Signal Processing Systems (521279S)
Part 2 : Implementing fixed-point filtering

Version 1.1

November 3, 2025

Study objectives. In this exercise, we consider the tool-based development of fixed-point DSP solutions. First, we take a look at a particular tool, Matlab's Fixed Point Toolbox, that will be used in the work. Then, a fixed-point FIR filter is designed to meet given specifications. The filter is simulated, which demonstrates some interesting phenomena. Simulation is also used to optimize the word lengths of the data path.

Contents

History
V1.0 11.10.2019 .
V1.1 09.11.2022 Minor additions to description in Sec. 3.

1 Matlab Fixed Point Toolbox

In the exercise, we will simulate fixed-point computations using Matlab. In the following, we present the basics of the associated fixed-point toolbox.

1.1 "fi" command

Fixed-point scaling methods were studied in the first design project. In the following, we take a look at the implementation of these methods in the Fixed Point Toolbox.

1.1.1 Binary point scaling

Fixed-point number objects can be constructed with the 'fi' command. The basic expression to use it has the form

```
fi(value,signedness,word_length,fraction_length)
```

which can be used to map values to *any* unsigned or signed *binary-point scaled* fixed point format (u$p.n$, s$p.n$). Note that the fraction length n can be any positive or negative integer.

Example 1. For example, the command

```
a = fi(0.42,1,8,4)
```

approximates $v = 0.42$ with s8.4 and the approximation is $\tilde{v} = 0.4375$ since the rounding mode is round-to-nearest by default. The fixed-point number object is stored to the workspace variable `a`. As another example,

```
b = fi(1.5,1,8,7)
```

would return a value in the s8.7 format. As this is a fractional format, 1.5 is out of the range. Because the operation saturates by default, the approximation is $\tilde{v} = 1 - 2^{-7} = 0.9921875$, the largest representable positive value. ■

1.1.2 Slope-bias scaling and fractional slope

To deal with slope-bias scaling, you may use the command:

```
fi(value,signedness,word_length,slope,bias)
```

Example 2. The command

```
c = fi(0.42,1,8,0.0625,0)
```

approximates $v = 0.42$ in s8.4 format as the bias is zero, and $0.0625 = 2^{-4}$ is the slope of the s8.4 format. ■

However, it is possible to do the following. Say that you want to use slope s for your coefficients. First, you represent the slope as a product $s = f \times 2^{-n}$, where $f \in [1,2)$ is the *fractional slope*. Then, you divide the coefficients first by f, and use the binary point scaled format sp.n for the result. The number of fraction bits and the fractional slope are determined by equations

$$n = -\lfloor \log_2(s) \rfloor, \quad (1)$$
$$f = s/2^n, \quad (2)$$

where $\lfloor \cdot \rfloor$ denotes the floor operation. The reason for introducing the fractional slope here is that it may be more convenient to use just the binary point scaling with the Fixed Point Toolbox.

Example 3. The *bin* and *storedInteger* commands provide the bit strings and integers corresponding to the fixed-point object. For example, the commands

```
slope = 0.04;
d1 = fi(0.42,1,8,slope,0);
bin(d1)
storedInteger(d1)

n = -floor(log2(slope));
f = slope / 2^-n;
d2 = fi(0.42 / f,1,8,n);
bin(d2)
storedInteger(d2)
```

provide alternative ways for finding the bit strings and integers, that correspond to the slope $s = 0.04$. ■

1.2 "fimath" command

In Matlab, we may add and multiply 'fi' objects in ordinary way if they match each other. The behaviour of overloaded '+' and '*' operators is specified using 'fimath' objects.

The 'fimath' object has five mode parameters:

- **SumMode** and associated fields define how the data type for the results of addition is determined. There are four options:
  1. 'FullPrecision' lets Matlab decide both the word length and fraction length. It is quaranteed that there will be no overflows and loss of precision;
  2. 'SpecifyPrecision' lets the user decide the sum data type, both the word and fraction lengths used;
  3. 'KeepLSB': user specifies the word length. The fraction length is chosen by Matlab so that the precision is maximized, but there is a danger of overflow;
  4. 'KeepMSB': user specifies the word length. Overflows are completely avoided, but there can be loss of precision.
- **ProductMode** and associated fields define how the product data type is determined. There are the same four options as with the SumMode.
- **RoundingMethod** (or **RoundMode** in older toolbox versions) defines the rounding mode. There are six options corresponding to the methods mentioned in the introduction of the previous exercise, 'floor', 'ceil', 'round', 'nearest', 'convergent' and 'fix'.
- **OverflowMode** defines how overflow events are handled. There are two options, 'saturate' and 'wrap'.
- **CastBeforeSum**: if this is true (=1) then both operands of addition are casted to the sum data type (applying selected overflow and rounding modes if necessary) and addition is done after that. Otherwise, the addition is carried first out with full precision, and the result is casted.

