Project Description

The goal of this project is to build a pair of wireless microcontrollers with the goal of measuring a temperature distribution through an apartment. The temperature data would then be processed and displayed visually on a 2D map of the space.

The description of this project will be broken into two sections. In the first section, I will describe how temperature was measured, transmitted, and stored using an 8-bit microcontroller, 10 kΩ thermistors, and 437 kHz radio receiver-transceivers. In the second section I will describe how I created a 2D floor map, interpolated temperatures to every pixel, and mapped temperature to a color using python and OpenCV.

# Part 1

Required devices:

2x Arduino development boards (I used Arduino UNO)

1x MicroSD card adapter for GPIO

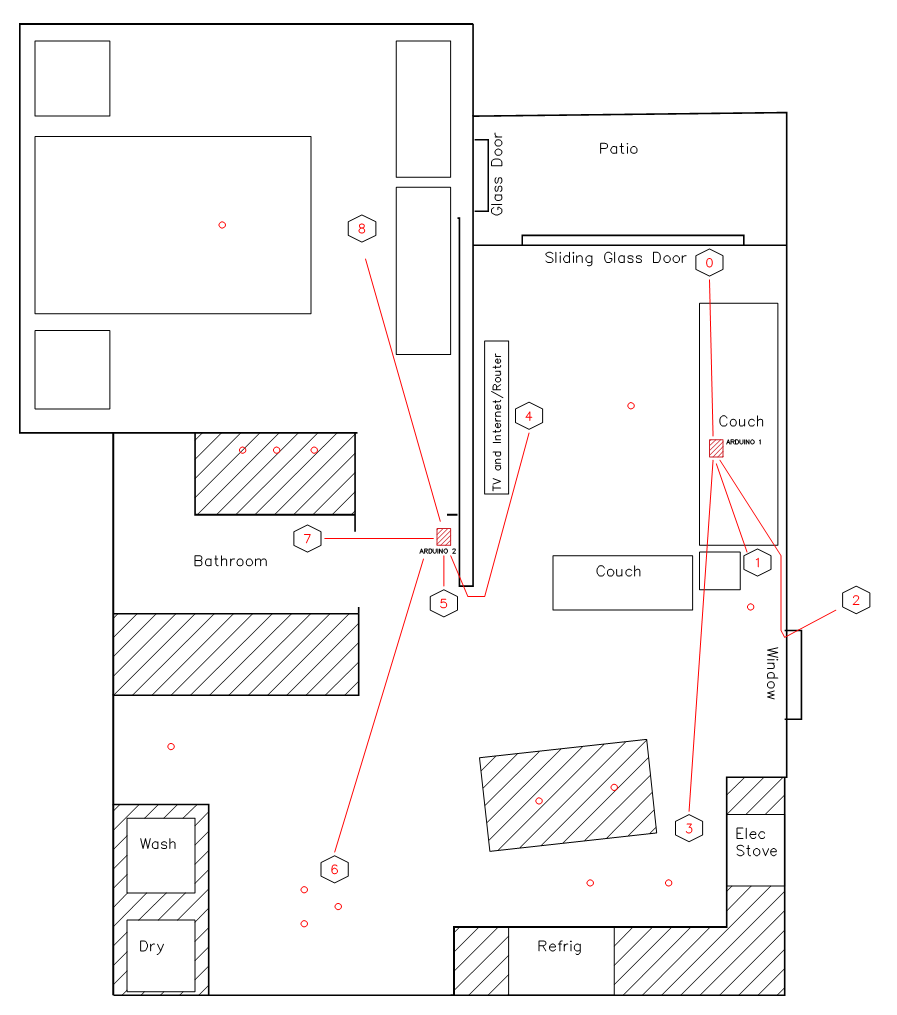
1x Real Time Clock (I used a DS3231)

1x pair of radio transmitter and receiver 433MHz

Breadboards and wires

9x (more or less depending on what you want) 10 kΩ thermistor type 2

Arduino development boards are used for measurement, transmission, and datalogging of temperature measurements. Two Arduino boards are used in this project. The first board will act as a datalogger, measurement station, and receiver for a remote board. The second board will take distributed measurements and transmit them to the receiver board (Figure X).



In my project, I placed distributed thermistors throughout my apartment. Each hexagon labeled 0-8 represents a thermistor. Thermistors 0-3 are connected to the receiver, and thermistors 4-8 are connected to the transmitter board. The goal is to get temperature measurements for all 9 thermistors to the receiver board so that temperature data can be logged and saved in a csv file for image processing (part 2).

## Datalogger

We need a convenient way to store measurements and save them in a format that is easily readable by a PC for use in part 2. A relatively simple way to do this is with Secure Digital (SD) card. The SD card will communicate with the Arduino via Serial Peripheral Interface (SPI) with the help of an SD adapter. There will be six wires between our Arduino and SD card. These wires are serial clock (SCK), master in slave out (MISO), master out slave in (MOSI), chip select (CS), power, and ground.

