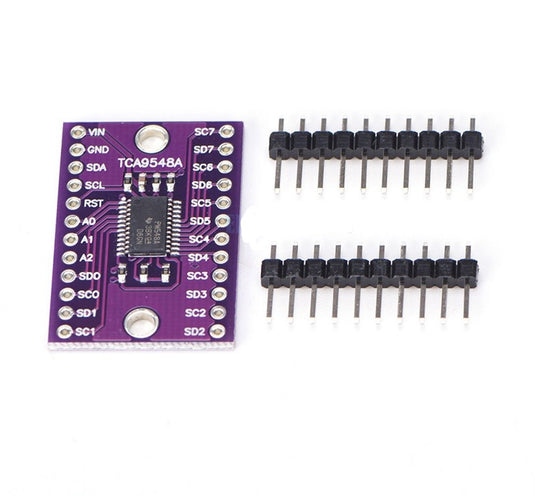
**Temperature Sensor**

|  |  |
| --- | --- |
| **Temperature Sensors** | **Specifications** |
| K Type Thermocouple | **Extra Amplifier**  MAX6675 Module  Generic 201608290937065860656  [https://robu.in/product/max6675-module-k-type- thermocouple-sensor-measure-1024-degrees-temperature/](https://robu.in/product/max6675-module-k-type-thermocouple-sensor-measure-1024-degrees-temperature/) |
| 10K NTC Thermistor | [https://robocraze.com/products/10k-thermistor-temparature-sensor-module? variant=40192469237913&currency=INR&utm\_medium=product\_sync&utm \_source =google&utm\_content=sag\_organic&utm\_campaign=sag\_organic&https:/ /robocraze.com/?utm\_source=googleads&utm\_medium=ppc&gad=1&gclid= Cj0KCQjwqs6lBhCxARIsAG8YcDgjyH6DUUuG0nHkRHh71QIqUFPsjn BMPRXw45aDuYMh4CvJM-rm0VoaAvGDEALw\_wcB](https://robocraze.com/products/10k-thermistor-temparature-sensor-module?variant=40192469237913&currency=INR&utm_medium=product_sync&utm%20_source=google&utm_content=sag_organic&utm_campaign=sag_organic&https://robocraze.com/?utm_source=googleads&utm_medium=ppc&gad=1&gclid=Cj0KCQjwqs6lBhCxARIsAG8YcDgjyH6DUUuG0nHkRHh71QIqUFPsjnBMPRXw45aDuYMh4CvJM-rm0VoaAvGDEALw_wcB)  NTC Thermistor Temperature Sensor B3950 High Precision 10K Wire Ear Fixed Head M4  <https://probots.co.in/ntc-thermistor-temperature-sensor-b3950-high-precision-10k-wire-ear-fixed-head-m4.html?gclid=Cj0KCQjwqs6lBhCxARIsAG8YcDi3B_xU2n7J7gLZSYt2jX-Uh4gAQ2b9zpAPI-4IwAdz2ay8KmqEmh4aAlryEALw_wcB> |
| TMP36  Digital | TMP36 - Temperature Sensor |
| DS18B20  Waterproof  Digital | <https://robocraze.com/products/ds18b20-waterproof-digital-thermometer-sensor-probe?currency=INR&variant=40193205928089&https://robocraze.com/?utm_source=googleads&utm_medium=ppc&campaignid=20040135259&adgroupid=&keyword=&device=c&gad=1&gclid=Cj0KCQjwqs6lBhCxARIsAG8YcDiZr2HE-781eCriJcHgnZTpegDSnUfMF9nDNVKAkjOAU02ttjSixYgaAmhLEALw_wcB>    4.7K Ohm |
| SparkFun IR Thermometer Evaluation Board - MLX90614  I2C  IR | <https://evelta.com/sparkfun-ir-thermometer-evaluation-board-mlx90614-temperature-sensor-evaluation-board/?campaignid=19631771445&adgroupid=&network=x&device=c&utm_source=19631771445&utm_medium=&gclid=Cj0KCQjwqs6lBhCxARIsAG8YcDgTfFHUWd4OgNbyAP7XuItsGY9eoazMA-3CaE6FEYAmBCLf7aDSezEaApu1EALw_wcB>  2-5 cm Distance range  <https://github.com/thecomputationguy/raspberrypi-thermal>  <https://github.com/4bitFox/MLX90640_python>  <https://www.instructables.com/Infrared-Thermal-Imaging-Camera-With-MLX90640-and-/>  <https://maker.pro/arduino/projects/diy-a-thermal-imaging-monitor> |
| AMG8833 Grid-EYE Infrared Array Breakout I2C Qwiic SparkFun  I2C  IR | <https://evelta.com/sen-14607-amg8833-grid-eye-infrared-array-breakout-i2c-qwiic-sparkfun/?campaignid=19631771445&adgroupid=&network=x&device=c&utm_source=19631771445&utm_medium=&gclid=Cj0KCQjwqs6lBhCxARIsAG8YcDjlpZL6UegYpBpKLguHMLO72oKBoAHuQr_I2kcPaslrxXPpreyIZ0oaAv1dEALw_wcB>  Distance <7m or 22.966 ft  Typical ±2.5 °C  SEN-14607 - AMG8833 Grid-EYE Infrared Array Breakout I2C Qwiic SparkFun  <https://www.hackster.io/jdanielse/amg8833-thermal-camera-fc8478?f=1>  <https://github.com/JaapDanielse/AMG8833-Thermal-Camera/wiki/Reference/>  <https://diyprojectslab.com/amg8833-thermal-camera-sensor-with-raspberry-pi/>  The AMG8833 is a thermal sensor array that provides 64 (8x8) individual thermal readings, which can be used to generate a thermal image. When applied to a plant, these readings can provide a detailed view of the plant's temperature distribution. However, determining the "exact temperature" of the whole plant body can be a bit more complex. Here are a few potential approaches:   1. **Average Temperature:** One approach is to simply average all the readings to get the mean temperature of the plant. This gives you a single number that represents the average temperature over the entire plant body, but it might not capture significant variation within the plant. For example, if one part of the plant is much warmer than the rest, the average might not accurately represent the temperature of the majority of the plant. 2. **Mode Temperature:** Another approach could be to find the mode, or the most common temperature value in your 64 readings. This would give you the temperature that is most typical for the plant, but again, it might not capture significant variations within the plant. 3. **Thermal Distribution:** Instead of trying to represent the whole plant with a single number, you might consider analyzing the full distribution of temperatures. This could involve calculating the mean, median, mode, and standard deviation of the temperatures, or even using more complex statistical measures. This approach provides a more detailed picture of the plant's temperature, but it is also more complex to analyze and interpret. 4. **Region-Specific Temperatures:** If you have knowledge about the structure of the plant, you could potentially assign specific temperature readings to specific parts of the plant, and then analyze the temperature of each part separately. For example, you might separate the leaves, stems, and flowers, and calculate separate average temperatures for each part.   import numpy as np  import matplotlib.pyplot as plt  from scipy import stats  # Let's assume you have a 2D numpy array representing temperature values captured from AMG8833  # I'll use a random array for demonstration  temperature\_array = np.random.rand(8,8) \* 30 # temperature range 0 to 30 degrees  # Calculate average and mode temperatures  average\_temp = np.mean(temperature\_array)  mode\_temp = stats.mode(temperature\_array, axis=None).mode[0]  # Print average and mode temperatures  print(f"Average temperature: {average\_temp}")  print(f"Mode temperature: {mode\_temp}")  # Plot the temperature distribution  plt.figure(figsize=(12,6))  # Plot the thermal image  plt.subplot(1, 2, 1)  plt.imshow(temperature\_array, cmap='hot', interpolation='nearest')  plt.title('Thermal Image')  plt.colorbar(label='Temperature (°C)')  # Plot the histogram  plt.subplot(1, 2, 2)  plt.hist(temperature\_array.flatten(), bins=30, color='skyblue', edgecolor='black')  plt.title('Temperature Distribution')  plt.xlabel('Temperature (°C)')  plt.ylabel('Frequency')  plt.tight\_layout()  plt.show()  # Calculate region-specific temperatures (arbitrary regions)  region1 = temperature\_array[0:4, 0:4]  region2 = temperature\_array[4:, 4:]  avg\_temp\_region1 = np.mean(region1)  avg\_temp\_region2 = np.mean(region2)  print(f"Average temperature of Region 1: {avg\_temp\_region1}")  print(f"Average temperature of Region 2: {avg\_temp\_region2}") |
| DHT22 (AM2302 Module)  For ambient Temperature | 0-100% RH  -40 to 80 degree celcius, ± 0.5°C  Sampling rate 0.5 Hz (reading 2 times in 1 second)  Generic Am2302 Dht22 Digital Temperature And Humidity Sensor Module 3 |

