Changing Signal Scale or Sampling Rate "Gently" By Fractional Delay Filtering

Çağatay Candan, *Senior Member, IEEE*
Department of Electrical and Electronics Engineering,
METU, Ankara, Turkey.
ccandan@metu.edu.tr

***Abstract*—An efficient method for signal scaling and sampling rate conversion is presented. The method is particularly suitable for the sampling rate conversion ratios, or equivalently signal scaling ratios, close to unity, such as 24/25. Within the scope of the present work, the rate changes around unity is called as "gentle" changes. The implementation of gentle rate changes through conventional techniques (cascade application of upsampling and downsampling) is not feasible in many applications. An efficiently implementable solution, typically requiring less than 5 multiplications per output sample, is described as a remedy for such applications.**

***Index Terms*—Signal Scaling, Sampling Rate Change, Fractional Delay**

I. Introduction

A time-varying filtering scheme for signal scaling, which is also applicable to the sampling rate conversion, is presented. The main goal is to produce the samples of the scaled signal, $x_\beta[n] = x(\beta t)|_{t=nT}$, by digitally processing the samples of $x(t)$, that is $x[n] = x(t)|_{t=nT}$. The problem of producing $x_\beta[n]$ from $x[n]$ can also be interpreted as the change of sampling rate from T to βT. Our main interest is the case of $\beta \approx 1$, such as $\beta = 24/25$, which is called as a "gentle" change in the context of the present study. If we follow the conventional techniques, such gentle changes require quite demanding implementations. For example, the rate change of $24/25$ requires upsampling by 25 units followed by downsampling by 24 units. Our goal is to describe a computationally attractive scheme to realize gentle rate changes.

The signal scaling problem with $\beta \approx 1$ may arise due to the deviation of the clocks running the analog-to-digital-converters (A/D) or digital-to-analog-converters (D/A). In the processing chain, the master clock of the converters may inadvertently "run faster" than the nominal speed. This may lead to a shorter than desired time period between samples. Specifically, if the oscillator has a frequency drift of 1000 ppm (parts per million), the actual sampling period can lie in the interval of $\left[\frac{999}{1000}T, \frac{1001}{1000}T\right]$ where T is the nominal sampling period. The textbook solution to correct the sampling rate deviation, which is the cascade application of upsampling and downsampling, is not a feasible approach for such conversion ratios.

Another application area requiring gentle corrections in the sampling rate is the compensation of the wideband Doppler effect in sonar systems. As noted in [1, p.52], the time dilation/compression on the order of 1 ± 0.001 can deteriorate the processing of the received echo. A proper signal design for the reduction of this undesired effect is suggested in [1]. Here, we suggest an alternative approach that scales the received sonar echo so that the dilation/compression due to the Doppler effect can be efficiently compensated.

The principle of the suggested method has been first given in the context of audio format conversion from CD (sampled at 44.1 kHz) to DAT (sampled at 48 kHz), [2]. Some earlier studies also study the application of fractional delay filters for the sampling rate conversion, [3], [4]. Most recently, Blok has described several fractional delay filtering schemes for the solution of the same problem, [5]. The main contribution of the present work can be stated as the recognition of the importance of fractional delay based systems for a gentle change in the signal scale and the description of a highly efficient discrete Newton series based implementation for the realization of fractional delay filters, [6].

II. Preliminaries

The signal $x(t)$ and its scaled version $x_\beta(t) = x(\beta t)$ are assumed to be sampled with the sampling period T, i.e. $x[n] = x(t)|_{t=nT}$ and $x_\beta[n] = x(\beta t)|_{t=nT}$. Note that the signal $x_\beta[n]$ can also be considered as the samples of $x(t)$ with the sampling period βT, that is $x_\beta[n] = x(\beta t)|_{t=nT} = x(t)|_{t=n(\beta T)}$. Hence, the process of generating $x_\beta[n]$ from $x[n]$ can be interpreted as either the change of sampling rate from T to βT or the change of signal scale from 1 to β.

For some applications mentioned in the introduction, the desired sampling rate change (or the scale factor) can be close to unity, $\beta \approx 1$. The presented scheme is specifically developed for such applications. The working principle of the scheme can be described by writing $\beta = 1 - \alpha$, with $|\alpha| \ll 1$, and then re-expressing the scaling relation as follows:

$$x_\beta(t) = x(\beta t) = x(t - \alpha t). \tag{1}$$

It is possible to interpret αt in (1) as the amount of delay (or advance, if $\alpha < 0$) of signal $x(t)$ at time t. Here, delay is an increasing function of time and if it can be properly realized for all t, the scaled signal can be generated. In terms of sampled data, the delay of αt seconds corresponds to $\alpha t/T$ samples of delay. The delay value is typically not integer valued and fractional delay filters is required for its realization.

