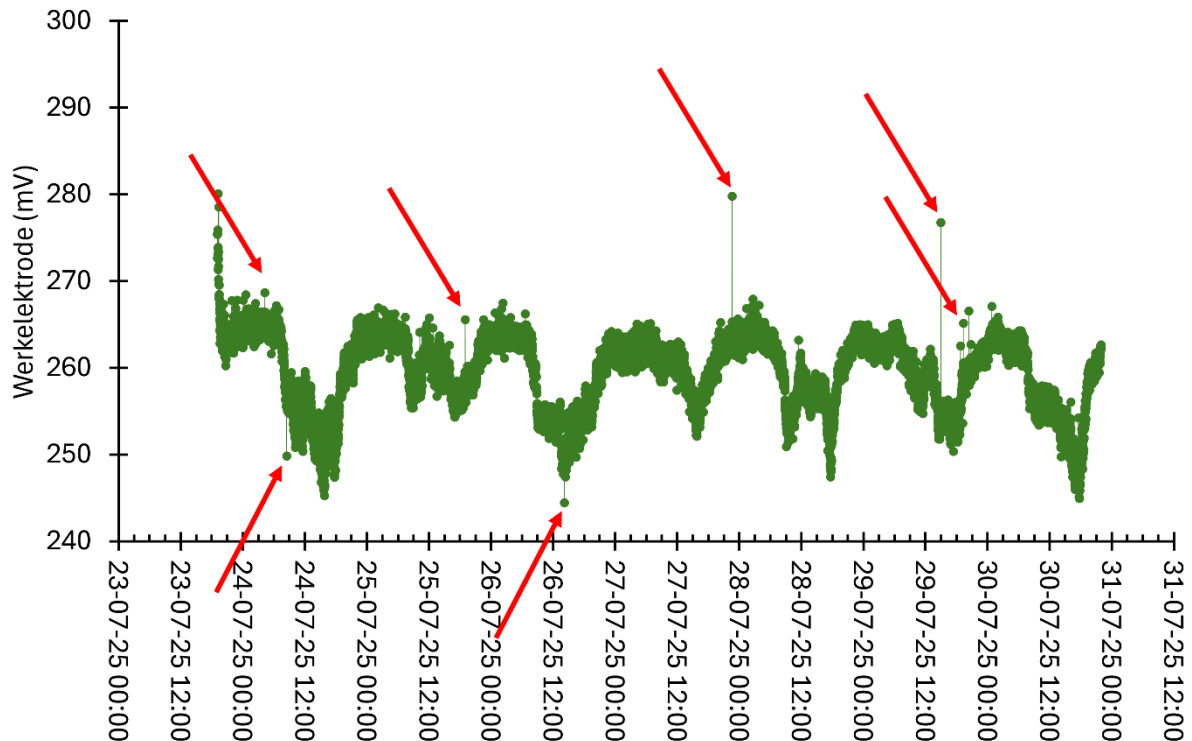
2. Kalibratie op basis van minimale en gemiddelde referentiepunten

De laagste sensorspanning wordt gelijkgesteld aan de laagste concentratie van een nabijgelegen VMM-metstation. Als tweede referentiepunt wordt de gemiddelde VMM-waarde genomen. Tussen deze twee punten wordt een lineaire interpolatie toegepast om de sensorspanning om te zetten naar een concentratie in $\mu\text{g}/\text{m}^3$, met lineaire extrapolatie voor waarden boven het gemiddelde.

5.1 Filteren en interpreteren van sensorresultaten

Tijdens de analyse van meetgegevens is het essentieel om zorgvuldig om te gaan met uitschieters en ruis, aangezien deze de betrouwbaarheid en interpretatie van de resultaten aanzienlijk kunnen beïnvloeden. **Fout! Verwijzingsbron niet gevonden.** toont hoe een enkele datapunten plots sterk stijgen of dalen ten opzichte van de omringende waarden, een duidelijk voorbeeld van een uitschieter die eerst gefilterd dient te worden.

Dit hoofdstuk bespreekt verschillende benaderingen voor het detecteren en hanteren van uitschieters, zowel visueel als analytisch. Aan de hand van concrete grafische voorbeelden wordt geïllustreerd hoe ruis en uitschieters zich manifesteren binnen de meetreeksen, en welke impact zij kunnen hebben op centrale tendenties, spreiding en foutdetectie. Het doel van dit hoofdstuk is om inzicht te bieden in de complexiteit van uitschieters en om te verduidelijken onder welke omstandigheden het uitsluiten, behouden of corrigeren van dergelijke waarden toegepast is binnen deze thesis.



Figuur 29: Uitschieters in het ruwe signaal van de werkelektrode van de NO₂-sensor

Bron: Eigen werk

5.1.1 Filtering van uitschieters met de Hampel-filter

Een methode binnen de robuuste statistiek is de Hampel-filter, die gebaseerd is op de mediaan en de mediaan van de absolute afwijkingen, in het Engels "*Median Absolute Deviation*" (MAD), binnen een bewegend venster. De Hampel-filter maakt gebruik van robuuste schatters die ongevoelig zijn voor individuele uitschieters. Door binnen elk venster te berekenen in welke mate een datapunt afwijkt van de centrale tendens in zijn lokale omgeving, kan systematisch worden bepaald of een waarde behouden blijft of vervangen wordt.

In de kalibratie van de gassensoren wordt gewerkt met een venstergrootte van 11 opeenvolgende meetpunten (symmetrisch: vijf ervoor, vijf erna rond een centraal punt) en een drempelwaarde van 3 maal de MAD. Datapunten waarvan de afwijking ten opzichte van de mediaan groter is dan deze grens worden vervangen door de mediaan van hun lokale venster. Deze aanpak zorgt ervoor dat lokale uitschieters onderdrukt worden, zonder de onderliggende structuur van het signaal te verstoren. De methode is bijzonder geschikt voor toepassingen waarbij lokale consistentie belangrijk is en waar globale filters, zoals de bewegende mediaan, onvoldoende onderscheid maken tussen structurele trends en ruis. De Hampel-filter kan bekomen worden door de volgende formule:

$$Z = \frac{|x_i - \tilde{x}|}{\text{MAD}}$$

Waarbij:

$$\text{MAD} = \text{mediaan}(|x_i - \tilde{x}|)$$

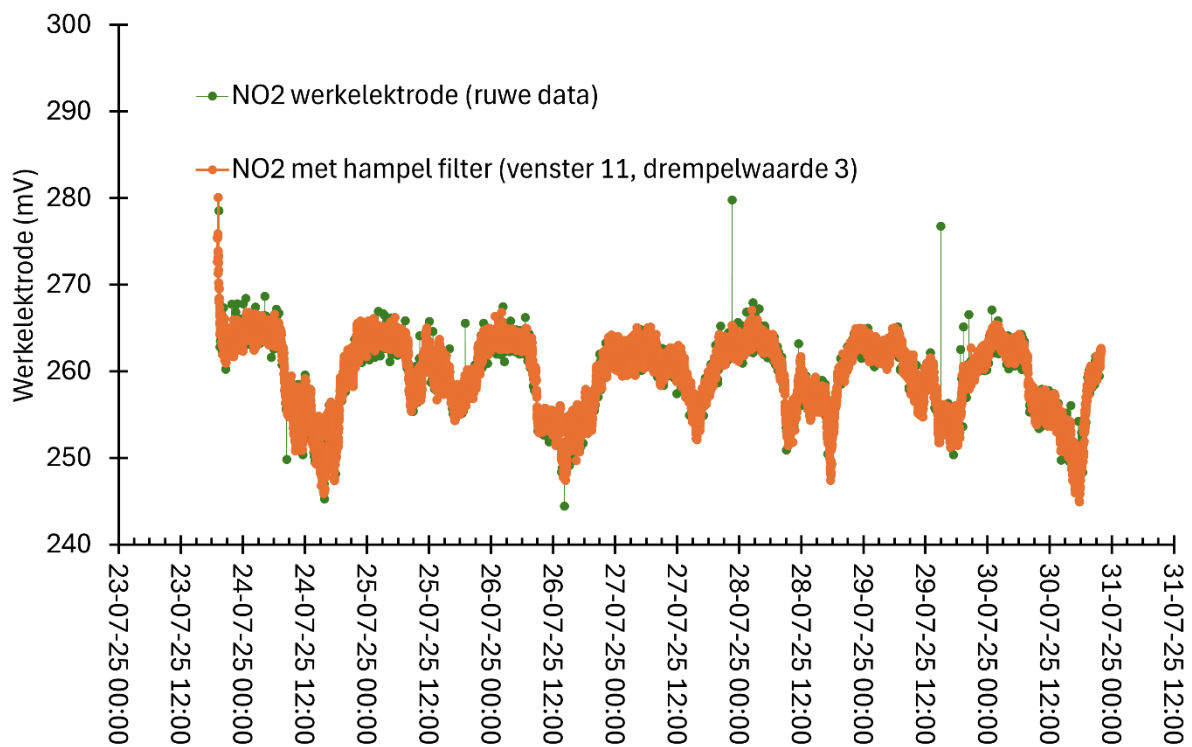
Met:

x_i De individuele meetwaarde voorstelt;

$\tilde{x}$ De mediaan van de reeks voorstelt;

En $|x_i - \tilde{x}|$ De absolute afwijking ten opzichte van de mediaan.

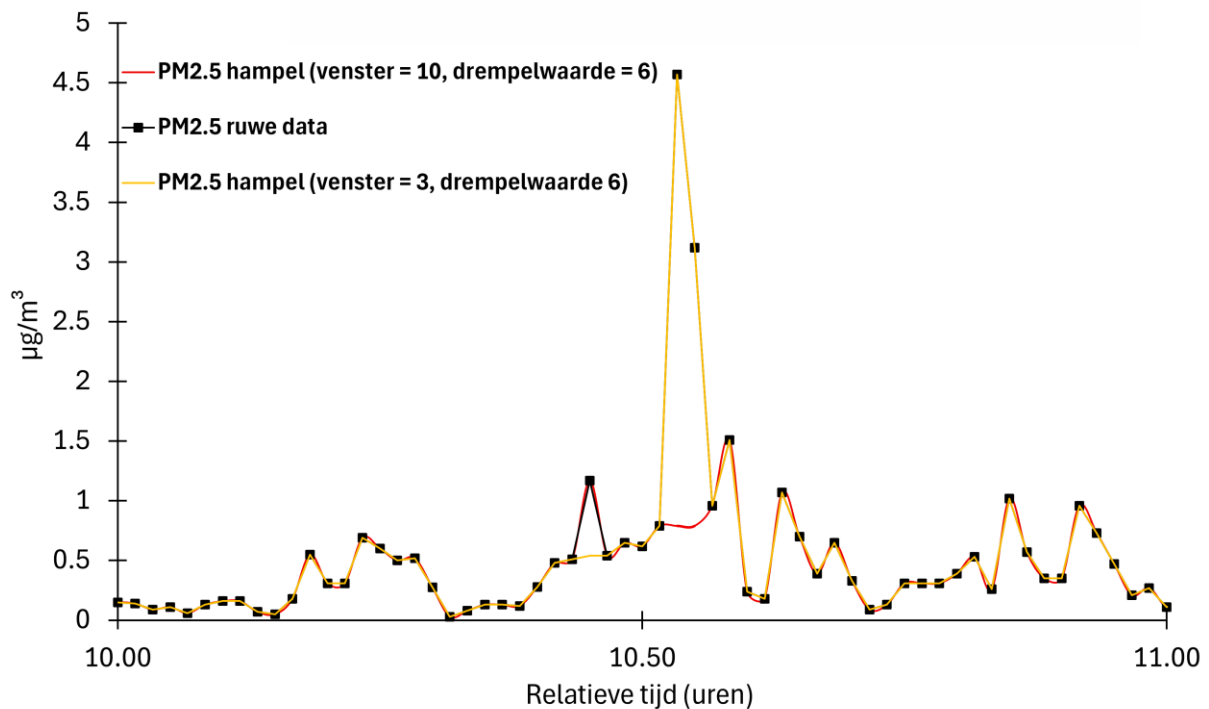
Wanneer $Z > 3$ wordt de waarde x_i als uitschieter beschouwd en vervangen door de waarde $\tilde{x}$, anders blijft de originele waarde x_i behouden.



Figuur 30: Toepassing van de Hampel-filter op de globale meetdata om de gassensoren te kalibreren.

Bron: Eigen werk

Tot slot is het belangrijk om kritisch te beoordelen of zogenaamde uitschieters wel daadwerkelijk statistische uitschieters zijn. Het is bijvoorbeeld mogelijk dat er een kortstondige piek in de concentratie van een bepaalde stof in de lucht heeft plaatsgevonden, die slechts tweemaal werd gedetecteerd door de SeaCanairy. In de onderstaande figuur zijn twee ruwe meetpunten voor $PM_{2.5}$ met een zwart vierkant aangeduid. Bij gebruik van een Hampel-filter met een venster van 10 worden deze punten als uitschieters aangeduid, terwijl dezelfde filter met een kleiner venster van 3 deze punten binnen het normale bereik plaatst. Dit illustreert hoe de keuze van het venster direct invloed heeft op de detectie van uitschieters en eventuele verdere kalibratie.



Figuur 31: Gevoeligheid van de Hampel-filter voor uitschieterdetectie bij verschillende vensters

Bron: Eigen werk

5.1.2 Reductie van signaalruis met de Savitzky-Golay-filter

Omdat elektrische signalen ruis kunnen veroorzaken in het bekomen resultaat, is het nuttig om deze ruis te onderdrukken. Daarom werd de Savitzky-Golay-filter toegepast op de dataset, nadat deze eerst werd gecorrigeerd met een Hampel-filter.

Hiervoor is wederom een venster van 11 gekozen met een polynoom van de 2^{de} graad. Een orde van de 3^{de} graad kan nuttig zijn als er plotse fluctuaties worden verwacht, maar binnen deze kalibratie is er gekozen voor een meer evenwichtigere orde van de 2^{de} graad. Om deze filter toe te passen dient er voor elke waarde binnen het lokaal venster van de datareeks vermenigvuldigt te worden met een coëfficiënt. Deze coëfficiënten zijn af te leiden uit tabel 1 in Figuur 32: Schermafdruck van Tabel 1 uit het werk van Abraham Savitzky en Marcel J. E. Golay. *Smoothing and Differentiation of Data* van het originele werk van de uitvinders van deze filter. [14] In de onderstaande afbeelding staat het venster van 11 aangeduid in het rood. De waarde 429 is de som van de bovenste

waarden. Om de uiteindelijke coëfficiënten te bekomen dienen de verschillende waarden gedeeld te worden door 429.

TABLE I

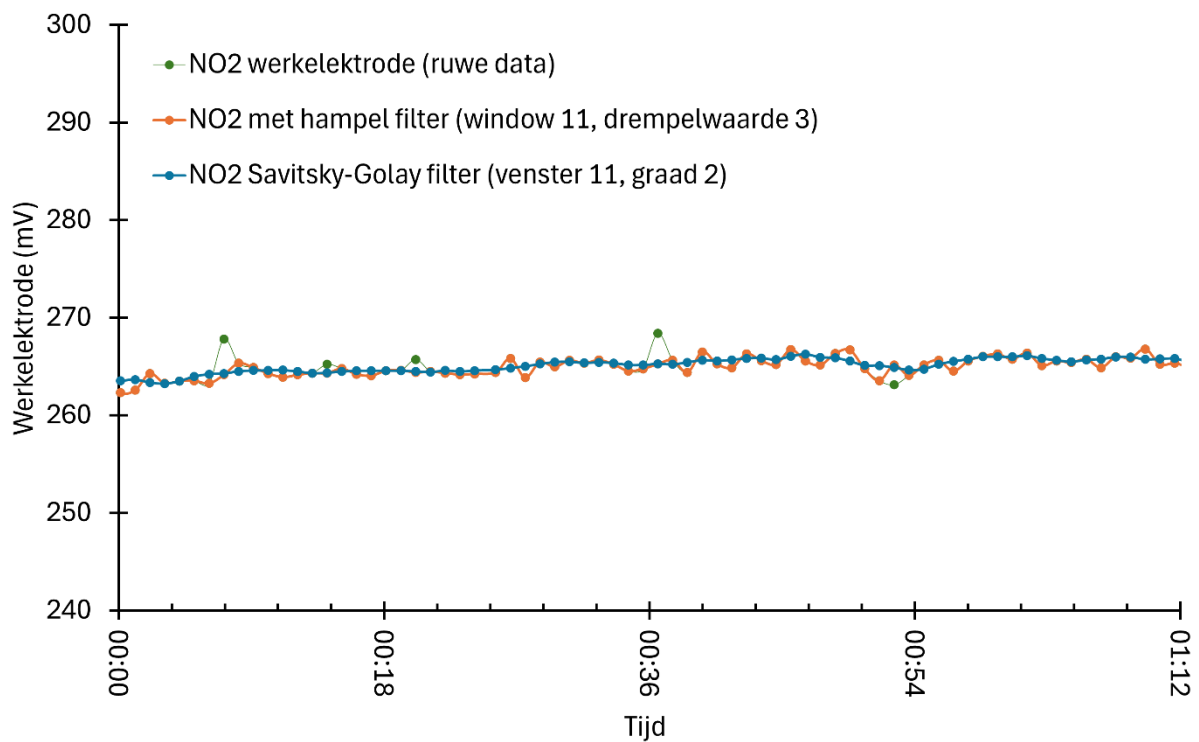
CONVOLUTES	SMOOTHING		QUADRATIC		CUBIC		A20	A30			
POINTS	25	23	21	19	17	15	13	11	9	7	5
-12	-253										
-11	-138	-42									
-10	-33	-21	-171								
-09	62	-2	-76	-136							
-08	147	15	5	-51	-21						
-07	222	30	84	24	-6	-78					
-06	287	43	149	89	7	-13	-11				
-05	322	54	204	144	18	42	0	-36			
-04	387	63	249	189	27	87	9	9	-21		
-03	422	70	284	224	34	122	16	44	14	-2	
-02	447	75	309	249	39	147	21	69	39	3	-3
-01	462	78	324	264	42	162	24	84	54	6	12
00	467	79	329	269	43	167	25	89	59	7	17
01	462	78	324	264	42	162	24	84	54	6	12
02	447	75	309	249	39	147	21	69	39	3	-3
03	422	70	284	224	34	122	16	44	14	-2	
04	387	63	249	189	27	87	9	9	-21		
05	322	54	204	144	18	42	0	-36			
06	287	43	149	89	7	-13	-11				
07	222	30	84	24	-6	-78					
08	147	15	9	-51	-21						
09	62	-2	-76	-136							
10	-33	-21	-171								
11	-138	-42									
12	-253										
NORM	5175	8059	1059	2261	323	1105	143	429	231	21	35

Figuur 32: Schermafdruck van Tabel 1 uit het werk van Abraham Savitzky en Marcel J. E. Golay. *Smoothing and Differentiation of Data*

Bron: Savitzky, A. & Golay, M.J.E. (1964) [14]

Tabel 10: Berekening van de coëfficiënten voor de Savitsky-Golay-filter met een 2^{de} orde polynoom en een venster van 11.

$-\frac{36}{429}$	$\frac{9}{429}$	$\frac{44}{429}$	$\frac{69}{429}$	$\frac{84}{429}$	$\frac{89}{429}$	$\frac{84}{429}$	$\frac{69}{429}$	$\frac{44}{429}$	$\frac{9}{429}$	$-\frac{36}{429}$
-0.0839	0.0210	0.1026	0.1608	0.1958	0.2075	0.1958	0.1608	0.1026	0.0210	-0.0839



Figuur 33: Toepassing van de Savitzky-Golay-filter na de Hampel-filter

Bron: Eigen werk

Omdat zowel de Hampel- als de Savitzky-Golay-filter werken op basis van een glijdend venster, treden er aan het begin en einde van de dataset problemen op. In deze randgebieden is het venster onvolledig, waardoor de filtering minder betrouwbaar wordt. Daarom worden deze randwaarden enkel gebruikt voor het filterproces zelf, maar niet meegenomen in de uiteindelijke kalibratie van de sensorsignalen.

5.2 Kalibratie op basis van minimale en gemiddelde referentiepunten

De vier gassensoren van Alphasense (voor NO₂, O₃, CO en SO₂) geven een uitgangssignaal in millivolt (mV). Om deze signalen om te zetten naar concentraties in µg/m³ wordt gebruikgemaakt van referentiedata van de Vlaamse Milieumaatschappij (VMM)¹.

Voor zowel de sensorsignalen als de VMM-metingen wordt het minimum en het gemiddelde bepaald over dezelfde meetperiode. Hierbij wordt aangenomen dat:

- de laagste gemeten sensorspanning overeenkomt met de laagste VMM-concentratie;
- de gemiddelde sensorspanning overeenkomt met de gemiddelde VMM-concentratie.

Op basis van deze twee punten wordt een lineaire interpolatie toegepast om de overige waarden om te rekenen naar µg/m³. Voor waarden boven het gemiddelde wordt lineaire extrapolatie toegepast.

Hoewel de afstand tussen de meetlocatie en de referentiestations relatief groot is, werd toch getracht een kalibratie uit te voeren die in de toekomst kan dienen om de resultaten te verbeteren. Door tijdsgebrek was het echter niet mogelijk om de SeaCanairy gedurende langere tijd bij de specifieke meetstations voor elk doelgas te plaatsen.

¹ Officiële data is te raadplegen op de volgende link: <https://www.irceline.be/en/air-quality/measurements/monitoring-stations>

5.2.1 Opbouw van de Alphasense-gassensoren

Om de werking van de gassensoren te begrijpen is het belangrijk de begrippen werkelektrode (WE) en hulpelektrode (AE, ook wel counter electrode genoemd) toe te lichten. De werkelektrode is de elektrode waar de reactie met het doelgas plaatsvindt: oxidatie of reductie zorgt hier voor een stroom die evenredig is met de concentratie van het gas. De hulpelektrode sluit de elektrische kring en zorgt ervoor dat de benodigde stroom geleverd kan worden om de werkelektrode in evenwicht te houden. Omdat beide elektroden ook gevoelig zijn voor algemene elektrische offsets en storingen, geeft Alphasense aan om het signaal van de hulpelektrode af te trekken van dat van de werkelektrode ($WE - AE$). Hierdoor wordt het uiteindelijke meetsignaal gecorrigeerd voor ongewenste achtergrondstromen en invloeden die niet door het doelgas veroorzaakt zijn.

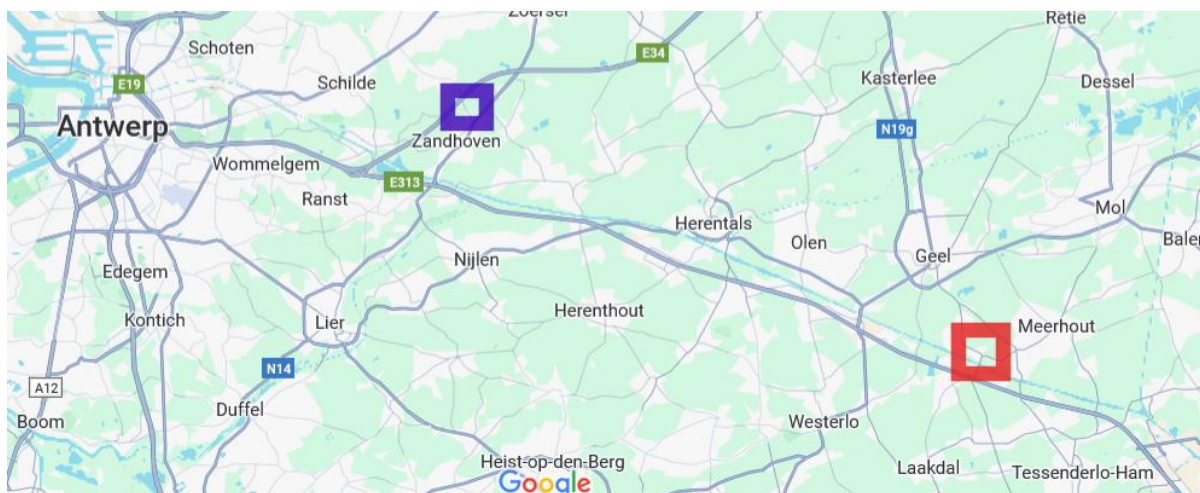
De volgende gassensoren zijn in de SeaCanairy geïnstalleerd:

Tabel 11: De gebruikte Alphasense-gassensoren en hun serienummer

Doelgas	Fabrieksnaam	Serienummer
NO ₂	NO2-A43F	212870427
O ₃ + NO ₂	OX-A431	214810129
SO ₂	SO2-A4	134330003
CO	CO-A4	132330502

5.2.2 Kalibratie van de NO₂-sensor

Voor de kalibratie van de NO₂-sensor werd gebruikgemaakt van het meetstation 40LD02 – Laakdal, gelegen op circa 27,5 km van de meetlocatie in Zandhoven. Dit station werd gekozen omdat het, net als Zandhoven, in een landelijk gebied ligt en daardoor representatiever is dan de stedelijke meetstations in Antwerpen, waar verkeersinvloeden de luchtkwaliteit sterk beïnvloeden.



Figuur 34: In het rood de locatie van het meetstation 40LD02 – Laakdal en in het blauw meetlocatie in Zandhoven

Bron: Eigen werk

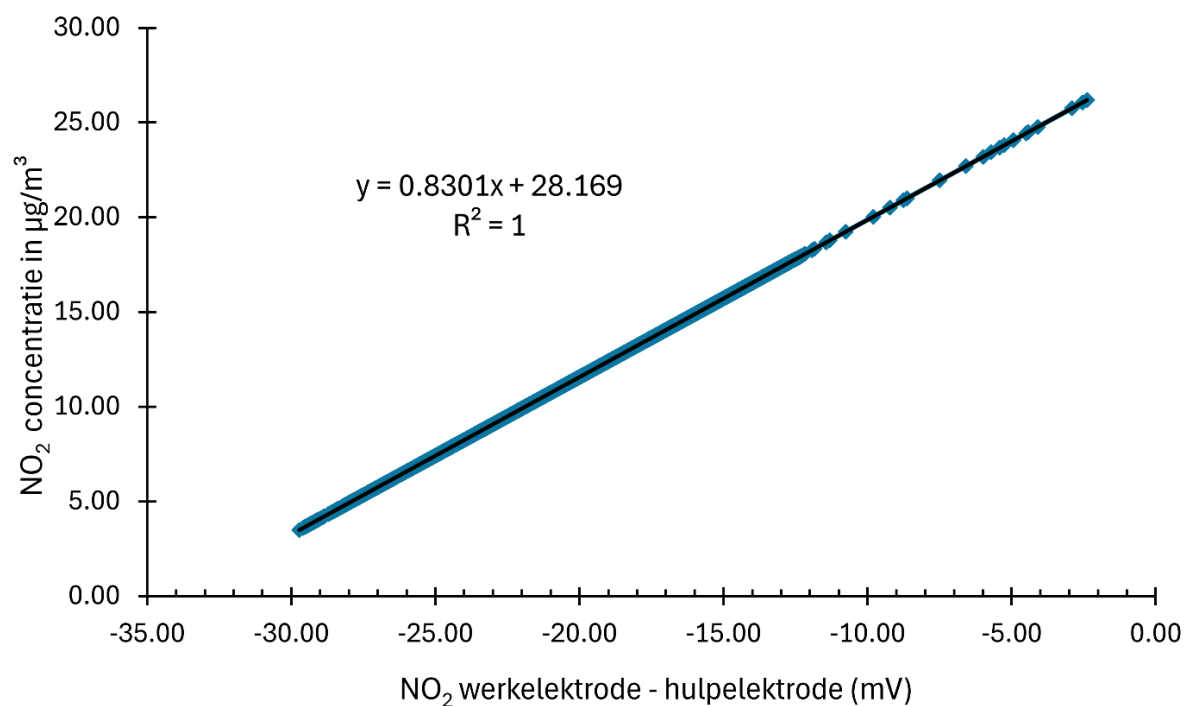
Op basis van de meetdata zijn de minimale en gemiddelde waarden berekend voor zowel het referentiemeetstation als de Alphasense-sensoren. Deze referentiepunten zijn samengevat in Tabel 12 hieronder. Met behulp van een Pythonprogramma (zie Bijlage XI Python-programma om de gassensoren van Alphasense om te zetten in een waarde naar $\mu\text{g}/\text{m}^3$) is vervolgens een lineaire interpolatie uitgevoerd, waarmee de gemeten signalen in mV zijn omgerekend naar concentraties in $\mu\text{g}/\text{m}^3$.

Figuur 35 toont hoe de geconverteerde waarden samen met de oorspronkelijke mV-signalen in één grafiek kunnen worden weergegeven, zodat voor elke spanning een overeenkomstige concentratie beschikbaar is en een lineaire functie kan worden afgeleid. Omdat lineaire interpolatie wordt toegepast, is de determinatiecoëfficiënt (R^2) per definitie gelijk aan 1.

Tot slot illustreert Figuur 36 de evolutie van de data tijdens de meetcampagne. Zowel de door het VMM-meetstation gemeten concentraties als de omgerekende concentraties in $\mu\text{g}/\text{m}^3$ worden weergegeven, waaruit blijkt dat beide reeksen een vergelijkbare trend volgen. Daarnaast toont de werkelektrode dat zelfs zonder kalibratie veranderingen in de luchtkwaliteit duidelijk waarneembaar zijn.