TCA9458A 1-to-8 I2C Multiplexer



**Positioning of Temperature Sensors Around the Plant**

|  |
| --- |
| **Leaf:** MLX90614 SparkFun IR Thermometer - A non-contact infrared sensor that's ideal for the delicate structure of leaves. |
| **Stem:** LM36, 10K NTC Thermistor, and K-type Thermocouple - A combination of contact sensors for the sturdier stem. |
| **Whole body:** AMG8833 Grid-EYE Infrared Array - A sensor that can generate a 2D thermal image, allowing you to monitor the temperature distribution of the entire plant. |
| **Soil:** DS18B20 - A digital temperature sensor to monitor soil temperature, which is a critical factor in plant health. |
| **Ambient:** DHT22 - A sensor for measuring ambient temperature and humidity, both of which can have a significant impact on plant health. |

To determine whether you need an I2C multiplexer like the TCA9548A, it's essential to know if any of your sensors share the same I2C address. The I2C protocol is a two-wire serial communication protocol where each device has a unique address. However, manufacturers sometimes use the same address for different devices, which means that two devices can't be used on the same I2C bus. In this case, you'd need an I2C multiplexer to allow the microcontroller to communicate with multiple devices that share the same address.

In your list:

* The MLX90614 and AMG8833 use the I2C protocol.
* The DS18B20 uses the 1-Wire protocol.
* The DHT22 uses a custom single-wire protocol.
* The LM36, 10K NTC Thermistor, and K-type Thermocouple are analog devices and don't use any digital communication protocol.

Looking at the I2C addresses:

* The MLX90614 has a factory pre-set I2C address of 0x5A.
* The AMG8833 has an I2C address that can be either 0x68 or 0x69, depending on the state of the AD\_SELECT pin.

Since the two I2C sensors don't share the same address, you don't need an I2C multiplexer for this particular set of sensors.

Now, let's discuss the Arduino code for reading these sensors. I'll provide a brief outline for each sensor, which you can use as a starting point:

1. MLX90614: You can use the Adafruit MLX90614 library to read the temperature from this sensor.
2. AMG8833: The Adafruit AMG88xx library can be used to read the 2D temperature array.
3. DS18B20: The DallasTemperature library is typically used with this sensor.
4. DHT22: The DHT library from Adafruit is a common choice for reading temperature and humidity.
5. LM36, 10K NTC Thermistor, and K-type Thermocouple: These are analog sensors and can be read directly using the analogRead() function in Arduino. Note that the K-type Thermocouple will need a signal conditioning circuit or module (like the MAX6675 or MAX31855) to convert the thermocouple voltage to a digital signal.

Here's a basic outline of how the code might look. Note that this is pseudo-code and will not compile or run as-is. You need to fill in the specific functions and logic based on the actual libraries you use:

|  |
| --- |
| // Importing necessary libraries  #include <Wire.h>  #include <Adafruit\_MLX90614.h>  #include <Adafruit\_AMG88xx.h>  #include <OneWire.h>  #include <DallasTemperature.h>  #include <Adafruit\_Sensor.h>  #include <DHT.h>  #include <Adafruit\_MAX31855.h>  #define ONE\_WIRE\_BUS 2 // DS18B20 on arduino pin2  #define DHTPIN 3 // DHT22 on arduino pin3  // Setup a oneWire instance to communicate with any OneWire devices  OneWire oneWire(ONE\_WIRE\_BUS);  // Pass our oneWire reference to Dallas Temperature sensor  DallasTemperature sensors(&oneWire);  // Initialize DHT sensor for normal 16mhz Arduino  #define DHTTYPE DHT22  DHT dht(DHTPIN, DHTTYPE);  // Initialize the MLX90614  Adafruit\_MLX90614 mlx = Adafruit\_MLX90614();  // Initialize AMG88xx  Adafruit\_AMG88xx amg;  #define DO 4  #define CS 5  #define CLK 6  Adafruit\_MAX31855 thermocouple(CLK, CS, DO);  #define LM36\_PIN A0 // Connect LM36 to Analog pin A0  #define NTC\_PIN A1 // Connect 10K NTC Thermistor to Analog pin A1  void setup() {  Serial.begin(9600);    // Start the MLX90614  mlx.begin();    // Start the AMG88xx  amg.begin();    // Start the DS18B20  sensors.begin();    // Start the DHT22  dht.begin();  }  void loop() {  // Getting temperature from MLX90614  Serial.print("Leaf Temperature = ");  Serial.println(mlx.readObjectTempC());  // Getting temperature from AMG88xx  float pixels[AMG88xx\_PIXEL\_ARRAY\_SIZE];  amg.readPixels(pixels);  Serial.println("Whole plant body temperature distribution:");  for(int i=1; i<=AMG88xx\_PIXEL\_ARRAY\_SIZE; i++){  Serial.print(pixels[i-1]);  Serial.print(", ");  if(i%8==0) Serial.println();  }  // Getting temperature from DS18B20  sensors.requestTemperatures();  Serial.print("Soil Temperature = ");  Serial.println(sensors.getTempCByIndex(0));  // Getting temperature and humidity from DHT22  float h = dht.readHumidity();  float t = dht.readTemperature();  // Check if any reads failed and exit early (to try again).  if (isnan(h) || isnan(t)) {  Serial.println("Failed to read from DHT sensor!");  return;  }  Serial.print("Ambient Humidity = ");  Serial.println(h);  Serial.print("Ambient Temperature = ");  Serial.println(t);  // Read the value from the LM36  int rawLM36 = analogRead(LM36\_PIN);  float voltageLM36 = (rawLM36 / 1024.0) \* 5.0; // Convert to voltage  float tempLM36 = voltageLM36 \* 100.0; // Convert to temperature (Celsius)  // Read the value from the NTC thermistor  int rawNTC = analogRead(NTC\_PIN);  // You need a formula to convert the raw NTC value to temperature, which depends on the specifics of the NTC thermistor  // Read the value from the K-type thermocouple  double tempK = thermocouple.readCelsius();  // Print the temperatures  Serial.print("Stem Temperature (LM36) = ");  Serial.println(tempLM36);  Serial.print("Stem Temperature (NTC) = ");  Serial.println(rawNTC); // You need to replace this with the actual temperature once you have the conversion formula  Serial.print("Stem Temperature (K-Type Thermocouple) = ");  Serial.println(tempK);  delay(2000); // Wait a bit before scanning again  } |

Please note: In this code, the NTC thermistor reading is still a raw analog value. To convert this raw value to a temperature, you'll need to use a specific conversion formula which depends on the characteristics of the thermistor and how it's wired in your circuit. If you need help with this, please provide more details about your NTC thermistor and how it's wired.

