
# ASN Filter Designer

## Filter Script user's guide

March 2017

ASN15-DOC002, Rev. 8

**For public release**

# Legal notices



Advanced Solutions Nederland B.V.
www.advsolned.com
support@advsolned.com



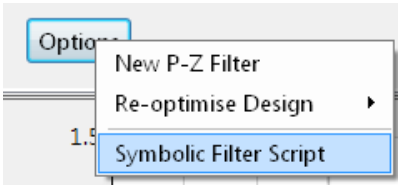
# Technical documentation feedback

If you would like to make a suggestion or report an error in our documentation, please feel free to contact us. You are kindly requested to provide as much information as possible, including a full description of the error or suggestion, the page number and the document number/description. All suggestions or errors may be sent to: documentation@advsolned.com

# Summary

This user's guide is intended to provide users of the ASN filter designer with a concise overview of the symbolic math scripting tool.

The symbolic filter script session may be started by selecting `Symbolic Filter Script` in the options menu in the P-Z editor.



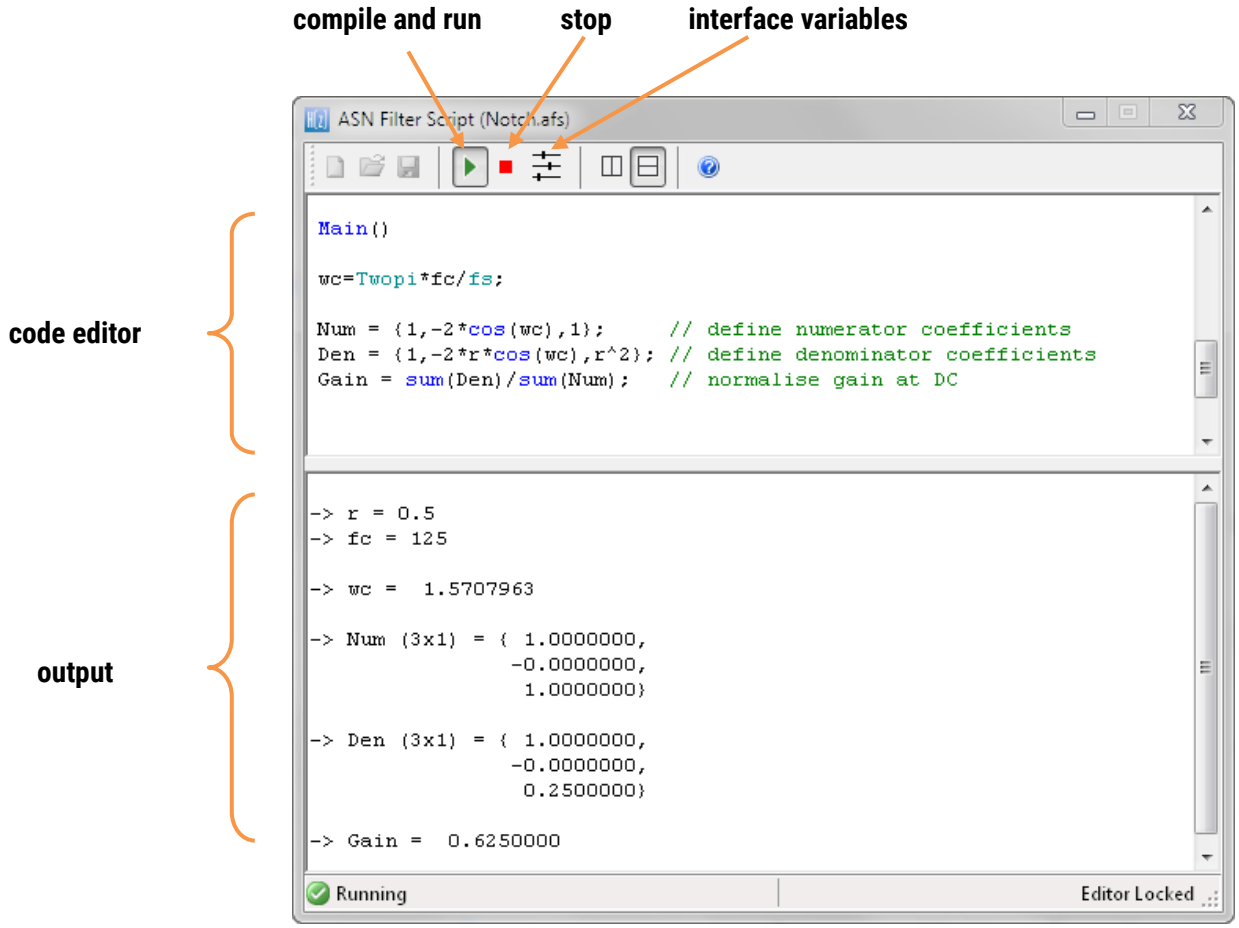

The scripting language supports over 40 scientific commands and provides designers with a familiar and powerful programming language, while at the same time allowing them to implement complex symbolic mathematical expressions for their filtering applications. The scripting language offers the unique and powerful ability to modify parameters on the fly with the so called interface variables, allowing for real-time updates of the resulting frequency response. This has the advantage of allowing the designer to see how the coefficients of the symbolic transfer function expression affect the frequency response and the filter's time domain dynamic performance.

