

iSCAPE Sensor Platforms Documentation

Smart Citizen sensor platform documentation for the iSCAPE project

Table of contents

1. Introduction	3
2. Sensor Analysis Framework	5
3. A modular tool for citizen action	9
4. Sensor Platform	10
5. Smart Citizen Kit	13
6. Smart Citizen Station	23
7. Troubleshooting	34
8. Use cases	38
9. Frequently asked questions	40
10. Components	45
11. Sensors	152
12. Guides	203
13. Sensor Analysis Framework	307
14. Sensor Platform	311

1. Introduction

1.1 Sections

- **Main:** Contains the **Smart Citizen Kit** and **Smart Citizen Station** documentation to help you use them.
- **Platform:** Contains all the documentation on the **online platform** where data is collected, stored and visualised.
- **Data Analysis:** Contains all the documentation on the **data post-processing framework** to obtain insights from the data calibrated by the sensors.
- **Use cases:** Contains documentation and **use cases** examples how to use our tools with your local community.
- **Guides:** Contains step-by-step **guides** for different features of the kit, how to get started, use the shell, or make some more advanced analysis of the sensor readings!

1.2 Guides

The documentation contains multiple guides as step-by-step tutorials to perform essential tasks as installing a kit or upgrading it's firmware as well.



Example guides

- Installing the Smart Citizen Kit
- Installing the Smart Citizen Station
- Installing the Smart Citizen Kit 1.0 / 1.1
- Onboarding new Sensors
- Uploading SD Card Data
- Update the Firmware
- Edit the Firmware
- Use Machine Learning to Create Models for Sensors Calibration

1.3 Open Source

Licensing

The entire project is released under open source licenses:-

- Hardware components: CERN Open Hardware License v1.2
- Core firmware: GNU GPL v3.0
- Software platform: GNU AGLP v3.0

Info

Check the **Source files** section for each component and explore the software source code and the hardware blueprints.

Source files

 Download the documentation

 Check the documentation source code

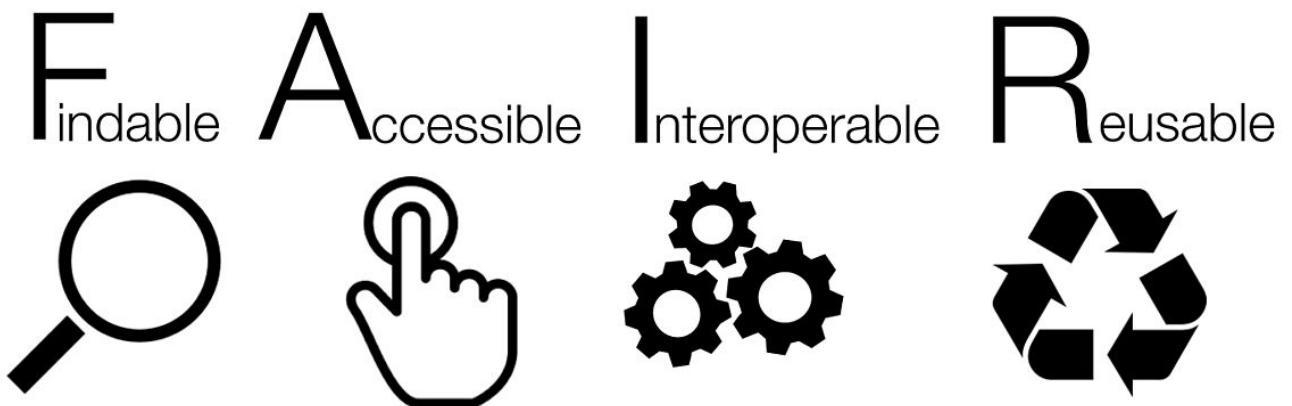
2. Sensor Analysis Framework

DOI 10.5281/zenodo.2566518

When dealing with sensor data, specially with low cost sensors, a great part of the **effort needs to be dedicated to data analysis**. After a careful data collection, this stage of our experiments is fundamental to extract meaningful conclusions and prepare reports from them. For this reason, we have developed a data analysis framework that we call the **Sensor Analysis Framework**. In this section, we will detail how this framework is built, how to install it, and make most use of it.

2.1 We care for open science

The framework is written in Python, and can be run using Jupyter Notebooks or Jupyter Lab. It is intended to provide an **state-of-the art data analysis environment**, adapted for the uses within the Smart Citizen Project, but that can be easily expanded for other use cases. The ultimate purpose of the framework, is to allow for reproducible research by providing a set of tools that can be replicable, and expandable among researchers and users alike, contributing to FAIR data principles.



By SangyaPundir - Own work, CC BY-SA 4.0

The framework integrates with the Smart Citizen API and helps with the analysis of **large amounts of data in an efficient way**. It also integrates functionality to **generate reports** in *html* or *pdf* format, and to **publish datasets** and documents to Zenodo.

More familiar with R?

R users won't be left stranded. R2PY provides functionality to send data from *python* to *R* quite easily.

Check the source code

2.2 How we use it

The main purpose of the framework is to make our lives easier when dealing with various sources of data. Let's see different use cases:

Get sensor data and visualise it

This is probably the most common use case: exploring data in a visual way. The framework allows **downloading data** from the **Smart Citizen API** or **other sources**, as well as to load local csv files. Then, different data explorations options are readily available, and not limited to them due to the great visualisation tools in python. Finally, you can generate html, or pdf **reports** for sharing the results.

examples

Check the example on how to load data from the API, and to make plots from it. Check here for an example about **reports** and it's result.

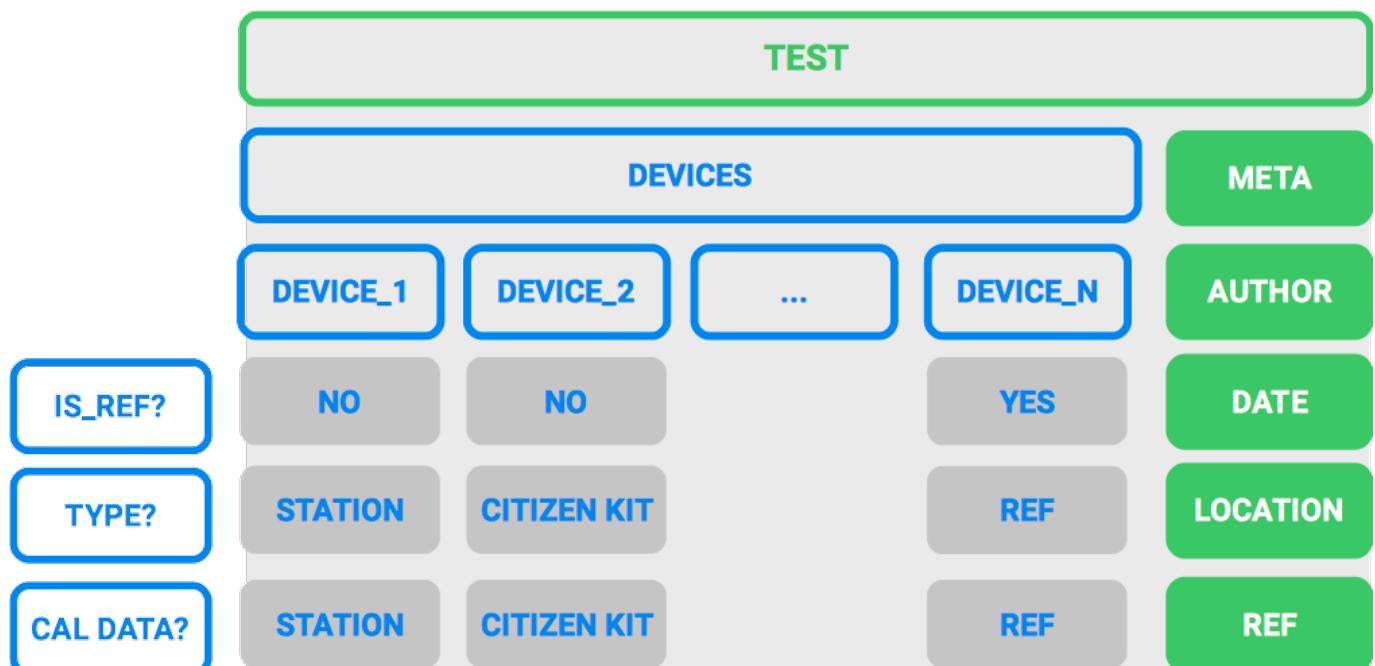
Organise your data in tests

Handling a lot of different sensors can be at times difficult to organise and have traceability. For this, we created the concept of *test*, which groups a set of devices, potentially from various sources. This is convenient since metadata can be added to the test instance describing, for instance, what was done, the calibration data for the device, necessary preprocessing for the data, etc. This test can be later loaded in a separate analysis session, modified or expanded, keeping all the data findable.

Some example metadata that can be stored would be:

- Test Location, date and author
- Kit type and reference
- Sensor calibration data or reference
- Availability of reference equipment measurement and type

A brief schema of the test structure is specified below:



examples

Follow the guide to organize your data in tests.

Clean sensor data

Sensor data never comes clean and tidy in the real world. For this reason, data can be cleaned with simple, and not that simple algorithms for later processing. Several functions are already implemented (filtering with convolution, Kalman filters, anomaly detection, ...), and more can be implemented in the source files.

Model sensor data

Low cost sensor data needs calibration, with more or less complex regression algorithms. This can be done at times with a simple linear regression, but it is not the only case. Sensors generally present non-linearities, and linear models might not be the bests at handling the data robustly. For this, a set of models are rightly implemented, using the power of common statistics and machine learning frameworks such as sci-kit learn, tensorflow, keras, and stats models.

Guidelines on sensor development

Check our guidelines on sensor deployment to see why this is important in some cases.

Batch analysis

Automatisation of all this tools can be very handy at times, since we want to spend less time programming analysis tools than actually doing analysis. Tasks can be programmed in batch to be processed automatically by the framework in an autonomous way. For instance, some interesting use cases of this could be:

- Downloading data from many devices, do something (clean it) and export it to .csv
- Downloading data and generate plots, extract metrics and generate reports for many devices
- Testing calibration models with different hyperparameters, modeling approaches and datasets

Check the guides

Visit [here!](#)

Share data

One important aspect of our research is to share the data so that others can work on it, and build on top of our results, validate the conclusions or simply disseminate the work done. For this, integration with zenodo is provided to share datasets and reports:

Info

Check one example in this notebook and this guide

2.3 Source files

[Download](#)

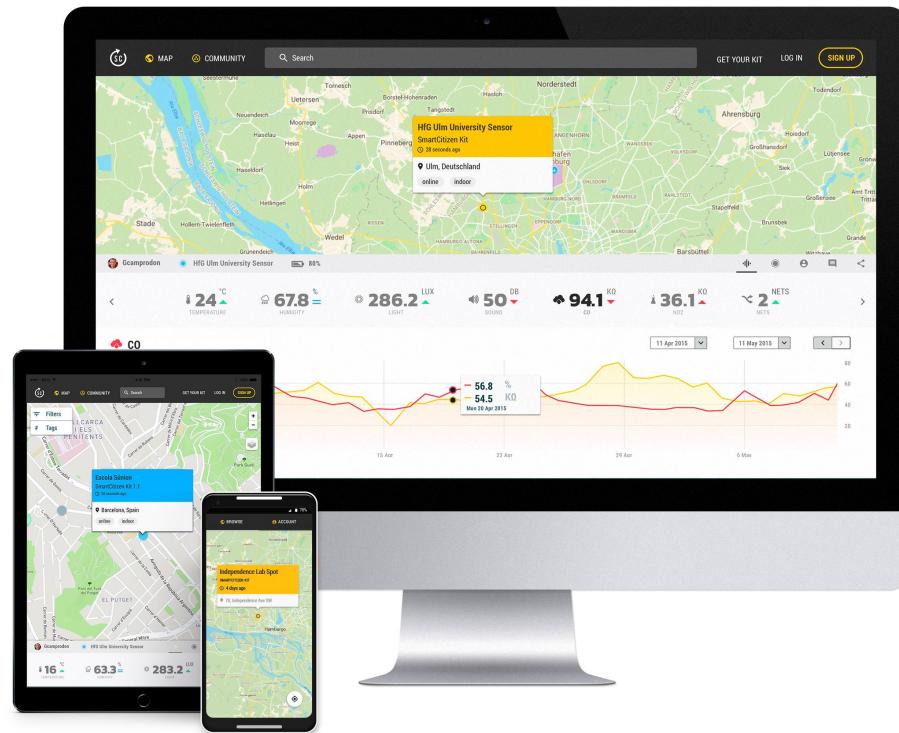
Check the source code

3. A modular tool for citizen action

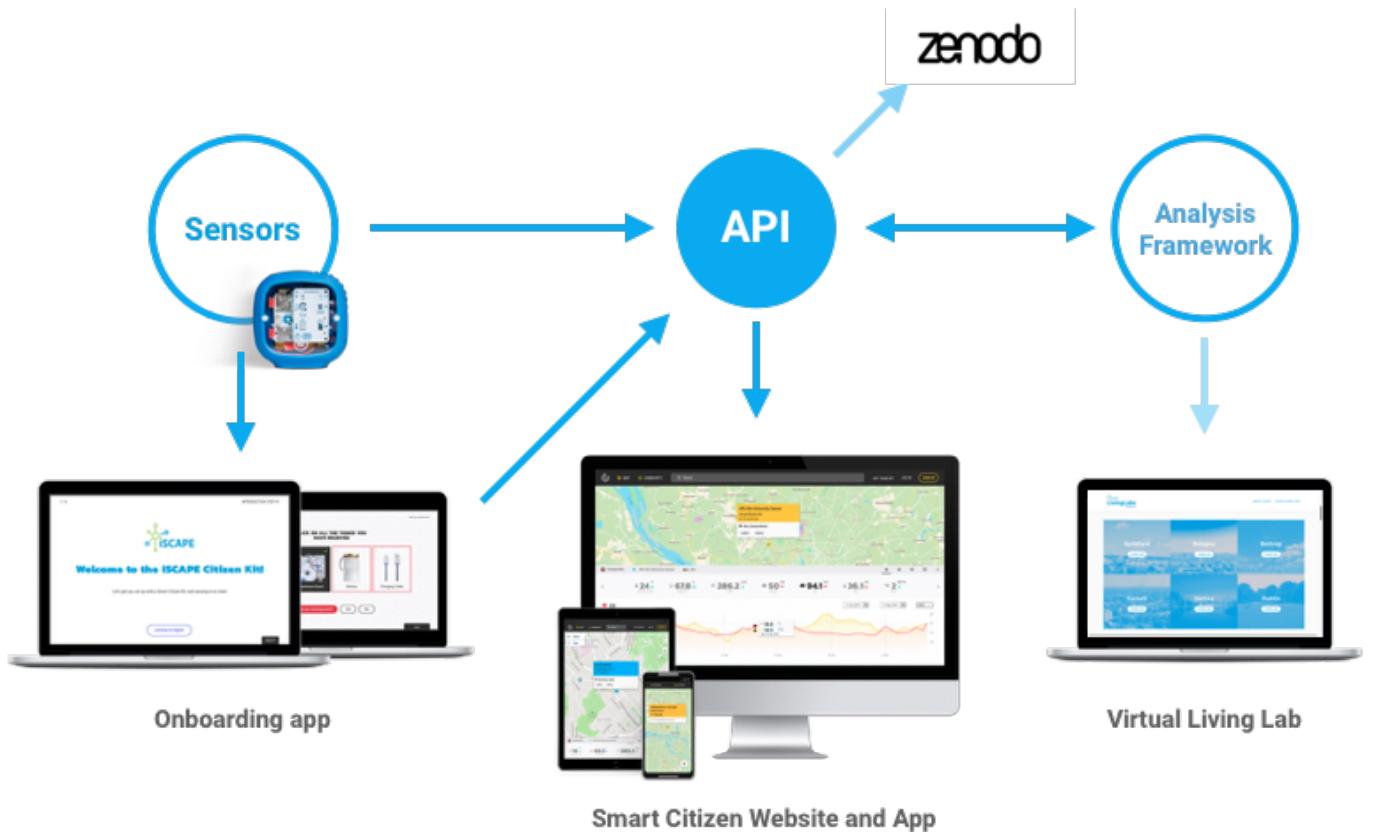
The guides in this section are aimed at creating a set of tools and resources around the SCK. This allows communities to develop their own sensing frameworks and strategies for participatory sensing. Find here guides that will help with making the best use of the Smart Citizen Kit, from a step-by-step guide on how to set up your kit to more advanced features like data analysis or using the SCK's shell.



4. Sensor Platform



The Smart Citizen platform supports the core features of the platform. That means this report documents new components, developed specifically for the project, but also existing components that already existed and made possible the platform.



We believe building modular and reusable software and using existing platforms is critical towards optimizing the research and development effort. By increasing the technology readiness levels of existing technologies, we can drastically improve the project exploitation strategy.

The previous requirements led to the decision of building the core platform on top of the existing Smart Citizen Platform. The platform is a front and backend solution for ingesting, storing and interacting with public data with a particular focus on crowd sensing applications.

Check the guides

We prepared a series of guides to help you on the most common features you will use

- Onboarding Sensors
- Uploading SD Card Data
- Downloading data

Want to learn more?

Check the developers ready [API Documentation](#)

4.1 Software components

- **Smart Citizen Website:** It aims to provide a visual website where the project environmental sensors can be accessed in near real time to facilitate the exploration of data with other contextual data (maps, keywords) and processed reports. This is especially important towards citizens engaging at each local site having a sense of ownership over a technology intervention has been associated with sustained community engagement (*Balestrini et al. 2014*). The main instance its available at smartcitizen.me/kits. You can explore and contribute to the source. This is free software available under GNU Affero General Public License (AGPL).
- **Smart Citizen API:** The platform provides a REST interface for all the functionalities available on the Website. That allows applications to be developed on easily on top having access to all the features to create complex and rich tools. The main instance its available at api.smartcitizen.me. You can explore and contribute to the source. One examples of this tools is the Sensors Analysis Framework or the iSCAPE Virtual Living Lab, both developed during the iSCAPE project) This is free software available under GNU Affero General Public License (AGPL).
- **Onboarding app:** It aims to facilitate the process of sensor setup to ensure that users, irrespective of technical expertise, can install the sensors. It guides the user through the process of the setup using simple language and a friendly graphic language. It is built as a separate tool from the core Smart Citizen Webpage in order it can be customized for each deployment. It exchange data with the core platform using the Smart Citizen API. The main instance its available at start.smartcitizen.me. There are also customized instances for specific projects such us onboarding.iscape.smartcitizen.me or start.decode.smartcitizen.me. You can explore and contribute to the source. This is free software available under a MIT License.

4.2 Source files

Check the source code

5. Smart Citizen Kit

A note about versions

The **SCK 2.0** was the development version for the now commercially available **SCK 2.1** sponsored thanks to the iSCAPE project under European Community's H2020 Programme under Grant Agreement No. 689954

Quick links

[Buy: seeedstudio.com](#)

[Installation: start.smartcitizen.me](#)

[Platform: smartcitizen.me](#)

[Discuss: forum.smartcitizen.me](#)

[Support: support@smartcitizen.me](#)

5.1 What is it?

The Smart Citizen Kit is the core of what we call the Smart Citizen System: a complete set of **modular hardware components** aiming to provide tools for **environmental monitoring**, ranging from **citizen science** and **educational activities** to more **advanced scientific research**. The system is designed in a extendable way, with a central data logger (the Data Board) with network connectivity to which the different components are branched. The system is based on the principle of reproducibility, also integrating non-hardware components such as a dedicated Storage platform and a Sensor analysis framework.

On top of that, the system is meant to serve as a **base solution for more complex settings**, not only related with air quality monitoring. For that purpose, in addition to the Urban Board, the system also provides off-the-shelf support for a wide variety of third party sensors, using the expansion bus as a common port. One example is what we call the Smart Citizen Station: a full solution for low cost air pollution monitoring.

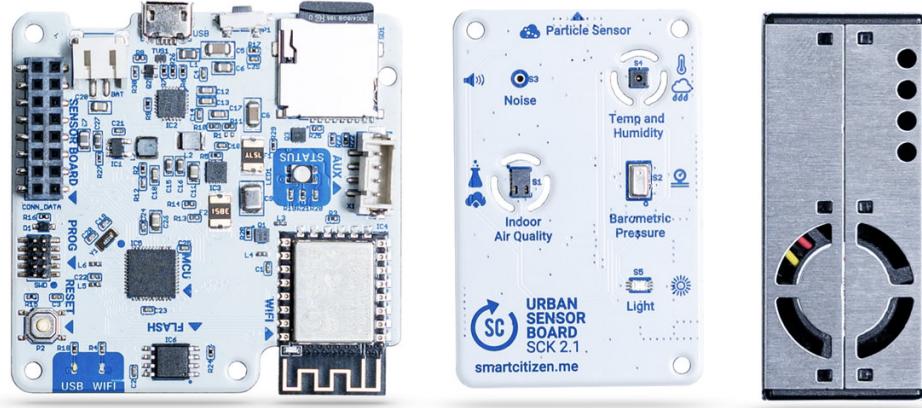
The sensors

Have a look at the supported sensors in the Firmware!

5.2 Measurements

All the Smart Citizen Kit new sensors generation measure **at least** air temperature, relative humidity, noise level, ambient light, barometric pressure and particulate matter (PM).

5.2.1 SCK 2.1



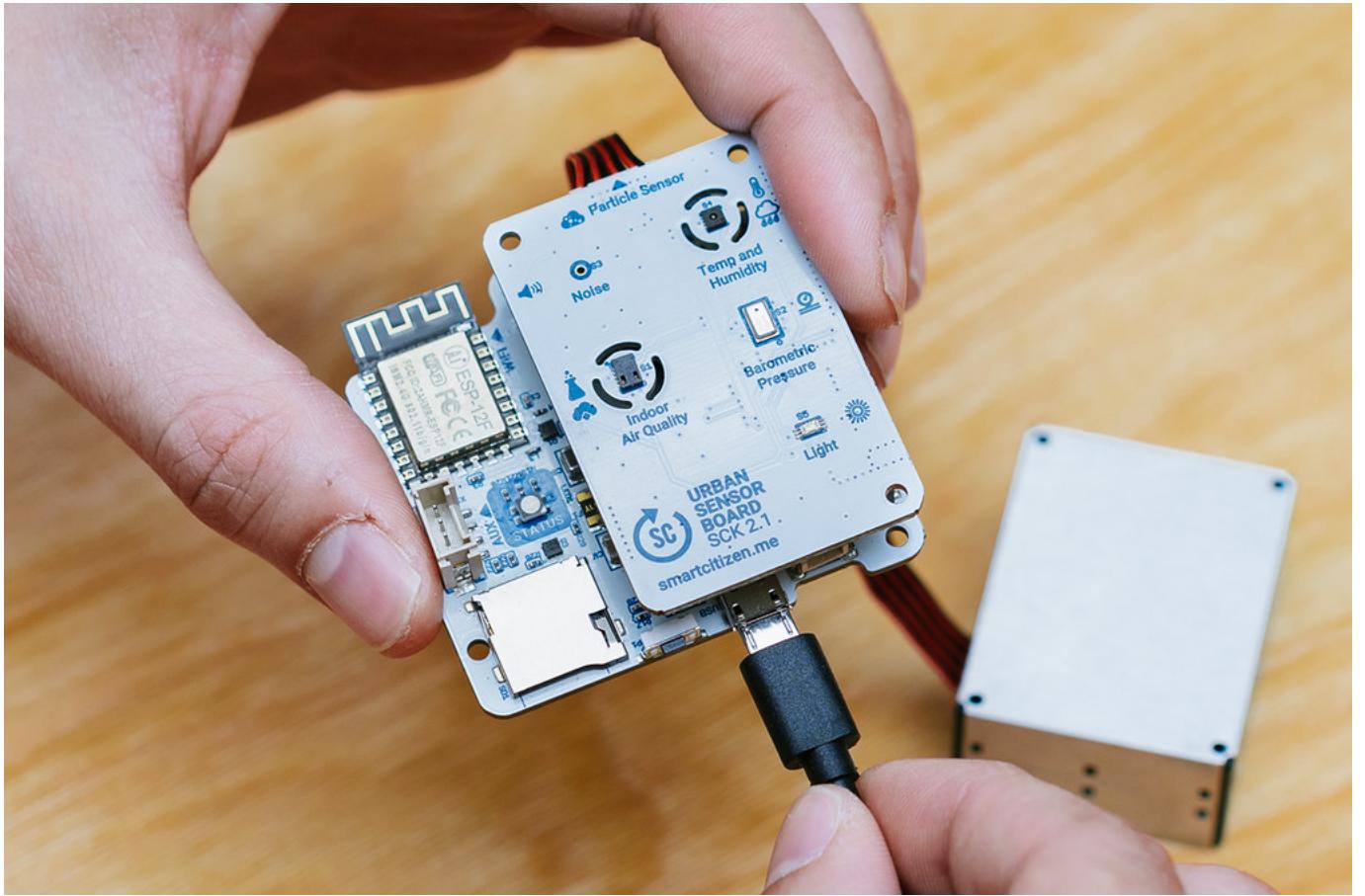
The SCK 2.1 components are listed below:

1. Smart Citizen Kit 2.1 with Particle Sensor and battery (brackets or rain-proof enclosure currently not included)
2. MicroSD card and microSD adapter to SD.
3. USB cable and a USB charger.

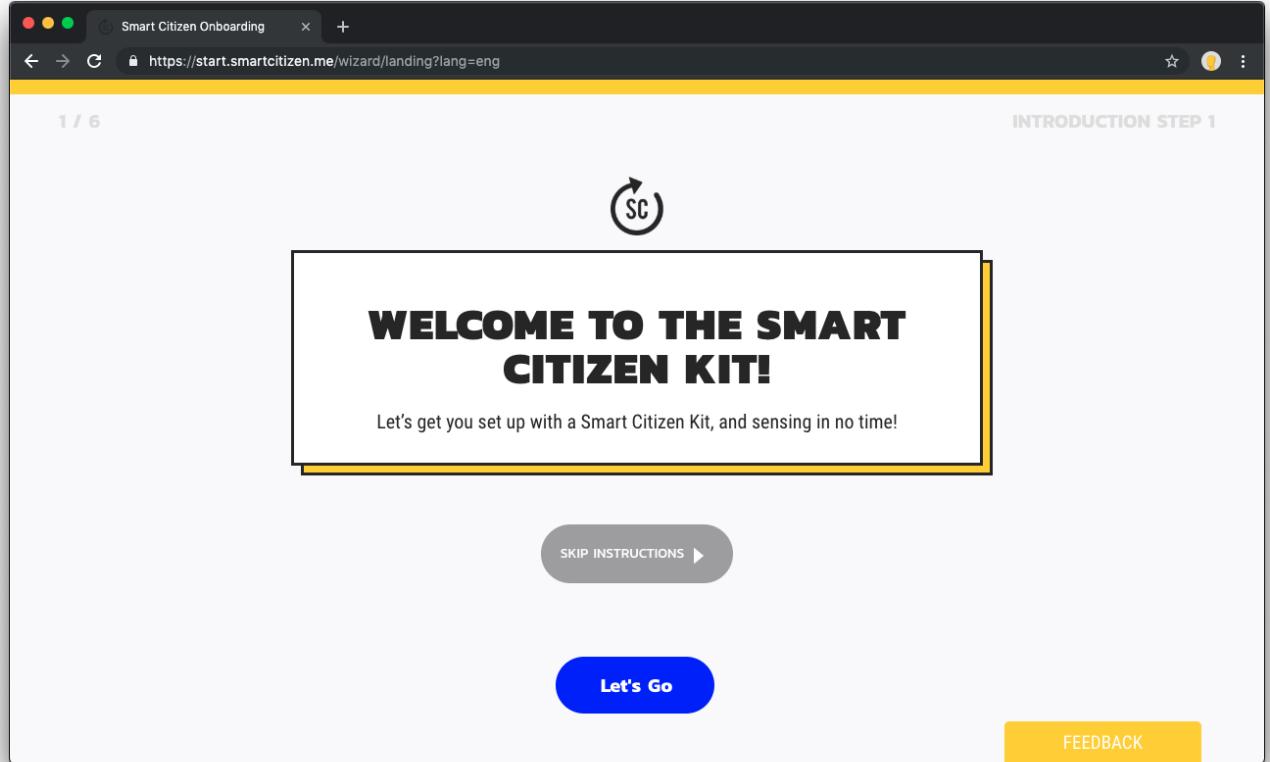
Measurement	Units	Sensors
Air temperature	°C	Sensirion SHT-31
Relative Humidity	% REL	Sensirion SHT-31
Noise level	dBA	Invensense ICS-434342
Ambient light	Lux	Rohm BH1721FVC
Barometric pressure	Pa	NXP MPL3115A26
Equivalent Carbon Dioxide	ppm	AMS CCS811
Volatile Organic Compounds	ppb	AMS CCS811
Particulate Matter PM 1 / 2.5 / 10	µg/m3	Planttower PMS 5003

5.3 Installation instructions

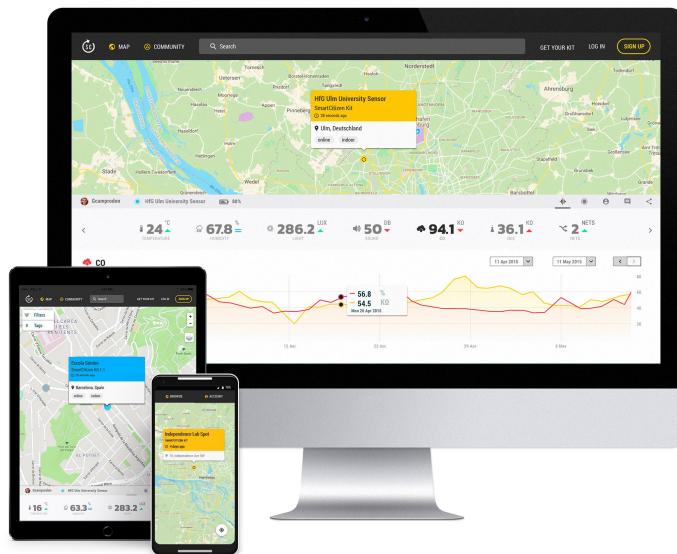
The sensor comes mounted and almost ready to be used:



The first step is to connect the battery. The kit will light in red (configuration mode) and we will be able to configure it by following the instructions at start.smartcitizen.me.



After the configuration process, data will be available on the SmartCitizen platform. You can explore the data there or download it using the [CSV Download](#) option (guide here)



5.4 Power management

5.4.1 Battery duration

The SCK comes with a 2000mAh LiPo battery. The battery is meant to be a complete power option for short-term measurements and a backup solution when the kit it is used for long periods. For long exposures, we recommend to permanently connect the USB to kit. The battery duration is dependent on which sensors are enabled or disabled:

- All sensors publishing over Wi-Fi: 12 hrs.
- All sensors publishing on SD card: 13 hrs.
- Without air quality sensors over Wi-Fi: 10 days
- Without air quality sensors on SD card: 25 days

You will note that the kit *turns itself off* while operating on battery. Actually, this is what we call `sleep-mode`, an operation mode implemented to reduce consumption while on battery operation.

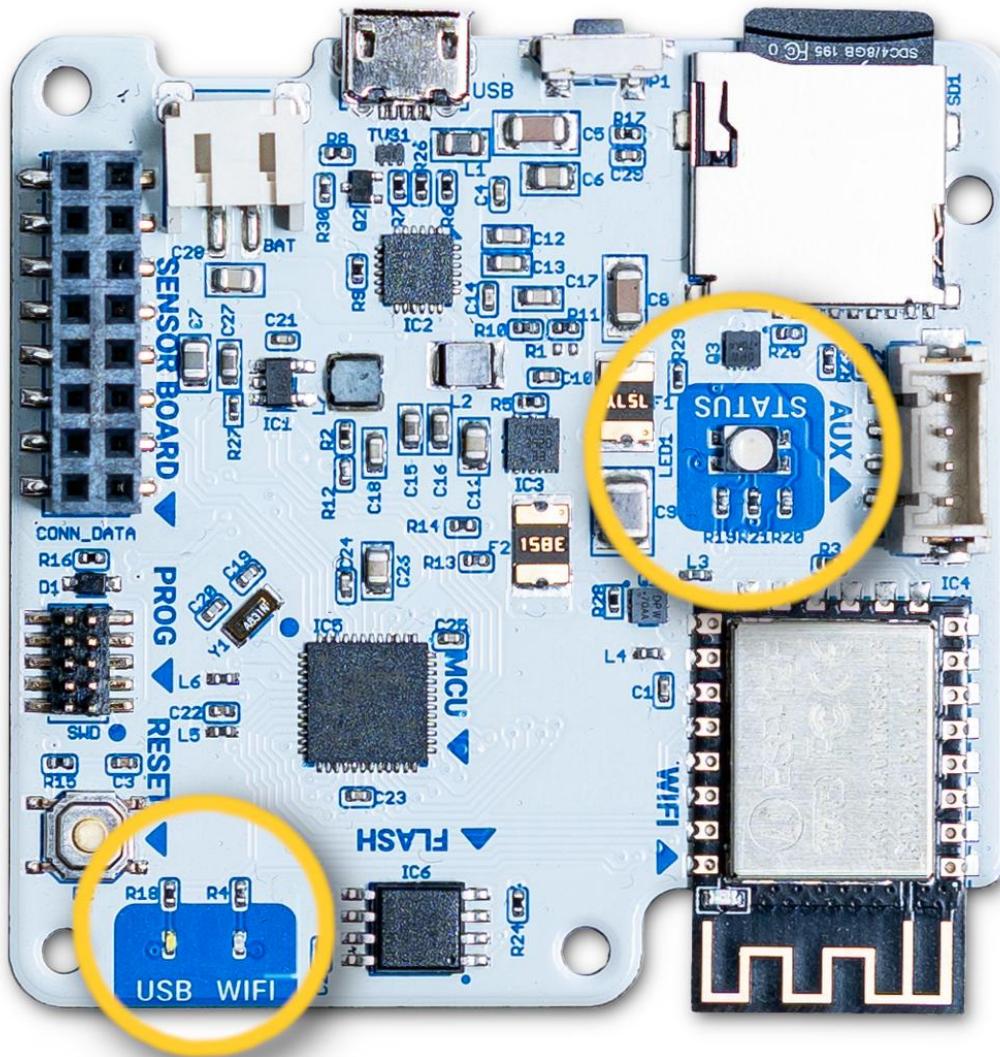
5.4.2 Battery charging

The SCK has a micro USB port and can be charged like any Smartphone or Tablet using a dedicated adapter or a computer USB port.

We recommend using a tablet power adaptor, instead of a computer USB port, for quicker charging. Autonomy can be extended by using a Power Bank, or a 5V PV Panel.

5.4.3 User feedback

The LED serves as an indication of the battery status. If the LED is flashing orange  it indicates that the battery must be charged. The battery takes about 4 hours to fully charge. When the battery is fully charged, the LED will change from orange to green .



Remember that in addition to these colors you will have the state color of the kit: configuration, network and sd.

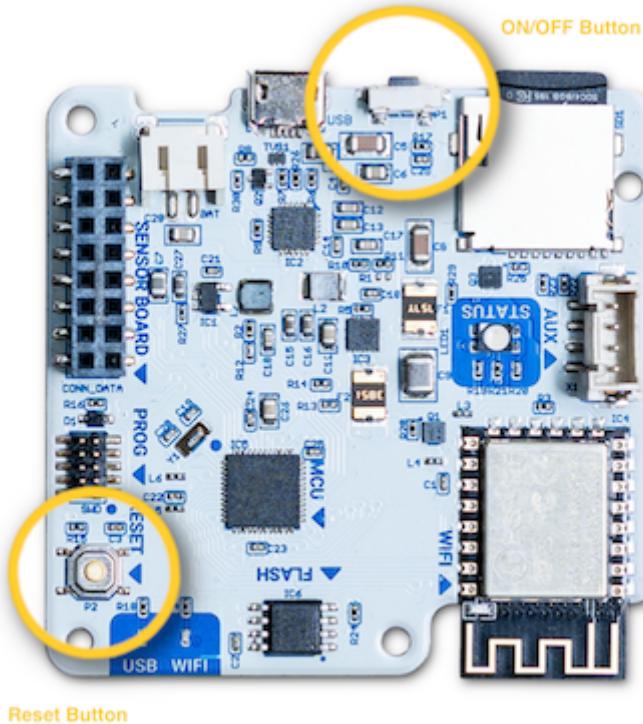
More details

Find more details under the data board section

5.5 User interfaces

The data board features a set of user interfaces which provide feedback to the user, as well as two buttons with different functionalities. The main RGB LED provides general feedback of the data board status. Additionally, two buttons are

provided for user action. A hardware reset button, which forces a power cut to the board, and a power button, used to change the device's mode, turn on and off the device, and to perform a factory reset. You can see both buttons below:

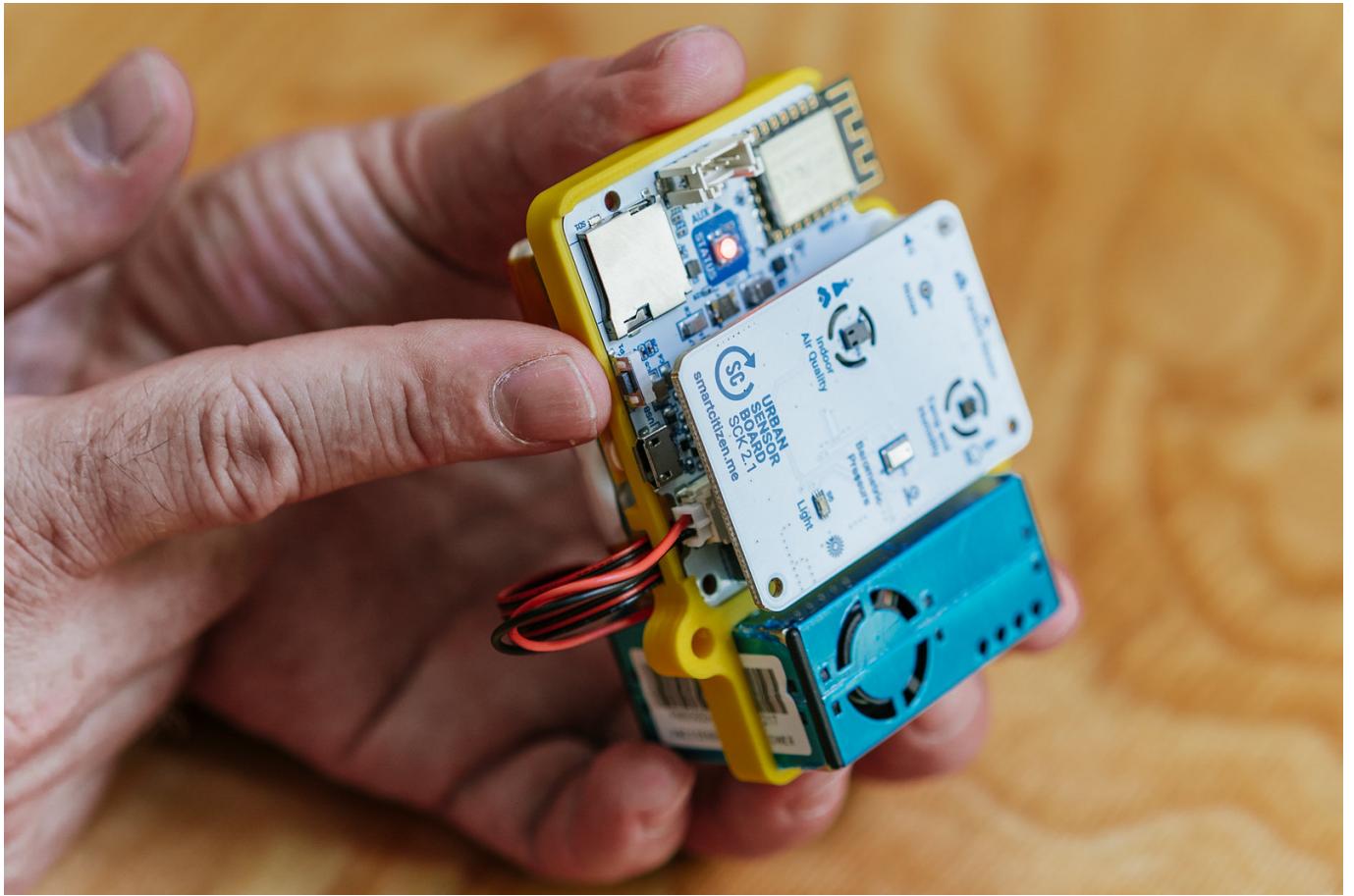


5.5.1 The button

The main button interaction is detailed below:

Function	Button action
ON	Push the button
OFF	Push the button for 5 seconds
CHANGE MODE	Push the button multiple times to choose: <i>Setup</i> <i>Wi-Fi</i> <i>Pink</i>
FACTORY RESET	Push the button 15 seconds for a full reset

An example is shown below:



Troubleshooting

Have a look at the troubleshooting section to check how you can use the buttons in case of problems with your SCK!

5.5.2 Operation modes

● Setup mode

In this mode, the Kit is ready to be configured in **network** mode or **SD card** in start.smartcitizen.me.

LED color	Kit status
Yellow	Ready to be setup
Green	Ready to be setup but battery is low, charge the Kit
Blue	Ready to be setup, battery charging
White	Ready to be setup, battery charged

● Wi-Fi mode

This is the standard mode for a network that requires a Wi-Fi connection. In this way, the device will publish the data every minute on the smartcitizen.me platform. If there is an inserted micro SD card, the data will be stored in duplicate.

LED color	Kit status
	👉 Collecting data online
	⚠ Error while collecting data
	🔋 Collecting data online but battery is low, charge the Kit
	⚡ Collecting data online, battery charging
	⚡ Collecting data online, battery charged

Warning

- ✓ The kit supports Wi-Fi WEP, WPA/WPA2 and open networks that are common networks in domestic environments and small businesses.
- ✗ But, it **does not** support WPA/WPA2 Enterprise networks such as EDUROAM or networks with captive portals such as those found in Airports and Hotels

● SD card mode (offline)

If we do not have an internet connection we can use the SD mode. In this case the device will record the data on the micro SD card. Later we can read the card using a card reader. The data can be visually spaced in a spreadsheet but also published on the smartcitizen.me platform using the **UPLOAD CSV** option.

LED color	Kit status
	👉 Collecting data offline
	⚠ Error while collecting data
	🔋 Collecting data offline but battery is low, charge the Kit
	⚡ Collecting data offline, battery charging
	⚡ Collecting data offline, battery charged

Guide

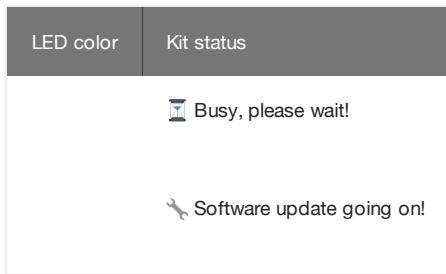
Check the guide on how to upload the sd card data here

Weird files?

The files in the sdcard have the following naming: YYYY-MM-DD.CSV, however, you will find in the some extra files (.01, .02...) These are data files that the sensor creates once there is a reset and, to avoid corruption, it creates a new file in the sd-card, by changing the file-extension.

A reset takes place every night at 3-4am with the purpose to avoid data loss because a problem. The SCK then stores the data in a file with a sequential name, and does so by changing the filename to YYYY-MM-DD.01, .02... etc depending on the amount of resets it sees during that day. You can see the data and work with it by changing the name from YYYY-MM-DD.01 to YYYY-MM-DD_01.CSV. Check the guide on how to organise your data to automatise this.

Especial status



5.6 Software Updates

Sofware updates are release frequently in the Firmware repository. These updates will need to be applied periodically to the two main components of the SCK: the SAMD21 (main processor) and the ESP8266 (Wi-Fi module). Check the instructions under the Update the Firmware section for more information.

6. Smart Citizen Station



The Smart Citizen Station was born with the idea to provide the iScape Living Labs with a system for monitoring the performance of their interventions. The Station aims at providing a solution that can be used by the Living Labs not just from a scientific point of view but also as a tool to engage local communities on air pollution related issues.



The station is designed with a modular principle where sensors can be added easily added expanding the capabilities of the installation or replaced when they are damaged or the sensors lifetime is over. From a costs perspective while being more expensive than the Smart Citizen Kit it is also conceived as a low-cost solution.



The design builds on top of the Smart Citizen Kit adding an extra set of more accurate sensors especially aimed at measuring air pollutants. The sensors include the Gas Sensor Board, featuring EC Carbon Monoxide, Nitrogen Dioxide and Ozone sensors and the PM Sensor Board, featuring a PM 2.5 / PM 10 sensor.

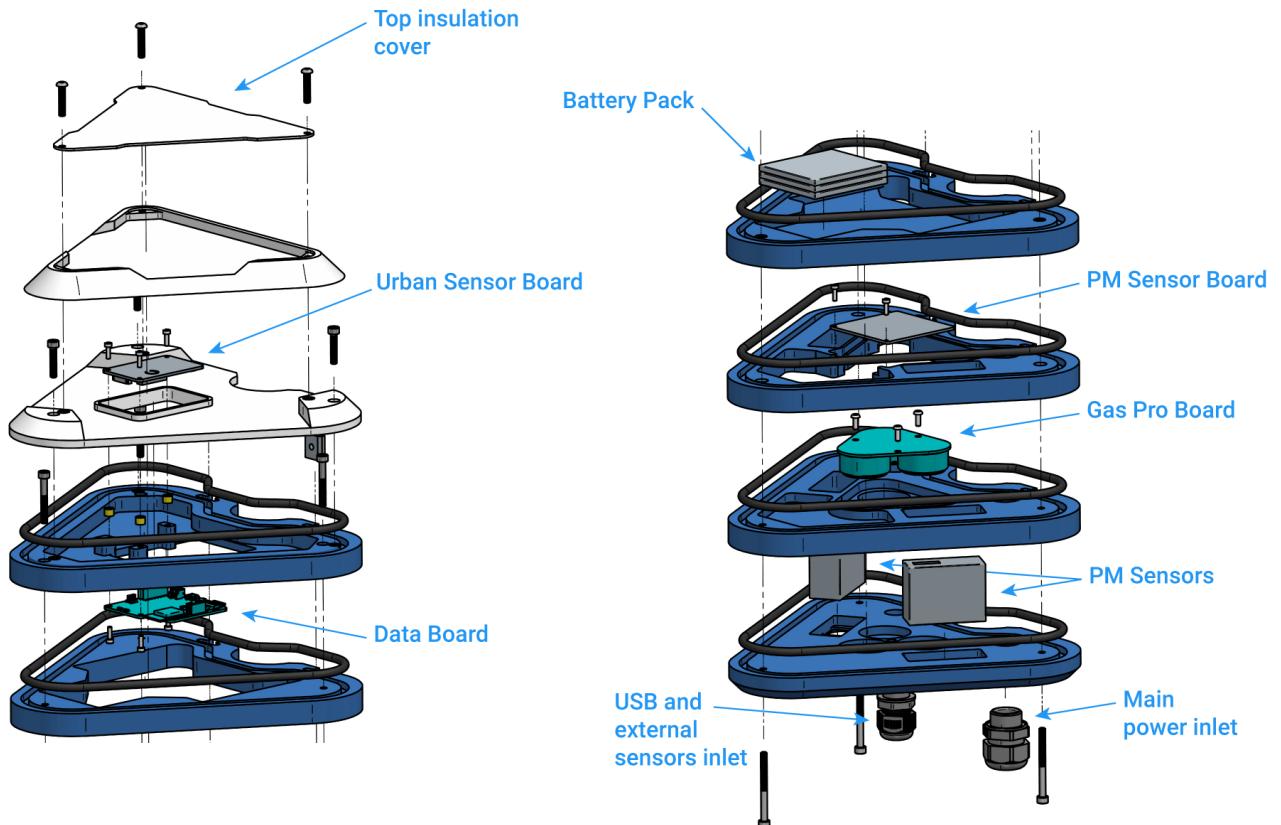
With all the sensor together this Kit provides information on Air Temperature, Relative Humidity, Noise Level, Ambient Light, Barometric Pressure, Particles Matter (PM 2.5 / 10), Carbon Monoxide, Nitrogen Dioxide and Ozone. The sensors are later described in detail in the document at the Sensor Components section.

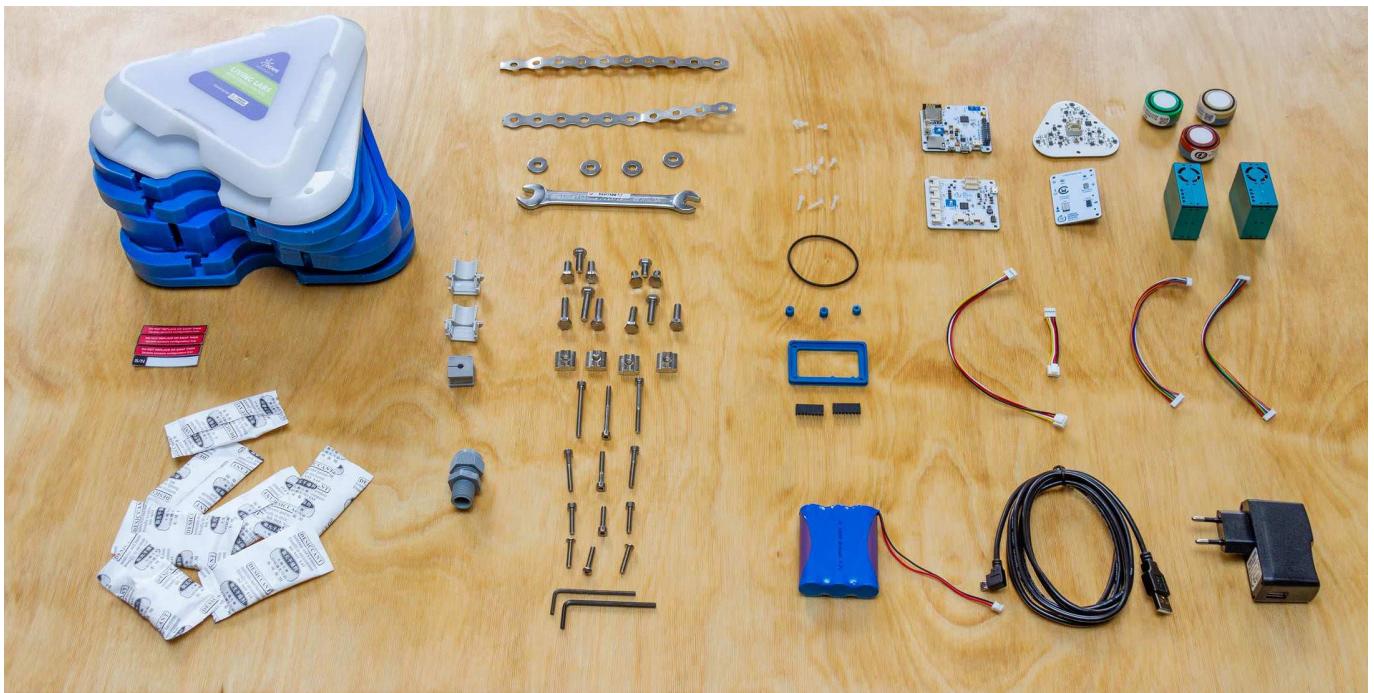
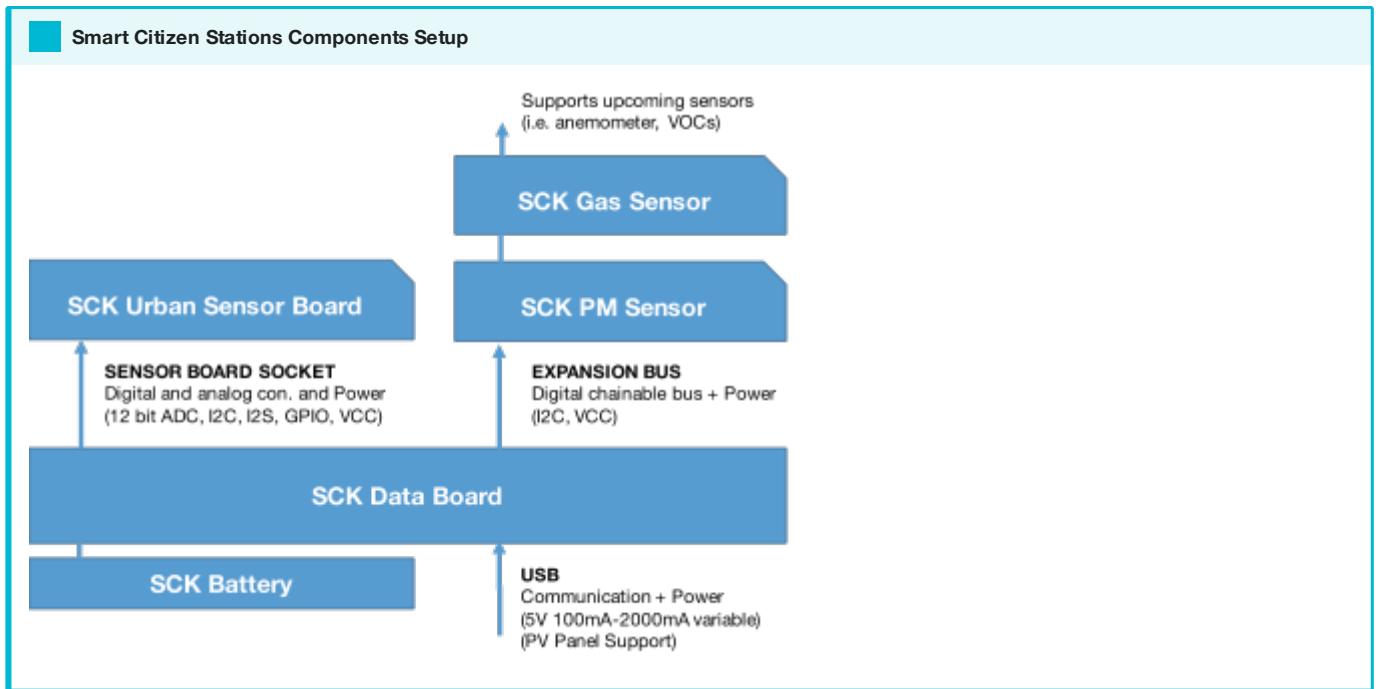
A note about versions

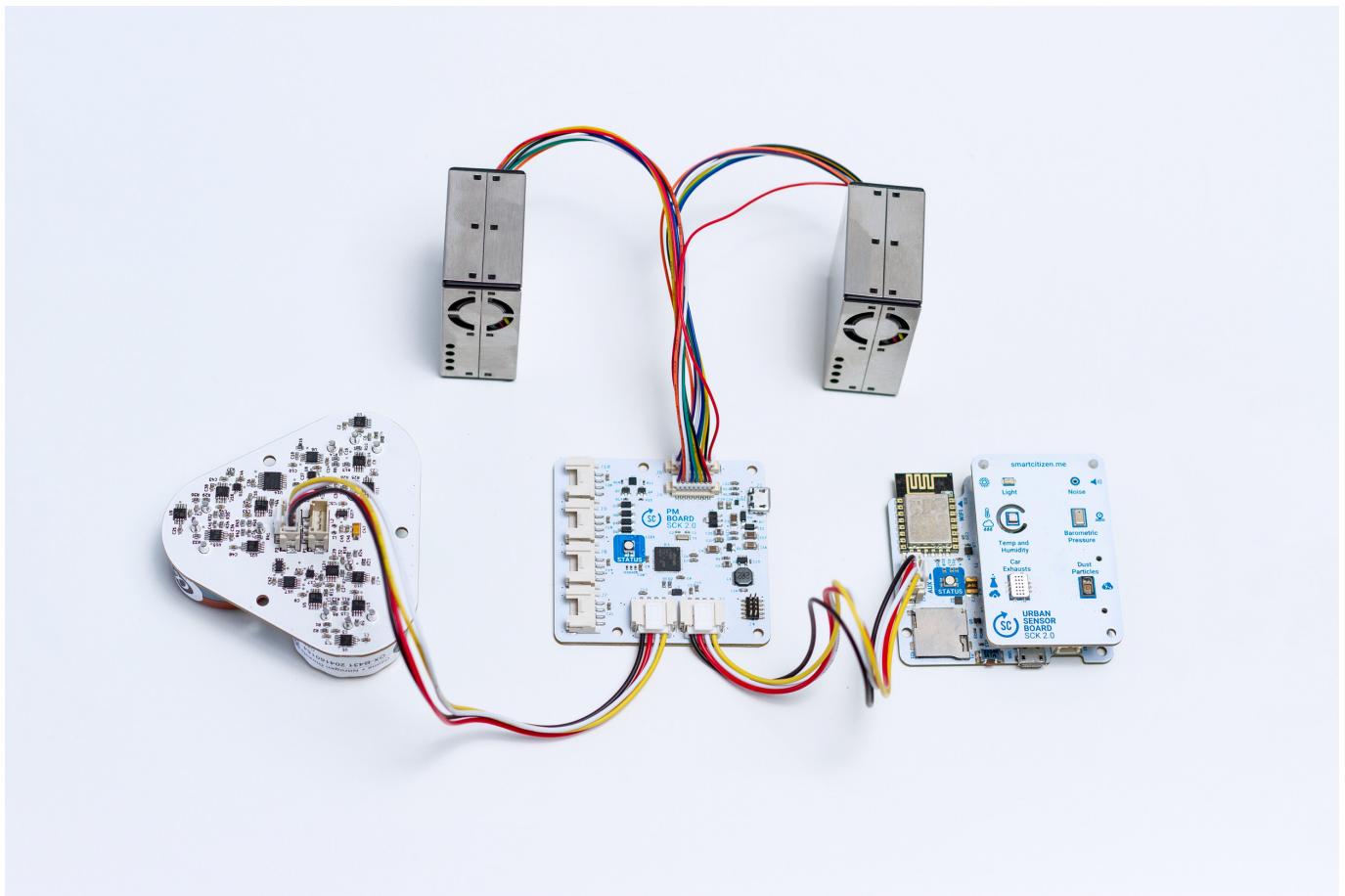
The **iScape Living Lab Station 1.0** was the development version for the 2.0 version. It was sponsored thanks to the iSCAPE project under European Community's H2020 Programme under Grant Agreement No. 689954

6.1 Components

The Station is a modular system based on different sensor board that connected to a central datalogger.







6.2 Sensors

Measurement	Units	Sensor	Component
Air Temperature	°C	Sensirion SHT-31	Urban Sensor Board
Relative Humidity	% REL	Sensirion SHT-31	Urban Sensor Board
Noise Level	dBA	Invensense ICS-434342	Urban Sensor Board
Ambient Light	Lux	Rohm BH1721FVC	Urban Sensor Board
Barometric pressure and AMSL	Pa and Meters	NXP MPL3115A26	Urban Sensor Board
Carbon Monoxide	ppm	Alphasense CO-B4	Gas Sensor Pro Board
Nitrogen Dioxide	ppb	Alphasense NO2-B43F	Gas Sensor Pro Board
Ozone	ppb	Alphasense OX-B431	Gas Sensor Pro Board
Gases Board Temperature	°C	Sensirion SHT-31	Gas Sensor Pro Board
Gases Board Rel. Humidity	% REL	Sensirion SHT-31	Gas Sensor Pro Board
PM 1	µg/m3	Plantower PMS5003 Dual System	PM Sensors Board
PM 2.5	µg/m3	Plantower PMS5003 Dual System	PM Sensors Board
PM 10	µg/m3	Plantower PMS5003 Dual System	PM Sensors Board

6.3 The Pack

Versions

Below the detailed list of components for the Smart Citizen Station V2.0. You can find more information regarding the iScape Living Lab Station V1.0, visit here.

- Smart Citizen Station

- Urban Board 2.1
- Data Board 2.1
- PM Board 2.0 + 2 PM sensors
- Gas Pro Board 2.0 with 3 EC sensors
- 6Ah Battery

- Accessories

- MicroSD card 512MB
- USB Charger
- MicroSD to SD card adapter
- Smart Citizen Power Supply (Traco P.S. 230AC in - DC5V out)
- 2m 3-Wire 220V cable
- Mounting brackets
- Mounting tools (1 x Allen Key)
- Enclosure
- Mounting bracket
- Thermoconformed Umbrella

6.4 Instructions

To start the installation simply visit the setup website stations.iscape.smartcitizen.me

INTRODUCTION STEP 3

THE ISCAPE LIVING LAB STATION

This sensor is a complete solution for environmental sensing. It measures sound, air quality, humidity, and lots of other things

[Continue](#)

[FEEDBACK](#)

Warning

We keep track internally of all sensor deployments and it is very important not to swap the internal components between Station to avoid mismatchs on the calibration data.

More info

Deployment considerations are listed here

6.4.1 Sensor considerations

Electrochemical sensor

The electrochemical sensors **need stabilisation time under the testing conditions** they will be at. It is important to set and power the sensors with sufficient time (1-2 days) on the test environment for them to adapt. The newer the sensor, the more stabilisation time it requires. For this deployment, you will be receiving brand new sensors.

Humidity and temperature extremes will require further sensor adaptation, in order to dry out or absorb the necessary humidity for their proper functioning.

Danger

Do not extract/attach the sensor capsule from the base board while powered, this could irreversibly damage the sensor.

Particle Sensor

The particle sensors measurements are delivered as averages of the two sensors with periodic validity checks. We are currently developing one-shot strategies for battery life improvement, but in the meantime, please make sure the sensor has reliable energy supply if you will use these sensors permanently.

6.5 Power

The kit has a battery life of 12 hours as is intended as a backup solution only. That's why a power supply needs to be installed as described below.

When we no longer want to publish or save more data for a few days we can turn off the kit. To do this, press the button for 5 seconds.

If the colors of the LED appear orange  indicates that the battery must be charged.

The battery takes about 4 hours to fully charge. When the battery is fully charged, change the orange to green .

Remember that in addition to the colors you will have the state color of the kit: configuration, network and sd.

6.5.1 Power supply

The Station can be directly powered at 220V AC (max consumption 5W).

Batteries

The Smart Citizen Station has a higher consumption than the kit, mostly due to the fans on the two PM sensors.

That means the internal battery last just for 20h, and it is only aimed at providing backup power.

For example, we can connect the station on the street light electric line, so the Station gets charged during the night when the lights are on.

Solar Panel

Unfortunately, we are having some problems with the PV Solar Panel system to power the Station independently. The system is currently under tests, and it will be available in the next few months.



Changing power supplies

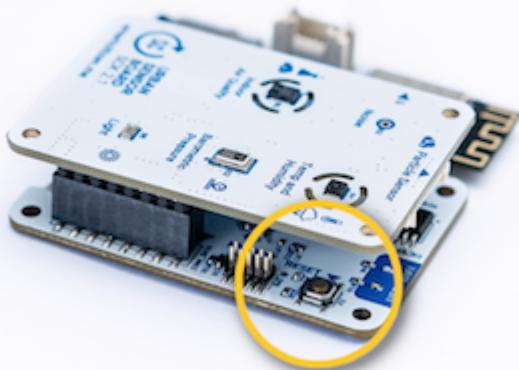
If you need to change power supplies (iScape Living Lab Station V1.0), please visit [here](#)

7. Troubleshooting



The magical reset button

Before trying anything else, the data board of your SCK comes with a very functional button that makes a hardware reset on the whole device. This is probably our best first try once the kit has any problem. You can see it here:



Some issues this might help solving:

- The kit hasn't been posting data for a while
- The kit doesn't respond to user interaction with the ON/OFF button
- The LED is fixed and does not react to anything
- ...

Pressing the reset button will not delete any configuration, it will simply restart your device. The light will go off and on and the device will start again with a white LED.

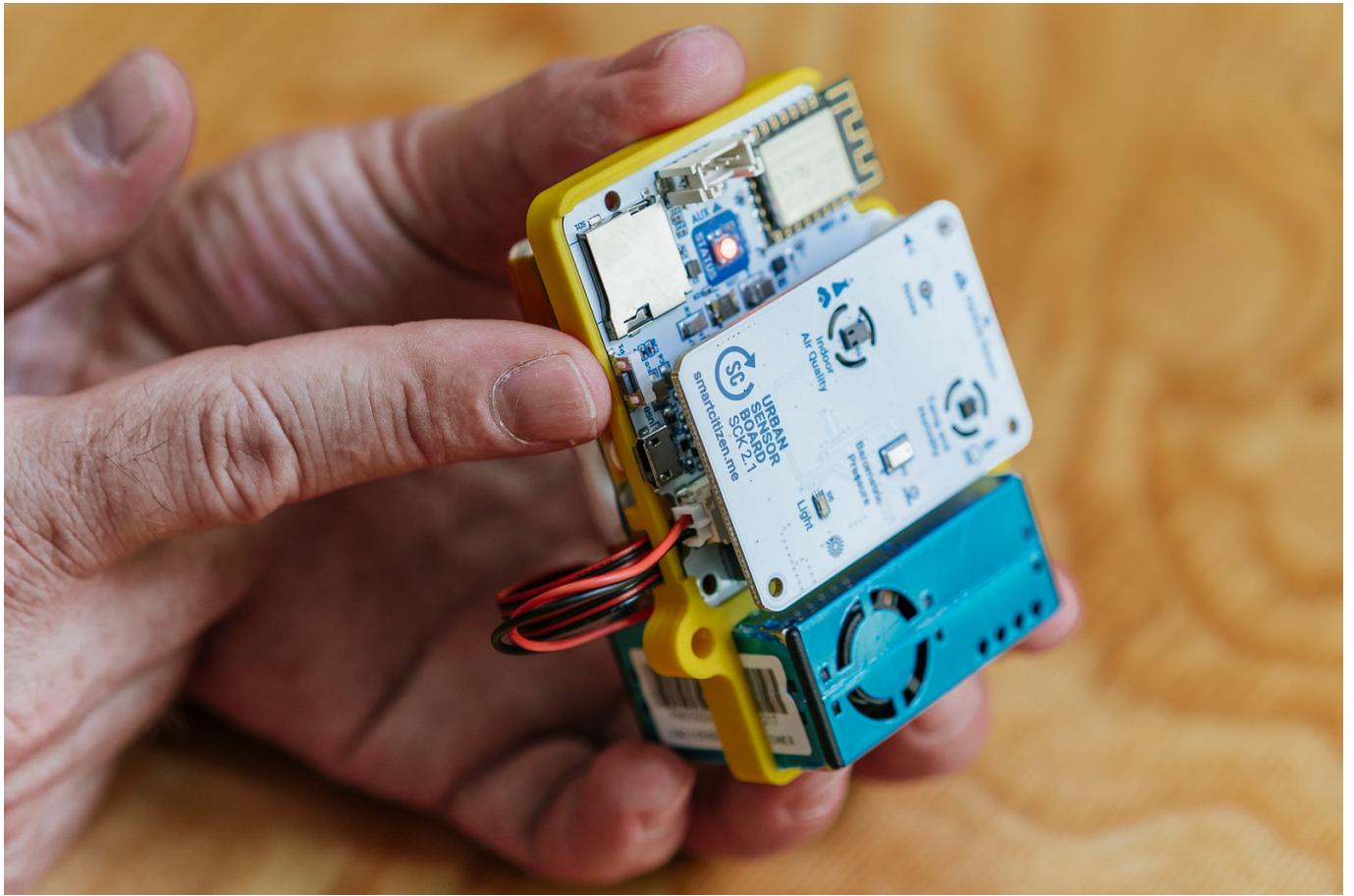
This button is also to be used when reflashing the firmware, by double clicking it. Have a look at the guide [here](#).

You can also perform a reboot by disconnecting the battery and the USB cable so that the kit is restarted. In this way we will not lose any data or configuration. However, if we are in `SD card mode`, the kit won't know *what time it is* and we will need to give it to him. For doing so:

- Press the ON/OFF button once. The LED should be breathing RED.
- Connect to the network `SmartCitizen[...]` and set it up again to log in `SD card mode`.

7.1 The network won't show up

Before configuring the Kit, if the `SmartCitizen[...]` network doesn't show up, make sure the LED is red. If not, press the button until the LED turns red.



7.2 Factory reset your kit

You can fully reset the Kit to the default settings so you can register again your device. Press the main button for **15 seconds**.

After 5 seconds the light will go off and will go on again after 15 seconds. Then you can release the button and your device will be fully resetted as a brand new Kit.

7.3 The LED does not turn on and the kit does not work

First of all, push the kit button. Maybe it's simply off.

If this does not work, most likely the kit has been left without battery. You will have to charge it using the USB charger. Any other mobile charger will also work.

We will know that it is charging when the LED emits orange pulses and once the battery is charged it will emit green pulses.

If the kit does not respond at all, it is probably worth trying with another USB cable, in case there is some problem there. If not, drop us an email or post on the forum

7.4 The kit does not store the data on the SD card

Some SD cards may have problems over time. We can try formatting it, but in case it does not work any micro SD card we buy at any mobile or computer store it will work. The size is not important and any micro SD or micro SDHC 512MB card up to 32GB will work.

7.5 The kit does not boot with the PMS Sensor

Make sure that you power the Smart Citizen Kit with a *good enough USB cable* and with an adaptor that can provide at least 1A. We have found some issues when powering the sensor with a thin cable, or from a weak power source, like a screen.

7.6 Known (fixed) issues

In this section, we will detail some problems you might have found in the early firmware versions of SCK 2.1.

7.6.1 Light sensor reads 0 and temperature/humidity sensor does not work

The issue is caused due to a firmware bug (light) and a problem with some SHT31 sensors (also fixed by firmware). A full explanation is detailed in the forum and the fix was released with V0.9.4 of the SAMD firmware.

8. Use cases

The Smart Citizen project has had the chance to be part of fantastic projects throughout its brief history. Ranging from public interventions in urban areas to the development of DIY sensors for agriculture, the team has carried out an enormous efforts to develop a technical platform that supports a wide range of communities on the creation of participatory sensing initiatives.



This section is a summary of these projects that might inspire you in future actions. They show the vast amount of possibilities offered by the Smart Citizen tools as key enablers of participatory sensing pilots and experiments.



By providing meaningful examples of novel appropriations and uses we seek to spire communities to conduct their own experiments and, hopefully, even their own custom tools.

Check our guides

All these projects left us with a great amount of experience and knowledge that we have compiled in this documentation. Find out how to use them in our guides section:

- Create interfaces for your data
- Use of third party sensors and adding data from other platforms
- DIY sensors for agriculture

9. Frequently asked questions



9.1 Can the sensors be placed outdoors?

Yes. The sensor is designed for both indoors and outdoors use. But if you're planning to use it outdoors, you will have to consider purchasing also a rainproof enclosure.

9.2 Can I make my own rainproof enclosure?

Of course! The manufacturing files for the 3D printed enclosure will be available to download in the Enclosures repository. Throughout the history of the Smart Citizen project, we've seen many inventive solutions for placing the sensor outdoors.

9.3 Can I charge the sensors with a solar panel?

Sure! But note that the sensor requires a 5V solar panel to work properly. Keeping that in mind, you can buy one of the photovoltaic panels that we provide, or run your own tests.

9.4 Can I add external sensors to the system?

Yes. The sensor has an independently configurable auxiliary bus at 3.3V with a SEEED Grove connector. The Bus has native support for I2C, but it can also be setup on firmware as a GPIO or UART. It can supply power up to 750mA, and it can be enabled or disabled by software.

9.5 What happens if there is a loss of network connectivity?

If the sensor is working in network mode and at any time the network is not available, it will store the data on its internal memory and publish all the collected data as soon as the network is available again.

9.6 Which external sensors can be added?

For the moment, the list of supported sensors includes some Atlas Scientific probes, some Seeed Grove sensors, and the Chirp moisture sensor, but the options are almost endless. We will add tutorials to use the additional sensors listed above in our documentation. However, in the short term we will only offer support for them via our custom hardware development services.

9.7 Will I be able to access the collected data?

Of course! The data collected by your sensor is available for anyone on the Smart Citizen Platform, and you can download it at any time as a CSV file. Besides, you can also use the API to built custom applications to interact with your device.

9.8 How does the kit record the data?

The sensor can work in network and SD card modes. In network mode, the sensor publish data to the SC platform over Wi-Fi every minute. In SD card mode, all the collected data is stored locally in CSV format, and it can be later uploaded manually to the platform using the “Manual Data Upload” option.

9.9 What networks does it support?

The SCK supports Wi-Fi WEP, WPA/WPA2 and open networks that are common networks in domestic environments and small businesses. However, like many other embedded devices such as Apple TV® or Chromecast®, it **does not** support networks with captive portals such as those found in Airports and Hotels. Currently, it also **does not** support WPA/WPA2 Enterprise networks such as EDUROAM. However, they will be supported in the future after a firmware updated.

9.10 Is there a mobile phone app that lets me view the data?

Currently there is an android app available, but we are working to make the website fully mobile device friendly, so that no mobile phone app is required. We would rather focus the time of our small team on the kits themselves instead of maintaining apps. So our final aim is to be app free, but fully mobile friendly.

9.11 How accurate are the measurements?

Weather, noise, light and PM sensor measurements have been calibrated and validated against reference sensors through both in-house and external validations and they provide accurate data. Chemical gas sensors are to be considered qualitatively rather than quantitatively while calibration algorithms are developed for data accuracy improvement.

9.12 Are there any notable case studies using similar sensors?

Yes! A particularly interesting case study is the Making Sense project at Plaça del Sol in Barcelona, where a group of 15 technology enthusiasts and environmentalists joined a community of neighbours from a middle-class district that has been suffering from noise issues due to the nightlife in the square. You can find more information about this case study at: www.making-sense.eu

9.13 What happens if I want to move the device or give it to someone else?

Just by pressing the button you can fully reset your sensor and configure it again using your account or a new one. All your previous data will remain available on the platform as it was before the reset.

9.14 What about using other wireless technologies?

We are working closely with Barcelona's The Things Network community to develop a TTN enabled sensor. A LoRA prototype has been tested, but we don't have dates for the final version yet. BLE, Zigbee, or others are not currently supported, and except for G5, we are not planning to implement them unless there is a custom hardware integration demand.

9.15 Can I remove my data from the platform?

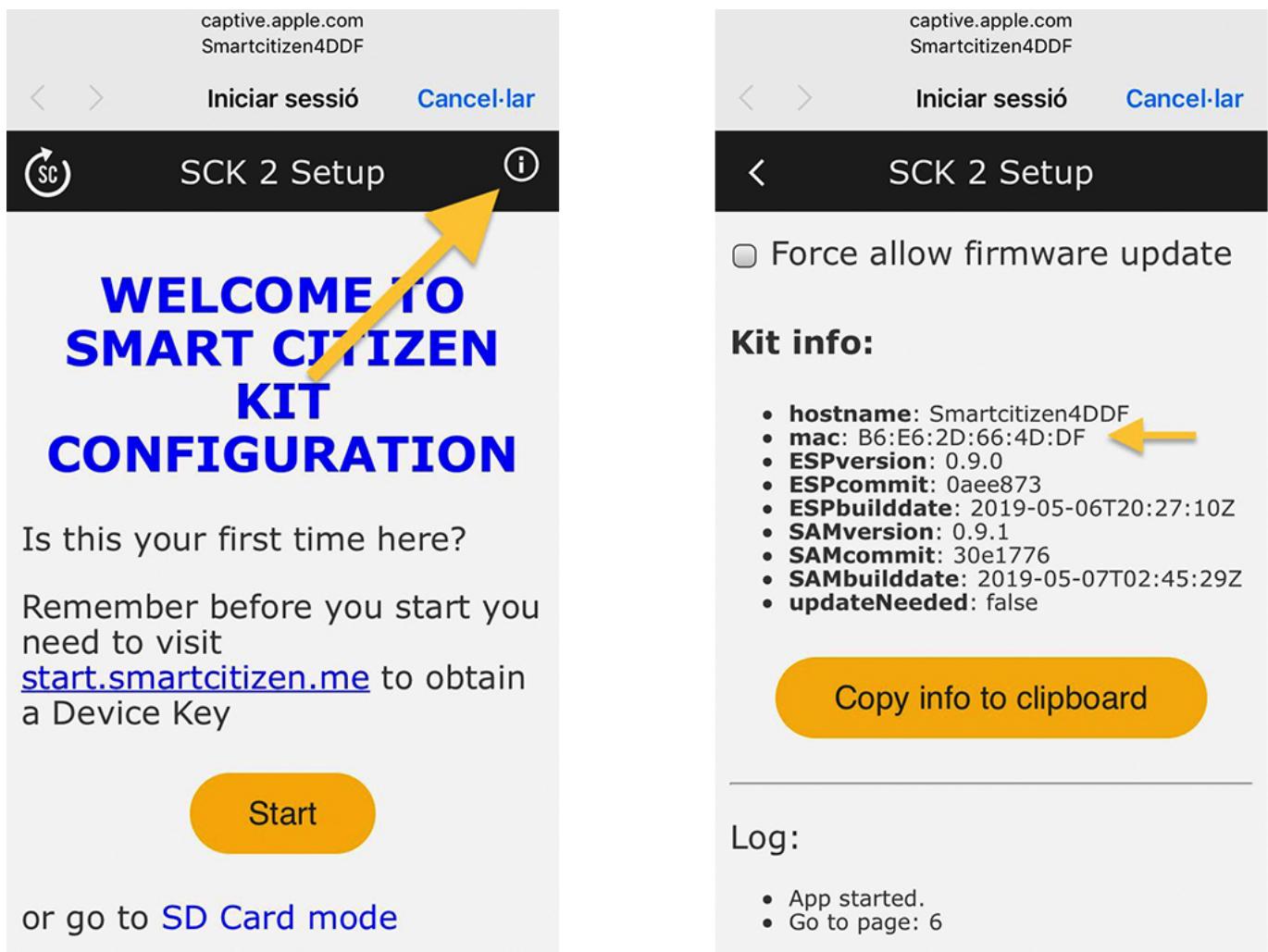
Of course. You are the owner of the data that you collect, and you can download and/or delete all your sensor data at any time.

9.16 How can I retrieve the MAC address from my device?

You can retrieve the MAC address with two methods: either you can use your phone (see below), or follow this guide if you want to try out the console interface in the kit.

Using your phone

1. Set the SCK in *setup mode* (press the button once, the LED should turn red).
2. With your phone, join the Wi-Fi network created by your Kit; it should be SmartCitizen[...].
3. Once you are on your Kit configuration page, go to the Info section. You will see a page with all the information about your Kit.
4. Your MAC address is listed as seen below:



9.17 What batteries are shipped with the kits?

The default SCK 2.1 Kits come with a 2000mAh LiPo battery model PL804050 (see datasheet and material safety data sheets).

For custom projects we also offer a bigger 6000mAh LiPo battery model DTP605068 (see datasheet and material safety data sheets).

We are working on a new dynamic battery calculator. Currently, you can find some approximate data here for the SCK 2.1.

9.18 Are the electronics waterproof?

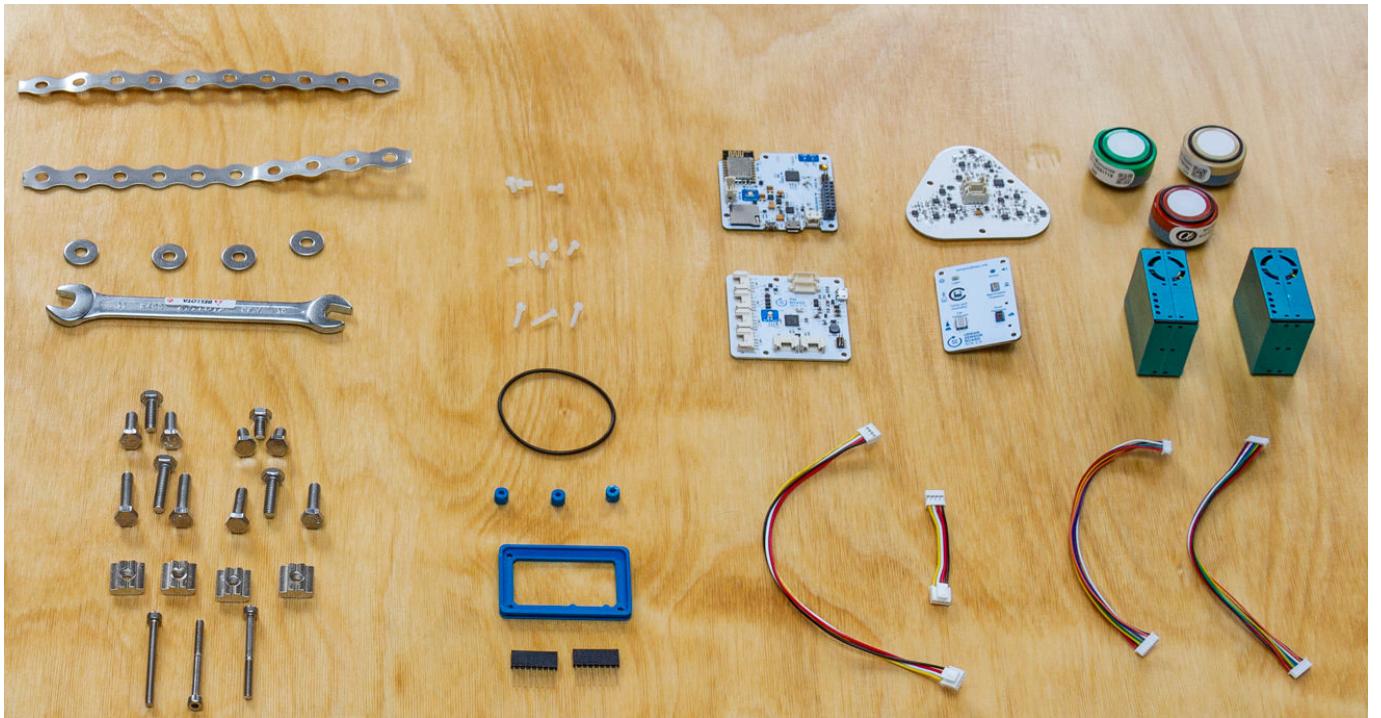
No. They cannot be exposed to water, high humidity, corrosive environments, or moisture. Always use an enclosure when exposed outdoors. Highly humid environments can provoke corrosion in the sensors (symptom of this is blue powder near the sensors in the urban board). To help protect them, we recommend using *transparent nail polish* in these areas. **Do not obscure the areas in red:**



If you are using any enclosure from the repository, we also recommend using a filtration foam (PPI-20/10) like this one. More info [here](#)

10. Components

10.1 Hardware Architecture

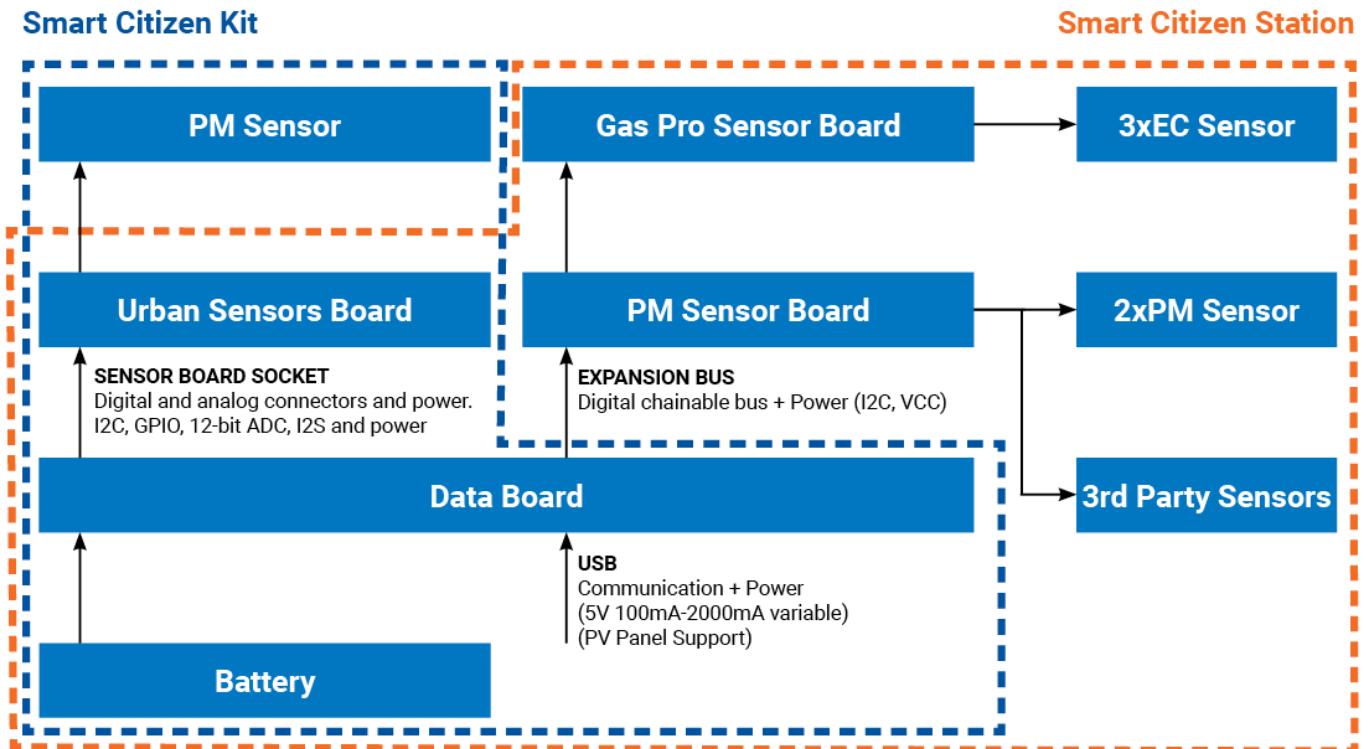


The project's sensor platform builds on the legacy of previous Smart Citizen Kit generations to develop a new set of tools especially aimed at providing meaningful data insights on a low budget. The system is designed in a extendable way, with a central data logger with network connectivity to which the different components are branched and aims to give support to a various activities ranging from education to more advanced scientific research.

We believe building modular and reusable hardware is critical towards optimizing the research and development effort. By increasing the technology readiness levels of existing technologies, we can drastically improve the project exploitation strategy.

10.1.1 Hardware

The core system bases its sensing capabilities in widely reviewed low cost sensors, and aims to provide a solid framework for environmental monitoring activities. Each of the modules is shown in the Figure below:



- **Data Board:** A datalogger at the heart of the sensors architecture supporting the Smart Citizen Kit and the Smart Citizen Stations.
- **Firmware:** The software running inside the sensors.
- **Sensor Board:** Multiple sensor board have been developed. They can be combined to built the different sensor solutions as the Smart Citizen Kit and the Smart Citizen Stations:
 - **Urban Sensor Board:** A selection of low-cost sensors in a board ready to measure the urban environment: temperature, humidity, noise, light, and PM2.5, among others. Together with the Data Board they create the Smart Citizen Kit.
 - **PM Sensor Board:** An auxiliary board capable of driving two Particulate Matter sensor as well as other auxiliary sensors required for specific deployments as an external temperature sensor or an anemometer. It is used in the Smart Citizen Stations.
 - **Gas Pro Sensor Board:** An auxiliary board driving 3 Alphasense Ltd. Electrochemical Series B Gas Sensors designed for ultra-low noise, high-performance and low power operation. It is used in the Smart Citizen Stations.

10.1.2 Open Source

We're against black boxes!

The entire project it is released under open source licenses:-

- Hardware components: CERN Open Hardware License v1.2
- Core firmware: GNU GPL v3.0
- Software platform: GNU AGPL v3.0

 Info

Check the **Source files** section for each component and explore the software source code and the hardware blueprints.

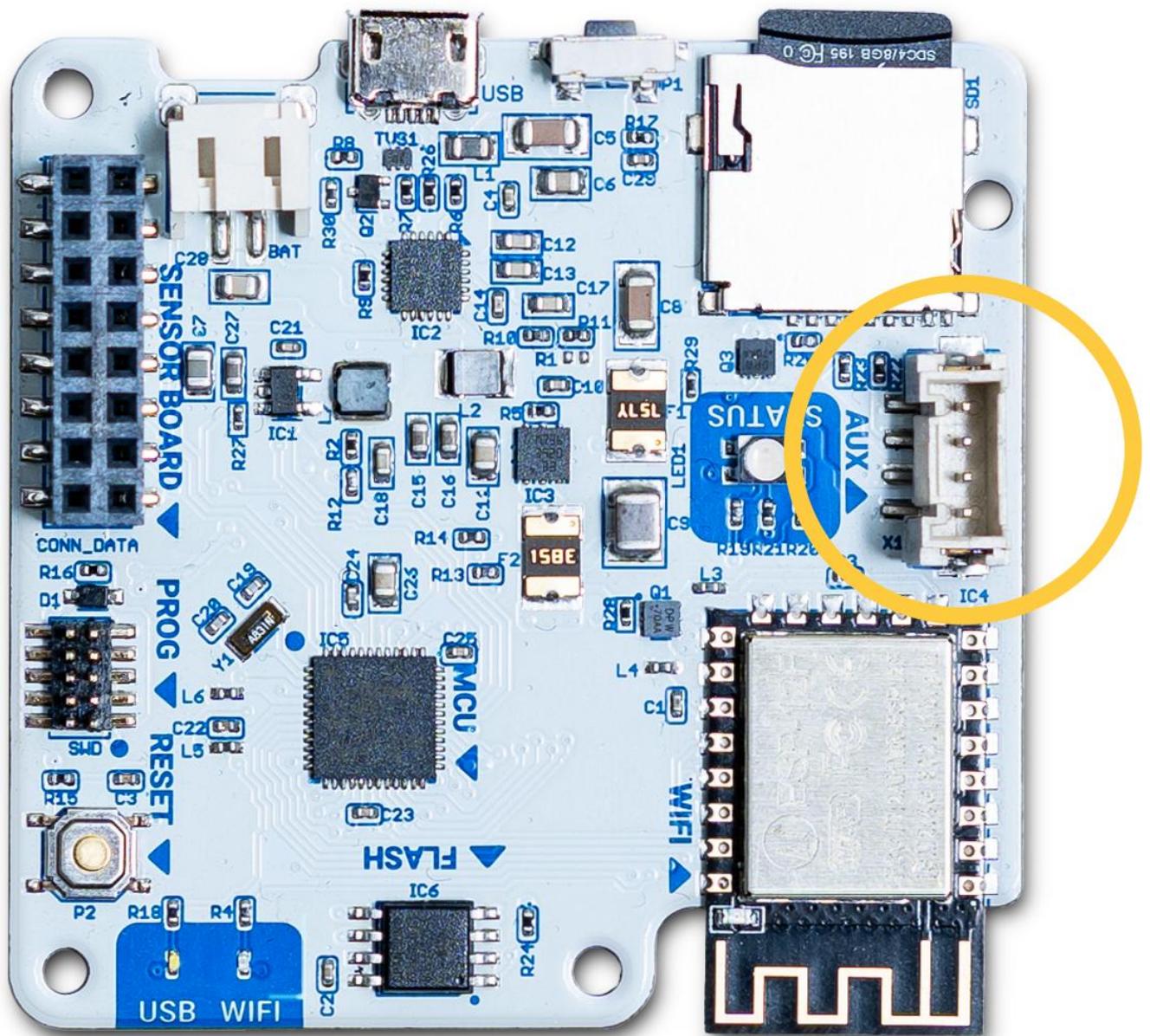
Source files

 [Download the documentation](#)

 [Check the documentation source code](#)

10.2 Auxiliary Connector

The data board features a standard Grove connector where off-the-shelf modules from the same manufacturer can be connected. The connector supports an independent I2C bus by default, but by software it can be configured to support other uses (GPIO, I2C and UART). It can supply power up to 750mA, and it can be enabled or disabled by software to save power.



There is a lot more to it!

The Smart Citizen Kit is designed with a modular approach in mind. This means that the Urban Board is only a selection of low cost sensors for air quality, but the hardware itself can be expanded for other use cases such as a more advanced air quality monitoring setup, soil monitoring, or water quality. Make sure you check our guide on how to use them.