**Arduino Boards**

* **The Arduino Mega 2560 has 54 digital I/O pins (of which 15 can be used as PWM outputs), 16 analog inputs, 4 UARTs (hardware serial ports), and a larger 256 KB of Flash memory for storing your program. It operates at 5V, which is compatible with most sensors.**
* ***The Arduino Due operates at 3.3V and has a more powerful ARM Cortex-M3 CPU. It also has more I/O pins and two DACs (digital-to-analog converters), which might be useful for some projects. However, it's important to note that some 5V sensors might require level shifting to work with the Due.***
* The Arduino MKR WiFi 1010, Arduino Nano 33 IoT, and Arduino RP2040 all have sufficient capabilities to run the code and manage the sensors you have mentioned, but there are a few important factors to consider:

Digital and Analog Pins: All of these boards have enough digital and analog I/O pins to interface with the sensors you've mentioned.

Operating Voltage: The MKR WiFi 1010 and Nano 33 IoT operate at 3.3V, whereas most Arduino boards operate at 5V. Some sensors might not be compatible with 3.3V and may require a level shifter. Please check each sensor's datasheet for voltage levels.

Memory: Each of these boards should have enough memory to handle the libraries and the sketch you're using.

Connectivity: Both the MKR WiFi 1010 and Nano 33 IoT have built-in WiFi, which could be a benefit if you want to send your sensor data over the internet.

Processor Power: All three boards are significantly more powerful than a basic Arduino Uno, so they should be able to handle the processing needs of your project.

In summary, all three boards you mentioned should work for your project, but you'll need to carefully check the voltage requirements of your sensors. If any of your sensors require 5V, you'll need to add a level shifter when using the MKR WiFi 1010 or Nano 33 IoT.

**Print all 6 sensor data on a same graph (except AMG8833 Grid-EYE). Mention legend for different sensor data (use different color) in real-time. Plot 6 different graphs on 6 subplots as pair of DHT22 and one of the 6 sensors (except AMG8833 Grid-EYE) . Plot an individual graph to only show the AMG8833 Grid-EYE data in real-time.**

ChatGPT

Your requirement involves creating several real-time plots:

A single plot with 6 different sensor data streams, with different colors and a legend.

6 subplots, each showing a pair of sensor data from DHT22 and one of the other 6 sensors.

An individual plot for the AMG8833 Grid-EYE data.

It's important to note that plotting this many graphs in real-time may be resource-intensive, and you may experience performance issues depending on the capabilities of your system.

Also, the AMG8833 Grid-EYE sensor data is a 2D array representing a thermal image, which is different from the other sensor data. It's typically visualized as a heatmap, not a line graph.

Here's a Python script that should achieve what you're asking for. This script assumes that the Arduino sends the sensor data as lines of text, with each line containing the sensor name and the reading separated by ' = ', similar to your previous Arduino code:

|  |
| --- |
| import serial  import matplotlib.pyplot as plt  # Open the serial port  ser = serial.Serial('/dev/ttyACM0', 9600)  # Set up the figure and subplots  plt.ion() # enable interactive mode  fig, axs = plt.subplots(8, 1, figsize=(10, 20)) # 8 subplots: 1 combined, 6 pairs, 1 Grid-EYE  fig.tight\_layout(pad=3.0)  # Create dictionaries to hold our data arrays and plot lines for each sensor  sensor\_data = {}  lines = {}  # Colors for each sensor  colors = ['b', 'g', 'r', 'c', 'm', 'y', 'k']  # Function to update plot with new data  def update\_plot(ax, line, data):  line.set\_ydata(data)  line.set\_xdata(range(len(data)))  ax.relim()  ax.autoscale\_view()  plt.draw()  plt.pause(0.01)  while True:  if ser.in\_waiting > 0:  line\_str = ser.readline().decode('utf-8').rstrip()  print(line\_str)    # Parse the line as a sensor name and reading  sensor\_name, reading\_str = line\_str.split(' = ')  reading = float(reading\_str)  # If this is a new sensor, initialize its data array and plot line  if sensor\_name not in sensor\_data:  sensor\_data[sensor\_name] = []  lines[sensor\_name], = axs[0].plot([], [], color=colors[len(sensor\_data)], label=sensor\_name) # combined plot  axs[0].legend(loc='upper left')    if sensor\_name != 'AMG8833 Grid-EYE': # Exclude Grid-EYE from pair plots  i = len(sensor\_data) # subplot index  lines[sensor\_name + '\_pair'], = axs[i].plot([], [], color=colors[i-1], label=sensor\_name) # pair plot  lines['DHT22\_pair'], = axs[i].plot([], [], color=colors[i], label='DHT22') # pair plot  axs[i].legend(loc='upper left')  # Add the new reading to the data array  sensor\_data[sensor\_name].append(reading)    # Update the plots for this sensor  update\_plot(axs[0], lines[sensor\_name], sensor\_data[sensor\_name])    if sensor\_name != 'AMG8833 Grid-EYE': # Exclude Grid-EYE from pair plots  i = list(sensor\_data.keys()).index(sensor\_name) # subplot index  update\_plot(axs[i], lines[sensor\_name + '\_pair'], sensor\_data[sensor\_name])  if 'DHT22' in sensor\_data: # Only update DHT22 pair plot if DHT22 data is available  update\_plot(axs[i], lines['DHT22\_pair'], sensor\_data['DHT22'])  # Update Grid-EYE plot separately (assuming it's a 1D array for simplicity)  if sensor\_name == 'AMG8833 Grid-EYE':  axs[7].imshow([sensor\_data[sensor\_name]], cmap='hot', interpolation='nearest') # heatmap plot  axs[7].set\_title('AMG8833 Grid-EYE')  plt.draw()  plt.pause(0.01) |

This script creates a total of 8 subplots: one for the combined sensor data, six for the sensor pairs with DHT22, and one for the Grid-EYE data. For simplicity, the Grid-EYE data is plotted as a 1D heatmap, but in reality, it should be a 2D heatmap representing an 8x8 grid of temperature readings. This will require a different format for the Grid-EYE data and additional processing to convert the data into a 2D array.

Explain the above code

ChatGPT

Sure, here's an explanation of the Python script:

1. The script starts by importing the necessary libraries (serial for serial communication with the Arduino and matplotlib.pyplot for creating the graphs), setting up a connection to the serial port where the Arduino is connected, and enabling interactive mode for pyplot to allow for real-time updates of the plots.
2. It then sets up a figure with 8 subplots: one for the combined sensor data, six for the sensor pairs with DHT22, and one for the Grid-EYE data. figsize=(10, 20) sets the size of the entire figure, and fig.tight\_layout(pad=3.0) adds padding between the subplots for better visibility of the axes and titles.
3. Two dictionaries are created to store the data arrays (sensor\_data) and line objects (lines) for each sensor. The line objects are created by calling plot() with an empty data set and are used to update the plot data later.
4. A list of colors (colors) is defined to be used for the different sensors in the plots.
5. A function update\_plot() is defined to update a plot with new data. It sets the y-data and x-data of the line object, recalculates the limits of the axes (ax.relim()), rescales the axes to fit the new data (ax.autoscale\_view()), and redraws the figure with the updated data (plt.draw() and plt.pause(0.01)).
6. The script then enters an infinite loop where it continuously checks for new data from the Arduino (if ser.in\_waiting > 0:). If new data is available, it reads the data as a line of text, decodes it from bytes to a string, and removes the trailing newline character.
7. The script expects each line of text to be in the format "sensor\_name = reading", so it splits the line into a sensor name and a reading string, then converts the reading string to a float.
8. If this is the first reading for a sensor, the script initializes an empty data array for the sensor and creates a new line in the combined plot and a new pair plot (unless it's the Grid-EYE sensor). It uses different colors for the lines based on the order of the sensors, and it sets a legend for each plot.
9. The script then adds the new reading to the data array for the sensor and updates the combined plot and the pair plot with the new data.
10. For the Grid-EYE sensor, the script updates a separate heatmap plot with the new data. For simplicity, the Grid-EYE data is treated as a 1D array in this script, but in reality, you would need to convert the data into a 2D array to represent the 8x8 grid of temperature readings from the Grid-EYE sensor.