It is possible to write to a SD card without using a microSD adapter, but you must consider the logic levels at which the Arduino and SD card operate at. Most SD cards will operate at 3.3V logic levels (some cards can step down to 1.8V levels), but the Atmel328 works on a 5V logic level. If you were to apply 5V to SD card connections, you would likely damage the card. It is possible to create a voltage divider between your Arduino and SD card, but this will increase the rise time when your data line is switching from LOW to HIGH. This might cause errors when communicating between the host device (Arduino) and SD card. For these reasons, it is advisable to use any type of microSD card adapter.

MicroSD adapters will mostly all consist of some sort of voltage regulator and a logic level shifter. In my adapter, there is a AMS1117 voltage regulator and a LVC125A (or SN774LVC125A) quadruple bus buffer gate with tri-state outputs.

Side note: while researching the LVC125A, I was unsure whether the LVC125A was being used as only a level shifter. Because SPI peripherals and master use a shared MISO line (peripherals communicating TO the master device), it is important that only the devices with a LOW chip select wire can communicate on the MISO line. I am unsure if the additional tri-state buffer is required on the SD converter, or if the SD card itself has a tri-state buffer built in to drive the peripheral output to HIGH Z while chip select is not enabled.

The real time clock communicates over I2C and provides the current year and time or unix time. We will store day, hour, minute, and second as a timestamp for each set of temperature measurements.

### Datalogger wiring

Wiring between our SD adapter and Arduino is easy. Remember when I mentioned there are six wires? Here they are:

|  |  |  |
| --- | --- | --- |
| Arduino |  | Adapter |
| SCK (13) | - | SCK |
| 10 | - | CS |
| MOSI (11) | - | MOSI |
| MISO (12) | - | MISO |
| 5V | - | VCC |
| GND | - | GND |

The real time clock communicate via I2C connected to SCL and SDA. For this single slave device I relied on the internal pullup resistors.

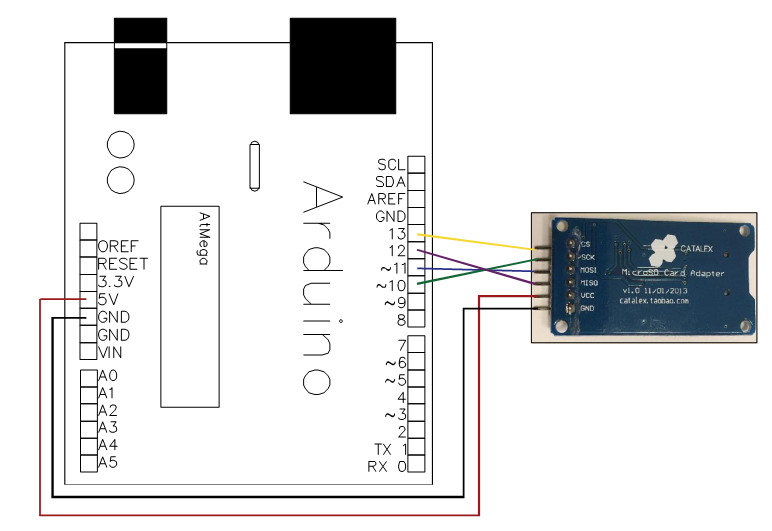


Figure - SD Adapter

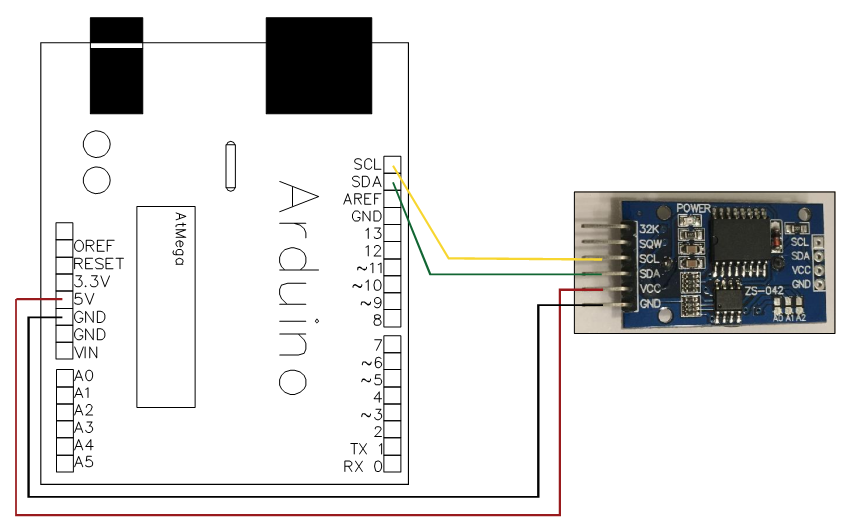


Figure – Real Time Clock

### Datalogger code

In this section we will break down code for the datalogger. We are using the SdFat library written by Bill Greiman.

Below you will see that we:

1. Define chip select output pin for our SD card
2. Give our file name
3. Initialize an object of class SdFat. This object will be used to initialize our sd card and file system
4. Initialize an object of class SdFile. This is the basic file class. We will use this object to write data to our SD card.
5. Define the data interval (how long between measurements and writing)
6. Create functions to make a header for data logged csv file, a function for writing collected data, and a function for logging time stamps

This code section is straight forward taken in pieces. The datalogging functions save information in a comma separated format, so in between each temperature entry and column header we use “file.print(“,”)”. P.S. there is no difference between calling file.write() and file.print(). File.print() and file.write() are derived from the Arduino print class, which writes ASCII characters.

