Escuela Universitaria
Politécnica - La Almunia
Centro adscrito
Universidad Zaragoza

ENERO/2023

REGULACIÓN Y CONTROL AUTOMÁTICO

FILTROS

PROFESOR **Dr. Diego Antolín Cañada**
AUTOR NAOUFAL EL RHAZZALI

UNIVERSIDAD DE ZARAGOZA

CONTENIDO

Objeto 7
Método operativo 8
Árbol del proyecto 8
Leyenda de archivos 8
Desarrollo 9
Función de transferencia Continua - Laplace 9
Función de Transferencia Discreta - Z 9
Implementación del filtro en C/C++ 9
Implementación regulador C/C++ 10
Generación del código fuente C/C++ 11
Simulación - Simulink 12
Script de Configuración 12
Filtro – Continuos 13
Filtro – Discreto 15
Filtro – Código C/C++ 15
Configuración del bloque “C Function” 16
Resultados 18
Filtro Paso Bajo (Low Pass Filter) 18
Fts – Filtro 18
FTz – Filtro 19
Implementación del filtro en C/C++ 20
Generación del código fuente C/C++ 20
Simulación - Simulink 21
Filtro/Continuo 22
Filtro/Discreto 23
Filtro – Código C/C++ 23
Filtro Paso Alto (High Pass Filter) 24
Fts – Filtro 24
FTz – Filtro 25
Implementación del filtro en C/C++ 26
Generación del código fuente C/C++ 26
Simulación - Simulink 27
Filtro – Continuos 28
Filtro – Discreto 29
Filtro – Código C/C++ 30
Filtro Paso Banda (Band Pass Filter) 30
Fts - Filtro 30
FTz - Filtro 31
Implementación del filtro en C/C++ 32

Generación del código fuente C/C++ 32
Simulación - Simulink 33
Filtro – Continuos 34
Anexos 36
Código de Regulador genérico 36

FIGURAS

Figura 1 - Ejecutando el comando Simulink en la línea de comandos 13
Figura 2 - Entorno Simulink de Matlab Online 13
Figura 3 - Espacio de trabajo con la librería, señalada con el puntero del ratón 13
Figura 4 - Librería desplegada 14
Figura 5 - Diagrama de bloques sistema Continuo 14
Figura 6 - Diagrama de bloques sistema Discreto 15
Figura 7 - Diagrama de bloques sistema C/C++ 16
Figura 8 - Cuadro de texto de configuración del bloque C Function 16
Figura 9 – Cuadro de texto de configuración del bloque C Function 17
Figura 10 - Configuración del Custom Code del bloque C Function 17
Figura 11 - Diagramas de bode del filtro Paso Bajo Continuo 18
Figura 12 - Respuesta al escalon unitario del filtro Paso Bajo Continuo 18
Figura 13 - Diagramas de bode del filtro Paso Bajo Discreto 19
Figura 14 - Respuesta al escalon unitario del filtro Paso Bajo Discreto 19
Figura 15 – Diagrama de bloques Filtro Paso Bajo en todos modos 22
Figura 16 – Respuesta del filtro paso bajo continuo en baja frecuencia 22
Figura 17 – Respuesta del filtro paso alto continuo en alta frecuencia 22
Figura 18 – Respuesta del filtro paso alto discreto en baja frecuencia 23
Figura 19 – Respuesta del filtro paso alto discreto en alta frecuencia 23
Figura 20 – Respuesta del filtro paso alto C/C++ en baja frecuencia 23
Figura 21 – Respuesta del filtro paso alto C/C++ en alta frecuencia 24
Figura 22 - Diagramas de bode del filtro Paso Alto Continuo 24
Figura 23 - Respuesta al escalon unitario del filtro Paso Alto Continuo 25
Figura 24 - Diagramas de bode del filtro Paso Alto Discreto 25
Figura 25 - Respuesta al escalon unitario del filtro Paso Alto Discreto 26
Figura 26 – Diagrama de bloques Filtro Paso Bajo en todos modos 28
Figura 27 – Respuesta del filtro paso alto continuo en baja frecuencia 28
Figura 28 – Respuesta del filtro paso alto continuo en alta frecuencia 29
Figura 29 – Respuesta del filtro paso alto discreto en baja frecuencia 29
Figura 30 – Respuesta del filtro paso alto discreto en alta frecuencia 29
Figura 31 – Respuesta del filtro paso alto C/C++ en baja frecuencia 30
Figura 32 – Respuesta del filtro paso alto C/C++ en alta frecuencia 30
Figura 33 - Diagramas de bode del filtro Paso Banda Continuo 31
Figura 34 - Respuesta al escalon unitario del filtro Paso Banda Continuo 31
Figura 35 – Diagrama de bloques Filtro Paso Banda en todos modos 34
Figura 36 – Respuesta del filtro paso Banda continuo en baja frecuencia 34
Figura 37 – Respuesta del filtro paso Banda continuo en media frecuencia 35
Figura 38 – Respuesta del filtro paso Banda continuo en alta frecuencia 35

TABLAS

Tabla 1 – Parámetros del filtro paso bajo continuo 18
Tabla 2 – Parámetros del filtro paso bajo discreto 19
Tabla 3 – Parámetros para el código C/C++ del filtro paso bajo 20
Tabla 4 – Parámetros del filtro paso alto continuo 24
Tabla 5 – Parámetros del filtro paso alto discreto 25

Tabla 6 – Parámetros para el código C/C++ del filtro paso bajo 26
Tabla 7 – Parámetros del filtro paso alto continuo 30
Tabla 8 – Parámetros del filtro paso banda discreto 31
Tabla 9 – Parámetros para el código C/C++ del filtro paso banda 32

ECUACIONES

[Ec. 1] – Fórmula de la frecuencia angular 9
[Ec. 2] – Fórmula de discretización de una Fts con bloqueador de orden 0 9
[Ec. 3] – Función de Transferencia en exponentes negativos 10
[Ec. 4] – Ecuación en diferencias 10
[Ec. 5] – Ecuación en diferencias en caso de a_0=1 10
[Ec. 6] – Definición de transformada Z a k 10
[Ec. 7] – Ecuación en diferencias transformada al tiempo discreto, k 10

OBJETO

El objetivo de esta práctica de regulación es comprobar que se ha realizado el cálculo de forma correcta en Matlab Simulink. Para ello se partirá de las ecuaciones del filtro calculadas en las prácticas de programación. Se hará el cálculo discreto del filtro con los comandos de Matlab, comparándolo con el resultado de cálculo manual.

Después se obtendrá la ecuación en diferencias del filtro y se escribirá el código correspondiente al filtro digital. Finalmente, se simulará el comportamiento de todos estos sistemas mediante Matlab Simulink y se discutirán los resultados obtenidos.

MÉTODO OPERATIVO

Para el desarrollo de estas prácticas, se empleará el software de Matlab para todos los cálculos, así mismo se empleará Simulink para todas las simulaciones.

Árbol del proyecto

Filtros

- **LowPass**
  - LP_config.m
  - LowPass_Sim.slx
  - LowPass_Sim.slxc
  - LowPass .c
  - LowPass .h
- **HighPass**
  - HP_config.m
  - HighPass_Sim.slx
  - HighPass_Sim.slxc
  - HighPass .c
  - HighPass .h
- **BandPass**
  - BP_config.m
  - BandPass_Sim.slx
  - BandPass_Sim.slxc
  - BandPass.c
  - BandPass.h

Leyenda de archivos

- Los archivos XX_config.m son scripts de configuración del filtro.
- Los archivos XX_Sim.slx son archivos de simulación de Simulink.
- Los archivos XX_Sim.slxc son archivos internos de simulación de Simulink.
- Los archivos XX.c son archivos de implementación de C/C++.
- Los archivos XX.c son archivos de cabecera de C/C++.

Desarrollo

Función de transferencia Continua - Laplace

Se pretende hallar la función de transferencia continua (**Fts**) del filtro. Para ello, se parte de la Fts general del filtro en cuestión, y se particulariza dicha función con la frecuencia de corte angular particular (**wc**).

Lo anterior, se consigue mediante hallar la frecuencia de corte lineal (**f_cutoff**), que es el resultado de una fórmula que involucra la NIA del estudiante.

$$F_c = f(A, B, C, D, E, F)$$

A partir de la **f_cutoff**, se halla **wc**, mediante **[Ec. 1]**.

[Ec. 1] – Fórmula de la frecuencia angular

$$\omega_c = 2\pi F_c$$

Conocida **wc** y la expresión del filtro en cuestión, proporcionada por los enunciados u obtenida a partir de los diagramas de bode, se obtiene la **Fts** particular del cada filtro en el dominio de Laplace.

Una vez se tenga dicha **Fts**, se obtendrán para la misma los diagramas de bode y la respuesta al escalón unitario. Así, se evaluarán el transitorio de la respuesta y los errores en estado estacionario.

Función de Transferencia Discreta - Z

Se pretende discretizar la **Fts**, obtenido la función de transferencia discretizada del filtro, en el dominio-Z. Para ello, se debe encontrar la transformada-Z del producto de **Fts** objetivo por un bloqueador (por simplicidad se escoge un bloqueador de orden 0, $\mathbf{B_0}$). Para ello, se resuelve **[Ec. 2].**

[Ec. 2] – Fórmula de discretización de una Fts con bloqueador de orden 0

$$G_z = \mathbf{Z}[G(s)B_0] = (1 - z^{-1})\sum[res_{(\frac{G(p)}{p})}[\frac{G(p)}{p}\frac{1}{1 - e^{pT}z^{-1}}]]$$

Resolver dicha transformación y el cómo se obtuvo dicho resultado, y toda la demostración teórico-matemáticamente detrás de ese proceso, se encuentra en el siguiente documento: Introducción a la discretización de reguladores continuos (Documento realizado por Naoufal El Rhazzali, el cual sigue estando en revisión y desarrollo). Tras resolver **[Ec. 2],** se obtiene **FTz** en tiempo discreto.

Implementación del filtro en C/C++

Se pretende implementar en **C/C++**, el filtro. Para ello, se debe obtenerse la ecuación en diferencias del filtro. Dicha ecuación se puede obtenerse de la **FTz** del apartado anterior.

Al hilo de lo anterior, se procede, inspeccionando la **FTz**, en busca del mayor grado. Dicho grado debería estar en el denominador, sino el regulador (o el filtro) no será causal y su implementación no se podría realizar.

Una vez Localizado dicho grado, y denotado como "**n**", se multiplica el numerador y denominador por $\mathbf{z^{-n}}$.

Hecho lo anterior, se igual la **FTz** obtenida, al cociente entre Salida, **U(z)**, y Entrada, **E(z)**, del filtro, obteniendo **[Ec. 3]**.

[Ec. 3] – Función de Transferencia en exponentes negativos

$$\frac{U(z)}{E(z)} = \frac{b_0 z^0 + b_1 z^{-1} + b_2 z^{-2} + b_3 z^{-3}, \ldots, + b_m z^{-m}}{a_0 z^0 + a_1 z^{-1} + a_2 z^{-2} + a_3 z^{-3}, \ldots, + a_n z^{-n}} = \frac{\sum_{i=0}^{m} b_i z^{-i}}{\sum_{j=0}^{n} a_i z^{-i}}$$

A partir de **[Ec. 3]** se obtienen **[Ec. 4]** y **[Ec. 5]**, ecuaciones en diferencias en Z

[Ec. 4] – Ecuación en diferencias

$$U(z) \sum_{j=0}^{n} a_i z^{-i} = E(z) \sum_{i=0}^{m} b_i z^{-i}$$

[Ec. 5] – Ecuación en diferencias en caso de a_0=1

$$U(z) = E(z) \sum_{i=0}^{m} b_i z^{-i} - U(z) \sum_{j=1}^{n} a_i z^{-i}$$

Finalmente, partiendo de la definición reflejada **[Ec. 6]**, se invierte la ecuación desde Z al tiempo discreto, en k. Por lo tanto, se obtiene **[Ec. 7]**.

[Ec. 6] – Definición de transformada Z a k

$$U_{k-n} = cte \cdot U(z) z^{-n},$$

[Ec. 7] – Ecuación en diferencias transformada al tiempo discreto, k

$$U_k = Z^{-1}[U(z)] = \sum_{i=0}^{m} b_i E_{k-i} - \sum_{j=1}^{n} a_i U_{k-i}$$

Teniendo la ecuación en diferencias, se procede con la implementación del código.

Implementación regulador C/C++

Se pretender realizar una función que ejecuta una ecuación en diferencias de cualquier regulador. Para ello, se cree en la implementación de una función reutilizable, siguiendo las buenas prácticas de programación.

La función recibe el error actual, $\mathbf{E_K}$, que tiene el sistema. Además, punteros a los coeficientes $\mathbf{a_i}$ y $\mathbf{b_i}$. Finalmente, la función retornaría la acción de regulación, $\mathbf{U_K}$.

Así, la firma o prototipo de la función sería:

```
double controller(double E, double* b, double* a)
```

La función entonces se llama **controller** y recibe el error actual por copia de valor, y recibe los coeficientes por referencia. Esto último queda justificado como sigue: hacer copia de grandes cantidades de datos no es eficiente.

El siguiente paso es la implementación de la función. Lo primero que se hace es declarar los arrays que contendrán los valores de acción y error. Ello contempla declarar arrays con las longitudes necesarias (n y m). Esto último se refiere a que cada regulador necesita recordar tantos valores pasados de acción (U_k, U_{k-1}, …, U_{k-n}) y error (E_k, E_{k-1}, …, E_{k-m}). El regulador emplea dichos valores para obtener el nuevo valor de la acción.