**Resources**

| | |
|---|---|
| *ASN15-DOC001* | ASN filter designer user's guide. |
| https://youtu.be/CUC0KxegbU0 | ASN Filter script in a nutshell (video). |
| Text Book | Understanding digital signal processing, R. Lyons. |

# 1. Filter script IDE

The filter script IDE (integrated development environment) provides you with all of the necessary features in order to design and evaluate your symbolic filter concept. The IDE output is coupled to the filter designer GUI, providing a fully interactive method of customising your filter transfer function on the fly.




As seen, the IDE is split up into a code editor and an output window. The IDE differs from other scripting IDEs in that all executed code appears in the output window, and there is no provision for entering and evaluating expressions in the output window directly.

As with all standard code editors, right clicking in the editor produces a standard options menu for copying, pasting, cutting and adding/removing comments respectively.

## 1.1. Code structure

The primary purpose of the symbolic filter script is to obtain values for the following three inputs:

- `Num`: the numerator coefficients
- `Den`: the denominator coefficients
- `Gain`: the filter gain

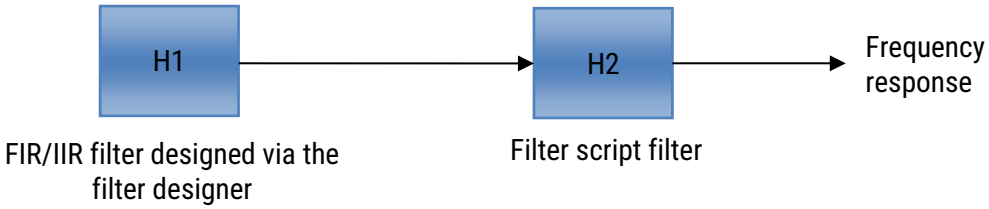
In order to provide a flexible means of modifying parameters on the fly (see section 1.2), the code is spilt up into two sections:

- *Initialisation*: contains all definitions of interface variables and any constant expressions. This section is run only once after compilation.
- *Main*: contains the bulk of the code, including the `Num`, `Den` and `Gain` expressions. Any expressions containing interface variables will be updated when modified via the interface variable GUI (see section 1.2.1).

The basic code structure is summarised below:

```
ClearH1;
interface variables
constant  expressions

Main()
{
        Num = {};
        Den = {};
        Gain = 1;
}
```

The `ClearH1` expression allows you to remove the H1 filter (primary) from the filter cascade and just use the H2 (secondary) filter. The relationship of the H1 and H2 filters is shown below:




As seen, the main FIR/IIR filter designed via the filter designer GUI is assigned to the *primary filter, H1*. All poles and zeros defined via the filter script are added to a *secondary filter block, H2*. The H2 filter block implements the filter as a single section (i.e. no biquads) IIR, which eases the implementation, but also the advantage of assigning poles to an FIR primary filter. In the case of no poles, the H2 filter becomes an FIR filter.

It should be noted that a direct form (single section) implementation may become problematic (due to numerical stability issues) for higher IIR filter orders, especially when poles are near to the unit circle.

## 1.2. Interface variables

Central to the interactivity of the tool are the so called *interface variables*. An interface variable is simply stated: *a scalar input variable that can be used modify a symbolic expression without having to re-compile the code*.

As discussed in section 1.1, interface variables must be defined in the initialisation section of the code, and may contain constants (such as, `fs` and `pi` - see section 2.7 for the complete list) and simple mathematical operators, such as multiply `*` and `/` divide. Where, adding functions to an interface variable is not supported.

An interface variable is defined as vector expression:

```
interface name = {minimum, maximum, step_size, default_value};
```

where, all entries must be real scalars values. Vectors and complex values will not compile.

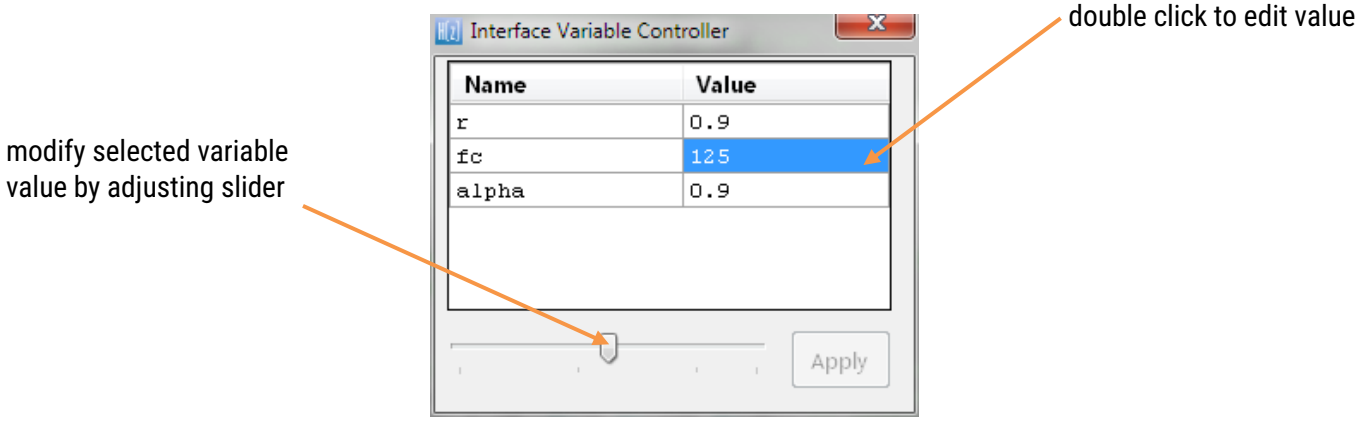
**Examples**

`interface alpha = {-1,1,0.1,0.3};` sets the variable `alpha` to `0.3`, and bounds the range to `±1`, in steps of `0.1`.

