FACULTY OF ENGINEERING TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING
ELECTRONICS AND ICT
ANDREAS VESALIUSSTRAAT 13 B-3000 LEUVEN

SPAI - Lab Assignment 2

DSP Implementation:
Real Time Adaptive Notch Filter

Benedicta Marietta Amanda
Wiktoria Radecka

29 December 2024

Q-factors

The number of bits used to represent each parameter in the algorithm plays a significant role in determining the sharpness of the filter. An appropriate choice for a Q-factor is one that ensures that the values of each parameter can be accurately represented while retaining as much numerical precision as possible. For this reason, the Q-factors were chosen based on its ability to represent the maximum positive and negative values of the parameter in as few integer bits as possible.

The input data y is chosen to be in Q15. Under the assumption that y is normalized, it can be established that its values will be in the range of (-1,1). As such, it is sufficient to represent these values in Q15, with a sign bit.

The adaptive coefficient a is chosen to be in Q13. To ensure filter stability, there is a constraint for this parameter of $|a| < 2$. Considering this, at first glance it would be intuitive to represent a in Q14. However, it is crucial that a is not overflowing and is truly within the given limits. To be able to check this, an extra integer bit is required. As such, Q13 was chosen to represent the values of a.

The pole radius ρ is chosen to be Q15. A pole radius is never negative and is constrained to be in the range of (0,1). As such, it can be represented accurately using Q15.

The circular data buffer s is a bit trickier to deduce the maximum value for. Its value depends on the other parameters y, a, ρ and it's past values. The equation to find the current value for s is:

$$s[k] = y + \rho \cdot a \cdot s[k-1] - \rho^2 \cdot s[k-2]$$

Considering the ranges of the necessary parameters, the case where maximum *s[k]* is dependent on the input signal. Using the python code, it was found that the maximum value of s[k] from the given test signal (a sine of 400 Hz and 1200 Hz) was 4.77. To represent this value, Q12 would be sufficient (3 integer bits to represent more than 4 but less than 8).

The output e is should be a fraction of the input signal y, as the ANF is an attenuating filter. Knowing this, it can be deduced that using a similar Q-factor to the input will be sufficient to represent the output. As such, the Q15 representation was chosen for e.

The step size μ was chosen to be Q15. It is known that for a stable adaptation, the value of μ should be between 0 and 2 divided by the maximum eigenvalue. This value is most often less than 1, and as a result it is sufficient to not use any integer bits.

The exponential decay time constant λ is chosen to be represented in Q15, as it is always in the range (0,1) and not requiring any integer bits.

Table 1 summarizes the chosen representations succinctly.

parameter	Q-factor	motivation
`y`	`Q15`	The normalized input is constrained to the range (1, -1)
`s`	`Q12`	The maximum values in the buffer are between +/-(4,8)
`a`	`Q13`	The values for a are constrained to (-2, 2)
`e`	`Q15`	the output is in the same representation as the input `y`
ρ	`Q15`	The values for ρ are constrained to (0,1)
μ	`Q15`	The value for μ is chosen to be less than 1
λ	`Q15`	The values for λ are constrained to (0,1)

Table 1: Q-factor choice motivation

C Implementation

This 2nd order ANF implementation in C uses fixed-point arithmetic which is generally faster than floating-point arithmetic on DSP processors. To prevent overflow, we use 32-bit registers such as `AC0` and `AC1` in intermediate calculations. Also, a circular buffer is used to manage the delay line.

There are stability checks that add some calculation overhead but are vital to keep the filter stable and ensure proper filter behavior.

Computational cost per sample

1. filter operation
   - 4 move operation
   - 1 find minimum
   - 2 branch operation
2. error calculation
   - 1 square and shift operation
   - 1 delay
   - 3 move operation
   - 1 xor logic operation
   - 1 address modification
3. coefficient update
   - 4 move operations
   - 1 call
   - 1 comparison
   - 1 addition
   - 1 repeat
   - 1 shift operation
4. index update
   - 1 saturation operation
   - 3 move operation
   - 1 branch operation
   - 1 multiplication
   - 1 quare and accumulate operation

Assembly Implementation

In a similar manner, the assembly implementation of adaptive notch filter uses fixed arithmetic and a circular buffer to store signal estimates. It uses `LAMBDA` and `MU` parameters to control filter adaptation. The filter coefficient `a` is adapted with each sample to minimise the the error signal `e`.