Figure 1 illustrates the approach for $\alpha = 1/3$. The presented value for α corresponds to the scale change of $\beta = 1 - \alpha = 2/3$ or the sampling rate change of $1/\beta = 3/2$. From Figure 1, it can be noted that the n'th output sample ($y[n]$) has a time delay of $n \times 1/3$ samples from the n'th input sample ($x[n]$).

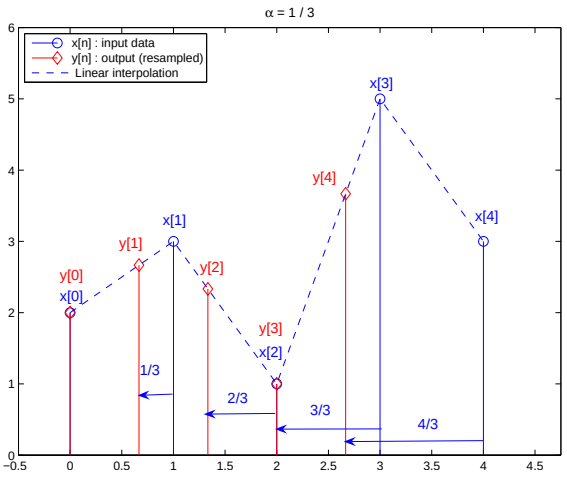

Fig. 1. Resampling for the sampling rate change of $1/\beta = 1/(1-\alpha)$ or the signal scaling of $\beta = 1-\alpha$ for $\alpha = 1/3$.

The dotted line in Figure 1 is the result of linear interpolation. By evaluating, the linear interpolation result at an increasing delay of $D[n] = \alpha n$ samples from the n'th input sample, the scaled output can be generated. Even though, this interpretation (output being the fractionally delayed version of input) is valid; we prefer to interpret the output as the resampled version of input. Our goal is to generalize the resampling operation with linear interpolator as illustrated in Figure 1 to arbitrary order polynomial interpolation and present an efficient mechanism for its realization. In this section, a fairly detailed description of the linear interpolation case is provided to assist the discussion of the general case.

It can be noted from Figure 1 that each output sample is generated by processing two closest samples around the output sample. For example, $y[1]$ is generated from $x[0]$ and $x[1]$; upon the arrival of $x[2]$, the next output sample $y[2]$ is generated from $x[1]$ and $x[2]$ and so on. The illustrated system is to increase the sampling rate change by $3/2$; hence, the system should produce more than one output sample per input sample on the average. The increase in the rate can be noted by studying the output sample $y[4]$. It should be noted that the required delay for $y[4]$ is $4/3$ samples, exceeding the unit delay for the first time. The sample $y[4]$ is generated from the input samples of $x[2]$ and $x[3]$ which are two neighboring input samples. It should also be noted that the same set of input samples ($x[2]$ and $x[3]$) is also used for the generation of $y[3]$. Hence, the set of samples $x[2]$ and $x[3]$ is utilized twice for the generation of two output samples and this leads to the increase in the sampling rate. It should also be noted that the case described for $y[4]$ occurs repeatedly whenever the output sample has a delay exceeding a full integer value.

Figure 2 shows a possible implementation for the realization of the fractional delays depicted in Figure 1. The delay of $D[n] = \alpha n$ ($\alpha = 1/3$) and the multiplier $w[n]$, appearing in the filtering diagram in Figure 2, are listed in Table I. The table is presented to emphasize the time-varying and asynchronous nature of the scheme. It should be noted that when $D[n]$ exceeds a full integer delay; the samples in the buffers, which are shown with dark squares, are not shifted; hence the same

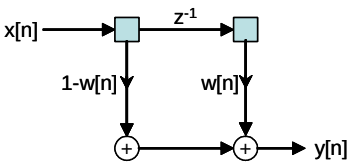

Fig. 2. Filtering scheme for the implementation of the resampling process shown in Figure 1. The multiplier values are given in Table I.

TABLE I
IMPLEMENTATION OF THE SCHEME SHOWN IN FIGURE 1

n	$D[n]$	$w[n]$	buffer contents
0	0	0	x[0], x[-1]=0
1	1/3	1/3	x[1], x[0] (updated)
2	2/3	2/3	x[2], x[1] (updated)
3	3/3	3/3	x[3], x[2] (updated)
4	4/3	1/3	x[3], x[2] (not updated)
5	5/3	2/3	x[4], x[3] (updated)
6	6/3	3/3	x[4], x[3] (updated)
7	7/3	1/3	x[4], x[3] (not updated)
⋮	⋮	⋮	⋮

buffer content is utilized twice to produce two output samples. This occurs at $n = 4$ and $n = 7$, where $D[n]$ exceeds a full integer delay, as noted in Table I.

