hackAIR

OPEN PLATFORM

D3.6: 2nd Design Guidelines for Open Sensor fabrication

WP3 – Collective sensing models and tools

Document Information

Grant Agreement Number	688363	Acronym	hackAIR	
Full Title	Collective awareness platform for outdoor air pollution			
Start Date	1st January 2016	Duration	36 months	
Project URL	www.hackAIR.eu			
Deliverable	D 3.5 – 1st Design Guidelines for Open Sensor fabrication			
Work Package	WP 3 – Collective sensing models and tools			
Date of Delivery	Contractual	1st July 2017	Actual	3rd July 2017
Nature	Report	Dissemination Level	Public	
Lead Beneficiary	DRAXIS			
Responsible Author	Ilias Stavrakas (DRAXIS), George Hloupis (DRAXIS), Demosthenes Triantis (DRAXIS), Konstantinos Moutzouris (DRAXIS)			
Contributions from	Hai-Ying Liu (NILU), Arne Fellermann (BUND)			

Document History

Version	Issue Date	Stage	Description	Contributor
0.1	20/05/2017	Draft	First draft	Ilias Stavrakas (DRAXIS), George Hloupis (DRAXIS)
0.2	10/06/2017	Draft	Integration of the COTS node	Demosthenes Triantis (DRAXIS), Konstantinos Moutzouris (DRAXIS)
1.0	20/06/2017	Draft	Internal review	Hai-Ying Liu (NILU), Arne Fellermann (BUND)
2.0	3/07/2017	Final	Integration of review comments	Ilias Stavrakas (DRAXIS), George Hloupis (DRAXIS)

Disclaimer

Copyright message

Contents

Table of figures

1 Executive summary

The current report is the second version of the design guidelines for the hackAIR open sensor fabrication that describes the implementation details of three open hardware solutions along with an initial version of user guidelines for the construction of each one. Herein, the final version of the open hardware sensors, is described after accumulating the experience of the first 18 months of the project. Specifically, hackAIR sensor node's networking capabilities are enriched as well as new improved versions of both the hardware and the software are implemented and are described here. The adopted amendments regarding the hardware and software originated from potential users' feedback as well as after consultation with the project partners. The solutions regarding the open hardware devices that are used as hackAIR sensors for data acquisition are the following:

1. An Arduino based air quality monitoring node
2. A Programmable System-on-Chip (PSoC) based air quality monitoring node (BLEAir)
3. A Commercial of the Shelf (COTS) air quality monitoring node.

Both the Arduino and the BLEAir nodes have similar aims and design principles. Unlike COTS, which aims to provide an air quality estimation using non-electronic materials that are easily available, the two nodes are designed to be constant measuring electronic stations providing relatively accurate data (when compared to expensive lab instruments).

The first and second chapters provide general information regarding fundamentals for the designed systems. Specifically, a justification for the main decisions that were taken during the project is presented. Main decisions deal with the networking of the designed hackAIR nodes, the power consumption and finally the selection of the most appropriate sensors that best fit in the requirements of low cost and high accuracy.

The third chapter describes the Arduino implementation details of one of the open hardware solutions that will be used for data acquisition in the hackAir platform. The proposed hardware solution is implemented using an Arduino device, an addon board for networking (WiFi or Ethernet) and an Air Quality sensor.

An Arduino node consists of the following:

1. A user-programmable Arduino board (ex. Arduino Uno)
2. An addon-board for internet connectivity
3. Arduino Ethernet Shield v2
4. hackAIR WiFi shield
5. An air-quality sensor

The forth chapter of the current report is to present the elements and describe the procedures needed to successfully develop a hackAIR Bluetooth Low Energy based node. The proposed solution can be implemented using a PSoC/PRoC BLE module which will transmit the acquired values from sensor to a smart hone (equipped with hackAIR app). The required elements for the proposed implementation are:

1. A PSoC/PRoC BLE module
2. A compatible air quality sensor
3. Mini USB breakout
4. Jumper cable
5. Battery

Each hackAIR BLE node will be programmed by a low cost programmer (which described in the following sections). A set of precompiled binaries (one for every supported sensor) was created in order for the user to be able to flash the

BLE module without further dependencies. In addition, and for experienced users that want to experiment further, we provide the code binaries totally open in a form that can be used by free development tools for BLE modules. Each procedure in the current report is described in a step-by-step approach allowing the familiarization of unknown actions in a shortened period.

Finally, the last chapter presents the hackAIR COTS sensor. The goal of this work is to research the optimal method for emblazonment and estimation of microparticles' dimensions, which accumulate on test surfaces with petroleum products coating. Thus, an estimation of the particles' concentration in the atmosphere is possible, with the usage of Computer Vision. It is desirable that system users be able to assess air quality by using commercial off the shelf (COTS) materials.

Through the hackAIR COTS sensor we aim to create, a low cost, low tech air quality sensor, which will provide a reliable qualitative measurement, regarding the atmosphere's concentration of Particulate Matter (PM). The sensor comprises solely with commercially available off-the-shelf products. In order to determinate the effectiveness of the proposed algorithm for blob detection, a series of combined measurements between the hackAIR COTS sensor and a commercially available air quality sensor was conducted.

For the purpose of calibrating the designed hackAIR COTS sensor, the DC1100 Air Quality Monitor was selected which displays the concentration of particles in the air, and specifically particles in 0.01 cubic foot of air, whereas the hackAIR COTS sensor detects the number of particles on the test surface after a layer of petroleum jelly is applied, through the usage of Computer Vision. The same commercial hardware (i.e. DC1100) was used in order to evaluate the Arduino and PSoC hackAIR nodes performance.

2 Design methodology-Sensors and processing node fundamentals

2.1 Introduction

During the proposal writing period it was scheduled to implement hackAIR sensing devices that fulfill certain specifications. The most critical of these specifications involve: the portability of the nodes, the low cost of node implementation, the nodes should incorporate open hardware and software technologies, the components that someone should use in order to implement a hackAIR node should comprise simple widely used electronic components and Commercial off the Shelf (COTS). After considering the available solutions and the state of the art technologies and devices it was decided to propose and implement three possible solutions that each of them has specific advantages depending on its field of application.

Primarily, as an initial step and in order to become confident of the measurements accuracy of the available in the market sensors it was decided to perform comparative measurements of commercial systems and low cost sensors. The tested sensors were separated into two main categories: LED and LASER sensors. It was concluded that the LASER sensors were of higher cost but their accuracy was significantly better. Taking into consideration the trend of rapid price reduction it was decided to include for the hackAIR nodes mainly the LASER sensors.

Another significant decision was to select the platform of the processing part of each node. Taking into consideration the wide Arduino users community and the significant networking flexibility of such a solution it was decided to design the Arduino hackAIR node following two networking solutions. The first incorporated the Ethernet networking capabilities and the second WiFi networking. To implement the latter one a complete Arduino shield was designed (https://github.com/hackair-project/hackAir-ArduinoWiFi) enabling the Arduino to register automatically to any WiFi access point that incorporates WPS security protocol. The main disadvantage of the proposed solution is the limited portability since it requires WiFi or Ethernet access to the internet as well as power supply plug. The WiFi shield is currently in version 2 that is considered as to be the latest version. Taking into consideration the above portability limitations it was decided to keep the low cost of Arduino hackAIR node avoiding the use of a GPS on the node. Instead it was decided to let a hackAIR user add his own geolcation when registering his node to the hackAIR portal.

The portable solution was selected to be implemented on the Cypress PSOC platform. This solution supports the same sensors set with the Arduino hackAIR node. The Bluetooth Low-Energy (BLE) transmission protocol was decided to be implemented and send PM measurements in beacon mode. This solution provides long battery lifetime since the PSOC hackAIR node may run for weeks without recharging the power bank that supports it. Any portable device that approaches the PSOC hackAIR node and includes the hackAIR mobile software may receive the data that are send over in beacon mode and transfer them to the central hackAIR portal after adding metadata like geolocation and the user credentials.

Finally, a computer vision algorithm was designed and all the required software is implemented in the hackAIR mobile application in order to be able to estimate the PM concentration using only Commercial off the Shelf components (COTS). The sensor consists of an empty milk box covered with petroleum jelly exposed for 24 hours on the PMs. After collecting the milk box and using a widely available smart phone macro-lens a picture is taken as close as possible. The smartphone runs the computer vision algorithms on the captured photo and send the results to the central hackAIR system. This solution is not as accurate as any other of the proposed solutions thus it was decided to provide only qualitative estimation of the PM concentration separated into three levels (low- medium – high)

It is significant to mention that all the networking protocols and packet structures that were implemented in order to submit the PM measurements to the central hackAIR system from any of the designed nodes are already published widely enabling any other community that works on similar projects to include the data structure in their hardware and with minimum effort to also submit their measurements to the hackAIR central system. The following table summarises the basic specifications of the adopted solutions regarding the processing unit of the designed hackAIR

nodes (i.e. Arduino node and the PSoC node). The two proposed implementations target specific needs regarding the PM measurements (portability, power consumption, networking etc.):

Table 1: Main characteristics of Arduino based and PSoC BLE hackAIR nodes

	Arduino hackAIR node	**PSoC BLE hackAIR node**
Processing Device	Arduino	PSoC/PRoC BLE
Main Purpose	Home Monitoring (indoor or outdoor)	Beacon measuring Node
Approximate Cost(€)	Arduino (25) + sensor(43[*1]) + power bank/wall mount (7) + WiFi/Ethernet (10/15)	PSoC/PRoC BLE module(10/20) + laser sensor(43 [*1]) + power bank (7)
Market availability	Widely available in electronic components stores and European DIY distributors (Farnell, Digikey, DfRobot, Mouser, CoolComponents, EBVElectronik, etc.)	
Power Supply	Wall mount PSU	Wall mount USB
Battery Requirements	5V Power Bank (min 2600mAh)	5V Power Bank (min 2600mAh)
Minimum Battery Operation	20 hours	2 days
Connectivity	Wireless and Ethernet network	Bluetooth Low Energy (BLE)
Connectivity to Mobile	During system initialization and configuration	YES
Method for Data transmission	Through WiFi	BLE beacon to smartphone and then any type of internet connection
WiFi Availability	YES	NO
Setup Procedure	Through Arduino IDE + WiFi	Low cost programmer (5-10€) + free software + hackAIR firmwares for uploading
User Programming	hackAIR library + Arduino IDE	Not required (but available if user wants to modify initial firmware code)
Geolocation	Yes using user's profile geolocation details (or from optional GPS shield)	Yes (From mobile)
Assembly	Guidelines using COTS	Guidelines using modules
Other Characteristics	Big Arduino community	More focused to specific users and up to date gadget users
		Low Power Consumption
	Other (lower cost) sensors available but with poorer accuracy	
	Better for ordinary people	Better for experimenters
	Can be used as school project	Can be used as maker/hacker project
	Ability to use power banks for extended operation time	

[*1]: Approximate cost since it depends on the selected PM sensor.

The design for the Arduino based node as well as PSoC BLE and COTS nodes are presented in detail in the current document. Apart from the different approaches that are implemented for each type of node, there are some features which are common to both devices regarding sensor interfaces. Thus, a short presentation of the selected sensors and their requirements is also included in this document.

Arduino and PSoC BLE implementations are presented in Figures 1 and 2 and, finally, instructions in the form of implementation guidelines are introduced. Regarding the design of COTS air quality estimation system, non-electronics materials that may be found in hardware stores are used so that users build a system that is able to

depict the particulate matter concentration using paper and/or fibrous filters. Using a smartphone's camera, a photo of the filter is taken and uploaded to the hackAIR platform in order to be analyzed by means of image processing algorithms.

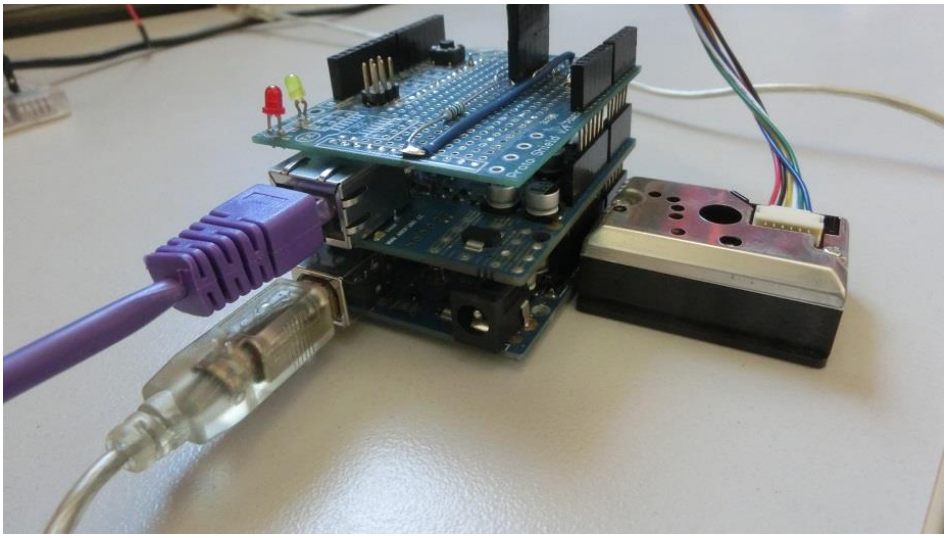

Figure 1: Assembled Arduino node

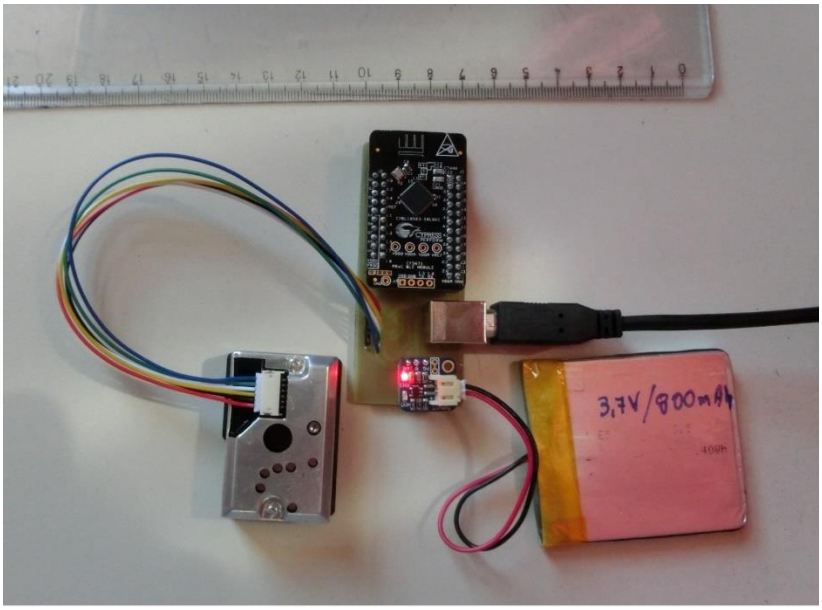

Figure 2. Assembled PSoC BLE hackAIR node

2.2 Low cost PM sensors

The Arduino based node as well as the PSoC BLE node share some common features along with many differences as presented in Table 1. Except from the common type of power supply unit that they are able to use (battery or power bank), their major common feature is the type of sensors that they will use to obtain PM measurements and transmit them to the hackAIR platform. The output of each of the proposed sensors (after the necessary conversions) is processed by the corresponding Micro Controller Unit (MCU) preparing the data that will be transmitted either wirelessly (by means of WiFi or Bluetooth protocols) or using Ethernet.

The sensors included in Table 2 are the sensing devices that are proposed to be used for PM sensing in the hackAIR open hardware design. A crucial requirement for the selection of a PM sensor is its low cost along with the high accuracy and repeatability specifications.

All the sensors are based on the same measurement principle, which is known as *optical scattering*: the airborne particles flowing across a light beam scatter the incident light; the scattered light reaches an optical detector which generates a signal proportional to the received light intensity.

Table 2: Main technical characteristics of the proposed sensors

id	SENSOR TYPE	Light beam	Signal Output	Output signal form	Forced Air flow	Result (Particles)	Sensing range (μg/m^3)	Relative Error	PRICE* (euro)
1	Shinyei PPD42NS	LED	Digital	Pulses (PWM)	No	Big (>$PM_{2.5}$) and small (<$PM_{1.0}$)	1-500**	5%	18
2	DFRobot SEN0117	Laser	Digital	Serial output	Yes (by fan)	$PM_{1.0}$, $PM_{2.5}$, PM_{10} individual values in μg/m^3	0-500	5%	42
3	iNovaFit SDS011	Laser	Digital	Serial output	Yes (by fan)	$PM_{2.5}$, PM_{10} individual values in μg/m^3	0-999	5%	35

*prices derived on 1/5/2017 from at least one international distributor

** Converted values from [1] since Shinyei PPD42NS output results in particles/283ml-0.01cf.

As presented earlier, there are three communication protocols to attach the sensors to the Arduino and the PSoC BLE processing devices. These types of data interface are:

a) Analog reading (range 0.5-3.6V max)

b) Digital (pulse width reading)

c) Serial by means of Universal Asynchronous Receiver/Transmitter (UART)

The proposed implementations for each device fulfill all the above interface requirements. In addition, since a) to c) requirements are the industry standards they will be included in any forthcoming PM sensors[3],[4]. In any case since the proposed solutions involve open hardware and software tools like Arduino and PSoC that are widely accepted and several technology communities work on them all the necessary steps (like documentation, software and hardware licensing, simplicity of implementation and network packet description) are considered in order to enable users/communities to further develop hackAIR systems making libraries for new sensors or/and processing units like nodemcu and others. Under this approach all the proposed implementations presented below are capable to support various PM sensors providing in this way guaranteed sustainability to the hackAIR project since the nodes are designed in a way that will also allow the use of other $PM_{2.5}$/PM_{10} sensors than the ones tested now (in case these sensors are compatible with data interfaces as presented above). The PM sampling rate is fully configurable for both the Arduino and the PSoC hackAIR nodes. As default sampling rate values it was decided for the Arduino hackAIR node to transmit one measurement per hour (such a node is designed to be placed at a specific place – limited portability), while the PSoC one measurement per 10 minutes (such a node is designed to be portable). This is applied in order to ensure the maximum lifetime of the sensors, and the user can easily change these time intervals in the software. According to the above proposed frequencies, the average life expectancy of the proposed hackAIR nodes is much longer than 3 years.

3 Arduino Node

Summary

The present chapter describes the implementation details of one of the open hardware solutions that will be used for data acquisition in the hackAIR platform. The proposed hardware solution is implemented using an Arduino device, an addon board for networking (WiFi or Ethernet) and an Air Quality sensor.

An Arduino node consists of the following:

- A user-programmable Arduino board (ex. Arduino Uno)
- An addon-board for internet connectivity
- Arduino Ethernet Shield v2
- hackAIR WiFi shield
- An air-quality sensor

3.1 Project hosting and learning resources

The necessary development tools for the Arduino node are an Arduino/Genuino UNO board and the Arduino IDE (ver. 1.6 and above). The usage of Arduino IDE is widely spread so that the users who are not familiar with it, can find many quick-start guidelines. It is beyond the scope of this report to provide a tutorial for the use of Arduino IDE.

Before IDE's first run, the hackAIR library should be installed. Instructions are available on the hackAIR Arduino wiki (hackair-project.github.io/hackAir-Arduino/general/) so they can remain up-to-date with any library or IDE changes.

To stay up to date with updates to the Arduino platform and to be able to quickly deploy new versions with enhancements and bugfixes the project is hosted on Github, a community for open source software (and hardware development). All related source files and hardware designs are available in the project's two repositories:

- https://github.com/hackair-project/hackAir-Arduino
- https://github.com/hackair-project/hackAir-ArduinoWiFi

Additionally, the project's wiki page contains detailed information on how to create an Arduino node, multiple examples and information on each supported sensor. This should be the user's main guide since it constitutes a dynamically updated user friendly and open access tutorial.

- hackair-project.github.io/hackAir-Arduino/general/

3.2 Implementation

3.2.1 Assembly guide for the WiFi shield

Note: for up-to-date information the wiki should be consulted.

I. Identify the components

1. Resistors: 1kΩ, 10kΩ, 100kΩ. The body of the resistor might not be blue, that is normal.
2. hackAIR WiFi shield PCB
3. ESP8266 module
4. ZVN4206A MOSFET
5. Electrolytic capacitors (2x 10uF @ 16V). They might be different colour; that is normal.
6. Screw terminals. You can create the four-connection one by combining 2 two-connection terminals
7. Standard 2.54mm pin header

8. 2x Capacitors 10nF. They might be different colour/shape.
9. 2x Capacitors 100nF. They might be different colour/shape.

II. Solder the level shifter IC

Solder U1 by putting it flat on the board. Be careful that the notch on the IC should match the mark on the PCB (where the U1 designation is printed).