This script allows you to visualize the sensor data in real-time as it's being sent from the Arduino to the Raspberry Pi. By plotting the data, you can see trends and patterns that might not be apparent from the raw numbers.

The code you've provided already includes Serial.print() commands, which sends data over USB to a connected device. Any device connected to the Arduino via USB, such as a Raspberry Pi, will be able to read this data from its corresponding serial port.

On the Raspberry Pi, you can use a Python script to read this data. Here's a simple script that reads data from the serial port and prints it out:

python

Copy code

import serial

# Open the serial port

ser = serial.Serial('/dev/ttyACM0', 9600)

while True:

if ser.in\_waiting > 0:

line = ser.readline().decode('utf-8').rstrip()

print(line)

In this Python script, replace '/dev/ttyACM0' with the correct serial port for your setup. You can find the correct port by running ls /dev/tty\* in the terminal on your Raspberry Pi and looking for a device that looks like /dev/ttyACM0 or /dev/ttyUSB0.

This script will continuously read lines of text from the serial port and print them out. Each line will be one sensor reading from the Arduino.

Remember to install the pyserial library on your Raspberry Pi before running the script. You can install it with the command pip install pyserial.

User

if arduino is not printing on serial monitor. then how the sensor data from arduino will be sent to raspberry pi?

ChatGPT

In order for the Arduino to send data to the Raspberry Pi over USB, the data must be printed to the Serial port. This is typically done using the Serial.print() or Serial.println() functions in the Arduino code. If nothing is being printed to the Serial port, then no data will be sent to the Raspberry Pi.

In the Arduino code you provided, data from the sensors is being printed to the Serial port. This means that the Raspberry Pi should be able to read this data from its corresponding serial port when the two devices are connected via USB.

If you're not seeing any data in the Arduino Serial Monitor or on the Raspberry Pi, you should check the following:

Make sure that the Arduino code is correctly reading from the sensors and printing to the Serial port. You can test this by connecting the Arduino to a computer and using the Arduino IDE's Serial Monitor to view the output.

Check the baud rate settings. The baud rate set in the Arduino code (in the Serial.begin() function) must match the baud rate used by the Raspberry Pi when opening the serial port.

Ensure the Arduino is properly connected to the Raspberry Pi via USB, and that the Raspberry Pi is reading from the correct serial port.

If you're using the Raspberry Pi's GPIO pins for serial communication instead of USB, ensure that the RX and TX pins are connected correctly.

Remember that for the Raspberry Pi to read data from the Arduino over serial, the Arduino must be actively sending data over serial. This is typically done by printing to the Serial port in the Arduino code. If no data is being sent, the Raspberry Pi will not be able to read anything.

User

Improve following code on raspberry pi where the data will be plotted in real time on a graph:

import serial

# Open the serial port (this depends on your setup)

ser = serial.Serial('/dev/ttyACM0', 9600)

while True:

if ser.in\_waiting > 0:

line = ser.readline().decode('utf-8').rstrip()

print(line)

ChatGPT

In order to plot the data in real time, you can use libraries such as matplotlib. However, because this requires a graphical interface, it's not typically done directly on a Raspberry Pi unless it's running a desktop environment.

Here's an example of how you could modify your script to plot the data in real time. This example assumes that the Arduino is sending a single floating point number per line. If the Arduino is sending multiple values per line or in a different format, you'll need to adjust the parsing of the line variable accordingly.

python

Copy code

import serial

import matplotlib.pyplot as plt

# Open the serial port

ser = serial.Serial('/dev/ttyACM0', 9600)

# Set up the figure and subplot

plt.ion() # enable interactive mode

fig = plt.figure()

ax = fig.add\_subplot(111)

ax.set\_ylabel('Sensor Reading')

ax.set\_xlabel('Time (seconds)')

data = []

line, = ax.plot(data)

while True:

if ser.in\_waiting > 0:

line\_str = ser.readline().decode('utf-8').rstrip()

print(line\_str)

# Parse the line as a float and append it to the data list

data.append(float(line\_str))

# Update the data and redraw the plot

line.set\_ydata(data)

line.set\_xdata(range(len(data)))

ax.relim()

ax.autoscale\_view()

plt.draw()

plt.pause(0.01)

Remember to run this script in an environment that can display the plot, such as a Jupyter notebook or a Python IDE with a graphical interface. If you're running this on a Raspberry Pi, the Pi will need to be running a desktop environment and you'll need to be running the script from a terminal within that environment.

This script will continuously update the plot as new data is read from the serial port. If you want to plot multiple data series, you would need to adjust this to parse the incoming data accordingly and maintain separate lists for each series.

User

display all sensor data

ChatGPT

Displaying all the sensor data on a real-time graph requires us to handle multiple data streams and plot them separately. The key is to parse the data correctly based on the format in which Arduino sends the data.

Assuming the Arduino sends a single line for each sensor reading and the line contains the sensor name followed by the reading (for example: Leaf Temperature = 24.6), we can parse and plot the data for each sensor as follows:

python

Copy code

import serial

import matplotlib.pyplot as plt

# Open the serial port

ser = serial.Serial('/dev/ttyACM0', 9600)

# Set up the figure and subplots

plt.ion() # enable interactive mode

fig, axs = plt.subplots(7, 1) # Assuming there are 7 different sensor data streams

fig.tight\_layout(pad=3.0)

# Create a dictionary to hold our data arrays and plot lines for each sensor

sensor\_data = {}

lines = {}

# Function to update plot with new data

def update\_plot(ax, line, data):

line.set\_ydata(data)

line.set\_xdata(range(len(data)))

ax.relim()

ax.autoscale\_view()

plt.draw()

plt.pause(0.01)

while True:

if ser.in\_waiting > 0:

line\_str = ser.readline().decode('utf-8').rstrip()

print(line\_str)

# Parse the line as a sensor name and reading

sensor\_name, reading\_str = line\_str.split(' = ')

reading = float(reading\_str)

# If this is a new sensor, initialize its data array and plot line

if sensor\_name not in sensor\_data:

sensor\_data[sensor\_name] = []

lines[sensor\_name], = axs[len(sensor\_data)-1].plot([], [])

axs[len(sensor\_data)-1].set\_ylabel(sensor\_name)

# Add the new reading to the data array

sensor\_data[sensor\_name].append(reading)

# Update the plot for this sensor

update\_plot(axs[len(sensor\_data)-1], lines[sensor\_name], sensor\_data[sensor\_name])

This script creates a new subplot for each sensor data stream. The sensor name is used as the y-label for each subplot. The sensor data is stored in a dictionary, with the sensor name as the key and the data array as the value.

