***DSP Education Kit***

**LAB 4**

**Infinite Impulse Response (IIR) Filters**

**Issue 1.0**

Contents

[1 Introduction 1](#_Toc29195183)

[1.1 Lab overview 1](#_Toc29195184)

[2 Requirements 1](#_Toc29195185)

[3 The Infinite Impulse Response Filter 1](#_Toc29195186)

[4 Design of a Simple IIR Low Pass Filter 2](#_Toc29195187)

[4.1 Impulse invariance method 3](#_Toc29195188)

[4.2 Exercise Part 1: Writing a C program to implement an example filter 5](#_Toc29195189)

[4.3 Exercise Part 2: Experimental measurement of the magnitude frequency response of the example filter 8](#_Toc29195190)

[5 Bilinear Transform Method of Digital Filter Implementation 10](#_Toc29195191)

[5.1 Exercise 10](#_Toc29195192)

[6 Implementing a Filter Designed Using MATLAB 12](#_Toc29195193)

[7 Recursive Generation of a Sine Wave 13](#_Toc29195194)

[7.1 Exercise 14](#_Toc29195195)

[8 Conclusions 14](#_Toc29195196)

# Introduction

## Lab overview

The examples in this exercise introduce some of the concepts of Infinite Impulse Response (IIR) filtering. Also explored are various methods of estimating the magnitude frequency response of a filter implemented in real time and the relative computational efficiency of different implementation options.

# Requirements

To carry out this lab, you will need:

* An STM32F746G Discovery board
* A PC running Keil MDK-Arm
* MATLAB
* An oscilloscope
* Suitable connecting cables
* An audio frequency signal generator
* Headphones

# The Infinite Impulse Response Filter

Whereas the output samples of a finite impulse response (FIR) filter depend only on the values of previous input samples (which are convolved with the FIR filter coefficients), the output samples of an infinite impulse response (IIR) filter depend also on the values of previous output samples. The difference equation describing the operation of an IIR filter may be written as

, (1)

where *y*(*n*) is the *n*th output sample and *x*(*n*) is the *n*th input sample. Comparing this with the difference equation describing the operation of an FIR filter

  (2)

It is apparent that FIR filters are a subset of IIR filters (in which all of the *a* coefficients in equation (1) are equal to zero.)

Many different approaches to the design of IIR filters are possible and often they are designed with the aid of software tools. Before using such a tool, and in order to appreciate better what such tools do, we will use a simple example to illustrate some of the basic principles of IIR filter design.

# Design of a Simple IIR Low Pass Filter

Traditionally, IIR filter design is based on the concept of transforming a continuous-time, or analogue, filter design into the discrete-time domain. Butterworth, Chebyshev, Elliptic, and Bessel classes of analogue filter are examples of the extensive knowledge that exists. For our example, we will use a second-order, type 1 Chebyshev, low pass filter with 2 dB of passband ripple and a cutoff frequency of 1500 Hz (9425 rad/s). We will use a sampling frequency of 8 kHz.

The continuous-time transfer function of this filter is

$H(s)=\frac{58072962}{s^{2}+7576s+73109527}$ (3)

and its frequency response is shown in Figure 1 and Figure 2. Our aim is to design an equivalent discrete-time IIR filter.



Figure 1: Magnitude frequency response of filter H(s)



Figure 2: Phase frequency response of filter H(s)

## Impulse invariance method

Starting with the filter transfer function (3), we can make use of the Laplace transform pair

, (4)

where in our example

The impulse response of a continuous-time filter is equal to the inverse Laplace transform of its transfer function, and hence the impulse response of the filter in this example is given by

 (5)

The *z*-transfer function of that impulse response may be determined from the *z*-transform pair

$Ζ\left\{Ae^{-αt}\sin(()ωt)\right\}=\frac{Ae^{-αt\_{s}}\sin(()ωt\_{s})z^{-1}}{1-2(e^{-αt\_{s}}\cos(()ωt\_{s}))z^{-1}+e^{-2αt\_{s}}z^{-2}}$ (6)