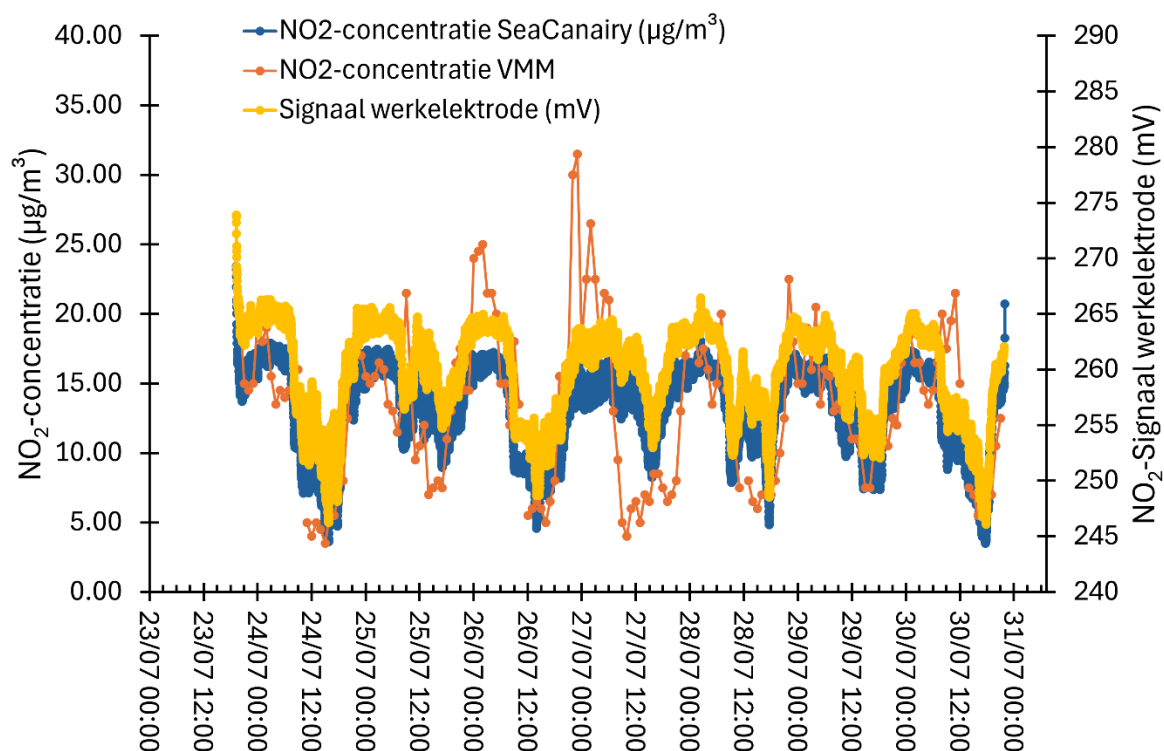
Tabel 12: Referentiepunten voor kalibratie van de NO₂-sensor met VMM-data

	Signaal WE - AE (mV)	Concentratie VMM (µg/m ³)
Minimum	-29.72	3.50
Gemiddelde	-18.01	13.22



Figuur 35: Relatie tussen sensorspanning en gekalibreerde concentratie voor NO₂

Bron: Eigen werk

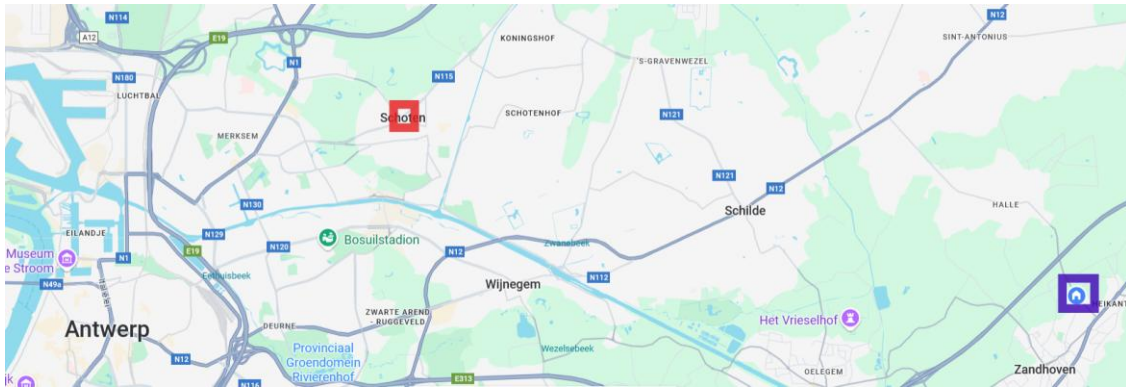


Figuur 36: Vergelijking van geconverteerde NO₂-concentraties, VMM-referentiedata en de werkelektrode tijdens de meetcampagne

Eigen werk samengevoegd met data van de Vlaamse Milieu Maatschappij [9]

5.2.3 Kalibratie van de O₃-sensor

Voor de kalibratie van de O₃-sensor is gebruikgemaakt van het meetstation 42R811-Schoten, gelegen op circa 12 km van de meetlocatie in Zandhoven. Hoewel het wenselijk zou zijn om voor alle gasen hetzelfde referentiestation te gebruiken, meet het station in Laakdal geen ozon. Daarom is gekozen voor Schoten, dat geografisch het dichtst bij Zandhoven gelegen is.

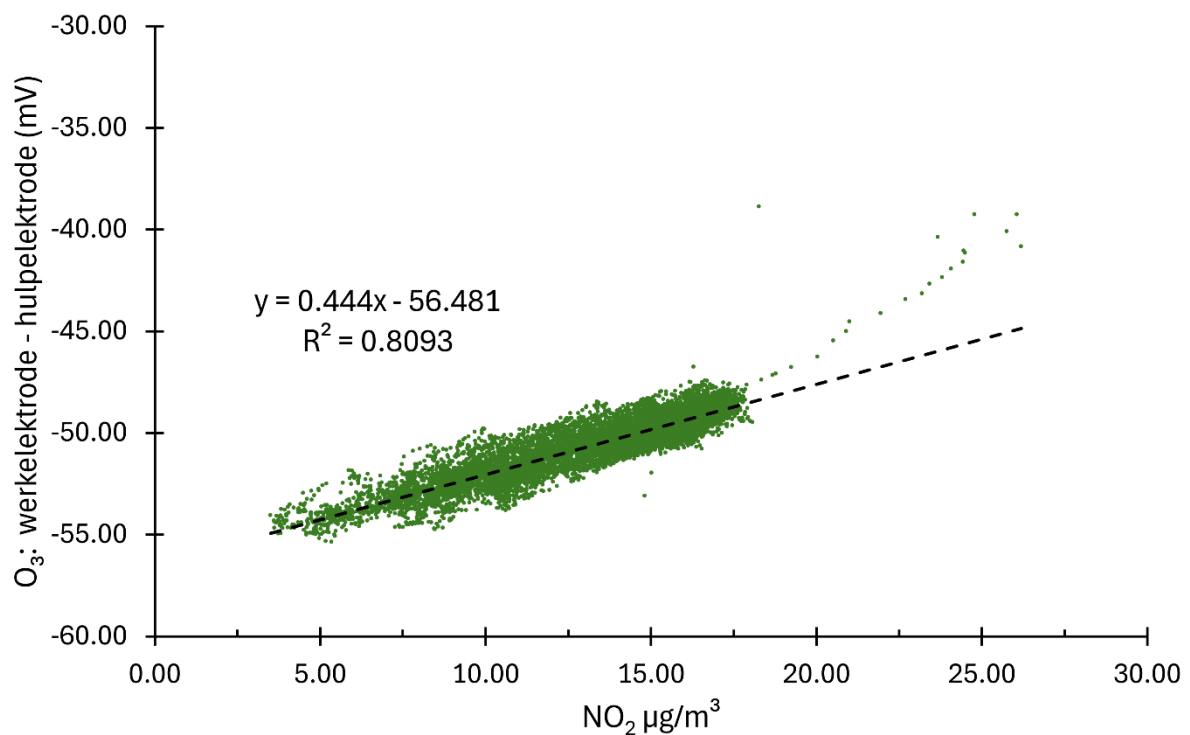


Figuur 37: In het rood de locatie van het meetstation 42R811 in Schoten en in het blauw meetlocatie in Zandhoven

Bron: Eigen werk

De sensor die ozon meet is ook gevoelig voor stikstofdioxide, een fenomeen dat kruissensitiviteit wordt genoemd. Om deze te corrigeren, is gebruikgemaakt van de reeds bepaalde NO_2 -concentraties. Hiervoor werd een grafiek opgesteld met op de X-as de NO_2 -concentratie en op de Y-as het verschil tussen werkelektrode en hulpelektrode van de O_3 -sensor. Uit deze relatie kon een correctiefunctie worden afgeleid waarmee het NO_2 -signaal werd gefilterd. Het resterende signaal in millivolt is vervolgens toe te schrijven aan de blootstelling aan ozon. Hieruit blijkt dat de determinantecoefficiënt gelijk is aan $R^2 = 0.8093$ wat een geloofwaardig verband aantoont. Hierbij is wel op te merken dat voor hogere waarden er een grotere afwijking van de correctiefunctie waar te nemen is.

Het is niet mogelijk om eenvoudigweg de NO_2 -concentratie af te trekken van de gecombineerde $\text{O}_3 + \text{NO}_2$ -concentratie, omdat de O_3 -sensor verschillende gevoeligheden heeft voor beide gassen. Het sensorsignaal is een som van bijdragen in millivolt, niet rechtstreeks van concentraties. Om kruissensitiviteit correct te compenseren, moet eerst het NO_2 -aandeel op signaalniveau (mV) worden bepaald en afgetrokken, waarna het resterende deel aan O_3 kan worden toegeschreven.



Figuur 38: Correctiefunctie voor NO₂-invloed op O₃-sensor

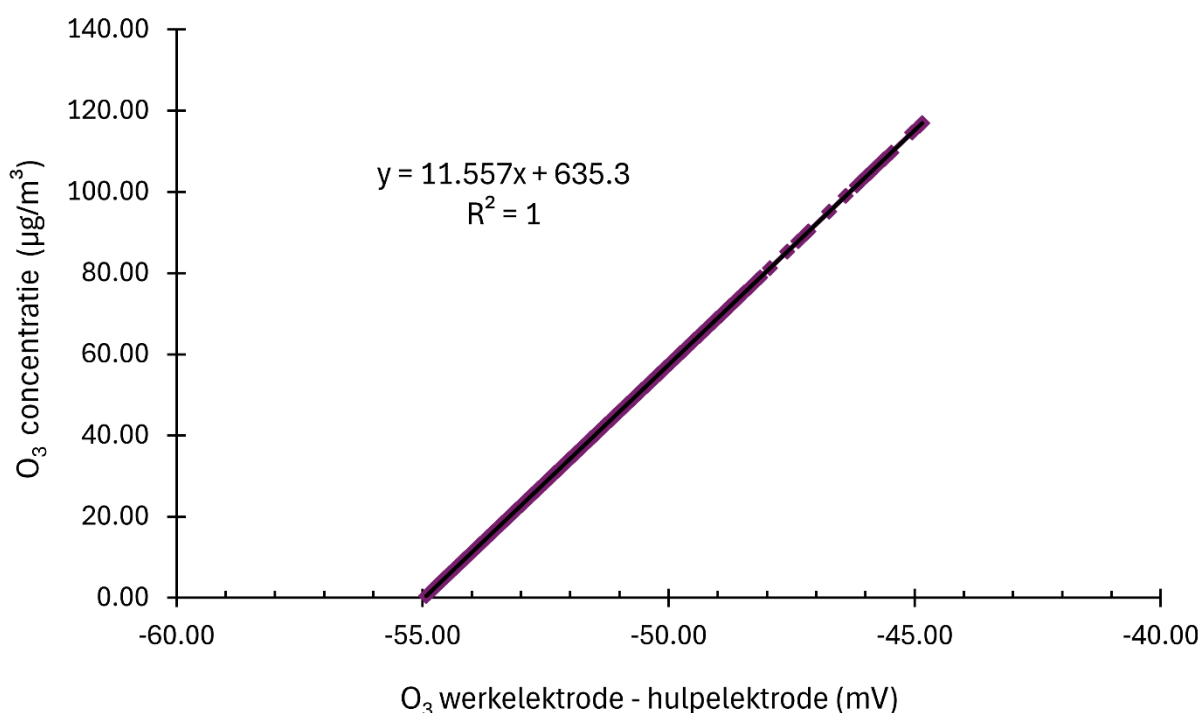
Bron: Eigen werk

Met behulp van de correctiefunctie kan een spanning in mV worden berekend die de bijdrage van ozon benadert alsof er geen kruissensitiviteit aanwezig zou zijn. Deze spanning wordt verkregen door in de afgeleide functie $y = 0.444x - 56.481$ de eerder berekende concentratie van NO₂ in te vullen. Door dit voor alle meetpunten toe te passen, wordt een reeks waarden bekomen die de ozonrespons in mV representeren, op basis waarvan vervolgens de twee referentiepunten voor kalibratie kunnen worden vastgesteld om vervolgens te interpoleren.

Figuur 39 toont hoe de geïnterpoleerde waarden samen met de oorspronkelijke mV-signalen in één grafiek worden weergegeven, zodat voor elke spanning een overeenkomstige ozonconcentratie in µg/m³ beschikbaar is en een lineaire functie kan worden afgeleid.

Tabel 13: Referentiepunten voor kalibratie van de O₃-sensor met VMM-data

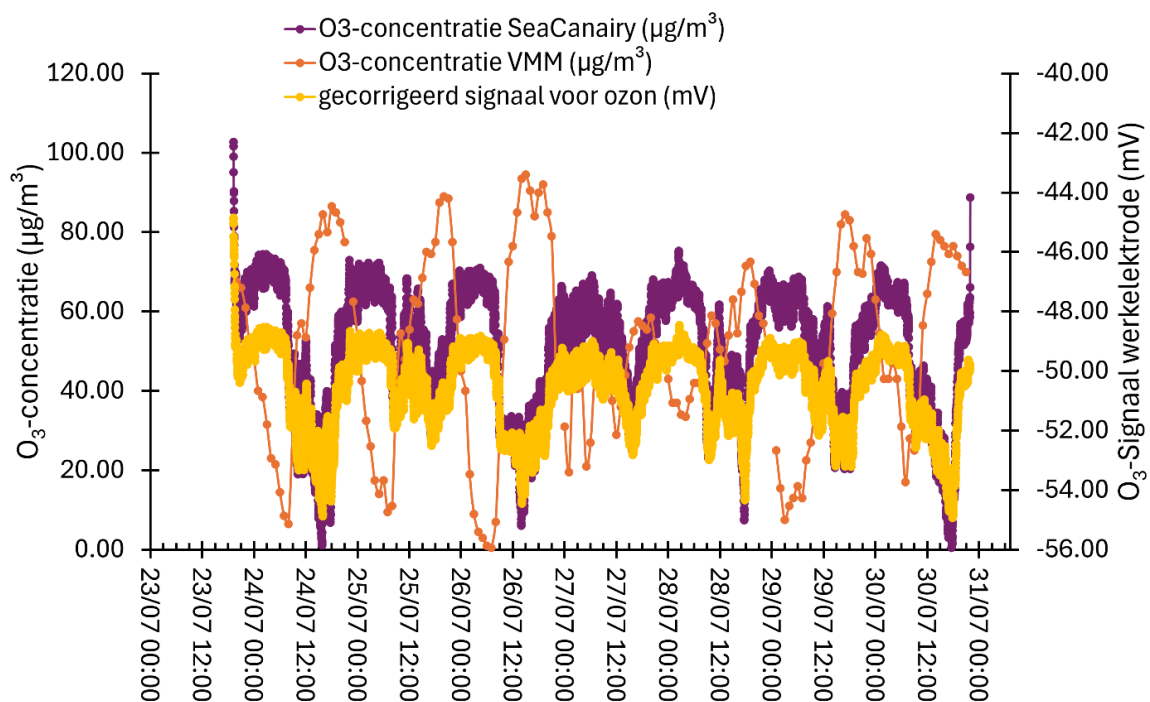
	Signaal WE - AE (mV)	Concenatratie VMM (µg/m ³)
Minimum	-54.93	0.50
Gemiddelde	-50.60	50.54



Figuur 39: Relatie tussen sensorspanning en gekalibreerde concentratie voor O₃

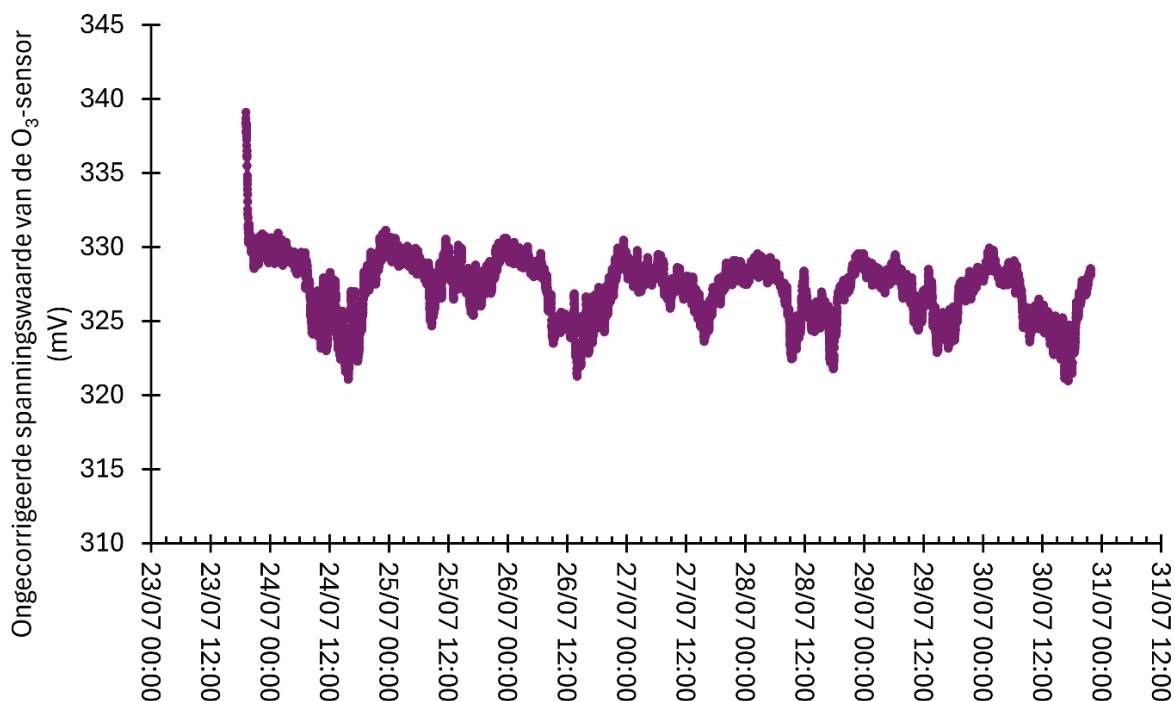
Bron: Eigen werk

Figuur 40 toont de evolutie van de data tijdens de meetcampagne voor zowel de referentiewaarden van het meetstation als de door de SeaCanairy omgerekende concentraties. De gecorrigeerde spanning is hierbij berekend met de correctiefunctie uit Figuur 38. Opvallend is dat bij een stijging of daling van de concentratie in µg/m³ de gecorrigeerde spanning in mV een overeenkomstige verandering vertoont. Hoewel doorgaans hogere ozonconcentraties overdag worden verwacht, volgt de gemeten trend eerder het patroon van NO₂ (zie Figuur 36). Ditzelfde gedrag wordt bevestigd door de ongecorrigeerde werkelektrode van de O₃-sensor in Figuur 41. De afwijking van het verwachte patroon valt hoogstwaarschijnlijk toe te schrijven aan de kruissensitiviteit van de sensor met NO₂. Een juiste kalibratie waarbij de doos naast een meetstation wordt geplaatst dat zowel NO₂ als O₃ meet zal dan ook betrouwbaardere resultaten opbrengen.



Figuur 40: Vergelijking van geconverteerde O₃-concentraties, VMM-referentiedata en de werkelektrode tijdens de meetcampagne

Bron: Eigen werk samengevoegd met data van de Vlaamse Milieu Maatschappij[9]

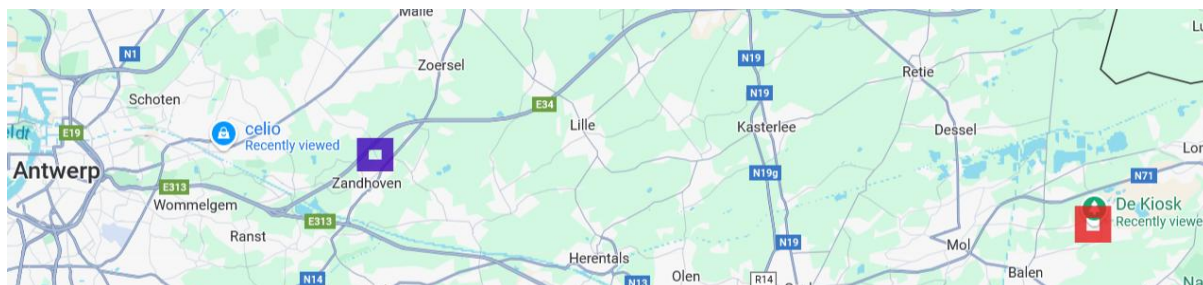


Figuur 41: Evolutie van de ongecorrigeerde werkelektrode van de O₃-sensor tijdens de meetcampagne

Bron: Eigen werk

5.2.4 Kalibratie van de SO₂-sensor

Voor de kalibratie van de SO₂-sensor is gebruikgemaakt van het meetstation 40WZ02 - Mol, gelegen op circa 38km van de meetlocatie in Zandhoven.



Figuur 42: In het rood de locatie van het meetstation 42R811 in Schoten en in het blauw meetlocatie in Zandhoven

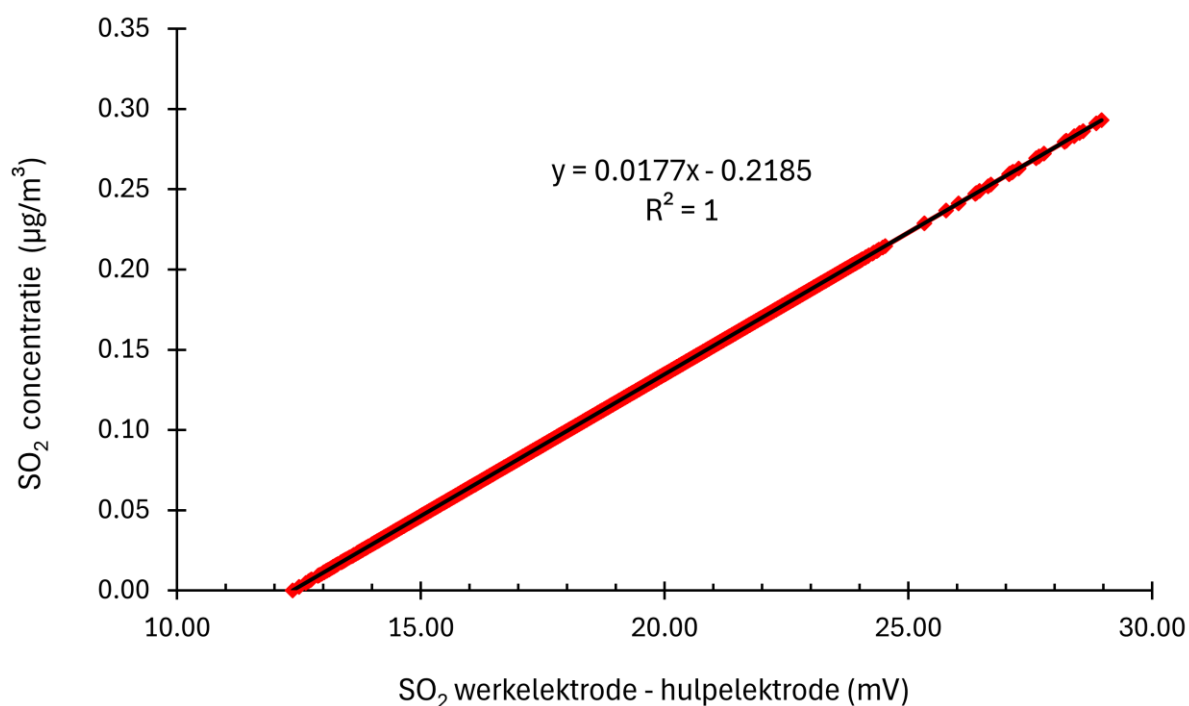
Bron: Eigen werk

Voor de metingen van SO₂ is het op te merken dat deze waarde zich over het algemeen zeer laag bevinden, niettemin valt er wederom een referentietabel op te stellen gevolgd door een grafiek in Figuur 43 waaruit de lineaire functie wordt bepaald. Tot slot geeft Figuur 44 de concentratie van het referentiestation in Mol samen met de geconverteerde meetwaarden van de SO₂-sensor. Verder geeft deze figuur opnieuw de werkelektrode.

Uit Figuur 44 blijkt dat de officiële SO₂-concentraties van het referentiestation in Mol niet geleidelijk toenemen, maar in sprongen van 0,5 µg/m³ worden weergegeven. Bovendien registreert het station vaak een concentratie van 0 µg/m³. De Alphasense-sensoren daarentegen tonen continue variaties boven het nulpunt. Opvallend is dat het meetstation enkele pieken detecteerde die niet zichtbaar zijn in de sensordata, mogelijk door lokale bronnen in de omgeving van het station. De werkelektrode daarentegen volgt wel de trend van de omgevormde concentratie.

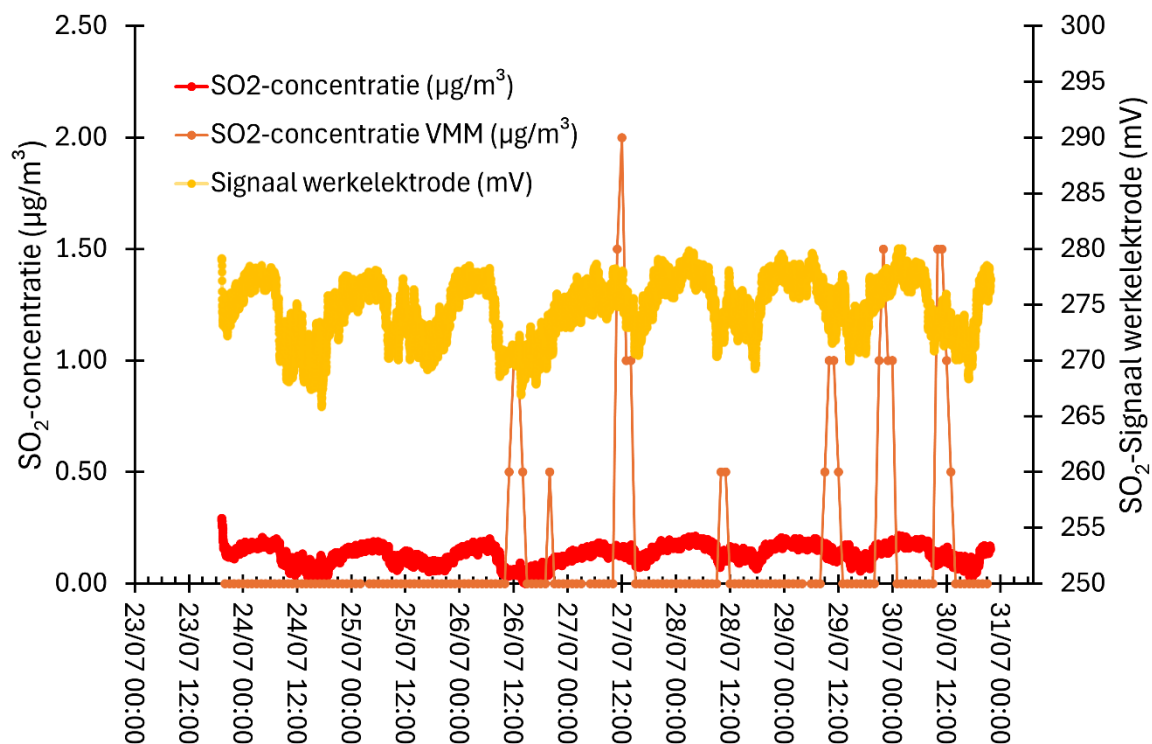
Tabel 14: Referentiepunten voor kalibratie van de SO₂-sensor met VMM-data

	Signaal WE – AE (mV)	Concentratie VMM (µg/m ³)
Minimum	12.37	0.00
Gemiddelde	19.73	0.13



Figuur 43: Relatie tussen sensorspanning en gekalibreerde concentratie voor SO₂

Bron: Eigen werk

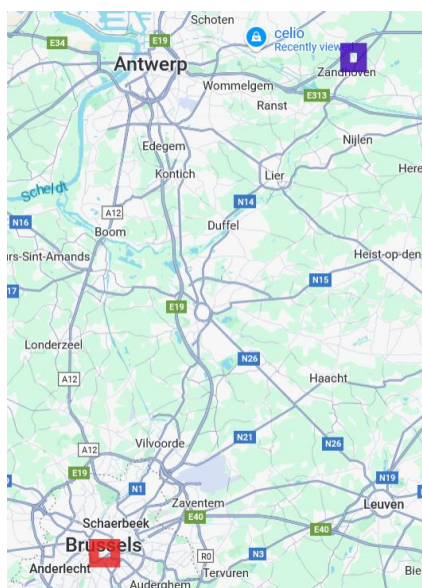


Figuur 44: Vergelijking van geconverteerde SO₂-concentraties, VMM-referentiedata en de werkelektrode tijdens de meetcampagne

Eigen werk samengevoegd met data van de Vlaamse Milieu Maatschappij [9]

5.2.5 Kalibratie van de CO-sensor

Voor de kalibratie van de CO-sensor is gebruikgemaakt van het meetstation 41B006 - Bruxelles (Parlement UE), gelegen op circa 38km van de meetlocatie in Zandhoven. Voor dit gas valt het op dat er in Vlaanderen geen meetstations meer actief zijn die CO meten en het dichtstbijzijnde meetstation zich in Brussel bevindt. Verder is het ook belangrijk te vermelden dat deze stations een waarde geven in mg/m^3 in plaats van $\mu\text{g}/\text{m}^3$.



