TEL AVIV UNIVERSITY אוניברסיטת תל-אביב

Home Kit – Autonomous Systems lab

מוגש על ידי:

שם פרטי: עמית   שם משפחה: דוידי   ת.ז: 208630103

Autonomous Systems lab

TAU Faculty of Engineering

2nd semester, 2022-2023

3. **Tasks:**

Blink another led, was done by changing the PIN number from 9 to either 10 or 11.

Modifying the blinking time was achieved by changing the delay amount.

Creating sequenced blinking effects was achieved by adding different delay patterns.

4. **Tasks:**

To view the measurement in volts we can use the map function*: map(potValue, 0, 1023, 0, 5).*

To change the led's brightness we can use the *map* function again and calculate the brightness by:

int brightness = map(analogRead(potPin), 0, 1023, 0, 255)

DC Motor

To run the motor in one direction, stop and then run it in the other direction.

We can use the *analogWrite* and *delay* functions, for example:

```
    motorSpeed = 255;
    analogWrite(IN2,  255 - motorSpeed);
    analogWrite(IN1,   motorSpeed);
```

will run the motor in the CCW direction.

To run the motor based on the potentiometer we simply combine the two previous programs, and we choose the parameter: *motorSpeed*, based on the potentiometer value, (using the *map* function).

Distance Sensor

To run the motor via readings from the distance sensor we can use the following:

```
distanceSensor.getDistance();
```

which will return a value based on the distance from the object in front of the sensor.

we can use 'map' to normalize the result:

```
motorSpeed = map(max_dist_val, 0, 1000, 0, 255);
```

we also use:

```
Serial.println(distance);
```

To plot the sensor data.

Magnetic Encoder – Interrupts

Based on our measurements we found that the gear ratio is 30. And indeed, we get 12 encoder counts per one motor revolution, thus in our code we used the following definitions:

```
#define COUNTS_PER_REVOLUTION 12
#define GEAR_RATIO 30
```

The motor’s Position and Velocity is calculated: (note that ***encoderCounts*** is updated by interrupts)

$$\Delta Time = \frac{CurrentTime - PreviousTime}{1000}\ [s]$$

$$CurrentPosition = \frac{EncoderCounts}{EncoderCountsPerRevolution \cdot GearRation}\ [Rotations]$$

$$\Delta Position = CurrentPosition - PreviousPosition$$

$$Velocity = \frac{\Delta Position}{\Delta Time} \cdot 60\ [RPM]$$

Arduino serial parser

The parser was implemented using the function:

```
Serial.readStringUntil('\n');
```

Which reads a string until the delimiter “\n” from serial.

After which, we parse the string by splitting it to substrings that ends with “,” with the function indexOf which returns the next index that the character ‘,’ is found.

```
int delimiterIndex = inputString.indexOf(',', index);
```

and then substring, which returns the substring [index to delimiter Index - 1]. (if the delimiter is not present we skip this part).

```
substring = inputString.substring(index, delimiterIndex);
```

Closed loop implementations

The closed loop implementations were done using a PID controller function, the EncoderCounts variable (that is being updated from interrupts), a LPF, and a 100 Hz loop.

