Lab 2 Notes – ECSE 426

Amjad Al-Rikabi: 260143211

Akeel Ali: 260275389

# Function Specification and Analysis of Requirements

The requirements of this lab were as follows:

1. Test the driver functions for the LSM303DLH sensor and the bus.
2. Write a program that reads and processes accelerometer readings to obtain tilt angles of the board.
3. Optimize the efficiency of the trigonometric calculations in processing the accelerometer readings.
4. Design and implement in code a filter for the accelerometer readings.

The challenge was solved by dividing it into 3 subproblems:

1. Initialize the bus and the sensor, then read out raw accelerometer readings.
2. Implement the necessary mathematical calculations to convert the raw readings into tilt angles.
3. Implement a filter for the raw readings.

# Software High Level Description

The software solution developed follows the pattern of the 3 subproblems identified in the previous section. The execution works in accordance to the following steps (each one of which will be fleshed out in the following paragraphs/sections):

1. First, a call to is made to setup the bus and the accelerometer.
2. Then, an intermediary step of calibration is executed whereby the program is interrupted using a breakpoint and the user is required to set the board on a flat surface.
3. The next steps then work in an infinite loop where:
   1. We first update the raw data stored by calling
   2. We then filter the raw data by calling
   3. Finally, we process the filtered data and calculate the pitch and roll of the board by calling

The software solution was modularized by placing the heavy lifting in separate source files and only exposing the needed functions using header files.

Two libraries of useful functions were hence created and they are outlined below along with a description of their exposed contents:

1. 1. : this function initializes the bus and the accelerometer using custom settings
   2. : this function calls one of the internal implementations that computes the arctan of the passed value

# Methodology/Implementation Notes

## Acc\_lib.c

The source file contains useful functions that perform two main services:

1. Deal with the accelerator and the bus
2. Compute the arctan using various implementations

The first service is fulfilled entirely by the function. It consists of a call to to initialize the I2C bus, followed by a call to which is passed a pointer to a struct containing the initializing parameters of the sensor

The second service presents an exposed interface which computes the inverse tangent of a passed value.

The implementation of this function was done in three ways which we outline below along with their advantages and disadvantages:

1. Simply using the function from the <math.h> library
   1. Advantage: this solution provides the most accurate results.
   2. Disadvantage: this solution presents an extra overhead and a performance hit due to the gained accuracy.
2. Using lookup tables to store pre-computed arctan results for discreet input values
   1. Advantage: this solution provides fast lookup as we use a function to map a value to an array index (.
   2. Disadvantage: this solution carries with it a precision loss and a storage overhead (a trade-off exists between the storage overhead the the precision: the larger the lookup tables, the more accurate will be our results, but the greater will be the space loss).
3. Using a set of if/else statements to direct values to pre-computed inverse tangent values
   1. Advantage: this solution speed advantages can only be proven using a performance experiment (see next section).
   2. Disadvantage: the excessive use of if/else statements, especially if not optimally placed, can present some performance hits. Moreover, the code is bloated as a result of the many cases (about 180 added lines for the arctan function).

With regards to method 3 above, the following snippet of code was used to generate all the cases of if/else statements (for a total of 90 degrees).

#include <math.h>

#define TO\_DEG\_FACTOR 57.29577951

**int** main(**void**){

**double** angle=1;

**double** limit=0;

**for**(angle=1;angle<91;angle++){

printf(**"else if (value <= %5.3f){\n"**,atan(angle/TO\_DEG\_FACTOR) );

printf(**" ans=%3.1f;\n"**,angle);

}

**return** 0;

}

Given the encapsulation of the code, the implementation chosen to compute the arctan does not affect the interface and can be easily changed.

## FIRFiler.c

Given the wide array of digital filters, our choice rested on the implementation of an FIR filter since it is stable for causal realtime events (all poles are set at zero), simpler to calculate for low-order filters, hence it is quick to develop and has the flexibility to be tweaked.

Using MATLAB’s *fdatool* utility, we have designed a fourth order filter (higher order filters are more accurate but were not recommended by the TAs since they become unnecessarily computationally expensive). Below is the frequency response of our filter:



The coefficients for the filter can be found to be hard coded in ‘fdacoefs.h’. Concerning storing previous values of the accelerometer reading, it was implemented in a circular buffer as that is more efficient for both memory and computationally (since it involves only moving pointers around). The code for this implementation can be seen in FIRFilter.c. The newly filtered output is calculated by performing a convolution of the filter’s impulse response coefficients with the most recent four accelerometer readings. This is performed for each axis separately, and separate circular buffers are stored separately for each corresponding axis.

Throughout different trial and error experiments, the filter was tweaked so that the final values that we have are more prone to noise and at the same time we aimed to not compromise the integrity of our signal nor the real-time need of performing the calculation quickly. In order to minimize memory usage and performance (since we did not have access to floating point math co-processor/hardware accelerators), the use of ‘float’ type was efficient.

# Performance Testing Method and Results

The 3 inverse tangent implementations were tested under differing circumstances to evaluate their speed performance.

The testing methodology consisted of setting a breakpoint just before the call to and another right after, and noting down the time elapsed between the two for different board orientations under each of the 3 arctan implementations. The results are shown in the table below:

|  |  |  |  |
| --- | --- | --- | --- |
| Orientation (degrees) | Lookup Table | Math.h Library | If/Else Statements |
|  | Time Elapsed (us) | Time Elapsed (us) | Time Elapsed (us) |
| 0,0 | 442.7 | 1095.3 | 375.1 |
| 45,0 | 413 | 1132.9 | 466.6 |
| 45,45 | 406.5 | 1122 | 513.4 |

The results present several features of interests. As expected, the math library takes about twice as much time to execute as the two other implementations. Since we are looking for an accuracy of about 2 degrees, this extra time investment is not worth it given that the precision it provides is not needed.

The if/else implementation of the inverse tangent function has varying execution times depending on the case that ends up handling the given value. Since we have 90 cases (for 90 degrees), if the particular orientation happens to fall in the first cases, the time elapsed can be even shorter than in the lookup table implementation. However, if the orientation falls in one of the later cases of the if statements, then we end up with a performance hit.

Variation in a system is usually an undesirable property and hence we have decided to discard this method on this basis. However, if we had decided to keep it, we could have improved the average time taken by putting the most common cases at the beginning of our if/else chain. Moreover, we could have used binary search to identify the particular value instead of linear search.

Finally, the lookup table implementation of the inverse tangent function provided the best and most consistent time measurements. Despite the slight space requirement to store the pre-computed values, we have opted for this implementation as our main arctan function as it provides the necessary accuracy along with the shortest time execution.

# Discussion

This lab provided us a great introduction to the use of drivers in communicating with a sensor using the I2C bus.

Implementing the algorithm to convert accelerometer readings to tilt angles taught us that raw sensor data can be used for various purposes.

Working on the inverse tangent function provided invaluable lessons in optimization at the embedded level. We decided to push the limits and experiment with 3 different implementations in order to gain better insights into time execution at the low level.