Figuur 45: In het rood de locatie van het meetstation 41B006 - Bruxelles (Parlement UE) en in het blauw meetlocatie in Zandhoven

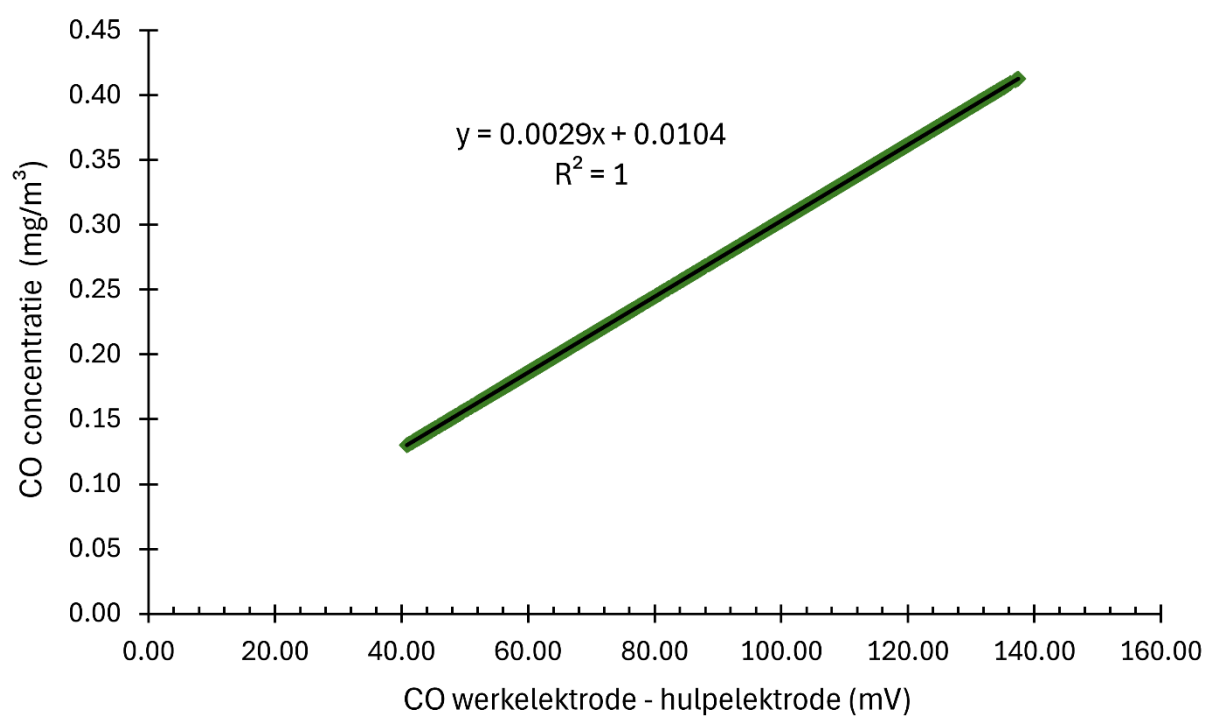
Bron: Eigen werk

Figuur 46 toont de geconverteerde waarde waarbij de functie bekomen wordt om voor een bepaalde waarde in mV een waarde in mg/m^3 bekomen kan worden, verder toont figuur 47 de concentratie van het referentiestation in Mol samen met de geconverteerde meetwaarden van de SO_2 -sensor. Verder geeft deze figuur opnieuw de werkelektrode.

In Figuur 47 zijn de waarde omgezet van mg/m^3 naar $\mu\text{g}/\text{m}^3$. Hoewel de waarden niet overeenkomen tussen het referentiestation en de geconverteerde waarde valt het wel op dat als de CO globaal stijgt, de gemeten waarde door de SeaCanairy ook licht zal stijgen. De afstand en het verschil tussen landelijk en stadsomgeving zorgen er wellicht voor dat er een groot verschil is tussen de gemeten waarde van het referentiestation in Brussel en de meetlocatie in Zandhoven.

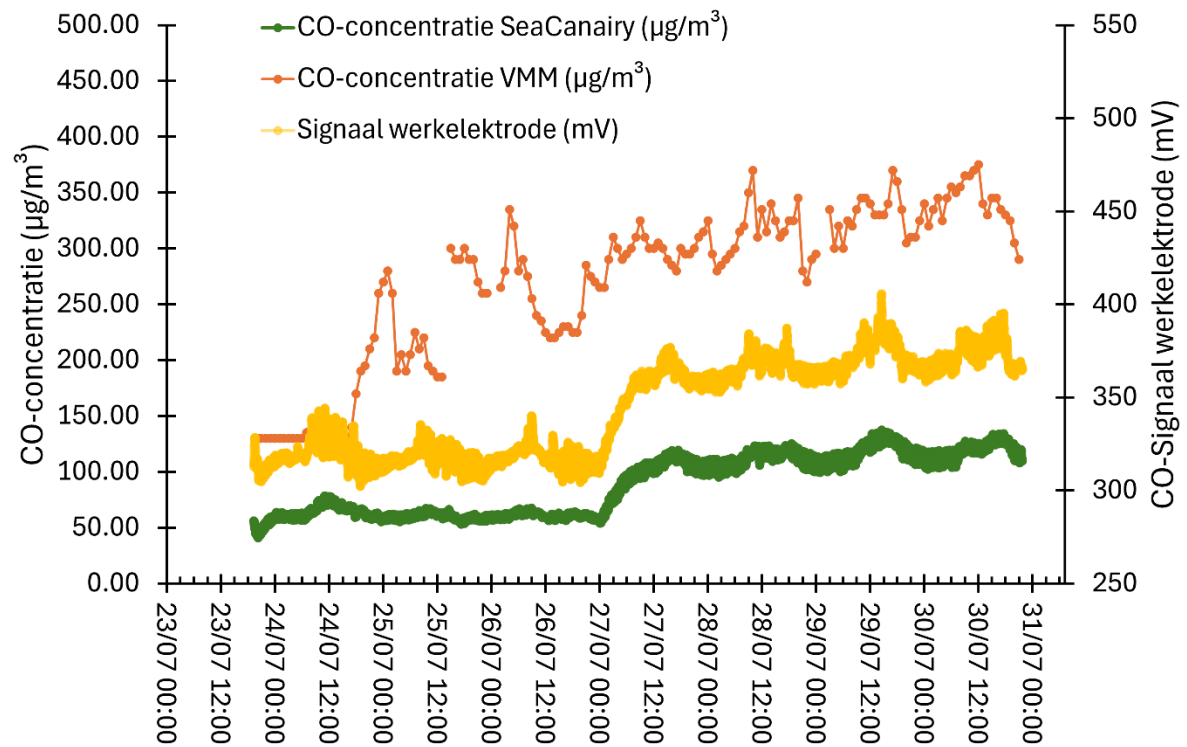
Tabel 15: Referentiepunten voor kalibratie van de SO₂-sensor met VMM-data

	Signaal WE – AE (mV)	Concentratie VMM (mg/m ³)
Minimum	40.83	0.13
Gemiddelde	88.63	0.27



Figuur 46: Relatie tussen sensorspanning en gekalibreerde concentratie voor CO

Bron: Eigen werk



Figuur 47: Vergelijking van geconverteerde CO-concentraties, VMM-referentiedata en de werkelektrode tijdens de meetcampagne

Eigen werk samengevoegd met data van de Vlaamse Milieu Maatschappij [9]

5.2.6 Resultaten van de kalibratie voor de Alphasense-gassensoren

Nu er voor al de sensoren een kalibratie uitgevoerd is geeft dit, zoals in elk hoofdstuk besproken, een omzettingfunctie weer waarmee het signaal in mV naar een concentratie omgezet kan worden.

Tabel 16: Lineaire omzettingfuncties van mV naar concentratie voor de Alphasense-gassensoren

Doelgas	Omzettingfunctie
NO ₂ (µg/m ³)	$y = 0.8301x + 28.169$
O ₃ (µg/m ³)	$y = 11.557x + 635.3$
SO ₂ (µg/m ³)	$y = 0.0177x - 0.2185$
CO (mg/m ³)	$y = 0.0029x + 0.0104$

Hoewel de kalibratie in een later stadium opnieuw moet worden uitgevoerd, aangezien de SeaCanairy niet op dezelfde locatie heeft kunnen meten als de referentiestations, toont dit hoofdstuk reeds hoe een praktische veldkalibratie kan worden uitgevoerd. Ondanks de beperkingen geeft dit proces een bruikbaar inzicht in de relatie tussen de ruwe sensorwaarden en officiële referentiedata. In toekomstig werk kan de methode verder worden verfijnd door de SeaCanairy langdurig naast referentiestations in te zetten, waardoor de nauwkeurigheid en betrouwbaarheid van de kalibratie verder zullen toenemen.

Hoofdstuk 6 Aanvullende analyse van de meetcampagne

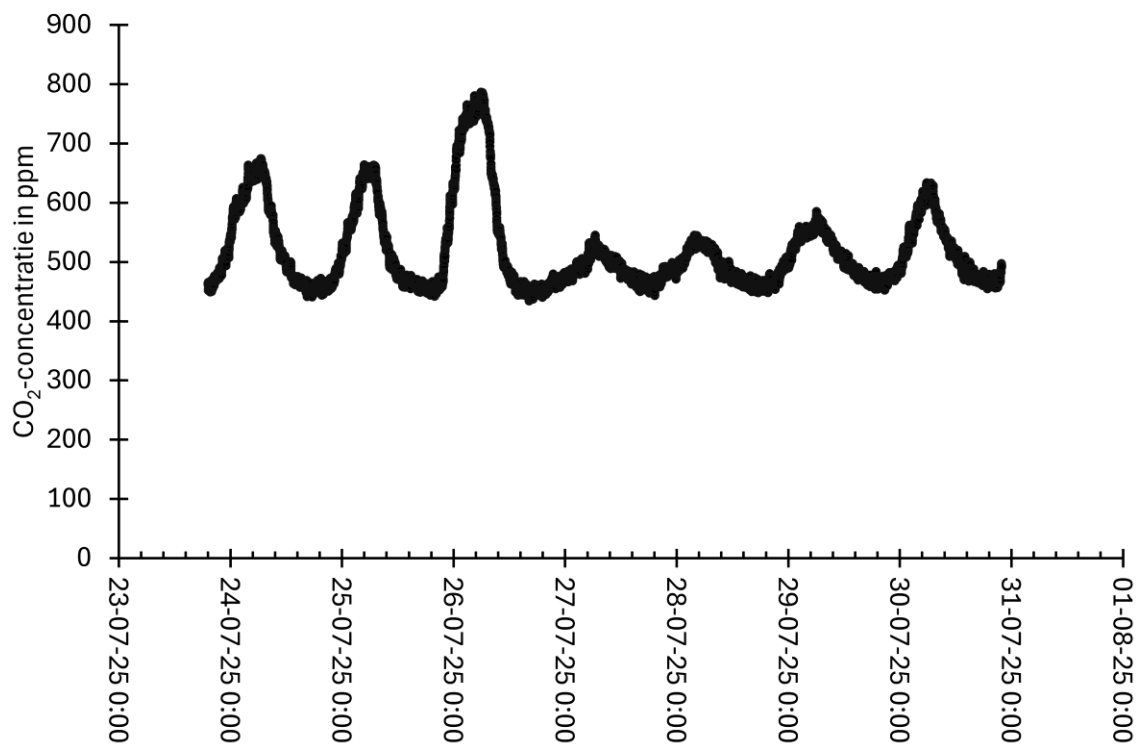
Tijdens de veldkalibratie van de Alphasense-gassensoren werd niet alleen de nodige kalibratiedata verzameld, maar ook bijkomende meetgegevens die extra inzichten bieden in de langdurige werking en dataverwerking van de SeaCanairy. In dit hoofdstuk wordt daarom aandacht besteed aan:

- de prestaties van de CO₂-sensor
- de prestaties van de OPC-N3-sensor.

6.1 De prestaties van de CO₂-sensor

De CO₂-sensor vertoonde aanvankelijk bij circa 50% van de metingen een checksumfout. Om dit te ondervangen herhaalt de software de communicatie tot maximaal zes keer, waardoor alsnog een geldige waarde wordt verkregen. De foutfrequentie kon aanzienlijk worden gereduceerd door een wachttijd van 100 ms in te lassen tussen het verzenden van het commando en het opvragen van de data, en door de communicatiesnelheid te verlagen van 100 kHz naar 50 kHz. In de uiteindelijke dataset van 10229 metingen trad slechts 66 keer een checksumfout op, waarbij de software er telkens in slaagde de juiste CO₂-waarde uit te lezen door de communicatie te herhalen.

Figuur 48 toont de gemeten CO₂-concentraties tijdens de meetcampagne. Hierbij is telkens een duidelijke piek waar te nemen tijdens de nacht en een dal tijdens de dag. Dit patroon wordt verklaard door het feit dat planten tijdens de nacht, wanneer fotosynthese stilvalt, CO₂ blijven uitstoten via respiratie, terwijl ze overdag CO₂ opnemen voor de fotosynthese.



Figuur 48: Gemeten CO₂-waarden door de SeaCanairy tijdens de meetcampagne

Bron: Eigen werk

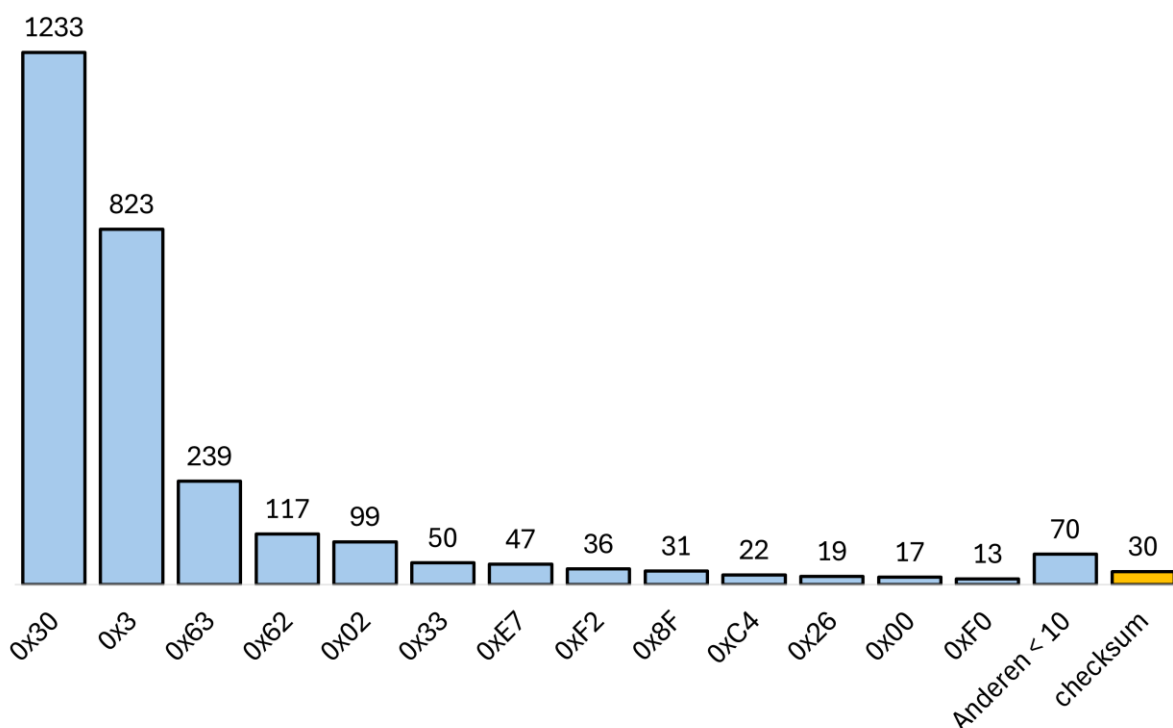
6.2 De prestaties van de OPC-N3-sensor

Er is reeds veel aandacht besteed aan de OPC-N3-sensor, die verantwoordelijk is voor het meten van fijnstof. Uit de meetdata blijkt dat er in totaal effectief 318 keer de fijnstofwaarden niet uitgelezen konden worden. Op 10299 meetpunten geeft dit een foutenpercentage van 3,11%

Bij elke start van een meetcampagne wordt automatisch een logboek aangemaakt waarin alle gebeurtenissen worden geregistreerd. In Bijlage XII

Python-programma voor het tellen en groeperen van logmeldingen staat een Python-programma om deze logmeldingen te tellen en te groeperen. Figuur 49 toont de meest voorkomende fouten tijdens de meetcampagne. Deze fouten doen zich vooral voor tijdens de initialisatiefase van de communicatie tussen de Raspberry Pi en de OPC-N3-sensor. Hieruit blijkt dat van de 2846 communicatiefouten die zich in totaal voordeden, 0x30 het vaakst voorkomt, gevolgd door 0x3. Door observatie is vastgesteld dat 0x3

meestal verschijnt wanneer de Raspberry Pi een tweede poging tot communicatie onderneemt. Na de twee maal een fout antwoord te hebben ontvangen geeft de sensor bij een 3^{de} poging tijdens de initialisatiefase wel een juist antwoord. Verder zijn er 30 checksumfouten vastgesteld, die leidden echter niet tot dataverlies, omdat de software bij een checksumfout automatisch een nieuwe uitleespoging uitvoert totdat geldige data beschikbaar zijn. Dit wil samengevat zeggen dat van de 2846 communicatiefouten die zich voordeden, bij 2528 het probleem werd opgelost door de aanpassingen in de software.

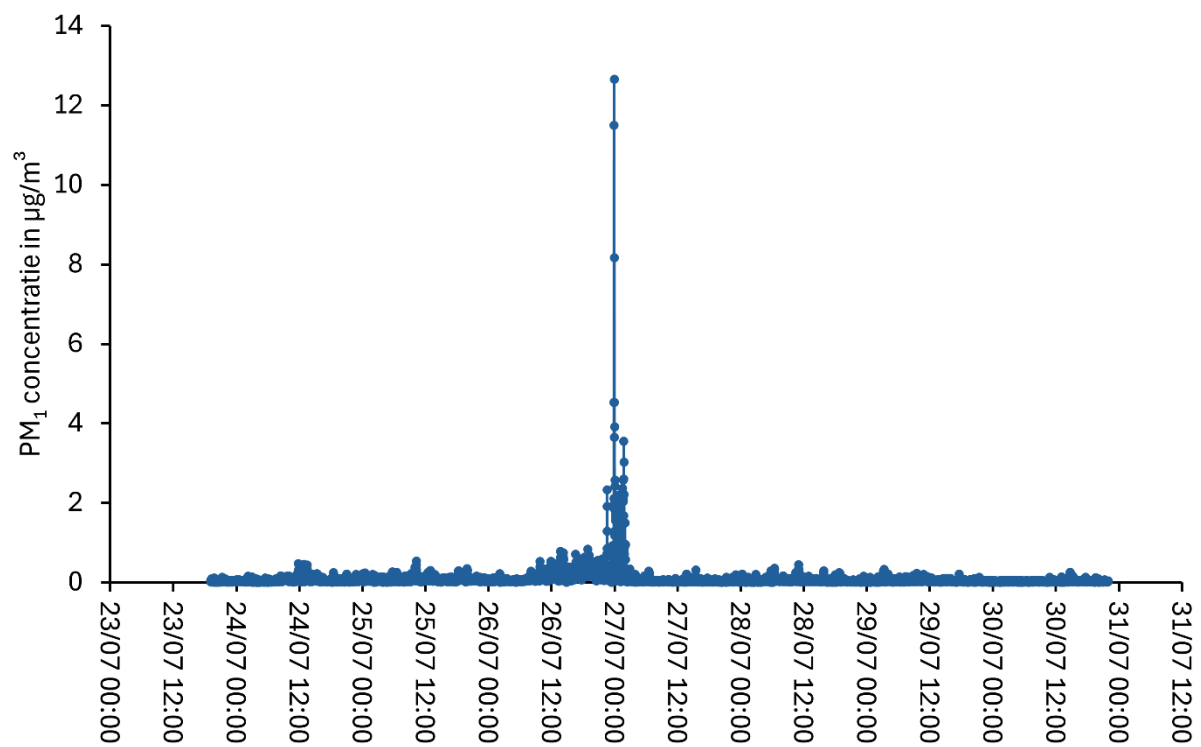


Figuur 49: Overzicht van de foutieve antwoorden van de OPC-N3-sensor tijdens 10299 meetpunten, waarbij de checksumfouten in het geel zijn aangeduid

Bron: Eigen werk

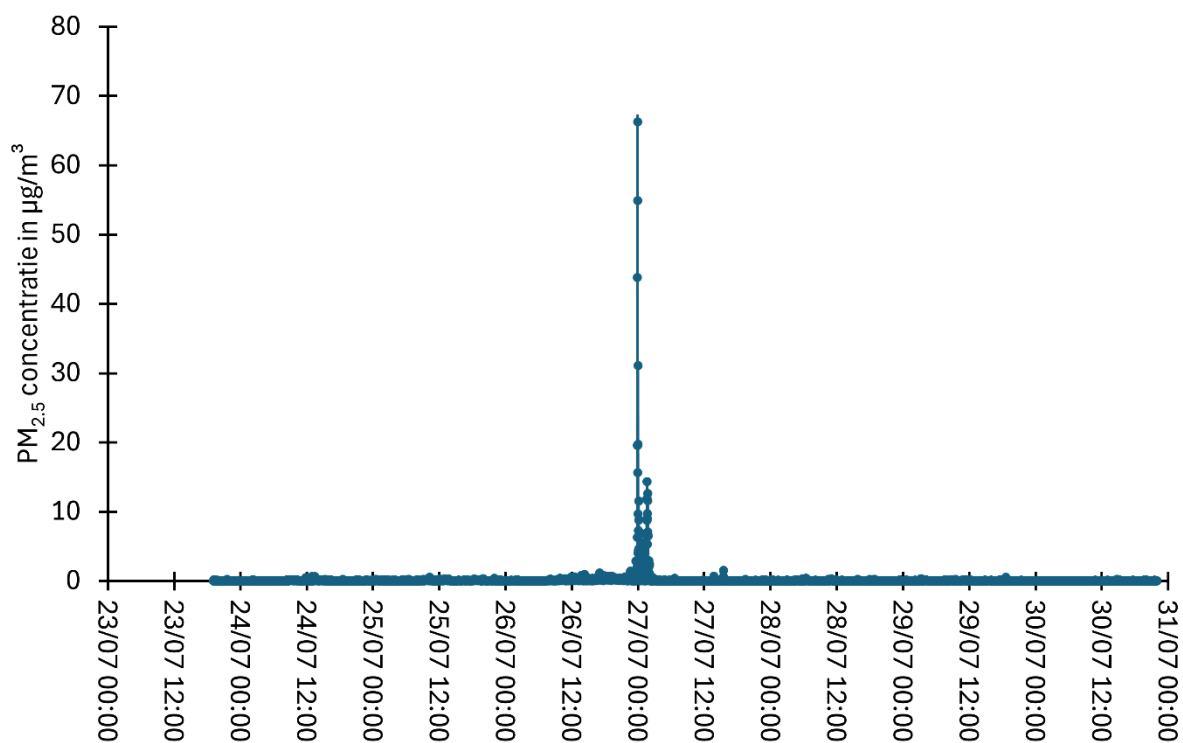
Verder is er ook de testdata zelf beschikbaar. Figuur 50 tot en met 52 tonen de waarden voor respectievelijk PM_1 , $PM_{2.5}$ en PM_{10} met een Hampel-filter met venster 11 en drempelwaarde 3. Er is duidelijk te zien dat er op zaterdag 27 juli omstreeks middernacht een plotse sterke stijging is van het fijnstofgehalte over de 3 verschillende waarden. PM_1 stijgt tot een maximum van $12,66 \mu g/m^3$, $PM_{2.5}$ tot $66,28 \mu g/m^3$ en PM_{10} tot $88 \mu g/m^3$.

Deze plotse stijging is niet waar te nemen op de meetstations van de VMM wat duidt op enerzijds een defect of anderzijds een lokale piek.



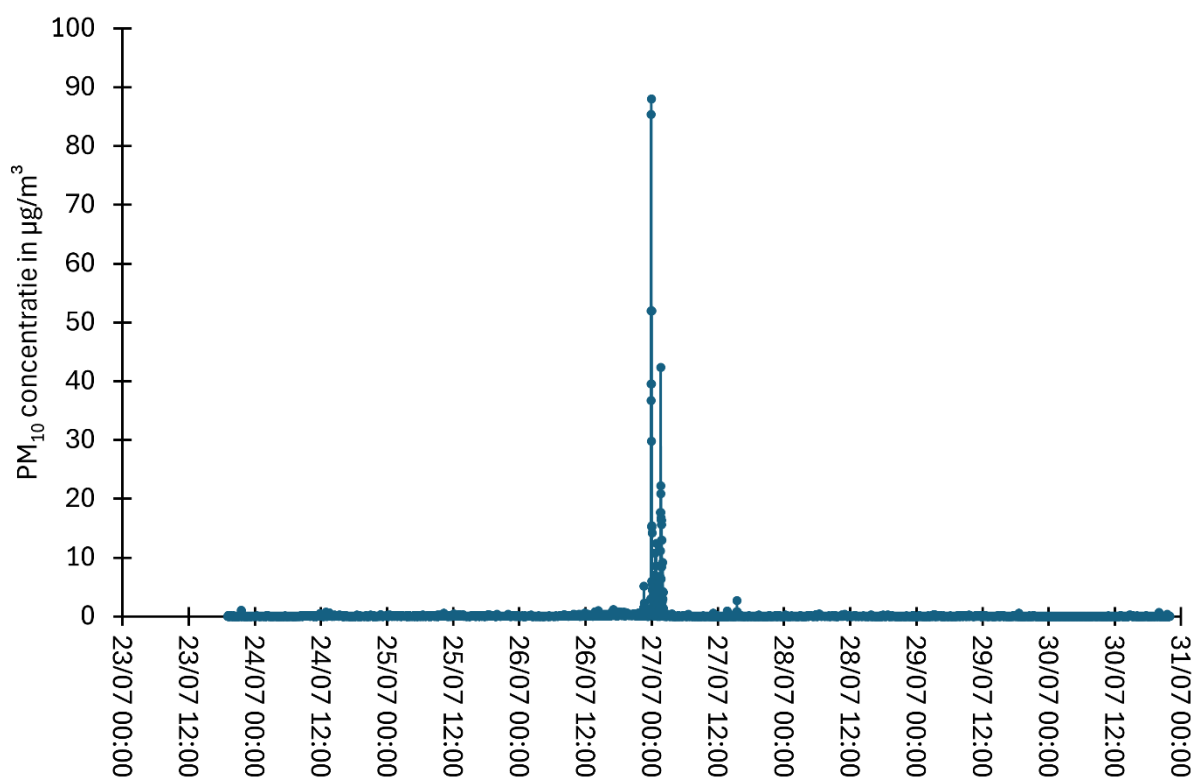
Figuur 50: PM₁-waarden door de SeaCanairy tijdens de meetcampagne

Bron: Eigen werk



Figuur 51: PM_{2.5}-waarden door de SeaCanairy tijdens de meetcampagne

Bron: Eigen werk



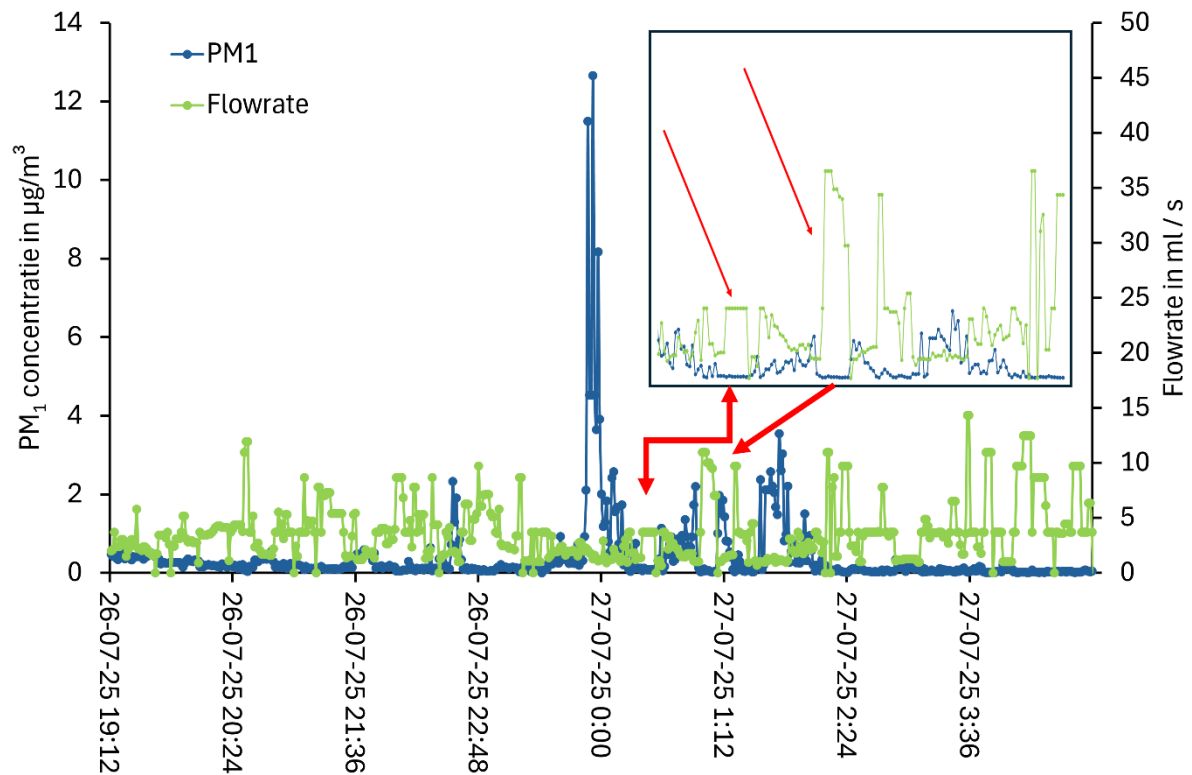
Figuur 52: PM₁₀-waarden door de SeaCanairy tijdens de meetcampagne

Bron: Eigen werk

6.2.1 Relatie tussen debiet en meetwaarden van de OPC-N3-sensor

Om na te gaan of de waargenomen piek het gevolg is van een defect of van een lokaal verschijnsel, geeft Figuur 53 een beter inzicht in de werking van de OPC-N3-sensor. In deze figuur is het debiet (in ml/s), zoals door de OPC-N3-sensor geregistreerd, uitgezet tegenover de gemeten fijnstofconcentratie. Tijdens de piek daalt het debiet merkbaar, maar ook op andere momenten blijkt het debiet niet constant te zijn. Steeds wanneer het debiet daalt, stijgt de gemeten fijnstofconcentratie, het omgekeerde effect is eveneens zichtbaar. Kort na de piek treedt een plotse stijging van het debiet op, wat gepaard gaat met een plotse daling van de fijnstofconcentratie. De grote piek in fijnstofconcentratie is naar alle waarschijnlijkheid het gevolg van een reële lokale verhoging, maar deze gebeurtenis bracht wel een bijkomend fenomeen aan het licht. In de toekomst moet

hiervoor een oplossing gezocht worden om de betrouwbaarheid van de metingen te verhogen.