In this section, the case of upsampling ($\alpha > 0$) is presented through an example having $\alpha = 1/3$. For $\alpha < 0$, the system becomes to a downsampling system. For the downsampling case, in analogy with the upsampling case, some buffer contents are discarded and are not used to produce an output. This leads to a reduction in the rate. We do not present further details of this case; but a detailed MATLAB implementation of the proposed scheme, efficiently implementing Nth order polynomial interpolation and handling both upsampling and downsampling cases, is given in [7].

III. Proposed Implementation

The proposed scheme generalizes the scheme described in the earlier section to higher order interpolators. Figure 3 illustrates the suggested scheme for $\alpha = 1/3$. As before, the first two samples are generated through the linear interpolator (shown with dotted lines) having the delays 0 and $1/3$, respectively. As shown in Figure 3, the next output sample, $y[2]$, is generated by combining the closest *two* input samples on each side of $y[2]$, instead of using one sample on each side as in the linear interpolator. The output sample is generated by finding the unique 3rd order polynomial passing through 4 input points in the neighborhood of $y[2]$. The method of interpolation by fitting an Nth degree polynomial to $N+1$ data inputs is called as the Lagrange interpolation, [8, p.81].

The Lagrange interpolation can be implemented in various ways. One of most efficient implementation is through the Newton series expansion, [6]. The Newton series implementation has the complexity of $O(N)$ multiplications and additions per output sample for the Nth degree interpolation.

The Newton series is the discrete time equivalent of Taylor series. The backward difference operator Δ, $\Delta\{f[n]\} = f[n] - f[n-1]$, in analogy with the derivative operator, and the Mth degree factorial polynomial

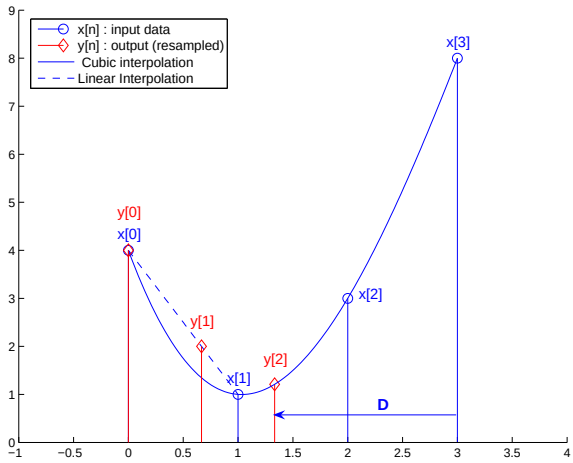

Fig. 3. The resampling process for the proposed scheme

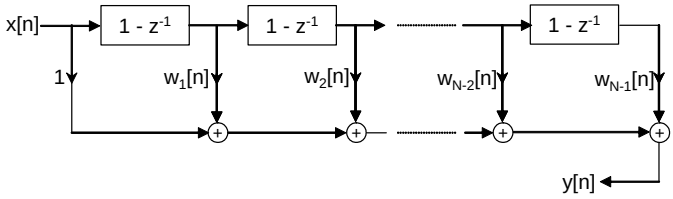

Fig. 4. Proposed structure for the sampling rate change / signal scaling.

$$x^{[M]} = \begin{cases} 1, & M = 0 \\ x(x+1)\dots(x+M-1), & M \geq 1 \end{cases}, \qquad (2)$$

in analogy with x^M, are defined to introduce the Newton series. The presented definitions lead to the fact that $\Delta\{x^{[M]}\} = Mx^{[M-1]}$, in analogy with $\frac{d}{dx}x^M = Mx^{M-1}$. Given these, Newton series can be defined as follows, [6]:

$$\widetilde{x}(k-D) \quad = \quad \sum_{i=0}^{N-1} \Delta^i\{x[k]\}\frac{(-D)^{[i]}}{i!} \qquad (3)$$

It can be noted from (3) that $\widetilde{x}(k-D) = x[k-D]$ for $D = \{0, 1, \dots, N-1\}$. This fact can be most easily verified by applying Δ operator on both sides of (3) and evaluating the result at $D = 0$, as described in [6]. It should be noted that the polynomial $\widetilde{x}(k-D)$ is the Lagrange interpolation polynomial; since it passes through $N+1$ consecutive data samples of $\{x[k-N], \dots, x[k]\}$.

