1-D Lookup Table

1 Introduction

Correlating the measurement voltage of a sensor into its real-world measurement value is an important function when developing mechatronic applications. There is an in-built Arduino mapping function, map(), that can perform this operation, but this function can only define a linear relationship, and will only work with integer values.

Section 2, of this document describes, the Arduino map() function and lists some of its draw backs of us use. Section 3 describes a simple implementation of 1-D lookup table, LUT, function that can be used with simple non-linear input/output characteristics, and describes some of the function's advantages and limitations. Finally, in section 0 an example application of using this LUT function is provided, alone with the source code, detailing its operation.

2 The Arduino map() Function

The Arduino map() function is limited to only being used to translate a linear relationship between the input variable and the output, as illustrated in Figure 1.

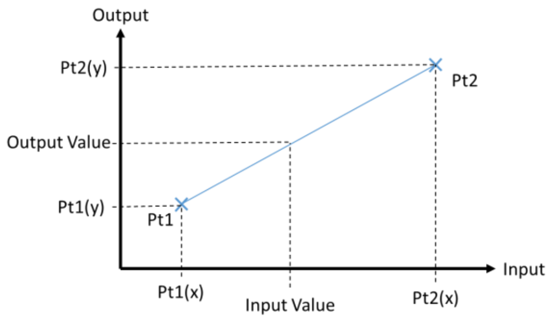

Figure 1. Graph illustrating the Arduino map() function

The syntax of the map() function, with respect to Figure 1, is:

```
Output value = map(input value, Pt1(x), Pt2(x), Pt1(y), Pt2(y))
```

Where: Pt1 and Pt2 are two points on the linear trend between the input and the output of the function. In many cases this can be a very useful function, but the map() function has two major limitations:

1. It can only describe a linear relationship between the input and the output
2. The mathematical operation is performed using integer variables. This reduces the precision of the function and may provide appropriate precision for your application.

3 Simple 1-D Lookup Table

A lookup table, LUT, is a function that can be used map an input variable through a non-linear characteristic, (generally based on experimental data), to find a corresponding output, as illustrated in Figure 2. The lookup table uses a search algorithm to find the most appropriate output for a given input.

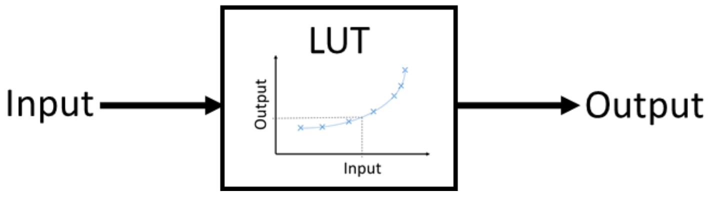

Figure 2. Block Diagram representation of a lookup table, LUT.

In this section the implementation of a simple LUT function is described that overcomes some of the limitations of the map() command, discussed in the previous section. This LUT function can be used to describe a non-linear input/output relationship, an example of which is illustrated in Figure 3.

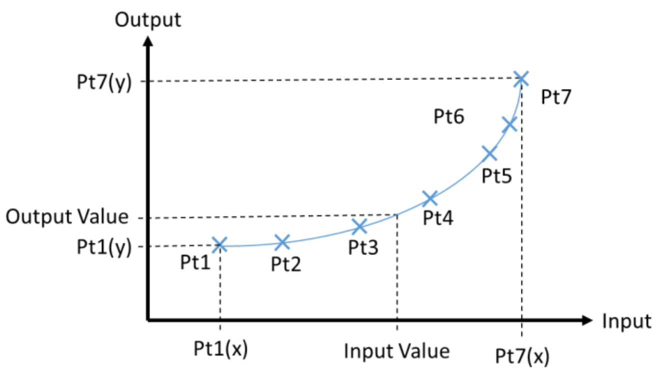

Figure 3. Graph illustrating the simple 1-D lookup table function

The non-linear characteristics shown in Figure 3 can be approximated using a piecewise linear function, described at the points: Pt1, Pt2….Pt7. Between these points, the non-linear characteristic is simplified to a linear function, i.e. a strait line between Pt1 and Pt2, another between Pt2 to Pt3, and so on.