`interface fc = {-fs/2,fs/2,1,fs/10};` sets the variable `fc` to `fs/10`, and bounds the range to `±fs/2`, in steps of `1`.

### 1.2.1. User interface

All interface variables are modified via the interface variable GUI, by clicking on 




As seen, a list of valid interface variables is presented together with their current values. Where, the list is automatically updated at compilation time in order to ensure that it matches the user code.

# 2. The Scripting language

The scripting language supports over 40 scientific commands and provides designers with a familiar and powerful programming language for designing filters with the most demanding specifications.

## 2.1. Trigonometrical functions

| Function | Syntax | Description |
|---|---|---|
| angle | y = angle(x) | Compute the inverse tangent (four quadrant) |
| cos | y = cos(x) | Compute the cosine |
| cosh | y = cosh(x) | Compute the hyperbolic cosine |
| sin | y = sin(x) | Compute the sine |
| sinh | y = sinh(x) | Compute the hyperbolic sine |
| tan | y = tan(x) | Compute the tangent |
| tanh | y = tanh(x) | Compute the hyperbolic tangent |

## 2.2. Vector functions

| Function | Syntax | Description |
|---|---|---|
| cols | y = cols(x) | Gets number of columns in the vector x |
| conv | y = conv(a,b) | Computes the linear convolution of input vectors a and b. Where, the length of y is equal to length(a) + length(b)-1 |
| diff | y = diff(x) | Gets the difference between adjacent values of the vector x |
| eldef | A(3,0) = eldef(5) | Sets the vector element value to the given value |
| length | y = length(x) | Gets the vector length |
| max | y = max(x) | Gets the maximum of the vector x |
| mean | y = mean(x) | Gets the mean of the vector x |
| min | y = min(x) | Gets the minimum of the vector x |
| ones | y = ones(N) | Vector of 1s of length N |
| poly | y = poly(x) | Convert roots to polynomial |
| reverse | y = reverse(x) | Flip vector elements up-to-down |
| roots | y = roots(x) | Get the roots of polynomial x |
| rows | y = rows(x) | Gets the number of rows in vector x |
| series | y = series(min,step,max) | Creates a data series - see section 2.6 |
| sortup | y = sortup(x) | Sort vector in ascending order: smallest first, largest last |
| sortdown | y = sortdown(x) | Sort vector in descending order: largest first, smallest last |
| stddev | y = stddev(x) | Gets the standard deviation of x |
| sum | y = sum(x) | Gets the sum of vector x |
| transpose | y = transpose(x) | Transpose vector x |
| zeros | y = zeros(N) | Vector of 0s of length N |

## 2.3. General functions

| Function | Syntax | Description |
|---|---|---|
| abs | y = abs(x) | Compute the absolute value(s). |
| ceil | y = ceil(x) | Round up to infinity. |
| conj | y = conj(x) | Compute the complex conjugate. |
| exp | y = exp(x) | Compute the exponential of the argument, i.e. $e^y$. |
| pow2 | y = pow2(x) | Compute the element to the power of 2, i.e $2^y$ |
| pow10 | y = pow10(x) | Compute the element to the power of 10, i.e $10^y$ |
| flip | y = flip(x) | Flip the real and imaginary components of x |
| floor | y = floor(x) | Round down to -∞ |
| imag | y = imag(x) | Get the imaginary component of x |
| ln | y = ln(x) | Natural log |
| log10 | y = log10(x) | Log base 10 |
| log2 | y = log2(x) | Log base 2 |
| logn | y = logn(N,x) | Log of x to base N |
| newpz | y = newpz(mag,freq) | Define a root: $0 \leq mag \leq 5$ and $0 \leq freq \leq \pm fs/2$<br>$roots = \{(z - r_1 e^{-i\theta_1}), (z - r_2 e^{-i\theta_2}), \ldots\}$<br>Where, $\theta_x = \frac{2\pi f}{f_s}$ |
| real | y = real(x) | Get the real component of x |
| round | y = round(x) | Round x |
| sqr | y = sqr(x) | Square of x |
| sqrt | y = sqrt(x) | Square root of x |

## 2.4. Math operators

| Operator | Example syntax | Description |
|---|---|---|
| + | a+b | Addition. |
| - | a-b | Subtraction. |
| * | A*B | Multiplication. |
| / | A/B | Division. |
| .* | A.*B | Element-by-element vector multiplication. |
| ./ | A./B | Element-by-element vector division. |
| .^ | a.^N | Element-by-element vector to the power. |
| ^ | a^N | Vector to the power. |
| ! | N! | Factorial. |

## 2.5. Specialised methods

| Function | Syntax | Description |
|---|---|---|
| savgolay | y = savgolay(n,p) | Design an FIR Savitzky-Golay lowpass smoothing filter of length n and polynomial p |

