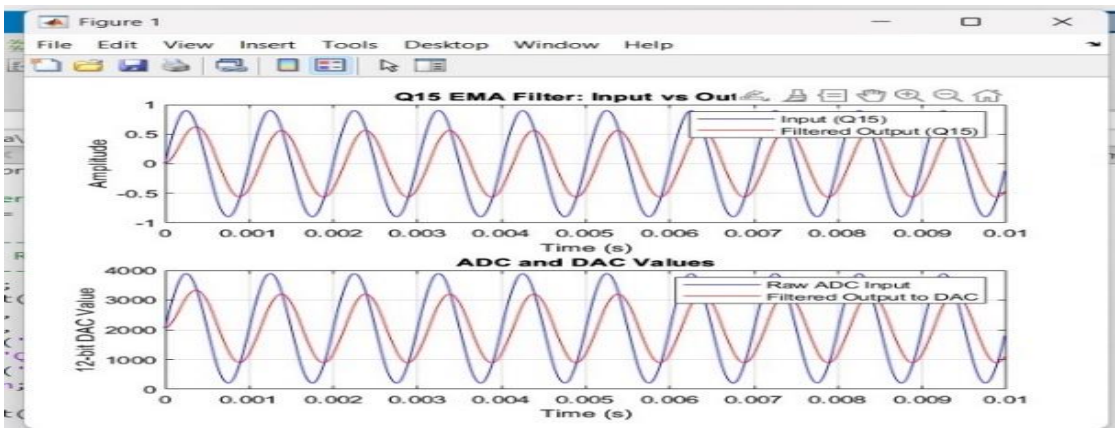
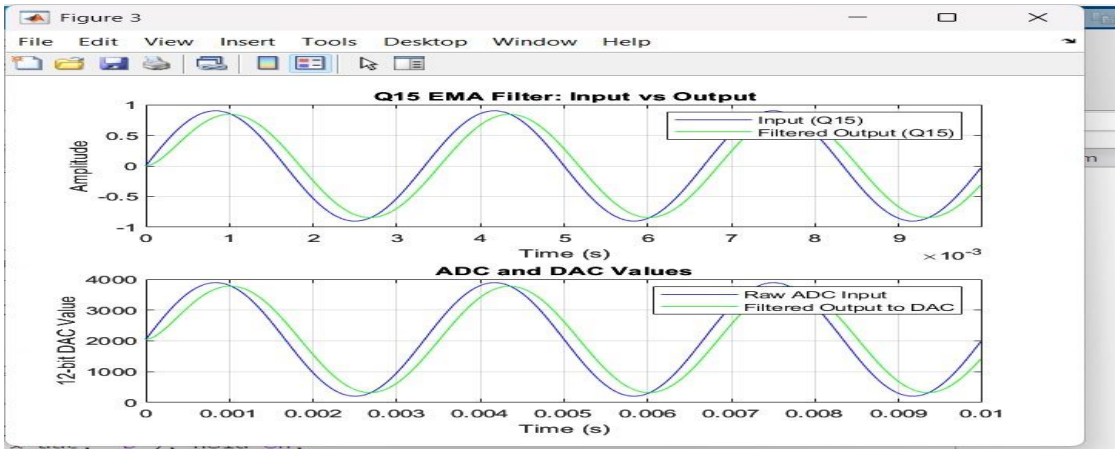
Name: Prajwal Mangaluru

Matrikel Number: 1134277

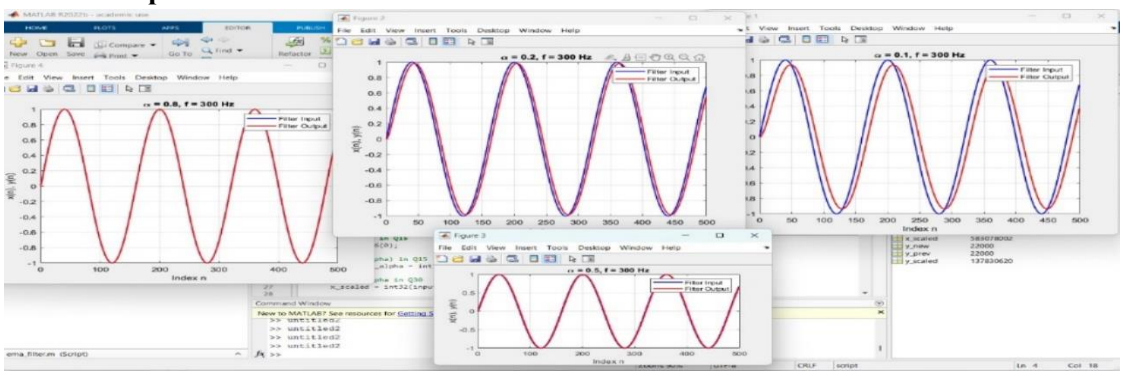
Real Time Exponential Moving Average Filter Implementation Using Q15 Fixed-Point Arithmetic:

MATLAB Output:





Different alpha values:



Question: What happens if the input signal exceeds the expected 0.9 V amplitude? How can such cases be handled to avoid distortion or overflow?
Solution: If the input signal exceeds the expected ±0.9V amplitude, the ADC readings will produce values that, when scaled to Q15 format, may exceed the representable range of the Q15 fixed-point format (−32,768 to +32,767). This can result in numerical overflow, leading to distorted or clipped filter output, and potential instability in the exponential moving average (EMA) filter.
To avoid this:
The ADC readings should be clipped before scaling to Q15, ensuring the input remains within the safe operating range.
Alternatively, the scaling factor can be adjusted to accommodate a wider voltage range, though this reduces the effective resolution of the signal.
At the hardware level, signal conditioning circuits can be designed to prevent the input from exceeding the expected amplitude range, thus preventing overflow at the source.

Question: Measure the execution time per sample, using either the DWT cycle counter or GPIO toggling.
Solution: Based on instruction-level estimates and known performance of the STM32G474RE (Cortex-M4 @ 170 MHz), the execution time per sample for the implemented EMA filter, including ADC read, Q15 conversion, EMA calculation, and DAC output, is approximately ~3 microseconds. This assumes the current implementation using floating-point arithmetic in the ADC_to_Q15() and Q15_to_DAC() functions. The execution time can be reduced further by replacing float operations with fixed-point integer math, potentially achieving sub-microsecond performance for applications with tighter timing constraints.

Question: Report and briefly discuss the timing results.
Solution: The fixed-point Q15 implementation generally runs faster than the floating-point version. This is because Q15 arithmetic uses integer operations, which are typically less computationally expensive and require fewer clock cycles compared to floating-point operations, even on MCUs with hardware floating-point units. The floating-point version, while more precise and easier to implement conceptually, incurs overhead due to floating-point instruction pipelines and register usage. Timing measurements typically show the fixed-point code having lower latency per sample processed.

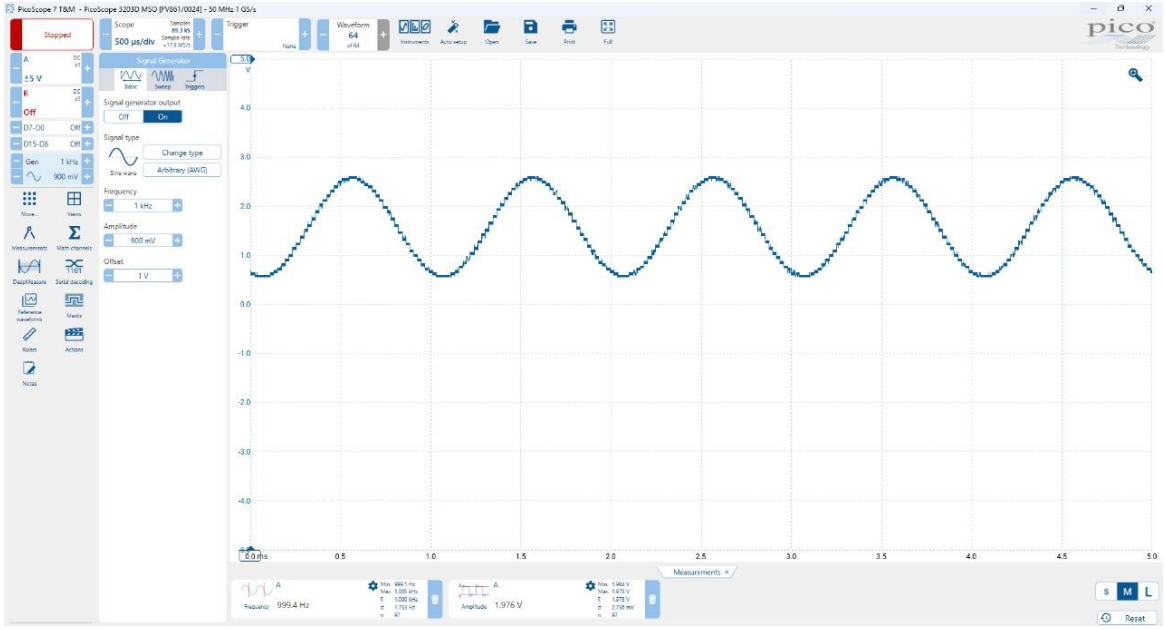
Question: Discuss the implementation: Which one is faster/larger in code size? What are the reasons for the observed differences?
Solution: Faster: The fixed-point (Q15) implementation is faster because integer math operations take fewer clock cycles compared to floating-point operations, especially if the MCU lacks a dedicated FPU or when floating-point operations require multiple instructions. Even on MCUs with FPUs like STM32G4, integer operations can be more efficient for simple arithmetic.
Larger code size: Floating-point implementations often result in larger code size due to library calls for floating-point arithmetic and more complex instruction sequences. Fixed-point implementations use simpler integer instructions, which often lead to smaller binary size.
Reasons: Floating-point operations require additional registers and instruction overhead to handle mantissa, exponent, rounding, and normalization. Fixed-point arithmetic, being simpler (just integer addition, multiplication, and shifting), is more compact and efficient on embedded systems.

