Analog-based Digital Low-Pass Filter Realization for Mains Noise Using Arduino Mega 2560 Default Parameters

Eduardo J. Pérez, Alejandro J. Del Cid, Pat H. Rodríguez
Department of Telecommunications Engineering
Universidad Tecnológica Centroamericana (UNITEC), Honduras

***Abstract*—In environments requiring remote monitoring, distance between microcontrollers and sensors leads to noisy measurements, particularly mains power noise. Noise mitigation in DC measurements can be achieved through digital low-pass filtering at code level. Design and performance of such filters depends of parameters such as sampling frequency, precision, and available memory, meaning the Arduino has specific design constraints. Specifications of clock speed, floating point precision and memory were determined, allowing the design of recursive filters based on analog parallels. High order filters were deemed unrealizable at low cut-off frequencies due to precision, whereas fewer poles required excessive memory, due to which designers must determine allowed variations and memory parameters during implementation. The procedure followed can be generalized to different frequency, response, precision and memory constraints.**

Keywords— Arduino Mega 2560, Digital Filter, Recurrence Coefficients, Sampling Frequency, Floating Point Precision, Memory.

I. Introduction

Remote monitoring may be a necessity in environments where limited access to microcontrollers requires the definition of a reduced amount of control units, such as hobbyist projects. In these cases, the distance between the microcontroller and the measurement point has an impact on the measurements when covering the span with electrical conductors. The conductor length increases susceptibility to noise, particularly to the ubiquitous mains power noise. The result is a superposition of the desired signal with AC components of varied frequencies. This compels designers to implement filters; however, the signal under measurement may have a frequency similar to that of the noise source, meaning that filtering must be implemented through software in order to mitigate noise selectively. This introduces new constraints that depend on the microcontroller involved in the system.

A particular case is that of the Arduino Mega 2560, which allows interaction with an ATmega2560 microcontroller through a proprietary interactive development environment and C++ coding [1]. When dealing with analog DC measurements, signal filtering can be achieved through digital low-pass filters at code level by processing discrete data values collected by the microprocessor. Such filters can be designed through different methods, one of which parts from analog circuitry with desirable properties. The filtering process is then limited by the clock speed, precision, and memory specifications of the onboard processor, providing specific parameters for digital filter design involving the Arduino Mega 2560.

II. Objectives

A. General Objective

• To study the realizability of analog-based digital low-pass filters intended for mains noise mitigation in the Arduino Mega 2560

B. Specific Objectives

• To determine the design constraints involved in filter design for the Arduino Mega 2560

• To compare the results of different order and cut-off frequency filters in terms of required memory and output variations

• To provide a replicable and modifiable procedure for designing and evaluating low-pass digital filters

III. Methodology

Measurements were taken using: Arduino Mega 2560 (henceforth, Arduino), PCE-SDS1022DL Oscilloscope, N1996A Spectrum Analyzer, 1μF Capacitors. The experimental setup was carried out at the "Universidad Tecnológica Centroamericana" (UNITEC) Networks Lab, consisting of a 75.6 m unshielded twisted pair cable with 14 connector-socket pairs folded into 2 loops, interconnecting the Arduino and a sensor with an output of voltage of 0.21 V. The setup is show in Fig. 1. Calculations and preliminary simulations were carried out in Matlab.

The sampling period of the Arduino was determined theoretically by studying the ATmega2560 manual, and corroborated through benchmarks found in the Arduino forums. For the specific Arduino used, the mean sampling period was determined. To reduce probe effect introduced by code commands other than sampling, a single sample was defined as the time taken to measure 960 values. A pilot test of 100 such samples yielded a standard deviation of 0.009 ms.

Universidad Tecnológica Centroamericana (UNITEC), Honduras

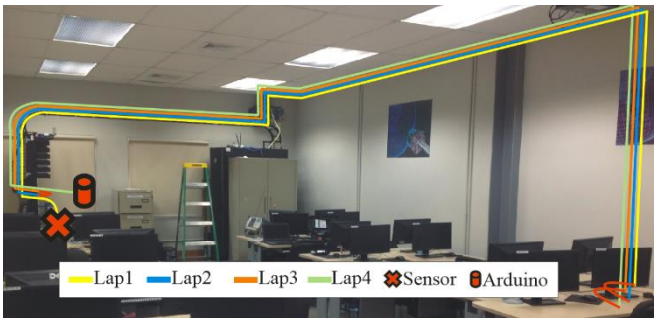

Fig. 1. *Experimental Setup in Networks Lab caption*

Choosing a margin of error of 0.001 ms, the mean sampling period was determined with 99% certainty.