Lo siguiente es inicializar dichos arrays, para asegurarse de que al inicio todos los valores estén a cero. Esto último se hace mediante **bucles-for**. Así mismo, se emplea

la variable **init_arrays**. Dicha variable es cero cuando se convoca la función por primera vez, se inicializan los arrays, y a partir de allí se pone a 1. Ello impide que a mitad de ejecución se vuelvan a reinicializarse los arrays.

La inicialización se podría haberse hecho a mano. No obstante, la idea detrás de esta función es que sea genérica, contemplando la posibilidad de que existan reguladores que se basen en 10 muestras anteriores de error y acción. Si se da el caso anterior, ponerse a asignar valores 0 a mano sería tedioso cada vez que se necesite dicho código.

El siguiente paso consiste en asignar a la primera posición del array de errores, el valor del error actual que se le pasa a la función cuando se le convoca.

Seguidamente, se declaran las variables **sum_Uk** y **sum_Ek**. La idea detrás de dichas variables se basa en que el regulador consiste en una sumatoria de coeficientes, $\mathbf{b_i}$, que multiplican a valores de error, y otra sumatoria de otros coeficientes, $\mathbf{a_i}$, que multiplican a valores de acción. Por ello, se calculan dichas sumas por separado, mediante bucles-for. Finalmente, se suman las dos variables dando el valor de la acción que se debería transmitirse hacia la planta. Por ello, es el valor del retorno.

Pero antes de retornar, se ejecutan dos bucles, que básicamente hacen un desplazamiento hacia la derecha de los arrays de acción y error. Esto último, se podría haberse implementado mejor mediante la conocida estructura de datos: **buffer circular** (o ring buffer). No obstante, debido al tiempo que se tiene asignado para este proyecto, prima más la facilidad de implementación que la eficiencia.

Lo último que queda por saber es cómo se van a definirse las longitudes de los arrays **N** y **M**. Ello se hace en un archivo de cabecera (o header). En dicho archivo se definen dichas longitudes, además del prototipo o firma de la función.

Finalmente, cuando se emplee dicho controlador para implementar un filtro, lo que se haría es definir una función-filtro, y dentro de la misma definir los coeficientes. Además, los delays o control del periodo de muestreo, y el cálculo de la acción de control se delega a la función **controller**.

Revisar en Anexos: **Código de Regulador genérico**

Lo siguiente sería implementar una función que ejecuta el filtro propiamente dicho. Para ello, se implementa un filtro genérico. Éste recibe $\mathbf{E_K}$ (valor actual del error), **Ts** (periodo de muestreo) que sería el retardo entre una ejecución y otra del filtro, el **time** que sería el reloj del sistema, y finalmente un puntero a **aux**, la cual es una variable auxiliar que permitiría la depuración del código mediante la visualización de parámetros de ejecución.

No existe una versión genérica del código de un filtro. Dependerá de qué filtro se trata.

Generación del código fuente C/C++

Para generar el código fuente del filtro que se necesitará para la simulación, hay que organizar las funciones obtenidas anteriormente en dos ficheros: uno de cabeza o ".h" y otro de implementación o ".c". Así, se realizan los siguientes ficheros.

- "headerFile.h"
- "ImplementationFile.c"

headerFile.h es un archivo de cabecera (Header), y en ello se declaran los prototipos (o firmas) de funciones. Así mismo, se definen macros o se declaran constantes que se utilizarán en el fichero de implementación. Mientras que **ImplementationFile.c** es un

archivo de implementación. En ello se implementan las funciones declaradas en el header. Dichos archivos se deben alojar en la carpeta del filtro en cuestión, como se muestra mostrada en el apartado de **Árbol del proyecto**.

Simulación - Simulink

Se pretende simular el comportamiento del filtro para todas las formas en la que se ha implementado: **Continuo**, **Discreto** y en **C/C++**. Para ello, se procede con cada forma.

Antes de empezar con la simulación es fundamental, realizar un script de configuración, donde se tengan definidas todas las variables y constantes que se necesitarán para la simulación. Es cierto, que no es necesario. No obstante, supone una buena práctica y agiliza la ejecución de la simulación, la depuración y realización de pruebas de evaluación.

Script de Configuración

Al hilo de lo anterior se crea un script con las iniciales en mayúscula del nombre del filtro inglés, seguido de un guion bajo y luego config. Por ejemplo, en caso del filtro paso bajo, (en inglés es Low Pass Filter) el nombre del script sería: **LP_config**.

La primera parte de estos scripts, suele empezar con definir los parámetros de la excitación: amplitud, offset, frecuencia lineal y frecuencia angular.

A continuación, se agrega la parte correspondiente a la definición de los parámetros continuos del filtro. Así, se declara primero la frecuencia máxima que permite un correcto funcionamiento para el **Atmega328p** (**Fc_Limit_Arduino**). A partir de allí se obtiene la **f_cutoff**, calculada a partir del NIA. O bien, se definen dos frecuencias de cortes: una superior y otra inferior. En el primer caso, se continua con un pequeño **if-end**, que comprueba que la **f_cutoff** no es mayor que **Fc_Limit_Arduino**. En el caso contrario, se divide **f_cutoff** entre 10. Conocidas las frecuencias lineales necesarias, se calculan las **wc's necesarias**. Con éstas últimas, ya se pueden declarar el numerador (**numGs**) y el denominador (**denGs**). Con ello, se obtiene la **Fts** (**Gs**). Finalmente, se obtienen los diagramas de bode y la respuesta al escalón unitario del filtro.

La siguiente parte del script corresponde con los parámetros discretos del filtro. Para conseguir lo anterior, se procede declarando la frecuencia de muestreo (**f_sample**). Dicha frecuencia como mínimo debe ser 2 veces mayor a la frecuencia de la menor componente sinusoidal de la señal muestreada. No obstante, las recomendaciones por parte de los expertos van en factores de 10 y mayores. Por esto último se opta por 10. Lo siguiente es obtener el periodo de muestreo (**Ts**) que es el inverso de **f_sample**. Con ello, se obtiene la **FTz**. Esto último, se consigue mediante el comando **c2d(Fts, Ts)**, de Matlab. Seguidamente, con la **FTz** en mano, se sacarían los numerador y denominador necesarios para la simulación en discreto. Para ello, sabiendo que **tf(…)** devuelve un objeto, basta con acceder a los atributos que interesan: **Numerator** y **Denominator**. No obstante, **Numerator** y **Denominator**, son del tipo dato **cell** de Matlab. Sin embargo, interesa el tipo de dato **mat**. Seguiendo la filosofía del lenguaje de Matlab, se puede sospechar que existiría alguna función del tipo: "xx2yy". Efectivamente: **cell2mat(…)**. Finalmente, se obtienen los diagramas de bode y la respuesta al escalón unitario.

Los scripts de configuración pueden incluir algún código más, por alguna particularidad de cada filtro en cuestión. Como por ejemplo un apartado **Simulación**. No obstante, es la forma más genérica de un script de configuración.

Filtro – Continuos

Para ello, se ejecuta en la línea de comandos de Matlab,

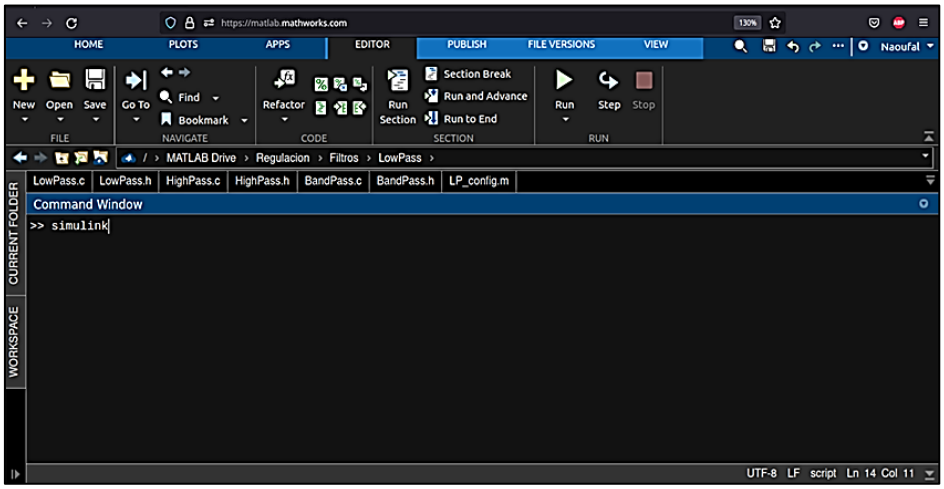

Figura 1 - Ejecutando el comando Simulink en la línea de comandos

Ese último comando hará que aparezca la siguiente ventana emergente,

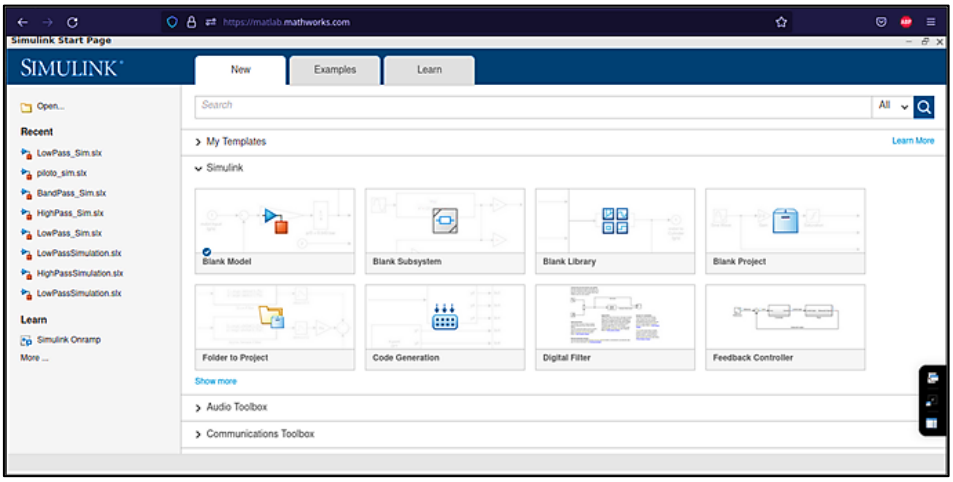

Figura 2 - Entorno Simulink de Matlab Online

El siguiente paso es crear un modelo en blanco. Para ello, se da clic sobre **Blank Model**. Ello hará arrojar un nuevo espacio de trabajo vacío, que se irá llenando con los componentes que se necesiten para ejecutar la simulación.

Para encontrar dichos componentes, se recurre a la librería, a la cual se accede dando clic sobre el icono sobre el cual está el puntero del mouse en la **Figura 3**

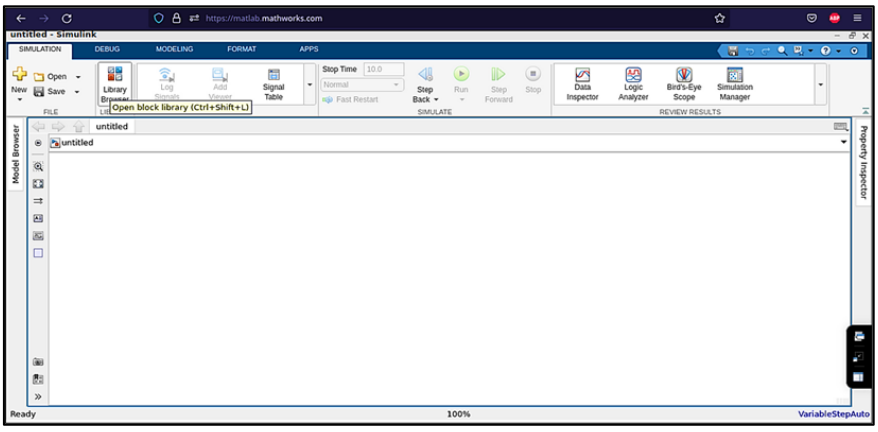

Figura 3 - Espacio de trabajo con la librería, señalada con el puntero del ratón

Tras pulsar aparece una barra de herramientas lateral, dónde sale un listado. Allí es dónde hay que bucear para encontrar los componentes interés.

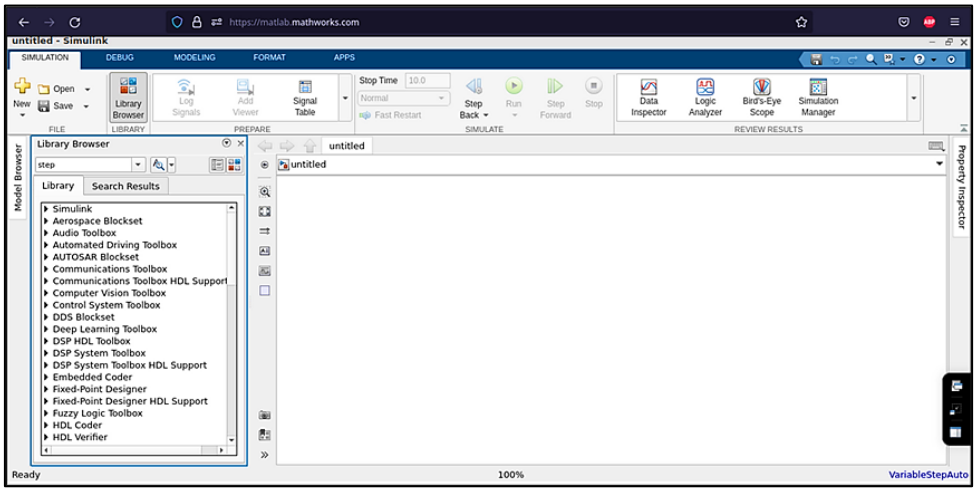

Figura 4 - Librería desplegada

El siguiente paso es dirigirse al árbol de búsqueda y llegar hasta “Sine Wave” (Onda sinusoidal),

Simulink
↳Sources
↳Sine Wave

Ahora se busca “Transfer Fcn” (función de Transferencia en tiempo continuo),

Simulink
↳Continuous
↳Transfer Fcn

Ahora se busca “Scope” (Oscilocopio),

Simulink
↳Commonly Used Blocks
↳Scope

Finalmente, se conectan todos los elementos del sistema, quedándose el diagrama de bloques como muestra **Figura 5**.

