

# LIBAIRTY | SIOT PROJECT REPORT

## OVERALL PROJECT FLOW

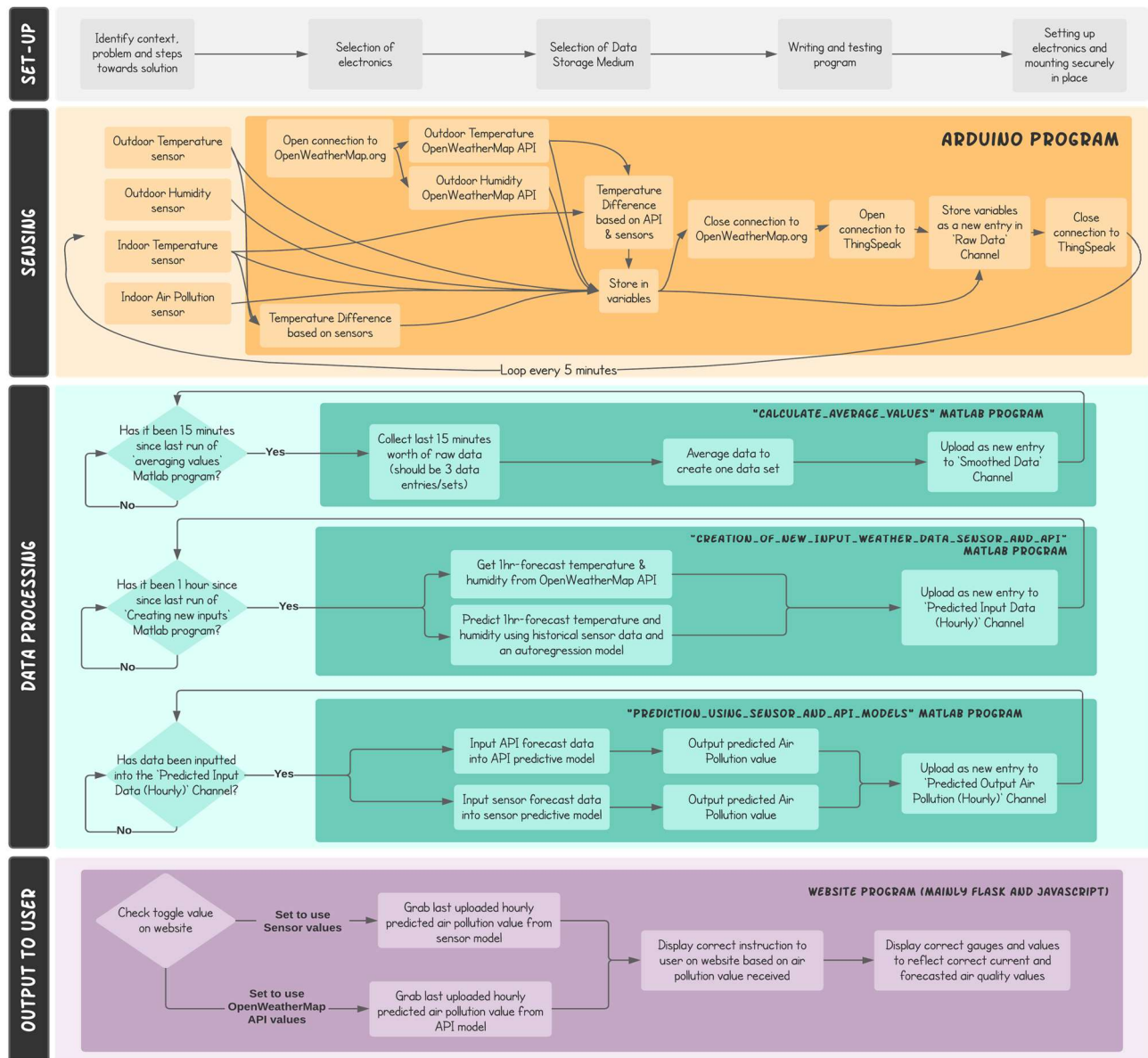


Figure 1. Flowchart depicting overall project flow

## SECTION 1: SENSING

### INTRODUCTION, OBJECTIVES & SUMMARY

Alongside my Master's research, which focuses on air purification in large indoor spaces, I was interested in looking into preventative solutions. One idea was to explore the correlation between outdoor weather and indoor air pollution, to see if we could then predict outdoor weather variables and thus predict indoor air pollution at a given time in the future. This would allow us to respond ahead of time and better control the quality of our indoor air.

Existing research already proves there to be a correlation between outdoor temperature & humidity and indoor air pollution<sup>1, 2</sup> so this project aims to develop a proof of concept prototype to demonstrate its predictive and preventative capabilities, to then potentially embed into the Master's project end-deliverable.

To gather the necessary data, the correct electronics were acquired, a program was written, and the system was then mounted and fixed in its environment to collect data continuously for over a week. The below sections cover the 'set-up', 'sensing' and the first loop of the 'data processing' sections of the project flowchart in figure 1.

(1) UCAR Centre For Science and Education. How Weather Affects Air Quality | UCAR Center for Science Education. 2020

(2) Levasseur M, Poulin P, Campagna C, Leclerc J. Integrated Management of Residential Indoor Air Quality: A Call for Stakeholders in a Changing Climate. *International Journal of Environmental Research and Public Health*. 2017; 14 (12): 1455.

## ELECTRONICS SELECTION

The selected **indoor air pollution sensor** was the **MQ-135 Gas sensor**, as it was the best within its series to measure air quality control, and can detect as well as measure NH<sub>3</sub>, NO, Alcohol, Benzene and Smoke<sup>3</sup>. It combines the readings and outputs a value in the form of PPM (parts per million), the standard unit to measure air quality.

To gather **outdoor temperature** and **humidity** values, the **DHT11** module was considered the most reliable option<sup>4</sup>, measuring well within the range of its environment. It was also a 2-in-1 module, reducing cost, wiring and overall energy. The same type of module was used indoors, to gather **indoor temperature** values (the humidity feature was ignored).

**Arduino** was chosen over Raspberry Pi, as I did not need to perform any complex or simultaneous processes, and the Arduino board was fully capable of all the tasks required of this project.