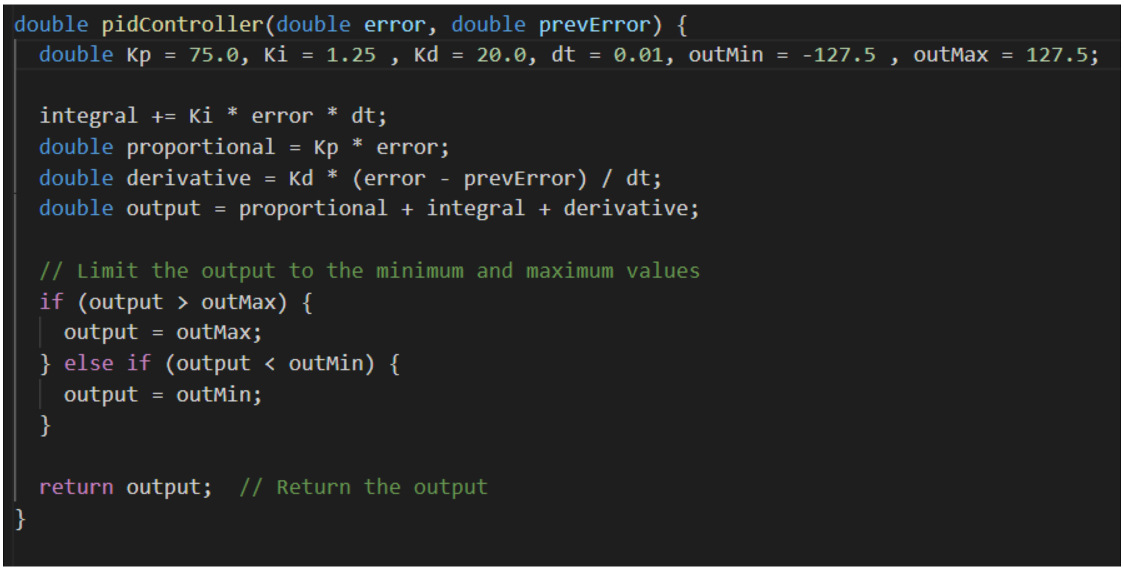
Each loop iteration begins with getting a user input (from serial or from one of the sensors (potentiometer or distance)), then we calculate the Position (using EncoderCounts).

With the position we can calculate the RPM and feed it to a LPF (if the design is a velocity control).

We calculate the error (between the user input and the measurement) and insert it to the PID controller function, which results in a control signal which we feed to the DC motors (the control signal is being fed with minor changes according to the system).

Position control close loop

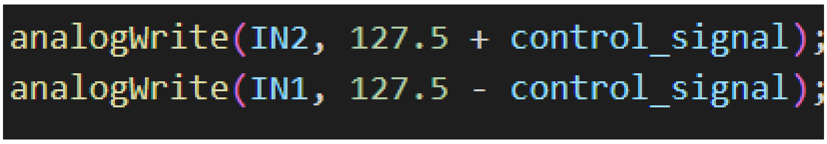
We implemented a PID controller with the following function:



```
double pidController(double error, double prevError) {
  double Kp = 75.0, Ki = 1.25 , Kd = 20.0, dt = 0.01, outMin = -127.5 , outMax = 127.5;

  integral += Ki * error * dt;
  double proportional = Kp * error;
  double derivative = Kd * (error - prevError) / dt;
  double output = proportional + integral + derivative;

  // Limit the output to the minimum and maximum values
  if (output > outMax) {
    output = outMax;
  } else if (output < outMin) {
    output = outMin;
  }

  return output;  // Return the output
}
```

The control parameters chosen are $K_p = 75.0, K_i = 1.25, K_d = 20.0$.

We chose to limit the output to be ∈ $(-127.5, 127.5)$ since we control the motor with:

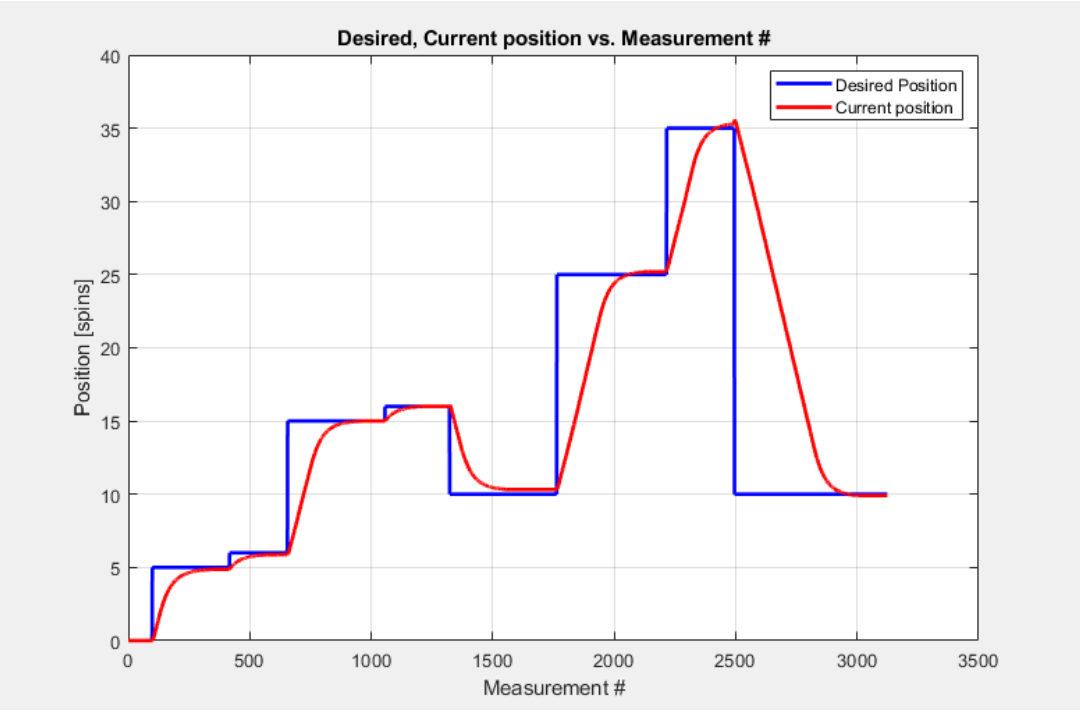


```
analogWrite(IN2, 127.5 + control_signal);
analogWrite(IN1, 127.5 - control_signal);
```