## 2.6. Variables and data initialisation

| Function | Description |
|---|---|
| General | All variables may contain upper and lower case characters, and numbers. e.g. `Num1`, `myGain`, `alpha15` |
| Interface Variables | The `interface` keyword must be used to define all interface variables. |
| Matrices | A generalised matrix is defined as `A(rows,columns)`. Although *matrix assignment is not supported*, certain vector operations may result in a matrix result, e.g. the vector multiplication: `A=a*transpose(a)`. All data indexes run from `0...N`, you may access a matrix element at row `R` and column `M` as: `y=A(R,M)`. However, you may also access a range of values using the `:` keyword, e.g. `Y=A(3:5,1:2)` which produces a new matrix `Y`. For modifying a value of a matrix/vector, use the `eldef` function, e.g. `a(2,1)=eldef(5)` |
| Vector assignment | By default, a vector is defined as an array with multiple rows and one column. It may contain expressions, variables and constants and must be enclosed in braces `{ }` with comma delimitation.<br><br>Example: `b = {1,0,3.4,0,1};`<br>Example: `A = {1,-2*cos(TwoPi*fc/fs),1};` |
| Vector manipulation | In order to accommodate transposed vectors, all vectors are defined as a generalised matrix, i.e. `A(rows,columns)`. By default, a vector of length `N` is defined as `A(N,1)`, whereas a transposed vector is defined as `A(1,N)`. As all data indexes run from `0...N`, you may access vector element `M` as: `y=A(M,0)`. However, you may also access a range of values using the `:` keyword, e.g. `y=A(3:5,0)`. For modifying a value of a vector, use the `eldef` function.<br><br>**Example**<br><br>`a = {1,0,3.4,0,1};` // assign five elements to vector `a`<br>`a(2,0)=eldef(5);` // set element three to 5<br>`y=a(0,0);` // get element zero and assign it to `y` |
| Data series | A real valued data series can be created with the following syntax:<br><br>`y = series(min,step,max)`<br><br>where, `step` represents the step size between `min` (minimum) and `max` (maximum).<br><br>**Example**<br><br>`a=series(-12,1,1.2);` |
| User comments | All user comments must be preceded with the `//` keyword. Where, the `/* */` syntax is not supported. |

## 2.7. System variables and reserved constants

There are several system variables and constants which can be used in every script and expression.

| Variable | Description |
|---|---|
| `fs` | The `fs` variable specifies the system sampling frequency in its frequency scale, i.e. 50MHz is given as 50, rather than $50e^6$ Hz |
| `Ts` | The `Ts` variable specifies the system sampling period `Ts=1/fs` |
| `pi` | `3.14159265358979` |
| `Twopi` | `6.28318530717959` |
| `i` | Complex number token, $\sqrt{-1}$ |

## 2.8. Mandatory keywords

The following keywords must be present in every script.

| Variable | Description |
|---|---|
| `Main()` | `Main()` is used to seperate the initialisation code from the "main" code - see section 1.1 for more information. |
| `ClearH1` | The `ClearH1` keyword **is not mandatory**, but if included will delete the H1 filter from the cascade. |
| `Den` | `Den` specifies the denominator filter coefficients. This must be a vector. |
| `Num` | `Num` specifies the denominator filter coefficients. This must be a vector. |
| `Gain` | `Gain` specifies the filter gain. This must be real. |

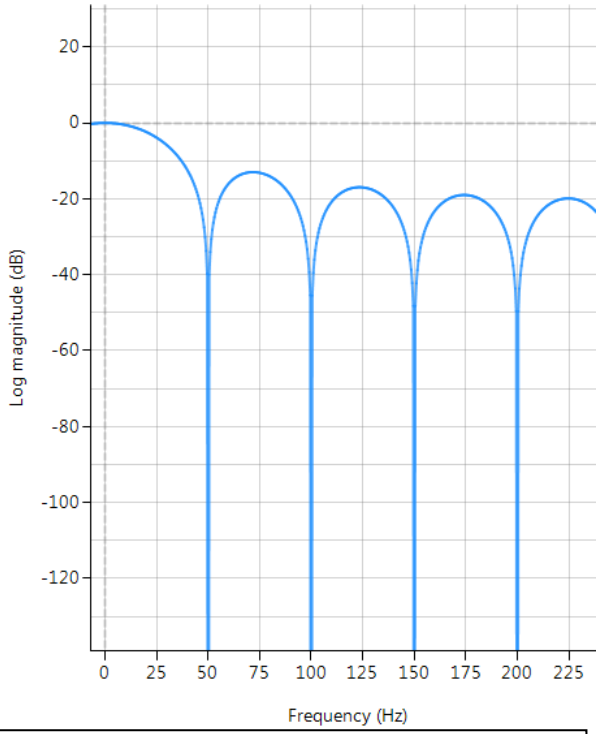
# 3. Example scripts

The following section is a collection of example scripts bundled with the software. All script files are `.afs` files which can be found in the `Scripts\Examples` directory.

## 3.1. Moving average filter (movingaverage.afs)

The moving average (MA) filter is probably one of the most widely used FIR filters due to its conceptual simplicity and ease of implementation. However, despite its simplicity, the moving average filter is optimal for reducing random noise while retaining a sharp step response. Where a simple rule of thumb states that the amount of noise reduction is equal to the *square-root of the number of points in the average*. For example, an MA of length 9 will result in a factor 3 noise reduction.

**Reference:** Understanding Digital Signal Processing, Chapter 5, R. G. Lyons




The following script implements an adjustable length moving average filter. The interface variable `L` is used to set the filter length between `1` and `100`.