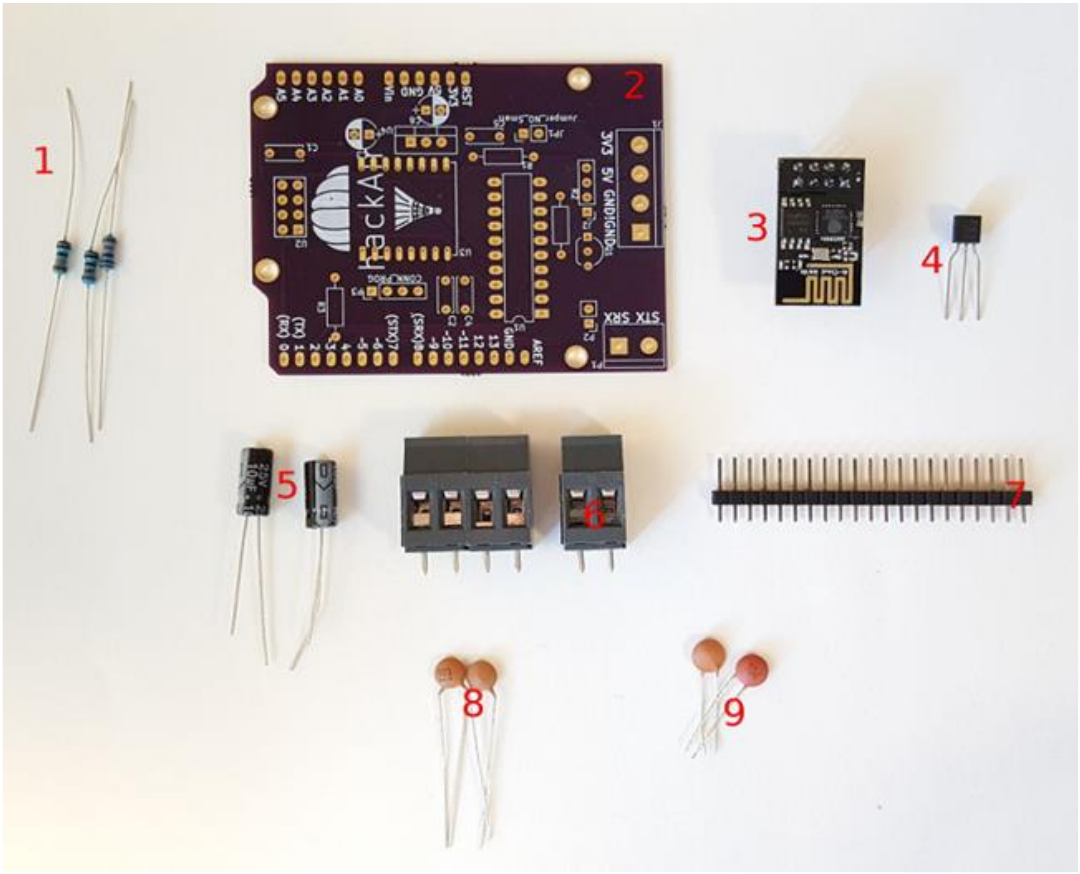

Figure 3 Necessary components for the WiFi shield

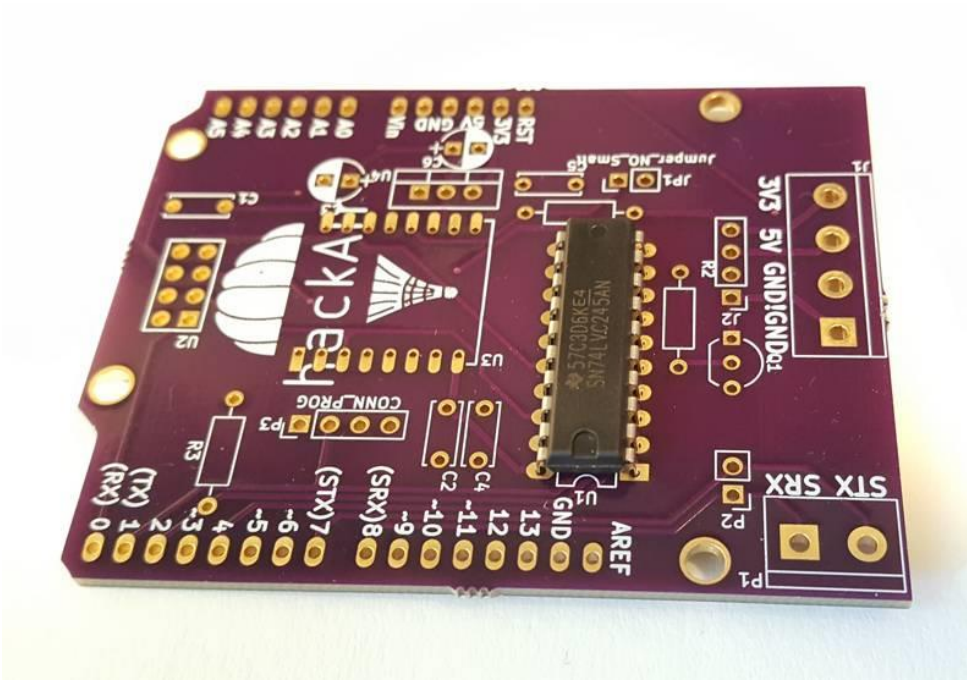

Figure 4 Solder U1 as seen in the picture above

III. Solder the resistors

Insert the resistors as following:

- R1: 1kΩ
- R2: 100kΩ
- R3: 10kΩ

You can bend the leads on the underside to make soldering easier as shown in the picture. If you are having trouble with parts staying in place paper tape is a good way to temporarily hold them in place.

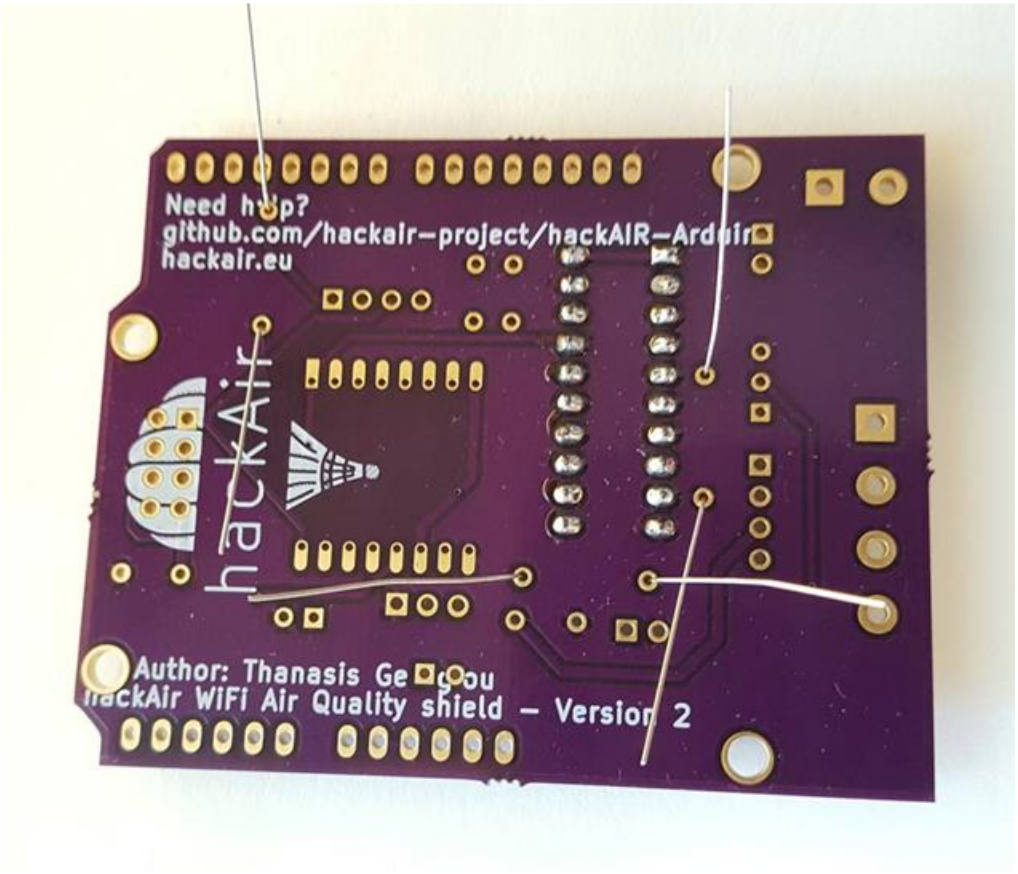

Figure 5 Bend the leads, in order to keep the resistors temporaliry in place.

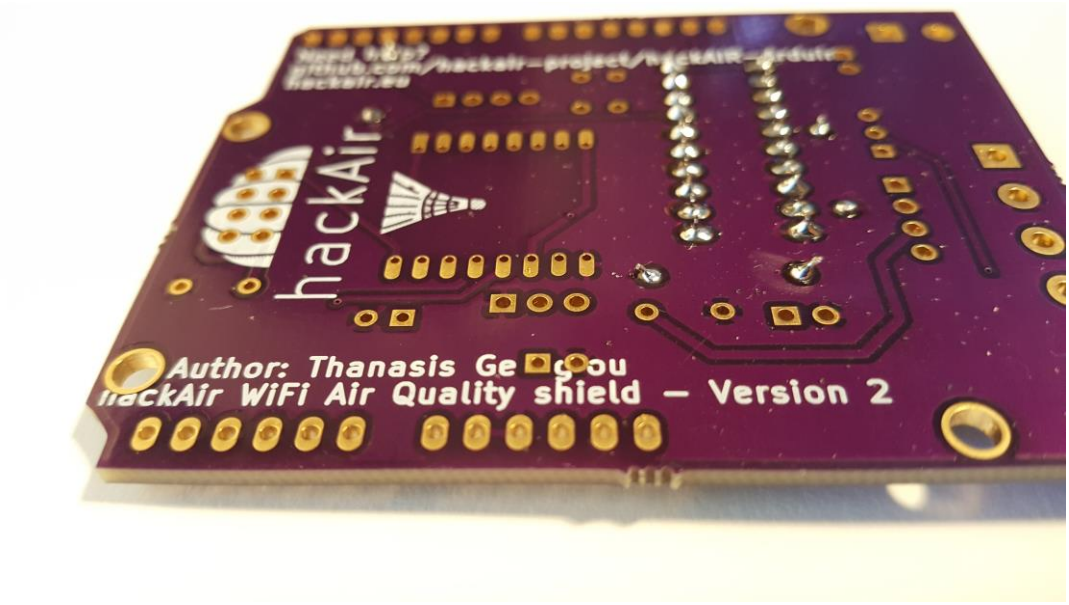

Figure 6 Once the resistors have been soldered, the result should be as pictured above

After soldering the long leads should be cut short. Be careful not to cut directly on the blob, just slightly higher.

IV. Soldering the capacitors

Some of the capacitors are electrolytic (the small towers) and must be soldered in a specific orientation. The long lead must go in the hole marked with the + sign, while the coloured band on the capacitor must be matched with the white mark on the PCB.

The small capacitors can be soldered either way since they are not polarised

- C1: 10nF (ceramic)
- C2: 100nF (ceramic)
- C3: 10uF (electrolytic)
- C4: 10nF (ceramic)
- C5: 100nF (ceramic)
- C6: 10uF (electrolytic)

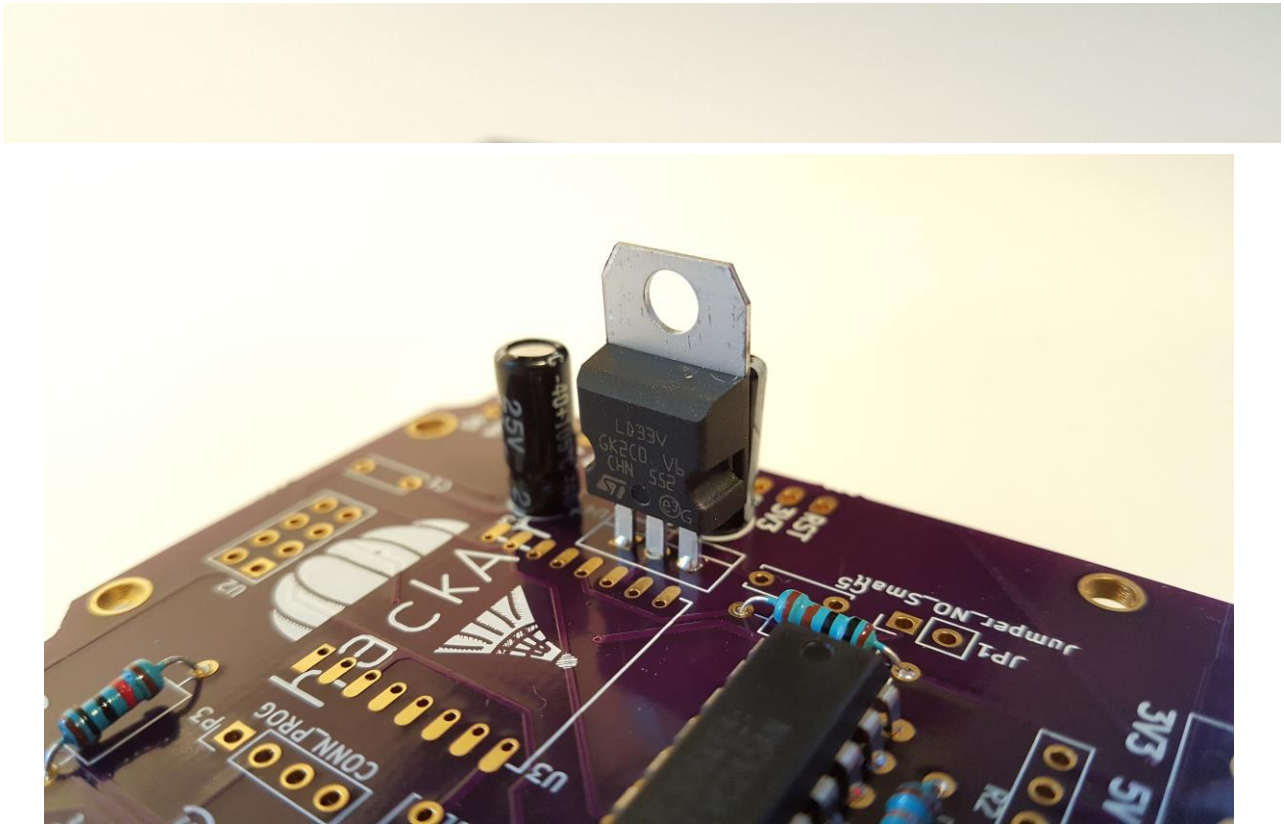

Figure 8 Make sure the metal surface is oriented as seen above

V. Solder the voltage regulator

Be careful when soldering the regulator, the metal heatsink must be on the side of the white markings on the board.

VI. Solder the ESP Module

There are two footprints for soldering either the ESP-01 or the ESP-12 module. You should use only one of those two. We recommend the ESP-01 (pictured) because it's easier to handle for beginners. The antenna should be placed outside of the board.

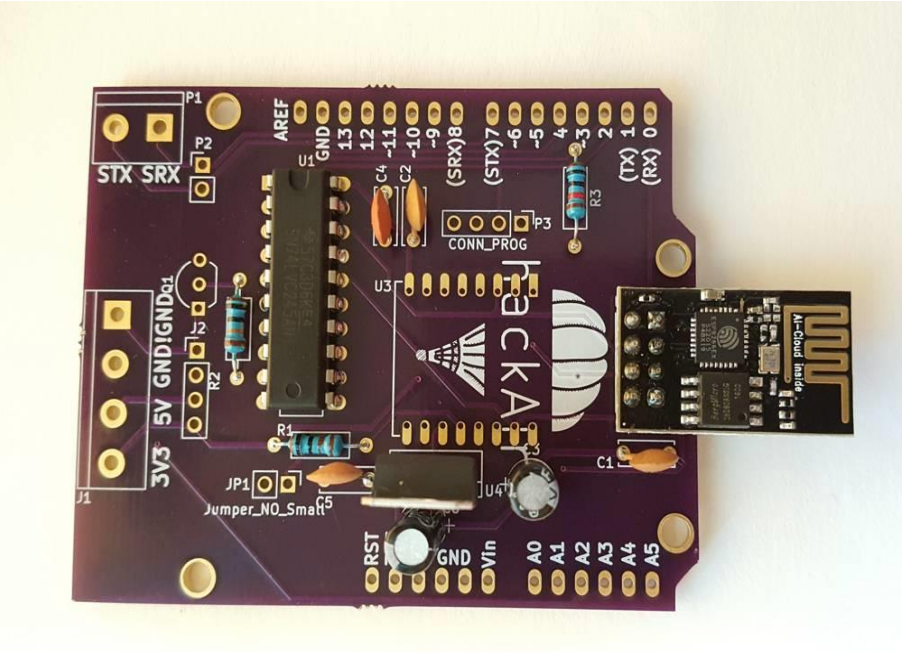

Figure 9 Make sure the antenna is placed outside of the board.

VII. Finish by soldering the connections

We left the connectors for the end because they are bulky and limit the stability of the board when trying to solder the rest of the components later. Great attention should be paid when soldering the pin headers that connect to the Arduino because if they are not aligned correctly, it might make inserting the shield difficult. Paper tape can be used to hold the components on the board.

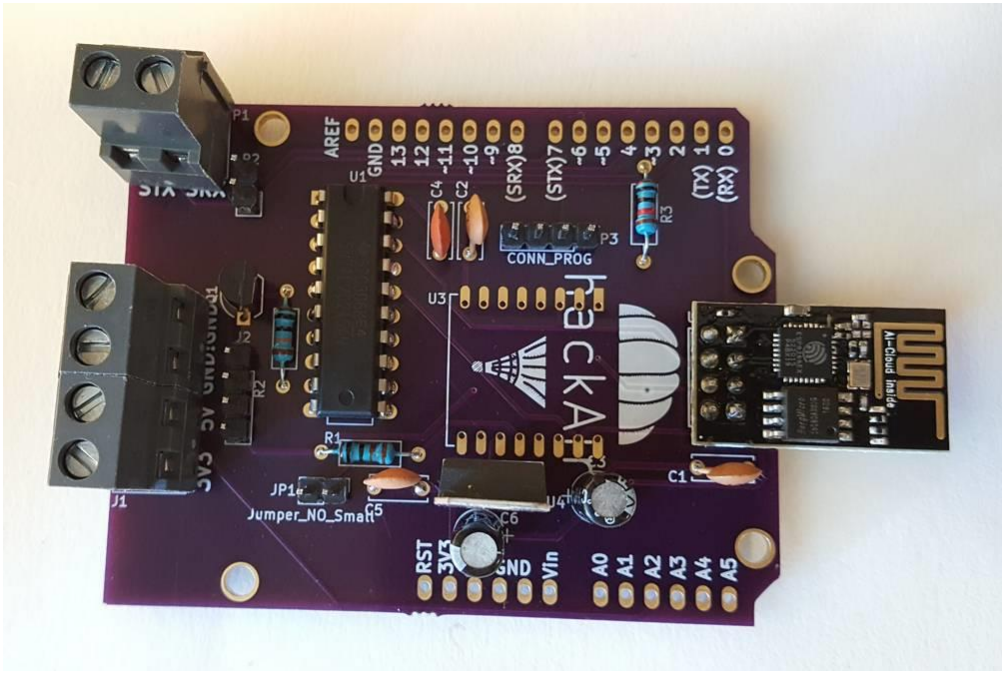

Figure 10 Not pictured: the Arduino shield pins. The female side should go on the top of the PCB, where the pin numbers (0, 1, 2…) are printed.

3.3 Software

The Arduino hackAIR software support is split in two distinct subprojects:

a. The hackAIR library, responsible for sensor support (includes Ethernet support)
b. The hackAIR WiFi library and firmware for the hackAIR WiFi shield

When the user installs the library, both subprojects become available for use. It's possible to use any part without the other, for example use the WiFi shield with an unsupported sensor (without the hackAIR library) or measure using a supported sensor without internet connectivity.

3.3.1 Arduino-hackAIR (Sensor support & Ethernet)

The core Arduino library handles the measurement of data so the users don't have to research the inner workings of each sensor. A library example for sending data to a computer through USB follows (Serial-PlainText.ino).

```
/**
 * @file Serial Sensor example
 * This example reads data from a sensor (default choice is the serial SDS011)
 * and sends it to a computer using USB serial communications.
 *
 * This example is part of the hackAIR Arduino Library and is available
 * in the Public Domain.
 */

#include "hackair.h"

// Specify your sensor
hackAIR sensor(SENSOR_SDS011);

void setup() {
  // Initialize the sensor
  sensor.begin();

  // Open serial port
  Serial.begin(9600);
}

void loop() {
  // Take a reading
  struct hackAirData data;
  sensor.refresh(data);

  // If it was invalid, print error
```

```
  if (data.error != 0) {
    Serial.println("Error!");
  } else {
    // Print the values to serial
    Serial.print("PM2.5: ");
    Serial.println(data.pm25);
    Serial.print("PM10: ");
    Serial.println(data.pm10);
    Serial.println("---");
  }

  // Wait a bit so the data is readable
  delay(2500);
}
```

As seen, it's very simple to read data from the sensor (single line of code) and it requires no complicated code from the user. Additional examples are included for printing data on an LCD or sending to the hackAIR servers using Ethernet/WiFi.

It is deemed out-of-scope to include more source code in this document since it is possible that it will be out-of-date very soon due to the frequent updates to our codebase. The Github repository along with the Wiki should be the primary source for source code and documentation.

3.3.2 HackAIR WiFi shield

The ESP8266 is a powerful SoC which is capable of handling both sensor measurements and internet connectivity. It is, however, hard for beginners to program the ESP module due to the lack of simplicity offered by something like the Arduino IDE. For this reason we opted to use the ESP module only for connectivity and let the Arduino handle everything related to the sensors. This means that for WiFi to work there are two codebases:

a. Data handling and connectivity firmware for the ESP module
b. Thin abstraction layer for communicating between the two systems for the Arduino

The user interacts only with a very small layer that just makes it easy to communicate to the hackAIR shield. For a simple measurement without averaging, only three lines of code are required (excluding setup).

```
struct hackAirData data;
sensor.refresh(data);
wifi_sendData(data);
```

The ESP Module firmware is based on the NodeMCU platform. NodeMCU is a Lua interpreter for the ESP platform that lets the user avoid writing complicated C code when programming the ESP. We chose this platform to make the code easier to understand for any potential users that want more control over their WiFi shield.

When the shield is powered on for the first time it will enter Access Point (AP) mode. At this state the user can connect to it using their smartphone and they will be prompted to enter the SSID and passphrase for the WiFi network that the node should use. After the setup is complete, the node will switch over to station mode and send data when instructed by the Arduino. The user can reset the saved password through the Arduino. Additionally, the WiFi shield can be turned off when unused so battery life is maximized.

Flashing the hardware to the WiFi shield is a onetime process involving a USB-to-Serial adapter. These low-cost adapters are easy to use and are used to create a communication path between the user's computer and the shield to upload the firmware. After the first flash, the shield is capable of downloading its own firmware updates from the internet.