When the control signal is zero, the motor will be still, when the control signal is 127.5 one pin will get 255. The other will get 0, And vice versa.

We chose those parameters so that the overshoot will be minimal, further optimization could be done to improve the controller according to our needs.

Plotting Desired positions vs. actual position with this controller:



Velocity control close loop

The velocity control closed loop was implemented similarly to the previous section, with a few changes:

The controller used was a PI controller with the chosen parameters:

$$K_p = 1.0, K_I = 8.0, OutMax = 255.0, OutMin = 0.0$$

We feed the motors the following:

```
  analogWrite(IN2, control_signal);
  analogWrite(IN1, 255 - control_signal);
```

Furthermore, a Low-Pass filter was implemented that smooths the current RPM measurement, before calculating the error and inserting it to the PI controller.

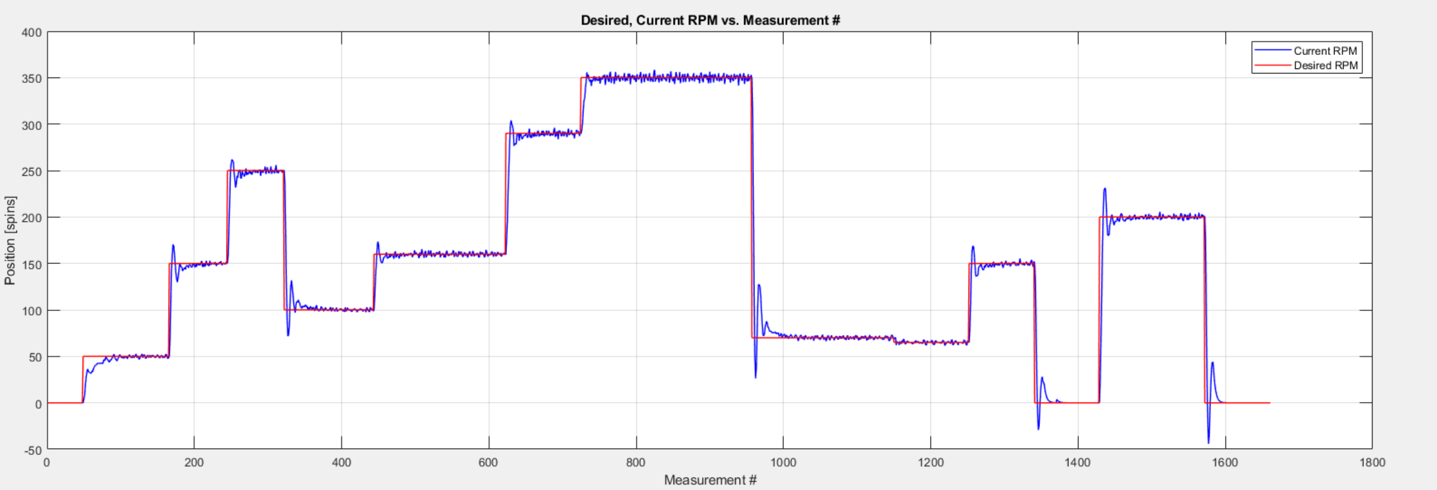
$$filtered\ measurement = \alpha \cdot \boldsymbol{current\ measurement} \quad + (1.0 - \alpha)\ \boldsymbol{previous\ measurement}$$