The Pretzel Board (fig. 2) was initially the chosen IOT board, as I already owned the board and aimed to reduce the overall cost to the project by using it. However, as I describe further in the ‘Sensing Program’ Section below, this board did not have the suitable specifications for this project, so I switched over to the **Arduino Nano IOT 33 Board** (fig. 3), as it contained an in-built Wifi Module and had enough memory for the purposes of this project.

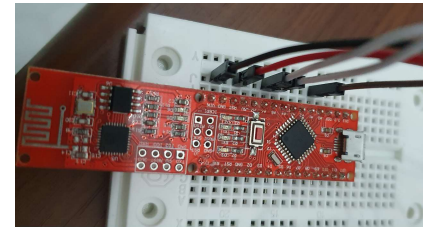


Figure 2. Pretzel Board (IOT Wifi Board)

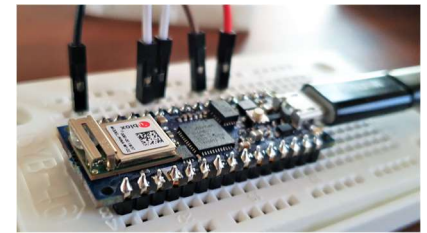


Figure 3. Arduino Nano 33 IOT Board

## SENSING SETUP

Figure 5 represents the circuit diagram, with the left-most two breadboards placed inside the room and the right-most board placed outdoors. Figure 4 shows the physical circuit set-up.

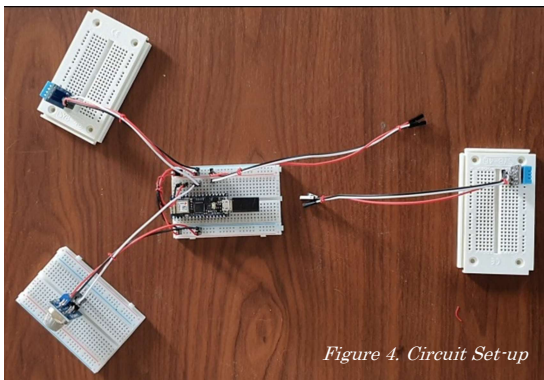


Figure 4. Circuit Set-up

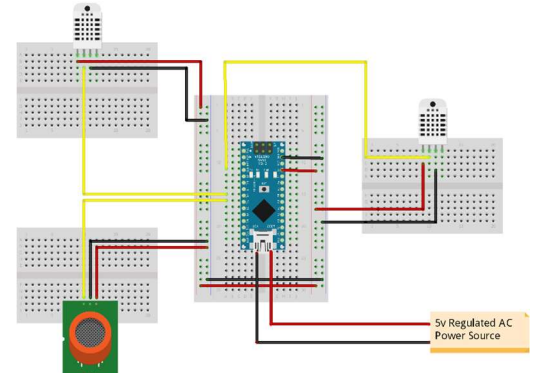


Figure 5. Circuit Diagram

fritzing

As the MQ135 sensor required a 5v input, the Nano 33 board's 5v pads were soldered together to enable the 5v output source.

The board and indoor sensors were placed on a high cupboard in my room, so that there would be minimal interference from human body heat, or accidental damage to the system. The air pollution sensor heats up to measure air quality and this heat can thus affect temperature readings. To avoid this, the air pollution sensor was kept as far away from the indoor DHT11 sensor, and facing the opposite direction (fig. 6). Wires were pulled out into the balcony, where the outdoor DHT11 sensor was mounted securely high up on the outside wall, in case of heavy winds (fig. 7). The first reason for not having separate boards situated outside and inside the house was because of the limitation by ThingSpeak to upload more than one entry for the same timestamp of data. This meant that all the data had to be sent to one board before uploading it all to ThingSpeak in one go. The second reason was that the home WiFi did not reach the balcony area so adding a second board to sync with the first via WiFi was also not

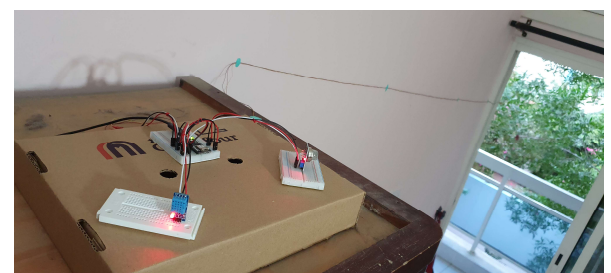


Figure 6. Board and indoor sensors fixed to top of cupboard, with wire going out to outdoor sensors

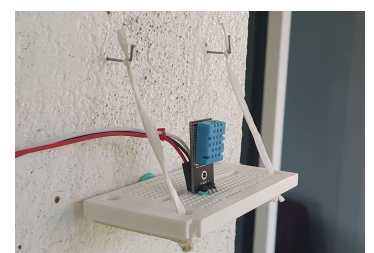


Figure 7. Outdoor DHT11 sensor fixed to balcony wall and connected to board indoors

(3) Presenting MQ sensors: low-cost gas and pollution detectors. Open Electronics. -02-16T15:20:27+00:00. 2018.

(4) Asad S. Best Temperature and Humidity Sensor Modules for Arduino – Linux Hint. "Nov 01, ". 2020.

feasible. Thus, it was decided to connect the outdoor components using a clean wiring set-up. To ensure the long cable runs did not affect the data transmitted to the Arduino board, the wires were placed far from any house wiring to avoid electromagnetic interference.

## DATA STORAGE SELECTION

To store the data as it was being collected, I initially looked at using Google Sheets because of the potential to then integrate graphs into the website in the second half of the project. However, after doing a significant amount of research, I chose to go with ThingSpeak, as its message upload limit, refresh rate, and data storage size limitations exceeded my requirements. It also had the added benefit of providing live graphs as each data entry was uploaded to the relevant channel (fig. 8). Its most important feature was its online Matlab Analysis, where Matlab programs could be written and run using timed (*TimeControl*) or response-based (*React*) triggers. These programs could pull data from channels, analyse and process them, and then write new data to another channel.