Substituting for , ,  and setting in equation (6) yields

 (7)

|  |
| --- |
| Note: The analogue filter example that this digital IIR filter design is based upon has a magnitude of 0.7943 (20 log10 0.7943 = -2.0dB) at DC (low frequencies), as shown in Figure 1. This can also be determined from the H(s) equation in equation (3) when evaluated at s=0, where 58072962/73109527 = 0.7943. From Figure 1, the maximum magnitude gain of the filter is approximately 0dB at around 1kHz. You will notice that if you substitute the , ,, and $t\_{s}$ values in equation (6), you will get a numerator coefficient of approximately 3860.4. With this numerator, the gain of this digital filter is significantly higher (20 log10 3860.4=71dB) than the gain of the analogue filter gain used in this example (Figure 1).To match the digital design filter gain as closely as possible with the example analogue filter, the digital filter gain needs to be reduced or scaled down.The assumption is that the Laplace Transform implementation of the Impulse Invariance method results in a discrete time filter whose impulse response is samples of the impulse response of the analogue prototype filter scaled by the sampling rate. That is, the gain of the discrete-time filter is fs times greater than that of the analogue prototype filter. In this example, the sampling rate is fs=8000, therefore the numerator coefficient 3860.4 is 8000 times too big. Therefore, 3860.4/8000=0.48255, which is the numerator coefficient in equation (7). This fs-scaled impulse response approach is also used in MATLAB’s impinvar() function. An alternative approach to using the fs-scaled impulse response approach is to scale H(z) so that the DC gain is -2dB, as shown in Figure 1. This means that we evaluate H(z) at z=1, in which case the numerator coefficient would be approximately 0.5335. However, you would notice that if you were to plot the frequency response for the 0.5335 numerator coefficient, the magnitude would exceed 0dB slightly, compared to the 0.48255 numerator coefficient, which stays slightly below 0dB and closely matches the analogue filter response in Figure 1. |

From equation (7), the following difference equation may be derived

 (8)

In terms of equation (1), we can see that = 0.71624315, = - 0.38791310, = 0.0000, and  = 0.48255

This discrete-time filter has the property that its impulse response *h*(*n*) is equal to samples of the continuous time impulse response *h*(*t*) (scaled by the sampling period *Ts* = 0.000125 s).



Figure 3: Impulse responses h(t) (scaled by Ts = 0.000125) and h(n) of the continuous time filter and its impulse invariant discrete-time implementation

## Exercise Part 1: Writing a C program to implement an example filter

Modify program stm32f7\_impinviir\_intr.c, listed in the code snippet below, in order to implement the filter described by the difference equation

 (8)

// stm32f7\_impinviir\_intr.c

#include "stm32f7\_wm8994\_init.h"

#include "stm32f7\_display.h"

#define SOURCE\_FILE\_NAME "stm32f7\_impinviir\_intr.c"

extern int16\_t rx\_sample\_L;

extern int16\_t rx\_sample\_R;

extern int16\_t tx\_sample\_L;

extern int16\_t tx\_sample\_R;

void BSP\_AUDIO\_SAI\_Interrupt\_CallBack()

{

 float32\_t xn, yn; // input and output values

 xn =(float32\_t)(rx\_sample\_L);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 insert code to compute new output sample here, i.e.

 y(n) = 0.48255x(n-1) + 0.71624315y(n-1) - 0.38791310y(n-2)

 also update stored previous sample values, i.e.

 y(n-2), y(n-1), and x(n-1)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

 tx\_sample\_L = (int16\_t)(yn);

 tx\_sample\_R = tx\_sample\_L;

 return;

}

int main(void)

{

 stm32f7\_wm8994\_init(AUDIO\_FREQUENCY\_8K,

 IO\_METHOD\_INTR,

 INPUT\_DEVICE\_INPUT\_LINE\_1,

 OUTPUT\_DEVICE\_HEADPHONE,

 WM8994\_HP\_OUT\_ANALOG\_GAIN\_0DB,

 WM8994\_LINE\_IN\_GAIN\_0DB,

 WM8994\_DMIC\_GAIN\_0DB,

 SOURCE\_FILE\_NAME,

 NOGRAPH);

 while(1){}

}

You will need to:

1. Declare variables in which to store previous input and output sample values *y*(*n*-1), *y*(*n*-2), and *x*(*n*-1).
2. In the interrupt service routine BSP\_AUDIO\_SAI\_Interrupt\_CallBack(), read the value of a new input sample from the Analog to Digital Converter (ADC), compute the value of a new output sample *y*(*n*) using equation (8), and write that value to the DAC.
3. Update the stored values of previous input and output values, i.e., the newly computed output value *y*(*n*) will need to be stored as *y*(*n*-1) and the value of *y*(*n*-1) will need to be stored as y(n-2).

If you implement the filter in this straightforward manner, you will almost certainly have used a *direct form I* implementation as shown in Figure 4.



Figure 4: Direct form I implementation of a second-order IIR filter

From the block diagram of Figure 4, it can be seen that

 (9)

An alternative IIR filter structure, *direct form II*, is illustrated in Figure 5.



Figure 5: Direct form II implementation of a second-order IIR filter

From that block diagram, it can be seen that

 (10)

 (11)

The two filter structures share the same z-transfer function

 (12)

Equations (10) and (11) constitute a two-stage method of computing a filter output value *y*(*n*) that may very simply be implemented in C code.

**Question: What advantage does direct form II implementation have over direct form I?**

## Exercise Part 2: Experimental measurement of the magnitude frequency response of the example filter

Use a signal generator and oscilloscope to measure the magnitude frequency response of the filter, entering your results in Table 1 and plotting them on the graph in Figure 6 (which shows the magnitude frequency response of the analogue prototype filter).

You can expect to see some high-frequency noise in the output signal, and you will get the most accurate results if the output voltage is as large as possible relative to that noise. However, if you use an input signal too large, you will exceed the range of the ADC and introduce distortion. Try to ensure that the output signal does not appear distorted before measuring its magnitude.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Frequency (Hz) | Vin (volts) |  | Vout (volts) | Vout/Vin | Gain (dB) |
| 100 |  |  |  |  |  |
| 200 |  |  |  |  |  |
| 300 |  |  |  |  |  |
| 500 |  |  |  |  |  |
| 1000 |  |  |  |  |  |
| 1500 |  |  |  |  |  |
| 1600 |  |  |  |  |  |
| 1800 |  |  |  |  |  |
| 2000 |  |  |  |  |  |
| 2400 |  |  |  |  |  |
| 2800 |  |  |  |  |  |
| 3200 |  |  |  |  |  |

Table 1: Experimentally measured magnitude frequency response of the impulse invariant discrete-time filter



Figure 6: Experimentally measured magnitude frequency response of the impulse invariant discrete-time filter (red) (magnitude frequency response of analogue prototype filter shown for comparison (blue))

**Comment, in the space below, on the result plotted in Figure 6. You are looking for discrepancies between the magnitude frequency responses of the digital filter and of the analogue prototype.**

# Bilinear Transform Method of Digital Filter Implementation

The bilinear transform method is relatively straightforward, often involving less algebraic manipulation than the impulse invariant method. It is achieved by making the substitution

  (13)

in the continuous-time transfer function of the analogue prototype filter *H*(*s*), where *T* is the sampling period of the digital filter, that is,

 (14)

## Exercise

Apply the bilinear transform method to the example filter of equation (3) and write down the z-transfer function *H*(*z*) and difference equation describing the filter in the space below. Show your work in the space provided below:

Implement this filter by modifying program stm32f7\_impinviir\_intr.c again and measure its magnitude frequency response, entering your results in Table 2 and plotting on the axes of Figure 7.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Frequency (Hz) | Vin (volts) |  | Vout (volts) | Vout/Vin | Gain (dB) |
| 100 |  |  |  |  |  |
| 200 |  |  |  |  |  |
| 300 |  |  |  |  |  |
| 500 |  |  |  |  |  |
| 1000 |  |  |  |  |  |
| 1500 |  |  |  |  |  |
| 1600 |  |  |  |  |  |
| 1800 |  |  |  |  |  |
| 2000 |  |  |  |  |  |
| 2400 |  |  |  |  |  |
| 2800 |  |  |  |  |  |
| 3200 |  |  |  |  |  |