$$\alpha = 0.3$$

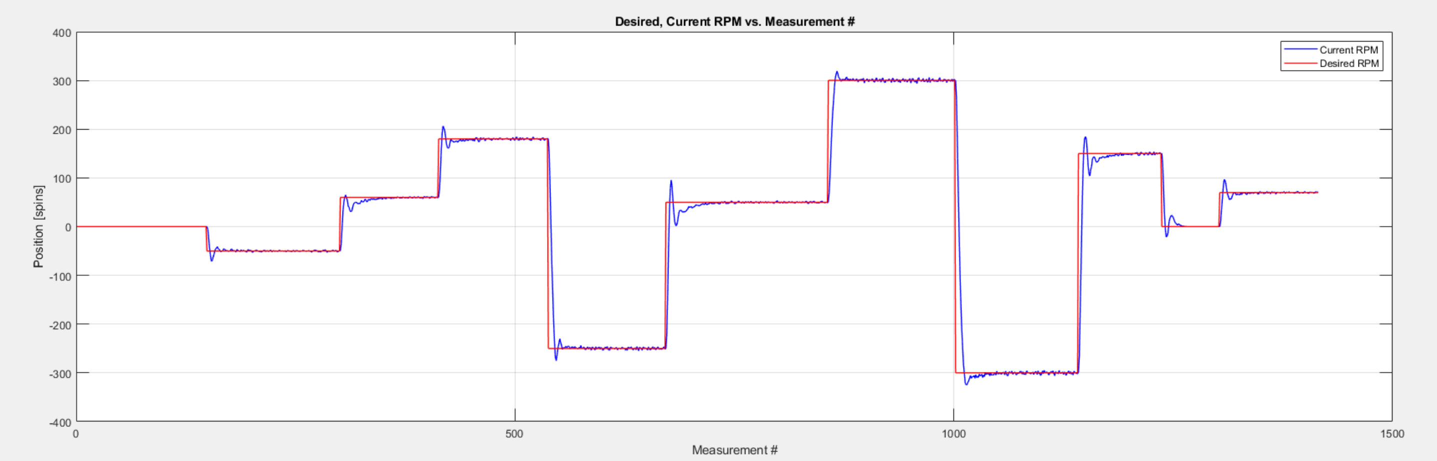
An overflow protection was implemented as well (since EncoderCounts can change from $32767 \leftrightarrow -32768$):

```
  if (abs (currentReading - previousReading) > 60000)
    currentReading = previousReading
```

Plotting Desired RPM vs. actual RPM with this controller:



With negative RPMs:



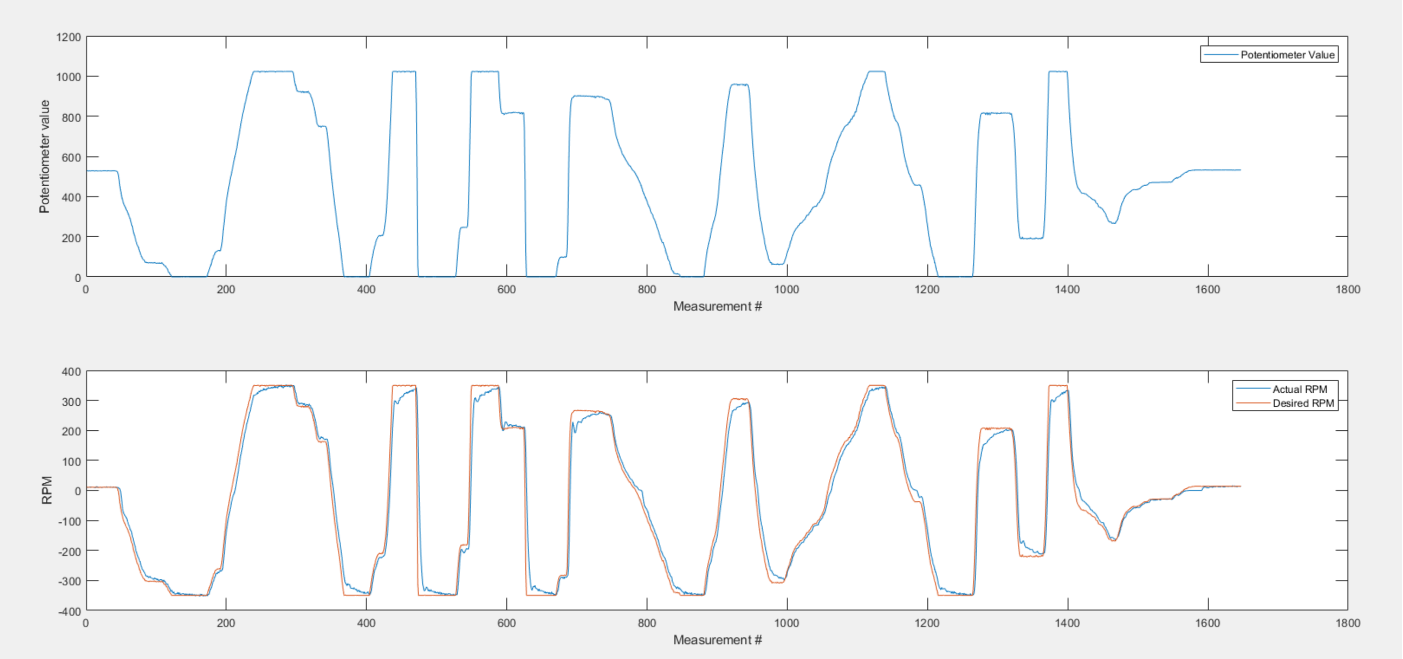
Potentiometer / Distance sensor feedback velocity control

To control the motor’s rpm via the potentiometer, the core of the program remained the same, the input source was changed from serial input to be:

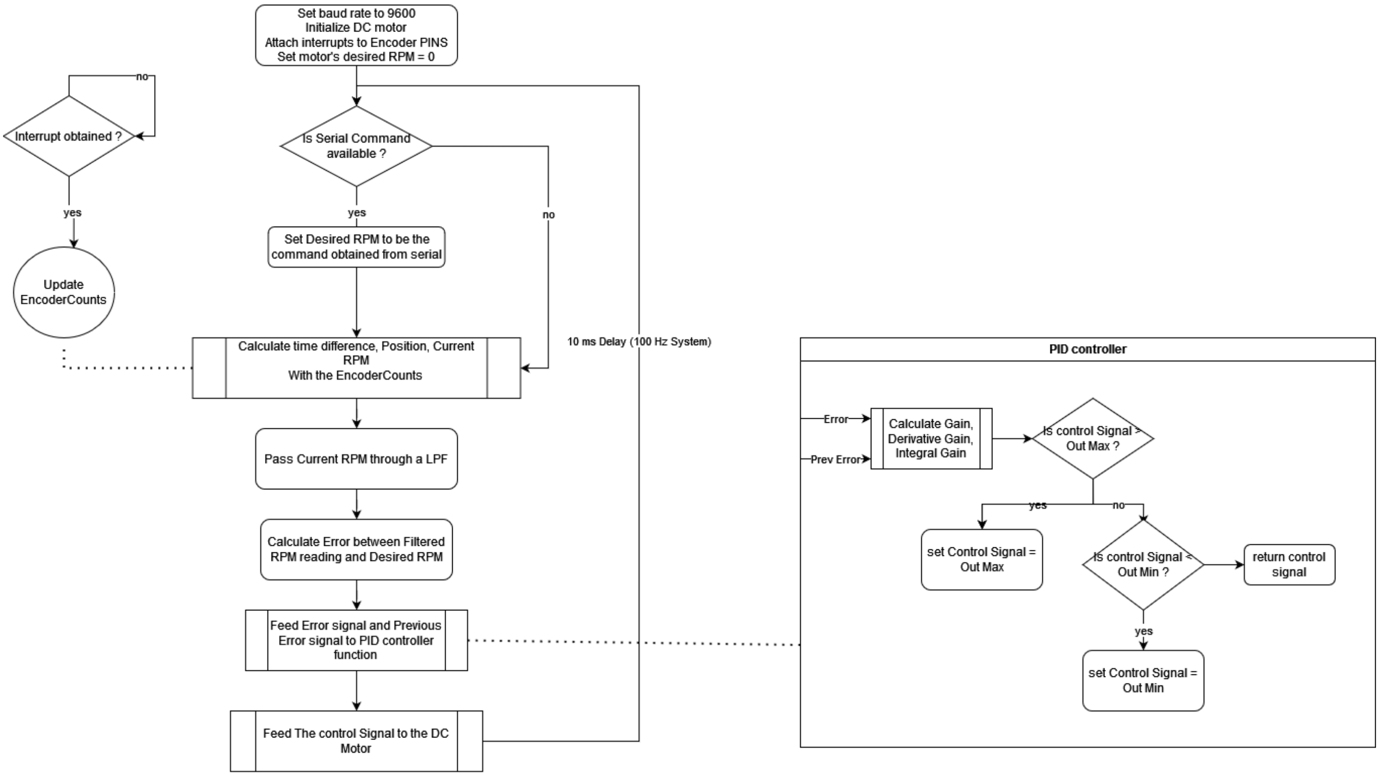
```
potVal = analogRead(potPin);
desiredRPM = map(potVal, 0, 1023, -350, 350);
```