Figuur 53: PM₁-concentratie en debiet tijdens de gemeten piek tijdens de meetcampagne die een verband weergeeft tussen de waarde van het debiet en de waarde van de gemeten fijnstofconcentratie

Bron: Eigen werk

Hoofdstuk 7 Suggesties voor verdere verbeteringen aan de SeaCanairy

Hoewel in deze thesis zoveel mogelijk oplossingen werden gezocht voor de opgetreden problemen, bleken niet alle knelpunten volledig oplosbaar. Er blijven daarom nog een aantal verbeterpunten over voor de huidige opstelling van de SeaCanairy.

7.1 Verbeteringen met betrekking tot de OPC-N3-sensor

Een belangrijk aandachtspunt blijft het instabiele debiet van de OPC-N3-sensor. Een onregelmatig debiet beïnvloedt rechtstreeks de nauwkeurigheid van de gemeten fijnstofwaarden. In toekomstig onderzoek kan daarom geëxperimenteerd worden met alternatieve pomptypes die een stabiel debiet leveren, of met een regelkring die het debiet actief monitort en corrigeert. Het inzetten van een pomp met ingebouwde flow control of een externe flow controller kan hierbij een mogelijke oplossing zijn.

7.2 Verbeteringen met betrekking tot kalibratie

De kalibratie van de Alphasense-gassensoren in deze thesis dient hoofdzakelijk als referentie en voorbeeld voor toekomstig gebruik. Voor de CO₂-sensor geldt echter dat er nog een geschikt referentiestation gevonden moet worden, aangezien de gebruikte applicatie geen meetstation omvat dat CO₂ registreert. Een koppeling met een officieel referentiestation zou de nauwkeurigheid en betrouwbaarheid van toekomstige kalibraties aanzienlijk verbeteren.

7.3 Verbeteringen met betrekking tot dataverwerking

De huidige software van de SeaCanairy blijft vrij beperkt in functionaliteit. De gebruikersinterface toont enkel de actuele meetwaarden, waardoor afwijkingen of storingen pas later ontdekt worden. Een uitbreiding waarbij de data in realtime in een grafiek wordt weergegeven zou toelaten om tijdens metingen de concentraties direct te volgen en de datakwaliteit beter in te schatten.

Daarnaast is de foutafhandeling van de OPC-N3-sensor voorlopig nog rudimentair. Ongeveer 3% van de meetpunten resulteert in een foutmelding, die pas achteraf via een Pythonprogramma geïnterpoleerd wordt. In de toekomst kan deze interpolatie rechtstreeks in de software geïntegreerd worden, zodat foutieve datapunten automatisch herkend en verwerkt worden. Dit zou zowel de datakwaliteit verbeteren als de verwerkingstijd reduceren.

Tot slot verdient ook het logboek aandacht. Momenteel worden alle gebeurtenissen regel per regel geregistreerd, wat een overzicht bemoeilijkt. Door de loggegevens automatisch te groeperen, bijvoorbeeld door foutcodes te tellen en per sensor samen te vatten, kan in één oogopslag duidelijk worden welke problemen zich voordoen en met welke frequentie. Hierdoor evolueert het logboek van een lineair archief naar een praktisch diagnostisch hulpmiddel.

Conclusie

Deze thesis heeft belangrijke vooruitgang geboekt in de verdere ontwikkeling van de SeaCanairy. Het luchtcircuit werd geoptimaliseerd en het elektrisch schema volledig herontworpen. Daarnaast had deze studie ook als doel om de werking van het systeem inzichtelijker te maken voor gebruikers met minder ervaring in programmeren en sensortechnologie.

Een belangrijke stap vooruit werd gerealiseerd met de kastverwarming, deze bleek een duidelijke positieve invloed te hebben op de metingen van de OPC-N3-sensor, doordat hygroscoische groei succesvol werd beperkt. Ook werden verschillende problemen die tijdens de ontwikkeling opdoken aangepakt, zoals de invloed van de pompconfiguratie op de fijnstofsensor, de noodzaak van een extra I²C-lijn door interferentie met de CO₂-sensor, en de elektromagnetische storingen veroorzaakt door de pomp. Door zowel hardwarematige verbeteringen als softwarematige verbeteringen konden de foutmeldingen van de OPC-N3-sensor sterk worden teruggebracht: van circa 60% naar ongeveer 3%, een resultaat dat aanzienlijke inspanningen vergde.

Toch zijn niet alle doelstellingen bereikt. De kalibratie van de Alphasense-gassensoren moet in de toekomst opnieuw uitgevoerd worden, aangezien door tijdsgebrek geen langdurige vergelijking met referentiestations van de Vlaamse Milieumaatschappij kon plaatsvinden. Bovendien werd op het einde van de testfase nog een bijkomend probleem vastgesteld, het debiet van de OPC-N3-sensor bleek niet stabiel, wat een significante invloed heeft op de gemeten fijnstofconcentraties. Ook dit vraagt in de toekomst om verdere studie en structurele oplossingen.

Ondanks de resterende uitdagingen toont deze thesis dat de SeaCanairy een veelbelovend platform is voor draagbare en betrouwbare luchtkwaliteitsmetingen. Met verdere optimalisaties in kalibratie, debietstabilisatie en software kan het systeem robuuster en nauwkeuriger worden, wat de inzetbaarheid in langdurige meetcampagnes en maritieme toepassingen vergroot.

Bibliografie

- [1] 3105360 *Kastverwarming zonder ventilator*. (z.d.). 3105360. Geraadpleegd 18 juni 2025, van <https://www.rittal.com/be-nl/products/PG20231215KLI101/PG20231245KLI1006/PRO0307?variantId=3105360>
- [2] 3110000 *Thermostaat Kastthermostaat*. (z.d.). 3110000. Geraadpleegd 18 juni 2025, van <https://www.rittal.com/be-nl/products/PG20231215KLI101/PG20231215KLI1008/PG20240123KLI204/PRO12215?variantId=3110000>
- [3] A9F74202—*Modulaire automaat—iC60N - 2 polen—2 A - C curve* | *Schneider Electric België*. (z.d.). Geraadpleegd 2 juni 2025, van <https://www.se.com/be/nl/product/A9F74202/modulaire-automat-ic60n-2-polen-2-a-c-curve/>
- [4] *Alphasense_opc-n3_datasheet_en_1.pdf*. (z.d.). Geraadpleegd 22 april 2025, van https://ametekcdn.azureedge.net/mediafiles/project/oneweb/oneweb/alphasense/products/datasheets/alphasense_opc-n3_datasheet_en_1.pdf?revision:29541b07-612a-42ba-b362-f41a48cf2e48
- [5] Baguley, D. A., Evans, G. S., Bard, D., Monks, P. S., & Cordell, R. L. (2025). Review of volatile organic compound (VOC) emissions from desktop 3D printers and associated health implications. *Journal of Exposure Science & Environmental Epidemiology*, 1-18. <https://doi.org/10.1038/s41370-025-00778-y>
- [6] Chih-Sheng Chen, Yung-Hui Liu, Cheng Ying Kuo, & Yun-Liang Chu. (2012). *REDUCTION OF CONDUCTIVE EMI NOISE RESULTED FROM THE COMMERCIAL POWER SUPPLY*. IEEE.
- [7] *Datasheet_SHT3x_DIS.pdf*. (z.d.). Geraadpleegd 22 april 2025, van https://sensirion.com/media/documents/213E6A3B/63A5A569/Datasheet_SHT3x_DIS.pdf
- [8] Dewez, C. (2021). *Esigning a portable and autonomous air pollution measuring instrument*. Antwerp Maritime Academy.
- [9] *Interactive viewer—English*. (z.d.). [Page]. Geraadpleegd 19 augustus 2025, van <https://www.irceline.be/en/air-quality/measurements/monitoring-stations>
- [10] Kunić, Z., Mršić, L., Đambić, G., & Ražov, T. (2024). Innovative Air-Preconditioning Method for Accurate Particulate Matter Sensing in Humid Environments. *Sensors*, 24(17), 5477. <https://doi.org/10.3390/s24175477>

- [11] Lukas Van Der borcht. (2019). *Constructie en kalibratie van een toestel voor het meten van luchtkwaliteit aan boord*. Hogere Zeevaartshool Antwerpen.
- [12] M9F21102—Miniature circuit breaker (MCB), Multi9 C60SP, 1P, 2A, B curve, 10kA (UL1077), 10kA (IEC/EN 60947-2) | Schneider Electric Belgium. (z.d.). Geraadpleegd 2 juni 2025, van <https://www.se.com/be/en/product/M9F21102/miniature-circuit-breaker-mcb-multi9-c60sp-1p-2a-b-curve-10ka-ul1077-10ka-iec-en-609472/>
- [13] PTIO 1,5/S/4—Initiator/actuator terminal block—3244452 | Phoenix Contact. (z.d.). Geraadpleegd 19 augustus 2025, van <https://www.phoenixcontact.com/en-be/products/sensor-actuator-terminal-block-ptio-15-s-4-3244452>
- [14] Savitzky, Abraham., & Golay, M. J. E. (1964). Smoothing and Differentiation of Data by Simplified Least Squares Procedures. *Analytical Chemistry*, 36(8), 1627-1639. <https://doi.org/10.1021/ac60214a047>
- [15] *What are the WHO Air quality guidelines?* (z.d.). Geraadpleegd 16 juni 2025, van <https://www.who.int/news-room/feature-stories/detail/what-are-the-who-air-quality-guidelines>
- [16] *What does B,C,D curve mean for Acti 9 MCB?* | Schneider Electric Australia. (2016, juni 6). WwW.Se.Com. Geraadpleegd 22 april 2025, van <https://www.se.com/au/en/faqs/FA290880/>

Lijst van bijlagen

Bijlage I Foutenrapport

Bijlage II Elektrisch diagram van de SeaCanairy

Bijlage III Code en toelichting voor de SHT30-buitentemperatuursensor

Bijlage IV Het configuratiebestand van de Raspberry Pi

Bijlage V 3D-ontwerp voor de ingang van de OPC-N3-sensor in OpenSCAD

Bijlage VI 3D-ontwerp voor de uitgang van de OPC-N3-sensor in OpenSCAD

Bijlage VII 3D-ontwerp voor het bovenste stuk van de CO₂-sensor in OpenSCAD

Bijlage VIII 3D-ontwerp voor het middenste stuk van de CO₂-sensor in OpenSCAD

Bijlage IX 3D-ontwerp voor het onderste stuk van de CO₂-sensor in OpenSCAD

Bijlage X Aanvullend experiment rond verwarming van fijnstof en hygroscopische groei

Bijlage XI Python-programma om de gassensoren van Alphasense om te zetten naar $\mu\text{g}/\text{m}^3$

Bijlage XII Python-programma voor het tellen en groeperen van logmeldingen

Bijlage XIII Overzicht van de SeaCanairy-software

Bijlage XIV De aangepaste SeaCanairy.py-code

Bijlage XV De aangepaste OPCN3.py-code

Bijlage XVI De aangepaste CO₂.py-code

Bijlage I

Foutenrapport

1. Probleemstelling: thermische uitschakeling van zekering

De zekering schakelt onverwacht uit, vermoedelijk als gevolg van te hoge temperaturen veroorzaakt door de interne kastverwarming.

Oplossing:

Volgens de datasheet bedraagt de maximale werkingstemperatuur van de gebruikte zekering 70 °C. Door de afstand tussen de zekering en het verwarmingselement te vergroten, of door de temperatuur op de thermostaat te verlagen, kan thermische oververhitting worden vermeden.

2. Probleemstelling: I²C-conflict tussen CO₂- en een ander I²C apparaat

De communicatie met de E+E Elektronik EE894 CO₂-sensor valt volledig weg wanneer de SHT30-sensor wordt toegevoegd aan de standaard I²C-bus.

Oplossing:

Er werd een aparte I²C-bus gerealiseerd via software op GPIO-pinnen 5 (SDA) en 6 (SCL). Hierdoor is de SHT30 gescheiden van de overige sensoren, wat I²C-busconflicten voorkomt en de stabiliteit ten goede komt.

3. Probleemstelling: Foutmeldingen en checksumproblemen bij E+E Elektronik EE894 CO₂-sensor

De CO₂-sensor genereert regelmatig foutmeldingen, met name checksumfouten bij het uitlezen van meetwaarden.

Oplossing:

Door een wachttijd van 1 ms in te bouwen tussen het verzenden van het commando en het uitlezen van de response, werd de betrouwbaarheid van de communicatie verbeterd. Daarnaast werd de I²C-kloksnelheid verlaagd van 100 kHz naar 50 kHz,

wat het aantal transmissiefouten aanzienlijk verminderde. Ook de fysieke bekabeling werd aangepast: een te korte, gesoldeerde kabel werd vervangen door een langere, soepelere verbinding, wat bijdroeg aan een stabielere datacommunicatie.

4. Probleemstelling: Beperkte toegangsrechten tot automatisch aangemaakte projectmappen

Telkens wanneer een nieuw project wordt opgestart door de projectnaam te wijzigen in het bestand `seacanairy_settings.yaml`, worden automatisch nieuwe submappen aangemaakt. Deze mappen krijgen standaard beperkte toegangsrechten, waardoor de standaardgebruiker geen lees- of schrijfrechten heeft. Dit verhindert onder andere het openen, kopiëren of exporteren van logbestanden zonder tussenkomst van rootrechten.

Oplossing:

Om dit probleem te verhelpen, dient de operator in de Linux-terminal het volgende commando uit te voeren. Hiermee worden de bestandsrechten aangepast zodat de normale gebruiker volledige toegang krijgt tot de bestanden en submappen:

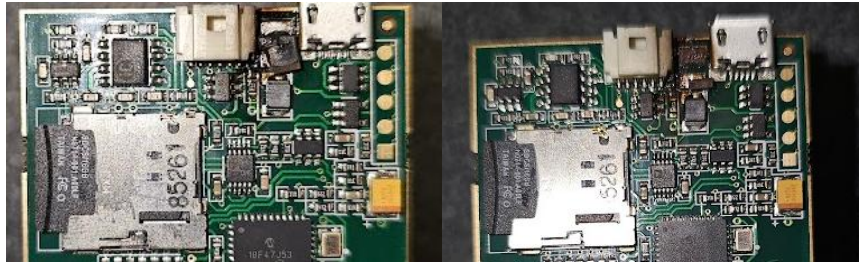
```
sudo chmod -R 777 /home/pi/seacanairy_project/data/
```

5. Probleemstelling: Kortsluiting op de printplaat van de OPC-N3 sensor

Tijdens het testen van de OPC-N3-sensor van Alphasense met een nieuwe datakabel is er een kortsluiting ontstaan in de sensor zelf. Tijdens het openen van de behuizing is het duidelijk zichtbaar dat tussen de USB poort en de molex connector een kortsluiting was. Het component is hoogstwaarschijnlijk een condensator.

Oplossing:

Door dit component weg te solderen van de PCB plaat en met een multimeter te verzekeren dat er tussen de 5V en grond geen verbinding meer was zonder weerstand functioneert de sensor weer. Onderstaande figuur geeft links de kortsluiting weer en rechts met het elektrisch component verwijderd.



Figuur 54: Doorgebrand elektrisch component op de PCB plaat van de OPC-N3-fijnstofsensor

Bron: Eigen werk

6. Probleemstelling: stroomproblemen op het wisselstroomnetwerk

Er is ogenschijnlijk geen stroom meer en de voedingen krijgen geen stroom meer.

Oplossing:

Haal de SeaCanairy van het lichtnet en kijk met een multimeter of er verbinding is tussen de terminal en de stekker. Als dit niet het geval is dient met te kijken naar waar de stroom de koffer binnengaat en vervolgens kan met de draden uit de terminal halen om vervolgens opnieuw in de terminal te bevestigen. Dit heeft in het verleden al een aantal keer gewerkt. Let op dat de SeaCanairy niet onder spanning mag staan tijdens deze werkzaamheden.

7. Probleemstelling: elektrisch geluid en uitvallen van de SeaCanairy bij het bewegen van de voedingskabel

Bij het bewegen van de voedingskabel was een elektrisch geruis hoorbaar, waarna de SeaCanairy sporadisch volledig uitviel.

Oplossing:

Een interne inspectie van de voedingskabel wees uit dat de stroomdraden losgekomen waren uit hun bevestiging. Daarnaast waren er sporen zichtbaar die wijzen op een eerdere kortsluiting. Aangezien de voedingskabel gedurende het verloop van deze thesis niet werd aangepast of gemanipuleerd, wordt aangenomen dat het defect reeds voorafgaand aanwezig was. Het herbevestigen van de drie interne geleiders heeft het probleem met het elektrische geruis en het onverwachte uitvallen opgelost.



Figuur 55: Brandplek in de rode cirkel te zien aan de terminal van de voedingskabel wat op een eerdere kortsluiting kan wijzen.

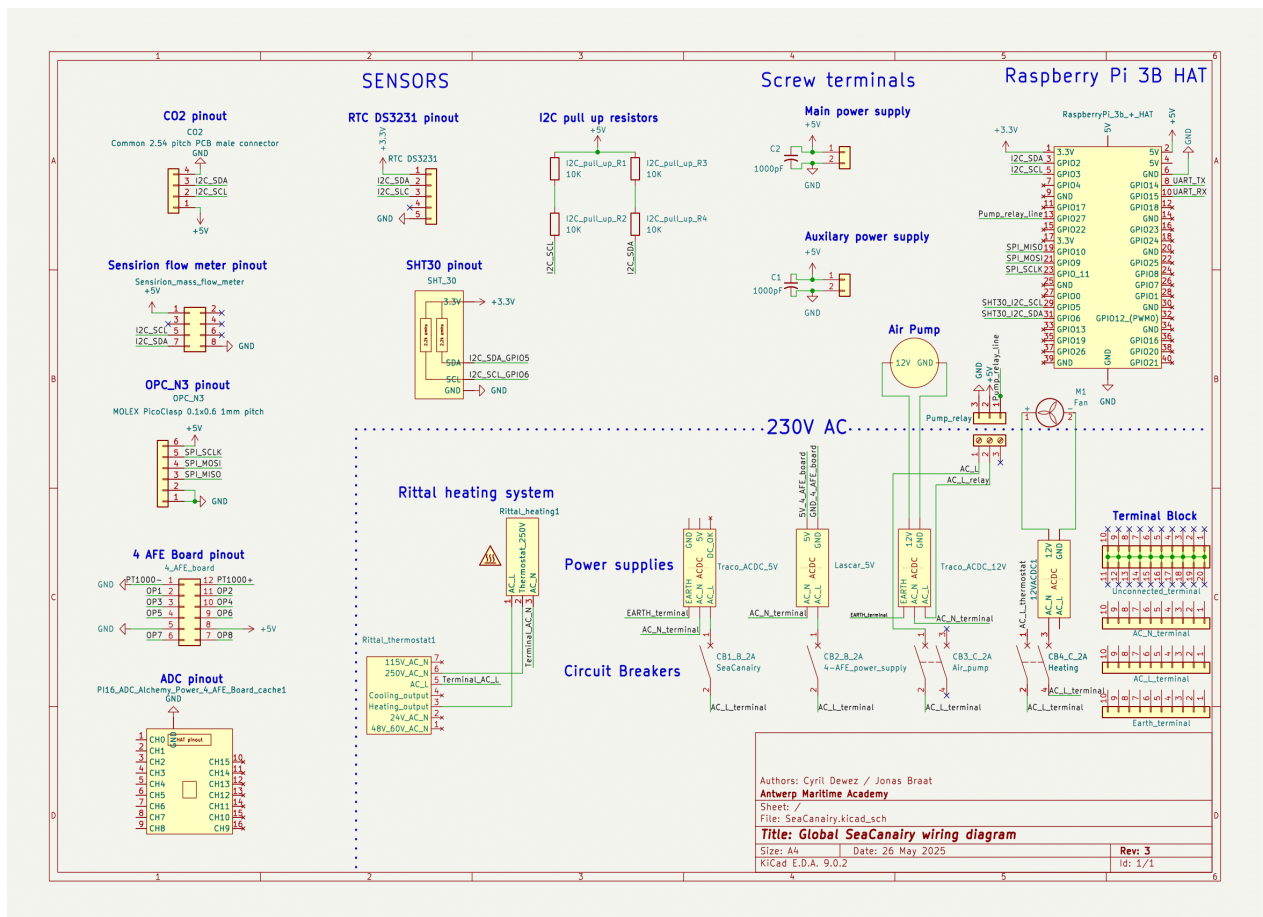
Bron: Eigen werk

8. Probleemstelling: Het debiet van de OPC-N3-sensor is niet constant wat tot fluctuaties leidt van de meetresultaten

Het debiet van de OPC-N3-sensor is niet constant. Bij dalingen in het debiet worden hogere fijnstofconcentraties geregistreerd, terwijl bij een plots stijgend debiet de gemeten concentraties kunstmatig dalen. Deze variaties verstoren de nauwkeurigheid en consistentie van de meetresultaten.

Bijlage II

Elektrisch Diagram van de SeaCanairy



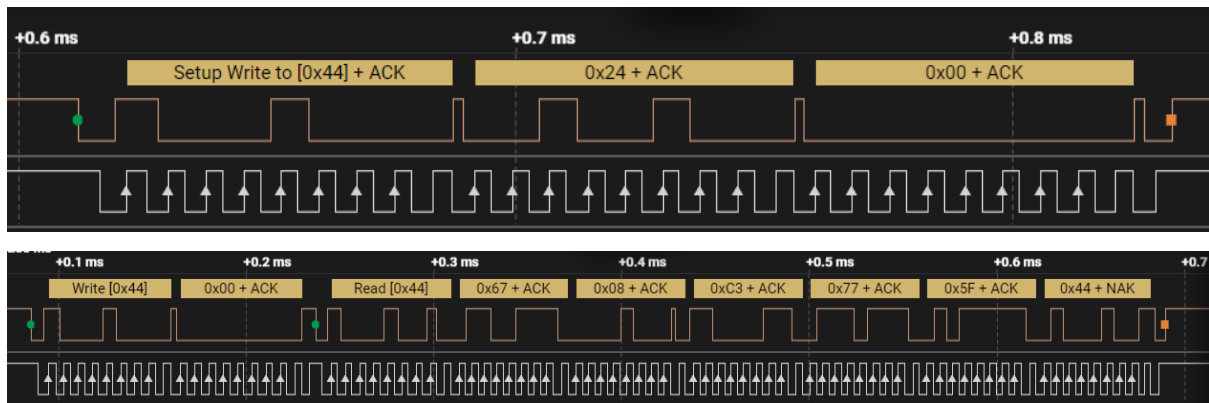
Voor het diagram op A4 grootte, zie volgende pagina.

Bijlage III

Code en toelichting voor de SHT30 buitentemperatuursensor

```
# SHT30.py
import smbus2
import time
# Initialiseer I2C bus 11
bus = smbus2.SMBus(11)
SHT30_ADDR = 0x44 # I2C-adres van de SHT30-sensor
def read_sht30():
    try:
        # Start een enkele meting door een commando te sturen
        bus.write_i2c_block_data(0x44, 0x24, [0x00])
        time.sleep(0.015) # Wacht minimaal 15 ms voor sensorrespons
        # Lees 6 bytes: 2 temperatuur, 1 CRC, 2 vochtigheid, 1 CRC,
        data = bus.read_i2c_block_data(0x44, 0x00, 6)
        # Combineer temperatuurbytes tot 16-bits ruwe waarde
        temp_raw = (data[0] << 8) | data[1]
        # Combineer vochtigheidsbytes tot 16-bits ruwe waarde
        humidity_raw = (data[3] << 8) | data[4]
        #Byte 1 (MSB): 01100110 → Shifted left 8 bits → 01100110 00000000
        #Byte 2 (LSB): 11001100
        #combineer beide bytes:
        #01100110 00000000 | 11001100 → 01100110 11001100
        # Reken ruwe waarden om naar fysische grootheden
        temperature = -45 + 175 * (temp_raw / 65535.0)
        humidity = 100 * (humidity_raw / 65535.0)
        return {
            "temperature": temperature,
            "humidity": humidity
        }
    except OSError as e:
        # Foutafhandeling: sensor is mogelijk niet aangesloten of
        onbereikbaar zonder dat het hoofdprogramma vastloopt
        print(f"SHT30 sensor error: {e}")
```

```
return {  
    "temperature": None,  
    "humidity": None }
```



Bovenaan stuurt de Raspberry Pi het adres 0x44 gevolgd door de het commando 0x24 0x00. Onderaan stuurt de Raspberry Pi het adres opnieuw gevolgd door 0x00.

De sensor antwoordt vervolgens met zes bytes in de volgende volgorde:

0x67, 0x08, 0xC3, 0x77, 0x5F, 0x44.

- Byte 1 en 2 (0x67, 0x08) bevatten de ruwe temperatuurwaarde
- Byte 3 (0xC3) is een checksum voor de temperatuur
- Byte 4 en 5 (0x77, 0x5F) bevatten de ruwe luchtvochtigheidswaarde
- Byte 6 (0x44) is een checksum voor de vochtigheid

Aangezien de huidige Pythonimplementatie geen CRC-controle toepast, worden enkel de temperatuur- en vochtigheidsbytes (byte 1, 2, 4 en 5) verder verwerkt in de dataconversie naar fysische eenheden.