10.2.1 Supported sensors

General purpose

- Seeed Grove ADC - 12 bit ADC from Seeed Studio
- Adafruit INA219 - Supports Bus voltage, shunt voltage, current and load voltage
- SparkFun ToF Range Finder Sensor - VL6180 - supports distance and light. Can be used for water level measurements

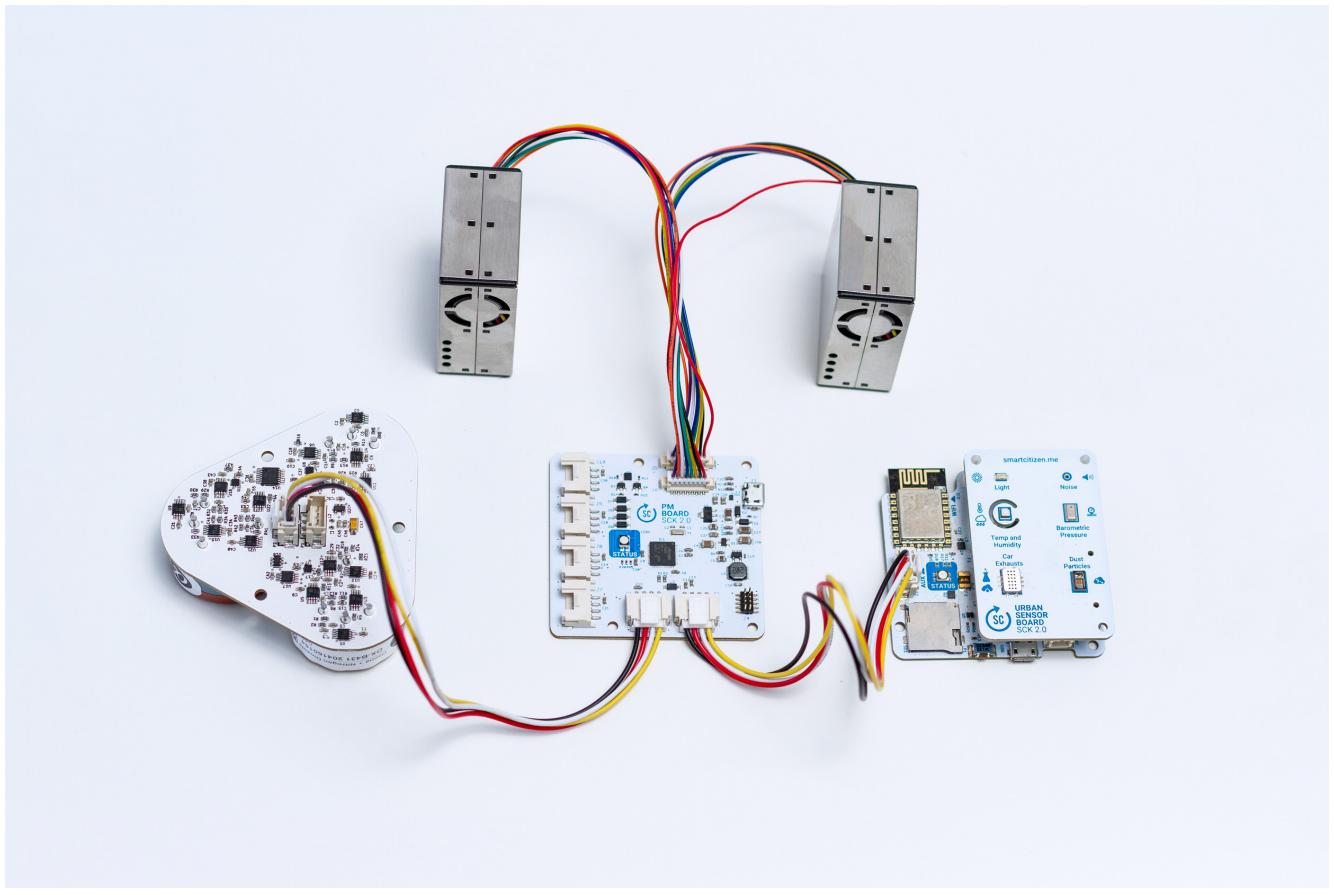
Environmental and air quality

- Seeed Grove SHT31 Temperature/Humidity
- Adafruit BME680 - supports temperature, humidity, barometric pressure and VOC gas
- Atlas Scientific Temperature - can be used with any PT-100 or PT-1000 temperature probes

Smart Citizen Station

Expanding the base air quality solution, the Smart Citizen Station is a more advanced setup in a more rugged enclosure. The sensors below can be directly plugged in and detected by the SCK:

- Smart Citizen Gases Pro Board: supports 3 electrochemical alphasense sensors, temperature and humidity
- Smart Citizen PM Board: supports 2 Plantower PMS5003 sensors, I2C extension, 4 ADC pins, 2 GPIO and a UART Serial port



Water measurements

Check the water measurements documentation with examples on sensors such us the Atlas Scientific Dissolved Oxygen and the DS18B20 Water Temperature

Soil measurements

Check the soil measurements documentation with examples on sensors such us the Chirp Soil Moisture - supports soil moisture (requires calibration), temperature and ambient light.

10.2.2 Other auxiliaries

- Seeed Groove OLED screen (96x96) - the screen cycles through sensor readings

Implement your own

Contact on how to implement sensors made by others.

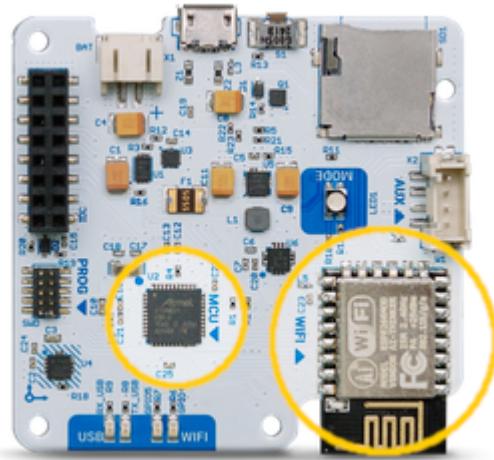
10.3 Data Board

The data board is a data-logger at the core of the sensors architecture supporting the Smart Citizen Kit and the Smart Citizen Station. This module is powered by an ARM M0+ 32-bits 48Mhz **SAMD21** running the Smart Citizen Firmware), combining the low power consumption of the ARM M0 family with the power of a 32-bits processor with 32KB of RAM and 256KB of FLASH memory. This solution offers enough program storage and memory space to support multiple auxiliary sensors. This chip is used by the Arduino Zero and MKR boards, therefore benefiting from the open community built around these boards in particular and the Arduino project in general.



Check the source code

The data board also includes a Wi-Fi module, a micro SD card slot, an internal Flash and a battery management solution. In addition, it includes 4MB of extra Flash Memory for offline data storage, in case of network brownouts. The Wi-Fi Module is the well-known Espressif ESP8266 IEEE 802.11 b/g/n Wi-Fi with 4MB Internal Flash for web content storage:



The Data Board connects to the sensor board providing power, analog and digital communications (12 bits ADC, GPIO, I2C, I2S, VCC). The data board also includes a Seeed Studio standard Grove connector where off-the-shelf modules from the same manufacturer can be connected. The connector supports an independent I2C bus by default, but by software it can be configured to support other uses (GPIO, I2C and UART). It can supply power up to 750mA, and it can be enabled or disabled by software to save power.

The board includes a power unit, with a battery management system, capable of handling a variety of Lithium polymer cells. The batteries are connected to a standard JST-2 pin battery connector. The Smart Citizen Kit by default uses a 2000mAh battery, but larger capacities can be used. Under normal conditions, and depending on the sensors enabled, a 2000mAh battery can last between 24 hours (with all sensors enabled, and a 1-minute recording frequency) to more than a week. The board also features a *sleep mode*, through which drastically lower average consumption are achieved.

The controller allows the batteries to be easily charged using the boards micro USB connector using any standard USB power adapter like the ones used on Smartphones. On remote areas, it can also be powered using a selection of PV Panels like Voltaics Systems 6W panel.

10.3.1 Firmware

The Smart Citizen Kit firmware is comprised of two parts: 1) the primary processing tasks are done by the SAMD21 microcontroller firmware; 2) the tasks related to network communication are run through the ESP8266. The SAMD21 is built on top of the Arduino Zero with a custom variant for the Data Board main MCU. The ESP8266 is also built using the Arduino ESP Core. Both firmwares are built and managed with Platform IO, an open-source IDE for embedded development. Platform IO features built-in dependency management and allows you to compile and upload both processors with a single command. Using the SWD ARM connector you can change the MCU bootloader and debug the firmware using Open Source tools.

Info

Learn more about the software running inside the Data Board on the [Firmware section](#).

Software guides

Check the firmware guides and learn how to update and even modify the software:

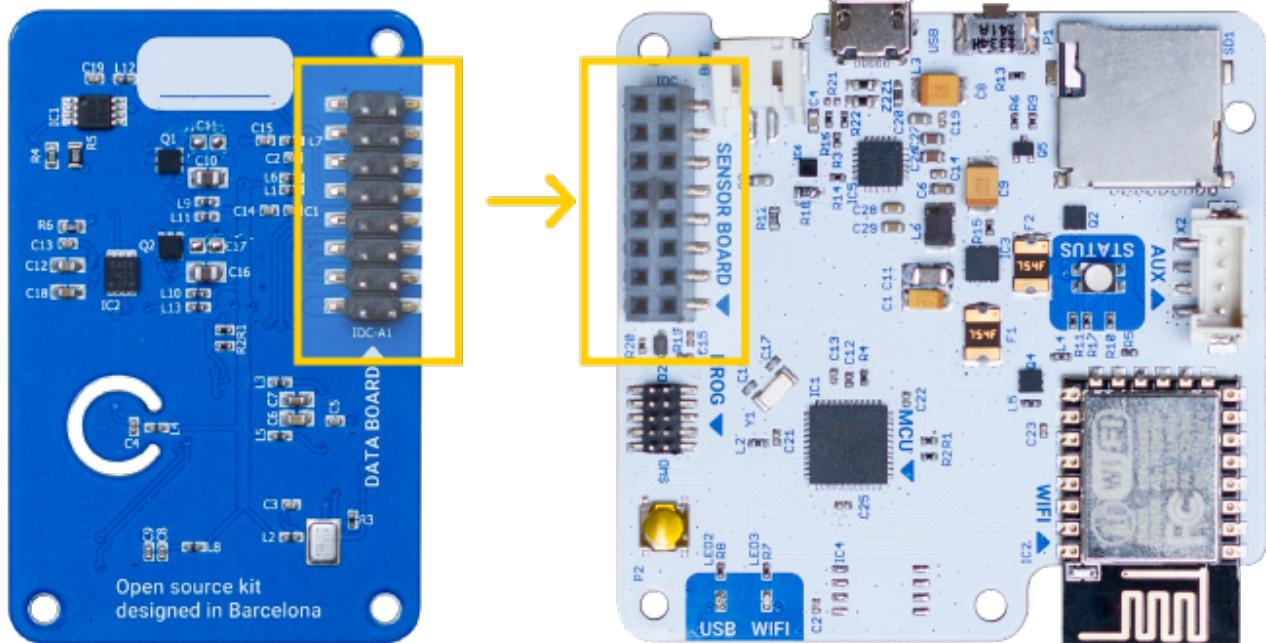
- Debug the Firmware
- Edit the Firmware
- Update the Firmware

10.3.2 Buses

Sensor Boards connector

The Kit features a modular architecture where sensors can be updated independently by replacing any individual Sensor Board. The Sensor Boards features GPIO, ADC, I2C, UART and I2S connections at 3.3V. Currently, we only offer the Urban Sensor Board, but more boards are on the way, and you can even design and build a custom one.

Example of a Sensor Board

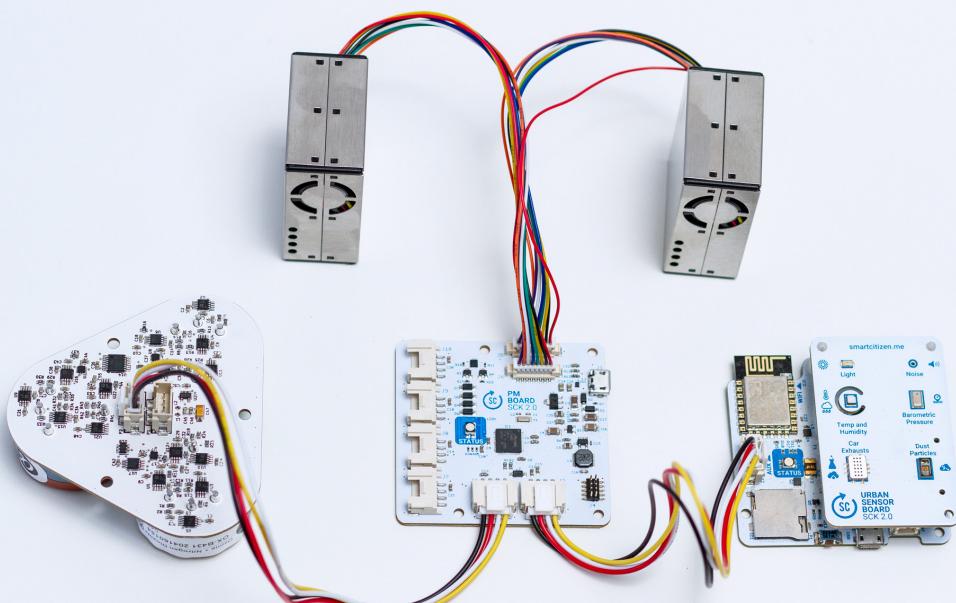


SAMD21 Pins	Arduino Zero Pin	SCK Pins	SCK Conector	SCK Conector	SCK Pins	Arduino Zero Pin	SAMD21 Pins
GND	GND	GND	16	15	GND	GND	GND
GND	GND	GND	14	13	GND	GND	GND
PA11	0	I2S_FS	12	11	TX	A5	PB2
PA7	9	I2S_SD	10	9	RX	25	PB3
PA10	1	I2S_SCK	8	7	5V	5V	5V
PA22	20	SDA	6	5	PWM_CO	13	PA9
PA23	21	SCL	4	3	PWM_NOX	14	PA8
VCC	VCC	VCC	2	1	VCC	VCC	VCC

Auxiliary connector

The Data Board features an independent configurable auxiliary bus at 3.3V with a SEEED Studio Grove connector. The Bus has native support for I2C, but it can also be setup on firmware as a GPIO or UART port. It can supply power up to 750mA, and it can be enabled or disabled by software.

 Example of devices connected via the AUX connector.



10.3.3 Power management

The Smart Smart Citizen Kit gives us the possibility of running directly from a USB power source with or without lithium battery, using the BQ24259 USB Charger. The charger manages external power regulation, battery fast charging (up to 2Ah) and USB OTG that allow us powering other devices from the SCK (currently not implemented).

Normaly the SCK uses a 2000 mAh Lithium polymer battery but it is possible to take advantage of larger batteries. The charging current is regulated with a manual imposed limit that can be configured, and also auto adjusts to the connected USB charger capacity. It is also possible to use solar panel (5v) to charge the SCK.

The power consumption of the kit depends on which sensors are enabled and how often they are read/published. Between readings the kit goes to *sleep mode* turning off almost all the subsystems and reducing the power consumption.

In previous versions of the kit (V2.0 and before), the most power-hungry sensors were the SGX MICS gas sensors (NO_2 and CO) which need an always-on heater with a permanent consumption of around 50 mAh (35 hours per charge). In V2.0 and V2.1, the PM sensor needs a fan with a consumption of 35 mAh (50 hours per charge). To improve the power consumption, the PM Sensor works on *one-shot mode* which turns the sensor off for $\frac{3}{4}$ ths of the time, and only taking a reading after the sensor has stabilised.

The **kit normal operation cycle** on battery is: read sensors, post, and then go to sleep. Until the battery charge is below 3%. When that threshold is passed it will enter an emergency sleep mode and interrupt all the normal functions until the charge goes over 5%.

Power consumption

The base power consumption of the device is 16mA (no sensors or wifi connection). While posting data online, the consumption can go up to 75mA accounting for the ESP8266, with an additional 90mA if all the sensors are to be working at the same time (Urban Sensor Board + PM sensor).

10.3.4 Source files

Download

Check the source code

10.4 Deployments

This page is a compilation of information regarding the operation in the field of the Smart Citizen Station. Since there are different versions, please, refer to their section accordingly.

10.4.1 Living Lab Station V2

 WIP

This version is in production stage and no information is currently available.

10.4.2 iScape Living Lab Station V1

The Pack

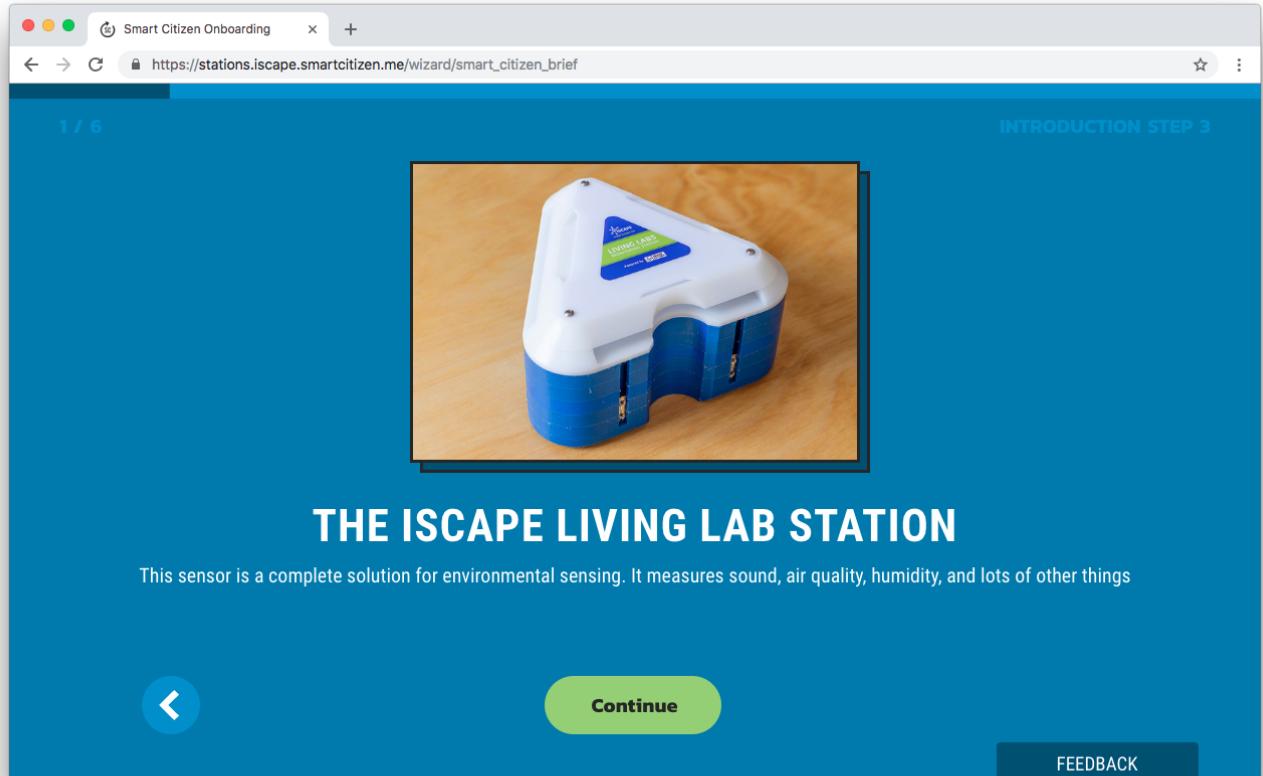


- iSCAPE Living Lab Station
 - Urban Board 2.0
 - Data Board 2.0
 - PM Board 2.0 + 2 PM sensors
 - Gas Pro Board 2.0 with 3 EC sensors
 - 6Ah Battery
- Accessories
 - MicroSD card 512MB
 - USB Charger
 - MicroSD to SD card adapter
 - USB Power Supply
 - 2m 3 Wire 220V cable
 - Mounting brackets and screws
 - Mounting tools (1x Wrench + 2 Allen Keys)

Instructions

ON BOARDING

To start the installation simply visit the setup website **stations.iscape.smartcitizen.me**.



Warning

⚠ We will need you to send us the following information once you are done with the setup: the *device ID*, which appears in the URL of your device <https://smarcticitizen.me/kits/XXXX> and the physical station ID that corresponds to that *device ID*, which can be found in a sticker underneath.

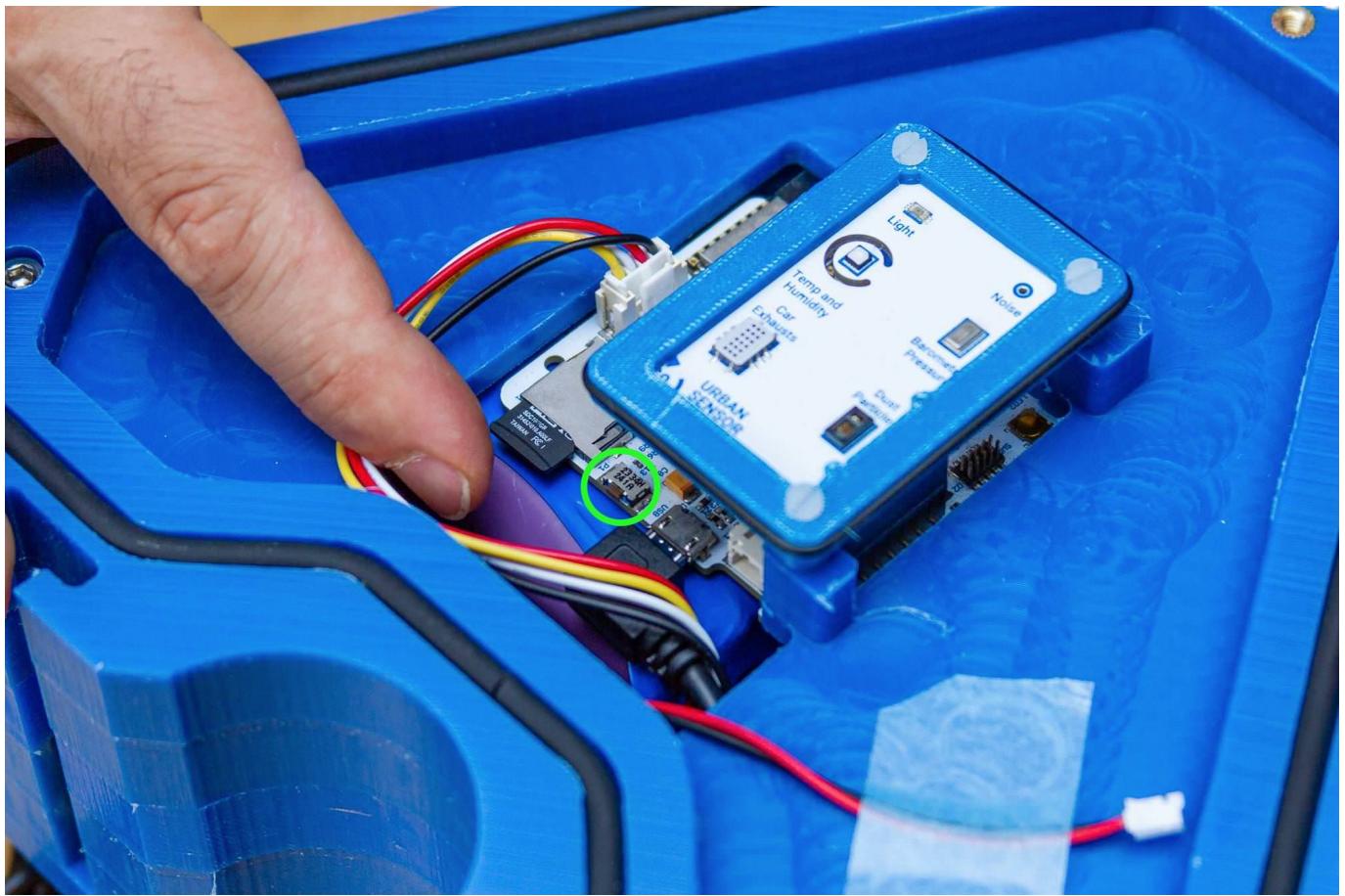
STATION ID GOES HERE

**GET DATA FROM THE SD CARD**

You will need to access the Kit in order to get the SD card. For this, first unscrew the **two white layers at the top of the station** with the keys provided in the Pack:



Then turn off your Kit by pressing the button for **5 seconds** and remove the micro SD card. You can plug the card on your computer using a Micro SD card reader.



Warning

Handle the SD card with care! It might drop inside the station

You will find inside a `YYYY-MM-DD.CSV` with all the data. You can follow the [Manual CSV data upload](#) guide to manually upload the data to the platform.

Power it back on!

Once you are done uploading the data and you want to keep on logging, put the SD card back in with the Kit OFF and press the button. It will come back to life!

Outdoor installation

Use the perforated steel tape and the M6 provided to mount the Station on any street light or pole. The Pack also includes the required wrench:



Also, a temperature probe needs to be extracted from the bottom of the station:



And it should look like this:



UMBRELLA COVER INSTALLATION

Due to some issues with the waterproofness of the Living Lab Station, we have developed a solution to protect it from the rain. This solution is shown in the pictures below, and it's meant to solve these problems for the current version of the LLS. The newer version of the LLS has a simpler setup, already including such cover to protect it from the rain or sun radiation.

**Beware of collisions**

As you can see, the cover is a rugged piece and it's only meant for the current version of the station. Please, be careful and do not fit it in places where people could bump into it.

This is what you get in the package (except the wrench):

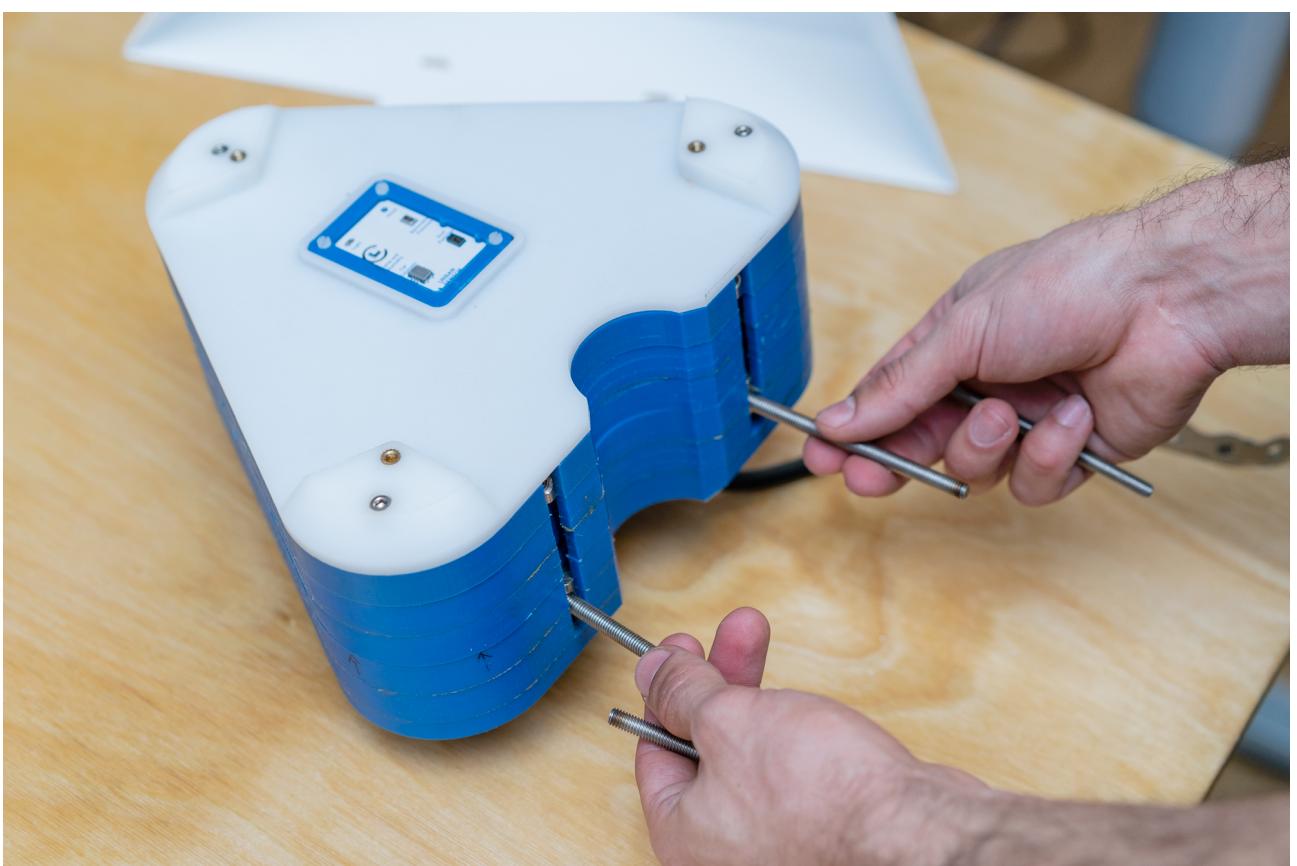


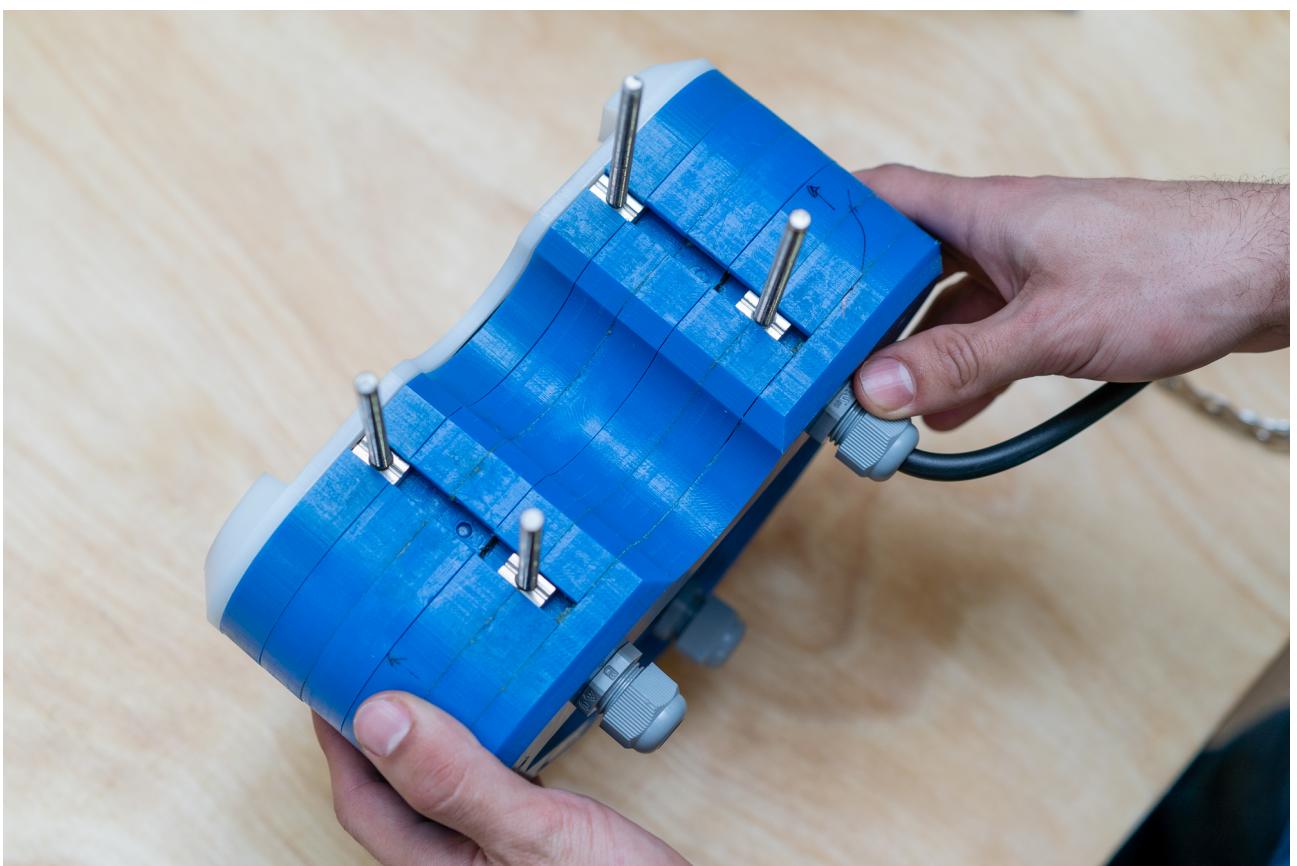
 Step by step

- If you have the 3D printed cover on the Smart Citizen Kit, it's time to remove it.
- There is no need to remove the two top white layers (in the pictures we did it without them)

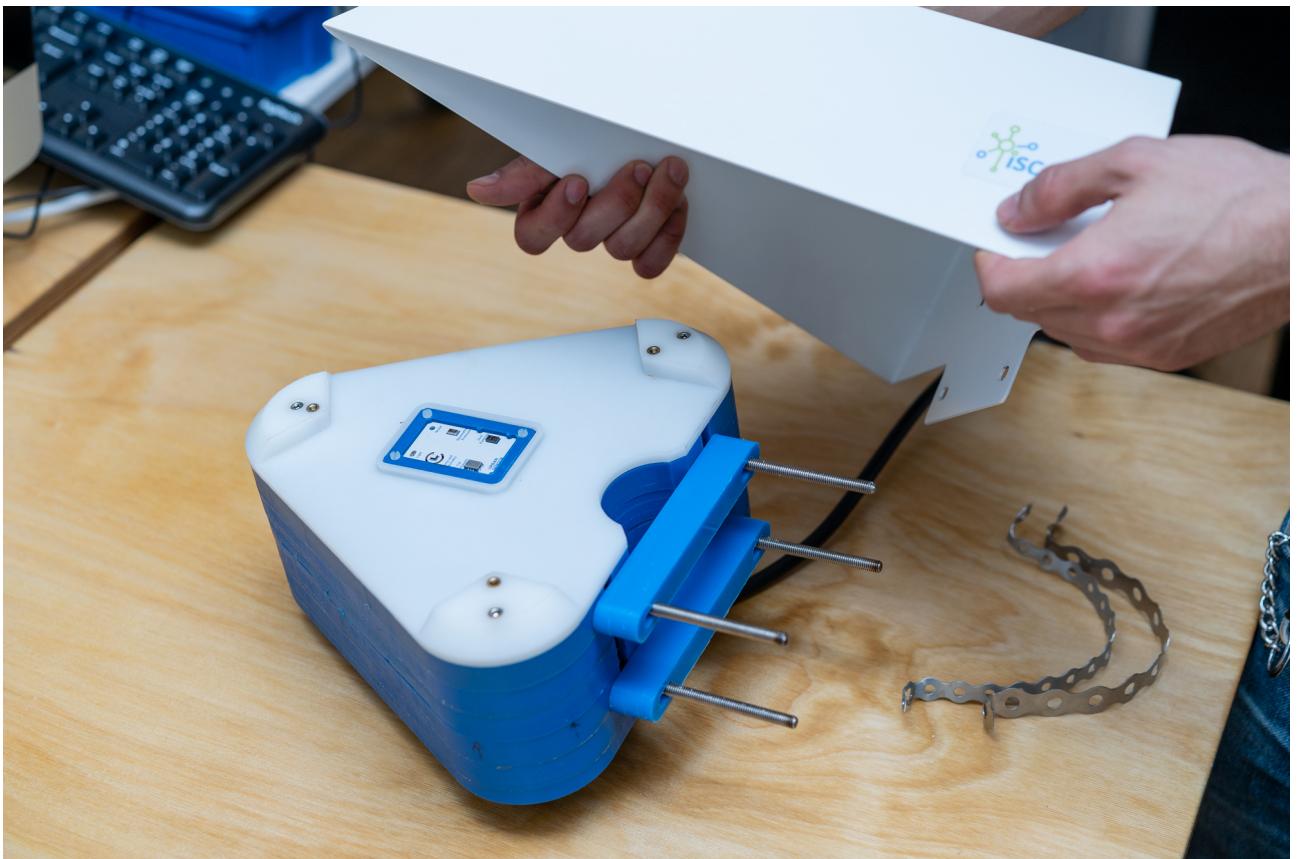


- Insert the threads in the already mounted t-slots. The distance between them is ~50mm





- Insert the 4x flat spacer in the threads



- Place the cover on the station





- Place the serrated spacers, with the serrated side on the outer part (they help to hold the station in place)



- Place the perforated steel stripe in one of the sides. Don't tight it too much, so that you have room to place it in the pole



- Put the station in it's final location, and tighten it with the perforated steel stripes. Play with both sides, so that the stripes are tight on the pole



• You are done!

Power supply

The Station can be directly powered at 220-240V AC (Max. consumption with the AC supply is 5W). It can also be powered via USB, with a normal phone charger (5V and 750mA max). However, there is a bit to do in order to change it. Let's see how!



Batteries

The Living Lab Station has a higher consumption, mostly due to the fans on the two PM sensors.

That means the internal battery last just for 20h, and it is only aimed at providing backup power.

For example, we can connect the station on the street light electric line, so the Station gets charged during the night when the lights are on.

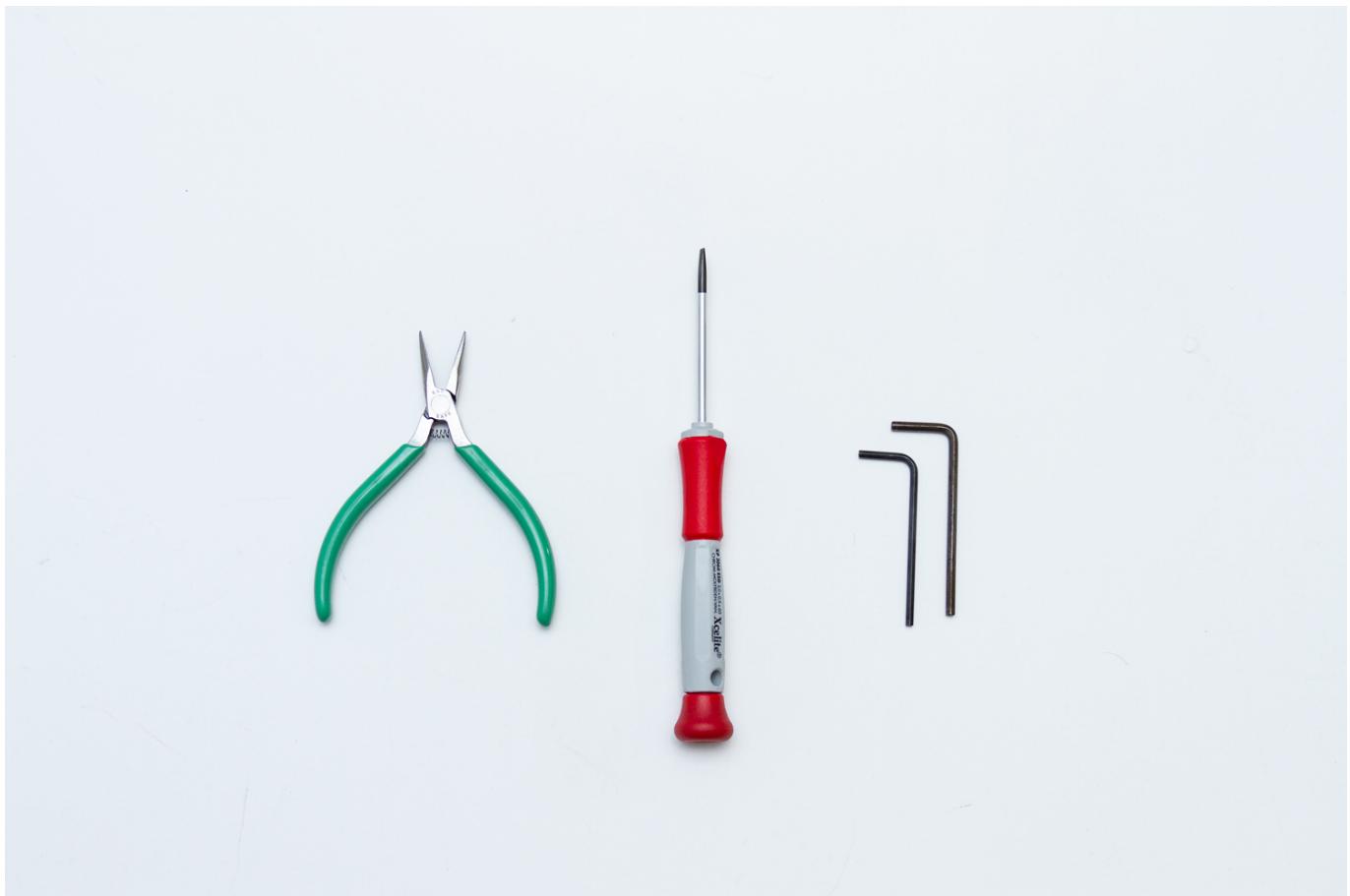
Solar Panel

Unfortunately, we are having some problems with the PV Solar Panel system to power the Station independently. The system is currently under tests, and it will be available in the next few months.



CHANGING POWER SUPPLIES

Before we start, some tools that will be helpful during the process:



 **Danger**

Unplug the station before starting this process from any type of external supply

 Step by step

- Remove the two covers using the allen keys as explained on the setup instructions.



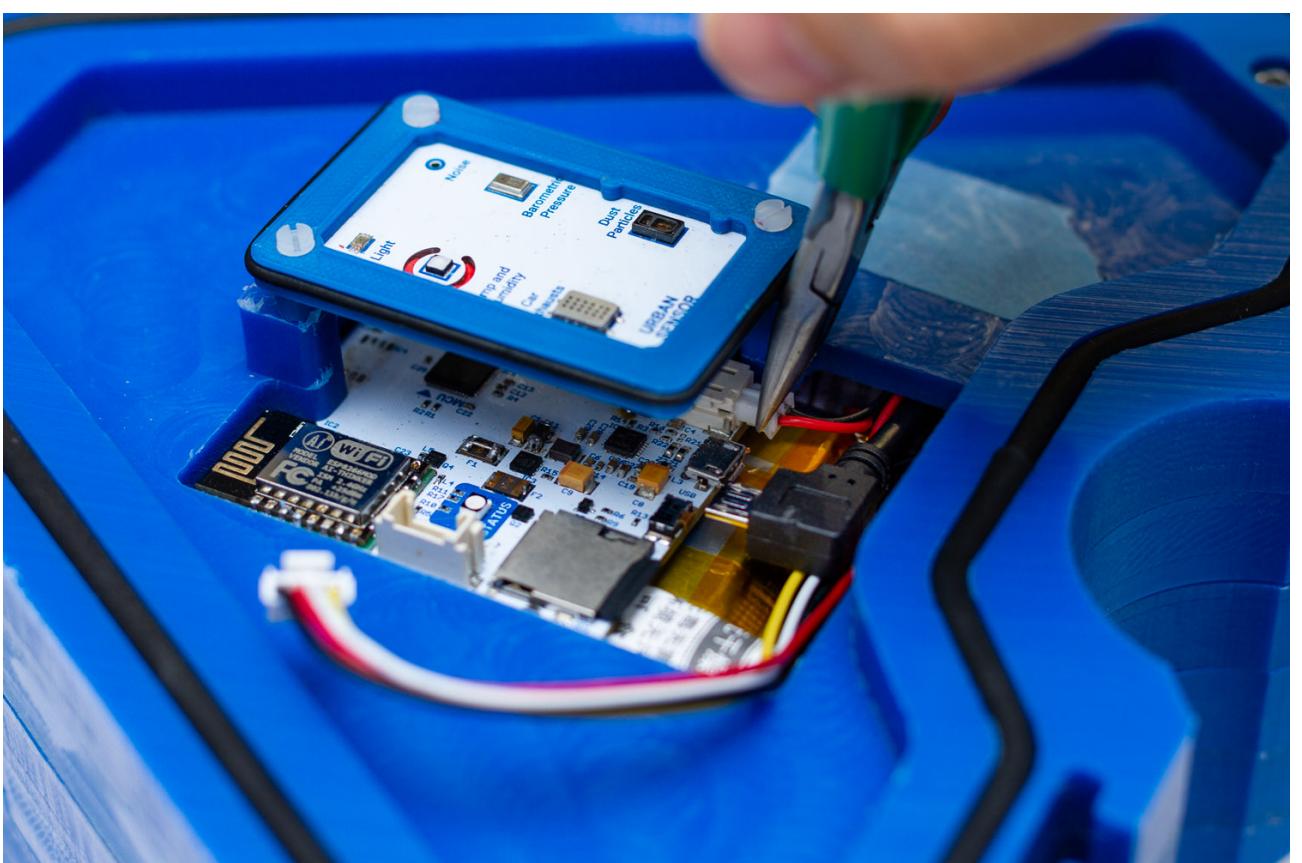
- Remove the layer which contains the kit. The kit is attached to the layers below, as seen in the image



- Unplug the different connectors in the kit: I2C, battery and USB



- You can use nose pliers for the USB and the battery





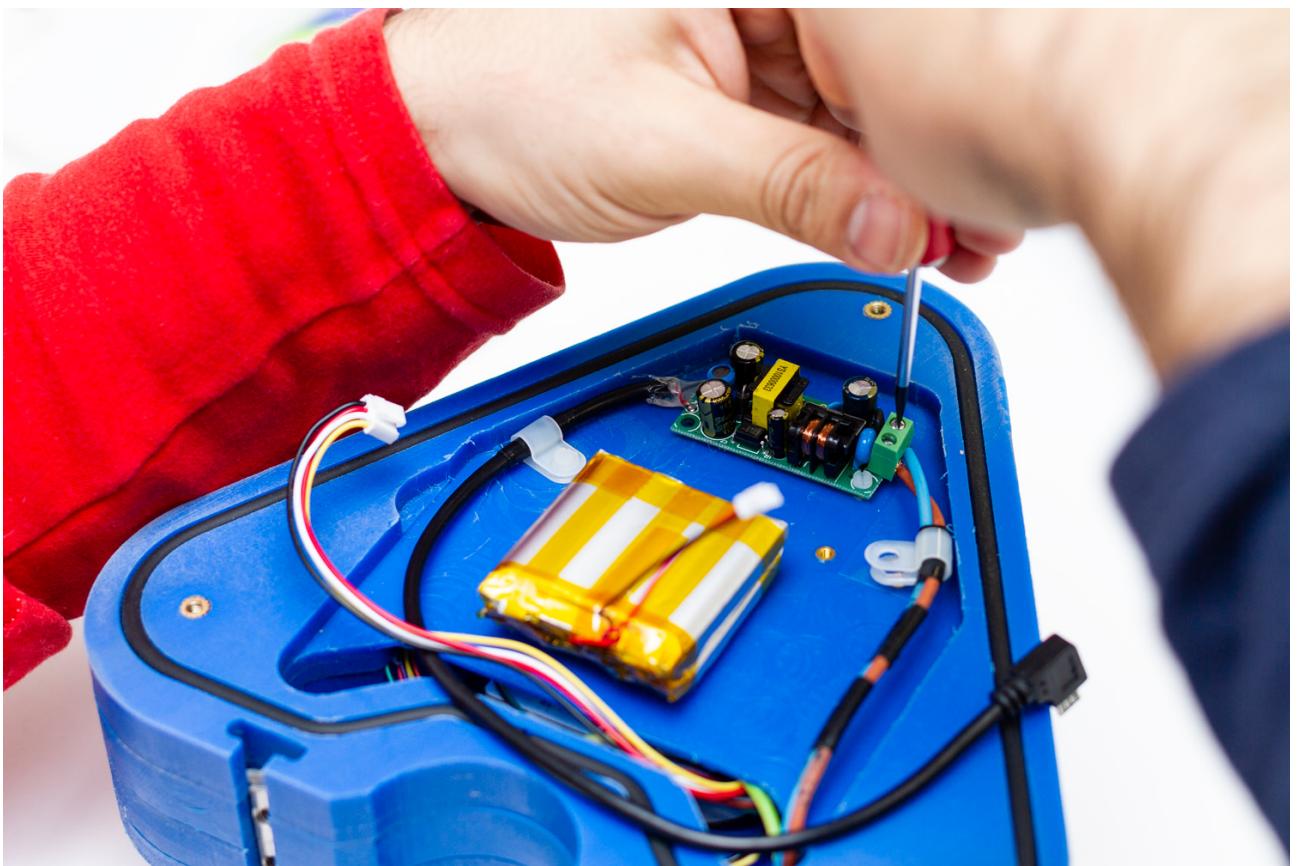
- Time to get to the power layer, this time, two blue layers will come off

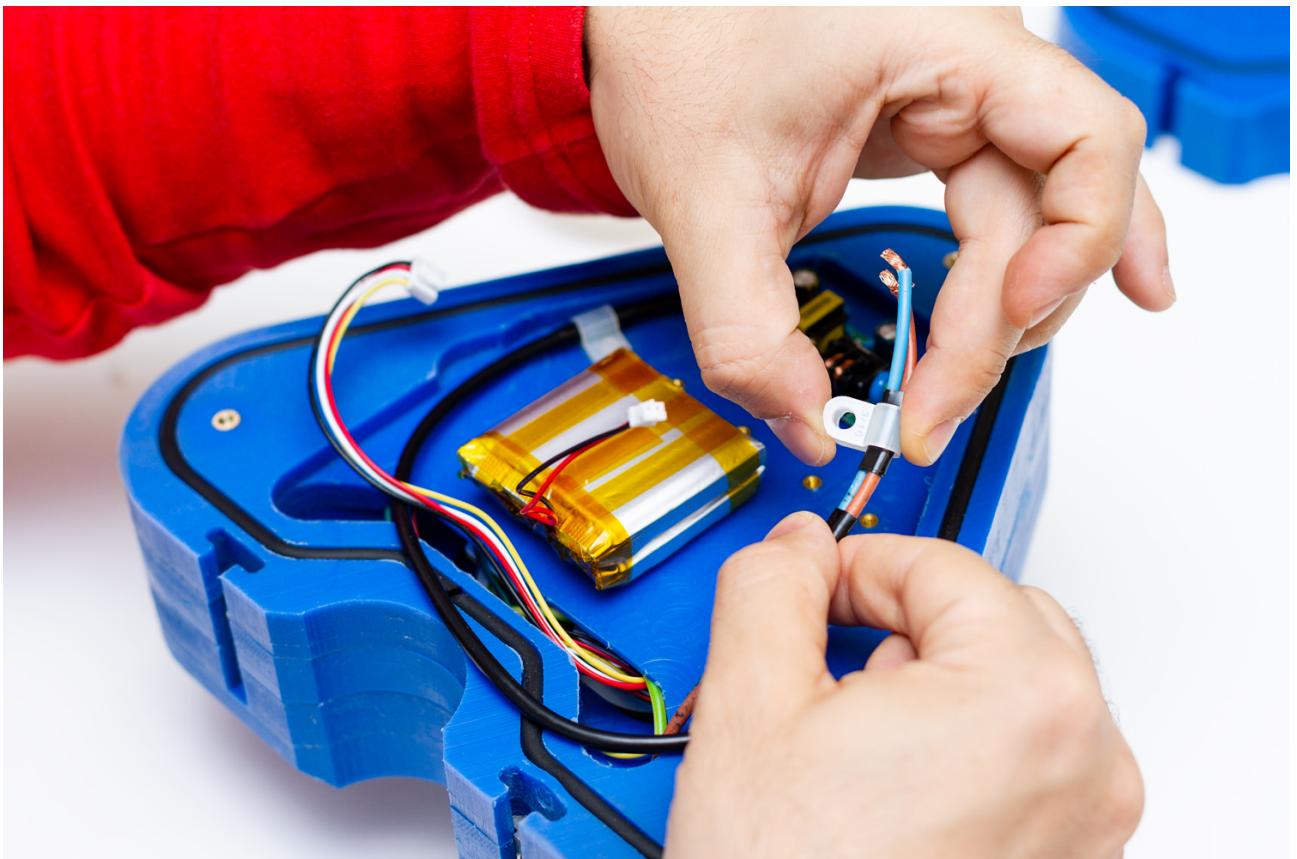


- Unscrew the cover for the power area



- Make sure there is no energy left in the power supply by checking that there is no LED on in it. Then, remove the cables from the power supply and the white brackets





- Extract the cable from the base's cable gland



- Cover the cable gland again and remove the square cable gland on the other side

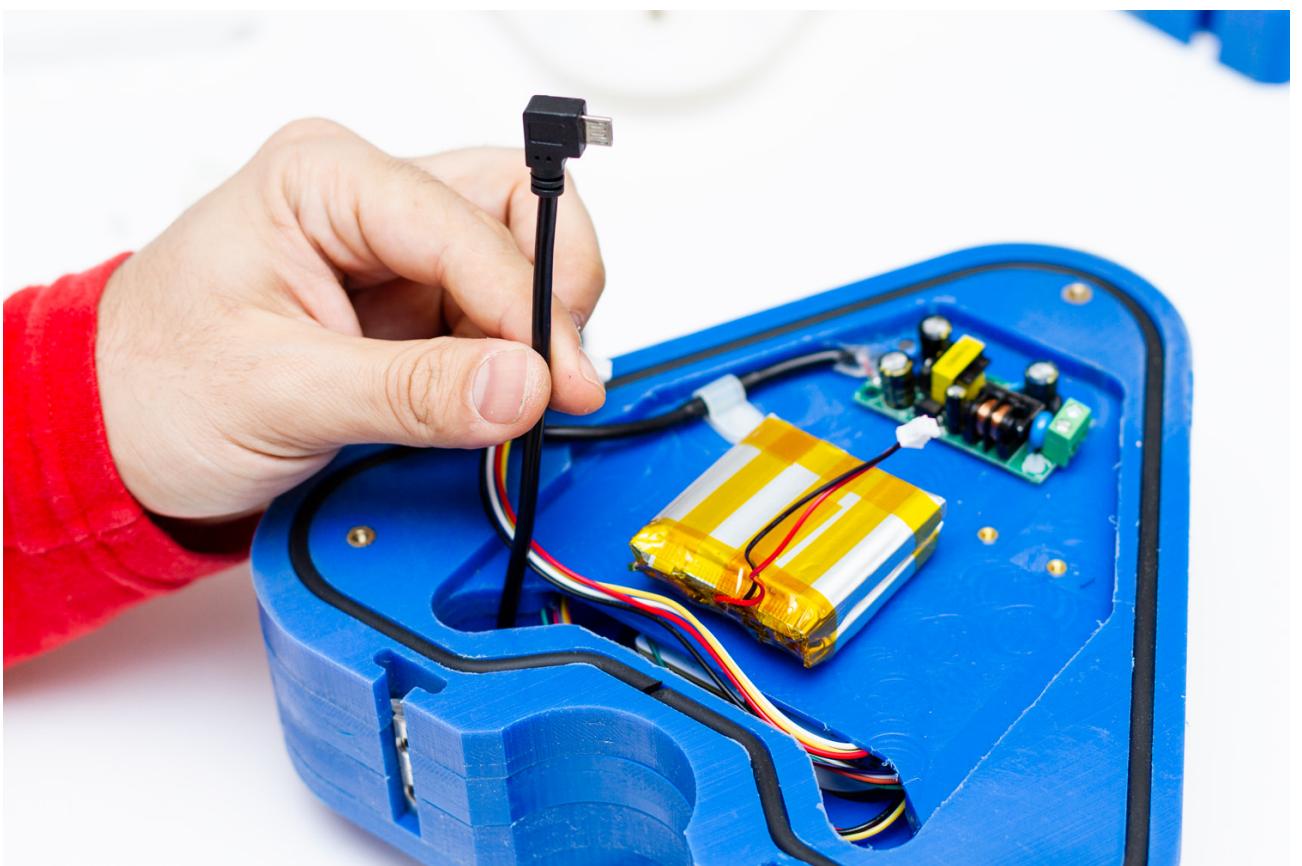
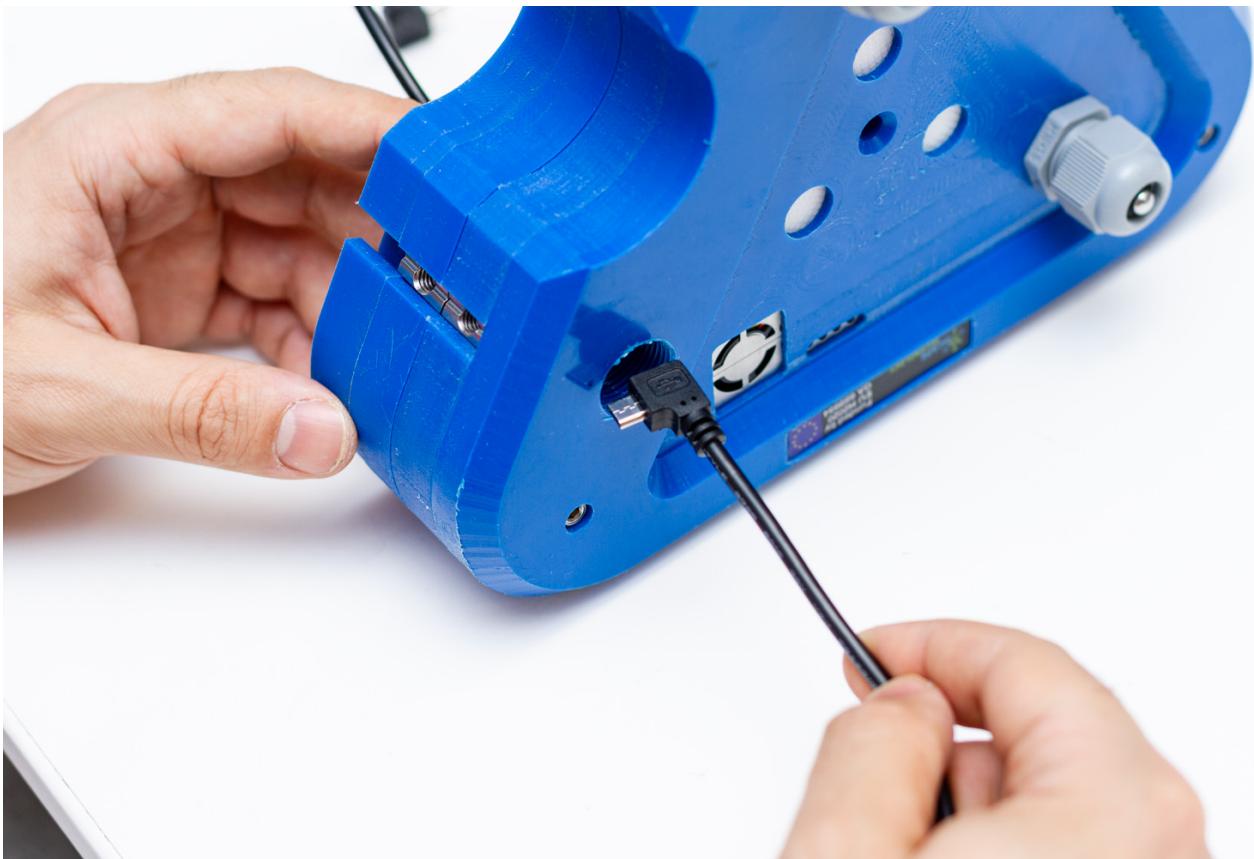


- Exchange the rubber in the cable gland with the one provided with a hole

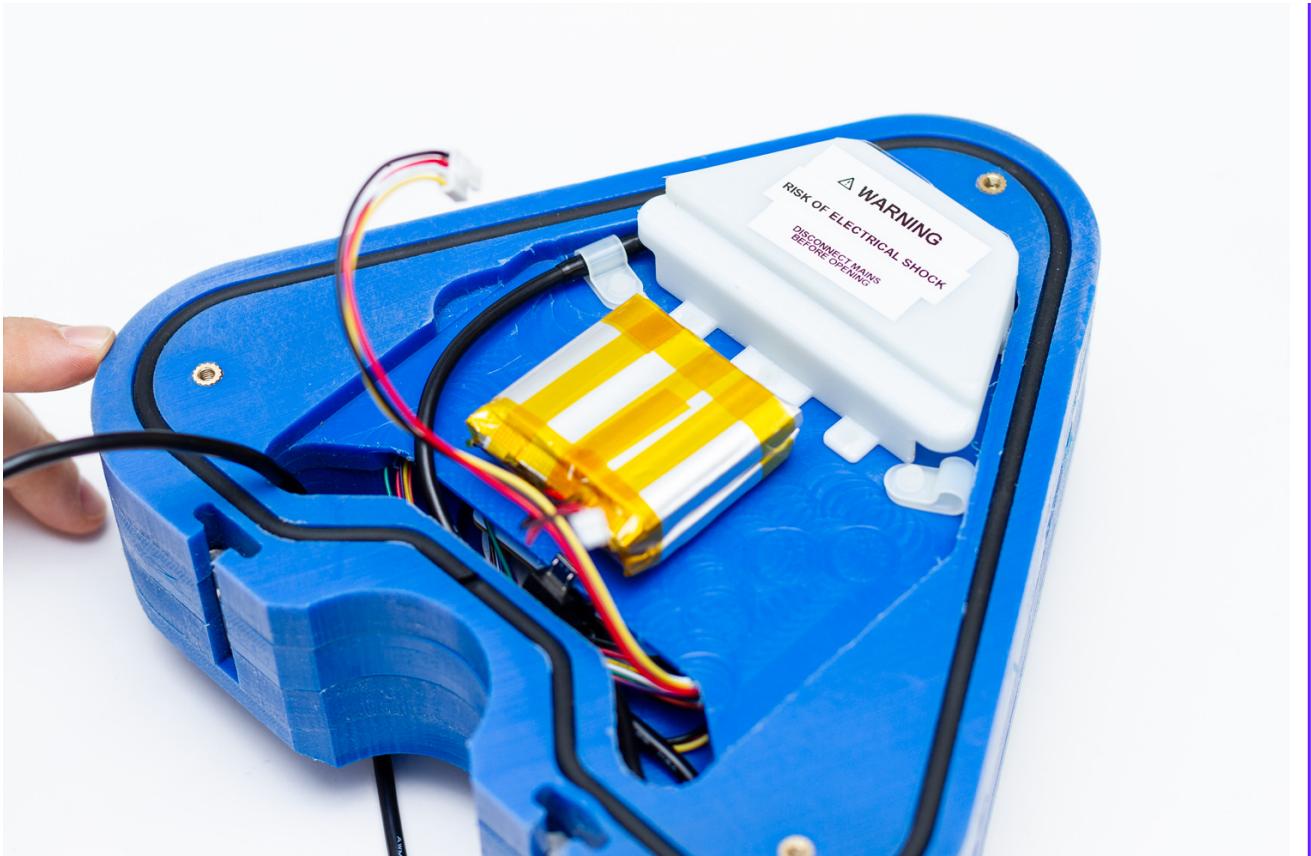




- Put the cable in and fix the gland in place. Leave sufficient overhead in the cable to be able to connect it to the kit



- Put the power cover back on



- Put the kit's layer back on and pass the cables through



- Connect everything in this order: first, the I2C connector, second, the battery, third, the USB



- Put the kit's layer on again. Verify that the o-ring fit's in properly. Close everything and put both layers back on on

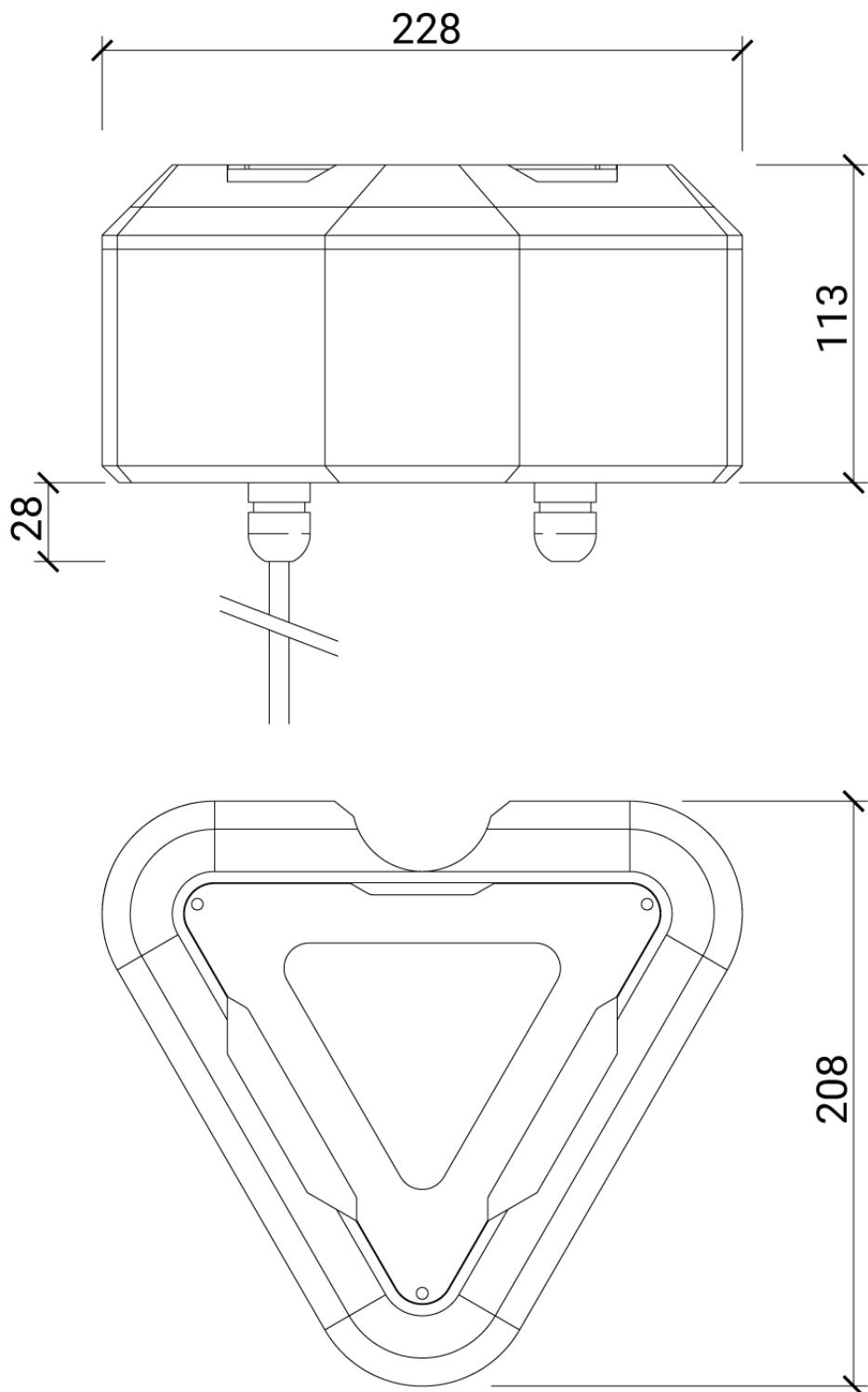




- Now, you can use the USB power supply or the battery pack!





Dimensions

Troubleshooting

BEFORE SETUP

Before configuring the Station setup make sure the LED is red. If not, press the button multiple times until the LED turns red.

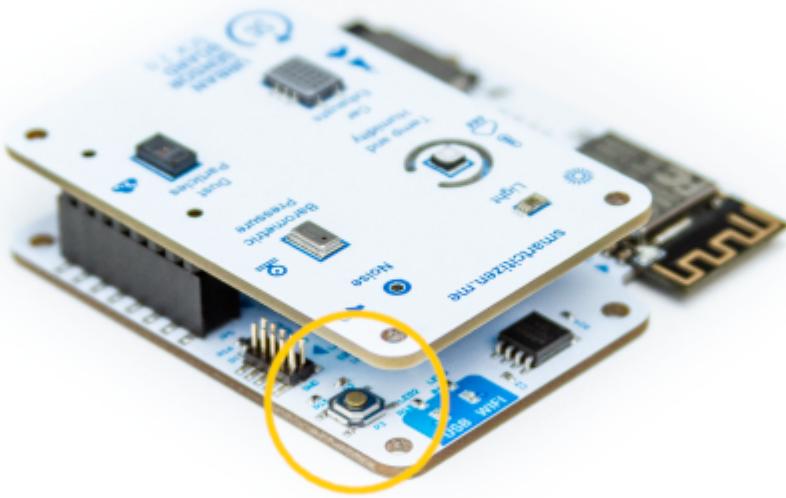


THE STATION DOES NOT RESPOND

If the station does not respond or does not work properly you can do two things:

Reboot your Station

You can fully reboot your Station by pressing the reset button located under the sensors board as seen on the picture. That will not delete any configuration, it will simply restart your device. Press the `RESET` button for a second. The light will go off and on and the device will start again.



You can also perform a reboot by disconnecting the battery and the USB cable so that the station is restarted. In this way we will not lose any data and configuration except the time in case of being in **SD mode**.



Factory reset your Station

You can fully reset the Station to the default settings so you can register again your device. Press the main button for **15 seconds**.



After 5 seconds the light will go off and will go on again after 15 seconds. Then you can release the button and your device will be fully resetted as a brand new Station.

THE LED DOES NOT TURN ON AND THE STATION DOES NOT WORK

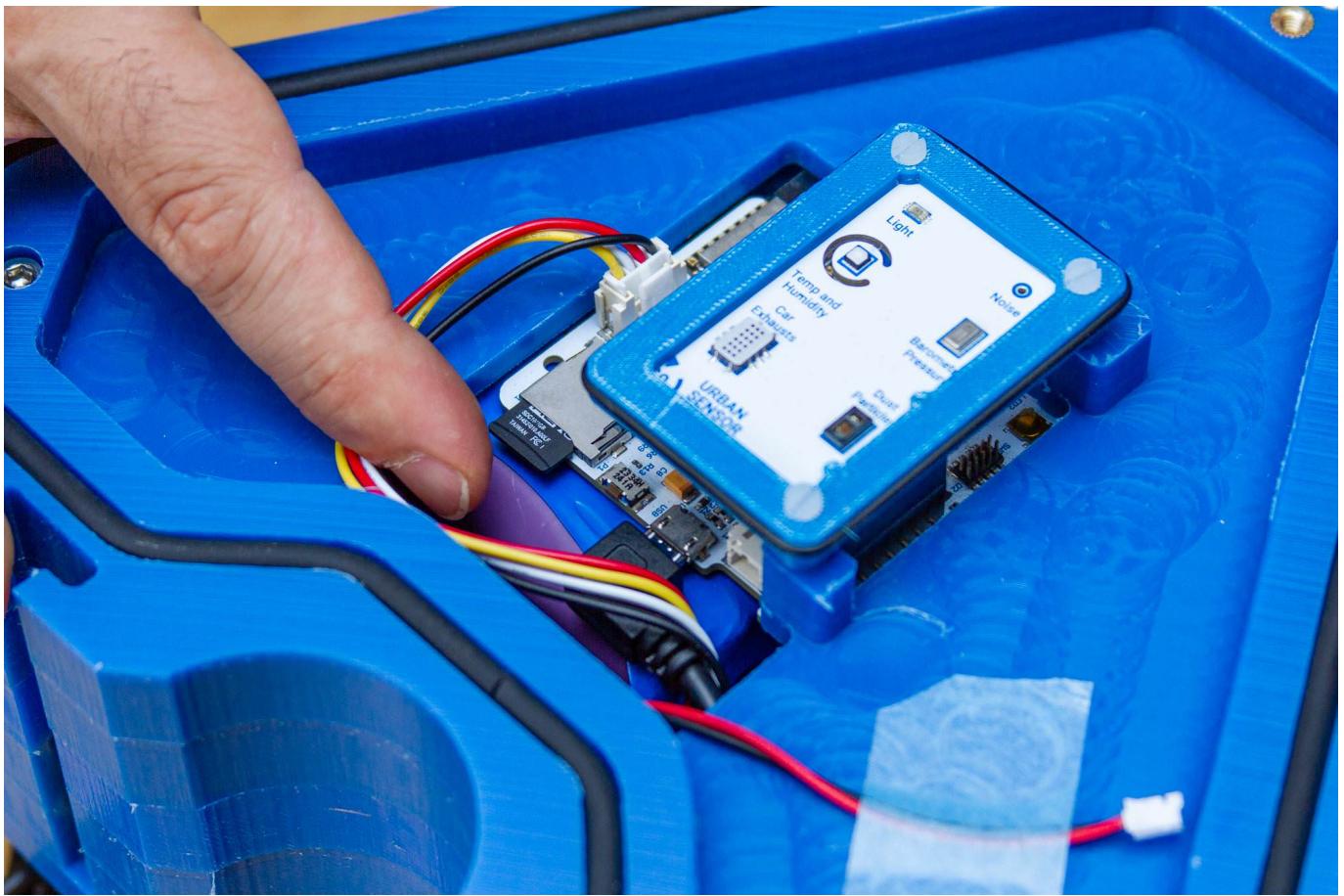
First of all, push the station button. Maybe it's simply off.

If this does not work, surely the station has been left without battery. You will have to charge it using the USB charger. Any other mobile charger will also work.

We will know that it is charging when the LED emits  orange pulses and once the battery is charged it will emit green 

The station does not store the data on the SD card.

Some SD cards may have problems over time. We can try formatting it but in case it does not work any micro SD card we buy at any mobile or computer store it will work. The size is not important and any micro SD or micro SDHC 512MB card up to 32GB will work.



Sensor Evaluation Campaign

Prior to sensor deployment for the intervention monitoring, some of the Living Lab Stations will be evaluated and compared against reference measurement under different conditions. They will be deployed in several cities among the iScape partners in order to 1. develop models for sensor calibration under different climatic and pollutant exposure conditions and 2. assess data quality. This campaign intends to evaluate the Living Lab Station before it's deployment, and trying to prevent concerns raised about data quality that other low-cost sensor platforms^{2 3 4}.

This evaluation will focus on the **real-world conditions calibration**, under wide range of exposure and climatic conditions, rather than developing tests in controlled conditions, as prior studies show discrepancies in the accuracy resulting from evaluation in laboratory conditions, versus that of outdoor conditions^{2 5 6}. The tests will be conducted by co-location of at least two stations per site with high-end sensors under the conditions indicated in the test section below.

The duration of the tests will be of 2,5 months, with two location changes. This is a compromise between the indications given in¹ for at least 3-months campaign and the availability of high-end sensors for the evaluation. Nevertheless, this campaign intends to cover a range of conditions by the deployment of the Living Lab Station in diverse conditions, not only climatic but also exposure-wise. The location changes will also intend to evaluate how well the sensors are able to adapt to these exposure and climatic changes¹⁰. The data will be uploaded to the SmartCitizen Platform and will be analysed using the Sensor Analysis Framework. The results of this evaluation in terms of models will be uploaded to a dedicated repository and will be implemented on the SmartCitizen Platform for on-the-fly sensor data processing. This processing aims to provide an open platform for sensor analysis using data analysis techniques, need which has been highlighted by^{2 5 6 9}.

As well, as stated in^{2 10 11}, it is necessary to perform individual field calibration for low-cost sensors if measurements comparable to those of high-end solutions are targeted. However, this calibration might not always be feasible in a wide range of conditions, leading to non-generalised models which can perform badly out of the training datasets. This test

campaign also aims to study this concern, with an evaluation for a cross calibration methodology, in which results from a limited subset of observations are applied to the complete dataset⁷. If successful, this would be set ground for the development of calibration strategies where the sensors are co-located with a high-end sensor and posteriorly deployed for citizen-science activities, or long term monitoring of the iScape Living Labs interventions, where high end sensors might not be available. This co-location could be performed in a recurrent manner, performing sequences of calibration-deployment-calibration, using merging calibrations as suggested in¹¹.

As a summary, this field campaign aims to cover the following points:

- Assess data quality levels and positioning with respect to the DQO set by the European Air Quality Directive
- Establish match scores for the different range of sensors available in the Living Lab Station
- Validation and assessment of EC sensor methodology for NO₂ and O₃ compounds in urban conditions (urban background and traffic) in various sites
- Validation of PMS PM raw data accuracy and effect of climatic conditions
- Calibration of Alphasense's EC sensors and PMS PM sensors for model quality improvement accounting for climatic conditions
- Feasibility assessment for the calibration of metal oxide sensor models with the use of reference data and/or Living Lab station data
- Validation of climatic sensors of the station itself (temperature, humidity, pressure)
- Drifts and stability:
 - Drifts and possible root causes for EC sensor sensitivities variations over time
 - Calibration stability for SGX MOS sensors
 - Sensor decay and recoverability of PMS sensors due to dust accumulation or others

TEST

The table below shows a description of the proposed test campaign:

Stage	Duration	Exposure	Reference equipment	Purpose
Pre-test	2 weeks	Urban Background	No	Stabilise electrochemical sensors to urban background on site. and verify overall functioning
Low Exposure test	1 month	Urban Background	Yes	Evaluate response in low transient areas and evaluate repeatability of urban background measurements in higher exposure testing phases
High Exposure test	1 month	Urban with traffic (canyon or junction)	Yes	Evaluate response in high transient / high concentration areas and validate current model and post-processing approach. Propose further models with more variables

The sites at which these calibration deployments are planned are:

Site	Season	Reference equipment	Duration
Bologna (Italy)	Summer	YES	1 month
Guildford (England)	Autumn	YES	2.5 months
Dublin (Ireland)	Autumn	YES	2.5 months
Bottrop (Germany)	Autumn	YES	2.5 months
Barcelona (Spain)	Spring	YES	>3 months

SENSOR INSTALLATION

Guidelines for representativeness of the results are given below:

Height

Between 2,5 and 3,5m. Not reachable by hand.

Reference equipment position

Within <2m and with similar exposure, air flow (both either on wall, or lamppost)⁷

Desirable measurements

- Chemical compounds (higher priority above):
 - NO₂
 - CO, O₃
 - NO_x, NO
 - NMHC

- Particulate Matter (higher priority above)
 - PM 2.5
 - PM 1.0, PM 10

- Climatic conditions (higher priority above)
 - Temperature and relative humidity
 - Wind speed and direction

Important Guidelines

Please, refer to the sensor considerations section for general information about the sensors. As well, take into account the following:

- Avoid direct exposure to intense sunlight for long periods of time, since this can severely affect the measurements (direct sun or intense transients).
- Avoid locations where high temperature or humidity transients are present since the sensor response is affected by these rapid changes.
- Avoid locations with low air flow or with direct exposure to air conditioning exhausts.

References

1. Spinelle L., Aleixandre M., Gerboles M. - 2013: Protocol of Evaluation and Calibration of Low-cost Gas Sensors for the Monitoring of Air Pollution. Joint Research Centre (Report EUR 26112 EN) []
2. Nuria Castell, Franck R. Dauge, Philipp Schneider, Matthias Vogt, Uri Lerner, Barak Fishbain, David Broday, Alena Bartonova - 2018: Can commercial low-cost sensor platforms contribute to air quality monitoring and exposure estimates? []
3. Snyder E., Watkins T., Solomon P., Thoma E., Williams R., Hagler G., Shelow D., Hindin D., Kilaru V., Preuss P. - 2013: The changing paradigm of air pollution monitoring. Environ. Sci. Technol. 47, 11369–11377 []
4. Lewis A., Edwards P. - 2016: Validate personal air-pollution sensors []
5. Spinelle L., Gerboles M., Villani M.G., Aleixandre M., Bonavitacola F. - 2015: Field calibration of a cluster of low-cost available sensors for air quality monitoring: Part A: Ozone and nitrogen dioxide []
6. Spinelle L., Gerboles M., Villani M.G., Aleixandre M., Bonavitacola F. - 2015: Field calibration of a cluster of low-cost available sensors for air quality monitoring: Part B: NO, CO and CO₂ []
7. David H. Hagan, Gabriel Isaacman-VanWertz, Jonathan P. Franklin, Lisa M. M. Wallace, Benjamin D. Kocar, Colette L. Heald, Jesse H. Kroll - 2018: Calibration and assessment of electrochemical air quality sensors by co-location with regulatory-grade instruments []
8. Olalekan A.M.Popoola, Gregor B.Stewart, Mohammed I.Mead, Roderic L.Jones - 2016: Development of a baseline-temperature correction methodology forelectrochemical sensors and its implications for long-term stability []
9. Sun L., ChunWong K., Wei P., Ye S., Huang H., Yang F., Westerdahl D., Louie P.K.K., Luk C.W.Y., Ning Z. - 2016. Development and application of a next generation air sensor network for the Hong Kong Marathon 2015. Air quality monitoring. Sensors 16, 211–229 []
10. A. Ripoll , M. Viana, M. Padrosa, X. Querol, A.Minutolo, K.M. Houc, J.M. Barcelo-Ordinas, J. Garcia-Vidal - 2018: Testing the performance of sensors for ozone pollution monitoring in a citizen science approach []
11. Philip J. D. Peterson, Amrita Aujla, Kirsty H. Grant, Alex G. Brundle, Martin R. Thompson, Josh Vande Hey and Roland J. Leigh - 2017: Practical Use of Metal Oxide Semiconductor Gas Sensors for Measuring Nitrogen Dioxide and Ozone in Urban Environments, Sensors 2017, 17, 1653 []

10.5 Enclosures

If we want to leave the kit on the outside for a few days you will need to provide it with extra protection. Below you can see the well-known 3D printed enclosure (for versions without PM sensor):



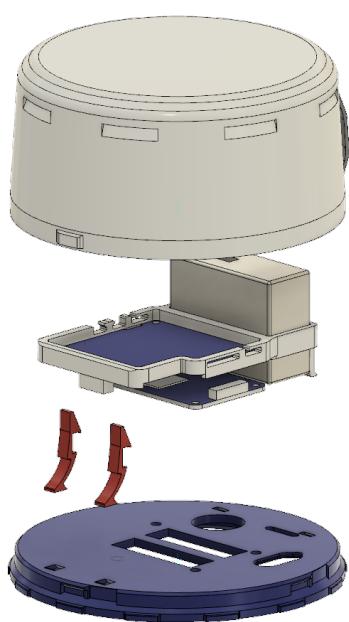
A note about the enclosures

Previous versions of the Smart Citizen Kit, without the PMS5003 sensor, included a 3D printed enclosure that holds the Data Board and Urban Sensor Board, as well as the lithium batteries. Different versions are available for the SCK 2.1, either with a CNC'ed version, or 3D printed one. Currently, there is no mass produced version at SEEED.

See the CNC'ed HDPE version:



Or the 3D printed one:



Want to contribute?

Visit the Smart Citizen Enclosures repository to download, modify, or add your own!

Build your own**Warning**

Keep in mind that casing is designed for short outdoor deployments. If you want a case for long exhibitions abroad, we will soon have a much more rugged enclosure ready! Also, feel free to explore all our enclosures repository for this and other versions of our hardware.

Step by step

First, you will need the two 3D printed clips. You can download the STL file and print them easily on any RepRap or similar FDM printer. If you don't know how to find a 3D printer, you can look for your nearest Fab Lab or use 3D Hubs.

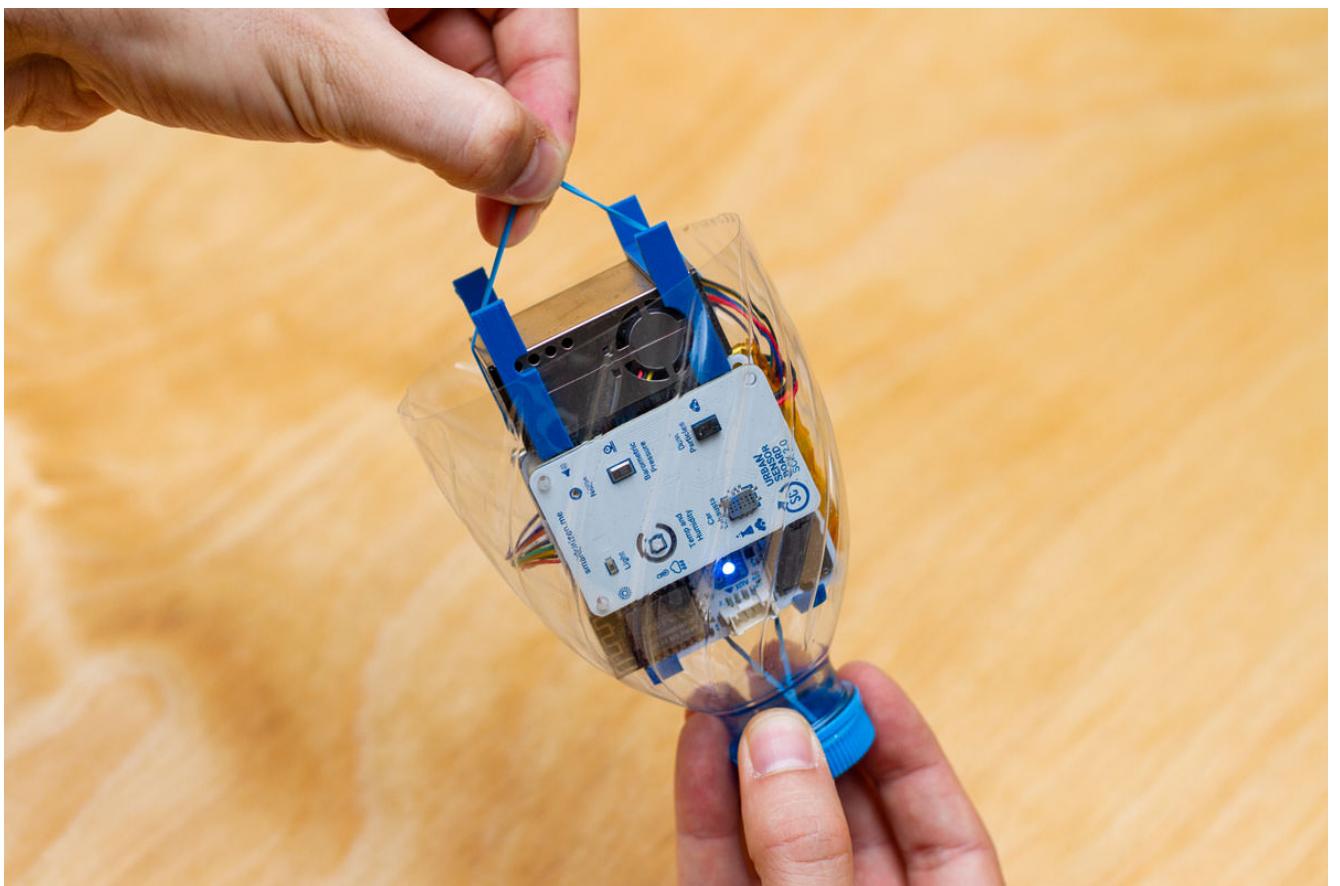
1. Use scissors to cut an empty plastic bottle at about 12 cm from the top



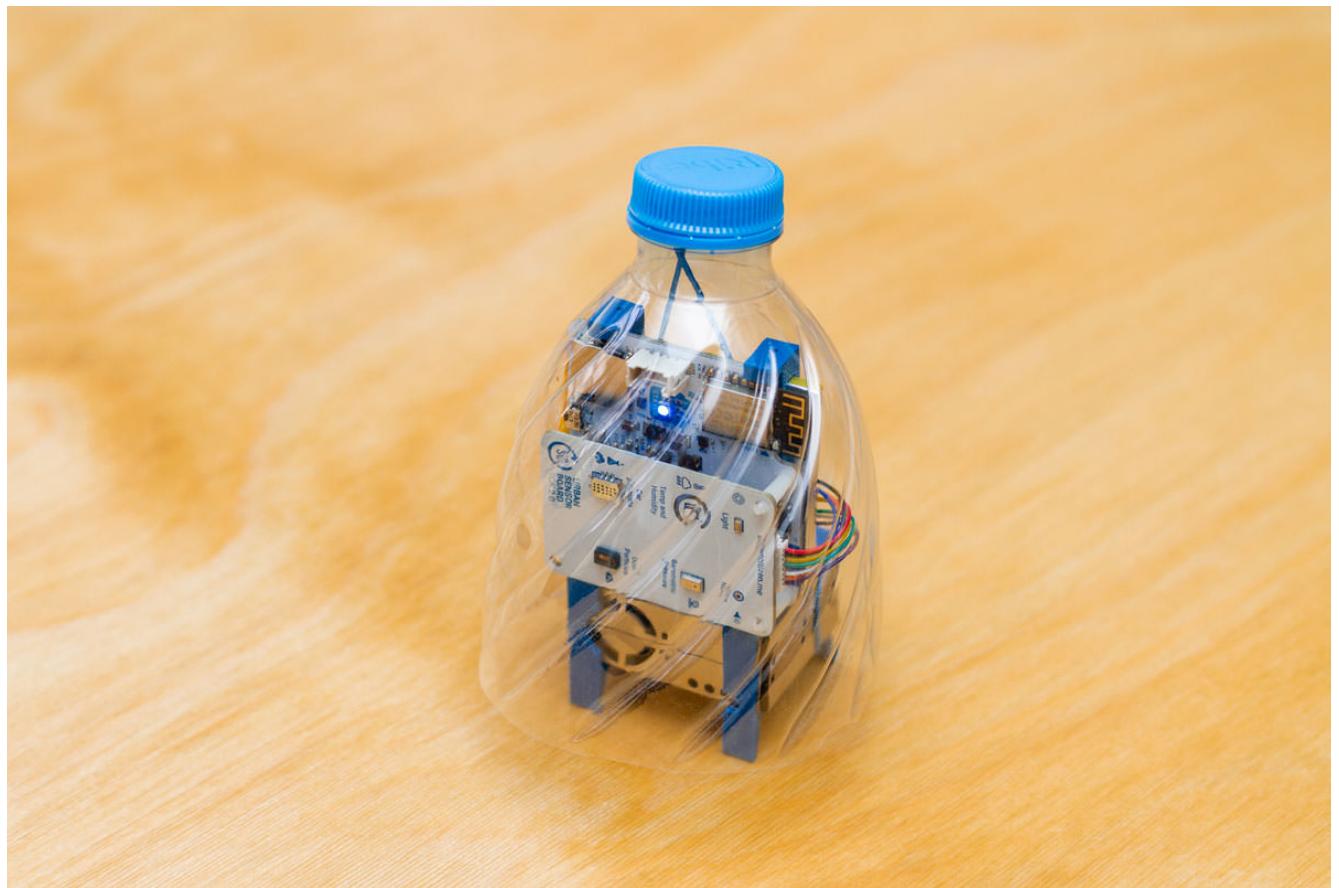
2. Use the rubber band to fix it using the bottle cap



3. Place the Kit inside and use the rubber band to hold it



4. You have now a simple enclosure to use your Kit outdoors for short measurement periods!



You can now install the sensor outdoors!