3.3.3 Creating a WiFi node (case study: DFRobot sensor)

To create a WiFi node you require:

- An Arduino Uno
- The hackAIR WiFi shield
- Any compatible air quality sensor (DFRobot is used in this example)
- Female-to-Male jumper wires
- The Arduino IDE with the hackAIR library installed

The WiFi shield should be stacked on top of the Arduino Uno, making sure all pins are aligned and inserted correctly. The sensor should then be connected to its daughterboard and the daughterboard to the Arduino following the table below.

Table 3 Correct pairing of pins between the sensor and the WiFi Shield

Sensor Pin	Description	WiFi Shield Pin
GND	Power Ground	GND
VCC	Power	5V
RST	Reset	-
Rx	Receive data	STX
Tx	Transmit data	SRX

Notice that the Tx/Rx pairs are connected backwards. This is because the 'Transmit' pin of the sensor must be connected to the 'Receive' pin of the Arduino and vice versa.

After the connections are formed, you should use the Arduino IDE to upload the included 'WiFiSensor' example, also shown below. This sketch will take 60 readings (one, every second), average them and then, send the result to the hackAIR servers.

You must declare which sensor is used at line 13 and the hackAIR API token at line 25. You will receive the API token after registering at the hackAIR portal and register your sensing device. The hackAIR application will ask you to

provide details of your geolocation in order for the hackAIR PM estimation algorithms to be able to combine your measurements with other users' measurements and provide interpolated PM concentration information.

```
/**
 * @file WiFi Sensor example
 * This example reads data from a sensor (default choice is the serial SDS011)
 * and sends it to the hackAIR servers using the hackAIR WiFi Shield.
 *
 * This example is part of the hackAIR Arduino Library and is available
 * in the Public Domain.
 */

#include "hackair.h"
#include "hackair_wifi.h"

hackAIR sensor(SENSOR_SDS011);

void setup() {
  // Initialize the sensor
  sensor.begin();

  // Boot WiFi module
  wifi_begin();
  // Wait for WiFi connection
  // At this point you should use your mobile phone to setup the WiFi connection
  wifi_waitForReady();
  // Set authentication token
  wifi_setToken("REPLACE WITH AUTHENTICATION TOKEN");
}

void loop() {
  // Struct for storing data
  struct hackAirData data;
  sensor.refresh(data);

  // Average readings
  float pm25 = data.pm25;
  float pm10 = data.pm10;

  // We will take 60 averages
  for (int i = 0; i < 60; i++) {
    // Read from the sensor
    sensor.refresh(data);

    // If error is not zero something went wrong with this measurment
    // and we should not send it.
    if (data.error == 0) {
      // Calculate average between the new reading and the old average
      pm25 = (pm25 + data.pm25) / 2;
      pm10 = (pm10 + data.pm10) / 2;
    }

    delay(1000); // Wait one second
  }

  // Send data to the hackAIR server
  data.pm25 = pm25;
  data.pm10 = pm10;
  wifi_sendData(data);
}
```

3.4 Hardware Design

3.4.1 Arduino Uno

The requirements described in the previous section, can be easily fulfilled by the Arduino node. The Arduino Uno can be powered by a USB Power supply (phone charger) or a 9V wall adapter (higher voltages are supported, recommended max is 12V). The design is based on the assumption that a USB power supply will be used which means a maximum current draw of 500mA. The Arduino itself requires ~50mA of electric current so the rest is available for powering sensors.

The sensors can be attached directly on the Arduino pins. Each sensor, depending on its data interface, should possibly be connected on different pins. Detailed information about every supported sensor is available in the Wiki.

3.4.2 Ethernet

If the user opts for Ethernet connectivity the supported method is using the official Arduino Ethernet shield version 2. It's available for sale together with the Arduino Uno and is very easy to use. Third party compatible shields are available but not officially supported by our resources.

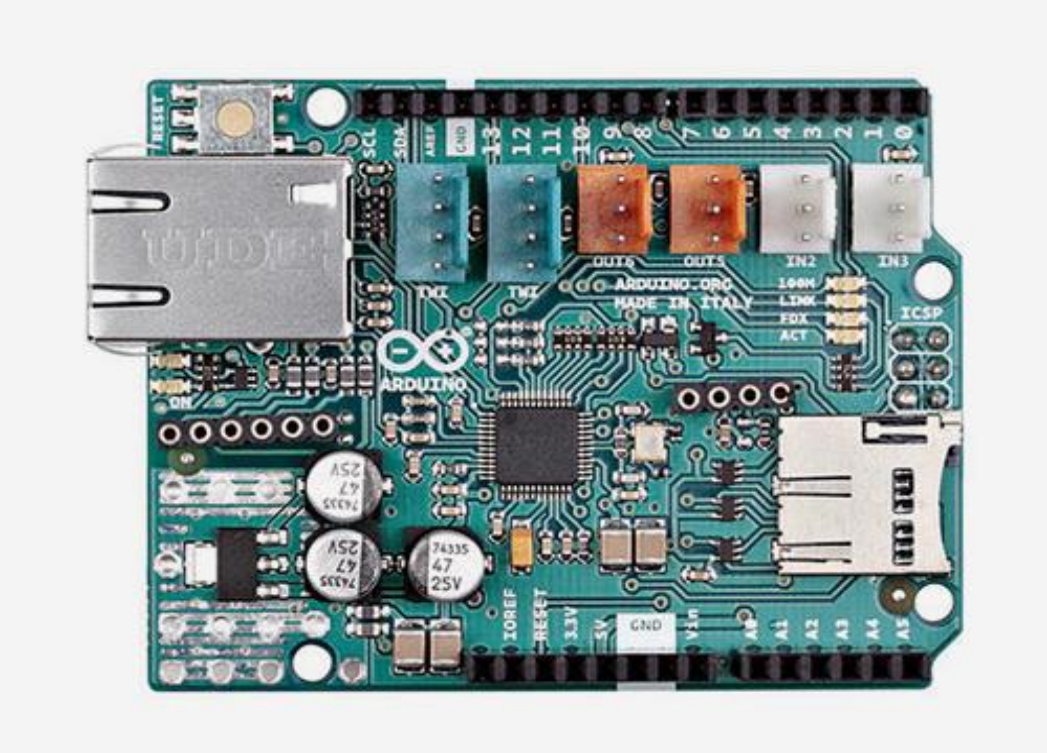

Figure 11: Arduino Ethernet Shield v.2

3.4.3 HackAIR WiFi shield

At the time of writing, all the available solutions regarding WiFi connectivity didn't fit the project's use case either due to high cost, low availability or complexity. To solve this problem, the hackAIR WiFi shield was designed to make using the ESP8266 easier for beginners.

The WiFi shield consists of an ESP8266 WiFi System-on-Chip and it's supporting circuitry in an Arduino shield board. Included on the board are screw terminals to connect sensors and a 3.3V regulator so supply power to the SoC. All the Arduino Pins are available with the exception of those used to communicate with the WiFi SoC.

Before using the shield, the user must first assemble it and flash the hackAIR firmware to the WiFi SoC. The wiki page contains information on how to use the flashing utilities (esptool, CuteESP) to upload the firmware.

The PCB design files are available so anyone can download, modify and manufacture the shield. Additionally a project is created at OSHPark, a PCB manufacturing services for hobbyist. The price is $30 for three PCB (including shipping). Unfortunately no other manufacturer seems to offer the convenience of OSHPark while selling single PCBs.

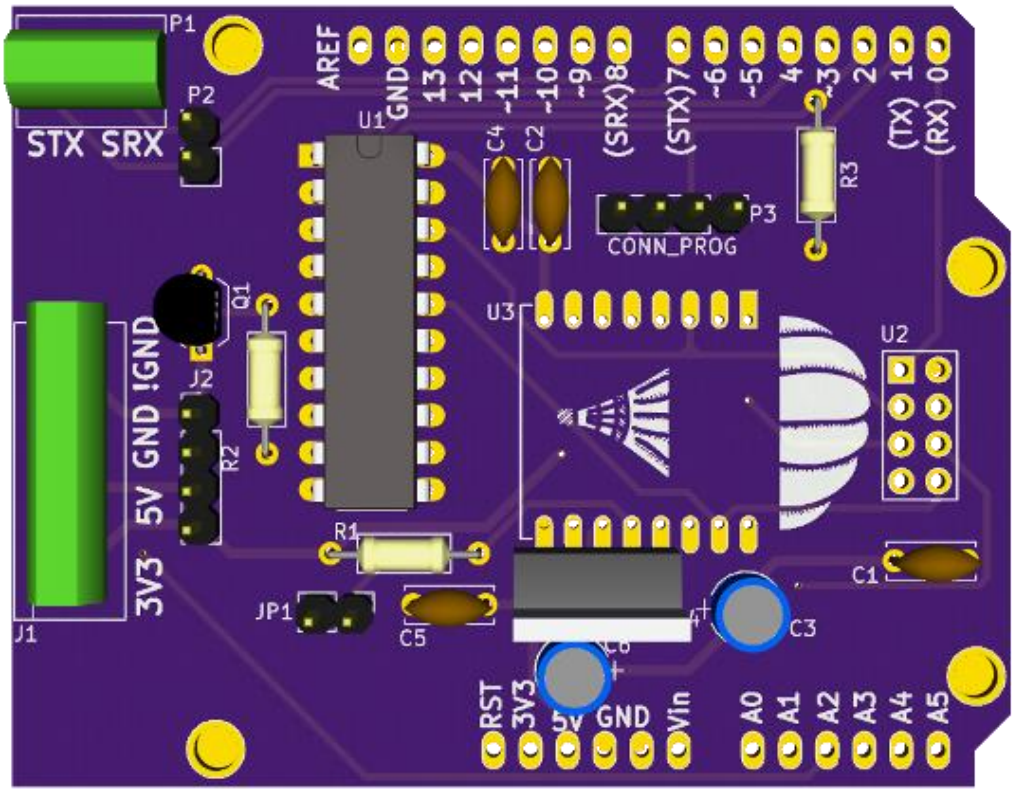

Figure 12: 3D Render of the hackAIR WiFi shield without the ESP module

Type	Name
Resistors	R1 (1kΩ), R2 (100kΩ), R3 (10kΩ)
Capacitors	C1 (10nF), C2 (100nF), C3 (10uF Electrolytic), C4 (10nF), C5 (100nF), C6 (10uF Electrolytic)
Voltage Regulator	U4 (LD1117S33)
WiFi SoC	ESP8266-01 or ESP8266-12, only one required
Level shifter IC	U1 (SN74LVC245), optionally use 20-pin DIP socket
Switching FET	Q1 (ZVN4206A)
Connectors	One set of Arduino Long headers, one strip of 2.54mm male pins, one 4-Screw terminal and one 2-Screw terminal

The total cost is around 30€, including one PCB. Detailed instructions on assembly and up-to-date schematics are available at the project's Wiki.

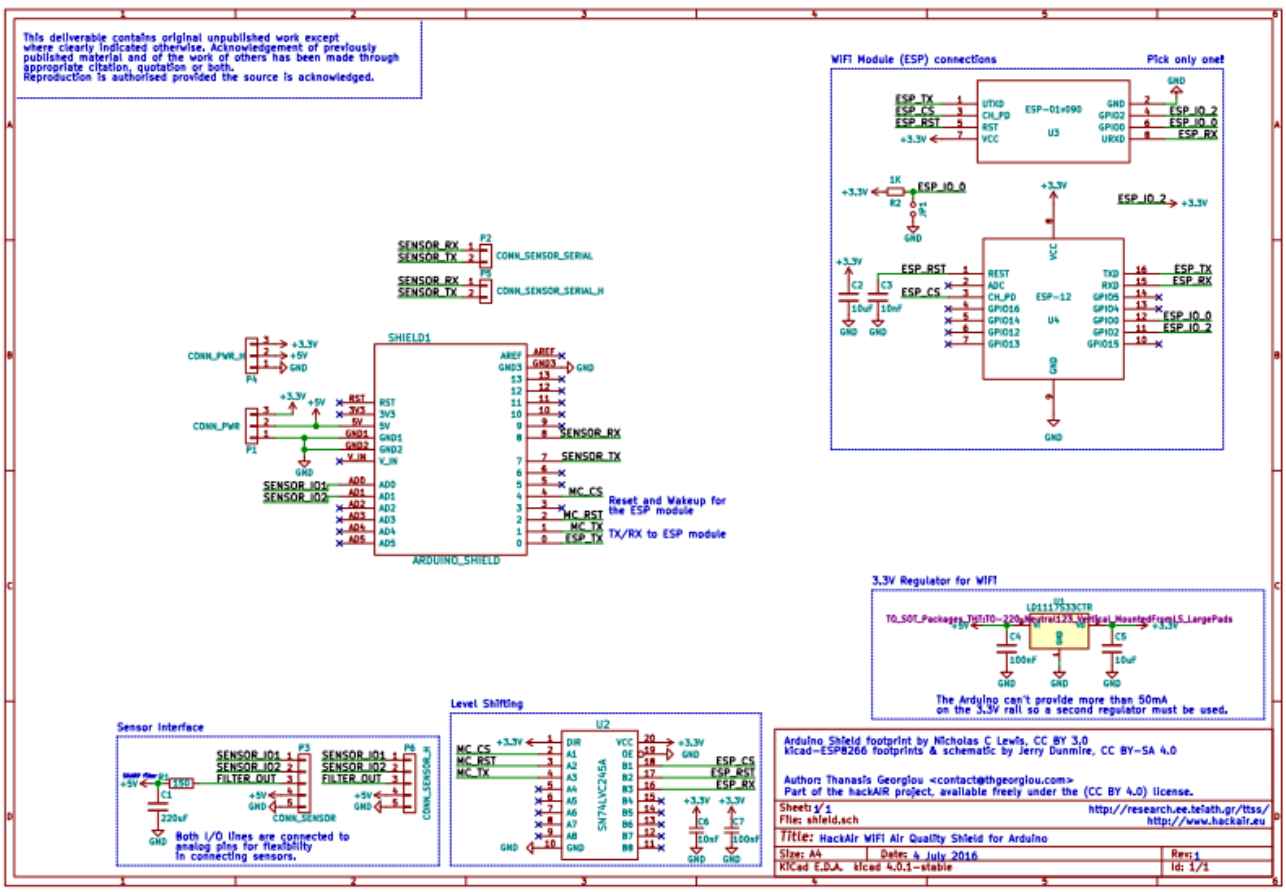

Figure 13. Schematic capture of the proposed hackAIR WiFi shield

4 BLE Node

Executive summary

The purpose of the current section is to present the elements and describe the procedures needed to successfully develop a hackAIR Bluetooth Low Energy based node. The proposed solution can be implemented using a PSoC/PRoC BLE module which will transmit the acquired values from sensor to a smart hone (equipped with hackAIR app). The required elements for the proposed implementation are:

- A PSoC/PRoC BLE module
- A compatible air quality sensor
- Mini UBS breakout
- Jumper cable
- Battery

Each hackAIR BLE node will be programmed by a low cost programmer (which described in the following sections). A set of precompiled binaries (one for every supported sensor) were created in order for the user to be able to flash the BLE module without further dependencies. In addition, and for experienced users that want to experiment further, we provide the code binaries totally unlocked in a form that can be used by free development tools for BLE modules. Each procedure in the current report is described in a step-by-step approach allowing the familiarization of unknown actions in a shortened period.

4.1 Project hosting and learning resources

To stay up to date with updates regarding the hackAIR BLE node and to be able to quickly deploy new versions with enhancements and bugfixes the project is hosted on Github, a community for open source software (and hardware development). All related source files and hardware designs are available in the project's repository:

- https://github.com/hackair-project/hackAIR-PSoC

Additionally, the project's wiki page contains detailed information on how to create a 4 BLE Node, multiple examples and information on each supported sensor. This should be the user's main guide.

- https://github.com/hackair-project/hackAIR-PSoC/wiki

4.2 Implementation

The following guidelines are provided in order to help the users implement the corresponding solution.

4.2.1 Suggested materials

The user will need the following components and materials to set up a BLEAir node:

- PSoC BLE Module (CY5671 or equivalent)
- KitProg programmer (included in CY8CKIT-059)
- Dust density sensor (One of three supported sensors: SEN0177, SDS011 and PPD42)
- Sensor connector or cable harness (provided by vendor)
- A USB power bank (at least 3000mAh) for power supply
- USB Module (micro USB breakout)
- Jumper Wires (Male/Male jumper wires and Male/Female jumper wires)
- Pin Header (5-pin 2.54mm male header)

After gathering the above follow the next steps to assemble your node.

4.2.2 STEP 1: Programming the PSoC 4/PRoC BLE Module using KitProg programmer

Before connecting the Bluetooth module to the sensor, it has to be programmed with the proper algorithm using a USB Programmer and Cypress's PSoC Programmer software which can be downloaded from their website. To choose the proper code file for the sensor, users should follow these guidelines:

- Use SerialLaserSensor_SEN0177.hex for SEN0177
- Use SerialLaserSensor_SDS011.hex for SDS011
- Use PwmLedSensor_PPD42.hex for PPD42

Once the proper file is chosen, users should proceed to the following steps to program the module:

1. **After receiving your PSoC 5LP Prototyping Kit proceed to detach the KitProg module by breaking at the area indicated in figure 14.**

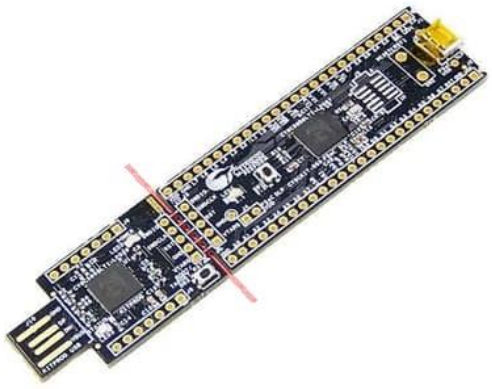

Figure 14 KitProg separation from PSoC 5LP Prototyping Kit

2. Connect KitProg through the 5 pin header marked at the bottom left of the PSoC BLE module (check for matching pin names). KitProg example is shown in figure 15