You will notice that I used for loops for writing data to the SD card, and I was a little bit sloppy about my conditions. Also, after a trial I decided to separate my day of week/hour/minute/second with commas. If they are not separated, it makes our job of extracting good time information much more difficult because of the lack of leading zeros.



The next section of code is placed in our void setup() function. Here we:

1. Automate creation of file names
2. Set SPI communications to 50 MHz and test if our SD is communicating
3. Find and automate creation of an unused file name. This will be useful for preventing concatenating to existing files
4. Open our file and error check
5. Write header data

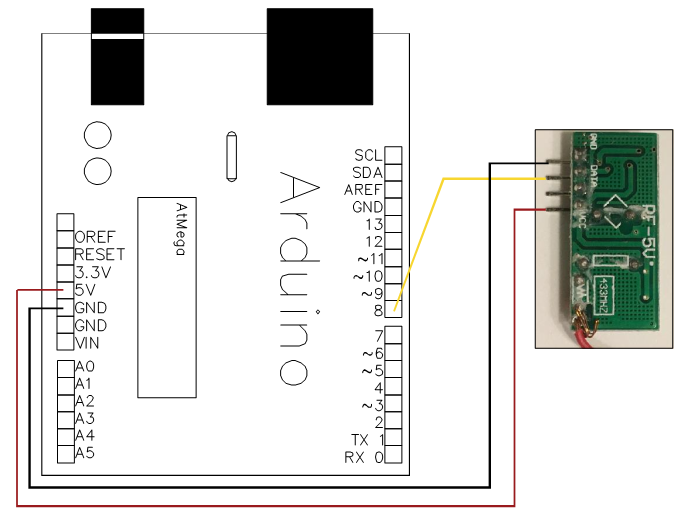


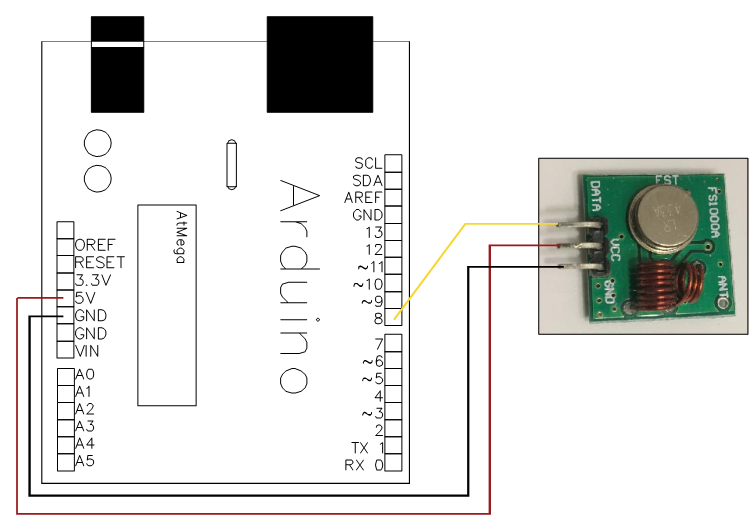
## Radio Transmitter & Receiver

We will be making use of one 433 MHz transmitter/receiver pair (FS1000A) for transmitting temperature information form a remote Arduino to the datalogging Arduino. These will be used in combination with the VirtualWire library written by Mike McCauley. VirtualWire is a library used to transmit short radio messages using Amplitdue Shift Keying (ASK).

### Radio Wiring

The wiring is simple and configurable. Connect your data line to pin 8 for both the transmitter and receiver. This pin can be changed to any digital input pin available. The receiver requires 5V and the transmitter can be 3V-12V powered. Increased voltage will increase the range, but I found that 5V worked well about 10 meters apart with some walls and couches in between. Be sure to attach a 9cm antennae to the transmitter and receiver.

Figure - Receiver

Figure - Transmitter

### Radio Code

Snippet of receiver code

The radio code is more simple than the datalogger. First, we define variables. The data pin is pin 8; we define some byte arrays for use in data reception, and finally some reception parameters. I am unsure if the VirtualWire library uses the “vw\_set\_ptt\_pin” parameter if it is not enabled, but for safety I defined its pin to six instead of the default 11.

Each bytes# variable will be used to store four sequential bytes transmitted by the remote Arduino. These four bytes represent what used to be a floating point number in the transmitter Arduino, and we will have to reconstruct four individual bytes into a single floating point number for storage.



In the next section of the receiver code, we need to define the “ReceiveData” function. It will retrieve messages stored in “buf”.

WARNING: the code below is a bad example of how to use VirtualWire. Instead of transmitting temperature readings as floating point numbers, I transmitted it byte by byte. This means that data I receive will not be in the same form as I sent it, and it needs to be converted from groups of 4 bytes to one floating number. It is important to note that VirtualWire can transmit floating point data, and there was no need to send numbers byte by byte. However, when I started this project I was using old documentation from when this library could only transmit arrays of 20 bytes each. So, in your project it will be more efficient to simply send and read floating numbers.

