

# Real Time Feature Extraction of Autoregression Coefficients for Digitized Signals Using the CMSIS DSP Library in ARM Cortex M4/M7 MCUs

Mwangi Alex W.

October 10, 2024

---

## Introduction

The principle of autoregression stipulates that any value within a time series dataset can be predicted from its past values given that the dataset is not random in nature. This is to imply that each value in a non-random dataset has a dependency on its past values within the dataset and that quality is what we refer to as autoregression. The autoregression of a dataset can be quantified through an autoregression model that accepts the data within the dataset as inputs and produces coefficients that reflect the autoregression of the dataset. By studying the determined coefficients, we can generally examine how each value within the dataset depends on its past values (the dataset's predictability). This autoregression feature can be adopted not only to examine predictability but to gather other insights and information about the dataset. In this article I intend to extract the autoregression coefficients for sEMG signals in real time for a DAQ board I designed. It happens that an sEMG signal's autoregression coefficients helps us understand various characteristics of muscles such as muscle fatigue and diagnostic classification and can be fed to machine learning(ML) algorithms for motion planning, muscle health assessment and gesture recognition. The concept of autoregression can however be adopted in various other signals such as audio signals for speech recognition, mood recognition etc. In embedded systems applications, when most signals are digitized and their data stored in memory, the data stored fits the description of time series data. Further, if this data is stored in buffers, the instantaneous buffer values fit the description of datasets hence these signals can be subjected to autoregression models which would offer insights about segments of the signals (segment size determined by the buffer size) that would otherwise be indeterminable through direct observation. Such is the advantage of digitization. Before I wrote a single line of code, I first focused on understanding the concept of autoregression mathematically. The task at hand can be divided into two steps; 1. Calculating the autocorrelation values for a chosen dataset. 2. Using the calculated autocorrelation values to compute the autoregression coefficients for the given dataset. The mathematics behind this two steps will be explained later. After understanding the mathematics, I went ahead to relate it to the context of my project. I was developing firmware for an sEMG signals data acquisition unit (DAQ). The data for which I intended to study its autoregression was from a buffer that stored the outputs of a moving average filter implemented still within the DAQ's microcontroller which in this case was an STM32G4. The buffer was of size 256 and stored instantaneous datasets as it was a circular buffer. I therefore needed to compute the autoregression coefficients each time after the buffer was filled. The second consideration was that since the mathematical computations of the model were rigorous as they contained high order matrix manipulations, I had to develop or employ optimized functions that would optimize CPU cycle economy and memory usage during computations hence minimize power consumption and save on time. The CMSIS DSP functions offer exactly that—a library that contains optimized functions for various applications including dot products and matrix manipulations. For that reason, I sort out the functions that I would use as will be presented later in the programming section. These functions can be utilized in arm cortex M4/M7 MCUs hence the code I develop in this article can be exported to other MCUs that utilize the arm cortex M4/M7 with minimal editing. After developing the code, checking for any possible syntax or logical errors and successfully compiling it, I had to validate it. No better to validate the model than to put it to test in real life. Currently, I am in this phase and as soon as the model is tested, the results will be updated on the Github repo for this article [here](#).

## The Mathematical Model

This section explains the concepts of autoregression from a mathematical perspective that will be paramount in the code development phase. In an autoregression model, the current value in a time series dataset can be predicted from its previous values. An autoregression model of order  $p$  can be written as follows;

$$X_t = c + \phi_1 X_{t-1} + \phi_2 X_{t-2} + \dots + \phi_p X_{t-p} + \epsilon_t$$

where:

- $X_t$  is the value of the time series at time  $t$ ,
- $c$  is a constant (intercept term),
- $\phi_1, \phi_2, \dots, \phi_p$  are the autoregressive coefficients,
- $X_{t-1}, X_{t-2}, \dots, X_{t-p}$  are the past values of the time series,
- $\epsilon_t$  is the white noise or error term at time  $t$ .
- $p$  is the order of the model (How far the model looks into the past).

To calculate the predicted values however, we need the coefficients. The AR coefficients are based on the autocorrelation of the time series data. Given a time series dataset, the autocorrelation at lag  $k$  is defined as:

$$R_k = \frac{1}{N} \sum_{t=k+1}^N X_t X_{t-k}$$

where:

- $X_t$  is the time series data at time  $t$ ,
- $N$  is the total number of data points,
- $R_k$  is the autocorrelation at lag  $k$ .