For other parameters, see Matlab documentation (help fi, doc fi).

In practice, you work by setting the 'fimath' property of the 'fi' object. The rule is that 'fimath' settings associated with 'fi' objects must be equal in order to perform additions and multiplications with them. Then, when we add or multiply two objects, the result will take the format specified in 'fimath'. Overflow and round modes of 'fimath' are applied to the result if necessary.

The result of an operation contains also the same 'fimath' settings so any consecutive additions and multiplications with the result will use the same rules.

Example 4. The commands

```
A = fi(1.42,1,8,4);
F = fimath('ProductMode','SpecifyPrecision',...
'ProductWordLength',8,'ProductFractionLength',3);
A.fimath = F;
B = A * A
```

generate two fixed point numbers, `A` approximates 1.42 is in the format s8.4 and the product `B` is in the format s8.3. ■

1.3 "fipref" command

The 'fipref' object is used to control the behaviour of the 'fi' objects.

The object has following attributes:

- **FimathDisplay**, **NumericTypeDisplay**. **NumberDisplay**: attributes used to control how 'fi' objects are displayed.
- **DataTypeOverride**: The default value of this attribute is 'ForceOff'. If you want that numbers are represented as floating-point numbers, you may set this value to 'ScaledDoubles', 'TrueDoubles' or 'TrueSingles'. The problems in a fixed-point system can be associated with word/fraction lengths. If the system does not work in the 'ForceOff' mode you may quickly check with other options that the system works with floating-point numbers.
- **LoggingMode**: When logging is 'On' information about variable ranges is collected which can be useful in the choice of the fixed-point format.

The Simulink tool also offers similar data type override and data logging capabilities.

1.4 Working with the toolbox

In Matlab, the *fi* command is used to construct 'fi' objects:

```
>> h = exp(1);
>> h_fixed = fi(h,1,8,3)
h_fixed =
    2.7500
          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 3
```

The *bin* command shows the object bit string:

```
>> bin(h_fixed)
ans =
    00010110
```

The *get* command shows information about the object's fixed-point properties, and behaviour of arithmetic commands:

```
>> get(h_fixed)
                    DataTypeMode: 'Fixed-point: binary point scaling'
                        DataType: 'Fixed'
                         Scaling: 'BinaryPoint'
                          Signed: 1
                      Signedness: 'Signed'
                      WordLength: 8
                  FractionLength: 3
                   FixedExponent: -3
                           Slope: 0.1250
          SlopeAdjustmentFactor: 1
                            Bias: 0
                       RoundMode: 'nearest'
                  RoundingMethod: 'Nearest'
                    OverflowMode: 'saturate'
                  OverflowAction: 'Saturate'
                     ProductMode: 'FullPrecision'
               ProductWordLength: 32
            MaxProductWordLength: 65535
           ProductFractionLength: 30
            ProductFixedExponent: -30
                    ProductSlope: 9.3132e-10
    ProductSlopeAdjustmentFactor: 1
                     ProductBias: 0
                         SumMode: 'FullPrecision'
                   SumWordLength: 32
                MaxSumWordLength: 65535
               SumFractionLength: 30
                SumFixedExponent: -30
                        SumSlope: 9.3132e-10
        SumSlopeAdjustmentFactor: 1
                         SumBias: 0
                  CastBeforeSum: 1
                            Tag: ''
```

Addition and multiplication of objects can be done using the corresponding operators:

```
>> h_fixed + h_fixed
ans =
    5.5000
          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 9
        FractionLength: 3

>> h_fixed * h_fixed
ans =
    7.5625
          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 6
```