Since the minimum and maximum RPMs are around ± 350, the map function was used with those boundaries.

Plotting the Desired and Actual RPMs with this controller:



A Block diagram that summarizes the Velocity Control Loop (Serial Input Controlled)


Set baud rate to 9600
Initialize DC motor
Attach interrupts to Encoder PINS
Set motor's desired RPM = 0
Interrupt obtained ?
no
yes
Update EncoderCounts
Is Serial Command available ?
yes
no
Set Desired RPM to be the command obtained from serial
Calculate time difference, Position, Current RPM With the EncoderCounts
10 ms Delay (100 Hz System)
Pass Current RPM through a LPF
Calculate Error between Filtered RPM reading and Desired RPM
Feed Error signal and Previous Error signal to PID controller function
Feed The control Signal to the DC Motor
PID controller
Error
Prev Error
Calculate Gain, Derivative Gain, Integral Gain
Is control Signal > Out Max ?
yes
no
set Control Signal = Out Max
Is control Signal < Out Min ?
return control signal
yes
set Control Signal = Out Min

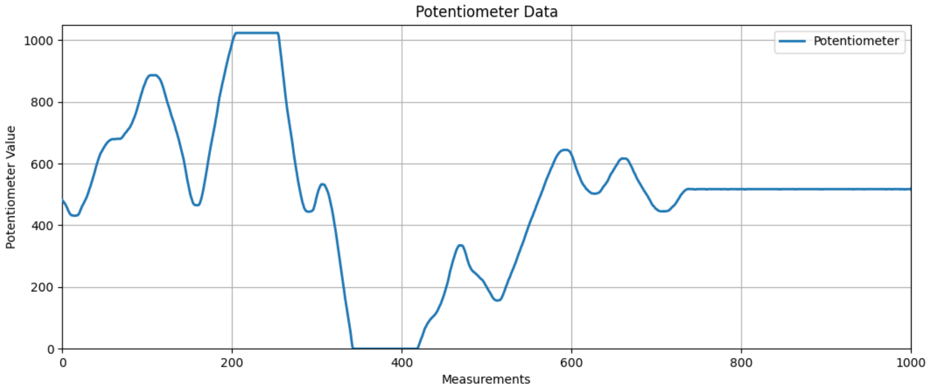
Note – the difference between the Serial controlled Velocity Control and the Position Control \ Potentiometer controlled velocity control Is minor.

Python - Arduino communication

To make the readings work with the potentiometer value, all we need to change is:

```
const int SENSOR_1_PIN = 0;
```

The plot obtained from the Python script:



To make the readings work with the **distance sensor**:

We must include the following code:

```
#include "SparkFun_VL53L1X.h"
#include <Wire.h>
```