This equation is computed using a dot product CMSIS DSP function within the MCU as we shall see later in the programming section. The autocorrelation values are computed up to order  $p$ . Roughly, correlation values quantify the extent to which each particular value within a dataset correlates with their previous  $k$ th term where  $k$  is the lag. Now that we have the autocorrelation values, we determine the autoregression coefficients from the Yule-Walker Equations. The Yule-Walker equations can be written as;

$$\begin{aligned} R_0 \cdot \phi_1 + R_1 \cdot \phi_2 + \dots + R_{p-1} \cdot \phi_p &= R_1 \\ R_1 \cdot \phi_1 + R_0 \cdot \phi_2 + \dots + R_{p-2} \cdot \phi_p &= R_2 \\ &\vdots \\ R_{p-1} \cdot \phi_1 + R_{p-2} \cdot \phi_2 + \dots + R_0 \cdot \phi_p &= R_p \end{aligned}$$

In matrix form:

$$\begin{pmatrix} R_0 & R_1 & \dots & R_{p-1} \\ R_1 & R_0 & \dots & R_{p-2} \\ \vdots & \vdots & \ddots & \vdots \\ R_{p-1} & R_{p-2} & \dots & R_0 \end{pmatrix} \begin{pmatrix} \phi_1 \\ \phi_2 \\ \vdots \\ \phi_p \end{pmatrix} = \begin{pmatrix} R_1 \\ R_2 \\ \vdots \\ R_p \end{pmatrix}$$

This system of linear equations can be solved through matrix inversion to obtain the coefficients since we have all the  $R$  values. The coefficients can be used for various purposes. One is to predict the next value within the dataset and the other is to analyze the autoregression of the dataset by studying the said coefficients. In our case, the goal is the latter. To understand the autoregression of sEMG signal data segments using an AI model in real time.

## Programming

With a sound understanding of the mathematics model behind autoregression, we can now embark on developing the applications in embedded C that are responsible for extracting the autoregression coefficients feature for sEMG signals in real time. I used the STM32Cube IDE as my development environment. I started by creating a header file to declare all the functions responsible for calculating the autocorrelation of data within the moving average buffer (the source buffer) and the functions that cater for coefficients calculation. It is important to state that I calculated the autoregression coefficients for two channels all within the ADC1. That is channel 1 and channel 2. The code presented here therefore runs the autoregression model on both channels. The well commented header file is as shown on Listing 1.

Listing 1: Header file for the autoregression model

```

1  /*
2  *
3  * _ADCn_INx_AR.h
4  *
5  * Created on: Oct 7, 2024
6  * Author: Mwangi Alex. W
7  *
8  * Header file that stores the functions declarations for the autoregression algorithm
9  */
10
11 #ifndef INC__ADCN_INX_AR_H_
12 #define INC__ADCN_INX_AR_H_
13
14 //INCLUDES
15 #include "stm32g4xx_hal.h"
16 #include "arm_math.h" // Header file necessary to run the CMSIS DSP functions
17 #include <stdlib.h>
18
19 //DEFINES
20 #define AR_ORDER 10 // Order for the auto-regression model (how much it looks into the past)
21
22 //FUNCTION DECLARATION
23 void ADC1_IN1_autocorr_calc(void); // Calculates the autocorrelation between data and
    generates auto-correlation values that are stored in a buffer.
24 void ADC1_IN2_autocorr_calc(void);
25
26
27 float32_t* ADC1_IN1_autoreg_coeffs(void); // Calculates the autoregression coefficients from
    the Yule-Walker equations. The coefficients are stored in a buffer. This function returns
    the coefficients array
28 float32_t* ADC1_IN2_autoreg_coeffs(void);
29
30 #endif /* INC__ADCN_INX_AR_H_ */