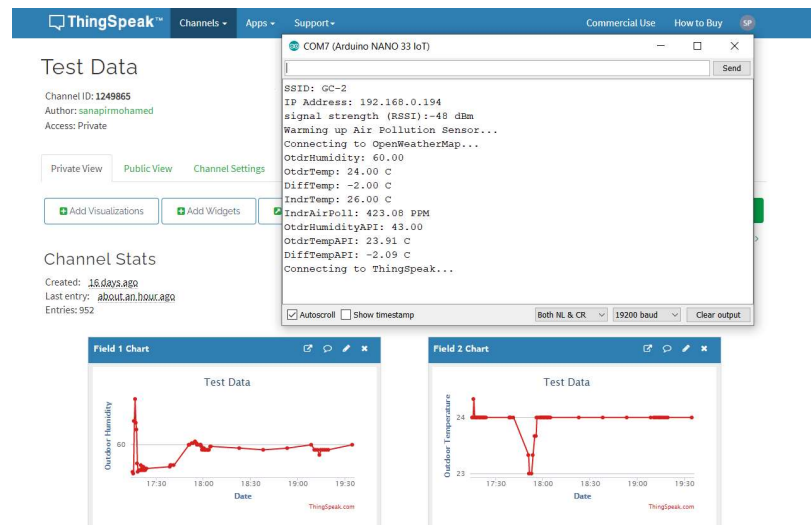


Figure 8. Screenshot of Raw Data Channel in ThingSpeak, demonstrating its live graph visuals, with Serial Monitor of Arduino displayed to assist during live testing

Two channels were created in ThingSpeak for the first half of this project: Raw Data and Smoothed Data. [Accessible in: Libairty\_SIOT > ThingSpeak Files > ThingSpeak Channel Details.txt] The first channel was used to directly store data from the sensors and API, using the Arduino's program (explained further in the 'Arduino Program' section below). The Arduino pushed data to the Raw Data channel every 5 minutes. The idea was to then use a variant of a smoothing technique by averaging 15-minutes worth of data (3 sets of 5-minute-spaced data) to come up with a average set of values every 15 minutes. This would help smooth out what I correctly expected to be a relatively noisy graph, and allow me to more clearly identify trends in the data. To achieve this, I wrote a short Matlab program on ThingSpeak to extract the last 15-minutes worth of data, average it and then upload the new data values to the 'Smoothed Data' channel. [Code available in: Libairty\_SIOT > ThingSpeak Files > Matlab Online Analysis Code > Calculate\_Average\_Values.m] This code was run every 15 minutes using ThingSpeak's TimeControl trigger function.

Therefore it essentially sampled data every 15 minutes and stored it to the Smoothed Data channel. This sample rate was chosen, because any higher was considered excess data gathered to then be analysed which was unnecceary given the not-so-frequent changes in each weather factor being recorded. And any lower may have resulted important changes in weather being missed and unrecorded. To ensure a decent window of data was averaged and written to the Smoothed Data channel, a sampling rate of 1 sample of raw data every 5 minutes was chosen to allow three sets of raw data to average to one set of Smoothed Data. The Smoothed Data was then used for second half of the project.

As a safety check, I added an if statement in lines 41-64 of the Matlab code where it would check if the last 15 minutes worth of data collected from the Raw Data was empty. This could occur if the physical system was disconnected from WiFi and unable to push data to ThingSpeak, or if there was a power failure and the

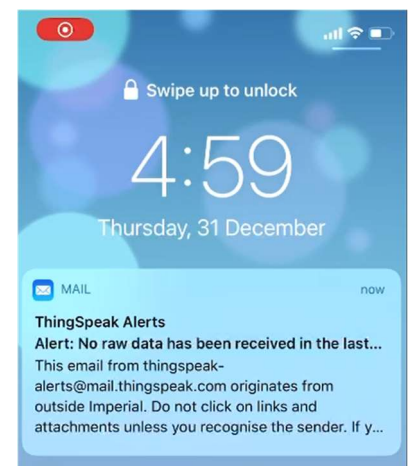


Figure 9. Screenshot of Email Alert notification if there has been no raw data in the last 15 minutes.

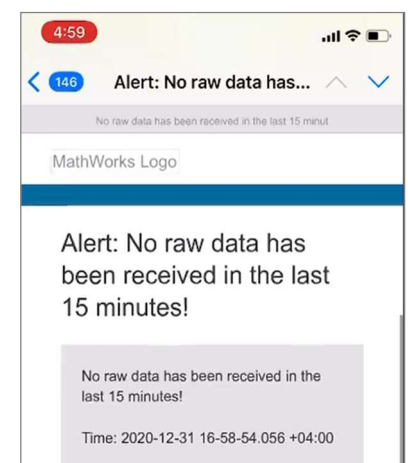


Figure 10. Screenshot of content of Email Alert if there has been no raw data in the last 15 minutes.



physical system was down. If this was the case, the Matlab Program would not calculate average values, nor upload these new (blank) values to the Smoothed Data channel as this would affect future trend analyses on the data. Instead, it would trigger an email notification (fig. 9, 10) to my email address, to make me aware of the problem and troubleshoot to fix it. This proved to be very useful as I experienced a power failure during the 1-week data collection period as well as disconnection of the wireless router, and thus failure to connect successfully to the WiFi. In both instances, I was instantly made aware of the problem and was quickly able to fix it.

## SENSING PROGRAM – DATA COLLECTION

As mentioned earlier, the project was initially started on a Pretzel Board with a WiFi module. This required me to learn to communicate with the WiFi module using AT commands, via a software serial port. However, the board did not have enough memory to store and run the entire program so I switched to the Arduino Nano 33 IOT board. The communication method with this board was much easier, using a WiFi client and associated library.