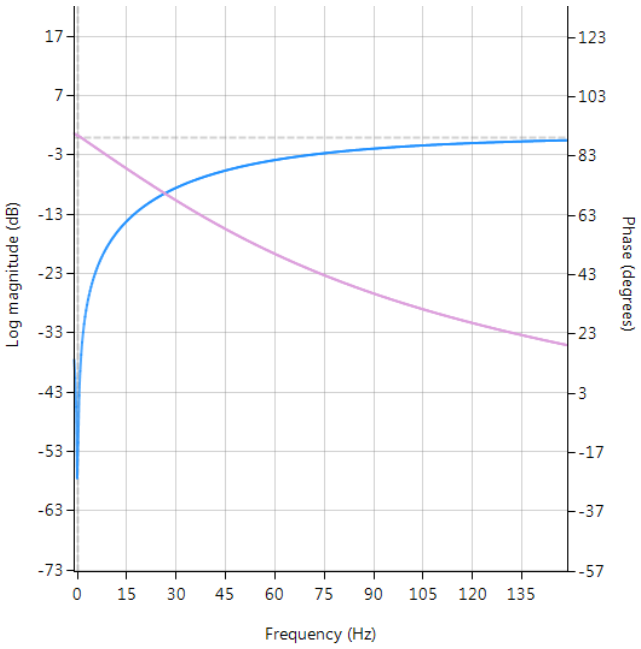
```
ClearH1;  // clear primary filter from cascade

interface L = {1,100,1,10};   // model length (order = length - 1)

Main()
Num = {ones(L)};        // moving average filter coefficients
Den = {1};
Gain = 1/L;
```

## 3.2. HPF (BilinearHPF.afs)

It is sometimes useful to transform an analogue filter into its digital/discrete equivalent. Although there are several transformation methods, the Bilinear z-transform (BZT) is a very popular method and is therefore used for this example. Central to the BZT concept is the S-Z transformation which maps an analogue transfer function, $H(s)$ into its digital equivalent $H(z)$:

$$s = \frac{2}{T}\frac{z-1}{z+1}$$

where, *T* is the discrete system's sampling period. However, substituting $s = e^{j\Omega}$ and $z = e^{jw}$ into the above equation and simplifying, we see that there is actually a non-linear relationship between the analogue, $\Omega$ and discrete, *w* frequencies. This relationship is shown below and is due to the nonlinearity of the arctangent function.




$$w = 2tan^{-1}\left(\frac{\Omega T}{2}\right)$$

Design example

A first order Laplace highpass transfer function is given by:

$$H(s) = \frac{s}{s+w} \quad ; \; \mathrm{w} = \tan\left(\frac{\pi f}{fs}\right)$$

Applying the BZT to $H(s)$, we obtain:

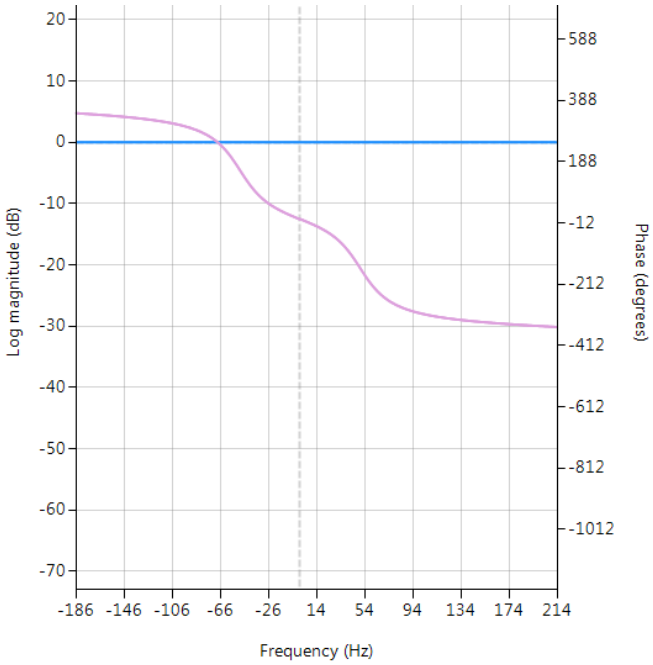
$$H(z) = \frac{1}{(w+1)}\left[\frac{1-z^{-1}}{1+\frac{w-1}{w+1}z^{-1}}\right]$$

The implementation of $H(z)$ is given below, where the cutoff frequency (-3dB point) is adjustable between $0 \leq f \leq fs/2$

```
ClearH1;  // clear primary filter from cascade
interface f = {0,fs/2,1,10}; // interface variable definition

Main()
w=tan(f*pi/fs);

Num = {1,-1}; // define numerator coefficients
Den = {1,(w-1)/(w+1)}; // define denominator coefficients
Gain = 1/(w+1); // define gain
```

## 3.3. Second order all-pass filter (SecondOrderAllPass.afs)

All-pass filters provide a simple way of altering/improving the phase response of an IIR without affecting its magnitude response. As such, they are commonly referred to as phase equalisers and have found particular use in digital audio applications.




A second order all-pass filter is defined as:

$$H(z) = \frac{r^2 - 2r\cos\left(\frac{2\pi fc}{fs}\right)z^{-1} + z^{-2}}{1 - 2r\cos\left(\frac{2\pi fc}{fs}\right)z^{-1} + r^2 z^{-2}}$$

Notice how the numerator and denominator coefficients are arranged as mirror image (mirror-image pair) of one another.