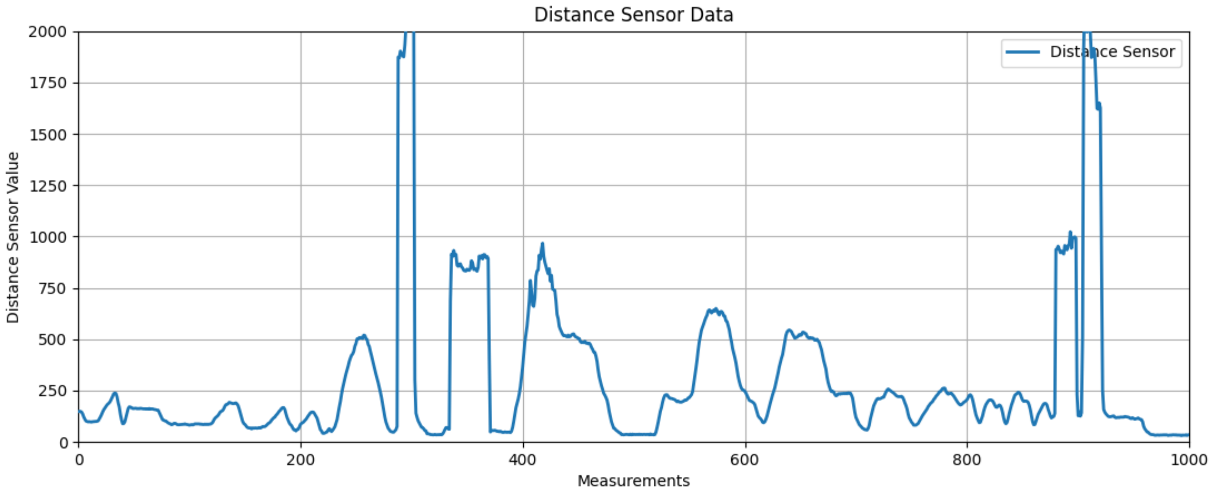
Declaring the sensor:

```
SFEVL53L1X distanceSensor;
```

Initializing it in setup(), and adding to loop() the following code (inside the if statement):

```
    distanceSensor.startRanging(); //Write configuration bytes to initiate measurement.
    while (!distanceSensor.checkForDataReady()) { delay(1); }
    int sensor1_value = distanceSensor.getDistance();
    distanceSensor.clearInterrupt();
    distanceSensor.stopRanging();
```

The plot obtained from the Python script:



How does the python \ Arduino code works in this example?

Python:

The given python script establishes a serial communication with an Arduino board (serial with baud-rate 115200), receives data, saves it to a file, and plots the data in real-time using Matplotlib.

In the main function, an ArduinoCommunication and SensorDataPlot classes are being created (the With statement ensures that the function close(self), is being called at the end of the script.)

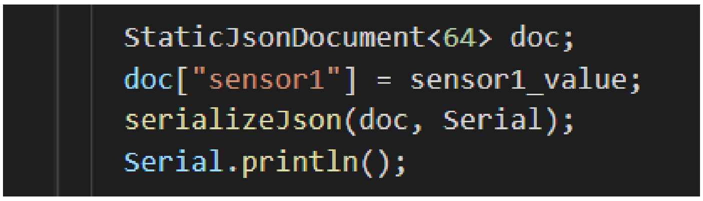
The ArduinoCommunication is handled in a thread that continuously checks the serial for messages and writes them to file. The data_callback function is declared to be plotter’s handle_data method. Which means that after each message obtained from the serial, the plotter’s method “handle_data” is being called with the data obtained.

The plotter’s handle_data method, gets a value from the dictionary “data” (in the example given, the key is “sensor1”), and then adds it to a “deque” type (a list-like structure).

The update_plot function is being called repeatedly by animation.FuncAnimation(), which updates the X and Y axis data