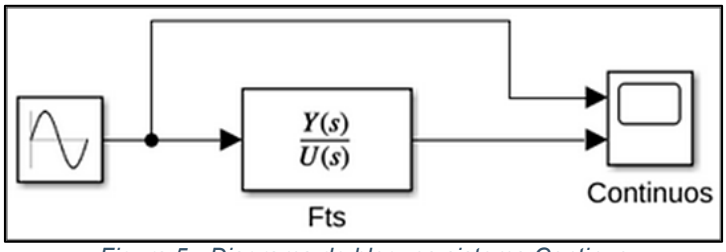

Figura 5 - Diagrama de bloques sistema Continuo

El siguiente paso es configurar los bloques. Ello, se hace dando doble clic sobre cada bloque. Así para “Sine Wave”, se rellenan los apartados de amplitud, offset y frecuencia angular con las variables generadas mediante el script de configuración. Para “Transfer Fcn”, lo mismo, se rellenan los apartados de numerador y denominador. Por último, la única configuración que se hace para “Scope” es el aumentar el número de entradas. Ello se consigue dando doble clic sobre el bloque, y pinchando en el icono de engranaje (Configuración de propiedades) y poniendo al **Number of input ports: 2**.

Filtro – Discreto

Lo único que hay que hacer es copiar el diagrama obtenido del Filtro-Continuo, salvo que el bloque “Transfer Fcn” se sustituye por “Discrete Transfer Fcn”. Para encontrarlo se bucea como sigue,

Simulink
↳**Discrete**
↳Discrete Transfer Fcn

Así se obtiene le siguiente Diagrama de bloques de la **Figura 6**

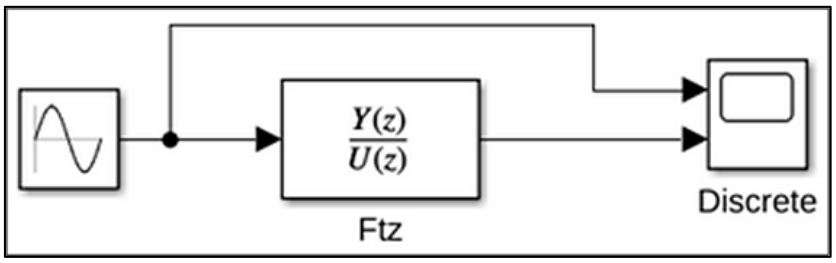

Figura 6 - Diagrama de bloques sistema Discreto

Filtro – Código C/C++

En este caso hay que buscar cuatro nuevos componentes. **Const**, **Digital Clock**, **Display** y finalmente, **C Function**. Nuevamente hay que dirigirse al árbol y empezar a buscar:

Ahora se busca Const (Constante),

Simulink
↳**Commonly Used Blocks**
↳**Const**

Ahora se busca **Digital Clock** (Reloj digital),

Simulink
↳**Continuous**
↳**Digital Clock**

Ahora se busca **Display** (Pantalla para ver valores),

Simulink
↳**Sinks**
↳**Display**

Finalmente, se busca **C Function** (Bloque de función en C),

Simulink
↳**User-Defined Functions**
↳**C Function**

Finalmente, conectamos todos los elementos del sistema, quedándose el diagrama de bloques como muestra **Figura 6**.

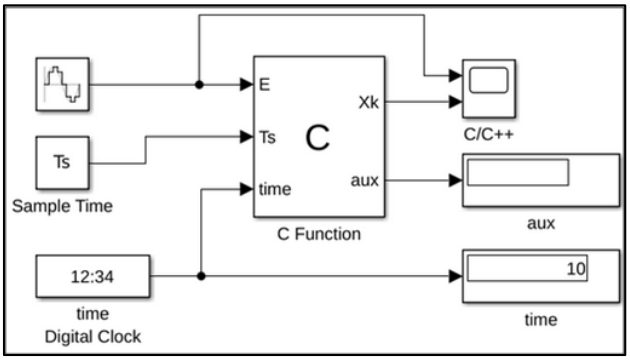

Figura 7 - Diagrama de bloques sistema C/C++

El siguiente paso es configurar los bloques. Así para **Const**, se rellana con el periodo de muestreo: **Ts**, definido en el script de configuración. Para **Digital Clock**, lo mismo, se rellena con el periodo (**Tsim**) que debería tener la onda cuadrada que genera el reloj. Dicho **Tsim**, ya se encuentra definido en el script de configuración. Los **Displays** no se configuran, generalmente. Finalmente, se de configurar **C-Function**, ello puede resultar un poco tedioso, por ello se le dedica a su configuración un apartado.

Configuración del bloque "C Function"

Tras dar dos **clics** sobre el bloque de configuración, aparece el cuadro de texto de la **Figura 8**. En la misma, se han **señalado** tres cuadros. Se empezaría por el cuadro rojo, el más de arriba. Dicha sección está destinada a la configuración de las entradas y salidas del bloque. Por ello, al dar "Add", **aparecerá** abajo una nueva fila, allí se podrían elegir el nombre de la variable, si es entrada (input) o salida (output), y finalmente el tamaño de **variable**, para este caso será siempre "double".

Hay que señalar que las variables añadidas en dicha sección deben corresponder en cuanto a nombre, exactamente con los nombres de las variables que estarán en la firma de la función de C/C++ a ejecutar.

El siguiente cuadro rojo, el mediano, está destinada a qué código va a ejecutar el bloque. Dado que se tendrá una función del filtro diseñada basta con escribir la variable de salida del bloque igual a la función diseñada con las variables de entrada.

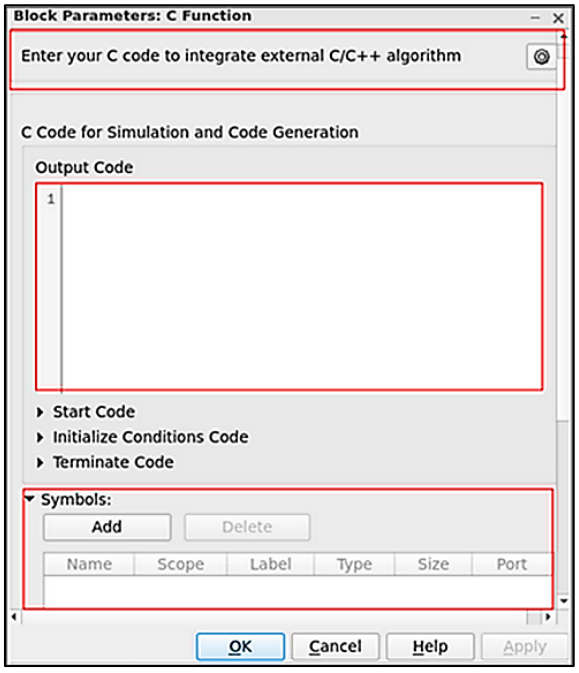

Figura 8 - Cuadro de texto de configuración del bloque C Function

Finalmente, en el cuadro rojo de más arriba, se podría ver que hay un icono de engranaje, si se pincha dicho icono, se llega a otro cuadro de texto, el cual se muestra a continuación el **Figura 9.**

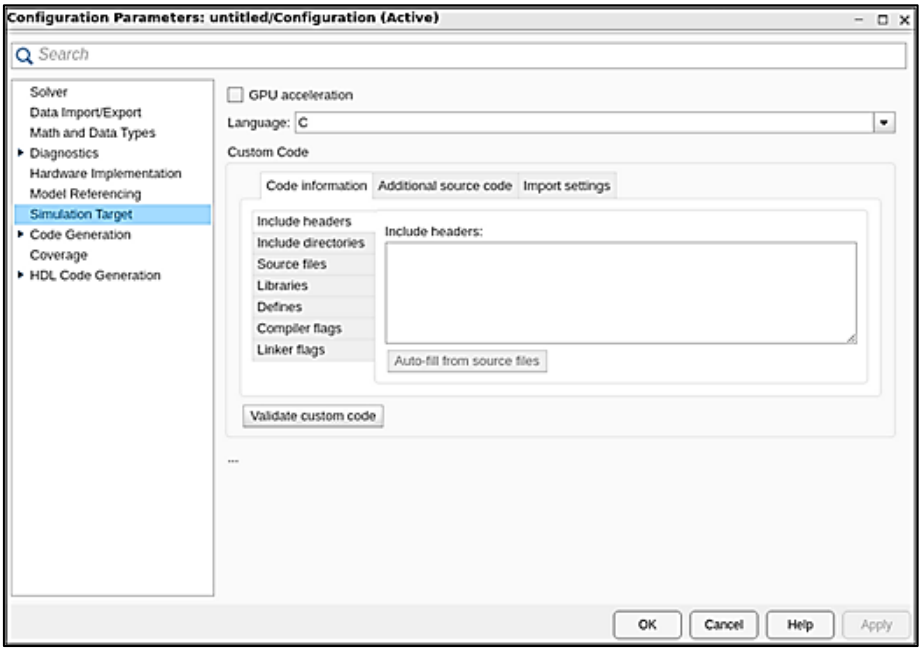

Figura 9 – Cuadro de texto de configuración del bloque C Function

En dicho cuadro, en el apartado de **Simulation Target**, se ha de indicar los archivos fuente de la función a ejecutar. Dichos archivos son de extensiones ".c" y ".h" y se deben encontrar alojados en la carpeta del proyecto. Simplemente, en dónde pone **Include headers**, se agrega lo que sigue: *"#include "nombreArchivo.h""*. En dónde pone **Sources files**, se agrega: "*nombreArchivo.c*" (sin comillas). En dónde pone **Include directories**, se agrega la ruta absoluta de dónde estén alojados los archivos.
Antes de cerrar el cuadro de texto, se debe dirigirse al apartado de Code Generation/Costum Code**. A**llí dejar con un tick el cuadro dónde pone Use the same code settings as Simultation Target. Ello se ilustra en la **Figura 10**.

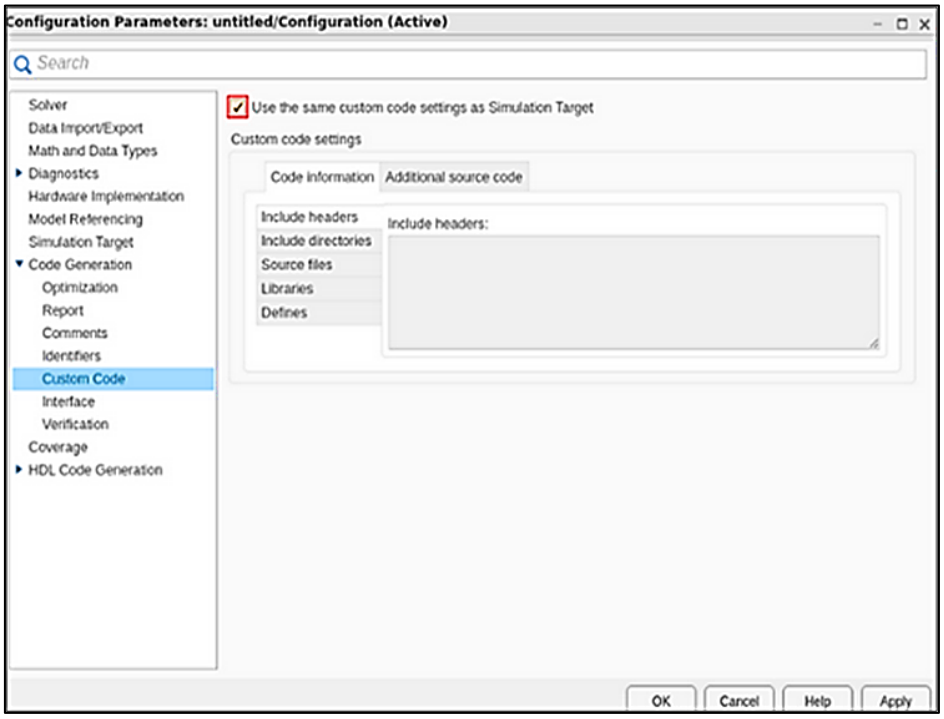

Figura 10 - Configuración del Custom Code del bloque C Function

RESULTADOS

Filtro Paso Bajo (Low Pass Filter)

Fts – Filtro

Parámetros	Resultados
NIA	787246
f_{cutoff}	$63\ Hz$
ω_c	$395.8407\ \frac{rad}{s}$
G_s	$\frac{395.8}{s+395.8}$