As *SumMode* is 'FullPrecision' the word length adds one bit to the word length. Also, as *ProductMode* is 'FullPrecision' the output word length is two times input word length (conservation of bits). This default behaviour can be changed easily. For example, if we want to use the format s15.6 for all multiplication results we can change the settings as

```
>> h_fixed.ProductMode = 'SpecifyPrecision';
>> h_fixed.ProductWordLength = 15;
>> h_fixed.ProductFractionLength = 6;
>> h_fixed * h_fixed
ans =
    7.5625
          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 15
        FractionLength: 6

        RoundingMethod: Nearest
        OverflowAction: Saturate
           ProductMode: SpecifyPrecision
     ProductWordLength: 15
 ProductFractionLength: 6
               SumMode: FullPrecision
```

The default overflow mode is saturate which means that if an overflow occurs in an operation the output is the representable value closest to the true value. For example,

```
>> h_fixed.ProductMode = 'SpecifyPrecision';
>> h_fixed.ProductWordLength = 9;
>> h_fixed.ProductFractionLength = 6;
>> h_fixed * h_fixed
ans =
    3.9844
```

Note: in one exercise task, two different numbers are multiplied. Change the 'ProductMode', 'ProductWordLength', and 'ProductFractionLength' of **both** fixed-point objects before multiplication.

2 Conversion from floating-point to fixed-point

2.1 Error sources

The error sources in digital signal processing can be classified to four groups (Ifeachor & Jervis 2002, p. 818):

- coefficient quantization errors,
- analog-to-digital conversion (ADC) quantization noise,
- roundoff errors associated with internal arithmetic operations using finite word lengths and production of the output, and
- overflow errors.

These items are illustrated in Fig. 1. When you design a DSP system, you must be aware of and counteract these error sources in order to quarantee that your system works as expected.

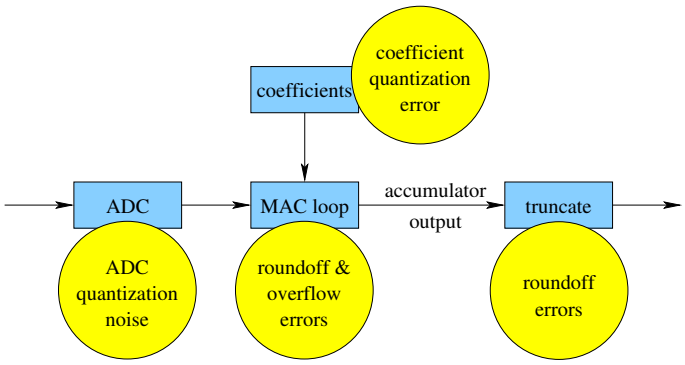


Figure 1: Error sources in DSP and the case of fixed-point FIR filtering.

Loss of precision can occur when parameters such as filter coefficients must be quantized (e.g. type conversion from floating point to fixed point during design). Approximation of filter coefficients changes the transfer function $H(z)$ of the filter, and we get another filter with a transfer function $H_q(z)$. As a result, the filter might not fulfill the original specifications, and it may even become unstable.[1]

Also quantization of signal values (rounding) lowers the system performance. To keep it sufficient for an application one must consider the system structure

[1]Recall that coefficient quantization error was mentioned in Introduction to Part 1 (**intro1.pdf**), Section 3.4.1., where FIR filtering was discussed. The idea was that ratios of filter coefficient values do not match the original ratios.

(interconnection of adder, multiplier and delay elements) in addition to the data types and word/fraction lengths used. *Roundoff* noises affect the filter output in the form of random disturbances which can be analyzed by suitable noise modelling and use of linear system theory; see (Ifeachor & Jervis 2002, p. 419, pp. 836-859).

Quantization is also a *nonlinear* operation, which may lead to effects such as limit cycles, jump phenomenon and subharmonic response in an IIR structure[2]. Increasing word length and error spectral shaping may alleviate these phenomena; see (Ifeachor & Jervis 2002, p. 846-860).

In filtering, one is typically interested in the power of the quantization noise; this is a measure of average error. However, in the design of fixed-point algorithms it may be necessary to consider *error bounds* by analyzing the propagation of the errors in computation and thereby determining how large the error can be in the worst case; see (Kreyszig 1993, pp. 922-924).

Finally, one must address the problem of overflows in design. Hardware mechanisms for managing the problem were mentioned earlier, namely the *guard bits* and *saturation arithmetic*. In the filter design, one may also add scaling operations to alleviate the problem (Ifeachor & Jervis 2002, pp. 828-835).