Arduino

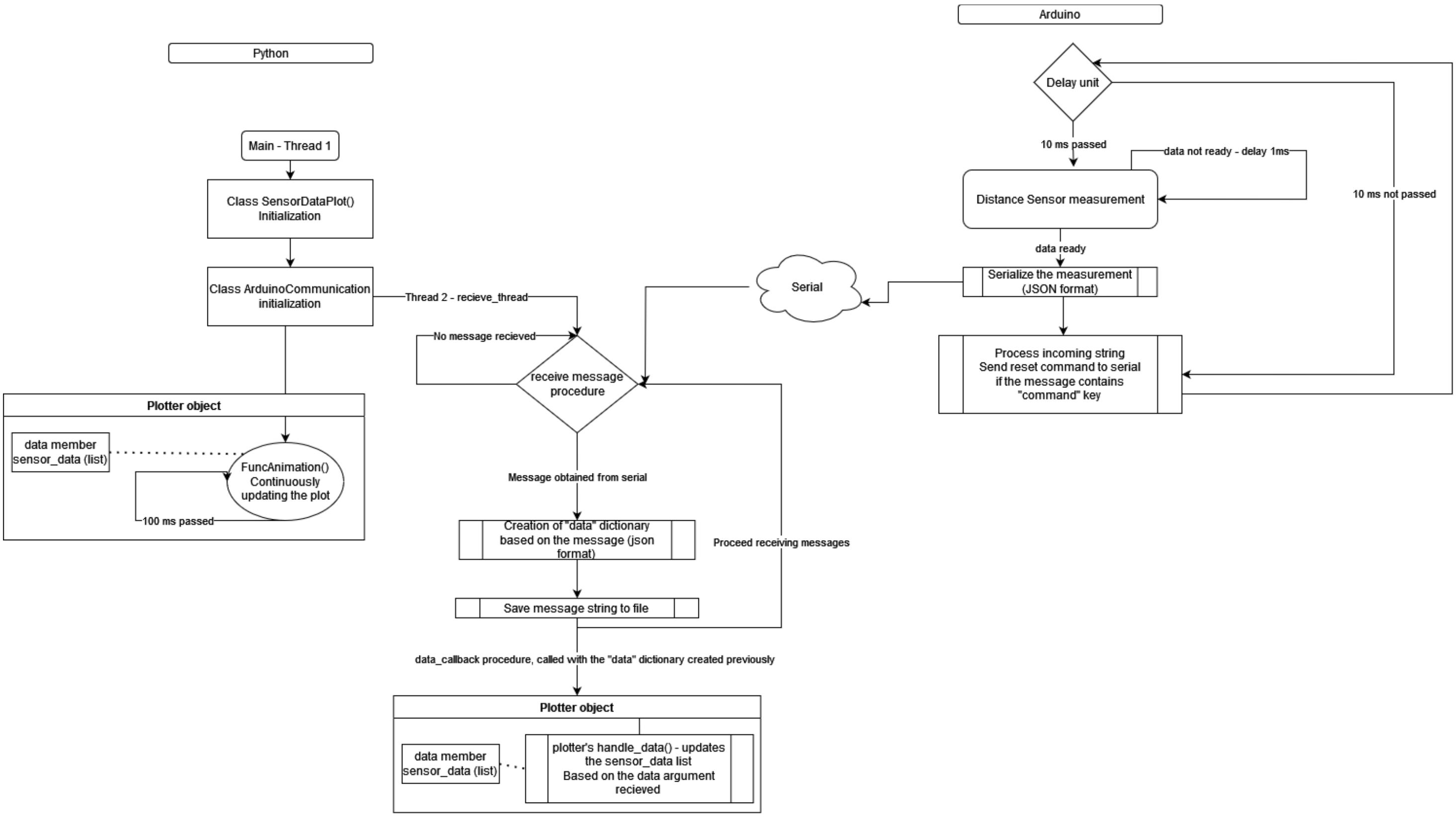
The code segment that sends the sensor data in a json format is:



```
StaticJsonDocument<64> doc;
doc["sensor1"] = sensor1_value;
serializeJson(doc, Serial);
Serial.println();
```

processIncomingString() takes an incoming string, parses it as JSON, checks if it contains a specific command (in this case, "reset"), and responds with a JSON message over the serial connection if the command is recognized.

A block diagram that summarizes the procedure of the Arduino board and the Python script:


Python
Main - Thread 1
Class SensorDataPlot() Initialization
Class ArduinoCommunication initialization
Plotter object
data member sensor_data (list)
FuncAnimation() Continuously updating the plot
100 ms passed
Thread 2 - recieve_thread
No message recieved
receive message procedure
Message obtained from serial
Creation of "data" dictionary based on the message (json format)
Proceed receiving messages
Save message string to file
data_callback procedure, called with the "data" dictionary created previously
Plotter object
data member sensor_data (list)
plotter's handle_data() - updates the sensor_data list Based on the data argument recieved
Serial
Arduino
Delay unit
10 ms passed
10 ms not passed
data not ready - delay 1ms
Distance Sensor measurement
data ready
Serialize the measurement (JSON format)
Process incoming string Send reset command to serial if the message contains "command" key