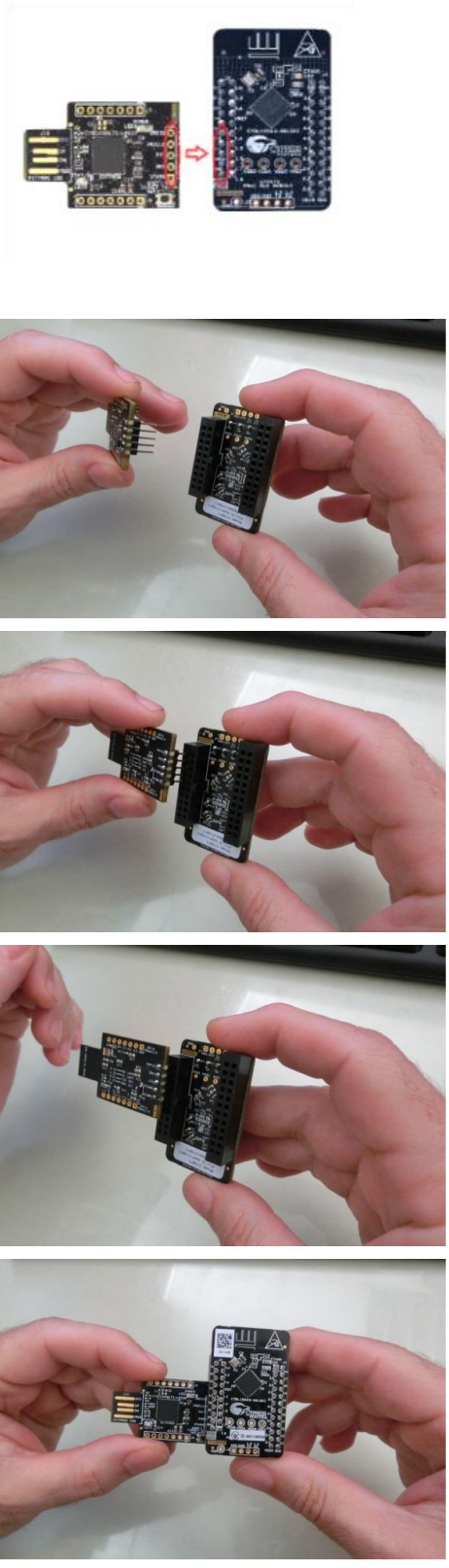

Figure 15 Connecting KitProg to BLE module for programming

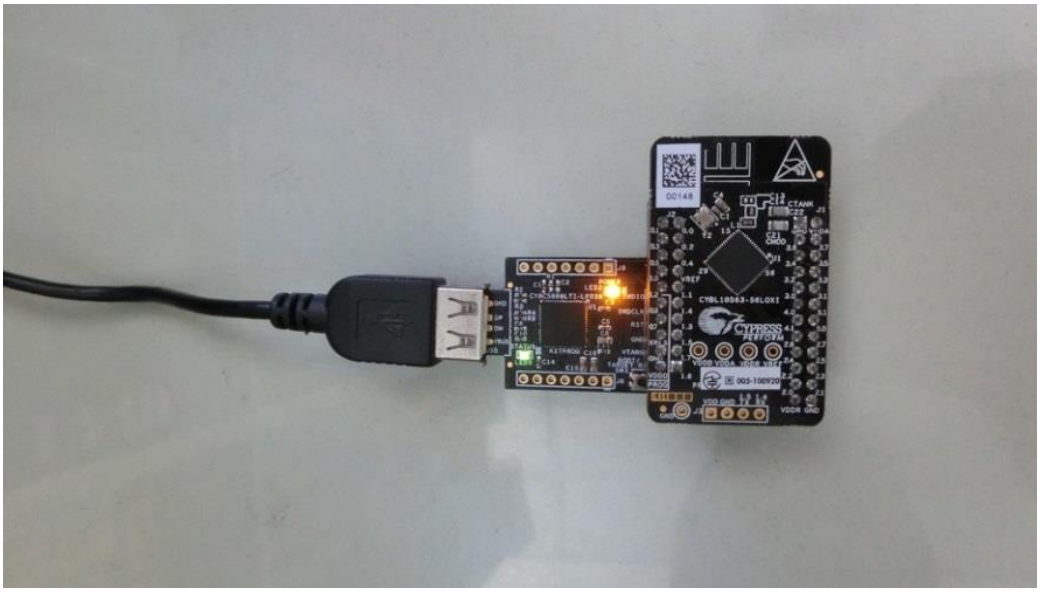

Figure 16 KitProg connection to USB port

3. Connect programmer to a USB 2.0 (or above) port (Figure 16)

4. Launch PSoC Programmer software

5. Open the proper .hex file (Figure 17)

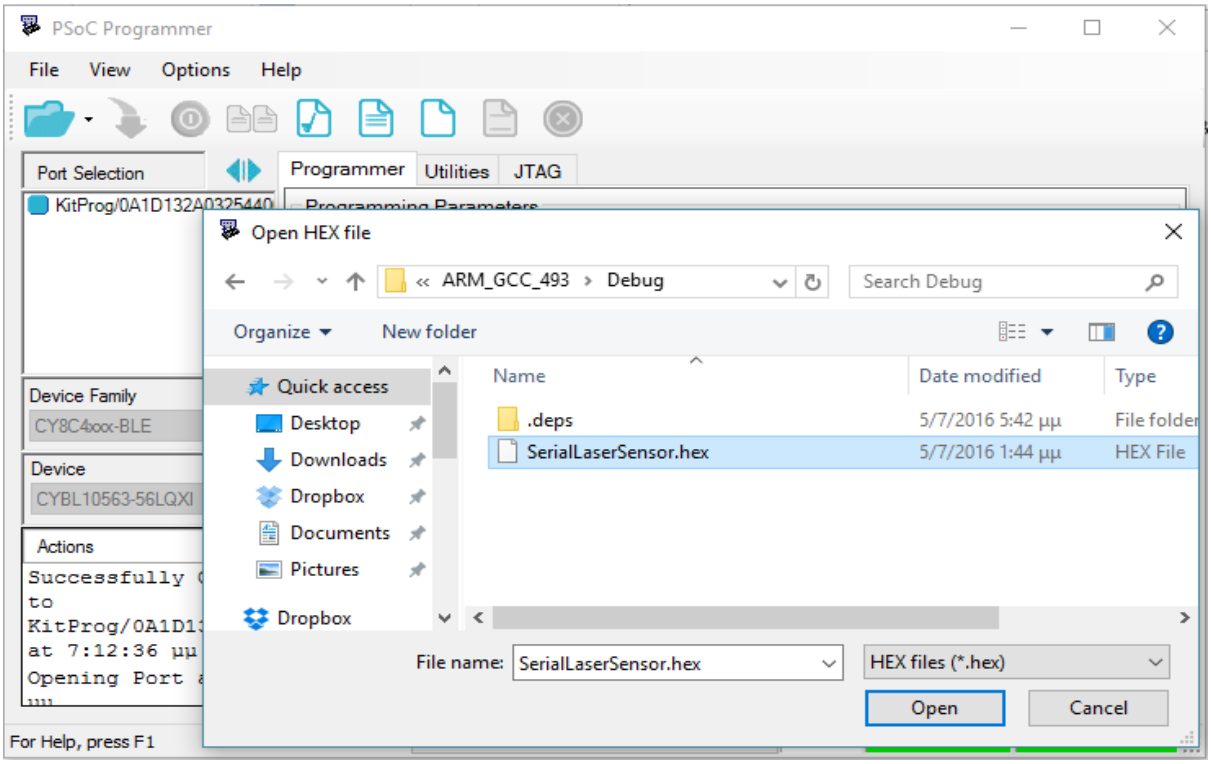

Figure 17 Opening a .hex file in PSoC Programmer

6. Select the device and click 'Program' (Figure 18)

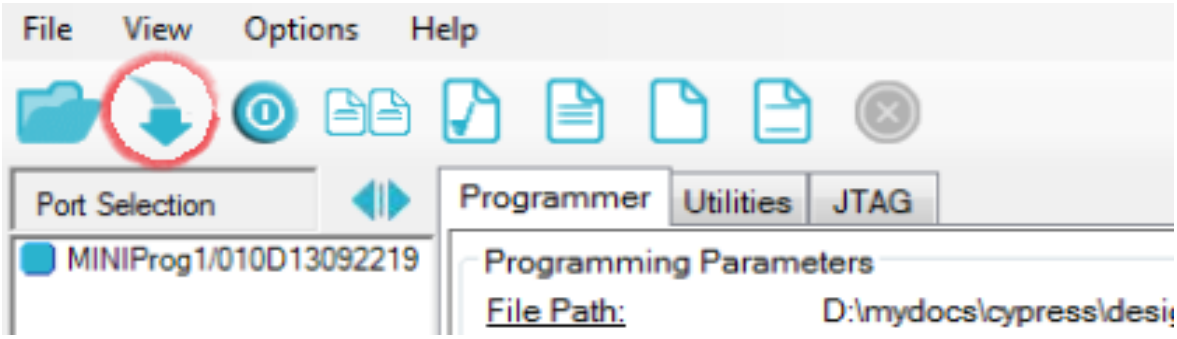

Figure 18 Starting the programming process

4.2.3 STEP 2: Soldering the USB Module

Use two M/M jumper wires to solder the micro-USB module to the PSoC BLE module as seen in Figure 19

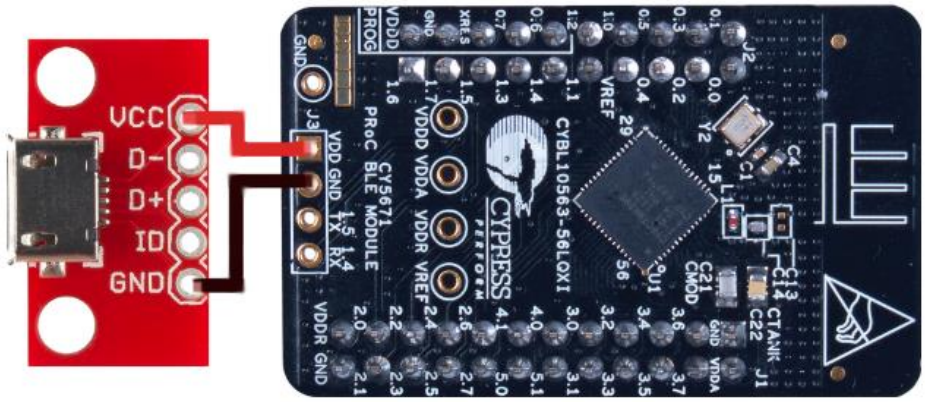

Figure 19. Soldering the USB module

4.2.4 STEP 3: Sensor Connection

For each type of sensor, users perform the following steps:

I. For SEN0177 connect (as shown in Figure 20):

- TX ➔ PIN 1.4
- VCC ➔ VDDD
- GND ➔ GND

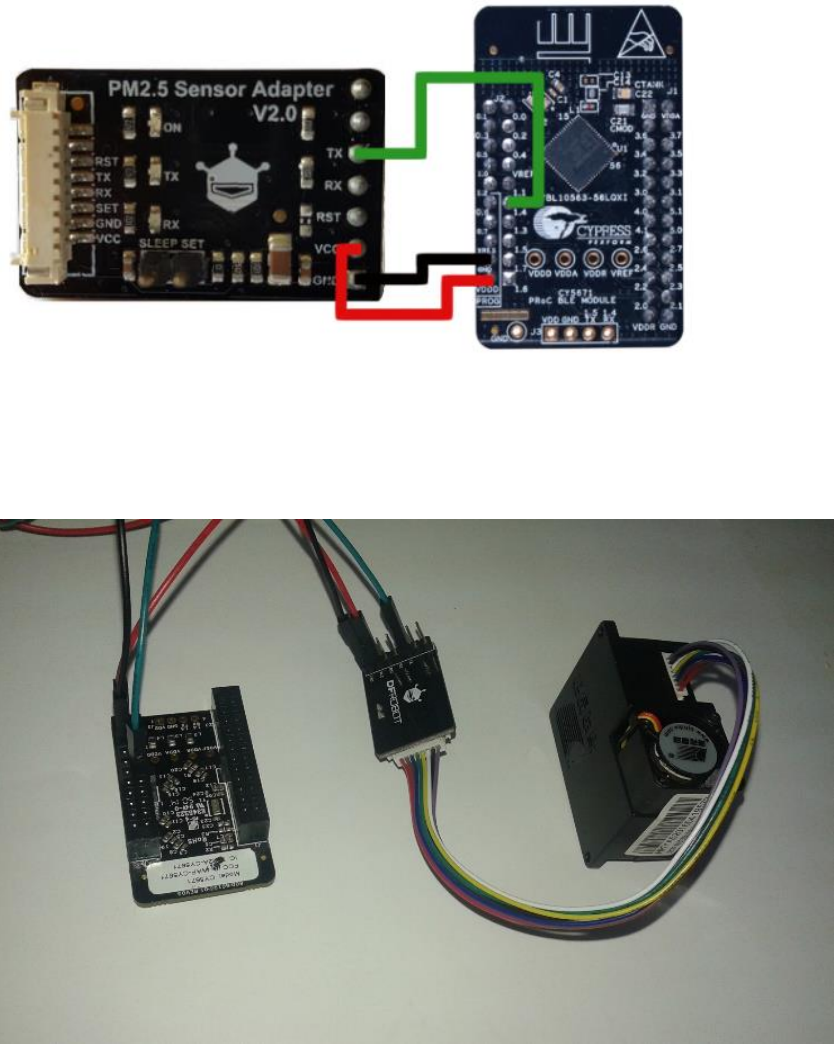

Figure 20 Connecting SEN0177

II. For SDS011 connect (as shown in Figure 21)::

- TXD ➔ PIN 1.4
- 5V ➔ VDDD
- GND ➔ GND

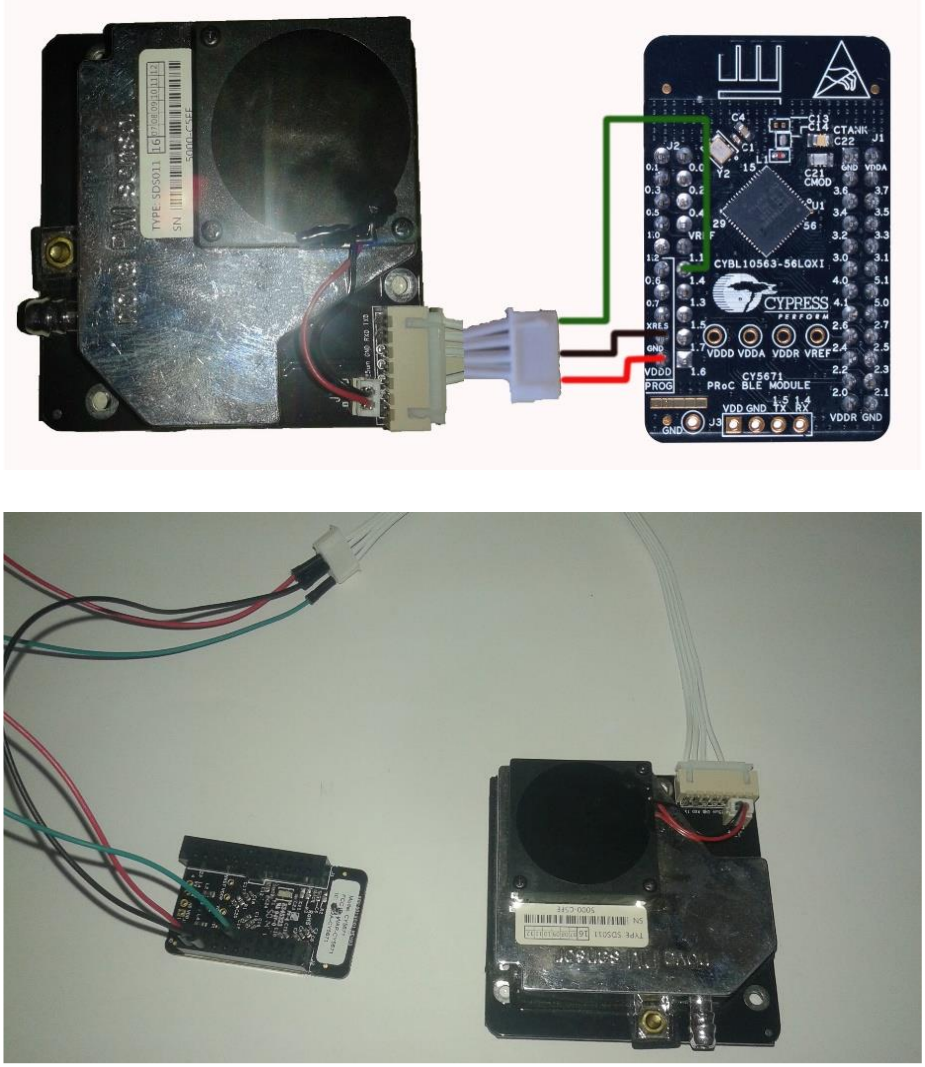

Figure 21 Connecting SDS011

III. For PPD42 connect (as shown in Figure 22):

- PIN 1 ➔ GND
- PIN 3 ➔ VDDD
- PIN 4 ➔ PIN 1.6

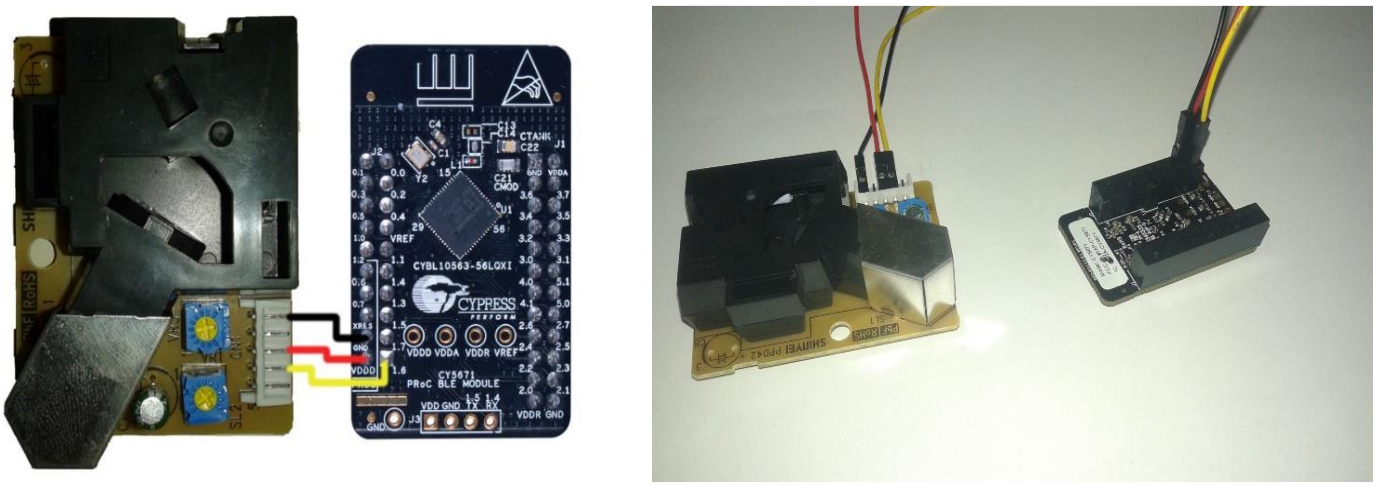

Figure 22 Connecting PPD42

4.2.5 STEP 4: Powering the device

Power the device using a power bank or any micro-USB charger

4.2.6 Customize the PSoC Creator Project (Optional)

a) Install PSoC Creator (available on Cypress's website) and run on any Windows computer.

b) Go to the PSoC Creator Project folder of the sensor you are using and double click on the .cyprj file.

c) Perform any desired changes (Learn how to use PSoC Creator to design BLE applications using the tutorials on Cypress's website)

d) Click the program button after connecting your module

4.3 Software

The software for BLEAir node comes in two forms:

a) Ready-made project files for using with PSoC creator
b) Precompiled. hex files for using with PSoC programmer

Ready-made project files are complete packages including all the elements (main routines, libraries and support files) needed by users that will use the PSoC Creator software. This approach ensures that the PSoC solution, it is fully customizable and provides the ultimate configuration solution but it is suggested for more experienced users. In the initial stage the users can tweak the readymade source files for each type of sensor as presented below (comments are in green).

4.3.1 MAIN.C routine for SEN0177

```
/* ========================================
 * Air Quality Beacon, part of the hackAIR project.
 * Freely available under CC BY 4.0
 *
 *
 * Author: George Doxastakis gdoxastakis.ee@gmail.com
 * Technology Transfer & Smart Solutions
 * TEI of Athens
 * http://research.ee.teiath.gr/ttss/
 *
 * http://www.hackair.eu/
*/
#include <project.h>

/* Function prototypes */
void StackEventHandler(uint32 event, void* eventParam);
char* readParticles();

/* ADV payload dta structure */
extern CYBLE_GAPP_DISC_MODE_INFO_T cyBle_discoveryModeInfo;
#define advPayload    (cyBle_discoveryModeInfo.advData->advData)


int main()
{
    CyGlobalIntEnable; /* Enable global interrupts. */
    Serial_Start();
    /* Start CYBLE component and register the generic event handler */
    CyBle_Start(StackEventHandler);

    for(;;)
    {
        STATUS_Write(1);CyDelay(300);STATUS_Write(0);
        char *val=readParticles(); //Perform sensor measurement (waits for result)

        /* Dynamic payload will be continuously updated */
        advPayload[19] =  0x01; //Sensor ID: 0x01 for SEN0177
        advPayload[21] =  val[4]; //High byte of PM1 value
        advPayload[22] =  val[5]; //Low byte of PM1 value
        advPayload[23] =  val[6]; //High byte of PM2.5 value
        advPayload[24] =  val[7]; //Low byte of PM2.5 value
        advPayload[25] =  val[8]; //High byte of PM10 value
        advPayload[26] =  val[9]; //Low byte of PM10 value
```

```
        CyBle_GapUpdateAdvData(cyBle_discoveryModeInfo.advData,
cyBle_discoveryModeInfo.scanRspData); //Update Advertisment Packet
        CyBle_ProcessEvents();
    }
}

void StackEventHandler(uint32 event, void *eventParam)
{
    switch(event)
    {

            case CYBLE_EVT_STACK_ON:
                  /* BLE stack is on. Start BLE advertisement */
                  CyBle_GappStartAdvertisement(CYBLE_ADVERTISING_FAST);
                break;

        default:
            break;
    }

}

/*******************************************************************
* NAME :            char* readParticles()
*
* DESCRIPTION :     Read sensor measurement
* OUTPUTS :
*       char* Sensor TX packet
*/
char* readParticles(){
    int idx;
    static char senData[32];
    senData[0]=0x42;
    while(Serial_UartGetChar()!= 0x42); // Wait for start character
    for(idx=1;idx<32;idx++){ // Read TX packet
        CyDelay(10); // Wait for data
        senData[idx]=Serial_UartGetChar();// Get next byte
    }
    return senData;
}

/* [] END OF FILE */
```

4.3.2 MAIN.C routine for SDS011

```
/* ========================================
 * Air Quality Beacon, part of the hackAIR project.
 * Freely available under CC BY 4.0
 *
 *
 * Author: George Doxastakis gdoxastakis.ee@gmail.com
 * Technology Transfer & Smart Solutions
 * TEI of Athens
 * http://research.ee.teiath.gr/ttss/
 *
 * http://www.hackair.eu/
*/
#include <project.h>

/* Function prototypes */
void StackEventHandler(uint32 event, void* eventParam);
char* readParticles();

/* ADV payload dta structure */
extern CYBLE_GAPP_DISC_MODE_INFO_T cyBle_discoveryModeInfo;
#define advPayload   (cyBle_discoveryModeInfo.advData->advData)


int main()
{
    CyGlobalIntEnable; /* Enable global interrupts. */
    Serial_Start();
    /* Start CYBLE component and register the generic event handler */
    CyBle_Start(StackEventHandler);

    for(;;)
    {
        STATUS_Write(1);CyDelay(300);STATUS_Write(0);
        char *val=readParticles(); //Perform sensor measurement (waits for result)
        uint16 pmsmall= (val[3]*256 + val[2])/10; //PM2.5 value
        uint16 pmlarge= (val[5]*256 + val[4])/10; //PM10 value
        /* Dynamic payload will be continuously updated */
        advPayload[19] =  0x02; //Sensor ID: 0x02 for SDS011
        advPayload[21] =  0x00; //High byte of PM1 value - Not Available
        advPayload[22] =  0x00; //Low byte of PM1 value - Not Available
        advPayload[23] =  pmsmall>>8; //High byte of PM2.5 value
        advPayload[24] =  pmsmall&0xFF; //Low byte of PM2.5 value
        advPayload[25] =  pmlarge>>8; //High byte of PM10 value
        advPayload[26] =  pmlarge&0xFF; //Low byte of PM10 value
        CyBle_GapUpdateAdvData(cyBle_discoveryModeInfo.advData,
cyBle_discoveryModeInfo.scanRspData); //Update Advertisment Packet
        CyBle_ProcessEvents();
    }
}

void StackEventHandler(uint32 event, void *eventParam)
{
    switch(event)
    {

            case CYBLE_EVT_STACK_ON:
                  /* BLE stack is on. Start BLE advertisement */
                  CyBle_GappStartAdvertisement(CYBLE_ADVERTISING_FAST);
                break;

        default:
            break;
```

```
    }

}

/*******************************************************************
* NAME :            char* readParticles()
*
* DESCRIPTION :     Read sensor measurement
* OUTPUTS :
*       char* Sensor TX packet
*/
char* readParticles(){
    int idx;
    static char senData[10];
    senData[0]=0xAA;
    while(Serial_UartGetChar()!= 0xAA); // Wait for start character
    for(idx=1;idx<10;idx++){ // Read TX packet
        CyDelay(10); // Wait for data
        senData[idx]=Serial_UartGetChar();// Get next byte
    }
    return senData;
}

/* [] END OF FILE */
```

4.3.3 MAIN.C routine for PPD42

```
/* ========================================
 * Air Quality Beacon, part of the hackAIR project.
 * Freely available under CC BY 4.0
 *
 *
 * Author: George Doxastakis gdoxastakis.ee@gmail.com
 * Technology Transfer & Smart Solutions
 * TEI of Athens
 * http://research.ee.teiath.gr/ttss/
 *
 * http://www.hackair.eu/
*/
#include <project.h>
#include <math.h>
/* Function prototypes */
void StackEventHandler(uint32 event, void* eventParam);
int16 readParticles();

/* ADV payload dta structure */
extern CYBLE_GAPP_DISC_MODE_INFO_T cyBle_discoveryModeInfo;
#define advPayload   (cyBle_discoveryModeInfo.advData->advData)


int main()
{
    CyGlobalIntEnable; /* Enable global interrupts. */

    /* Start CYBLE component and register the generic event handler */
    CyBle_Start(StackEventHandler);
    for(;;)
    {
        STATUS_Write(1);CyDelay(300);STATUS_Write(0);
        int16 val=readParticles(); //Perform sensor measurement

        /* Dynamic payload will be continuously updated */
        advPayload[19] =  0x05; //Sensor ID: 0x05 for PPD42
        advPayload[23] =  val>>8; //High byte of sensor measurement
        advPayload[24] =  val&0xFF; //Low byte of sensor measurement
        CyBle_GapUpdateAdvData(cyBle_discoveryModeInfo.advData,
cyBle_discoveryModeInfo.scanRspData); //Update Advertisment Packet

        CyBle_ProcessEvents();

        CyDelay(500); //Measurement delay
    }
}

void StackEventHandler(uint32 event, void *eventParam)
{
    switch(event)
    {

            case CYBLE_EVT_STACK_ON:
                  /* BLE stack is on. Start BLE advertisement */
                  CyBle_GappStartAdvertisement(CYBLE_ADVERTISING_FAST);
                break;

        default:
            break;
    }
```

```
}

/*******************************************************************
* NAME :            int16 readParticles()
*
* DESCRIPTION :     Read sensor measurement
* OUTPUTS :
*       int16 Dust concentration in pcs/0.01cf
*/
int16 readParticles(){
    int i;
    int dur=0;

    for(i=0;i<2000000;i++){
      CyDelayUs(1);
      dur+=!PWM_IN_Read();//Count low pulse Duration
    }
    int ratio = dur/20000;
    uint16 senDat = (uint16)(1.1 * pow(ratio, 3.0) - 3.8 * pow(ratio, 2.0) + 520 *
ratio + 0.62); // Sensor transfer function

    return senDat;
}

/* [] END OF FILE */
```

All the corresponding files (project files as well as. hex files) will be available to users through the hackAIR platform.