In my implementation, I transmitted data as unsigned 8 bit integers (see transmitter code). However, the data was originally stored as 4 byte floating point numbers. Because I was transmitting 5 temperature readings at once, the total message length to be received is 20 bytes (5 readings \* 4 bytes/reading = 20 bytes). My procedure for receiving temperature data and getting ready to save it goes as follows:

1. Receive temperature data byte-wise
2. Copy buffer data in increments of 4 bytes to a dummy variable (bytes4 etc.)
3. Use the C memcpy function to copy byte data to the variable Temp\_Store which holds floating point number. Memcpy() copies binary data from the source to destination memory blocks. The destination memory block can then be read as its intended data type which is floating point.
4. (not shown) save data in Temp\_Store to SD card



The transmitter code is simple. After constructing Temp\_Store which holds 5 floating point temperature measurements, we send that data byte-wise.



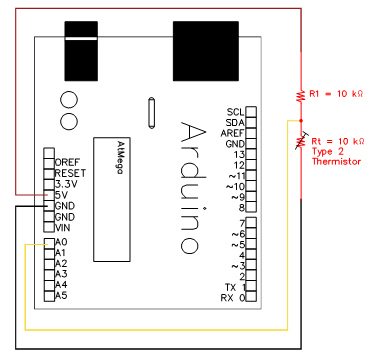
## Temperature Measurements

We are taking temperature measurements using 10 kΩ type 2 thermistors. We will set up a voltage divider and measure voltage across the thermistor. Then we will calculate thermistor resistance from the voltage reading, and finally interpolate resistance to temperature using a lookup table.

### Thermistor Wiring

We want to setup a voltage divider to measure the variable resistance of each thermistor. I used 10 kΩ and 4.7 kΩ resistors. We are measuring voltage between the thermistor and ground. We expect resistance to decrease as temperature increases (and vice versa).See the wiring diagram below.

Make wiring diagram for thermistor



Make equation for resistance

Thermistor resistance can be calculated with the equation below (Eq #). is measured thermistor voltage, is our known resistance, and is input voltage (5V).

Make equation for linear interpolation

After resistance is calculated, we can linearly interpolate between calculated resistance and lookup table values. See the next section for code implementation.

### Thermistor code

Show thermistor code snippet

First we define constants for calculating temperature. Input voltage to the voltage divider is 5V, known resistance (R1) is 10 kΩ, and temperature calculations will occur on analog pins 0-5. The variables “Temp\_Array” and “Resistance\_Array” are taken straight from a type 2 thermistor lookup table found on the internet.



Calculating temperature will be achieved with the following logic:

Read voltage across thermistor

Calculate thermistor resistance using equation #

Interpolate temperature from resistance using equation #. We do this by locating the ith index of Resistance\_Array where measured resistance is greater than the tabled resistance. Then knowing that index, we interpolate to estimate temperature.



In the main loop we:

1. Calculate temperature from thermistors on pins A0-A3
2. Read and process data received via RF from the transmitter board
3. Write a time stamp to the SD card
4. Write temperature data to the SD card



# Part 2

## Introduction

Redefine goal:

After finishing part 1, I was left with a CSV file containing 9 thermistor readings time stamped every 10 seconds. From this data, I want to:

1. Calibrate readings (discussed later)
2. Map discrete temperature readings to a dynamic, colorized space map representing my apartment
3. Combine each space map from each sample period into a video to show dynamic temperature changes over time

Each step came with special challenges which I will talk about in the following sections.

## Python Scripts

### Data Import & Calibration

After part 1 I was left with a CSV file representing 9 different time stamped temperature measurements (Table X).

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Day | Hour | Minute | Second | T0 | T1 | T2 | T3 | T4 | T5 | T6 | T7 | T8 |
| 6 | 9 | 0 | 1 | 80.16 | 79.51 | 76.77 | 78.54 | 81.38 | 80.11 | 81.01 | 80.29 | 80.65 |
| 6 | 9 | 0 | 11 | 80.16 | 79.67 | 77.08 | 79.35 | 81.19 | 80.11 | 81.19 | 80.11 | 80.65 |
| 6 | 9 | 0 | 21 | 80.16 | 79.01 | 76.61 | 77.9 | 81.19 | 80.11 | 81.01 | 80.29 | 80.83 |

Table X – Sample of temperature readings

As you can see, I chose to take measurements every 10 second. Due to uncertainties in resistance of the static resistor and thermistor, temperature measurements vary even when the probes are exposed to the same temperature media. To reduce error due to these uncertainties, I need to correct each temperature reading I take. I chose to do this by finding steady state error between a “true” media temperature, and the reported temperature.

Each thermistor was placed in a water bath of uniform temperature, and measurements were taken every 10 seconds (Table X). I chose the true temperature to be the mode reported value between all 8 thermistors. Then, I determined steady state error: where was 79.19, was the measured temperature array, and E is the steady state error matrix. Each column of E = {corresponding to a thermistor was averaged: where is the average error for the ith thermistor. Finally, I added the error vector to my measured data: to get a dataset which is compensated for steady state error.