2.2 Simulation-based development

Design of the fixed-point solution can be based on analysis of the effects of error sources as outlined above. In addition, solution can be based on a tool-based design process where simulation of the system is used to study behaviour of the system in computations. This can be based on writing C/C++ code. Alternatively, Matlab and its Fixed Point Toolbox can be used for implementing the simulation. Also the Simulink environment of Matlab offers support for fixed-point simulations.

[2]Recall that, opposite to FIR filters, an infinite impulse response (IIR) filter has feedback connections from the output.

3 Filter design

We start by considering the coefficient quantization problem.

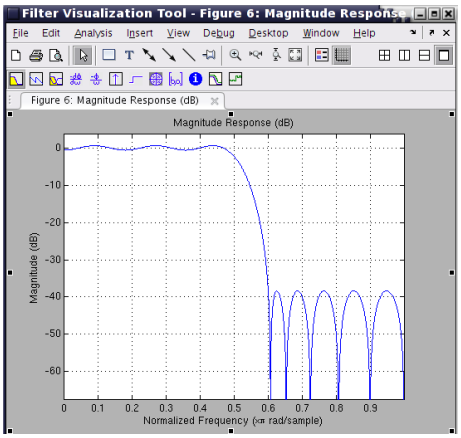
3.1 Design of a floating-point version

As a first step we use Filter Design & Analysis Tool to find out the filter coefficients. The Matlab command **fdatool** opens the main dialog of this tool. There you can make your specification. Various displays of the design are available (magnitude response, phase response, impulse response etc).

In the **File** menu, you can save the session with the command **Save Session**. You can generate an M-file which returns a DFILT object that contains the coefficient values. Use the command **Generate MATLAB Code**>**Filter Design Function** (in some versions of Matlab, it is **Generate M-file**).

Example 5. An equiripple FIR filter was designed using fdatool. Unzip accompanying package **dtask2.zip**. The session is stored to **x_session.fda** which you can open in fdatool. The file **x_filter.m** contains the generated M-file. You can view the filter characteristics with the Filter Visualization Tool (**fvtool**).

```
>> Hd = x_filter

Hd =
     FilterStructure: 'Direct-Form FIR'
           Numerator: [1x24 double]
    PersistentMemory: false

>> fvtool(Hd)
```



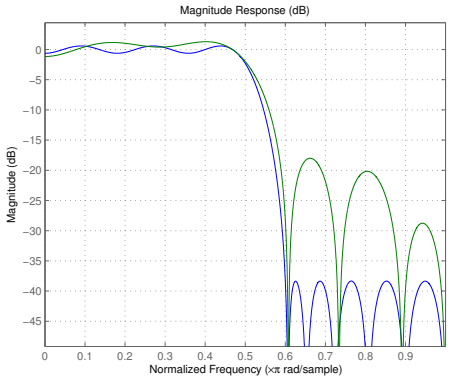
Note: Button for the **Zoom-in** tool is below the menus (magnifying class with plus sign). It is useful for inspecting the passband behavior. ■

3.2 Coefficient quantization

To quantize coefficients we can use functions like **round**, **fix** and **fi**. The number of integer bits depends on the largest coefficient magnitude as the following example shows. Note that, for simplicity, the *binary point scaling* is used here and also in the design task. Slope-bias scaling could give better results, but would require more work.

Example 5 (continued). Let us quantize the filter coefficients to a fixed-point representation, which uses 4 fraction bits, and compare the resulting filter to the original one. This can be done with the commands

```
>> Hdq = dfilt.dffir;
>> fraction_bits = 4;
>> mplier = 2^fraction_bits;
>> Hdq.Numerator = round(Hd.Numerator * mplier) / mplier

Hdq =
     FilterStructure: 'Direct-Form FIR'
           Numerator: [0 0 0 0 0 -0.0625 0 0.0625 -0.0625
                       -0.125 0.125 0.5 0.5 0.125 -0.125
                       -0.0625 0.0625 0 -0.0625 0 0 0 0 0]
    PersistentMemory: false

>> fvtool(Hd,Hdq);
```