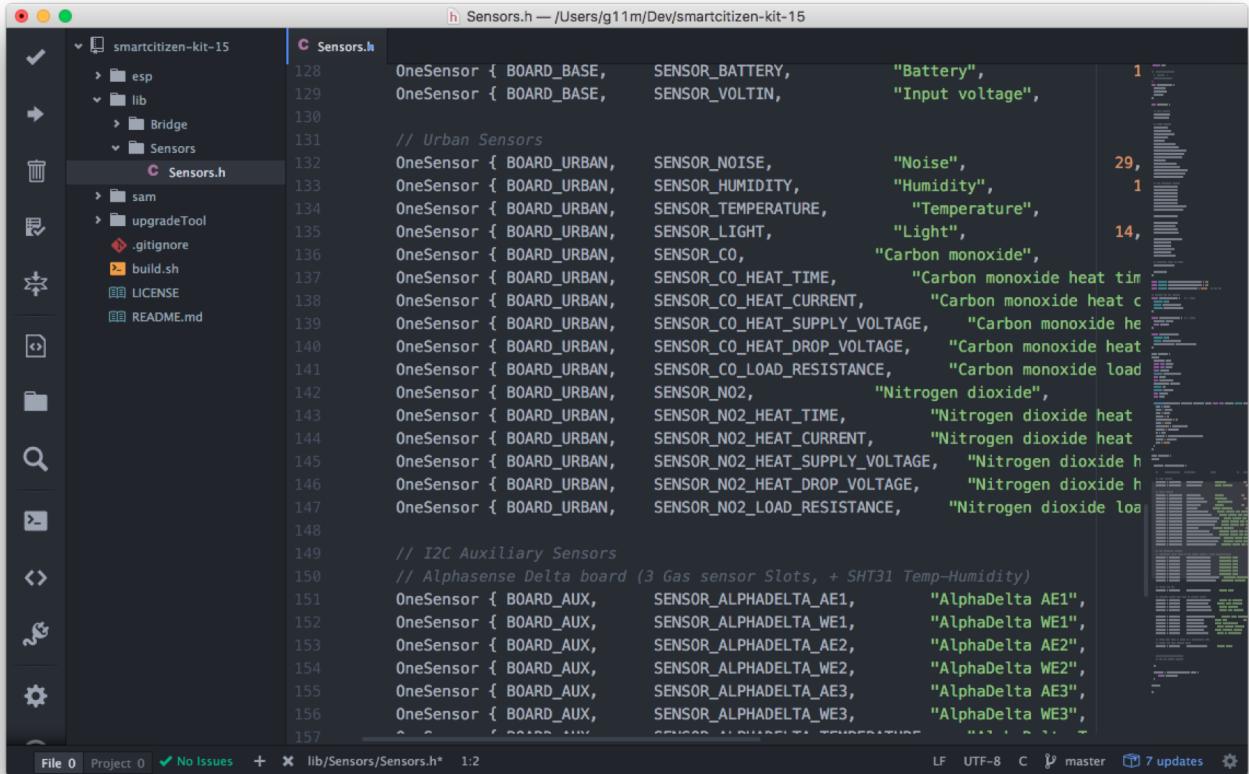
10.6 Firmware

The firmware is OOP and is entirely written in C++. Both processes the core ARM MCU and the ESP8266 WIFI are developed as part of the same framework integrating seemingly by using a set of bridge libraries that provide a unifies the RPC architecture.

A note about versions

The first version of the software was initially developed for the Making Sense project under European Community's H2020 Programme under Grant Agreement No. 688620.

The current version has been funded by the iSCAPE project project under European Community's H2020 Programme under Grant Agreement No. 689954.



```

Sensors.h — /Users/g11m/Dev/smartsitizen-kit-15
128 OneSensor { BOARD_BASE, SENSOR_BATTERY, "Battery",
129 OneSensor { BOARD_BASE, SENSOR_VOLTIN, "Input voltage",
130
131 // Urban Sensors
132 OneSensor { BOARD_URBAN, SENSOR_NOISE, "Noise",
133 OneSensor { BOARD_URBAN, SENSOR_HUMIDITY, "Humidity",
134 OneSensor { BOARD_URBAN, SENSOR_TEMPERATURE, "Temperature",
135 OneSensor { BOARD_URBAN, SENSOR_LIGHT, "Light",
136 OneSensor { BOARD_URBAN, SENSOR_CO, "Carbon monoxide",
137 OneSensor { BOARD_URBAN, SENSOR_CO_HEAT_TIME, "Carbon monoxide heat tim
138 OneSensor { BOARD_URBAN, SENSOR_CO_HEAT_CURRENT, "Carbon monoxide heat c
139 OneSensor { BOARD_URBAN, SENSOR_CO_HEAT_SUPPLY_VOLTAGE, "Carbon monoxide he
140 OneSensor { BOARD_URBAN, SENSOR_CO_HEAT_DROP_VOLTAGE, "Carbon monoxide heat
141 OneSensor { BOARD_URBAN, SENSOR_CO_LOAD_RESISTANCE, "Carbon monoxide load
142 OneSensor { BOARD_URBAN, SENSOR_N02, "Nitrogen dioxide",
143 OneSensor { BOARD_URBAN, SENSOR_N02_HEAT_TIME, "Nitrogen dioxide heat
144 OneSensor { BOARD_URBAN, SENSOR_N02_HEAT_CURRENT, "Nitrogen dioxide heat
145 OneSensor { BOARD_URBAN, SENSOR_N02_HEAT_SUPPLY_VOLTAGE, "Nitrogen dioxide h
146 OneSensor { BOARD_URBAN, SENSOR_N02_HEAT_DROP_VOLTAGE, "Nitrogen dioxide h
147 OneSensor { BOARD_URBAN, SENSOR_N02_LOAD_RESISTANCE, "Nitrogen dioxide lo
148
149 // I2C Auxiliary Sensors
150 // Alphasense Delta board (3 Gas sensor Slots, + SHT31 Temp-Humidity)
151 OneSensor { BOARD_AUX, SENSOR_ALPHADELTA_AE1, "AlphaDelta AE1",
152 OneSensor { BOARD_AUX, SENSOR_ALPHADELTA_WE1, "AlphaDelta WE1",
153 OneSensor { BOARD_AUX, SENSOR_ALPHADELTA_AE2, "AlphaDelta AE2",
154 OneSensor { BOARD_AUX, SENSOR_ALPHADELTA_WE2, "AlphaDelta WE2",
155 OneSensor { BOARD_AUX, SENSOR_ALPHADELTA_AE3, "AlphaDelta AE3",
156 OneSensor { BOARD_AUX, SENSOR_ALPHADELTA_WE3, "AlphaDelta WE3",
157

```

Firmware updates are done via the micro USB port using the Platform IO software available for Linux, Mac and Windows.

10.6.1 Architecture

Core Microcontroller

Name	Functions
Pins	Definition for the MCU pinout
Sensors	Definition for all the sensors supported
Sensors	Absracts the sensors on a common interface
SckBase	Manages the core operations: power, connectivity, peripherials
SckAux	Manages the sensors connected on the AUX connector
SckUrban	Manages the sensors on the Urban Sensor Board
SckCharger	Manages the battery charging process
SckButton	Manages users button interaction actions
SckLed	Manages light status for user feedback
Commands	Library to absracts the core features on to a simple shell interface
ReadLight	Manages configuration over light
ReadSound	Manages configuration over sound

DEPENDENCIES

- Adafruit INA219 Library
- Adafruit MPL3115A2 Library
- Adafruit SHT31 Library
- Arduino Json Library
- Arduino Low Power@ Library
- ArduinoZero PMUX Report Library
- DS2482 Library
- FlashStorage Library
- FlashStorage
- MCP342X Library
- RadioHead Library
- RTCZero Library
- SdFat Library
- SmartSmart Citizen Kit Gases Pro Board Library
- SparkFun BQ27441 Arduino Library
- SparkFun MAX3010x Library
- SPIFlash Library
- U8g2 Library

WiFi Module

Name	Functions
SckESP	Runs all the Wi-Fi networking related functions

Dependencies

- Time Library
- ArduinoJson Library
- RemoteDebug Library
- RemoteDebug Library
- RadioHead Library
- RadioHead Library
- PubSubclient Library

Shared

Name	Functions
Config	Provides a shared configuration between the two MCUs

10.6.2 Data management

The board is capable of storing the recorded data offline on its internal dedicated flash memory of 8MB and later publish this over Wi-Fi connectivity provided by an Espressif ESP8266. Data is published using MQTT messages to the Smart Citizen Platform. NTP is used for syncing the built-in RTC. For long term offline storage, the board provides a standard microSD socket where card in the orders of GB can be employed. That ensures extended periods of data in the order of decades can be stored.

Configuration

The board firmware is fully customizable without requiring any changes to the core software. That includes enabling or disabling sensors, the sampling frequency of the sensors or the operation mode. There different configuration options via the Serial Shell available when the board is connected over USB.

```
Detecting: AlphaDelta 1A... found, Enabling AlphaDelta 1A
Detecting: AlphaDelta 1W... found, already enabled!!!
Detecting: AlphaDelta 2A... found, already enabled!!!
Detecting: AlphaDelta 2W... found, already enabled!!!
Detecting: AlphaDelta 3A... found, already enabled!!!
Detecting: AlphaDelta 3W... found, already enabled!!!
Detecting: AlphaDelta Temperature... found, already enabled!!!
Detecting: AlphaDelta Humidity... found, already enabled!!!
Detecting: Groove ADC... nothing!
Detecting: INA219 Bus voltage... nothing!
Detecting: INA219 Shunt voltage... nothing!
Detecting: INA219 Current... nothing!
Detecting: INA219 Load voltage... nothing!
Detecting: DS18B20 Water temperature... nothing!
Detecting: Atlas PH... nothing!
Detecting: Atlas Conductivity... nothing!
Detecting: Atlas Specific gravity... nothing!
Detecting: Atlas Dissolved Oxygen... nothing!
Detecting: Atlas DO Saturation... nothing!
Detecting: Groove OLED... nothing!
```

10.6.3 Shell

The firmware provides a comprehensive command shell over USB to manage all the kits functionalities for advanced users.

Use any Serial console as `screen`, `platformio device monitor`, or the serial monitor on the Arduino IDE

Info

Have a look at the guide for different platforms here

10.6.4 Storage

Readings files YYYY-MM-DD.CSV

These files are generated and updated by the kit in a daily manner. When a SD is detected the SCK will automatically save the sensors into it.

The SCK creates an additional CSV file once there is a hardware reset. A reset takes place every night at 3-4am with the purpose to avoid data loss because a software problem (i.e. blocked software). The SCK then stores the data in a file with a sequential name, and does so by changing the filename to YYYY-MM-DD.01, .02... depending on the amount of resets it sees during a certain day. The latest data is always in the file with .CSV extension. An example of a day with two resets (ad hoc and the programmed one):

```
YYYY-MM-DD.01 -> first reset  
YYYY-MM-DD.02 -> second reset  
YYYY-MM-DD.CSV -> latest file
```

The user can safely change the extension of these files back to .CSV and concatenate them:

```
YYYY-MM-DD.01 -> YYYY-MM-DD_01.CSV  
YYYY-MM-DD.02 -> YYYY-MM-DD_02.CSV  
YYYY-MM-DD.CSV -> YYYY-MM-DD.CSV
```

Warning

If there is a problem with the device, sometimes it can be that the SD card contains many files for a single day. These resets might go unnoticed, and the SD card files can be a way of detecting an issue.

Debug log file DEBUG.txt

The debug file is generated and updated by the kit, only if the debug mode is enabled on the configuration.

When the debug mode is enabled the verbosity level of this file is defined by the outlevel (*normal, verbose or silent*).

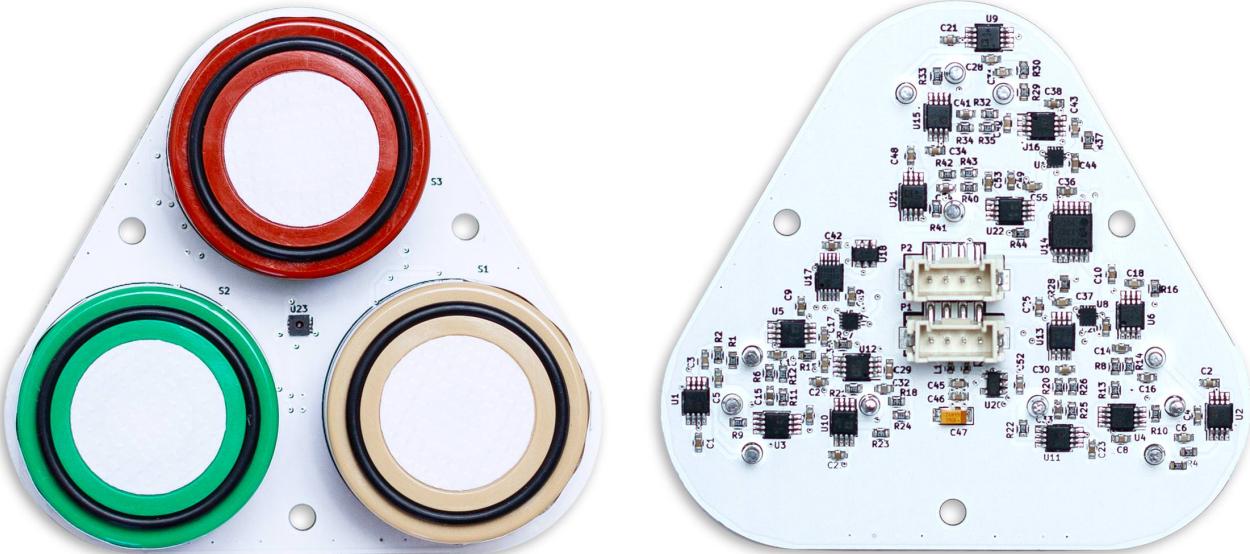
10.6.5 Source files

[Download](#)

[Check the source code](#)

10.7 Gases Pro Sensor Board

The Gases Sensor Board is a custom, ultra-low noise, high-performance, low power, digital output driver for 3 Alphasense Ltd. Electrochemical Series B Gas Sensors specifically designed for the project from the ground up.



Check the source code

10.7.1 Sensor measurements

Measurement	Units	Sensor
Carbon Monoxide	ppm	Alphasense CO-B4
Nitrogen Dioxide	ppb	Alphasense NO2-B43F
Ozone	ppb	Alphasense OX-B431

10.7.2 Sensors selection

The following characteristics have been considered for the sensor choice

- The driver's board designed includes a temperature and humidity sensor for calibrating the temperature dependence of the sensing subsystem.
- Same technology as the A4 series but more robust when exposed to outdoor environments 24/7.
- Designed for fixed site air quality networks which demand longer term reliability.
- Manufacturers provide the baseline resistance calibration values per sensor allowing corrections to be easily applied.
- Low power consumption

The Alphasense EC Sensors were selected to provide a higher linearity, repeatability and resolution than the SGX MICS MO Gas Sensors found on the Urban Sensor Board.

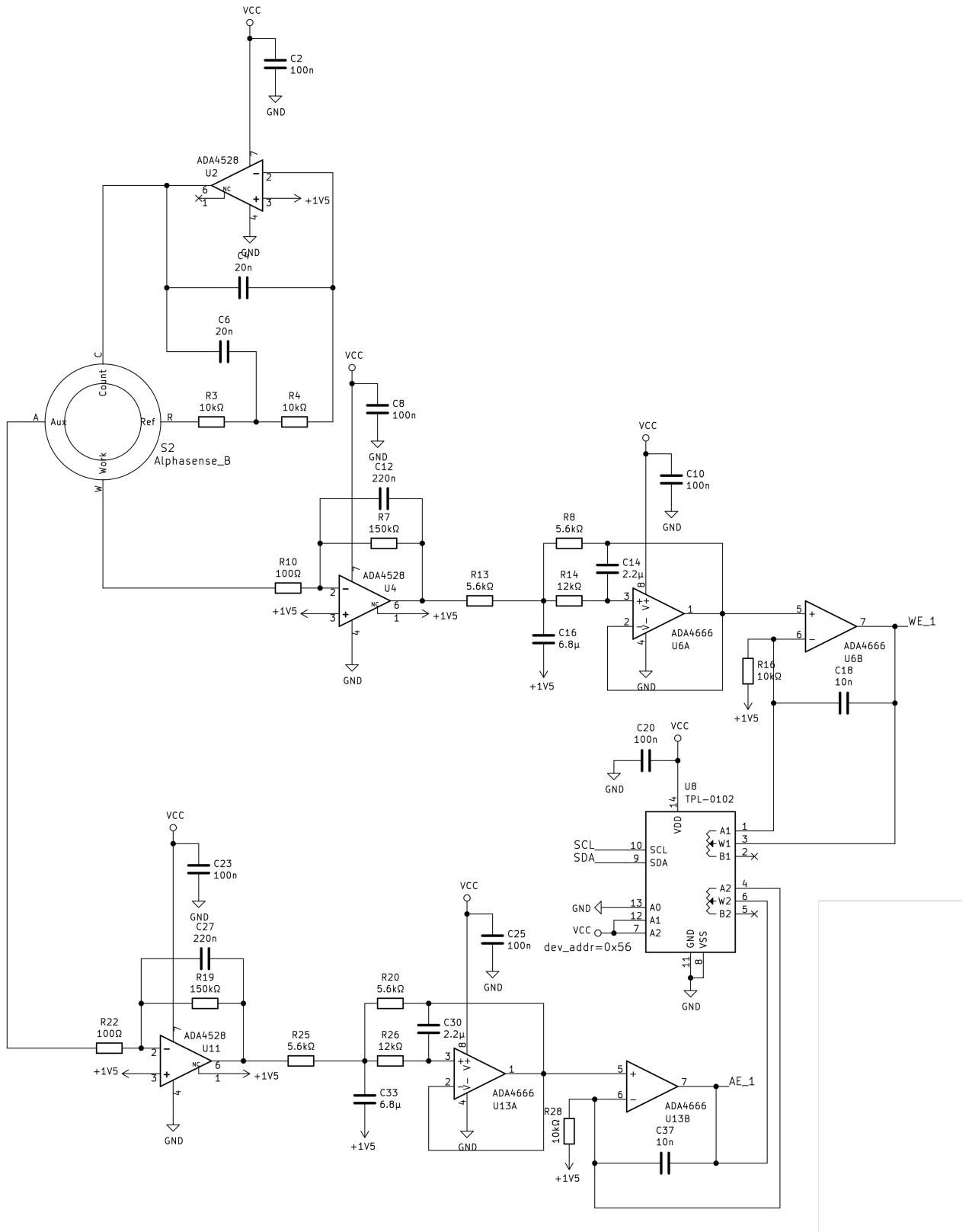
The selection of the sensors was based on the wide variety of literature available on them. Both Penza and EuNetAir Consortium (2014) and Mead et al. (2013) test the NO₂A1-A3 against reference instruments, in the laboratory as well as in the field, with well-correlated results. The former concluded that the Data Quality Objective for "indicative measurements" (European Parliament and Council of the European Union, 2008) is fulfilled, and the latter report sensitivity in the low ppb region with high linearity. Spinelle et al. tested the Alphasense NO₂B4 and O₃B4 in a field experiment, with various calibration approaches. Performance evaluation of the same sensors was performed later including a test on a wide range of performance parameters (e.g. response time, calibration function, repeatability, drift, hysteresis effect, and matrix effect) (Spinelle et al. 2017). The experiment found a strong correlation with reference instruments ($R^2 > 0.9$) and identified some cases with significant hysteresis effect related to humidity. In chamber conditions, the performances of the Alphasense CO-B4 was found to be excellent, with the R^2 values being greater than 0.9 (Castell et al. 2017) (Mead et al. 2013) (Sun et al. 2016). Two field studies reported moderate to excellent R^2 values (0.53--0.97) for the CO-B4 sensor (Castell et al. 2017) (Mead et al. 2013). Finally, some calibration approaches as detailed in Popoola et al. (2016) and Hagan et al. (2018) are used in the post-processing stage as a basis for pollution concentration calculations.

10.7.3 Design

Each of the three drivers for Alphasense Ltd. Series B Sensors is built around the same design. They include a three stage adjustable amplifier design for the working electrode and another symmetrical design for the auxiliary electrode. Both signals are then feed to a high accuracy delta-sigma A/D converter with differential inputs 18 bits of resolution. All the parameters are digitally adjustable via I²C from the **Data Board**. Each board also include a unique identifier chip allowing the firmware on the **Data Board** to identify the board and apply the corresponding calibration values and a humidity and temperature sensor.

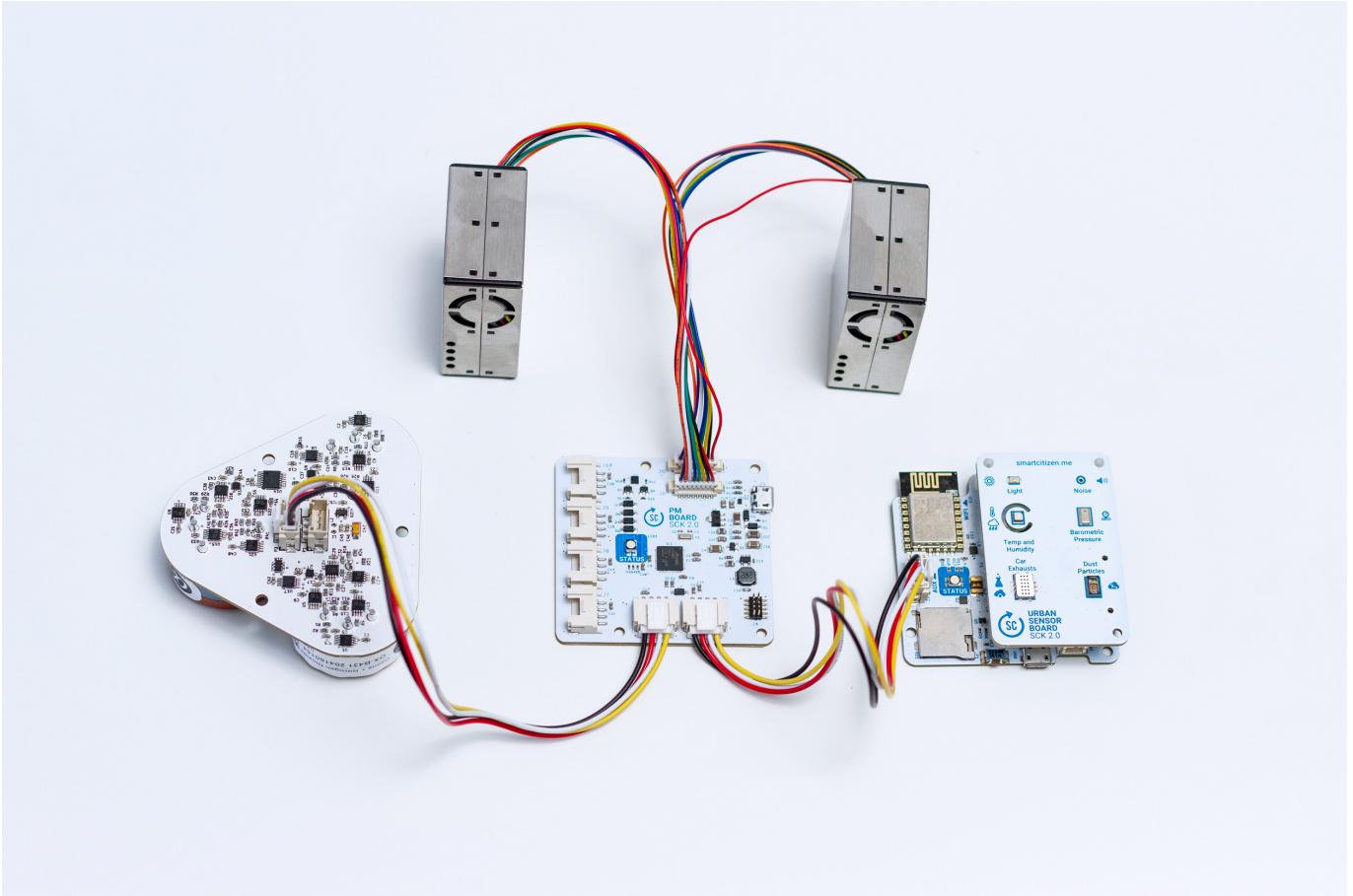
Info

Visit the source files section to download the complete schematics.



10.7.4 Setup

The board is connected to the Data Board using the AUX connector. Before, the Alphasense sensors need to be in place and properly registered using the board id. The board will be autodetected by the main Firmware running on the Data Board. Multiple sensor board can be daisy-chained as seen on the image.



Validation

This board is currently being evaluated under the iScape Project and the results will be public soon!

10.7.5 Source files

[Download](#)

[Check the source code](#)

1. ALPHASENSE NO2-B43F Technical Datasheet

<http://www.alphasense.com/WEB1213/wp-content/uploads/2017/07/NO2B43F.pdf>

2. ALPHASENSE OX-B431 Technical Datasheet

<http://www.alphasense.com/WEB1213/wp-content/uploads/2017/07/OX-B431.pdf>

3. ALPHASENSE CO-B4 B Technical Datasheet

<http://www.alphasense.com/WEB1213/wp-content/uploads/2015/04/COB41.pdf>

10.8 Noise Sensor Implementation

10.8.1 Firmware

Audio I2S Library

A custom library for audio analysis.

 Work in progress

Audio I2S Base library, intenedt to be generic purpose audio analysis library for an I2S Microphone on the SAMD21 with:

- FFT Analysis
- FIR Analysis
- Custom window selection
- Custom weighting function selection
- Custom buffer size and custom fft bin size (in case of FFT analyser)
- Custom equalisation
- Octave auto generation of .h files for coefficients and so on

Smart Citizen Firmware

Smart Citizen Firmware Firmware implementation in the SmartCitizen Kit 2.0 and 2.1, with a better usage of memory and SCK related functionalities:

- FFT analysis
- Selection of A or C weighting through LUT
- Two user cases:
 - General audio analysis with fixed buffer size and fixed FFT bins size ($f_s = 44,1\text{kHz}$)

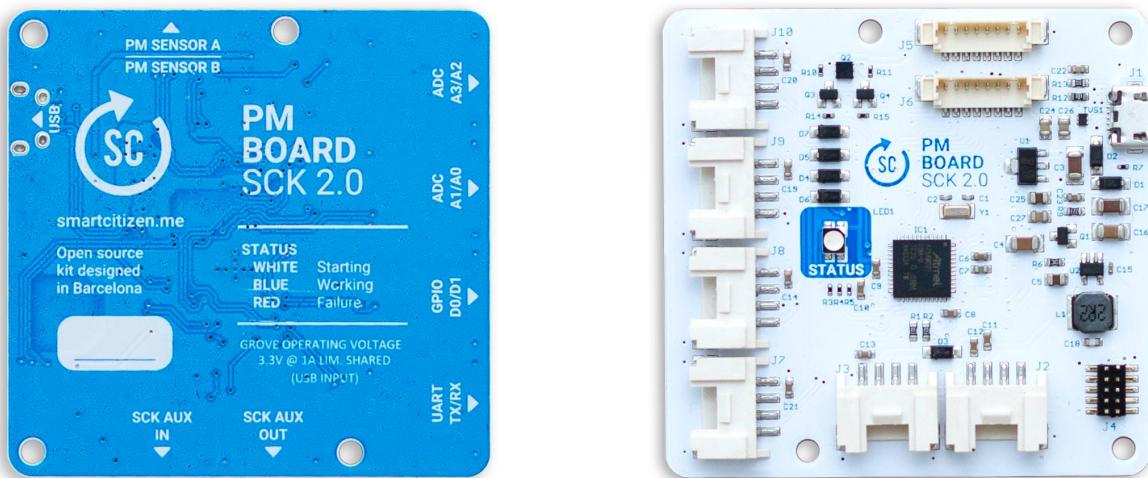
10.8.2 Source files

Download

Check the source code

10.9 PM Sensor Board

The PM Sensor Board is based around Plantower PMS 5003¹ a digital particle concentration sensor that uses the Laser Scattering principle to obtain the number of suspended particles in the air. This includes a custom designed PCB with an MCU to provide I2C connectivity with the Data Board.



Check the source code

10.9.1 Sensor measurements

Measurement	Units	Sensor
PM 1	µg/m3	Plantower PMS5003 Dual System
PM 2.5	µg/m3	Plantower PMS5003 Dual System
PM 10	µg/m3	Plantower PMS5003 Dual System

10.9.2 Sensors selection

The following characteristics have been considered for the sensor choice:

- Provides PM 2.5 and PM 10 measurements in $\mu\text{g}/\text{m}^3$
- Minimal distinguishable particle diameter of 0.3 μm
- No need for external ADC or linearization circuits. The sensor includes an internal MCU capable of dealing with all the light emitting and sensing processing. All the communication is done using the I2C protocol. A dedicated driver has been designed for this.
- Ultra Low Cost when compared to other commercial solutions with similar performance
- Low Power

The selection is based on the academic references selected above. For a complete Low-Cost Sensors Evaluation see recommendations for application and the subsequent publication (*Rai et al. 2017*).

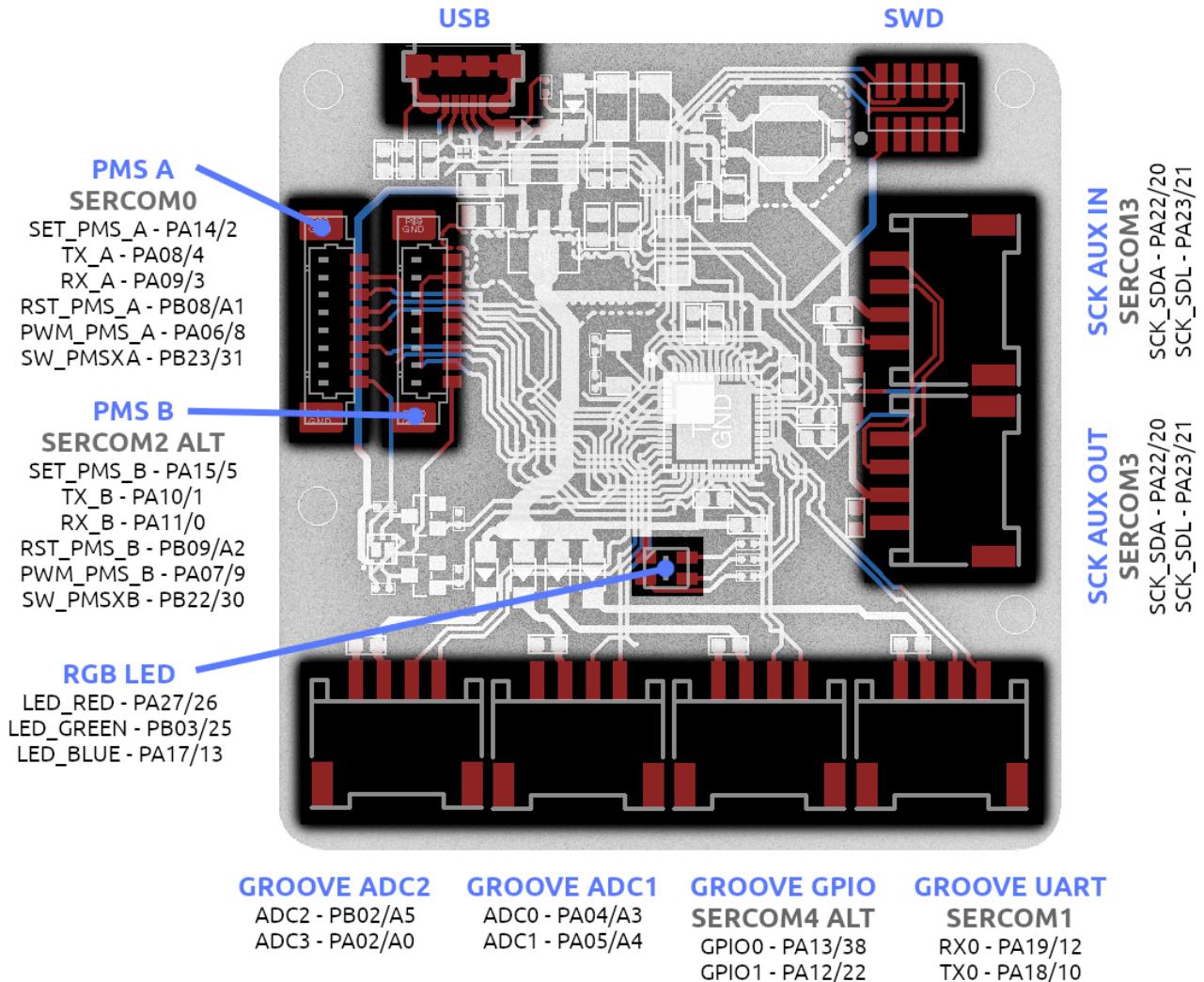
10.9.3 Design

The PM Sensor Board runs a dedicated ARM M0+ 32-bits, the same as the Data Board to provide a unified hardware architecture. The board includes an highly efficient step up to provide 5V to drive the PM sensors and a disable/enable circuit to turn off the sensor by software.

Info

Visit the source files section to download the complete schematics.

Pinout



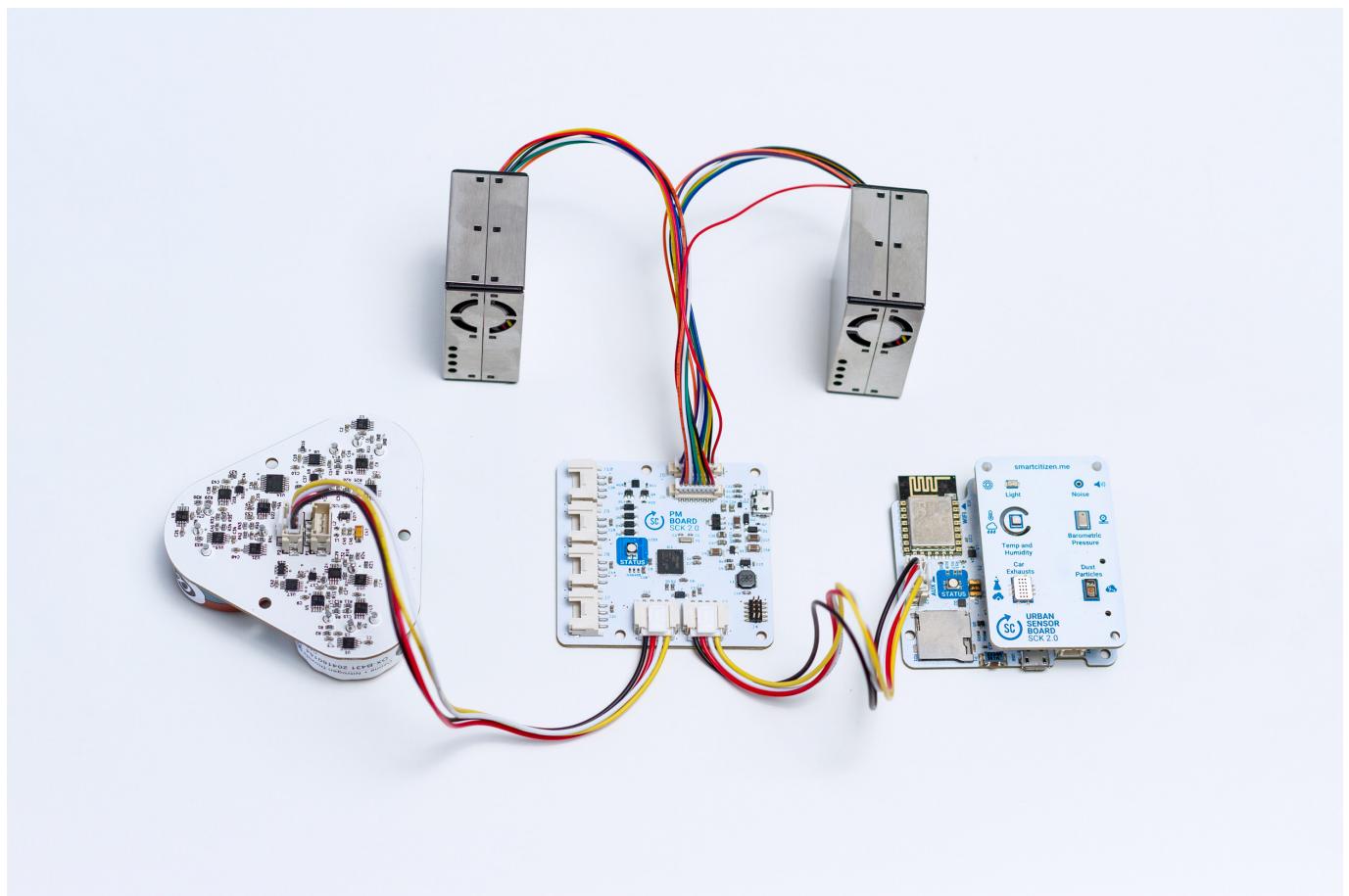
PIN_SAM	Pin	PM 1.0 (rev10)	PM driver 1.1	Arduino pin	SERCOM	SERCOM (alt)	SERCOM USED PM 1.1
1	PA00			Used by xtal		SERCOM1.0	
2	PA01			Used by xtal		SERCOM1.1	
3	PA02		ADC3	A0			
4	PA03			AREF			
7	PB08	RST_PMS_A	RST PMS A	A1		SERCOM4.0	
8	PB09	RST_PMS_B	RST PMS B	A2		SERCOM4.1	
9	PA04	TX_A	ADC0	A3		SERCOM0.0	
10	PA05	RX_A	ADC1	A4		SERCOM0.1	
11	PA06	TX_B	PWM_PMS_A	D8		SERCOM0.2	
12	PA07	RX_B	PWM_PMS_B	D9		SERCOM0.3	
13	PA08	ADC0	TX_A	D4	SERCOM0.0	SERCOM2.0 SERCOM2.0 (alt)	
14	PA09	ADC1	RX_A	D3	SERCOM0.1	SERCOM2.1 SERCOM2.1 (alt)	
15	PA10	PWM_PMS_A	TX_B	D1	SERCOM0.2	SERCOM2.2 SERCOM0.2	
16	PA11	PWM_PMS_B	RX_B	D0	SERCOM0.3	SERCOM2.3 SERCOM0.3	
19	PB10			D23 / MOSI		SERCOM4.2	
20	PB11			D24 / SCK		SERCOM4.3	
21	PA12	GPIO0	GPIO1	D22 / MISO	SERCOM2.0	SERCOM4.0 SERCOM4.0 (alt)	
22	PA13	GPIO1	GPIO0	Used by EDBC	SERCOM2.1	SERCOM4.1 SERCOM4.1 (alt)	
23	PA14	SET_PMS_A	SET PMS A	D2	SERCOM2.2	SERCOM4.2	
24	PA15	SET_PMS_B	SET PMS B	D5	SERCOM2.3	SERCOM4.3	
25	PA16	TX0		D11	SERCOM1.0	SERCOM3.0	
26	PA17	RX0	LED_BLUE	D13 / LED	SERCOM1.1	SERCOM3.1	
27	PA18	BLUE	TX0	D10	SERCOM1.2	SERCOM3.2	SERCOM1.2
28	PA19	GREEN	RX0	D12	SERCOM1.3	SERCOM3.3	SERCOM1.3
29	PA20	RED		D6	SERCOM5.2	SERCOM3.2	
30	PA21			D7	SERCOM5.3	SERCOM3.3	
31	PA22	SDA	SCK_SDA	D20 / SDA	SERCOM3.0	SERCOM5.0	SERCOM3.0
32	PA23	SCL	SCK_SCL	D21 / SCL	SERCOM3.1	SERCOM5.1	SERCOM3.1
33	PA24	D-	D-	Used by USB	SERCOM3.2	SERCOM5.2	
34	PA25	D+	D+	Used by USB	SERCOM3.3	SERCOM5.3	
37	PB22	SW_PMSXB	SW_PMSXB	D30 / EDBG TX		SERCOM5.2	
38	PB23	SW_PMSXA	SW_PMSXA	D31 / EDBG RX		SERCOM5.3	
39	PA27	ID	LED_RED	TX			
41	PA28	pull up	ID	Used by USB			
45	PA30	SWCLK	SWCLK	Used by SWCLK		SERCOM1.2	
46	PA31	SWDIO	SWDIO	Used by SWDIO		SERCOM1.3	
47	PB02	ADC2	ADC2	A5		SERCOM5.0	
48	PB03	ADC3	LED_GREEN	D25 / RX LED		SERCOM5.1	

SERCOM DISTRIBUTION

SERCOM0.0	SERCOM1.0	SERCOM2.0 (alt)	SERCOM3.0	SERCOM4.0 (alt)	SERCOM5.0
PMA	UART	PMB (alt.)	I2C	I2C/UART (alt.)	-
TX_B (0.2)	TX0 (1.2)	TX_A (2.0)	SCK_SDA (3.0)	GPIO1 (4.0)	
RX_B (0.3)	RX0 (1.3)	RX_A (2.1)	SCK_SCL (3.1)	GPIO0 (4.1)	

10.9.4 Setup

The board is connected to the Data Board using the AUX connector. Before, the Plantower PMS sensors need to be connected. The board will autodetect the PMS sensors and present them seamlessly to the main Firmware running on the Data Board. Multiple sensor board can be daisy-chained as seen on the image.



10.9.5 Extra sensors

Dallas OneWire support



See [datasheet](#)

Support for rugged version of DS18B20 was added in this commit, the sensor is autodetected on boot when connected to the PM board GPIO Grove port.

10.9.6 Source files

[Download](#)

[Check the source code](#)

1. PLANTOWER PMS5003 Technical Datasheet <https://aqicn.org/air/view/sensor/spec/pms5003.pdf>

10.10 Soil

This page compiles examples of sensors for soil and water monitoring, developed during the GROW project.

A note about versions

The following research has been funded by the Grow Observatory project under European Community's H2020 Programme under Grant Agreement No. 690199.



10.10.1 Moisture Sensor

The Chirp Sensor is a low cost moisture and temperature sensor developed by WeMakeThings: a hackers and engineers collective based in Vilnius, Lithuania. Their hardware and software are fully open-source, and it can be easily integrated but also replicated and customized for new projects.

Chirp

The sensor uses capacitive sensing to measure soil's moisture. A 1MHz square wave is output from the chip through a resistor into a big pad that, together with the surrounding ground plane, it forms a parasitic capacitor. The resistor and the capacitor create a low pass filter which cut-off frequency changes with changing capacitance. The soil around the sensor acts as an electrolyte whose dielectric constant changes depending on the amount of moisture in it, so the capacitance of our makeshift capacitor changes too. The filtered square wave is then fed into a peak detector formed of out a diode and a capacitor. An ADC measures this voltage in the microcontroller. The sensor also includes a temperature sensor with a calculated absolute measurement accuracy around 2%.

There are different versions of the Chirp sensor, and for this application we chose the Chirp I2C sensor. The sensor was integrated on to the SCK's firmware, and it is automatically recognized by the board once it is plugged into the SCK using the Aux sensor connector. A Grove 4 pin Female Jumper to Grove will need to be used with the sensor to connect it to the SCK. The original Chirp sensors come coated with PRF202 - a moisture resistant varnish for electronics, but it is not enough for actual deployment. For such, one must add additional protection to the whole sensor. We suggest polyester or epoxy resin. However, you must note that sensitivity of the sensor will decrease depending on how thick the layer you are going to apply and might need to be recalibrated. We also recommend covering the electronics with heat shrink to fully waterproof the sensor. Some versions already include a pre-ruggedized sensor, which is a recommended solution for a faster use.

Sensor calibration

The soil moisture sensor can be used for schedule irrigations (i.e. determine when to water the plants); or for calculating soil water deficit to work out how much water to apply. Depending on the application, the sensor would need to be calibrated in with different procedures, but as a general guideline, we need to normalise its readings. Without this process, the raw sensor readings will be meaningless to the user and only some trends could be analysed. This section is a digest of some of these procedures, and more information is given in the notes below.

More references

- Capacitance probe calibration
- The importance of soil moisture sensor calibration

In case of **irrigation scheduling**, it is generally sufficient to simply match the raw readings from each sensor at both 0% (held in air) and 100% water levels (submerged in water). This is, of course, an approximation and will need some further analysis from the user to determine when to irrigate. When a **more accurate measurement is required**, the sensor needs to be calibrated with the actual soil where it's going to be deployed, since different types of soil will have different capacities. A valid approach is to prepare different samples of the soil with different levels of saturation, and adapt the sensor readings for it.

Image Source: Edaphic Scientific

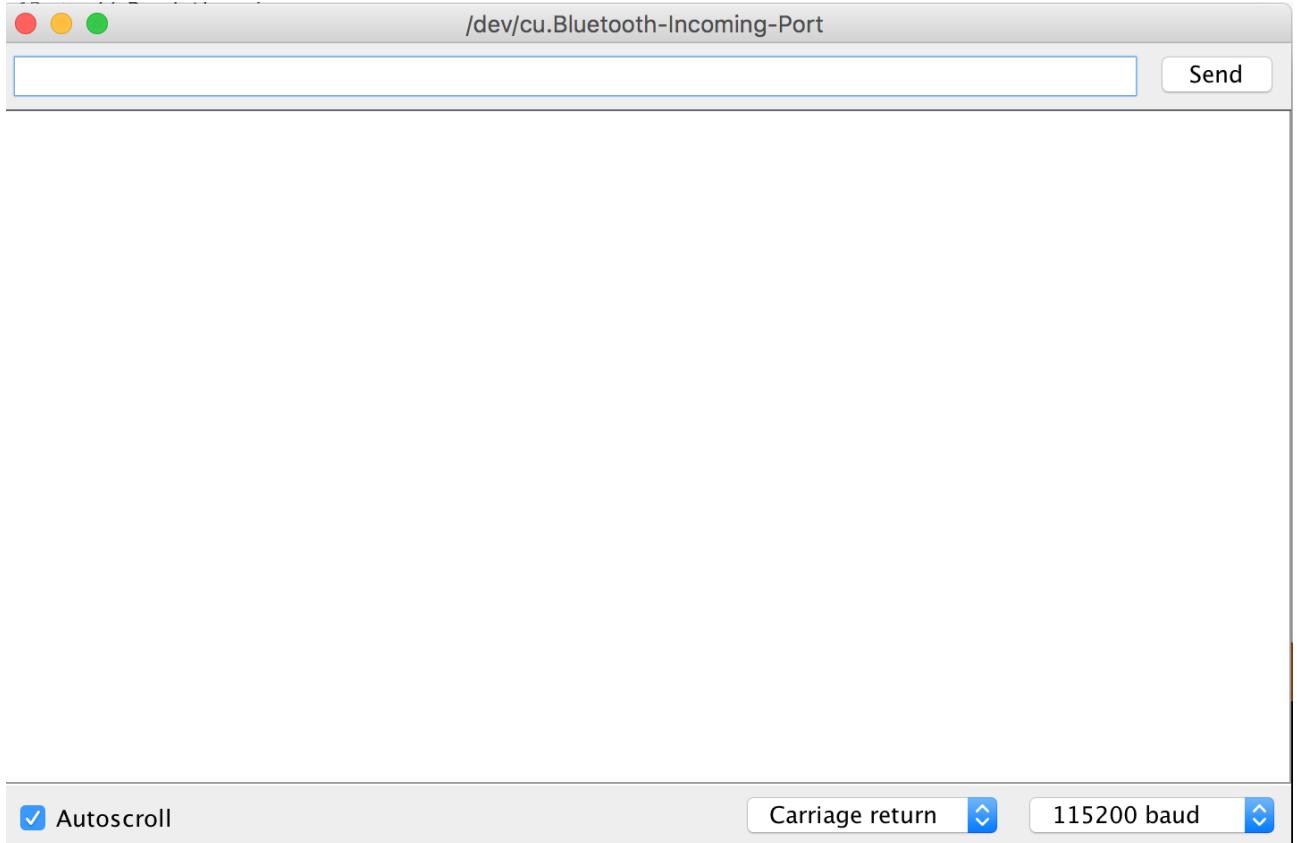
Calibrate your sensor

If we are not aiming to get a full-fledged sensor reading, we will only need to measure the sensor in dry air and fully submerged in water. For that, **we will use:**

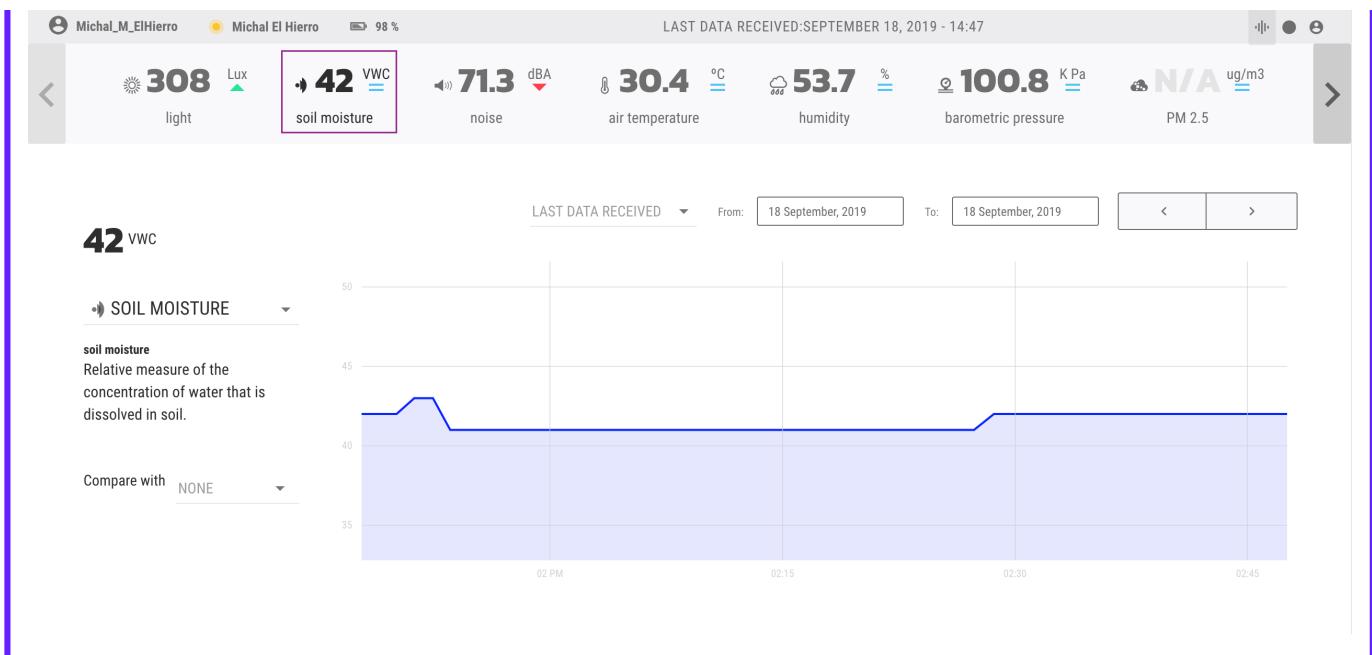
- A laptop with a serial interface. For instance, the Arduino IDE
- Our sensor
- A cup filled up with water and a napkin

The sensor can be calibrated using the shell interface. The process is as follows:

1. Connect your kit to a computer and open the terminal for the SCK. If you use the Arduino IDE, go to Tools > Serial Monitor and select 115200 baud at the bottom right corner



2. If you use the IDE type `sensor` on the top and click `Send`
3. Check if the output has something like `Soil Moisture Raw (60 sec) after Enabled`
4. If it's `Enabled`, **dry the sensor** and type in: `read soil moisture raw`. Repeat this command 5-10 times until you get an stable output (repeat command with arrow up)
5. Put the sensor in a cup of water (until the line). Then read the value again `read soil moisture raw` several times.
6. Once you have both values, type in: `control moisture cal XXX YYY` where XXX and YYY are the dry and wet values that you just measured
7. Check that the reading is OK by: `read soil moisture percent`. You should receive an answer in rh%
8. Now you should see the data online (if in network mode):



Find out more

Check the project source code files.

Sensor validation

Three Chirp sensors were compared to the Parrot Flower Power (now discontinued). The Flower Power can measure several metrics, such as light, temperature, fertilizer and soil moisture. In this test, we compared the soil moisture readings for three Flower Parrot sensors, compared to three Chirp sensors. Both sensors show a good behaviour and the values can be correlated with good R2 scores. The approach for this low-cost sensors, in general, should be more qualitative than quantitative (analyse the trends rather than the absolute values), since their values appear to differ between sensors, even when normalised. In the particular case of the Chirp sensor, the sensor seems to be fairly normalised with simply a two calibration values (water and air) as a first approach.

Full analysis here

Find the full analysis here!

10.10.2 Tensiometer

WIP

This version is a WIP but is not fully functional with the SCK 2.1. It is shown here as a showcase of the project's capabilities. Have a look at the forum or drop us an email to discuss this. Check the source files.

Soil Moisture data as the one provided by the Chirp Moisture Sensor is interesting for research, but when it comes to crops irrigation management, we usually like to know the soil water tension (SWT). That is because Soil Moisture in water is not directly related to the water plants roots might be able to extract because it is deeply affected by the soil composition. Even soil irrigation can be inferred from soil moisture when the soil type is known we think a soil tensiometer. Also when it is a simple solution, it is a useful tool for crops management.

Watermark Tensionmeter Demo

The design is entirely open source and it is deeply inspired by the work of Reinier Van der Lee from the Vinduino project, using an already calibrated commercial probe like the **Watermark 200SS9**. The sensor itself is straightforward and it consists of two stainless steel screws that work as electrodes cast inside a piece of plaster and covered by a plastic mesh to prevent erosion. As water is added more electrons can pass between the electrodes of the probe reducing the amount of resistance between them. By using this range of values, you can determine the amount of water that exists in your soil. To avoid interferences and degradation of the electrodes the design only applies voltage for a very short time and uses alternating electric polarities. For the sensor to work, we need a minimal circuit that uses two resistors and two diodes. The resistors work together with the electrodes to build a voltage divider. We can calculate the resistance value between the two electrodes by knowing the value of the resistors and the voltage. However to be able to alternate the electric current we need to duplicate the circuit and add two diodes. In total, we need 4 Pins to be connected to a microcontroller like the Arduino or the Smart Citizen Kit.

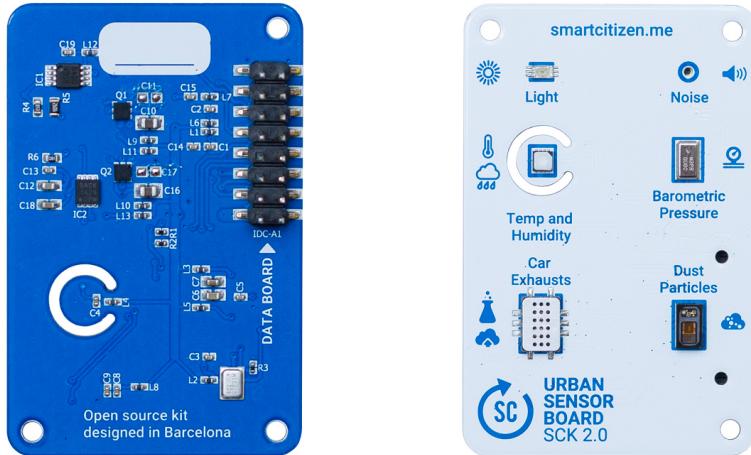
10.11 Urban Sensor Board

10.11.1 What is it?

The Urban Sensor Board is a solution that contains a selection of low-cost sensors for environmental monitoring. Its main purpose is to serve as a tool for citizen science and awareness activities, and for that reason, metrics such as temperature, pressure, and humidity, as well as noise levels, ambient light, air quality indicators and PM sensors are included. The Urban Sensor Board has undergone several modifications throughout its development, and its **current version is V2.1**:



An iteration with a different set of sensors was developed as part of the iScape Project and is shown in the image below:

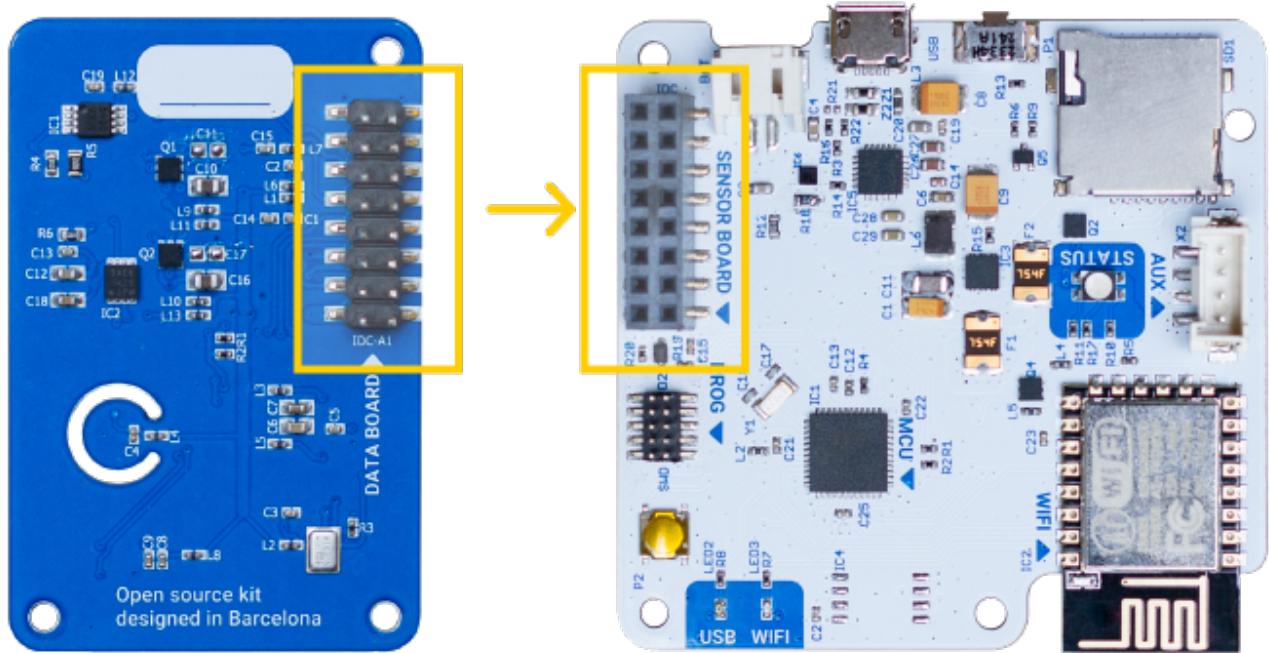


Check the source code

A major effort has been carried out on this design to improve the accuracy of the data provided. The sensors on the board include: Air Temperature, Relative Humidity, Noise Levels and Spectrum, Ambient Light and Barometric Pressure. The board also features a section especially focused on Air Quality including a Particle Matter Sensor, and, in version V2.1, an eCO₂ and TVOC sensor. Previously, in version V2.0, a Carbon Monoxide and a Nitrogen Dioxide sensors was included, but due to the high power consumption and the need of important calibration efforts, these were removed. The sensor density of the board design offers more than ten different environmental metrics at a very low cost and differentiates the design from other existing solutions. The following sections describe in detail each of the sensors available.

Board assembly

The Urban Sensor Board connect to the Data Board connector named **Sensor Board**



V2.1 Sensors

Measurement	Units	Sensor
Air Temperature	°C	Sensirion SHT-31
Relative Humidity	% REL	Sensirion SHT-31
Noise Level and Spectrum	dBA, dBC, dBZ	Invensense ICS-434342
Ambient Light	Lux	Rohm BH1721FVC
Barometric pressure and AMSL	Pa and Meters	NXP MPL3115A26
eCO2 and TVOC	ppm/ppb	AMS CCS811
Particulate Matter PM1/PM2.5/PM10	µg/m3	PMS 5003

V2.0 Sensors

Measurement	Units	Sensor
Air Temperature	°C	Sensirion SHT-31
Relative Humidity	% REL	Sensirion SHT-31
Noise Level and Spectrum	dBA, dBC, dBZ	Invensense ICS-434342
Ambient Light	Lux	Rohm BH1721FVC
Barometric pressure and AMSL	Pa and Meters	NXP MPL3115A26
Carbon Monoxide	ppm (Periodic Baseline Calibration Required)	SGX MICS-4514
Nitrogen Dioxide	ppb (Periodic Baseline Calibration Required)	SGX MICS-4514
Particulate Matter PM2.5 (external - power req)	µg/m³	PMS 5003

10.11.2 Metal Oxyde Sensors (all versions)

The metal oxyde sensors section is so extense, that we decided to dedicate a full section to them. Have a look at it here!

Read more

More on the MICS working principle and field validation

What are normal values?

More on the AMS CCS811, eCO2 and TVOC

10.11.3 Noise Level Sensor (V2.0 onwards)

The noise sensor is based on the INVENSENSE ICS-43432² high-performance, low power, digital output, omnidirectional MEMS microphone with a bottom port and I2S interface. The sensors are similar to the one found on some high-end smartphones. It delivers the information directly in a digital format to the MCU where a custom library has been developed to provide noise data in dB scales A, C and Z. The raw FFT is also accessible to support characterization of specific noise frequencies. The sensor has been calibrated specifically for the project on an anechoic chamber using standard microphone calibration procedures.

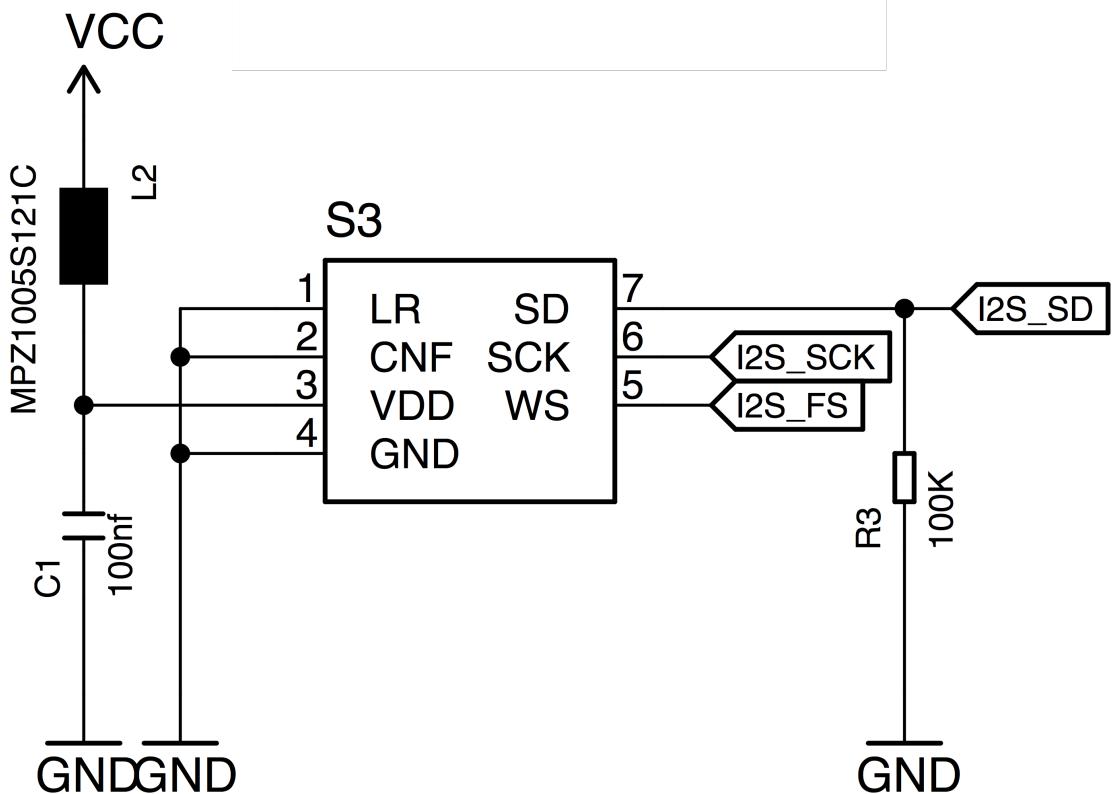
The following characteristics have been considered for the sensor choice

- High 65 dBA SNR with a -26 dB FS Sensitivity
- Low Sensitivity Tolerance ±1 dB
- Wide Frequency Response from 50Hz to 20kHz
- High Acoustic Overload Point 116 dB SPL
- Low Power

Info

Check the Noise sensor implementation full documentation

Sensor integration



10.11.4 Relative Humidity and Air Temperature Sensor (V2.0 onwards)

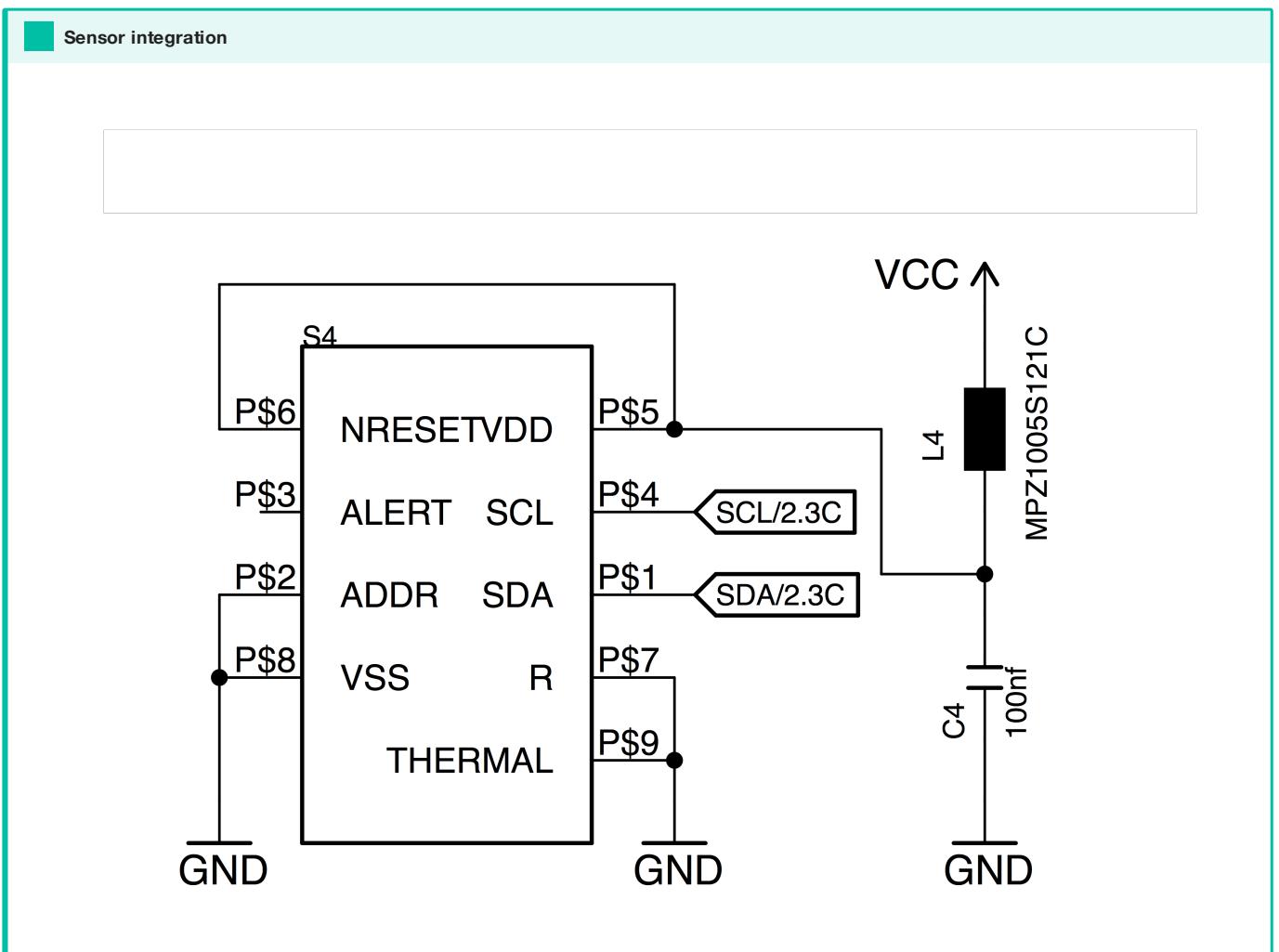
Relative Humidity and Air Temperature Sensor are provided by a SENSIRION SHT31³ module.

Sensor upgrade

Preliminary tests during the project shown a absolute calibration issues affecting the previously selected sensor, the SENSIRION SHT31. Those we updated the sensor to the newest SHT 31 with a PTFE layer for protection obtaining better results.

The following characteristics have been considered for the sensor choice

- Calibrated, linearized sensor signals in digital, I2C format straight to the MCU where data is provided in degrees Celsius and Relative Humidity.
- Wide measurement range with high resolution. The relative humidity range of 0-100% RH with a 0.03% resolution and a repeatability of 0.1%, together with a temperature operating range from -40 to +125°C with a temperature resolution of 0.01 °C and a repeatability of 0.1%.
- No need for calibration and long-term stability.
- Low power consumption
- Commonly found in many commercial weather stations as the Davis Vantage Pro.

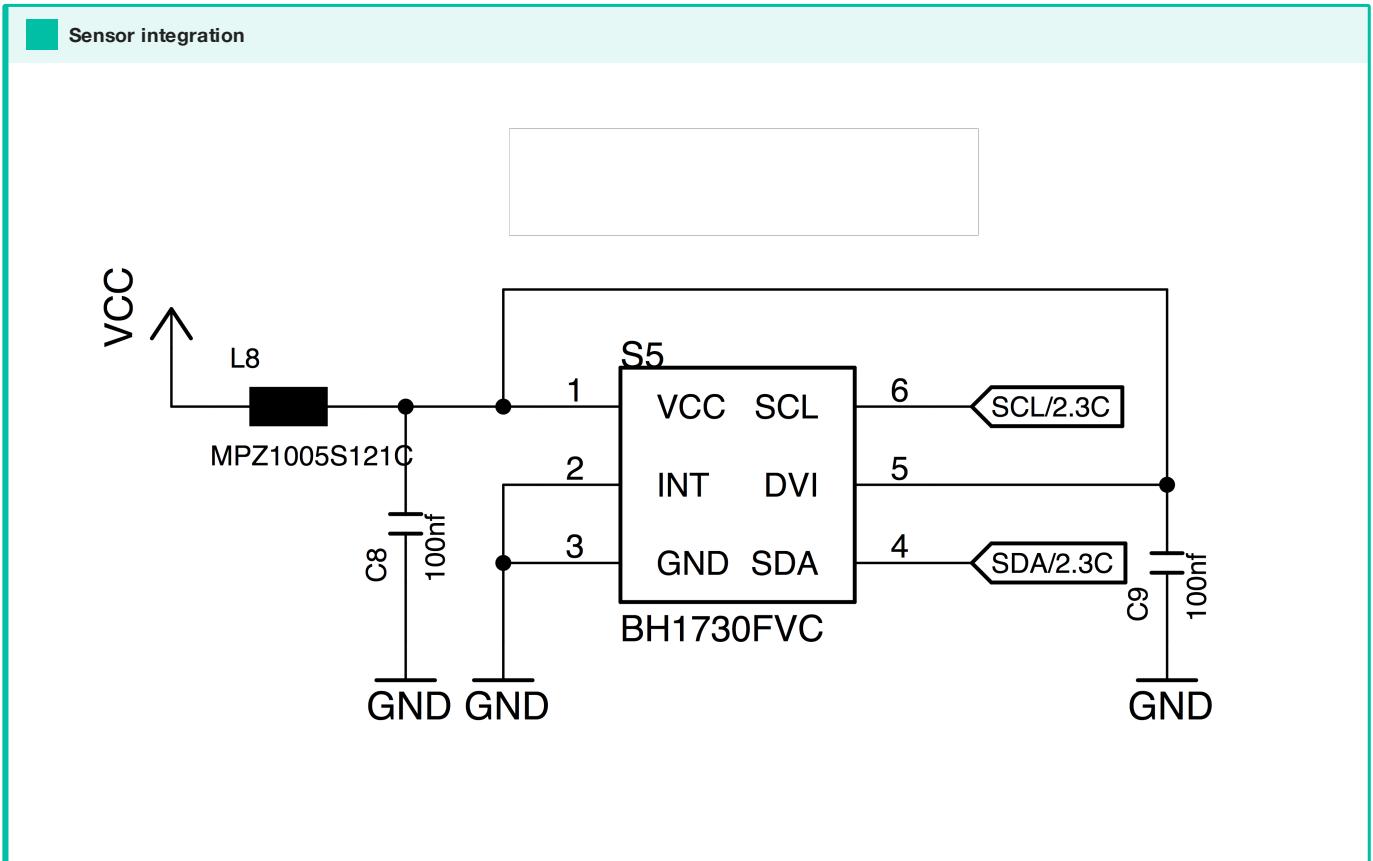


10.11.5 Ambient Light Sensor (V1.5 onwards)

The Ambient Light Sensors is based around the ROHM BH1721FVC⁴ which uses an LDR10 combined with an ADC and the corresponding circuit that allows communicating with the device with the I2C protocol.

The following characteristics have been considered for the sensor choice:

- No need of external ADC or linearization circuits uses the well-known I2C protocol
- Measures ambient light data in a wide range from 1lx to 65528 lx a repeatability of 15% and a resolution of 8 lx.
- Possibility to adjust by an I2C command the kind of light that it should measure (visible or infrared).
- Low power consumption.
- 50Hz/60Hz (electric network frequency) light rejection. Filtering the interference of most artificial light sources.



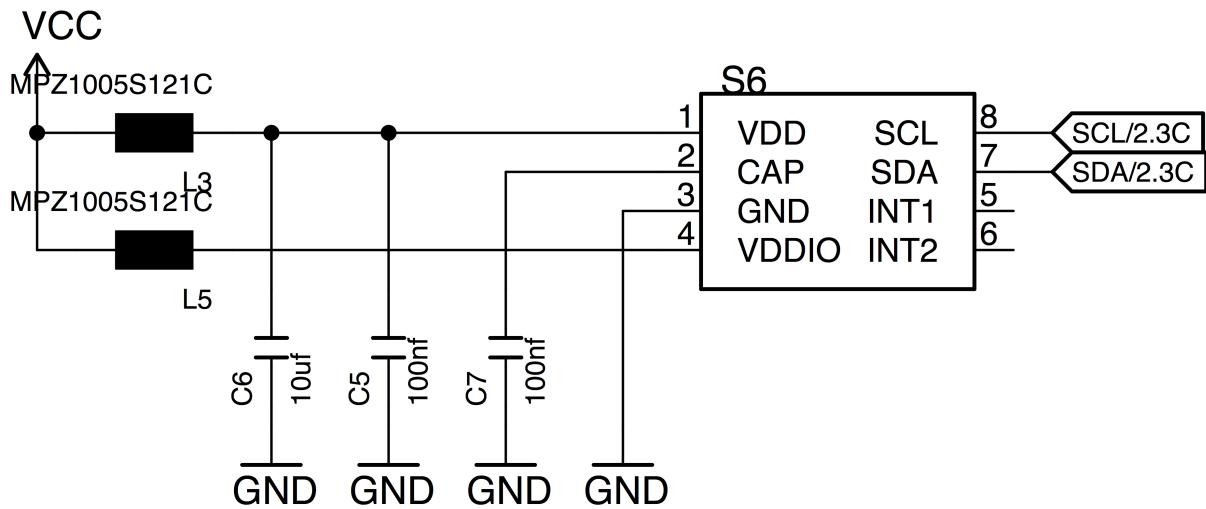
10.11.6 Barometric Pressure (V2.0 onwards)

The Barometric Pressure sensor is based around the NXP MPL3115A2⁵ is a compact, piezoresistive, absolute pressure sensor with an I2C digital interface.

The following characteristics have been considered for the sensor choice:

- Wide operating range of 20 kPa to 110 kPa.
- Temperature compensated utilizing an on-chip temperature sensor.
- No need for an external ADC or linearization circuits. The pressure and temperature data is fed into an internal high-resolution ADC to provide fully compensated and digitized outputs for pressure in Pascals and temperature in °C using the well-known I2C protocol
- Barometric pressure is also processed by the MCU as height above mean sea level (AMSL) helping to determine the location of the device.
- Low power consumption.

Sensor integration

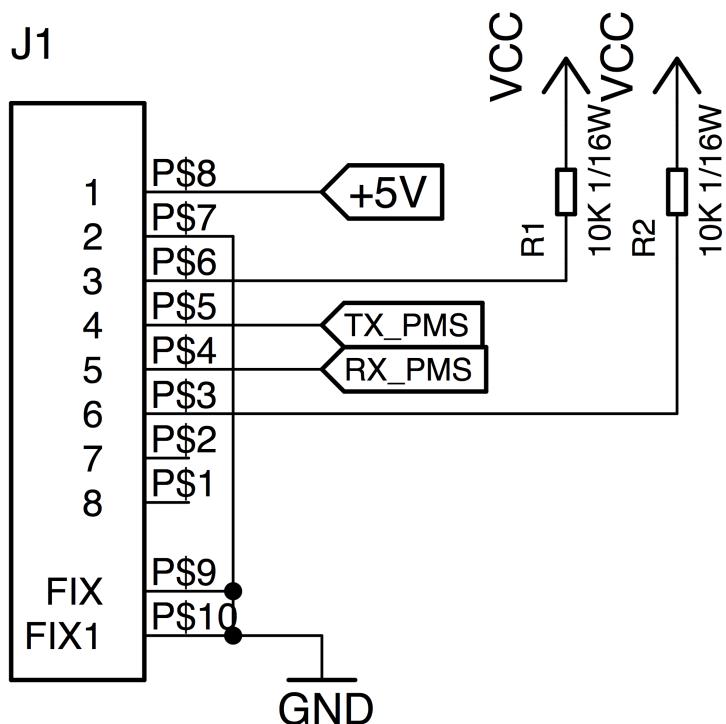
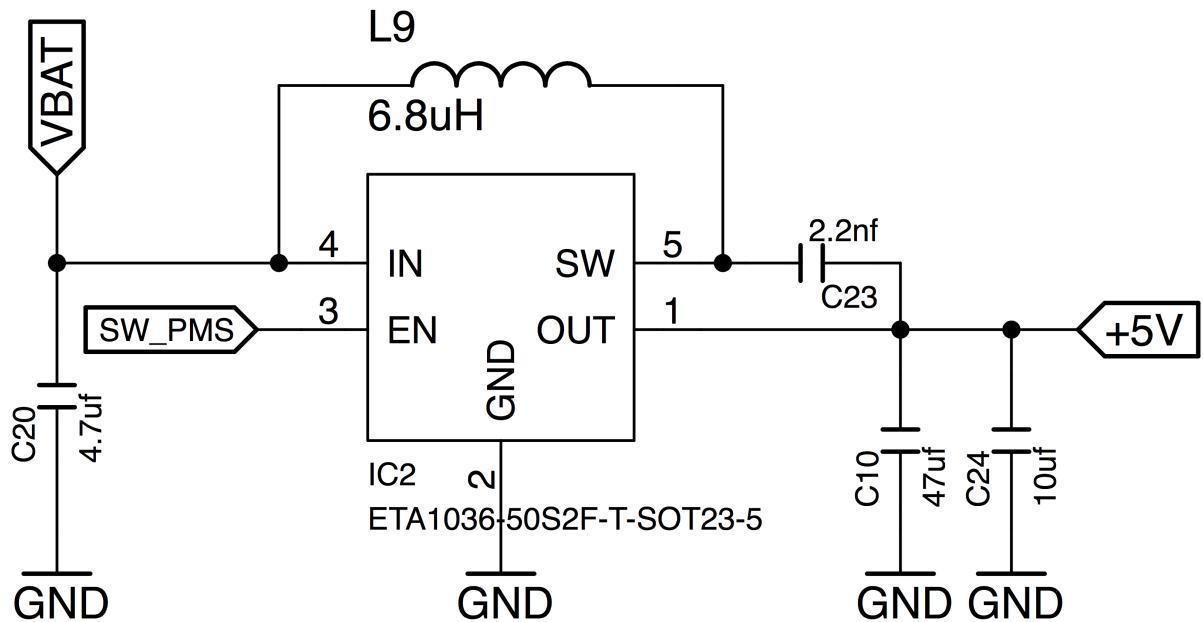


10.11.7 External PM Sensor (V2.0 onwards)

An external connector on the board supports the connection of a Plantower PMS 5003 or PMS 7003⁷. The device is a digital particle concentration sensor that uses the Laser Scattering principle to obtain the number of suspended particles in the air. The sensor can be fully enabled or disabled in software to save energy when not in use.

The following characteristics have been considered for the sensor choice:

- Provides PM 2.5 and PM 10 measurements in $\mu\text{g}/\text{m}^3$
- Minimal distinguishable particle diameter of 0.3 μm
- No need for external ADC or linearization circuits. The sensor includes an internal MCU capable of dealing with all the light emitting and sensing processing. All the communication is done using the I2C protocol. A dedicated driver has been designed for this.
- Ultra Low Cost when compared to other commercial solutions with similar performance
- Low Power

 Sensor integration


10.11.8 Source files

Download

Check the source code

1. SGX MICS 4514 Technical Datasheet

https://sgx.cdistore.com/datasheets/sgx/0278_Datasheet%20MiCS-4514%20rev%2017.pdf

2. INVENSENSE 43432 Technical Datasheet

<https://www.invensense.com/wp-content/uploads/2015/02/ICS-43432-data-sheet-v1.3.pdf>

3. SENSIRION SHT31 Technical Datasheet

https://www.sensirion.com/fileadmin/user_upload/customers/sensirion/Dokumente/2_Humidity_Sensors/

4. ROHM BH1730 Technical Datasheet

<http://rohmfs.rohm.com/en/products/databook/datasheet/ic/sensor/light/bh1721fc-e.pdf>

5. NXP MPL3115A2 Technical Datasheet

<http://www.nxp.com/docs/en/data-sheet/MPL3115A2.pdf>

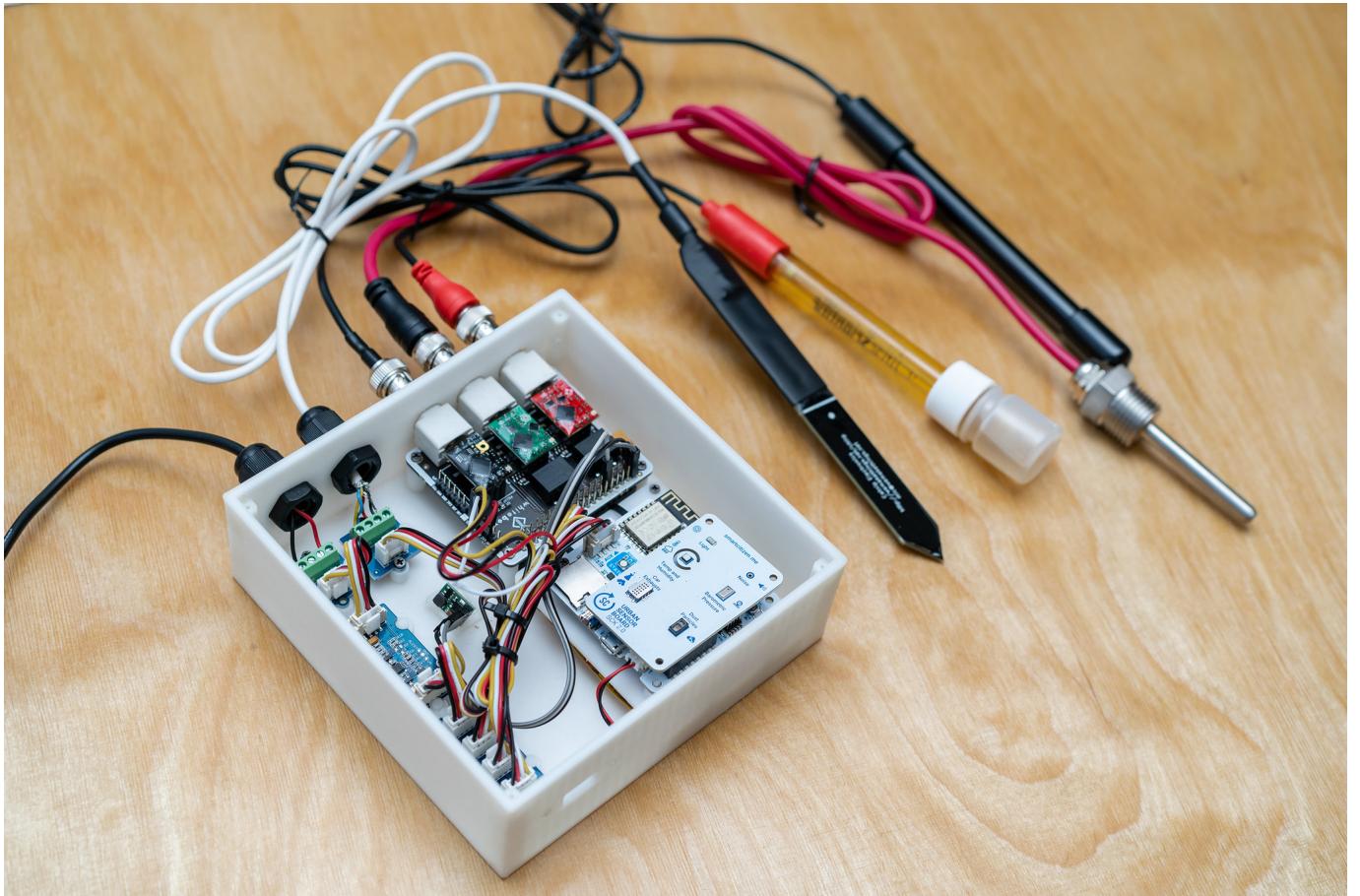
6. MAXIM 30105 Technical Datasheet

<https://datasheets.maximintegrated.com/en/ds/MAX30105.pdf>

7. PLANTOWER PMS5003 Technical Datasheet <https://aqicn.org/air/view/sensor/spec/pms5003.pdf>

10.12 Water

Having a robust portfolio of the sensor for measuring soil and water characteristics is a need found by many research communities. In this direction, we include a collection of sensors that despite not being low cost or open source, they are still affordable and well documented when compared to other commercial solution. From a cost perspective, they are not aimed at being massively deployed but instead used individually in a specific site for specific needs.

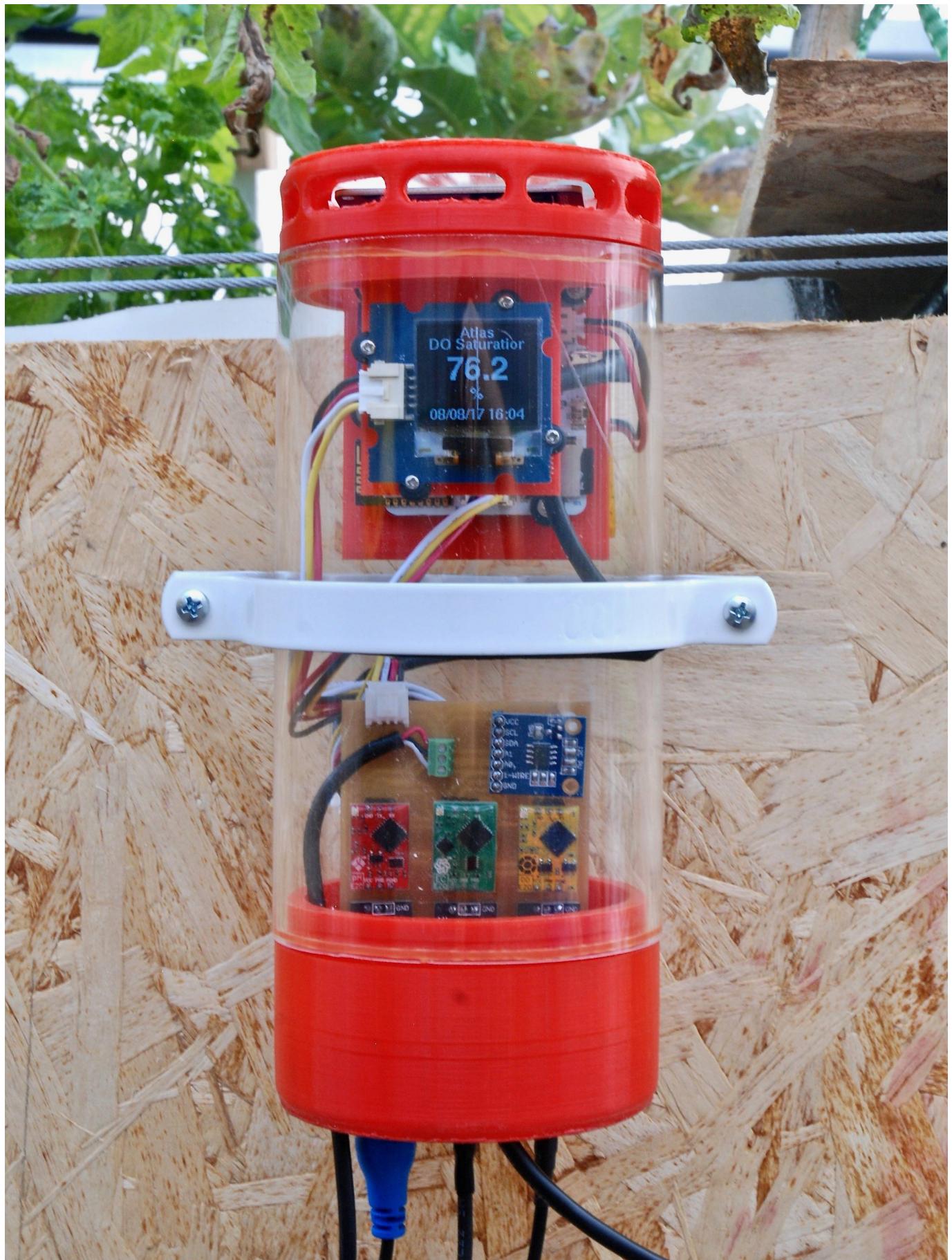


The sensors selected are from Atlas Scientific, a New York-based company that converts devices that were originally designed to be used by humans into devices that are specifically designed to be used by robots. As already mentioned the sensors are not entirely open source as the other sensors documented on this section. However, they are modular and exceptionally well documented by the manufacturer. That includes documentation on how to install, calibrate and integrate them with additional existing hardware. In this direction, we developed a full library for the SCK to support the sensors via the Auxiliary sensor connector. We also developed a Python script to simplify the calibration process of the sensors. As the sensors can be configured in different ways, we do not provide a full step-by-step guide. Instead, we refer to the documentation on the project's repository.

The setup is built out of the following main components:

- Atlas Scientific Sensor Probe: The physical probe we will insert on to the soil (or water).
- Atlas Scientific EZO Circuit: The driver that will read the analog signal coming from the Sensor Probe and turn it into a meaningful numeric value by applying the different calibration operations.
- Whitebox Labs Tentacle T3: The motherboard that puts everything together and hosts up to 3 Atlas Scientific Probes. It connects to the SCK via the Aux sensor connector. This boards can be chained to support more sensors, but this is not documented at the moment.
- SEEED Grove - 4 pin Female Jumper to Grove 4 pin Conversion
- Cable needs to be used to connect the board to the SCK.

Different sensor probes can be selected for different needs. For example the setup shown above is designed for soil measurements and includes Atlas Scientific temperature, conductivity and PH probes. It also consists of a Chirp Moisture Sensor as described in the above section. As an additional example the setup in the figure below is designed for water monitoring on aquaponics systems and includes Atlas Scientific probes for PH, conductivity and dissolved oxygen.