Tabla 1 – Parámetros del filtro paso bajo continuo

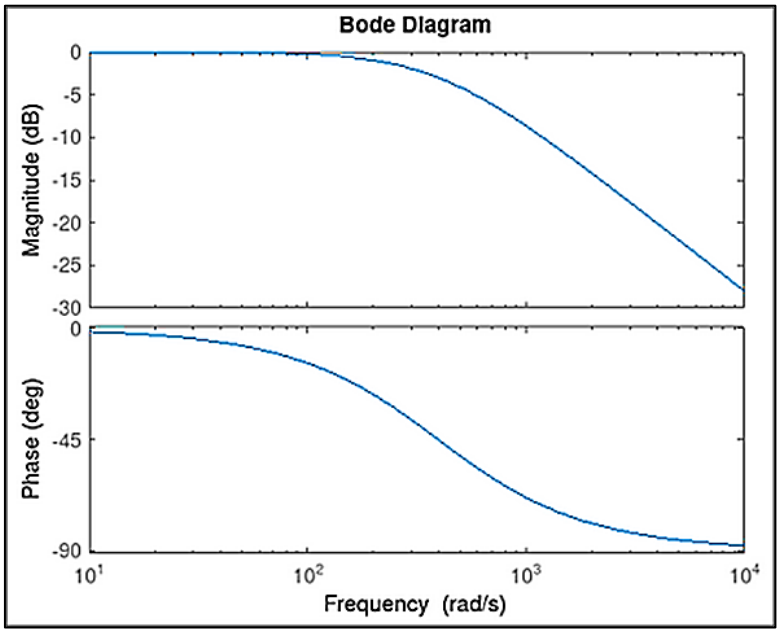

Figura 11 - Diagramas de bode del filtro Paso Bajo Continuo

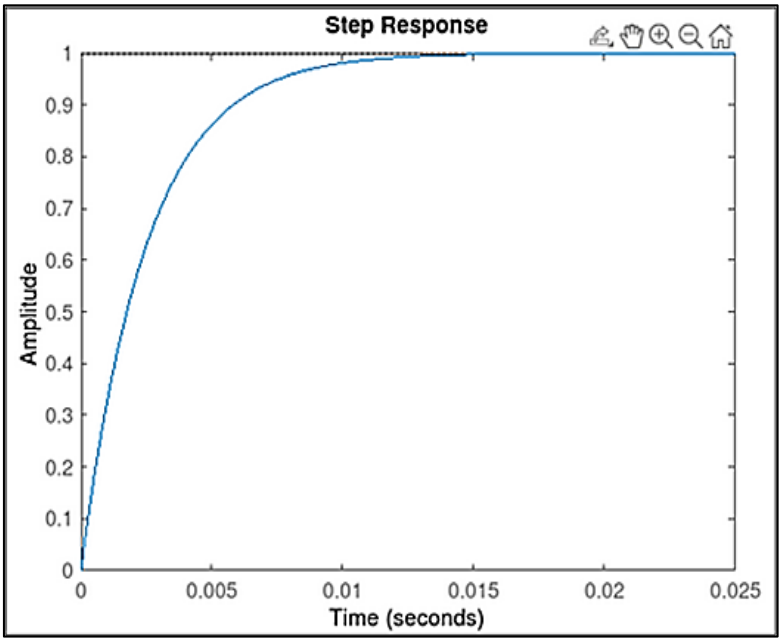

Figura 12 - Respuesta al escalon unitario del filtro Paso Bajo Continuo

FTz – Filtro

Parámetros	Resultados
NIA	787246
f_{cutoff}	$63\ Hz$
f_{sample}	$630\ Hz$
T_S	$0.0016\ s$
ω_c	$395.8407 \frac{rad}{s}$
G_z	$\frac{0.4665}{z - 0.5335}$

Tabla 2 – Parámetros del filtro paso bajo discreto

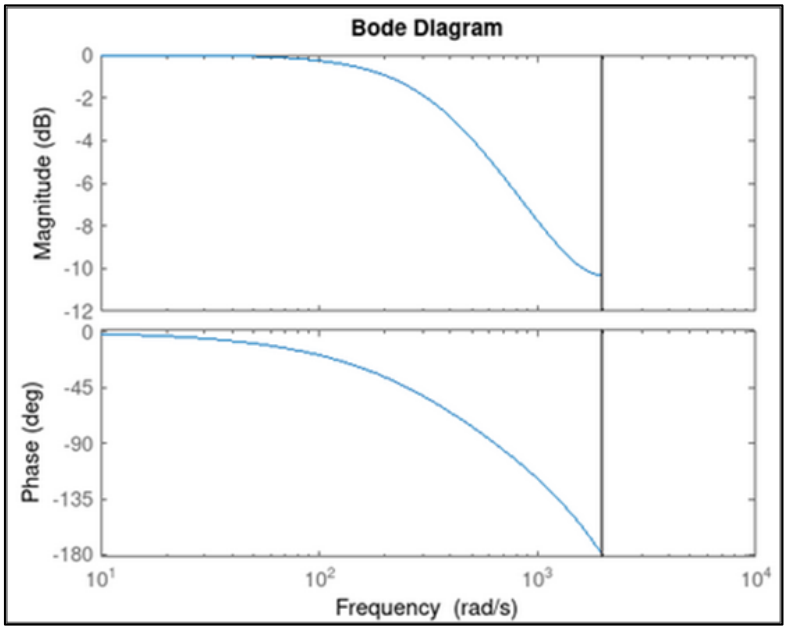

Figura 13 - Diagramas de bode del filtro Paso Bajo Discreto

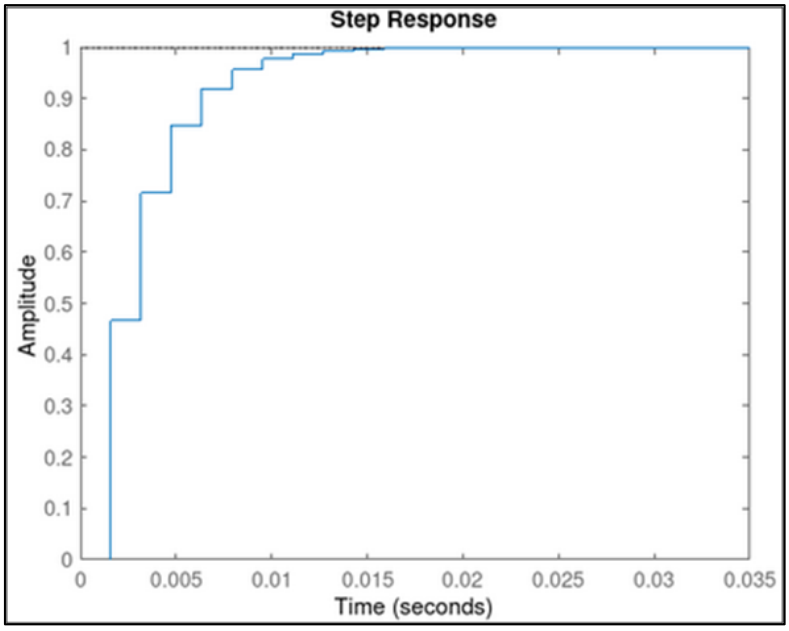

Figura 14 - Respuesta al escalon unitario del filtro Paso Bajo Discreto

Implementación del filtro en C/C++

$$G_s = \frac{U(z)}{E(z)} = \frac{0.4665}{z - 0.5335}\frac{z^{-1}}{z^{-1}} = \frac{0.4665z^{-1}}{1 - 0.5335z^{-1}}$$

$$U(z)(1 - 0.5335z^{-1}) = E(z)(0.4665z^{-1})$$
$$U(z) = 0.4665E(z)z^{-1} + 0.5335U(z)z^{-1}$$

$$U_k = 0.4665E_{k-1} + 0.5335U_{k-1}$$

Parámetros	Resultados
M	2
N	2
b[M]	{0, 0.4665}
a[N]	{0, 0.4665}

Tabla 3 – Parámetros para el código C/C++ del filtro paso bajo

Generación del código fuente C/C++

- LowPass.h
- LowPass.c

LowPass.h

```
#ifndef BANDPASS_H
#define BANDPASS_H

#define N   3
#define M   3

double controller(double Ek, double* b, double* a);
double BandPass(double Ek, double Ts, double time, double* aux);

#endif /*   LOWPASS_H   */
```

LowPass.c

```
#include "BandPass.h"

double BandPass(double Ek, double Ts, double time, double* aux){
    static int k=0;
    static double Xk=0;

    double b[M]={0.0, 0.4523, -0.4494};
    double a[N]={0.0, 1.0473, -0.5010};

    if(time < Ts){
        k=1;
    }
    else if(time >= k*Ts){
        Xk=controller(Ek, b, a);
        k++;
        *aux=k*Ts;
    }
    return Xk;
}

double controller(double E, double* b, double* a){
    static int init_arrays = 0;
    static float U[N];
```

```
    static float E_values[M];

    if(!init_arrays){
        for(int i=0;i<N; i++) U[i]=0;
        for(int i=0;i<M; i++) E_values[i]=0;
        init_arrays = 1;
    }

    E_values[0]=E;

    float sum_Ek, sum_Uk;
    sum_Ek=0; sum_Uk=0;

    for(int k=0; k<M; k++)  sum_Ek = sum_Ek + b[k]*E_values[k];
    for(int k=0; k<N; k++)  sum_Uk = sum_Uk + a[k]*U[k];

    U[0]=sum_Ek+sum_Uk;

    for(int k=(M-1); k>0; k--)  E_values[k]=E_values[k-1];
    for(int k=(N-1); k>0; k--)  U[k]=U[k-1];

    return U[0];
}
```

Simulación - Simulink

LP_config.m

```
%% INPUT SIGNAL
amplitude_input=1;      % Volt
offset_input=2.5;       % Volt
f_input=630;           % Hz
w_input=2*pi*f_input;   % rad/s

%% Continous LOW PASS FILTER
Fc_Limit_Arduino = 150; % Hz
A=7; B=8; C=7; D=2; E=4; F=6;
f_cutoff = (1*C+1*D)*1*A*10; % Cuttoff frecuency (Hz)

if f_cutoff > Fc_Limit_Arduino
    f_cutoff=f_cutoff/10;
end

wc=2*pi*f_cutoff; % Angular frecuency (rad/s)
numGs=[wc];
denGs=[1 wc];
Gs=tf(numGs, denGs); % Transfer function (Tf) in the Laplace domain
bode(Gs);
step(Gs);

%% DIGITAL LOW PASS FILTER
f_sample=10*f_cutoff;   % Sample frecuency (Hz)
Ts=1/f_sample;          % Sample rate (seconds = s)
Gz=c2d(Gs, Ts);
numGz=cell2mat(Gz.Numerator);
denGz=cell2mat(Gz.Denominator);

bode(Gz);
step(Gz);

%% Simulation
Tsim=0.01*Ts;
```

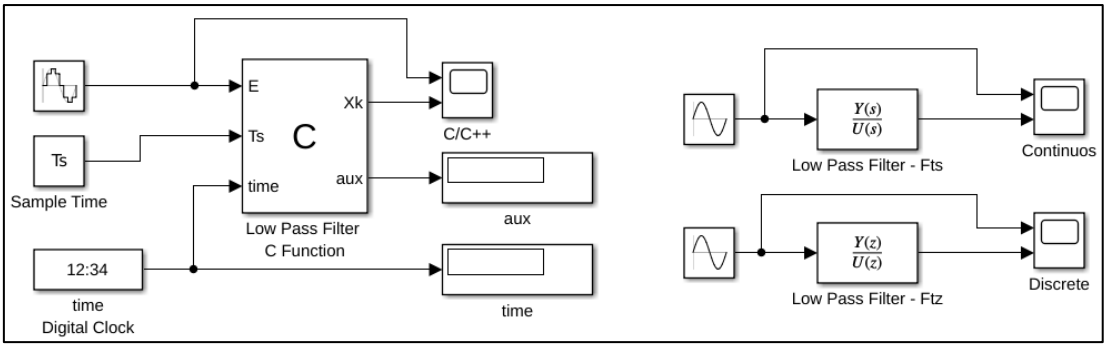

Figura 15 – Diagrama de bloques Filtro Paso Bajo en todos modos

Se ha ensayado al filtro paso bajo en todos sus modos (Continuo, Discreto y C/C++), con una entrada sinusoidal de 10Hz (baja frecuencia) y otra de 630Hz (alta frecuencia). A continuación, se presentan los resultados.