The simple LUT algorithm works by searching alone the characteristic to determine between which two points the input value is located. In the case of the example illustration in Figure 3, the input value is located on the linear segment between Pt3 and Pt4. The output value can be calculated using a simple linear interpolation between these two points:

$$TempFraction = \frac{Pt4|_x - Input\ Value}{Pt4|_x - Pt3|_x}$$
$$Output\ Value = TempFraction \times \left(Pt4|_y - Pt3|_y\right) + Pt3|_y$$

Equation 1

Where: TempFraction is a temporary variable, $Pt3|_x$ and $Pt4|_x$ are the x-axis components of Pt3 and Pt4, and $Pt3|_y$ and $Pt4|_y$ are the y-axis components of Pt3 and Pt4.

Note: The Equation 1 is only valid for the input values located between Pt3 and Pt4.

3.1 Defining the lookup table

The lookup table is defined using two arrays, xElements and yElements. The xElements array represent the input characteristics of the system, and the yElements represents how this input characteristic is mapped to the output. It can be seen from the graph in Figure 4 that the xElements from the table form the x-locations of the graph points, and the yElements form the y-locations of the graph points.

The LUT, in Figure 4, is only defined at points P1 to Pt11. To resolve input values between the discrete values of the xElements array, the linear interpolation function, described above, must be used.

Note: This method is not limited to 11 points, and the example implementation can work with LUT sizes up to 255 points.

The example code provided in section 4.2 defines a lookup table as described in Figure 4.

	xElements	yElements
Pt1	10.0	0.0
Pt2	100.0	51.0
Pt3	200.0	102.0
Pt4	300.0	153.0
Pt5	400.0	204.0
Pt6	500.0	255.0
Pt7	600.0	204.0
Pt8	700.0	153.0
Pt9	800.0	102.0
Pt10	900.0	51.0
Pt11	1000.0	0.0

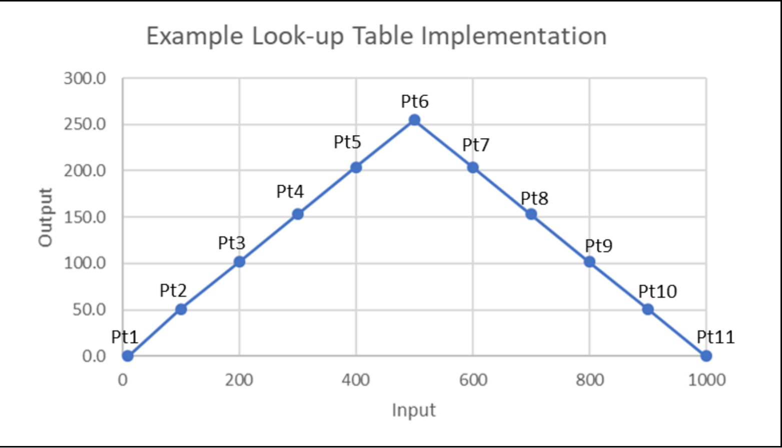

Figure 4. Details of the example lookup table implementation

It should be noted: the xElements and yElements arrays, defined for the LUT, are floating point variables, and depending on the number of elements you define in the arrays, this will result in a larger data usage than using the map() command. Furthermore, because this LUT implementation is performed in floating point, and you will generally be defining multiple LUT segments between these array points, therefore, there will be a higher processing overhead for this implementation than with the map() function. On the other hand, this LUT implementation can provide more accurate results, especially if you are attempting to map a non-linear characteristic.

3.2 Lookup table constraints

The LUT output is only defined within x-axis bounds of the data points, i.e. the algorithm can interpolate between the xElements array values, but not extrapolate outside of the x-axis range defined by the points Pt1 to Ptn, where Ptn is the last defined point.

Furthermore, the points contained within the xElements array must be singular, and increasing in value, as index of the array increases, but the elements do not need to be equally spaces, as illustrated in Figure 4. The yElements array does not have these constraints.

The Arduino implementation of the 1-D lookup table, LUT, will return an ‘overflow’ if the input variable is outside the bounds of the xElements Array, e.g. for the LUT described in Figure 4, if a zero value is input into the LUT, then an ‘overflow’ will be returned.

It should be noted that the size of the xElement array and the size of yElement array must be the same.

4 Example Application, Using the Simple 1-D Lookup Table

The example application that we have provided for the 1-D lookup table is based around the Arduino circuit shown in Figure 5, and the lookup table characteristics are shown in Figure 4.