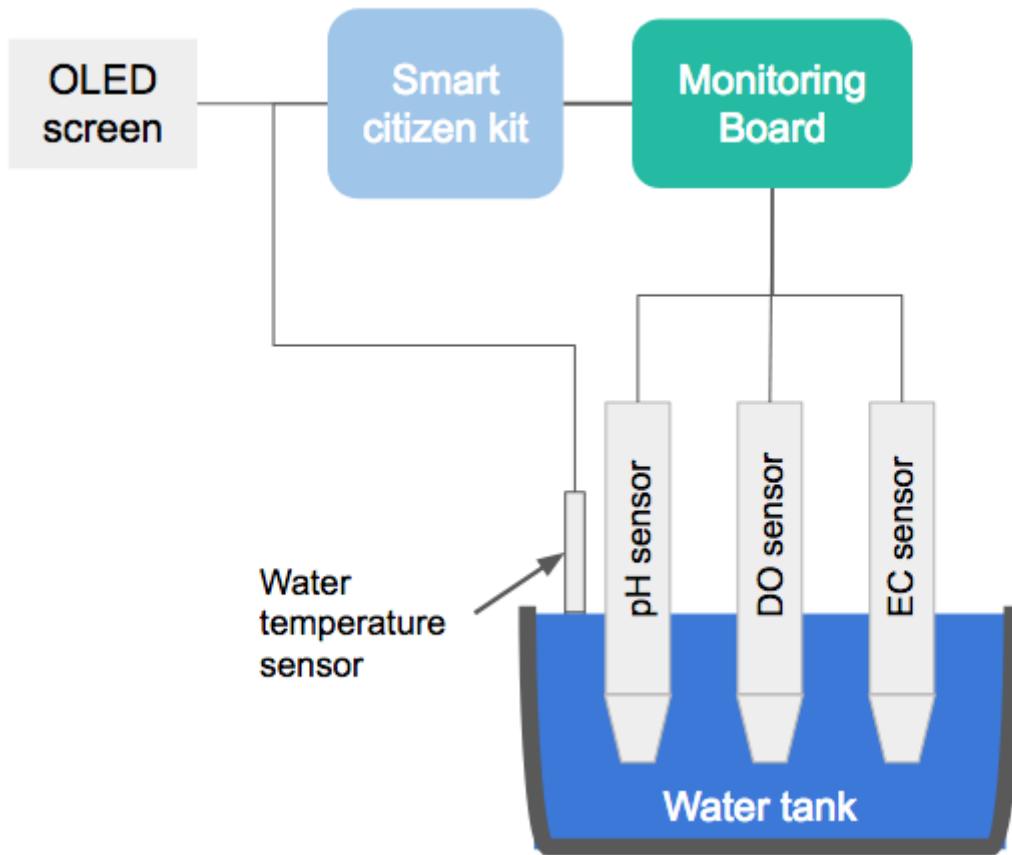




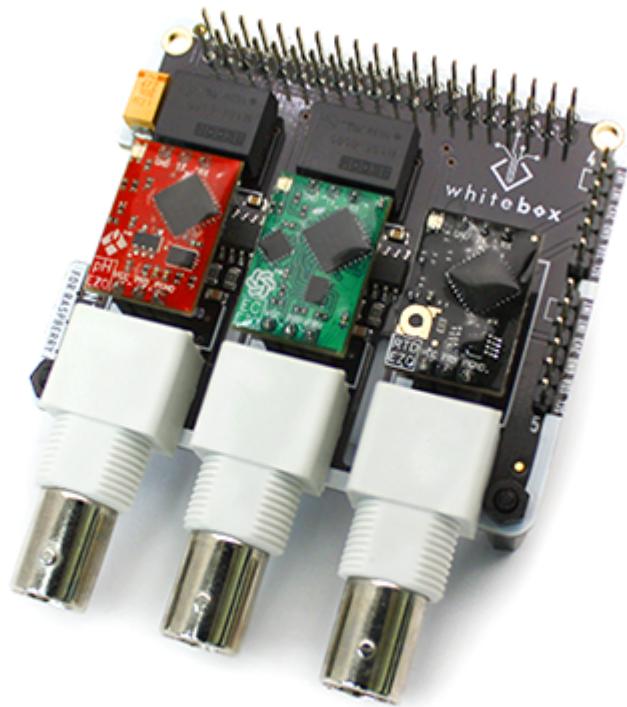
10.12.1 Metrics and Sensors

ID	Name	Description	Unit	Measurement	Description
10	Battery	Custom Circuit	%	battery	The SCK remaining battery level in percentage.
13	HPP828E031	Humidity	%	humidity	Relative humidity is a measure of the amount of moisture in the air relative to the total amount of moisture the air can hold. For instance
12	HPP828E031	Temperature	°C	air temperature	Air temperature is a measure of how hot or cold the air is. It is the most commonly measured weather parameter. Air temperature is dependent on the amount and strength of the sunlight hitting the earth
14	BH1730FVC	Digital Ambient Light Sensor	Lux	light	Lux is a measure of how much light is spread over a given area. A full moon clear night is around 1 lux
15	MiCS-4514	NO2	kOhm/ ppm	no2	Nitrogen dioxide is a toxic gas. It is a pollutant in some urban areas due the excess air required for complete combustion of fuels introduces nitrogen into the combustion of internal combustion engines (cars)
16	MiCS-4514	CO	kOhm/ ppm	co	Carbon monoxide is a colorless
29	MEMS Mic	MEMS microphone with envelope follower sound pressure sensor (noise)	dBC	noise	dB's measure sound pressure difference between the average local pressure and the pressure in the sound wave. A quiet library is below 40dB
42	DS18B20	Submergible Water Temperature sensor	°C	water temperature	Water temperature is a measure of how hot or cold the water is.
46	AS EZO Specific Gravity	Atlas Scientific EZO™ Specific Gravity	SG	specific gravity	The density of a substance to the density of a reference substance.
49	AS EZO Oxygen Saturation	Atlas Scientific EZO™ Oxygen Saturation	%	oxygen saturation	Relative measure of the concentration of oxygen that is dissolved in Water
43	AS EZO PH	Atlas Scientific EZO™ pH	PH	pH	pH is a numeric scale used to specify the acidity or basicity of an aqueous solution.
48	AS EZO Dissolved Oxygen	Atlas Scientific EZO™ Dissolved Oxygen	mg/L	dissolved oxygen	Absolute measure of the concentration of oxygen that is dissolved in Water
45	AS EZO Electrical Conductivity	Atlas Scientific EZO™ Electrical Conductivity	µS/cm	electrical conductivity	How strongly a given material opposes the flow of electric current.

10.12.2 Hardware

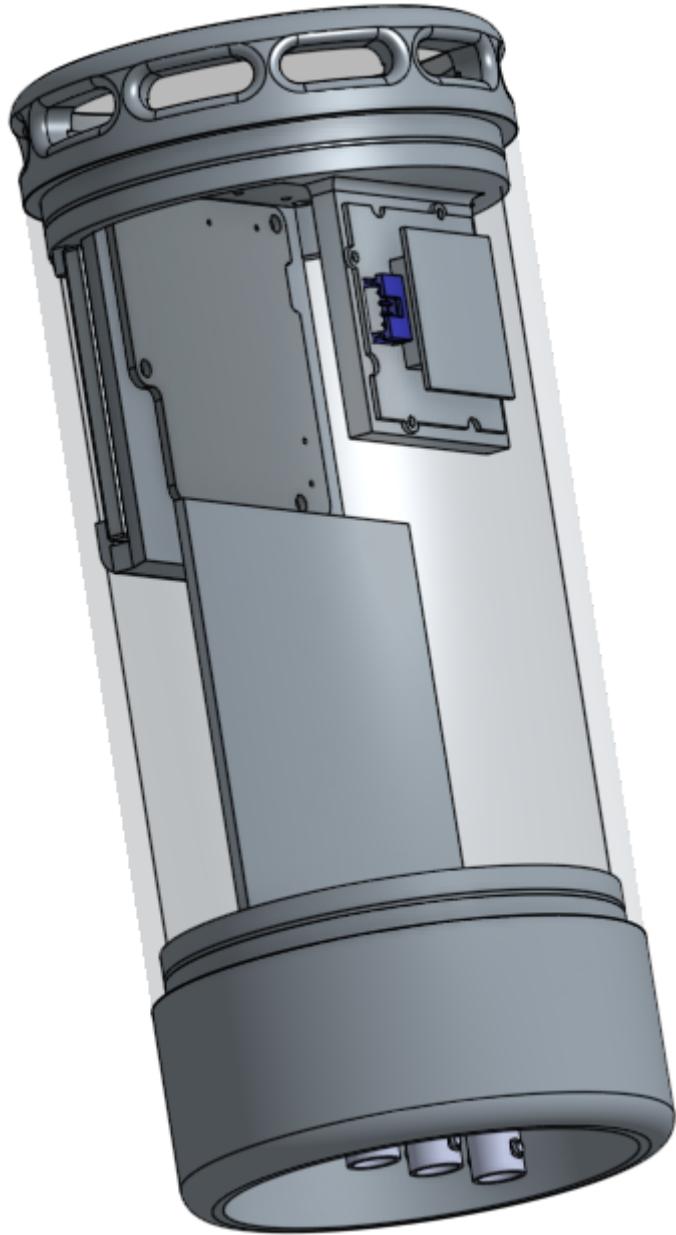
**Atlas Scientific Carrier board board**

We recommend using Whitebox Labs Tentacle T3 that hosts up to 3 Atlas Scientific Probes. It connects to the SCK via the Aux sensor connector. This boards can be chained to support more sensors, but this is not documented at the moment. However, before it existed we designed a custom board in collaboration in with Aquapioners.

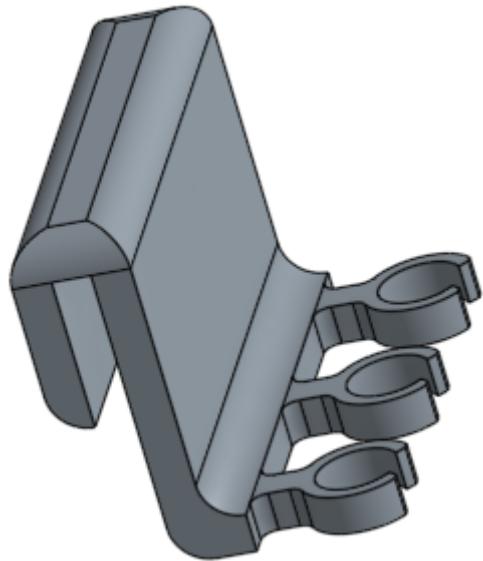


Enclosures

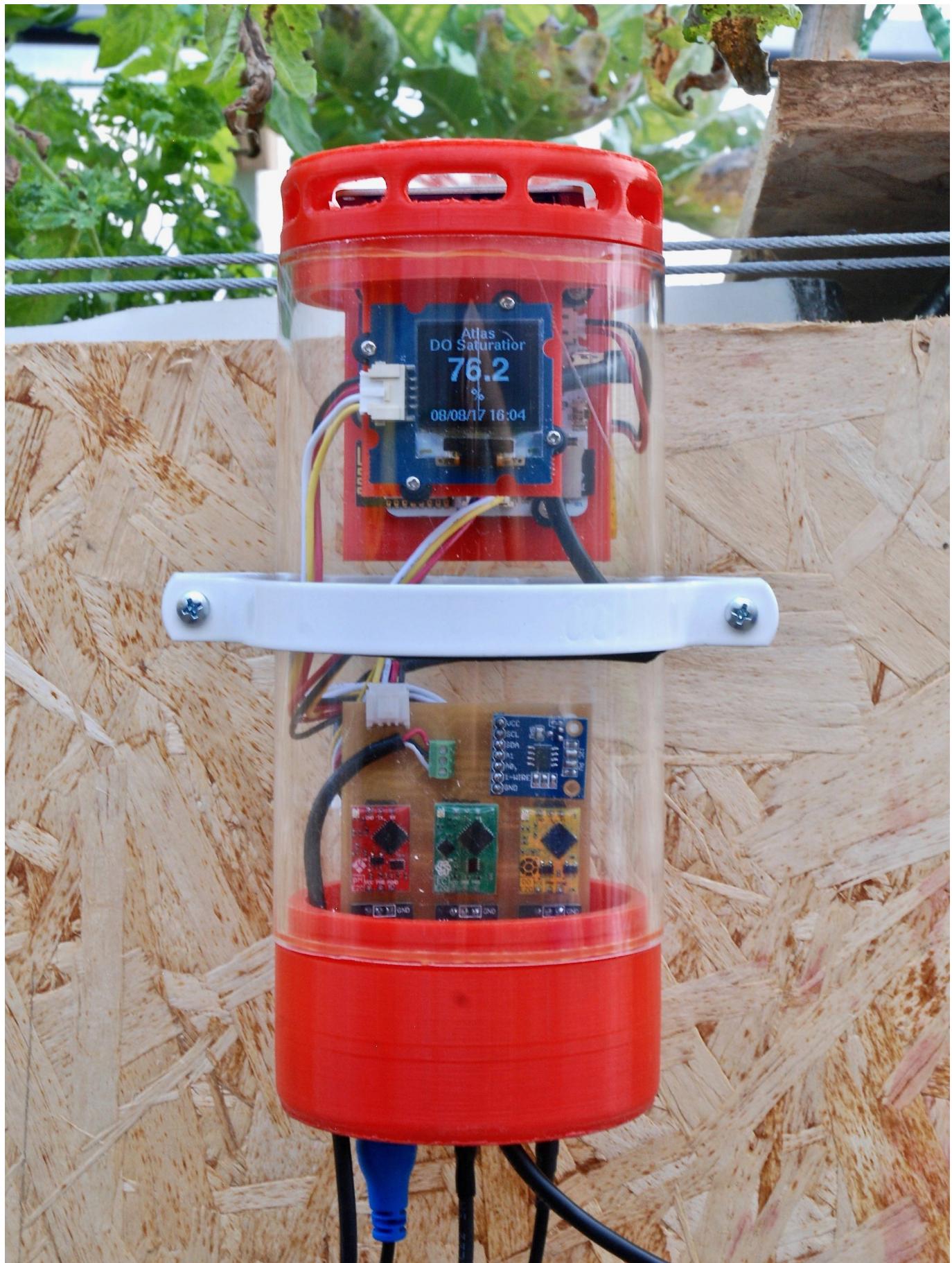
There is not an standard enclosure and in many cases a simple IP65 box could work. However, in collaboration in with Aquapioners we designed the custom enclosure below.



The enclosure of the monitoring board and the smart citizen have been designed on Onshape, you can either download the STL files or copy the project to your onshape account and modify them as you wish : The Onshape documents of the monitoring case



We have also designed a probe holder if you want to hold your probes on the side of you fish tank : The Onshape document of the probes holder

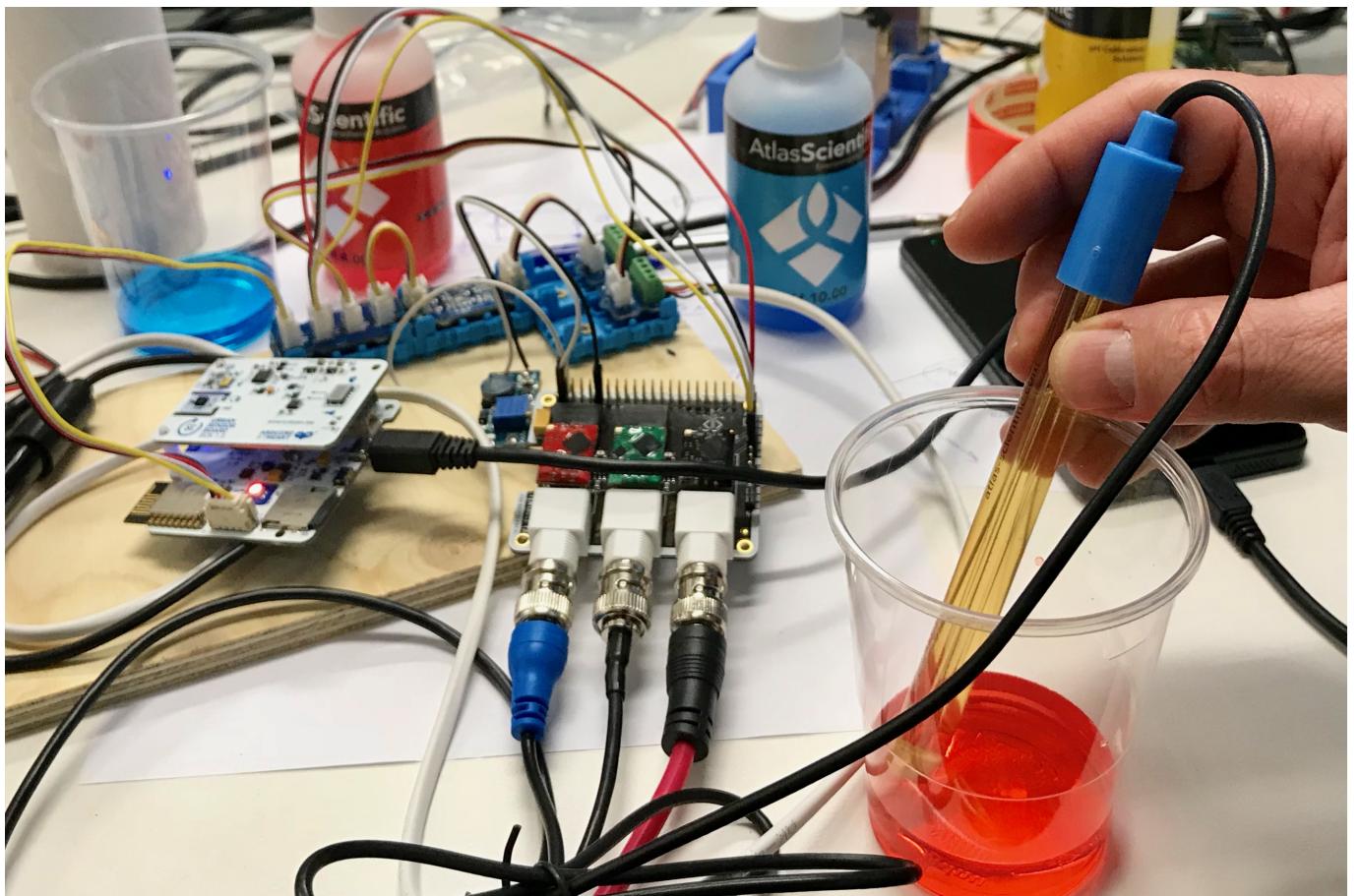


10.12.3 Firmware and setup

The Smart Citizen firmware has the support for the monitoring board built-in. To enable the sensors you just need to plug your board to the Smart Citizen kit aux port and reboot the Smart Citizen Kit and the sensors will be handled by the board. However, you will need to register your device again using the Advanced Kit Selection. At the moment the closest Kit Blueprint will be #22 BioPV Kit . You can request in the forum for a custom blueprint with the specific sensors you are using.

10.12.4 How to calibrate the sensors

In order to calibrate the sensors you will need to use the USB Shell.



The pH sensor

format of the command line

```
control atlas ph com,[point],[pH value at current temperature]
```

The pH value at current temperature can be found on the reference table on the calibration solution bottle. If the current temperature is not on it, use the closest value.

THE 3 POINTS CALIBRATION

First start a serial communication with the Smart Citizen Kit with screen or pio device monitor or even the serial monitor of the Arduino IDE.

Order of the calibration :

1. mid point
2. low point
3. high point

important ! : Always calibrate the mid point first because it calibration erase all the previous calibration done.

Always clean the probe with distilled water between each calibration

- **The mid point calibration :** Put the sensor in the pH 7 calibration solution and run the command below :

```
control atlas ph com cal,mid,[value of pH at current temperature]
```

example at 30°C :

```
control atlas ph com cal,mid,6.99
```

- **The low point calibration :** Put the sensor in the pH 4 calibration solution and run the command below :

```
control atlas ph com cal,low,[value of pH at current temperature]
```

- **The high point calibration :** Put the sensor in the pH 10 calibration solution and run the command below :

```
control atlas ph com cal,high,[value of pH at current temperature]
```

Note : (not tested) If your calibration solutions are not 4, 7 and 10, you can still use them and replace [value of pH at current temperature] by your values.

The EC Sensor

THE 2 POINTS CALIBRATION

Order of the calibration :

1. dry point
2. high point

3. **The dry point calibration :** Check that the sensor is dry and run the command below :

```
control atlas ec com cal,dry
```

- **The high point calibration :** Put the sensor in the high point calibration solution (12,880 µS/cm) and run the command below :

```
control atlas ec com cal,high,[value of EC at current temperature]
```

example at 30°C :

```
control atlas ec set cal,high,14,120
```

Important ! : Do not forget the , between the hundreds and the thousands or else the calibration will not occur !

Note : (not tested) If your calibration solution is not 12880 µS/cm, you can use another one and replace [value of pH at current temperature] by your value of electroconducivity.

10.12.5 The DO Sensor

The 2 points calibration

Order of the calibration

1. dry point
2. 0 mg/L point (optional)

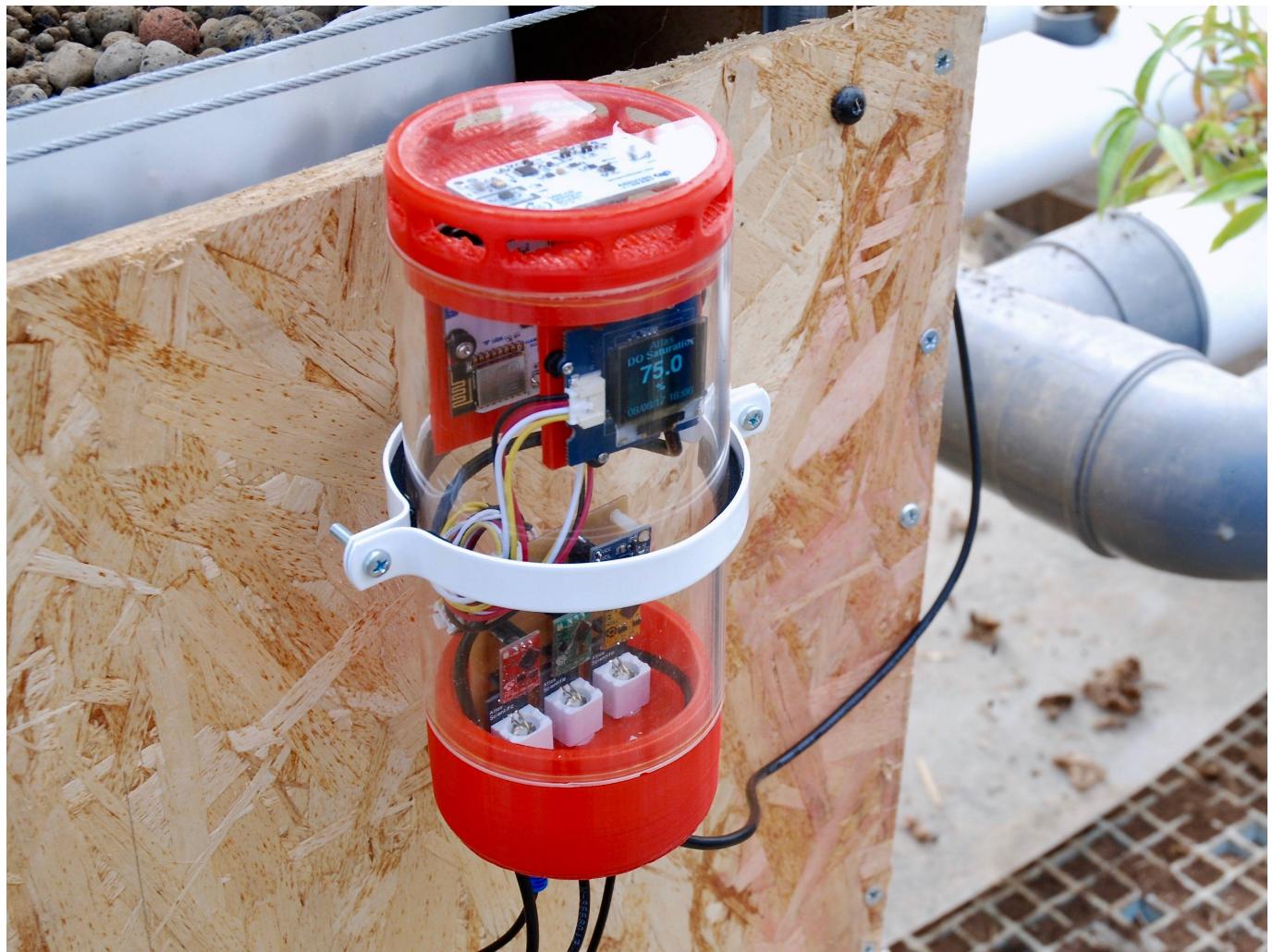
3. The dry point calibration : Check that the sensor is dry and run the command below :

```
control atlas do com cal
```

• **The 0mg/L point calibration :** Put the sensor in the 0mg/L calibration solution and run the command below :

```
control atlas do com cal,0
```

10.12.6 Deployment





11. Sensors

11.0.1 eCO₂ and TVOC sensor

The AMS CCS811 is a Metal Oxide Sensor with I₂C connectivity which is capable of measuring a **volatile organic compounds** (VOCs for short). This sensor was introduced in the SCK2.1, in replacement of the SGX MICS4514 from previous designs. As mentioned in other parts of the documentation, this decision was mainly due to the **lower power consumption of the CCS811** and the easy implementation of reading processing provided by the manufacturer.

Does it measure CO₂?

No. Despite the name, the sensor does not measure CO₂. See below for a detailed description of the sensor measurements.

Measurements

eCO₂ and **eTVOC** are two related measurements. The first stands for *equivalent CO₂*, and it's an indication of the concentration of CO₂ that would cause the same level of radiative forcing as a given type and concentration of greenhouse gas. The eCO₂ measurement is therefore a derived measurement from the reactions all these substances in the air with the metal oxide substrate in the sensor. eCO₂ bottomline starts at 400ppm (current background CO₂, sadly) and can reach several thousands.

On the other hand, eTVOC stands for *equivalent total volatile organic compounds* and is a measurement of the total amount of any emitted gases coming from toxins and chemicals. They come from a wide range of everyday items including paints and varnishes, wax and cosmetics, cleaning and hobby products, and even cooking. When you have an enclosed space like a home or office, these emitted gases accumulate and pollute our fresh air.

GLOBAL WARMING POTENTIAL

To understand the **eCO₂ readings**, we need to know what is the **Global Warming Potential** (GWP for short). GWP is an estimate of how much a given greenhouse gas contributes to Earth's radiative forcing.

We know that CO₂ is one of the major contributors to global warming, but there are others (and that are much worse). For example: CO₂ has a value of 1 GWP, whereas methane has a GWP of 72 over 20 years, but a lower GWP of 25 over 100 years. This is because it is very potent in the short-term but then breaks down to CO₂ and water in the atmosphere, meaning that the longer the period you consider it over, the more similar its effect is to that of CO₂ alone. This means that 1ppm of CH₄ is much more worrying for the global warming of the planet than 1ppm CO₂ in the short term, because it can produce a higher increment of the atmosphere temperature. All this is at the same time interesting and worrying, because many products used for painting, solvents, varnish, refrigeration, and more contain pollutants with high GWP. A very interesting article can be found here.

Working principle

As any metal oxide sensor, the CCS811 measures the resistance of a sensitive layer, exposed to ambient air. This layer is heated up with a *heater element* (a resistance) up to several hundred °C, and some oxidation reactions take place on it. The characteristics of the sensitive element vary from sensor to sensor, and with time, depending as well on the exposure to different chemical components and ambient conditions. For this reason an individual sensor characterisation is *very tricky*, and relative measurements are used, using an internal processing that monitors the baseline resistance of the sensor (i.e., the resistance of the sensitive layer when exposed to clean air).

The sensor generally targets pollutants that can get oxidised in the sensor substrate. This oxidation process modifies the resistance of the sensor, and the more oxidation reactions we have the lower the resistance is. The concept of baseline resistance in this sense can be confusing, but basically can be explained as: the higher resistance, the cleaner the environment.

Temperature and humidity are used internally to compensate the readings, as the sensor compares it's actual resistance with the *clean air one* (baseline), and inputs the ambient conditions in the correction.

Sensor considerations

Like any sensor, the CCS811 has some limitations. As mentioned above, the sensitive layer will decrease it's resistance when in presence of VOCs, but other pollutants can have the opposite effect on it. For instance, Ozone (O₃) will increase the sensor's resistance, and it could be seen as clean air by the sensor. This could explain why in some outdoor environments (generally with traffic and high levels of sun radiation), the sensor can present an unstable behaviour.

Additionally, humidity is known to affect the sensor resistance. The internal humidity correction can limit this effect up to a certain extent, but a perfect correction is not possible.

We also recommend setting the sensor in a stable environment, in which temperature and relative humidity changes are not abrupt. When moving the sensor to another location, beware that any high resistance could be seen as *the new baseline resistance*, and this value might not apply to the previous environment, should you put the sensor back into it's original location.

Resetting the baseline resistance

The sensor keeps track of the baseline resistance even after a factory reset. Currently, there is no method to reset the baseline resistance in the firmware, but it will soonly be introduced. When changing locations, a baseline resistance reset should be performed.

Finally, although the sensor is considered to be an indoor sensor, *it can be placed outdoor*, but keeping in mind, that the environment will be very different and that the sensor might behave in unexpected ways.

Sources

Have a read to the Datasheet Or some other Application notes

EARLY-LIFE (BURN-IN)

The CCS811 performance in terms of resistance levels and sensitivities will change during early life. The change in resistance is greatest over the first 48 hours of operation, although this process can last up to 5 days. CCS811 controls the burn-in period allowing eCO₂ and eTVOC readings to be used from first power-on after 60 minutes of operation.

CONDITIONING PERIOD (RUN-IN)

After early-life (Burn-In) the conditioning or run-in period is the time required to achieve good sensor stability before measuring VOCs after long idle period. The sensor will need to run for 20 minutes, before accurate readings are generated.

11.1 Inside the Electrochemical Sensors

11.1.1 Sensor working principle

The electrochemical cells used are toxic gas sensors from alphasense that operate in an amperometric mode. That is, they generate a current that is linearly proportional to the fractional volume of the toxic gas in the environment:

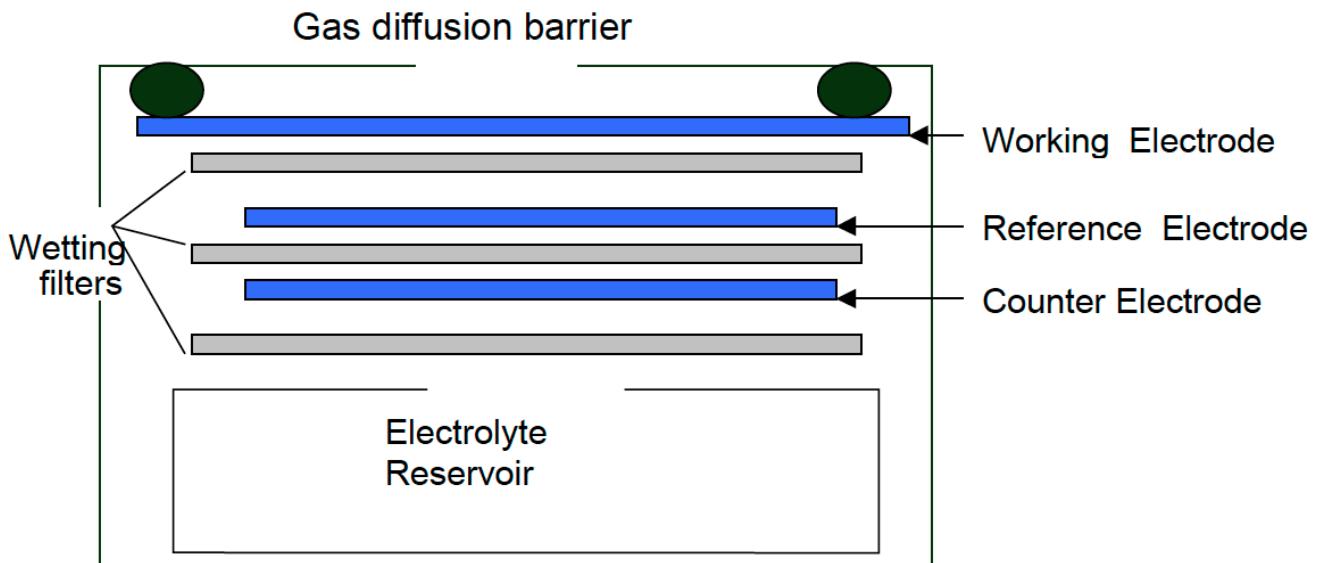


Image Source: Alphasense Ltd.

These electrochemical sensors are comprised of four electrodes:

- Working electrode
- Auxiliary electrode
- Counter electrode
- Reference electrode

The **working electrode** is where the oxidation (CO, H₂S, NO, SO₂) or reduction (NO₂, Cl₂) of the toxic gas to be measured takes place. This electrode is exposed to the outside air and directly exposed to all gases in the air including the gas to be measured. This electrode may as well be **poisoned** if it is exposed to certain gases that either adsorb onto the catalyst (such as acetylene onto CO sensors), or react, creating by-products which inhibit the catalyst (NO₂ or aromatics onto H₂S sensors).

The **auxiliary electrode** is an electrode of the same characteristics to those of the working electrode, but it is buried inside an electrolyte and, hence, it is not in contact with the target gas. Since it is isolated from external conditions that could affect the **working electrode**, it serves as a reference to the measurements provided by the latter.

The **counter electrode** balances the reaction of the working electrode – if the working electrode oxidises the gas, then the counter electrode must reduce some other molecule to generate an equivalent current, in the opposite sense. For example, where carbon monoxide will be oxidised on the working electrode, oxygen will be reduced on the counter electrode.

The **reference electrode** anchors the working electrode potential to ensure that it is always working in the right conditions. It is important that the reference electrode has a stable potential, keeping the working electrode at the right electrochemical potential to maintain a constant sensitivity, good linearity and minimum sensitivity to interfering gases.

Therefore, while the sensor response is exposed to the target gas, it creates a current flowing from the working to the counter electrode or viceversa (depending on the oxidative or reductive nature of the target gas). This relationship can be characterised and follows a curve such as:

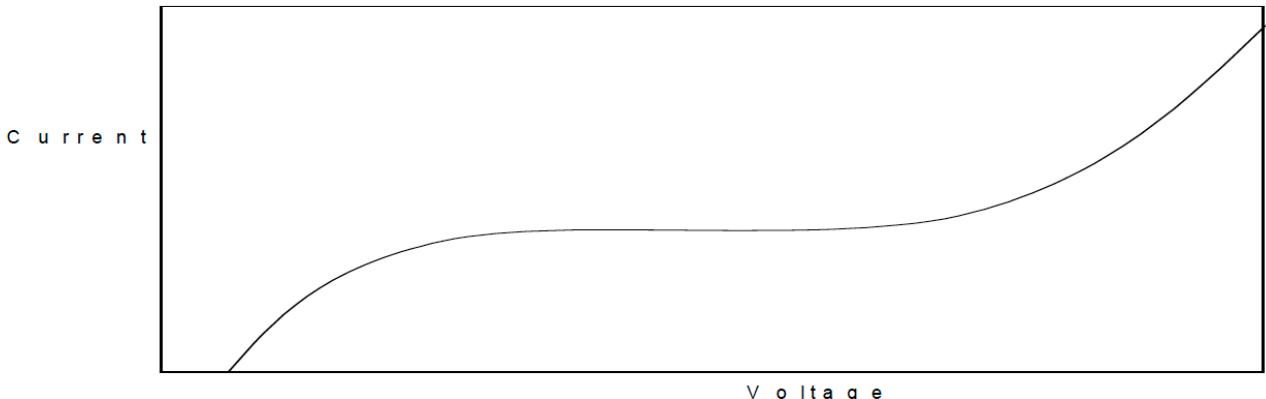


Image source: Alphasense Ltd.

When operating in the so called *transport limited current plateau* the measured current (I_L) should be linearly dependent on the concentration or fractional volume of the toxic gas (CT) in the external environment:

$$I_L = k C_T$$

where k is a proportionality constant. This constant is provided by the manufacturer as Sensitivity and is explained below.

Electronics design considerations

A potentiostat circuit is built in order to ensure that the counter electrode is provided with as much current as it needs, also maintaining the working electrode at a fixed potential, irrespective of how hard it is working.

Manufacturer data

The manufacturer provides the calibration data in laboratory conditions for each of the electrochemical cells used. This data is listed below: - **Sensor sensitivity**: the sensor response in nA per each ppm of target pollutant in *nominal* conditions - **Electrode zero current**: the electrode reading in nA to zero air (pure air at 25degC). This is provided for both, working and auxiliary electrodes, in the case of 4-electrode sensors - **Sensor response** (t_{90}) - **Sensor range**

The manufacturer suggests using the following equation in order to determine the sensor's corrected reading in the presence of target gas:

$$\text{Concentration [ppm]} = \frac{(I_{WE} - n(I_{AE}))}{\text{Sensitivity [nA/ ppm]}}$$

Where:

$$I_{WE} (\text{nA}) = K (\text{nA/mV}) V_{WE} (\text{mV})$$

$$I_{AE} (\text{nA}) = K (\text{nA/mV}) V_{AE} (\text{mV})$$

Where: * I_{PCBWE} and I_{PCBAE} are the electronic offsets for each electrode * $n = \{I_{0WE}\} / \{I_{0AE}\}$, the ratio between alphasense's zero currents * k is a **constant conversion factor** (~ 6.36 in the case of the SCK Gas Pro Board electronics)

With regards to **sensor ranges**, the following are available from the manufacturer:

- **NO₂**: 20ppm
- **O₃ + NO₂**: 20ppm both
- **CO**: 1000ppm

Finally, toxic gas sensors' sensitivity will **drift downwards with time, typically 0.5% to 2% per month**, depending on the sensor type, relative humidity and gas concentration/temperature conditions.

Reduction vs Oxidation Electrochemical Sensor

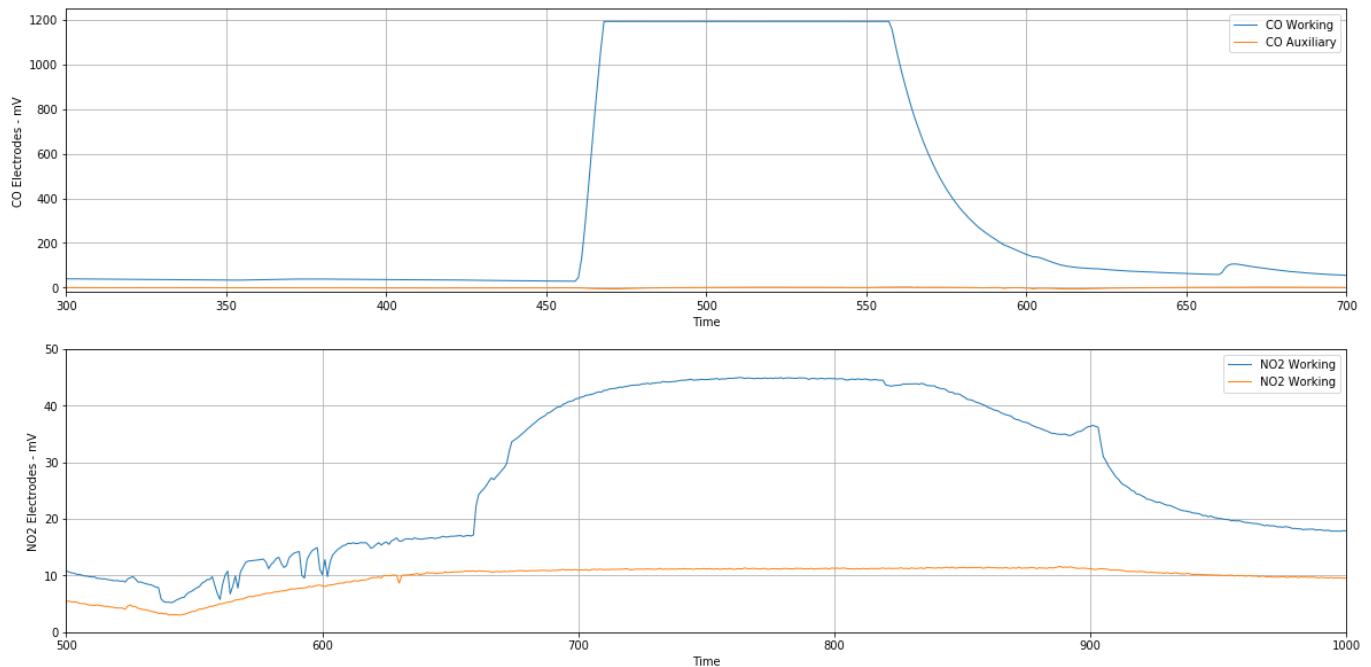
As mentioned above, the **counter electrode** is meant to balance the reaction of the working electrode. This determines the current direction within the board: whether it goes *from the working electrode to the counter electrode* or viceversa.

- Oxidation sensors, such as CO, provoke a positive current **out of the working electrode** and the larger the amount of CO present, the larger (positive) is this current.
- Reduction sensors, such as NO₂, provoke a negative current, i.e: **going into the sensor** and the larger the amount of NO₂ present, the larger (negative) is this current

As an example, this is reflected in the different signs of the sensor sensitivity:

- NO₂-B43F Average Batch Sensitivity: -347nA/ppm
- CO-B4F Average Batch Sensitivity: 588nA/ppm

Although this is in principle directly related with the sensor itself, there are further signal transformations to be taken into account. For instance, the currents seen in the electrodes, if comparing between CO and NO₂, should be different in sign, however, for both, CO and NO₂ sensors, we see positive currents which grow positively with higher CO and NO₂ concentrations:



Hence, the sensor sensitivity provided by the manufacturer should be considered in absolute terms ($\text{abs}(\text{Sensitivity})$) for the calculations to yield always positive results in pollutant fractional volumes.

11.1.2 Sensor Calibration

The model described in the following section is based on the findings of ¹. This study uses alphasense's 3-electrode sensors, and here it is further extended to the case of 4-electrode sensors, taking into account the auxiliary electrode.

Baseline correction based on temperature

The mentioned work described the correction method based on temperature using a baseline correction algorithm which is described in ². This is summarised below:

1. For each day of gas working electrode readings, and for each point in the time series (i), the minimum value of the working electrode value that is contained within the interval $(i-\delta < i < i+\delta)$ is determined, where δ is an interval ranging from 0 to a day length. The outcome of this procedure is an array where each column is a vector of minimum working electrode values calculated for each δ_i value (this is, from now on, $\text{baseline}_{\{\delta_i\}}$).
2. The correlation between each $\text{baseline}_{\{\delta_i\}}$ and the temperature is calculated. Relative humidity is not considered in this study since it's generally inversely correlated with the temperature.
3. The correlation coefficients for each correlation ($R^2_{\{\delta_i\}}$) are calculated. The maximum R^2 within this array is obtained.
4. For the equation at which the maximum $R^2_{\{\delta_i\}}$ is found, the temperature reading is used to calculate the corrected baseline.
5. The corrected baseline is subtracted from the actual working electrode reading
6. The final pollutant concentration is calculated based on the corrected working electrode reading and the manufacturer's data.

The readings are treated in a day-to-day basis in order to avoid non-stationary temperature trends over several days, but still to account for temperature variations within each day.

Finally, a background pollutant concentration is assumed from ³ which is also summarised below for each pollutant. This background concentration is added to the final result.

Table 1*Initial mixing ratios (ppb) used in the RCS.*

<i>Species</i>	<i>Chemical formula</i>	<i>Mixing ratio/ppb</i>
Nitric oxide	NO	2
Nitrogen dioxide	NO ₂	8
Ozone	O ₃	40
Carbon monoxide	CO	200
Nitric acid	HNO ₃	2
Methane	CH ₄	1800
Water vapour	H ₂ O	0.02%
VOCs		
Ethene	C ₂ H ₄	0.91
Propene	C ₃ H ₆	0.29
Formaldehyde	HCHO	3.14
Acetaldehyde	CH ₃ CHO	2.98
Isoprene	C ₅ H ₈	0.28
Methanol	CH ₃ OH	7.38
Ethanol	C ₂ H ₅ OH	2.37
Peroxyacetyl nitrate	PAN	0.46

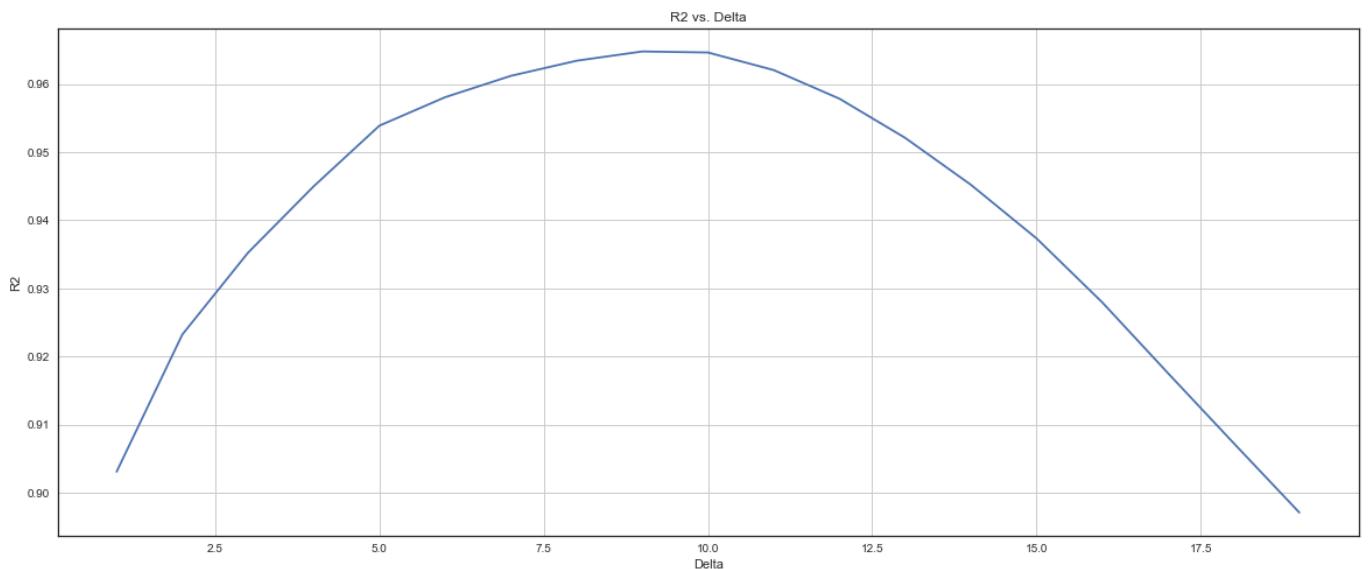
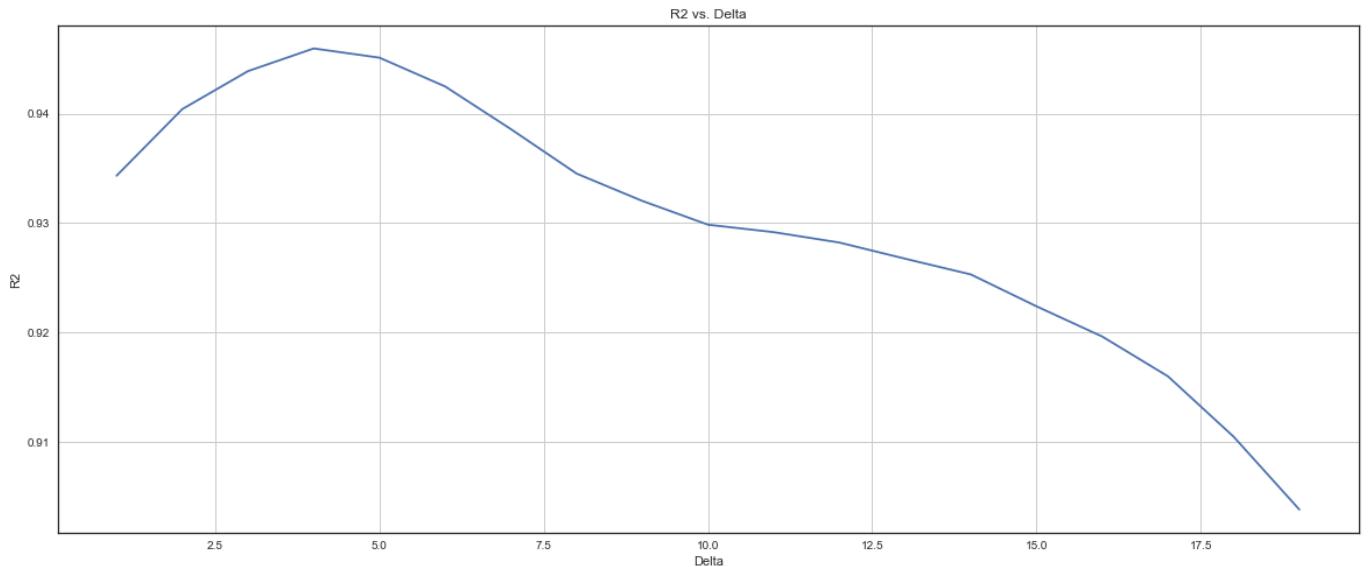
Background concentrations. Source ³

APPLICATION ON 4-ELECTRODE SENSORS

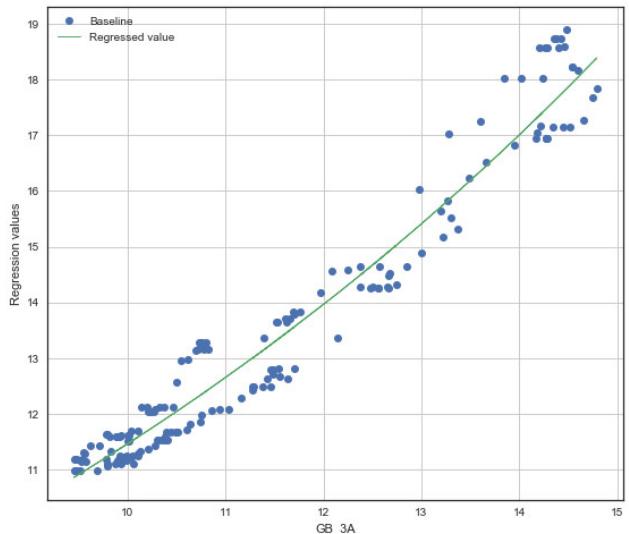
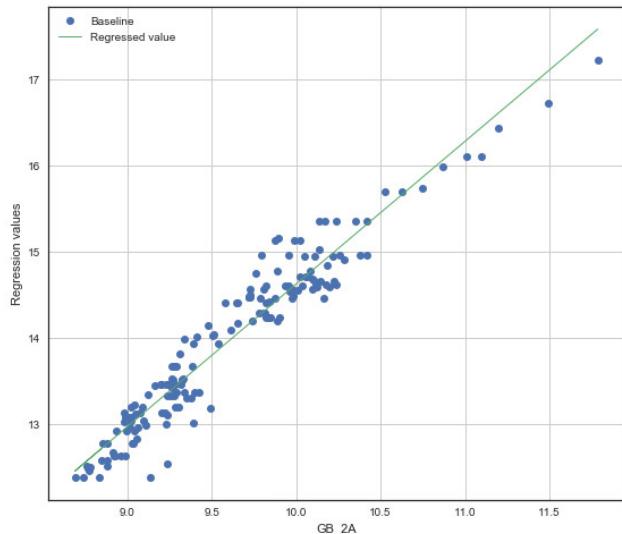
This algorithm can be used to correct temperature effects on the working electrode based on the temperature in 4-electrode sensors. The results are discussed below for tests validation campaigns performed within the iScape project. These tests are summarized below:

- *University of Bologna*: data collected from 23/January to 13/February. The measured pollutants with reference equipments were CO, NO₂, NO, NOx and O₃. Two prototype Smart Citizen Stations were deployed in two different sites, with two Smart Citizen Kits.
- *University College Dublin*: data collected from 27/March to 17/April. The measured pollutants with reference equipments were NO, NO₂ and NOX. One prototype Smart Citizen Station was deployed with two Smart Citizen Kits.

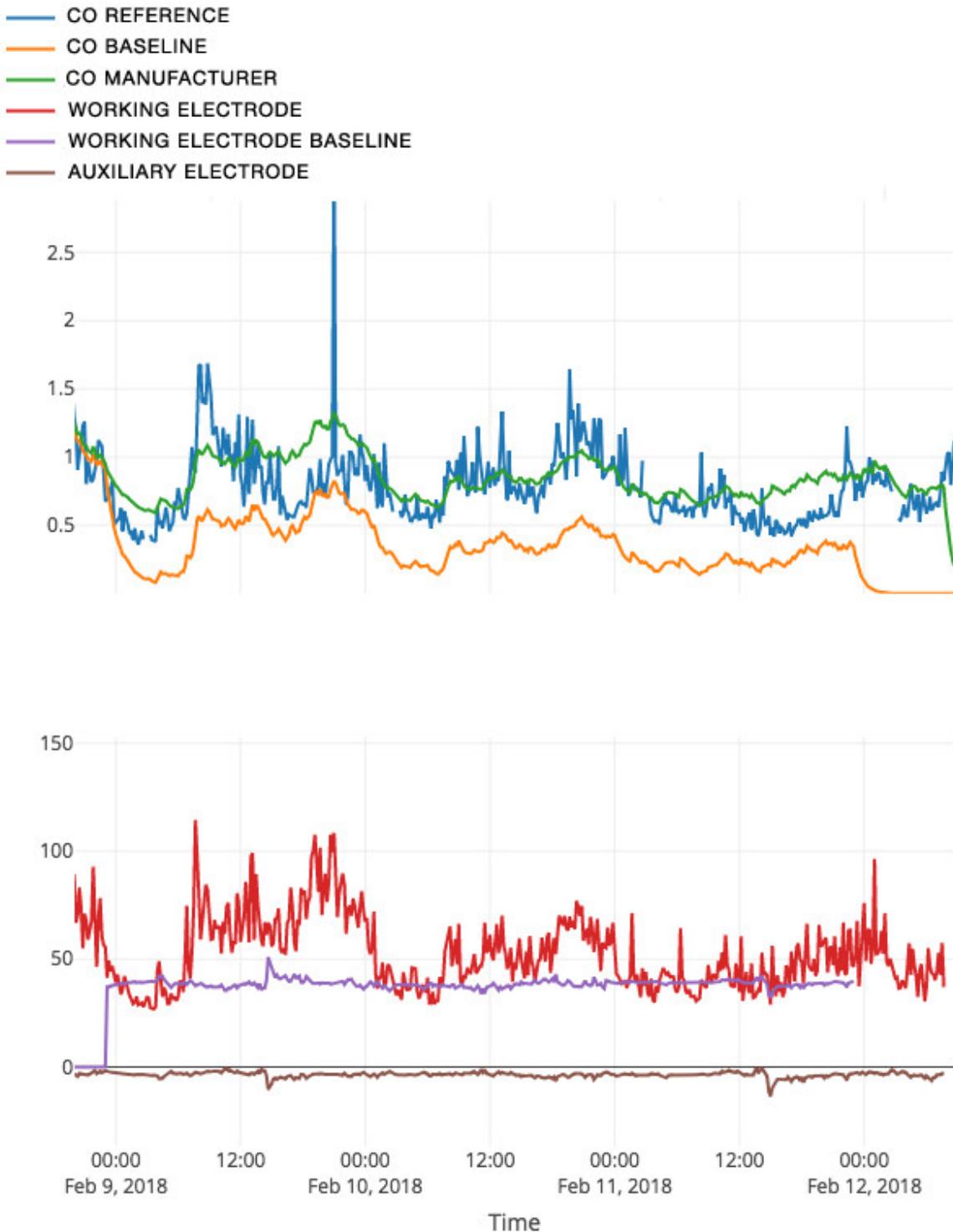
The results found with this methodology in the reduction sensors (NO₂, O₃) are significant in a daily basis. Two examples of the variation of the correlation coefficient with respect to the delta used to calculate the baseline are shown below:



The algorithm is set to apply the best performing correlation function from either a linear or an exponential fit, basing this decision on the one that yields better correlation coefficient. NO₂ and O₃ at high concentrations yield better results with an exponential fit, whilst lower concentrations reflect a linear trend:



Furthermore, the study from which this methodology is drawn from states that oxidation sensors do not yield a proper baseline correlation methodology and so is validated. The result is indeed far better correlated with the reference measurement if using the manufacturer's methodology:



This methodology reads as follows:

$$\text{Concentration } [\text{ppm}] = \frac{\{I_{\text{WE}}\} - n(I_{\text{AE}})}{[\text{nA}]} \text{ over Sensitivity } [\text{nA}/\text{ppm}]$$

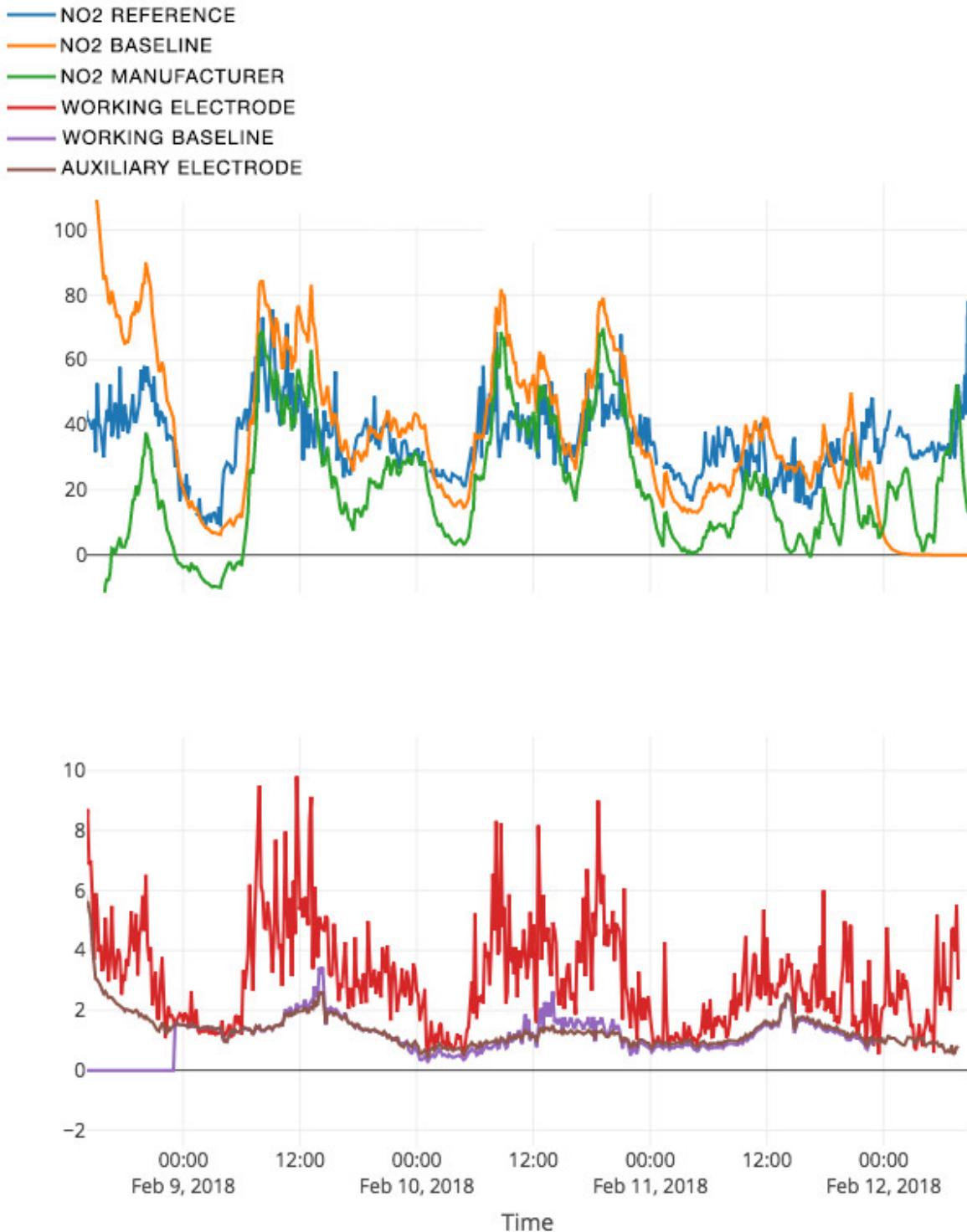
Where:

$$I_{\text{WE}} \text{ (nA)} = K \text{ (nA/mV)} V_{\text{WE}} \text{ (mV)}$$

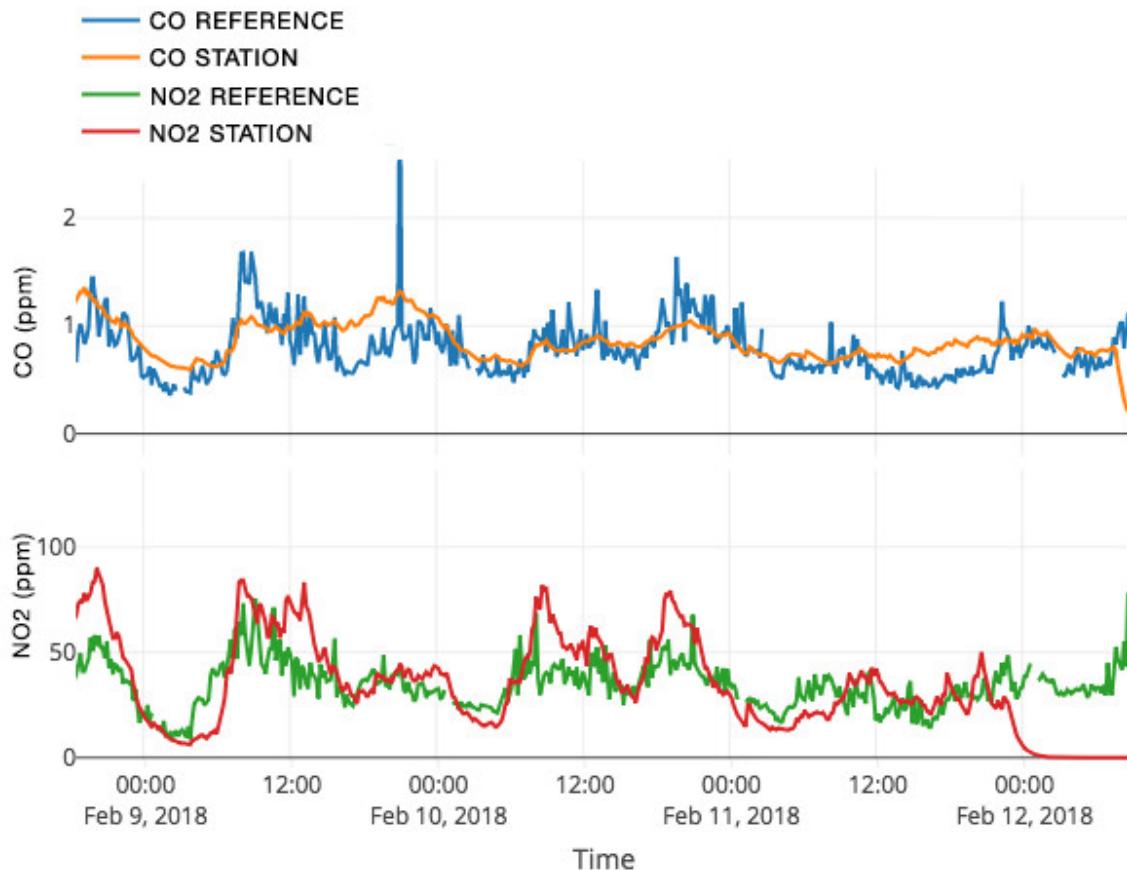
$$I_{\text{AE}} \text{ (nA)} = K \text{ (nA/mV)} V_{\text{AE}} \text{ (mV)}$$

Where: * I_{PCBWE} and I_{PCBAE} are the electronic offsets for each electrode * $n = \{I_{\text{0WE}}\} / \{I_{\text{0AE}}\}$, the ratio between alphasense's zero currents * k is a **constant conversion factor** (~ 6.36 in the case of the SCK Gas Pro Board electronics)

In the case of NO₂, the results provided by this baseline correction algorithm yield better results:



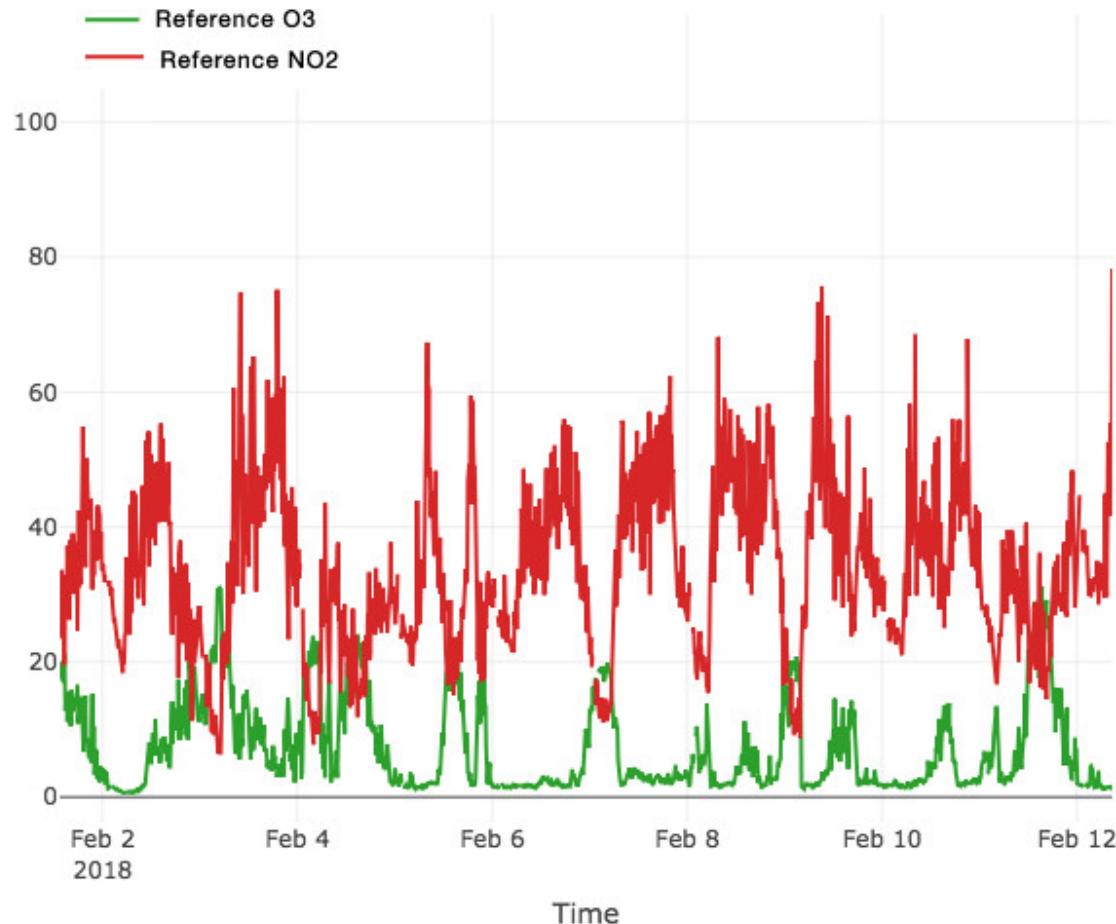
Both, CO and NO₂ pollutants, using the best method for each calculation, are shown below:



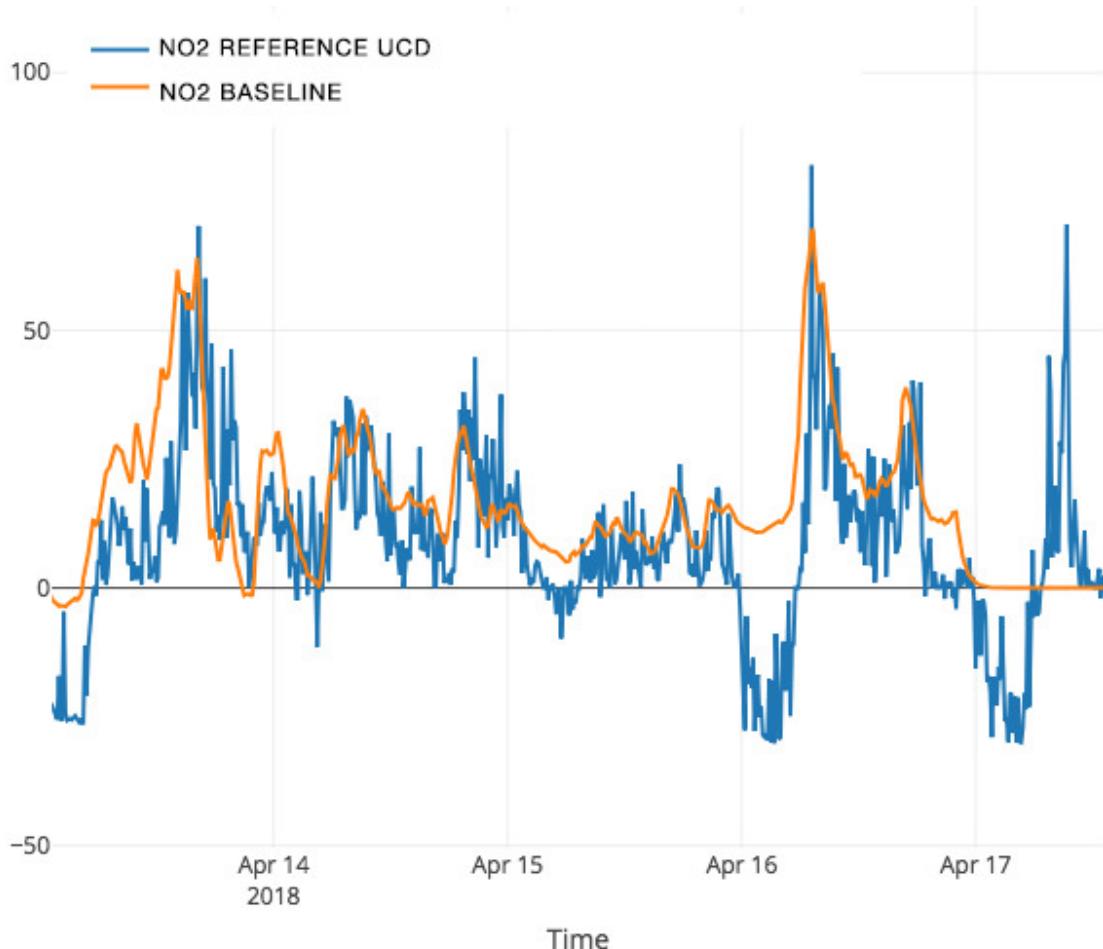
Finally, a comparison between the reference measurement results from both methods is detailed below:

	Manufacturer Method	Baseline Method
Pollutant	RMSE / R2	RMSE / R2
CO (ppm)	0.2-0.3 / 0.3-0.5	>2 / <0.01
NO ₂ (ppb)	21-24 / 0.3-0.5	6 - 12 / 0.4 - 0.6
O ₃ (ppb)	20-40 / 0.1-0.3	4-9 / 0.1 - 0.3

As seen above, the NO₂ correlation with both methods yields significant results for non-corrected signals, whilst the RMSE values are higher in the case of the manufacturer's proposal. Therefore, for this pollutant, the selected methodology will be the baseline method. On the contrary, the CO measurements are highly uncorrelated with the baseline method, whilst the original manufacturer's proposal yields decent results. Finally, the O₃ correlation levels are lower than the CO and NO₂ measurements. This is possibly due to the O₃ reference measurement equipment used in the Bologna campaign, since it shows an inverse relationship with NO₂ which suggests a biased pollutant calculation in the reference equipment:



As well, the results from UCD that are used as a reference for NO₂, suggest a poor zero/span calibration of the equipment as it yields negative results that could spoil the NO₂ correlation/model errors from those tests:



Baseline correction based on auxiliary electrode

As seen above, the results from applying this methodology to a low concentration, urban environment measurement with 4-electrode sensors yield significantly correlated results in the case of the reductive sensors. It was also seen that oxidation measurements are significantly correlated with the reference measurements while using the manufacturer's suggested method.

However, as detailed in the following section, the use of the auxiliary electrode as the source of the correction yields better results due to:

- The auxiliary electrode is accounting for both, temperature and absolute humidity. The latter could be discarded if the relative humidity is not considered.
- Since data is treated in a day to day basis, variations of mean temperatures during different days could provoke significant correlations to be found at different timelapses. This provokes gaps in the prediction during night hours that are reduced by the use of the auxiliary electrode.
- Finally, it is preferably to use data contained in a single sensor (such as the auxiliary electrode for the EC sensor) rather than including additional sensors in the algorithm.

A comparison between the results using this proposed method and the reference measurement from both test campaigns is seen below:

	Manufacturer Method	Baseline Method With Temperature	Baseline Method With Auxiliary Electrode
Pollutant	<i>RMSE / R2</i>	<i>RMSE / R2</i>	<i>RMSE / R2</i>
CO (ppm)	0.2-0.3 / 0.3-0.5	>2 / <0.1	>2 / <0.01
NO₂ (ppb)	21-24 / 0.3-0.5	6-12/0.1-0.4	6 - 12 / 0.4 - 0.6
O₃ (ppb)	20-40 / 0.1-0.3	4-12 / <0.2	4-9 / 0.1 - 0.3

11.1.3 References

1. The use of electrochemical sensors for monitoring urban air quality in low-cost, high-density networks - *M.I. Mead, O.A.M. Popoola, G.B. Stewart, P. Landshoff, M. Calleja, M. Hayesb, J.J. Baldovi, M.W. McLeod, T.F. Hodgson, J. Dicks, A. Lewis J. Cohen, R. Baron, J.R. Saffell, R.L. Jones*
2. Development of a baseline-temperature correction methodology for electrochemical sensors and its implications for long-term stability - *Olaekan A.M. Popoola*, Gregor B. Stewart, Mohammed I. Mead, Roderic L. Jones*
3. Modelling atmospheric composition in urban street canyons - *Vivien Bright, William Bloss and Xiaoming Cai*

11.2 Metal Oxide sensors

The Smart Citizen Kit has been using metal oxide sensors for air quality metrics for a long time, and we thought that it would be interesting to dedicate a section for them!

Learn More

Check this link for more information about the specifics of the eCO2 - TVOC sensor

Looking for the CO/NO2 MOs?

Check the Legacy Hardware Section!

11.2.1 A word about Metal Oxide Sensors

Metal Oxide Sensors measure the resistance (R_S) of a sensitive layer after heating it up with a *heating element* (normally another resistor). However, this reading cannot be considered as an absolute measurement of the target pollutant concentration, since the resistance varies from sensor to sensor, and it's affected by several conditions, such as temperature, humidity and other non-target pollutant affectations. To mitigate this problem, the output of the sensor is normalized using the baseline resistance (R_A): R_S is divided by R_A . This baseline resistance is the resistance that the sensor sees in clean air, and the cleaner the air is, the higher the resistance is.

Unfortunately, since R_A varies with the deployment conditions, R_A cannot be determined by a one-time calibration; and in the case of the AMS CCS811 included in the SCK V2.1, is maintained on-the-fly in software. This process is known as **baseline correction**.

Previous versions of the SCK (V1.5, V2.0 and others) included the SGX MICS4514, which was meant to measure CO and NO₂, and a lot of effort was put in V2.0 to improve the driver for the sensor, aiming to reduce power consumption and improve sensor readings. Unfortunately, this didn't match our expectations in terms of data quality and power consumption, and since individual sensor calibration is not feasible in our case (as some scientific publications have suggested), we decided to focus efforts in simpler, more robust and understandable set of sensors.

That being said, the SCK V2.1 includes the AMS CCS811 for Air Quality indicative measurements for indoor air quality in the Urban Sensor Board, and the PMS5003 for outdoor PM exposure. More complex outdoor set-ups will be also possible, for instance using the Gas Pro Sensor Board (featuring up to three Alphasense Electrochemical Sensors)^{[8][9][^10]}. This board is currently under evaluation and will be available soon.

What to expect from Metal Oxide Sensors

As said above, this type of sensors **is not meant for fine pollution monitoring**, but is more oriented for **air quality indications and trends detection**. Our approach is to use them for indicative measurements, and progressively tend towards a more reliable, fine and robust system, once the technology is capable of providing so.

While deploying them, since the air quality is expected to vary in a typical environment, the minimum time over which a baseline correction is applied is 24 hours. This means that the sensor output will change with time, until the baseline is roughly stable. Since the sensor monitors the baseline resistance periodically, if a cleaner air is found, the new baseline resistance is used to calculate the sensor readings (although this is only done for future readings). This also means that the

SCK should not be interrupted with an *ad hoc* power cut since this could erase the baseline resistance and the sensor could always yield wrong readings since it never sees *clean air*.

11.3 Introduction to Noise Sensor Design

11.3.1 Basics of MEMs I2S Microphone

The new Urban Sensor Board SCK 2.0 comes with a digital **MEMs I2S microphone**. There is a wide range of possibilities in the market, and our pick was the INVENSENSE (now TDK) ICS43432: a tiny digital MEMs microphone with I2S output. There is an extensive documentation at TDK's website coming from the former and we would recommend to review the nicely put documents for those interested in the topic.

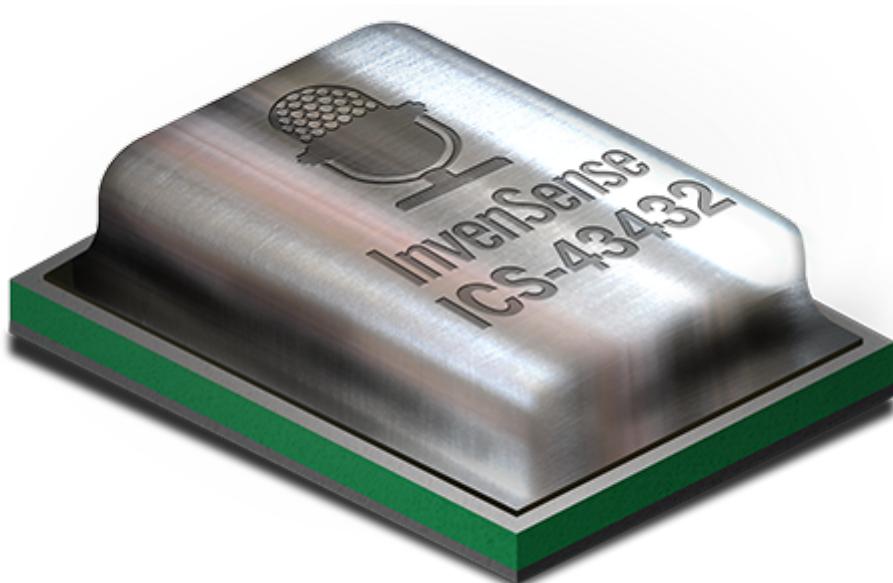


Image credit: Invensense ICS43432

To begin with, we'll talk about the microphone itself. The **MEMs microphone** comes with a transducer element which converts the sound pressure into electric signals. The sound pressure reaches the transducer through a hole drilled in the package and the transducer's signal is sent to an ADC which provides with a signal which can be pulse density modulated (PDM) or in I2S format. Since the ADC is already in the microphone, we have an all-digital audio capture path to the processor and it's less likely to pick up interferences from other RF, such as the WiFi, for example. The I2S has the advantage of a decimated output, and since the SAMD21 has an I2S port, this allows us to connect it directly to the microcontroller with no CODEC needed to decode the audio data. Additionally, there is a bandpass filter, which eliminates DC and low frequency components (i.e. at $f_s = 48\text{kHz}$, the filter has -3dB corner at 3,7Hz) and high frequencies at $0,5 \cdot f_s$ (-3dB cutoff). Both specifications are important to consider when analysing the data and discarding unusable frequencies. The microphone acoustic response has to be considered as well, with subsequent equalisation in the data treatment in order.

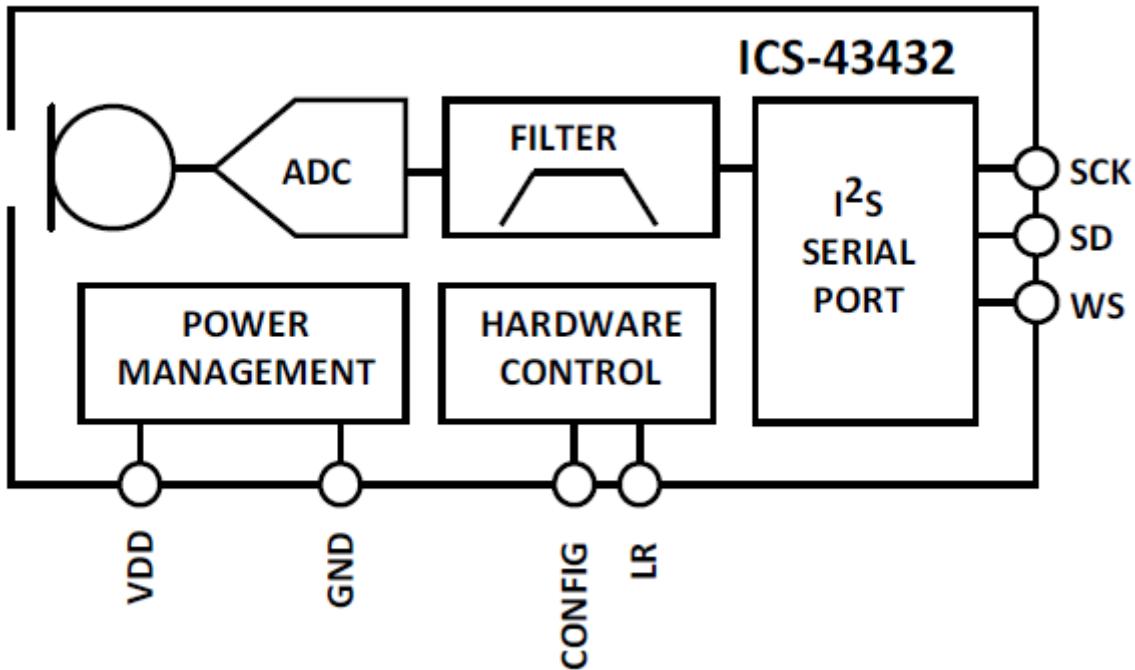


Image credit: ICS43432 Datasheet - TDK Invensense

I2S Protocol

The **I2S protocol** (*Inter-IC-Sound*) is a serial bus interface which consists of: a bit clock line or Serial Clock (SCK), a word clock line or Word Select (WS) and a multiplexed Serial Data line (SD). The SD is transmitted in two's complement with MSB first, with a 24-bit word length in the microphone we picked. The WS is used to indicate which channel is being transmitted (left or right). In the case of the ICS43432, there is an additional pin which corresponds with the L/R, allowing to use the left or right channel to output the signal and the use of stereo configurations. When set to left, the data follows WS's falling edge and when set to right, the WS's rising edge. For the SAMD21 processor, there is a well developed I2S library that will take control of this configuration.

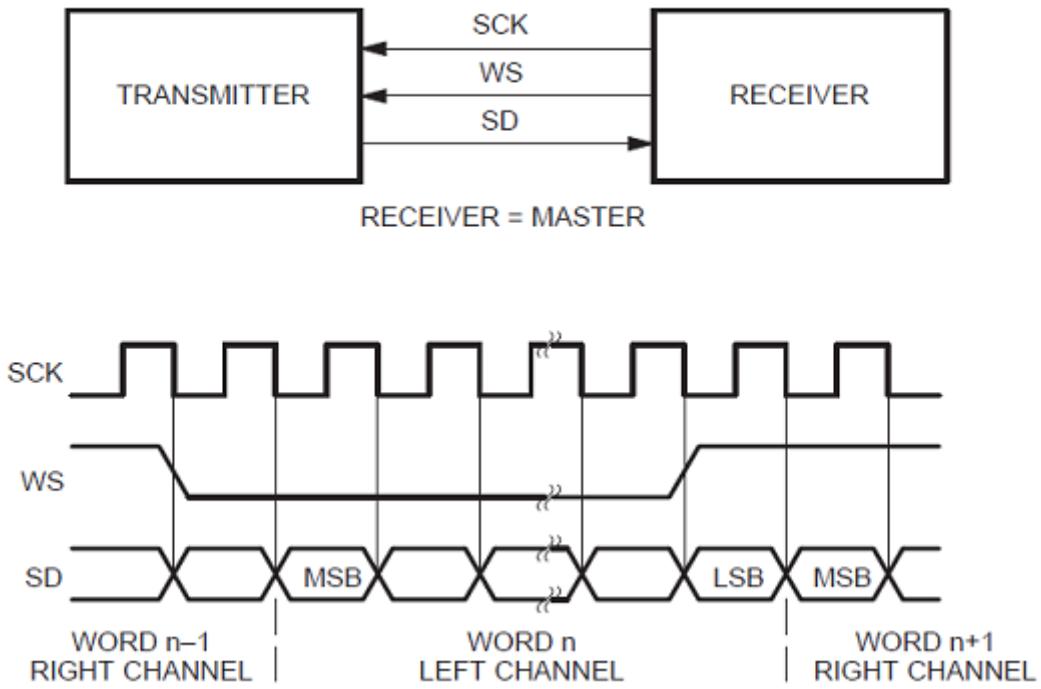


Image credit: I2S bus specification - Philips Semiconductors

Also, we would like to highlight that the SD line of the I2S protocol is quite delicate at high frequencies and it is largely affected by noise in the path the line follows. If you want to try this at home (for example with an Arduino Zero and an I2S microphone like this one, it is important not to use cables in this line and to connect the output pin directly to the board, to avoid having interfaces throughout the SD line. One interesting way to see this is that every time the line sees a medium change, part of it will be reflected and part will be transmitted, just like any other wave. This means that introducing a cable for the line will provoke at least three medium changes and a potential signal quality loss much higher than a direct connection. Apart from this point, the I2S connection is pretty straight forward and it is reasonably easy to retrieve data from the line and start playing around with some FFT analysis.

11.3.2 Basics of weighting and human hearing

The world of acoustics and signal processing for audio analysis is worth several book-length discussions. We might as well try to give an insight of our intentions within this world since we introduced ourselves in it by picking a digital microphone with a quite nice range of capabilities.

The very first thing we would like to do is to be able to perform **weighting** on the buffer we receive from the microphone through the I2S. To explain a bit further on what *weighting* is, it is no more than a transformation from the real-world sound pressure levels (SPL) travelling around in the air to what our ears can perceive. Just that.

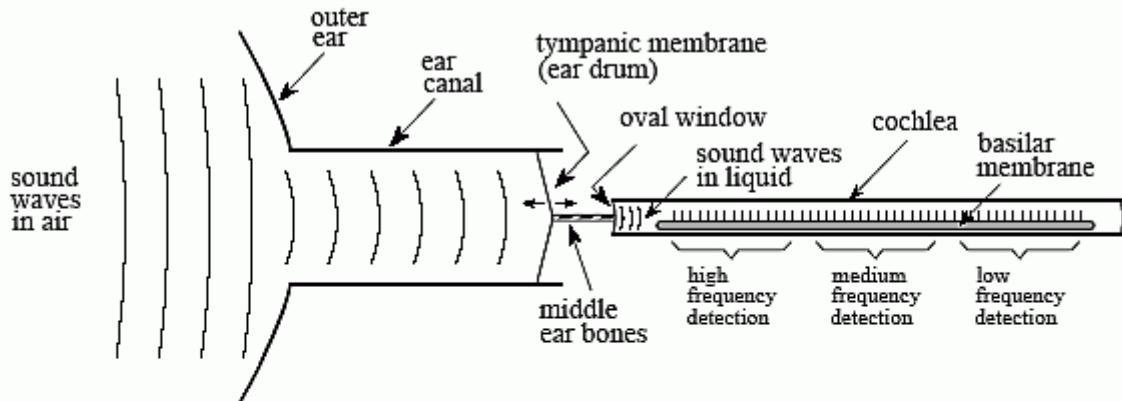


FIGURE 22-1

Functional diagram of the human ear. The outer ear collects sound waves from the environment and channels them to the tympanic membrane (ear drum), a thin sheet of tissue that vibrates in synchronization with the air waveform. The middle ear bones (hammer, anvil and stirrup) transmit these vibrations to the oval window, a flexible membrane in the fluid filled cochlea. Contained within the cochlea is the basilar membrane, the supporting structure for about 12,000 nerve cells that form the cochlear nerve. Due to the varying stiffness of the basilar membrane, each nerve cell only responds to a narrow range of audio frequencies, making the ear a frequency spectrum analyzer.

Image credit: Human hearing - DSP Guide

There are several studies and models of what we actually perceive and depending on them, we have several types of so called **weighting functions**. Some of them have been standardised for the purpose of SPL measurement, finding different types like **A-weighting** (the most common one), B-weighting, D (both in disuse) and others. In the frequency domain, they look like this:

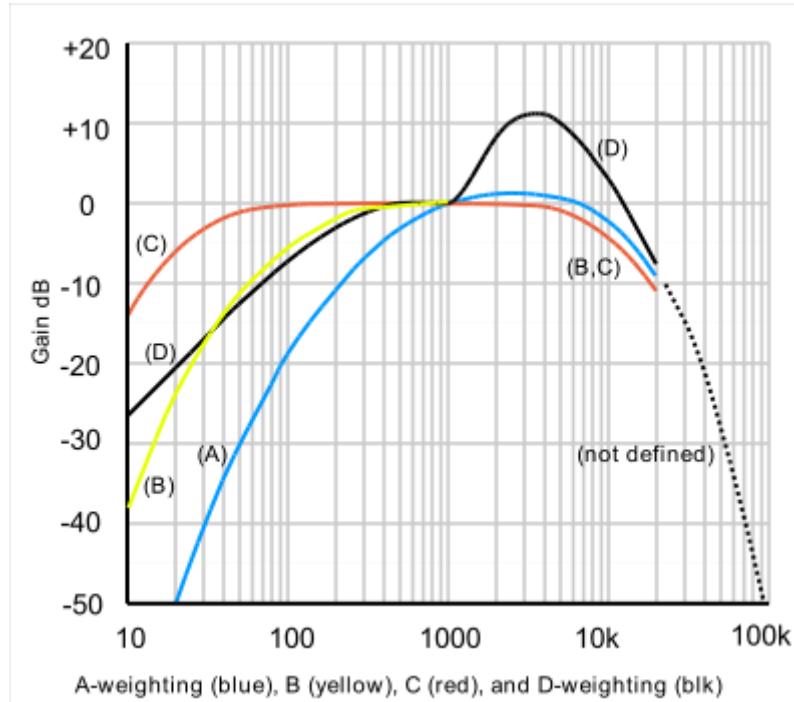


Image credit: A-weighting - Wikipedia

This means that, even if there are high sound pressure levels floating around in the air, we might not hear them just because of the frequency they are at. Normally humans can hear from something around 20Hz to 20kHz, although most adults might not hear anything in out-of-laboratory conditions above 15kHz. Some animals though, can perceive a great range of frequencies, and for example mouses can hear up to 80kHz! So, now we know what this all is about, the I2S microphone is going to help us understand better how *beluga whales* communicate among themselves...

But also! The I2S microphone is interesting in order to **understand sources of urban noise pollution** since it provides us with a raw SPL buffer we can play with. As well, we can obtain dBA levels (SPL with a-weighting correction) by processing this buffer in several ways and calculate the RMS level of the resulting signal.

11.3.3 Signal postprocessing

RMS and FFT algorithm simplified

In this paragraph we'll continue with some bits and pieces about *acoustics and signal processing*. In the previous section we introduced the concept of **weighting** and our interest on calculating the **sound pressure level** in different scales. Normally, SPL is expressed in **RMS** levels, or *root mean square*. This is nothing more than a modified arithmetic average, where each term of the expression is added in its square form. Therefore, to keep the same units, we then take the square root of all the average and we have:

$$x = \{\sqrt{x_1^2 + x_2^2 + \dots + x_N^2} / N\}$$

The interesting thing about the RMS level, is that it expresses an average signal level throughout the signal, and it actually relates to the peak level of sinusoid wave by $\sqrt{2}$. Therefore, it is a very interesting way to express average levels for signals and for that reason, it's the common standard used.

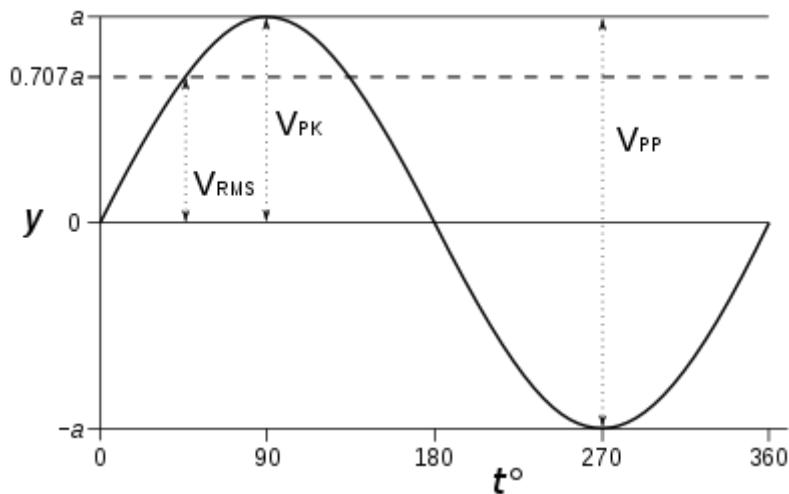


Image credit: Sine wave parameters- Wikipedia

Now that we know how to calculate the RMS level of our signal, let's go into something more interesting: *how do we actually perform the weighting?* Well, if you recall the previous section, when we talked about hearing, we were talking about the different hearing capabilities in terms of **frequencies** (in humans, mouses, beluga whales...). Therefore, something interesting to know about our signal is its **frequency content**, so that we are able to perform the weighting. For this purpose, we have the **FFT algorithm**, which we won't tell you is easy, but we'll try to put it simply here.