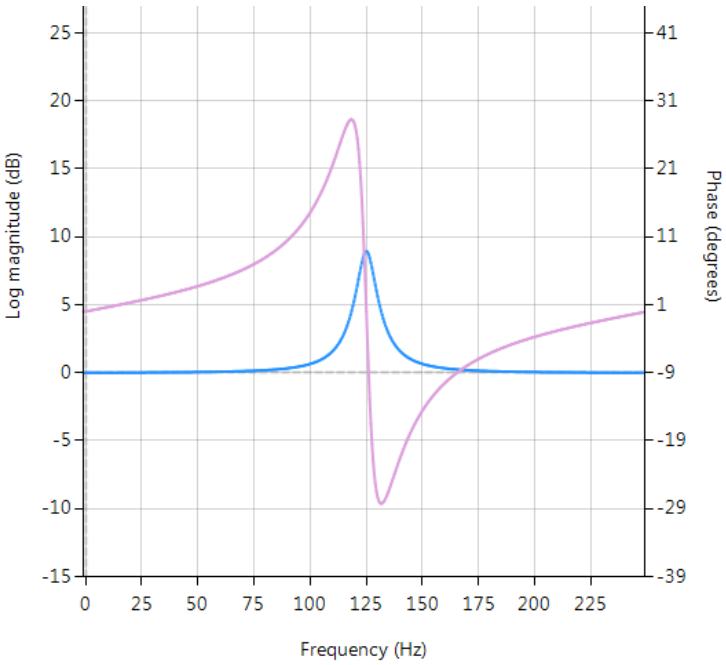
*Reference*: The digital All-pass Filter: A versatile signal processing building block, Regalia, Mitra et al., Proceedings IEEE, vol 76, January 1988.

The following script implements the symbolic transfer function with two interface variables `radius` and `fc`.

```
ClearH1;  // clear primary filter from cascade

interface radius = {0,2,0.01,0.5};    // radius value
interface fc = {0,fs/2,1,fs/10};     // frequency value

Main()
Num = {radius^2,-2*radius*cos(Twopi*fc/fs),1};  // mirror image pair
Den = reverse(Num);
Gain = 1;
```

## 3.4. Allpass Peaking/Bell filter (AllpassPeaking.afs)

A Bell or Peaking filter is a type of audio equalisation filter that boosts or attenuates the magnitude of a specified set of frequencies around a centre frequency in order to perform magnitude equalisation. As seen in the plot on the right-hand side, the filter gets its name from the shape of the its magnitude spectrum (blue line) which resembles a Bell curve.




A Bell filter can be constructed from an all-pass configuration (see section 3.3) by the following transfer function:

$$H(z) = \frac{(1+K) + A(z)(1-K)}{2}$$

where, $A(z)$ is the all-pass filter component:

$$H(z) = \frac{1}{2}\left[(1+K) + \underbrace{\frac{k_2 + k_1(1+k_2)z^{-1} + z^{-2}}{1 + k_1(1+k_2)z^{-1} + k_2 z^{-2}}}_{\text{all-pass filter}}(1-K)\right]$$

```
ClearH1;  // clear primary filter from cascade
interface BW = {0,2,0.1,0.5}; // filter bandwidth
interface fc = {0, fs/2,fs/100,fs/4}; // peak/notch centre frequency
interface K = {0,3,0.1,0.5}; // gain/sign

Main()

k1=-cos(2*pi*fc/fs);
k2=(1-tan(BW/2))/(1+tan(BW/2));

Pz = {1,k1*(1+k2),k2}; // define denominator coefficients
Qz = {k2,k1*(1+k2),1}; // define numerator coefficients
Num = (Pz*(1+K) + Qz*(1-K))/2;
Den = Pz;
Gain = 1;
```

## 3.5. AllpassNotch (AllpassNotch.afs)

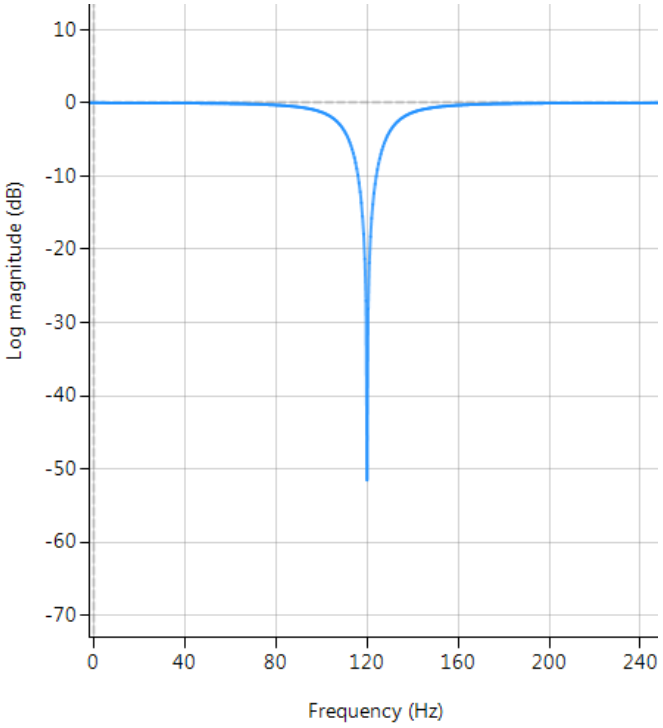
A notch filter can be constructed from an all-pass configuration (see section 3.3) by the following transfer function:

$$H(z) = \frac{1}{2}[1 + A(z)]$$

where, $A(z)$ is the all-pass filter component:

$$H(z) = \frac{1}{2}\left[1 + \underbrace{\frac{k_2 + k_1(1+k_2)z^{-1} + z^{-2}}{1 + k_1(1+k_2)z^{-1} + k_2 z^{-2}}}_{\text{all-pass filter}}\right]$$

$k_1 = -\cos(\frac{2\pi f}{fs})$ controls the centre frequency of the notch, and $k_2 = \frac{1-\tan(BW/2)}{1+\tan(BW/2)}$ controls the bandwidth of the notch.