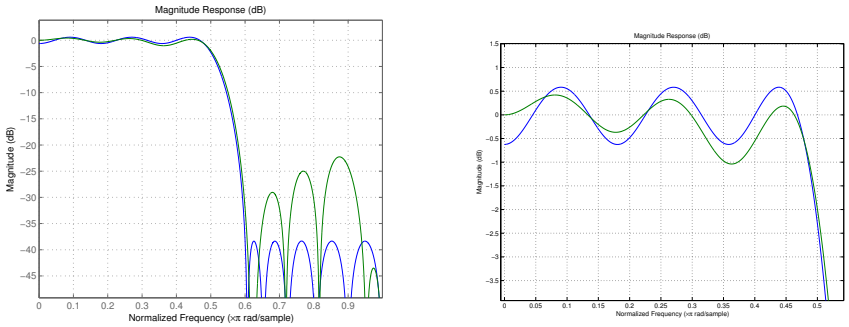
Here, the blue color corresponds to the unquantized filter, and the green color to the quantized one.

We can also use the Fixed Point Toolbox. No integer bits are needed as the largest absolute coefficient value is $0.4701 < 1$. Therefore, the word length equal to 5 gives the same result as above:

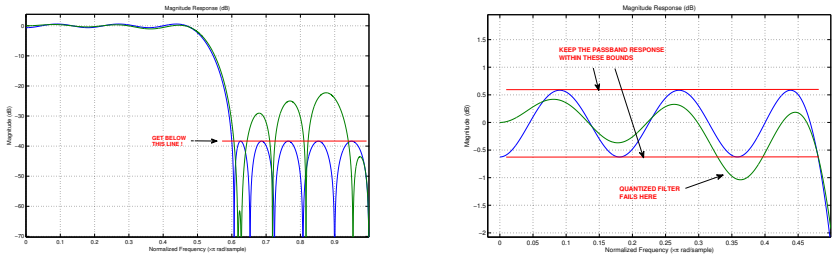
```
>> max(Hd.Numerator)

ans =
    0.4701

>> Hdq = dfilt.dffir;
>> Hdq.Numerator = fi(Hd.Numerator,1,5,4); % s5.4
>> fvtool(Hd,Hdq);
```

Note that $0.4701 < 0.5$, and 5 fraction bits give different approximation to it. With the same word length, we can get improvement in the magnitude response:

```
>> Hdq.Numerator = fi(Hd.Numerator,1,5,5); % s5.5
>> fvtool(Hd,Hdq);
```



However, especially the stopband behavior is still far away from the original response:



Increasing the word length helps. For example, try

```
>> Hdq.Numerator = fi(Hd.Numerator,1,10,10); % s10.10
>> fvtool(Hd,Hdq);
```

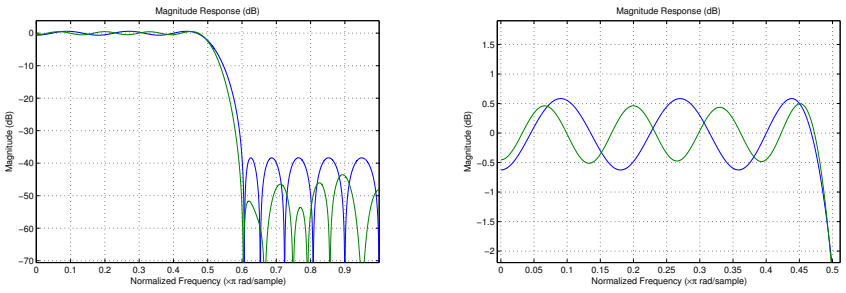
However, it is still not as required. So, one should increase the word length more to improve the behaviour. ■

3.3 Tightening of specification

Another approach to the fixed-point filter design problem is that you start from a *tighter* specification (i.e. compared to the specification, the bandpass ripple is set to a lower value and stopband attenuation is increased) and design an unquantized filter for that. As a result you get a longer filter, but the fixed-point coefficient word length requirement is reduced.

Example 5 (continued). In the original specification, the passband ripple is 1 dB, and the stopband attenuation 40 dB (open **x_session.fda** in fdatool and look at the magnitude specification). Another session is stored to **x_session2.fda**, where the ripple is set to 0.75 dB (i.e. reduced 25%), and the stopband attenuation to 60 dB (i.e. increased 20 dB). The M-file generated from that specification is in **x_filter_2.m**.

```
>> Hd_2 = x_filter_2

Hd_2 =

     FilterStructure: 'Direct-Form FIR'
          Arithmetic: 'double'
           Numerator: [1x35 double]
    PersistentMemory: false

>> max(Hd_2.Numerator)

ans =
    0.5232

>> Hdq_2 = dfilt.dffir;
>> Hdq_2.Numerator = fi(Hd_2.Numerator,1,10,9);
>> fvtool(Hd,Hdq_2)
```



Note that the maximum value 0.5232 implies, that we must now use a format s$p.(p-1)$. By trial, it was found that for $p \geq 10$ the filter response meets the specification. ■

4 Roundoff error study

4.1 Motivation

Roundoff errors require careful study in many cases. For example, with periodic inputs quantization errors may correlate, that is, we can see periodicities in the differences between the results computed using fixed-point and more accurate floating-point modes. For example, signals in telecommunication often involve periodic signals and correlated noises can be expected.