So **FFT** stands for **Fast Fourier Transform**, and it's an algorithm capable of performing a Fourier Transform in a simplified and efficient way (that's where the *fast* comes in). What it does in a detailed mathematical way is something quite

complicated and we don't want to bore you and ourselves with the details; but being practical, it is basically a conversion between the signal in time domain and its frequency domain components. Interestingly, this process is reversible and the other way around it is called **IFFT** (*I* for *Inverse*, obviously...).

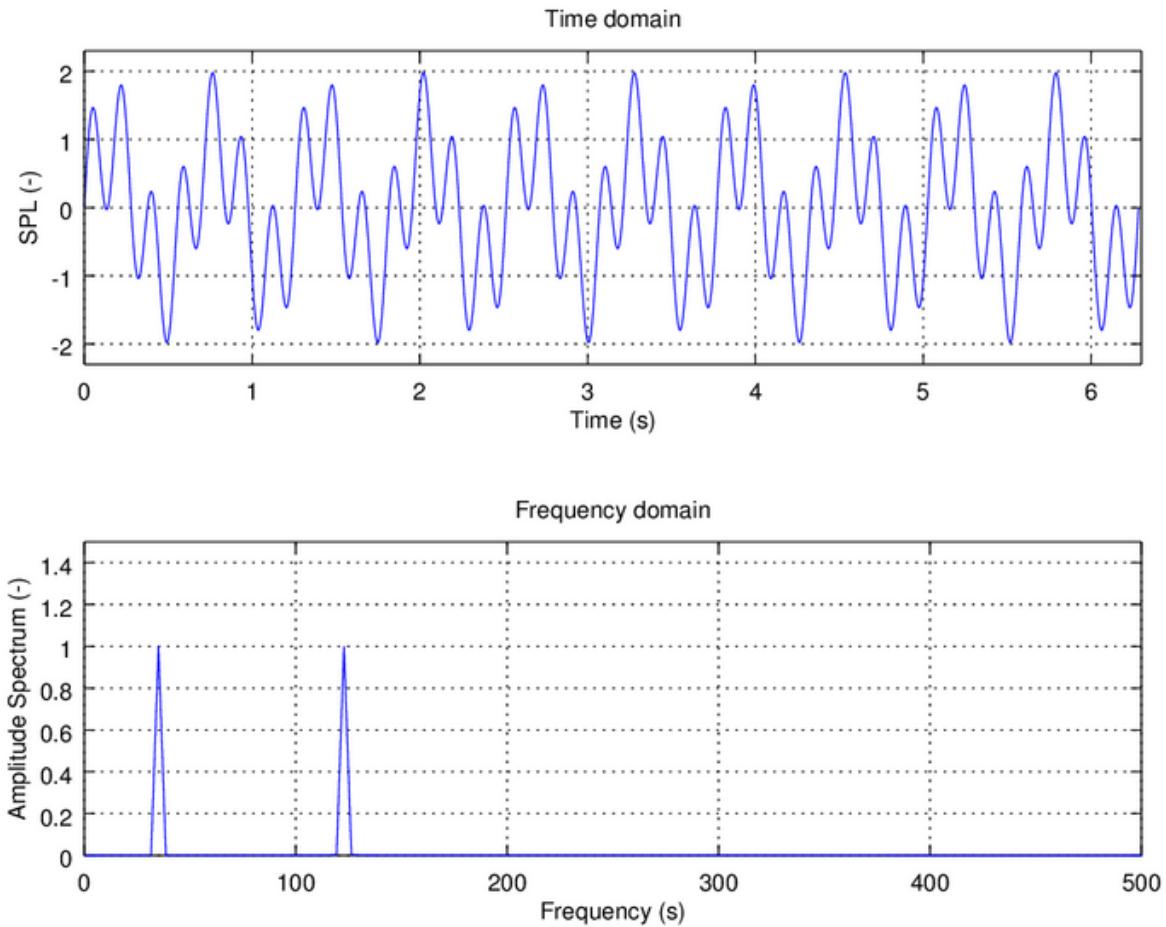


Image credit: Smart Citizen

In the example above, things in the time domain get a bit messy, but in the frequency domain we can *clearly* see the composition of two sine waves of the same amplitude of roughly 40Hz and 120Hz. The FFT algorithm hence helps us digest the information contained in a signal in a more visually understandable way.

For this introduction, let's move on to what we actually want to do: *the much anticipated weighting*. At this point, our task is fairly easy: we just have to multiply both: our signal in the frequency domain with the weighting function and that's it! If we have a look at the figure below, in the time and frequency domain, the signals look like this:

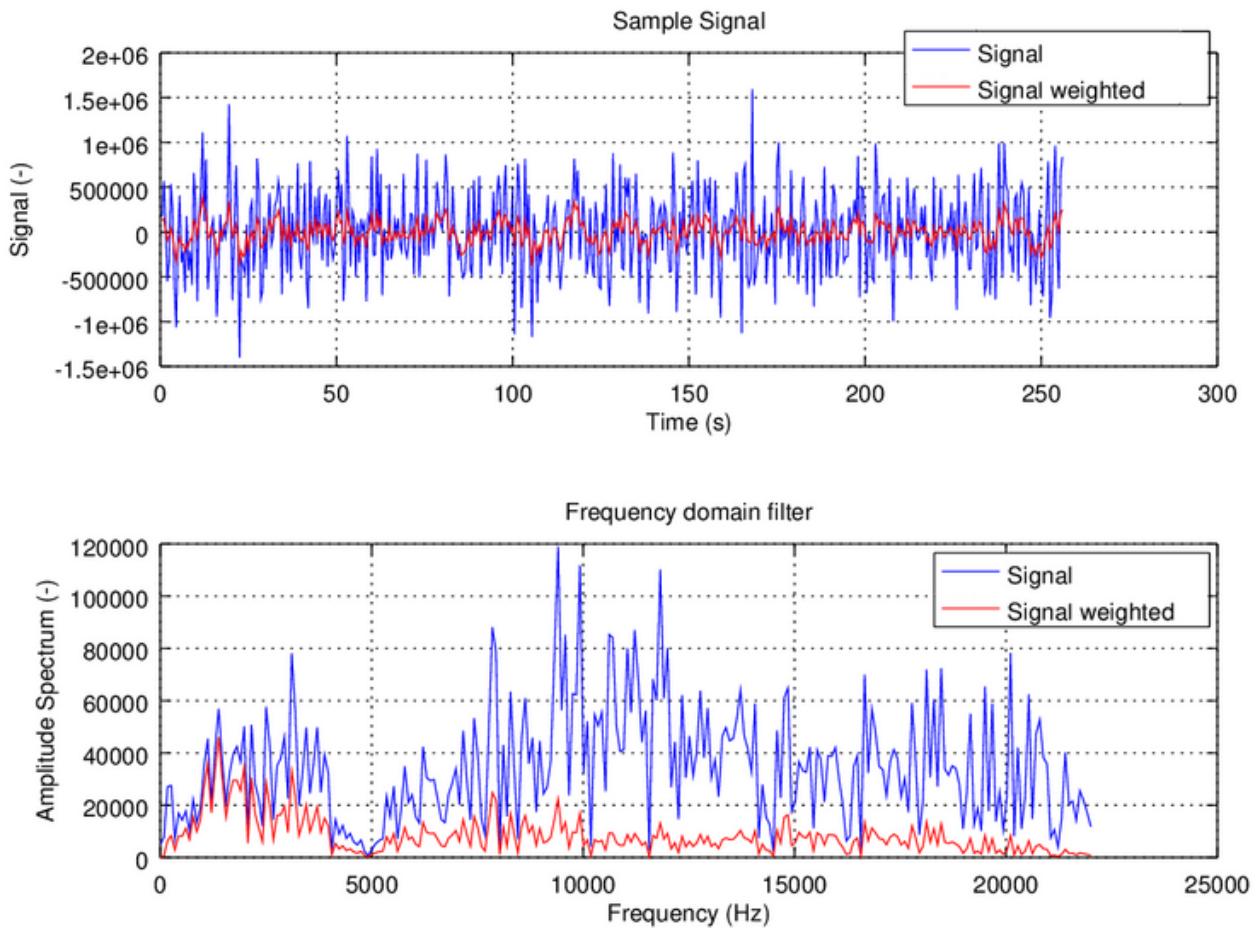


Image credit: Smart Citizen

This example shows how our ears are only capable of perceiving the signal in red, but the actual sound components are in blue -- being much higher in the amplitude spectrum. If you want to get into the thick of it, here you have the actual implementation in Matlab of the A-weighting function that we'll use in the SCK V2.0.

And finally, to close, let's take a look at the whole chain of processing, where we will continue in future sections:

1. Signal acquisition
2. Windowing
3. FFT
4. Spectrum Normalisation
5. Equalisation
6. A-weighting
7. RMS calculation

This is the whole signal treatment process we use for the I2S microphone ICS43432. We will have a look at *windowing* and its use in future sections, as well as its implementation in the SAMD21 Cortex M0+ for our firmware.

NB: Being mathematical purist, there is yet another possibility for this procedure using convolution in time domain, which we will cover in future sections.

Pre/post processing: signal windowing and equalisation

SIGNAL WINDOWING

In this section we are going to describe how we have to pre-post process our signals in order to obtain the results in the manner we are expecting. These are very important steps in our processing chain, since the FFT algorithms -or convolution FIR Filters- won't be able to cope with our system's limitations. These limitations might not be obvious at the beginning, but you really don't want to ignore them while designing your system, since they'll invalidate many of your measurements.

The very first of these limitations, is the fact that our microphone is, in fact, taking **discrete samples** of the ambient noise surrounding it. This means that, from the very beginning, we are missing some pieces of information and we will never be able to process them. But it's OK! For the purpose of our analysis, we don't need to sample continuously and this situation is easily bypassed.

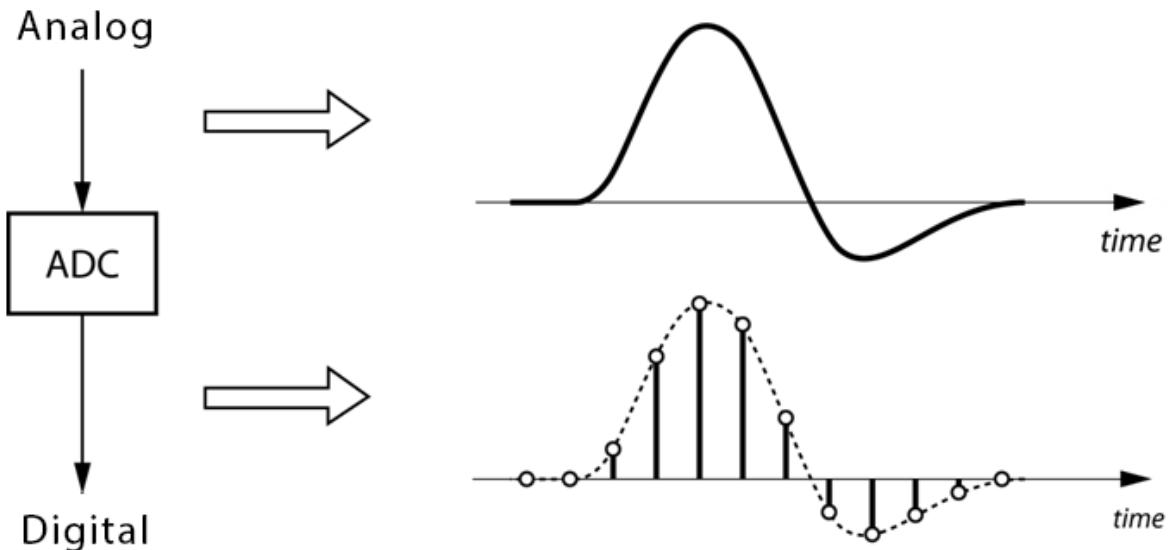


Image credit: NUTAQ - Signal processing

Discrete sampling has two main consequences for us: the first one is that we are taking samples once every $1/f_s$, where f_s is the sampling frequency. Normal audio systems sample at 44,1kHz, but this number might vary depending on the application. If you remember this chart, you might be wondering why we have to sample at such a high frequency.



Image credit: Signal acquisition - Adinstruments

This is due to the Nyquist sampling criterion, which states that **at a minimum, we have to sample at double the maximum frequency we want to analyse**. Since humans hearing has a limited frequency range that goes up to 20kHz in some cases, it is reasonable to use something around 40kHz. With this, the *Nyquist criterion* solves the so called **aliasing problem**, in which several sinusoid signals could fit the same sampling pattern if the number of samples is too low:

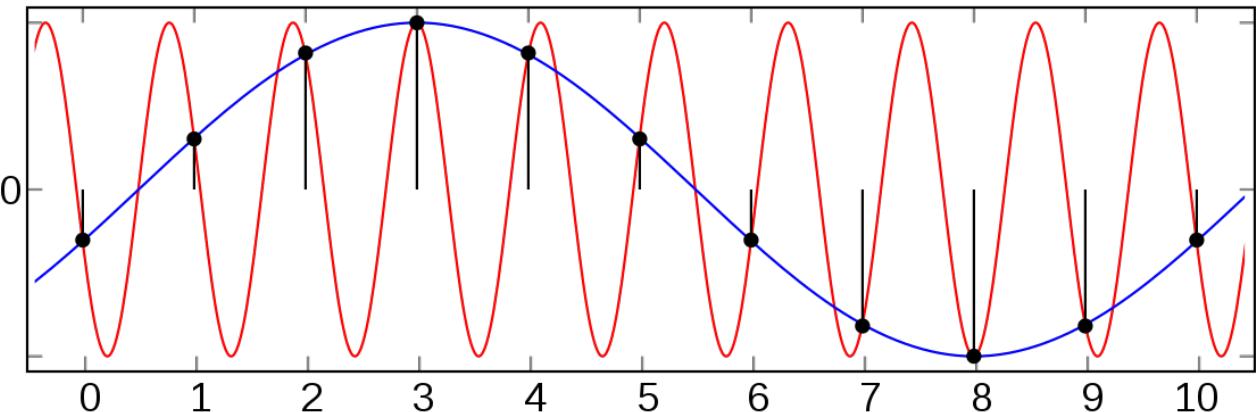


Image credit: Wikipedia - Aliasing

The second of the discrete sampling limitation comes from the **amount of samples we are able to handle at a time**. Normally, this is due to memory limitations in the RAM, although we'll see in the future where to allocate them. Nevertheless, it is not useful to handle buffers that are *too long*, since at some point, the increase of buffer length does not provide any additional information. Buffer length requirements in our case come from the minimum frequency we want to sample, which is *around 20Hz*. Doing some quick math, we need 0,05s worth of sample buffer, which at 44,1kHz is roughly *2200 samples*. This is equally too many samples, considering that each could be allocated as a `uint8_t`, taking up to 16kB just for the raw buffer!

This is where **signal windowing** kicks in. Imagine that we have a very-low-frequency sinusoid and that we are not able to sample completely the whole sine wave, due to buffer limitations. By definition, our system is assuming that the discrete samples we measure are constantly being repeated in the environment, one after the other:

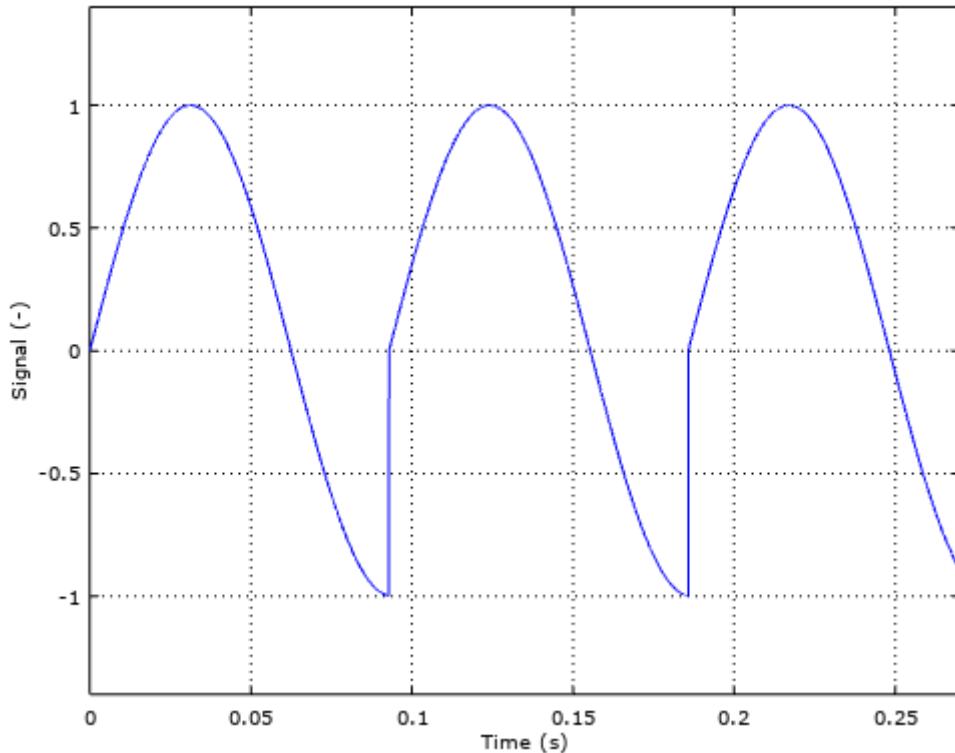


Image credit: Smart Citizen

When we take the FFT of this signal, we see undesired frequencies that make our frequency spectrum invalid. This is called **spectral leakage** and it's mitigated by the use of *windows* (math functions, not the OS). These windows operate by **smoothing the edges** of our measurement and preventing the *jumps* in the signal helping the FFT algorithm to properly analyse the signals.

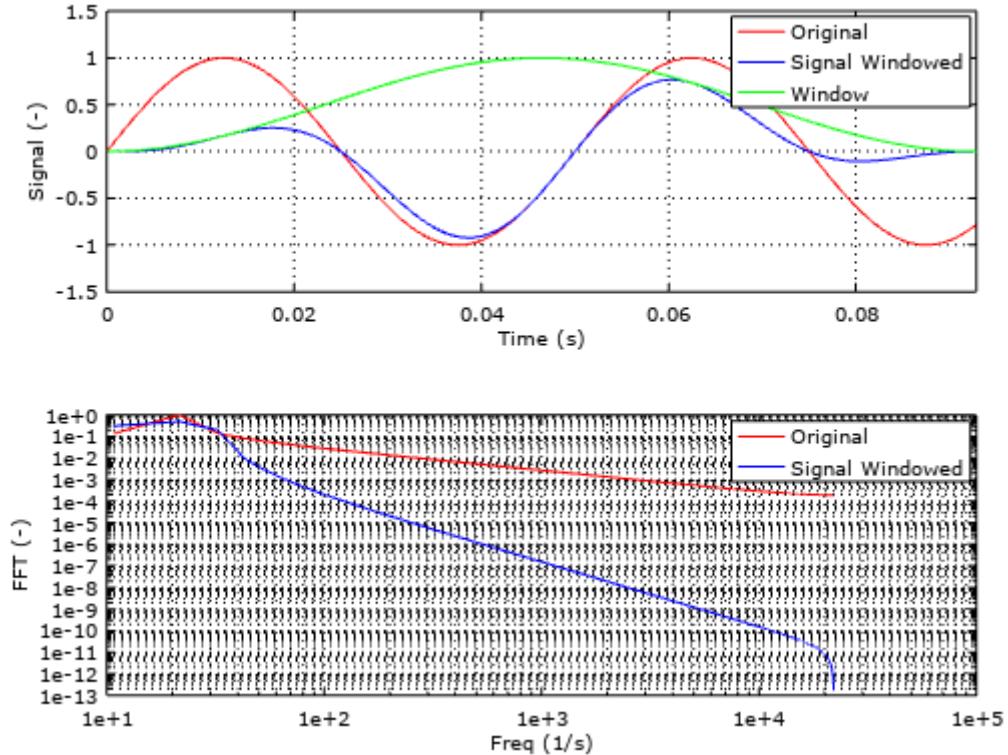


Image credit: Smart Citizen

With the use of *signal windowing*, more specifically with the use of the *hamming window*, we are then able to reduce the amount of samples needed to roughly 1000 samples. Now we are down to **50% of the memory allocation needed without windowing**. You can see the effect on the *RMS relative errors* in the image below, where the trend of the Hann (another common window) and the Hamming treated buffers, with respect to the frequency tends to stabilise *much more quickly* than the *raw buffers*.

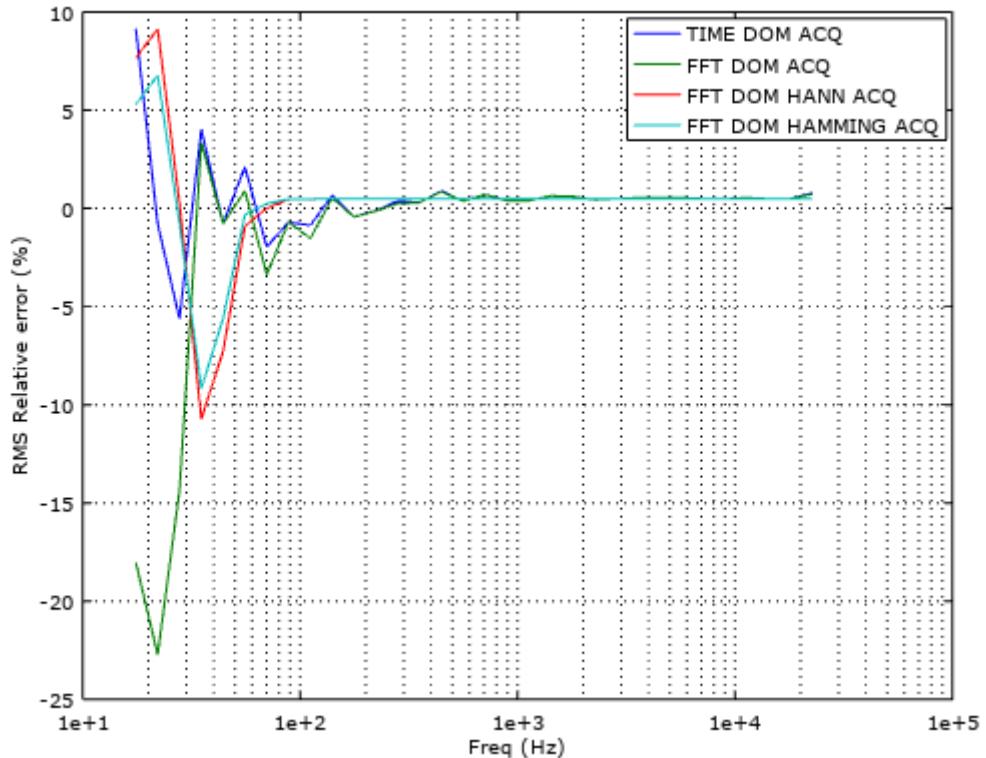


Image credit: Smart Citizen

There is a wide range of functions to use and the decision depends on your application. For audio applications, the most common ones are the Hann, Hamming, and Blackmann. We chose the Hamming because it's trend is to stabilise a bit more quickly than the rest, although the differences are minimal. For your reference, there is a very interesting description of all these phenomena in this article, where you'll find a more mathematical approach.

Info

Talk about the microphone response and how to correct it.

Filtering and convolution

In this section we are going to talk about a different approach to the FFT Analysis we have seen in previous sections. *What if* we don't like the FFT algorithm and we only want to obtain a dBA or dBC results? There is a fairly simple solution to this problem, and it's called **filtering**.

Filtering is a very common technique in signal acquisition that eliminates some frequency components of the raw signal. Examples of filters you very likely have heard of are *low-pass, high-pass and band-pass filters*. These only *let pass* the low, high or a defined interval range of frequencies, mostly cancelling out the rest. In the frequency domain, they basically multiply the spectrum of our signal with its filter spectrum. Exactly what we have done with the weighting.

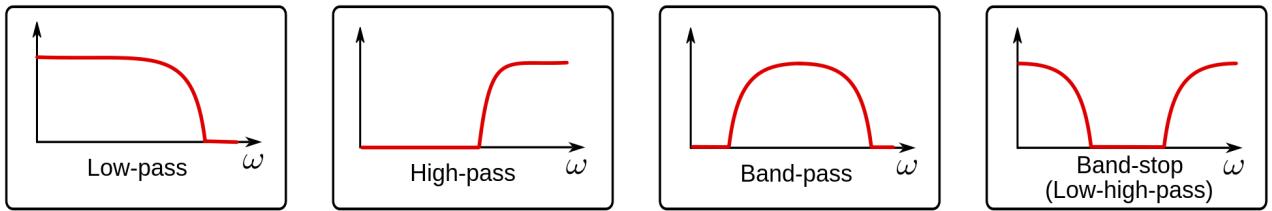


Image credit: Norwegian Creations

First, it is important to get a glimpse of the math behind the filters and why they do their magic. And for this, the most important thing we need to know is called **convolution**.

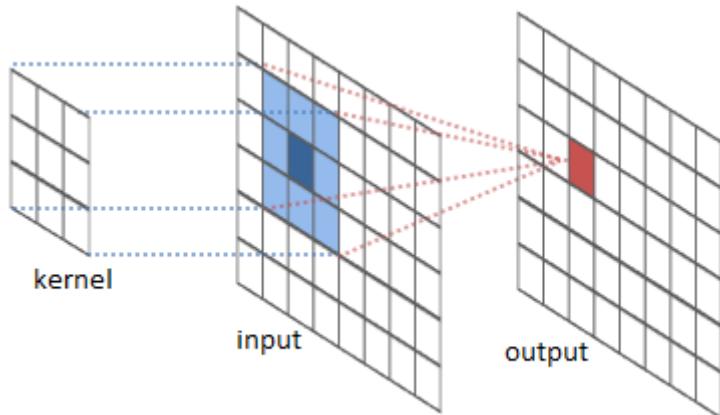


Image credit: River Trail

For the purpose of **audio analysis**, let's consider we have an input vector, a filter kernel and an output vector. Our input vector can be the raw audio signal we have captured, being the output signal the result of the convolution operation. The filter kernel is the characteristic of the filter and will be, for this example, a one dimension array. What the convolution operation is going to do, in a *very very very simplified way*, is to **sweep through the input sample** and multiply each component with its corresponding filter kernel component, then sum the results and put them in the corresponding output sample. If we put some math notation and call $x[n]$ to the input vector, $h[n]$ to the filter kernel and $y[n]$ to the output vector, it all ends up looking like this:

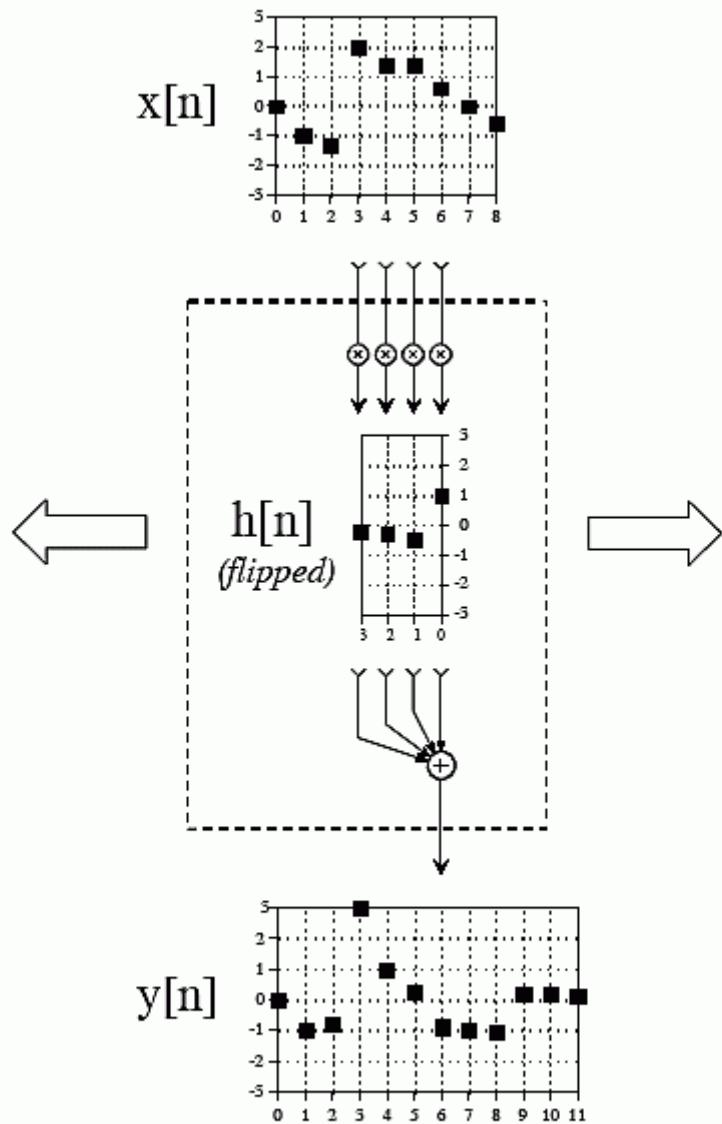


FIGURE 6-8

The convolution machine. This is a flow diagram showing how each sample in the output signal is influenced by the input signal and impulse response. See the text for details.

Image credit: DSP Guide

Now, the most interesting thing of all this theory is that **convolution and multiplication are equivalent operations when we jump from the time to the frequency domain**. This means that multiplication in time domain equals to convolution in frequency domain, and more importantly for us, **convolution in the time domain, equals to multiplication in the frequency domain**. To sum up, the relationship between both domains would look like:

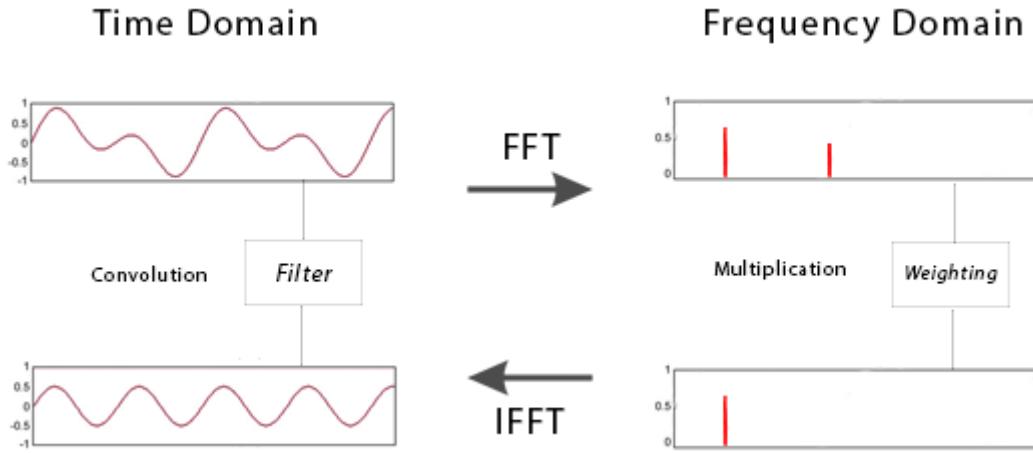


Image credit: SmartCitizen

Therefore, what we could do is to define a **custom filter function** and apply it via convolution to our input buffer. This is basically a **FIR filter**, where *FIR* stands for *Finite Impulse Response*. There is another type of filters called **IIR**, where *IIR* stands for *Infinite impulse response*. The difference between them is that *FIR* uses *convolution* and *IIR* uses *recursion*. The concept of **recursion** is very simple and it's nothing else than a simplification of the convolution, given that in the convolution algorithm, there are many recursive operations that we repeat over and over and we can implement into a smarter algorithm. Normally, *IIR* filters are *more efficient in terms of speed and memory*, but we need to specify a series of coefficients, and it's tricky, if not impossible, to create a custom filter response.

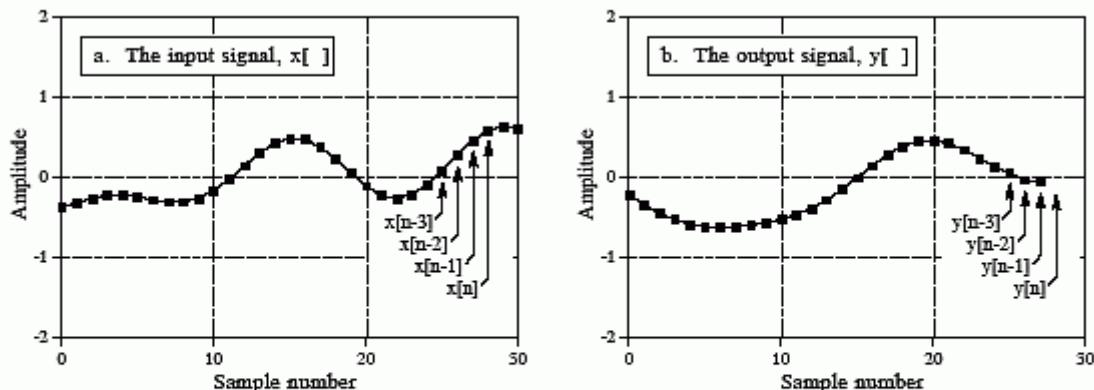


FIGURE 19-1
Recursive filter notation. The output sample being calculated, $y[n]$, is determined by the values from the input signal, $x[n]$, $x[n-1]$, $x[n-2]$, ..., as well as the previously calculated values in the output signal, $y[n-1]$, $y[n-2]$, $y[n-3]$, ... These figures are shown for $n = 28$.

Image credit: DSP Guide

So finally! How can we avoid using the FFT algorithm to extract the desired frequency content of a signal and recreate the signal without it? Sounds complex, but now we know that we can use a **FIR filter**, with a **custom frequency response** and apply it via convolution to our input buffer. As simple as that. The custom frequency response, with the proper math, can be obtained by applying the IFFT algorithm to the desired frequency response (for example, the A-weighting function). You can have a look to this example if you want to create a custom filter function in octave, with A or C weighting and implement it to a FIR filter in C++.

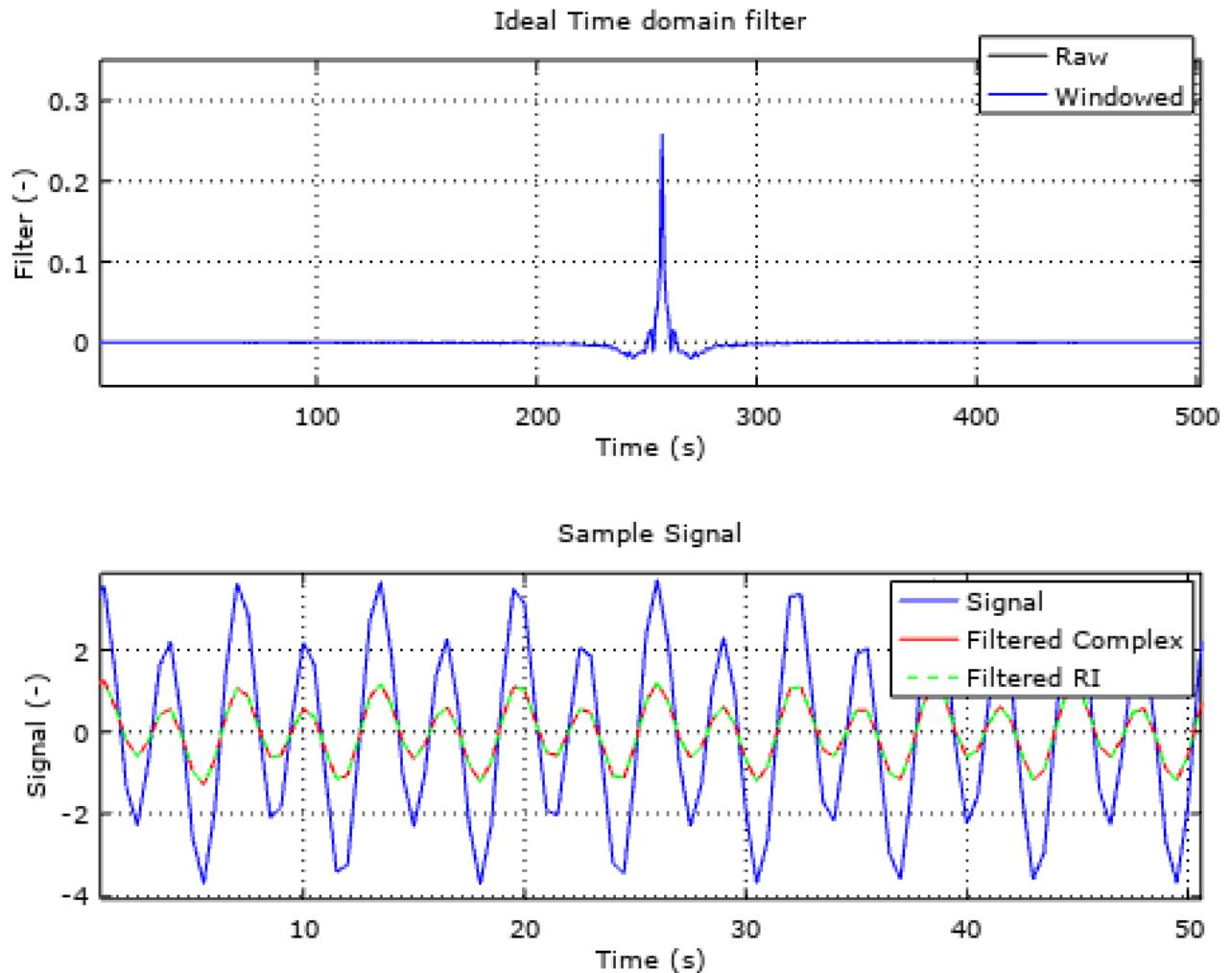


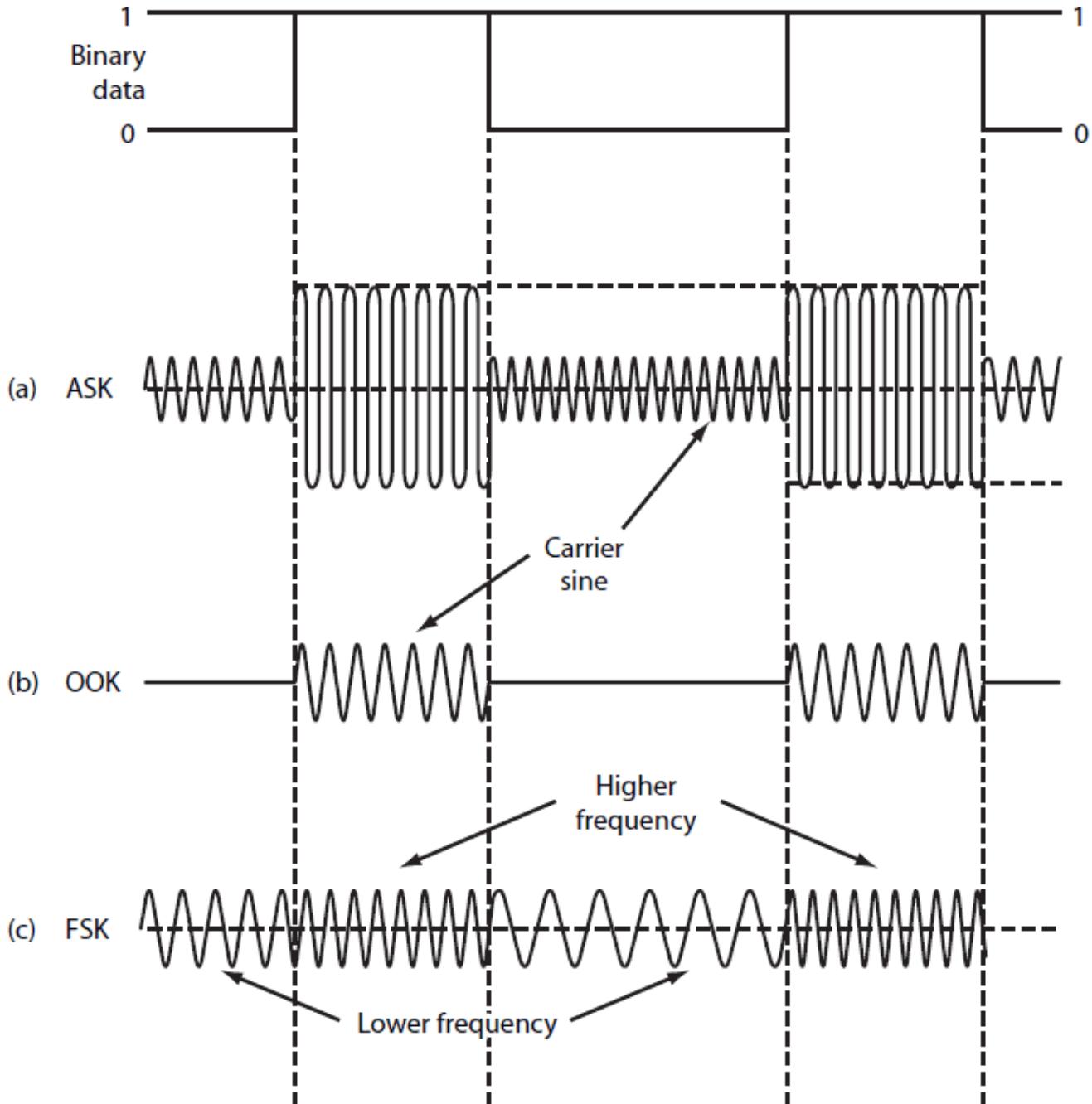
Image credit: SmartCitizen

Also, if you are really into it, you can read more about convolution and other DSP topics, we would recommended to go through this fantastic guide.

AFSK Analyser

In this section we are going to talk about a new feature we are planning to introduce in the upcoming version of the SCK: a FSK communication protocol via Audio (A-FSK). You might have read about this technique and its usage in the Amazon Dash configuration process, and on the post today we are going to describe very briefly the work in progress for this feature.

So! FSK stands for Frequency Shift Keying, which is a form of transmission through frequency variations on the carrying waveforms. Its major counterpart is the so called ASK, or Amplitude Shift Keying, in which the transmission is carried out via amplitude variations. A very simple form of ASK is OOK, which stands for On-Off-Keying, in which the amplitude of the carrier wave oscillates between a value and nothingness:



1. Three basic digital modulation formats are still very popular with low-data-rate short-range wireless applications: amplitude shift keying (a), on-off keying (b), and frequency shift keying (c). These waveforms are coherent as the binary state change occurs at carrier zero crossing points.

Image credit: Electric Stack Exchange

As in many other situations, there is a trade off between the options on the table: ASK or FSK? Maybe another one? The main disadvantage of ASK it is said to have a higher probability of error with respect to FSK, since noise interference affects amplitude of the transmitted wave. FSK, on the other hand, it is said to have a lower bandwidth efficiency. However, since

we have talked about FFT quite a lot now, we thought FSK would be our best bet and also, because maybe bandwidth is not such a big deal after all as we can see below.

Then, the idea is to implement an algorithm that is able to identify if the sound transmitted from an emitter (i.e. a smartphone) contains a series of reference frequencies in certain known spots. Following this principle, our aim is to transmit a byte per sound wave, hence, in a sound wave containing up to 8 possible carrier frequencies that might or not be activated. The activation (or not) of these frequencies in the analysed spectrum will yield a 1 or a 0, that we can use on a bit mask and extract 8-bit ASCII characters codes:

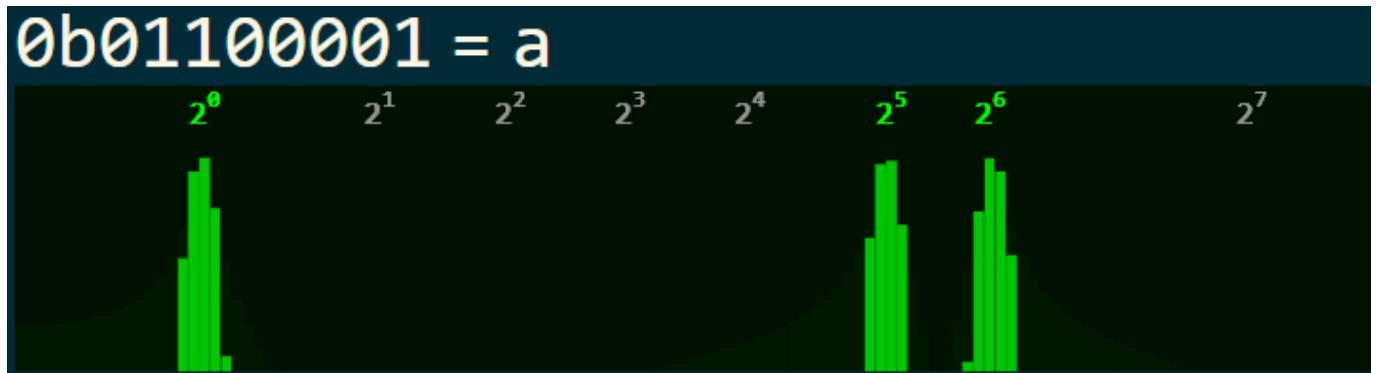


Image credit: Martin Melhus

The emitter could be based on the Web Audio API, as the example from Martin Melhus above from his project on a Web Audio Modem. Finally, the receiver would be our beloved I2S Mems microphone that we have been talking about for so long now, doing a FFT algorithm and detecting the peaks in it, identifying the carrier frequencies activation.

11.3.4 Field Evaluation

The sensor is calibrated in an anechoic chamber with a reference microphone to obtain sensor characteristics for spectrum equalisation. The TDK ICS43432 (former Invensense) has a clear non-linear response, which is specified in its datasheet and is characterised in an anechoic chamber as specified above:

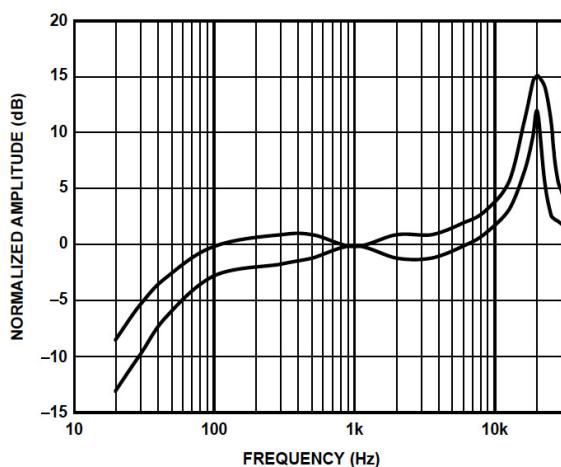


Figure 4. Frequency Response Mask

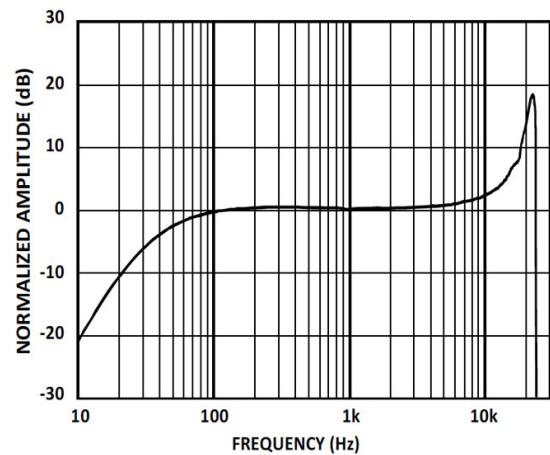
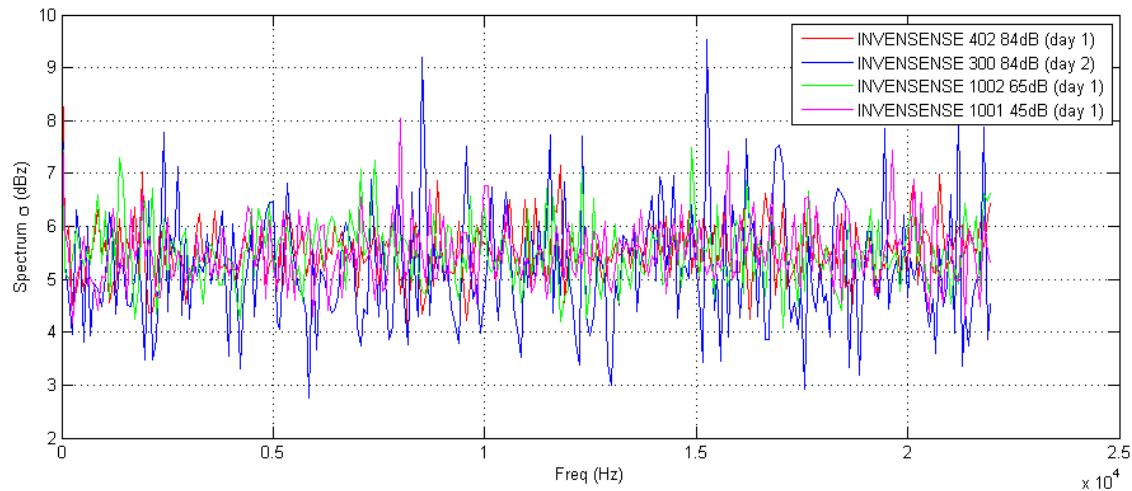
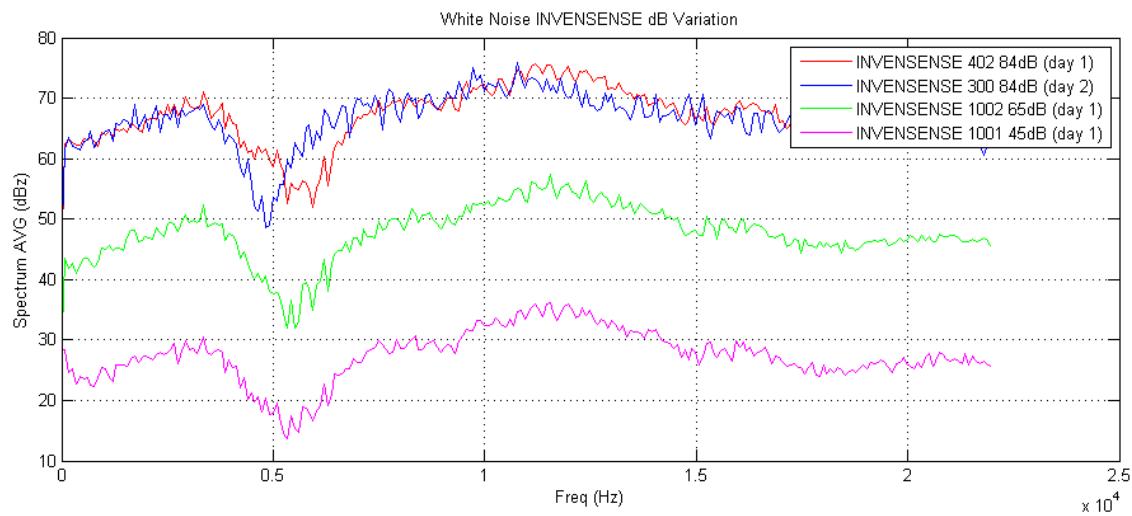


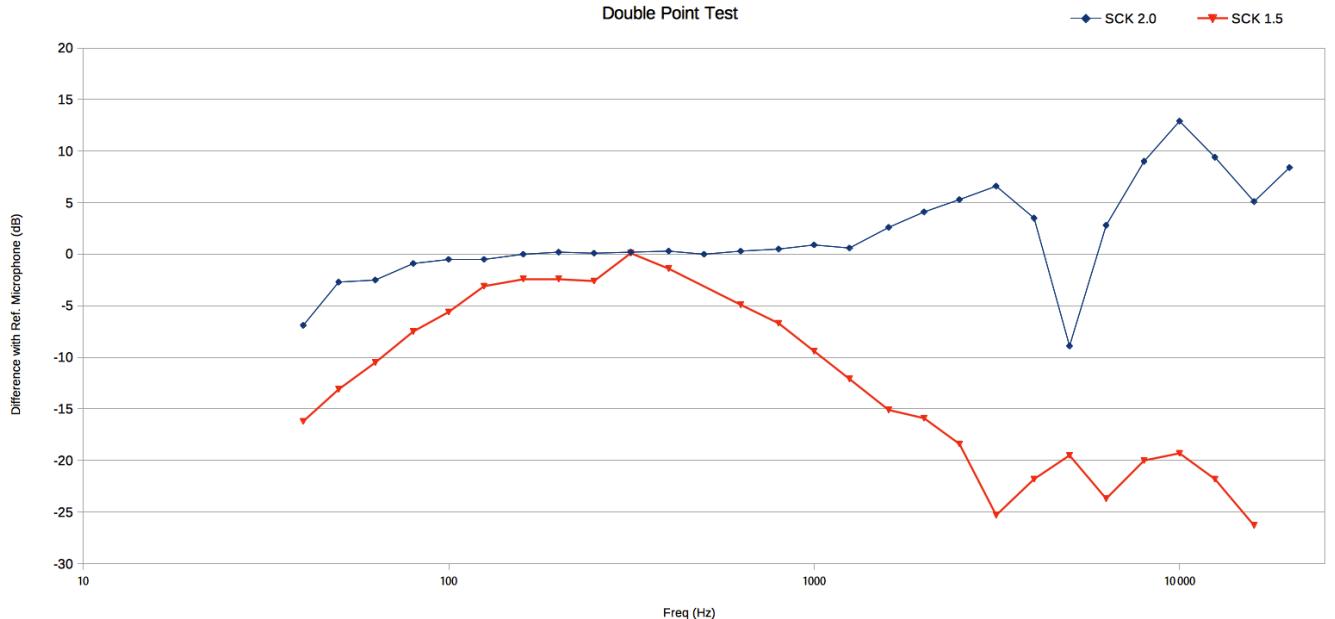
Figure 5. Typical Frequency Response (Measured)

Image credit: Invensense ICS43432

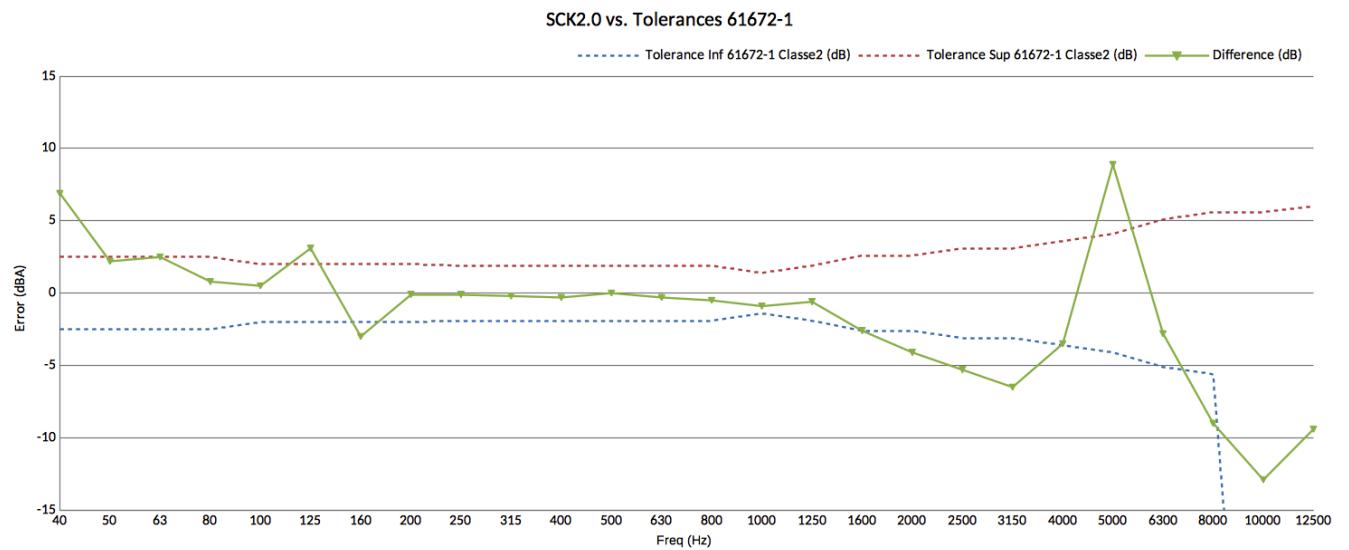
The results for this characterisation, for different SPLs are shown below:



The microphone's spectrum response is not dependent on the SPL, but only on the frequency. The above response is corrected in the Smart Citizen Kit on real time. A double point validation is performed on both microphones, from the SCK1.5 and the SCK2.0, yielding the following results:



Finally, if comparing these with the thresholds, in dBA scale IEC 61672-1, without accounting for the previous equalisation:



Which yields a very good linearity off-the-shelf over the common urban frequency range (below 2000Hz).

11.3.5 Source files

Download

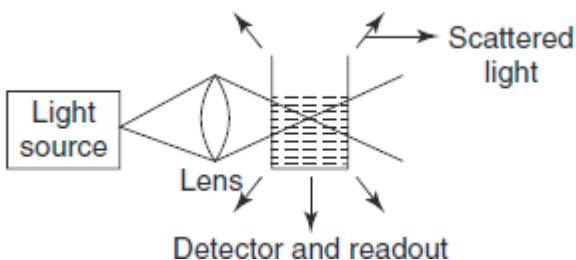
Check the source code

11.4 PM Sensors

11.4.1 Working principle



The PM sensors available in the Smart Citizen Kit (one sensor per Kit) and the Smart Citizen Station (two sensors per Station), are the Plantower PMS5003 sensor. The PMS5003 is a nephelometer, and this type of measures suspended particulates by employing a light beam and a light detector set to one side (often 90°) of the source beam. Particle density is then a function of the light reflected into the detector and the particle mass is a calculation derived from this density, assuming certain properties of the particles, such as shape, color and reflectivity, among others.



What the sensor does, is to analyse the readings from the sensing element and count how many particles are there, for different particles sizes, or bins. This means that the sensor will group, for instance, the particles that have a diameter between 1um and 2.5um in one bucket, and count them. Once it has the **particle number** calculated for all the buckets, it **estimates the Particle Mass** for PM1 (particles with a diameter below 1um), PM2.5 (particles with a diameter below 2.5um)

and PM10 (particles with a diameter below 10um). For estimating this, it makes quite a few assumptions (the internal calculations are unknown to us), such as:

- Particle shape (normally a sphere, but with some shape factors)
- Particle color, and hence reflectivity index
- Particle composition, and hence density

The performance of the sensor

We have been part of a study in which we characterised a few low cost sensors. You can check it in [here](#)

11.4.2 Sensor considerations

Sources

Have a read to the Datasheet

These sensors are used in some other projects, such as Purple Air and have been evaluated in laboratory by the Finnish Meteorological Institute - FMI and in outdoor conditions the South Coast AQMD (Air Quality Management District), USA. The study by the FMI did not yield good results for this sensor (specially in PM10), but given the cost we still think is a good citizen awareness sensor and that can be used for certain studies. The AQMD study shows better results for PM10 and PM2.5 with high correlation results with respect to reference equipment ($R^2 > 0.9$ in most cases), although we are not aware of actual testing conditions, or the reference equipment calibration. Other authors also show good results and recommend the usage of these sensors, although in some measurement conditions (like specific types of particles) they perform better, which makes sense given the assumptions mentioned above (read the academic article [here](#)). Similar sensors are used in the Luftdaten project (with a SDS011 in this case).

Relative humidity affects this type of sensor, since particles can absorb water and grow in size, hence modifying the fractions and the calculated mass. Additionally, particle's chemistry can affect these assumed properties, and these assumptions may not be usable in every type of environment. However, a relative humidity correction is being tested, correcting size distribution based on particle higroscopicity.

Dusty environments

The sensor might get clogged in a very dusty environment (like a workshop) and might need some periodic cleaning. It is safe to use a vaccum cleaner to do so, but be careful not to damage the light sensor, the laser emitter or the fan during the process.

Powering the sensor

Make sure that you power the Smart Citizen Kit with a *good enough USB cable* and with an adaptor that can provide at least 1A. We have found some issues when powering the sensor with a thin cable, or from a weak power source, like a screen.

11.5 About CO-NO₂ Metal Oxide Sensors

The SGX Mics is a Metal Oxide Resistive sensor capable of reacting to different substances in the atmosphere. In a simplified way, it is comprised of two main elements:

- A **SnO₂ substrate** that acts as a sensor element
- A **heater element** to keep the substrate in an optimal working area

The SnO₂ is a chemically sensitive **metal oxide** which has interactions with molecules to be detected in the target gas. The reactions that can occur on SnO₂ surface are **adsorption and catalytic reactions**, which basically mean that the gas molecules can be adsorbed onto the surface or can catalyse reactions (trigger or enhance them). They take place at the so called **active sites** or grain boundaries, which are areas where the grains that constitute the sensor resistance are in contact with the air (e.g. with metallic contacts). Hence, metal oxide substrate is basically a collection of sites at which different molecules can be absorbed and therefore interact in various manners with the species present in the atmosphere: either through catalytic reaction, surface reaction, grain boundary reaction (*among others*). ².

The sensor element is typically heated to a few hundred degrees (°C) using a small **resistive heater**. The regions within the sensor can be described as in Peterson et al. ¹: **the surface, which interacts with the gas, the bulk, which is unaffected by it, and the particle boundary, which lies in between these two**. The particle boundary is situated at a distance from any material exposed to the atmosphere into the sensor that chemical electrostatic effects can propagate (the so called Debye length), and this is related to the material's physical properties. At high temperatures, oxygen atoms bond onto the boundary, extracting electrons in the process from the semiconductor's surface region. The oxygen either then directly reacts with ambient gases, or these gases also bond onto the sensor, which causes more charge carriers to be withdrawn or injected into the surface region. All these effects change the sensor resistance and it is measured accordingly in ¹:

In the case of the SGX 4514, the detection of the pollution gases is achieved by measuring the sensing resistance of both sensors. In a generic way, we could characterise the sensor resistance as follows:

- RED sensor resistance **decreases** in the presence of CO and hydrocarbons.
- OX sensor resistance **increases** in the presence of NO₂.

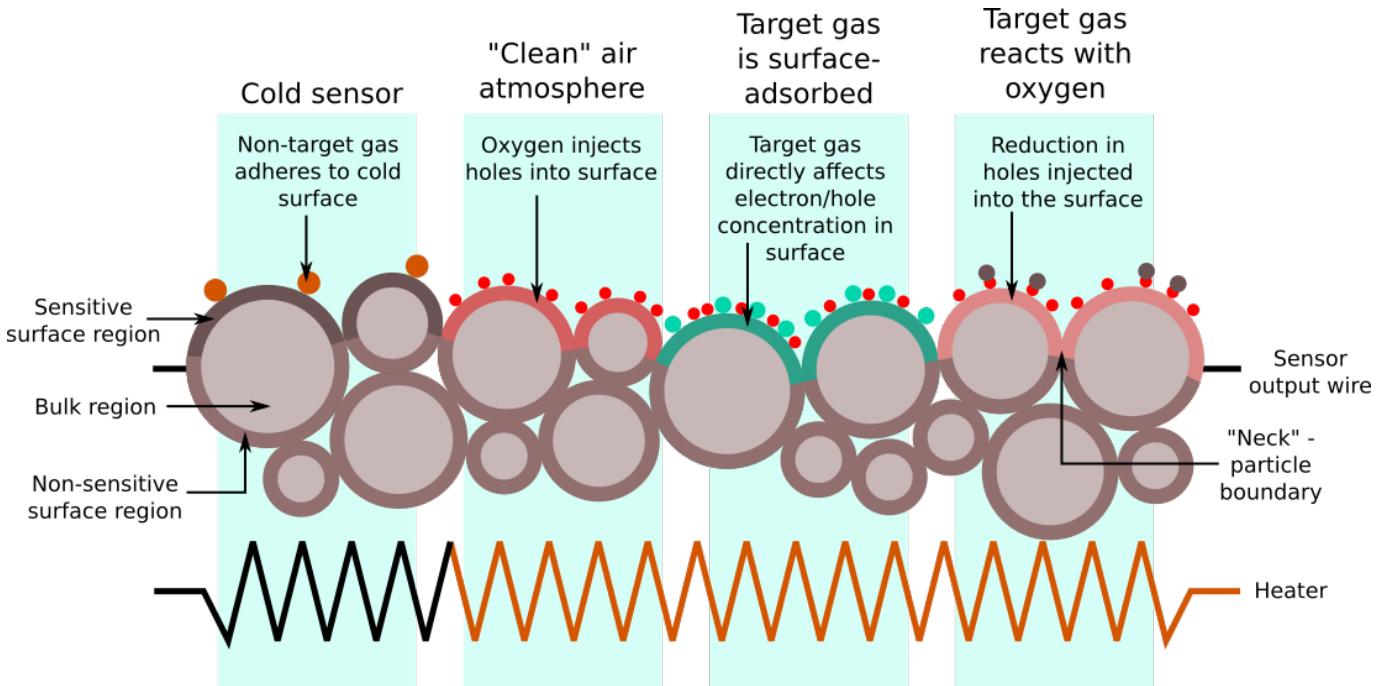
Finally, the chemical reactions within the resistive element are directly related to temperature and follow an Arrhenius equation type of behaviour. Each sensor's type has a different optimal operation temperature, which is translated into different heating powers for the heater element. Depending on the heating power and transition speeds, different reactions can be facilitated and this can lead to positive effects such as sensor clean up or battery compsuption savings, for example, when heated up in a pulsed profile. On the other hand, it can facilitate sensor poisoning or ageing, which highlights the need of proper sensor characterisation.

11.5.1 Sensor Calibration

The SGX4514 is a low cost sensor originally ment to detect instances or trends of target gas in the atmosphere ⁴⁵. The applications intended for these sensors are 'event sensing' applications and the level of accuracy required is not necessarily within regulatory standards. Furthermore, these sensors should not be used with safety related issues.

However, despite the low cost nature of these sensors, they have been subject of a great deal of research¹²³ and have been reported to give considerably good results in field applications. Before delving into the details of sensor calibration, we will try to understand what these sensors are and how they should be handled. Some important definitions are:

- **Sensor baseline resistance:** is the resistance that the sensor exhibits when it's not powered
- **Sensor sensitivity:** is the resistance variation with variations in the target gas
- **Sensor cross-sensitivity:** is the resistance variation with variations of gases other than the target gas
- **Sensor poisoning:** an irreversible resistance variation provoked by the reaction of gases other than the target gas



Source: Peterson et al.¹

Peterson et al.¹ describes the various types of interactions between atmospheric gases and a MOS sensor surface. In the image above, the leftmost region describes the unpowered behaviour, or **base resistance**. The three other regions of the diagram describe different processes that actually occur simultaneously to varying degrees. The sensor's output is the resistance across the whole of the sensor material, which forms a resistor network with contributions from both the bulk and surface regions. The model described in¹ also explains the wide variation in base resistance between individual sensors of the same type, as the random nature of the surface geometry means an equally random network of resistances. This diagram is a two-dimensional representation of a three-dimensional material; in an actual sensor, the sensitive region is spread into the surface with a distance dependent on the grain size and arrangement resulting from the sintering.

Each sensor will then have a different resistance in air and how much this baseline resistance changes with the concentration of the target gas will also differ (what we defined above as sensitivity). Therefore to convert from resistance readings to concentration it is necessary to derive a **calibration curve for each sensor**. This will require measuring the resistances in air and at a number of gas concentrations over the desired range. It is important that the concentrations are in a background of air as Oxygen is needed for the sensor to work correctly. As stated in², the sensor's response is only partially a function of the amount of gas to which the surface is exposed. Instead, the sensor will have a baseline resistance that is related to the bulk and particle boundary resistance. Because of the random geometry of the granular sensor surface, the baseline resistance will vary between individual sensors.

The change in resistance with the change in gas concentration is generally not a linear response. The response can be measured and fitted to a **polynomial relationship**, with interactions from other metrics such as temperature, humidity and

other gases. It has been proved that air flow around the sensor yields better sensor reactivity, and that the usage of PTFE filters also helps reducing cross-sensitivity and sensor poisoning. An important practical consideration with any in situ air quality sensor design is ensuring adequate flow of sampling air through the device. **Stale air inside a casing will produce unrepresentative results**, and even sensors mounted outside a casing might not get a properly-mixed sample.

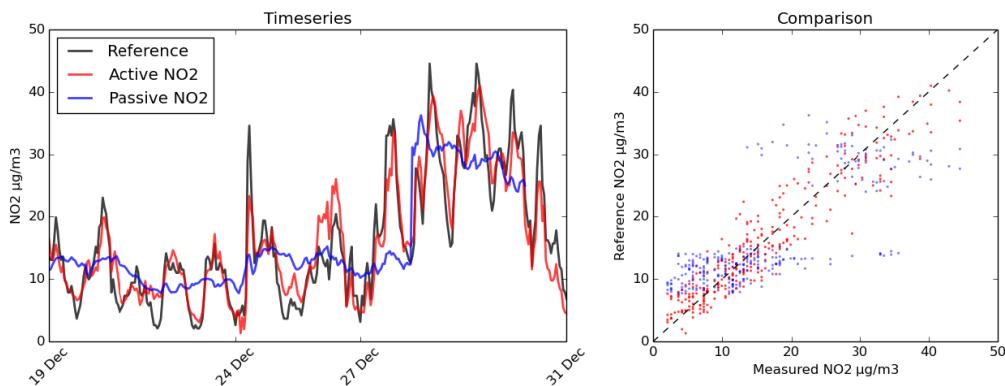


Figure 17. The effectiveness of aspiration (red) in comparison with the effectiveness of passive airflow for a sensor exposed to the elements (blue). The AURN (black) was used as a reference for actual NO₂ concentrations. On the left is a time series; on the right is a scatter plot of the same data demonstrating the diminished sensitivity with passive airflow.

Source: Peterson et al.¹

Although the deployment of multiple different sensors can compensate for the cross-sensitivity issues in calibration, it cannot eliminate it. MOS sensors can thus be used only in situations where any interfering species can either be measured by other means, or they must be calibrated regularly and used in locations where the background varies in concentration slowly compared with the target gases. As well, the **sensor drift** over time is an important issue that requires sensor recalibration over time.

There are two major factors in the longevity of a sensor's calibration. The first is the **natural degradation** of the heater element, which becomes hotter over prolonged use and causes the sensor's response profile to vary. The second is the effect of slowly-varying interfering gases, which over the course of months shifts the sensor's baseline. The first problem may have an engineering solution, but the second will involve taking the results of the tests in an artificial atmosphere, identifying the most critical species and either measuring or possibly modelling their likely concentrations during deployments.

An analytical approach to counteracting this drift might be "merging calibrations", where a sensor is calibrated at the start and end of a four-month campaign, and the coefficients gradually change from one end of the experiment run to the other.

Having all this in mind, the sensor calibration we follow is comprised of the following steps:

- Sensor behaviour characterisation under different temperature profiles
- Sensor baseline and sensitivity characterisation in controlled conditions
- Sensor deployment with reference measurements collocation and model calibration

The use of deployment campaigns is of utmost importance in order to develop sensor models that are *reality proof*. With the possibility of collecting the data with the SmartCitizen Platform and the data treatment provided by the Sensor Calibration Framework, we are able to iterate over the different sensor calibration possibilities, ranging from Ordinary Linear Regression or more advanced techniques such as ML models such as LSTMs networks.

11.5.2 Field results

In this section, we will detail some of the MOS related results obtained during the sensor validation campaigns detailed below:

- *University of Bologna*: data collected from 23/January to 13/February. The measured pollutants with reference equipments were CO, NO₂, NO, NOx and O₃. Two prototype Smart Citizen Stations were deployed in two different sites, with two Smart Citizen Kits.
- *University College Dublin*: data collected from 27/March to 17/April. The measured pollutants with reference equipments were NO, NO₂ and NOX. One prototype Smart Citizen Station was deployed with two Smart Citizen Kits

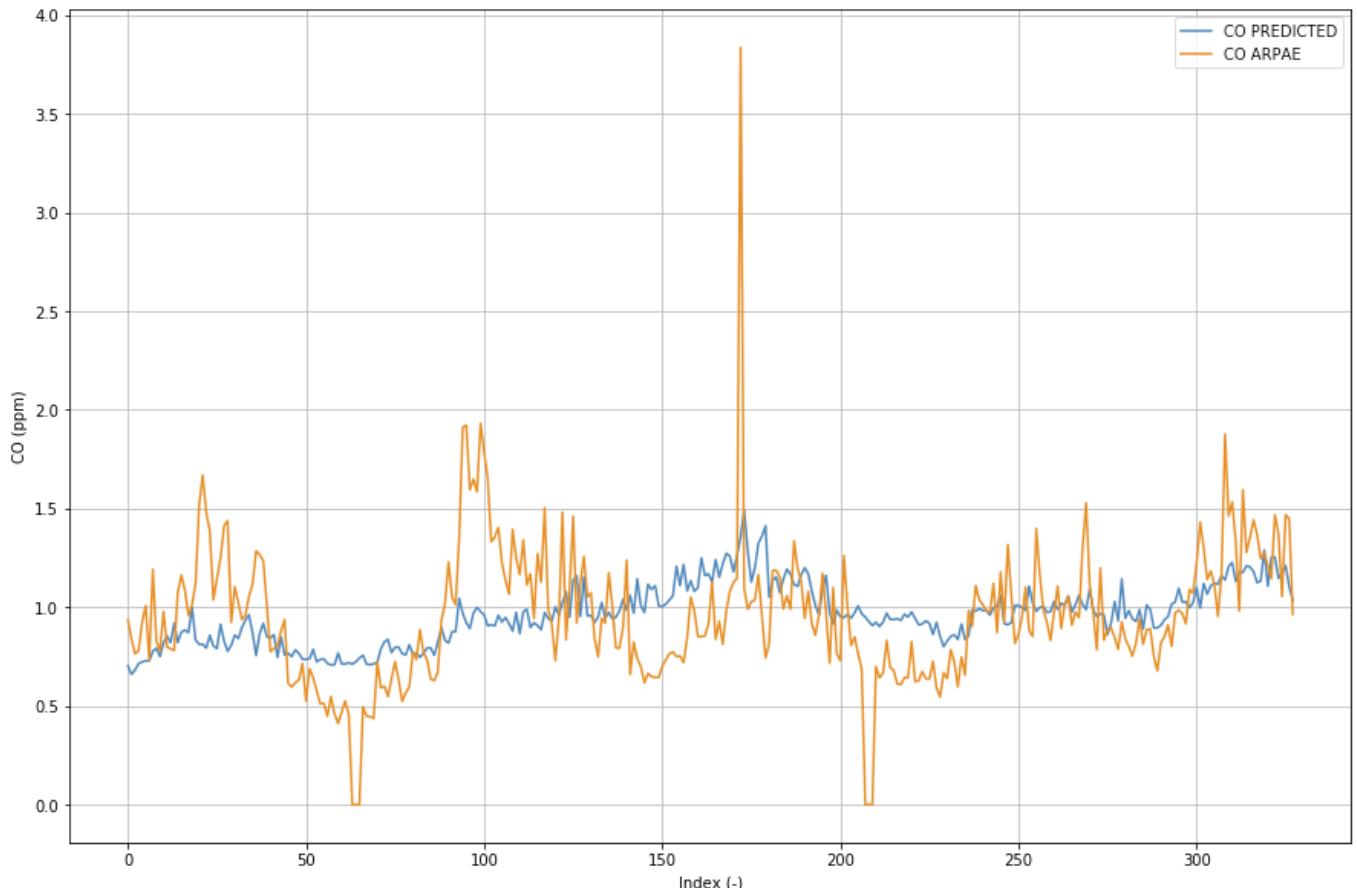
For both results shown below, we used an LSTM with 200 epochs training and the following structure:

```
from keras.models import Sequential
from keras.layers import Dense, Activation, LSTM, Dropout

model = Sequential()
layers = [50, 100, 1]
model.add(LSTM(layers[0], return_sequences=True, input_shape=(train_X.shape[1], train_X.shape[2])))
model.add(Dropout(0.2))
model.add(LSTM(layers[1], return_sequences=False))
model.add(Dropout(0.2))
model.add(Dense(output_dim=layers[2]))
model.add(Activation("linear"))
model.compile(loss='mse', optimizer='rmsprop')
```

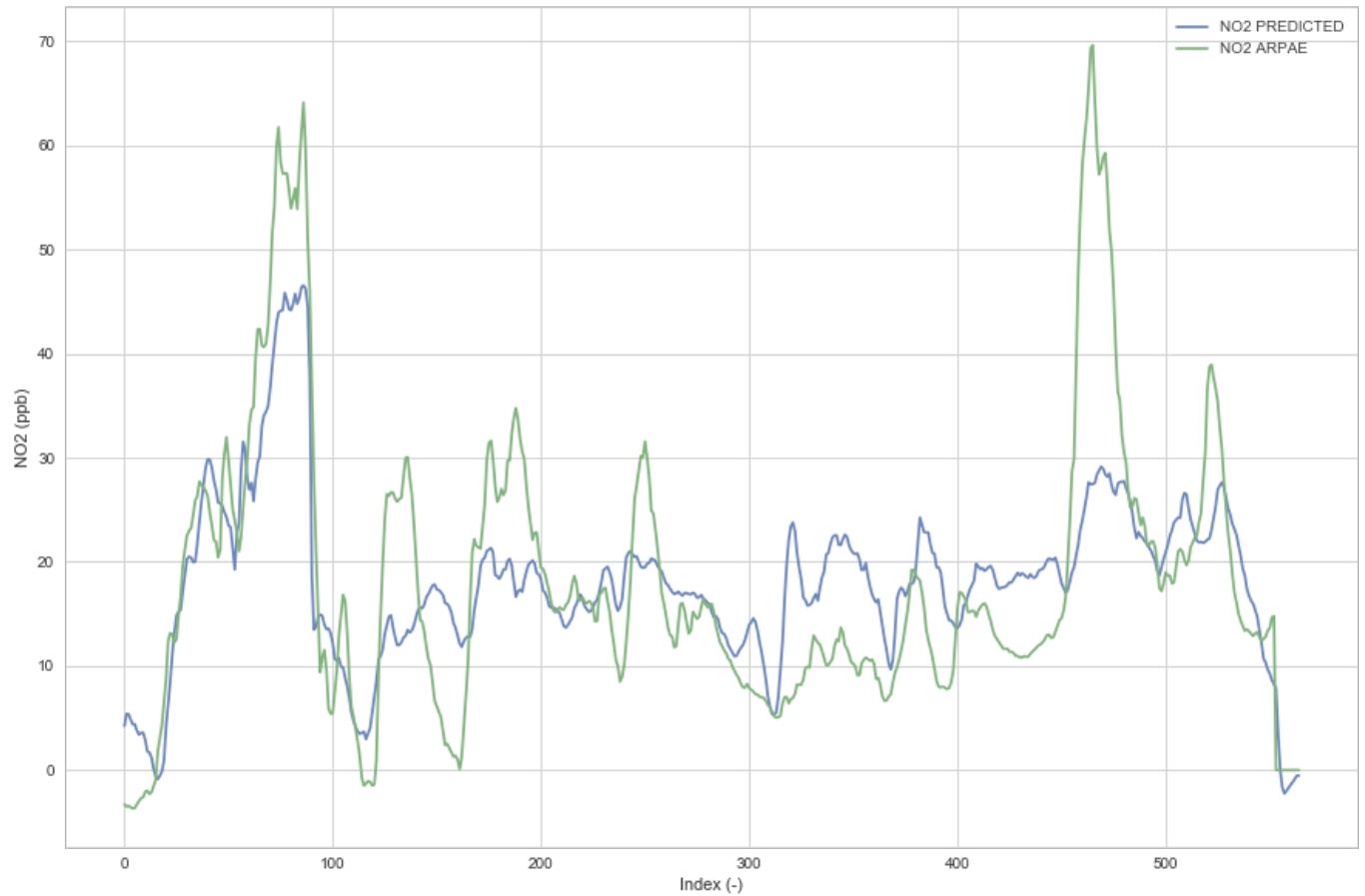
Carbon Monoxide

The CO model included the following features: CO_{R}^{-1}, CO_R^{-2}, Temp and Temperature^2. The results can be seen below:



Nitrogen Dioxide

The NO₂ model included the following features: NO₂~{R}, NO₂~{R}^{-2}, Light, Temp and Temperature^2. The results can be seen below:



Warning

This test campaign contains a short amount of data to be used as a training dataset for a LSTM algorithm. Therefore, this is just to be considered as an use case example and further tests and data should be carried out to train broader models.

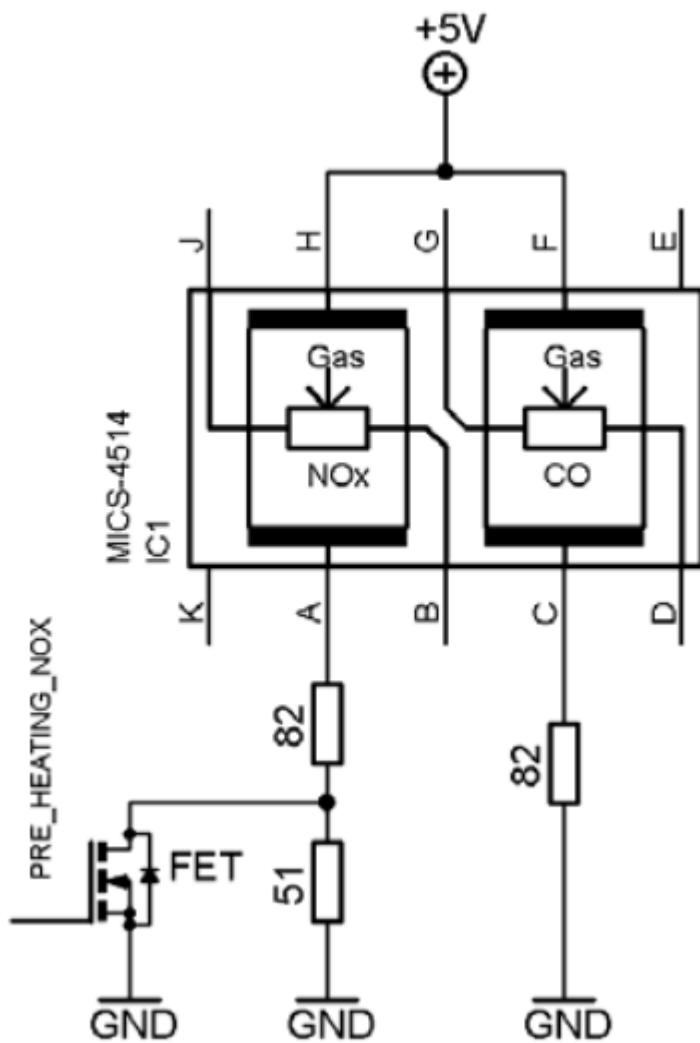
11.5.3 Metal Oxide Sensors Implementation

Heating stage

The solution present at Urban Sensor Board V2.0 for MICS-4514 sensor's heaters excitation, pretends to make it compatible with a 3.3V global voltage source.

OPERATING CONDITIONS (RED Sensor/OX Sensor)

Parameter	Symbol	Typ	Min	Max	Unit
Heating power	P _H	76/43	71/30	81/50	mW
Heating voltage	V _H	2.4/1.7	-	-	V
Heating current	I _H	32/26	-	-	mA
Heating resistance at nominal power	R _H	74/66	66/59	82/73	Ω



The manufacturer recommend the following circuit topology, with a global supply voltage of 5V. In the datasheet are collected the electrical nominal conditions for that resistors, in order to operate safely with the heater, without damaging it.

Besides that, several other possible conditions could also damage early the heater resistors, like the fact of consider a pure PWM signal, with source 5V and subsequent duty cycle, as excitation. Even if the frequency is relatively high (100kHz), the resistors are forced to operate briefly with 5V, and this accelerates the destruction of this part of the MICS sensor.

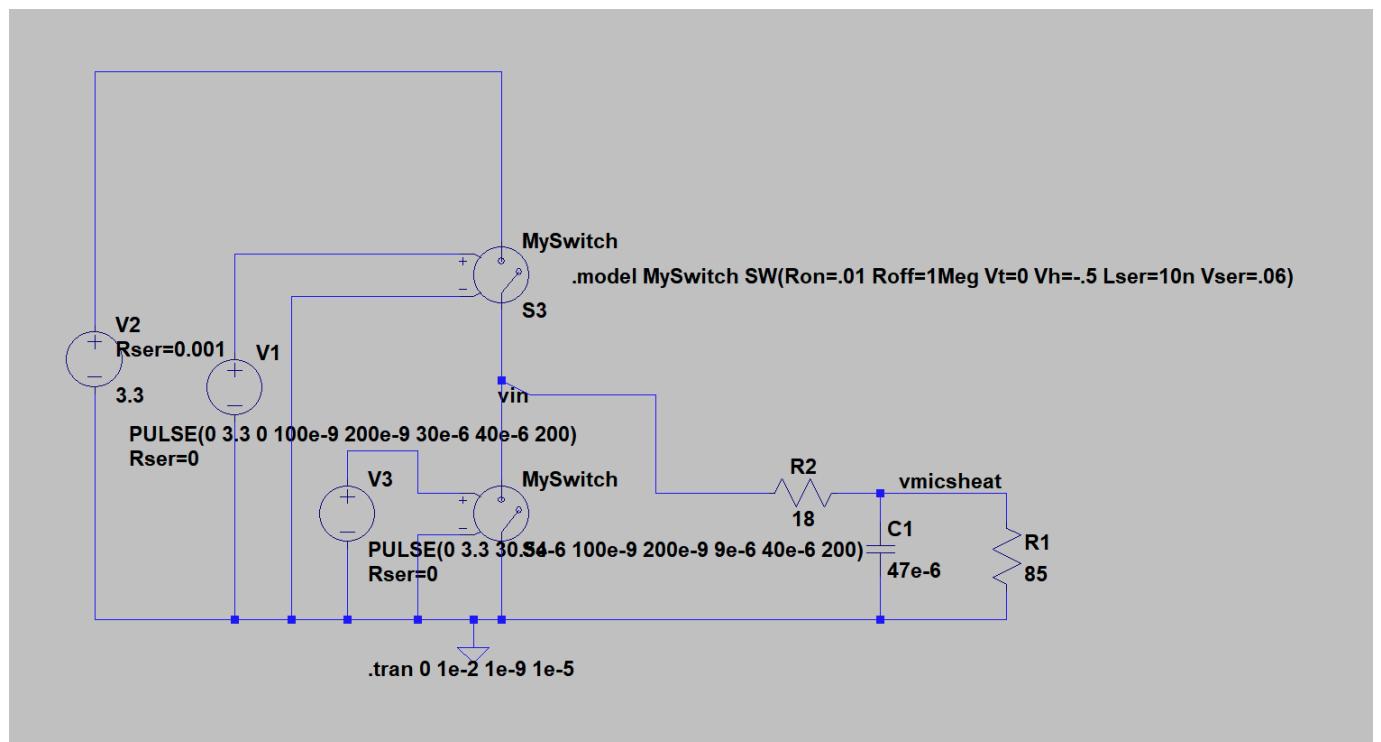
So it's possible to provide the nominal voltages for heater resistors from a 3.3V source, if we replace the auxiliar resistors (from recommended topology) with corresponding values, to preserve the total power dissipated and current at same normal operating conditions.

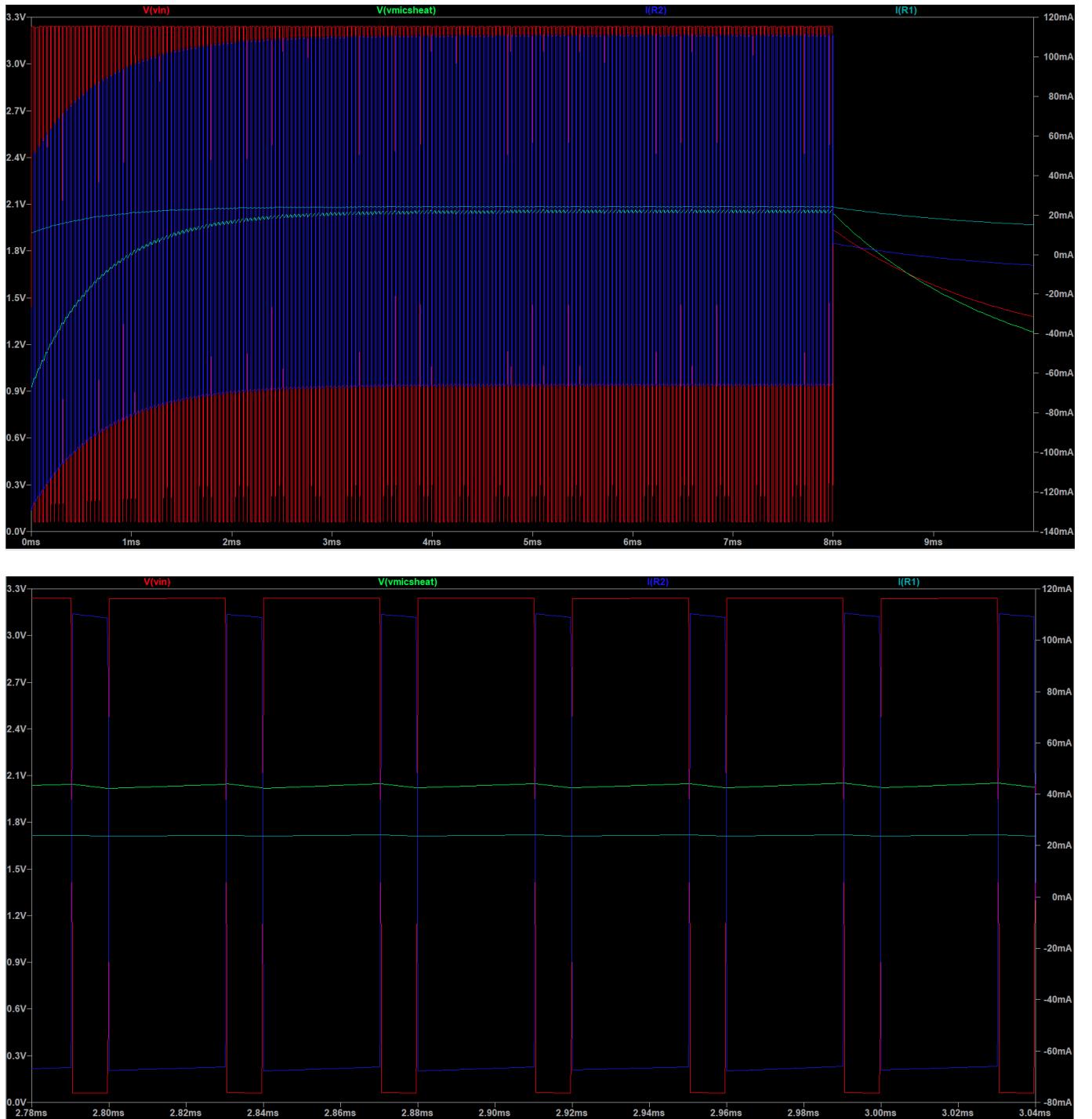
Even more, we can upgrade the function of the auxiliar resistors adding a capacitor to form a passive RC filter. In the DC or continuous operation, the capacitor is fully charged and the current is limited by the auxiliar resistor. In AC or pulsed operation, the capacitor can be selected to remove this AC component, and feed the heater resistor with a nearly constant voltage or at least with small variations (<1%).

The source for the PWM signal must be buffered, because the resistive load of the system demands currents above the SAMD21 can supply. For this purpose, the solution selected is to use a digital hex-inverter buffer, which can drive up to 32mA with each output pin, which we can parallelize to operate under proper safety factor for the buffer.

Simulations

The first simulations and given values lead to the selection of the RC components values if we set a PWM frequency around 40 kHz.