Digital filter design was centered on the calculation of recurrence coefficients. The process involved analog filter design through Laplace transform, inverse Laplace transform to obtain the time domain transfer function, discretization of the time domain based on sampling period, Z transform and gain normalization. Subsequently, the resulting filters were simulated on a sample noisy signal prior to implementation in the experimental setup.

IV. Description

When measuring analog DC signals with the Arduino, the goal is to obtain significant information from the values collected. This becomes difficult when noise comes into play, as is the case of sensors connected to a microcontroller by means of a span of conductor. A setup consisting of a 75.6 m unshielded twisted pair connecting an Arduino and a sensor with a 0.21 V allows the measurement of the superposed noise signal. A noise of 30 mV peak maximum is present in the experimental setup, representing an error of ± 14% in the measurement. By inspection, the shape of the noise follows that of a 60 Hz sinusoid, as expected from the spectrum analyzer output of -10 dBm at 60 Hz in the Networks Lab, as shown in Fig. 2. The information of interest can be obtained through filtering.

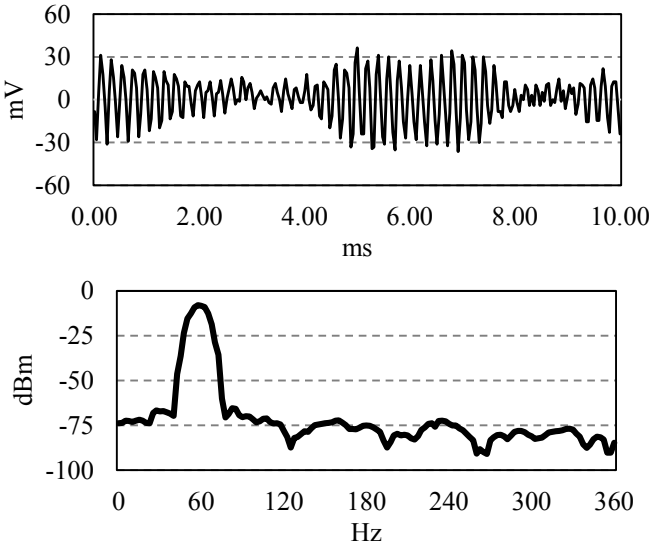

Fig. 2. *Experimental Setup in Networks Lab caption*

The sampling theorem states that the maximum frequency that can be unambiguously characterized based on the sampled values of a signal depends on the sampling frequency used [2]. For the Arduino, the sampling frequency is dictated by a prescalar in the Analog to Digital Converter (ADC) in the ATmega2560 with a default value of 128 [3]. The ADC is specified to take 13 clock cycles for successive approximation in the default single conversion mode, plus a minimum of 1 clock cycle before commencing the next measurement [3]. From [1], [2], and [3], the Arduino follows that

$$\frac{16*10^6}{128*14}\text{Hz} > 2f_m \text{ Hz}. \quad (1)$$

From (1), the Arduino sampling frequency is approximately 8929 Hz, characterizing signals up to a maximum frequency f_m of approximately 4465 Hz. This corresponds to the findings in [4], where the sampling period is measured as 112.01 µs. For the particular Arduino used, the mean period for measuring 960 samples is 107.528 ms with 99% certainty, equivalent to 8928 Hz with an error of 0.011% with respect to the theoretical value, due to which a frequency of 8929 Hz is used.

With (1) in mind, a 1µF capacitor is placed parallel to the ADC input. This value is determined by defining the desired reactance at the frequency of interest; in this case, finding the capacitor that provides a reactance with a magnitude close to 30 Ω at 4465 Hz, decoupling signals at this frequency and higher, which satisfies (1) and attenuates other AC signals under 4465Hz. Samples collected by the Arduino at this point represent an unambiguous signal that can be treated with digital filters.