This is the incorrect way to calibrate thermistors. Ideally, I would modify the Stein-Hart constants based on “true” temperature readings and measured resistance values. However, this method requires having a pre-calibrated thermistor, and an accurate DMM (which I do not have).

Another source of error could come from inaccurate resistor estimates. I read resistance from the resistors color bands (again, because I do not have a DMM) which may vary from their actual resistance.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Day | Hr | Min | Sec | T0 | T1 | T2 | T3 | T4 | T5 | T6 | T7 | T8 |
| 6 | 9 | 20 | 35 | 79.18 | 78.7 | 79.18 | 78.7 | 81.01 | 79.38 | 79.19 | 79.19 | 79 |
| 6 | 9 | 20 | 45 | 79.18 | 78.7 | 79.18 | 78.7 | 81.19 | 79.56 | 79.19 | 79.19 | 79.19 |
| 6 | 9 | 20 | 55 | 79.18 | 78.7 | 79.18 | 78.7 | 81.19 | 79.56 | 79.38 | 79.19 | 79.19 |

Table X – Partial dataset showing steady state thermistor readings while in a water bath

Bow is the python code used to import and calibrate collected data. The importData() function uses the pandas library to import an excel file to a dataframe object called “cal”. In the fixData() function I first remove transient data before temperature readings reached steady state. Finally I calculate the error vector that is used to correct my measured dataset.





### Temperature Mapping

Now that I have my corrected dataset, I can start mapping temperature readings to a colorized map. I created a map of my apartment using AutoCAD and a tape measure to approximate the location of important furniture and walls (Figure X).

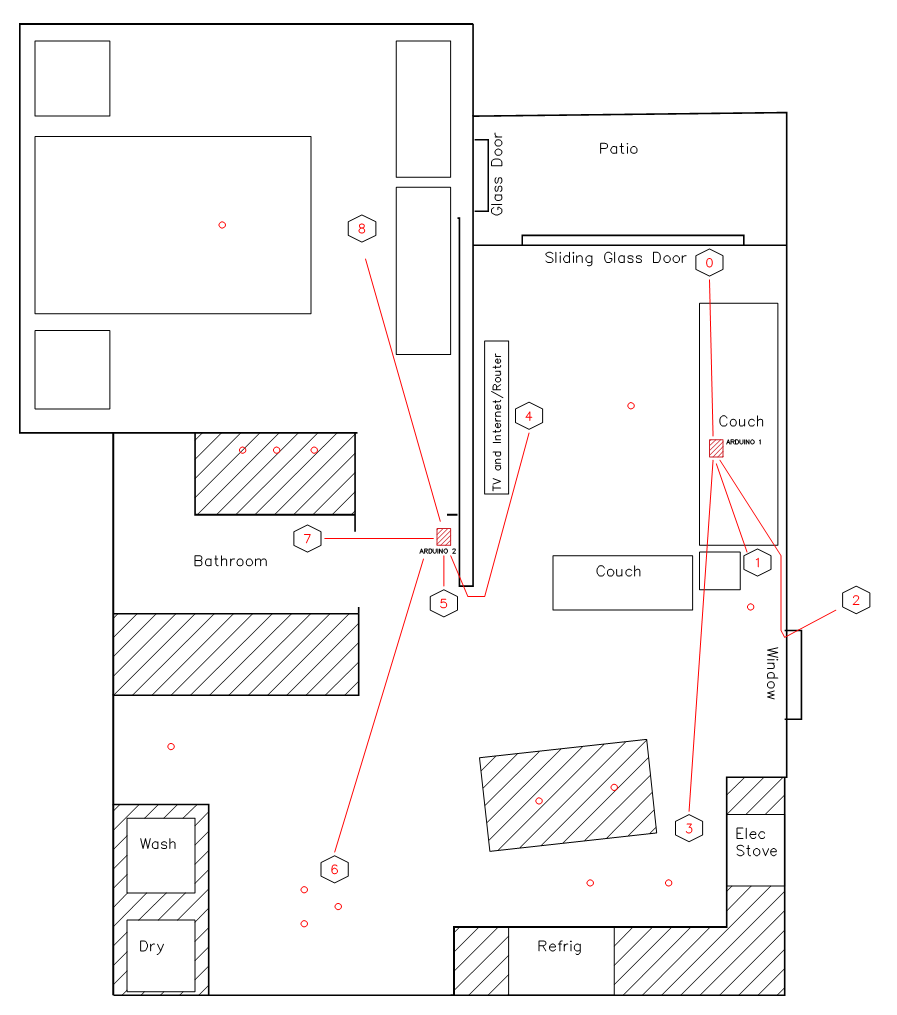


Figure X – Detailed layout of apartment

There are a total of 9 functions I used to manipulate Figure X to its final form. They are:

importImages(): imports a detailed and basic background layout. The basic layout is used to get a mask and the detailed image is used to display data



resizeImages(): changes the image size from its original 1920x1080 to a smaller 960x540 image. The image is cropped to show only important areas.



convertToGray(): changes the mask to grayscale for edge detection

1. **def** convertToGray():  #Convert to grayscale
3. **global** img1\_gray
4. **global** img1\_inv
5. ret, img1\_gray = cv2.threshold(img1,220,255,0)
6. img1\_gray = cv2.cvtColor(img1\_gray, cv2.COLOR\_BGR2GRAY)
7. img1\_inv = cv2.bitwise\_not(img1\_gray)

getMask(): a convoluted way to get an image that has white where I want to map temperatures to color (white is represented as 255 or 1 and black is represented as 0 in OpenCV images). While making this function I was learning the uses of openCV’s thresholding and bitwise functions. This process could certainly be simplified while still getting the same result.