```

To define the functions, I proceeded to create a .c file as shown in Listing 2

Listing 2: .C file for the autoregression algorithm

```

1  /*
2  *
3  * _ADCn_INx_AR.c
4  *
5  * Created on: Oct 7, 2024
6  * Author: Mwangi Alex. W
7  *
8  *
9  * The auto-regression algorithm is a feature
10 * extraction technique that intends to determine how values within a dataset depends on their
    past within a window that the designer of the algorithm specifies. The number of past
    values (window) chosen determines the order of the algorithm. That way, the algorithm
    accepts time series data as its input arguments and produces coefficients from the Yule-
    walker linear equations that reflect the dependency of a particular value within a data
    set with its past values still within the dataset. The functions developed here
    therefore first of all compute the autocorrelation values within the dataset
11 which in this case is data from the output of a moving average DSP scheme and go ahead to
    compute the coefficients using the determined autocorrelation values
12 */
13
14 //INCLUDES
15 #include "_ADCn_INx_AR.h"
16
17 /*I have only commented for one channel--channel 1.The same holds for the second channel*/
18 //VARIABLES
19 float32_t AutoCorr_1[AR_ORDER + 1]; // Buffer holding R0 to R10, e.g., autocorr_buffer[0] = R0
    , ..., autocorr_buffer[10] = R10
20 float32_t AutoCorr_2[AR_ORDER + 1]; // For the second channel
21
22
23 float32_t AR_Coeffs_1[AR_ORDER]; // Buffer that holds the coefficient values.Should be
    initialized to zero at running before computation
24 float32_t AR_Coeffs_2[AR_ORDER];

```

```

25
26
27 ADC1_IN1_MA AR_ADC1_IN1; // Declaring an instance of autoregression feature
28 ADC1_IN2_MA AR_ADC1_IN2;
29
30
31
32 arm_matrix_instance_f32 ADC1_IN1_YW_mtx; // Creating a Yule-Walker matrix instance for the
    CMSIS DSP matrix initialization function
33 arm_matrix_instance_f32 ADC1_IN2_YW_mtx;
34
35
36
37 arm_matrix_instance_f32 ADC1_IN1_Inv_YW_mtx; // Creating an inverse Yule-Walker matrix
    instance for the CMSIS DSP matrix initialization function
38 arm_matrix_instance_f32 ADC1_IN2_Inv_YW_mtx;
39
40
41 arm_matrix_instance_f32 ADC1_IN1_AC_mtx; // Creating an autocorrelation matrix instance for
    the CMSIS DSP matrix initialization function
42 arm_matrix_instance_f32 ADC1_IN2_AC_mtx;
43
44
45 arm_matrix_instance_f32 ADC1_IN1_coeffs_mtx; // Creating a coefficients matrix instance for
    the CMSIS DSP matrix initialization function
46 arm_matrix_instance_f32 ADC1_IN2_coeffs_mtx;
47
48
49 arm_status StatusCoeffs_1; // Variable to monitor the status of the coefficients calculation
    operation
50 arm_status StatusCoeffs_2;
51
52
53 //FUNCTION DEFINITIONS
54 void ADC1_IN1_autocorr_calc(void)
55 {
56     /* Iterate through each lag */
57     for(uint32_t n = 0; n <= AR_ORDER; n++) // AR_ORDER is 10, so n runs from 0 to 10 (
        inclusive)
58     {
59         float32_t result_1 = 0.0f; // Initializing the result of the dot product in
            autocorrelation equation to zero before calculation
60
61         /* Number of valid points for the dot product at this lag for the data set*/
62         uint32_t Blocksize_1 = ADC_DMA_SIXTEENTHBUFFERSIZE - AR_ORDER; // The last AR_ORDER
            (10) values can not be computed for since the buffer does not hold all their past
            values
63
64         arm_dot_prod_f32(&(AR_ADC1_IN1_MA_ADC1_IN1_OutBfr[n]), AR_ADC1_IN1_MA_ADC1_IN1_OutBfr,
            Blocksize_1, &result_1); // A CMSIS DSP function for computing dot products. Its
            array input arguments are a moving average output buffer (earlier defined) which
            we intend to calculate autoregression coefficients for.
65
66         AutoCorr_1[n] = result_1 / ADC_DMA_SIXTEENTHBUFFERSIZE; // By the end of the loop. We
            have an array of autocorrelation (R) values
67     }
68 }
69
70 float32_t* ADC1_IN1_autoreg_coeffs(void)
71 {
72     float32_t AC_Matrix_1 [AR_ORDER]; // Declare an array to hold the autocorrelation matrix
        R1 - R10 as with the RHS of the Yule-Walker equation
73
74     for(uint32_t n=0; n < AR_ORDER; n++)
75     {
76         AC_Matrix_1[n] = AutoCorr_1[n + 1]; // Fills the array with the supposed values. R1 to
            R10 corresponds to indices 1 to 10
77     }
78
79     arm_mat_init_f32(&ADC1_IN1_AC_mtx, AR_ORDER, 1, AC_Matrix_1); // Initializes the
        autocorrelations matrix as shown on the LHS of the Yule-Walker equation
80
81
82     float32_t Yule_Walker_Matrix_1[AR_ORDER * AR_ORDER]; // Array that hold the Yule-Walker
        matrix data (100 elements) according to the Yule-Walker equations
83