Digital filters can be implemented by programming their characteristic difference equation [5]:

$$y[n] = a_0x[n] + a_1x[n-1] + \cdots + b_1y[n-1] + b_2y[n-2] + \cdots \quad (2)$$

Values a_i and b_j are the recurrence coefficients of the filter, assigning a weight to the components $x[n-i]$ and $y[n-j]$, which represent the value of the input and output, respectively, i and j samples ago, where $i = 0,1,2,\ldots$ and $j = 1,2,3,\ldots$. This equation accounts only for current and past values of $x[n]$ and $y[n]$, due to which filters of this form are causal and physically realizable [2]. Design of causal filters revolves around the calculation of adequate coefficients, producing a desired behavior when applied to a discrete time signal, such as ADC measurements in the Arduino.

Recurrence coefficients can be taken from the z transform of the filter's transfer function, which is equivalent to taking the z transform of both sides of (2) and solving for the transformed output divided by the transformed input [5]:

$$\frac{Y[z]}{X[z]} = \frac{a_0+a_1z^{-1}+a_2z^{-2}+\cdots}{1-b_1z^{-1}-b_2z^{-2}+\cdots} \quad (3)$$

In (3), $Y[z]$ and $X[z]$ are the respective z transforms of the input and output, and coefficients a_i and b_j are the same recurrence coefficients as in (2). Since (2) and (3) are causal, a physical filter can be made to have the same transfer function $H[z] = \frac{Y[z]}{X[z]}$. This transfer function can be obtained by taking the Laplace transform of a time domain transfer function $h(t)$ after it has been sampled, represented by $h^*(t)$, yielding $H^*(s)$ [2]. The transfer function $H[z]$ can be obtained from $H^*(s)$ through the substitution $z = e^{sT}$ [2] with prior knowledge of sampling period T.

A continuous system can be analyzed through the Laplace transform to obtain a transfer function $H(s)$ and a corresponding $h(t)$. This time domain function can be discretized by taking samples every T seconds, from which $h^*(t)$ and $H[z]$ can be found. This procedure allows the calculation of recurrence coefficients so that (2) is a discrete representation of a system with transfer function $H(s)$, shifting the focus onto common analog filters.

It is desired that the output of the filter remains the same as the input for the frequency band of interest; in this case, frequencies under 60 Hz. A Butterworth response filter achieves a maximally flat gain below the cut-off frequency [6], solving the problem by using a unit gain; additionally, the topologies selected as a basis for $H(s)$ are a simple RC combination for one pole, a Sallen-Key topology for two poles, and a cascaded combination of both for 3 poles, finding the appropriate gain value for the two pole Sallen-Key Butterworth response in [6]. Transfer functions $H(s)$ for these low-pass analog filters are found to be:

$$H_1(s) = (1 + s\beta)^{-1} \quad (4)$$

$$H_2(s) = (1.586^{-1}(1 + s\beta)(2 + s\beta) - 1.586^{-1} - (s\beta))^{-1} \quad (5)$$

$$H_3(s) = H_2(s)H_1(s) \quad (6)$$

$$\beta = RC = (2\pi F_c)^{-1} \quad (7)$$

In these equations, index n in $H_n(s)$ denotes the number of poles in the filter. This representation was derived with simplicity of input into mathematical software in mind. The symbol β substitutes the product RC as the specific component values are unimportant; rather, filter cut-off frequency F_c is emphasized.

Though any 1, 2 or 3 pole low-pass filter may be designed through (4-7), care must be taken to ensure that $F_c < 4465\ Hz$, limit imposed by the Arduino's default characteristics following from (1). Sampling period T must be fixed to 8929^{-1} s when discretizing the continuous time transfer function $h(t)$ obtained from $H(s)$.

$H[z]$ may be obtained by successive transformations following the sequence $H(s) \rightarrow h(t) \rightarrow h^*(t) \rightarrow H[z]$, or directly $H(s) \rightarrow H[z]$ through software aid or by use of transform tables. Once $H[z]$ is obtained, a few modifications must be made. In order for the expression to match (3), the numerator and denominator must be divided by the highest power of the z variable to produce negative exponents; they must also be divided by the constant value in the denominator to ensure a leading value of 1 followed by negative exponents of z.

