# SPAI Lab assignment 2 ANF filter

Quinten Vansina

## ANF filter in C

Designing the ANF filter in C was tougher than first expected. I started by looking at the MATLAB file ANF\_fixedpoint.m from Toledo to analyze the expected values after each operation. The test signal used is generated by the MATLAB section ‘signal generation v’ from Toledo. Using the debugger in MATLAB, I came to the conclusion that the following variables had a certain range which had to be kept in mind when designing the filter in C.

|  |  |  |  |
| --- | --- | --- | --- |
| Variable | Range in Matlab (\*) | Appropriate Q-factor | Range in C implementation |
| x\_debug (filter coef.) | [-2.4, 2.4] | 16Q11(\*\*) | [-16,15.99..] |
| AC (1st calculation) | [-1.9, 1.9] | 32Q14 | [-2,1.99..] |
| AC (2nd calculation) | [-3.6, 3.6] | 32Q13 | [-4,3.99..] |
| a\_i | [1, 1.9] | 16Q12 (\*\*\*) | [-8,7.99..] |
| AC1 (1st calculation) | [-4.4, 4.4] | 32Q12 | [-8,7.99..] |
| AC1 (2nd calculation) | [-2.5, 2.5] | 32Q13 | [-4,3.99..] |
| AC (3rd calculation) | [-2.1, 2.1] | 32Q13 | [-4,3.99..] |
| e and y | [-1, 1] | 16Q15 | [-1,0.99..] |
| AC (4th calculation) | [-0.015, 0.015] | 16Q15 | [-1,0.99..] |
| AC (5th calculation) | [-0.002, 0.002] | 16Q15 | [-1,0.99..] |

(\*) This is a rough approximation

(\*\*) The filter coefficients have a broader range, when testing with other signals, it was necessary to give the coefficients a wider range.

(\*\*\*) a\_i has a broader range to check for the |a| < 2 constraint, if it was limited by the Q-factor, it might possibly overflow and it would therefore be impossible to detect if the constraint was met or not.

The algorithm took approximately 5 seconds to finish when compiled in Debug-mode, however I had trouble getting the algorithm to work when compiled in Release-mode. In MATLAB, it took 0.0436 seconds to finish.

## ANF filter in assembly

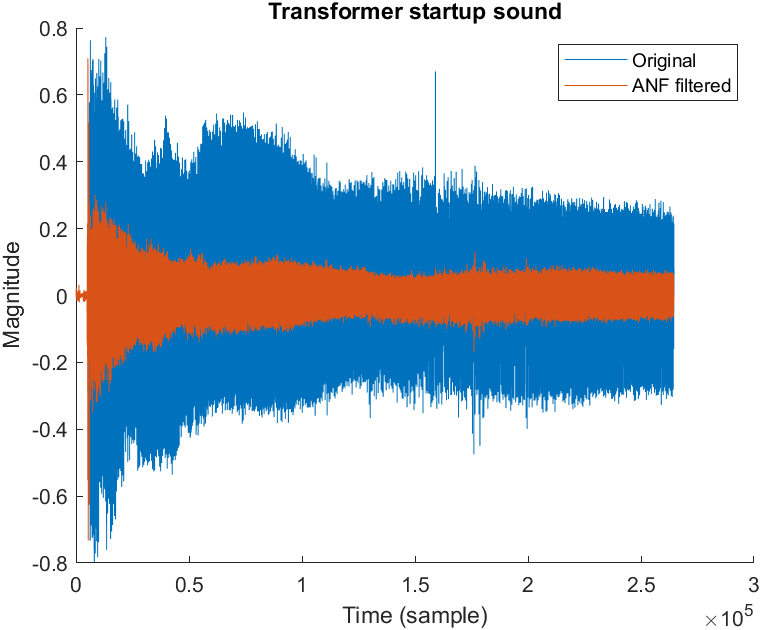
I manually tried to go from C to assembly by looking at the available datasheets and the solution of the circular FIR filter on Toledo. The code mostly performs the same calculations in the same order except when rounding the results of multiplications. What I first did was move the appropriate number to add after the multiplication in the register and then use the multiply and accumulate operation, this seemed better than doing a multiplication and an addition separately.

I used 4 Words (16 bit) of stackspace, but I eventually only needed 1 word to temporarily store some variables, I also used AC2 to put some temporary calculations; for example the rho squared was calculated and placed in AC2 before being multiplied by one of the filter coefficients. The memory and register usage should’ve been done better.