5 COTS

Executive summary

The goal of this work is to research the optimal method for blob detection and estimation of microparticles' dimensions, which accumulate on test surfaces with petroleum products coating. Thus, an estimation of the particles' concentration in the atmosphere is possible, with the usage of Computer Vision. It is desirable that system users are able to assess air quality by using commercial off the shelf (COTS) materials.

Through the hackAIR COTS sensor we aim to create, a low cost, low tech air quality sensor, which will provide a reliable qualitative measurement, regarding the atmosphere's concentration of PM. The sensor comprises solely with commercially available off-the-shelf products. In order to determinate the effectiveness of the proposed algorithm for blob detection, a series of combined measurements between the hackAIR COTS sensor and a commercially available air quality sensor was conducted.

For the purpose of the experiment, the DC1100 Air Quality Monitor was selected which displays the concentration of particles in the air. More precisely particles in 0.01 cubic foot of air, whereas the hackAIR COTS sensor, detects the number of particles on the test surface after a layer of petroleum jelly is applied, through the usage of Computer Vision.

5.1 Project hosting and learning resources

To stay up to date with updates to the COTS node platform and to be able to quickly deploy new versions with enhancements and bugfixes the project is hosted on Github, a community for open source software (and hardware development). All related source files and hardware designs are available in the project's repository:

- https://github.com/Electronicdevicesandmaterials/cots

Additionally, the project's wiki page contains detailed information on how to create an COTS node, multiple examples and information on each supported sensor. This should be the user's main guide.

- https://github.com/Electronicdevicesandmaterials/cots/wiki

5.2 Implementation

In this section, we present the step by step implementation for the hackAIR COTS sensor.

5.2.1 What you will need

1. A used carton of milk
2. One small jar of petroleum jelly
3. One Butter knife
4. Commonly Used Thread
5. Mobile macro lens of x6 or greater magnification
6. Counter weight (batteries, paper clippers, coins etc.)

5.2.2 Where to buy

1. Local supermarket
2. Online

5.2.3 How to assemble

1. Get an empty carton of milk, clean it and let it dry.

Figure 23 Once the carton is dry, open it using a Stanley knife as pictured above.

2. Cut square pieces with dimensions 5cm x 5cm.

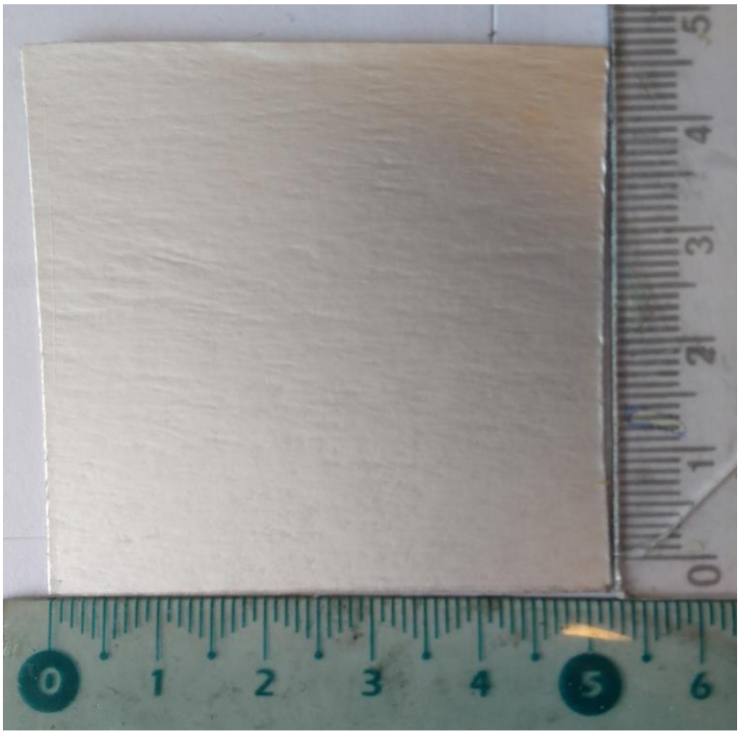

Figure 24 Using a Stanley knife cut square pieces

3. Along the diagonal axis of your choice draw two small dots with the tip of a pencil, near the centre of the test surface.

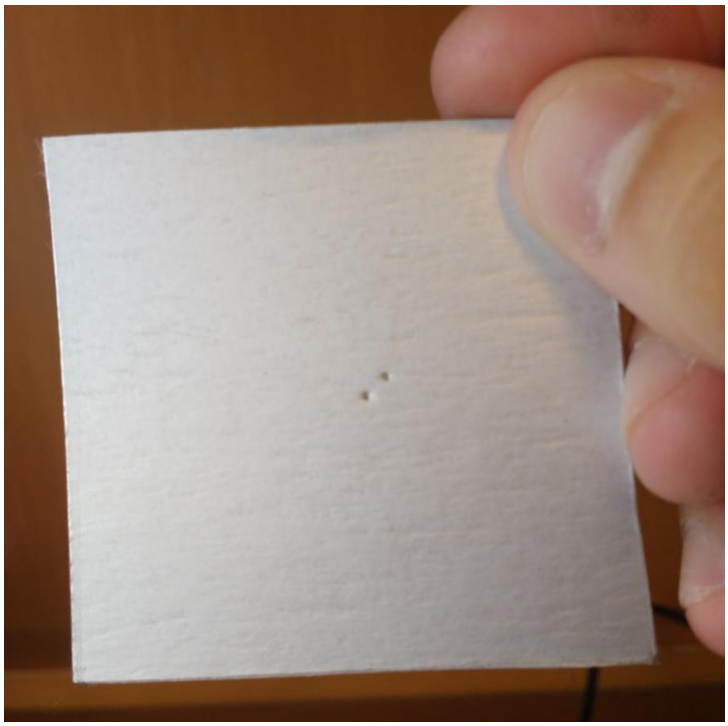

Figure 25 The small dots will serve as benchmark level for the blobs' dimensions.

4. Bend the surface slightly outward, along the same diagonal axis you chose earlier. This, will result to a constant capture of air particles, on the surface.

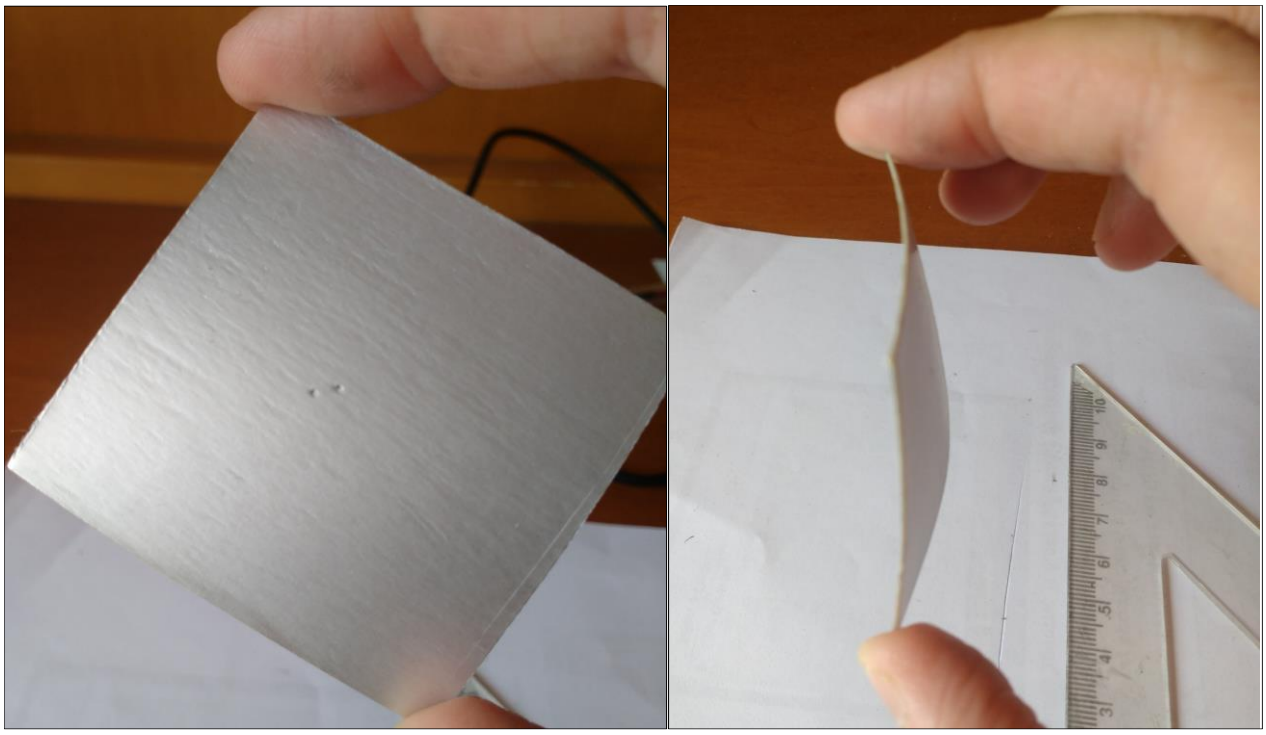

Figure 26 Bending the test surface, will allow it to turn along the general wind direction

5. Pierce the upper right corner and put a thread through the hole you just created it.

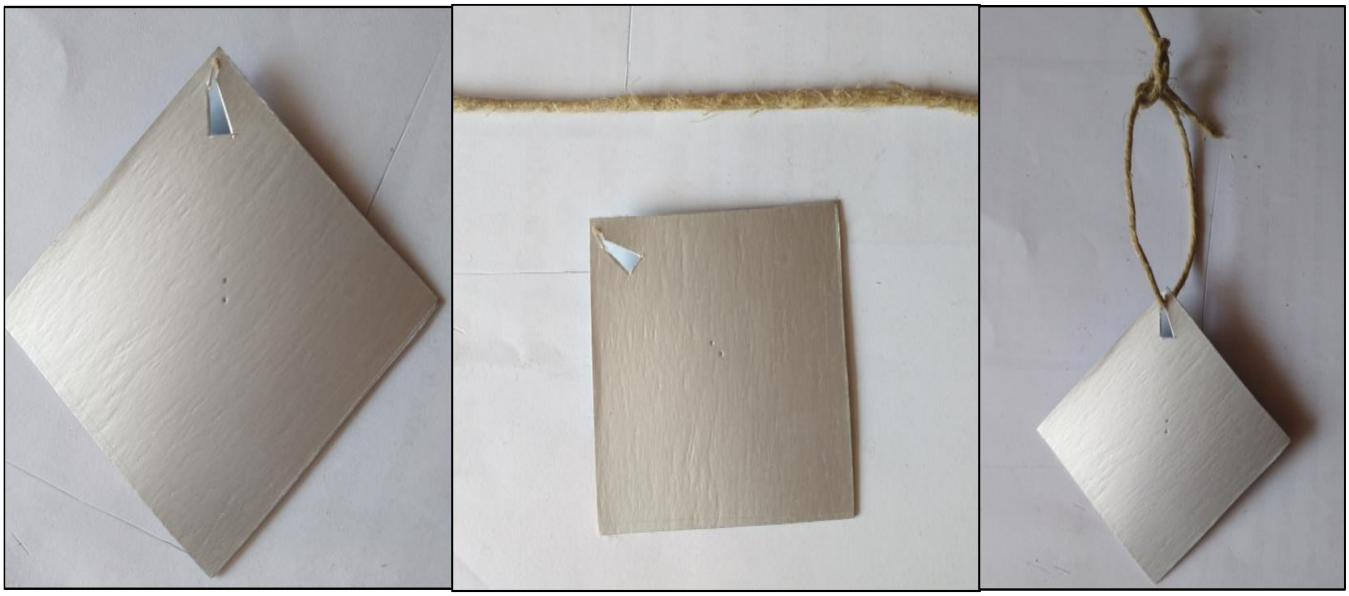

Figure 27 Make a hole using a Stanley knife or any other sharp tool, put the thread through the hole. Make a knot with the thread's loose ends.

6. At the lower left corner, adjust a counter weight (batteries, paper clippers, coins etc).

Figure 28 Using duct tape or any other mean, adjust a counter weight, as pictured above.

7. Apply a small amount of petroleum jelly on the test surface using the aforementioned butter knife. Before applying the petroleum jelly, make sure the test surface is clean and completely dry.

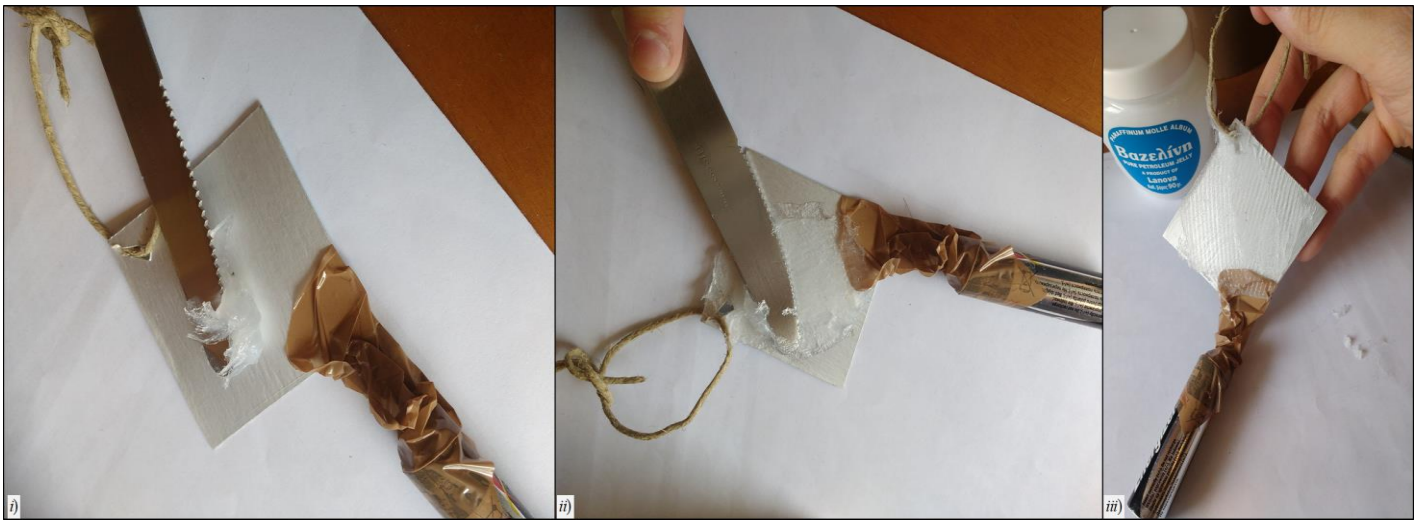

Figure 29 While applying the small amount of petroleum jelly, make sure you apply it as smoothly as possible.

5.2.4 Where should I place my sensor

Place the sensor at your balcony or outside your window. Make sure the test surface (the one with the petroleum jelly), is facing towards the general wind direction. For better results, it is recommended to keep the hackAIR COTS sensor dry, and the test surface clean before the application of petroleum jelly.

Figure 30: Final result of hackAIR COTS sensor

After 24 hours of constant exposure, retrieve the sensor. Place the aforementioned macro lens on top of the two small dots, you previously drew. Let your mobile adjust the focus of its camera and capture your image. Use flash if necessary. Upload the photo on the hackAIR app (there are instructions there).

Depending on your lens model the necessary distance between the lens and the test surface may differ. Make sure you read the manual of your lens model first.

Figure 31: Capturing the image for blob detection using super macro x6 lens

5.3 Software

5.3.1 Main algorithm code* in Matlab:

*free and available for anyone to use it and modify it.

```
% Code to simple blob detection, measurement, and filtering.

%% Startup code.
clear all
clc; % Clear command window.
clearvars; % Get rid of variables from prior run of this m-file.
close all;  % Close all imtool figures.
format long g;
format compact;
captionFontSize = 14;
warning('off')

%% variables
baseFileName = 'test12.jpg'; %name of the exammed picture
areaInMeters = 300*10^-6;
bright_pixels = 1; %Presentage_of_desirable_high_level_pixels [0-1]
%% find & read image
original_Image = imread(baseFileName);
original_Image = rgb2gray(original_Image);
original_Image = adapthisteq(original_Image);

% %% get histogram of the image
% [pixelCount, grayLevels] = imhist(original_Image);

%% Binarize image by thresholding using otsu's method

[threshold_Value, EM] = graythresh(original_Image);
thresh = multithresh(original_Image);
binaryImage=im2bw(original_Image,threshold_Value);
% binaryImage = imbinarize(original_Image,'adaptive')

binaryImage = imfill(binaryImage, 'holes');% Do a "hole fill" to get rid of
any background pixels or "holes" inside the blobs.

%% Blob detection

% Identify individual blobs by seeing which pixels are connected to each
other.
% Each group of connected pixels will be given a label, a number, to identify
it and distinguish it from the other blobs.
% Do connected components labeling with either bwlabel() or bwconncomp().

labeledImage = bwlabel(binaryImage, 8);      % Label each blob so we can make
measurements of it

% Get all the blob properties.
blobMeasurements    =    regionprops(labeledImage,    original_Image,    'Area',
'Centroid', 'MeanIntensity');

numberOfBlobs = size(blobMeasurements, 1);
```

```
boundaries = bwboundaries(binaryImage);

% Get the centroids of ALL the blobs into 2 arrays,
allBlobCentroids = [blobMeasurements.Centroid];
centroidsX = allBlobCentroids(1:2:end-1);
centroidsY = allBlobCentroids(2:2:end);

%% Blob Selection
allBlobIntensities = [blobMeasurements.MeanIntensity];
allBlobAreas = [blobMeasurements.Area];
areaInPixels = meterstopixels_cov(areaInMeters);
%% Calculating Threshold index using CDF function
cdf = cumsum(allBlobIntensities)/sum(allBlobIntensities);
Intensity_Threshhold_Index = find(cdf < bright_pixels, 1, 'last');
Intensity_Threshold_Value = allBlobIntensities(Intensity_Threshhold_Index);
%%
% Get a list of the blobs that meet our criteria and we need to keep.
allowableIntensityIndexes = (allBlobIntensities > Intensity_Threshold_Value);
allowableAreaIndexes = allBlobAreas <= areaInPixels;
keeperIndexes = find(allowableIntensityIndexes & allowableAreaIndexes);
% keeperIndexes = find(allowableAreaIndexes);

% Extract only those blobs that meet our criteria, and eliminate those blobs
that don't meet our criteria.
keeperBlobsImage = ismember(labeledImage, keeperIndexes); %Result will be an
image - the same as labeledImage but with only the blobs listed in
keeperIndexes in it.

% Re-label with only the keeper blobs kept.
labeledBlobsImage = bwlabel(keeperBlobsImage, 8);     % Label each blob so we
can make measurements of it

% %Keeper blobs properties
KeeperblobMeasurements = regionprops(keeperBlobsImage, original_Image, 'Area',
'Centroid', 'MeanIntensity');

KeeperBlobCentroids = [KeeperblobMeasurements.Centroid];
KeeperBlobArea = [KeeperblobMeasurements.Area];
KeeperBlobIntensities = [KeeperblobMeasurements.MeanIntensity];
KeepercentroidsX = KeeperBlobCentroids(1:2:end-1);
KeepercentroidsY = KeeperBlobCentroids(2:2:end);
numberOfKeeperBlobs = size(KeeperblobMeasurements, 1)
%% plots & figures


% Plot the centroids in the original image in the upper left.
imshow(original_Image);
% axis tight; % Make sure image is not artificially stretched because of
screen's aspect ratio.
title('Centroid and Boundaries of Blobs', 'FontSize', captionFontSize);

hold on; % Don't blow away image.

for k = 1 : numberOfKeeperBlobs           % Loop through all keeper blobs.
    KeeperBlob = allBlobAreas(k) <= areaInPixels;
    if KeeperBlob
        % Plot Blobs' Centroids with a red +.
        h(k)   =       plot(KeepercentroidsX(k),   KeepercentroidsY(k),   'r.',
'MarkerSize', 12, 'LineWidth', 2);
```

```
    end

end

hold on;

Keeperboundaries = bwboundaries(keeperBlobsImage);
KeepernumberOfBoundaries = size(Keeperboundaries, 1);

for k = 1 : KeepernumberOfBoundaries

    thisBoundary1 = Keeperboundaries[2, 8, 10-31];
    p(k)= plot(thisBoundary1(:,2), thisBoundary1(:,1), 'g', 'LineWidth', 2);

end

legend([h(end) p(end)],'Centroids of Blobs','Outlines of Blobs');
% legend('boxoff')
hold off
```

5.3.2 Supplementary code for converting meters to pixels.

```
function [ pix ] = meterstopixels_cov( mtr )

pix = 3779.527559055*mtr;

end
```

5.3.3 COTS Java library for Android

CLASS DESCRIPTION

A Java implementation of the original COTS blob counting algorithm for use in Android devices.