Om de temperatuur te berekenen worden byte 1 en 2 samengevoegd tot een 16-bits waarde. In dit voorbeeld vormen 0x67 en 0x08 samen het getal 0x6708, wat overeenkomt met de decimale waarde 26376. Deze ruwe waarde wordt vervolgens ingevoerd in de conversieformule uit het datasheet:

$$T = -45 + \frac{175 \times 26376}{65535} = 25,43^{\circ}\text{C}$$

Voor de relatieve luchtvochtigheid kan hetzelfde gedaan worden met bytes 4 en 5 die samen de heximale waarde 0x775F geven. In decimale waarde geeft dit 30559.

$$RH = \frac{100 \times 30559}{65535} = 46,63\%$$

Bijlage IV

Het configuratiebestand van de Raspberry Pi

```
# For more options and information see
# http://rpf.io/configtxt
# Some settings may impact device functionality. See link above for details
# uncomment if you get no picture on HDMI for a default "safe" mode
#hdmi_safe=1
# uncomment this if your display has a black border of unused pixels visible
# and your display can output without overscan
disable_overscan=1
# uncomment the following to adjust overscan. Use positive numbers if console
# goes off screen, and negative if there is too much border
#overscan_left=16
#overscan_right=16
#overscan_top=16
#overscan_bottom=16
# uncomment to force a console size. By default it will be display's size
# minus
# overscan.
#framebuffer_width=1280
#framebuffer_height=720
# uncomment if hdmi display is not detected and composite is being output
#hdmi_force_hotplug=1
# uncomment to force a specific HDMI mode (this will force VGA)
#hdmi_group=2
#hdmi_mode=4
# uncomment to force a HDMI mode rather than DVI. This can make audio work
in
```

```

# DMT (computer monitor) modes
#hdmi_drive=2
# uncomment to increase signal to HDMI, if you have interference, blanking,
or
# no display
config_hdmi_boost=4
# uncomment for composite PAL
#sdtv_mode=2
#uncomment to overclock the arm. 700 MHz is the default.
#arm_freq=800
# Uncomment some or all of these to enable the optional hardware interfaces
dtparam=i2c_arm=on
#dtparam=i2s=on
dtparam=spi=on
# Uncomment this to enable infrared communication.
#dtoverlay=gpio-ir,gpio_pin=17
#dtoverlay=gpio-ir-tx,gpio_pin=18
# Additional overlays and parameters are documented /boot/overlays/README
# Enable audio (loads snd_bcm2835)
dtparam=audio=0

[pi4]
# Enable DRM VC4 V3D driver on top of the dispmanx display stack
dtoverlay=vc4-fkms-v3d
max_framebuffers=2

[all]
#dtoverlay=vc4-fkms-v3d
# NOOBS Auto-generated Settings:
# 1-wire settings
dtoverlay=w1-gpio
max_usb_current=1
hdmi_group=2
hdmi_mode=87
hdmi_cvt 1024 600 60 6 0 0 0
hdmi_drive=1

```

```
enable_uart=1
# Added by Cyril Dewez on the 07-04-21
# Disable the Bluetooth to free the PL011 UART port for use with the
# Seacanairy GPS receiver
dtoverlay=pi3-disable-bt
# Added by Cyril Dewez on the 09/05/21
# Add the RTC to the system
dtoverlay=i2c-rtc,ds3231
# Added by Jonas Braat on 19/03/25
# Create an extra I2C lane for the SHT30 sensor
dtoverlay=i2c-gpio,bus=3,i2c_gpio_sda=6,i2c_gpio_scl=5
```

Bijlage V

3D-ontwerp voor de ingang van de OPC-N3-sensor in OpenSCAD

```
// Entrance for OPC-N3

length = 130;

width = 95;

height = 20;


corner_hole_radius = 2;

corner_offset = 10; // Offset of the corner holes from the edges


cutout1_length = 90;

cutout1_width = 20;

cutout1_depth = 25;

cutout1_position_x = 20;

cutout1_position_y = width / 2 - cutout1_width / 2; // Center the first
cutout in the y direction


difference() {

    // The main rectangle

    cube([length, width, height]);
```

```

// Corner holes
for (x_offset = [corner_offset, length - corner_offset]) {
    for (y_offset = [corner_offset, width - corner_offset]) {
        translate([x_offset, y_offset, 0])
            cylinder(h = height, r = corner_hole_radius, $fn=20);
    }
}

// square cutout
translate([cutout1_position_x, cutout1_position_y, 0])
    cube([cutout1_length, cutout1_width, cutout1_depth]);
}

translate([length-0.8, width / 2, height / 2]) // Position the text on the
Y-Z axis
rotate([90, 0, 90]) // Rotate the text to align with the Y-Z plane
linear_extrude(height = 2)
    text("SeaCanairy", size = 8, valign = "center", halign = "center");

```

Bijlage VI

3D-ontwerp voor de uitgang van de OPC-N3-sensor in OpenSCAD

```
// Define the dimensions
```

```
length = 130; // Length of the rectangle in mm (13 cm)
```

```
width = 95; // Width of the rectangle in mm (9.5 cm)
```

```
height = 21; // Height of the rectangle in mm (2 cm)
```

```
corner_hole_radius = 2; // Radius of the corner holes in mm
```

```
corner_offset = 10; // Offset of the corner holes from the edges in mm (1 cm)
```

```
cutout1_length = 32; // Length of the first square cutout in mm (3 cm)
```

```
cutout1_width = 32; // Width of the first square cutout in mm (3 cm)
```

```
cutout1_depth = 15; // Depth of the first square cutout in mm (1 cm)
```

```
cutout1_position_x = 70; // Position of the first cutout in the x direction  
in mm (7 cm)
```

```
cutout1_position_y = width / 2 - cutout1_width / 2; // Center the first  
cutout in the y direction
```

```
cutout2_length = 55; // Length of the second square cutout in mm (4.5 cm)
```

```
cutout2_width = 55; // Width of the second square cutout in mm (4.5 cm)
```

```
cutout2_depth = 25; // Depth of the second square cutout in mm (1 cm)
```

```

cutout2_position_x = 0; // Position of the second cutout in the x direction
(from the bottom edge)

cutout2_position_y = 20; // Position of the second cutout from the bottom
in the y direction


cutout_tube_length = 30;

cutout_tube_width = 5.5;

cutout_tube_depth = 15;

cutout_tube_position_x = 50;

cutout_tube_position_y = 47.5;

cutout_tube_position_z = 12;


cutout_tube2_length = 10;

cutout_tube2_width = 8;

cutout_tube2_depth = 15;

cutout_tube2_position_x = 50;

cutout_tube2_position_y = 47.5;

cutout_tube2_position_z = 12;


// Create the rectangular prism with corner holes and cutouts
difference() {

    // The main rectangular prism

    cube([length, width, height]);


    // Corner holes

    for (x_offset = [corner_offset, length - corner_offset]) {

```

```

    for (y_offset = [corner_offset, width - corner_offset]) {
        translate([x_offset, y_offset, -10])
            cylinder(h = height + 20, r = corner_hole_radius, $fn=20);
    }
}

// First square cutout
translate([cutout1_position_x, cutout1_position_y, 0])
    cube([cutout1_length, cutout1_width, cutout1_depth]);

// Second square cutout from the bottom
translate([cutout2_position_x, cutout2_position_y, 0])
    cube([cutout2_length, cutout2_width, cutout2_depth]);

// Cutout tube oriented along the x-axis
    translate([cutout_tube_position_x,      cutout_tube_position_y,
cutout_tube_position_z])
        rotate([0, 90, 0]) // Rotate the cylinder 90 degrees about the y-
axis
        cylinder(h = cutout_tube_length, r = cutout_tube_width / 2, $fn=20);

// Cutout tube2 oriented along the x-axis
    translate([cutout_tube2_position_x,      cutout_tube2_position_y,
cutout_tube2_position_z])
        rotate([0, 90, 0]) // Rotate the cylinder 90 degrees about the y-
axis

```

```
        cylinder(h = cutout_tube2_length, r = cutout_tube2_width / 2,  
$fn=20);  
}
```

Bijlage VII

3D-ontwerp voor het bovenste stuk van de CO₂ sensor in OpenSCAD

```
//CO2 upper part

// dimensions cube

length = 40;

width = 40;

height = 15;

// dimension cylinder

cylinder_radius = 13;

cylinder_height = 8;


//holder_cutout

holder_cutout_thickness = 2.5;

holder_cutout_width = 23;

holder_cutout_height = cylinder_height;


// corner_holes

corner_hole_radius = 2;

corner_offset = 4.5;

difference()

{
```

```

// the main rectangular prism
cube([length, width, height]);

{
// corner holes
for (x_offset = [corner_offset, length - corner_offset])
    for (y_offset = [corner_offset, width - corner_offset]) {
        translate([x_offset, y_offset, -10])
            cylinder(h = height + 20, r = corner_hole_radius, $fn=50);
    }
}

// cylinder_cutout
translate([21.5, 21.5,0])
cylinder(cylinder_height, r=cylinder_radius, $fn=50);

//holder_cutout
translate([38, 21.5, 0])
rotate([0, 0, 134])
    cube([holder_cutout_width, holder_cutout_thickness,
holder_cutout_height]);
}

translate([20, 28, 14.5])
linear_extrude(1.0)

```

```
text("ANTWERP", size=5, valign="center", halign="center");  
translate([20, 21, 14.5])  
linear_extrude(1.0)  
text("MARITIME", size=5, valign="center", halign="center");  
translate([20, 13, 14.5])  
linear_extrude(1.0)  
text("ACADEMY", size=5, valign="center", halign="center");
```

Bijlage VIII

3D-ontwerp voor het middenste stuk van de CO₂ sensor in OpenSCAD

```
//CO2 middle part

// dimensions cube

length = 40;

width = 40;

height = 36;

// dimension cylinder

cylinder_radius = 13;

cylinder_height = 40;

// connection tube

connection_radius = 5.5;

connection_height = 30;

// bolt_cutout

bolt_radius = 11;

bolt_height = 8;
```

```

//holder_cutout

holder_cutout_thickness  = 2.5;

holder_cutout_width = 23;

holder_cutout_height = height;


// corner holes

corner_hole_radius = 2;

corner_offset = 4.5;

difference()
{
    // the main rectangular prism
    cube([length, width, height]);
    {
        // corner holes
        for (x_offset = [corner_offset, length - corner_offset])
            for (y_offset = [corner_offset, width - corner_offset]) {
                translate([x_offset, y_offset, -10])
                    cylinder(h = height + 20, r = corner_hole_radius, $fn=50);
            }
    }

    // cylinder_cutout
    translate([21.5, 21.5, 0])
        cylinder(cylinder_height, r=cylinder_radius, $fn=50);

    // connection_flowsensor // connection to flow meter

```

```

translate([23, 15, 16])
rotate([90, 0, 0])
cylinder(h = connection_height, r=connection_radius, $fn=50);

//bolt_flowsensor
translate([23, 15, 16])
rotate([90, 0, 0])
cylinder(h = bolt_height, r = bolt_radius);

// connection_alphasense // connection to gas sensors
translate([0, 23, 21.5])
rotate([0, 90, 0])
cylinder(h = connection_height, r=connection_radius, $fn=50);

//bolt_alphasense
translate([8, 23, 21.5])
rotate([0, 90, 0])
cylinder(h = bolt_height, r = bolt_radius);

//holder_cutout
translate([38, 21.5, 0])
rotate([0, 0, 134])
        cube([holder_cutout_width,        holder_cutout_thickness,
holder_cutout_height]);
}

```

Bijlage IX

3D-ontwerp voor het onderste stuk van de CO₂ sensor in OpenSCAD

```
//CO2 lower part
```

```
// dimensions cube
```

```
length = 40;
```

```
width = 40;
```

```
height = 9;
```

```
// dimension cylinder
```

```
cylinder_radius = 13;
```

```
cylinder_height = 11;
```

```
//dimensions cube cutout
```

```
cube_cutout_length = 15;
```

```
cube_cutout_width = 15;
```

```
cube_cutout_height = 16;
```

```
//wire_cutout
```

```
wire_length = 6;
```

```
wire_width = 15;
```

```

wire_depth = 0.7;

//holder_cutout
holder_cutout_thickness = 2.5;
holder_cutout_width = 23;
holder_cutout_height = height;

// corner_holes
corner_hole_radius = 2;
corner_offset = 4.5;

//corner_holes_bolts
corner_holes_bolt_length = 4;
corner_holes_bolt_depth = 2.2;

difference()
{
    // the main rectangular prism
    cube([length, width, height]);
    {
        // corner holes
        for (x_offset = [corner_offset, length - corner_offset])
            for (y_offset = [corner_offset, width - corner_offset]) {
                translate([x_offset, y_offset, -10])
                    cylinder(h = height + 20, r = corner_hole_radius, $fn=50);
                // corner hole cube cutouts (width 4mm, depth 2.2mm)
            }
    }
}

```

```

        translate([x_offset - 5, y_offset - 5, 0])

        cube([10, 10, 2.2]);

    }

}

// cylinder_cutout
translate([21.5, 21.5 ,4.5])
cylinder(cylinder_height, r=cylinder_radius, $fn=50);

//cube_cutout
translate([9., 9, 4.5])
cube([cube_cutout_length, cube_cutout_width, cube_cutout_height]);

//wire_cutout
translate([10,9,height-0.7])
rotate([0,0,90])
cube([wire_length, wire_width, wire_depth]);

//holder_cutout
translate([38, 21.5, 4.5])
rotate([0, 0, 134])

        cube([holder_cutout_width,          holder_cutout_thickness,
holder_cutout_height]);
}

```

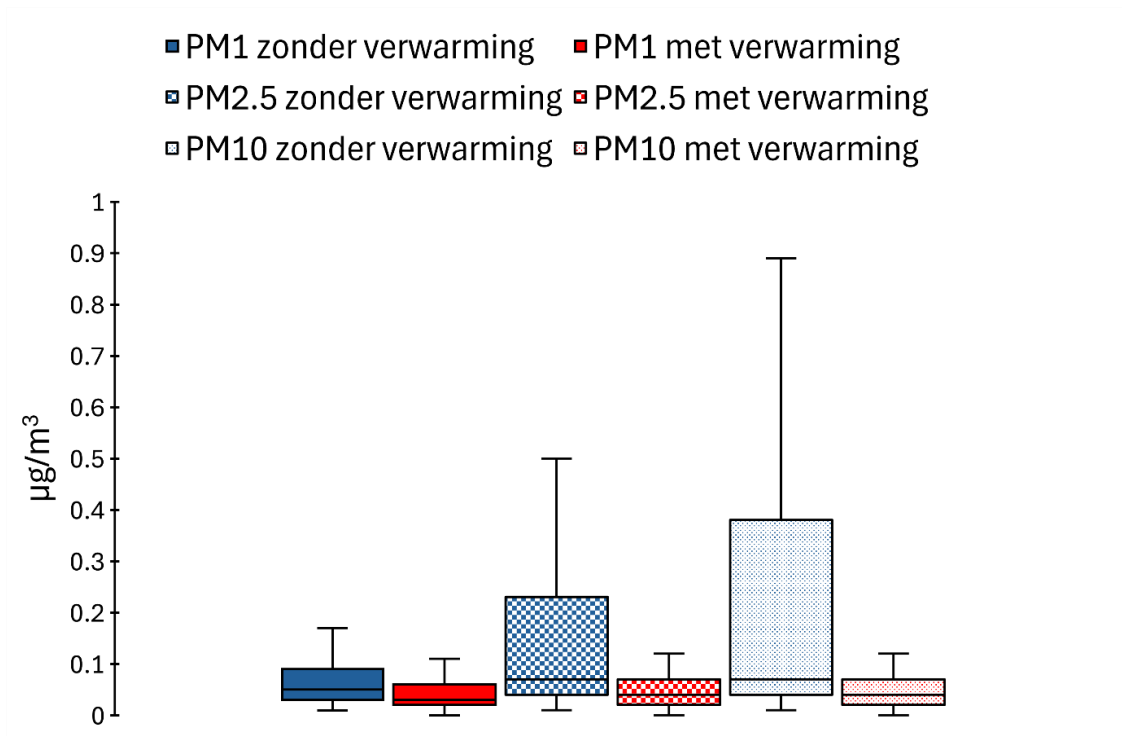

Bijlage X

Aanvullend experiment rond verwarming van fijnstof en hygroscopische groei

Deze test werd uitgevoerd in de achtertuin van een landelijke omgeving in Zandhoven.

Er werden eerst 60 metingen uitgevoerd zonder verwarming, gevolgd door 60 metingen met verwarming op 40°C ingesteld.

Bij het gebruik van verwarming was het noodzakelijk om eerst een stabilisatiefase af te wachten. Deze fase liet toe dat de binnenlucht in het meetsysteem voldoende opwarmde en dat de relatieve vochtigheid (RH) nabij de OPC-sensor zich stabiliseerde rond een constante waarde. De relatieve luchtvochtigheid zonder het verwarmingselement bedraagt gemiddeld 42.6% en met het verwarmingselement 32.1%.



Figuur 56: Effect van voorverwarming op de impact van hygroscopische groei op PM-waarden waarbij het verwarmingselement een positieve invloed heeft op de gemeten fijnstofdeeltjes.

Bron: Eigen werk

Bijlage XI

Python-programma om de gassensoren van Alphasense om te zetten in een waarde naar $\mu\text{g}/\text{m}^3$

#Deze code is deels ontworpen met het gebruik van AI

#Deze code voorziet de lineaire interpolatie tussen de verschillende gassensoren

#Deze code is enkel getest op Windows en niet op Linux, waar de Raspberry Pi op draait. De code wordt op het bureaublad opgeslagen en het kan zijn dat dit voor een linux omgeving aangepast moet worden

```
import os
```

```
import re
```

```
import csv
```

```
def interpolate(signal_mv, s_min, s_avg, c_min, c_avg):
```

```
    return (signal_mv - s_min) * (c_avg - c_min) / (s_avg - s_min) + c_min
```

```
def main():
```

```
    print("Welk gas wordt gemeten? (bijv. NO2, CO, O3,...)")
```

```
    gas = input("Gasnaam: ").strip()
```

```
    print("\nVoer de vier referentiepunten in:")
```

```
    try:
```

```
        s_min = float(input("Minimaal signaal (mV): "))
```

```

s_avg = float(input("Gemiddeld signaal (mV): "))
c_min = float(input("Concentratie bij minimaal signaal ( $\mu\text{g}/\text{m}^3$ ): "))
c_avg = float(input("Concentratie bij gemiddeld signaal ( $\mu\text{g}/\text{m}^3$ ): "))
except ValueError:
    print("Fout: Voer enkel numerieke waarden in.")
    return

print("\nPlak je signaalwaarden hieronder.")
print("Druk op Enter.")

lines = []
while True:
    line = input()
    if not line.strip():
        break
    lines.append(line)

# Splits alle waarden
raw_values = re.split(r"[,\s]+", " ".join(lines))
try:
    signals = [float(v) for v in raw_values if v.strip()]
except ValueError:
    print("Fout bij omzetten van waarden naar getallen.")
    return

concentrations = [interpolate(s, s_min, s_avg, c_min, c_avg) for s in signals]

# Opslaan op bureaublad
desktop_path = os.path.join(os.path.expanduser("~"), "Desktop")
filename = f"{gas.upper()}_resultaten.csv"
full_path = os.path.join(desktop_path, filename)

```

```
try:
    with open(full_path, "w", newline="") as f:
        writer = csv.writer(f)
        writer.writerow(["Signaal (mV)", "Concentratie (µg/m³)"])
        writer.writerows(zip(signals, concentrations))
        print(f"\nBestand opgeslagen op bureaublad als: {filename}")
except Exception as e:
    print(f"Fout bij opslaan: {e}")

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

Bijlage XII

Python-programma voor het tellen en groeperen van logmeldingen

#Deze code is ontworpen met de hulp van AI

#Deze code laat toe om het logboek met foutcodes te groeperen en te ordenen

```
import re
```

```
from collections import Counter, defaultdict
```

```
# --- Configure sensor detection here ---
```

```
SENSOR_PATTERNS = {
```

```
    "OPCN3": r"\bOPC[-_]?N3\b",
```

```
    "CO2": r"\bCO2\b|\bEE894\b|\bEE-?894\b",
```

```
    "O3": r"\bO3\b|\bOZONE\b|\bOX[-_]?A431\b|\bOX[-_]?B431\b",
```

```
    "NO2": r"\bNO2\b|\bNO2\b|NO2-A43F|NO2-B43F",
```

```
    "SO2": r"\bSO2\b|\bSO2\b",
```

```
    "CO": r"\bCO\b(?:\d+)\bCO-SENSOR\b",
```

```
    "RH/T": r"\bSHT30\b|\bSHT31\b|\bSHT\b|\bRH\b|\bTEMP\b",
```

```
}
```

```
LEVEL_RE = re.compile(r"\b(?:P<level>CRITICAL|ERROR|WARNING|INFO)\b",  
re.IGNORECASE)
```

```
CYCLE_RE = re.compile(r"(\d+/\d+)\b") # matches (1/3), (2/3), etc.
```

```
SPACE_RE = re.compile(r"\s+")
```

```
def detect_sensor(line: str) -> str:
```

```
    for sensor, pattern in SENSOR_PATTERNS.items():
```

```

    if re.search(pattern, line, re.IGNORECASE):
        return sensor
    return "Unknown"

def canonical_message(msg: str) -> str:
    """Normalize the log message by removing cycle counters and extra whitespace."""
    msg = CYCLE_RE.sub("", msg) # remove (1/3) etc.
    msg = SPACE_RE.sub(" ", msg) # collapse multiple spaces
    return msg.strip()

def is_returned_message(msg: str) -> bool:
    return msg.lower().startswith("returned")

def analyze_logs(log_text: str):
    by_sensor_level_msg = defaultdict(Counter)
    totals_by_sensor = Counter()
    totals_by_level = Counter()

    for raw in log_text.splitlines():
        line = raw.strip()
        if not line:
            continue

        m = LEVEL_RE.search(line)
        if not m:
            continue # skip lines without a recognized level

        level = m.group("level").upper()
        sensor = detect_sensor(line)

        # message after the level token
        msg = line[m.end():].strip() or "(no message text)"

```

```

# Normalize message text
msg_norm = canonical_message(msg)

by_sensor_level_msg[(sensor, level)][msg_norm] += 1
totals_by_sensor[sensor] += 1
totals_by_level[level] += 1

# --- Print summaries ---
print("\n=== Totals by sensor ===")
for sensor, count in totals_by_sensor.most_common():
    print(f"{sensor:10s} : {count}")

print("\n=== Totals by level ===")
for level, count in totals_by_level.most_common():
    print(f"{level:10s} : {count}")

print("\n=== Detailed counts by sensor and level ===")
sensors_sorted = sorted(set(s for s, _ in by_sensor_level_msg.keys()),
                        key=lambda s: (s == "Unknown", s)) # Unknown last
levels_order = ["CRITICAL", "ERROR", "WARNING", "INFO"]

for sensor in sensors_sorted:
    print(f"\n--- Sensor: {sensor} ---")
    for level in levels_order:
        key = (sensor, level)
        counter = by_sensor_level_msg.get(key)
        if not counter:
            continue

# Voor OPCN3: niet-‘Returned’ eerst, daarna ‘Returned’ onderaan
items = list(counter.items())

```

```

if sensor == "OPCN3":
    non_returned = [(m, c) for m, c in items if not is_returned_message(m)]
    returned = [(m, c) for m, c in items if is_returned_message(m)]
    # sorteer elk blok op frequentie (hoog naar laag)
    non_returned.sort(key=lambda x: x[1], reverse=True)
    returned.sort(key=lambda x: x[1], reverse=True)
    ordered = non_returned + returned
else:
    # standaard: sorteer enkel op frequentie
    ordered = sorted(items, key=lambda x: x[1], reverse=True)

print(f"\n[{level}]")
for msg, count in ordered:
    print(f"{count:5d} × {msg}")

if __name__ == "__main__":
    print("Paste your logs below:\n")
    lines = []
    while True:
        try:
            line = input()
        except EOFError:
            break
        if line.strip() == "":
            break
        lines.append(line)

    log_data = "\n".join(lines)
    analyze_logs(log_data)

```

Bijlage XIII

Overzicht van de SeaCanairy software

seacanairy_project/

├ AFE.py

├ Check-flowrate.py

├ CO2.py

├ Database.py

├ Flow.py

├ GPS.py

├ OPCBIS.py

├ OPCN3.py

├ SeaCanairy.py

├ SeaCanairy_settings.yaml

├ Set_system_time.sh

├ SHT31.py

├ Start_sampling.sh

└ Stop_seacanairy.py

Binnen deze thesis wordt enkel verwezen naar de bestanden SeaCanairy.py, SHT31.py, OPCN3.py en CO2.py, aangezien uitsluitend aan deze Python-code werd gewerkt. Een volledige back-up van het project kan verkregen worden bij Dr. W. Jacobs of Dr. O. Schalm.

Bijlage XIV

De aangepaste SeaCanairy.py-code

```
#!/home/pi/seacanairy_project/venv/bin/python3
"""
This code aims to make the whole Seacanairy works! It start the pump, read the date from
the CO2 sensor,
the OPC-N3, the AFE board, the GPS, SHT31, stop the pump, and store the whole in a file.
"""

# import all libraries
import time
from datetime import date, datetime, timedelta
import csv # for storing data in file
from progress.bar import IncrementalBar # to show beautiful loading bar on the screen
during sampling
import os # to be able to create new files/folders and see the current path
import yaml # to read the settings stored in the 'seacanairy_settings.yaml' file
import logging # to store the errors messages in a separate log file
import threading
import sys # for error messages
import SHT31

# -----
# HELLO
# -----

print("~~~~~")
print("~~~~~ SEACANAIRY ~~~~~")
```

```

print("~~~~~")

# -----
# CHECK THE TIME
# -----
print("\n##### TIME #####")
print("Current date and time is:", datetime.now())
print("If time is not correct, close this window and click on 'Check Date and Time'")

# -----
# SETTINGS
# -----
# Import the settings from the Yaml file
# Store the different settings in dedicated variables to get it more easy

print("\n##### FILES #####")

current_working_directory = str(os.getcwd()) # returns the path where the python script
is currently running

with open(current_working_directory + '/seacanairy_settings.yaml') as file:
    settings = yaml.safe_load(file) # load all the yaml file in a dictionary
    file.close() # always close the file after use

# Sampling period
sampling_period = settings['Seacanairy settings']['Sampling period']

# CO2 sampling period (CO2 sensor takes samples automatically)
CO2_sampling_period = int(sampling_period / settings['CO2 sensor']['Automatic
sampling frequency' \

                                '(number of sample during the above sampling period)'])

```

```

# Amount of time required for the CO2 sensor to take the measurement
CO2_startup_delay = settings['CO2 sensor']['Amount of time required for the sensor to
take the measurement']

# Name of the research (f-e 'Air measurement in my room')
project_name = settings['Seacanairy settings']['Sampling session name']

# Amount of time while the OPCN3 fan keep running without measurement to flush fresh
air inside the casing
OPC_flushing_time = settings['OPC-N3 sensor']['Flushing time']

# Amount of time the OPCN3's laser takes PM sample
OPC_sampling_time = settings['OPC-N3 sensor']['Sampling time']

# OPCN3 Fan speed (0-100)
OPC_fan_speed = settings['OPC-N3 sensor']['Fan speed']

# Number of reading AFE must take and average
AFE_readings_averaged = settings['AFE Board']['Noise reduction - number of reading
averaged']

# Let the user choose if he want to activate the following sensor or not
# Seen the problems encountered with GPS and OPCN3, could be good to disable the
unnecessary sensors (GPS f-e)
# This does not shut down the sensor powering
CO2_activation = settings['CO2 sensor']['Activate this sensor']
OPCN3_activation = settings['OPC-N3 sensor']['Activate this sensor']
GPS_activation = settings['GPS']['Activate this sensor']
AFE_activation = settings['AFE Board']['Activate this sensor']
air_pump_activation = settings['Seacanairy settings']['Activate M&C air pump']
flow_sensor_activation = settings['Air flow sensor']['Activate this sensor']
mysql = settings['MySQL database settings']['Activate database upload']

```

```
air_pump_minimum_running_time = settings['Seacanairy settings']['Air pump minimum
running time']
```

```
AFE_delay = settings['AFE Board']['Absorption time between air pump stop and reading']
```

```
SHT31_activation = settings['SHT31 sensor']['Activate this sensor']
```

```
# -----
```

```
# CREATE FILES
```

```
# -----
```

```
# Create a directory to store everything if it doesn't exit
```

```
directory_path = current_working_directory + "/" + project_name
```

```
print("Current working directory is", directory_path)
```

```
if not os.path.exists(directory_path):
```

```
    os.mkdir(directory_path) # create the directory
```

```
    print("Created directory", directory_path)
```

```
# Create a file to store the log if it doesn't exist
```

```
log_file = directory_path + "/" + project_name + "-log.log"
```

```
if not os.path.isfile(log_file):
```

```
    os.mknod(log_file) # create the file
```

```
    print("Created log file", log_file)
```

```
else:
```

```
    # Add a line to separate different executions
```

```
    file_logfile = open(log_file, "a")
```

```
    file_logfile.write("***** NEW SESSION *****\n")
```

```
    file_logfile.close()
```

```
# You can't go further in the program without creating the folder and the logger file
```

```
# -----
```

```
# IMPORT NECESSARY SENSORS
```

```
# -----
```

```
# Seen the above settings, import the necessary libraries
```

```

# Avoid starting module (i2c, SPI, UART) that will not be used

if CO2_activation:
    import CO2 # Library for the CO2 sensor

if OPCN3_activation:
    import OPCN3 # Library for the OPC-N3 sensor

if GPS_activation:
    import GPS # Library for the GPS

if AFE_activation:
    import AFE

if air_pump_activation:
    import RPi.GPIO as GPIO # to put GPIO high/low to switch the air pump relay on and
    off

if flow_sensor_activation:
    import flow as flow

if SHT31_activation:
    from SHT31 import read_sht31

if mysql:
    import database as db

# -----
# LOGGING
# -----

# Save the messages shown on the console in a dedicated file to understand the possible
issues afterwards

```

```

message_level = logging.INFO

# set up logging to file
logging.basicConfig(level=message_level,
                    format='%(asctime)s %(name)-12s %(levelname)-8s %(message)s',
                    datefmt='%d-%m %H:%M:%S',
                    filename=log_file,
                    filemode='a')