Please adjust this script according to your exact needs and the format in which Arduino sends the data. Also, please note that the number 7 in fig, axs = plt.subplots(7, 1) should be replaced with the actual number of sensor data streams.

User

Print all 6 sensor data on a same graph (except AMG8833 Grid-EYE). Mention legend for different sensor data (use different color) in real-time. Plot 6 different graphs on 6 subplots as pair of DHT22 and one of the 6 sensors (except AMG8833 Grid-EYE) . Plot an individual graph to only show the AMG8833 Grid-EYE data in real-time.

ChatGPT

Your requirement involves creating several real-time plots:

A single plot with 6 different sensor data streams, with different colors and a legend.

6 subplots, each showing a pair of sensor data from DHT22 and one of the other 6 sensors.

An individual plot for the AMG8833 Grid-EYE data.

It's important to note that plotting this many graphs in real-time may be resource-intensive, and you may experience performance issues depending on the capabilities of your system.

Also, the AMG8833 Grid-EYE sensor data is a 2D array representing a thermal image, which is different from the other sensor data. It's typically visualized as a heatmap, not a line graph.

Here's a Python script that should achieve what you're asking for. This script assumes that the Arduino sends the sensor data as lines of text, with each line containing the sensor name and the reading separated by ' = ', similar to your previous Arduino code:

python

Copy code

import serial

import matplotlib.pyplot as plt

# Open the serial port

ser = serial.Serial('/dev/ttyACM0', 9600)

# Set up the figure and subplots

plt.ion() # enable interactive mode

fig, axs = plt.subplots(8, 1, figsize=(10, 20)) # 8 subplots: 1 combined, 6 pairs, 1 Grid-EYE

fig.tight\_layout(pad=3.0)

# Create dictionaries to hold our data arrays and plot lines for each sensor

sensor\_data = {}

lines = {}

# Colors for each sensor

colors = ['b', 'g', 'r', 'c', 'm', 'y', 'k']

# Function to update plot with new data

def update\_plot(ax, line, data):

line.set\_ydata(data)

line.set\_xdata(range(len(data)))

ax.relim()

ax.autoscale\_view()

plt.draw()

plt.pause(0.01)

while True:

if ser.in\_waiting > 0:

line\_str = ser.readline().decode('utf-8').rstrip()

print(line\_str)

# Parse the line as a sensor name and reading

sensor\_name, reading\_str = line\_str.split(' = ')

reading = float(reading\_str)

# If this is a new sensor, initialize its data array and plot line

if sensor\_name not in sensor\_data:

sensor\_data[sensor\_name] = []

lines[sensor\_name], = axs[0].plot([], [], color=colors[len(sensor\_data)], label=sensor\_name) # combined plot

axs[0].legend(loc='upper left')

if sensor\_name != 'AMG8833 Grid-EYE': # Exclude Grid-EYE from pair plots

i = len(sensor\_data) # subplot index

lines[sensor\_name + '\_pair'], = axs[i].plot([], [], color=colors[i-1], label=sensor\_name) # pair plot

lines['DHT22\_pair'], = axs[i].plot([], [], color=colors[i], label='DHT22') # pair plot

axs[i].legend(loc='upper left')

# Add the new reading to the data array

sensor\_data[sensor\_name].append(reading)

# Update the plots for this sensor

update\_plot(axs[0], lines[sensor\_name], sensor\_data[sensor\_name])

if sensor\_name != 'AMG8833 Grid-EYE': # Exclude Grid-EYE from pair plots

i = list(sensor\_data.keys()).index(sensor\_name) # subplot index

update\_plot(axs[i], lines[sensor\_name + '\_pair'], sensor\_data[sensor\_name])

if 'DHT22' in sensor\_data: # Only update DHT22 pair plot if DHT22 data is available

update\_plot(axs[i], lines['DHT22\_pair'], sensor\_data['DHT22'])

# Update Grid-EYE plot separately (assuming it's a 1D array for simplicity)

if sensor\_name == 'AMG8833 Grid-EYE':

axs[7].imshow([sensor\_data[sensor\_name]], cmap='hot', interpolation='nearest') # heatmap plot

axs[7].set\_title('AMG8833 Grid-EYE')

plt.draw()

plt.pause(0.01)

This script creates a total of 8 subplots: one for the combined sensor data, six for the sensor pairs with DHT22, and one for the Grid-EYE data. For simplicity, the Grid-EYE data is plotted as a 1D heatmap, but in reality, it should be a 2D heatmap representing an 8x8 grid of temperature readings. This will require a different format for the Grid-EYE data and additional processing to convert the data into a 2D array.

User

Explain the above code

ChatGPT

Sure, here's an explanation of the Python script:

The script starts by importing the necessary libraries (serial for serial communication with the Arduino and matplotlib.pyplot for creating the graphs), setting up a connection to the serial port where the Arduino is connected, and enabling interactive mode for pyplot to allow for real-time updates of the plots.

It then sets up a figure with 8 subplots: one for the combined sensor data, six for the sensor pairs with DHT22, and one for the Grid-EYE data. figsize=(10, 20) sets the size of the entire figure, and fig.tight\_layout(pad=3.0) adds padding between the subplots for better visibility of the axes and titles.

Two dictionaries are created to store the data arrays (sensor\_data) and line objects (lines) for each sensor. The line objects are created by calling plot() with an empty data set and are used to update the plot data later.

A list of colors (colors) is defined to be used for the different sensors in the plots.

A function update\_plot() is defined to update a plot with new data. It sets the y-data and x-data of the line object, recalculates the limits of the axes (ax.relim()), rescales the axes to fit the new data (ax.autoscale\_view()), and redraws the figure with the updated data (plt.draw() and plt.pause(0.01)).

The script then enters an infinite loop where it continuously checks for new data from the Arduino (if ser.in\_waiting > 0:). If new data is available, it reads the data as a line of text, decodes it from bytes to a string, and removes the trailing newline character.

The script expects each line of text to be in the format "sensor\_name = reading", so it splits the line into a sensor name and a reading string, then converts the reading string to a float.

If this is the first reading for a sensor, the script initializes an empty data array for the sensor and creates a new line in the combined plot and a new pair plot (unless it's the Grid-EYE sensor). It uses different colors for the lines based on the order of the sensors, and it sets a legend for each plot.

The script then adds the new reading to the data array for the sensor and updates the combined plot and the pair plot with the new data.

For the Grid-EYE sensor, the script updates a separate heatmap plot with the new data. For simplicity, the Grid-EYE data is treated as a 1D array in this script, but in reality, you would need to convert the data into a 2D array to represent the 8x8 grid of temperature readings from the Grid-EYE sensor.

This script allows you to visualize the sensor data in real-time as it's being sent from the Arduino to the Raspberry Pi. By plotting the data, you can see trends and patterns that might not be apparent from the raw numbers.