The functions and flow of the program can be seen in the flowchart in figure 11. [Code available in: Libairty\_SIoT > Sensor Files > Sensing and Storing > Sensing\_and\_Storing.ino]

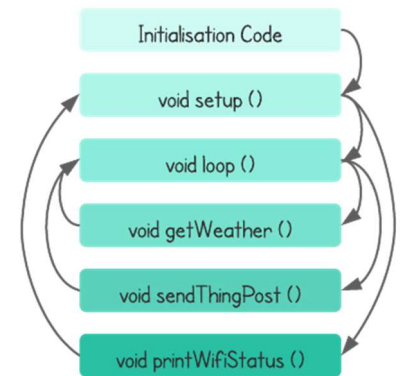


Figure 11. Arduino program functions and flow depiction

- ⊙ The **initialisation** section calls on the details in the 'secrets.h' file which contains my WiFi network and password details as well as OpenWeatherMap and ThingSpeak API keys.
- ⊙ In the **set-up** section, the board searches for the Wireless Router, checks its firmware, and then connects to it. The serial also begins in this section. A 10-second delay is provided to ensure enough time is given to establish a successful connection with the wireless router. Based on the MQ135 sensor's documentation, a further 20 seconds is also provided to allow the sensor to warm up so that it is ready to capture air quality data accurately when required.
- ⊙ In the **main loop**, the program pulls data from the physical sensors and stores them in variables. It also calls on the `getWeather()` function which pulls current weather data from OpenWeatherMap.org using an API. This data is also stored in variables. The temperature values obtained are then used to calculate temperature difference values. The final 8 weather variables below are then pushed to ThingSpeak using the `sendThingPost()` function and a 5 minute delay is triggered before the loop runs again.

### Final 8 weather variables sent to Raw Data Channel on ThingSpeak:

1. *Outdoor humidity* (from DHT11 sensor outdoors)
  2. *Outdoor temperature* (from DHT11 sensor outdoors)
  3. *Indoor temperature* (from DHT11 sensor indoors)
  4. *Temperature difference* (calculated as the difference between outdoor temp and indoor temp sensor values)
  5. *Indoor Air Quality* (from MQ135 sensor indoors)
  6. *Outdoor humidity API* (from OpenWeatherMap)
  7. *Outdoor temperature API* (from OpenWeatherMap)
  8. *Temperature difference API* (calculated as the difference between outdoor temp API and indoor temp sensor values)
- ⊙ The **getWeather()** function establishes a connection with `api.openweathermap.org` and uses an HTTP GET request and a private API key to retrieve data from the site. The data gathered is localised to my home containing the sensors (using lat-lon coordinates in the API call). The API call is also specified to exclude excess data and reduce size of extracted JSON file. I.e. it excludes daily, hourly and minutely data as only 'current' data is required. The JSON file is parsed to obtain the specific temperature and humidity values, which are then saved to their own variables as explained above. The connection to `openweathermap.org` is then closed.

- ⊙ The **sendThingsPost( )** function is similar to the **getWeather( )** function as it opens a connection to **api.thingspeak.com**, creates a message with all the variables to store and the specific channel ID it will be stored in, as well as the correct write API key for that channel. The message is created using the correct structure and syntax required by ThingSpeak and is then pushed to ThingSpeak. The program checks that this process was successful before moving back to the main loop.
- ⊙ The last function, **printWifiStatus( )** is mainly for debugging purposes. It allows me to view the connection details of the board with the wireless router as well as other information such as the signal strength, for troubleshooting purposes.

Prior to using the DHT11 sensors, both temperature and humidity levels of the indoor and outdoor sensors were checked by placing them next to each other in the same environment and taking measurements. This was done to check they were both precise and that one did not have a significant offset. This was seen to be the case.

The MQ135.h file (in the screenshot in figure 12) was used to calibrate the air quality sensor so that it is more precise and not just mapped directly from the voltage. The resultant RLOAD and RZERO values were inserted in the MQ135 Arduino Library file.

```
1. #include "MQ135.h"
2. const int ANALOGPIN=0;
3. MQ135 gasSensor = MQ135(ANALOGPIN);
4. void setup(){
5.   Serial.begin(9600);
6. }
7. void loop(){
8.   float rzero = gasSensor.getRZero();
9.   Serial.println(rzero);
10.  delay(1000);
11. }
```

```
24 // The load resistance on the board
25 #define RLOAD 10.0
26 // Calibration resistance at atmospheric CO2 level
27 #define RZERO 76.63
28 // Parameters for calculating ppm of CO2 from sensor resistance
29 #define FARA 116.6020682
30 #define PARB 2.769034857
```

Figure 12. Screenshot of MQ135.h file used to extract calibration values for the MQ135 sensor

Initially, I chose to collect data every 5 minutes, repeat this twice more, and then average the three sets of 5-min-interval data within the Arduino itself before sending the final smoothed data entry to ThingSpeak every 15mins. I then decided to reduce the processing load and time on the Arduino by utilising ThingSpeak's online manipulation features using Matlab. Thus, I decided to use the Arduino to capture data every 5 minutes and then use Matlab's platform on ThingSpeak to average the last 15-minutes worth of data and push new smoothed data to a new channel. This reduced processing time ensured that data could be sent quickly and accurately (from a time-stamp point of view). This method also reduced the required on-board memory, because fewer global variables and sketch characters were now needed.

## FUTURE IMPROVEMENTS

In the future, I would add interrupts to the Arduino program as opposed to using delays, for two reasons. The first is that we would not be relying on the count of the delay + the time taken to get to the delay from the start of the loop. The interrupts would accurately trigger regardless of the time taken to process a loop. Another reason is so that we can put the Arduino to sleep during the 5-min wait period as this would allow the system to use less energy and thus potentially run on a power pack instead of mains, mitigating power failure issues. An alternative to interrupts for time-accuracy could be the use of **millis( )** instead of **delay( )**, as that function also relies on a clock based system, as opposed to being added onto the time taken to get through the code to the delay command.

I would also improve the casing of the physical sensors to better protect them from external factors. For example, during the one-week period, I noticed that my cat had jumped up onto the cupboard and had situated himself very close to my DHT11 sensor measuring indoor temperature (fig. 13). The body heat emitted by the cat may have impacted the temperature readings had he been there long enough. Although in this instance I noticed him within minutes of him getting there, there may have been times I was unaware of this, potentially affecting the accuracy of the final readings.