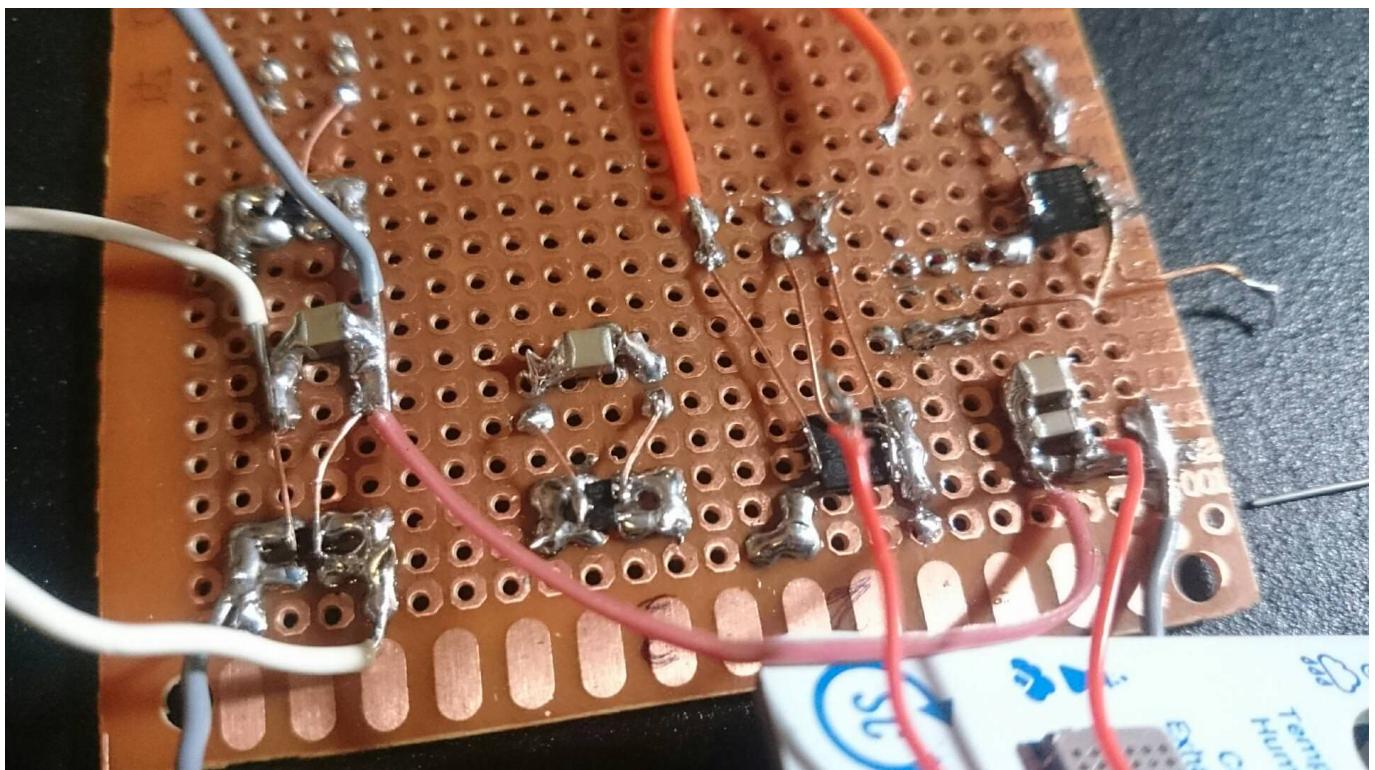
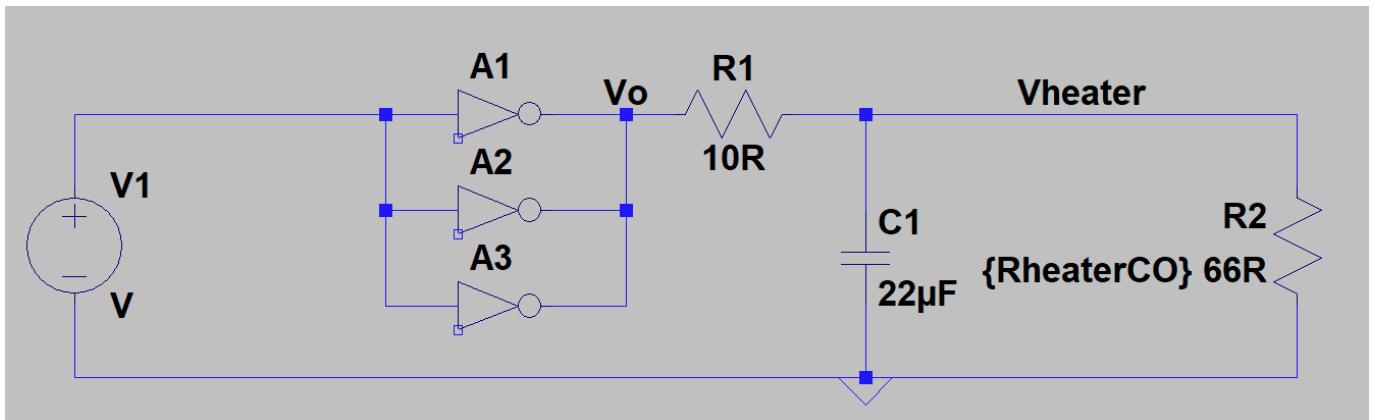




To evaluate the R part of the filter, is needed to take into account the output resistance of the hex-inverter buffer.

Prototypes

We build the circuit into a protoboard, with several IC HEX-INV manufacturers, based on the following schematic:



The measures are summarized in the following table, in which we compare four pre-selected devices, which can fit in our application for size and price considerations.

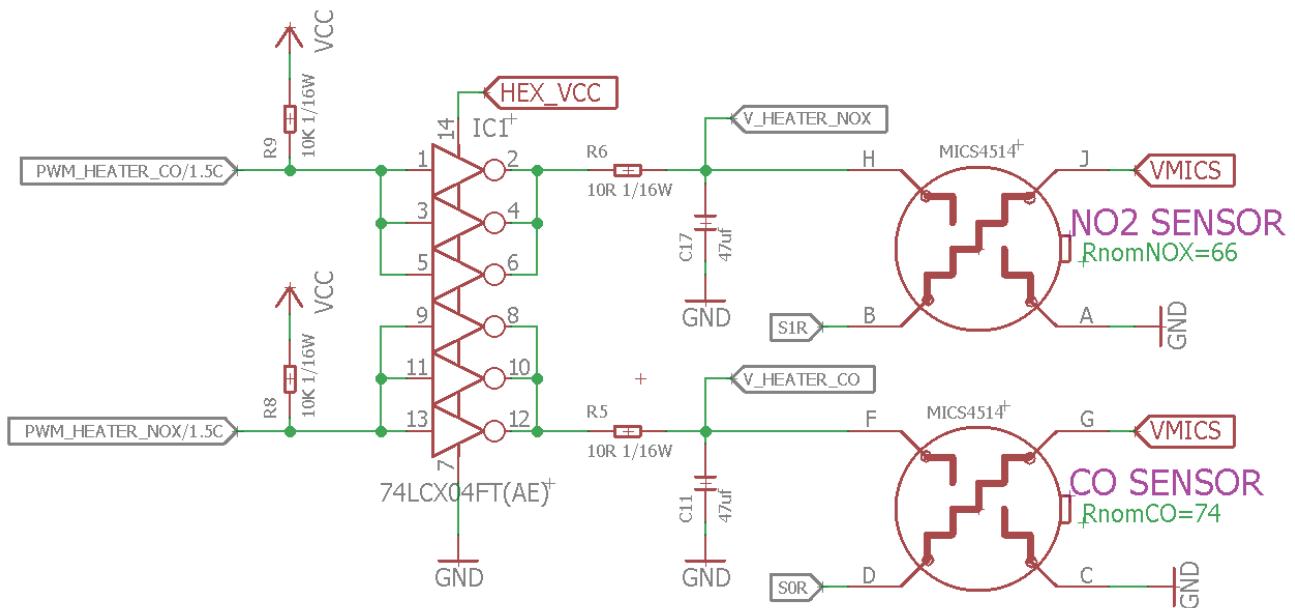
			Vout_H	Vout_L	Vheater_avg	Vheater_rizado	Io [mA]	RintH	RintL
74LCX04FT(AE)	Load: 70R	DC	3,12	0			44,5714	4,04	
	Load: 10R + RheaterCO ~60	DC	3,15	0	2,85-		41,4474		
	(66R) + 22uF	~30	2,8	0,45	1,72	0,02	26,0606		
			2,5	0,22	0,84	0,02	12,7273		
MC74LCX04DTG	Load: 70R	DC	3,15	0			45,0000	3,33	
	Load: 10R + RheaterCO ~60	DC	3,15	0	2,81-		41,4474		
	(66R) + 22uF	~30	2,85	0,55	1,785	0,02	27,0455		
			2,55	0,25	0,9	0,02	13,6364		
SN74LVC2G04DBVR	Load: 70R	DC	3,075				43,9286	5,12	
	Load: 10R + RheaterCO ~60	DC	3,075	0	2,775-		40,4605		
	(66R) + 22uF	~30	2,7	0,6	1,722	0,018	26,0909		
			2,25	0,28	0,837	0,018	12,6818		
NC7WZ14P6X & NC7WZ16P6X	Load: 70R	DC	3,1	0			44,2857	4,52	
	Load: 10R + RheaterCO ~60	DC	3,14	0	2,8-		41,3158		
	(66R) + 22uF	~30	2,75	0,7	1,817	0,018	27,5303		
			2,45	0,32	0,916	0,022	13,8788		

Four cases with parallelized inverters, for each device were performed: passive load 70R test with DC input, and three tests with 10R+Rheater load at DC input, 60% duty cycle and 30% duty cycle. The 74LCX04FT(AE) was selected because it has the lowest LOW output level (0.45V, 0.22V), which is considered here as the quality (or close to ideal) of the square wave input source.

Final implementation

The solution implemented in the PCB, has a constant auxilar R (10R+Rout_buff), and constant C (47uF), and also operates at constant frequency, then, the output power regulation is based on the PWM's duty cycle. The following circuit represent the implemented schematic.

MICS4514 INTERFACE



Operation

First of all, is needed to know the real implemented R heater of each sensor (which may vary among devices and time), and can be estimated by measuring the V_heater_* at 100% duty cycle, then:

$$R_{heater} = \frac{(R_{int_buff} + 10R) \cdot V_{heater}}{3.3V - V_{heater}}$$

Where Rint_buff can be approximated with 4 Ohm resistor.

The desired_reference_voltage is function of the desired_power_Rheater and duty cycle. If we set 80mW we can use the value of the R heater to obtain desired_reference_voltage through this formula:

$$V_{desiredRef} = \sqrt{(P_{desired} \cdot R_{heater})}$$

(Take into account this resistor has a drift over time, therefore is recommended to take periodic measurements of the value of R heater itself, and check the output power reachability).

With selected parameters, after 2ms of PWM operation, the RC reaches the permanent, and then is recommended to take measurements of V_HEATER_. The loop can be closed to determine the duty cycle as function of the difference (desired_reference_voltage - V_HEATER_ (averaged)).

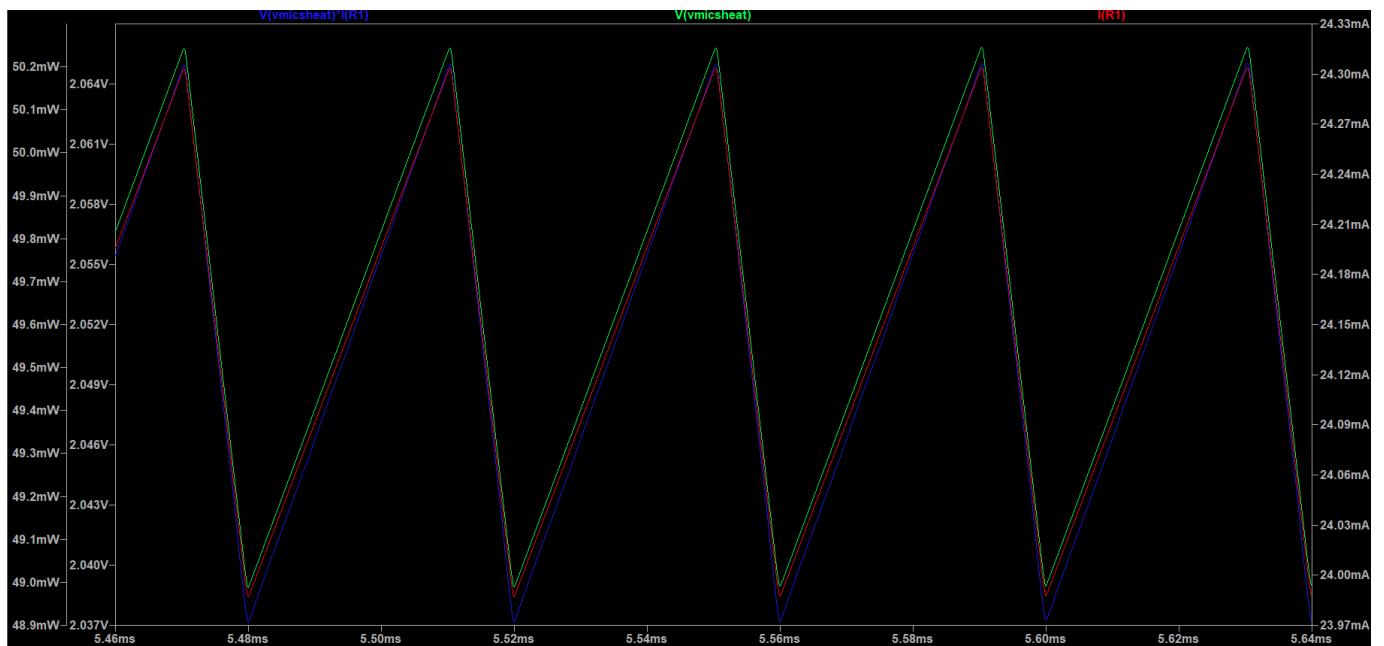
Is recommended to average several samples to remove the AC part of the signal. The measured DC signal has a noise of ±20mV peak to peak (with triangular distribution).

The sign of the PWM signal is inverted due to the action of the inverter, then, a desired x% duty is obtained as 100%-x%.

$$DutyCycle_{Set} = 100 - DutyCycle_{Desired}$$

As initial PWM approximation to begin to converge close to the regulated duty cycle can be obtained through this simplification:

$$DutyCycle_{InitialSet} = \left(1 - \frac{V_{heater}}{3.3V}\right) \cdot 100$$



References

1. Practical Use of Metal Oxide Semiconductor Gas Sensors for Measuring Nitrogen Dioxide and Ozone in Urban Environments 
2. Modelling Of Water Adsorption On SnO₂ Surface 
3. MICS-4514 Datasheet 
4. Frequently-Asked-Questions-for-MiCS-Gas-Sensors 
5. SGX Metal Oxide Gas Sensors - How to use and how they perform 
6. Sensors 2017, 17, 1653 

12. Guides

12.1 Analyse your data in batch

Sometimes we have a lot of devices to process, and the interfaces for analysis in smartcitizen.me or in the `jupyter notebook` might not be the most efficient way to do it. For this reason, we have developed a functionality to process the data in batches, defining the tasks to perform in a `json` descriptor file.

The descriptor file can be placed in the `src/tasks` directory (or any, actually). An example of how to run it is shown in `examples/batch_analysis.ipynb`:

```
# Load the object
from src.models.batch import batch_analysis",
tasks_file = '../tasks/batch.json"',
batch_session = batch_analysis(tasks_file, verbose = True)

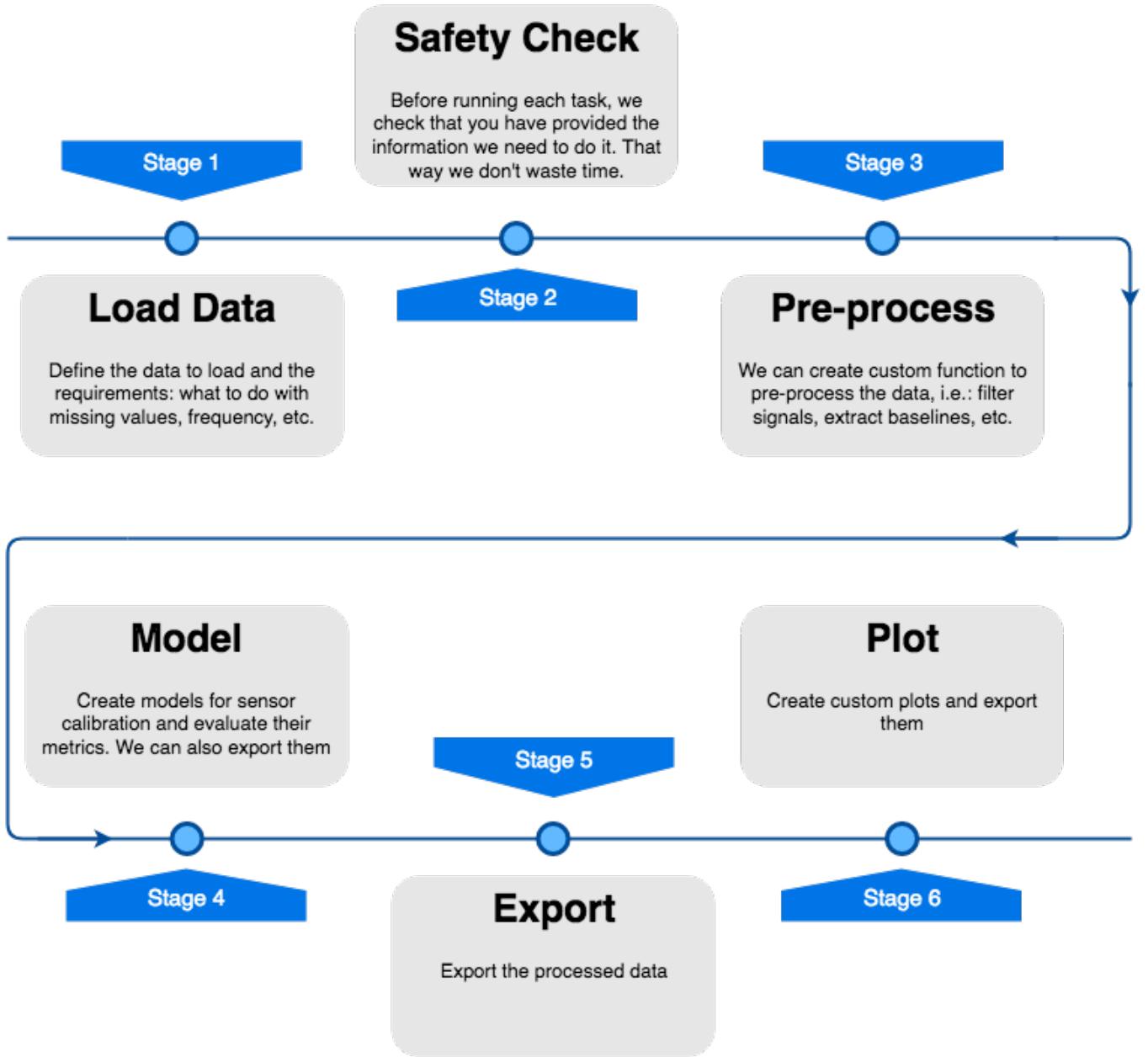
# Run the analysis
batch_session.run()
```

12.1.1 Functionality

These tasks are intended to automatise data analysis tasks as the following:

- Load, process and export processed data
- Generate models and apply them, extracting metrics and comparing if they extrapolate to different set of sensors in different datasets, without having to run extra code
- Make plots for different metrics in an automatic way, and export their renders

12.1.2 Workflow



12.1.3 Json task description

The descriptor file loaded is a `.json` containing keys for each task to be run. Several tasks can be included and will be run consecutively:

```
{
  "TASK_1": {...},
  "TASK_2": {...},
  "TASK_3": {...}
}
```

Each of the `tasks` contains different fields, depending on what the process is.

Define data

In case **no model needs to be calculated**, the data can be specified directly in the task:

```
{
  "TASK_1": {
    "data": {
      "datasets": {
        "TEST_1": ["DEVICE_11", "DEVICE_12"],
        "TEST_2": ["DEVICE_21", "DEVICE_22", "DEVICE_23"],
        ...
      },
      "data_options": {"avoid_processed": true,
                      "frequency": "1Min",
                      "clean_na": true,
                      "use_cache": true,
                      "clean_na_method": "drop",
                      "min_date": null,
                      "max_date": null,
                      "export_data": null,
                      "rename_export_data": false}
    }
  }
}
```

If a model is to be calculated, the data is defined within the model key as seen below.

DATA LOADING OPTIONS

- `frequency` : frequency at which load the data (as defined in `pandas` here)
- `clean_na` : clean or not `NaN`
- `clean_na_method` : `drop` or `fill` with back-forward filling
- `use_cache` : whether or not to use file chaching for the analysis. This adds a `cached` folder in the corresponding `test` directory, which allows faster download from the `API`. It is implemented so that the only data to be downloaded is the one that is not cached
- `min_date` , `max_date` : for limitting the amount of data to be loaded
- `export_data` : if the processed data (after pre-processing and modeling) has to be exported. It will be saved in the corresponding `test` folder, under `processed` . Options are:
 - `None` : don't export anything
 - `All` : all channels in the `pandas` `dataframe`
 - `Only Generic` : Export only channels that are under the `data/interim/sensorNamesExport.json`
 - `Only Processed` : Export only channels that are tagged as `processed` under the `data/interim/sensorNamesExport.json`
- `rename_export_data` : Rename the exported channels for better readability using the file `data/interim/sensorNamesExport.json`

None?

In `json`, we specify the `python None` as `null`.

Want to save time?

Enable `use_cache` and we will save some time by checking if the data we have downloaded previously from the API can be used. This also applies for pre-processing functions.

Model

Recommended to just get in touch

This can be overwhelming at first. Just get in touch

In the `model` sub-task, currently three possibilities are implemented:

- Linear methods: `OLS` or `RLM` regression
- Random Forest (`RF`) or Support Vector Regressor `SVR`
- XGBoost

In the case of having a `model` task, the `data` defined above is ignored, and only the one under `model: {"data":{}}` is used.

One model per task

It is better to generate only one model per task, since the memory used by the models can be very large.

An example is shown below:

```

"model": {
    "model_name": "Random_Forest_100",
    "model_type": "RF",
    "model_target": "ALPHASENSE",
    "data": {"train": {"TEST_1": {"devices": ["DEVICE_1"],
                                "reference_device": "REF_DEVICE_1"}},
              "test": {"TEST_1": {"devices": ["DEVICE_1", "DEVICE_2"],
                                "reference_device": "REF_DEVICE_2"},
                       "TEST_2": {"devices": ["DEVICE_3"],
                                "reference_device": "REF_DEVICE_3"},
                       "TEST_3": {"devices": ["DEVICE_4"],
                                "reference_device": "REF_DEVICE_4"},
                       "TEST_2": {"devices": ["DEVICE_5"],
                                "reference_device": "REF_DEVICE_5"}},
    "features": {"REF": "NO2_REF",
                 "A": "GB_2W",
                 "B": "GB_2A",
                 "C": "HUM"
               },
    "data_options": {"export_data": "All",
                    "rename_export_data": false,
                    "target_raster": "1Min",
                    "clean_na": true,
                    "clean_na_method": "fill",
                    "min_date": null,
                    "max_date": null,
                    "use_cache": true
                  },
    "hyperparameters": {"ratio_train": 0.75,
                        "n_estimators": 100,
                        "shuffle_split": true
                      },
    "model_options": {"session_active_model": false,
                      "export_model": false,
                      "export_model_file": false,
                      "extract_metrics": true,
                      "save_plots": false,
                      "show_plots": false
                    }
  },
}

```

- `model_name` : model name to be saved
- `model_type` : 'RF', 'SVR', 'LSTM' or 'OLS'
- `model_target` : if the model is to be stored under a specific category of models under the `models/` folder
- `data` : dict containing the data to use for training, and features description. Under `train`, we define which of the tests and devices is to be used for the model definition, with the format `{"TEST": "devices": ["DEVICE"], "reference_device": "REF_DEVICE_1"}`. Under `test`, we define a series of `test` in which we'll evaluate the model extracted from the `train` dataset:
 - `devices` : list of devices to use
 - `reference_device` : device that contains the reference data

Info

Multiple training datasets are possible as well, by combining them.

Additionally: + `features` : dict of devices tagged as `REF` , `A` , `B` , `C` ... to define the features of the model, being `REF` the reference channel in the `reference_device` - `hyperparameters` : dict containing different hyperparameters, depending on the type of model: + For all: * `ratio_train`: generic, train-test split ratio + OLS: * `formula_expression` : stats models formula type, accepting `numpy` expressions This formula has to reference the features described under the `data` section. Example: `REF ~ A * B + np.log(C/2) + Random Forest`: * `n_estimators` : only for `RF` . number of forests to use * `shuffle_split`: only for `RF` . whether or not use shuffle split + LSTM: * `n_lags` : number of lags to account in the LSTM input * `epochs` : number of epochs (100 or more recommended) * `batch_size` : batch size (72 recommended) * `verbose` : verbose output during training * `loss` : loss function ('mse' or others) * `optimizer` : optimizer to use (`adam` or others) * `layers` : specific layer structure. Example below:

```
```
"layers": [{"type": "lstm",
 "neurons": 100,
 "return_seq": true
},
{
 "type": "dropout",
 "rate": 0.05
},
{
 "type": "lstm",
 "neurons": 100,
 "return_seq": true
},
{
 "type": "lstm",
 "neurons": 50,
 "return_seq": false
},
{
 "type": "dropout",
 "rate": 0.05
},
{
 "type": "dense",
 "neurons": 1,
 "activation": "linear"
}
],
```

```

- `model_options` : different options for the model calculated

- `session_active_model` : keep the model active after the task is completed
- `export_model` : export the model (parameters, hyperparameters, weights) to the `model/model_type` folder after the task is completed
- `export_model_file` : export the model file (not recommended for RF) to the same folder as above
- `export_metrics` : export metrics for the model or not
- `save_plots` : save model plots or not
- `show_plots` : show model plots or not

Plots

The plot sub-task accepts two different libraries: `matplotlib` and `plotly`. The first one generates static images, that we can export for nice quality graphs, while `plotly` is more meant for exploratory analysis.

Many plots? No problem

In the case of the models, we only wanted one model per task, but it's not the case in the plots.

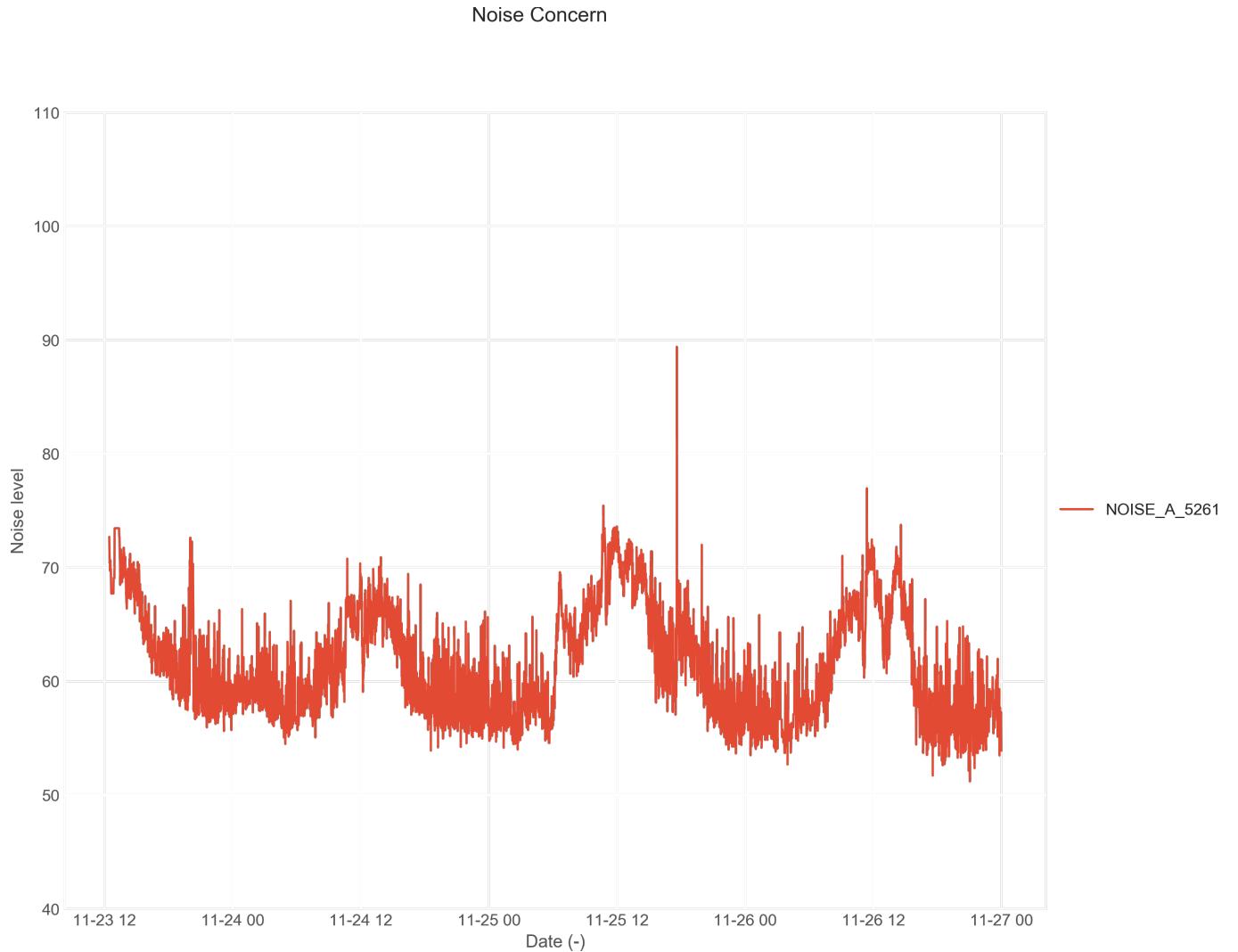
```

"plot": {
  "2": {"plot_type": "timeseries",
    "plotting_library": "matplotlib",
    "data": {"test": "TEST_1",
      "traces": {"1": {"device": "5262",
        "channel": "EXT_PM_10",
        "subplot": 1},
      "2": {"device": "5262",
        "channel": "EXT_PM_25",
        "subplot": 2}}},
    "options": {"show_plot": false,
      "separate_device_plots": false,
      "target_raster": "10Min",
      "max_date": null,
      "min_date": null,
      "export_path": "/Users/mac/Desktop",
      "file_name": "plot_pm"},
    "formatting": {" xlabel": "Date (-)",
      "ylabel": {"1": "Temp degC",
        "2": "Hum (%rh)" },
      "yrange": {"1": [0, 40],
        "2": [0, 100]},
      "title": "PM",
      "sharex": true,
      "grid": true,
      "height": 10,
      "width": 12}
  },
}

```

GENERAL DESCRIPTION

This description is suitable for timeseries plots. Check below for other types:



- `plot_type` : the plot type to be used. Currently we support `timeseries`, `violin`, `scatter_matrix`, `correlation_plot`, `heatmap`, `barplot`, `coherence_plot`
- `plotting_library`: `matplotlib` or `plotly`
- `data` : the data to represent. This data has to be loaded previously. This is only to define the plot:
 - `test` : each plot can only represent data from a single `test`
 - `traces` : numbered entries to specify the `device`, `channel` and, if applicable, the subplot (only for timeseries)

Same plot for many devices?

Set "device: 'all'" and "separate_device_plots": true , and we will make separate plots for each device.

- options :

- show_plot : whether or not to show the plot. Not recommended for large amounts
- separate_device_plots : true in case we want a single plot for each device in traces: devices . false in case we want to merge all the devices in a single plot (for comparison purposes). Applicable for timeseries
- target_raster : the frequency of the data to plot. Reduces processing time
- min_date , max_date : if not null , they crop the data with those dates
- export_path : where to put the plot. If null , we won't export anything
- file_name : how to name the plot. We will append the trace name in case separate_device_plots is set to true

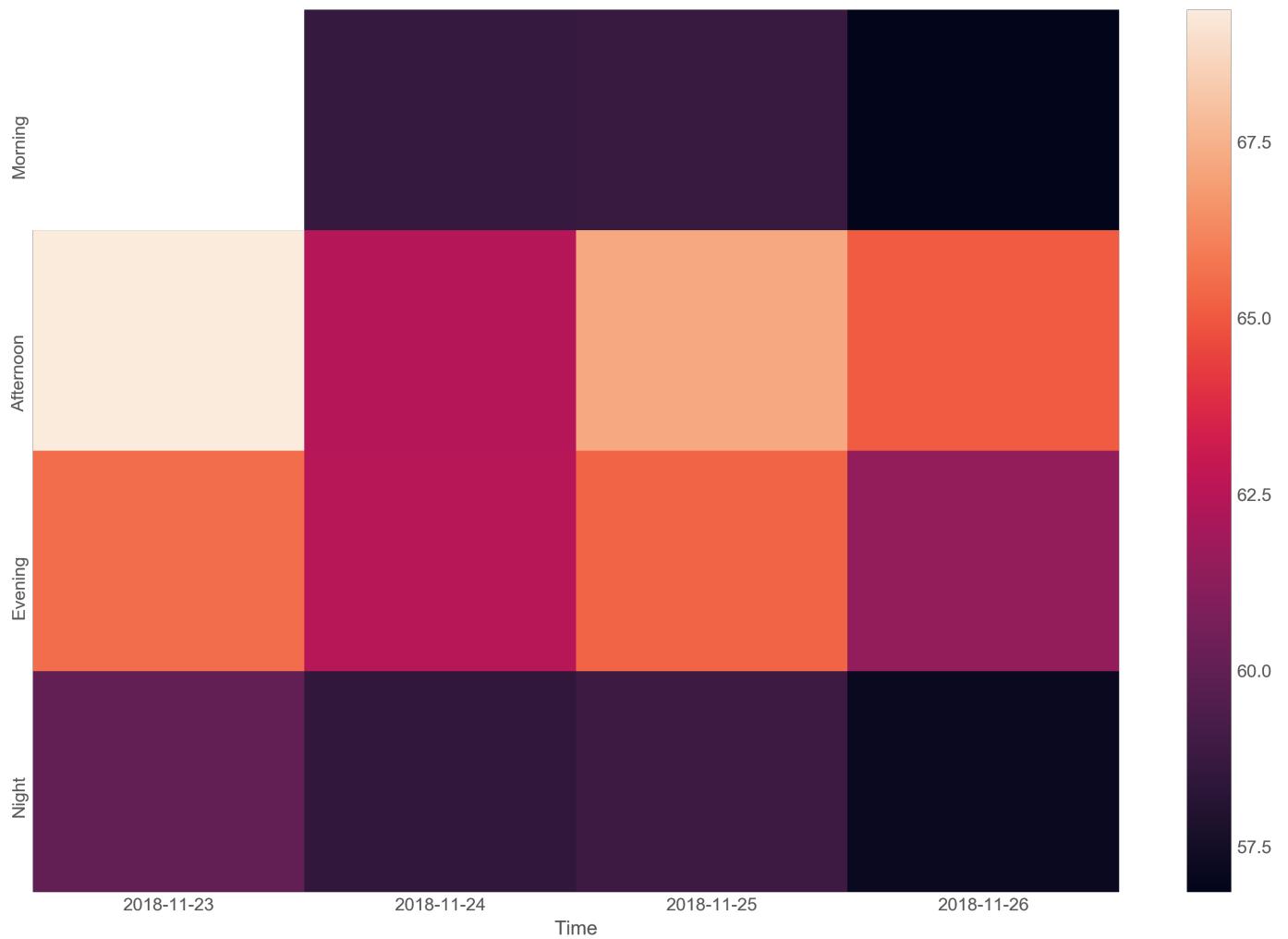
- formatting :

- xlabel : Name to tag the x axis of the plot
- ylabel : Name to tag the y axis(es) of the plot. It can be more than one value in a json style
- yrange : Range fro the y axis(es). Same as above
- title : plot title
- grid : to show the grid or not
- height , width : plot height and width

PLOTS SPECIFICS

Heatmap

Magnificent plot



```

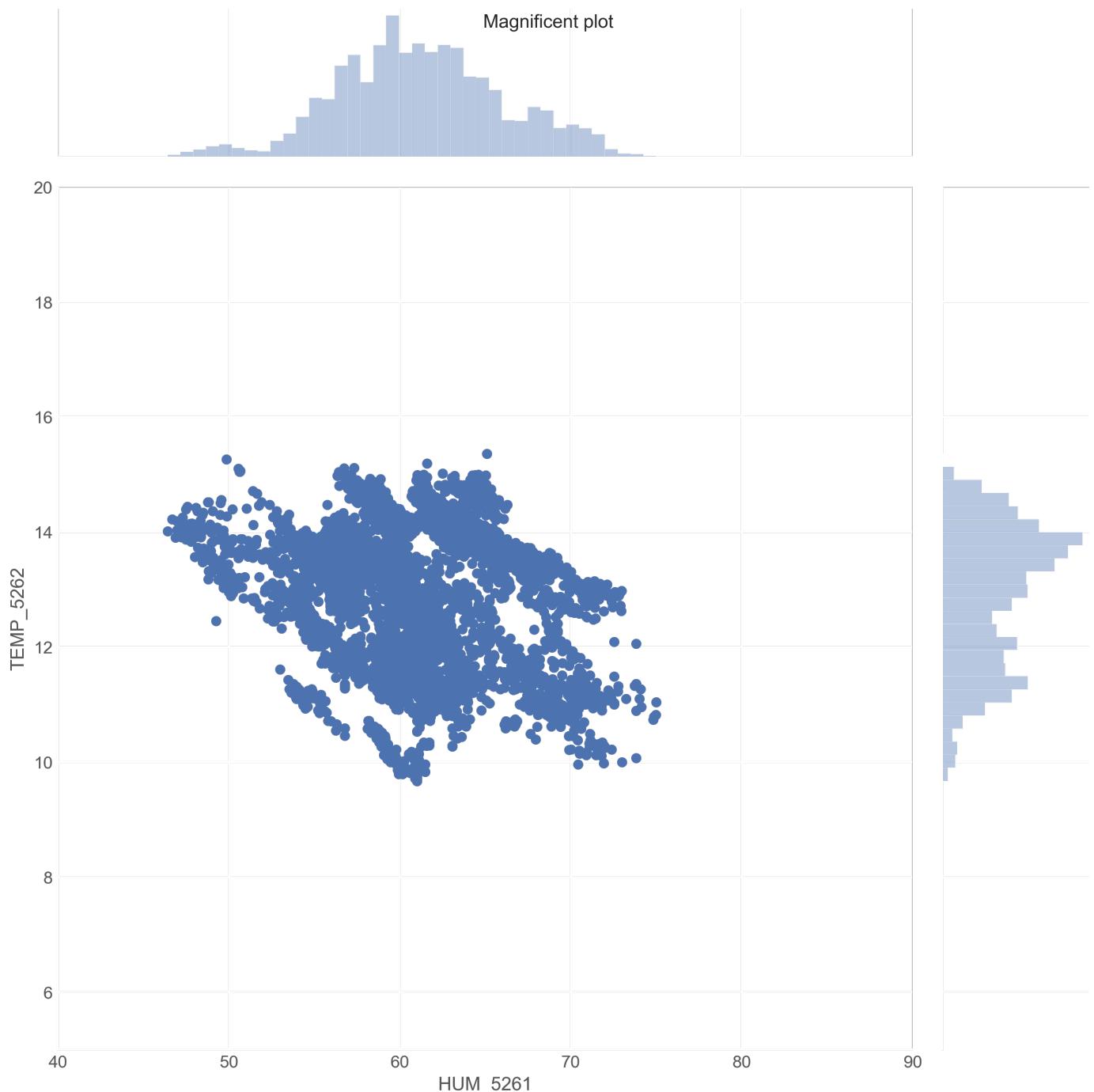
    "1" : {"plot_type": "heatmap",
            "plotting_library": "matplotlib",
            "data": {"test": "TEST_1",
                     "traces": {"1": {"device": "all",
                                     "channel": "NOISE_A",
                                     "subplot": 1}}},
            "options": {"show_plot": false,
                        "separate_device_plots": true,
                        "target_raster": "10Min",
                        "min_date": null,
                        "max_date": "2019-01-03",
                        "relative": false,
                        "export_path": "/Users/macoscarr/Desktop",
                        "file_name": "heatmap_noise"},
            "formatting": {"title": "Magnificent plot",
                          "grid": true,
                          "height": 10,
                          "width": 15,
                          "frequency_hours": 6}
        }
    }
}

```

Note that in this case it only makes sense to put one trace. If we define "device": "all", then "separate_device_plots": true

- `plotting_library` : recommended library is `matplotlib` (although we actually use `seaborn`)
- `formatting` :
 - `frequency_hours` : to choose between 1, 2, ..., 6, 12, 24. Resamples the data to make it fit in bins of that size to create the heatmap

Correlation plot



```

"1" : {"plot_type": "correlation_plot",
"plotting_library": "matplotlib",
"data": {"test": "TEST_1",
"traces": {"1": {"device": "5261",
"channel": "NOISE_A",
"subplot": 1},
"2": {"device": "5262",
"channel": "TEMP",
"subplot": 1}}},
"options": {"show_plot": True,
"separate_device_plots": false,
"export_path": "/Users/macoscar/Desktop",
"file_name": "plot_corr",
"target_raster": '10Min',
"min_date": None,
"max_date": None},
"formatting": {"jpkind": 'scatter',
"title": "Magnificent plot",
"xrange": [0, 100],
"yrange": [0, 40],
"grid": True,
"height": 10,
"width": 15}
}

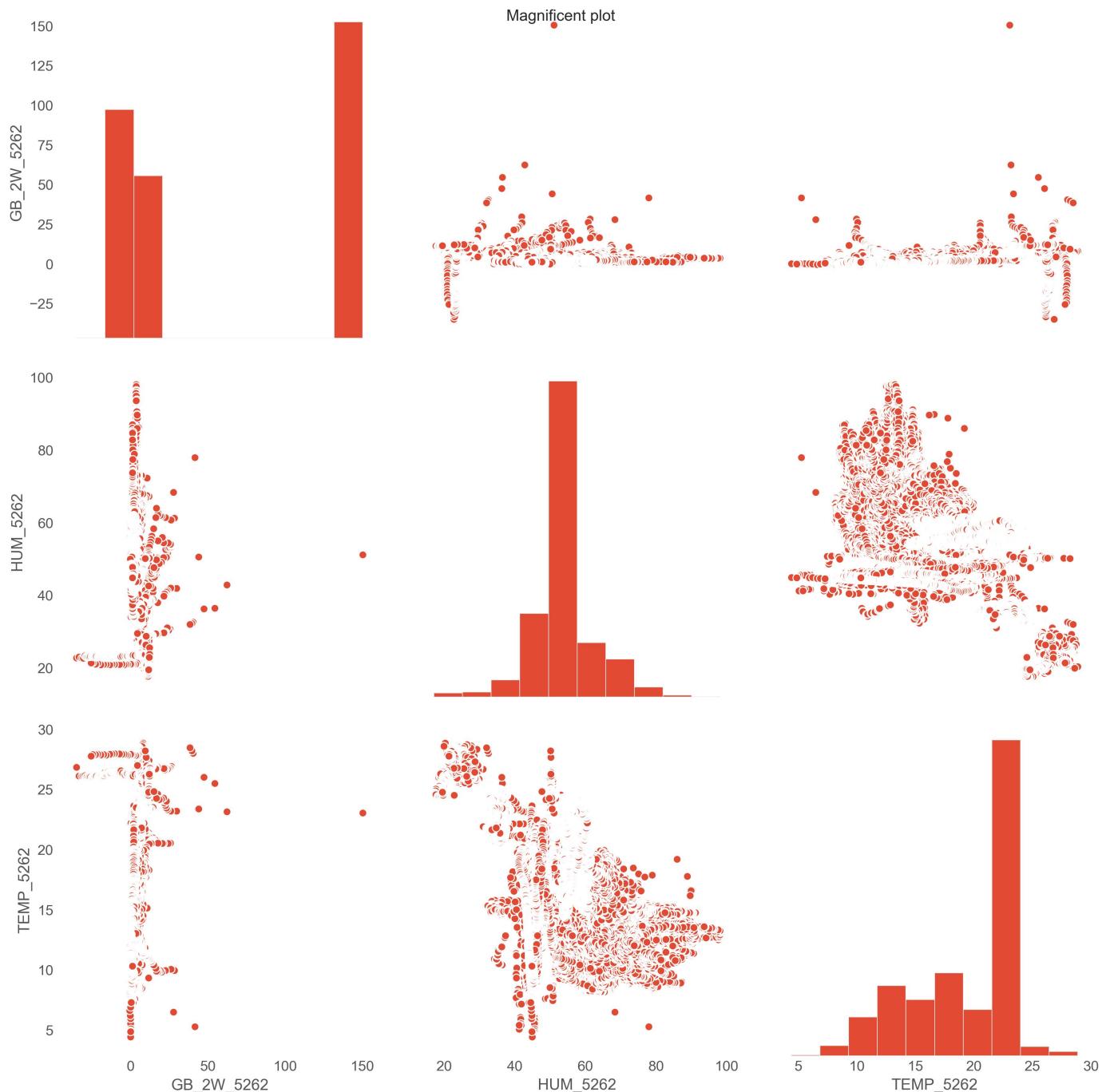
```

Note that in this case it only makes sense to put two traces, in the same subplot.

- **formatting :**
 - `jpkind`: type as defined in `sns.jointplot` documentation, to choose from { “scatter” | “reg” | “resid” | “kde” | “hex” }

Scatter plot matrix

The big brother of the `correlation` plot :



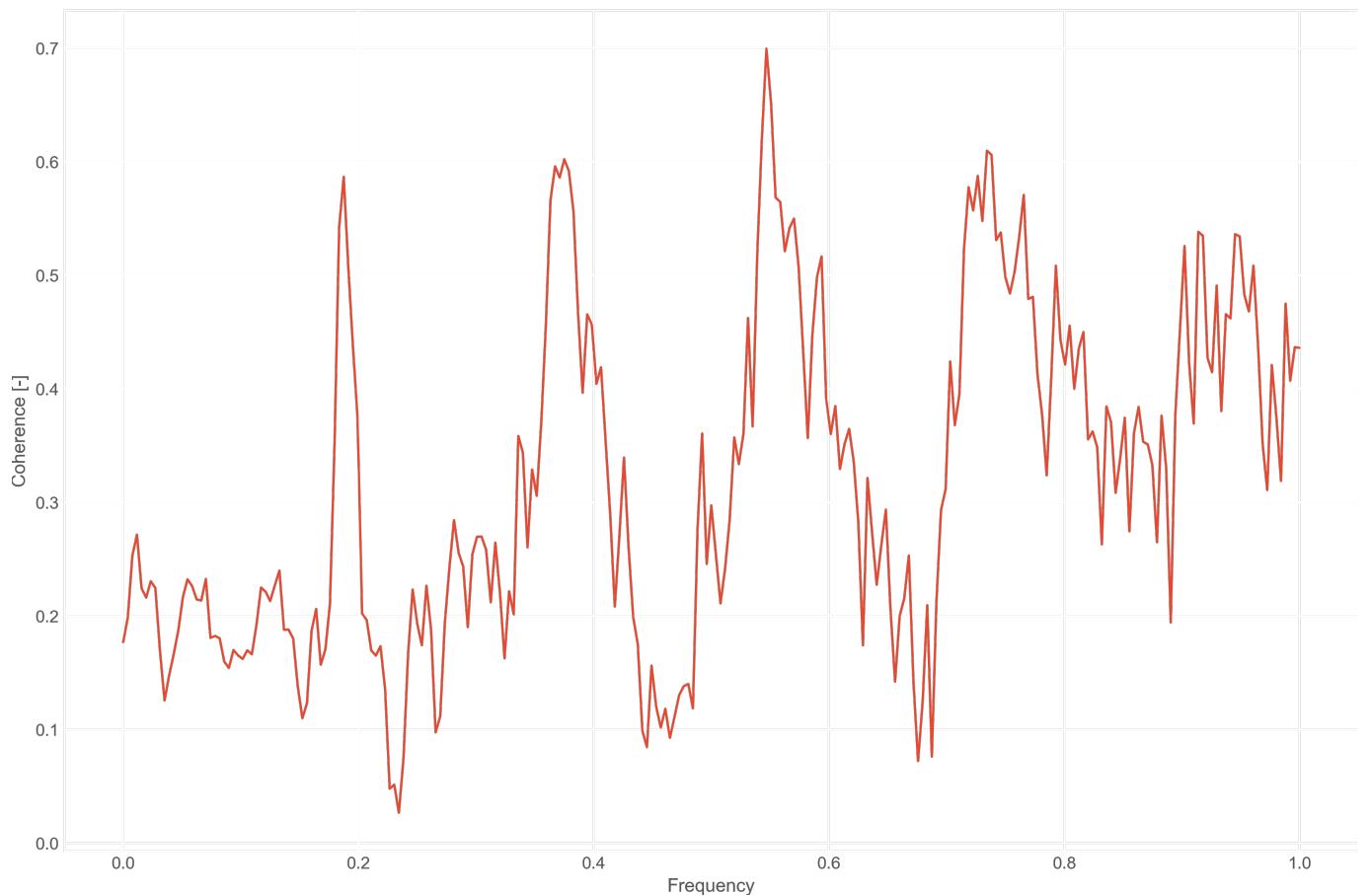
```

"1" : {"plot_type": "scatter_matrix",
"plotting_library": "matplotlib",
"data": {"test": "TEST_1",
"traces": {"1": {"device": "5262",
"channel": "GB_2W",
"subplot": 1},
"2": {"device": "5262",
"channel": "TEMP",
"subplot": 1},
"3": {"device": "5262",
"channel": "HUM",
"subplot": 1}}},
"options": {"show_plot": True,
"separate_device_plots": False,
"export_path": "/Users/macoscarr/Desktop",
"file_name": "plot_scatter",
"target_raster": '10Min',
"min_date": None,
"max_date": None},
"formatting": {"title": "Magnificent plot",
"grid": True,
"height": 4,
"width": 4}
}
}

```

Coherence plot

Magnificent plot

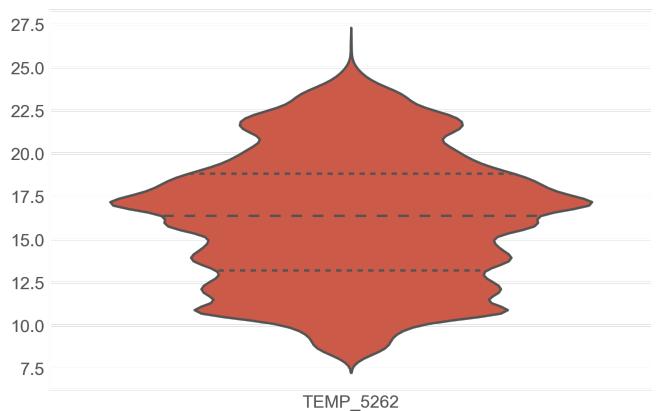
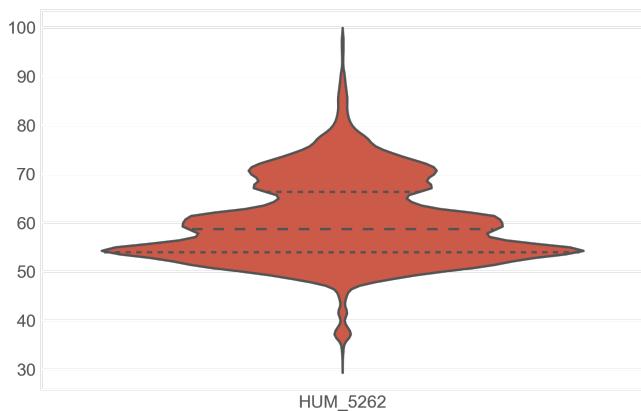
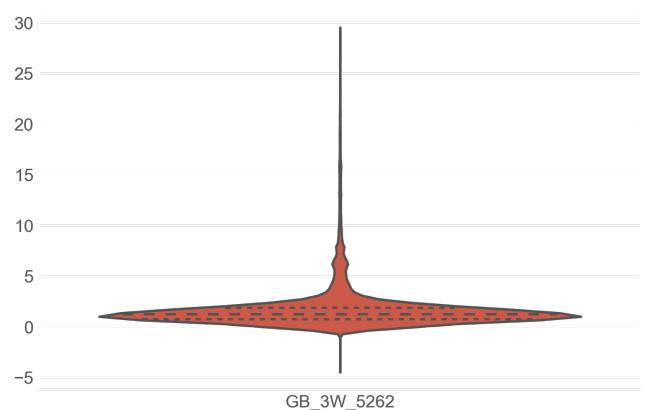
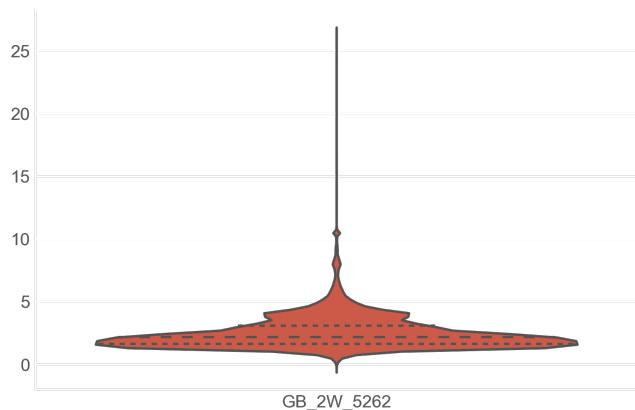
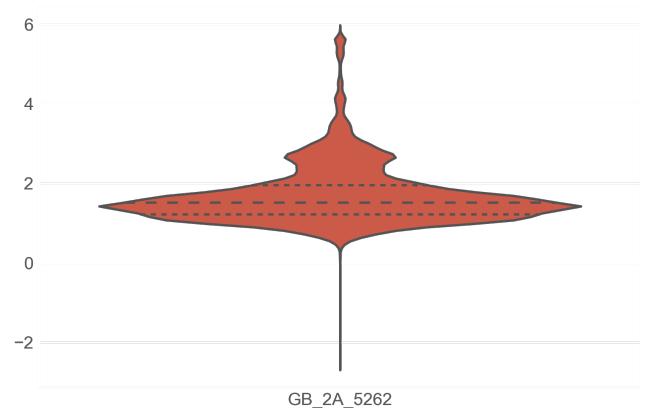
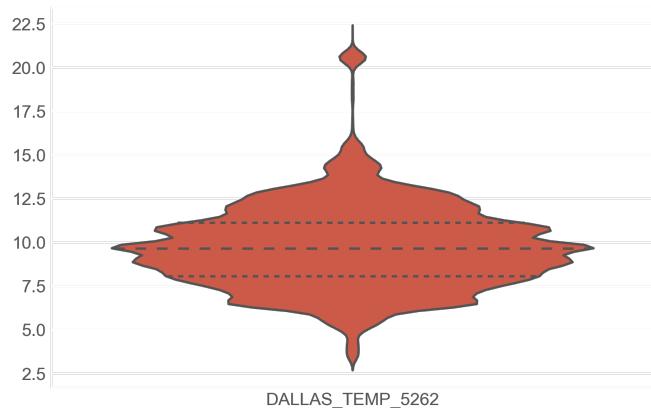


This plot it's used to plot the coherence between x and y. Coherence is the normalized cross spectral density. More info here:

```
"1" : {"plot_type": "coherence_plot",
"plotting_library": "matplotlib",
"data": {"test": "TEST_1",
"traces": {"1": {"device": "5262",
"channel" : "GB_2W",
"subplot": 1},
"2": {"device": "5262",
"channel": "TEMP",
"subplot": 1}}},
"options": {"show_plot": True,
"separate_device_plots": False,
"export_path": "/Users/macoscarr/Desktop",
"file_name": "plot_coherence",
"target_raster": '10Min',
"min_date": None,
"max_date": None},
"formatting": {"title": "Magnificent plot",
"grid": True,
"height": 10,
"width": 15}
}
```

Violin plot

Magnificent plot



This plot shows the signal distribution.

```

"1" : {"plot_type": "violin",
"plotting_library": "matplotlib",
"data": {"test": "TEST_1",
"traces": {"1": {"device": "5262",
"channel": "DALLAS_TEMP",
"subplot": 1},
"2": {"device": "5262",
"channel": "HUM",
"subplot": 1},
"3": {"device": "5262",
"channel": "TEMP",
"subplot": 1},
"4": {"device": "5262",
"channel": "GB_2W",
"subplot": 1},
"5": {"device": "5262",
"channel": "GB_2A",
"subplot": 1},
"6": {"device": "5262",
"channel": "GB_3W",
"subplot": 1}}},
"options": {"show_plot": True,
"separate_device_plots": False,
"export_path": "/Users/macoscari/Desktop",
"file_name": "plot_violin",
"target_raster": '10Min',
"min_date": None,
"max_date": '2019-01-03',
"relative": False,
"ylabel": {1: "External temperature",
2: "Humidity (%RH)",
3: "Temperature (degC)",
4: "Wir",
5: "Wor",
6: "Wur"},
"yrange": {1: [0, 90],
2: [300, 2000],
3: [0, 60],
4: [0, 60],
5: [0, 60],
6: [0, 60]}},
"formatting": {"title": "Magnificent plot",
"grid": True,
"height": 10,
"width": 15}
}

```

12.2 Creating Interfaces

12.2.1 Custom dashboards and notifications

When working on deployments that involve multiple devices a community might face the need to create their own page where the sensors' data is updated on real time. Also, it is sometimes useful to trigger notifications on different services.

This can help to look at data from different spots simultaneously and also to create a sense of community among the devices' owners. This feature can be easily built using Freeboards or Node-RED, both online free visual tool that supports the creation of dashboards. Additionally, [Node-RED] can be used to create notifications on common services such as Twitter or Telegram.

Node RED

Node-RED is an open-source visual tool that enabled the wiring of hardware devices, APIs and online services. The tool can be easily installed on any local computer or it can be used directly on the Smart Citizen infrastructure.

Examples

You can find the following examples in the toolkit repository:

- Trigger notifications
- Device dashboard

Freeboard

WIP

This is a work in progress. You can find a demo here

12.2.2 Talk to the world

Due to their unobtrusive nature, sensor technologies like Smart Citizen may easily blend in the background of users' attention. To bring the sensed data back to the surface and support sensemaking and awareness processes, it is possible to use the SCKs' data to trigger actions on the physical environment.

A Raspberry Pi is probably the best tool to do so, since it can also be a suitable tool to engage people with coding, creating new internet of things (IoT) and physical computing applications.

Raspberry Pi

Blink example

This example presents a small Python script that can turn two lights based on the real-time temperature data from a remote sensor on the Smart Citizen platform.

We will implement a simple logic: When temperature on the remote sensor reaches 25 degrees then turn the first light on. When temperature is below 25 degrees turn the first light off and then turn the other light on. We will use the *Raspberry Pi GPIOs (General Purpose Input Outputs)* to connect to LED's that represent the status of our sensor.

Step-by-step

We will need to wire the two LED's following the schematic below:

Once we have the Raspberry Pi running and connected to the internet we will need to save the Python script below on the desktop:

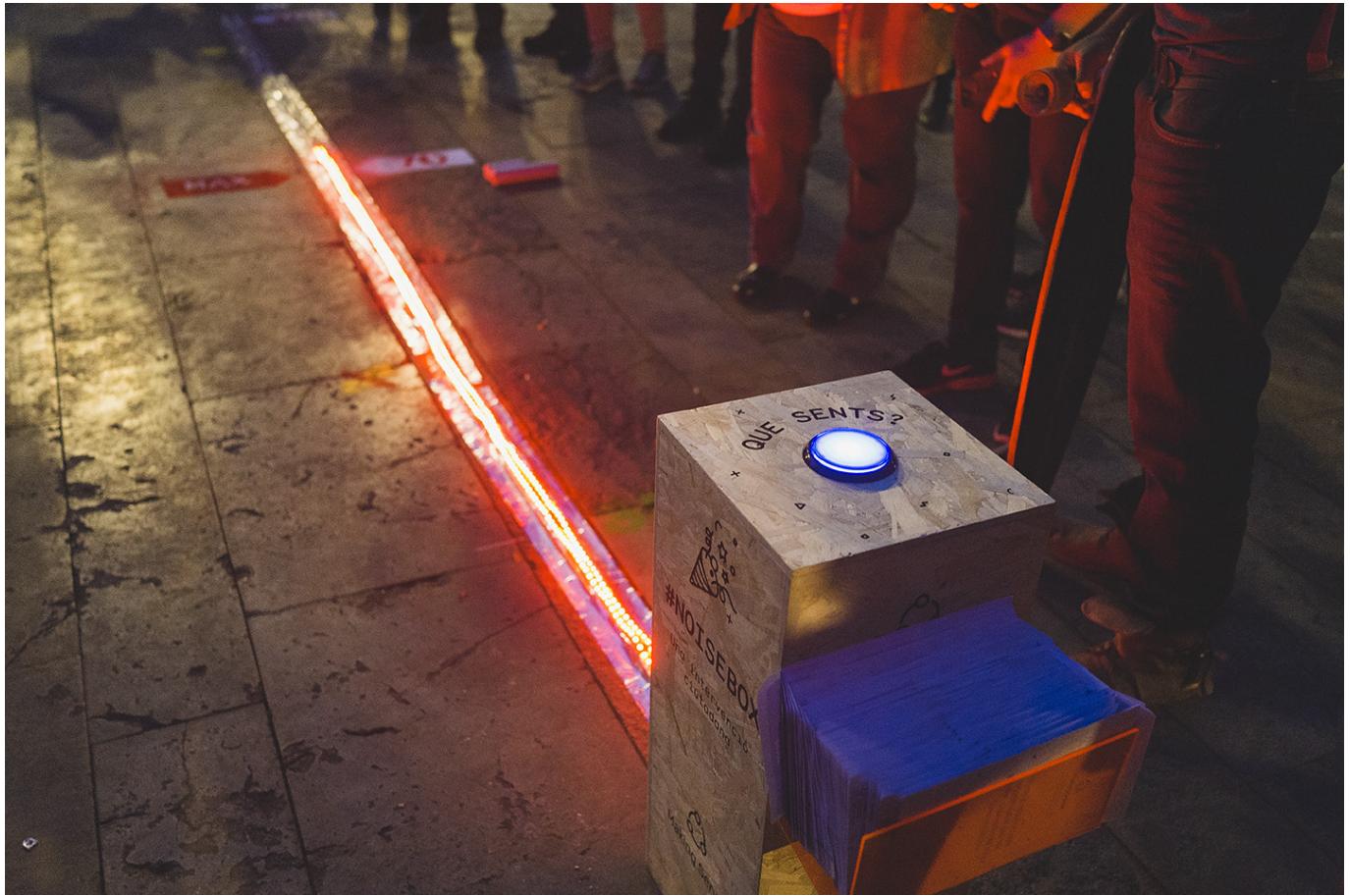
```
# Smart Citizen Examples for the Raspberry Pi
#
# http://smartcitizen.me
#
# Trigger 2 LEDs depending on the temperature
# For more information on the LEDs connection check: https://learn.sparkfun.com/tutorials/raspberry-gpio
# For more information on the SmartCitizen API check: http://developer.smartcitizen.me
#
import RPi.GPIO as GPIO
import json, requests, time
GPIO.setup(18, GPIO.OUT)
GPIO.setup(23, GPIO.OUT)
while True:
    r = requests.get('https://api.smartcitizen.me/v0/devices/3292')
    data = json.loads(r.text)
    for sensor in data['data']['sensors']:
        if sensor['description'] == 'Temperature': #C0, N02...
            print sensor['value']
            if sensor['value'] > 25:
                print 'LED ON'
                GPIO.output(18, GPIO.HIGH)
                GPIO.output(23, GPIO.LOW)
            else:
                print 'LED OFF'
                GPIO.output(18, GPIO.LOW)
                GPIO.output(23, GPIO.HIGH)
    time.sleep(15) #Update every 15 seconds
```

Finally, open the Terminal app and run:

```
pi@raspberrypi ~ $ cd Desktop
pi@raspberrypi Desktop $ sudo python smartcitizen-led.py
```

The sensor box

The **Sensor Box is display installation** aimed at engaging citizens to discuss about data on the public space. The installation was built by the Making Sense Barcelona community champions to talk about noise problems affecting neighbours around the Plaça del Sol area in Barcelona. However the installation was built from the ground up to be replicable ad easy to built in oder Fab Labs worldwide. This aims at creating a tool communities can built to engage citizens to discuss about issues by using the data provided by the Smart Citizen Kit.



The device comprises a wooden box equipped with a Smart Citizen Kit to which was a 5 meter long LED strip has been attached. Participants can press a button on the box to trigger the noise sensor. The original installation was battery powered but it can also be plugged to simplify the design and cost.

Step-by-step

Visit the public displays repository and download the directory.

The `/built` folder contains the files for building the installation: `NoiseBox.blend` the whole installation design in blender, `CableClip.stl` and `Hinge.stl` 3D printed parts for the cable clips and hinges used, `Acrylic.dxf` the acrylic cover lasercut file and `noiseBoxSchematic.fzz` the wiring diagram for the installation.

The `/code` folder contains the Arduino files to drive the installation. The Arduino sketch reads sensor data from an SCK 1.5 over the I2C bus when a user presses the button display the result on a WS2811 addressable LED strip. This code was originally created to display reading from the noise sensor in dB but it can quickly be changed to support any other sensor. It runs on an Arduino UNO but any compatible board can be used.

Check out the Making Sense D2.3 Smart Citizen Toolkit report and Making Sense D.24 Smart Citizen Toolkit report updates for more examples!

12.3 Model your sensor data

In this section, we will detail how to develop models for our sensors. We will try two different approaches for model calibration:

- **Ordinary Least Squares (OLS):** based on the statsmodels package, the model is able to ingest an expression referring to the kit's available data and perform OLS regression over the defined training and test data
- **Machine Learning (LSTM):** based on the keras package using tensorflow in the backend. This framework can be used to train larger collections of data, among others:
 - Robust to noise
 - Learn non-linear relationships
 - Aware of temporal dependence

Load some data first

We will need to load the data first, for this, check the guides to organise the data and to load it

12.3.1 Ordinary Least Squares example

Let's delve first into an OLS example.

Info

You can follow this example using this notebook

```
from src.models.model import model_wrapper

# Input model description
model_description_ols = {"model_name": "OLS_UCD",
                         "model_type": "OLS",
                         "model_target": "ALPHASENSE",
                         "data": {"train": {"2019-03_EXT_UCD_URBAN_BACKGROUND_API": {"devices": ["5262"],
                                                                      "reference_device": "CITY_COUNCIL"}},
                                  "test": {"2019-03_EXT_UCD_URBAN_BACKGROUND_API": {"devices": ["5565"],
                                                                     "reference_device": "CITY_COUNCIL"}},
                         "features": {"REF": "NO2_CONV",
                                      "A": "GB_2W",
                                      "B": "GB_2A",
                                      "C": "HUM"},
                         "data_options": {"frequency": '1Min',
                                         "clean_na": True,
                                         "clean_na_method": "drop",
                                         "min_date": None,
                                         "frequency": "1Min",
                                         "max_date": '2019-01-15'},
                         },
                         "hyperparameters": {"ratio_train": 0.75},
                         "model_options": {"session_active_model": True,
                                           "show_plots": True,
                                           "export_model": False,
                                           "export_model_file": False,
                                           "extract_metrics": True}
                     }
```

More info

Check the guide on batch analysis for a definition of each parameter.

We have to keep at least the key `REF` within the `"features"`, but the rest can be renamed at will. We can also input whichever `formula_expression` for the model regression in the following format:

```
"expression" : 'REF ~ A + np.log(B)'
```

Which converts to:

$$\text{REF} = \text{A} + \log(\text{B})$$

We can also define the ratio between the train and test dataset and the minimum dates to use within the datasets (globally):

```
min_date = '2018-08-31 00:00:00'
max_date = '2018-09-06 00:00:00'

# Important that this is a float, don't forget the .
"hyperparameters": {"ratio_train": 0.75}
```

If we run this cell, we will perform model calibration, with the following output:

```
OLS Regression Results
=====
Dep. Variable:           REF    R-squared:      0.676
Model:                 OLS    Adj. R-squared:  0.673
Method:                Least Squares   F-statistic:   197.5
Date:      Thu, 06 Sep 2018   Prob (F-statistic): 1.87e-135
Time:          12:25:17   Log-Likelihood:     1142.9
No. Observations:      575    AIC:             -2272.
Df Residuals:         568    BIC:             -2241.
Df Model:                  6
Covariance Type:    nonrobust
=====
            coef    std err        t    P>|t|      [0.025    0.975]
-----
Intercept   -3.7042    0.406    -9.133    0.000    -4.501    -2.908
A           0.0011    0.000     2.953    0.003     0.000     0.002
np.log(B)  -3.863e-05 7.03e-06    -5.496    0.000   -5.24e-05  -2.48e-05
=====
Omnibus:            7.316   Durbin-Watson:  0.026
Prob(Omnibus):       0.026   Jarque-Bera (JB): 10.245
Skew:              -0.076   Prob(JB):      0.00596
Kurtosis:            3.636   Cond. No.: 4.29e+05
=====
Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 4.29e+05. This might indicate that there are
strong multicollinearity or other numerical problems.
```

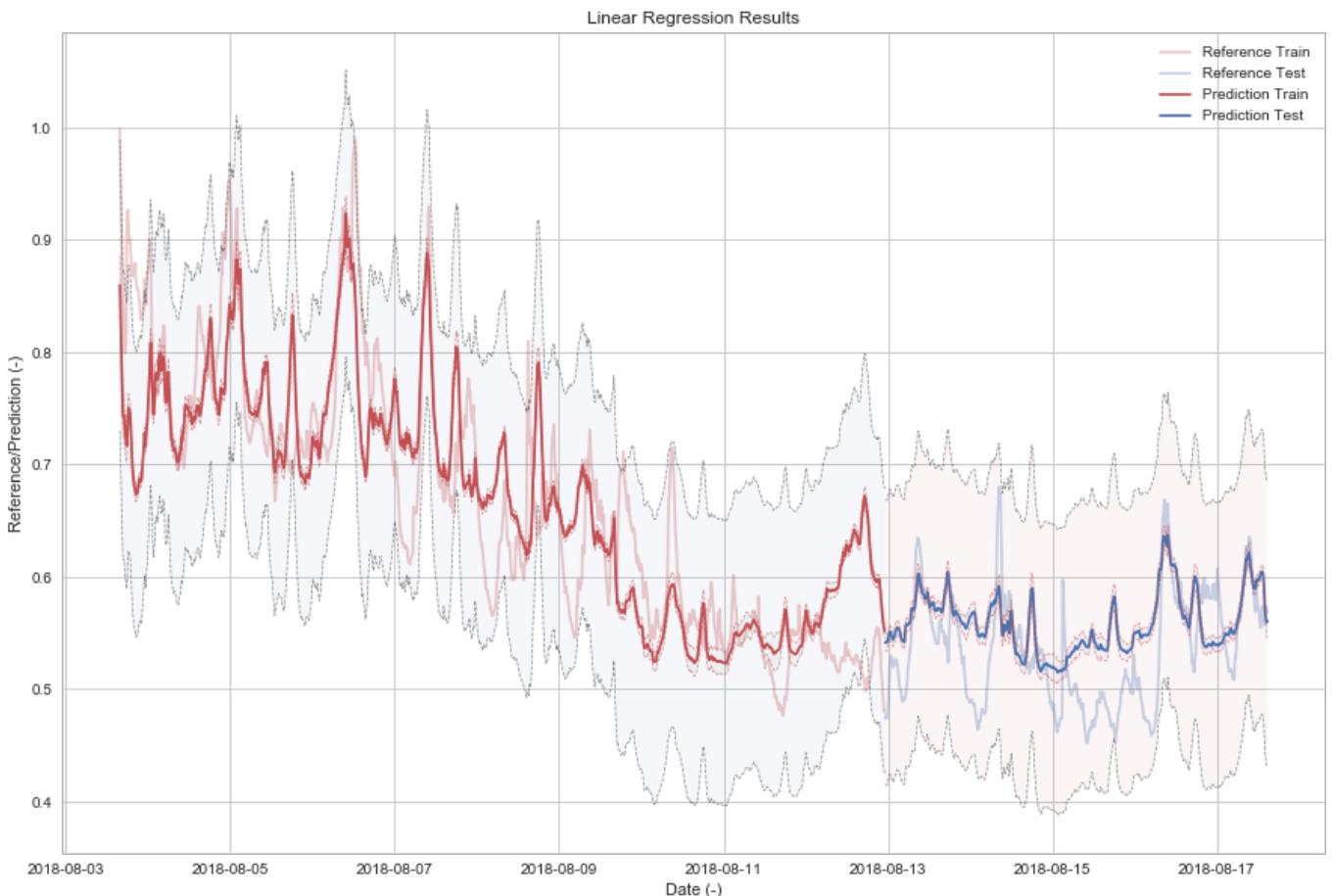
This output brings a lot of information. First, we find what the dependent variable is, in our case always `'REF'`. The type of model used and some general information is shown below that.

More statistically important information is found in the rest of the output. Some key data:

- *R-squared and adjusted R-squared*: this is our classic correlation coefficient or R^2 . The adjusted one aims to correct the model overfitting by the inclusion of too many variables, and for that introduces a penalty on the number of variables included
- Below, we can find a summary of the *model coefficients* applied to all the variables and the $P>|t|$ term, which indicates the significance of the term introduced in the model
- *Model quality diagnostics* are also indicated. Kurtosis and skewness are metrics for determining the distribution of the residuals. They indicate how the residuals of the model resemble a normal distribution. Below, we will review more on diagnosis plots. The Jarque Bera test indicates if the residuals are normally distributed (the null hypothesis is a joint hypothesis of the skewness being zero and the excess kurtosis being zero), and a value of zero indicates that the data is normally distributed. If the Jarque Bera test is valid (in the case above it isn't), the Durbin Watson is applicable in order to check for autocorrelation of the residuals (meaning that the residuals of our model are related among themselves and that we haven't captured some characteristics of our data with the tested model).

Finally, there is a warning at the bottom indicating that the condition number is large. It suggests we might have multicollinearity problems in our model, which means that some of the independent variables might be correlated among themselves and that they are probably not necessary.

Our function also depicts the results in a graphical way for us to see the model itself. It will show the training and test datasets (as `Reference Train` and `Reference Test` respectively), and the prediction results. The mean and absolute confidence intervals for 95% confidence are also shown:



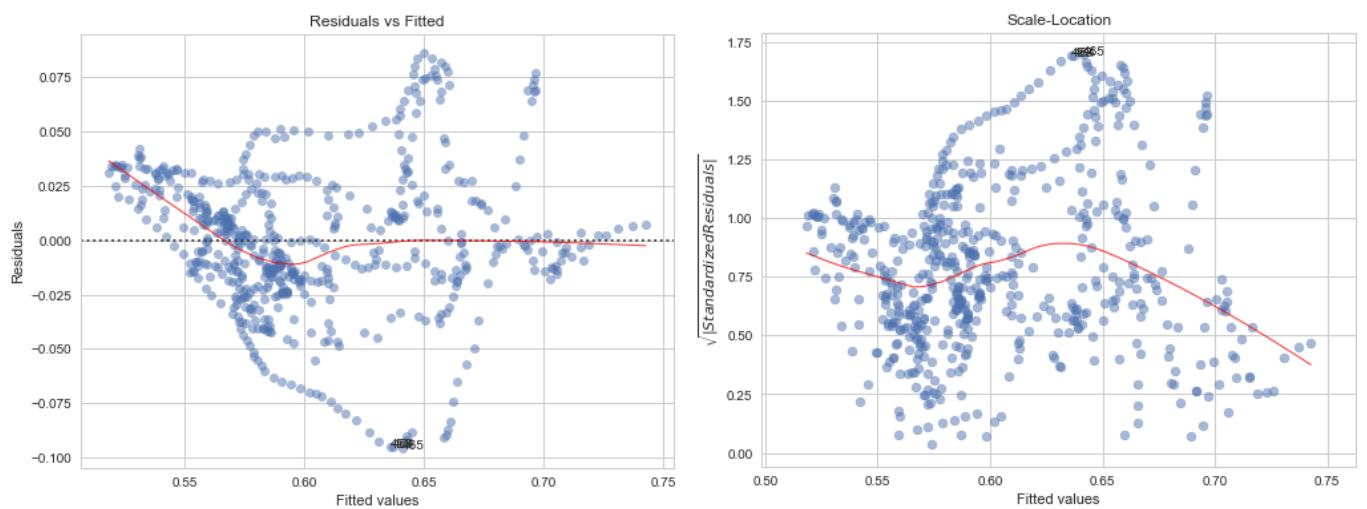
Now we can look at some other model quality plots. If we run the cell below, we will obtain an adaptation of the summary plots from R:

```
from linear_regression_utils import modelRplots
%matplotlib inline

modelRplots(model, dataTrain, dataTest)
```

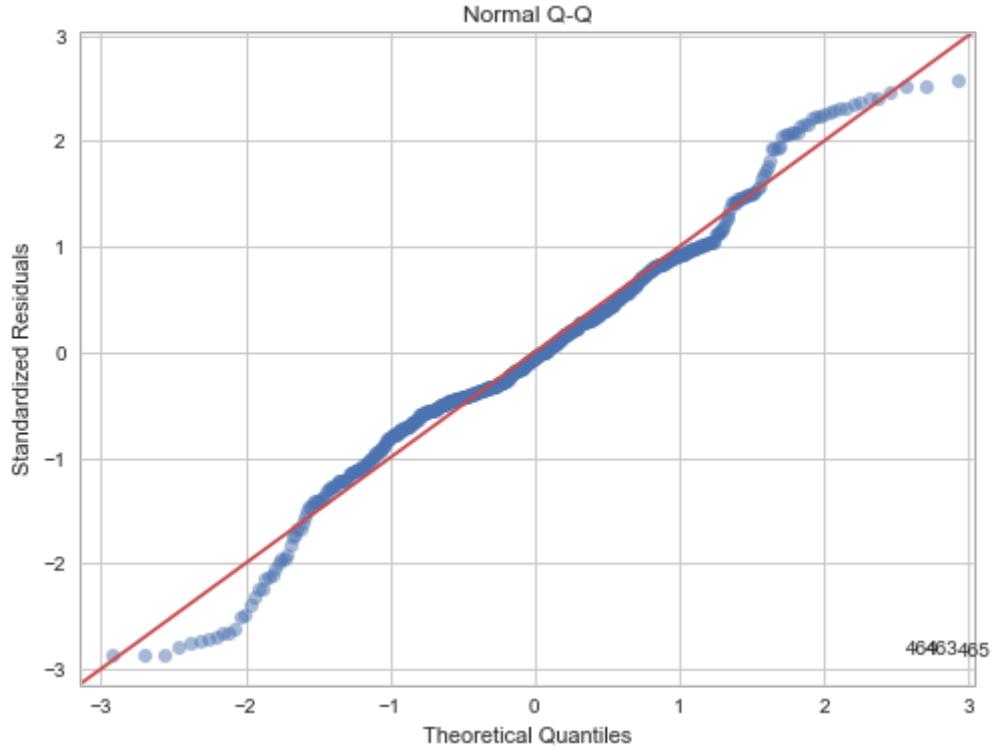
Let's review the output step by step:

- **Residual vs Fitted and Scale Location plot:** these plots represent the model heteroscedasticity, which is a representation of the residuals versus the fitted values. This plot is helpful to check if the errors are distributed homogeneously and that we are not penalising high, low, or other values. There is also a red line which represents the average trend of this distribution which we want it to be horizontal. For more information visit [here](#) and [here](#). Clearly, in this model we are missing something:



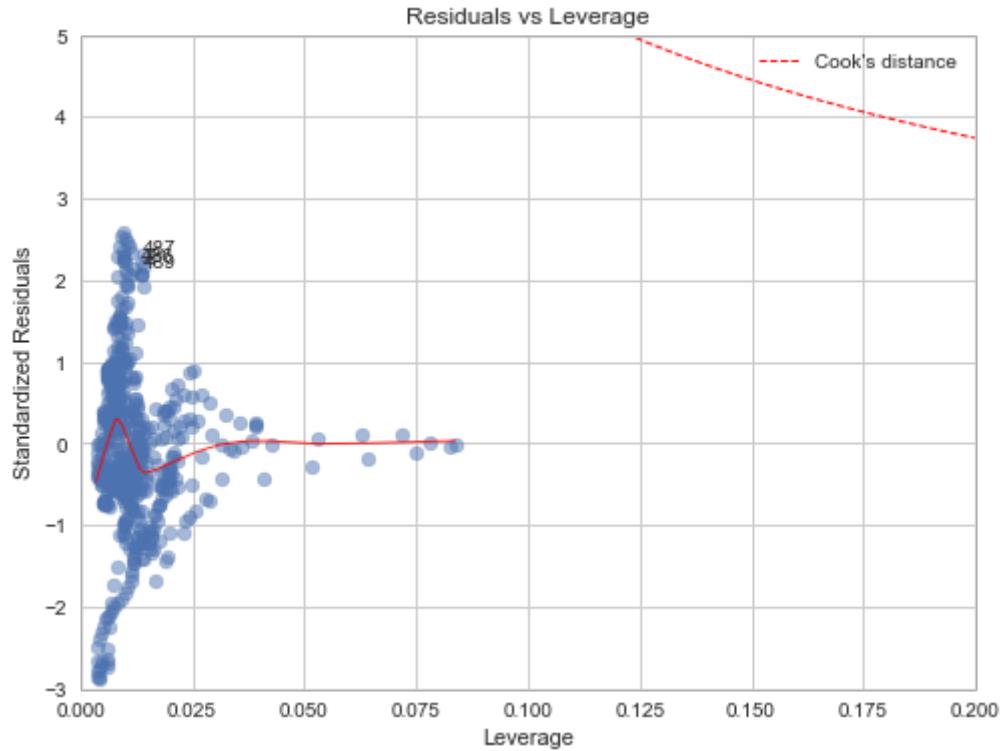
- **Normal QQ:** the qq-plot is a representation of the kurtosis and skewness of the residuals distribution. If the data were well described by a normal distribution, the values should be about the same, i.e.: on the diagonal (red line). For example, in our case the model presents a deviation on both tails, indicating skewness. In general, a simple rubric to interpret a qq-plot is that if a given tail twists off counterclockwise from the reference line, there is more data in that tail of your distribution than in a theoretical normal, and if a tail twists off clockwise there is less data in that tail of your distribution than in a theoretical normal. In other words:

- if both tails twist counterclockwise we have heavy tails (leptokurtosis),
- if both tails twist clockwise, we have light tails (platykurtosis),
- if the right tail twists counterclockwise and the left tail twists clockwise, we have right skew
- if the left tail twists counterclockwise and the right tail twists clockwise, we have left skew

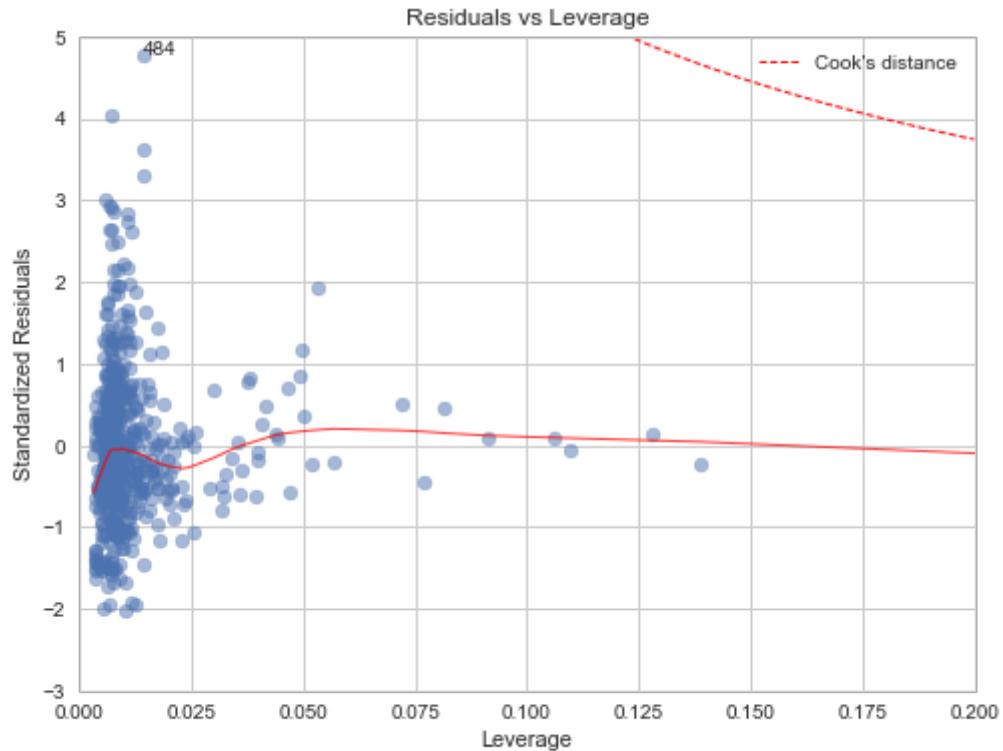


- **Residuals vs Leverage:** this plot is probably the most complex of them all. It shows how much leverage one single point has on the whole regression. It can be interpreted as how the average line that passes through all the data (that we are calculating with the OLS) can be modified by 'far' points in the distribution, for example, outliers. This leverage can be seen as how much a single point is able to pull down or up the average line. One way to think about whether or not the results are driven by a given data point is to calculate how far the predicted values for your data would move if your model were fit without the data point in question. This calculated total distance is called Cook's distance. We can have four cases (more information from source, here)

- everything is fine (the best)
- high-leverage, but low-standardized residual point
- low-leverage, but high-standardized residual point
- high-leverage, high-standardized residual point (the worst)



In this case, we see that our model has some points with higher leverage but low residuals (probably not too bad) and that the higher residuals are found with low leverage, which means that our model is safe to outliers. If we run this function without the filtering, some outliers will be present and the plot turns into:



Finally, we can export our model and generate some metrics to evaluate the results.