Computational cost per sample:

- set up
  - 1 memory allocation operation
  - 3 move operations
- rho update
  - 4 move operations
  - 1 addition
  - 2 multiplications
  - 2 shift operation
  - 1 squaring
- filter operation
  - 6 move operations
  - 1 addition and 1 subtractions
  - 3 multiplications
  - 3 shift operations
- error calculation
  - 4 move operations
  - 1 addition and 1 subtraction
  - 1 multiplication
  - 1 shift operation
- coefficient update
  - 3 move operations, maximum 4 move operations
  - 1 addition
  - 2 multiplications
  - 2 comparisons
  - 1 negate operation (toggles bits)
- update index
  - 1 move operation

Functional Tests

For functional tests, we used two signals: a concatenated sine wave of 400 Hz and 1200 Hz, and a field recording of a fridge hum.

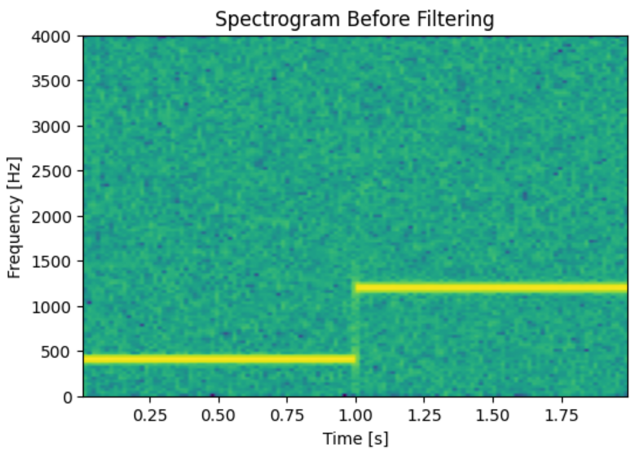

Figure 1: Sinusoidal Signal Spectrogram

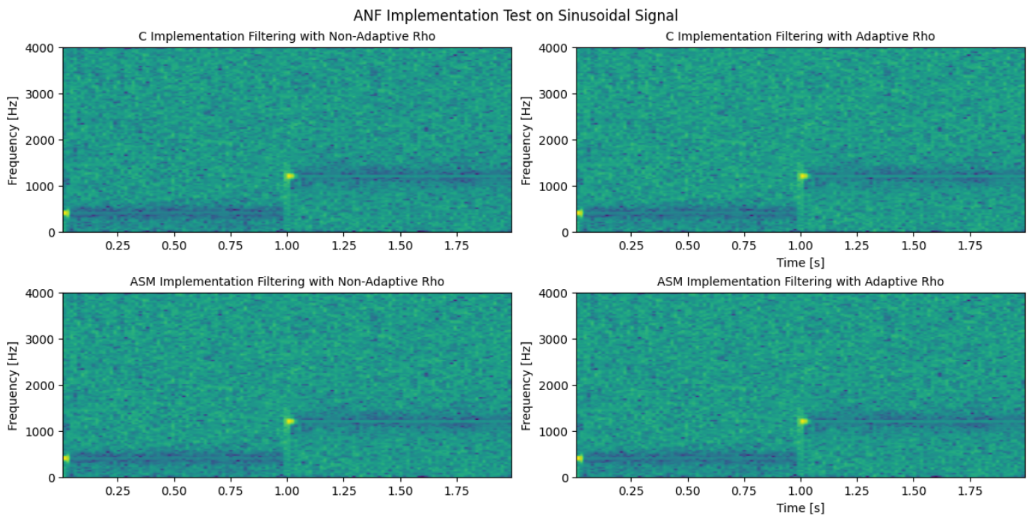

Figure 2: Filtered Sinusoidal Signal Spectograms

Fixed Sine Wave

As a preliminary test for the C and ASM implementations of the filter, we used the test signal proposed in the original python file. It was a signal made of concatenated sinusoids of different frequencies with Gaussian noise.

Using a spectrogram, we can clearly see how the implemented filters suppress the signal. Different patterns in shapes and colours in the spectograms mean that there are changing frequency components and energy distributions over time in the signals. For example, horizontal lines indicate consistent frequency components present throughout the signal duration and the areas of darker colours represent periods of relative low energy in certain frequency ranges. Here we can see in Figure 2 that the initial high energy at the frequencies 400 Hz and 1200 Hz from Figure 1 was successfully attenuated.

We can also observe that the filter converges very quickly, with very short yellow spots at the initialization of the signal and at the frequency change. Both implementations of the ANF are able to adapt and adjust to these changes rapidly, meaning our parameter values were well chosen.

In this case, there is no clear difference between the results using fixed ρ and adaptive ρ in the filtering process. Even so, an adaptive ρ is preferable as this allows the filter to flexibly change the notch bandwidth and adapt better to dynamic signals.

Fridge Hum

We also tested our implementations by removing the hum from a noisy field recording of a fridge. Each of the signal was inputted into the 2nd order adaptive notch filter and the output signal was plotted against its unfiltered version in Figure 3.