Table 2: Experimentally measured magnitude frequency response of bilinear transform discrete-time filter



Figure 7: Experimentally measured magnitude frequency response of the bilinear transform discrete-time filter (red) (magnitude frequency response of analogue prototype filter shown for comparison (blue))

**Comment on the result plotted in Figure 7 in the space below. You are looking for discrepancies between the magnitude frequency responses of the digital filter and the analogue prototype.**

# Implementing a Filter Designed Using MATLAB

To implement a filter designed using *fdatool*, carry out the following steps:

1. Design the IIR filter using *fdatool*—Enter the parameters for a fourth-order elliptical low pass IIR filter with a cutoff frequency of 800 Hz, 1 dB ripple in the pass band and 50 dB of stop band attenuation.
2. Click on *Design Filter* and then look at the characteristics of the filter using different options from the *Analysis* menu.
3. Click on *Export* in the *fdatool File* menu.
4. Choose *workspace*, *coefficient*, *SOS,* and *G* and click on *OK*
5. At the MATLAB terminal, type stm32f7\_iirsos\_coeffs(SOS,G) and enter a filename, e.g., elliptic.h.

Program stm32f7\_iirsos\_intr.c is a generic IIR filter program that uses cascaded second-order sections and reads the filter coefficients from a separate header file. In order to implement a filter that you have designed, edit the line in program stm32f7\_iirsos\_intr.c that reads

#include “bp2000.h”

To include the header file containing your filter coefficients, e.g.,

#include “elliptic.h”

You can observe the characteristics of an IIR filter implemented using program stm32f7\_iirsos\_intr.c using a signal generator and an oscilloscope. Alternatively, some variants of the program have been provided to enable you to see the filter characteristics more easily:

* Program stm32f7\_iirsosdelta\_intr.c uses as input a repeated sequence of a single non-zero sample value followed by 255 zero-value samples. This enables you to use an oscilloscope to see a close approximation to the impulse response of the filter.
* Program stm32f7\_iirsosprbs\_intr.c uses as input a pseudorandom binary noise sequence. This enables you to use the FFT function on an oscilloscope, or *GoldWave*, to see a close approximation to the magnitude frequency response of the filter.

# Recursive Generation of a Sine Wave

The *z*-transform of a sinusoidal sequence  is given by

 (15)

Considering *h*(*n*) as the impulse response and *H*(*z*) as the *z*-transfer function of a second-order filter, it is apparent that by appropriate choice of coefficients, we can configure the second-order filter to act as a sine wave generator, i.e., to have a sinusoidal impulse response.

The difference equation corresponding to the *z*-transfer function of equation (15) is

 (16)

Since the sinusoidal output we require is the response of the filter to an impulse, i.e., *x*(*n*-1) will be equal to zero for all values of *n* except *n* = 1, equation (16) may be simplified to

 (17)

with non-zero initial values for *y*(*n*-1) and *y*(*n*-2).

The frequency of the oscillator depends on the values of the coefficients *a*1 = 2cos(ω*t*) and *a*2 = 1.0. The amplitude of the output depends on the initial conditions *y*(*n*-1) and *y*(*n*-2).



Figure 8: Block diagram representation of equation (20).

One simple solution that will yield a sinusoidal output of amplitude *A* is



## Exercise

Write a program, based on stm32f7\_impinviir\_intr.c, that uses the recursive method to generate a sine wave of frequency 1500 Hz. Note that the WM8994 DAC expects a 16-bit signed integer input. You should think in terms of the amplitude *A* of the sine wave generated as an integer value of up to 32767.

#  Conclusions

This exercise has introduced the IIR filter. Typically, IIR filter designs are based on analogue filter prototypes that are transformed into the discrete time domain using either the impulse invariant method or (more commonly) the bilinear transform.

Many filter design packages, including *MATLAB*’s fdatool, use this approach. The impulse invariant method can suffer from aliasing problems. The bilinear transform eliminates this problem but distorts the frequency response of the prototype analogue design.