# define a Handler which writes INFO messages or higher to the sys.stderr/display
console = logging.StreamHandler()
console.setLevel(message_level)

# set a format which is simpler for console use
formatter = logging.Formatter('%(name)-12s: %(levelname)-8s %(message)s')

# tell the handler to use this format
console.setFormatter(formatter)

# add the handler to the root logger
logging.getLogger().addHandler(console)

logger = logging.getLogger('SEACANAIRY')

# Following logging messages must be called by logger.debug (...)

# -----
# FUNCTIONS
# -----

# Use the GPIO to turn on and off the air pump via relay

if air_pump_activation:
    GPIO.setwarnings(False) # avoid messages concerning GPIO (GPIO is already low,
    blablabla)

    GPIO.setmode(GPIO.BCM) # use the GPIO names (GPIO1...) instead of the processor

```

```

pin name (BCM...)

pump_gpio = 27
GPIO.setup(pump_gpio, GPIO.OUT, initial=GPIO.LOW)


# -----
# FUNCTIONS
# -----


def loading_bar(name, delay):
    """
    Show a loading bar on the screen during a a certain amount of time
    Make the user understand the software is doing/waiting for something
    :param name: Text to be shown on the left of the loading bar
    :param delay: Amount of time the system is waiting in seconds
    :return: nothing
    """

    bar = IncrementalBar(name, max=(2 * delay), suffix='% (elapsed)s/' + str(delay) + '
seconds')

    for i in range(2 * delay):
        time.sleep(0.5)
        bar.next()
    bar.finish()
    return


def pump_start():
    """
    It put tension on the GPIO number 27 to turn on the air pump relay
    :return: nothing
    """

```

```
GPIO.output(pump_gpio, GPIO.HIGH)
```

```
def pump_stop():
```

```
    """
```

```
    It remove tension from the GPIO number 27 to turn off the air pump relay
```

```
    :return: nothing
```

```
    """
```

```
GPIO.output(pump_gpio, GPIO.LOW)
```

```
def append_data_to_csv(*data_to_write):
```

```
    """
```

```
    Store all the arguments given in the -data.csv file
```

```
    All arguments will be separated by a coma in the csv file
```

```
    :param data_to_write: unlimited amount of arguments
```

```
    :return: nothing
```

```
    """
```

```
to_write = []
```

```
for i in data_to_write:
```

```
    if isinstance(i, list):
```

```
        to_write += i
```

```
    else:
```

```
        to_write += [i]
```

```
# print("To write in SD is:", to_write)
```

```
# to_write = [*data_to_write] # Concatenate all the arguments in one list
```

```
with open(csv_file, mode='a', newline='') as data_file:
```

```
    writer = csv.writer(data_file, delimiter=',', quotechar='"',  
quoting=csv.QUOTE_MINIMAL)
```

```

        writer.writerow(to_write)

data_file.close() # close file after use(general safety practice)


def wait_timestamp(starting_time):
    """
    Wait that the sampling period is passed out to start the next measurement
    :param starting_time: time at which the measurement has started ('time.time()'
    required)
    :return: Nothing, function will stop when the next measurement can start
    """
    finishing_time = time.time()
    next_launching = starting_time + sampling_period # time at which the next sample
    should start

    to_wait = next_launching - finishing_time

    # if the sampling process took more time than the expected sampling period
    if finishing_time >= next_launching:
        logger.error("Sampling cycle took " + str(round((finishing_time - next_launching), 1))
+
        "seconds more than expected (" + str(round(sampling_period, 0)) + "
seconds)")
        print("Starting new sample...")
        return # function stop and new sample start

    while time.time() < next_launching:
        now = time.time()
        for _ in range(10):
            now = time.time()
            if now < next_launching:
                time.sleep(.1)

```

```

        else:
            break

    to_wait = round(next_launching - now, 0) # amount of time system will wait
    print("Waiting before next measurement:", int(to_wait), "seconds (sampling time is
set on",
        sampling_period,
        "seconds) ", end="\r")
    if not now < next_launching:
        break

# Delete the waiting countdown and skip a line
# Print a whole blank to remove completely the previous line ("Waiting before next
measurement...")
print(" ")
print("Starting new sample...")
return # function stop and new sample start

# -----
# MAIN CODE
# -----

# INITIATE CSV FILE

# Create the file to store the data if it doesn't exist
csv_file = directory_path + "/" + str(project_name) + "-data.csv"

to_write = []
to_write += ["Date/Time"]

# for the OPC-N3 sensor:
to_write += ["PM 1 ( $\mu\text{g}/\text{m}^3$ )", "PM 2.5 ( $\mu\text{g}/\text{m}^3$ )", "PM 10 ( $\mu\text{g}/\text{m}^3$ )",

```

```

"Temperature OPC (°C)", "Relative Humidity OPC (%RH)",
"sampling time OPC (sec)", "sample flow rate OPC (ml/s)",
"bin 0", "bin 1", "bin 2", "bin 3", "bin 4", "bin 5",
"bin 6", "bin 7", "bin 8", "bin 9", "bin 10", "bin 11",
"bin 12", "bin 13", "bin 14", "bin 15", "bin 16", "bin 17",
"bin 18", "bin 19", "bin 20", "bin 21", "bin 22", "bin 23",
"bin 1 MToF", "bin 3 MToF", "bin 5 MToF", "bin 7 MToF",
"reject count glitch", "reject count long TOF", "reject count ratio",
"reject count out of range", "fan revolution count", "laser status"]

# for the flow sensor:
to_write += ["flow (sccm)", "flow (slm)", "flow (slh)", "flow slm min", "flow slm max"]

# for the AFE:
to_write += ["temperature (°C)", "temperature (mV)",
"temperature min", "temperature max",
"NO2 (ppb)", "NO2 main (mV)", "NO2 aux (mV)",
"NO2 main min", "NO2 main max", "NO2 aux min", "NO2 aux max",
"OX (ppb)", "OX main (mV)", "OX aux (mV)",
"OX main min", "OX main max", "OX aux min", "OX aux max",
"SO2 (ppb)", "SO2 main (mV)", "SO2 aux (mV)",
"SO2 main min", "SO2 main max", "SO2 aux min", "SO2 aux max",
"CO (ppb)", "CO main (mV)", "CO aux (mV)",
"CO main min", "CO main max", "CO aux min", "CO aux max"]

# for the GPS:
to_write += ["Date and time (UTC)",
"GPS fix date and time (UTC)", "latitude", "longitude", "SOG (kts)", "COG",
"horizontal dilution of precision", "accuracy",
"altitude (m)", "WGS84 correction (m)", "fix type", "GPS status"]

# for the CO2 sensor:

```

```
to_write += ["Relative Humidity (%RH)", "Temperature (°C)", "Pressure (hPa)",
            "CO2 average (ppm)", "CO2 instant (ppm)"]
```

```
#for the SHT31 sensor
```

```
to_write += ["SHT31 temperature (°C)", "SHT humidity (%)"]
```

```
db_header = []
```

```
db_data_type = []
```

```
db_header += ["Date_Time"]
```

```
db_data_type += ["DATETIME"]
```

```
# for the OPC-N3 sensor:
```

```
db_header += ["PM_1", "PM_2_5", "PM_10",
            "Temperature_OPC", "Relative_Humidity_OPC",
            "sampling_time_OPC", "sample_flow_rate_OPC",
            "bin_0", "bin_1", "bin_2", "bin_3", "bin_4", "bin_5",
            "bin_6", "bin_7", "bin_8", "bin_9", "bin_10", "bin_11",
            "bin_12", "bin_13", "bin_14", "bin_15", "bin_16", "bin_17",
            "bin_18", "bin_19", "bin_20", "bin_21", "bin_22", "bin_23",
            "bin_1_MToF", "bin_3_MToF", "bin_5_MToF", "bin_7_MToF",
            "reject_count_glitch", "reject_count_long_TOF", "reject_count_ratio",
            "reject_count_out_of_range", "fan_revolution_count", "laser_status"]
```

```
db_data_type += ["FLOAT"] * 41
```

```
# for the flow sensor:
```

```
db_header += ["flow_sccm", "flow_slm", "flow_slh", "flow_slm_min", "flox_slm_max"]
```

```
db_data_type += ["FLOAT"] * (3 + 2)
```

```
# for the AFE:
```

```
db_header += ["temperature_C", "temperature_mV",
            "temperature_min", "temperature_max",
```

```

"NO2_ppb", "NO2_main_mV", "NO2_aux_mV",
"NO2_main_min", "NO2_main_max", "NO2_aux_min", "NO2_aux_max",
"OX_ppb", "OX_main_mV", "OX_aux_mV",
"OX_main_min", "OX_main_max", "OX_aux_min", "OX_aux_max",
"SO2_ppb", "SO2_main_mV", "SO2_aux_mV",
"SO2_main_min", "SO2_main_max", "SO2_aux_min", "SO2_aux_max",
"CO_ppb", "CO_main_mV", "CO_aux_mV",
"CO_main_min", "CO_main_max", "CO_aux_min", "CO_aux_max"]
db_data_type += ["FLOAT"] * (32)

# for the GPS:
db_header += ["Date_time_UTC",
              "GPS_fix_date_and_time_UTC", "latitude", "longitude", "SOG", "COG",
              "horizontal_dilution", "accuracy",
              "altitude", "WGS84_correction", "fix_type", "GPS_status"]
db_data_type += ["TEXT"] * 12

# for the CO2 sensor:
db_header += ["Relative_Humidity", "Temperature", "Pressure",
              "CO2_average", "CO2_instant"]
db_data_type += ["FLOAT"] * 5

#for the SHT31 sensor
db_header += ["SHT31_temp", "SHT31_hum"]
db_data_type += ["FLOAT", "FLOAT"]

if mysql:
    db.create_new_table(db_header, db_data_type)

if not os.path.isfile(csv_file): # if the file doesn't exist
    os.mknod(csv_file) # create the file
    print("Initializing data file", csv_file)

```

```

# Write a first line to the file, this will be the column headers
append_data_to_csv(to_write)

else:
    logger.info(""" + str(csv_file) + "" already exist")
    logger.info("Appending data to this file")

print("\n##### STARTING SENSORS #####")

# Keep a trace of the day and time at which the system has started

now = datetime.now() # get time
logger.info("Starting of Seacanairy on the " + str(now.strftime("%d/%m/%Y at
%H:%M:%S"))) # delete time decimals

# WARN USER IF A SENSOR IS DEACTIVATED

if not CO2_activation:
    logger.warning("CO2 sensor has been disabled by the user in
'seacanairy_settings.yaml'")

if not OPCN3_activation:
    logger.warning("OPC-N3 sensor has been disabled by the user in
'seacanairy_settings.yaml'")

if not GPS_activation:
    logger.warning("GPS has been disabled by the user in 'seacanairy_settings.yaml'")

if not air_pump_activation:
    logger.warning("Air pump has been disabled by the user in 'seacanairy_settings.yaml'")

if not mysql:

```

```
logger.warning("Database upload has been disabled by the user in  
'seacanairy_settings.yaml'")
```

```
# SET INTERNAL CO2 SENSOR TIMESTAMP
```

```
if CO2_activation:
```

```
    # Read the internal timestamp of the CO2 sensor, and change the value if necessary
```

```
    if CO2_sampling_period != CO2.internal_timestamp(): # if internal timestamp is not  
the good one... change it
```

```
        CO2.internal_timestamp(CO2_sampling_period)
```

```
        # nothing in the brackets = read, timestamp in the brackets = write
```

```
else: # if user don't want to use the sensor...
```

```
    try:
```

```
        # set the internal timestamp to the longest period available to reduce its wear...
```

```
        CO2.internal_timestamp(3600)
```

```
        logger.info("CO2 sensor disabled, internal timestamp set on 3600 seconds to reduce  
its wear")
```

```
    except:
```

```
        # It means that CO2 sensor is maybe not plugged in...
```

```
        pass # nothing to do if it fails
```

```
if OPCN3_activation:
```

```
    # Initialize OPCN3 SPI, avoid loosing first communication
```

```
    OPCN3.initialization_SPI()
```

```
    time.sleep(.2)
```

```
    # Set the desired OPC fan speed
```

```
    OPCN3.set_fan_speed(OPC_fan_speed)
```

```
if CO2_activation:
```

```
    # Ask the CO2 sensor to take a new sample
```

```
    print("Synchronizing CO2 sensor with Seacanairy sampling period")
```

```
    CO2.trigger_measurement(True) # True == force
```

```
print("\n##### SAMPLING! #####")
```

```
# LOOP which will run continuously
```

```
try:
```

```
    for _ in range(2):
        pump_start()
        time.sleep(.25)
        pump_stop()
        time.sleep(.25)
    pump_start()
    print("Flushing air inside the system...", end='\r')
    time.sleep(10)
```

```
try: # necessary to stop the pump when the program is stopped
```

```
    while True:
```

```
        # Get the time in second at which the measurement start
```

```
        start = time.time() # return the time expressed in second since the python date
reference
```

```
        # a bit the same as 'millis()' on Arduino
```

```
        # easier to work with than with a complex string datetime format and a timedelta
function...
```

```
        # Get date and time to store in the Excel file
```

```
        now = datetime.now().strftime('%Y-%m-%d %H:%M:%S') # remove the decimals
and change the date order
```

```
    if air_pump_activation:
```

```
        # if OPCN3_activation:
```

```
            # if OPCN3.fan_on():
```

```
                # OPCN3.laser_on()
```

```

#   OPCN3_fan_started = 1
#   time.sleep(1)
pump_start() # start the pump
print("Air pump is on")
# time.sleep(.5) # let time to the pump to start and flow to happen

to_write = [now] # create the list in which data to store will be saved
# [a, b, c] + [d, e, f] = [a, b, c, d, e, f]

if air_pump_activation:
    print("Flushing air inside piping...", end='\r')
    while start + air_pump_minimum_running_time - OPC_flushing_time -
OPC_sampling_time - 2 > time.time():
        # wait until expected flushing time is reached
        time.sleep(.25)

if CO2_activation:
    print("Trigger CO2 measurement    ")
    CO2.trigger_measurement()

if flow_sensor_activation:
    # Start Flow Rate thread
    flow_period = 1
    flow_number_measurements = 10
    flow_delay = 1

    flow_thread = threading.Thread(target=flow.start_averaged_measurement,
                                   args=[flow_period, flow_number_measurements, flow_delay],
    daemon=True)

    print("Reading flow rate during", flow_period, "seconds after", flow_delay,
          "seconds delay...")
    flow_thread.start()

```

```

if OPCN3_activation:
    # Get OPC-N3 sensor data (see 'OPCN3.py')
    print("***** OPC-N3 *****")
    OPC_data = OPCN3.get_data(OPC_flushing_time, OPC_sampling_time,
start_fan_laser=True)
    to_write += [OPC_data["PM 1"], OPC_data["PM 2.5"], OPC_data["PM 10"],
        OPC_data["temperature"], OPC_data["relative humidity"],
        OPC_data["sampling time"], OPC_data["sample flow rate"],
        OPC_data["bin 0"], OPC_data["bin 1"], OPC_data["bin 2"], OPC_data["bin
3"],
        OPC_data["bin 4"], OPC_data["bin 5"], OPC_data["bin 6"], OPC_data["bin
7"],
        OPC_data["bin 8"], OPC_data["bin 9"], OPC_data["bin 10"], OPC_data["bin
11"],
        OPC_data["bin 12"], OPC_data["bin 13"], OPC_data["bin 14"],
OPC_data["bin 15"],
        OPC_data["bin 16"], OPC_data["bin 17"], OPC_data["bin 18"],
OPC_data["bin 19"],
        OPC_data["bin 20"], OPC_data["bin 21"], OPC_data["bin 22"],
OPC_data["bin 23"],
        OPC_data["bin 1 MToF"], OPC_data["bin 3 MToF"],
        OPC_data["bin 5 MToF"], OPC_data["bin 7 MToF"],
        OPC_data["reject count glitch"],
        OPC_data["reject count long TOF"], OPC_data["reject count ratio"],
        OPC_data["reject count out of range"],
        OPC_data["fan revolution count"], OPC_data["laser status"]]

else:
    to_write += ["-"] * (41)

if air_pump_activation:

```

```

print("***** AIR FLOW *****")
print("Reading flow rate...", end='\r')
flow_thread.join() # wait the end of the thread
print("          ", end='\r')
flow_OPC_data = flow.get_averaged_measurement()
to_write += [flow_OPC_data['average flow [sccm]'], flow_OPC_data['average
flow [slm]'],
            flow_OPC_data['average flow [slh]'],
            flow_OPC_data['slm min'], flow_OPC_data['slm max']]

else:
    to_write += ["-"] * (3 + 2)

if air_pump_activation:
    while start + air_pump_minimum_running_time > time.time():
        # wait until expected flushing time is reached
        print("Flushing air inside piping...", end='\r')
        time.sleep(.25)
    pump_stop()
    print("Air pump is off (pump has run", round(time.time() - start, 0), "seconds)")

if AFE_activation:
    # Get OPC-N3 sensor data (see 'AFE.py')
    print("***** AFE BOARD *****")
    AFE_data = AFE.start_averaged_data(AFE_readings_averaged, display=True)
    to_write += [AFE_data["temperature"], AFE_data["temperature raw"],
                AFE_data["temperature min"], AFE_data["temperature max"],
                AFE_data["NO2 ppb"], AFE_data["NO2 main"], AFE_data["NO2 aux"],
                AFE_data["NO2 main min"], AFE_data["NO2 main max"], AFE_data["NO2
aux min"],
                AFE_data["NO2 aux max"],
                AFE_data["OX ppb"], AFE_data["OX main"], AFE_data["OX aux"],

```

```

AFE_data["OX main min"], AFE_data["OX main max"], AFE_data["OX aux
min"],
AFE_data["OX aux max"],
AFE_data["SO2 ppb"], AFE_data["SO2 main"], AFE_data["SO2 aux"],
AFE_data["SO2 main min"], AFE_data["SO2 main max"], AFE_data["SO2
aux min"],
AFE_data["SO2 aux max"],
AFE_data["CO ppb"], AFE_data["CO main"], AFE_data["CO aux"],
AFE_data["CO main min"], AFE_data["CO main max"], AFE_data["CO aux
min"],
AFE_data["CO aux max"],
]

print("")
AFE.print_measurements(AFE_data)
# AFE.save_sensor_data()

else:
    to_write += ["-"] * (4 * 3 + 4 * 4 + 4)

if GPS_activation:
    # print("Wait switching from SPI to UART...", end='\r')
    # time.sleep(1) # avoid too close communication between SPI of OPC and UART
of GPS
    # Get GPS information
    print("***** GPS *****")
    GPS_data = GPS.get_position()
    to_write += [GPS_data["current time"], GPS_data["current date and time"],
        GPS_data["latitude"], GPS_data["longitude"],
        GPS_data["SOG"], GPS_data["COG"], GPS_data["horizontal precision"],
GPS_data["accuracy"],
        GPS_data["altitude"], GPS_data["WGS84 correction"], GPS_data["fix

```

```
status"], GPS_data["status"]]
```

```
else:
```

```
    to_write += ["-"] * 12
```

```
if CO2_activation:
```

```
    # Get CO2 sensor data (see 'CO2.py')
```

```
    print("***** CO2 SENSOR *****")
```

```
    print("Waiting for sensor readiness...", end="\r")
```

```
    while time.time() < start + 16:
```

```
        time.sleep(.5)
```

```
        print("          ", end='\r')
```

```
        CO2_data = CO2.get_data()
```

```
        #to_write += [CO2_data["temperature"], CO2_data["pressure"],
```

```
        #CO2_data["average"], CO2_data["instant"]]
```

```
        to_write += [CO2_data["relative humidity"], CO2_data["temperature"],
CO2_data["pressure"],
                    CO2_data["average"], CO2_data["instant"]]
```

```
#else:
```

```
    #to_write += ["-"] * 5
```

```
#Get data from SHT31 temp and hum
```

```
print("***** SHT31 *****")
```

```
print("***** OUTSIDE TEMP AND HUM *****")
```

```
sht31_data = SHT31.read_sht31()
```

```
sht31_temperature = sht31_data["temperature"]
```

```
sht31_humidity = sht31_data["humidity"]
```

```
if sht31_temperature is None or sht31_humidity is None:
```

```
    sht31_temperature, sht31_humidity = "-", "-"
```

```
    print("Failed to read SHT31 sensor")
```

```

else:
    #sht31_temperature, sht31_humidity = "", "" causes problems with the print
statements because '-' is a str
    print(f"SHT31 temperature: {sht31_temperature: .2f} °C")
    print(f"SHT31 humidity: {sht31_humidity: .2f} %\n\n\n")
    to_write += [sht31_temperature, sht31_humidity]

# Store everything in the database
if mysql:
    db.update(db_header, db_data_type, to_write)

# Store everything in the csv file
append_data_to_csv(to_write)

# Time at which the sampling finishes
finish = time.time() # as previously, expressed in seconds since reference date

# Calculate the amount of time the sampling process took, round to 0 to avoid
decimals
# int(...) to delete the remaining 0 behind the coma
logger.info("Sampling finished in " + str(int(round(finish - start, 0))) + " seconds")

# Wait that the sampling period is passed
wait_timestamp(start)

except KeyboardInterrupt:
    logger.info("Software stopped by the user")
    OPCN3.laser_off()
    OPCN3.fan_off()
except: # stop everything before exiting the program
    logger.critical("Software has been stopped (" + str(sys.exc_info()) + ")")
    pump_stop()

```

except:

 pump_stop()

Bijlage XV

De aangepaste OPCN3.py code

```
#!/home/pi/seacanairy_project/venv/bin/python3
"""
Library for the use and operation of the Alphasense OPC-N3 sensor
"""

import codecs
import csv
import spidev # driver for the SPI/serial communication
import time
import struct # to convert the IEEE bytes to float
import datetime
import sys
import os # to create folders/files and read current path
from datetime import datetime
from progress.bar import IncrementalBar # beautiful progress bar during sampling
# import RPi.GPIO as GPIO # used for CS (Chip Select line)

import logging # save logger messages into memory

# yaml settings
import yaml # read user settings

# -----
# YAML SETTINGS
# -----
```

```

# Get current directory
current_working_directory = str(os.getcwd())

with open(current_working_directory + '/seacanairy_settings.yaml') as file:
    settings = yaml.safe_load(file)
    file.close()

store_debug_messages = settings['CO2 sensor']['Store debug messages (important
increase of logs)']

project_name = settings['Seacanairy settings']['Sampling session name']

OPC_flushing_time = settings['OPC-N3 sensor']['Flushing time']

OPC_sampling_time = settings['OPC-N3 sensor']['Sampling time']

take_new_sample_if_checksum_is_wrong = \
    settings['OPC-N3 sensor']['
        'Take a new measurement if checksum is wrong (avoid shorter sampling periods
when errors)']

# -----
# LOGGING SETTINGS
# -----
# all the settings and other code for the logging
# logging = keep a trace of some messages in a file to be reviewed afterward (check for
errors f-e)

def set_logger(message_level, log_file):
    # set up logging to file

```

```

logging.basicConfig(level=message_level,
                    format='%(asctime)s %(name)-12s %(levelname)-8s %(message)s',
                    datefmt='%d-%m %H:%M',
                    filename=log_file,
                    filemode='a')

```

```

logger = logging.getLogger('OPC-N3') # name of the logger
# all further logging must be called by logger.'level' and not logging.'level'
# if not, the logging will be displayed as 'ROOT' and NOT 'OPC-N3'
return logger

```

```

if __name__ == '__main__': # if you run this code directly ($ python3 CO2.py)
    message_level = logging.DEBUG # show ALL the logging messages
    # Create a file to store the log if it doesn't exist
    log_file = current_working_directory + "/log/OPCN3-debugging.log" # complete file
location required for the Raspberry
    if not os.path.isfile(log_file):
        os.mknod(log_file)
    print("DEBUG messages will be shown and stored in '" + str(log_file) + "'")
    logger = set_logger(message_level, log_file)
    # define a Handler which writes INFO messages or higher to the sys.stderr/display
    console = logging.StreamHandler()
    console.setLevel(message_level)
    # set a format which is simpler for console use
    formatter = logging.Formatter('%(name)-12s: %(levelname)-8s %(message)s')
    # tell the handler to use this format
    console.setFormatter(formatter)
    # add the handler to the root logger
    logging.getLogger().addHandler(console)

```

```

else: # if this file is considered as a library (if you execute 'seacanairy.py' for example)

```

```

# it will only print and store INFO messages and above in the corresponding log_file
if store_debug_messages:
    message_level = logging.DEBUG
else:
    message_level = logging.INFO
log_file = current_working_directory + "/" + project_name + "/" + project_name + "-
log.log"

# no need to add a handler, because there is already one in seacanairy.py
logger = set_logger(message_level, log_file)

# -----
# SPI CONFIGURATION
# -----
# configuration of the Serial communication to the sensor

bus = 0 # name of the SPI bus on the Raspberry Pi 3B+, only one bus
device = 0 # name of the SS (Slave Selection) pin used for the OPC-N3
spi = spidev.SpiDev() # enable SPI (SPI must be enable in the RPi settings beforehand)
spi.open(bus, device) # open the spi port at start
spi.max_speed_hz = 307200 # must be between 300 and 750 kHz
# Personal experiment shown that UART and SPI speeds must be multiple
# UART baud rate is 9600 for the GPS sensor
#  $9600 * 2 * 2 * 2 * 2 * 2 = 307200$ 
# If not, both sensor data are corrupted
# If not, OPCN3 returns alternately  $\text{int}(48) = \text{hex}(0x30) = \text{bytes}(00110000)$ 
spi.mode = 0b01 #  $\text{bytes}(0b01) = \text{int}(1) \rightarrow$  SPI mode 1
# first bit (from right) = CPHA = 0  $\rightarrow$  data are valid when clock is rising
# second bit (from right) = CPOL = 0  $\rightarrow$  clock is kept low when idle
wait_10_milli = 0.015 # 15 ms
wait_10_micro = 1e-06
wait_reset_SPI_buffer = 5 # seconds

```

```
time_available_for_initiate_transmission = 15 # seconds - timeout for SPI response
```

```
# if the sensor is disconnected, it can happen that the RPi wait for its answer, which never comes...
```

```
# avoid the system to wait for unlimited time for that answer
```

```
# CS (chip selection) manually via GPIO - NOT CURRENTLY USED, to switch the OPCN3 CS line manually up and down
```

```
# GPIO.setmode(GPIO.BCM) # use the GPIO names (GPIO1...) instead of the processor pin name (BCM...)
```

```
# CS = 25 # GPIO number in which CS is connected
```

```
# GPIO.setup(CS, GPIO.OUT, initial=GPIO.HIGH)
```

```
# def cs_high(delay=0.010):
```

```
#     """Close communication with OPC-N3 by setting CS on HIGH"""
```

```
#     time.sleep(delay)
```

```
#     # GPIO.output(CS, GPIO.HIGH)
```

```
#     # time.sleep(delay)
```

```
#
```

```
#
```

```
# def cs_low(delay=0.010):
```

```
#     """Open communication with OPC-N3 by setting CS on LOW"""
```

```
#     time.sleep(delay)
```

```
#     # GPIO.output(CS, GPIO.LOW)
```

```
#     # time.sleep(delay)
```

```
# Safe OPCN3 data in a CVS file
```

```
def initiate_transmission(command_byte):
```

```
    """
```

```
    Initiate SPI transmission to the OPC-N3
```

First loop on the manufacturer's flow Chart

:param command_byte: byte to be sent during communication initiation

:return: True when SPI initiation has been done, False if it failed

"""

attempts = 0 # sensor is busy loop

cycle = 1 # SPI buffer reset loop (going to the right on the flowchart)

logger.debug("Initiate transmission with command byte " + str(hex(command_byte)))

stop = time.time() + time_available_for_initiate_transmission

time in seconds at which we consider it took too much time to answer

cs_low() # not used anymore

while time.time() < stop:

print('Trying to start the OPC-sensor')

logger.debug("attempts = " + str(attempts)) # disable to reduce the amount of time

between spi.xfer

reading = spi.xfer([command_byte]) # initiate control of power state

spi.xfer() means write a byte AND READ AT THE SAME TIME

if reading == [243]: # SPI ready = 0xF3 = 243 = 0b11110011

time.sleep(wait_10_micro)

return True # indicate that the initiation succeeded

if reading == [49]: # SPI busy = 0x31 = 49

time.sleep(wait_10_milli)

attempts += 1

elif reading in ([0x30], [0x62], [0x8F], [0x02], [0xCF], [0xC0], [0xE7], [0x80], [0x33],
[0x07]): #returned 0x30 but we expect 0x31

```

#         logger.error(f"Returned 0x{reading[0]:02X}, wait for buffer and send command
0xF3 to force a reaction")
#         time.sleep(wait_reset_SPI_buffer)
#         spi.xfer([0xF3])
#         attempts =0
#         cycle += 1
#         continue
#

```

elif reading == [3]: #returned 0x3. this is always the second wrong byte returned, see else statement below in the function