1. **def** getMask():
2. **global** img\_contour, img1\_inv, img2\_gray, img2\_mask
4. #Find the locations inside the walls
5. img\_contour, contours, heirarchy = cv2.findContours(img1\_inv, cv2.RETR\_LIST, cv2.CHAIN\_APPROX\_SIMPLE)
7. #draw the contours, send thickness = cv2.FILLED
8. img\_contour = cv2.drawContours(img1, contours, 3, [0,0,0], -1)
10. #This will be the binary representation of the areas we will want to color in
11. #once i get the temperature data
12. img\_contour = cv2.cvtColor(img\_contour, cv2.COLOR\_BGR2GRAY)
13. ret, img1\_inv = cv2.threshold(img1\_inv, 220, 255, 0)
14. res = cv2.bitwise\_or(img\_contour, img1\_inv)
15. temperature\_mask = cv2.bitwise\_not(res) #this is what i want
17. #now to subtract the areas that have important information like the couch
18. img2\_gray = cv2.cvtColor(img2, cv2.COLOR\_BGR2GRAY)
19. img2 = cv2.cvtColor(img2, cv2.COLOR\_BGR2GRAY)
20. ret, img2 = cv2.threshold(img2, 245, 255, cv2.THRESH\_BINARY)
21. ret, img2\_mask = cv2.threshold(img2\_gray, 220, 255, 0)
22. #now to remove the detailed walls
23. img2\_mask = cv2.bitwise\_or(img\_contour, img2\_mask)
25. img2\_mask = cv2.bitwise\_and(temperature\_mask, img2\_mask)
26. img2\_mask[np.where((img2\_mask >= 1) & (img2\_mask < 255))] = 0 #final

getNodes(): lets the user select regions in the image where thermistors were physically placed. The location of the thermistors in the image is important because they are used to calculate the temperature gradient between nodes.



getPixelWeightArray(): this function will create a 3D (H x L x t) array called calc\_temp where H and L represent the height and width of the image we are mapping colors to. The third dimension, t, is 11 long and holds information corresponding to the pixel at the same index H & L of the image we are mapping colors to. In order from 0:10 the slices represent the {image mask, node 1 weight, node 1, node 2 weight, node 2, node 3 weight, node 3, weighted temperature, temperature mapped to HSV hue, HSV saturation, and HSV value}. Additional slices will be defined in the next function.



calculateDistance() – Because I only measured temperature at 9 discrete locations, I need a way to calculate temperature of each pixel between multiple nodes. This function calculates the distance between each pixel and the three closest thermistor nodes. After this, the function determines the “weight” of each node temperature on a pixel.

First, the distance between each pixel in the image and the nodes is found with where d is the distance in pixels, is pixel x location, is pixel y location, is node x location, and is node y location. The distance between the pixel and the three closest nodes are found.

After the three closest nodes are found, those three distances are mapped to three magnitudes. Each magnitude represents the nodes influence on a pixels assigned temperature. The magnitude is found with an exponential function where a, b, and c are constants to be determined and d is the node to pixel distance. Figure X shows a plot with a = 0.7 and b = {0.1:0.1:0.9}. Appropriate constants for a, b, and c were found by iteration and feelings of how much I wanted a node to influence a pixels temperature assignment given how far away the node is. Values of b = 0.4 or 0.5 felt appropriate because these values offered a tradeoff between close nodes having significant influence over temperature, while far away nodes still having slight influence. Higher values of b would causes steeper gradients between nodes of different temperatures, while low values blur temperature differences until the pixels are very close to nodes.

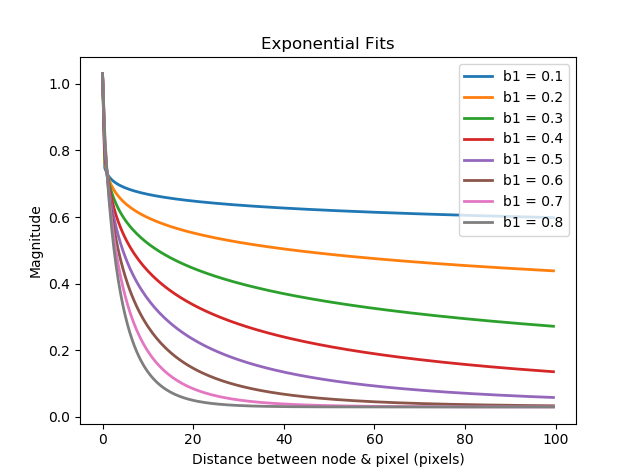


Figure X – Figure depicting iterative process of choosing weighting constants.