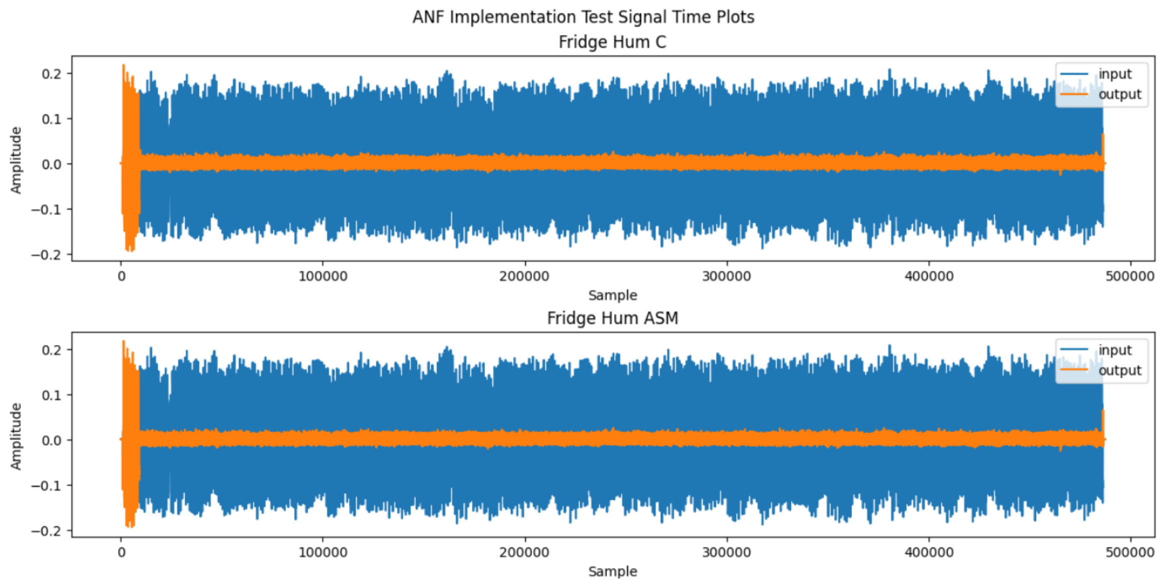

Figure 3: Fridge Hum Time Plots

In the plots of the Figure 3, we note that both implementations effectively reduce fridge noise as the amplitude of the signal diminished quite quickly. Furthermore, there isn't any visible difference between C and ASM implementations.

Next, we created spectograms as seen in Figure 4.

In the original fridge buzz signal we see that over the entire signal, high frequencies have low energy. We also note yellowish horizontal lines in low frequencies. This would mean that there is a consistent low-frequency component in the signal.

In the case of spectograms of Figure 4, we note significant difference between C and ASM implementations. It seems that C implementation manages to keep low energy high frequency components while removing the yellowish horizontal lines mentioned above. In contrast, in ASM implementation those yellowish lines seem to blend in the surroundings more as the entire energy of the signal seems to have boost in energy.

Lastly, in Figure 5 we see the magnitude spectrum of a signal fraction where we observe ANF operation. In both implementations ANF successfully removes frequency peaks present in fridge hum. There aren't any visible performance differences.