```

        logger.error("Returned 0x3, wait for buffer and send command 0xF3 to force a
reaction")
        time.sleep(1)
        attempts = 0
        cycle += 1
        continue

```

elif reading == [230] or reading == [99] or reading == [0]:

#0x6E or 0x63 or 0x00

During developing, I noticed that these were the answers given by the sensor when the CS line was

facing troubles.

This comes from personal experiment and not from the official documentation

To resolve it, try connecting the CS line directly to the ground (current setting)

logger.error("Problem with the SS (Slave Select) line "

"(error code " + str(hex(reading[0])) + "), skipping")

logger.error(f"Returned 0x{reading[0]:02X}, wait for buffer and send command 0xF3 to force a reaction")

spi.xfer([0xF3])

attempts =0

```

        cycle += 1
        continue
#         cycle += 1
#         logger.debug("Check that SS line is well kept DOWN (0V) during transmission."
#             " Try again by connecting SS Line of sensor to Ground")
#         print("Waiting SPI Buffer reset (" + str(reading) + ")", end='\r')
#         # time.sleep(wait_reset_SPI_buffer)
#         return False

    else:
        logger.error(
            "Failed to initiate transmission (unexpected code returned: " +
str(hex(reading[0])) + ") (" + str(
                cycle) + "/3)")
        print("Waiting SPI Buffer reset (" + str(reading) + ")", end='\r')
        if cycle >= 3:
            logger.critical("Failed to initiate transmission (reset 3 times SPI, still error)")
            return False

        #Catch all the errors and flush the memory of the OPC with the 0xF3 command,
        #this helps drastically improve sensor stability against electromagnetic interference
        #0xF3 requests an info string but this is only to flush the memory buffer of the
sensor
        #without it, you will just get errors and no readings;
        #so keep this part in it
        else:
            time.sleep(wait_reset_SPI_buffer)
            logger.error(f"Returned 0x{reading[0]:02X}, wait for buffer and send command
0xF3 to force a reaction")
            spi.xfer([0xF3])
            attempts = 0
            cycle += 1
            continue

```

```

if attempts > 60:
    # it is recommended to use > 20 in the Alphasense documentation
    # After experiment it seems that 60 is a good value
    # (does not take too much time, and let some chance to the sensor to answer
READY)
    logger.error("Failed 60 times to initiate control of power state, reset OPC-N3 SPI
buffer, trying again")
    # cs_high()
    print("Waiting SPI Buffer reset (" + str(reading) + ")", end='\r')
    time.sleep(wait_reset_SPI_buffer) # time for spi buffer to reset

    attempts = 0 # reset the "SPI busy" loop
    cycle += 1 # increment of the SPI reset loop
    # cs_low()

if cycle >= 3:
    logger.critical("Failed to initiate transmission (reset 3 times SPI, still error)")
    return False

logger.critical("Transmission initiation timeout (> "
    + str(time_available_for_initiate_transmission) + " secs)")
return False # function depending on initiate_transmission function will not continue,
indicate error

def fan_off():
    """
    Turn OFF the fan of the OPC-N3
    :return: False if it succeeded turning off the fan, True if it failed
    """
    print("Turning fan OFF", end='\r')

```

```

logger.debug("Turning fan OFF")
attempts = 1

while attempts < 4:
    # logger.debug("attempts = " + str(attempts)) # disable to reduce the amount of time
    # between spi.xfer
    if initiate_transmission(0x03):
        reading = spi.xfer([0x02])
        # cs_high()
        # spi.close() # close the serial port to let it available for another device
        # Avoid opening and closing ports too often.
        # Avoid getting "too much files opened" error after long running time
        if reading == [0x03]: # official answer of the OPC-N3
            print("Fan is OFF      ")
            # time.sleep(0.5) # avoid too close communication (AND let some time to the
            # OPC-N3 to stop the fan)
            return False
        else:
            time.sleep(1) # let some time to the OPC-N3 (to try to stop the fan)
            reading = read_DAC_power_status('fan')
            if reading == 0:
                return False
            elif reading == 1:
                attempts += 1
                logger.warning("Failed to stop the fan, trying again...")
                print("Waiting SPI Buffer reset", end='\r')
                time.sleep(wait_reset_SPI_buffer)
            else:
                attempts += 1
                print("Waiting SPI Buffer reset", end='\r')
                time.sleep(wait_reset_SPI_buffer)
    if attempts >= 3:

```

```

        logger.critical("Failed 3 consecutive times to stop the fan")
        return True
    else:
        logger.critical("Failed to stop the fan (transmission problem)")
        return True
return True

def fan_on():
    print("Wait for 5 seconds")
    time.sleep(5)
    """
    Turn ON the fan of the OPC-N3
    :return: True if it succeeded turning off the fan, False if it failed
    """
    print("Turning fan ON", end='\r')
    logger.debug("Turning fan ON")

    attempts = 1

    while attempts < 4:
        # logger.debug("attempts = " + str(attempts)) # disable to reduce the amount of time
        # between spi.xfer
        if initiate_transmission(0x03):
            logger.debug("attempts = " + str(attempts))
            reading = spi.xfer([0x03])
            # cs_high()
            # spi.close() # close the serial port to let it available for another device
            # Avoid opening and closing ports too often.
            # Avoid getting "too much files opened" error after long running time
            time.sleep(0.6) # wait > 600 ms to let the fan start
            if reading == [0x03]: # official answer of the OPC-N3

```

```

    print("Fan is ON      ")
    time.sleep(0.5) # avoid too close communication
    return True # indicate that fan has started

else:
    time.sleep(1) # let time to the OPC-N3 to try to start the fan
    reading = read_DAC_power_status('fan')
    if reading == 1:
        return True # indicate that fan has started
    elif reading == 0:
        logger.error("Failed to start the fan...")
        attempts += 1
        print("Waiting SPI Buffer reset", end='\r')
        time.sleep(wait_reset_SPI_buffer)
    else:
        attempts += 1
        print("Waiting SPI Buffer reset", end='\r')
        time.sleep(wait_reset_SPI_buffer)
    if attempts >= 4:
        log = "Failed 3 consecutive times to start the fan"
        logger.critical(log)
        return False # indicate that fan is OFF
    else:
        logger.critical("Failed to start the fan (transmission problem)")
        return False
return True

```

```
def laser_on():
```

```
    """
```

```
    Turn ON the laser of the OPC-N3
```

```
    :return: True if it succeeded turning off the laser, False if it failed
```

```

"""

print("Turning laser ON", end='\r')
logger.debug("Turning laser ON")
attempts = 0

while attempts < 4:
    # logger.debug("attempts = " + str(attempts)) # disable to reduce the amount of time
between spi.xfer
    if initiate_transmission(0x03):
        reading = spi.xfer([0x07])
        # cs_high()
        # spi.close() # close the serial port to let it available for another device
        # Avoid opening and closing ports too often.
        # Avoid getting "too much files opened" error after long running time
        if reading == [0x03]:
            print("Laser is ON      ")
            time.sleep(.5) # avoid too close communication
            return True # indicate that the laser is ON
        else:
            time.sleep(1) # let time to the OPC-N3 to try to start the laser
            reading = read_DAC_power_status('laser')
            if reading == 1:
                logger.info("Wrong answer received after SPI writing, but laser is well on")
                return True # indicate that the laser is ON
            elif reading == 0:
                logger.error("Failed to start the laser, trying again...")
                attempts += 1
                print("Waiting SPI Buffer reset", end='\r')
                time.sleep(wait_reset_SPI_buffer)
            else:
                attempts += 1
                print("Waiting SPI Buffer reset", end='\r')

```

```

        time.sleep(wait_reset_SPI_buffer)
    if attempts >= 3:
        logger.critical("Failed 3 consecutive times to start the laser")
        return False # indicate that laser is still off
    else:
        logger.critical("Failed to start the laser (transmission problem)")
        return False
return False

def laser_off():
    """
    Turn OFF the laser of the OPC-N3
    :return: False if it succeeded turning off the laser, True if it failed
    """
    print("Turning the laser OFF", end='\r')
    logger.debug("Turning laser OFF")
    attempts = 0

    while attempts < 4:
        # logger.debug("attempts = " + str(attempts)) # disable to reduce the amount of time
        # between spi.xfer
        if initiate_transmission(0x03):
            reading = spi.xfer([0x06])
            # cs_high()
            # spi.close() # close the serial port to let it available for another device
            # Avoid opening and closing ports too often.
            # Avoid getting "too much files opened" error after long running time
            if reading == [0x03]:
                print("Laser is OFF          ")
                # time.sleep(1) # avoid too close communication
                return False

```

```

else:
    time.sleep(1) # let time to the OPC-N3 to try to stop the laser
    reading = read_DAC_power_status('laser')
    if reading == 0:
        logger.info("Wrong answer received after writing, but laser is well off")
        return False
    elif reading == 1:
        logger.error("Failed to stop the laser (code returned is " + str(reading) + "), trying
again...")
        attempts += 1
        print("Waiting SPI Buffer reset", end='\r')
        time.sleep(wait_reset_SPI_buffer)
    else:
        attempts += 1
        print("Waiting SPI Buffer reset", end='\r')
        time.sleep(wait_reset_SPI_buffer)
    if attempts >= 3:
        logger.critical("Failed 4 times to stop the laser")
        return True # indicate that laser is still on
    else:
        logger.critical("Failed to stop the laser (transmission problem)")
        return True
return True

```

```

def read_DAC_power_status(item='all'):

```

```

    """

```

```

    Read the status of the Digital to Analog Converter as well as the Power Status

```

```

    Try only one time to read the byte(s)

```

```

    :param item: 'fan', 'laser', 'fanDAC', 'laserDAC', 'laser_switch', 'gain', 'auto_gain_toggle',
'all'

```

```

    :return: DAC power byte, 5 status bytes if argument is 'all'

```

"""

```
print("Reading DAC power status", end='\r')
if initiate_transmission(0x13):
    response = spi.xfer([0x13, 0x13, 0x13, 0x13, 0x13, 0x13])
    # cs_high()
    # spi.close() # close the serial port to let it available for another device
    # Avoid opening and closing ports too often.
    # Avoid getting "too much files opened" error after long running time
    time.sleep(0.5) # avoid too close communication

    if item == 'fan':
        logger.debug("DAC power status for " + str(item) + " is " + str(response[0]))
        return response[0]
    elif item == 'laser':
        logger.debug("DAC power status for " + str(item) + " is " + str(response[1]))
        return response[1]
    elif item == 'fanDAC':
        logger.debug("DAC power status for " + str(item) + " is " + str(response[2]))
        response = 1 - (response[2] / 255) * 100 # see documentation concerning fan pot
        logger.info("Fan is running at " + str(response) + "% (0 = slow, 100 = fast)")
        return response
    elif item == 'laserDAC':
        logger.debug("DAC power status for " + str(item) + " is " + str(response[3]))
        response = response[3] / 255 * 100 # see documentation concerning laser pot
        logger.debug("Laser is at " + str(response) + "% of its maximal power")
        return response
    elif item == 'laser_switch':
        logger.debug("DAC power status for " + str(item) + " is " + str(response[4]))
        return response[4]
    elif item == 'gain':
        response = response[5] & 0x01
        logger.debug("DAC power status for " + str(item) + " is " + str(response))
```

```

        return response
    elif item == 'auto_gain_toggle':
        response = response[5] & 0x02
        logger.debug("DAC power status for " + str(item) + " is " + str(response))
        return response
    elif item is 'all':
        logger.debug("Full DAC power status is " + str(list(response)))
        return response
    else:
        raise ValueError("Argument of 'read_ADC_power_status' is unknown, check your
code!")

else:
    print("Waiting SPI Buffer reset", end='\r')
    time.sleep(wait_reset_SPI_buffer)
    return False # indicate an error

def digest(data):
    """
    Calculate the CRC8 Checksum with the bytes received
    :param data: list containing an infinite number of bytes with which to calculate the
checksum
    :return: calculated checksum
    """
    crc = 0xFFFF

    for byteCtr in range(0, len(data)):
        to_xor = int(data[byteCtr])
        crc ^= to_xor
        for bit in range(0, 8):
            if (crc & 1) == 1:

```

```

        crc >>= 1
        crc ^= 0xA001
    else:
        crc >>= 1
    # log = "Checksum is " + str(crc)
    # logger.debug(log)
    return crc & 0xFFFF

```

```
def check(checksum, *data):
```

```
    """
```

```

    Check that the data received are correct, based on those data and the checksum given
    :param checksum: checksum sent by the sensor (the last byte in any transmission)
    :param data: bytes sent by the sensor, with which to calculate the checksum
    :return: True if data are corrects, False if they are not
    """

```

```
    to_digest = []
```

```
    for i in data:
```

```
        to_digest.extend(i)
```

```
    if digest(to_digest) == join_bytes(checksum):
```

```
        log = "Checksum is correct"
```

```
        logger.debug(log)
```

```
        return True
```

```
    else:
```

```
        log = "Checksum is wrong"
```

```
        logger.debug(log)
```

```
        return False

```

```
def convert_IEEE754(value):
```

```
    """
```

```

    Join bytes and convert them to float according to the IEEE754 encryption

```

:param value: list containing the two bytes to decrypt

:return: decrypted float

"""

value = join_bytes(value)

answer = struct.unpack('f', bytes(value))

return answer

def loading_bar(name, delay):

"""

Show a loading bar on the screen during a certain amount of time.

Make the user understand the software is doing/waiting for something

:param name: text to be shown on the left of the loading bar (waiting, sampling...)

:param delay: amount of time the system is waiting (seconds)

:return: nothing

"""

bar = IncrementalBar(name, max=(2 * delay), suffix='% (elapsed)s/' + str(delay) + ' seconds')

for i in range(2 * delay):

time.sleep(0.5)

bar.next()

bar.finish()

return

def PM_reading():

"""

Read the PM bytes only from the OPC-N3 sensor

Read the data and convert them in readable format, checksum enabled

Does neither start the fan nor start the laser

Recommended to use read_histogram() instead of this function

:return: List[PM 1, PM2.5, PM10]

```

"""

print("YOU SHOULD BETTER USE OPCN3.read_histogram()")
attempts = 1
while attempts < 4:
    if initiate_transmission(0x32):
        PM_A = spi.xfer([0x32, 0x32, 0x32, 0x32])
        PM_B = spi.xfer([0x32, 0x32, 0x32, 0x32])
        PM_C = spi.xfer([0x32, 0x32, 0x32, 0x32])
        checksum = spi.xfer([0x32, 0x32])
        # spi.close()

        PM1 = round(struct.unpack('f', bytes(PM_A))[0], 3)
        PM25 = round(struct.unpack('f', bytes(PM_B))[0], 3)
        PM10 = round(struct.unpack('f', bytes(PM_C))[0], 3)

        if check(checksum, PM_A, PM_B, PM_C):
            print("PM 1:", PM1, "mg/m3\t\tPM 2.5:", PM25, "mg/m3\t\tPM10:", PM10,
"mg/m3")
            time.sleep(0.5) # avoid too close SPI communication
            return [PM1, PM25, PM10]
        if attempts >= 4:
            log = "PM data wrong 3 consecutive times, skipping PM measurement"
            logger.critical(log)
            return ["error", "error", "error"]
        else:
            attempts += 1
            log = "Checksum for PM data is not correct, reading again (" + str(attempts) + "/3)"
            logger.error(log)
            time.sleep(0.5) # avoid too close SPI communication

def getPM(flushing_time, sampling_time, start_fan_laser=True):

```

```

"""
Get PM measurement from OPC-N3
Recommended to use get_data() instead of this function
:param flushing_time: time (seconds) during which the fan runs alone to flush the
sensor with fresh air
:param sampling_time: time (seconds) during which the laser reads the particulate
matter in the air
:return: List[PM1, PM2.5, PM10]
"""

print("YOU SHOULD BETTER USE OPCN3.read_histogram()")
try:
    if start_fan_laser:
        fan_on()
        time.sleep(flushing_time)
        laser_on()
        print("Starting sampling") # will be printed on the same line as "Laser is ON"
        time.sleep(sampling_time)
        PM = PM_reading()

        laser_off()
        fan_off()
    except SystemExit or KeyboardInterrupt: # to stop the laser and the fan in case of error
        or shutting down the program
        laser_off()
        fan_off()
        raise
    return PM

def read_histogram(sampling_period, delete_previous=True):

"""

```

Read all the available data from the OPC-N3

It first read the histogram to remove the old data remaining in the OPCN3 buffer

Then it let the sensor take sample during the defined sampling period

Finally it read a last time the histogram data returned by the sensor

It decode the bytes returned into readable format

It returns everything in a dictionary

:param: sampling_period: amount of time (seconds) during while the fan is running and laser is sampling

:return: Dictionary{"PM 1", "PM 2.5", "PM 10", "temperature", "relative humidity", "bin", "MToF", "sampling time",

"sample flow rate", "reject count glitch", "reject count longTOF", "reject count ratio",

"reject count out of range", "fan revolution count", "laser status"}

"""

```
logger.debug("Reading histogram...")
```

```
print("Reading histogram...", end='\r')
```

```
# Create a dictionary containing data to be returned in case of error
```

```
to_return = {
```

```
    "PM 1": "error",
```

```
    "PM 2.5": "error",
```

```
    "PM 10": "error",
```

```
    "temperature": "error",
```

```
    "relative humidity": "error",
```

```
    "sampling time": "error",
```

```
    "sample flow rate": "error",
```

```
    "reject count glitch": "error",
```

```
    "reject count long TOF": "error",
```

```
    "reject count ratio": "error",
```

```
    "reject count out of range": "error",
```

```
    "fan revolution count": "error",
```

```
    "laser status": "error",
```

```
    "bin 0": "error",
```

```

    "bin 1": "error",
    "bin 2": "error",
    "bin 3": "error",
    "bin 4": "error",
    "bin 5": "error",
    "bin 6": "error",
    "bin 7": "error",
    "bin 8": "error",
    "bin 9": "error",
    "bin 10": "error",
    "bin 11": "error",
    "bin 12": "error",
    "bin 13": "error",
    "bin 14": "error",
    "bin 15": "error",
    "bin 16": "error",
    "bin 17": "error",
    "bin 18": "error",
    "bin 19": "error",
    "bin 20": "error",
    "bin 21": "error",
    "bin 22": "error",
    "bin 23": "error",
    "bin 1 MToF": "error",
    "bin 3 MToF": "error",
    "bin 5 MToF": "error",
    "bin 7 MToF": "error"
}

```

```

# Delete old histogram data and start a new one

```

```

if delete_previous:

```

```

    if initiate_transmission(0x30):

```

```

    answer = spi.xfer([0x00] * 86)
    logger.debug("SPI reading is:\r" + str(answer))
    # spi.close()
    logger.debug("Old histogram in the OPC-N3 deleted, starting a new one")
else:
    logger.critical("Failed to initiate histogram, skipping this measurement")
    return to_return # indicate clearly an error in the data recording

delay = sampling_period * 2 # you must wait two times the sampling_period in order
that
# the sampling time given by the OPC-N3 respects your sampling time wishes
# first 5 seconds are with low gain, and the next seconds are with high gain
(automatically performed by OPC-N3)
print("                ", end='\r') # remove last line

# Reading the histogram delete all the data in the OPCN3's buffer
# If the checksum is wrong, seacanairy don't get the data as expected
# Nevertheless, OPCN3 clean its buffer and all data are lost
# So you must wait another x seconds to get sample
if not take_new_sample_if_checksum_is_wrong:
    loading_bar('Sampling PM', delay)

attempts = 1 # reset the counter for next measurement
while attempts < 4:
    # If the user want to take a nex sample in case the checksum is wrong (see
explanation above), then
    # the system must wait the required amount of time in the reading loop
    if take_new_sample_if_checksum_is_wrong:
        loading_bar('Sampling PM', delay)

    if initiate_transmission(0x30):
        # read all the bytes and store them in a dedicated variable

```

```

# see sensor documentation for more info

bin = spi.xfer([0x00] * 48)
MToF = spi.xfer([0x00] * 4)
sampling_time = spi.xfer([0x00] * 2)
sample_flow_rate = spi.xfer([0x00] * 2)
temperature = spi.xfer([0x00] * 2)
relative_humidity = spi.xfer([0x00] * 2)
PM_A = spi.xfer([0x00]*4)
PM_B = spi.xfer([0x00]*4)
PM_C = spi.xfer([0x00]*4)
reject_count_glitch = spi.xfer([0x00] * 2)
reject_count_longTOF = spi.xfer([0x00] * 2)
reject_count_ratio = spi.xfer([0x00] * 2)
reject_count_Out_Of_Range = spi.xfer([0x00] * 2)
fan_rev_count = spi.xfer([0x00] * 2)
laser_status = spi.xfer([0x00] * 2)
checksum = spi.xfer([0x00] * 2)

# spi.close()

# check that the data transmitted are correct by comparing the checksums
# if the checksum is correct, then proceed...

if check(checksum, bin, MToF, sampling_time, sample_flow_rate, temperature,
relative_humidity,
    PM_A, PM_B,
    PM_C, reject_count_glitch, reject_count_longTOF, reject_count_ratio,
reject_count_Out_Of_Range,
    fan_rev_count, laser_status):
    logger.debug("SPI reading is:\r" + str(bin) + " " + str(MToF) + " " +
str(sampling_time)
        + " " + str(sample_flow_rate) + " " + str(temperature) + " " +
str(relative_humidity)
        + " " + str(PM_A) + " " + str(PM_B) + " " + str(PM_C) + " " +

```

```

str(reject_count_glitch)

    + " " + str(reject_count_longTOF) + " " + str(reject_count_ratio) + " "
    + str(reject_count_Out_Of_Range) + " " + str(fan_rev_count)
    + " " + str(laser_status))

# return TRUE if the data are correct, and execute the below

# decode the bytes according to the IEEE 754 32 bytes floating point format into
decimals

# rounding until 2 decimals, as this is the accuracy of the OPC-N3 for PM values
PM1 = round(struct.unpack('f', bytes(PM_A))[0], 2)
PM25 = round(struct.unpack('f', bytes(PM_B))[0], 2)
PM10 = round(struct.unpack('f', bytes(PM_C))[0], 2)

# Create a list with the PM values
#pm_values = [PM1, PM25, PM10]
# Define the CSV file path
#csv_file = "/home/pi/Documents/pm_values.csv"
# Get the current time
#current_time = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
# Open the CSV file in append mode
#with open(csv_file, mode='a', newline='') as file:
    #writer = csv.writer(file)
    # Check if the file is empty (to write headers if needed)
    #is_empty = file.tell() == 0
    # Write the current time if the file is empty
    #if is_empty:
        # writer.writerow(["Time", "PM 1", "PM 2.5", "PM 10"]) # Write header row
    # Write the current time and PM values to the CSV file
    #writer.writerow([current_time] + pm_values)
#print("PM values saved to", csv_file)

print("PM 1:\t", PM1, " mg/m3", end="\t\t\t")

```

```

print("PM 2.5:\t", PM25, " mg/m3", end="\t\t\t")
print("PM 10:\t", PM10, " mg/m3")

relative_humidity = round(100 * (join_bytes(relative_humidity) / (2 ** 16 - 1)), 2)
temperature = round(-45 + 175 * (join_bytes(temperature) / (2 ** 16 - 1)), 2) #
conversion in °C

print("Temperature:", temperature, " °C (PCB Board)\t| \tRelative Humidity:",
relative_humidity,
      "%RH (PCB Board)")

sampling_time = join_bytes(sampling_time) / 100
print(" Sampling period:", sampling_time, "seconds", end="\t\t\t")
sample_flow_rate = join_bytes(sample_flow_rate) / 100
print(" Sampling flow rate:", sample_flow_rate, "mL/s |", round(sample_flow_rate
* 60, 2), "mL/min |",
      round(sample_flow_rate * 60 * 60 / 1000, 2), "L/h")
# This is the amount of air passing through the laser beam, not the total sampling
flow rate!

reject_count_glitch = join_bytes(reject_count_glitch)
print(" Reject count glitch:", reject_count_glitch, end="\t\t\t")
reject_count_longTOF = join_bytes(reject_count_longTOF)
print(" Reject count long TOF:", reject_count_longTOF)
reject_count_ratio = join_bytes(reject_count_ratio)
print(" Reject count ratio:", reject_count_ratio, end="\t\t\t")
reject_count_Out_Of_Range = join_bytes(reject_count_Out_Of_Range)
print(" Reject count Out Of Range:", reject_count_Out_Of_Range)
fan_rev_count = join_bytes(fan_rev_count)
print(" Fan revolutions count:", fan_rev_count, end="\t\t\t")
laser_status = join_bytes(laser_status)
print(" Laser status:", laser_status)

```

```

to_return = {
    "PM 1": PM1,
    "PM 2.5": PM25,
    "PM 10": PM10,
    "temperature": temperature,
    "relative humidity": relative_humidity,
    "sampling time": sampling_time,
    "sample flow rate": sample_flow_rate,
    "reject count glitch": reject_count_glitch,
    "reject count long TOF": reject_count_longTOF,
    "reject count ratio": reject_count_ratio,
    "reject count out of range": reject_count_Out_Of_Range,
    "fan revolution count": fan_rev_count,
    "laser status": laser_status,
    "bin 0": join_bytes(bin[0:1]),
    "bin 1": join_bytes(bin[2:3]),
    "bin 2": join_bytes(bin[4:5]),
    "bin 3": join_bytes(bin[6:7]),
    "bin 4": join_bytes(bin[8:9]),
    "bin 5": join_bytes(bin[10:11]),
    "bin 6": join_bytes(bin[12:13]),
    "bin 7": join_bytes(bin[14:15]),
    "bin 8": join_bytes(bin[16:17]),
    "bin 9": join_bytes(bin[18:19]),
    "bin 10": join_bytes(bin[20:21]),
    "bin 11": join_bytes(bin[22:23]),
    "bin 12": join_bytes(bin[24:25]),
    "bin 13": join_bytes(bin[26:27]),
    "bin 14": join_bytes(bin[28:29]),
    "bin 15": join_bytes(bin[30:31]),
    "bin 16": join_bytes(bin[32:33]),
    "bin 17": join_bytes(bin[34:35]),

```

```

"bin 18": join_bytes(bin[36:37]),
"bin 19": join_bytes(bin[38:39]),
"bin 20": join_bytes(bin[40:41]),
"bin 21": join_bytes(bin[42:43]),
"bin 22": join_bytes(bin[44:45]),
"bin 23": join_bytes(bin[46:47]),
"bin 1 MToF": MToF[0]/3,
"bin 3 MToF": MToF[1]/3,
"bin 5 MToF": MToF[2]/3,
"bin 7 MToF": MToF[3]/3,
}

```

```

print(" Bin number:\t", end="")
for i in range(0, 24):
    print(to_return["bin " + str(i)], end=" ")
print("") # go to next line
print(" MToF:\t\t", end="")

```

```

for i in range(0, 4):
    i = (i * 2) + 1
    print(to_return["bin " + str(i) + " MToF"], end=" ")
print("") # go to next line

```

```

if sampling_time > (sampling_period + 0.5): # we tolerate a difference of 0.5
seconds

```

```

    log = "Sampling period of the sensor was " \
        + str(round(sampling_time - sampling_period, 2)) + " seconds longer than
expected"
    logger.warning(log)

```

```

elif sampling_time < (sampling_period - 0.5):
    logger.warning("Sampling period of the sensor was "

```

```
        + str(round(sampling_period - sampling_time, 2)) + " seconds shorter  
than expected")
```

```
    return to_return
```

```
else:
```

```
    # if the function with the checksum return an error (FALSE)
```

```
    logger.warning(
```

```
        "Error in the data received (wrong checksum), reading histogram again... (" +  
str(attempts) + "/3)")
```

```
        logger.warning("Data received were:\n" + str(bin) + str(MToF) +  
str(sampling_time) +
```

```
            str(sample_flow_rate) + str(temperature) +
```

```
            str(relative_humidity) + str(PM_A) + str(PM_B) +
```

```
            str(PM_C) + str(reject_count_glitch) +
```

```
            str(reject_count_longTOF) + str(reject_count_ratio) +
```

```
            str(reject_count_Out_Of_Range) + str(fan_rev_count) +
```

```
            str(laser_status) + str(checksum))
```

```
    print("Waiting SPI Buffer reset", end='\r')
```

```
    time.sleep(wait_reset_SPI_buffer) # let some times between two SPI  
communications
```

```
    attempts += 1
```

```
else:
```

```
    logger.critical("Failed to read histogram (transmission initiation problem)")
```

```
    return to_return
```

```
if attempts >= 3:
```

```
    logger.error("Data were wrong 3 times (wrong checksum), skipping this histogram  
reading")
```

```
    logger.warning("Data received were:\n" + str(bin) + str(MToF) + str(sampling_time)  
+
```

```
        str(sample_flow_rate) + str(temperature) +
```

```

    str(relative_humidity) + str(PM_A) + str(PM_B) +
    str(PM_C) + str(reject_count_glitch) +
    str(reject_count_longTOF) + str(reject_count_ratio) +
    str(reject_count_Out_Of_Range) + str(fan_rev_count) +
    str(laser_status) + str(checksum))
print("Waiting SPI Buffer reset", end='\r')
time.sleep(wait_reset_SPI_buffer)
return to_return

```