User

In the following code, can you include descriptive statistics in real-time to measure min, max, mean, standard deviation, correlation plot with respect to DHT22? Also, include a CSV file to store all the sensor values except AMG8833 Grid-EYE

ChatGPT

Certainly, we can use the pandas and numpy libraries to calculate descriptive statistics in real-time, and use pandas to store the data in a CSV file. Here's how you can modify the previous script to do this:

python

Copy code

import serial

import matplotlib.pyplot as plt

import pandas as pd

import numpy as np

# Open the serial port

ser = serial.Serial('/dev/ttyACM0', 9600)

# Set up the figure and subplots

plt.ion() # enable interactive mode

fig, axs = plt.subplots(8, 1, figsize=(10, 20)) # 8 subplots: 1 combined, 6 pairs, 1 Grid-EYE

fig.tight\_layout(pad=3.0)

# Create dictionaries to hold our data arrays and plot lines for each sensor

sensor\_data = {}

lines = {}

data\_df = pd.DataFrame() # DataFrame to store all sensor data

# Colors for each sensor

colors = ['b', 'g', 'r', 'c', 'm', 'y', 'k']

# Function to update plot with new data

def update\_plot(ax, line, data):

line.set\_ydata(data)

line.set\_xdata(range(len(data)))

ax.relim()

ax.autoscale\_view()

plt.draw()

plt.pause(0.01)

while True:

if ser.in\_waiting > 0:

line\_str = ser.readline().decode('utf-8').rstrip()

print(line\_str)

# Parse the line as a sensor name and reading

sensor\_name, reading\_str = line\_str.split(' = ')

reading = float(reading\_str)

# If this is a new sensor, initialize its data array and plot line

if sensor\_name not in sensor\_data:

sensor\_data[sensor\_name] = []

lines[sensor\_name], = axs[0].plot([], [], color=colors[len(sensor\_data)], label=sensor\_name) # combined plot

axs[0].legend(loc='upper left')

if sensor\_name != 'AMG8833 Grid-EYE': # Exclude Grid-EYE from pair plots

i = len(sensor\_data) # subplot index

lines[sensor\_name + '\_pair'], = axs[i].plot([], [], color=colors[i-1], label=sensor\_name) # pair plot

lines['DHT22\_pair'], = axs[i].plot([], [], color=colors[i], label='DHT22') # pair plot

axs[i].legend(loc='upper left')

# Add the new reading to the data array

sensor\_data[sensor\_name].append(reading)

# Update the plots for this sensor

update\_plot(axs[0], lines[sensor\_name], sensor\_data[sensor\_name])

if sensor\_name != 'AMG8833 Grid-EYE': # Exclude Grid-EYE from pair plots

i = list(sensor\_data.keys()).index(sensor\_name) # subplot index

update\_plot(axs[i], lines[sensor\_name + '\_pair'], sensor\_data[sensor\_name])

if 'DHT22' in sensor\_data: # Only update DHT22 pair plot if DHT22 data is available

update\_plot(axs[i], lines['DHT22\_pair'], sensor\_data['DHT22'])

# Update the DataFrame and calculate descriptive statistics

data\_df = pd.DataFrame(sensor\_data)

print('--- Descriptive statistics ---')

print('Min:', np.min(data\_df[sensor\_name]))

print('Max:', np.max(data\_df[sensor\_name]))

print('Mean:', np.mean(data\_df[sensor\_name]))

print('Standard deviation:', np.std(data\_df[sensor\_name]))

print('Correlation with DHT22:', data\_df[sensor\_name].corr(data\_df['DHT22']))

print()

# Update Grid-EYE plot separately (assuming it's a 1D array for simplicity)

if sensor\_name == 'AMG8833 Grid-EYE':

axs[7].imshow([sensor\_data[sensor\_name]], cmap='hot', interpolation='nearest') # heatmap plot

axs[7].set\_title('AMG8833 Grid-EYE')

plt.draw()

plt.pause(0.01)

# Save the data to a CSV file every 100 iterations

if len(data\_df) % 100 == 0:

data\_df.to\_csv('sensor\_data.csv', index=False)

This script includes a DataFrame (data\_df) to store the sensor data. Whenever a new reading is received, it's added to the DataFrame, and the script calculates and prints the minimum, maximum, mean, standard deviation, and correlation with DHT22 for the sensor.

The script also saves the data to a CSV file every 100 iterations. The index=False argument is used to prevent the DataFrame's index from being written to the file. The file will be overwritten each time, so if you want to keep all data, you should append to the file instead (change the mode from 'w' to 'a' in the to\_csv() function). However, keep in mind that continuously writing large amounts of data to a file can slow down your script.

User

does the descriptive analysis is updated on real-time?

ChatGPT

Yes, the descriptive statistics are recalculated and printed in real-time every time a new reading is received from a sensor. This is done inside the main loop of the script, specifically in the following block of code:

python

Copy code

# Update the DataFrame and calculate descriptive statistics

data\_df = pd.DataFrame(sensor\_data)

print('--- Descriptive statistics ---')

print('Min:', np.min(data\_df[sensor\_name]))

print('Max:', np.max(data\_df[sensor\_name]))

print('Mean:', np.mean(data\_df[sensor\_name]))

print('Standard deviation:', np.std(data\_df[sensor\_name]))

print('Correlation with DHT22:', data\_df[sensor\_name].corr(data\_df['DHT22']))

print()

This block of code is executed every time a new reading is received from a sensor (excluding the AMG8833 Grid-EYE). The DataFrame is updated with the new reading, and then the minimum, maximum, mean, standard deviation, and correlation with DHT22 are calculated for the sensor's data and printed to the console. This happens in real-time as the data is being received from the sensors.

Keep in mind that continuously calculating and printing these statistics can slow down your script, especially as the amount of data grows. If you experience performance issues, you may want to calculate and print the statistics less frequently, for example, every 10 or 100 iterations.

User

can you plot the correlation in more 6 different subplots?

ChatGPT

Correlation is typically visualized using a scatter plot, where each point represents an observation and the position of the point is determined by the values of two variables (in this case, the sensor reading and the DHT22 reading). If there's a correlation between the two variables, it should be visible as a trend in the scatter plot.

Here's how you can modify the previous script to also create a scatter plot for the correlation with DHT22 for each sensor:

python

Copy code

import serial

import matplotlib.pyplot as plt

import pandas as pd

import numpy as np

# Open the serial port

ser = serial.Serial('/dev/ttyACM0', 9600)

# Set up the figure and subplots

plt.ion() # enable interactive mode

fig, axs = plt.subplots(14, 1, figsize=(10, 40)) # 14 subplots: 1 combined, 6 pairs, 6 correlations, 1 Grid-EYE

fig.tight\_layout(pad=3.0)

# Create dictionaries to hold our data arrays and plot lines for each sensor

sensor\_data = {}

lines = {}

data\_df = pd.DataFrame() # DataFrame to store all sensor data

# Colors for each sensor

colors = ['b', 'g', 'r', 'c', 'm', 'y', 'k']

# Function to update plot with new data

def update\_plot(ax, line, data):

line.set\_ydata(data)

line.set\_xdata(range(len(data)))

ax.relim()

ax.autoscale\_view()

plt.draw()

plt.pause(0.01)

while True:

if ser.in\_waiting > 0:

line\_str = ser.readline().decode('utf-8').rstrip()

print(line\_str)

# Parse the line as a sensor name and reading

sensor\_name, reading\_str = line\_str.split(' = ')

reading = float(reading\_str)

# If this is a new sensor, initialize its data array and plot line

if sensor\_name not in sensor\_data:

sensor\_data[sensor\_name] = []

lines[sensor\_name], = axs[0].plot([], [], color=colors[len(sensor\_data)], label=sensor\_name) # combined plot

axs[0].legend(loc='upper left')

if sensor\_name != 'AMG8833 Grid-EYE': # Exclude Grid-EYE from pair plots and correlation plots

i = len(sensor\_data) # subplot index

lines[sensor\_name + '\_pair'], = axs[i].plot([], [], color=colors[i-1], label=sensor\_name) # pair plot

lines['DHT22\_pair'], = axs[i].plot([], [], color=colors[i], label='DHT22') # pair plot

axs[i].legend(loc='upper left')