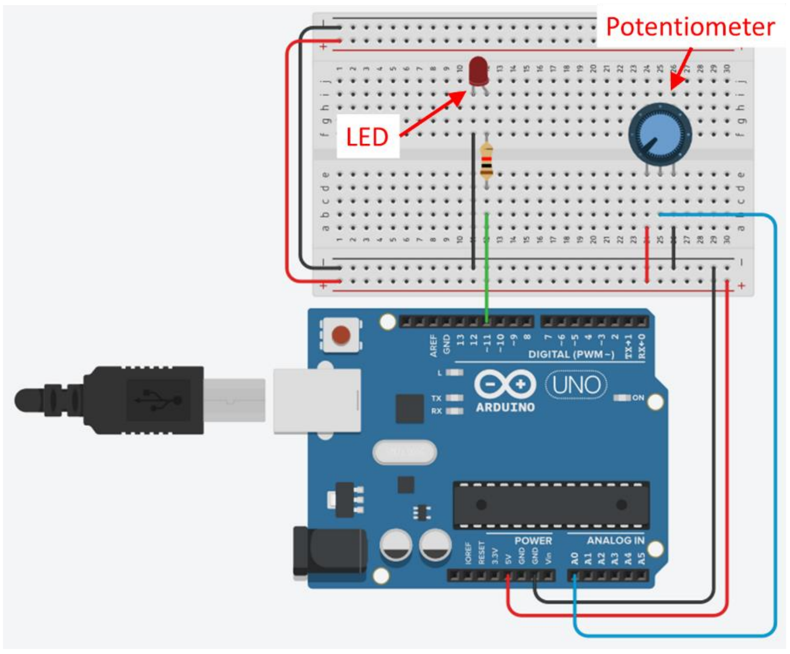

Figure 5. Test circuit for the simple 1-D lookup table example program

The input to the input to the lookup table is the ADC measurement value from the potentiometer, with a measurement range of 0 to 1023, and an LED is provided to a visual indication of the LUT function. The sample code also used the serial monitor to display the measured ADC value and the output value from the LUT.

Comments are provided in the code, shown in section 4.2, to further describe its operation.

4.1 Implementation of the Simple 1-D Lookup Table

The LUT is implemented in the code provided in section 4.2, in the following sections:

- **Global namespace:** This section defines and initialised the two LUT arrays used in this implementation: LuT_1_xElements and LuT_1_yElements. The LuT_1_xSize variable is required to pass the LUT array sizes to the LUT functions. The prototypes for the LUT functions are also located here: LookUpTable1D, CheckLuTArrays
- **setup():** The size of the LUT arrays are calculated and the CheckLuTArrays function is run to ensure the LUT arrays are formed correctly, as described in section 3.1.
- **loop():** The LookUpTable1D function is used to calculate an output value, based on the two LUT arrays and the specified input value.

It is recommended that when you implement the LUT functions in your user code, you include all the above elements in similar places in your code, to ensure that the code will only run if the arrays are correctly structured. (Remember to also copy the two LUT functions: LookUpTable1D and CheckLuTArrays, into the end of your Arduino Sketch.)

The functions are designed in such a way as to allow more than one LUT to be implemented in a in your code. To achieve this a 2nd set of arrays and LUT size variables need to be defined, and functions can be reused to check the array definitions and perform the LUT operation.

Comments are provided in the code, shown in section 4.2, to further describe its operation.