12.3.2 Machine learning example

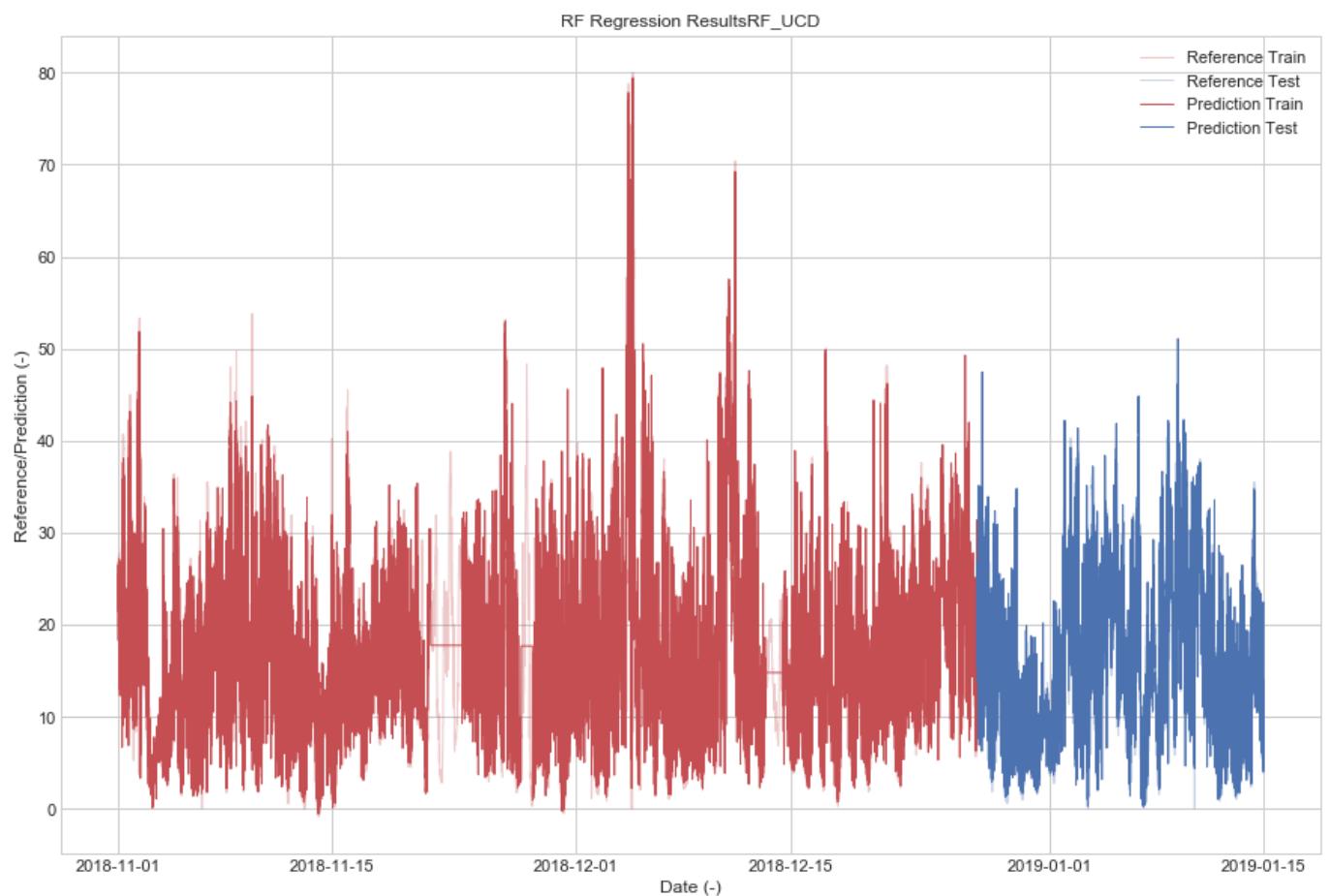
Machine learning algorithms promise a better representation of the sensor's data, being able to learn robust non-linear models and sequential dependencies. For that reason, we have implemented toolset based on keras with Tensorflow backend, in order to train sequential models³.

```
model_description_rf = {"model_name": "RF_UCD",
    "model_type": "RF",
    "model_target": "ALPHASENSE",
    "data": {"train": {"2019-03_EXT_UCD_URBAN_BACKGROUND_API": {"devices": ["5262"],
        "reference_device": "CITY_COUNCIL"}},
        "test": {"2019-03_EXT_UCD_URBAN_BACKGROUND_API": {"devices": ["5565"],
        "reference_device": "CITY_COUNCIL}}},
    "features": {"REF": "NO2_CONV",
        "A": "GB_2W",
        "B": "GB_2A",
        "C": "HUM"},
    "data_options": {"target_raster": '1Min',
        "clean_na": True,
        "clean_na_method": "drop",
        "min_date": None,
        "frequency": "1Min",
        "max_date": '2019-01-15'},
    },
    "hyperparameters": {"ratio_train": 0.75,
        "min_samples_leaf": 2,
        "max_features": None,
        "n_estimators": 100,
        "shuffle_split": True},
    "model_options": {"session_active_model": True,
        "show_plots": True,
        "export_model": False,
        "export_model_file": False,
        "extract_metrics": True}
}
```

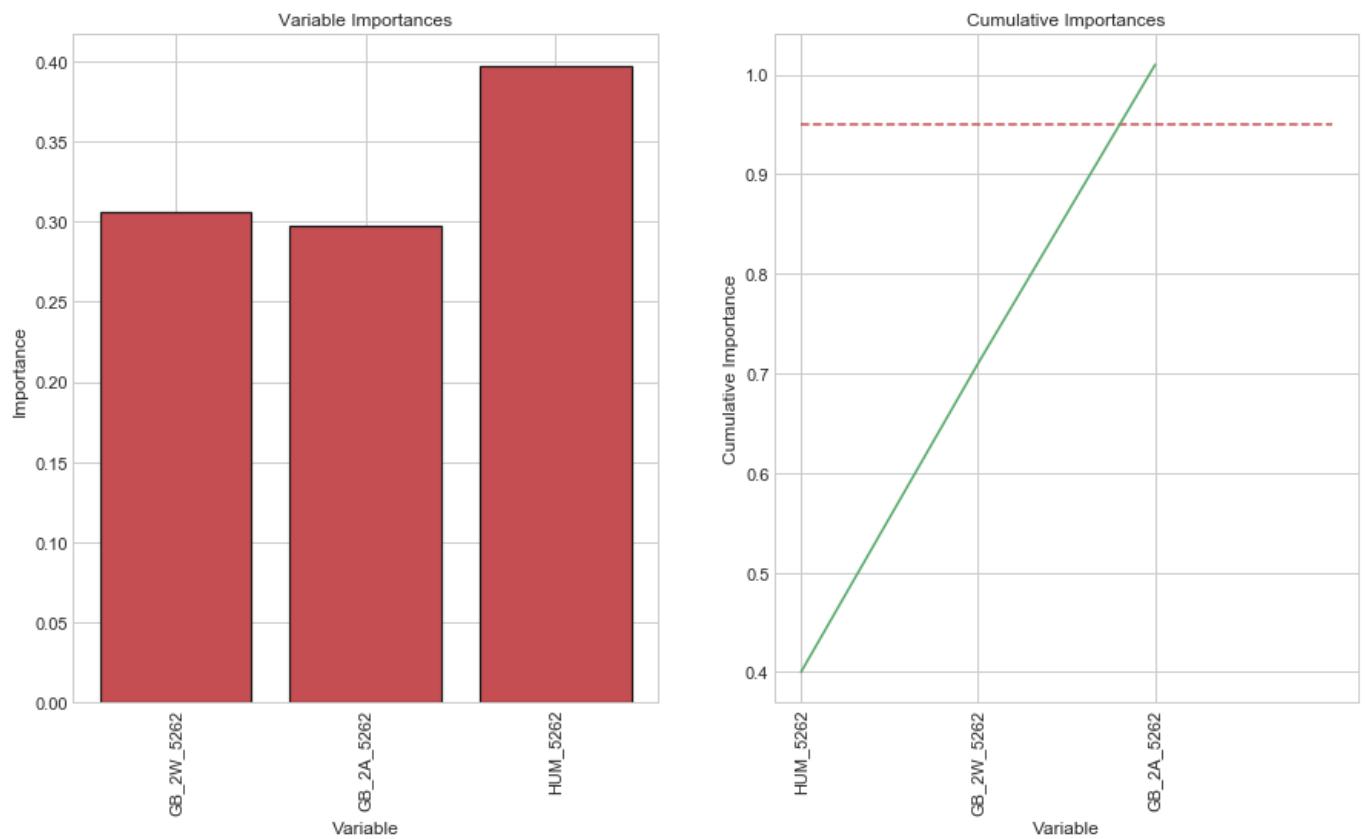
Output:

```
Using TensorFlow backend.
...
Beginning Model RF_UCD
Model type RF
Preparing dataframe model for test 2019-03_EXT_UCD_URBAN_BACKGROUND_API
Data combined successfully
Creating models session in recordings
Dataframe model generated successfully
Training Model RF_UCD...
Training done
Variable: HUM_5262 Importance: 0.4
Variable: GB_2W_5262 Importance: 0.31
Variable: GB_2A_5262 Importance: 0.3
Calculating Metrics...
Metrics Summary:
Metric      Train     Test
avg_ref     16.648   15.861
avg_est     16.666   16.000
sig_ref     11.438   10.584
sig_est     9.493    9.404
bias        0.019    0.139
normalised_bias 0.002    0.013
sigma_norm  0.830    0.888
sign_sigma  -1.000   -1.000
rsquared     0.799    0.879
RMSD        5.124    3.677
RMSE_norm_unb 0.453    0.351
No specifics for RF type
Preparing devices from prediction
Channel 5262_RF_UCD prediction finished
```

This will also output some nice plots for visually checking our model performance:

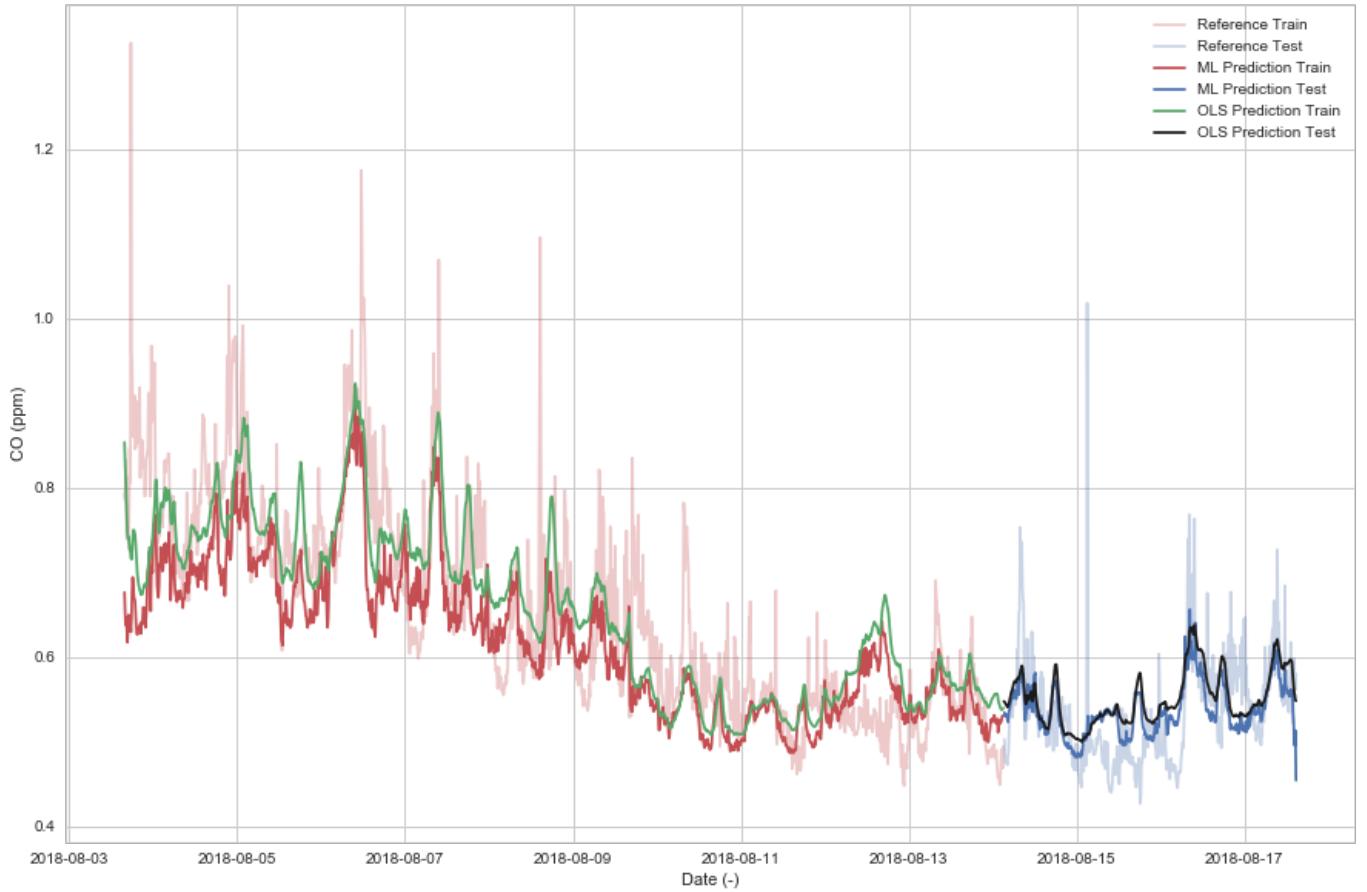


And some extras about variable importance:



12.3.3 Model comparison

Here is a visual comparison of both models:



It is very difficult though, to know which one is performing better. Let's then evaluate and compare our models. In order to evaluate it's metrics, we will be using the following principles¹²:

Info

In all of the expressions below, the letter m indicates the model field, r indicates the reference field. Overbar is average and σ is the standard deviation.

Linear correlation Coefficient A measure of the agreement between two signals:

$$R = \frac{1}{N} \sum_{i=1}^N (m_i - \bar{m})(r_i - \bar{r}) / (\sigma_m \sigma_r)$$

The correlation coefficient is bounded by the range $-1 \leq R \leq 1$. However, it is difficult to discern information about the differences in amplitude between two signals from R alone.

Normalized standard deviation A measure of the differences in amplitude between two signals: $\sigma_m = \sqrt{\frac{1}{N} \sum_{i=1}^N (m_i - \bar{m})^2}$

unbiased Root-Mean-Square Difference A measure of how close the modelled points fall to each other:

$$RMSE' = \sqrt{\frac{1}{N} \sum_{i=1}^N ((m_i - \bar{m}) - (r_i - \bar{r}))^2}$$

Potential Bias Difference between the means of two fields: $B = \bar{m} - \bar{r}$ **Total RMSD** A measure of the average magnitude of difference: $RMSE = \sqrt{\frac{1}{N} \sum_{i=1}^N ((m_i - \bar{m}) - (r_i - \bar{r}))^2}$

In other words, the unbiased RMSD (RMSD') is equal to the total RMSD if there is no bias between the model and the reference fields (i.e. $B = 0$). The relationship between both reads:

$$\text{RMSD}^2 = B^2 + \text{RMSD}'^2$$

In contrast, the unbiased RMSD may be conceptualized as an overall measure of the agreement between the amplitude (σ) and phase (ϕ) of two temporal patterns. For this reason, the correlation coefficient (R), normalized standard deviation (σ^*), and unbiased RMSD are all referred to as **pattern statistics**, related to one another by:

$$\text{RMSD}'^2 = \sigma_r^2 + \sigma_m^2 - 2\sigma_r\sigma_m R$$

Normalized and unbiased RMSD If we recast in standard deviation normalized units (indicated by the asterisk) it becomes:

$$\text{RMSD}'^* = \sqrt{1 + \sigma^*^2 - 2R}$$

NB: the minimum of this function occurs when $\sigma^* = R$.

Normalized bias Gives information about the mean difference but normalized by the $\sigma^* \text{ } B^* = \overline{m} - \overline{r} / \sigma_r$

Target diagrams The target diagram is a plot that provides summary information about the **pattern statistics as well as the bias** thus yielding an overview of their respective contributions to the total RMSD. In a simple Cartesian coordinate system, the unbiased RMSD may serve as the X-axis and the bias may serve as the Y-axis. The distance between the origin and the model versus observation statistics (any point, s , within the X,Y Cartesian space) is then equal to the total RMSD. If all is normalized by the σ_r , the distance from the origin is again the *standard deviation normalized total RMSD*:¹

$$\text{RMSD}^{*2} = B^{*2} + \text{RMSD}^{*2}$$

The resulting target diagram then provides information about:

- whether the σ_m is larger or smaller than the σ_r
- whether there is a positive or negative bias

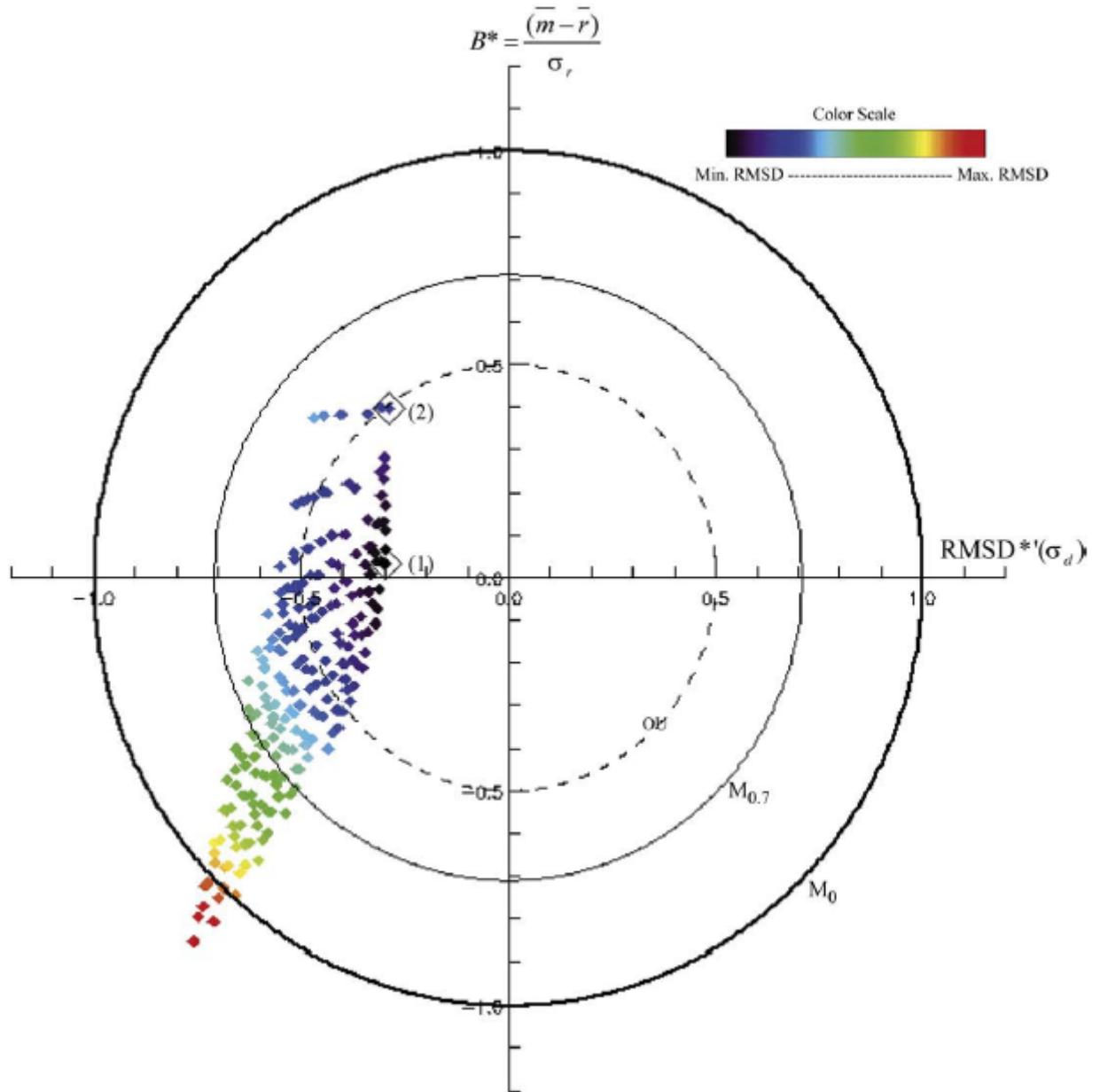


Image Source: Jolliff et al.¹

Any point greater than $\text{RMSD}^*=1$ is to be considered a poor performer since it doesn't offer improvement over the time series average.

Interestingly, the target diagram has no information about the correlation coefficient R, but some can be inferred, knowing that all the points within the $\text{RMSD}^* < 1$ are positively correlated ($R>0$), although, in ¹ it is shown that a circle marker with radius $M_{\{R1\}}$, means that all the points between that marker and the origin have a R coefficient larger than R1, where:

$$M_{\{R1\}} = \min(\text{RMSD}^*) = \sqrt{1+R1^2-2R1^2}$$

12.3.4 Results

Let's now compare both models with the target diagram:

```

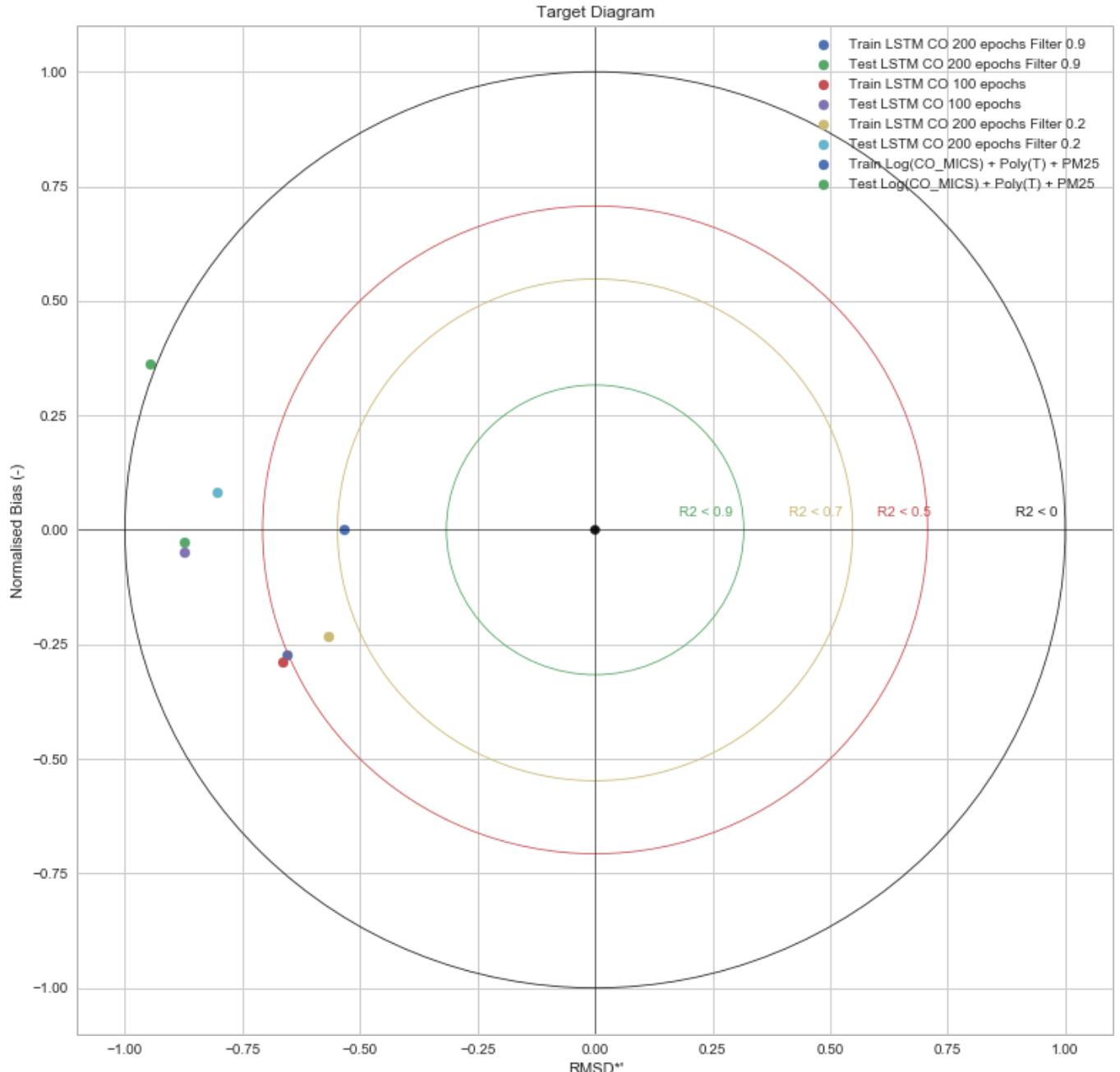
from src.visualization.visualization import targetDiagram
%matplotlib inline
models = dict()

group = 0
for model in [ols_model, rf_model]:
    for dataset in ['train', 'validation']:
        if dataset in model.metrics.keys():
            models[model.name + '_' + dataset] = model.metrics[dataset]
            models[model.name + '_' + dataset]['group'] = group

targetDiagram(models, True, 'seaborn-talk')

```

Output:



Here, every point that falls inside the yellow circle, will have an R^2 over 0.7, and so will be the red and green for R^2 over 0.5 and 0.9 respectively. We see that only one of our models performs well in that sense, which is the training dataset of the OLS. However, this dataset performs pretty badly in the test dataset, being the LSTM options much better. This target

diagram offers information about how the hyperparameters affect our networks. For instance, increasing the training epochs from 100 to 200 does not affect greatly on model performance, but the effect of filtering the data beforehand to reduce the noise shows a much better model performance in both, training and test dataframe.

12.3.5 Export the models

Let's now assume that we are happy with our models. We can now export them:

```
ols_model.export('directory')
rf_model.export('directory')
```

Output:

```
Saving metrics
Saving hyperparameters
Saving features
Model included in summary
```

And in our directory:

```
→ models ls -l
RF_UCD_features.sav
RF_UCD_hyperparameters.sav
RF_UCD_metrics.sav
```

Save the model file

If you want to save the model file into the disk, change the option "export_model": False, to True! Be careful though, it can take quite a lot of space. If you just want to keep the model to test in the current session, it is best to just use "session_active_model": True, .

12.3.6 References

1. Engineering statistics handbook 
2. Summary diagrams for coupled hydrodynamic-ecosystem model skill assessment (Jolliff et al.) 
3. Machine learning mastery 

12.4 Debug the firmware

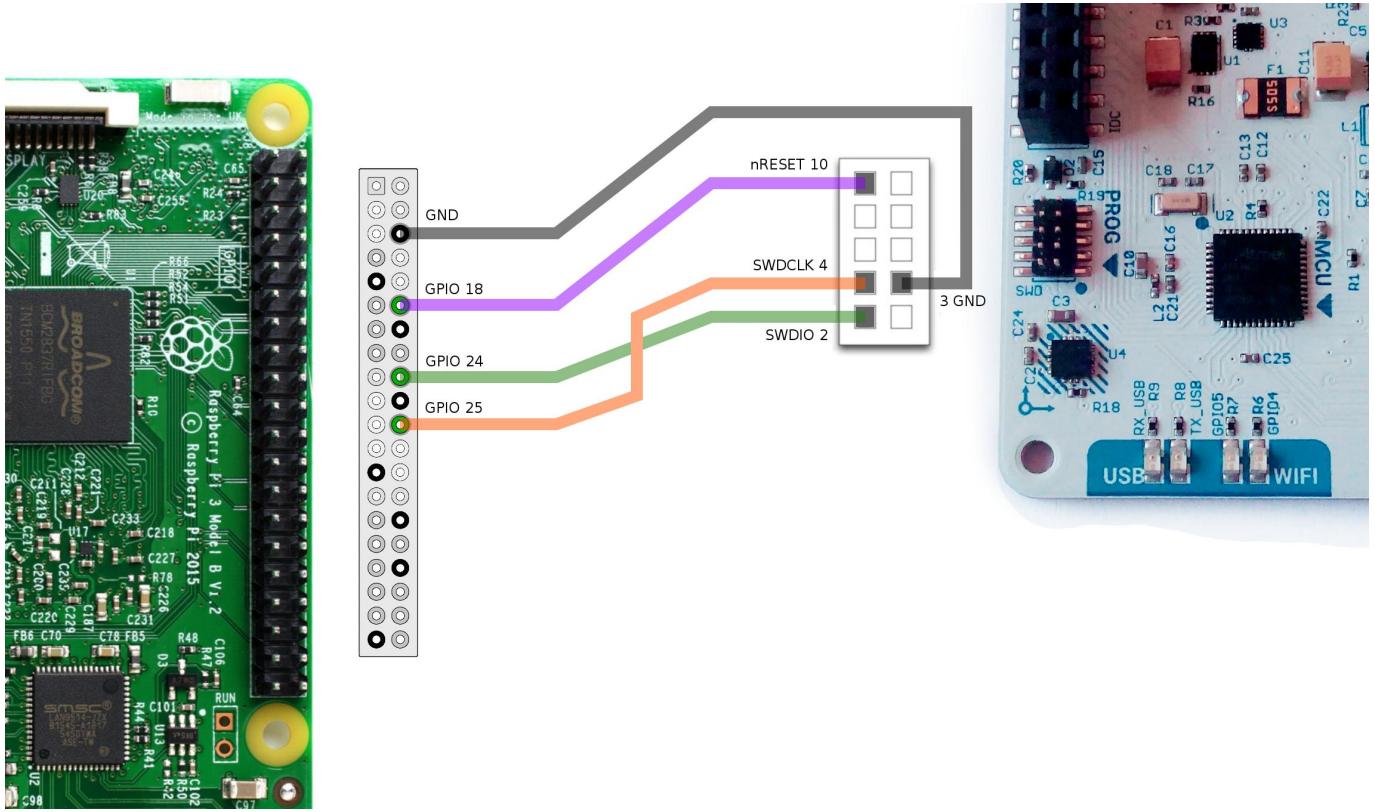
12.4.1 Introduction

Sometimes (many times actually), our code won't do what we want it to do and we need to take a look at what it's doing. By using a **debugger** we will be able to see what is going on *inside* another program while it executes or even crashes. This is fairly straight forward when you code for a modern day computer, since most IDEs have a proper interface integrated for it. However, debugging a chip like the SAMD21 can sometimes be tricky and here is where it's interesting to use a debugging kit.



To keep it simple: our final target is to be able to **interact** with the SAMD21 (or the chip) **while it's executing the program** and tell it to pause the execution, give us the value of some variables and then continue. We will release a fairly extensive report with documentation on this process, but for those interested in reading an overview on how to debug, this post can be a short introduction.

So, here we go! The first item we need is the **Open On-Chip Debugger** (OpenOCD) which provides debugging with the assistance of a **debug adapter**. This adapter is a small hardware module which helps provide the right kind of electrical signaling to the target being debugged. These are required since the debug host, on which OpenOCD runs (i.e. your computer, a Raspberry Pi...) won't usually have native support for such signaling, or the connector needed to hook up to the target.



These adapters are sometimes packaged as discrete dongles, which may generically be called **hardware interface dongles** (and are quite expensive). Some development boards also integrate them directly, which may let the development board connect directly to the debug host over USB (and sometimes also to power it over USB, like the Arduino Genuino Zero). In the case of the **Smart Smart Citizen Kit**, we have a **SWD Adapter** that supports *Serial Wire Debug* signaling to communicate with the ARM core. In our approach, **using a complete open toolchain**, OpenOCD is be running on a Raspberry Pi, and communicating with the SCK's SWD through the GPIO pins of the Pi.

Finally, to be able to actually **see what is going on inside our firmware while it executes**, we need something that is able to read and understand the machine code and hand it over to a human understandable interface. This is where **GDB** kicks in and helps us by:

- Starting our program, specifying anything that might affect its behavior.
- Make our program stop on specified conditions.
- Examine what has happened when our program has stopped.
- Change things in our program, so we can experiment with correcting the effects of one bug and go on to learn about another.

GDB and OpenOCD will be running in a Raspberry Pi hooked up to the SWD interface of the SCK, and we will see what's going on in them from our computer's terminal via SSH. Fairly *simple*, right? Now, we can make some changes to our code, make GDB flash it to the SCK and keep debugging in a completely open toolchain!

12.4.2 Debugger setup using a Raspberry Pi

- First download and copy Raspbian Lite to your SDcard, here are the installation docs.
- Add wifi configuration

Create a file name `wpa_supplicant.conf` on the `/boot` partition of the SD card, the content of this file should looks like this:

```
ctrl_interface=DIR=/var/run/wpa_supplicant GROUP=netdev
update_config=1

network={
    ssid="wifi_ssid"
    psk="wifi_password"
}
```

Replacing `wifi_ssid` and `wifi_password` with your actual wifi network information. The `wpa_supplicant.conf` file will be copied to `/etc/wpa_supplicant/` directory automatically once the Raspberry Pi is booted up.

- Enable SSH server.

SSH access is disabled as default for security reasons. To enable the SSH server when Raspberry Pi is booted up for the first time: create a file called `ssh` with no file extension and no contents, and copy it to the `/boot` partition on the SD card.

- Find your raspberry on the network

In order to find a raspberry pi over the network we can use commands like these:

Linux

```
MY_IP_RANGE=$(ip addr | grep 'state UP' -A2 | tail -n1 | awk '{print $2}') && nmap -sn $MY_IP_RANGE && IP=$(arp -na | grep b8:27:eb | grep -Eo '[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}') && ssh $IP;
```

Mac

```
MY_RANGE=$(ip addr | grep "UP" -A3 | grep '192' -A0 | awk '{print $2}') && nmap -sn $MY_RANGE && arp -na | grep b8:27:eb
```

- SSH login without password:

- If you have never generated a RSA key: `ssh-keygen` without passphrase
- Copy the key to the Raspberry: `ssh-copy-id -i ~/.ssh/id_rsa.pub raspi-address`

Once booted, it will connect to the network. The command above (`MY_IP...`) finds it and logs into it via SSH.

Once you are logged to your raspberry pi and connected to the internet, do a **system upgrade**:

```
sudo apt-get install rpi-update
sudo rpi-update
sudo apt-get update && sudo apt-get dist-upgrade
```

Install some **dependencies**:

```
sudo apt-get install git autoconf libtool make pkg-config libusb-1.0-0 libusb-1.0-0-dev telnet sshfs
```

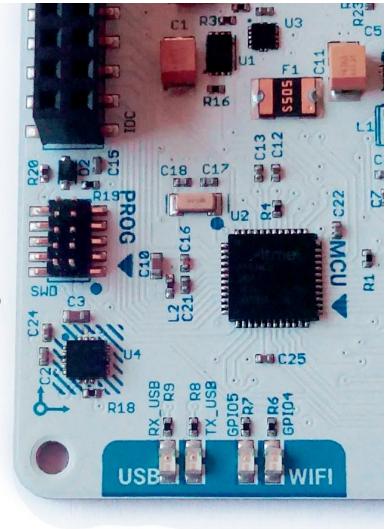
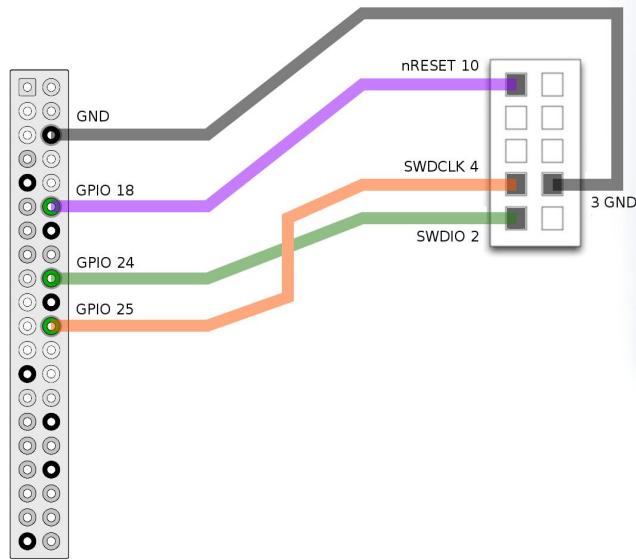
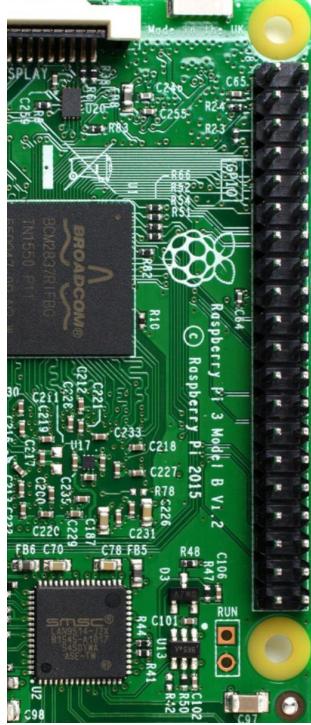
Openocd installation

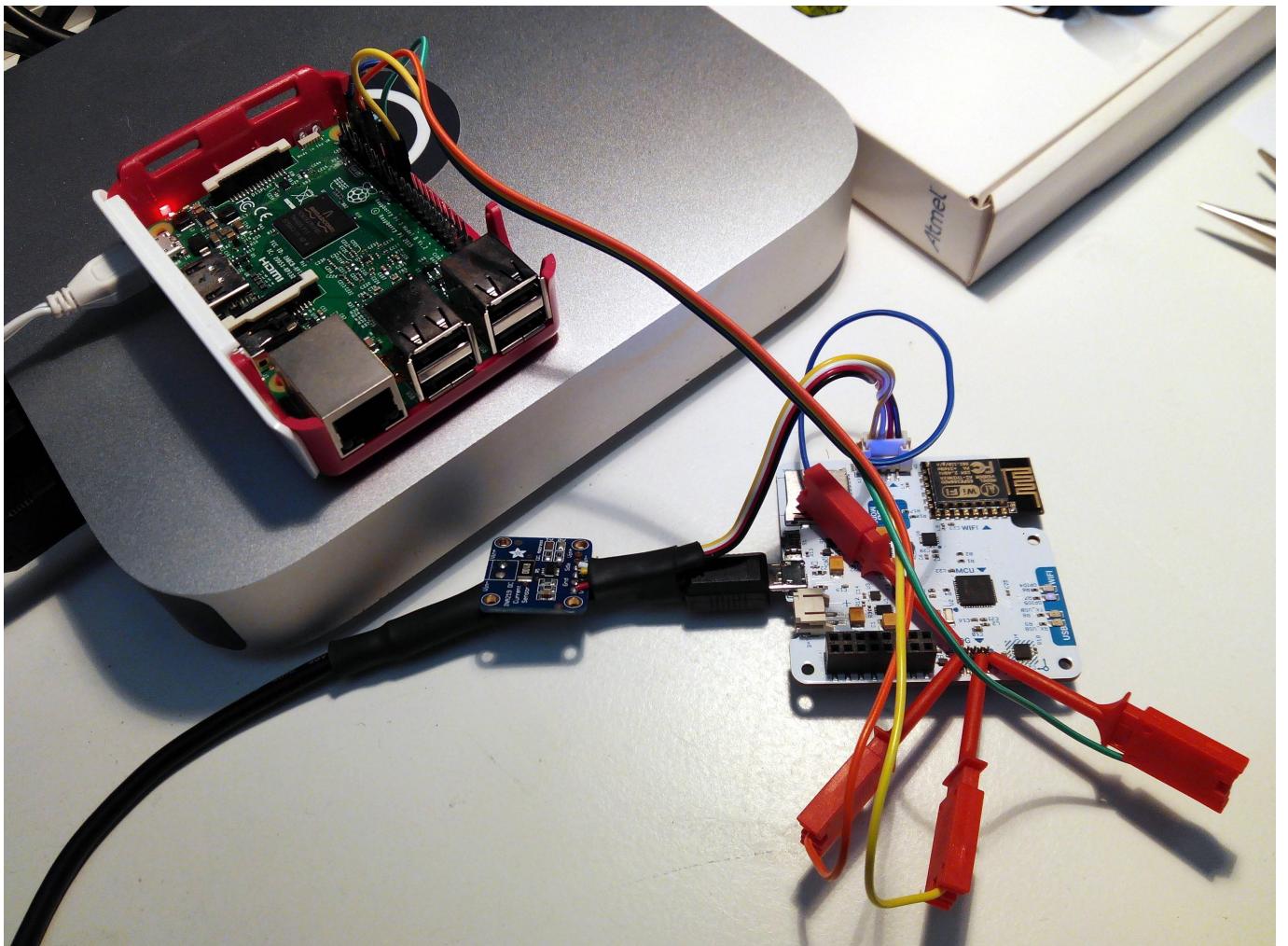
- Clone **openocd** repository and compile:

```
git clone git://git.code.sf.net/p/openocd/code openocd-code
cd openocd-code
./bootstrap
./configure --enable-sysfsgpio --enable-bcm2835gpio
make
sudo make install
```

The list of interfaces that openOCD can use is under: `/usr/local/share/openocd/scripts/interface`.

In order to use the SWD connector that the SCK features, by using *Bit Banging*, we connect it directly to the Raspberry Pi GPIOs:





Running OpenOCD on the raspberry pi

Once you are logged into the raspberry Pi you need a openOCD config file to start (ej. `sck.cfg`) with this content:

```
source [find interface/raspberrypi2-native.cfg]
transport select swd

set CHIPNAME at91samd21g18
source [find target/at91samdXX.cfg]

adapter_nsrst_delay 100
adapter_nsrst_assert_width 100

init
targets
reset halt
```

You can store this file in OpenOCD scripts dir so it will auto find it

```
sudo mv sck.cfg /usr/local/share/openocd/scripts/
```

and then run the OpenOCD server with:

```
sudo openocd -f sck.cfg
```

Then you can connect to OpenOCD, if you want to connect from an external computer, replace `127.0.0.1` with your Raspberry Pi IP address.

```
telnet 127.0.0.1 4444
```

Example

On a **arduino zero** go to the directory where the *.cfg is and:

```
openocd -f arduino_zero.cfg
Open On-Chip Debugger 0.10.0
Licensed under GNU GPL v2
For bug reports, read
  http://openocd.org/doc/doxygen/bugs.html
Info : auto-selecting first available session transport "swd". To override use 'transport select <transport>'.
none separate
adapter speed: 400 kHz
cortex_m reset_config sysresetreq
Info : CMSIS-DAP: SWD Supported
Info : CMSIS-DAP: Interface Initialised (SWD)
Info : CMSIS-DAP: FW Version = 02.01.0157
Info : SWCLK/TCK = 1 SMDIO/TMS = 1 TDI = 1 TDO = 1 nTRST = 0 nRESET = 1
Info : CMSIS-DAP: Interface ready
Info : clock speed 400 kHz
Info : SWD DPIDR 0x0bc11477
Info : at91samd21g18.cpu: hardware has 4 breakpoints, 2 watchpoints
```

12.4.3 Using OpenOCD remotely from other computer

We need to give the OpenOCD server access to your project files that are remotely stored. To do this you can mount your working directory remotely on the Raspberry Pi via SSH:

```
ssh pi@raspi_address
pi$ mkdir working_dir
pi$ sshfs user@computer_address:working_path working_dir
pi$ cd working_dir
pi$ sudo openocd -f sck.cfg
```

Then you can connect to OpenOCD from your computer with:

```
telnet raspi_address 4444
```

Uploading Arduino original bootloader

- Get the bootloader file here and build it.
- Connect to OpenOCD server and run:

```
reset halt
at91samd bootloader 0
at91samd chip-erase
program samd21_sam_ba.bin verify
at91samd bootloader 8192
reset run
```

If you don't see any error you're done!

Uploading SCK Firmware

- Install platformio, download and build SCK firmware
- Connect to OpenOCD server and run:

```
reset halt
flash write_image firmware.bin 8192
verify_image firmware.bin 8192
reset run
reset run
```

12.4.4 GDB

General description

The purpose of a debugger such as GDB is to allow you to see what is going on “inside” another program while it executes—or what another program was doing at the moment it crashed.

GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act: * Start your program, specifying anything that might affect its behavior. * Make your program stop on specified conditions. * Examine what has happened, when your program has stopped. * Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

Debugging session with Raspberry Pi as the OpenOCD server

Once your raspberry pi is setup with above instructions you can just do:

```
ssh pi@RaspberryAddress sudo openocd -f sck.cfg &
cd /platformio_project/path
arm-none-eabi-gdb ./pioenvs/zeroUSB/firmware.elf
(gdb) target remote RaspberryAddress:3333
(gdb) monitor reset run
```

If you are using *platformio*, you need to modify the compiling option to avoid optimisation with -Og message to the compiler. In case you are not using *platformio*, activate verbose compiling output at Arduino IDE and find your compiled .elf directory.

```
[env:zeroUSB]
platform = atmelsam
board = zeroUSB
framework = arduino
build_flags = -Og
```

Now we are all set and ready to go. The debugger is waiting for instructions on the execution, which we detail below.

Info

Quick handy instructions inside GDB environment 1. (gdb) appears in every line and you don't have to type it each time 2. In case you need to exit GDB, just type in `quit`, but remember always killing the process before, should you have a target running

```
(gdb) kill
(gdb) quit
```

3. RET repeats the previous command

GDB commands

All commands in gdb during debugging are detailed in the GDB guide, chapter GDB commands in detail (continue and stepping)

An **extract** of some useful commands are detailed below:

CONTINUING AND STEPPING

`continue [ignore-count]`

- Resumes program execution until next breakpoint. `[ignore-count]` argument allows to specify a further number of times to ignore a breakpoint.

```
(gdb) continue
Continuing.

Breakpoint 1, tick () at src/HOLA.cpp:9
9 void tick() {
```

`step count`

- Continues running your program until control reaches a **different source line**, only available for source lines and functions compiled with debugging information. `count` is optional and states the number of steps to be performed before stopping, if no breakpoint arrives earlier.

`next [count]`

- Continue to the next source line **without going into functions**. It has the same functionality as `step`, but it stays in the same stack frame. `count` works as in `step count`. As well, it understands jumps calls as in the end of `for` loops and return to the beginning of the loop.

Info

`set step-mode on/off` sets the behaviour of `(gdb)` when stepping into a function with no debugging information. In the case of `step-mode on`, it inspects the first line of code of the function, whereas on `step-mode off` it skips the function completely.

`finish`

- Continue running until **just after function in the selected stack frame returns**.

`until`

- Has the same behaviour as `step`, but it **ignores the jumps between lines due to loops** (for, whiles, etc), continuing to the next source code with incremental line number.

BREAKPOINTS

`info breakpoints`

- Retrieve information about breakpoints

```
(gdb) info breakpoints
Num      Type            Disp Enb
ress    What
1       breakpoint     keep y  0x00002140 in tick() at src/HOLA.cpp:9
breakpoint already hit 15 times
```

`break`

Info

Use the `tbreak` command instead of `break` if you want to stop the program once, and then remove the breakpoint. More **breakpoint condition** options can be found [here](#) you can find

watchpoint

- Set a watchpoint **watchpoint** to only stop once a variable has a certain value.

```
(gdb) watch timer
```

Info

Type in `info watchpoints` to get information about watchpoints.

commands

- Set a **list of actions** related to the breakpoint:

```
break main.cpp:50
commands
silent
printf "count is %d\n", count
cont
end
```

delete

- Delete a breakpoint

```
(gdb) delete 1
(gdb) info breakpoints
No breakpoints or watchpoints.
```

PRINTING / SETTING VARIABLES AND MORE**loop**

- Read what is around a certain function

```
(gdb) l loop
25 //While (!Serial) {
26     //; // wait for serial port to connect. Needed for native USB port only
27 //}
28 }
29
30 void loop() {
31 // put your main code here, to run repeatedly:
32 Serial.println("HOLA");
33 tick();
34 Serial.println(millis());
```

print

- Retrieve value of a specific variable

```
(gdb) print timer
$12 = 2
```

set

- Set variable to a certain value

```
(gdb) set timer = 0
```

TARGET COMMANDS (LOAD)

```
load filename offset
```

- Load it is meant to make filename (an executable) available for debugging on the remote system—by downloading it. load also records the filename symbol table in GDB, like the add-symbol-file command. The file is loaded at whatever address is specified in the executable, also into flash memory.

Making changes in the code

Anytime we make a change in the code, we don't need to reload the debugging session. We can easily do so by:

1. Compile the code: a. Define Shell build in Sublime Text and configure a build system with:

```
"shell_cmd": "cd .. && pio run"
```

Then, everytime you hit Ctrl+B (Cmd+B) and you use your custom build system, it will automatically use this option.

- b. Or hit `pio run` in another terminal located in your project root directory

2. In gdb, `load file`. This will reload the file defined at the beginning of your debugging session and upload it to the target

```
(gdb) load
Loading section .text, size 0x2e50 lma 0x2000
Loading section .ramfunc, size 0x60 lma 0x4e50
Loading section .data, size 0x110 lma 0x4eb0
Start address 0x2910, load size 12224
Transfer rate: 3 KB/sec, 4074 bytes/write.
```

3. Keep debugging

GDB Console**TUI**

GDB has a console GUI option available with the command line option `--tui` In the upper frame you can see the code that's being executed.

```

test.cpp
8     string s = "Solution";
9     int a = 1;
10    int b = 2;
11    int c = 3;
12    int d = 4;
13    int e = 5;
14
15    float f = (e * d * 2) / c;
B+> 16    f *= 2;
17
b+ 18        cout << s << "=" << f << endl;
19    }
20

child process 3746 In: main                                         Line: 16   PC: 0x400b67
Starting program: /home/ipp/html/yolinux.com/TUTORIALS/a.out

Breakpoint 1, main () at test.cpp:16
(gdb) print f
$1 = 13
(gdb) break 18
Breakpoint 2 at 0x400b75 in file test.cpp, line 18.
(gdb) 

```

GDB INIT FILE

From this example dashboard we can generate a custom .gdbinit file for the SCK which will be placed in the HOME directory... (ON GOING)

Info

Would be interesting to generate a custom option for production validation and one for internal debugging purposes

For references about where to locate the .gdbinit and more custom behaviour for gdb in general see here.

GDB from Sublime Text

Setup Platformio project with sublime Text Setup sublimeGDB

12.4.5 Reference

General GDB references and examples

Debugging with GDB - Book

Debugging example from GDB and OpenOCD

Arduino zero example

Additional notes from Platformio configuration

1. How to set other DEBUG FLAGS
2. About project configuration with Platformio init
3. Check [here](#) for building an *.ini file with custom build target for debugging and production.

12.5 Downloading the Data

Once you've added your SCK to the platform and it's capturing and sending data correctly, you can interact with the platform in several ways. Visualizing the data, downloading the data and interacting with the data through the API.

12.5.1 Download Data

If you are interested in use the data captured by your sensors, you can download all the data for later use. To do this, go to your device page, at the bottom there is a button called *DOWNLOAD DATA*. You will receive an email with a link to download your data on CSV format in less than a minute,

12.5.2 API

The Smart Citizen API allows you to request back information from your devices and do incredible things with it.

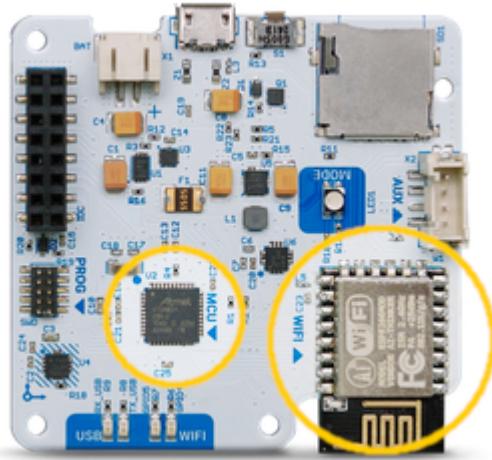
It is a REST API and it returns the information in JSON format. This means you can easily access the information from any language like Javascript, PHP, Processing.org, Python, and start doing things with it quickly.

Code examples

We are working to provide enhanced tutorials on how to interface the API. At the moment you can find some examples on smartcitizen-toolkit repository

12.6 Edit the Firmware

The data board of your SmartSmart Citizen Kit has two **two microcontrollers**:



The main one is an **Atmel SAMD21**, this chip is in charge of all the normal tasks like reading the sensors, saving data, interacting with the user, etc. For this chip we need two software components the bootloader and the main firmware.

For communications the SCK has an **ESP8266 microcontroller with Wifi capabilities**, this chip receives instructions from the SAMD21 via serial port and takes care of publishing the collected data through the network. This chip needs a main firmware and a filesystem that stores the web pages for the setup mode server.

12.6.1 Development environment

The SmartSmart Citizen Kit Firmware is on our repository on github so you will need git software installed.

To build the SmartSmart Citizen Kit firmware you need a linux computer with platformio installed, you don't need the full IDE installation (Atom). You can follow this instructions to install only the console version.

For bootloader upload you also need OpenOCD somewhere in your PATH. You can use platformIO provided binary, normally it is located in `~/platformio/packages/tool-openocd`.

12.6.2 Getting the firmware

To get the firmware just run:

```
git clone --recursive https://github.com/fablabbcn/smartsitizen-kit-21
```

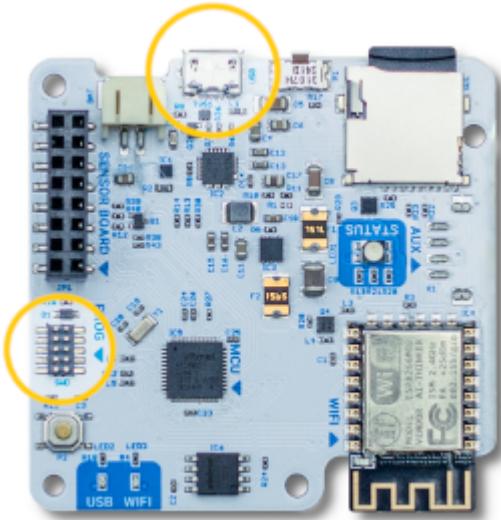
The bootloader repository is a submodule of the main firmware so you must do a `--recursive` clone to get it.

Info

If you download manually (with the *clone* or *download* button on github) you will **not** get the bootloader code, but you can get it from here.

12.6.3 SAMD21 bootloader

If your kit doesn't have the bootloader already flashed (all the kits that we ship come with it) you will need an ATMEL-ICE programmer. This process can also be done with a Raspberry Pi computer and the proper connector and cables, we are preparing the documentation for this process.



Connect the Atmel-ICE programmer to the 10 pin SWD connector and to your computer. Power the SCK via USB, you can use any USB charger or even your computer.

Open a terminal, go to the folder where you cloned the firmware repository and run:

```
cd smartcitizen-kit-20
./build.sh boot
```

```
Flashing SAM bootloader

Remeber to connect SAM-ICE programer to your kit!
Press any key to continue
[ ]
```

If you have everything connected click any key to continue, you will see a lot of output when compiling, the led on the SCK should *breathe* in **green** and you should see an output similar to this:

```

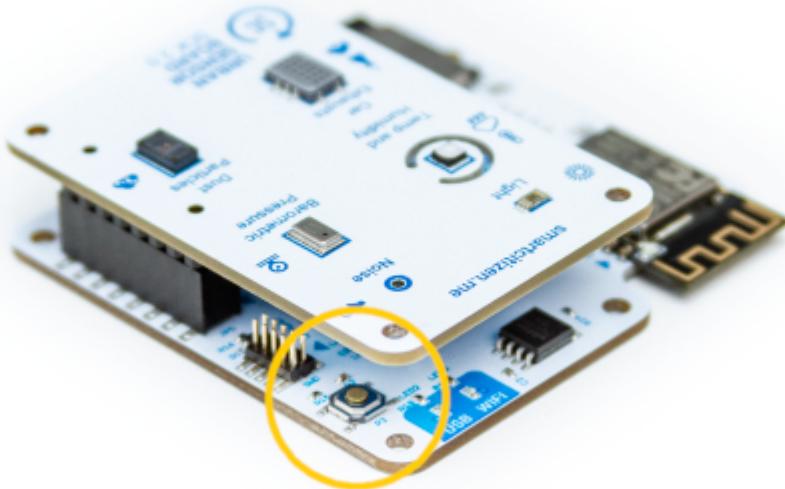
adapter speed: 400 kHz
cortex_m reset_config sysresetreq
Info : CMSIS-DAP: SWD Supported
Info : CMSIS-DAP: JTAG Supported
Info : CMSIS-DAP: Interface Initialised (SWD)
Info : CMSIS-DAP: FW Version = 01.00.0021
Info : SWCLK/TCK = 1 SWDIO/TMS = 1 TDI = 1 TDO = 1 nTRST = 0 nRESET = 1
Info : CMSIS-DAP: Interface ready
Info : clock speed 400 kHz
Info : SWD DPIDR 0x0bc11477
Info : at91samd21g18.cpu: hardware has 4 breakpoints, 2 watchpoints
  TargetName      Type      Endian TapName      State
  -----
  0* at91samd21g18.cpu cortex_m little at91samd21g18.cpu running
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x00000288 msp: 0x20002dd8
chip erased
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0xfffffff8 msp: 0xfffffff8
** Programming Started **
auto erase enabled
Info : SAMD MCU: SAMD21G18A (256KB Flash, 32KB RAM)
wrote 16384 bytes from file ./build/sck2.0/bootloader-sck2.0.bin in 1.670352s (9.579
iB/s)
** Programming Finished **
** Verify Started **
verified 8192 bytes in 0.702667s (11.385 KiB/s)
** Verified OK **
shutdown command invoked

```

You are ready for the next step, just remember to disconnect the Atmel-ICE programmer and connect the SCK to your computer with a USB cable.

12.6.4 SAMD21 firmware

The bootloader we just flashed allows a very simple way of uploading the SCK firmware based on the UF2 format, when you **double-click the reset button** of your kit it will expose a MSD interface to your computer and a new drive will popup where you can just drag the compiled firmware file (converted to UF2 format).



Build script

You can use the same script used to flash the bootloader (`build.sh`) that will do everything for you: compile the firmware, convert the binary to UF2 format and upload it to the kit:

```
./build.sh sam
```

```
Flashing SAM

Remember to double click the reset button of your kit!!!
Press any key to continue...
```

If you haven't already, **double-click the reset button of your kit** or click any key. If this is your first time building the software, platformio will take a while installing all the needed dependencies, be patient. If there are no errors you should see an output similar to this:

```
Current build targets ['buildprog', 'size']
Compiling .pioenvs/sck2/src/SmartCitizenKit.ino.cpp.o
Linking .pioenvs/sck2/firmware.elf
Calculating size .pioenvs/sck2/firmware.elf
Building .pioenvs/sck2/firmware.bin
text      data      bss      dec      hex filename
120120      592    12052   132764    2069c .pioenvs/sck2/firmware.elf
===== [SUCCESS] Took 6.10 seconds =====

Converting to uf2, output size: 241664, start address: 0x2000
Wrote 241664 bytes to SAM_firmware.uf2.
Flashing /media/vico/SCK-20 (SAMD21G18A-sck2-v2)
Wrote 241664 bytes to /media/vico/SCK-20/NEW.UF2.
```

The script will leave a copy of the compiled software in UF2 format called `SAM_firmware.uf2` you can use this file to reflash your kit without compiling it again.

Manual install

If you want to install the firmware manually (or you had some problem with the build script) just follow this steps:

```
cd sam
pio run
```

At the end you should see some output similar to this:

```
Compiling .pioenvs/sck2/src/SmartCitizenKit.ino.cpp.o
Linking .pioenvs/sck2/firmware.elf
Calculating size .pioenvs/sck2/firmware.elf
Building .pioenvs/sck2/firmware.bin
text      data      bss      dec      hex filename
120120      592     12052   132764    2069c .pioenvs/sck2/firmware.elf
===== [SUCCESS] Took 5.96 seconds =====
```

then do:

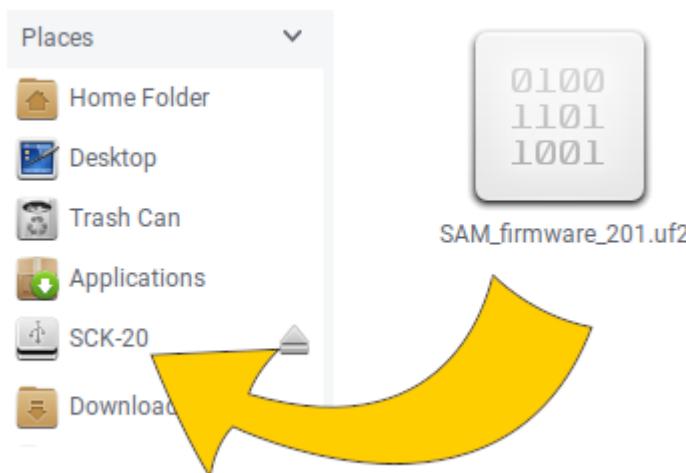
```
cd ..
tools/uf2conv.py -o SAM_firmware.uf2 sam/.pioenvs/sck2/firmware.bin
```

you should see:

```
Converting to uf2, output size: 241664, start address: 0x2000
Wrote 241664 bytes to SAM_firmware.uf2.

NO DEVICE WAS FLASHED.
```

don't worry about the **NO DEVICE WAS FLASHED** message, we are doing it manually. Now **double-click the reset button of your kit** open your favorite file browser and drag the file you just created to the SCK-2.0 drive. The kit will reset and run the new firmware.



Info

Keep in mind that if your computer is not configured to automount new drives you will need to mount your sck manually (as any other USB drive).

12.6.5 ESP8266 firmware

Just like the other parts of the process this is also covered by our `build.sh` script. So you can just do:

```
./build.sh esp
```

As before, if this is the first time you do it, it will take a while on downloading dependecies and building the firmware.

In this case the upload process is different, since the ESP8266 chip is not connected to the USB interface the data must be uploaded through the SAMD21 chip. Our upload script takes care of searching for a SCK on the USB bus, sending a command to the kit so it puts himself in what we call *bridge mode* (white led) and uploading the firmware. This is the expected output:

```
Building .pioenvs/esp12e/firmware.bin
before_upload(["upload"], [".pioenvs/esp12e/firmware.bin"])

Searching for a Smartcitizen kit...
Smartcitizen kit found on /dev/ttyACM0
Asking for upload bridge...
Configuring upload protocol...
Looking for upload port...
Use manually specified: /dev/ttyACM0
Uploading .pioenvs/esp12e/firmware.bin
Uploading 363344 bytes from .pioenvs/esp12e/firmware.bin to flash at 0x00000000
..... [ 22% ]
..... [ 45% ]
..... [ 67% ]
..... [ 90% ]
..... [ 100% ]
after_upload(["upload"], [".pioenvs/esp12e/firmware.bin"])
All good!!!
===== [SUCCESS] Took 42.58 seconds =====
```

Info

Sometimes the ESP8266 and the uploader software don't get synced and the upload fails. Normally if you try again it will work.

12.6.6 ESP8266 filesystem

This process is very similar to the previous one you just need to add the letters `fs`, and wait a little longer ;)

```
./build.sh espfs
```

```
Current build targets ['uploadfs']
Building SPIFFS image from 'data' directory to .pioenvs/esp12e/spiffs.bin
/vue.min.js
/index.html.dev
/css.css
/index.gz
/index.html
/images/sck.png
/main.js
/favicon.ico
before_upload(["uploadfs"], [".pioenvs/esp12e/spiffs.bin"])

Searching for a Smartcitizen kit...
Smartcitizen kit found on /dev/ttyACM0
Asking for upload bridge...
Looking for upload port...
Use manually specified: /dev/ttyACM0
Uploading .pioenvs/esp12e/spiffs.bin
Uploading 1028096 bytes from .pioenvs/esp12e/spiffs.bin to flash at 0x00300000
[    7% ]
[   15% ]
[   23% ]
[   31% ]
[   39% ]
[   47% ]
[   55% ]
[   63% ]
[   71% ]
[   79% ]
[   87% ]
[   95% ]
[  100% ]
after_upload(["uploadfs"], [".pioenvs/esp12e/spiffs.bin"])
All good!!!
===== [SUCCESS] Took 103.51 seconds =====
```

12.7 How to install the framework

The following data analysis framework is a set of tools built on Python 3.7 to help you analyse your data. It can be used with Jupyter Notebooks or Jupyter Lab, although it is not mandatory.

12.7.1 Prerequisites

We recommend you use these two tools for managing the different versions of the framework and keep it updated.

1. Download and install `git` from here
2. Download and install `Anaconda` for Python 3.7 from here

12.7.2 Installation

Open your favourite shell on the directory you have your project. (`cmd.exe` on windows)

Get the repo

Make a directory and `clone` the repository in it:

```
→ mkdir data_analysis
→ cd data_analysis
→ git clone https://github.com/fablabbcn/smartcitizen-iscape-data.git
...
```

Want to stay up-to-date?

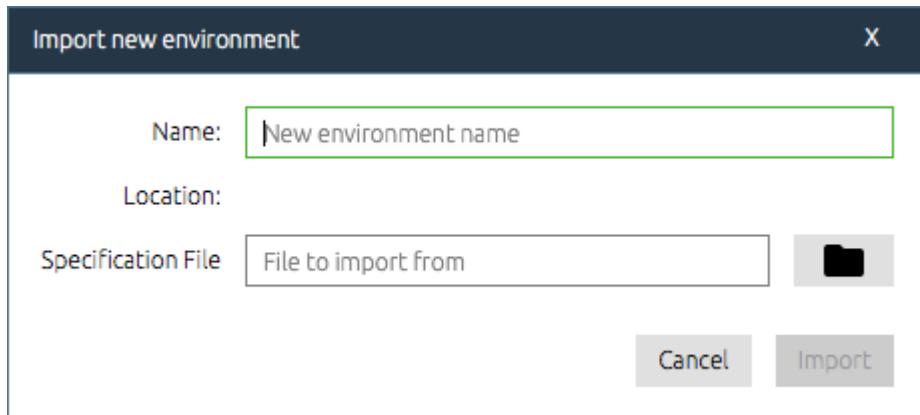
The framework is being constantly updated and the only version that will always be up-to-date is in the master branch of the github repository. We do not recommend to simply download the repository but to clone it with `git`. If you want to learn more about `git` and why it can help you in your projects, check here

Create the environment

Navigate to the cloned repository and create an environment for anaconda. We provide an `environment.yml` file to help with the process:

```
→ cd smartcitizen-iscape-data
→ smartcitizen-iscape-data git:(master) x conda env create -f environment.yml
```

Alternatively, you can open Anaconda Navigator and `Import` an environment in the `Environment` section:



Note about different platforms

Different platforms (Windows, Mac, Linux...) may have different dependencies for each package. We have tried to make the environment as clean as possible, but there might be still some `ModuleNotFoundError`s. Please, use the issue tracker in the project to help us improve!

Finish it up

The code in the framework is managed as internal dependencies. To activate this, you can run:

```
→ pip install --editable . --verbose
```

Additional commands to install jupyter lab extensions are given in the `.dotfile`:

```
jupyter labextension install @jupyter-widgets/jupyterlab-manager@1.0
jupyter labextension install @jupyterlab/toc
jupyter labextension install jupyterlab-plotly@1.0.0
conda install -c conda-forge jupyter_nbextensions_configurator
```

With an optional one for plotly chart studio:

```
jupyter labextension install jupyterlab-chart-editor@1.2
```

You can run it by:

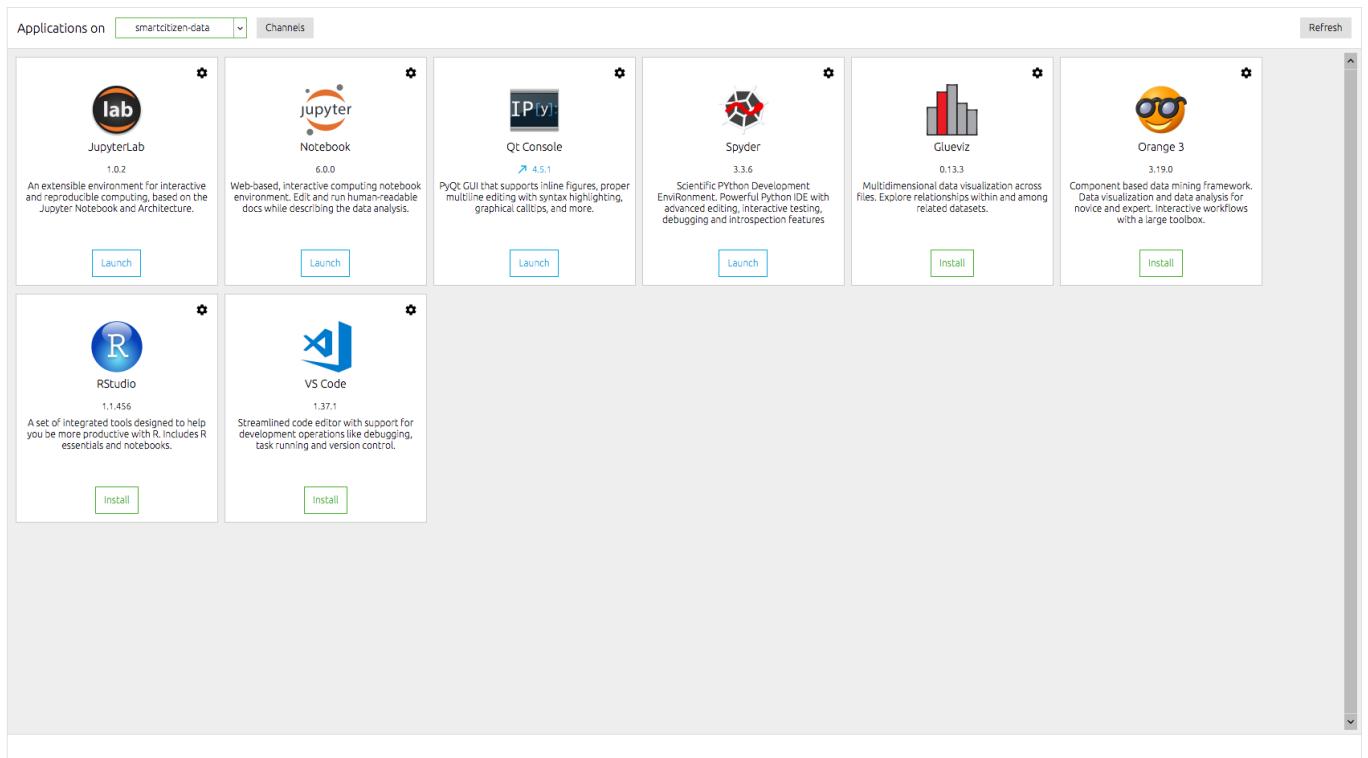
```
→ chmod +x .dotfile
→ ./dotfile
```

12.7.3 Run

You should now be ready to go! You can verify the installation by either running this command:

```
(smartcitizen-data) → ~ jupyter lab --version
1.0.2
```

Or opening Anaconda Navigator:



To run the framework, in your terminal type:

```
(smartcitizen-data) → smartcitizen-iscape-data git:(master) ✘ jupyter lab
```

This will open a web browser instance (by default localhost:8888/lab) which gives access to the tools in the framework.

Learn More

Still wondering what this is? Read this introduction to Jupyter

12.8 Make reports of your data

Tools are provided to generate test or analysis reports, with a custom template. These are generated with the `jupyter nbconvert` using the preprocessor and tools in the `notebooks` and `template` folder. To generate a report, follow the steps:

1. Tag the cells in your notebook. You can use the Jupyter Lab Celltags extension. Don't tag the cells you want to hide, and tag the ones you want to show with `show_only_output`. This can be changed and add more tags, but we keep it this way for simplicity
2. Go to the notebooks folder:

```
cd notebooks
```

3. Type the command:

```
jupyter nbconvert --config sc_nbconvert_config.py notebook.ipynb --sc_Prepocessor.expression="show_only_output" --to html --TemplateExporter.template_file=../templates/full_sc --output-dir=../reports --output=OUTPUT_NAME
```

Where:

- `sc_nbconvert_config.py` is the config
- `notebook.ipynb` is the notebook you want
- `"show_only_output"` is a boolean expression that is evaluated for each of the cells. If true, the cell is shown
- `../templates/full_sc` is the default template we have created
- `../reports` is the directory where we will put the `html` report
- `OUTPUT_NAME` is the name for the export

This generates an `html` export containing only the markdown or code cell outputs, without any code. Examples can be found in the source code repository.

Don't like the template?

You can modify these templates in the `templates` folder

And here is the result!

Analysis Report



Summary

Test name: 2019-09_INT_DELIVERIES_SEPTEMBER

Number of devices tested: 25

Test date: 2019-09-19

Test author: Óscar González

Warnings

No warning for this test

Conclusions

Devices show normal behaviour.

- One faulty device for temperature/humidity sensor (SHT31) detected (9967)
- One device had the PM sensor unplugged (9977)
- One device was tested further to assess learning period of CSS811 (9974)

Time Series Plots

Min Date available: None

Max Date available: None

12.9 Onboarding Sensors

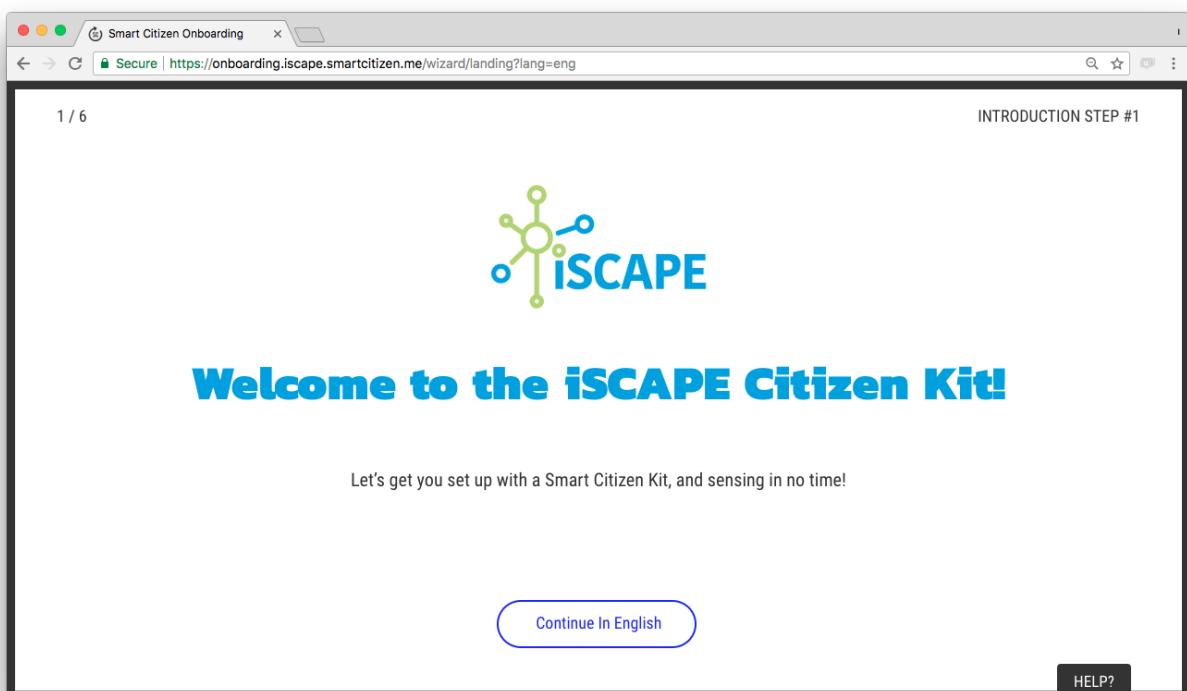
The onboard app guides you through the process of the setup using simple language and a friendly graphic language. It is built as a separate tool from the core Smart Citizen Webpage in order it can be customized for each deployment.

Onboarding app

Visit the onboarding app at start.smartcitizen.me

 Step by step

1. Welcome page



2. Select all the parts you received to ensure you are not missing any part

Smart Citizen Onboarding https://onboarding.iscape.smartcitizen.me/wizard/kit_parts

2 / 6 WHAT'S IN THE BOX STEP #2

CLICK ON ALL THE THINGS YOU HAVE RECEIVED

We need to know this to make the set up work smoothly

Sensor Board Hardware Board Battery Charging Cable

HELP?

Smart Citizen Onboarding https://onboarding.iscape.smartcitizen.me/wizard/kit_parts

2 / 6 WHAT'S IN THE BOX STEP #2

CLICK ON ALL THE THINGS YOU HAVE RECEIVED

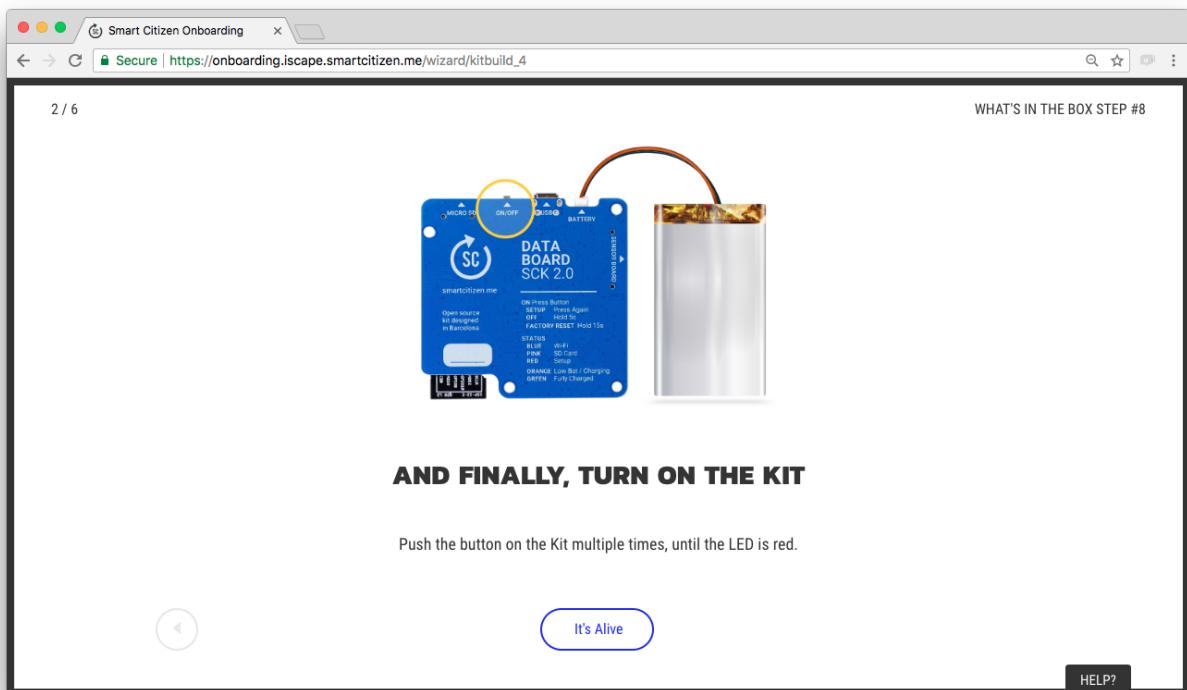
We need to know this to make the set up work smoothly

Sensor Board Hardware Board Battery Charging Cable

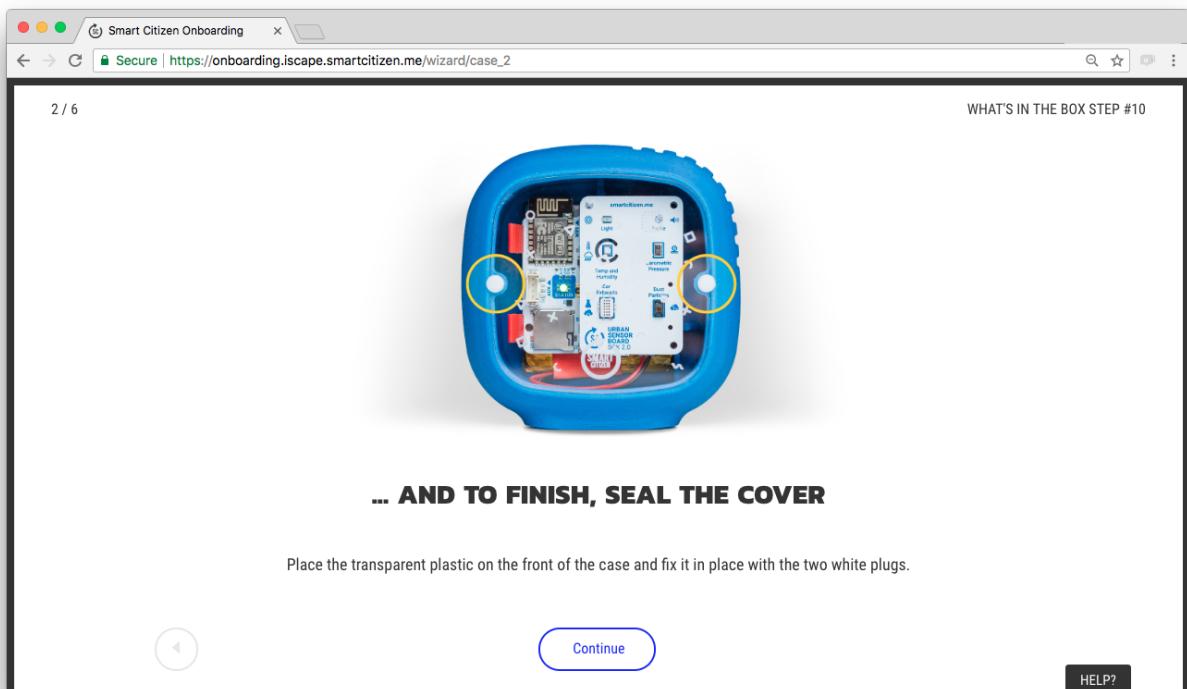
Are You Missing Parts? Yes No

HELP?

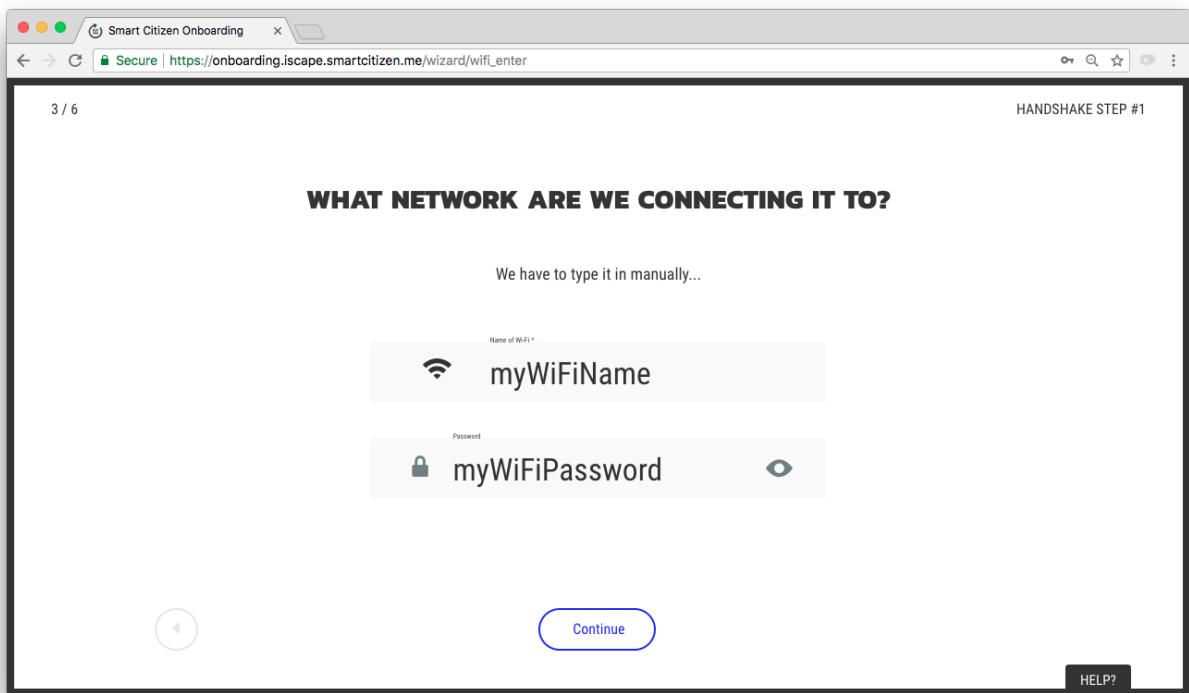
3. Turn on your Kit



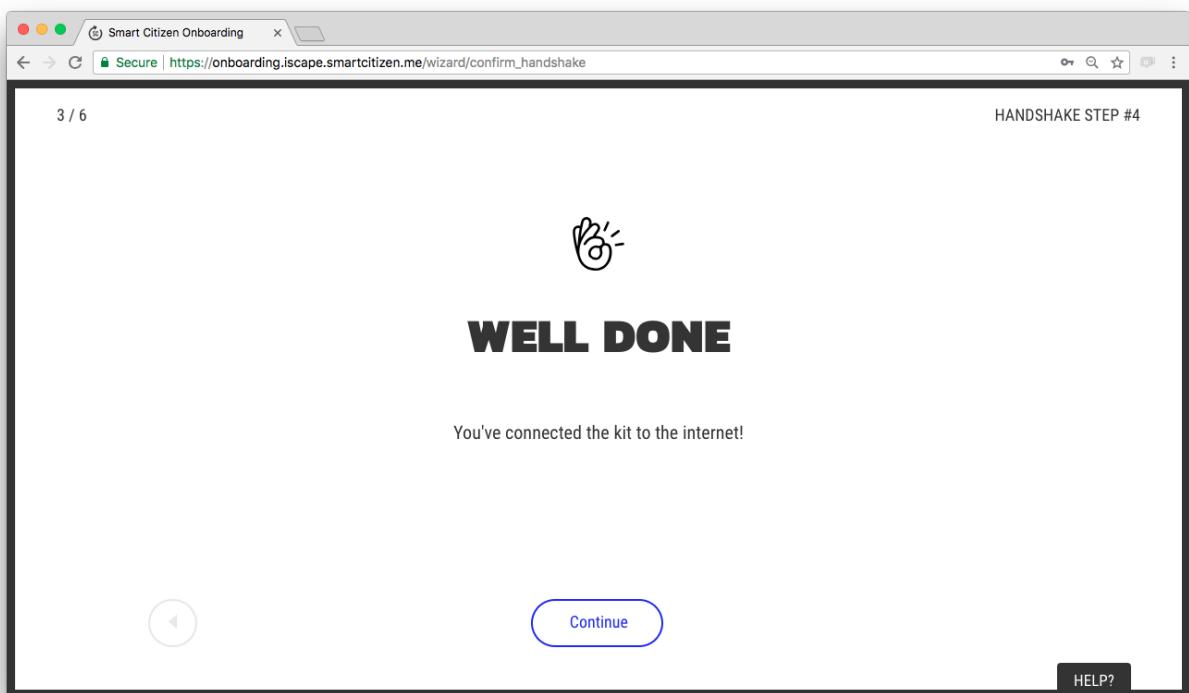
4. Close the cover of your device



5. Choose the Wi-Fi network you want to connect to



6. You will receive a message when the Kit it is connected



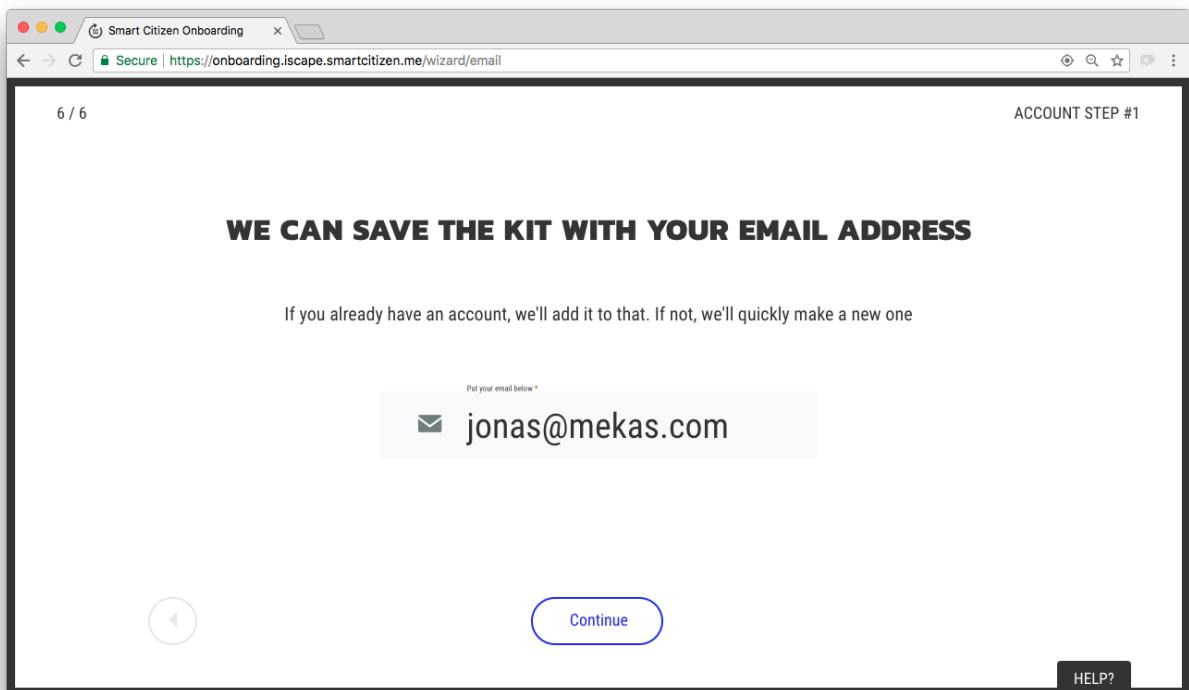
7. Add a name to your sensor

The screenshot shows a web browser window titled "Smart Citizen Onboarding". The URL is <https://onboarding.iscape.smartcitizen.me/wizard/sensorName>. The page is labeled "NAMING STEP #2". At the top, it says "4 / 6". Below that is a large heading "WHAT SHALL WE NAME THE SENSOR?". A sub-instruction reads "You can name it pretty much anything. This is how it will appear on the Smart Citizen map." There is a text input field with a placeholder "Enter the sensor name *". Inside the field, the text "Maria Sensor" is entered, preceded by a user icon. Below the input field is a button labeled "CHOOSE A RANDOM NAME". At the bottom of the form are two buttons: a circular "Done" button and a rectangular "HELP?" button.

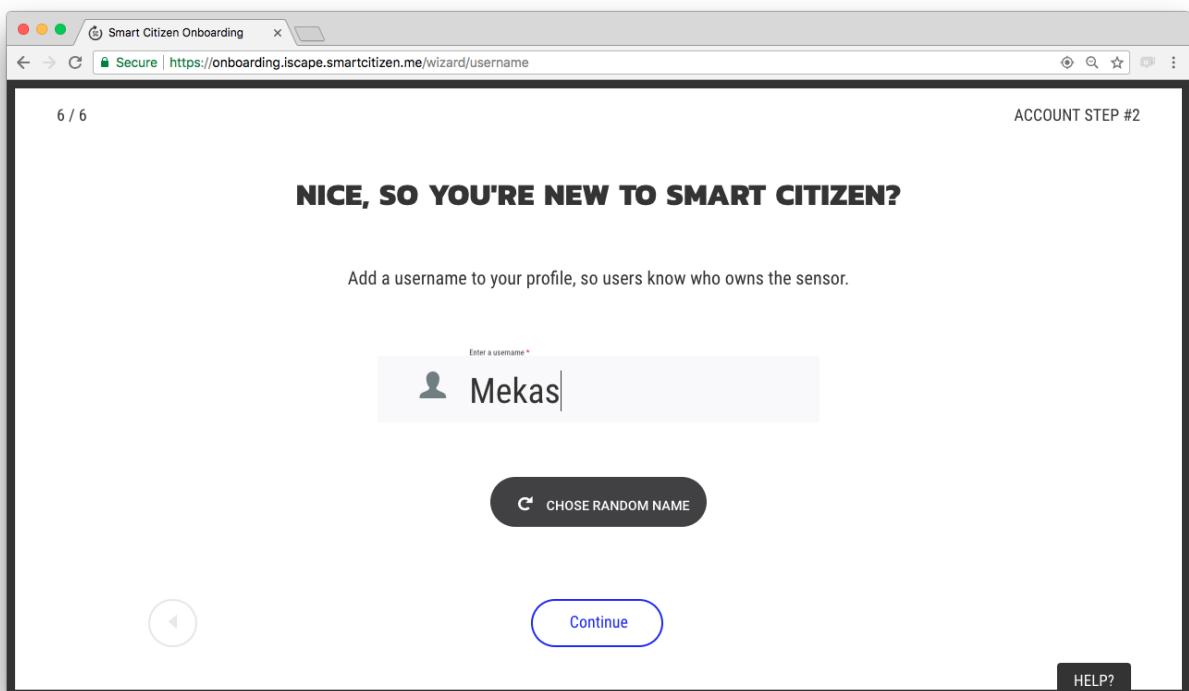
8. Select the location for your sensor

The screenshot shows a web browser window titled "Smart Citizen Onboarding". The URL is https://onboarding.iscape.smartcitizen.me/wizard/location_map. The page is labeled "LOCATION STEP #2". At the top, it says "5 / 6". Below that is a heading "IF YOU WANT TO ADJUST IT OR PIN IT ELSEWHERE, YOU CAN DO THAT HERE". A sub-instruction reads "Remember wherever your device goes, it will need wi-fi. Otherwise you'll have to go get it every now and again and connect it to your computer to sync the data". Below this is a map interface. The map shows the University College Dublin (UCD) campus in Belfield, Dublin 4, Ireland. A red pin is placed on the main building complex. The map includes labels for the Health Science Building, UCD Science Centre North, Computer Science, UCD School of Veterinary Medicine, UCD Swimming Pool, UCD Cinema, UCD Students' Union, UCD Science Centre (West), and UCD National Virus Reference Laboratory. There are also green and blue shaded areas representing different zones or coverage areas. The map has "Map" and "Satellite" tabs at the top left. At the bottom right of the map is a "HELP?" button.