Dependencies

Use of the library requires the Open CV Library (version 3.0.0 or higher).

The following classes are imported:

import org.opencv.core.*;

import org.opencv.imgcodecs.Imgcodecs;

import org.opencv.imgproc.Imgproc;

Fields

static double	DEFAULT_PM_AREA

Static Methods

static Integer	*countBlobs(String imageFile, Double maxarea)*
static Integer	*countBlobs(Mat img, Double maxarea)*
static double	*pixelsFromPMarea(double micrometers)*

DEFAULT_PM_AREA

public static final double DEFAULT_PM_AREA

Particle area as defined by experimental COTS results

countBlobs

public static Integer countBlobs(String imageFile, Double maxarea)

Returns the number of blobs smaller than the defined area using a filename as input

public static Integer countBlobs(Mat img, Double maxarea)

Returns the number of blobs smaller than the defined area using an image matrix as input

pixelsFromPMarea

public static double pixelsFromPMarea(double microMeters)

Converts area from micrometers to pixels

Example code

```
int particles = COTS.countBlobs("/mnt/sdcard/COTS/testimage.jpg",
```

```
COTS.pixelsFromPMarea(COTS.DEFAULT_PM_AREA));
```

```
package gr.teiath.hackAIR.COTS;

import org.opencv.core.*;
import org.opencv.imgcodecs.Imgcodecs;
import org.opencv.imgproc.Imgproc;

import java.util.ArrayList;
import java.util.Iterator;

public class COTS {
    public static final double DEFAULT_PM_AREA = 300;
    public static Integer countBlobs(String imageFile,Double maxarea){
        Mat img = Imgcodecs.imread(imageFile);
        Mat imggray = new Mat();
        Imgproc.cvtColor(img, imggray, Imgproc.COLOR_BGR2GRAY);
        Imgproc.equalizeHist(imggray,imggray);
        Imgproc.GaussianBlur(imggray,imggray,new Size(3,3),0);
        Imgproc.threshold(imggray,imggray,0,255,Imgproc.THRESH_OTSU);
        ArrayList<MatOfPoint> contours = new ArrayList<MatOfPoint>();
        Imgproc.findContours(imggray,contours,new
Mat(),Imgproc.RETR_LIST,Imgproc.CHAIN_APPROX_SIMPLE);
        int particles = 0;
        Iterator<MatOfPoint> cont = contours.iterator();
        while (cont.hasNext()) {
            MatOfPoint contour = cont.next();
            double area = Imgproc.contourArea(contour);
            if(area<maxarea)particles+=1;
        }
        return particles;
    }
    public static Integer countBlobs(Mat img,Double maxarea){
        Mat imggray = new Mat();
        Imgproc.cvtColor(img, imggray, Imgproc.COLOR_BGR2GRAY);
        Imgproc.equalizeHist(imggray,imggray);
        Imgproc.GaussianBlur(imggray,imggray,new Size(3,3),0);
```

```
        Imgproc.threshold(imggray,imggray,0,255,Imgproc.THRESH_OTSU);
        ArrayList<MatOfPoint> contours = new ArrayList<MatOfPoint>();
        Imgproc.findContours(imggray,contours,new
Mat(),Imgproc.RETR_LIST,Imgproc.CHAIN_APPROX_SIMPLE);
        int particles = 0;
        Iterator<MatOfPoint> cont = contours.iterator();
        while (cont.hasNext()) {
            MatOfPoint contour = cont.next();
            double area = Imgproc.contourArea(contour);
            if(area<maxarea)particles+=1;
        }
       return particles;
    }

    public static double pixelsFromPMarea(double microMeters){
        return 3779.527559055*microMeters*0.0000001;
    }
}
```

5.4 Measurement and results

The experiment lasts for 24h or less, the presented results were acquired after a 24h exposure of the haickAir COTS sensor in the air and the pictures were captured using a x6 magnification super macro lens. Below, at table 4 we present some recommended lens models for capturing the aforementioned image.

Currently a new batch of measurements is taking place, where the pictures are captured with the usage of SmartMicroOptics: Blips Lens with x10 magnification.

Preliminary tests were conducted using a the commercial DC1100 PM measurement system and the results were put in contrast with the corresponding results after applying the computer vision algorithm on the photos captured from the hackAIR COTS node. Figure 32 shows comparative plots that depict the temporal variation of the DC1100 measurements and the estimated PM concentration from the COTS node. It may be seen that the trend of the variation obtains qualitative similarities. The numerical values of these tests are also presented in Table 5.

Model	Description	Provider	Webpage Link	Cost
MoKo Universal Phone Camera Lens Kit	0.45X Super Wide Angle Lens and 15X Super Macro Lens	Sold by BSCstore and Fulfilled by Amazon.	Link	10€
BLIPS Basic Kit	The world's thinnest and lightest micro-lenses.	Sold by Smart MicroOptics	Link	22€
WESTERN OPTICS Professional Quality HD Lens Kit.	0.45x Ultra Wide Angle Lens & 12.5x Super Macro Lens	Sold by Cirrus Industries and Fulfilled by Amazon.	Link	24€
BSR International 2 in 1HD Camera Lens Kit	0.45X Super Wide Angle Lens and 12.5X Macro Lens	Sold by BSR International™ and Fulfilled by Amazon	Link	12€
AFAITH Universal 3 in 1 Camera Lens Kit	Fish Eye Lens + 2in1 Micro/Wide Angle Lens	Sold by AFAITH and Fulfilled by Amazon	Link	9€
APEXEL HD Clip-on Mobile Phone Super Macro Lens	Macro/Close Up: 18X	Sold bby snowbird2016	Link	18€

After completing a series of tests involving more macro lenses it is scheduled to compare the results and define the level of PM concentration in order to be able to distinguish and define a qualitative approach to the PM concentration in the three selected levels.

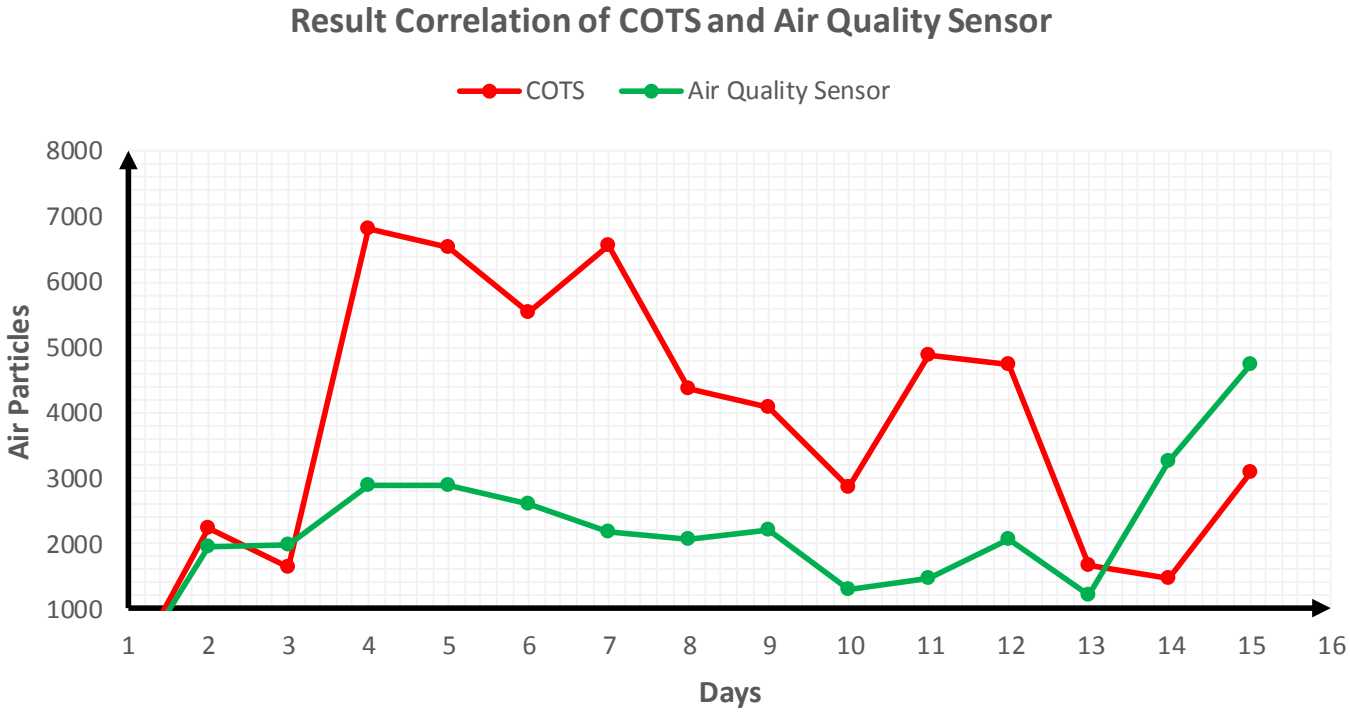

Figure 32 Result correlation between hackAIR COTS Sensor and DC1100 Air Quality Monitor

Number of blobs		
Days	COTS (Blobs Detected)	Air Quality Sensor (Particle Concentration)
Day 1	2242	1948
Day 2	1634	1988
Day 3	6827	2890
Day 4	6531	2885
Day 5	5536	2597
Day 6	6564	2183
Day 7	4359	2073
Day 8	4079	2220
Day 9	2862	1294
Day 10	4881	1471
Day 11	4744	2078
Day 12	1675	1202
Day 13	1461	3274
Day 14	3099	4730

Table 5 Measurement results between hackAIR COTS Sensor and DC1100 Air Quality Monitor

5.5 Theoretical background

The present chapter deals with the necessary theoretical background behind the proposed algorithm. In the beginning, we present some basic pixel relationship concepts such as pixel neighbourhoods and connectivity [32]. Afterwards, we introduce the concept of thresholding, the border tracing algorithm and histogram. Lastly, we introduce the region filling algorithm [33].

5.5.1 Basic Relationships between Pixels

An image may be defined as a two-dimensional function, $f(x,y)$, where x and y are spatial coordinates, and the amplitude of f at any pair of coordinates (x, y) is called the intensity or gray level of the image at that point. When x, y, and the amplitude values of f are all finite, discrete quantities, we call the image a digital image. In this section, we consider that an image is denoted by $f(x, y)$.When referring to a particular pixel, we use lowercase letters, such as p and q.

5.5.2 Neighbours of a Pixel

A pixel p at coordinates (x, y) has four *horizontal* and *vertical* neighbours whose coordinates are given by

$$(x+1, y),(x-1, y),(x, y+1),(x, y-1)$$

This set of pixels, called the 4-*neighbours* of p, is denoted by $N_4(p)$.Each pixel is a unit distance from (x, y), and some of the neighbours of p lie outside the digital image if (x, y) is on the border of the image. The four *diagonal* neighbours of p have coordinates:

$$(x+1, y+1),(x+1, y-1),(x-1, y+1),(x-1, y-1)$$

and are denoted by $ND(p)$.

These points, together with the 4-neighbours, are called the 8-*neighbours* of p, denoted by $N_8(p)$. As before, some of the points in $ND(p)$ and $N_8(p)$ fall outside the image if (x, y) is on the border of the image[33].

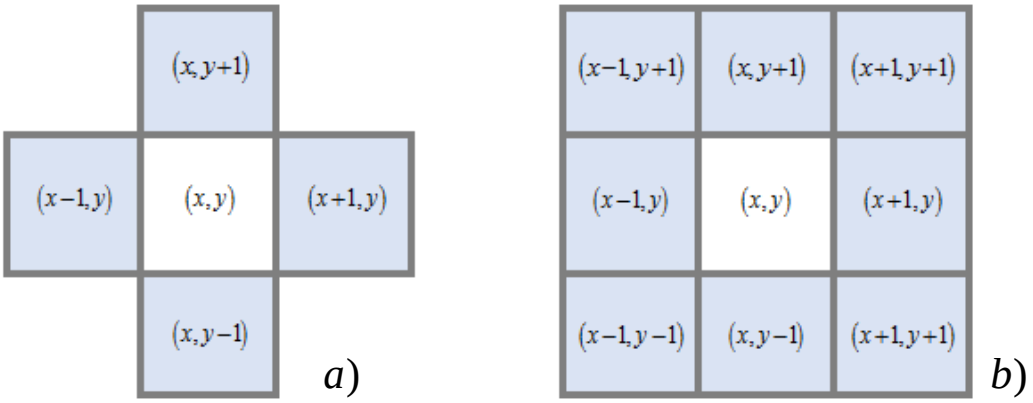

Figure 33 a) 4-Pixel neighbourhood and b) 8-pixel neighbourhood

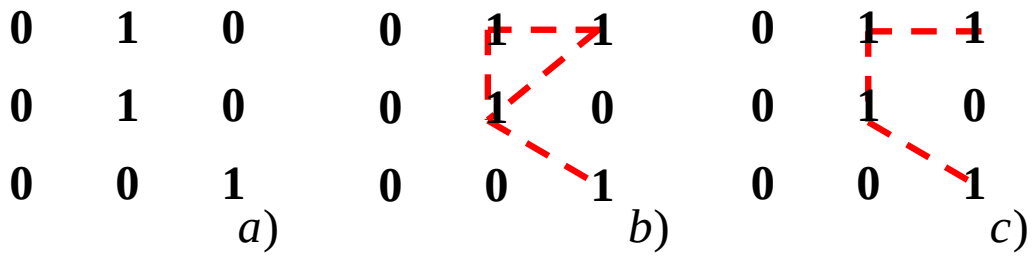

Figure 34 a) Arrangement of pixels, b) pixels that are 8-adjacent (shown dashed) to the centre pixel; c) m-adjacency

5.5.3 Connectivity

Connectivity between pixels is a fundamental concept that simplifies the definition of numerous digital image concepts, such as regions and boundaries. To establish if two pixels are connected, it must be determined if they are neighbours and if their gray levels satisfy a specified criterion of similarity (for example, if their gray levels are equal) [18]. For instance, in a binary image with values 0 and 1, two pixels may be 4-neighbours, but they are said to be connected only if they have the same value.

Let $\mathbf{V}$ be the set of gray-level values used to define adjacency. In a binary image, $\mathbf{V}=\{1\}$ if we are referring to adjacency of pixels with value 1. In a grayscale image, the idea is the same, but set $\mathbf{V}$ typically contains more elements. For example, in the adjacency of pixels with a range of possible gray-level values 0 to 255, set $\mathbf{V}$ could be any subset of these 256 values. We consider three types of adjacency:

a) 4-adjacency. Two pixels p and q with values from $\mathbf{V}$ are 4-adjacent if q is in the set $N_4(p)$.

b) 8-adjacency. Two pixels p and q with values from $\mathbf{V}$ are 8-adjacent if q is in the set $N_8(p)$

c) m-adjacency (mixed adjacency). Two pixels p and q with values from $\mathbf{V}$ are m-adjacent if:

   i. q is in $N_4(p)$, or

   ii. q is in $ND(p)$ and the set $N_4(p)\cap N_4(q)$ has no pixels whose values are from $\mathbf{V}$.

Mixed adjacency is a modification of 8-adjacency. It is introduced to eliminate the ambiguities that often arise when 8-adjacency is used. For example, consider the pixel arrangement shown in Figure 2 a) for $\mathbf{V}=\{1\}$. The three pixels at the top of Figure. 37 b) show multiple 8-adjacency, as indicated by the dashed lines. This ambiguity is removed by using m-adjacency, as shown in Fig. 37 c). Two image subsets S_1 and S_2 are adjacent if some pixel in S_1 is adjacent to some pixel in S_2. It is understood here and in the following definitions that adjacent means 4-, 8-, or m-adjacent.

A (digital) path (or curve) from pixel p with coordinates $(x,\ y)$ to pixel q with coordinates $(s,\ t)$ is a sequence of distinct pixels with coordinates:

$$(x_0,y_0),(x_1,y_1),\ldots,(x_n,y_n)$$

where $(x_0,y_0)=(x_1,y_1),(x_n,y_n)=(s,t)$ and pixels (x_i,y_i) and (x_{i-1},y_{i-1}) are adjacent for $1\le i\le n$.

In this case, n is the length of the path. If $(x_0,y_0)=(x_n,y_n)$ the path is a closed path. We can define 4-, 8-, or m-paths depending on the type of adjacency specified. For example, the paths shown in Figure 2 b) between the

northeast and southeast points are 8-paths, and the path in Figure 2 c) is an m-path. Note the absence of ambiguity in the m-path [22, 32, 33].

Let S represent a subset of pixels in an image. Two pixels p and q are said to be connected in S if there exists a path between them consisting entirely of pixels in S. For any pixel p in S the set of pixels that are connected to it in S is called a connected component of S. If it only has one connected component, then set S is called a connected set.

Let R be a subset of pixels in an image. We call R a region of the image if R is a connected set. The boundary (also called border or contour) of a region R is the set of pixels in the region that have one or more neighbours that are not in R. If R happens to be an entire image (which we recall is a rectangular set of pixels), then its boundary is defined as the set of pixels in the first and last rows and columns of the image. This extra definition is required because an image has no neighbours beyond its border. Normally, when we refer to a region, we are referring to a subset of an image, and any pixels in the boundary of the region that happen to coincide with the border of the image are included implicitly as part of the region boundary.

The concept of an edge is found frequently in discussions dealing with regions and boundaries. There is a key difference between these concepts, however. The boundary of a finite region forms a closed path and is thus a "global" concept. Edges are formed from pixels with derivative values that exceed a pre-set threshold. Thus, the idea of an edge is a "local" concept that is based on a measure of gray-level discontinuity at a point. It is possible to link edge points into edge segments, and sometimes these segments are linked in such a way that correspond to boundaries, but this is not always the case. The one exception in which edges and boundaries correspond is in binary images. Depending on the type of connectivity and edge operators used the edge extracted from a binary region will be the same as the region boundary [34].

5.5.4 Thresholding

Image segmentation is a fundamental process in many image, video, and computer vision applications. It is often used to partition an image into separate regions, which ideally correspond to different real-world objects. It is a critical step towards content analysis and image understanding [35]. The gray levels of pixels belonging to the object are entirely different from the gray levels of the pixels belonging to the background, in many applications of image processing. Thresholding becomes then a simple but effective tool to separate those foreground objects from the background. We can divide the pixels in the image into two major groups, according to their gray-level. These gray-levels may serve as "detectors" to distinguish between background and objects is considering as foreground in the image [36].

Select a gray-level between those two major gray-level groups, which will serve as a threshold to distinguish the two groups (objects and background). Image segmentation is performed by such as boundary detection or region dependent techniques. But the thresholding techniques are more perfect, simple and widely used [20]. Different binarization methods have been performed to evaluate for different types of data. The locally adaptive binarization method is used in gray scale images with low contrast, Varity of background intensity and presence of noise. Niblack's method was found for better thresholding in gray scale image, but still it has been modified for fine and better result [28].

Segmentation involves separating an image into regions corresponding to objects. We usually try to segment regions by identifying common properties. Or, similarly, we identify contours by identifying differences between regions (edges) [34].

The simplest property that pixels in a region can share is intensity. So, a natural way to segment such regions is through thresholding, the separation of light and dark regions. Thresholding creates binary images from grey-level ones by turning all pixels below some threshold to zero and all pixels about that threshold to one.

If $g(x,y)$ is a thresholded version of $f(x,y)$ at some global threshold T,

$$g(x,y)=\begin{cases}1 & f(x,y)\geq T\\ 0 & otherwise\end{cases} \qquad \textit{eq. 5.1}$$

The major problem with thresholding is that we consider only the intensity, not any relationships between the pixels. There is no guarantee that the pixels identified by the thresholding process are contiguous. We can easily include extraneous pixels that aren't part of the desired region, and we can just as easily miss isolated pixels within the region (especially near the boundaries of the region). These effects get worse as the noise gets worse, simply because it's more likely that a pixels intensity doesn't represent the normal intensity in the region. When we use thresholding, we typically have to play with it, sometimes losing too much of the region and sometimes getting too many extraneous background pixels. (Shadows of objects in the image are also a real pain—not just where they fall across another object but where they mistakenly get included as part of a dark object on a light background.)