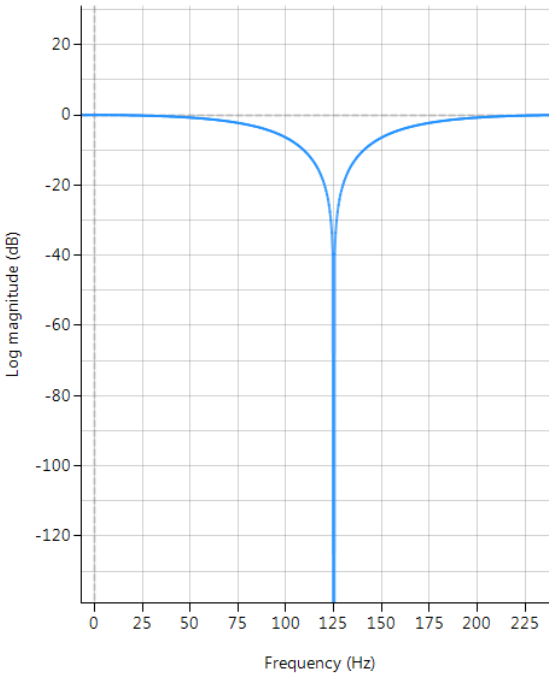



*Reference*: The digital All-pass Filter: A versatile signal processing building block, Regalia, Mitra et al., Proceedings IEEE, vol 76, January 1988.

```
ClearH1;  // clear primary filter from cascade
interface BW = {0,2,0.1,0.5}; // interface variable definition
interface fc = {0, fs/2,fs/100,fs/4};

Main()

k1=-cos(2*pi*fc/fs);
k2=(1-tan(BW/2))/(1+tan(BW/2));

Den = {1,k1*(1+k2),k2}; // define denominator coefficients
Num = {k2,k1*(1+k2),1}; // define numerator coefficients
Num = (Num+Den)/2;
Gain = Num(0,0)/Den(0,0);   // compsensate gain for normalisation
Num=Num/Num(0,0);   // normalise numerator
Den=Den/Den(0,0);   // normalise denominator
```

## 3.6. Notch (Notch.afs)

The primary purpose of a Notch filter is to attenuate (minimize) a specific frequency point in the spectrum, while leaving the rest of the spectrum unaffected. Notch filters are extensively used in audio and sensor signal processing applications in order to minimize the effects of 50/60Hz powerline interference on measured signals.




A notch filter may be defined as:

$$H(z) = \frac{1 - 2\cos w_c z^{-1} + z^{-2}}{1 - 2r\cos w_c z^{-1} + r^2 z^{-2}}$$

where, $w_c = \frac{2\pi f_c}{fs}$ controls the centre frequency, $f_c$ of the notch, and $r$ controls the bandwidth of the notch. The symbolic expressions are implemented as follows:

```
ClearH1;  // clear primary filter from cascade
interface r = {0,1,0.1,0.5};  // radius range
interface fc = {0, fs/2,fs/100,fs/4}; // centre frequency range

Main()

wc=Twopi*fc/fs;

Num = {1,-2*cos(wc),1};      // define numerator coefficients
Den = {1,-2*r*cos(wc),r^2}; // define denominator coefficients
Gain = sum(Den)/sum(Num);    // normalise gain at DC
```

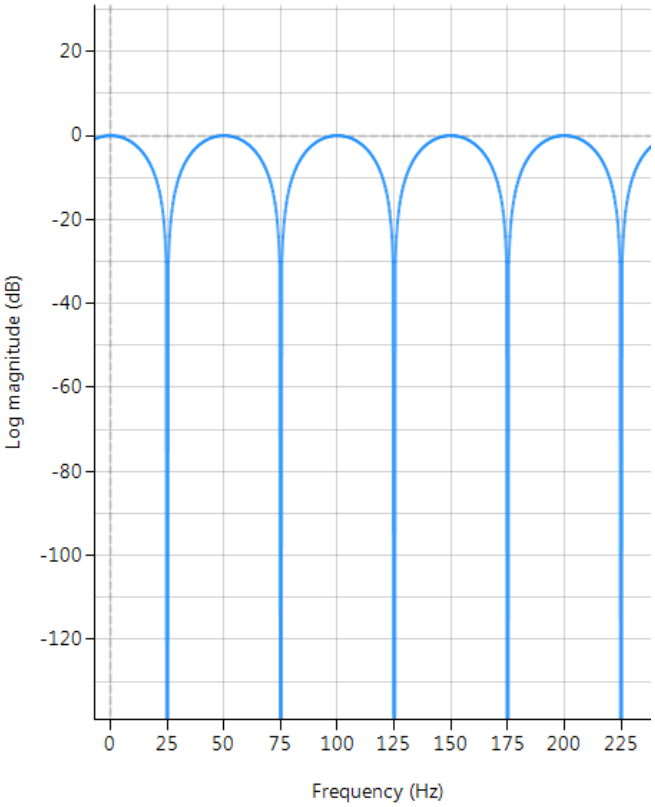
## 3.7. Comb (comb.afs)