The expected output of a recurrent filter built from (3) and (2) should be equal to the input when the frequency is within the range of interest. To ensure this, the filter's gain must be normalized. This procedure can be done by taking the preliminary expression of the recurrent filter of form (2) and enforcing $x[n-i] = 1$, $y[n-j] = G$, and $y[n] = G$, and then solving for G, where G is the filter's gain [5]. If $G \neq 1$, then a_i must be substituted by $\frac{a_i}{G}$ [5]. Alternatively, to test whether the filter is normalized, assume all inputs and outputs equal to 1. If (2) holds true under this assumption, the filter is normalized; otherwise, the previous procedure must be carried out.

Recursive filters intended for implementation on the Arduino have constraints other than sampling frequency. The floating point precision in Arduino uses 4 bytes (henceforth B) and allows for up to 7 digits aside from exponents [7]. The limitation comes from the lack of double precision variables, meaning that variables declared as doubles have the same characteristics as floats [8]. Filters with certain coefficient combinations are subject to rounding error when coded into the Arduino, whereas environments with higher precision will only experience this in more extreme cases.

The final limitation present in the Arduino environment is memory. Arduino boards using the ATmega2560 have 8 kB of SRAM memory for variable storage [9]. A maximum of 4000 integers may be stored at a time, based on their size [10]; this scenario is not realistic as other variables must be stored in usual codes, limiting the available memory further. Recurrent filters require different amounts of samples to settle into a steady state. To reduce jitter and deviation from the mean clock speed of the microcontroller, all samples should be taken consecutively and stored prior to application of the filter, and because of this, filters requiring large amounts of samples to reach their final state are not realizable in the Arduino.

V. Results

Fixing $T = 8929^{-1}$ s and using (4-7) along with consecutive transformations $H(s) \rightarrow h(t) \rightarrow h^*(t) \rightarrow H[z]$, recurrence coefficients were calculated for filters of first, second and third order, with cut-off frequencies of 1 Hz, 10 Hz, 20 Hz, 30 Hz and 40 Hz. Many of these filters were determined to be unrealizable in the Arduino mainly due to floating point precision. Coefficients for realizable filters are shown in Table I rounded to 6 decimal places; an explanation for this rounding is provided below. Notation xi and yj in Table I corresponds to $x[n-i]$ and $y[n-j]$.

To determine whether a filter can be realized or not in the Arduino, the order of the coefficients must be examined. As only 7 digits are allowed, coefficients that differ in magnitude by several powers of 10 will be rounded to 0. The following criterion can be used to evaluate the low-pass filters found:

$$\lfloor Log_{10}(|b_m|) \rfloor - \lfloor Log_{10}(|a_i|) \rfloor > 7 \ \forall i, i = 0,1,2, \ldots. \quad (8)$$

TABLE I. RECURRENCE COEFFICIENTS

Cut-off Frequency	1 Pole	2 Poles	3 Poles
1 Hz	0.000703*x+0.999297*y1	NA	NA
10 Hz	0.007012*x+0.992988*y1	0.000049*x1+1.99005*y1-0.990099*y2	NA
20 Hz	0.013975*x+0.986025*y1	0.000197*x1+1.980100*y1-0.980297*y2	NA
30 Hz	0.020889*x+0.979111*y1	0.000439*x1+1.970152*y1-0.970591*y2	NA
40 Hz	0.027755*x+0.972245*y1	0.000776*x1+1.960205*y1-0.960981*y2	0.000011*x1+0.000011*x2+2.93245*y1-2.866781*y2+0.934309*y3

In (8), b_m is the recurrence coefficient with the largest magnitude in the denominator of $H[z]$. When (8) is satisfied, all numerator coefficients will round to 0 and the filter cannot be realized on Arduino. If the result is exactly equal to 7, care must be taken to see whether the coefficients will round up or down. When the result is less than 7, recurrence coefficients will be nonzero, but normalization testing is necessary.

The third order filter with cut-off frequency of 30 Hz is a case of an unrealizable filter due to normalization. The expression for the filter is:

$$y[n] = 0.000005x[n-1] + 0.000005x[n-2] + 2.949263y[n-1] - 2.899588y[n-2] + 0.950316y[n-3]. \quad (9)$$

Repeated attempts at normalization yield $1 \neq 1.000001$ and $1 \neq 0.999999$, oscillating about 1. This alters the output by multiplying a gain different from 1, making it unrealizable in Arduino. A special case can be made for modifying the coefficients of $x[n-i]$ arbitrarily to obtain a sum of 1, though this would require analysis of the frequency response for any modifications in the cut-off frequency of the filter.