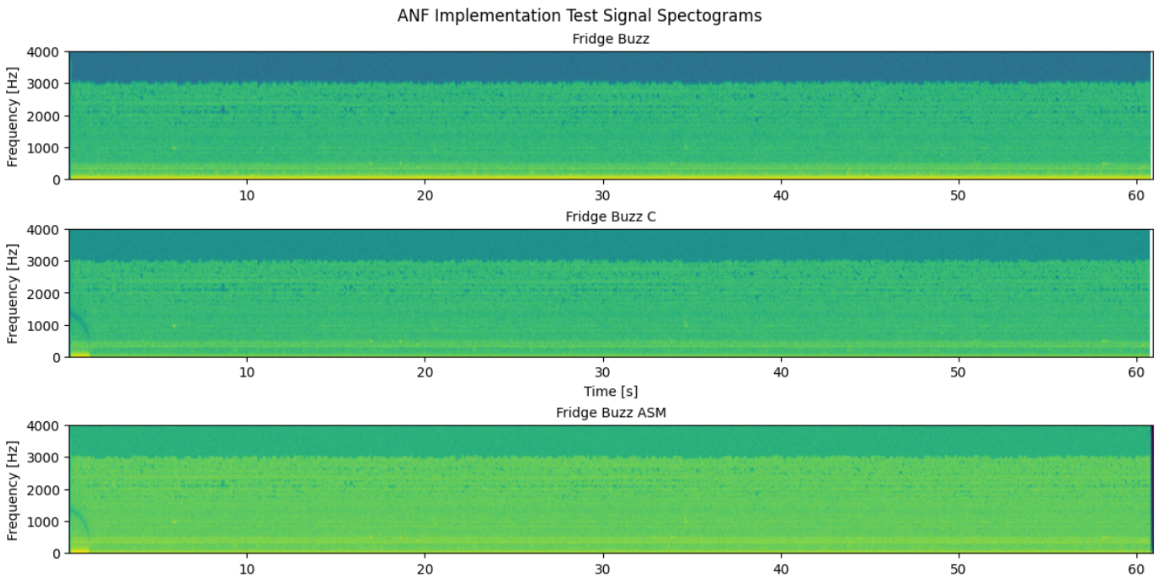

Figure 4: Fridge Hum Spectogram

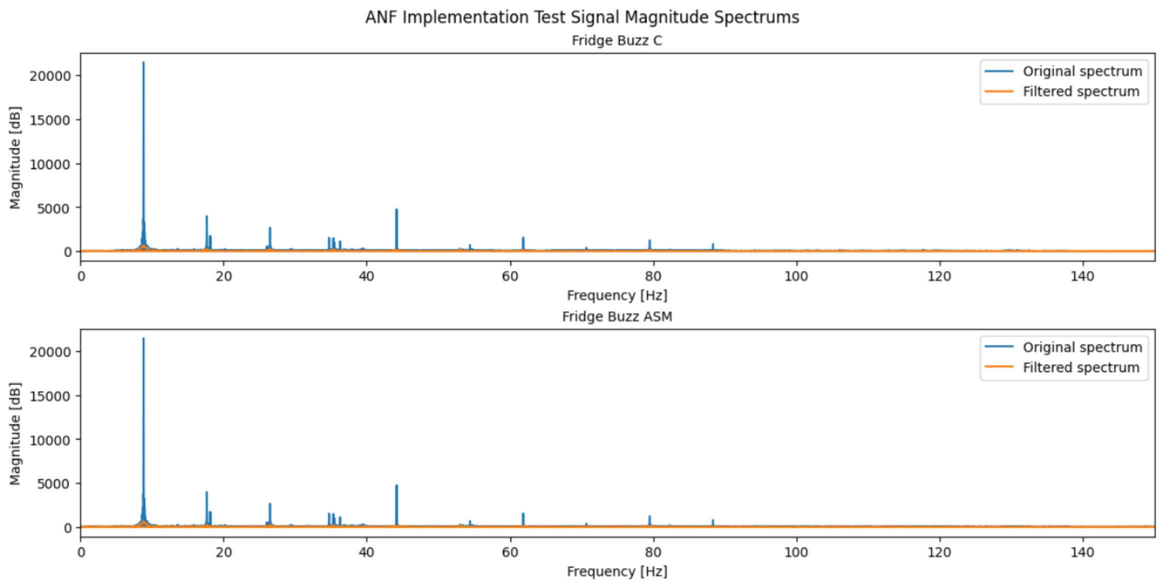

Figure 5: Fridge Hum Magnitude Spectrum

Comparison

In terms of operation count, C implementation has more operations, due to compiler-generated overhead. What this means is that there might be additional instructions for memory management and variable handling that add to the performance overhead. In contrast, ASM implementation allows us to have a direct control over instruction count. Thanks to manual optimisation we can reduce the number of operations compared to the C implementation.

Furthermore, C implementation has more memory accesses due to inefficient register utilisation as opposed to ASM implementation where we have direct control over data movement. The C implementation also uses stack operations for function calls[1] which generally requires more clock cycles.

To get a more concrete view of this, the profile clock was run on Code Composer Studio to calculate the number of cycles it would take to run one instance of the anf function in both C and ASM implementations. It was found that the C implementation took a total of 640 cycles, while the ASM took only 207. This means that the ASM code is more than 3 times more efficient than the C code.

While the ASM code is more efficient, the C implementation was faster to develop due to higher-level abstractions that allowed us to read and understand the code more easily. It was also simpler to debug. On the other hand, ASM implementation included low-level, machine-oriented instructions that were not always intuitive to understand. At the cost of using taking advantage of available hardware architecture, part of the development time had to be set aside to gain detailed understanding of available resources of the DSP development board.

To summarize, the C implementation is a good choice when a quicker, simpler solution is required while the ASM implementation is a better option when computational efficiency and speed is more important.

[1]Each function call requires pushing parameters onto stack, saving return address, saving previous frame pointer and allocating local variables. This takes few clock cycles as well as memory resources.