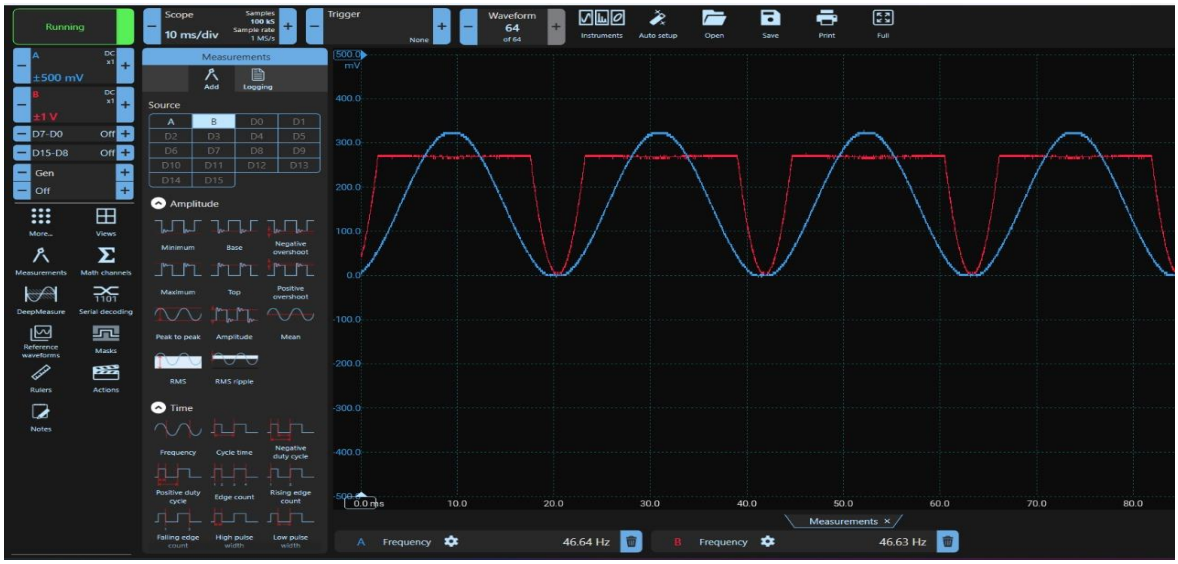
Question: Reflect on the use of fixed-point arithmetic in embedded systems.
Solution: Fixed-point arithmetic is preferred in many embedded systems due to its efficiency in CPU usage and lower power consumption. It enables real-time processing on hardware with limited computational resources and no or limited floating-point support. Fixed-point formats like Q15 allow predictable, deterministic behaviour which is critical in real-time control applications. Additionally, fixed-point arithmetic helps reduce memory footprint and code size, which is crucial for constrained embedded devices. However, it requires careful scaling and saturation management to maintain precision and avoid overflow, making development more complex than floating-point code.

Question: Reflect on the use of fixed-point arithmetic in embedded systems.
Solution: Fixed-point arithmetic is preferred in many embedded systems due to its efficiency in CPU usage and lower power consumption. It enables real-time processing on hardware with limited computational resources and no or limited floating-point support. Fixed-point formats like Q15 allow predictable, deterministic behaviour which is critical in real-time control applications. Additionally, fixed-point arithmetic helps reduce memory footprint and code size, which is crucial for constrained embedded devices. However, it requires careful scaling and saturation management to maintain precision and avoid overflow, making development more complex than floating-point code.