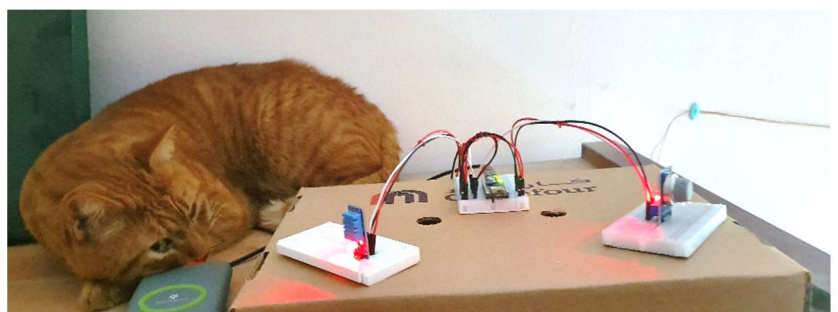


Figure 13. Cat situated near DHT11 temperature sensor (on the left-hand breadboard), potentially increasing temperature readings due to localised body heat

## SECTION 2: INTERNET OF THINGS

### INTRODUCTION & OBJECTIVES

---

The first section resulted in a smoothed output of weather data stored in a ThingSpeak channel, 'Smoothed Data'. The aim for this section was to develop a model between weather factors and indoor air pollution levels. I could then use the historical sensor data, or grab the hourly-forecast data from OpenWeatherMap.org, to predict the weather in the next hour. Feeding these values into the predictive models would give me what the indoor air pollution levels would be predicted to be in the next hour, should the room windows remain open.

The aim would then be that if the air pollution levels are predicted to enter the danger-zone (500+ PPM) in the next hour, the user would be prompted to close their windows. This way, we could prevent highly polluted air entering the room and bringing the air pollution levels as high as predicted. The next time we forecast the pollution levels in the room to drop below the danger-zone, we would trigger a prompt to open the room windows and allow the air to circulate again. This would be safe as the air pollution level that the room would reach in the next hour with the windows open would be considered 'safe' and below the danger zone.

The below sections cover the 'data processing' and 'output to user' parts of the project flowchart in figure 1.

### PRE-REQUISITES REQUIRED

---

The chosen end product for this project was a custom-made website to communicate directly with the user and instruct them on whether to open or close their windows. In order to achieve this end goal, I had to learn the below actions from scratch, having little to no previous experience in such web development projects:

- ⦿ Developing different types of regression models on Matlab
- ⦿ Extracting the models themselves and uploading them to the Matlab platform on ThingSpeak to be used during the live processing steps
- ⦿ Developing a Flask app (Flask is a Python web framework) and code in Python
- ⦿ Coding in HTML and design the bare structure of the website
- ⦿ Coding in CSS and dress the HTML code into a more user-friendly format
- ⦿ Coding in JavaScript to allow the user to interact with the page and increase its functionality
- ⦿ Using the Git command-line to create a local and online repository to store my code during the development process
- ⦿ Hosting my final Flask app on the Heroku server so that it is fully accessible to a standard user

### DATA ANALYSIS, INFERENCING & INSIGHTS

---

As shown in the 'Data Processing' section in the flowchart in figure 1, there were three loops running simultaneously. The first loop was explained in the previous section, focusing on smoothing the data. The second loop's role was to output hourly predicted input weather data. This was done by using TimeControl on ThingSpeak to trigger the online ThingSpeak-Matlab program 'Creation\_of\_New\_Input\_Weather\_Data\_Sensor\_and\_API' every hour. [Code available in: Libairty\_SIoT > ThingSpeak Files > Matlab Online Analysis Code > Creation\_of\_New\_Input\_Weather\_Data\_Sensor\_and\_API.m]. This program creates two sets of input weather data to use in the predictive models later on: Hourly predictions based on historical sensor data, and hourly forecasts straight from openweathermap.org.

For the first set of data, line 8 of the program captures the previous 24hrs of smoothed data from the Smoothed Data channel and saves it under the variable 'historicalData'. Lines 13-22 are a safety check that triggers an email alert if no historical data has been imported over the last 24hrs, as this would mean no data would have been collected by the sensors/API. Lines 30-64 contain the code to create an autoregression model that uses the historical sensor data to predict what the weather values would be in an hour's time. These values are stored in variables.

For the second set of data, I used Matlab's ability to retrieve data directly from OpenWeatherMap, using the *webread* function and the relevant parameters, and saved the received content as a struct. I extracted the

relevant data from the first row/item of the struct, as it contained the info for the first hourly forecast, and stored these in variables. The following variables were then pushed as a data entry to the 'Predicted Input Data (Hourly)' channel:

1. *Forecast Outdoor humidity* (from historical sensor data and respective AR model)
2. *Forecast Outdoor temperature* (from historical sensor data and respective AR model)
3. *Forecast Temperature difference* (from historical sensor data and respective AR model)
4. *Forecast Outdoor humidity* (directly from OpenWeatherMap)
5. *Forecast Outdoor temperature* (directly from OpenWeatherMap)
6. *Forecast Temperature difference* (calculated as the difference between direct value from OpenWeatherMap and forecast indoor temperature)

The final loop in the flowchart in figure 1 relies on the React app in ThingSpeak, which triggers the online ThingSpeak-Matlab program 'Prediction\_using\_Sensor\_and\_API\_Models' when it detects that data has just been uploaded to the 'Predicted Input Data (Hourly)' channel. [Code available in: Libairty\_SIoT > ThingSpeak Files > Matlab Online Analysis Code > Prediction\_using\_Sensor\_and\_API\_Models.m]. Line 31 of the program extracts this last entered data from the Predicted Input Data (Hourly) channel to use as inputs into the predictive models that will output the predicted air pollution levels in an hour.

Before generating the predictive models, the data was analysed for correlations using the *corrcoef(x,y)* function on the Matlab Desktop app, which showed all independant variables (from both sensors and OpenWeatherMap) against indoor air pollution to have absolute correlations between 0.35 and 0.45, suggesting weak but present correlations, confirmed graphically.