Filtro/Continuo

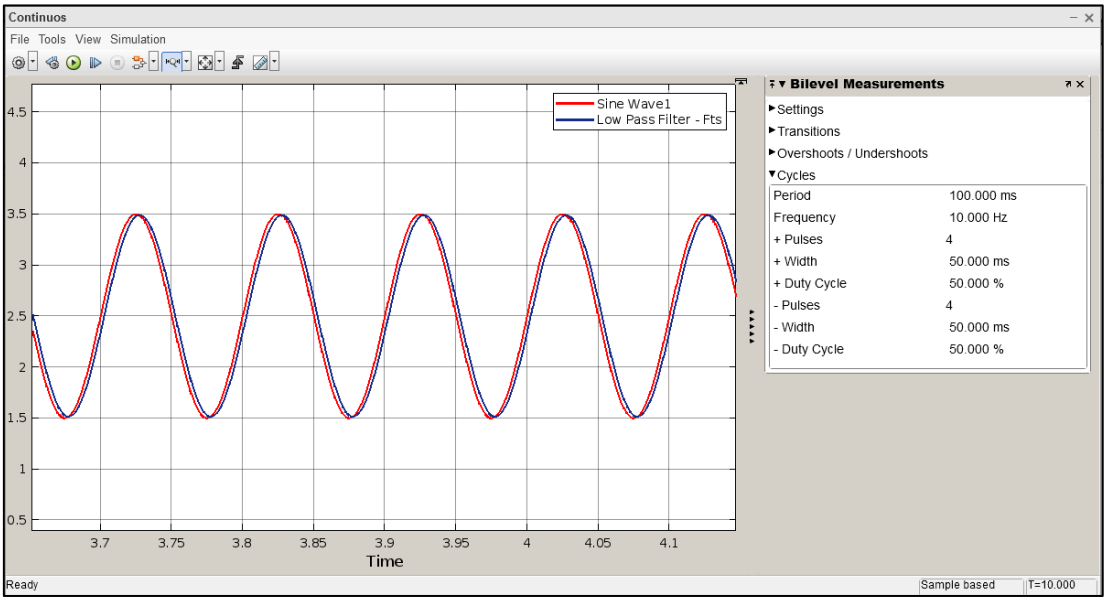

Figura 16 – Respuesta del filtro paso bajo continuo en baja frecuencia

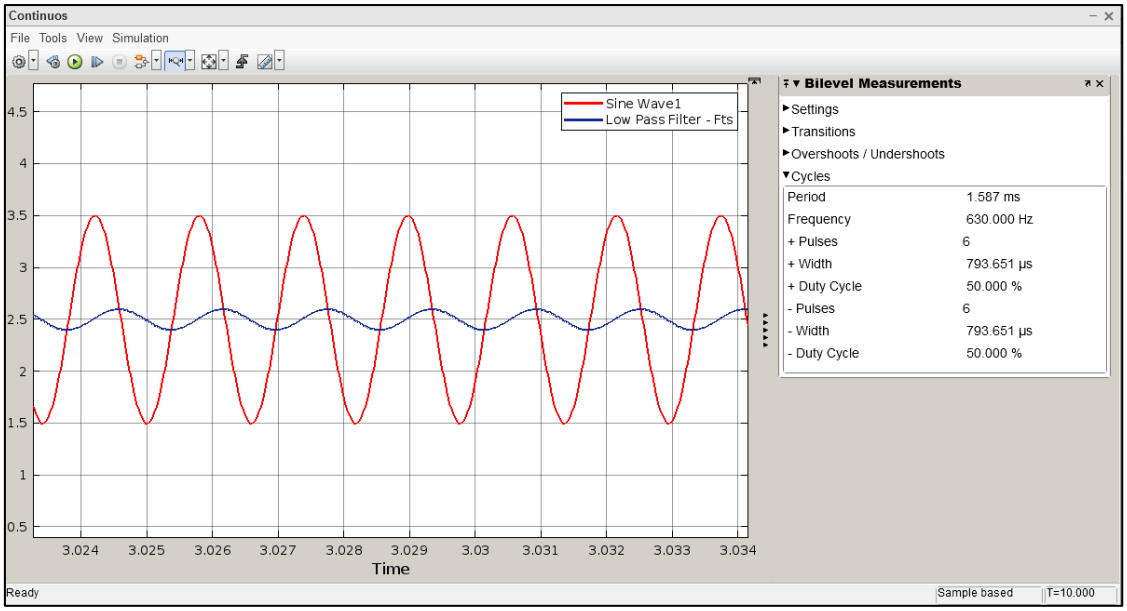

Figura 17 – Respuesta del filtro paso alto continuo en alta frecuencia

Filtro/Discreto

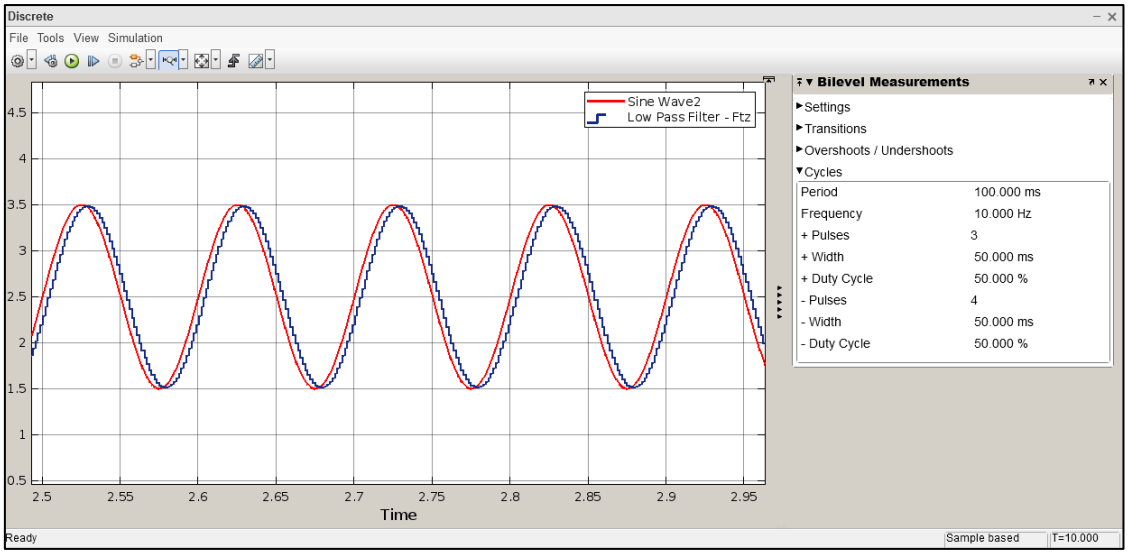

Figura 18 – Respuesta del filtro paso alto discreto en baja frecuencia

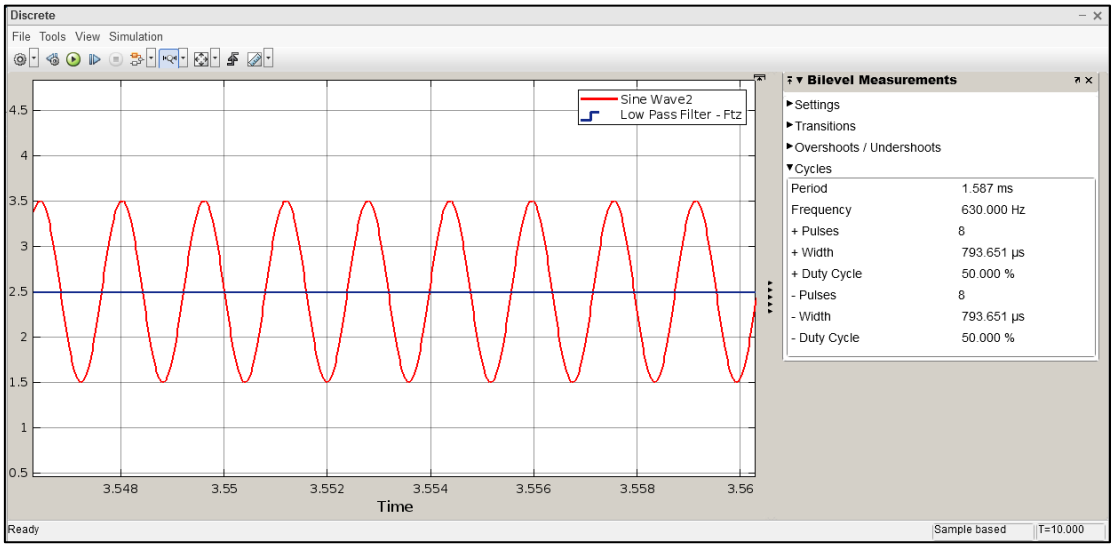

Figura 19 – Respuesta del filtro paso alto discreto en alta frecuencia

Filtro – Código C/C++

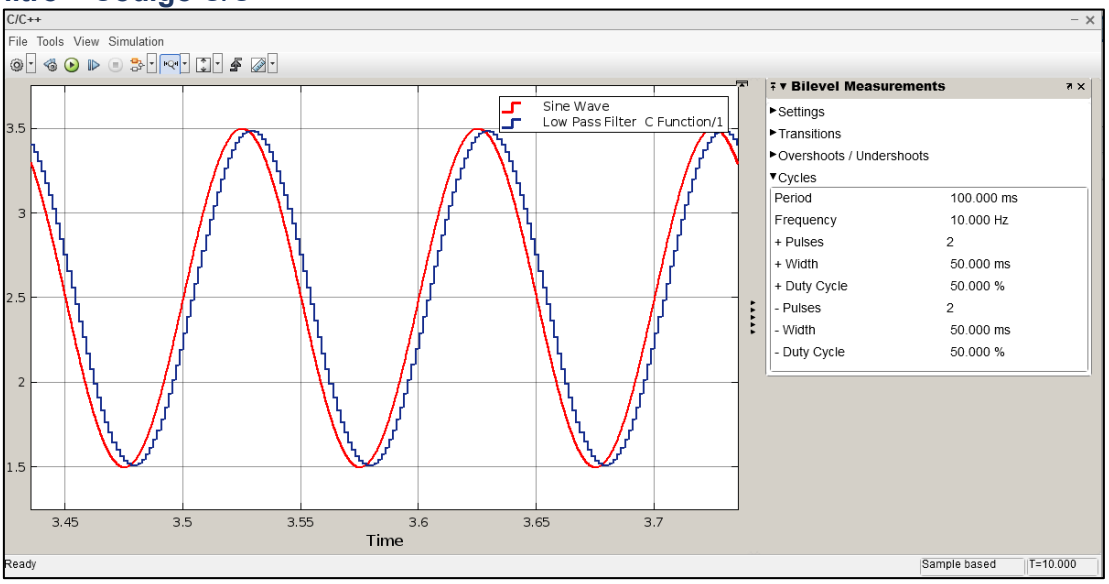

Figura 20 – Respuesta del filtro paso alto C/C++ en baja frecuencia

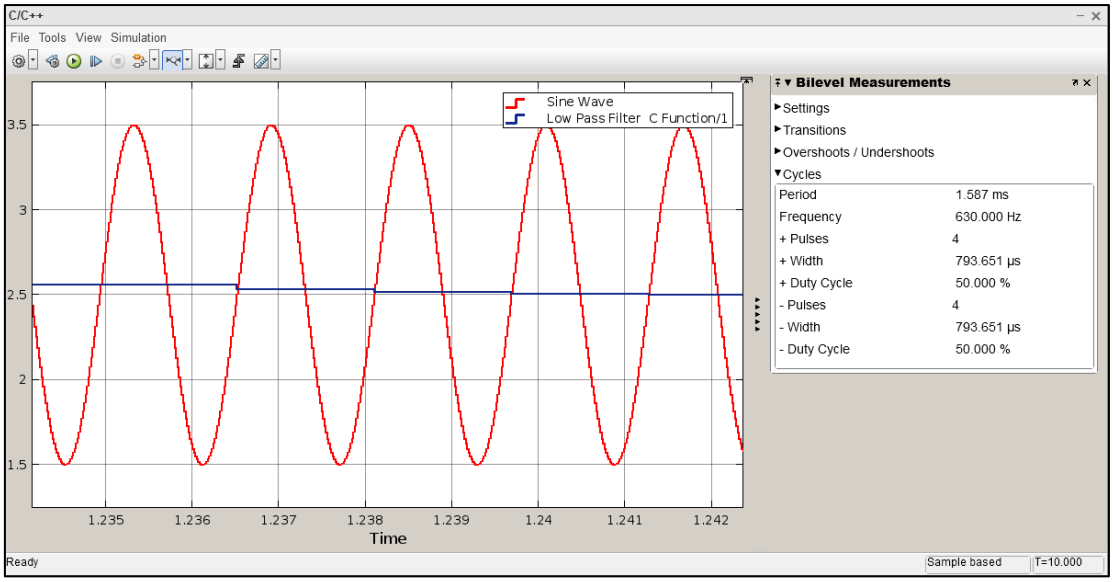

Figura 21 – Respuesta del filtro paso alto C/C++ en alta frecuencia

Filtro Paso Alto (High Pass Filter)

Fts – Filtro

Parámetros	Resultados
NIA	787246
f_{cutoff}	$70\ Hz$
ω_c	$439.8230\ \frac{rad}{s}$
G_s	$\frac{s+43.98}{s+439.8}$