An individual nodes influence over pixel temperature is described with etc. where is the previously calculated magnitudes for each of the three closest nodes, and is the influence of node 1 over the temperature assigned to the pixel. Pixel temperature will be calculated later by where , , are the temperatures corresponding to the first, second, and third closest nodes.

The intended purpose of exponentially weighing a nodes influence over a pixels assigned temperature is to give the viewer the perception of a smooth gradient between spaces.



calculateTemperature2(): Assigns temperatures to each pixel based on weights from the three closest nodes. Next, it finds the Z score (number of deviations from mean in a normal distribution) for each calculated temperature point. The Z score is then mapped to a HSV hue based on a linear fit between blue and red.

First, the mean and standard deviation of measured data is calculated at a time dataTime. Then, weighted temperatures are calculated using the summation of node temperatures times node influences for each pixel: . is the resultant (H x L) matrix which contains temperatures assigned to each pixel of the original (H x L) Image X and is stored in calc\_temp[:, :, 7]. **, ,** are the influence matrices calculated previously and reside in calc\_temp[:, :, 1], calc\_temp[:, :, 3], and calc\_temp[:, :, 5] respectively. **, ,** andare matrices holding node temperatures corresponding to the 1st, 2nd, and 3rd closest nodes for each element of the image array.

A Z score is calculated for each element in **: .** Now each element in is mapped to a hue in the range of 120 to 180 which represents blue to red in OpenCV according to: which results in the linear fit . Any elements that are significantly hotter than the average will get mapped to 180 (red) or 120 (blue) if they are colder.



writeToVideo() – Iterates calculateTemperature2() to create sequential images based on sequential temperature measurements. Converts the HSV image to BGR and writes each image to the video output file using the OpenCV videoWriter object.

The calculateTemperature2() function needs to be called for each data collection period to create a sequential video. This function takes in a starting temperature period and an ending, then creates an output video of all images created for each temperature period.



## Discussion of Results

### How well can the user visualize data?

The paramount goal of this part of the project is to let the viewer see:

1. Relative temperature gradients throughout a space
2. Absolute changes in space temperature relative to previous times (aka heating up during the day)
3. Hot and cold spots

There are a few programming choices that affected how well the three goals were achieved. Among these are:

1. Color mapping
   1. Using colors as opposed to single dimensional change (light/dark)

While researching HSV coloration in OpenCV I came across a document that claimed viewers understand data best when it is represented in one dimension (such as light/dark). It is hard to say if color is one dimension. In traditional HSV coloration, there are three parameters that all contribute to the color including color wavelength, purity or how much white is contained, and color intensity. I held saturation and value constant in my application, which made it so there is only one parameter to change. It is up to the viewer to interpret what these colors mean, unless a legend is added to the figure. Even then, the viewer needs to internalize what colors mean relative to each other.

* 1. Specific colors used

I found it to be a huge challenge choosing the specific colors to map between. Intuitively, most people would agree blue is cold and red is hot. However, this results in the average ranging temperatures being purple. In this mid-range where most measurements lie, lots of meaning is lost because different hues of purple are less distinguishable. Ideally, the algorithm would map between blue and red as follows: dark blue -> light blue -> light red -> red. This requires two hue and saturation be changed simultaneously in a non-linear way.

A third option could be a curve of the form or even higher order. This type of curve would require iterating between constants to find an appropriate tradeoff between temperature change and color change (Similar to Figure X).

* 1. Mapping based on standard deviation instead of static range

My other big design decision was mapping colors based on deviations away from the mean, as opposed to a static mapping. In static mapping, colors would be translated to color based on a fit between color and some temperature boundaries. I initially tried this and found this method to be lacking because:

1. The temperature range needs to be wide enough to be including of most (or all) temperatures measured. Because temperature changes through the day depending on time and outside conditions, this led to a larger temperature range than was usually experienced at any moment. This led to the colors representing *average* differences throughout the day, as opposed to instantaneous differences across space.
2. Assigning a “low” value to blue and “high” value to red is a little arbitrary. Low or high is relative to what we are used to/surroundings. What is low in one moment may be high in another.

By using the Z score, I was able to map to colors based on what is hot or cold in the instant. A drawback to this is the large resolution mentioned previously. Large discrete measurements changes cause the sample mean and standard deviation to shift relative to what it was previously, which contributes to a “choppy” feeling to the viewer.

This issue might be fixed by applying a low pass filter to the temperature data. The filter must be chosen carefully to not limit the bandwidth of the system.

* 1. Exponential radial calculation of weighted temperatures

Exponential fit constants mentioned in Figure X above must be chosen carefully. Values of b1 close to 1 dramatize the data being represented because pixels close to temperature nodes will be heavily influenced by the node and will rapidly change to be influenced by other nodes. Conversely, values of b1 close to 0 minimize the influence of any specific node over a pixels weighted temperature. Figure X shows an example where pixels close to a node are colored differently due to a combination of all above influences (color mapping, specific color used, mapping based on deviation, and exponential weighting.)