4.2 Sample Code for the Simple 1-D Lookup Table Test Program Code

```
// This example program demonstrates the use and operation of a simple 1-Dlookup table
//
```

```
// Author: Ben Taylor
// University of Sheffield
// Date: September 2020
//

//Declare Global variables
int potAnalogPin = 0; // The Potentiometer is connected to analogue 0
int LEDpin = 11;      // The LED to pin 11 (PWM pin)
int potReading;      // the analogue reading from the FSR resistor divider
float interpVal;
int LEDbrightness;

// Define the elements in lookup table 1
float LuT_1_xElements[] = {10, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000};
float LuT_1_yElements[] = {0, 51,  102, 153, 204, 255, 204, 153, 102, 51,  0};

// initialise  the lookup Table Array size variables
float LuT_1_xSize = 0; // Initialise variable as 0, to be calculated in setup()


// Function prototypes for the Lookup Table functions
float LookUpTable1D(float inVal, float elementsArraySize, float xElements[], float
yElements[]);
void CheckLuTArrays (float xArraySize, float yArraySize, float xElements[], float
yElements[]);

// ----------------------------------------------
// Arduino setup function
void setup() {
  Serial.begin(9600);  // Initialise the serial communications bus with a 9600 baud rate
  pinMode(LEDpin, OUTPUT); // Set the pin mode for the LED pin

  //Calculate the size of the lookup table, LuT_1, xElements Array and yElements Array
  LuT_1_xSize = sizeof(LuT_1_xElements) / sizeof(LuT_1_xElements[0]);
  float LuT_1_ySize = sizeof(LuT_1_yElements) / sizeof(LuT_1_yElements[0]); //Note this is a
local variable

  // Check the LuT_1 Arrays for size and increasing xElement Values
  CheckLuTArrays(LuT_1_xSize, LuT_1_ySize, LuT_1_xElements, LuT_1_yElements);

}

// ----------------------------------------------
// Arduino loop function
void loop() {

  // Read the potentiometer Value
  potReading = analogRead(potAnalogPin);
  // Perform the lookup table operation
  interpVal = LookUpTable1D(potReading, LuT_1_xSize, LuT_1_xElements, LuT_1_yElements);

  //Display results

  Serial.print("Analog reading = ");
  Serial.print(potReading);
  Serial.print(", Look - up Values = ");
  Serial.println(interpVal);

  // Use the output value from lookup table to control the brightness of an LED
  LEDbrightness = interpVal;
  analogWrite(LEDpin, LEDbrightness); // here you are using PWM !!!

  // Delay the next iteration of the loop
  delay(200);
}

// ----------------------------------------------
// This function performs the 1-D lookup table operation.
//
// The function will scan through the gaps between the elements in the xElement array, and
find
// where the inVal is located. A simple linear interpolation will be performed between indexes
n and n+1,
// of xElements and yElements, to calculate the lookup output value relating to the input
value
//
```

```
// If the input value does not lie in the range of values between the first and last elements
of the xElements
// array, an overflow variable will be returned
//
float LookUpTable1D (float inVal, float elementsArraySize, float xElements[], float
yElements[]) {

  //declare and initialise local variables
  float fraction = 0.0;

  // Scan through the gaps between the elements in the xElement array, and find where the
inVal is
  // located. Perform a simple linear interpolation between indexes n and n+1, of xElements
  // and yElements, to calculate the lookup output value relating to the input value

  for (int n = 0; n <= elementsArraySize - 2; n++) {
    if ((inVal >= xElements[n]) && (inVal <= xElements[n + 1])) {
      fraction = (inVal - xElements[n]) / (xElements[n + 1] - xElements[n]);

      // Return the output value and exit the function
      return (fraction * (yElements[n + 1] - yElements[n])) + yElements[n];
    }

  }
  // Return an overflow value of the input value is not within the input range of the
xElements array
  return 0xFFFFFFFF;
}

// ------------------------------------------------
// This function performs a simple check on the lookup table arrays to check the data is
valid:
// 1. Check the xElements and yElements arrays are the same size
// 2. Check that the values in the xElements array are increasing with index
//
// If an error is discovered, an error message is displated, and the code is held in an
infinite
// loop to block further execution
//
void CheckLuTArrays (float xArraySize, float yArraySize, float xElements[], float yElements[])
{

  // Check for array size missmatch between the xElements and yElements arrays
  // If found display an error message, then block the execution of the remainder
  // of the code
  if (xArraySize != yArraySize) { // check for array size missmatch
    Serial.println();
    Serial.println("Error: Number of elements in xElements array is not equal to the number
of");
    Serial.println("elements in the yElements array.");
    Serial.println();
    Serial.println("This program will not execute further, and sit in an infinite loop");
    while (1); // Hold the execution at this point
  }

  //Check that the values in xElements are increasing as the index values increase
  // If not display an error message then block the execution of the remainder of the code
  for (int n = 0; n <= xArraySize - 2; n++) {
    // Scan through the xElement indexes
    if (LuT_1_xElements[n] >= LuT_1_xElements[n + 1]) {
      Serial.println();
      Serial.println("Error: the values in xElements are not all increasing with increasing");
      Serial.println("index values");
      Serial.println();
      Serial.println("This program will not execute further, and sit in an infinite loop");
      while (1); // Hold the execution at this point
    }
  }
}
```