Tabla 4 – Parámetros del filtro paso alto continuo

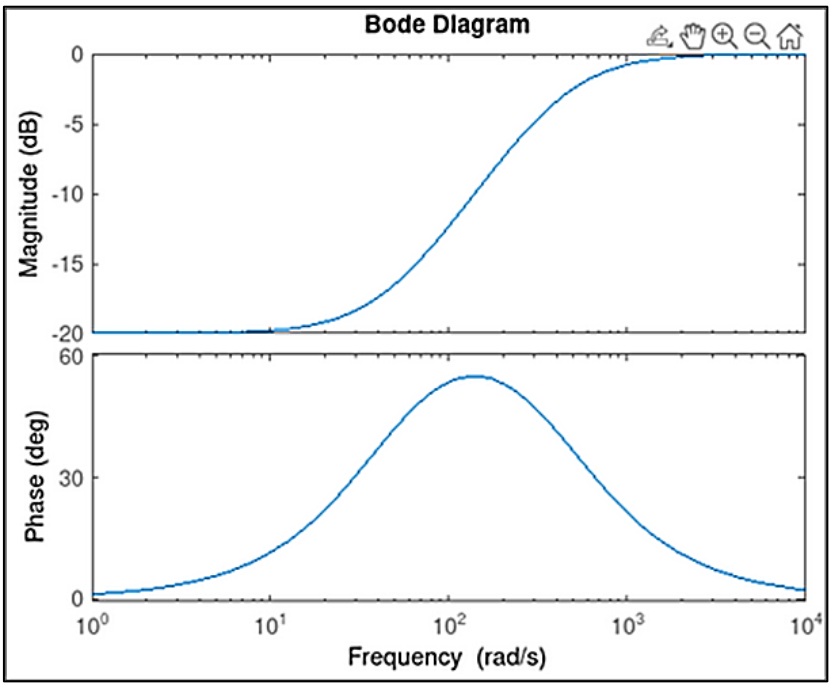

Figura 22 - Diagramas de bode del filtro Paso Alto Continuo

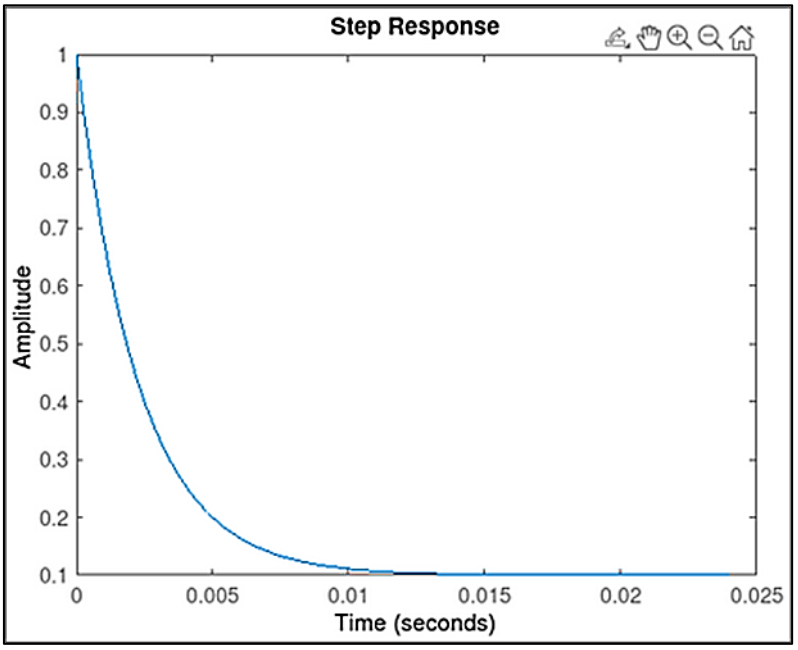

Figura 23 - Respuesta al escalon unitario del filtro Paso Alto Continuo

FTz – Filtro

Parámetros	Resultados
NIA	787246
f_{cutoff}	$70\ Hz$
f_{sample}	$70000\ Hz$
T_S	$1.4286 \cdot 10^{-5}\ s$
ω_c	$439.8230 \frac{rad}{s}$
G_z	$\frac{z - 0.9994}{z - 0.9937}$

Tabla 5 – Parámetros del filtro paso alto discreto

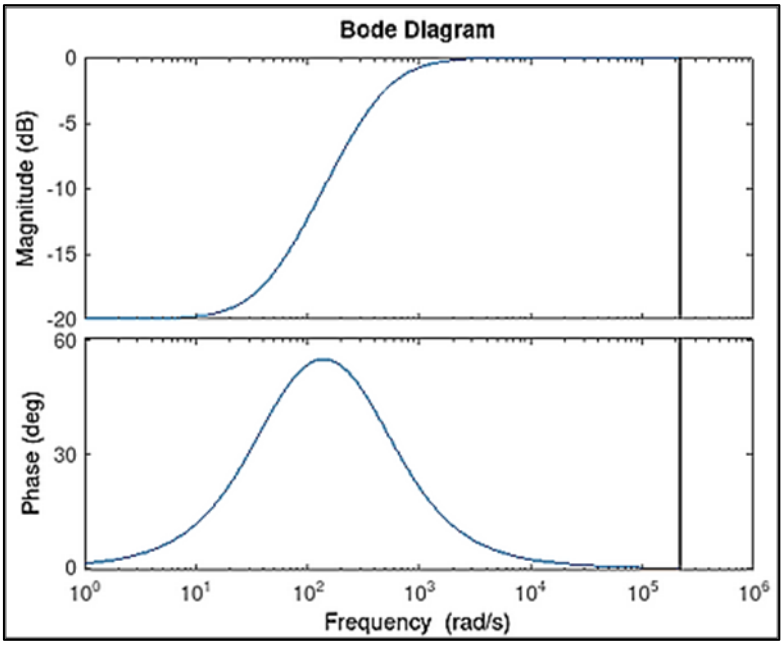

Figura 24 - Diagramas de bode del filtro Paso Alto Discreto

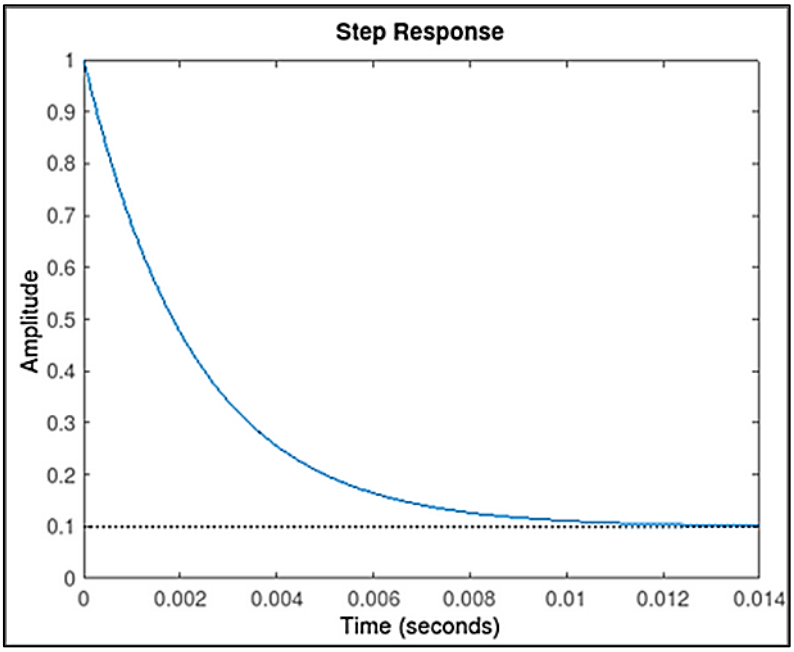

Figura 25 - Respuesta al escalon unitario del filtro Paso Alto Discreto

Implementación del filtro en C/C++

$$G_s = \frac{U(z)}{E(z)} = \frac{z - 0.9994\,z^{-1}}{z - 0.9937\,z^{-1}} = \frac{1 - 0.9994z^{-1}}{1 - 0.9937z^{-1}}$$

$$U(z)(1 - 0.9937z^{-1}) = E(z)(1 - 0.9994z^{-1})$$
$$U(z) = E(z) - 0.9994E(z)z^{-1} + 0.9937U(z)z^{-1}$$

$$U_k = E_k - 0.9994E_{k-1} + 0.9937U_{k-1}$$

Parámetros	Resultados
M	2
N	2
b[M]	{1, -0.9994}
a[N]	{0, 0.9937}

Tabla 6 – Parámetros para el código C/C++ del filtro paso bajo

Generación del código fuente C/C++

- HighPass.h
- HighPass.c

HighPass.h

```
#ifndef HIGHPASS_H
#define HIGHPASS_H

#define N   2
#define M   2

double controller(double E, double* b, double* a);
double HighPass(double Ek, double Ts, double time, double* aux);

#endif /*   HIGHPASS_H   */
```

HighPass.c

```
#include "HighPass.h"
```

```
double HighPass(double Ek, double Ts, double time, double* aux){
    static int k=0;
    static double Xk=0;

    double b[M]={1, -0.9994};
    double a[N]={0, 0.9937};

    if(time < Ts){
        k=0;
    }
    else if(time >= k*Ts){
        Xk=controller(Ek, b, a);
        k++;
        *aux=k*Ts;
    }

    return Xk;
}

double controller(double E, double* b, double* a){

    static int init_arrays = 0;
    static double U[N];
    static double E_values[M];

    if(!init_arrays){
        for(int i=0;i<N; i++) U[i]=0;
        for(int i=0;i<M; i++) E_values[i]=0;
        init_arrays = 1;
    }

    E_values[0]=E;
    double sum_Ek, sum_Uk;
    sum_Ek=0; sum_Uk=0;

    for(int k=0; k<M; k++)  sum_Ek = sum_Ek + b[k]*E_values[k];
    for(int k=0; k<N; k++)  sum_Uk = sum_Uk + a[k]*U[k];

    U[0]=sum_Ek+sum_Uk;
    for(int k=1; k<M; k++)  E_values[k]=E_values[k-1];
    for(int k=1; k<N; k++)  U[k]=U[k-1];

    return U[0];
}
```

Simulación - Simulink

HP_config.m

```
%% INPUT SIGNAL
amplitude_input=1; % Volt
offset_input=0; % Volt
f_input=10; %Hz
w_input=2*pi*f_input; % rad/s

%% Continous HIGH PASS FILTER
Fc_Limit_Arduino = 150; % Hz
A=7; B=8; C=7; D=2; E=4; F=6;
f_cutoff = (1*F+1*E)*1*C*10; % Cuttoff frecuency (Hz)

if f_cutoff > Fc_Limit_Arduino
    f_cutoff=f_cutoff/10;
end
```

```
wc=2*pi*f_cutoff; % Angular frecuency (rad/s)
numGs=[1 wc/10];
denGs=[1 wc];

Gs=tf(numGs, denGs); % Transfer function (Tf) in the Laplace domain
bode(Gs);
step(Gs);

%% DIGITAL HIGH PASS FILTER
f_sample=1000*f_cutoff; % Sample frecuency (Hz)
Ts=1/f_sample;          % Sample rate (seconds = s)
Gz=c2d(Gs, Ts);

numGz=cell2mat(Gz.Numerator);
denGz=cell2mat(Gz.Denominator);

bode(Gz);
step(Gz);

%% Simulation
Tsim=0.1*Ts;
```