5.5.5 Global Thresholding

Global thresholding method is used when there the intensity distribution between the objects of foreground and background are very distinct. When the differences between foreground and background objects are very distinct, a single value of threshold can simply be used to differentiate both objects apart. Thus, in this type of thresholding, the value of threshold T depends solely on the property of the pixel and the grey level value of the image. Some most common used global thresholding methods are Otsu method, entropy based thresholding, etc. Otsu's algorithm is a popular global thresholding technique. [30]

5.5.6 Clustering (The Otsu Method)

In image processing, segmentation is often the first step to pre-process images to extract objects of interest for further analysis. Segmentation techniques can be generally categorized into two frameworks, edge-based and region based approaches. As a segmentation technique, Otsu's method is widely used in pattern recognition, document binarization, and computer vision. In many cases Otsu's method is used as a pre-processing technique to segment an image for further processing such as feature analysis and quantification. Otsu's method searches for a threshold that minimizes the intra-class variances of the segmented image and can achieve good results when the histogram of the original image has two distinct peaks, one belongs to the background, and the other belongs to the foreground or the signal.

The Otsu's threshold is found by searching across the whole range of the pixel values of the image until the intra-class variances reach their minimum. As it is defined, the threshold determined by Otsu's method is more profoundly determined by the class that has the larger variance, be it the background or the foreground. As such, Otsu's method may create suboptimal results when the histogram of the image has more than two peaks or if one of the classes has a large variance [24].

Another way of accomplishing similar results is to set the threshold so as to try to make each cluster as tight as possible, thus minimizing their overlap. Obviously, we can't change the distributions, but we can adjust where we separate them. As we adjust the threshold one way, we increase the spread of one and decrease the spread of the other. The goal then is to select the threshold that minimizes the combined spread. We can define the within-class variance as the weighted sum of the variances of each cluster:

$$\sigma_w^2(T) = n_B(T)\sigma_B^2(T) + n_O\sigma_O^2(T) \qquad \textit{eq. 5.2}$$

where

$$n_B(T) = \sum_{i=0}^{T-1} p(i) \qquad \textit{eq. 5.3}$$

$$n_O(T) = \sum_{i=T}^{N-1} p(i) \qquad \textit{eq. 5.4}$$

$\sigma_B^2(T)$: the variance of the pixels in the foreground (below threshold)

$\sigma_O^2(T)$: the variance of the pixels in the foreground (above threshold)

and $[0, N-1]$ is the range of intensity levels.

Computing this within-class variance for each of the two classes for each possible threshold involves a lot of computation, but there's an easier way. If you subtract the within-class variance from the total variance of the combined distribution, you get something called the between-class variance:

$$\sigma_B^2(T) = \sigma^2 - \sigma_w^2(T) = n_B(T)\left|\mu_B(T) - \mu^2\right| + n_O(T)\left|\mu_O(T) - \mu^2\right| \qquad \textit{eq. 5.5}$$

where σ^2 is the combined variance and μ is the combined mean.

Notice that the between-class variance is simply the weighted variance of the cluster means themselves around the overall mean. Substituting

$$\mu = n_B(T)\mu_B(T) + n_O(T)\mu_O(T) \qquad \textit{eq. 5.6}$$

and simplifying, we get

$$\sigma_B^2(T) = n_B(T)\,n_O(T)\left|\mu_B(T) - \mu_O(T)\right| \qquad \textit{eq. 5.7}$$

So, for each potential threshold T we

a) Separate the pixels into two clusters according to the threshold.
b) Find the mean of each cluster.
c) Square the difference between the means.
d) Multiply by the number of pixels in one cluster times the number in the other.

This depends only on the difference between the means of the two clusters, thus avoiding having to calculate differences between individual intensities and the cluster means. The optimal threshold is the one that maximizes the between-class variance (or, conversely, minimizes the within-class variance) [14].

This still sounds like a lot of work, since we have to do this for each possible threshold, but it turns out that the computations aren't independent as we change from one threshold to another. We can update $n_B(T)$, $n_O(T)$, and the respective cluster means $\mu_B(T)$ and $\mu_O(T)$ as pixels move from one cluster to the other as T increases. Using simple recurrence relations, we can update the between-class variance as we successively test each threshold:

$$n_B(T) = n_B(T) + n_T \qquad \textit{eq. 5.8}$$

$$n_O(T) = n_O(T) - n_T \qquad \textit{eq. 5.9}$$

$$\mu_B(T) = \frac{\mu_B(T)\,n_B(T) + n_T}{n_B(T+1)} \qquad \textit{eq. 5.10}$$

$$\mu_O(T) = \frac{\mu_O(T)\,n_O(T) - n_T}{n_O(T+1)} \qquad \textit{eq. 5.11}$$

This method is sometimes called the Otsu method, after its first publisher [17, 20, 28, 36].

5.5.7 Contour Tracing

One method of finding and analysing the connected components of T is to scan T in some manner until a border pixel of a component P is found and subsequently to trace the boundary of P until the entire boundary is obtained. The boundary thus obtained can be stored as a doubly linked circular list of pixels or as a polygon of the boundary of the pixels in question. Such procedures are generally termed *contour tracing*. There is frequently no standard formal definition given of the boundary of a connected pattern. In practice, the boundary is what seems appropriate as the boundary to humans and algorithms are judged in accordance to how well they agree with human perception. However, in order to make more precise statements about our algorithms there are two useful definitions of "border" points which are intimately related to the boundary. We say that a black pixel of P is a 4-border point if at least one of its 4-neighbours is white. We say that a black pixel of P is an 8-border point if at least one of its 8-neighbours is white. [15]

The proposed algorithm for the hackAIR COTS sensor, implements the Moore-Neighbourhood-Tracing algorithm, which is presented is the following section.[15, 30, 34, 37]

5.5.7.1 Moore-Neighbourhood-Tracing Algorithm

Input: *A square tessellation* T *containing a connected component* P *of black cells.*

Output: *A sequence* $B(b_1, b_2, \ldots, b_k)$ *of boundary pixels.*

Begin

{Initialization: find a pixel in P*, initialize* B*, define* $M(p)$ *to be the Moore neighbourhood of the current pixel* p *}*

1. Set B *to be empty.*

2. From bottom to top and left to right scan the cells of T *until a pixel s of* P *is found (until the current pixel* $p = s$ *is a black pixel of* P*), insert* s *in* B*. Let* b *denote the current boundary point, i.e.,* $b = s$

3. Backtrack (move the current pixel to the pixel from which s was entered) and advance the current pixel p being examined to the next clockwise pixel in $M(b)$.

while p *is not equal to* s ***do***

> ***if*** p *is black, insert* p *in B, set* $b = s$ *and backtrack (move the current pixel to the pixel from which* $p = b$ *was entered);*

else *advance the current pixel p to the next clockwise pixel in* $M(b)$.

end while

End

5.5.8 Histogram

Histograms are the basis for numerous spatial domain processing techniques [38]. Histogram manipulation can be used effectively for image enhancement. In addition to providing useful image statistics, the information inherent in histograms also is quite useful in other image processing applications, such as image compression and segmentation. Histograms are simple to calculate in software and also lend themselves to economic hardware implementations, thus making them a popular tool for real-time image processing [31, 39, 40].

The histogram of a digital image with gray levels in the range $[0, L-1]$ is a discrete function $h(r_k) = n_k$, where r_k is the kth gray level and n_k is the number of pixels in the image having gray level r_k. It is common practice to normalize a histogram by dividing each of its values by the total number of pixels in the image, denoted by n. Thus, a normalized histogram is given by

$$p(r_k) = \frac{n_k}{n}, \text{ for } k = 0,1,2,\ldots L-1. \qquad \text{eq. 5.12}$$

Loosely speaking, $p(r_k)$ gives an estimate of the probability of occurrence of gray level r_k. Note that the sum of all components of a normalized histogram is equal to 1 [41].

5.5.8.1 Histogram Equalization

Consider for a moment continuous functions, and let the variable r represent the gray levels of the image to be enhanced. In the initial part of our discussion we assume that r has been normalized to the interval $[0,1]$, with $r = 0$ representing black and $r = 1$ representing white we consider a discrete formulation and allow pixel values to be in the interval $[0,\ldots L-1]$ [38]. For any r satisfying the aforementioned conditions, we focus attention on transformations of the form:

$$s = T(r), \qquad 0 \le r \le 1 \qquad \text{eq. 5.13}$$

that produce a level s for every pixel value r in the original image. For reasons that will become obvious shortly, we assume that the transformation function $T(r)$ satisfies the following conditions:

a. $T(r)$ is single-valued and monotonically increasing in the interval $0 \le r \le 1$ and

b. $0 \le T(r) \le 1$ for $0 \le r \le 1$.

The requirement in a. that $T(r)$ be single valued is needed to guarantee that the inverse transformation will exist, and the monotonicity condition preserves the increasing order from black to white in the output image. A transformation function that is not monotonically increasing could result in at least a section of the intensity range being inverted, thus producing some inverted gray levels in the output image [39, 40]. While this may be a desirable effect in some cases, that is not what we are after in the present discussion. Finally, condition (b) guarantees that the output gray levels will be in the same range as the input levels. The inverse transformation from s back to r is denoted

$$r = T^{-1}(s) \qquad 0 \le s \le 1 \qquad \text{eq. 5.14}$$

It can be shown that even if $T(r)$ satisfies conditions (a) and (b), it is possible that the corresponding inverse $T^{-1}(s)$ may fail to be single valued.

The gray levels in an image may be viewed as random variables in the interval $[0,1]$. One of the most fundamental descriptors of a random variable is its probability density function (PDF). Let $p_r(r)$ and $p_s(s)$ denote the probability density functions of random variables r and s, respectively, where the subscripts on p are used to denote that p_r and p_s are different functions. A basic result from an elementary probability theory is that, if $p_r(r)$ and $T(r)$ are known and $T^{-1}(s)$ satisfies condition (a), then the probability density function $p_s(s)$ of the Transformed variable s can be obtained using a rather simple formula:

$$p_s(s) = p_r(r)\left|\frac{dr}{ds}\right| \qquad \textit{eq. 5.15}$$

Thus, the probability density function of the transformed variable, s, is determined by the gray-level PDF of the input image and by the chosen transformation function.

A transformation function of particular importance in image processing has the form

$$s = T(r) = \int_0^r p_r(w)\,dw \qquad \textit{eq. 5.16}$$

where w is a dummy variable of integration. The right side of eq. 1.16 is recognized as the cumulative distribution function (CDF) of random variable r. Since probability density functions are always positive, and recalling that the integral of a function is the area under the function, it follows that this transformation function is single valued and monotonically increasing, and, therefore, satisfies condition (a). Similarly, the integral of a probability density function for variables in the range $[0,1]$ also is in the range $[0,1]$, so condition (b) is satisfied as well. Given transformation function $T(r)$,we find $p_s(s)$ by applying eq. 5.17. We know from basic calculus (Leibniz's rule) that the derivative of a definite integral with respect to its upper limit is simply the integrand evaluated at that limit. In other words,

$$\frac{ds}{dr} = \frac{T(r)}{dr} = \frac{d}{dr}\left[\int_0^r p_r(w)\,dw\right] = p_r(r) \qquad \textit{eq. 5.18}$$

Substituting this result for dr/ds into eq. 1.15, and keeping in mind that all probability values are positive, yields

$$p_s(s) = p_r(r)\left|\frac{ds}{dr}\right| = p_r(r)\left|\frac{1}{p_r(r)}\right| = 1 \qquad 0 \le s \le 1 \qquad \textit{eq. 5.19}$$

Because $p_s(s)$ is a probability density function, it follows that it must be zero outside the interval $[0,1]$ in this case because its integral over all values of s must equal 1. We recognize the form of $p_s(s)$ given in Eq. 1.18 as a uniform probability density function. Simply stated, we have demonstrated that performing the transformation function given in Eq. 1.16 yields a random variable s characterized by a uniform probability density function. It is important to note from Eq. 1.16 that $T(r)$ depends on $p_r(r)$, but, as indicated by Eq. 1.18, the resulting $p_s(s)$ always is uniform, independent of the form of $p_r(r)$.

For discrete values, we deal with probabilities and summations instead of probability density functions and integrals. The probability of occurrence of gray level r_k in an image is approximated by

$$p_r(r_k) = \frac{n_k}{n} \qquad k = 0,1,2\ldots,L-1 \qquad \textit{eq. 5.20}$$

where, as noted at the beginning of this section, n is the total number of pixels in the image, n_k is the number of pixels that have gray level r_k, and L is the total number of possible gray levels in the image.

The discrete version of the transformation function given in Eq. 1.16 is

$$s_k = T(r_k) = \sum_{j=0}^{k} p_r(r_j) = \sum_{j=0}^{k} \frac{n_j}{n} \qquad k = 0,1,2,\ldots,L-1 \qquad \textit{eq. 5.21}$$

Thus, a processed (output) image is obtained by mapping each pixel with level r_k in the input image into a corresponding pixel with level s_k in the output image via Eq. 1.20. As indicated earlier, a plot of $p_r(r_k)$ versus r_k is called a histogram. The transformation (mapping) given in Eq. 1.20 is called histogram equalization or histogram linearization.

Unlike its continuous counterpart, it cannot be proved in general that this discrete transformation will produce the

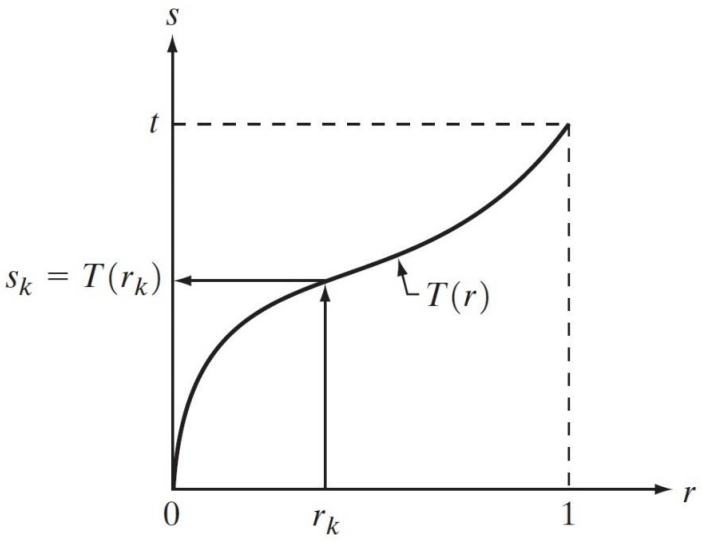

Figure 35 A gray-level transformation function that is both single valued and monotonically increasing.

discrete equivalent of a uniform probability density function, which would be a uniform histogram. However, as will

be seen shortly, use of Eq. 1.20 does have the general tendency of spreading the histogram of the input image so that the levels of the histogram-equalized image will span a fuller range of the gray scale [39].

We discussed earlier in this section the many advantages of having gray-level values that cover the entire gray scale. In addition to producing gray levels that have this tendency, the method just derived has the additional advantage that it is fully "automatic." In other words, given an image, the process of histogram equalization consists simply of implementing Eq. 1.20, which is based on information that can be extracted directly from the given image, without the need for further parameter specifications. The inverse transformation from s back to r r is denoted by

$$r_k = T^{-1}(s_k) \qquad k = 0,1,2\ldots,L-1 \qquad \textit{eq. 5.22}$$

It can be shown that the inverse transformation in eq. 1.21 satisfies conditions (a) and (b) stated previously in this section only if none of the levels, $r_k, k = 0,1,2\ldots,L-1$, are missing from the input image.

5.5.8.2 Contrast limited adaptive histogram equalization(CLAHE)

As mention earlier, the histogram of a digital image with intensity levels in the range $[0,L-1]$ is a discrete function: $h(r_k) = n_k$,

Where r_k is the kth intensity value and n_k is the number of pixel in the image with intensity r_k .

A normalized histogram is given by:

$$p_r(r_k) = \frac{n_k}{n} \qquad k = 0,1,2\ldots,L-1 \qquad \textit{eq. 1.12}$$

Loosely speaking $p_r(r_k)$ is an estimated of the probability of occurrence of intensity level r_k in an image. The sum of all components of normalized histogram is equal to 1.

The histogram equalization is a method in image processing of contrast adjustment using the image's histogram. This method usually increases the global contrast of many images, through transforming the original image histogram to a uniform histogram, that is, trying to make uniform the distribution intensity pixels of the image. The histogram equalization is obtained by next equation:

$$s_k = (L-1)\sum_{j=0}^{k} p_r(r_j) \qquad k = 0,1,2,\ldots,L-1 \qquad \textit{eq. 5.23}$$

where s_k is the new distribution of the histogram.

This procedure is based on the assumption that the image quality is uniform over all areas and one unique grayscale mapping provides similar enhancement for all regions of the image. However, when distributions of grayscales change from one region to another, this assumption is not valid. In this case, an adaptive histogram equalization technique can significantly outperform the standard approach. In this case, the image is divided into a limited number of regions and the same histogram equalization technique is applied to pixels in each region [38].

Even in some cases this method cannot resolve the problem, when grayscale distribution is highly localized, it might not be desirable to transform very low-contrast images by full histogram equalization. In these cases, the mapping curve may include segments with high slopes, meaning that two very close grayscales might be mapped to

significantly different grayscales. This issue is resolved by limiting the contrast that is allowed through histogram equalization [30].

The combination of this limited contrast approach with the aforementioned adaptive histogram equalization results in what is referred to as Contrast Limited Adaptive Histogram Equalization (CLAHE) proposed in [10]. The CLAHE procedure consists in [42, 43]:

a) First the image has to be divided into several nonoverlapping regions of almost equal sizes.
b) Secondly the histogram of each region is calculated. Then, based on a desired limit for contrast expansion, a clip limit for clipping histograms is obtained.
c) Next, each histogram is redistributed in such a way that its height does not go beyond the clip limit. The clip limit β is obtained by:

$$\beta = \frac{n}{L}\left(1+\frac{a}{100}\left(s_{\max}-1\right)\right) \qquad \textit{eq. 5.24}$$

4. where a is a clip factor, if clip factor is equal to zero the clip limit becomes exactly equal to $\left(\frac{n}{L}\right)$ moreover if clip limit is equal to 100 the maximum allowable slope is $s_{\max}$

d) Finally, cumulative distribution functions (CDF) of the resultant contrast limited histograms are determined for grayscale mapping.

The pixels are mapped by linearly combining the results from the mappings of the four nearest regions.

5.5.9 Boundary Extraction

The boundary of a set A denoted by $\beta(A)$ can be obtained by first eroding A by B and then performing the set difference between A and its erosion. That is

$$\beta(A) = A - (A!\ B) \qquad \textit{eq. 5.25}$$

where B is a suitable structuring element.

The following figure illustrates the mechanics of boundary extraction. It shows a simple binary object, a structuring element B and the result of using eq. 1.24

Although the structuring element shown in the afore mentioned figure is among the most frequently used, it is by no means unique. For example, using a $5x5$ structuring element of 1's would result in a boundary between 2 and 3 pixels thick. Note that when the origin of B is on the edges of the set, part of the structuring element may be outside the image. The normal treatment of this condition is to assume that values outside the borders of the image are 0 [25, 37].

Figure 40 further illustrates the use of eq. 1.24 with the structuring element of Figure 4 b). In this example, binary 1's are shown in the white and 0's in black, so the elements of the structuring, which are 1's, also are treated as white. Because of the elements used, the boundary shown in Figure 5 is one pixel thick.

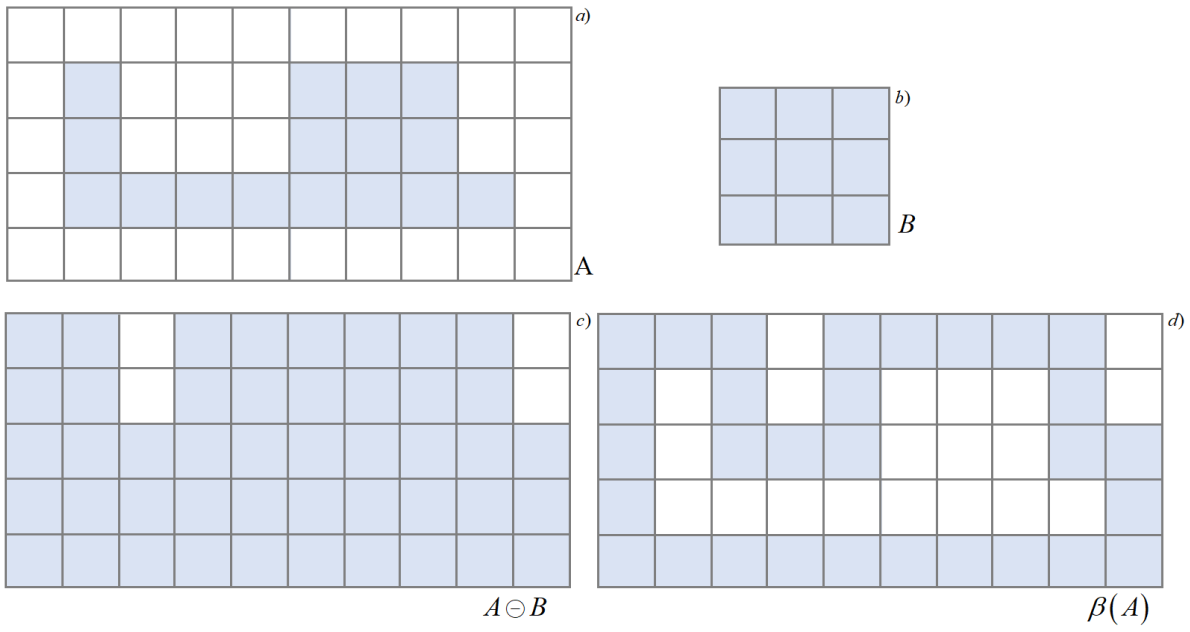

Figure 36 a) Set A, b) Structuring element B, c) A eroded by B and d) Boundary, given by the set deference of A and its erosion

Figure 37 Boundary extraction by morphological processing in a simply binary image.