Simple linear models, which are often used to obtain analytic results, make assumptions about uncorrelated noises. Therefore, the results of analysis may not tell the truth and simulations are necessary to study the actual behavior of the system. If the filter does not work as required, some remedy must be found.

4.2 FIR/MAC simulator

An M-file **FIR_MAC_Simulator.m** was written to experiment with fixed-point FIR filtering. The function can be used for simulating A/D conversion (ADC) followed by transversal FIR filtering implemented on a MAC unit. In order to study errors, it provides access to the signals at four datapath points, which are shown in Fig. 2.

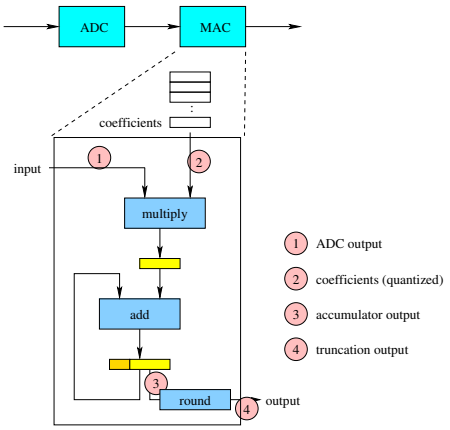


Figure 2: Overview of the simulated system.

Implementation utilizes fi, fimath, fipref and quantizer objects. The function has the syntax

```
h = FIR_MAC_Simulator(qxp,qap,qpp,gbits,qop)
```

where `qxp` specifies the ADC parameters, `qap` specifies coefficient quantization, `qpp` specifies the pipeline register for multiplication results, `gbits` is the number of guard

bits in the accumulator, and `gop` specifies the output format. The output `h` is a structure that contains nested function handles.

After simulator construction, a simulation is run by calling

```
sigall = h.simulate(mode,x,a)
```

where `mode` selects the execution mode (0 = full fixed-point simulation, 1-9 switch off quantizations in different ways), and `x` and `a` are the input data (analog) and filter coefficients. The round-to-nearest and saturate-on-overflow modes are used in all parts of the MAC unit. The output `sigall` is a structure that contains the output values and the intermediate values shown in Fig. 2, and input/output differences related to the ADC and rounding operations.

We can also compare the current simulation result with a reference result by calling

```
diff = h.computeDiff(ref_sigall)
```

where `ref_sigall` is the reference output produced by another simulation run.

4.3 Simulator in the design task

You may take a look at the implementation of **FIR_MAC_Simulator** to learn about fixed-point programming in Matlab. However, in the exercise you do not have to call it directly. Another M-file, **FIR_Experiment.m**, was written which

1. generates test signals,
2. calls **FIR_MAC_Simulator** twice to compute both fixed-point and reference (double precision floating-point) filtering results,
3. visualizes the results in the time domain, and
4. returns the filtering results.

The syntax of the function is

```
[y_fixpt,y_float,y_error] = FIR_Experiment(signal,a,bitspec)
```

where

- `signal` selects the test signal,
- `a` contains the unquantized filter coefficients,
- `bitspec` contains parameters, that control the word and fraction lengths used in the simulation,
- `y_fixpt` contains the fixed-point filtering result,
- `y_float` contains the floating-point result, and

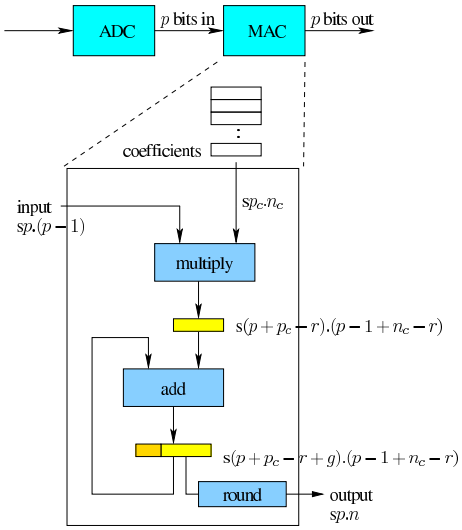


Figure 3: Word and fraction length specifications in the simulation.