# Initialize correlation plot

axs[i+7].scatter([], [], color=colors[i-1], label=sensor\_name) # correlation plot

axs[i+7].legend(loc='upper left')

# Add the new reading to the data array

sensor\_data[sensor\_name].append(reading)

# Update the plots for this sensor

update\_plot(axs[0], lines[sensor\_name], sensor\_data[sensor\_name])

if sensor\_name != 'AMG8833 Grid-EYE': # Exclude Grid-EYE from pair plots and correlation plots

i = list(sensor\_data.keys()).index(sensor\_name) # subplot index

update\_plot(axs[i], lines[sensor\_name + '\_pair'], sensor\_data[sensor\_name])

if 'DHT22' in sensor\_data: # Only update DHT22 pair plot and correlation plot if DHT22 data is available

update\_plot(axs[i], lines['DHT22\_pair'], sensor\_data['DHT22'])

axs[i+7].scatter(sensor\_data[sensor\_name], sensor\_data['DHT22'], color=colors[i-1]) # correlation plot

axs[i+7].relim()

axs[i+7].autoscale\_view()

plt.draw()

plt.pause(0.01)

# Update Grid-EYE plot separately (assuming it's a 1D array for simplicity)

if sensor\_name == 'AMG8833 Grid-EYE':

axs[13].imshow([sensor\_data[sensor\_name]], cmap='hot', interpolation='nearest') # heatmap plot

axs[13].set\_title('AMG8833 Grid-EYE')

plt.draw()

plt.pause(0.01)

# Save the data to a CSV file every 100 iterations

if len(data\_df) % 100 == 0:

data\_df.to\_csv('sensor\_data.csv', index=False)

This script creates an additional 6 subplots for the correlation plots, and each time a new reading is received from a sensor, it adds a point to the correlation plot for that sensor. The correlation plot shows the sensor's readings on the x-axis and the DHT22 readings on the y-axis. If there's a correlation between the two, it should be visible as a trend in the scatter plot.

**ChatGPT 3.5 Needs Checking and Modifications**

show the trend line formula as legend and print it right side of each of 6 subplots where it is being done

ChatGPT

To display the trend line formula as a legend and print it on the right side of each of the 6 correlation subplots, you can modify the previous script as follows:

python

Copy code

import serial

import matplotlib.pyplot as plt

import pandas as pd

import numpy as np

# Open the serial port

ser = serial.Serial('/dev/ttyACM0', 9600)

# Set up the figure and subplots

plt.ion() # enable interactive mode

fig, axs = plt.subplots(14, 1, figsize=(10, 40)) # 14 subplots: 1 combined, 6 pairs, 6 correlations, 1 Grid-EYE

fig.tight\_layout(pad=3.0)

# Create dictionaries to hold our data arrays and plot lines for each sensor

sensor\_data = {}

lines = {}

data\_df = pd.DataFrame() # DataFrame to store all sensor data

# Colors for each sensor

colors = ['b', 'g', 'r', 'c', 'm', 'y', 'k']

# Function to update plot with new data

def update\_plot(ax, line, data):

line.set\_ydata(data)

line.set\_xdata(range(len(data)))

ax.relim()

ax.autoscale\_view()

plt.draw()

plt.pause(0.01)

# Function to calculate and display trend line equation as a legend

def display\_trendline\_equation(ax, x, y, color):

coeffs = np.polyfit(x, y, 1)

m = coeffs[0]

b = coeffs[1]

equation = f'y = {m:.2f}x + {b:.2f}'

ax.text(0.95, 0.95, equation, verticalalignment='top', horizontalalignment='right', transform=ax.transAxes,

color=color, fontsize=10, bbox=dict(facecolor='white', edgecolor='none', alpha=0.7))

while True:

if ser.in\_waiting > 0:

line\_str = ser.readline().decode('utf-8').rstrip()

print(line\_str)

# Parse the line as a sensor name and reading

sensor\_name, reading\_str = line\_str.split(' = ')

reading = float(reading\_str)

# If this is a new sensor, initialize its data array and plot line

if sensor\_name not in sensor\_data:

sensor\_data[sensor\_name] = []

lines[sensor\_name], = axs[0].plot([], [], color=colors[len(sensor\_data)], label=sensor\_name) # combined plot

axs[0].legend(loc='upper left')

if sensor\_name != 'AMG8833 Grid-EYE': # Exclude Grid-EYE from pair plots and correlation plots

i = len(sensor\_data) # subplot index

lines[sensor\_name + '\_pair'], = axs[i].plot([], [], color=colors[i-1], label=sensor\_name) # pair plot

lines['DHT22\_pair'], = axs[i].plot([], [], color=colors[i], label='DHT22') # pair plot

axs[i].legend(loc='upper left')

# Initialize correlation plot

axs[i+7].scatter([], [], color=colors[i-1], label=sensor\_name) # correlation plot

axs[i+7].legend(loc='upper left')

# Add the new reading to the data array

sensor\_data[sensor\_name].append(reading)

# Update the plots for this sensor

update\_plot(axs[0], lines[sensor\_name], sensor\_data[sensor\_name])

if sensor\_name != 'AMG8833 Grid-EYE': # Exclude Grid-EYE from pair plots and correlation plots

i = list(sensor\_data.keys()).index(sensor\_name) # subplot index

update\_plot(axs[i], lines[sensor\_name + '\_pair'], sensor\_data[sensor\_name])

if 'DHT22' in sensor\_data: # Only update DHT22 pair plot and correlation plot if DHT22 data is available

update\_plot(axs[i], lines['DHT22\_pair'], sensor\_data['DHT22'])

axs[i+7].scatter(sensor\_data[sensor\_name], sensor\_data['DHT22'], color=colors[i-1]) # correlation plot

axs[i+7].relim()

axs[i+7].autoscale\_view()

display\_trendline\_equation(axs[i+7], sensor\_data[sensor\_name], sensor\_data['DHT22'], colors[i-1]) # Display trend line equation

# Update Grid-EYE plot separately (assuming it's a 1D array for simplicity)

if sensor\_name == 'AMG8833 Grid-EYE':

axs[13].imshow([sensor\_data[sensor\_name]], cmap='hot', interpolation='nearest') # heatmap plot

axs[13].set\_title('AMG8833 Grid-EYE')

plt.draw()

plt.pause(0.01)

# Save the data to a CSV file every 100 iterations

if len(data\_df) % 100 == 0:

data\_df.to\_csv('sensor\_data.csv', index=False)

In this modified script, I added a function called display\_trendline\_equation which calculates the coefficients of the trend line using np.polyfit and then displays the equation as a legend on the right side of each correlation subplot. The equation is positioned at (0.95, 0.95) using ax.text and styled with a white background and a semi-transparent box.

This way, the trend line equation will be displayed as a legend in each correlation subplot, showing the formula for the trend line that represents the relationship between the respective sensor reading and the DHT22 reading.