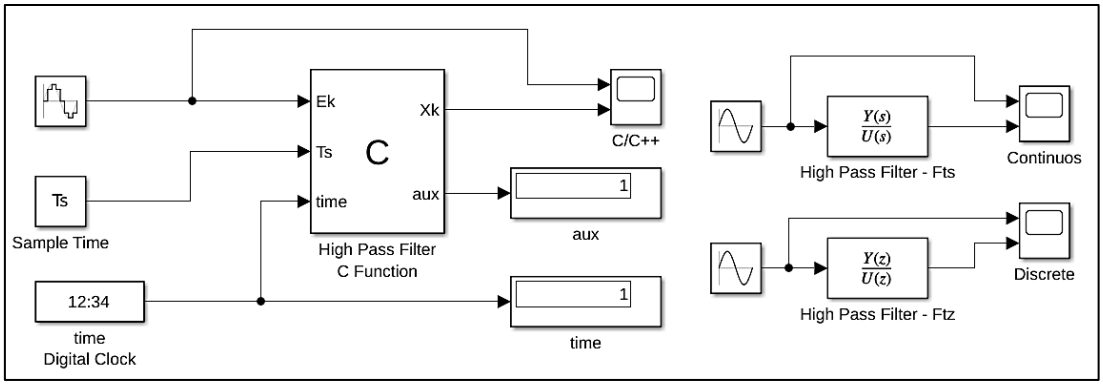

Figura 26 – Diagrama de bloques Filtro Paso Bajo en todos modos

Filtro – Continuos

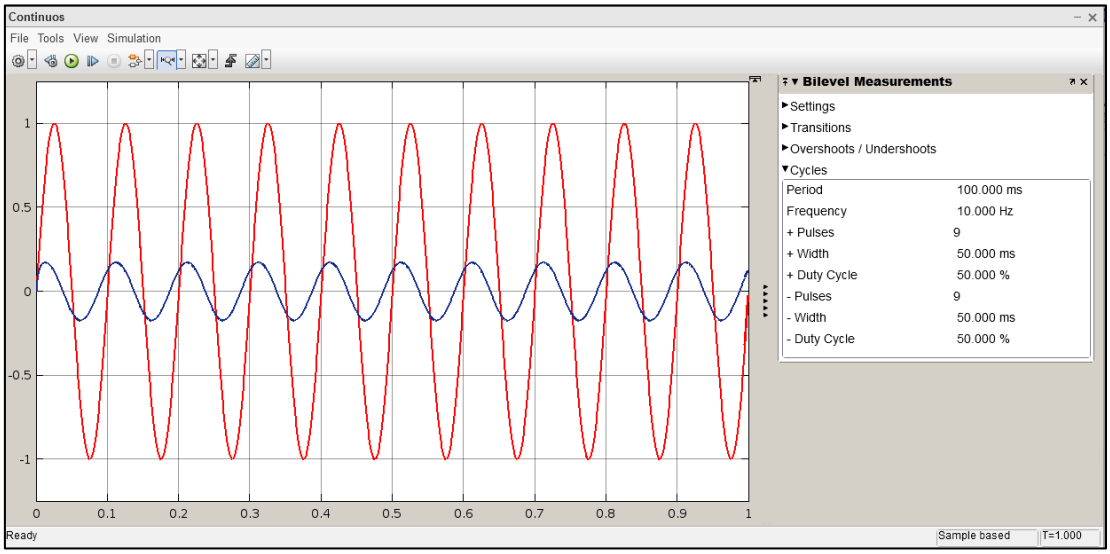

Figura 27 – Respuesta del filtro paso alto continuo en baja frecuencia

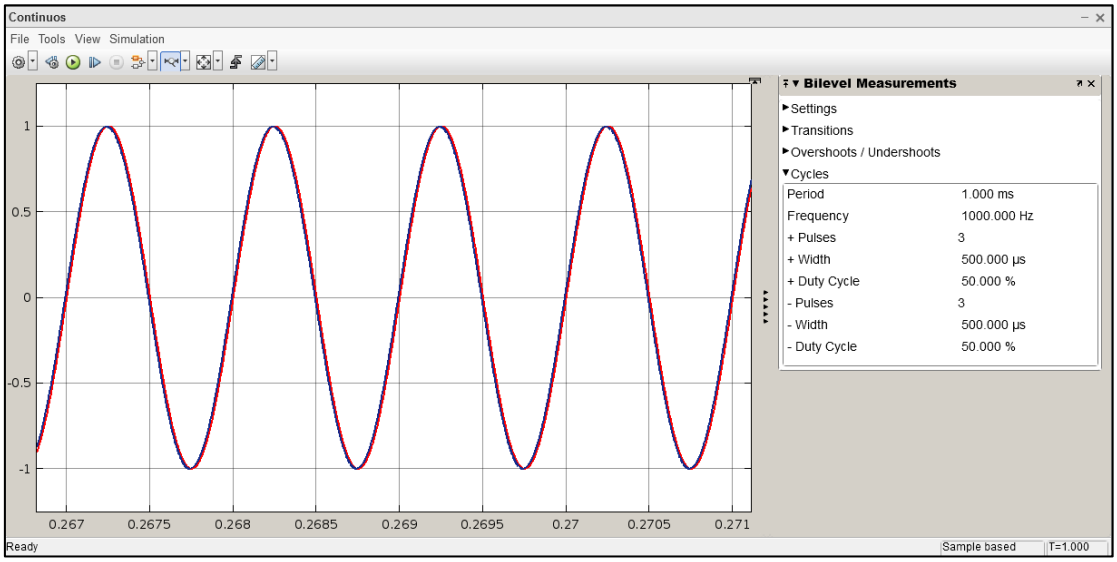

Figura 28 – Respuesta del filtro paso alto continuo en alta frecuencia

Filtro – Discreto

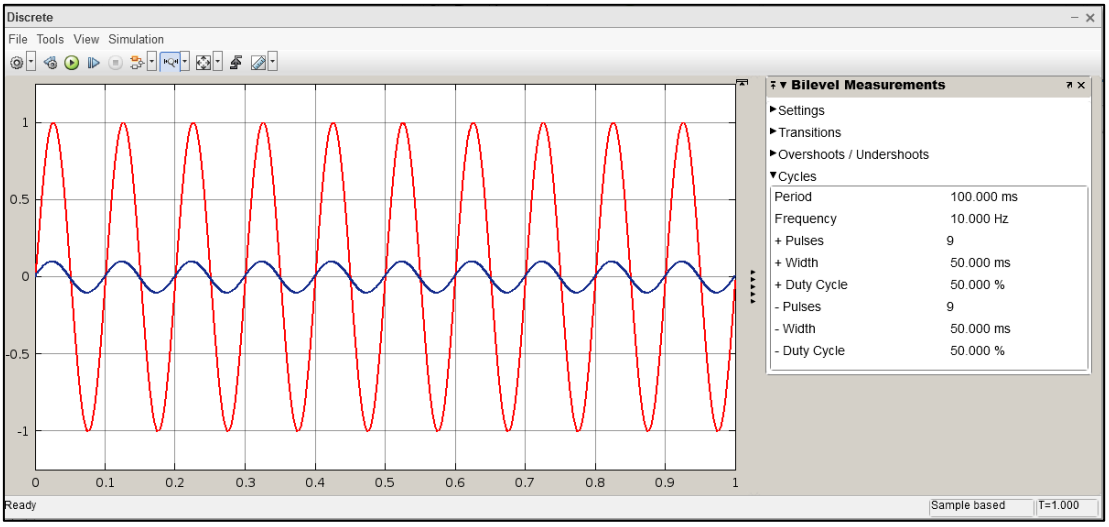

Figura 29 – Respuesta del filtro paso alto discreto en baja frecuencia

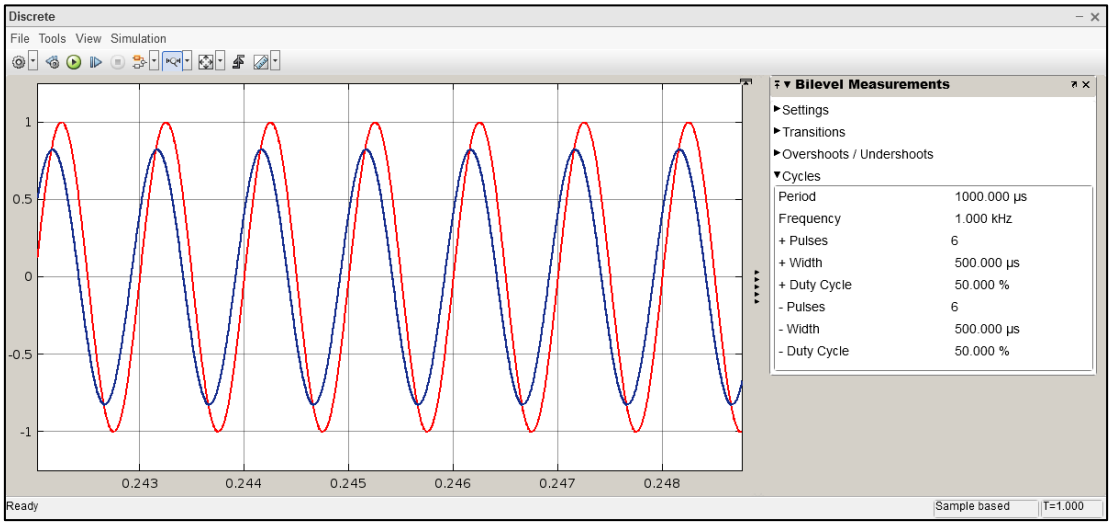

Figura 30 – Respuesta del filtro paso alto discreto en alta frecuencia

Filtro – Código C/C++

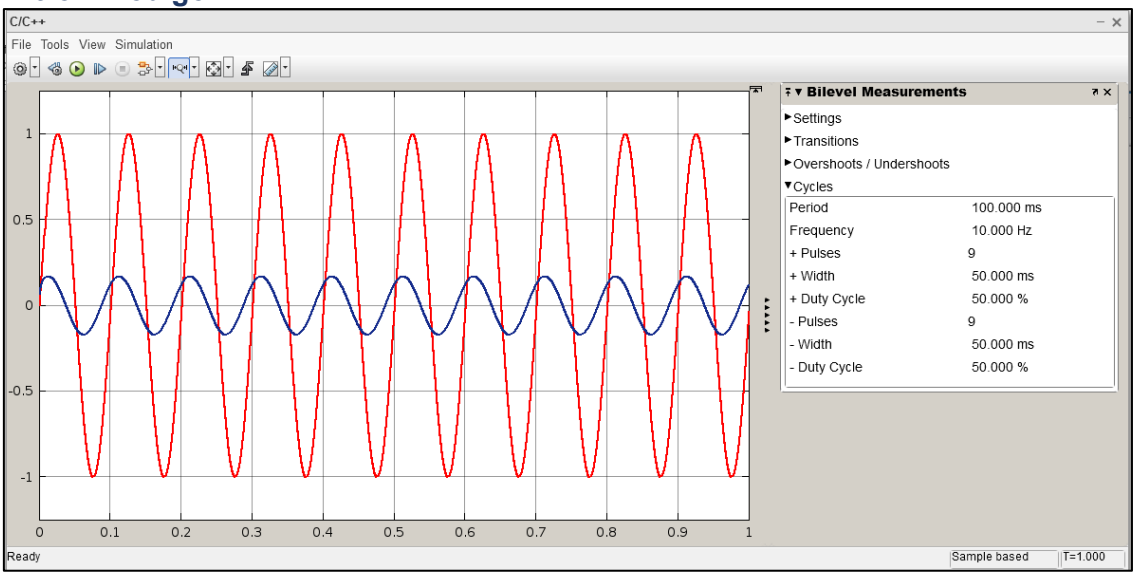

Figura 31 – Respuesta del filtro paso alto C/C++ en baja frecuencia

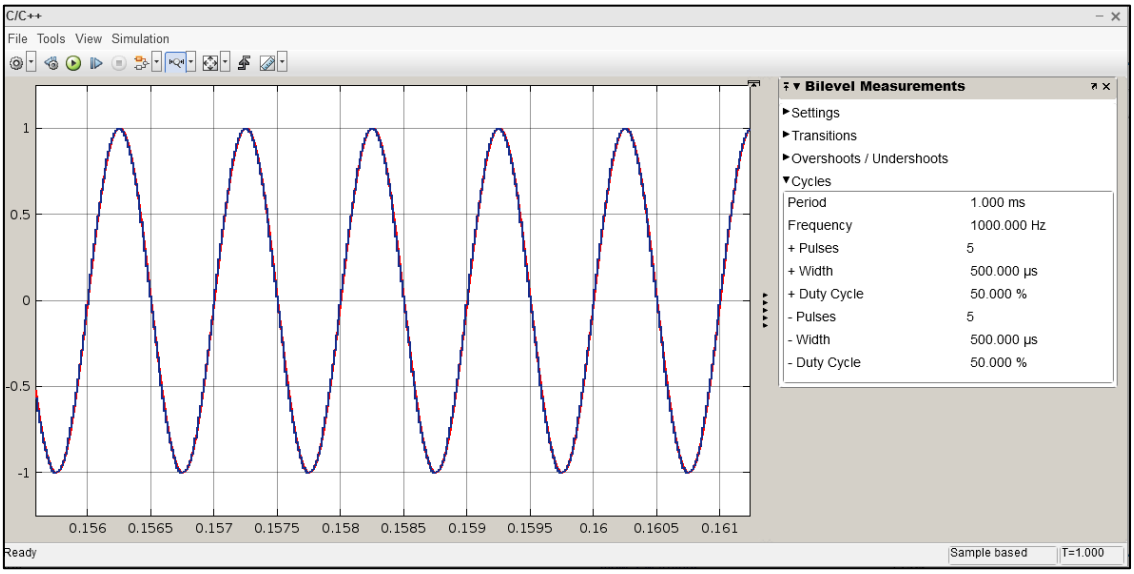

Figura 32 – Respuesta del filtro paso alto C/C++ en alta frecuencia

Filtro Paso Banda (Band Pass Filter)

Fts - Filtro

Parámetros	Resultados
NIA	787246
f_{cutoff_L}	$130\ Hz$
f_{cutoff_H}	$30\ Hz$
ω_{c_L}	$816.8141\ \frac{rad}{s}$
ω_{c_H}	$188.4956\ \frac{rad}{s}$
G_s	$\frac{816.8\,s\ +\ 15400}{s^2 + 1005s + 154000}$

Tabla 7 – Parámetros del filtro paso alto continuo

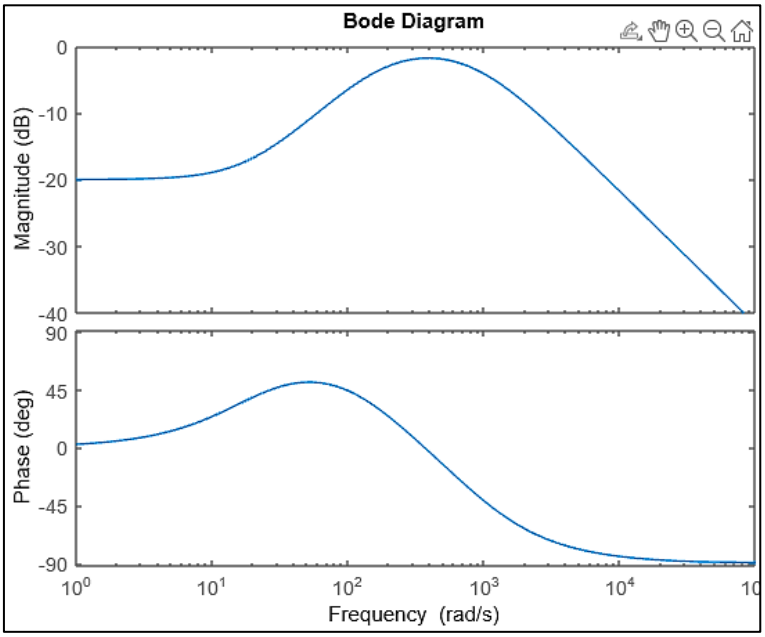

Figura 33 - Diagramas de bode del filtro Paso Banda Continuo

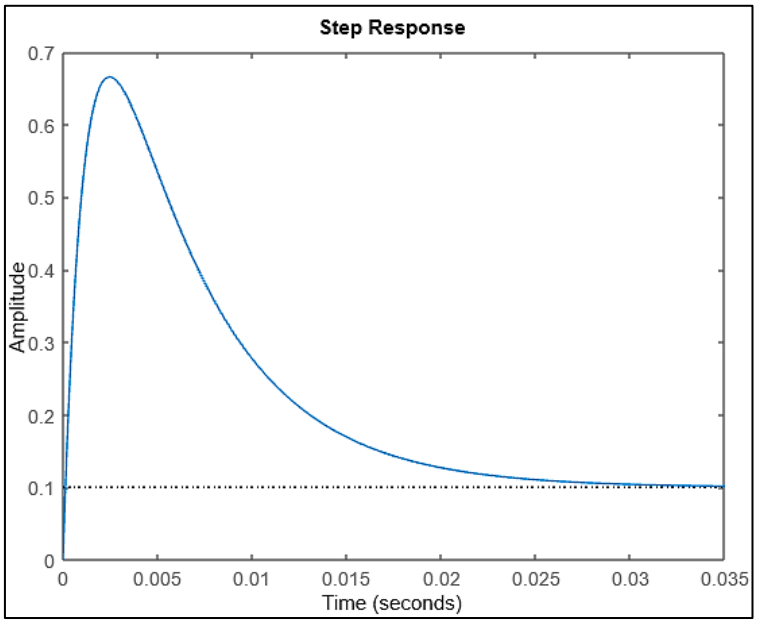

Figura 34 - Respuesta al escalon unitario del filtro Paso Banda Continuo

FTz - Filtro

Parámetros	Resultados
NIA	787246
f_{cutoff_L}	$130\ Hz$
f_{cutoff_H}	$30\ Hz$
f_{sample}	$1300\ Hz$
T_S	$0.00076923\ s$
ω_{c_L}	$816.8141\ \frac{rad}{s}$
ω_{c_H}	$188.4956\ \frac{rad}{s}$
G_z	$\frac{0.4345\ z\ -\ 0.4283}{z\text{^}2\ -\ 1.399\ z\ +\ 0.4615}$