- `y_error` is their difference.

The argument `bitspec` is a vector of six parameters, $[p, p_c, n_c, r, g, n]$. Their explanation is as follows:

- ADC input signals have the range -1 .. +1.
- ADC output is encoded in the fractional p-bit format, s$p.(p-1)$.
- filter coefficients are encoded in the s$p_c.n_c$ format.
- According to the law of conservation of bits, the multiplier output has the format s$(p+p_c).(p-1+n_c)$. Then, there is no loss in precision. In the simulation, the precision can be weakened by r bits, which gives the output format s$(p+p_c-r).(p-1+n_c-r)$.
- The accumulator has the format s$(p+p_c-r+g).(p-1+n_c-r)$, where g is the number of guard bits.
- Finally, the contents of the accumulator are mapped to the output format s$p.n$. So, the MAC output has the same word length as the input, and the number of fraction bits is n.

The specification of fixed-point numbers is illustrated in Fig. 3.

Example 6. Let us simulate a fixed-point implementation of the filter, whose coefficients are provided by **x_filter.m**. The input/output word length is $p = 8$ bits, coefficients are

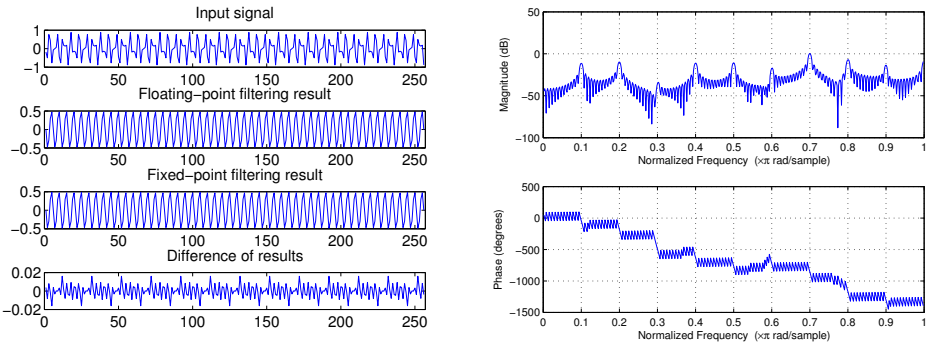
encoded with the same word length ($p_c = 8$) using $n_c = 7$ fraction bits, full precision is used for the product pipeline register ($r = 0$), and the accumulator contains 7 guard bits ($g = 7$). We infer the number of fraction bits for the output:

- The L_1 norm of the filter coefficients, that is, the sum of their absolute values is 1.8531.[3]
- As the filter input values are in the range $[-1, 1)$, the output values must be within the range $1.8531 \times [-1, 1)]$. We need one integer bit, in addition to the sign bit, to represent the values in that range. Therefore, $n = p - 2 = 6$.

The `bitspec` used is $[8, 8, 7, 0, 7, 6]$. The simulation for the 'intro' signal is run with the following commands:

```
>> Hd = x_filter; a = Hd.Numerator;
>> bitspec = [8, 8, 7, 0, 7, 6];
>> [y_fixpt,y_float,y_error] = FIR_Experiment('intro',a,bitspec);

I/O word length: 8
MAC input format: s8.7
Coefficient format: s8.7
Product pipeline register format: s16.15
Accumulator register format: s23.15
MAC output format: s8.6

>> freqz(y_error); % Spectrum of the error signal
>> errpow_dB = 10 * log10(mean(y_error.^2)); % Error power
```

Signal plots in time and frequency domains show that the result is noisy:



■

[3]Calculated with the Matlab's **norm** function. Be careful when you use it as it can compute also other kinds of norms.

References

Ifeachor, E. C. & Jervis, B. W. (2002) Digital Signal Processing: a Practical Approach. 2nd edition. Pearson Education Ltd., Harlow, England.

Kreyszig, E. (1993) Advanced Engineering Mathematics. 7th Edition. John Wiley & Sons, Inc., New York.