```

```

84  /* Filling the Yule-Walker matrix with the appropriate values (100 elements) from the auto
    -correlations buffer values */
85  for (int r = 0; r < AR_ORDER; r++)
86  {
87      for (int32_t c = 0; c < AR_ORDER; c++)
88      {
89          /* Access the autocorrelation buffer using the absolute difference of indices */
90          Yule_Walker_Matrix_1[r * 10 + c] = AutoCorr_1[abs(r - c)]; // We are placing the
            elements in a one-dimensional array Yule-Walker matrix as if it represents a
            10x10 matrix. This is done by calculating the correct index for each matrix
            element using (i * 10 + j)
91      }
92  }
93
94  arm_mat_init_f32(&ADC1_IN1_YW_mtx, AR_ORDER, AR_ORDER, Yule_Walker_Matrix_1); //
    Initializes the Yule-Walker matrix
95
96
97  float32_t Yule_Walker_Matrix_Inv_1 [AR_ORDER * AR_ORDER]; // Array that holds the inverse
    Yule-Walker matrix data (100 elements)
98
99  arm_mat_init_f32(&ADC1_IN1_Inv_YW_mtx, AR_ORDER, AR_ORDER, Yule_Walker_Matrix_Inv_1); //
    Initializes the inverse Yule-Walker matrix
100
101  /* Calculate the inverse of the Yule-Walker matrix and return status of the operation */
102  arm_status StatusInv_1 = arm_mat_inverse_f32(&ADC1_IN1_YW_mtx, &ADC1_IN1_Inv_YW_mtx);
103
104  if(StatusInv_1 == ARM_MATH_SUCCESS) // Check if operation was successful
105  {
106      memset(AR_Coeffs_1, 0, AR_ORDER * sizeof(float32_t)); // Initializes the entire
            autoregression coefficients array to values zero
107
108      arm_mat_init_f32(&ADC1_IN1_coeffs_mtx, AR_ORDER, 1, AR_Coeffs_1); //Initializes the
            autoregression coefficients Matrix
109
110      StatusCoeffs_1 = arm_mat_mult_f32(&ADC1_IN1_Inv_YW_mtx, &ADC1_IN1_AC_mtx, &
            ADC1_IN1_coeffs_mtx); //A CMSIS DSP function. Computes the coefficients of the
            datasets and returns the status of the computations. The inverser matrix is a 10
            by 10 while the aurocorrelations matrix is 10 by 1. Meaning that the output will
            be a 10 by 1 matrix holding the coefficients values
111  }
112  else
113  {
114      // Do something to indicate that the process has failed
115  }
116
117  return AR_Coeffs_1; // The array now holds the coefficient values which can now be
    transmitted
118 }
119
120
121 /* The above process is repeated for the second channel*/
122 void ADC1_IN2_autocorr_calc(void)
123 {
124     /* Iterate through each lag */
125     for(uint32_t n=0; n<AR_ORDER; n++)
126     {
127         float32_t result_2 = 0.0f; // Initializing it to zero before calculation
128
129         /* Number of valid points for the dot product at this lag */
130         uint32_t Blocksize_2 = ADC_DMA_SIXTEENTHBUFFERSIZE - AR_ORDER;
131
132         arm_dot_prod_f32(&(AR_ADC1_IN2.MA_ADC1_IN2_OutBfr[n]), AR_ADC1_IN2.MA_ADC1_IN2_OutBfr,
            Blocksize_2, &result_2);
133
134         AutoCorr_2[n] = result_2 / ADC_DMA_SIXTEENTHBUFFERSIZE;
135     }
136 }
137
138 float32_t* ADC1_IN2_autoreg_coeffs(void)
139 {
140     float32_t AC_Matrix_2 [AR_ORDER];
141
142     for(uint32_t n=0; n < AR_ORDER; n++)
143     {
144         AC_Matrix_2[n] = AutoCorr_2[n + 1]; // R1 to R10 corresponds to indices 1 to 10