I then decided to go ahead and create the predictive models on the Matlab Desktop app, using the Smoothed Data gathered over the 1-week period. One model was made using the OpenWeatherMap weather values as inputs, and a second was made using the physical sensor values gathered as inputs. Both models used the indoor air pollution sensor readings as the outputs of the models, training them to use input weather data to then output a matching expected air pollution level. [Code available in: Libairty\_SIoT > ThingSpeak Files > Matlab Models > Sensor\_Model\_Code (or API\_Model\_Code).m]. Using the *fitlm* function in Matlab, I compared single-variable and multi-variate linear regression models between the independent variables and the corresponding air pollution values. The multi-variate regression models considering all three inputs – outdoor humidity, outdoor temperature and temperature difference, using the sensor/API data respectively for each model – showed the highest r-squared value, so were chosen for this project. Both final models were evaluated and assessed for usefulness to be used in the project. It was found that whilst they showed relationships between the independent and dependant variables, the variance of the data against the fit line was high, hence resulting in a low R-squared value. This R-squared value was higher for the sensor-based model compared to the OpenWeatherAPI-based model, but minimally so. I decided to use these models nonetheless as there was an underlying, albeit weak, correlation that was sufficient for proof-of-concept predictive purposes.

Following this, the models themselves were extracted and saved as .mat files. [Files available in: Libairty\_SIoT > ThingSpeak Files > Matlab Models > Sensor\_Model (or API\_Model).mat]. These were then zipped and saved in my DropBox. A shared link to the files in DropBox were generated and used to import the models into the online ThingSpeak-Matlab program 'Prediction\_using\_Sensor\_and\_API\_Models' in lines 4-12. In the program, weather data inputs 1-3 from the above list of variables that were sent to the Predicted Inputs channel are pushed through the sensor-based model, whilst data inputs 4-6 are pushed through the API-based model. Both predicted air pollution outputs are saved as variables and then written to the 'Predicted Output Air Pollution (Hourly)' channel on ThingSpeak.

It may be worth noting that I tested the effectiveness of these predicted air pollution results by opening and closing my room window accordingly when the predicted air pollution values surpassed the safety limit. Over three days, I noticed an overall average drop of 22.7% in the indoor air pollution levels, using the sensor-based predictive system. I repeated this with the OpenWeatherAPI-based model and noticed an overall drop of 20.8%. Whilst these are relatively small improvements, they are still significant, very similar to each other and were also found to be

quite consistent over the three days of testing. This proves the system and models do work but have the potential to be improved further.

## DATA VISUALISATION - WEBSITE FUNCTIONALITY, DESIGN & TESTING

As I was using Flask for my web framework, the first stage was to create a .py file to essentially run the app. This was the 'main.py' file. [File available in: Libairty\_SIoT > main.py]. Its main function is to initialise the app (in line 7); link the route/url with the correct html pages; and, perform back-end commands before sending information through to the associated web pages that are being opened. For example, in lines 14-52 for the home page, I use the HTTP GET method to grab the latest two predicted output air pollution values as well as the current air pollution value, and they are stored in variables. The values of the predicted air pollution values are then used to decide what instruction is sent through to the home page to be displayed, as well as what quote should be displayed based on the current air pollution value. This code is placed within the home function, def home(), so that the code can be run every time the page is refreshed, to update the air pollution values obtained.

Templates were used to make the html editing process easier and cleaner. The 'layout.html' file [available in: Libairty\_SIoT > templates > layout.html] is the main template for the website, containing the navigation bar code, main custom CSS file and bootstrap CSS. Dedicated blocks are added to incorporate the differing content for each of the Home, Sbout and Live Data pages which are extended from the Layout page. *Block content* addresses the key content going onto each page. *Block extendedCSS* contains the additional CSS file related to that specific page, e.g. the home page. *Block extendedHead* was added to give automated refresh functionality to the home page by adding line 5 in 'home.html' to the head of its main layout template page [file available in: Libairty\_SIoT > templates > home.html]. Javascript code was also directly linked in the layout file to allow for further functionality.

One custom javascript file linked was 'customJavaScript.js' [file available in: Libairty\_SIoT > static > customJavaScript.js]. It contained all the javascript code I had written for use throughout the entire website, hence was linked in the main layout template page, thus accessible to all extended pages. It was also used within the layout page itself which contained code for the navigation bar. When clicking on a navigation button, the JavaScript function *function()* would be triggered to create a bouncing motion and give the page more of a user-friendly feel (fig. 14). This animation type was chosen following implementation of multiple alternatives, to see which one felt 'fun' yet clean and fitting with the theme of the page.



Figure 14. Demonstration of About page navigation button bouncing up when hovered on

The home page mainly displayed the predicted and current air pollution data in the room, and instructed the user on what to do at the given instant. As seen in figure 15, gauges were created and imported directly from ThingSpeak to visually show live and regularly updated values for both the current and predicted air pollution. From the main.py file, the air pollution values were passed through to the home.html file and displayed on the home page, as well as their associated instructions and quotes. Customised quotes depending on the current air pollution were included to give the user more of an engaged feel, where they could understand their current situation and get feedback that matched accordingly. User feedback showed this to be useful in achieving high engagement with the site, as people became more intrigued after knowing the state of their air in the given moment and receiving fun, relatable and applicable feedback.