The frequency response of a comb filter consists of a series of regularly-spaced troughs, giving the appearance of a comb. Where the spacing of each trough appears at either odd or even harmonics of the desired fundamental frequency. Thus, an FIR comb filter can be described by the following transfer function:

$$H(z) = 1 + \alpha z^{-L}$$

where, $\alpha$ is used to set the Q (bandwidth) of the notch and may be either positive or negative depending on what type of frequency response is required. In order to elaborate on this, negative values of $\alpha$ have their first trough at DC and their second trough at the fundamental frequency. Clearly this type of comb filter can be used to remove any DC components from a measured waveform if so required. All subsequent troughs appear at even harmonics up to and including the Nyquist frequency.

Positive values of $\alpha$ on the other hand, only have troughs at the fundamental and odd harmonic frequencies, and as such cannot be used to remove any DC components.




```
ClearH1;  // clear primary filter from cascade
interface L = {4,20,1,5}; // filter length
interface alpha = {-1,1,0.01,0.99};

Main()
Num = {1,zeros(L-1),alpha}; // numerator coefficients
Den = {1};
Gain = 1/sum(abs(Num));
```

## 3.8. Fractional Farrow Delay

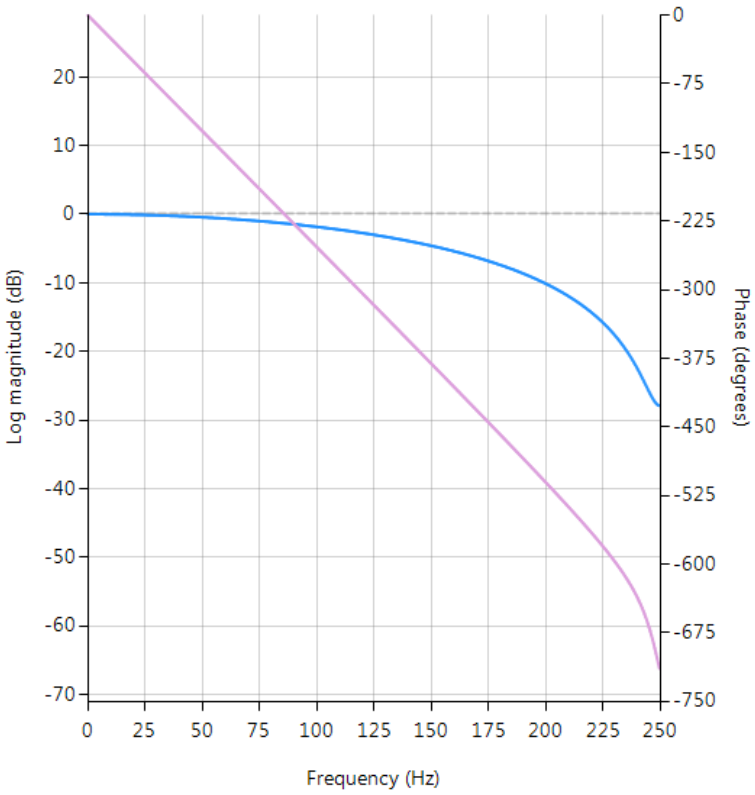
In signal processing, the need sometimes arises to nudge or fine-tune the sampling instants of a signal by a fraction of a sample. An FIR Farrow delay filter is typically employed to achieve this task, and may be combined with a traditional integer delay line in order to achieve a universal fractional length delay line.

A Fractional delay based on an FIR Farrow structure may be defined as:

$$H(z) = (1-\alpha) + \alpha z^{-1}; \quad 0 \leq \alpha \leq 1$$

Which produces a fractional linear delay of $\alpha$ between 0 and 1 samples. However, a more universal building block can be achieved by combining the Farrow delay structure with an integer delay, Δ

$$H(z) = (1-\alpha)z^{-\Delta} + \alpha z^{-(\Delta+1)}$$




The plot shown on the right shows the magnitude (blue) and phase (purple) spectra for $\Delta = 9, \alpha = 0.52$. As seen, the fractional delay element results in a non-flat magnitude spectrum at higher frequencies.

```
ClearH1;  // clear primary filter from cascade

interface alpha = {0,1,0.02,.5}; // fractional delay
interface D = {1,30,1,10};       // integer delay

Main()
Num = {zeros(D),1-alpha,alpha}; // numerator coefficients
Den = {1};                      // denominator coefficient
Gain = 1/sum(Num);              // normalise gain at DC
```

# Document Revision Status

| Rev. | Description | Date |
|---|---|---|
| 1 | Document released. | 26/06/2015 |
| 2 | Updated examples. | 12/10/2015 |
| 3 | Updated examples and text. | 03/12/2015 |
| 4 | Updated examples and text. | 25/01/2016 |
| 5 | Added section 3.8. | 18/02/2016 |
| 6 | Updated text. | 06/06/2016 |
| 7 | Updated references. | 30/06/2016 |
| 8 | Updated text and added section 2.5. | 07/03/2017 |