Implementation of filters in Table I yielded the results in Fig. 3-5. There is a tradeoff between performance and order of poles. Filters with more poles reach a steady output in fewer samples and therefore require less memory; however, as shown in Table I, more poles leads to smaller recurrence coefficients in the numerator of $H[z]$. The size of these coefficients increases with the cut-off frequency, so higher order filters are realizable at higher cut-off frequencies. Resulting oscillations are not evident in Fig. 3-5, due to which simulation with noisy signals is advised. The utility of a bypass capacitor to limit the frequency of signals prior to sampling is shown in Fig. 5.

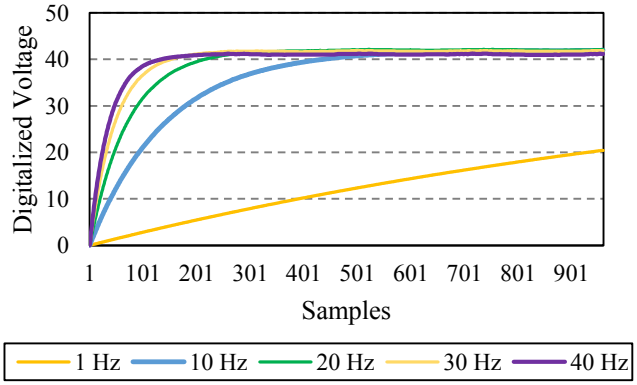

Fig. 3. *First order filter output*

The suggested implementation of these filters is iterative rather than recursive. The motivation is that recursion elicits greater memory usage by creating copies of variables involved in calculations; in contrast, iterative implementations can be realized by means of a single swap variable. In the case of the analog-based low-pass filters discussed, a loose upper bound for realization can be found based on filters requiring the fewest variables, corresponding to first order. These filters require 2 recurrence coefficients, the present input, a previous output, and a number of input samples to be cycled through in order to obtain a steady output. All input measurements are stored first as integers and are casted to floats during application, meaning that an array of integer inputs is needed along with 5 floats for input, output, previous output, and recurrence coefficients.

$$M = 4 * 5 * p + 2 * p \quad (10)$$

In (10), M represents available memory in bytes, and p is the number of input samples needed, as well as cycles necessary to reach steady output. The first term represents 5 floating point variables of 4 bytes each with p copies for output calculation, while the second shows an array of integers of size p for input samples. Substituting the Arduinos SRAM, p is found to be approximately 363 cycles and samples, eliminating most of the filters in Table I.

$$M = 4 * 5 + 4 + 2 * p \quad (11)$$

Variables in (11) have the same meaning as in (10). Here, the first term represents 5 floats, the second term is an additional float for swapping values, and the last term is the same as in (10). Substituting M yields p of approximately 3988, meaning iterative implementations have a higher cycle limit for realization. Usage of a large input array can be circumvented by performing calculations immediately after each measurement, though this might affect the mean sampling frequency and produce jitter. In this case, measurements should be carried out to characterize sampling frequency in the specific application.