```

```

145     }
146
147     arm_mat_init_f32(&ADC1_IN2_AC_mtx, AR_ORDER, 1, AC_Matrix_2); //Initializes the
        autocorrelations matrix
148
149
150     float32_t Yule_Walker_Matrix_2[AR_ORDER * AR_ORDER]; // Array that hold the matrix data
        (100 elements) according to the Yule-Walker equations
151
152     /* Filling the Yule-Walker matrix with the appropriate values (100 elements) from the auto
        -correlations buffer values */
153     for (int r = 0; r < AR_ORDER; r++)
154     {
155         for (int32_t c = 0; c < AR_ORDER; c++)
156         {
157             /* Access the autocorrelation buffer using the absolute difference of indices */
158             Yule_Walker_Matrix_2[r * 10 + c] = AutoCorr_2[abs(r - c)]; // We are placing the
                elements in a one-dimensional array Yule-Walker matrix as if it represents a
                10x10 matrix. This is done by calculating the correct index for each matrix
                element using (i * 10 + j)
159         }
160     }
161
162     arm_mat_init_f32(&ADC1_IN2_YW_mtx, AR_ORDER, AR_ORDER, Yule_Walker_Matrix_2); //
        Initializes the Yule-Walker matrix
163
164
165     float32_t Yule_Walker_Matrix_Inv_2[AR_ORDER * AR_ORDER]; // Array that hold the inverse
        matrix data (100 elements)
166
167     arm_mat_init_f32(&ADC1_IN2_Inv_YW_mtx, AR_ORDER, AR_ORDER, Yule_Walker_Matrix_Inv_2); //
        Initializes the inverse Yule-Walker matrix
168
169     /* Calculate the inverse of the Yule-Walker matrix and return status of the operation */
170     arm_status StatusInv_2 = arm_mat_inverse_f32(&ADC1_IN2_YW_mtx, &ADC1_IN2_Inv_YW_mtx);
171
172     if(StatusInv_2 == ARM_MATH_SUCCESS) // Check if operation was successful
173     {
174         memset(AR_Coeffs_2, 0, AR_ORDER * sizeof(float32_t)); // Initializes the entire
            autoregression coefficients array to values zero
175
176         arm_mat_init_f32(&ADC1_IN2_coeffs_mtx, AR_ORDER, 1, AR_Coeffs_2); //Initializes the
            coefficients Matrix
177
178         StatusCoeffs_2 = arm_mat_mult_f32(&ADC1_IN2_Inv_YW_mtx, &ADC1_IN2_AC_mtx, &
            ADC1_IN2_coeffs_mtx);
179     }
180     else
181     {
182         // Do something to indicate that the process has failed
183     }
184
185     return AR_Coeffs_2;
186 }

```

I go ahead to show how I called the functions in the main.c file in Listing 3

Listing 3: Header file for the sorting algorithm

```

1  /* USER CODE BEGIN Includes */
2  #include <_ADCn_INx_AR.h>
3  /* USER CODE END Includes */
4  /* USER CODE BEGIN PV*/
5  /*.
6  .
7  . (Other private variables)
8  .*/
9  float32_t* AR_1;
10 float32_t* AR_2;
11 /* USER CODE END PV */
12
13 /* USER CODE BEGIN WHILE */
14 while (1)
15 {
16     if(TKE0_1 == 1) // Checks if there is muscle activation for channel 1. (Assumes the
17         reader has a function to do so. Here, I utilize the Teager Kaiser energy operator.
18         Threshold it and return a value )
19     {
20         /*Computes the autocorrelation values and the autoregression coefficients from the
21         moving average buffer and returns the latter*/
22         ADC1_IN1_autocorr_calc();
23         AR_1 = ADC1_IN1_autoreg_coeffs();
24     }
25
26     /*For channel 2*/
27     if(TKE0_2 == 1)
28     {
29         ADC1_IN2_autocorr_calc();
30         AR_2 = ADC1_IN2_autoreg_coeffs();
31     }
32 }
33 /* USER CODE END WHILE */

```

## Conclusion

The autoregression model offers rich insights when applied to time series data and can be applied to signals to accrue information that would be indeterminable by mere observation. Here I explained the mathematical model behind autoregression and went ahead to implement it on an arm cortex M4 MCU such that we produce autoregression coefficients that we can feed to an ML algorithm such as a support vector machine to evaluate various characteristics of the signal at the window we observe it. Autoregression coefficients are rich in information hence this concept can be extended to various other signals to derive various insights and information. Future editions of this article can be tracked through my public GitHub repository [here](#).