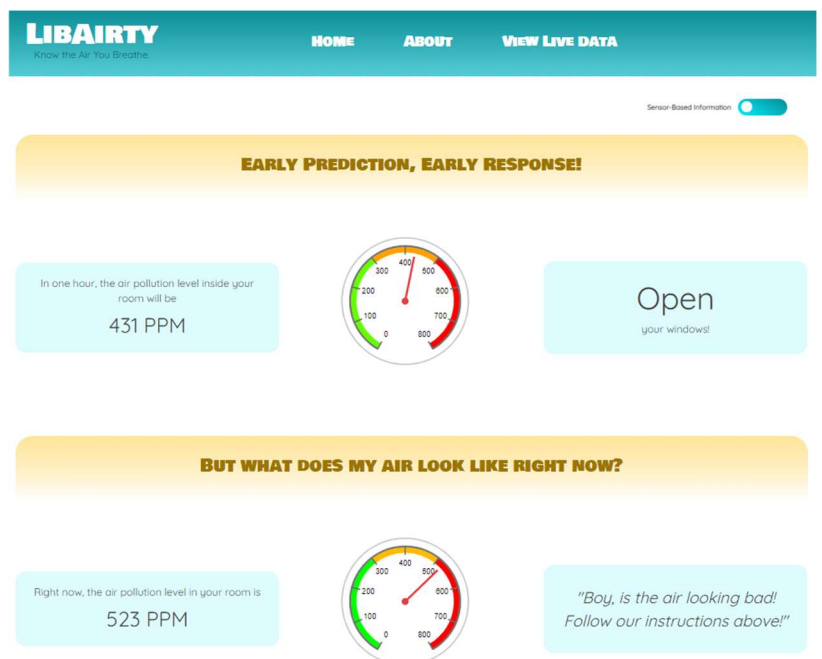


Figure 15. Screenshot of scrolled-down home page of LibAirty website



Also visible in the top-right of figure 15 is a toggle, which I coded in to provide the user full functionality over which result-type to view (i.e. the predicted air pollution based on the sensor model and sensor inputs or based on the OpenWeatherMap model and inputs). Flicking the toggle triggers a custom JavaScript function *toggleFunctions()*, which swaps the gauge, air pollution values and associated instructions from sensor to OpenWeather and back according to the toggle value (as shown in lines 17-35 in customJavaScript.js, and figure 16).

```
17 function toggleFunctions(sensorOutputAirpollData,APIOutputAirpollData) {
18   var x = document.getElementById("toggleValue");
19   var airpoll = document.getElementById("outputAirPollValue");
20   var winAct = document.getElementById("windowActionValue");
21   var airpollgauge = document.getElementById("airpollgauge");
22
23
24   if (x.innerHTML === "Sensor-Based Information") {
25     x.innerHTML = "OpenWeatherMapAPI-Based Information";
26     airpoll.innerHTML = APIOutputAirpollData + ' PPM';
27     winAct.innerHTML = APIWindowAction;
28     airpollgauge.innerHTML = '<iframe width="450" height="260" sty
29
30
31   } else {
32     x.innerHTML = "Sensor-Based Information";
33     airpoll.innerHTML = sensorOutputAirpollData + ' PPM';
34     winAct.innerHTML = sensorWindowAction;
35     airpollgauge.innerHTML = '<iframe width="450" height="260" sty
36
```

Figure 16. Snapshot of toggle function in customJavaScript.js

After demonstrating the website and its functionality to a few users, it quickly became obvious that they were very unclear as to what the predicted value was referring to and what the instructions meant. To provide clarity, I decided to add some user-friendly introductory text to explain the project from an end-user point of view.

The About page was created to provide a space to explain the project from a more technical standpoint. Here, I chose to: embed a link to the project video via YouTube; include some accompanying information on how the system works; add a hyperlink to the project's GitHub Repository; and, add an image of the project flow (included in this report in figure 1). An 'About the Designer' section was also added to allow the users to connect with a face behind the project, and as a way of containing everything within the website itself.

The final page, View Live Data, was a place to easily view all the data gathered and processed for this project, which had been stored on the four previously mentioned ThingSpeak channels. Clean heading designs were used to separate each of the four channels, with short descriptions included to provide clarity on the purpose of each channel. Dropdowns were used to control the information shown and further minimise the clutter on the page. These dropdowns allowed the user to select which particular graph

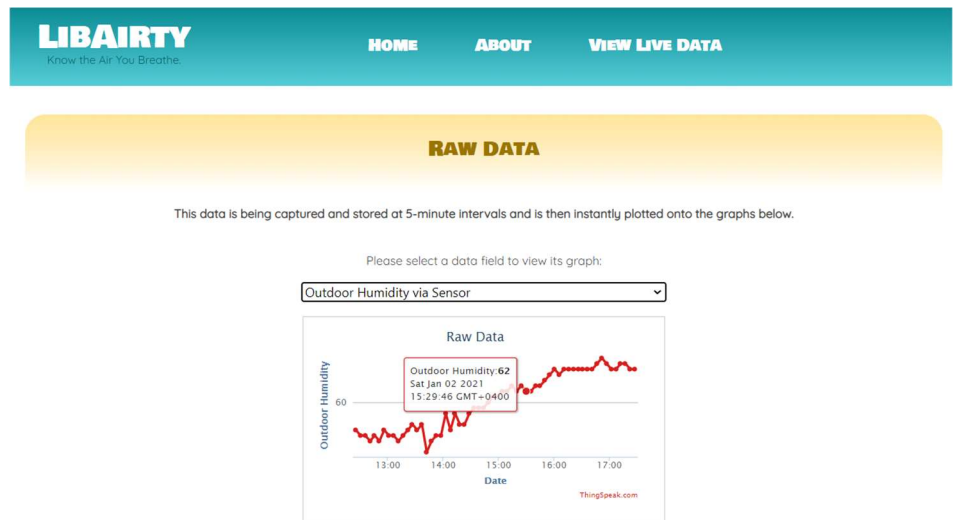


Figure 17. Screenshot of View Live Data page and interaction with embedded graph

of the specific channel to view. The code was written such that when clicking an option in the dropdown, the JavaScript function *rawgraphchangeFunction()* (for the Raw Data section in this example) would be triggered. It would call on the related graph from ThingSpeak depending on the option selected by the user. I chose to use ThingSpeak's iframe embedding functionality as it retained the interactive aspect of the graphs (fig. 17) and automatically updated them as new data points were entered into the associated channels and fields.

## HOSTING ON HEROKU

To host the Flask app on Heroku, I used the Git Bash terminal to obtain certain files, including a Procfile to let Heroku know I was using Gunicorn as the professional web server [available in: Libairty\_SIOT > Procfile]. I also created a Requirements.txt file [available in: Libairty\_SIOT > Requirements.txt] which contained all the libraries required to be downloaded by Heroku in order to run the Flask app successfully.

I then initialised a local git repository and committed all files to the repository before pushing the files to a dedicated url on Heroku ([www.libairty.herokuapp.com](http://www.libairty.herokuapp.com)). I then linked the local repository to an online GitHub repository to provide public access to the project files. This way, it became easier to update code files within the local git repo, online repo and the Heroku server. It also allowed me to utilise the version control features of Git and safely make larger changes with the ability to go backwards if needed.