```

def get_data(flushing_time, sampling_time, start_fan_laser=True):
    """
    Get all the possible data from the OPC-N3 sensor
    Start the fan, flush air during defined time, start the laser,
    sample the air during defined time, turn off the laser and the fan
    :param flushing_time: time during which the ventilator is running without sampling
                        to refresh the air inside the casing
    :param sampling_time: time during which the sensor is sampling
    :return: Dictionary{"PM 1", "PM 2.5", "PM 10", "temperature", "relative humidity", "bin",
    "MToF", "sampling time",
                        "sample flow rate", "reject count glitch", "reject count longTOF", "reject count
    ratio",
                        "reject count out of range", "fan revolution count", "laser status"}
    """
    # return "error" everywhere in case of error during the measurement
    (fan_on/laser_on/read_histogram...)
    # seacanairy.py need to find the items in the dictionary, if not if crash
    to_return = {
        "PM 1": "error",
        "PM 2.5": "error",
        "PM 10": "error",
        "temperature": "error",

```

"relative humidity": "error",
"sampling time": "error",
"sample flow rate": "error",
"reject count glitch": "error",
"reject count long TOF": "error",
"reject count ratio": "error",
"reject count out of range": "error",
"fan revolution count": "error",
"laser status": "error",
"bin 0": "error",
"bin 1": "error",
"bin 2": "error",
"bin 3": "error",
"bin 4": "error",
"bin 5": "error",
"bin 6": "error",
"bin 7": "error",
"bin 8": "error",
"bin 9": "error",
"bin 10": "error",
"bin 11": "error",
"bin 12": "error",
"bin 13": "error",
"bin 14": "error",
"bin 15": "error",
"bin 16": "error",
"bin 17": "error",
"bin 18": "error",
"bin 19": "error",
"bin 20": "error",
"bin 21": "error",
"bin 22": "error",

```

"bin 23": "error",
"bin 1 MToF": "error",
"bin 3 MToF": "error",
"bin 5 MToF": "error",
"bin 7 MToF": "error"
}

```

```

try: # necessary to put an except condition (see below)

```

```

    if start_fan_laser:
        if not fan_on():
            logger.critical("Skipping histogram reading")
            return to_return

    print("Flushing fresh air", end='\r')
    time.sleep(flushing_time / 2)

    if start_fan_laser:
        if not laser_on():
            logger.critical("Skipping histogram reading")
            fan_off()
            return to_return

    print("Flushing fresh air", end='\r')
    time.sleep(flushing_time / 2)

    to_return = read_histogram(sampling_time)
    laser_off()
    fan_off()
    # spi.close()
    return to_return

```

```

except KeyboardInterrupt, SystemExit): # in case of error AND if user stop the software
during sampling

```

```

    # Avoid that the laser and the fan keep running indefinitely if system crash
    print(" ") # go to the next line
    logger.info("Python instance has been stopped, shutting laser and fan OFF...")
    laser_off()

```

```
fan_off()
```

```
raise
```

```
def join_bytes(list_of_bytes):
```

```
    """
```

```
    Join bytes to an integer, from byte 0 to byte infinite (right to left)
```

```
    :param list_of_bytes: list [bytes coming from the spi.readbytes or spi.xfer functions]
```

```
    :return: bytes concatenated to an integer
```

```
    """
```

```
    val = 0
```

```
    for i in reversed(list_of_bytes):
```

```
        val = val << 8 | i
```

```
    return val
```

```
def set_fan_speed(speed_percent):
```

```
    """
```

```
    Set the sensor fan speed
```

```
    Reduce fan speed can decrease dust deposition in the sensor casing
```

```
    Argument in percent, calibrated from the slowest as possible to the fastest
```

```
    :param speed_percent: number between 0 and 100 (0 = slowest, 100 = fastest)
```

```
    :return: nothing
```

```
    """
```

```
    if speed_percent < 0 or speed_percent > 100:
```

```
        raise ValueError("Fan speed of OPC-N3 sensor must be a number between 0 and 100  
(0 = slowest, 100 = fastest)")
```

```
    value = int((45 + speed_percent / 100 * 55) / 100 * 255)
```

```
    # Personal investigations shows that the fan don't work below 45%
```

```
    # Formula makes a calculation to convert 0% as 45% --> easier for user input
```

```
    if initiate_transmission(0x42):
```

```

        reading = spi.xfer([0, value])

        logger.info("Fan speed is set on " + str(speed_percent) + " (0 = the slowest, 100 = the
fastest)")
    else:
        logger.error("Failed to set the fan speed")

```

```

def initialization_SPI():
    """
    Initialize the OPCN3 SPI system
    To be executed once after Seacanairy power up
    To be executed on time only after powering up the OPCN3
    :return: nothing
    """
    print("Initializing OPCN3 SPI...", end='\r')

```

```

# Make any communication to start the Sensor SPI
# Personal investigations shows that first communication is always lost
if initiate_transmission(0x3F):
    answer = spi.xfer([0x3F] * 60)
    string = ""
    for x in range(len(answer)):
        string += chr(answer[x])

    print("OPCN3 infostring: " + str(string) + "")

return

```

```

if __name__ == '__main__':
    # The code below runs if you execute this code from this file (you must execute OPC-

```

N3 and not seacanairy)

```
while True:
```

```
    logger.debug("Code is running from the OPC-N3 file itself, debug messages shown")
```

```
    # fan_on()
```

```
    # read_DAC_power_status('fan')
```

```
    # time.sleep(1)
```

```
    # laser_on()
```

```
    # read_DAC_power_status('laser')
```

```
    # time.sleep(1)
```

```
    # laser_off()
```

```
    # read_DAC_power_status('laser')
```

```
    # time.sleep(1)
```

```
    # fan_off()
```

```
    # read_DAC_power_status('fan')
```

```
    # print("sleep")
```

```
    # time.sleep(3)
```

```
get_data(2, 3)
```

```
print("sleep")
```

```
time.sleep(5)
```

Bijlage XVI

De aangepaste CO2.py-code

```
#!/home/pi/seacanairy_project/venv/bin/python3
"""
Library for the use of E+E Elektronik EE894 CO2 sensor via I2C communication
"""
# -----
# USEFUL VARIABLES
# -----

# get the time
import time
from datetime import date, datetime

# Get the errors
import sys

# Create folders and files
import os

# smbus2 is the new smbus, allow more than 32 bits writing/reading
from smbus2 import SMBus, i2c_msg

# 'SMBus' is the general driver for i2c communication
# 'i2c_msg' allow to make i2c write followed by i2c read WITHOUT any STOP byte (see
sensor documentation)

# logging
import logging
```

```

# yaml settings
import yaml

# progress bar during sampling
from progress.bar import IncrementalBar

# I2C address of the CO2 device
CO2_address = 0x33 # i2c address by default, can be changed (see sensor doc)

# emplacement variable
bus = SMBus(1) # make it easier to read/write to the sensor (bus.read or bus.write...)

# -----
# YAML SETTINGS
# -----

# Get current directory
current_working_directory = str(os.getcwd())

with open(current_working_directory + '/seacanairy_settings.yaml') as file:
    settings = yaml.safe_load(file)
    file.close() # close the file after use

store_debug_messages = settings['CO2 sensor']['Store debug messages (important
increase of logs)']

project_name = settings['Seacanairy settings']['Sampling session name']

measurement_delay = settings['CO2 sensor']['Amount of time required for the sensor to
take the measurement']

max_attempts = settings['CO2 sensor']['Number of reading attempts']

```

```

# -----
# LOGGING SETTINGS
# -----

# all the settings and other code for the logging
# logging = tak a trace of some messages in a file to be reviewed afterward (check for
errors fe)

def set_logger(message_level, log_file):
    # set up logging to file
    logging.basicConfig(level=message_level,
                        format='%(asctime)s %(name)-12s %(levelname)-8s %(message)s',
                        datefmt='%d-%m %H:%M',
                        filename=log_file,
                        filemode='a')

    logger = logging.getLogger('CO2 sensor') # name of the logger
    # all further logging must be called by logger.'level' and not logging.'level'
    # if not, the logging will be displayed as 'ROOT' and NOT 'OPC-N3'
    return logger

if __name__ == '__main__': # if you run this code directly ($ python3 CO2.py)
    message_level = logging.DEBUG # show ALL the logging messages
    # Create a file to store the log if it doesn't exist
    log_file = current_working_directory + "/log/CO2-debugging.log"
    if not os.path.isfile(log_file):
        os.mknod(log_file)
    print("CO2 Sensor DEBUG messages will be shown and stored in " + str(log_file) + "")
    logger = set_logger(message_level, log_file)
    # The following HANDLER must be activated ONLY if you run this code alone
    # Without the 'if __name__ == '__main__'' condition, all the logging messages are

```

displayed 3 TIMES

(once for the handler in CO2.py, once for the handler in OPCN3.py, and once for the handler in seacanairy.py)

define a Handler which writes INFO messages or higher to the sys.stderr/display (= the console)

```
console = logging.StreamHandler()
```

```
console.setLevel(message_level)
```

set a format which is simpler for console use

```
formatter = logging.Formatter('%(name)-12s: %(levelname)-8s %(message)s')
```

tell the handler to use this format

```
console.setFormatter(formatter)
```

add the handler to the root logger

```
logging.getLogger().addHandler(console)
```

else: # if this file is considered as a library (if you execute seacanairy.py for example)

if the user asked to store all the messages in 'seacanairy_settings.yaml'

if store_debug_messages:

```
    message_level = logging.DEBUG
```

if the user don't want to store everything

else:

```
    message_level = logging.INFO
```

Create a file to store the log if it doesn't exist yet

```
log_file = current_working_directory + "/" + project_name + "/" + project_name + "-
```

```
log.log"
```

```
logger = set_logger(message_level, log_file)
```

no need to add a handler, because there is already one in seacanairy.py

all further logging must be called by logger.'level' and not logging.'level'

if not, the logging will be displayed as ROOT and NOT 'CO2 sensor'

```
def loading_bar(name, delay):
```

```
    """
```

```
    Show a loading bar on the screen during a a certain amount of time
```

```
    Make the user understand the software is doing/waiting for something
```

```
    :param name: Text to be shown on the left of the loading bar (waiting, sampling...)
```

```
    :param length: Amount of time the system is waiting (seconds)
```

```
    :return: nothing
```

```
    """
```

```
    bar = IncrementalBar(name, max=(2 * delay), suffix='% (elapsed)s/' + str(delay) + ' '
seconds')
```

```
    for i in range(2 * delay):
```

```
        time.sleep(0.5)
```

```
        bar.next()
```

```
    bar.finish()
```

```
    return
```

```
def digest(buf):
```

```
    """
```

```
    Calculate the CRC8 checksum (based on the CO2 documentation example)
```

```
    :param buf: List of bytes to digest [bytes to digest]
```

```
    :return: checksum
```

```
    """
```

```
    # Translation of the C++ code given in the documentation
```

```
    crcVal = 0xff
```

```
    _from = 0 # the first item in a list is named 0
```

```
    _to = len(buf) # if there are two items in the list, then len() return 1 --> range(0, 1) = 2
```

```
    loops
```

```

for i in range(_from, _to):
    curVal = buf[i]

    for j in range(0, 8): # C++ stops when J is not < 8 --> same for python in range
        if ((crcVal ^ curVal) & 0x80) != 0:
            crcVal = (crcVal << 1) ^ 0x31

        else:
            crcVal = (crcVal << 1)

    curVal = (curVal << 1) # this line is in the "for j" loop, not in the "for i" loop

checksum = crcVal & 0xff # keep only the 8 last bits

return checksum

def check(checksum, data):
    """
    Check that the data transmitted are correct using the data and the given checksum
    :param checksum: Checksum given by the sensor (see sensor doc)
    :param List of bytes transmitted by the sensor before the checksum (see sensor doc)
    :return: True if the data are correct, False if not
    """
    calculation = digest(data)
    if calculation == checksum:
        logger.debug("CRC8 is correct, data are valid")
        return True
    else:
        logger.debug("CRC8 does not fit, data are wrong")
        logger.error("Checksum is wrong, sensor checksum: " + str(checksum) +
            ", seacanairy checksum: " + str(calculation) +

```

```

        ", bytes returned:" + str(data) + str(checksum))
if data[0] and data[1] == 0:
    logger.debug("Sensor returned 0 values, it is not ready, waiting a little bit")
    print("Sensor not ready, waiting...", end='\r')
    time.sleep(3)
return False

def status(print_information=True):
    """
    Read the status byte of the CO2 sensor
    !! It will trigger a new measurement if the previous one is older than 10 seconds
    :param: print_information: Optional: False to hide the messages
    :return: True if last measurement is OK, False if NOK
    """
    logger.debug("Reading sensor status")
    try:
        with SMBus(1) as bus:
            # reading = read_from_custom_memory(0x71, 1)
            reading = bus.read_byte_data(CO2_address, 0x71)
            # see documentation for the following decryption
            CO2_status = reading & 0b00001000
            temperature_status = reading & 0b00000010
            humidity_status = reading & 0b00000001
            if print_information: # if user/software indicate to print the information
                if CO2_status == 0:
                    logger.debug("CO2 status is OK")
                else:
                    logger.warning("CO2 status is NOK")
            if temperature_status == 0:
                logger.debug("Temperature status is OK")
            else:

```

```

        logger.warning("Temperature is NOK")
    if humidity_status == 0:
        logger.debug("Humidity status is OK")
    else:
        logger.warning("Humidity status is NOK")
    if CO2_status or humidity_status != 0:
        # Only CO2_status and humidity_status, because for no known reason
        temperature status is always NOK
    return False
else:
    # Everything is OK
    return True
except:
    logger.critical("Status reading failure")
    return True # try to go ahead in all cases

def getRHT():
    """
    Read the last Temperature and Relative Humidity measured, process the bytes, check
    checksum, convert in °C and %RH
    :return: Dictionary with the following items {"RH", "temperature"}
    """
    logger.debug("Reading RH and Temperature from CO2 sensor")

    write = i2c_msg.write(CO2_address, [0xE0, 0x00]) # see documentation, example for
    reading t° and RH
    read = i2c_msg.read(CO2_address, 6)

    attempts = 0 # trial counter for the checksum and the validity of the data received
    reading_trials = 0 # trial counter for the i2c communication

```

```

# In case there is a problem and it return nothing, return "error"
data = {
    "relative humidity": "error",
    "temperature": "error"
}

# all the following code is in a loop so that if the checksum is wrong, it start a new
measurement
while attempts <= max_attempts:

    while reading_trials <= max_attempts: # reading loop, will try again if the i2c
communication fails
        try: # SMBUS stop working in case of error, avoid the software to crash in case of
i2c error
            with SMBus(1) as bus:
                bus.i2c_rdwr(write, read)
            break # break the loop if the try has not failed at the previous line, jump to the
process of data

        except: # what happens if the i2c fails
            if reading_trials == max_attempts:
                logger.critical("i2c failure "
                    + str(max_attempts) + "consecutive times, skipping this RH and
temperature reading")
                return data # indicate clearly that data are wrong

            logger.error("i2c failure (" + str(sys.exc_info())
                + "), trying again... (" + str(reading_trials + 1) + "/" + str(max_attempts) + ")")
            reading_trials += 1 # increment of reading_trials
            time.sleep(3) # if transmission fails, wait a bit to try again (sensor is maybe busy)

# process the data given by the sensor

```

```

reading = list(read)

# if the two checksums are correct...
if check(reading[2], [reading[0], reading[1]]) and check(reading[5], [reading[3],
reading[4]]):
    # reading << 8 = shift bytes 8 times to the left, say differently, add 8 times 0 on the
    right
    temperature = round((((reading[0] << 8) + reading[1]) / 100 - 273.15, 2)
    relative_humidity = ((reading[3] << 8) + reading[4]) / 100

    print("Temperature is:", temperature, "°C", end="")
    print("\t| Relative humidity is:", relative_humidity, "%RH")

    # Create a dictionary containing all the data
    data = {
        "relative humidity": relative_humidity,
        "temperature": temperature
    }

    return data

else: # if one or both checksums are not corrects
    if attempts == max_attempts:
        logger.error("Data were wrong "
            + str(max_attempts) + " consecutive times, skipping this RH and
temperature reading")
        return data # indicate on the SD card that data are wrong

    else:
        attempts += 1
        logger.warning("Error in the data received (wrong checksum), reading data
again... ("
            + str(attempts) + "/" + str(max_attempts) + ")")

```

```
time.sleep(4) # avoid to close i2c communication
```

```
def getCO2P():
```

```
    """
```

```
    Read the last CO2 instant, CO2 average and pressure measurements, process the  
bytes, check checksum,
```

```
    convert in hPa and ppm
```

```
    :return: Dictionary containing the following items {"average", "instant", "pressure"}
```

```
    """
```

```
    logger.debug('Reading of CO2 and pressure')
```

```
    write = i2c_msg.write(CO2_address, [0xE0, 0x27]) # see documentation, reading of  
CO2 and pressure example
```

```
    time.sleep(0.1)
```

```
    read = i2c_msg.read(CO2_address, 9)
```

```
    attempts = 0 # trial counter for the checksum and the validity of the data received
```

```
    reading_trials = 0 # trial counter for the i2c communication
```

```
    # Create a dictionary containing the data, return "error" in case of error
```

```
    data = {
```

```
        "average": "error",
```

```
        "instant": "error",
```

```
        "pressure": "error"
```

```
    }
```

```
    # all the following code is in a loop so that if the checksum is wrong, it start a new  
measurement
```

```
    while attempts <= max_attempts:
```

```
        while reading_trials <= max_attempts: # reading loop, will try again if the i2c
```

communication fails

```
try: # SMBUS stop working in case of error, avoid the software to crash in case of  
i2c error
```

```
    with SMBus(1) as bus:
```

```
        bus.i2c_rdwr(write, read)
```

```
        break # break the loop if the try has not failed at the previous line, jump to the  
process of data
```

```
except: # what happens if the i2c fails
```

```
    if reading_trials == max_attempts:
```

```
        logger.critical("i2c failure "
```

```
                        + str(max_attempts) + " consecutive times, skipping CO2 and pressure
```

```
reading (" +
```

```
        str(sys.exc_info()) + ")")
```

```
        return data # indicate clearly that the data are wrong
```

```
    logger.error("i2c failure, trying again... (" + str(sys.exc_info()) + ")")
```

```
    reading_trials += 1 # increment of reading_trials
```

```
    print("Waiting 3 seconds...", end='\r')
```

```
    time.sleep(3) # if I2C comm fails, wait a little bit and try again (sensor is maybe  
busy)
```

```
# process the data given by the sensor
```

```
reading = list(read)
```

```
# if the two checksums are correct...
```

```
if check(reading[2], [reading[0], reading[1]]) and check(reading[5], [reading[3],  
reading[4]]) and check(
```

```
    reading[8], [reading[6], reading[7]]):
```

```
    pressure = (((reading[6]) << 8) + reading[7]) / 10 # reading << 8 = shift bytes 8 times  
to the left
```

```
    print("Pressure is:", pressure, "mbar")
```

```
CO2_average = (reading[0] << 8) + reading[1] # reading << 8 = shift bytes 8 times to
the left
```

```
print("CO2 average is:", CO2_average, "ppm", end="")
```

```
CO2_raw = (reading[3] << 8) + reading[4]
```

```
print("\t| CO2 instant is:", CO2_raw, "ppm")
```

```
data = {
    "average": CO2_average,
    "instant": CO2_raw,
    "pressure": pressure
}
```

```
return data
```

```
else: # if one or both checksums are not corrects
```

```
    if attempts == max_attempts:
```

```
        logger.error("Error in the data received (wrong checksum), skipping this CO2 and
pressure reading")
```

```
        return data # indicate clearly that the data are wrong
```

```
    else:
```

```
        attempts += 1
```

```
        logger.warning("Error in the data received (wrong checksum), reading data
again... (" +
```

```
            str(attempts) + "/" + str(max_attempts) + ")")
```

```
        time.sleep(3) # avoid too close i2c communication
```

```
def get_data():
```

```
    """
```

```
    Get all the available data from the CO2 sensor (CO2 instant/average, pressure,
```

temperature, humidity

:return: Dictionary containing the following items

{"pressure", "temperature", "CO2 average", "CO2 instant", "relative humidity"}

"""

Read status byte

attempts = 1

while True:

if status(True):

break

else:

print("Waiting for data to be ready...", end='\r')

time.sleep(2)

attempts += 1

if attempts >= 6:

print("Sensor not ready, trying to read...", end='\r')

break

Get CO2 and pressure

data1 = getCO2P()

Get RH and temperature

data2 = getRHT()

Append those two dictionary

data1.update(data2)

return data1

Settings

def internal_timestamp(new_timestamp=None):

"""

Read the internal sampling period of the CO2 sensor

To change the value, write it between the brackets (in seconds)

:param new_timestamp: None or empty to read, new value in seconds to change it

:return: Actual internal sampling period of the sensor

"""

```
if new_timestamp is not None: # if user write something as input in the brackets
    (arguments)
```

```
    if not 15 <= new_timestamp <= 3600:
```

```
        logger.warning("Sampling period should be a number between 15 and 3600
seconds (see sensor documentation)")
```

```
        to_write = new_timestamp * 10 # see sensor documentation
```

```
        msb_timestamp = (to_write & 0xFF00) >> 8
```

```
        lsb_timestamp = (to_write & 0xFF)
```

```
        reading = write_to_custom_memory(0x00, msb_timestamp, lsb_timestamp)
```

```
    else: # if user doesn't write anything between the brackets
```

```
        reading = read_from_custom_memory(0x00, 2)
```

```
if reading is not False: # read_from_custom_memory() returns False in case of error...
```

```
    # ...Python crash if it tries to make calculations with a boolean (True or False)
```

```
    measuring_time_interval = (reading[1] + reading[0] * 256) / 10
```

```
    if new_timestamp is None: # adapt the message in function of the wishes of the user
    (here he want to read)
```

```
        logger.info("Internal measuring time interval is " + str(int(measuring_time_interval))
+ " seconds")
```

```
    else: # (here he want to write)
```

```
        logger.info(
```

```
            "Internal measuring time interval set successfully on " +
str(int(measuring_time_interval)) + " seconds")
```

```
        return measuring_time_interval
```

```
    else:
```

```
        logger.error("Failed to change the internal timestamp to " + str(new_timestamp) + "
seconds")
```

```

def trigger_measurement(force=False):
    """
    Request a new CO2, t°, pressure and RH measurement IF the previous one is older than
    10 seconds
    Force to avoid the previous 10 seconds condition
    Same function as 'status()'
    :param: force: True to apply the function two consecutive times to be sure that the
    sensor is well
        synchronized with the seacanairy
        False to apply it once (during the main loop of the Seacanairy for example)
    :return: True or False if status if OK or NOK
    """
    print("Triggering a new measurement...", end='\r')

    # The sensor will not take a new sample if the previous one is older than 10 seconds
    sensor_status = status(False) # trigger new measurement

    if force: # if force is True
        if measurement_delay != 0: # if user/software want to wait for the data to be ready
            loading_bar("Waiting for sensor sampling", measurement_delay) # usually 10
seconds (see doc)
            # sensor documentation, let time to the sensor to perform the measurement

        # That way, we ensure that the sensor will trigger a new measurement RIGHT now
        sensor_status = status(False)
        loading_bar("Waiting for sensor sampling", measurement_delay)
        # sensor documentation, let time to the sensor to perform the measurement

    return sensor_status # same function as 'status()', but here we don't want to print the
status on the screen

```

```

def read_internal_calibration(item):
    """
    Read the internal temp_calib of a particular sensor item
    :param item: indicate which internal temp_calib to read: 'relative humidity',
    'temperature', 'pressure', 'CO2', 'all'
    :return: List containing the temp_calib settings [offset, gain, lower_limit, upper_limit]
    """
    if item == 'relative humidity':
        index = 0x01
        unit = "%RH"
        factor = 1 / 100
    elif item == 'temperature':
        index = 0x02
        unit = "Kelvin"
        factor = 1 / 100
    elif item == 'pressure':
        index = 0x03
        unit = "mbar"
        factor = 1 / 10
    elif item == 'CO2':
        index = 0x04
        unit = "ppm"
        factor = 1
    elif item == "all":
        for i in ['relative humidity', 'temperature', 'pressure', 'CO2']: # iterate this function for
each parameter
            read_internal_calibration(i)
            time.sleep(0.5) # avoid too close i2c communication
        return # exit the function once the iteration is finished
    else:

```

```

        raise TypeError("Argument of read_internal_calibration is wrong, must be: 'relative
humidity', "
                        "'temperature', 'pressure', 'CO2' or 'all'")

reading = read_from_custom_memory(index, 8)

if reading is False: # if read_from_custom_memory() function doesn't work, will return
False...
    logger.error("Failed to read the internal temp_calib of the CO2 sensor")
    return False # indicate error
print(reading)
offset = (reading[0] << 8 + reading[1]) * factor
gain = (reading[2] << 8 + reading[3]) / 32768
lower_limit = (reading[4] << 8 + reading[5]) # factor taken into account further
upper_limit = (reading[6] << 8 + reading[7]) # factor taken into account further

logger.info("Reading temp_calib for " + str(item) + ":")
logger.info("\tOffset: " + str(offset) + " " + str(unit))
logger.info("\tGain: " + str(gain))
if lower_limit == 0xFFFF:
    logger.info("\tNo last lower limit adjustment")
    lower_limit = 0
else:
    lower_limit += factor
    logger.info("\tLower limit: " + str(lower_limit) + " " + str(unit))
if upper_limit == 0xFFFF:
    logger.info("\tNo last upper minute adjustment")
    upper_limit = 0
else:
    upper_limit *= factor
    logger.info("\tUpper limit: " + str(upper_limit) + " " + str(unit))
return [offset, gain, lower_limit, upper_limit]

```

```

def read_from_custom_memory(index, number_of_bytes):
    """
    Read bytes from specified custom memory address in the CO2 sensor internal memory
    :param index: index of the data to be read (see sensor doc)
    :param number_of_bytes: number of bytes to read (see sensor doc)
    :return: list[bytes] from right to left
    """

    logger.debug("Reading " + str(number_of_bytes) + " bytes from customer memory at
index " + str(hex(index)) + "...")

    write = i2c_msg.write(CO2_address, [0x71, 0x54, index]) # usual bytes to send/write to
initiate the reading
    attempts = 1
    read = [] # avoid return issue

    while attempts < 4:
        try:
            with SMBus(1) as bus:
                bus.i2c_rdwr(write)
            read = i2c_msg.read(CO2_address, number_of_bytes)
            with SMBus(1) as bus:
                bus.i2c_rdwr(read)
            break # break the trial loop if the above has not failed
        except: # if i2c communication fails
            if attempts >= 3:
                logger.warning("i2c communication failed 3 times while writing to customer
memory, skipping reading")
                return False # indicate that the writing process failed, exit this function
            else:
                logger.error("i2c communication failed to read from customer memory (" +
str(attempts) + "/3)")

```

```

        attempts += 1

        print("Waiting 3 seconds...", end='\r')

        time.sleep(3) # avoid too close i2c communication, let time to the sensor, may
be busy

    reading = list(read)

    logger.debug("Reading from custom memory returned " + str(reading))

    return reading


def write_to_custom_memory(index, *bytes_to_write):
    """
    Write data to a custom memory address in the CO2 sensor internal memory
    :param index: index of the customer memory to write (see sensor doc)
    :param bytes_to_write: unlimited amount of bytes to write into the internal custom
memory at index (see sensor doc)
    :return: True (Success) or False (Fail)
    """

    logger.debug("Writing " + str(bytes_to_write) + " inside custom memory at index " +
str(hex(index)) + "...")

    crc8 = digest([index, *bytes_to_write]) # calculation of the CRC8 based on the index
number and all the bytes sent

    attempts = 1 # trial counter for writing into the customer memory
    cycle = 1 # trial counter for i2c communication

    try:
        with SMBus(1) as bus:
            write = i2c_msg.write(CO2_address, [0x71, 0x54, index, *bytes_to_write, crc8]) #
see sensor doc

            bus.i2c_rdwr(write)

            logger.debug("i2c writing succeeded")

            # i2c writing function worked, and sensor didn't replied a NACK on the SCK line

```

```

        # (see i2c working principle/theory)

except:
    logger.critical("i2c failure while writing to custom memory")
    return False

# check that the data are written correctly
time.sleep(0.3)
reading = read_from_custom_memory(index, len(bytes_to_write))
if reading == [*bytes_to_write]: # because reading returns a list
    logger.debug("Success in writing " + str(bytes_to_write) + " inside custom memory at
index " + str(index))
    return reading # indicate that the writing process succeeded

else:
    logger.error("Failed in writing " + str(bytes_to_write) + " inside custom memory at
index " + str(hex(index)))
    logger.debug("Value read is " + str(reading) + " in place of " + str(bytes_to_write))

# -----
# Test Execution
# -----

# __name__ = '__main__' indicate that the Python sheet has been executed directly
# in opposition with __name__ = '__CO2__' when the Python sheet is executed as a library
from another Python sheet

# What is below will be executed if user execute this Python code directly ($ python3
CO2.py)

# Code below is used to make trials to the CO2 sensor while developing

```

```

if __name__ == '__main__':
    now = datetime.now()
    logger.info("-----") # add a line in the log file
    logger.info("Launching a new execution on the " + str(now.strftime("%d/%m/%Y
%H:%M:%S")))

    print("Reading internal timestamp")
    internal_timestamp()
    trigger_measurement(force=True)

    while True: # unstopped loop
        get_data()
        print("waiting 10 seconds...")
        time.sleep(10) # wait 10 seconds

```