5.5.10 Region Filling

Next, we develop a simple algorithm for region filling based on set dilations, complementation, and intersections. In Figure 6 A denotes a set containing a subset whose elements are 8-connected boundary points of a region. Beginning with a point p inside the boundary, the objective is to fill the entire region with 1's. If we adopt the convention that all non-boundary (background) points are labelled 0, the we assign a value of 1 to p to begin. The following procedure then fill the region with 1's:

$$X_k = (X_{k-1} \oplus B) \cap A^C \qquad k = 1, 2, 3, \ldots \qquad \text{eq. 5.26}$$

Where $X_0 = p$ and B is the symmetric structuring element shown in Figure 7.

The algorithm terminates at iteration step k if $X_k = X_{k-1}$. The set union of X_k and A contains the filled set and its boundary.

The dilation process of eq. 1.25 would fill the entire area it left unchecked. However, the intersection at each step with A^C limits the result to inside the region of interest. This is our first example of how a morphological process can be conditioned to meet a desired property. In the current application, it is appropriately called conditional dilation. The rest of Figure 41 illustrates further mechanics of eq. 1.25. Although this example has only one subset, the concept applies to any finite number of subsets, assuming that a point inside each boundary is given [26, 27, 30].

Figure 42 a) shows an image composed of white circles with black inner spots. An image such as this might result from thresholding into two levels a scene containing polished spheres (e.g. ball bearings). The dark spots inside the spheres and Figure 42 b) shows the result of filling that component. Finally, Figure 8 (c) shows the result of filling all the spheres.

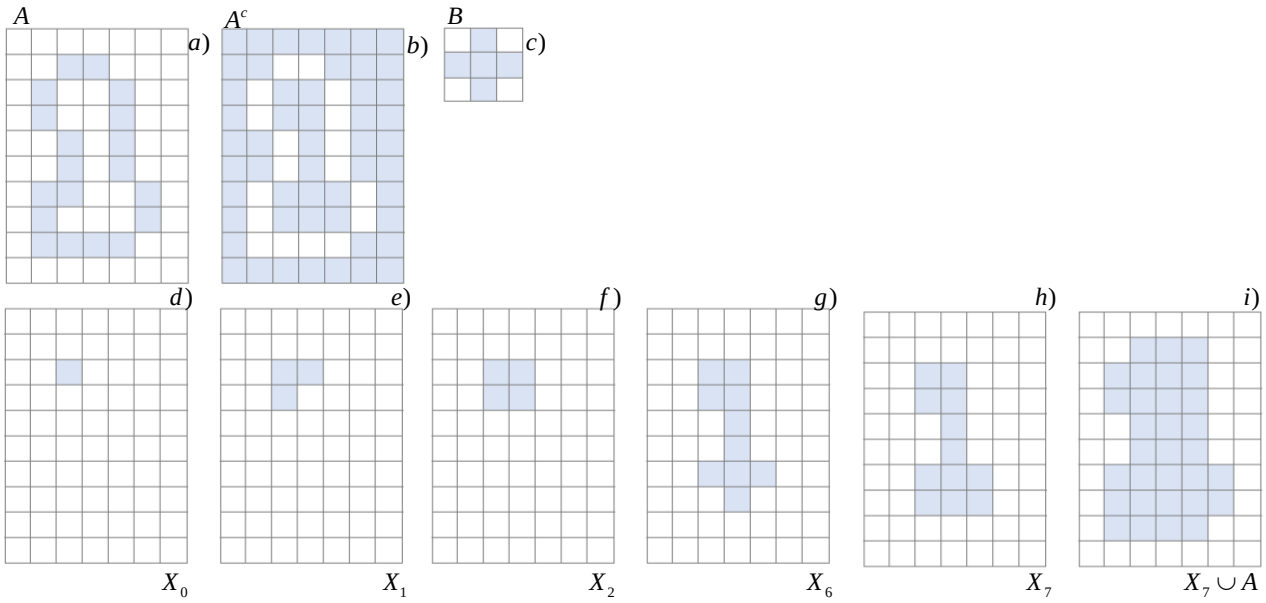

Figure 38 a)Set A, b) Complement of A, c) Structuring element B, d) Initial point inside the boundary, e)-h) Various steps of eq. 1.25, i) final result

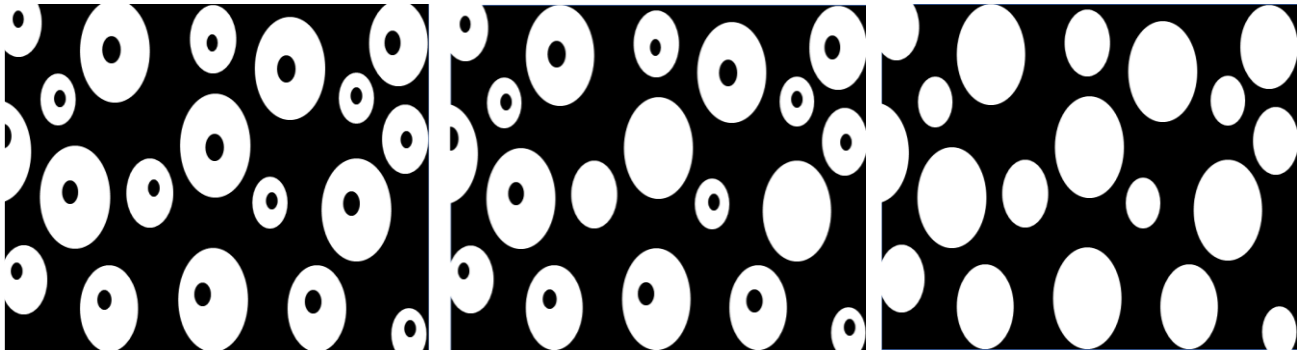

Figure 39 a) Binary image, the black dot inside one of the regions is the starting point of the region filling algorithm, b) Result of filling that region c) Result of filling of all the regions.

5.5.11 Extraction of Connected Components

The concepts of connectivity and connected components were discussed earlier. In practice, extraction of connected components in a binary image is a central to many automated image analysis applications [15, 30, 37]. Let Y represent a connected component contained in a set A and assume that a point p of Y is known. Then the following iterative expression yields all the elements of Y :

$$X_k = (X_{k-1} \oplus B) \cap A \qquad k = 1,2,3,\ldots \qquad \text{eq. 5.27}$$

Where $X_0 = p$, and B is a suitable structuring element, as shown in Figure 9. If $X_k = X_{k-1}$, the algorithm has converged and we let $Y = X_k$.

Eq. 5.28 is similar in form with eq. 1.25. The only difference is the use of A instead of its complement. This difference arises because all the elements sought (that this, the elements of the connected component) are labelled 1.

The inter section with A at each iterative step eliminates dilations centred on elements labelled 0. Figure 43 illustrates the mechanics of eq. 5.29. Note that the shape of the structuring element assumes 8-connectivity between pixels. As in the region filling algorithm, the results just discussed are applicable to any finite number of connected components contained in A, assuming that a point is known in each component.

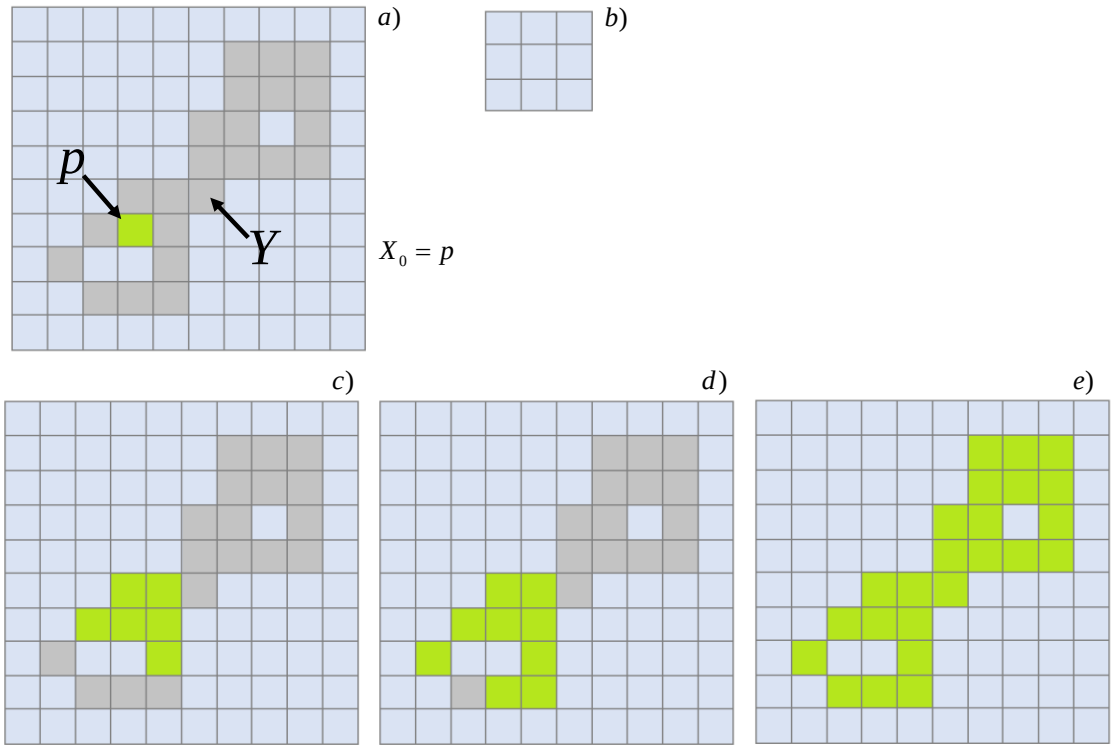

Figure 40 a) Set A showing initial point p (all shaded points are valued 1, but are shown different from p to indicate that they have not yet found by the algorithm. b) Structuring element, c) Result of the first iterative step d) Result of second step. e) Final result.

5.5.12 Proposed Algorithm

The 2nd chapter presents the proposed algorithm for blob detection. The algorithm is programmed using Mathworks' MATLAB which is a multi-paradigm numerical computing environment and fourth-generation programming language.

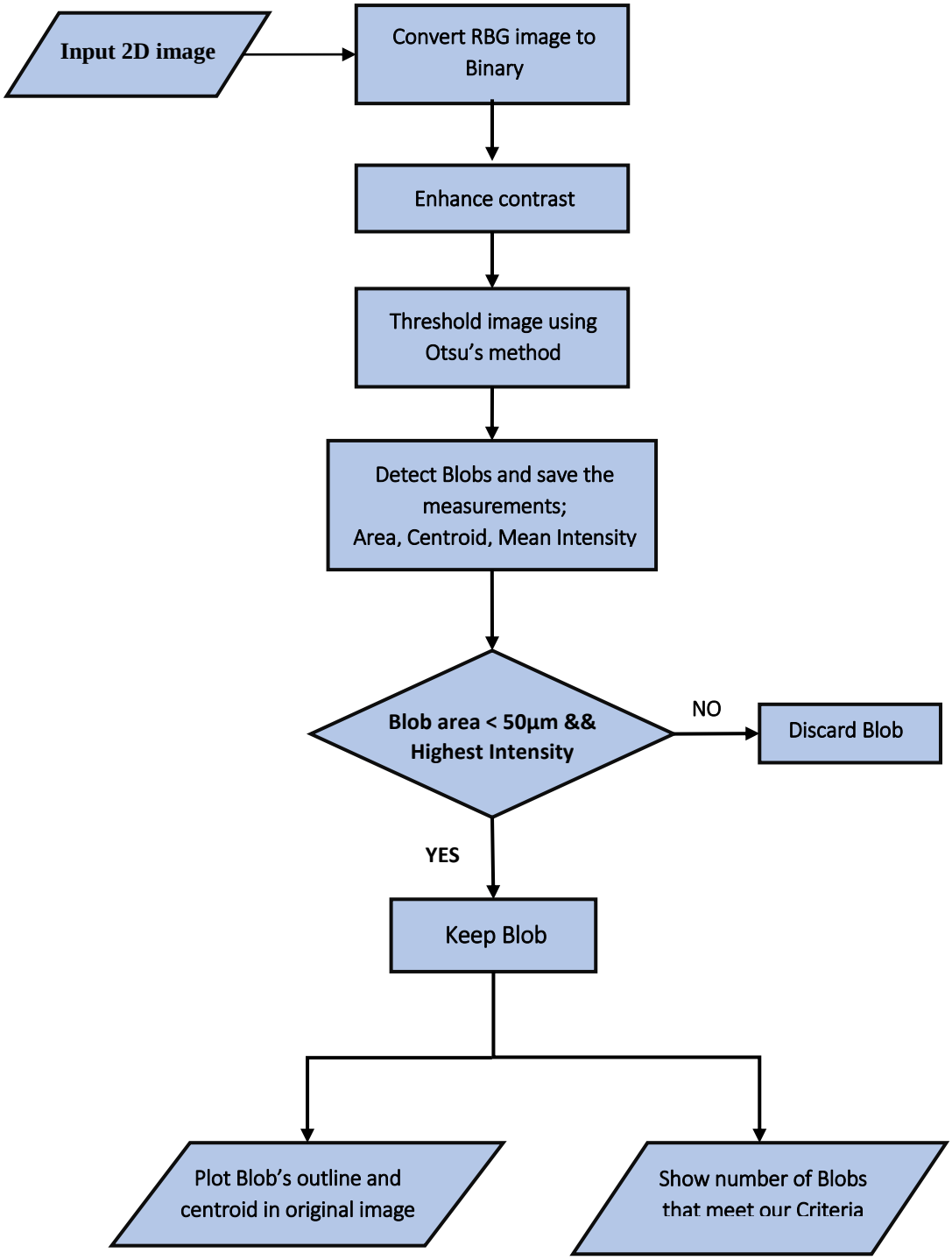

Figure 41 Proposed algorithm for blob selection.

6 Conclusions

The up to now work has set all the hackAIR nodes requirements and all the hardware and software has been developed. Since all hardware is ready, it is expected to conduct minor level modifications that will not change any of the specifications but they will increase the user friendliness of the designed systems and will provide further features as extras on the existing ones. These changes will be reported under D7.6 : Report on hackAIR updated support services and methodologies [M36].

The designed hardware and software targets to activate on the hackAIR concept (particulate matter measurements) large technology friendly communities such as Arduino and PSoC users. Additionally, users of lower technical capabilities will be able to estimate the outdoor PM concentrations using simple devices like macro lenses and mobile phones to qualitatively approach the air quality level. Provided that all the above tools (i.e. that are used as fundamentals on the hackAIR nodes design) rapidly and dynamically change, all the necessary measures are taken in order to enable users of compatible systems and processing nodes to integrate their systems and add their data on the hackAIR portal. Such measures involve well established and detailed hardware and software documentation available on the github in form of wiki, open format of the hackAIR data packet, widely known and available hardware versions, wide range and expandable networking capabilities.

It is considered that with the designed and released tools the air quality awareness will be increased since each user may see their contribution on a global map and see the impact their activities have on the local air quality of the air having in mind that minor changes of habits have major impact on the environment and sequentially the quality of life.

References

1. Austin, E., et al., *Laboratory Evaluation of the Shinyei PPD42NS Low-Cost Particulate Matter Sensor.* PloS one, 2015(dx.doi.org/10.1371/journal.pone.0137789).
2. Arling, J., Connor, K., Mercieca, M., *Air quality sensor network for Philadelphia*. 2010.
3. Sharp, http://www.sharp-world.com/products/device/lineup/selection/opto/dust/index.html.
4. Plantower, http://www.plantower.com/content/?94.html.
5. Morpugro, A., F. Pedersini, and A. Reina. *A low-cost instrument for environmental particulate analysis based on optical scattering*. in *IEEE Instrumentation and Measurement Technology Conference (I2MTC),*. 2012.
6. http://www.dylosproducts.com/dcproairqumo.html. Dylos.
7. Sharp, i., http://www.sharpsma.com/webfm_send/1488.
8. Sharp, i., http://media.digikey.com/pdf/Data%20Sheets/Sharp%20PDFs/DN7C3CA006_Spec.pdf.
9. Amphenol, http://gr.mouser.com/pdfdocs/01Cdustsensordatasheet-English1-22-3.pdf.
10. Sharp, i., http://media.digikey.com/pdf/Data%20Sheets/Sharp%20PDFs/GP2Y1023AU0F%20Specs.pdf.
11. Shinyei, http://wiki.seeedstudio.com/wiki/Grove_-_Dust_sensor.
12. Plantower, i., *https://*www.google.gr/url?sa=t&rct=j&q=&esrc=s&source=web&cd=2&cad=rja&uact=8&ved=0ahUKEwiK7rSc85XPAhWhCsAKHeuPBB0QFgghMAE&url=http%3A%2F%2Fwww.plantower.com%2Fen%2Fcontent%2F%3F108.html&usg=AFQjCNFvycVx7qJ7CMsnFUHZtrA0Rhz3pw&sig2=ZnHJl-dXa_OecgCMt--KnQ.
13. NovaFitness, *https://*www.google.gr/url?sa=t&rct=j&q=&esrc=s&source=web&cd=4&ved=0ahUKEwjPq_P78pXPAhXHDMAKHUL3BasQFgg4MAM&url=http%3A%2F%2Fbreathe.indiaspend.org%2Fwp-content%2Fuploads%2F2015%2F11%2Fnova_laser_sensor.pdf&usg=AFQjCNFp3mrXRfBTTe2iyLFZldxzqXgmhw&sig2=pKN0.
14. Bradski, G. and A. Kaehler, *Learning OpenCV*. 1st ed. 2008: O'Reilly Media, Inc.
15. Foltz, M.A., *Connected Components in Binary Images.* Machine Vision, 1997. **6**(866).
16. Haralick, R.M. and L.G. Shapiro, *Computer and Robot Vision*. 1st ed. Vol. 1. 1992: Addison-Wesley.
17. Ismai, A.H. and M.H. Marhaban. *A simple approach to determine the best threshold value for automatic image thresholding*. in *Signal and Image Processing Applications (ICSIPA), 2009 IEEE International Conference on*. 2009. Kuala Lumpur, Malaysia: IEEE.
18. Jain, A.K., *Fundamentals of Digital Image Processing*. PRENTICE HALL INFORMATION AND SYSTEM SCIENCES SERIES, ed. T. Kailath. 1989: Prentice-Hall.
19. Kazlouski, A. and R.K. Sadykhov. *Plain objects detection in image based on a contour tracing algorithm in a binary image*. in *Innovations in Intelligent Systems and Applications (INISTA) Proceedings, 2014 IEEE International Symposium on*. 2014. Alberobello, Italy: IEEE.
20. Lee, S., S. Chung, and R. Park, *A comparative performance study of several global thresholding techniques for segmentation.* Computer Vision, Graphics, and Image Processing, 1990. **52**(2): p. 171-190.
21. Lindh, J., *Bluetooth Low Energy beacons*. 2015, Texas Instruments.
22. Nixon, M.S. and A.S. Aguado, *Feature Extraction & Image Processing for Computer Vision*. 3rd ed. 2012: Elsevier Ltd.
23. Nixon S, M. and A. Aguado S, *Feature Extraction and Image Processing*. 2002: Newnes.
24. Otsu, N., *A Threshold Selection Method from Gray-Level Histograms.* IEEE Transactions on Systems, Man, and Cybernetics, 1979. **9**(1): p. 62 - 66.
25. PRATT, W.K., *DIGITAL IMAGE PROCESSING: PIKS Inside*. 3rd ed. 2001: Wiley-Interscience.
26. Sedgewick, R., *Algorithms in C*. 3rd ed. 1998: Addison-Wesley.
27. Sedgewick, R. and K. Wayne, *Algorithms* 2001: Pearson Education, Inc.
28. Senthilkumaran, N. and S. Vaithegi, *Image segmentation by using Thresholding techniques for medical images* Computer Science & Engineering: An International Journal (CSEIJ),, 2016. **6**(1).
29. Theodoridis, S. and K. Koutroumbas, *Pattern Recognition (Third Edition).* Academic Press, 2006: p. 837.
30. Wilson, J.N. and G.X. Ritter, *HANDBOOK OF Computer Vision Algorithms in Image Algebra*. 2nd ed. 2000: CRC Press.

31. Zuiderveld, K., *Contrast limited adaptive histogram equalization*, in *Graphics gems IV*. 1994: Academic Press Professional, Inc. San Diego, CA, USA. p. 474-485
32. Easton Jr., R.L., *Fundamentals of Digital Image Processing*. 2010.
33. Gonzalez, R.C. and R.E. Woods, *Digital Image Processing Second Edition*. 2nd ed. 2002: Pearson Prentice Hall.
34. Toussaint, G.T., *Grids, Connectivity and Contour Tracing*, in *Computational Morphology*. 1988.
35. Zhang, H., J.E. Fritts, and S.A. Goldman, *Image segmentation evaluation: A survey of unsupervised methods.* Computer Vision and Image Understanding, 2008. **110**(2): p. 260–280.
36. Milstein, N., *Image Segmentation by Adaptive Thresholding.* Spring, 1998.
37. Soille, P., *Morphological Image Analysis: Principles and Applications*. 1999: Springer-Verlag.
38. Benitez-Garcia, G., et al., *Face Identification Based on Contrast Limited Adaptive Histogram Equalization (CLAHE).* Mechanical and Electrical Engineering School of National Polytechnic Institute of Mexico. Mexico, Mexico D.F., 2012.
39. Garima, Y., M. Saurabh, and A. Anjali. *Contrast Limited Adaptive Histogram Equalization Based Enhancement For Real Time Video System*. in *Advances in Computing, Communications and Informatics (ICACCI, 2014 International Conference on*. 2014. New Delhi, India: IEEE.
40. Sasi, N. and V. Jayasree, *Contrast Limited Adaptive Histogram Equalization for Qualitative Enhancement of Myocardial Perfusion Images.* Engineering, 2013. **5**: p. 326-331.
41. Schilling, R.J. and S.L. Harris, *Fundamentals of Digital Signal Processing Using MATLAB®*. 2nd ed. 2012: Cengage Learning.
42. Forsyth, D.A. and J. Ponce, *Computer Vision: A Modern Approach*. 2011: Pearson
43. Hartley, R. and A. Zisserman, *Multiple View Geometry in Computer Vision*. 2nd ed. 2004: Cambridge University Press.