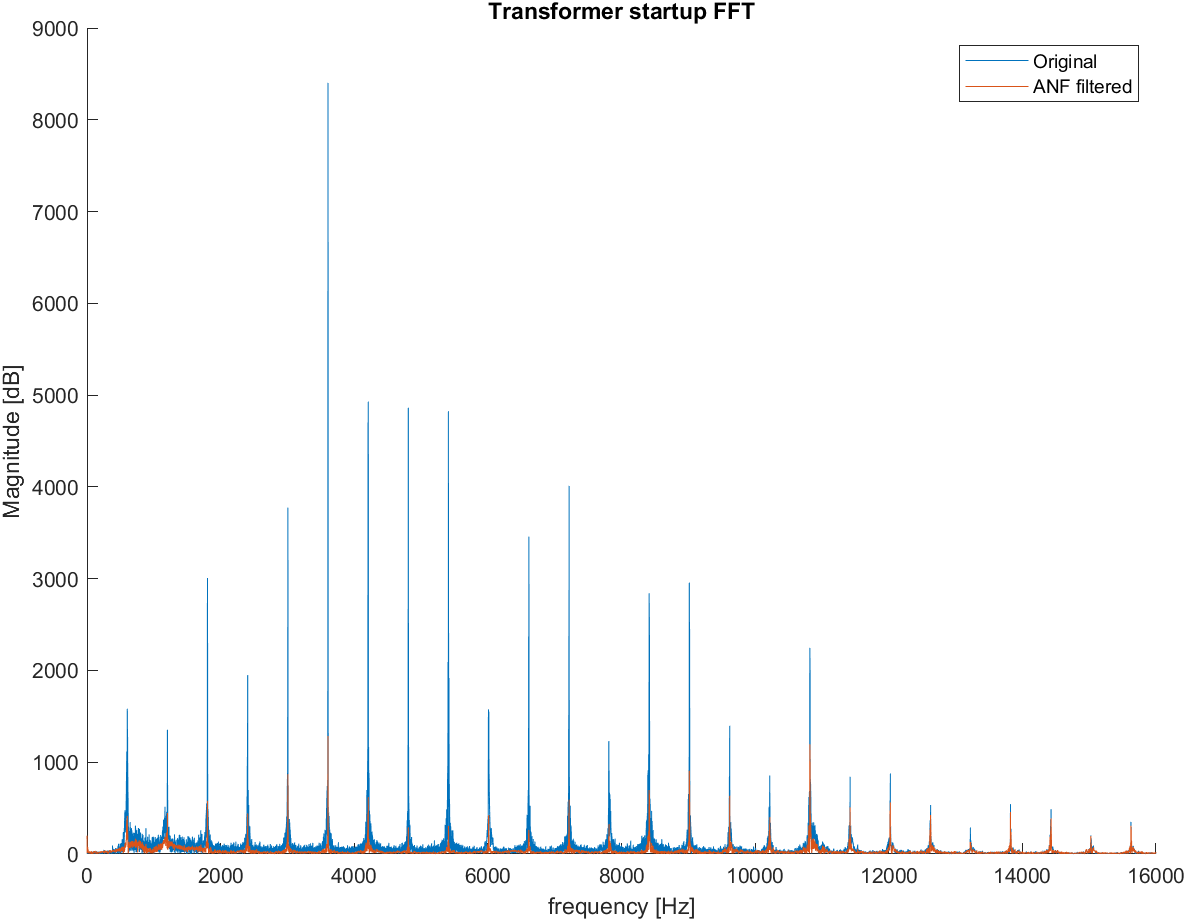
The performance wasn’t significantly different when compared to the C-implementation. Although the assembly implementation should be faster since the compiler in debug mode, doesn’t make optimizations and adds additional stuff for debugging purposes that slows down the performance. However, I don’t expect my assembly code to be faster when the C-code is compiled in release mode since I’m probably not implementing the assembly in the fastest way possible. On top of that, the Code Composer studio compiler probably knows better ways to optimize C code for the TMS320C5515 instruction set than me.

## Results 2nd order ANF (both assembly and C)

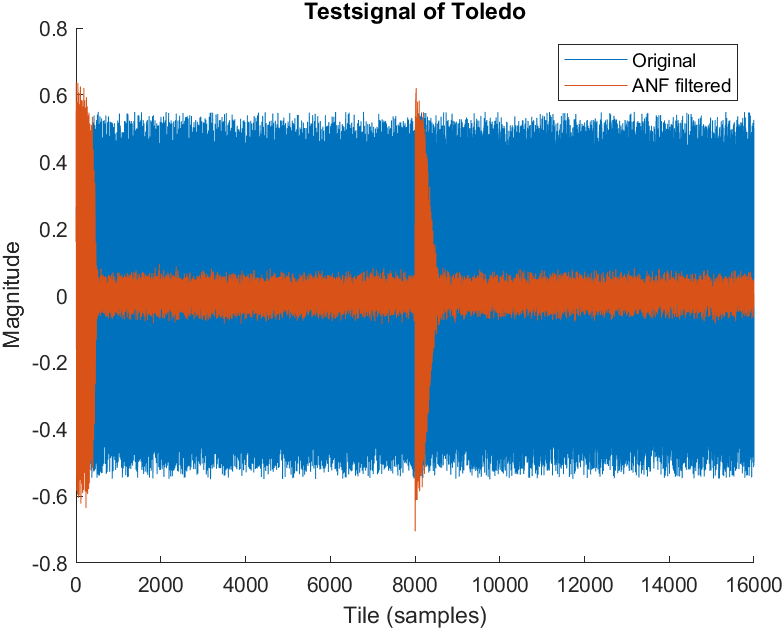
The test signal generated by the ANF\_fixedpoint.m file on Toledo generates a sine wave that changes midway with added noise. For a self-chosen test signal, I chose the recording of a transformer starting up as it contains several harmonics of 50Hz.



The filtered curve has significantly dropped in magnitude.



From the looks of it, it seems like the ANF had a really wide notch bandwidth because a lot of frequencies are deattenuated, to prove that it actually works, I’ll post the results from the test signal of the ANF\_fixedpoint.m as well. The results from the assembly and C code have minor differences which are negligible.



## Cascade attempt

I put the ANF algorithm twice in series but didn’t get any decent results. This is because when the test signal contains 2 prominent sine waves, it will go to the frequency which outputs the smallest error, not the one that filters a single frequency completely out. A better solution would be to put it in parallel so if the algorithm is run, the notch frequency can converge to 2 separate frequencies without interfering into each other.

## Online attempt

I copied the necessary functions/headers from lab 4 and outputted it to one of the audio jacks, when plugging in with my headphone, I get a jittery result, I have no idea why this is.