User

you need to modify the above code by including Flask

ChatGPT

Certainly! To include Flask in the code, we can create a simple web application that displays the real-time plots using the Flask framework. Here's the modified code that includes Flask:

python

Copy code

import serial

import matplotlib.pyplot as plt

import pandas as pd

import numpy as np

from flask import Flask, render\_template, Response

# Open the serial port

ser = serial.Serial('/dev/ttyACM0', 9600)

# Set up the figure and subplots

plt.ion() # enable interactive mode

fig, axs = plt.subplots(14, 1, figsize=(10, 40)) # 14 subplots: 1 combined, 6 pairs, 6 correlations, 1 Grid-EYE

fig.tight\_layout(pad=3.0)

# Create dictionaries to hold our data arrays and plot lines for each sensor

sensor\_data = {}

lines = {}

data\_df = pd.DataFrame() # DataFrame to store all sensor data

# Colors for each sensor

colors = ['b', 'g', 'r', 'c', 'm', 'y', 'k']

# Function to update plot with new data

def update\_plot(ax, line, data):

line.set\_ydata(data)

line.set\_xdata(range(len(data)))

ax.relim()

ax.autoscale\_view()

plt.draw()

plt.pause(0.01)

# Function to calculate and display trend line equation as a legend

def display\_trendline\_equation(ax, x, y, color):

coeffs = np.polyfit(x, y, 1)

m = coeffs[0]

b = coeffs[1]

equation = f'y = {m:.2f}x + {b:.2f}'

ax.text(0.95, 0.95, equation, verticalalignment='top', horizontalalignment='right', transform=ax.transAxes,

color=color, fontsize=10, bbox=dict(facecolor='white', edgecolor='none', alpha=0.7))

# Flask setup

app = Flask(\_\_name\_\_)

@app.route('/')

def index():

return render\_template('index.html')

def generate\_plots():

while True:

if ser.in\_waiting > 0:

line\_str = ser.readline().decode('utf-8').rstrip()

print(line\_str)

# Parse the line as a sensor name and reading

sensor\_name, reading\_str = line\_str.split(' = ')

reading = float(reading\_str)

# If this is a new sensor, initialize its data array and plot line

if sensor\_name not in sensor\_data:

sensor\_data[sensor\_name] = []

lines[sensor\_name], = axs[0].plot([], [], color=colors[len(sensor\_data)], label=sensor\_name) # combined plot

axs[0].legend(loc='upper left')

if sensor\_name != 'AMG8833 Grid-EYE': # Exclude Grid-EYE from pair plots and correlation plots

i = len(sensor\_data) # subplot index

lines[sensor\_name + '\_pair'], = axs[i].plot([], [], color=colors[i-1], label=sensor\_name) # pair plot

lines['DHT22\_pair'], = axs[i].plot([], [], color=colors[i], label='DHT22') # pair plot

axs[i].legend(loc='upper left')

# Initialize correlation plot

axs[i+7].scatter([], [], color=colors[i-1], label=sensor\_name) # correlation plot

axs[i+7].legend(loc='upper left')

# Add the new reading to the data array

sensor\_data[sensor\_name].append(reading)

# Update the plots for this sensor

update\_plot(axs[0], lines[sensor\_name], sensor\_data[sensor\_name])

if sensor\_name != 'AMG8833 Grid-EYE': # Exclude Grid-EYE from pair plots and correlation plots

i = list(sensor\_data.keys()).index(sensor\_name) # subplot index

update\_plot(axs[i], lines[sensor\_name + '\_pair'], sensor\_data[sensor\_name])

if 'DHT22' in sensor\_data: # Only update DHT22 pair plot and correlation plot if DHT22 data is available

update\_plot(axs[i], lines['DHT22\_pair'], sensor\_data['DHT22'])

axs[i+7].scatter(sensor\_data[sensor\_name], sensor\_data['DHT22'], color=colors[i-1]) # correlation plot

axs[i+7].relim()

axs[i+7].autoscale\_view()

display\_trendline\_equation(axs[i+7], sensor\_data[sensor\_name], sensor\_data['DHT22'], colors[i-1]) # Display trend line equation

# Update Grid-EYE plot separately (assuming it's a 1D array for simplicity)

if sensor\_name == 'AMG8833 Grid-EYE':

axs[13].imshow([sensor\_data[sensor\_name]], cmap='hot', interpolation='nearest') # heatmap plot

axs[13].set\_title('AMG8833 Grid-EYE')

plt.draw()

plt.pause(0.01)

# Save the data to a CSV file every 100 iterations

if len(data\_df) % 100 == 0:

data\_df.to\_csv('sensor\_data.csv', index=False)

# Convert the figure to a PNG image

plt.savefig('static/plot.png')

# Yield the image as a byte stream

with open('static/plot.png', 'rb') as f:

yield b'--frame\r\nContent-Type: image/png\r\n\r\n' + f.read() + b'\r\n\r\n'

@app.route('/plot.png')

def plot\_png():

return Response(generate\_plots(), mimetype='multipart/x-mixed-replace; boundary=frame')

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True)

In this modified script, we have added the Flask framework and defined a Flask web application. The main route (/) returns the index.html template, which can be created in the same directory as the Python script.

The generate\_plots() function has been modified to continuously update the plots and save the figure as a PNG image (plot.png) in the static directory. The plot\_png() route serves the image as a multipart response, allowing the web page to display the real-time plots.

To view the real-time plots, you can access the Flask web application in your browser at http://localhost:5000 (or the appropriate IP address and port number).

Make sure to create the templates and static directories in the same directory as the Python script, and place an index.html file in the templates directory. The index.html file should include an image element to display the real-time plot:

html

Copy code

<!DOCTYPE html>

<html>

<head>

<title>Real-Time Sensor Data Plot</title>

</head>

<body>

<h1>Real-Time Sensor Data Plot</h1>

<img src="{{ url\_for('plot\_png') }}" alt="Real-Time Plot">

</body>

</html>

With this modification, the Flask web application will continuously update the plots and serve them to the web page in real-time.

User

Design the html with more details such as a nice heading where "Plant Body Temperature Monitoring" is written. Put some nice styles. Plot the graphs in suitable positions.

ChatGPT

Certainly! Here's an example of an enhanced HTML template with improved styling and a more visually appealing design for the Plant Body Temperature Monitoring application:

html

Copy code

<!DOCTYPE html>

<html>

<head>

<title>Plant Body Temperature Monitoring</title>

<style>

body {

font-family: Arial, sans-serif;

background-color: #f7f7f7;

margin: 0;

padding: 20px;

}

h1 {

text-align: center;

color: #333;

margin-bottom: 30px;

}

.container {

display: flex;

flex-wrap: wrap;

justify-content: center;

}

.plot {

width: 400px;

height: 300px;

margin: 10px;

background-color: #fff;

box-shadow: 0 2px 6px rgba(0, 0, 0, 0.1);

}

</style>

</head>

<body>

<h1>Plant Body Temperature Monitoring</h1>

<div class="container">

<div class="plot">

<h2>Combined Plot</h2>

<img src="{{ url\_for('plot\_png', plot='combined') }}" alt="Combined Plot">

</div>

<div class="plot">

<h2>Temperature Pair Plots</h2>

<img src="{{ url\_for('plot\_png', plot='pair') }}" alt="Temperature Pair Plots">

</div>

<div class="plot">

<h2>Correlation Plots</h2>

<img src="{{ url\_for('plot\_png', plot='correlation') }}" alt="Correlation Plots">

</div>

<div class="plot">

<h2>AMG8833 Grid-EYE</h2>

<img src="{{ url\_for('plot\_png', plot='grid\_eye') }}" alt="AMG8833 Grid-EYE">

</div>

</div>

</body>

</html>

In this updated HTML template, we have added CSS styles to enhance the appearance of the web page. The h1 element has a centered alignment, a dark color, and a margin-bottom for spacing. The body element has a light background color and padding for better readability.

The plots are organized using a flexbox layout with the .container class. Each plot is contained within a .plot class, which has a fixed width, height, margin, and a light background color. Each plot also includes an h2 heading to provide a descriptive title.

Feel free to customize the CSS styles further to match your desired design preferences.