Tabla 8 – Parámetros del filtro paso banda discreto

Implementación del filtro en C/C++

$$G_s = \frac{U(z)}{E(z)} = \frac{0.6118\,z\ -\ 0.5624}{z\text{^}2\ -\ 0.5275\,z\ +\ 0.02093}\frac{z^{-2}}{z^{-2}} = \frac{0.6118\,z^{-1}\ -\ 0.5624z^{-2}}{1\ -\ 0.5275\,z^{-1}\ +\ 0.02093z^{-2}}$$

$$U(z)(1\ -\ 0.5275\,z^{-1}\ +\ 0.02093z^{-2}) = E(z)(0.6118\,z^{-1}\ -\ 0.5624z^{-2})$$
$$U(z) = 0.6118E(z)z^{-1}\ -\ 0.5624E(z)z^{-2} + 0.5275U(z)z^{-1} - 0.02093U(z)z^{-2}$$

$$U_k = 0.6118E_{k-1}\ -\ 0.5624E_{k-2} + 0.5275U_{k-1} - 0.02093U_{k-2}$$

Parámetros	Resultados
M	3
N	3
b[M]	$\{0.0, 0.6118, -0.5624\}$
a[N]	$\{0.0, 0.5275, -0.02093\}$

Tabla 9 – Parámetros para el código C/C++ del filtro paso banda

Generación del código fuente C/C++

- BandPass.h
- BandPass.c

BandPass.h

```
#ifndef BANDPASS_H
#define BANDPASS_H

#define N   3
#define M   3

double controller(double Ek, double* b, double* a);
double BandPass(double Ek, double Ts, double time, double* aux);

#endif /*   LOWPASS_H   */
```

BandPass.c

```
#include "BandPass.h"

double BandPass(double Ek, double Ts, double time, double* aux){
    static int k=0;
    static double Xk=0;

    double b[M]={0.0, 0.4345, -0.4283};
    double a[N]={0.0, 1.3990, -0.4615};

    if(time < Ts){
        k=1;
    }
    else if(time >= k*Ts){
        Xk=controller(Ek, b, a);
        k++;
        *aux=k*Ts;
    }
    return Xk;
}

double controller(double E, double* b, double* a){
```

```
    static int init_arrays = 0;
    static float U[N];
    static float E_values[M];

    if(!init_arrays){
        for(int i=0;i<N; i++) U[i]=0;
        for(int i=0;i<M; i++) E_values[i]=0;
        init_arrays = 1;
    }

    E_values[0]=E;

    float sum_Ek, sum_Uk;

    sum_Ek=0;
    sum_Uk=0;

    for(int k=0; k<M; k++)  sum_Ek = sum_Ek + b[k]*E_values[k];
    for(int k=0; k<N; k++)  sum_Uk = sum_Uk + a[k]*U[k];

    U[0]=sum_Ek+sum_Uk;

    for(int k=(M-1); k>0; k--)  E_values[k]=E_values[k-1];
    for(int k=(N-1); k>0; k--)  U[k]=U[k-1];

    return U[0];
}
```

Simulación - Simulink

BP_config.m

```
%% INPUT SIGNAL
amplitude_input=1; % Volt
offset_input=0; % Volt
f_input=60; % Hz (5 veces la fc_H)
w_input=2*pi*f_input; % rad/s

%% CONTINUOUS BAND PASS FILTER
fc_L=130;
fc_H=30;

% Angular frecuency (rad/s)
wc_L=2*pi*fc_L;
wc_H=2*pi*fc_H;

% Transfer function (Tf) in the Laplace domain
numGs=[wc_L (wc_H*wc_L)/10];
denGs=[1 (10*wc_H+10*wc_L)/10 (10*wc_H*wc_L)/10];
Gs=tf(numGs, denGs);

bode(Gs);
step(Gs);

%% DISCRETE BAND PASS FILTER (DIGITAL)
f_sample=10*fc_L;     % Sample frecuency (Hz)
Ts=1/f_sample;        % Sample rate (seconds = s)

% Tf in Z domain
Gz=c2d(Gs, Ts);
numGz=cell2mat(Gz.Numerator);
denGz=cell2mat(Gz.Denominator);
```

```
len_numGz=length(numGz);
len_denGz=length(denGz);

bode(Gz)
step(Gz);

%% Simulation
Tsim=0.1*Ts;
```

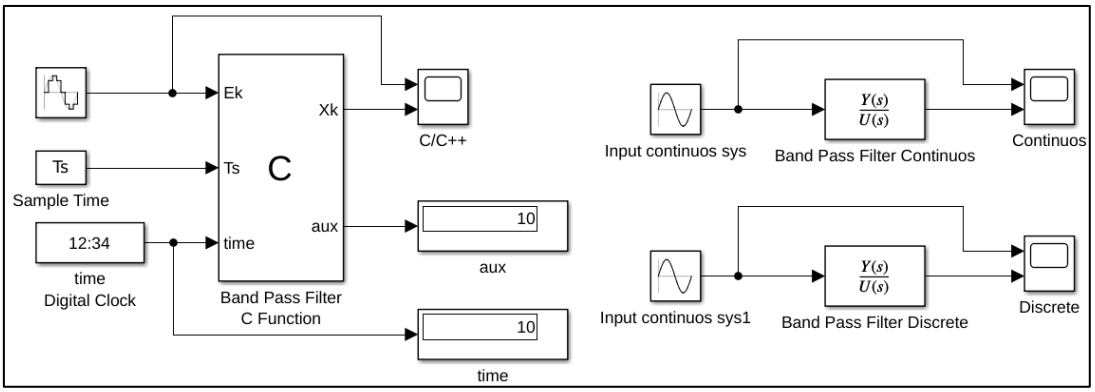

Figura 35 – Diagrama de bloques Filtro Paso Banda en todos modos

Filtro – Continuos

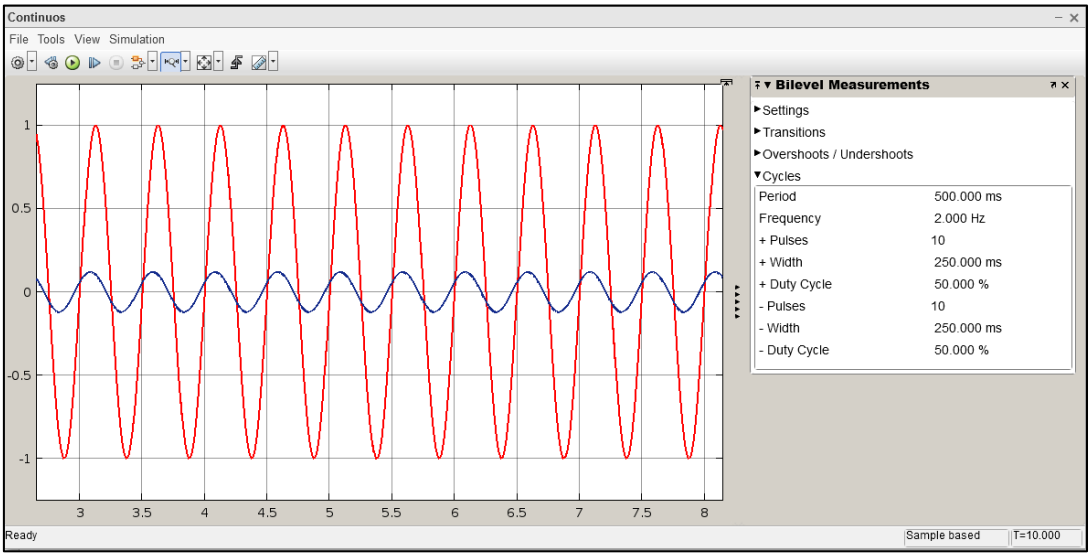

Figura 36 – Respuesta del filtro paso Banda continuo en baja frecuencia

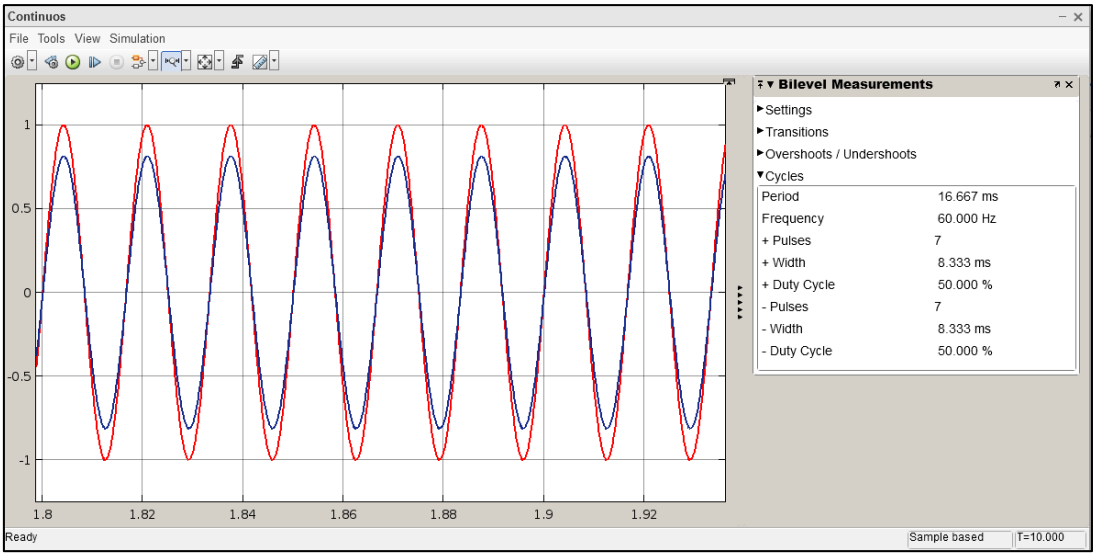

Figura 37 – Respuesta del filtro paso Banda continuo en media frecuencia

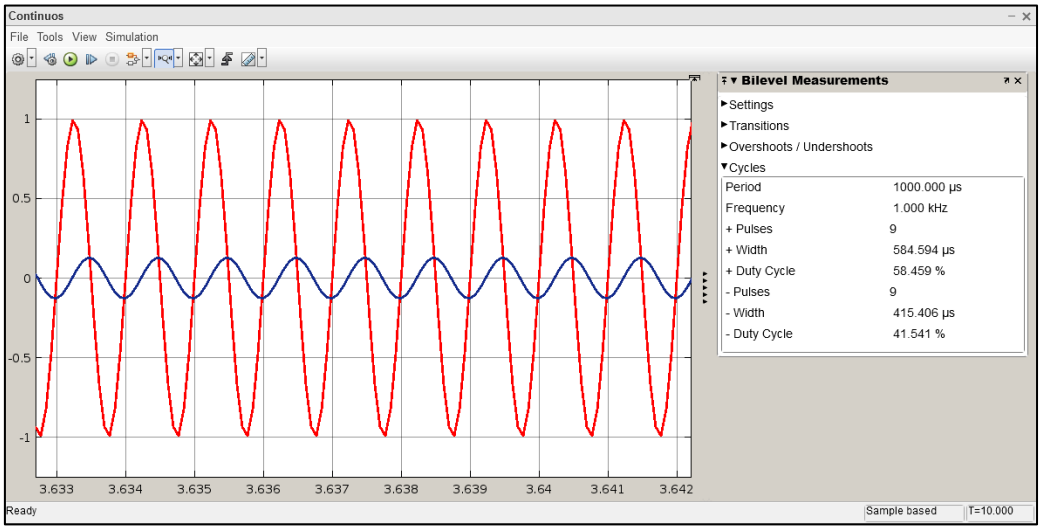

Figura 38 – Respuesta del filtro paso Banda continuo en alta frecuencia

ANEXOS

Código de Regulador genérico

```
double controller(double E, double* b, double* a){
    static int init_arrays = 0;
    static double U[N];
    static double E_values[M];

    if(!init_arrays){
        for(int i=0;i<N; i++) U[i]=0;
        for(int i=0;i<M; i++) E_values[i]=0;
        init_arrays = 1;
    }

    E_values[0]=E;

    double sum_Ek, sum_Uk;

    for(int k=0; k<M; k++)  sum_Ek=b[k]*E_values[k];
    for(int k=0; k<N; k++)  sum_Uk=a[k]*U[k];

    U[0]=sum_Ek+sum_Uk;

    for(int k=1; k<M; k++)  E_values[k]=E_values[k-1];
    for(int k=1; k<N; k++)  U[k]=U[k-1];

    return U[0];
}
```