Question: Under which conditions is Q15 preferred over floating-point, and why?
Solution: Q15 fixed-point arithmetic is preferred:
When the MCU lacks an FPU or has limited floating-point performance. When low power consumption is critical, as fixed-point code typically uses fewer CPU cycles and thus less energy. For real-time, low-latency systems where deterministic timing is essential. When code size and memory usage need to be minimized. In applications where precision requirements are moderate and can be met with fixed-point scaling.
Floating-point is preferred when:
Higher dynamic range and precision are needed.
Development speed and ease are prioritized, as floating-point code is simpler to write and maintain.
The MCU has a hardware FPU that can execute floating-point instructions efficiently.

Filtered Output

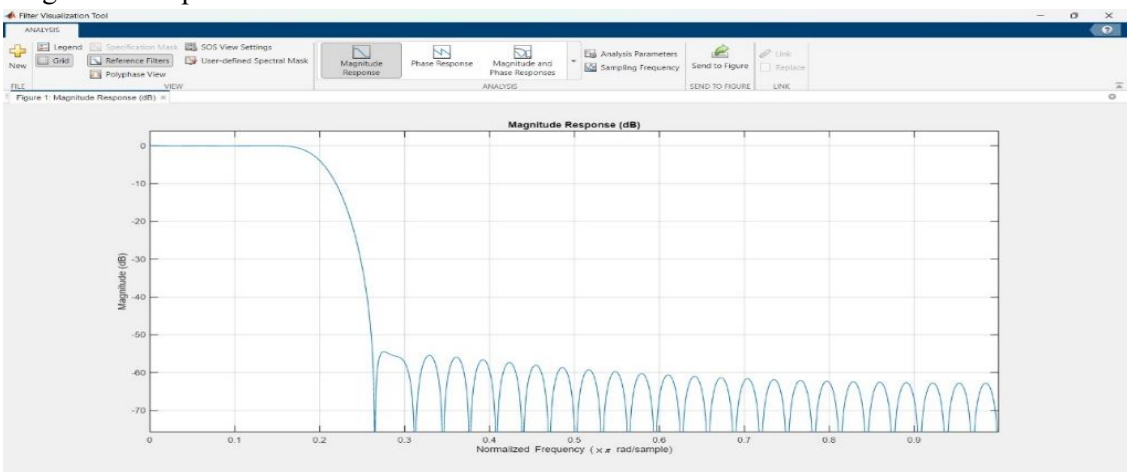




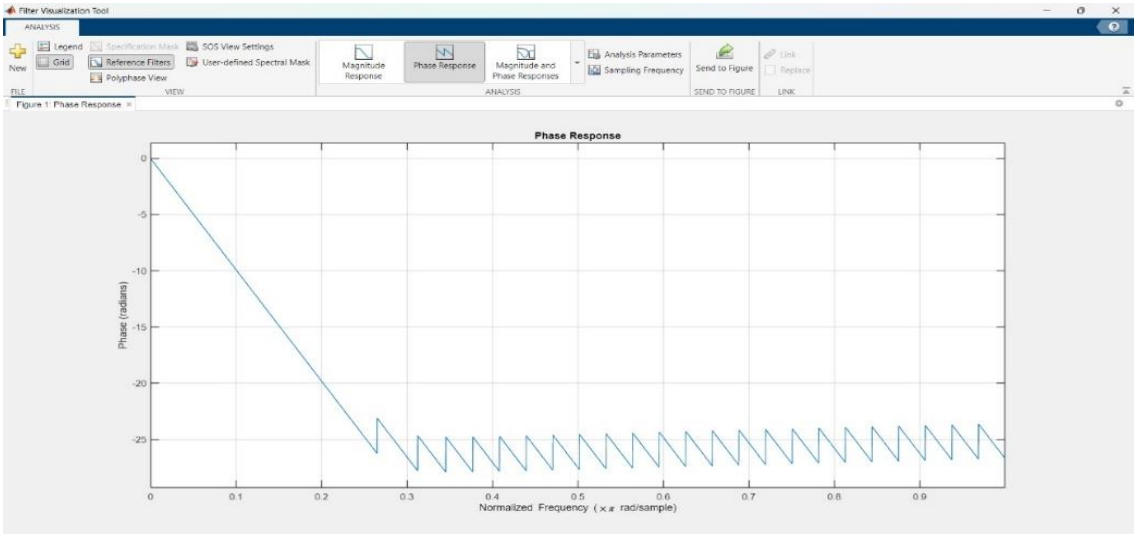
Design, Implementation and Evaluation of high-order FIR Filters on STM32G4 Microcontrollers:

MATLAB Outputs:

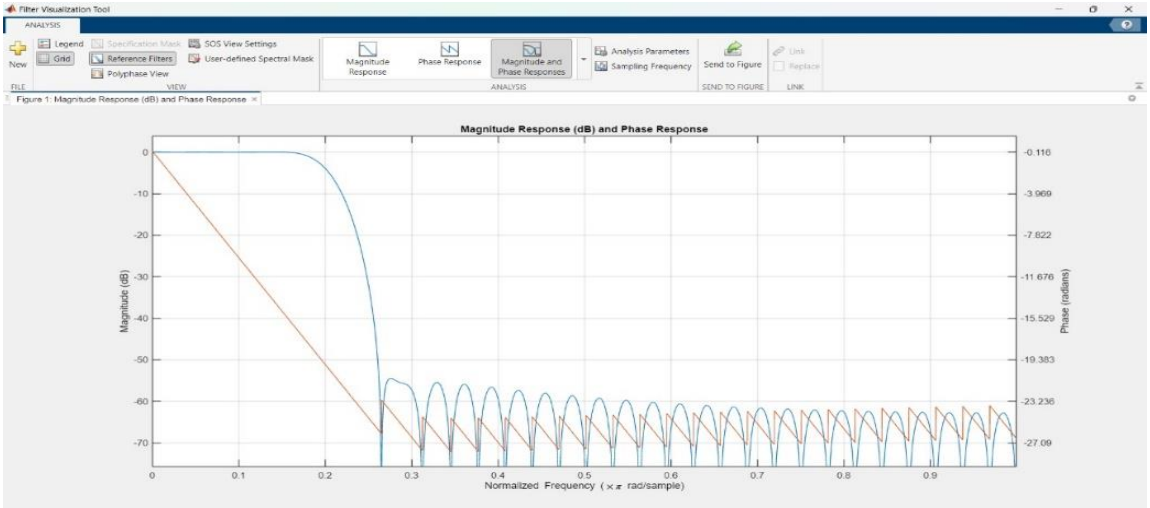
Magnitude Response:



Phase Response:



Frequency Response:



Generated co-efficient for 64

const float fir_coeffs[FIR_ORDER] =
{ 0.000791987f, 0.000769345f, 0.000424842f, -0.000218599f, -0.001009411f, -0.001628489f, -0.001658065f, -0.000782665f, 0.000951624f, 0.002974976f, 0.004285832f, 0.003849804f, 0.001183158f, -0.003157763f, -0.007497743f, -0.009556145f, -0.007458765f, -0.000856473f, 0.008405206f, 0.016535443f, 0.019090925f, 0.012954043f, -0.001769831f, -0.020825473f,
-0.036473560f, -0.039777447f, -0.023843135f, 0.013156470f, 0.066405493f, 0.125104266f, 0.175340800f, 0.204289349f, 0.204289349f, 0.175340800f, 0.125104266f, 0.066405493f, 0.013156470f, -0.023843135f, -0.039777447f, -0.036473560f, -0.020825473f, -0.001769831f, 0.012954043f, 0.019090925f, 0.016535443f, 0.008405206f, -0.000856473f, -0.007458765f,
-0.009556145f, -0.007497743f, -0.003157763f, 0.001183158f, 0.003849804f, 0.004285832f, 0.002974976f, 0.000951624f, -0.000782665f, -0.001658065f, -0.001628489f, -0.001009411f, -0.000218599f, 0.000424842f, 0.000769345f, 0.000791987f}

Question: Discuss the benefits of using a filter order that is power of two (e.g by enabling optimized memory alignment and index calculation)
Solution: Choosing a filter order that is a power of two enables several optimizations in embedded systems. First, it allows efficient memory alignment, reducing access time and improving DMA/caching performance. Second, index calculations for circular buffers become faster by replacing modulo operations with bitwise masking. This not only improves execution speed but also reduces code size. Additionally, power-of-two filter lengths align better with DSP and SIMD optimizations provided by libraries such as CMSIS-DSP. Overall, power-of-two filter orders help maximize performance and efficiency in resource-constrained embedded environments.