## FUTURE IMPROVEMENTS

In the future, I would develop an accompanying mobile app using Flutter, to provide users with easier access to the instructions. I would also incorporate email or mobile alert notifications when instructions change from ‘Open’ to ‘Close’ windows or vice versa for an easier user-experience.

It could also be useful to add an element of actuation to the window where it would automatically respond to the given instructions and open or close as required, removing the need for human interference all-together. This would increase the reliability and effectiveness of the system.

It would also be important to look further in the predictive modelling methods and select more thorough models – such as a neural network – over the currently used linear regression models. This could significantly increase the accuracy of the predicted values. Incorporating lags of explanatory variables may be a way of taking into account the time delay when outdoor weather variables change and how soon they impact indoor air pollution. Using higher quality sensors may also increase the accuracy of the data collected and thus better models and predictions could be generated. For the purposes of this project however, where the main aim was to provide a proof-of-concept for this predictive system, I believe that the models used were satisfactory.

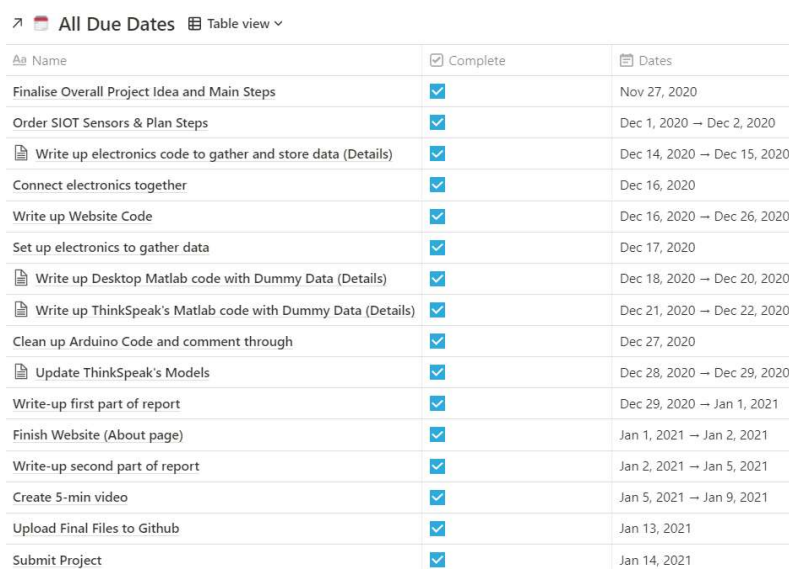
## AVENUES FOR FUTURE WORK AND POTENTIAL IMPACT

As mentioned above, this project successfully proves the idea of using predictive systems to improve indoor air quality. With further development of the predictive models, I believe this concept has the potential to be embedded in my Master’s Project end-deliverable, which involves the design of a new and innovative air purification unit. The predictive system could connect directly to window actuation systems in new buildings to reduce the overall air pollution indoors, and as a result the air purification unit would not need to work as hard to keep the air clean. This could be beneficial in terms of saving energy, time, lowering noise, reducing maintenance frequency, and increasing the lifespan of the unit.

Following improved design of the sensor compartment for increased protection and easy mounting, I believe the system can be scaled up from a prototype to useable product, regardless of the potential to actuate windows in one’s room. The system would perhaps need to rely on Bluetooth communication between indoor and outdoor sensors, as opposed to pulling wires between the two sub-systems. As mentioned earlier, this project has also proven that using either OpenWeatherMap.org or the physical sensors as the basis for the predictive system provides relatively accurate and very similar air pollution predictions, with both methods reducing the overall indoor air pollution in my room when tested. Thus, it could be that aside from the indoor air pollution sensor, all other physical sensors are removed from the system, and that the OpenWeatherMap predictive model and system is used. This could significantly save on cost, maintenance and assembly, making LibAirty a much more feasible system to implement in homes or other indoor spaces.

## TIME PLAN

Notion was used to set out a feasible time-scale in which to complete everything as well as track my progress. The main table view of the time plan can be seen in figure 18. Throughout the project, I tried to maximise efficiency by completing tasks simultaneously, in anticipation of needing time to troubleshoot during the coding stages. For example, during the 1-week data collecting period, I began writing up the predictive modelling code using dummy data. This way, when the 1-week data was ready for use, it was a simple update in the code to call the correct ThingSpeak data, followed by an update of the predictive models and their associated .mat files.



All Due Dates Table view		
Name	Complete	Dates
Finalise Overall Project Idea and Main Steps	✓	Nov 27, 2020
Order SIOT Sensors & Plan Steps	✓	Dec 1, 2020 → Dec 2, 2020
Write up electronics code to gather and store data (Details)	✓	Dec 14, 2020 → Dec 15, 2020
Connect electronics together	✓	Dec 16, 2020
Write up Website Code	✓	Dec 16, 2020 → Dec 26, 2020
Set up electronics to gather data	✓	Dec 17, 2020
Write up Desktop Matlab code with Dummy Data (Details)	✓	Dec 18, 2020 → Dec 20, 2020
Write up ThingSpeak's Matlab code with Dummy Data (Details)	✓	Dec 21, 2020 → Dec 22, 2020
Clean up Arduino Code and comment through	✓	Dec 27, 2020
Update ThingSpeak's Models	✓	Dec 28, 2020 → Dec 29, 2020
Write-up first part of report	✓	Dec 29, 2020 → Jan 1, 2021
Finish Website (About page)	✓	Jan 1, 2021 → Jan 2, 2021
Write-up second part of report	✓	Jan 2, 2021 → Jan 5, 2021
Create 5-min video	✓	Jan 5, 2021 → Jan 9, 2021
Upload Final Files to Github	✓	Jan 13, 2021
Submit Project	✓	Jan 14, 2021

Figure 18. Screenshot of Project Time Plan on Notion (Table View)

## PROJECT LINKS

---

Video: <https://youtu.be/H50gOqxOeks>

Website: <https://libairty.herokuapp.com/>

GitHub: [https://github.com/sanafp/LibAirty\\_SIoT](https://github.com/sanafp/LibAirty_SIoT)