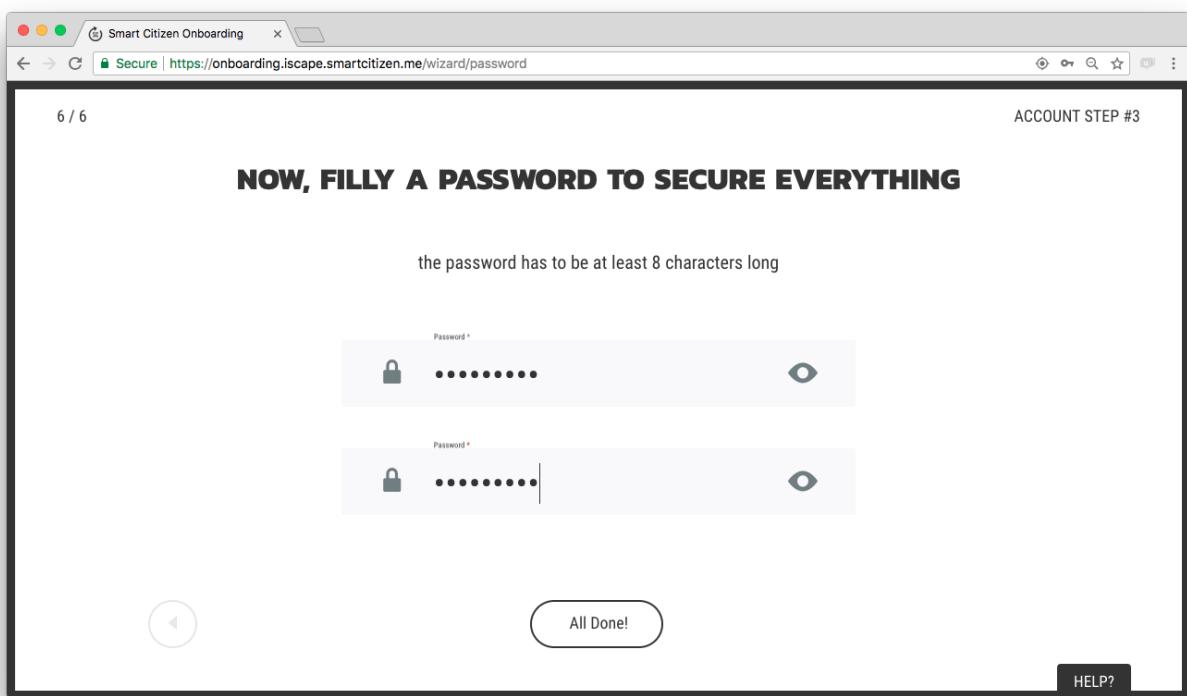
9. Add your email to register the Kit under your name



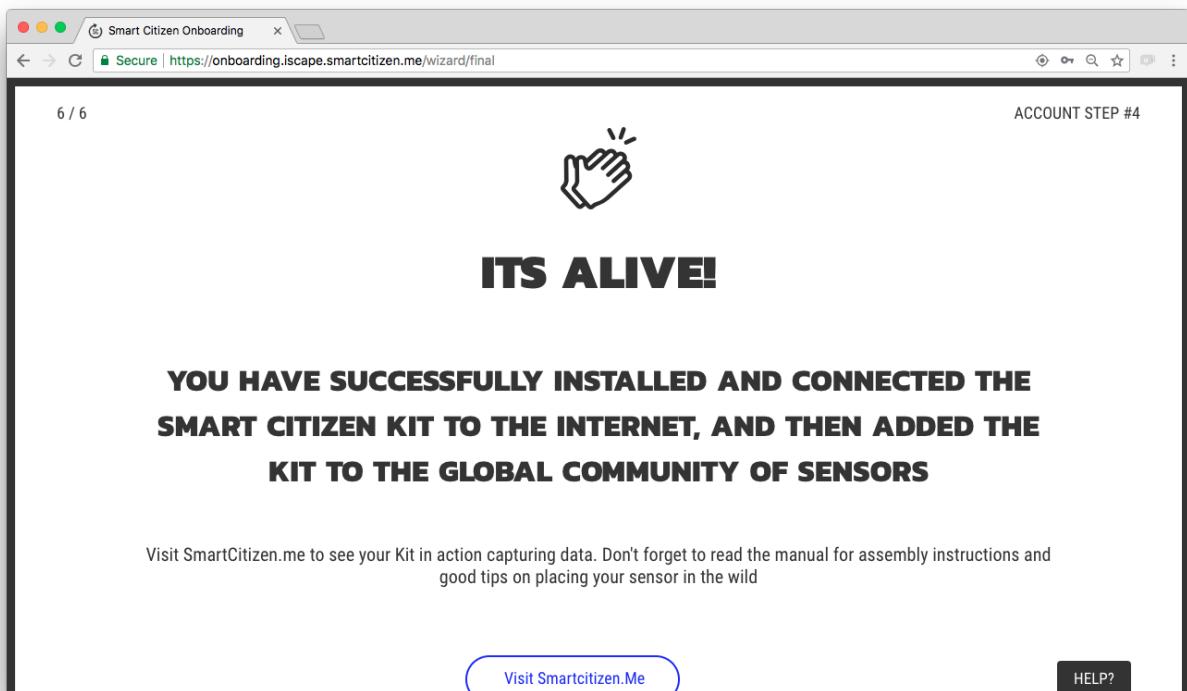
10. Add a user name for people to see you on the platform



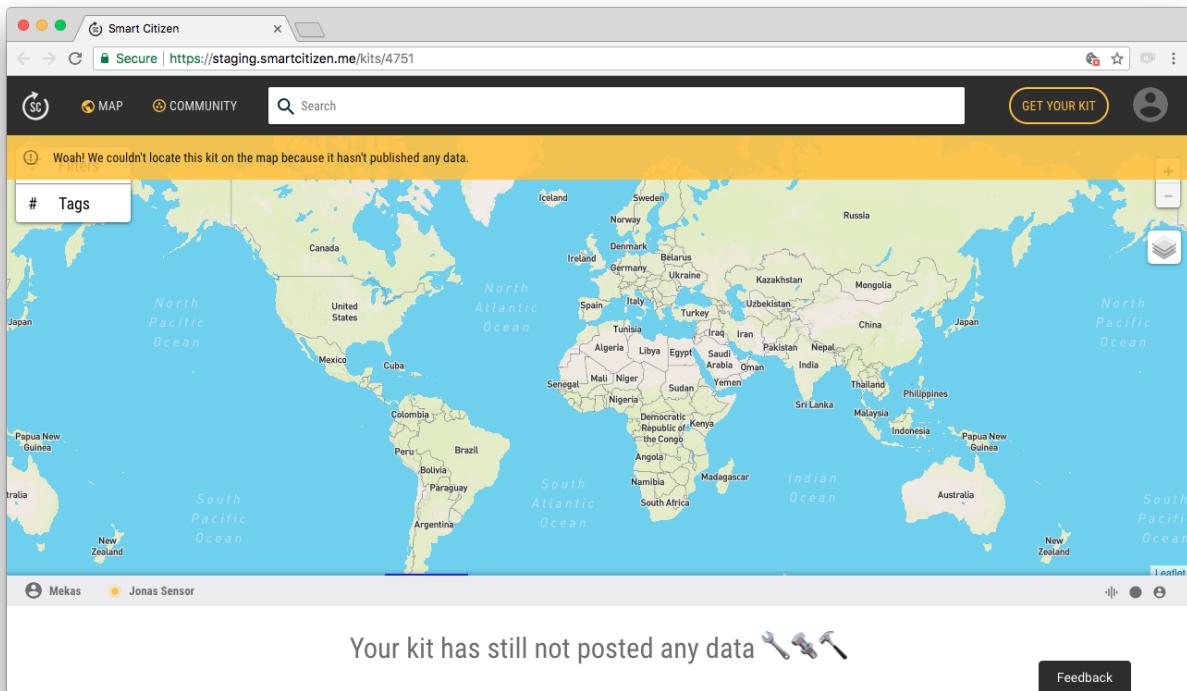
11. Add a password to protect your account



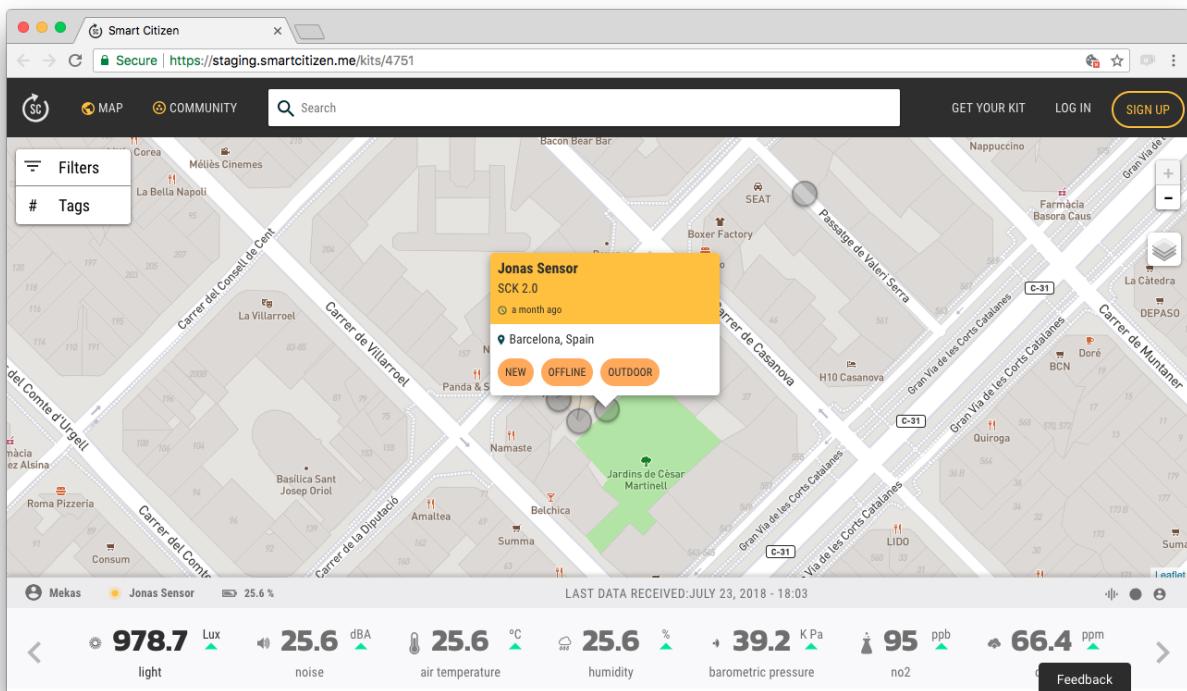
12. You are done!



13. Visit the Kit on the platform. Wait one minute till it publishes data



14. Look at the data!

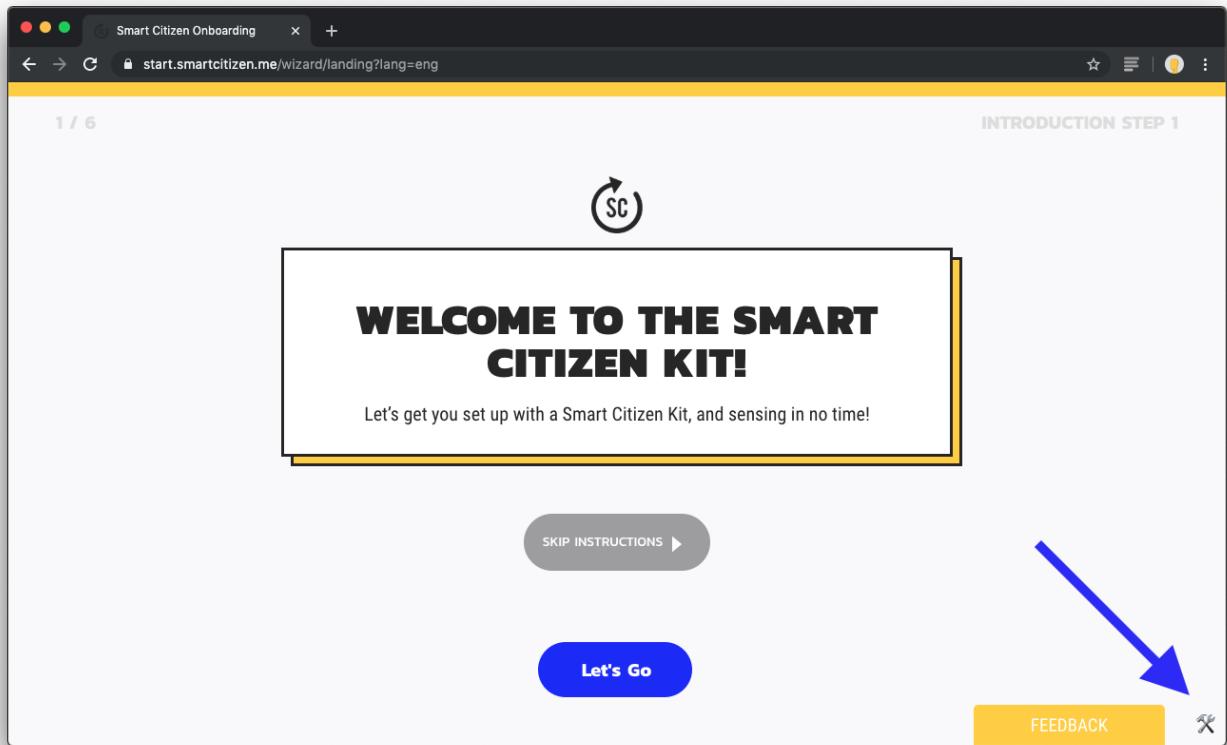


12.9.1 Advanced Kit Selection

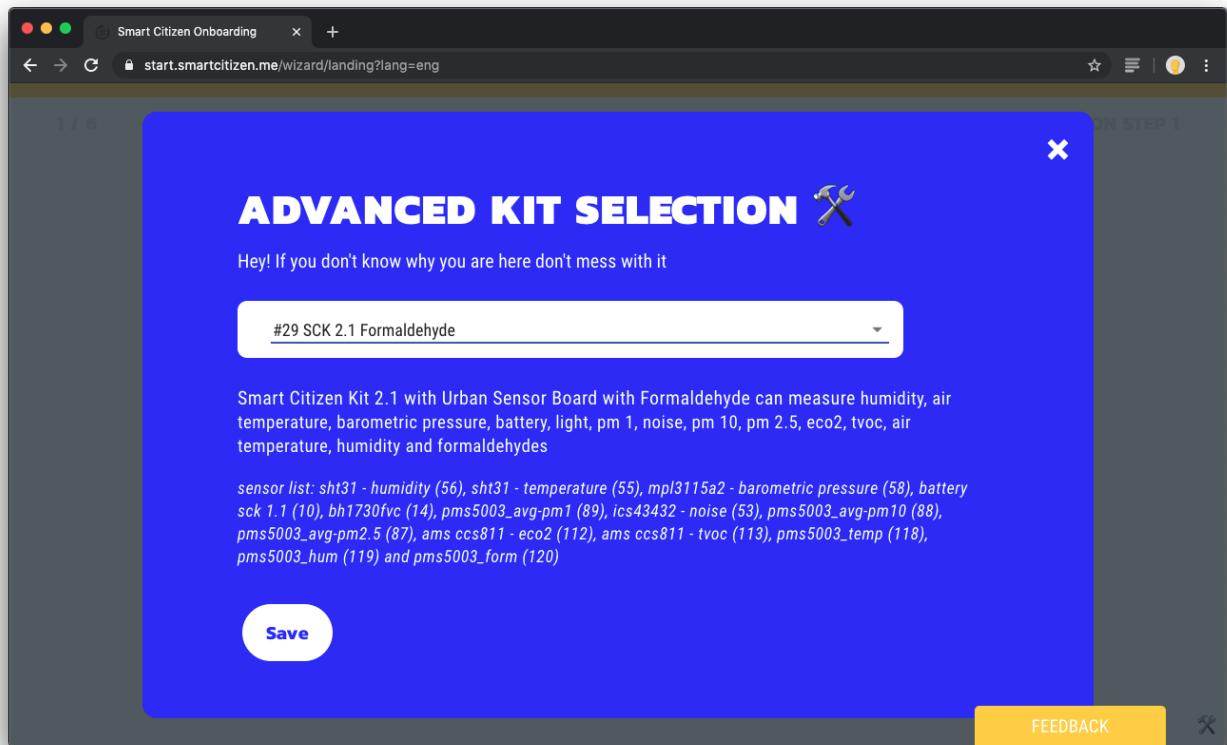
The feature below is only recommended for advanced users who know how to configure new or customized sensor devices

How to choose a custom

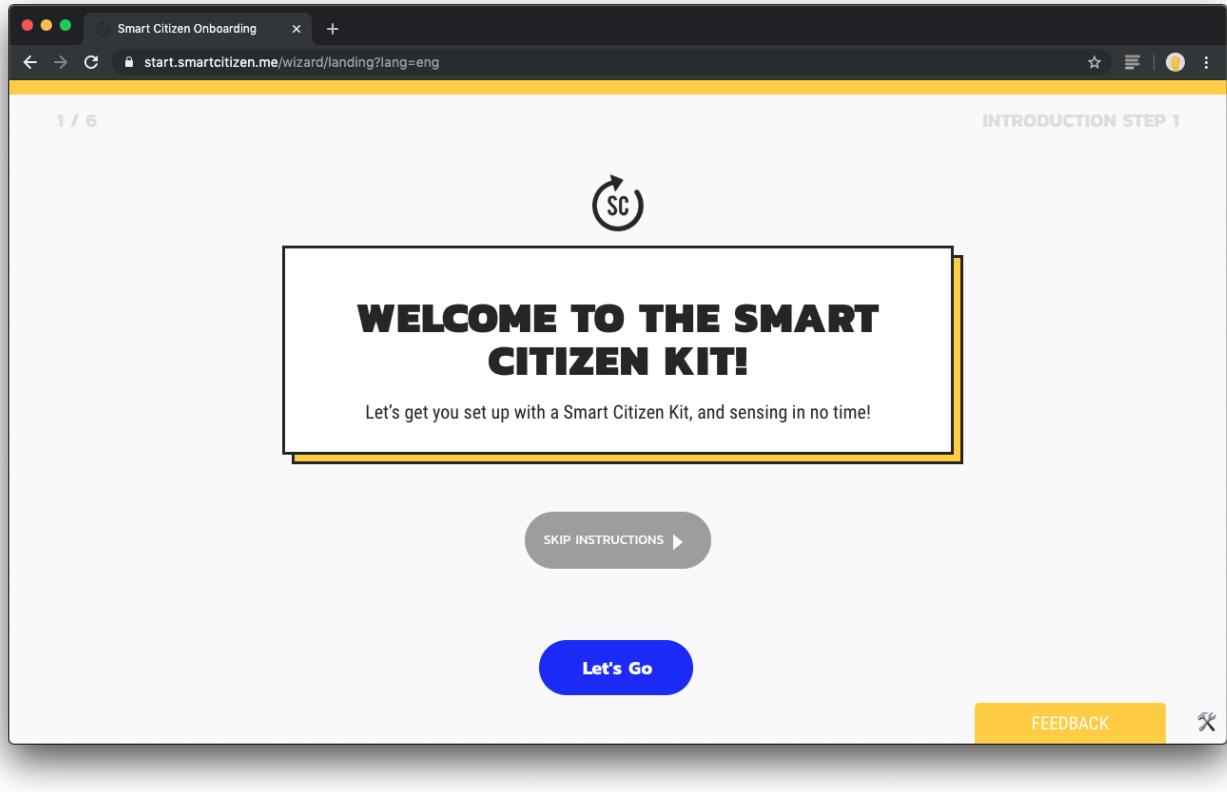
1. Click the  icon in the bottom right corner



1. Choose the blueprint of the device you want to setup



1. Click save and continue the process as usually



12.10 Organise your data

Step by step

Go through the installation guide first before jumping into this guide.

When downloading the framework, the following folder structure is copied:

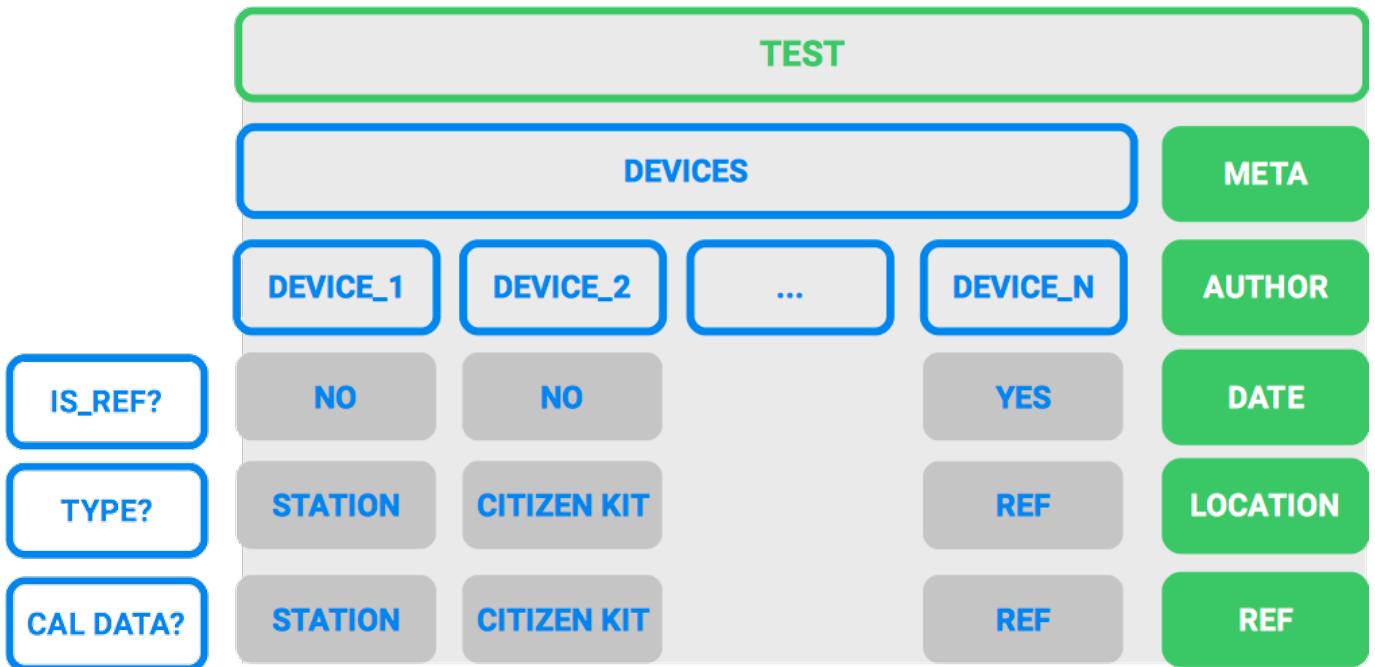
```
└── environment.yml
└── LICENSE
└── README.md
└── src.egg-info
└── data
    ├── export
    ├── interim
    ├── processed
    ├── raw
    ├── reports
    └── scripts
└── docs
└── models
└── notebooks
└── references
└── src
└── tasks
```

The `data` directory will be the place where we will organise our data. Each of the sub-directories is explained below:

- `export` : we will export processed data or images here
- `interim` : we will place files used internally for the framework, such as sensor calibration data
- `processed` : the most important folder of our data directory. This will contain a hierarchy with the different tests after loading into our database. The tests in this folder will have an unified format to easily navigate them. It's the folder we will load our data from, once we have created a test.
- `raw` : raw sdcard data can be placed here. It will feed our test creation script below
- `scripts` : a compilation of handy scripts to parse excel files, concatenate csvs and more

12.10.1 The test

we will organise our data in tests. The concept is very simple. A test is a collection of devices (SCKs or not), that have a common purpose. They can be located in the same spot (co-location) or not, and the main idea is that we centralise the data in a common instance for easy analysis.



Tests have dates, authors, can have comments (for us to remember what we did), a report attached and can be in different conditions (outdoor, indoor)... We will put all this metadata in a `.yaml` file to help use look for the tests later on.

Tests can also have different sources. They could be csv files, xls files or data from the Smart Citizen API. For the local files, we will first pre-process them with the scripts in the `data/scripts` folder and then place them in the `data/raw` folder, in csv format. We use this format because it's a common usable format and, although it is not the most efficient way of archiving the data, it can be easily explored by other common applications.

Pre-process the sd card data

In order to make our files usable, we will need to have them in a format like `YY-MM-DD.CSV`. However, if the kit has been reset, we can find some files like: `YY-MM-DD.01`, `YY-MM-DD.02` and they should be something like `YY-MM-DD_01.CSV`, `YY-MM-DD_02.CSV` ...

Pro-tip

We can rename them manually, but if we have many of them, we can use this one-liner (works in oh-my-zsh):

```
autoload -Uz zmv
zmv '(*).(0*)' '$1_${2}.CSV'
```

We can now concatenate all the sd card data from a device with the `concat_script.py` in the `data/scripts` folder.

We can access the script's help by typing in the terminal:

```
(smartcitizen-data) → scripts git:(master) ✘ python concat_script.py -h
usage: concat_script.py [-h] [--output OUTPUT] [--index INDEX] [--keep]
                        [--ignore IGNORE] [--directory DIRECTORY]

optional arguments:
  -h, --help            show this help message and exit
  --output OUTPUT, -o OUTPUT
                        Output file name, including extension
  --index INDEX, -i INDEX
                        Final name of time index
  --keep, -k             Keep full CSV header
  --ignore IGNORE, -ig IGNORE
                        Ignore files in concatenation
  --directory DIRECTORY, -d DIRECTORY
                        Directory for csv files to concatenate
```

Create a directory (`FILES`) and put the sd card data of one device in it. Then, run the script with:

```
(smartcitizen-data) → scripts git:(master) ✘ python concat_script.py -k -d FILES
Using files in /Users/macoscarr/Documents/04_Projects/02_FabLab/01_SmartCitizen/01.Repositories/DataAnalysis/smartcitizen-iscape-data/data/scripts/FILES
Files to concat:
18-11-08.CSV
18-11-09.CSV
18-11-13.CSV
18-11-14.CSV
18-11-15.CSV
18-11-16.CSV
18-11-17.CSV
18-11-18.CSV
Updating header
Saving file to: Log_Concat.csv
```

And if we now navigate to the `FILES` directory:

```
(smartcitizen-data) → FILES git:(master) ✘ ls
18-11-08.CSV  18-11-13.CSV  18-11-15.CSV  18-11-17.CSV  Log_Concat.csv
18-11-09.CSV  18-11-14.CSV  18-11-16.CSV  18-11-18.CSV
```

We can find our concatenated file!

Create a test

Once we have all our concatenated files, we can proceed to create our test. For this, you can launch `jupyter lab` and open the `examples/test_creation.ipynb` notebook.

In it, we can follow the instructions in the notebook by filling up the input data.

A note about the timestamps

To synchronise our tests, we will **always** need to specify the location. This means that all our tests will be in 'UTC'.

TEST INFORMATION

```
-----
# Is you are creating a new test (True) or if the test is going to be updated (False)
isnewtest = True
# Date when you performed the test in 'YYYY-MM-DD' format
date = '2020-01-24'
# INTernal or EXTernal test (made by your organisation or others)
who = 'INT'
# Short title for the test
short_name = 'AMAZING_TEST'
# Comment for your test. Make this as long as you need to describe fully the purpose of this test
comment = ''
...
# Project
project = 'SmartCitizen'
# Firmware version
commit = ''
# Who made the test
author = 'Myself'
# Test type (indoor, outdoor, anything that helps you organise later on)
type_test = 'outdoor'
# If you are going to document the results somewhere, you can put a link below
report = ''
notes = ''
-----
#-----
```

ADD DEVICE INFORMATION

An example for each type of device is given below. A test can contain as many devices you need, each with an unique identifier.

- Data of a simple SC KIT from the API, downloading all the available data, with a frequency of 1Min:

```
device_1 = device_wrapper({'device_id': '8739',
    'type': 'KIT',
    'version': '2.1',
    'pm_sensor': 'PMS5003',
    'location': 'Europe/London',
    'frequency': '1Min',
    'source': 'api'})
```

- Data of a simple SC KIT from the API, downloading the data from a certain date, up to the last available data:

```
device_2 = device_wrapper({'device_id': '8739',
    'type': 'KIT',
    'version': '2.1',
    'pm_sensor': 'PMS5003',
    'location': 'Europe/London',
    'frequency': '1Min',
    'source': 'api',
    'min_date': '2019-07-12',
    'max_date': None})
```

- Data of a SC STATION from the local csv data, located in London. The csv file is in `data/raw/5527.csv`. Also, the device is archived in the device history calibration described in this section:

```
device_3 = device_wrapper({'name': '5527',
    'type': 'STATION',
    'version': '2.1',
    'pm_sensor': 'PMS5003',
    'location': 'Europe/London',
    'frequency': '1Min',
    'device_history': '5527',
    'source': 'csv_new',
    'fileNameRaw': device_name + '.csv',
    'fileNameInfo': ''})
```

- Data of a SC STATION from the API, located in London. The device is archived in the device history calibration:

```
device_4 = device_wrapper({'device_id': 5262,
    'type': 'STATION',
    'version': '2.1',
    'device_history': '5262',
    'pm_sensor': 'PMS5003',
    'location': 'Europe/London',
    'frequency': '1Min',
    'source': 'api'})
```

- Data of another device (reference equipment in this case), with local csv file, with frequency of 15Min. Extra information has to be given so that we can process the units of the different channels, the time index format and the location. The framework will convert the names from the `source_channel_names` to `target_channel_names`, and convert the `units`, in case they are available.

```
device_5 = device_wrapper({'name': 'PARROT_3',
    'type': 'OTHER',
    'location': 'Europe/London',
    'fileNameRaw': 'grow_parrot_u6dODj2Dnm1570629407808.csv',
    'fileNameInfo': '',
    'source': 'csv',
    'equipment': 'PARROT_SENSOR',
    'index': {'name': 'Time', 'format': '%Y-%m-%d %H:%M:%S', 'frequency': '15Min'},
    'channels': {'source_channel_names': ('air_temperature_celsius', 'battery_percent', 'calibrated_soil_moisture_percent', 'fertilizer_level', 'light',
    'soil_moisture_percent', 'water_tank_level_percent'),
        'units': ('degC', '%', '%', '-', 'lux', '%', '%'),
        'target_channel_names': ('TEMP', 'BATT', 'Cal Soil Moisture', 'Fertilizer', 'Soil Moisture', 'Water Level')
    },
    'location': 'Europe/Madrid'})
```

PROCESS EVERYTHING

Once we have all the information, we can then process the files:

```
list_devices = [device_1, device_2, device_3, device_4, device_5]
newtest.create(details, list_devices)
```

12.10.2 Devices history and calibration

Each device can have different sensors throughout its life and, for each sensor, we can have different calibration parameters. These two sets of information are stored in `data/interim`.

Devices History

Stored in `data/interim/sensorData.yaml`, it's a file to be manually filled with all the devices ID's you want to manage. It contains basic references of the internal sensors that can be later on used in the different calibration files. An example of a Smart Citizen Station is shown below. Since the device can have different sensors at different times, the dates are also important. The ID (below 4773), should be the same as in the test defined above in the field `device_history`:

```
'4773':
    hardware_id: '8ce207d2-504e4b4b-372e314a-ff03180c'
    sck_id: 'SCK2118080014'
    id: 'SCS21001'
    type: 'station'
    mac: '07D7'
    internal_ref: 'DEMO'
    date_from: '2018-09-17'
    date_to: '2018-09-21'
    gas_pro_board:
        CO: 162581720
        NO2: 202160405
        O3: 204160159
        slots: !!python/tuple [CO, NO2, O3]
    pm_board:
        PMS5003_1: 2017123000701
        PMS5003_2: 2017123000703
```

Devices calibration

Stored in `data/interim/CalibrationData/`, each json file contains a descriptor file for each sensor calibration data. An example for Alphasense's Electrochemical sensors is shown below:

```
{"Serial No": "162031254", "Target 2": "na", "Target 1": "CO", , "Sensitivity 1": "568.3", "Sensitivity 2": "0", "Zero Current": "-34", "Aux Zero Current": "-20.8"}  
{"Serial No": "162031257", "Target 2": "na", "Target 1": "CO", , "Sensitivity 1": "493.1", "Sensitivity 2": "0", "Zero Current": "-69.4", "Aux Zero Current": "-18.6"}
```

Calibration data

Currently, we store unique calibrations only for Alphasense sensors. This calibration is provided by the manufacturer directly and stored in this file.

12.10.3 Load the data

We have two main methods to load data into the framework: a test (coming from sd card data or API), and the API, directly. Run the following piece of code in your notebook :

Info

Check how to load test data in this example and how to download data from the API in this one

Loading data from a test

If you have local tests created as defined here, the tests will be accessible when you do:

```
[print (test) for test in data.get_tests(data.dataDirectory).keys()]
```

You can load as many as you want, and select the `frequency` and what to do with `Nan` values.

Decide this now

It is better to decide this now and not when we create the test. Better to have the information available always and process it each time on load than vice-versa.

The set of options below will try to:

```
options = {'clean_na': True, 'clean_na_method': 'drop', 'frequency': '3Min', 'load_cached_API': True, 'store_cached_API': True}
```

- Clean NaNs and how (`drop` or `fill`)
- Frequency for the data (`frequency`)
- Save API data (`store_cached_API`) : will try to cache this data in the disk for later use, and faster load next time (see next option)
- Load cached API data (`load_cached_API`) : if you have loaded this test previously, and if you have previously marked the above option, it will load the data directly from your disk, to avoid loading it again from the API

A normal output is shown below:

```
Test Load
Loading test 2018-01_INT_BOLOGNA_RELEASE
Comment: Pre Bologna release validation with 6 sensors (4 kits and 2 alphasenses)
Device ID SCK73FD
No metadata found - skipping
Kit SCK73FD located Europe/Madrid
Kit SCK73FD has been loaded
...
```

Loading data from the API

The devices are the devices IDs from: <https://smartcitizen.me/kits/1234>. An example is shown below:

```
device = api_device('1234', verbose = True)
device.get_device_data(start_date = None, end_date = None, frequency = '1Min', clean_na = False, clean_na_method = None);
```

You should see something like this:

```
Loading device 1234 from API
Kit ID 19
...
```

Note that if the device ID is in the devices history, it will also load the calibration data for you.

Info

A more advanced guide can be found [here](#)

12.10.4 Data structure

Data is organised internally using tests. Each test can contain multiple devices. Each device, can contain data from one single source, either an API, or CSV files.

```
# Tests
data.tests.keys()
```

Devices within a test.

```
data.tests[testname].devices.keys()
```

Data inside the device:

```
data.tests[testname].devices[device_name].readings
```

We can check the columns in it by:

```
data.tests[testname].devices[device_name].readings.columns
```

12.10.5 Export data

Once we have concluded our analysis, we can export the process data.

```
# Test name where the device is
testname = 'mytest'
# List of devices to export
devices = list(data.tests[mytest].devices.keys())
# Example exporting only the first. We can iterate over devices with a for loop and export them all in separate CSV files
devicename = devices[0]
# Export it
data.export_data(testname, devicename, export_path = '/path/to/folder', all_channels = True, forced_overwrite = True)s
```

Some options are available:

- Copy to test folder: it will export the csv to the `/processed` folder in the test
- Include All / Raw / Processed: Options to include all the channels in your test, or only the ones tagged as `raw` or `processed` from the descriptor file in: `data/interim/sensorNamesExport.json`
- Rename channels: if you want to rename channels with the descriptor file mentioned above

12.11 Third party sensors

This page reflects examples on how to use and implement compatible third party sensors.

What are *third party sensors*?

By third party sensors, we mean sensors that have been developed by others, with no affiliation to the Smart Citizen Team.

This page is a digest and updated version of the Making Sense D2.3 Smart Citizen Toolkit report and Making Sense D.24 Smart Citizen Toolkit report updates. Both these reports reflect information for the **SCK 1.5**, which is not a commercially available version of the kit. This guide is an update version for the SCK 2.1.

12.11.1 Use of already supported sensors

The auxiliary port is designed to expand the sensor board by adding new sensors via the common I2C standard. However other protocols are supported, such as SPI or UART. The pins have the following default configuration:

PIN	PORT	Function
1	SCL	I2C (by software: 1-WIRE or other)
2	SDA	I2C (by software: 1-WIRE or other)
3	VCC	Voltage
4	GND	Ground

By connecting any of the supported sensors to the SCK, it will automatically be detected and data will be logged into the SD-card. You can check the output of the `sensor` command in the Serial output:

```
> sensor
Enabled
-----
Temperature (60 sec)
Humidity (60 sec)
Ext Temperature (60 sec)
Ext Humidity (60 sec)
Battery (60 sec)
Light (60 sec)
Noise dBA (60 sec)
Barometric pressure (60 sec)
PM 1.0 (60 sec)
PM 2.5 (60 sec)
PM 10.0 (60 sec)
```

Publishing data using custom devices

The Smart Citizen Platform supports data from any sensor that has a **numerical digital output**. The Smart Citizen API supports other devices to publish data to the platform by previously agreeing with the Smart Citizen terms and conditions.

For each device type, a new device blueprint needs to be created. **A device blueprint defines the sensors and the metrics that your devices will have.** This will include the hardware details of your sensors and the kind of data that will be published to the platform. Custom calibration formulas to be applied to the data when processed in the platform can be also added.

How to do it?

Once a device blueprint is added to the platform, any user can create as many devices as needed and publish data to them following the standard Smart Citizen API. It is important to note that Device Blueprint currently cannot be created by users and should be requested by contacting support@smartcitizen.me.

The minimal Device Blueprint includes all the necessary data that a user might provide in order to create a Kit. It is composed of Components and those can reuse existing Sensors and Measurements definitions. Sensors define the hardware or software component that records the data. Measurements are descriptions of what sensors are recording. Blueprints can be shared across many devices or can be tailored per device in order to provide dedicated calibration formulas per individual sensor. This is achieved with the *Components* binding.

The following example shows a basic Device Blueprint in JSON. This is the minimum of information that a blueprint needs:

```
{
  "name": "The Frog",
  "description": "Custom Arduino Humidity Sensor",
  "slug": "ms:0,5",
  "components": [
    {
      "map": "hum",
      "equation": "(125.0 / 65536.0 * x) + 7",
      "sensor": {
        "name": "HPP828E031",
        "description": "Humidity",
        "unit": "%",
        "measurement": {
          "name": "relative humidity",
          "description": "Relative humidity is a measure..."
        }
      }
    }
  ]
}
```

The following examples expand the previous Device Blueprint with the complete data model:

```
{
  "id": 10,
  "uuid": "056e452d-41c4-436d-a640-2157a278037d",
  "slug": "ms:0,5",
  "name": "The Frog",
  "description": "Custom Arduino Humidity Sensor",
  "created_at": "2016-06-18T16:25:02Z",
  "updated_at": "2016-06-18T16:25:02Z",
  "components": [
    {
      "id": 35,
      "uuid": "22da9377-5151-4547-a71b-6fd8958e1330",
      "equation": "(125.0 / 65536.0 * x) + 7",
      "map": "hum",
      "sensor": {
        "id": 13,
        "uuid": "1c19ae8f-b995-460f-87a3-a9d0c140abfb",
        "parent_id": 19,
        "name": "HPP828E031",
        "description": "Humidity",
        "unit": "%",
        "created_at": "2015-02-02T18:24:30Z",
        "updated_at": "2015-07-05T19:54:54Z",
        "measurement": {
          "id": 2,
          "uuid": "9cbbd396-5bd3-44be-adc0-7ffba778072d",
          "name": "relative humidity",
          "description": "Relative humidity is a measure of the amount of moisture in the air relative to the total amount of moisture the air can hold. For instance, if the relative humidity was 50%, then the air is only half saturated with moisture."
        }
      }
    }
  ]
}
```

Too much information?

Drop an email to support@smartcitizen.me and we will try to help!

Using SEEED Studio Grove bricks

You can use off-the-shelf sensors from the extensive Grove open hardware sensor library, removing the need to build our own sensor add-ons from scratch. Foto seeed sensors Seeed Grove Bricks

12.11.2 Implementing other sensors

This is a WIP

This section is under heavy development. Thanks for your patience!

Implementation of other sensors goes through the modification of the Firmware. This is an advanced user feature, and previous programming experience in C++ is necessary.

The workflow we normally follow for this goes like:

1. Find out if there is an already existing library for the desired sensor. Good places to look at are Adafruit's repository, Sparkfun's repository or a global Github search
2. Implement this library in the firmware. The library needs certain functions to be valid. (More info soon!)
3. If the device needs to log the data on the platform, you can email us at support@smartcitizen.me with a request for a new device blueprint. However, it is easier to simply log the data in SD-card in this case, if the online recording is not fully mandatory.

Contribute it back to the community

Make a pull request with your contribution back to the firmware so that other can use it!

12.12 Update the firmware

When new features are developed or bugs are fixed we will release new versions of the SCK firmware.

Info

If you already configured your kit on the smartcitizen platform **you will need the token that the platform gave you during the onboarding process**, to recover it from your kit:

1. **Click your kit button** until the kit is in setup mode, the led should be red.
2. **Connect to the kit** with your mobile device as you did during the onboarding process.
3. **Write down the token** of your kit.

After updating the firmware follow this same steps to input the token and wifi credentials, after this your kit will be publishing on the same registered device than before.

A note about versions

-  The guide below applies to both, SCK 2.0 and SCK2.1.

Updating your kit is very simple

- **Connect your kit** with a micro USB cable to your computer.
- **Double click the reset button** of your SCK, the SCK led should turn green and a new drive called SCK-20 should appear on your computer file browser.
- Inside the SCK-20 drive you should see some files, **double click the INDEX.HTM** file and our github releases page will open in your browser. **Download the new firmware** called SAM_firmware_XXX.uf2 and save it to your computer.

Tip

You can backup your current firmware version just saving the file called CURRENT.UF2.

- Simply **drag the firmware file you downloaded over the SCK-20 drive**, your kit led will blink in green and after some seconds it will reset and start with the new version.
- If your **Wi-Fi module needs a firmware update** when you connect to your kit to setup the network you will see a screen that will ask for the new file. You can find it in our github releases page, look for the file called ESP_firmware_XXX.bin. If you don't see it, check in a previous release (some releases don't include Wi-Fi firmware).
- After the update you just done, you can configure your kit as a new device following the onboarding process or use your previous token as explained before.

Obtain your firmware version remotely (advanced)

If you are an advance user managing a big deployment of devices you can obtain remotely the version of all the Kits you have registered by looking at the `hardware_info` property of each of your devices using the platform API `/v0/devices/`. When your Kit is in Wi-Fi mode, it publishes the information daily.

```
"hardware_info": {  
    "id": "DFD098A758515157382E3120FF101D12",  
    "mac": "B6:E6:20:66:47:6D",  
    "time": "2020-04-14T03:00:24Z",  
    "esp_bd": "",  
    "hw_ver": "2.1",  
    "sam_bd": "2019-11-27T12:49:13Z",  
    "esp_ver": "",  
    "sam_ver": "0.9.6-4e90c77"  
}
```

More info in the platform API documentation.

12.13 Updating the Wi-Fi

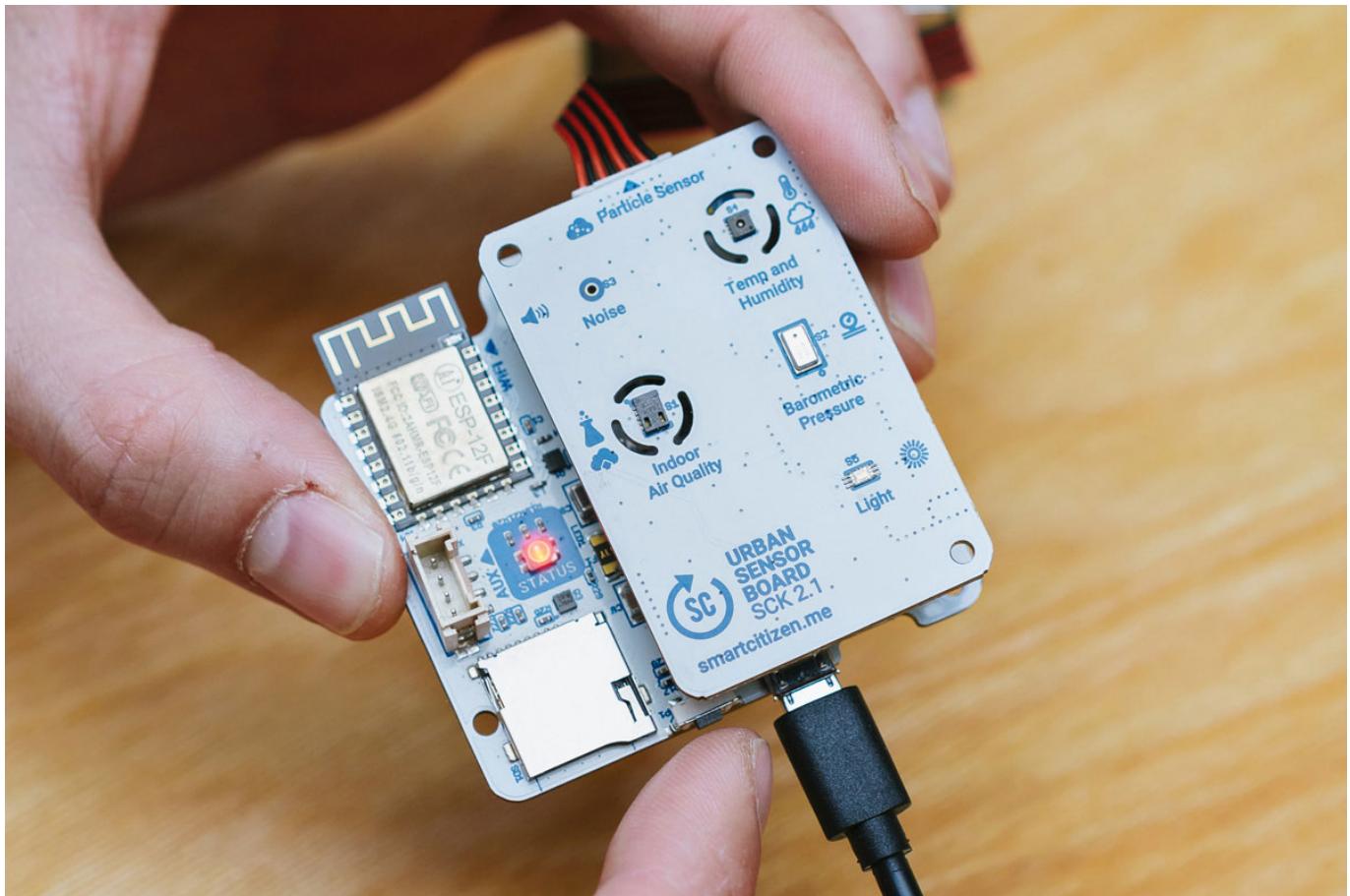
Info

This guide will help you update your sensor Wi-Fi settings as well as other parameters you can set on the Setup App

Warning

- ✓ The kit supports Wi-Fi WEP, WPA/WPA2 and open networks that are common networks in domestic environments and small businesses.
- ✗ But, it **does not** support WPA/WPA2 Enterprise networks such as EDUROAM or networks with captive portals such as those found in Airports and Hotels

1. Click your kit button until the kit is in setup mode, the led should be red.

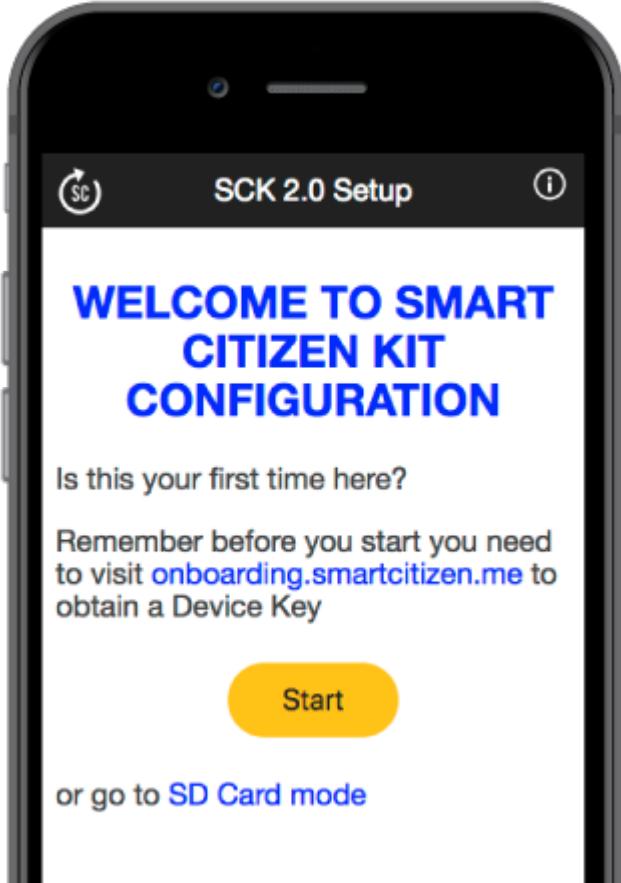


1. Connect to the kit with your mobile device as you did during the installation process. You will need to search for a **Wi-Fi network** called `SmartCitizen[...]`. If you have multiple kits `[...]` is the unique identifier of your kit.
2. Once connected you should see the **Setup App** in your phone. If it doesn't show up automatically you can open <http://192.168.1.1> on your phone browser.

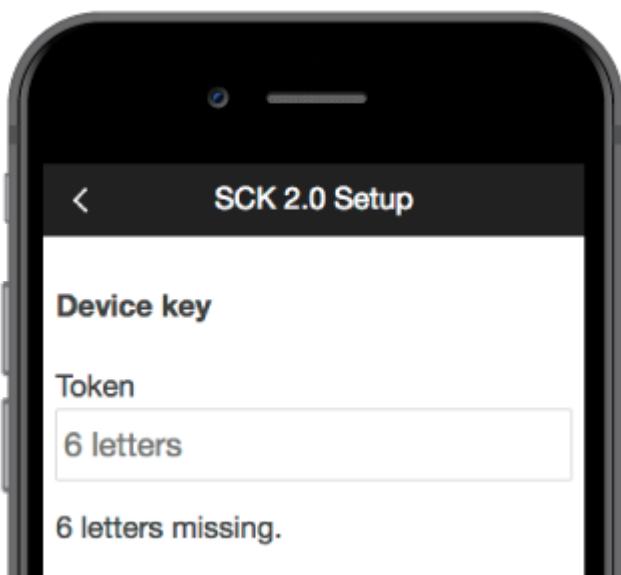
Info

The process also works on laptops, desktops and tablets, connected than can connect to the SmartCitizen[...] network.

1. On the app **choose start** to move to the next page.



1. You will see your **device key** or token. Do not change it and **choose next**.



1. Set your new Wi-Fi settings on the next screen as you did during the installation process.



1. The Led will go blue and **your kit should start to publish again using the new Wi-Fi**. You can confirm it by visiting your device on smartcitizen.me/kits. You can repeat the process as many times as you want by going back to step 1.

Info

If your Kit is not publishing check the LED status chart to know more about it.

12.14 Upload data to zenodo

Uploading results to Zenodo is also possible using the data analysis framework.



Once you have your data organised in a test, you can upload it directly to Zenodo and share it with others. An example of this is provided in the example notebook.

For this to work, we need to have a token. Put this in a `secrets.py` file in the `src` directory:

```
> cd src
> ls -la
.
..
__init__.py
__pycache__
config.yaml
data
models
saf.py
secrets.py
src.egg-info
tools
visualization
```

In this file, include your token as below:

```
zenodo_token='yourtokenhere'
```

Next, you can define a `upload.yaml` file to describe the upload (see one example here: [https://github.com/ISCAPE-D3.13/Sensor_Platforms/blob/main/upload.yaml](#))

```

example_upload_1.json:
  title: 'Example Data 1'
  description: 'This field accepts <strong>HTML</strong>'
  upload_type: 'dataset'
  keywords: ['Low-Cost Sensors', 'Air Quality', 'Citizen Science']
  creators: [{name: 'Author 1', affiliation: 'Affiliation 1', orcid: '0000-0000-0000-0001'}, {name: 'Author 2', affiliation: 'Affiliation 2', orcid: '0000-0000-0000-0002'}]
  tests: ['TEST_1', 'TEST_2']
  access_right: 'open'
  options:
    include_processed_data: false
    include_footer_doi: true
  communities: [{ identifier: "community_in_zenodo"}]
  grants: [{"id": "GRANT_ID"}]
  report: ['report.pdf']

example_upload_2.json:
  title: 'Example Data 2'
  description: 'This field accepts <strong>HTML</strong>'
  upload_type: 'dataset'
  keywords: ['Low-Cost Sensors', 'Air Quality', 'Citizen Science']
  creators: [{name: 'Author 1', affiliation: 'Affiliation 1', orcid: '0000-0000-0000-0001'}, {name: 'Author 2', affiliation: 'Affiliation 2', orcid: '0000-0000-0000-0002'}]
  tests: ['TEST_3', 'TEST_4']
  access_right: 'open'
  options:
    include_processed_data: false
    include_footer_doi: false
  communities: [{ identifier: "community_in_zenodo"}]
  grants: [{"id": "GRANT_ID"}]
  report: ['report_2.pdf']

```

Info

All the keys below are linked to the zenodo documentation

Different uploads can be defined by each main key: `example_upload_1.json` and `example_upload_2.json`. Each of them contains the following information (all of them can be later modified or added in the web interface in zenodo.org)

Metadata for Zenodo

- `title` : Name for the dataset
- `description` : dataset description (mandatory). This field accepts HTML
- `upload_type` : 'dataset' (mandatory - always, for now)
- `keywords` : list of keywords, such as `['Low-Cost Sensors', 'Air Quality', 'Citizen Science']`
- `creators` : dictionary containing the authors. Can contain the name, the affiliation and an orcid
- `access_right` : 'open' (other options in the zenodo documentation)
- `communities` : id for the zenodo data community
- `grants` : grant id

Upload

- `tests` : `['TEST_1', 'TEST_2']`
- `options` :
 - `include_processed_data` : true or false. Whether or not to include the processed data from the `processed` folder in the test directory
 - `include_footer_doi` : true. If there is a report attached, add a nice footer to it with the DOI
- `report` : list of reports to attach. Must be also in the `data/uploads` folder

To upload the datasets, we can use the zenodo sandbox and a `dry_run`, to check everything is running well. Then, make these defaults to `False` to actually upload it:

```
# You can use the sandbox.zenodo.org for tests, as well as a dry_run. When you are happy with your upload, set these variables to False  
# Then go to uploads in the zenodo section and publish whenever you are ready  
data.upload_to_zenodo('example_zenodo_upload', sandbox = False, dry_run = True)
```

Warning

Note that a .json file will be created in the data/uploads folder containing the metadata necessary for the upload (as liked by zenodo API). You can securely delete this file once you are done. Note that, in case `include_footer_doi=true`, the actual pdf to upload will be `report_doi.pdf`

Finally, **to actually deploy the dataset**, you need to visit the deposit section and aprove it manually.

Info

Get the iSCAPE datasets from Zenodo here:

- <https://zenodo.org/record/3570700>
- <https://zenodo.org/record/3570680>
- <https://zenodo.org/record/3570688>

12.15 Uploading SD Card Data

Here some instructions on how to upload CSV files to Smartcitizen platform. First be sure to be logged and go to your profile.

Weird files?

Check what they mean here

Step by step

On your kits' list, click on the wheel and then on "Upload CSV".

The screenshot shows the Sensor Kits platform interface. At the top, there are navigation links: SC (selected), MAP, COMMUNITY, and a search bar. On the right, there are buttons for 'GET YOUR KIT' and a user profile icon. Below the header, the user profile 'dsmrs' is shown with status 'No data' and 'No website'. The main area displays a list of kits under 'FILTER KITS BY' (ALL, ONLINE, OFFLINE). One kit, 'Elite Stack Mushroom', is highlighted with a yellow box around its 'EDIT' and 'UPLOAD CSV' buttons. Another kit, 'Outstanding Stack Stag', is also visible. A yellow arrow points from the 'EDIT' button to the 'UPLOAD CSV' button.

Once on the upload page, you can add some files by clicking on the "Load CSV files" button.

The screenshot shows the 'Upload CSV Files' page. At the top, there are navigation links: SC (selected), MAP, COMMUNITY, and a search bar. On the right, there is a 'BACK TO PROFILE' button. The main area has a heading 'Upload CSV Files' and a 'LOAD CSV FILES' button, which is highlighted with a yellow arrow. Below it, there is a section titled 'Upload your files' with the instruction 'Select the files you want and upload them into your kit!'. There are 'ACTIONS' and 'APPLY' dropdown menus. A message box says 'There are no files here. Let's upload something!'.

Select some files as much as you like, to be them ready to upload. Then on the drop-down menu select the "Upload" option

LOAD CSV FILES

Upload your files

Select the files you want and upload them into your kit!

NONE **APPLY**

- SELECT ALL**
- DESELECT ALL**
- UPLOAD** **▼** **APPLY**
- REMOVE**

<input checked="" type="checkbox"/> 18-05-1.csv	NEW DATA	DELETE
<input checked="" type="checkbox"/> 18-05-2.csv	NEW DATA	DELETE
<input checked="" type="checkbox"/> 18-05-3.csv	NEW DATA	DELETE
<input checked="" type="checkbox"/> 18-05-4.csv	NEW DATA	DELETE
<input checked="" type="checkbox"/> 18-05-5.csv	NEW DATA	DELETE

Click on the "Apply" button to upload them

LOAD CSV FILES

Upload your files

Select the files you want and upload them into your kit!

Actions

UPLOAD **▼** **APPLY**

<input checked="" type="checkbox"/> 18-05-1.csv	NEW DATA	DELETE
<input checked="" type="checkbox"/> 18-05-2.csv	NEW DATA	DELETE
<input checked="" type="checkbox"/> 18-05-3.csv	NEW DATA	DELETE
<input checked="" type="checkbox"/> 18-05-4.csv	NEW DATA	DELETE
<input checked="" type="checkbox"/> 18-05-5.csv	NEW DATA	DELETE

Congrats! You just uploaded your files CSV files on the Smartcitizen platform.

MAP COMMUNITY Search GET YOUR KIT

LOAD CSV FILES

Upload your files

Select the files you want and upload them into your kit!

ACTIONS ▾ APPLY

<input checked="" type="checkbox"/> 18-05-1.csv ✓	NEW DATA	
<input checked="" type="checkbox"/> 18-05-2.csv ✓	NEW DATA	
<input checked="" type="checkbox"/> 18-05-3.csv ✓	NEW DATA	
<input checked="" type="checkbox"/> 18-05-4.csv ✓	NEW DATA	
<input checked="" type="checkbox"/> 18-05-5.csv ✓	NEW DATA	

12.16 Using the Shell

Warning

This guide is a work in progress!

The SCK (from V2.0 onwards) has an integrated command shell over USB to manage all the kits functionalities for advanced users. In this guide, we will cover how to access to this functionality in different platforms, and some examples.

12.16.1 What is it?

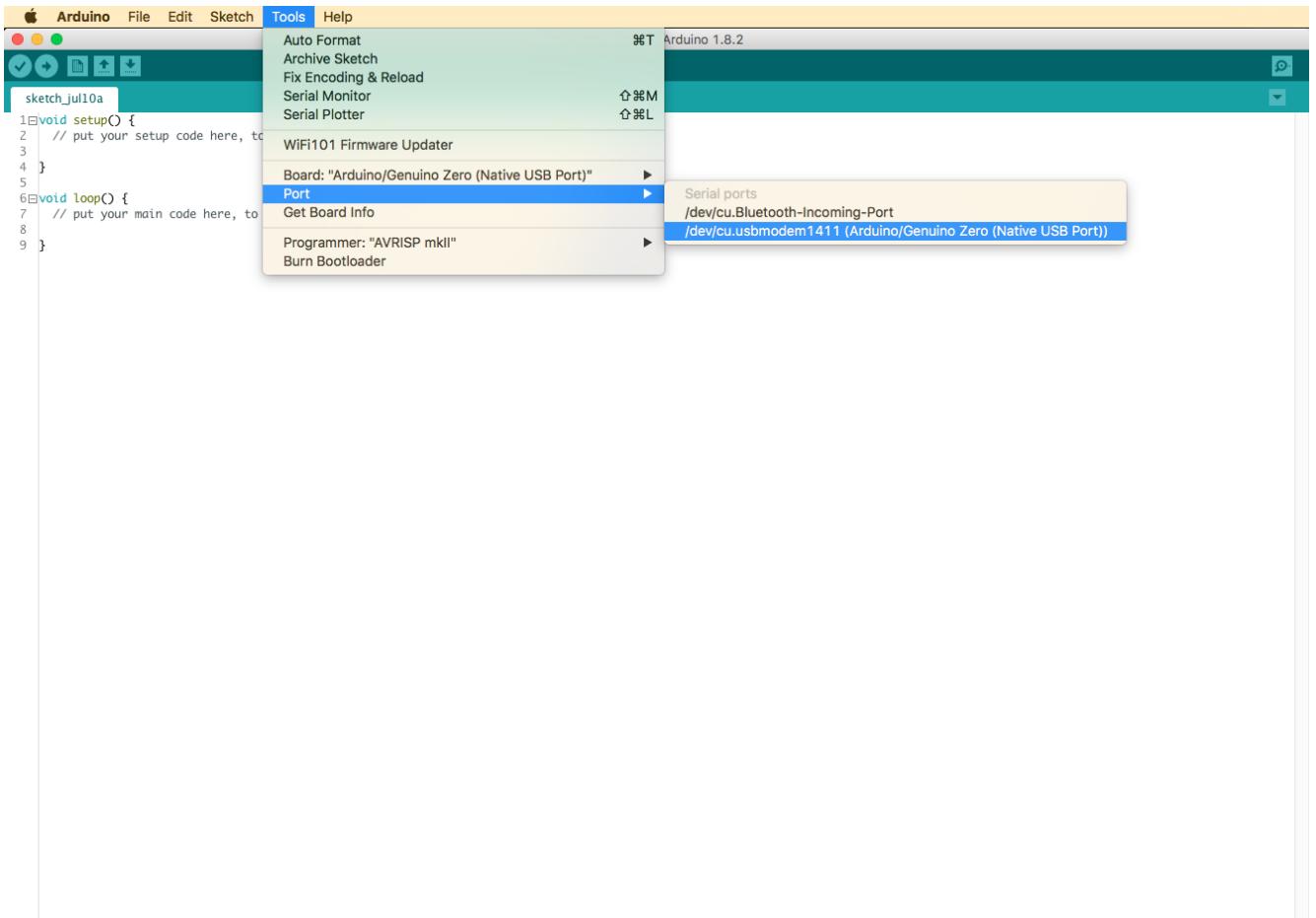
We could define the `shell` as a text-based interface to access almost any SCK functionality. In terms of hardware, it relies on the serial communication between the SCK and your computer, so any **decent** USB cable connected between them will do.

12.16.2 How to access it?

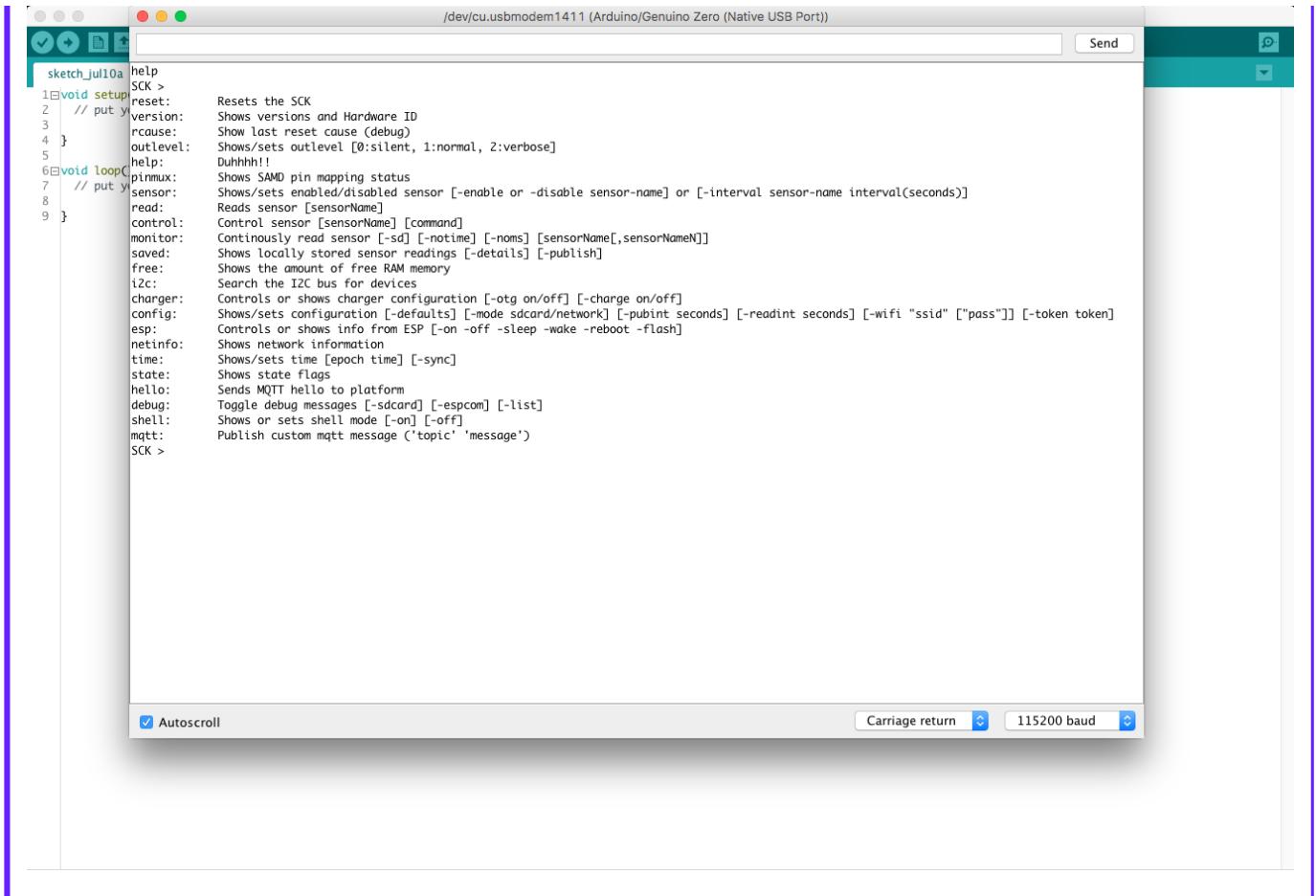
Software-wise, different platforms will have different interfaces. The easiest and most reliable for all of them would be through the Arduino IDE.

Using the Arduino IDE

- Launch the *Arduino IDE* and select the port under *Tools > Port >*:



- Launch the *Serial Monitor* under *Tools > Serial Monitor*. Make sure that the dropdowns in the bottom are set as in the image below (Carriage return and 115200 baud)
- Type in `help` to get started.



More advanced users would probably rather use a more *rugged* interface. In this case, you could use `screen` in your terminal of choice:

```
> ls /dev/cu | grep usb
cu.usbmodem1411
tty.usbmodem1411
> screen /dev/cu.usbmodem1411
SCK >
...
```

If you already installed platformio to edit the firmware you can use it here, too

```
> pio device monitor
SCK >
...
```

Be patient!

The port will take a little time to appear in your list of devices. Normally the LED of your SCK will be **static white** during that period.

12.16.3 Some examples!

The `help` command outputs a quite intuitive explanation of all the commands:

```
SCK > help
reset: Resets the SCK
version: Shows versions and Hardware ID
rcause: Show last reset cause (debug)
outlevel: Shows/sets outlevel [0:silent, 1:normal, 2:verbose]
help: Duhhh!!
pinmux: Shows SAMD pin mapping status
sensor: Shows/sets enabled/disabled sensor [-enable or -disable sensor-name] or [-interval sensor-name interval(seconds)]
read: Reads sensor [sensorName]
control: Control sensor [sensorName] [command]
monitor: Continously read sensor [-sd] [-notime] [-noms] [sensorName[,sensorNameN]]
saved: Shows locally stored sensor readings [-details] [-publish]
free: Shows the amount of free RAM memory
i2c: Search the I2C bus for devices
charger: Controls or shows charger configuration [-otg on/off] [-charge on/off]
config: Shows/sets configuration [-defaults] [-mode sdcard/network] [-pubint seconds] [-readint seconds] [-wifi "ssid" ["pass"]] [-token token]
esp: Controls or shows info from ESP [-on -off -sleep -wake -reboot -flash]
netinfo: Shows network information
time: Shows/sets time [epoch time] [-sync]
state: Shows state flags
hello: Sends MQTT hello to platform
debug: Toggle debug messages [-sdcard] [-espcom] [-list]
shell: Shows or sets shell mode [-on] [-off]
mqtt: Publish custom mqtt message ('topic' 'message')
```

Pro tip

The SCK outputs a lot of information via serial. This can be sometimes confusing while typing commands. You can silent it a bit with this command:

```
SCK > shell -on
Shell mode: on
```

This will turn your LED static yellow, and no output except responses to your commands will be given.

Remember to turn it off after you are done experimenting!

```
SCK > shell -off
Shell mode: off
```

Set the recording configuration

If you want to change your recording mode to, for instance, `sdcard` mode, you could do so by typing:

```
SCK > config -mode sdcard
```

To set it up in `network` mode:

```
SCK > config -mode network -wifi "SSID" "PASSWORD" -token 123456
```

Warning

Note that the token is not between quotes since it's always 6 digits

To modify your wifi:

```
SCK > config -wifi "NEWSSID" "NEWPASSWORD"
```

You can check your current configuration by typing `config`:

```
SCK > config
Mode: sdcard
Publish interval: 60
Reading interval: 60
Wifi credentials: not configured
Token: not configured
Mac address: 11:22:33:44:55:66
```

Get version data

Check your **hardware and firmware version** data with the command `version`:

```
SCK > version
Hardware Version: 2.1
SAM Hardware ID: 5934C4B550515157382E3120FF151210
SAM version: 0.9.1-30e1776
SAM build date: 2019-05-07T02:45:29Z
ESP MAC address: 86:0D:8E:A7:7F:CC
ESP version: not synced
ESP build date: not synced
```

List/modify the active sensors

By typing in `sensor`, a **list of enabled and supported sensors** is displayed:

```
SCK > sensor
Disabled
-----
PM board Dallas Temperature
[...]
Enabled
-----
Temperature (60 sec)
Humidity (60 sec)
Battery (60 sec)
Light (60 sec)
Noise dBA (60 sec)
Barometric pressure (60 sec)
VOC Gas CCS811 (60 sec)
eCO2 Gas CCS811 (60 sec)
PM 1.0 (60 sec)
PM 2.5 (60 sec)
PM 10.0 (60 sec)
```

To **disable** one sensor, you can type in part of the sensor name:

```
SCK > sensor -disable Noise
Disabling Noise dBA
Saved configuration on eeprom!!
```

To **enable** it, if it's present:

```
SCK > sensor -enable Noise
Enabling Noise dBA
Saved configuration on eeprom!!
```

Only if available!

If the sensor you are trying to connect is not recognised, the kit will complain:

```
SCK > sensor -enable atlas
Failed enabling Atlas Temperature
```

Read/Monitor some sensors

If one sensor is enabled, you can `read` it (once) or `monitor` it (as fast as the SCK can):

```
SCK > read Noise
Noise dBA: 53.85 dBA
```

To monitor **one sensor**:

```
SCK > monitor light
Time   Milliseconds   Light
2019-07-10T17:58:07Z   6      137
2019-07-10T17:58:07Z   98     137
2019-07-10T17:58:07Z   98     136
2019-07-10T17:58:07Z   98     137
2019-07-10T17:58:07Z   108    137
2019-07-10T17:58:07Z   98     137
2019-07-10T17:58:07Z   98     137
2019-07-10T17:58:07Z   108    137
2019-07-10T17:58:07Z   98     137
2019-07-10T17:58:08Z   108    137
2019-07-10T17:58:08Z   98     136
...
...
```

Or all of them, with no arguments:

Time	Milliseconds	Battery	Light	Temperature	Humidity	Noise dBA	Barometric pressure	VOC Gas CCS811	eCO2 Gas CCS811	PM 1.0	PM 2.5	PM 10.0
2019-07-11T09:13:04Z	5	37	137	28.75	57.72	1.5	101.29	1.00	408.00	1.5	1.5	1.5
2019-07-11T09:13:07Z	3195	37	138	28.78	57.65	1.5	101.30	1.00	408.00	1.5	1.5	1.5
2019-07-11T09:13:08Z	694	37	136	28.77	57.62	1.5	101.29	1.00	408.00	1.5	1.5	1.5

If you don't need to output the `milliseconds` column (the time since last reading) or the `timestamp`, you can do so by:

```
SCK > monitor -noms light
Time   Light
2019-07-10T17:58:58Z   136
2019-07-10T17:58:58Z   136
2019-07-10T17:58:58Z   137
2019-07-10T17:58:58Z   137
2019-07-10T17:58:59Z   136
...
...
```

```
SCK > monitor -notime light
Milliseconds   Light
6      137
98     137
98     137
99     137
108    137
...
...
```

Warning

If your kit has no time configured (the LED should be flashing), the output would look like:

```
SCK > monitor Noise
Time   Milliseconds   Noise dBA
0      1      52.83
0      187    50.36
0      187    52.05
0      187    51.95
0      187    48.28
0      187    48.72
0      187    50.81
...
...
```

Something cool to do with the `monitor`, is to log the sensor output into a file for later analysis. For instance, in your terminal you could do:

```
> echo "monitor pm light" > /dev/cu.usbmodem1411 && screen -L /dev/cu.usbmodem1411
```

Then, if we check the contents of the file (normally something like `screenlog.X`):

```
monitor light
Time    Milliseconds    Light
2019-07-11T09:10:05Z    6      141
2019-07-11T09:10:05Z    98     141
2019-07-11T09:10:05Z    99     141
2019-07-11T09:10:05Z    98     141
2019-07-11T09:10:05Z    108    141
2019-07-11T09:10:05Z    98     141
2019-07-11T09:10:05Z    98     141
2019-07-11T09:10:05Z    98     141
2019-07-11T09:10:06Z    98     141
2019-07-11T09:10:06Z    108    141
2019-07-11T09:10:06Z    98     141
2019-07-11T09:10:06Z    98     141
...
...
```

This can be useful in case you want to log data as fast as possible, with little delay between readings (~100ms).

12.16.4 Advanced (but cool) example!

Making most of the digital microphone

- The digital microphone in your SCK uses an FFT algorithm to calculate the final sound pressure level (SPL) in different scales (A, C, Z). The FFT spectrum is also available for user analysis. Let's have a look!
- First, enable it with:

```
SCK > sensor -enable fft
Enabling Noise FFT
```

- Then, monitor and log it in a file with:

```
echo "monitor fft" > /dev/cu.usbmodem1411 && screen -L /dev/cu.usbmodem1411
```

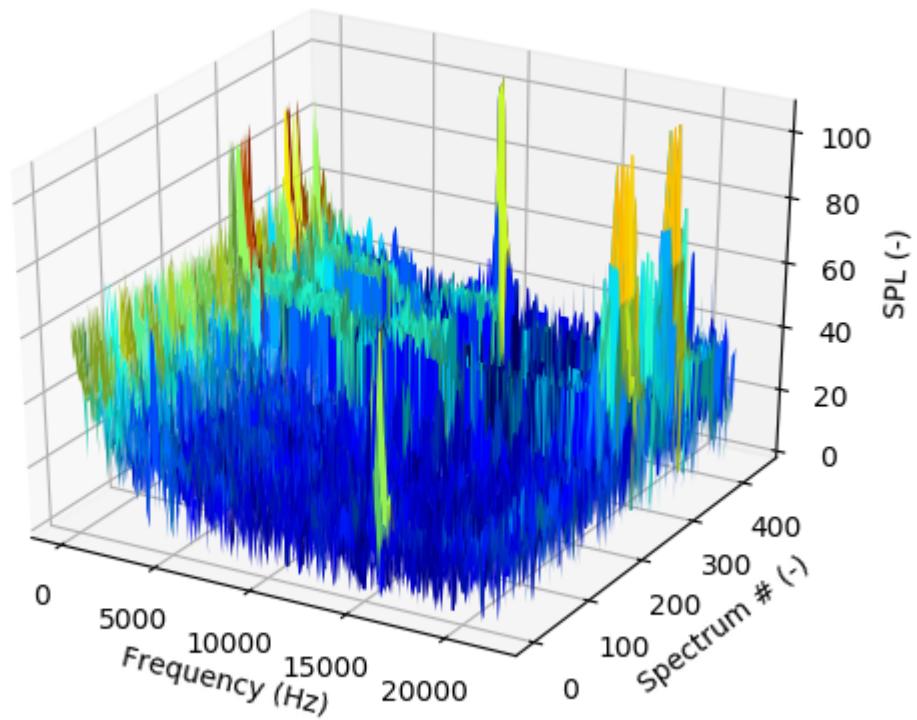
- In this file, we will have something like:

```
12
15
19
20
0
...
12
23
2019-07-11T09:38:01Z    5  null
...
```

- The values between the dates are the actual FFT spectrum. We will now clean the lines with the dates and then plot the data. For this, we will use a `python` code to make things easier. You can download the code [here](#).
- If we run this code in `python3` in the same folder where the screenlog from before is:

```
> python spectrum_example.py
```

We will have two outputs: a `csv` file with the spectrums in rows, and a `png` image that looks like this!

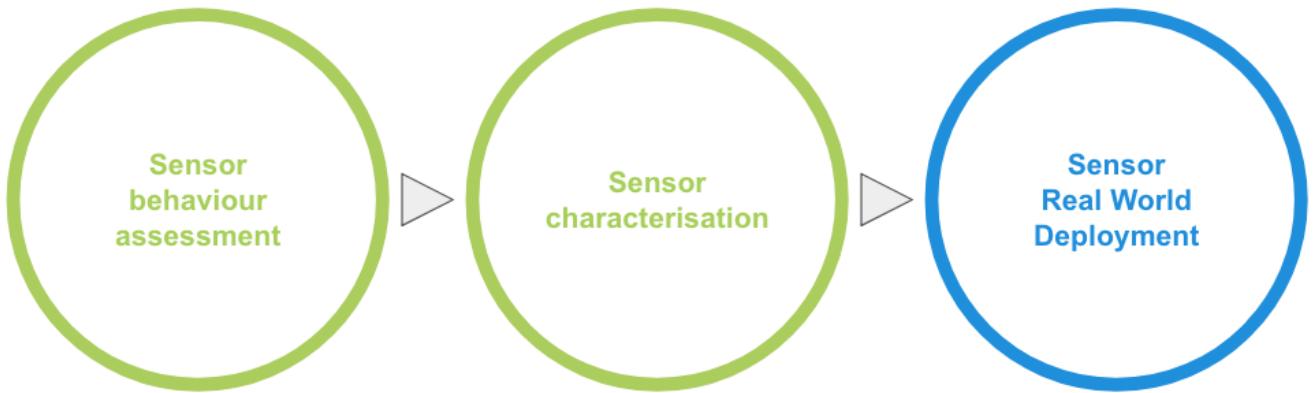


You can see that we were playing with a tone generator to make some high pitch noises at 10kHz and 20kHz.

13. Sensor Analysis Framework

13.1 Low Cost Sensors Calibration

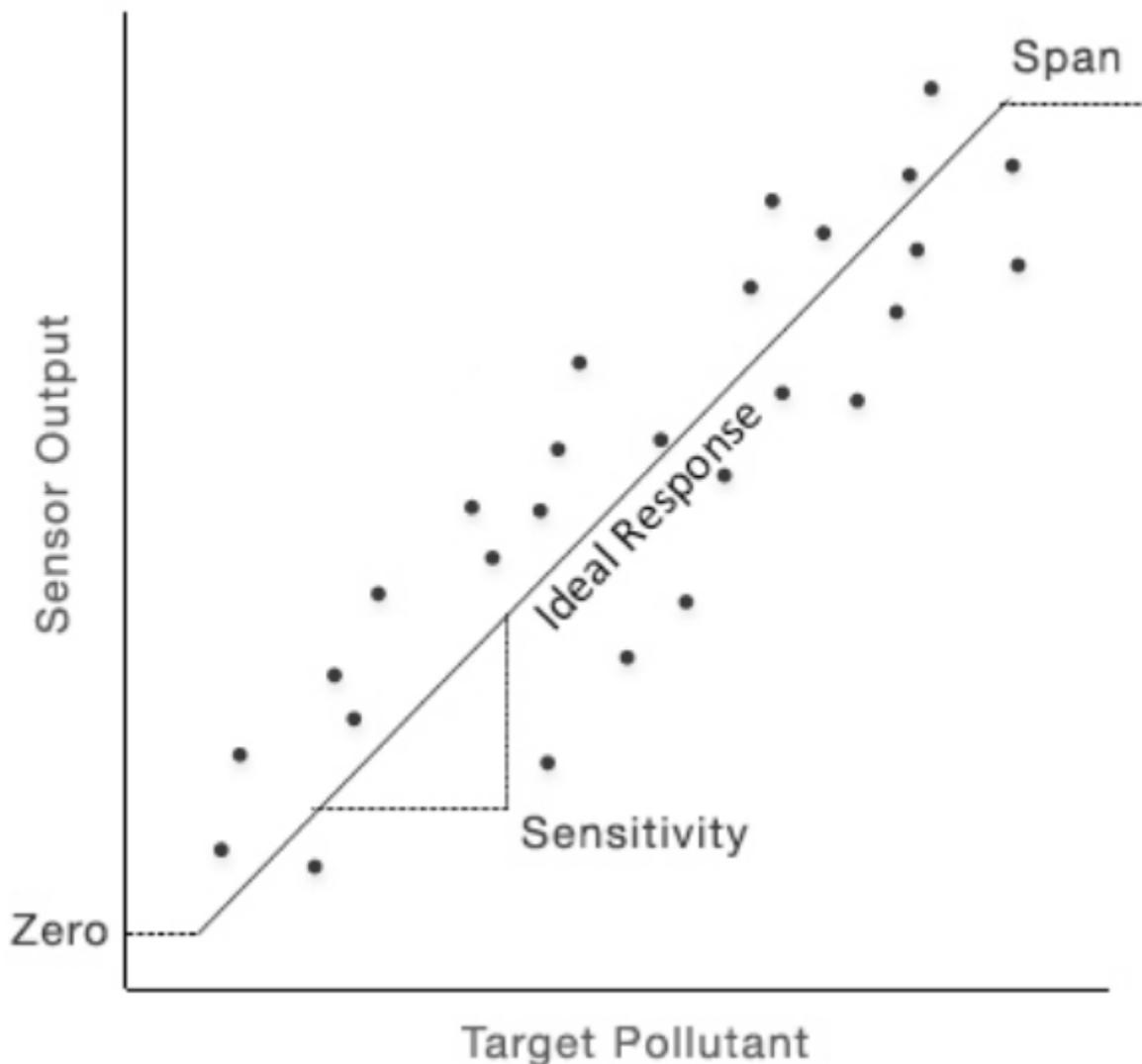
Low cost sensor calibration and assessment pose a great challenge for data quality objectives. We follow this **sensor calibration procedure**, which can be split into three stages:



1. **Behaviour assesment:** in laboratory conditions, serving as base testing for assessing general sensor behaviour.
2. **Characterisation:** also in laboratory conditions, assess generic sensor parameters as sensitivity, zero and span.
3. **Modelisation with real world deployment:** including other variables such as environmental factors and sensor cross-sensitivity.

Each of these stages apply differently depending on the type of sensor. For instance the electrochemical sensors present in the *Station* are already characterised by the manufacturer, while the old SGX MICS4514 Metal Oxyde Sensors in the *Urban Board* of the *Smart Citizen Kit* are not. The different characteristics of these sensors make different calibration approaches to be carried out.

Base calibration parameters need to be determined in controlled conditions. In this stage, the aim would be to find parameters such as:



- **Sensor sensitivity:** the sensor response per each ppm of target pollutant in *nominal* conditions
- **Zero:** the sensor reading in zero air (pure air at 25degC).
- **Sensor response (t_{90})**
- **Sensor range:** maximum and minimum readings for the sensor

Finally, after this initial calibration assessment, it is critical to gather as much data as possible from **long term sensor deployments**. These deployments should aim to cover the widest range of sensor exposure conditions, in order to generate robust models. While dealing with low cost sensors this stage is very important, as it is detailed in the sections below.

These sensor deployments serve for two main purposes: to generate **quantitative classification methods** that can classify the air quality in predefined ranges (i.e. 'poor', 'fair', 'good'); and to generate **predictive qualitative models** for more accurate

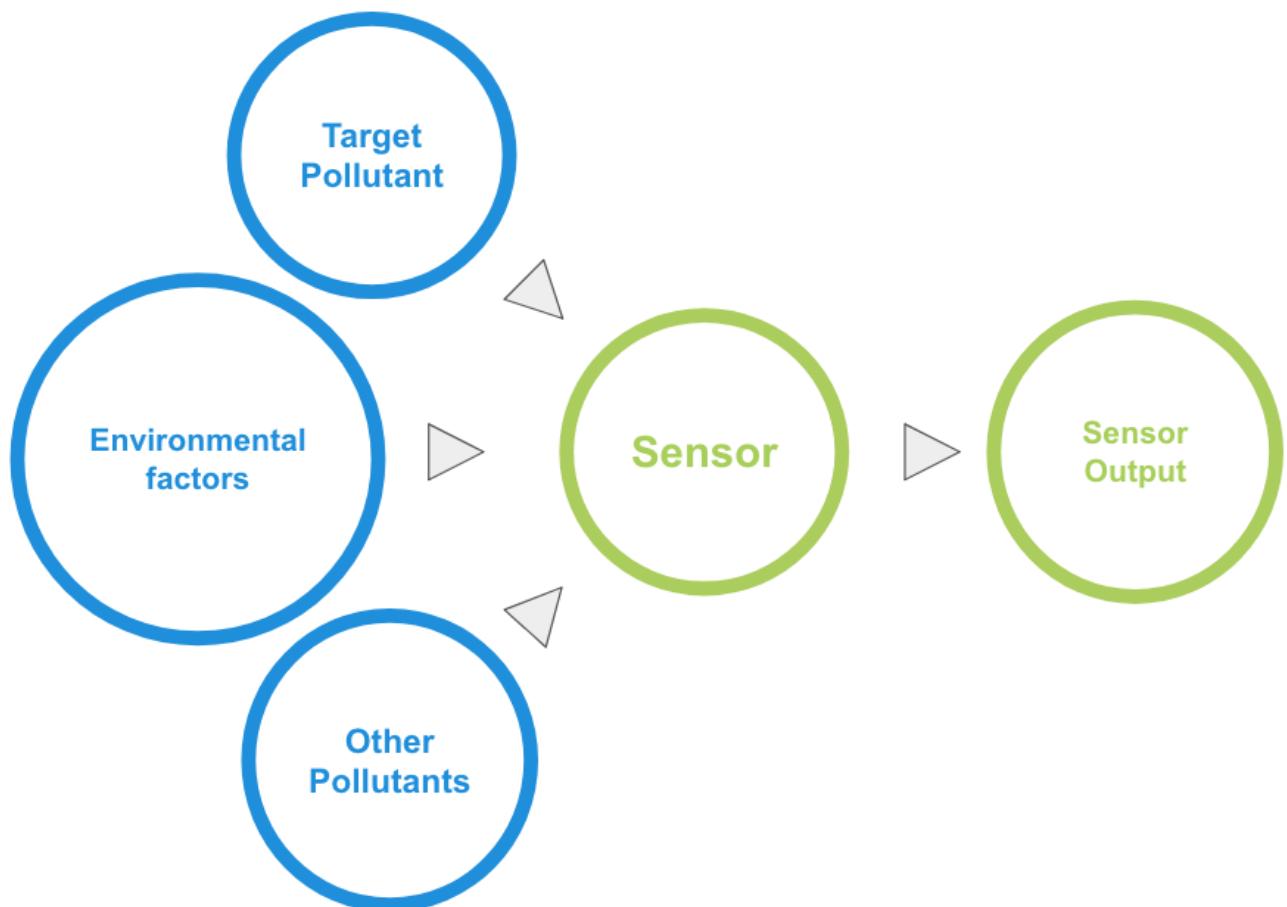
values. Either of them need large amounts of data if the models are aimed to be representative. Additionally, by the mere nature of the data and the sensors themselves, these models would need to be:

- Robusts to noise
- Capable of learning non-linear relationships
- Handle multivariate inputs
- Capable of learning temporal dependence

These needs make **machine learning** methods great candidates for modeling the data. These methods are implemented in the Sensor Analysis Framework, as well as other *more traditional* linear methods. The combination of these algorithms with large amounts of data gathered during, for instance, the iScape project, offers a great opportunity to demonstrate the use of low cost sensors for air quality monitoring.

Smart Citizen Kits

Due to their construction, low-cost metal oxyde sensors suffer from high levels of spread in their baseline resistance and sensitivity. As well, these sensors are generally reactive to other pollutants in the atmosphere, with a low selectivity of the actual target pollutant and drifts in their behaviour can be seen after some weeks of exposure. As well, metal oxyde sensors show short and long term drifts in their calibrations.



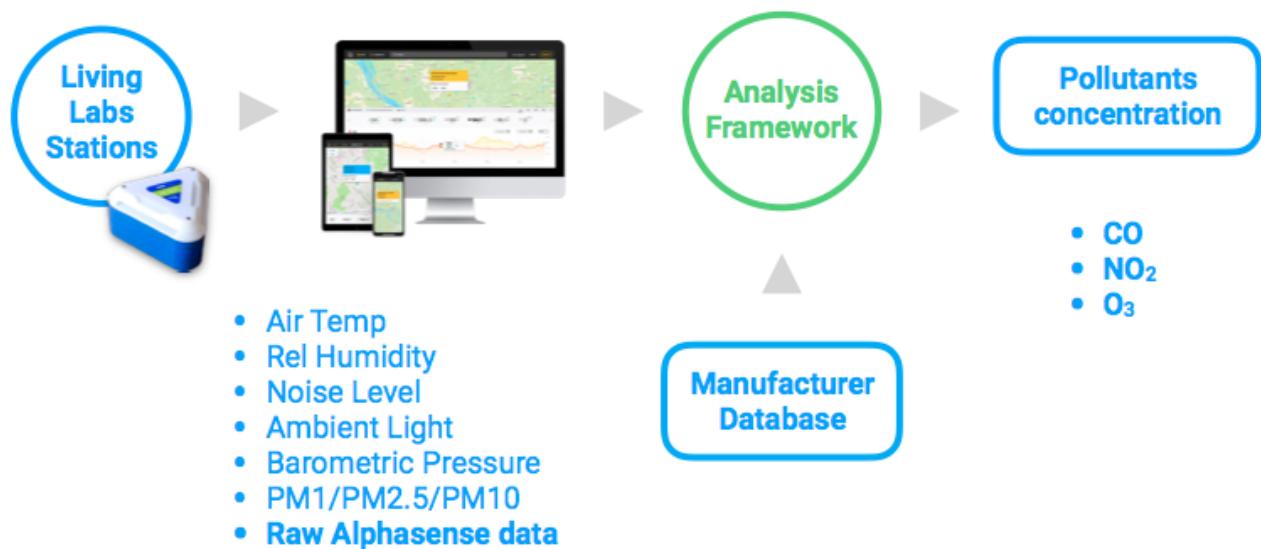
Ideally, a **sensor characterisation** in laboratory conditions is needed to assess sensitiviy, baseline resistances sensor-to-sensor spread, aiming to obtain normalising factors for each sensor or group of sensors. Even if possible, the variability of

the sensor behaviour during the deployment stage, makes the individual characterisation and calibration of the Metal Oxyde sensors unrealistic. For this reason, indicative measurements are to be expected from this type of low cost sensors. More information about the sensors present in the urban board of the SCK can be found in this section.

Smart Citizen Stations

Electrochemical sensors

These sensors can achieve significant accuracy, but they require a particular data post-processing that combines the measurement at the sensor electrodes with the sensor characterisation on the factory as well as other environmental parameters as air temperature and relative humidity (i.e. absolute humidity). Luckily the manufacturer of these sensors, Alphasense, provides us with that reference data. However, the complexity of the operations performed can not be done inside the sensing device as it uses advanced operations as well as historical data from the same device. For that reason, the data needs to be post processed using the Sensors Analysis Framework. The algorithm is **in a beta stage** and later it will be applied automatically on the data once it arrives at the platform. More details can be found in the *Electrochemical sensor baseline methodology* Section.



This process doesn't require any on-site reference data but requires the data to be processed using the manufacturer calibration reference per sensor as well as other environmental values as temperature and humidity.

PM sensor

The selected PM sensor is internally characterised by the manufacturer and, its readings are currently being evaluated. Preliminarily, the measurements can be as well improved when reference data is available, as some have noted that the PM sensors can be affected by relative humidity.

Plantower PMS 5003

Read more on the Plantower PMS 5003 implementation on the **PM Sensor Board**.

14. Sensor Platform

14.1 Smart Citizen API

The cloud-based data engine supporting: data ingestion, aggregation and retrieving. It is entirely independent of any web front-end exposing all the functionalities over a clear REST API. That allows applications to be developed on easily on top having access to all the features to create complex and rich tools. The main instance its available at api.smartcitizen.me. You can explore and contribute to the source. One examples of this tools is the Sensors Analysis Framework or the iSCAPE Virtual Living Lab, both developed during the iSCAPE project) This is free software available under GNU Affero General Public License (AGPL).

 TD;LR

Check the developers ready [API Documentation](#)

14.1.1 Data Ingestion Flow

As already described in the Supported Assets sections, above, the Platforms supports multiple sensor types and even data coming from other platforms. On the following section, we describe all the features supported when it comes to sending data to the platform.

14.1.2 Ingestion protocols

Two protocols are supported for data to be sent to the platform: MQTT and HTTP

- MQTT is the one used by constrained devices as the Citizen Sensors and the Living Lab Stations. It allows the devices to post data to the platform after they are registered. It also allows them to receive configuration options (i.e. sensors reading interval) and report errors (i.e. sensors are malfunctioning).
- HTTP is aimed at applications publishing data to the platform (i.e. an existing sensors platform that also wants to make all the data available to the platform). This API gives access to all the platform functionalities as it is part of the core Smart Citizen API. Over this API we are not just limited to publish data but to register new devices or even users.

Both protocols support transport encryption with TLS to ensure secure communication between the client and the server over the Internet.

14.1.3 Authorisation and authentication

Knowing who posts what is a serious problem when it comes to hundreds of sensor data being published per minute. Constrained hardware devices using the MQTT API use a unique device token given to the device every time it is registered on the platform. The token authenticates the devices against the platform, and it can be expired at any time to prevent a device to keep publishing. Instead, the HTTP API supports authentication using an OAuth2 or a private token. Both mechanisms work at a user level allowing a single process to manage all the devices created by a user.

14.1.4 Kits blueprints

Each device sensors configuration needs to be previously registered on the platform to ensure each datapoint published is associated with the required metadata. This information is called a Kit blueprint. The minimal blueprint includes all the necessary data that a user might provide to create a Kit. It is composed of Components, and those can reuse existing Sensors and Measurements. Sensors are the hardware or software components that record the data. Measurements are descriptions of what the sensors are recording.

14.1.5 Data Storage

Once the steps above are completed data is stored in a database cluster performing asynchronous masterless replication to ensure data backup and availability. Each datapoint is stored with the following items:

- *Component*: A reference to the component type that generated the datapoint.
- *Device*: A reference to the device that generated the datapoint.
- *Raw Data*: The datapoint as received to the platform
- *Processed Data*: The datapoint after applying post processing, when implemented.
- *Timestamp*: The time the datapoint was generated. Once stored historical data available via the Smart Citizen API. All the other services, as the Smart Citizen Webpage, access the data from there. The API also exposes a method where data is processed to a CSV file and email to the user. That allows loading the data offline to any software capable of dealing with CSV files (i.e. Microsoft Excel, MATLAB, etc.)