Add Figure – showing choppy colors close to nodes

1. Displaying outside & window temperatures

Displaying outside air and window temperature would help the viewer understand time-of-day, and how temperature changes through the day. The viewer may also better understand how windows transfer heat quicker than other building materials. These ideas were not added in this project because I moved onto other projects.

### Accuracy of measurements?

This setup had two types of accuracy errors: offset and gain error. The offset error could be corrected by calibrating each thermistor to a defined standard. Gain error occurs because temperature is estimated from resistance, which varies non-linearly for thermistors.

Offset error is lager in this experiment and hurts how well the viewer understands the data. For example, Figure X shows range measurements for all data points. Measurement range stays close to 2 Deg F over the entire experiment, and I attribute this to offset error. If there were no offset error, range should be close to zero when there is no temperature deviation in a room (when there is no signal transience).

### Resolution of measurements?

The atmega328 has a 10 bit ADC converter. Using this information, I can calculate the approximate resolution of temperature measurements. The approximate temperature resolution is defined as: where is the thermistor sensitivity, is the derivative of thermistor resistance with respect to voltage, and is the ADC bit resolution. can be found by looking at the thermistor lookup table, and this value varies as temperature varies. Because our operating range is narrow, it can be approximated to be which are the temperature and resistance operating point between 81 and 79 Deg F. can be found explicitly from Equation X and is . Evaluated at our operating point of = 2.5 V, this results in 8000 Ω/V. is where n is our bit resolution. In many sensors bandwidth and electrical noise are the dominant determiners of resolution. However, in this setup electrical noise did not appear to be greater than the ADC resolution because temperature measurements were constant during steady state calibration. In order to confirm this I could measure the signal noise with an oscilloscope.

Evaluating these values gives us an approximate resolution of 0.175 Deg F, which agrees closely with our measured data where we observe that temperature measurements always seem to shift by approximately 0.19 Deg F when they do change. This means that the most discernable temperature change measurable with this setup is about ±0.19 Deg F.

It is assumed that temperature does not change significantly within ±0.19 Deg F while measurements are being made. This setup will not be able to discern true temperature changes between ±0.19 Deg F.

The system resolution will also affect how the user interprets temperature gradients. To give an intuitive understanding of the relative magnitude of resolution to the temperature changes within the room, the resolution and standard deviation of measured temperature can be compared.

Figure X shows standard deviation and range of temperature measurements across measurement periods. Visually we see that temperature deviates between 0.6 and 1 Deg F, which means resolution is of temperature deviation. This means that temperature changes may appear choppy or sudden between measurement periods. If resolution were a lower percentage of measured variance, then the viewer would experience smoother changes in temperature.

The system used has a very high precision which can be calculated as where is the value of the nth measurement and is the average value of a set of n calibration measurements (cite national instruments). This is useful for this application because the viewer is meant to see relative changes in temperature as opposed to the true temperature.

I am unsure if system resolution contributes to poor measurements of variance. I assume that the system resolution must be appropriately smaller than true variance for measured variance to reflect true variance, similar to the Nyquist measuring rule.

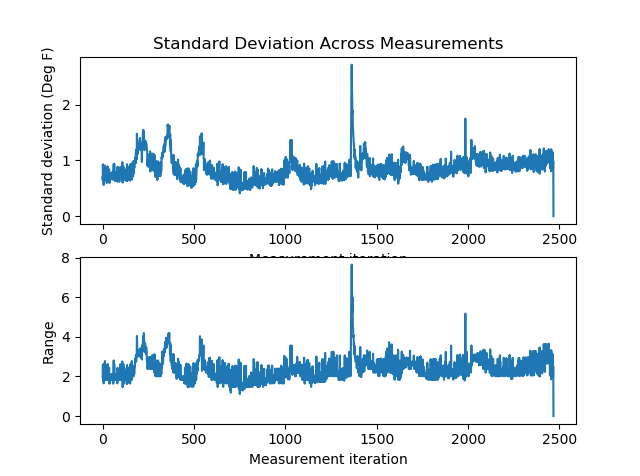


Figure X – Standard deviation and range of temperature measurements through ~2500 10 second cycles

# Project Conclusion

The project goal was to communicate data visually to a viewer. This setup was a fun experiment, but it did not communicate data as effectively as I hoped. In summary:

1. Offset error (accuracy) was large between individual thermistors,
2. sensor resolution was large compared to the “true” temperature,
3. and color mapping algorithms were lacking.

Sensors should be calibrated to a known standard; ADC resolution should be increased, and a different color mapping scheme should be developed. With money, time, material, and some matrix algebra this could all be accomplished.

There are questions I am unable to answer:

How can I un-confound “real” temperature fluctuations from sensor resolution and offset error?

There are questions I effectively answered:

How can I improve sensor/system accuracy and resolution?

What metrics can I use to relate empirical measurements with viewer experiences?

What are important metrics of instrumentation sensors?

How to empirically calculate sensor resolution?

Beginner uses for C and Ptyhon

Data visualization using standard programming libraries

Vector/matrix algebra for faster calculatinos