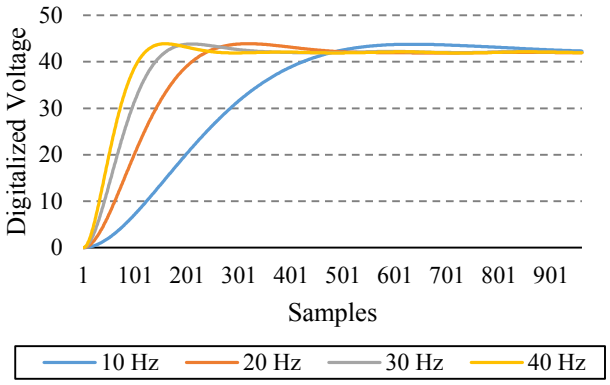

Fig. 4. *Second order filter output*

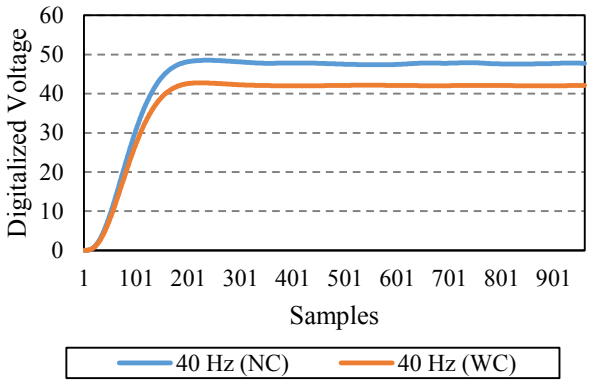

Fig. 5. *Third order filter output*

VI. Conclusions

- The Arduino Mega 2560 is limited by default parameters of 8929 Hz sampling frequency, a maximum of 4000 samples at a time, and 7 digit precision. This limits which signals can be characterized completely by the Arduino to 4465 Hz and requires usage of physical filters. Size of recurrence coefficients is limited through (8), and the number of samples allowed before reaching a steady output are limited to values under 4000 by the surrounding code that applies the filtering.
- Due to constraints in the Arduino environment, filters must be selected based on design requirements. A smoother response is obtained by using lower cut-off frequencies, while a steady output is achieved with less memory at higher frequencies. At the same time, memory usage can be decreased by increasing the number of poles in filters; however, this reduces the size of recurrence coefficients and can thus be achieved only at higher cut-off frequencies. Selection of cut-off frequency and filter order is a function of available memory and allowed variations of output.
- By means of (4-8) and use of transform tables or mathematical software, low-pass filters aimed at DC measurements can be designed for the Arduino with its default parameters. Values in Table I are ready for usage in Arduino code. Additionally, this procedure is applicable to different sampling frequencies, as is the case where the ADC prescalar is modified, by substitution of the appropriate sampling period. Determining realizable filters can be done through (8) in environments with double precision by substituting an appropriate margin in the inequality. Furthermore, filters other than low-pass can be designed in the same fashion for Arduino and other environments if corresponding expressions for $H(s)$ are found. Usage of intuitive, physically realizable analog filters as basis for recursive digital filter is meant to facilitate intuitive design, along with simple test cases to determine if such a filter is realizable in Arduino.

References

[1] Arduino. "Arduino MEGA 2560 & Genuino MEGA 2560," Arduino. [Online]. Available: https://www.arduino.cc/en/Main/ArduinoBoardMega2560. [Accessed: May 1, 2016].

[2] B. C. Kuo, Sistemas de Control Digital [Digital Control Systems]. Mexico: Grupo Editorial Patria, 2011.

[3] Atmel Corportaion (2014, Feb.). Atmel ATmega640/V-1280/V-1281/V-2560/V-2561/V: 8-bit Atmel Microcontroller with 16/32/64KB In-System Programmable Flash, pp. 268-288. [Online]. Available: http://www.atmel.com/devices/atmega2560.aspx. [Accessed: May 2, 2016].

[4] Riva (2015, Jun. 2). Microcontroller I/O & ADC Benchmarks. [Forum]. Available: http://forum.arduino.cc/index.php?topic=326944.0. [Accessed: May 2, 2016].

[5] S. W. Smith, "The z-Transform", in The Scientist and Engineer's Guide to Digital Signal Processing, pp. 605-630. [Online]. Available: http://www.dspguide.com/ch33.htm. [Accessed: May 7, 2016].

[6] J. Karki, "Active Low-Pass Filter Design", Texas Instruments Application Report, pp. 9-10, Sep. 2012. [Online]. Available: http://www.ti.com/lit/an/sloa049b/sloa049b.pdf. [Accessed: May 6, 2016].

[7] Arduino. "Float," Arduino. [Online]. Available: https://www.arduino.cc/en/Reference/Float. [Accessed: May 20, 2016].

[8] Arduino. "Double," Arduino. [Online]. Available: https://www.arduino.cc/en/Reference/Double. [Accessed: May 20, 2016].

[9] Arduino. "Memory," Arduino. [Online]. Available: https://www.arduino.cc/en/Tutorial/Memory. [Accessed: May 23, 2016].

[10] Arduino. "Int," Arduino. [Online]. Available: https://www.arduino.cc/en/Reference/Int. [Accessed: May 20, 2016].