The main advantage of Newton series formulation is the ease of its digital implementation. Figure 4 shows the proposed Newton Series based implementation which exactly follows the analytical relation given in (3). The implementation shown has N multipliers and N first order differencing blocks, shown with $1 - z^{-1}$. The value of the multiplier at the output of ith differencing block at the time instant of n is denoted by $w_i[n]$. The time-varying nature of the filtering scheme is evident from the implementation.

The weights $w_i[n]$ can be specified as follows. As before, $D[n]$ represents the value of the desired delay at the sample n. The proposed N point ($N-1$'th order) Lagrange interpolator scheme uses the nearest $N/2$ input samples on each side of the output sample for the interpolation. The delay at the time instant n can be written as follows:

$$D[n] = \frac{N-2}{2} + \alpha n, \quad N : \text{even} \qquad (4)$$

The term $\frac{N-2}{2}$ in (4) corresponds to the bulk delay due to the usage of $N/2$ input points on the right side of the output sample. (The bulk delay is required for the causality of the implementation.) The term of αn in (4) corresponds to the time varying delay. When $N = 2$, the equation (4) reduces to the definition given in the previous section for the linear interpolator.

Following the formulation of (3), the filter coefficients $w_i[n]$ $i = \{0, \dots, N-1\}$ can be explicitly written as follows:

$$d_i[n] = \frac{1}{i!}\left(-\left(\frac{N-2}{2} + \text{rem}(\alpha n, 1)\right)\right)^{[i]} \qquad (5)$$

The term of $(N-2)/2 + \text{rem}(\alpha n, 1)$ corresponds to the effective fractional delay. The function $\text{rem}(\cdot, 1)$ represents the non-integer (fractional) part of the argument. More specifically, $\text{rem}(\cdot, A)$ corresponds to the remainder operation when the argument is divided by $A + \epsilon$ for an arbitrary small positive ϵ, i.e. the range of $\text{rem}(\cdot, A)$ is $[0, A]$.

The appearance of $\text{rem}(\cdot, 1)$ in (5) is due to the time-variation of the delay. When the delay $D[n]$ exceeds a full integer value, the buffer contents are not updated. By choosing not to update the buffer contents for such instances, the additional unit sample increase in the delay is integrated into the system. The remaining part of the delay after the integration of its integral part becomes $\text{rem}(\alpha n, 1)$.

The described operation can be most easily seen from Table I. At the sample of $n = 4$, the desired delay is $4/3$ samples. To implement this delay, the buffer contents for $n = 3$ is used to the produce the 4th output sample and the required fractional delay becomes $4/3 - 1 = 1/3$ (which is identical to the multiplier $w[n]$) for the linear interpolation scheme (also see Figure 1).

The present discussion is focused on the case of upsampling ($\alpha > 0$) with even number of input points (N : even) for the interpolation. Both restrictions can be easily lifted as follows:

Case of Odd N: The description given above assumes the number of points utilized in the Lagrange interpolator is even. The approach and the filtering scheme given in Figure 4 can be used as it is also for the odd values of N by adopting $D[n] = \frac{N-3}{2} + \alpha n$ instead of (4). This choice corresponds to usage of $\frac{N-1}{2}$ samples on the right side of output sample and $\frac{N+1}{2}$ samples on the left side of the output sample. (There can be other alternatives.) The bulk delay in (5) for the calculation of multiplication coefficients becomes $\frac{N-3}{2}$. This applies to both upsampling and downsampling cases.

Case of Downsampling ($\alpha < 0$): The downsampling case follows quite easily from the given discussion. One important change is the change in the argument of $\text{rem}(\cdot, 1)$ appearing in (5) from αn to $(1+\alpha)n$. For further details, readers are invited to examine the detailed MATLAB code given for both downsampling and upsampling cases, [7].

IV. Numerical Results

In all three sets of experiments presented in this section, the input is taken as $x[n] = \cos(\omega n)$. The output produced by the scheme is compared with the desired output of $x_\beta[n] =$

$\cos(\beta\omega n)$ in the root mean square (RMS) sense for various values of signal frequency (ω) and the scaling factor (β).

Figure 5 shows the output produced by the scheme for $\omega = \pi/4$ and $\beta = 4/5$; stated differently, the input $x[n] = \cos(\pi/4n)$ is resampled such that the output is $x_\beta[n] = \cos(\pi/5n)$. Figure 5 shows the scheme output for linear ($N = 2$) and cubic interpolation ($N = 4$). It can be noted that the case of $N = 4$ is almost identical to the true output for the given ω and β values.

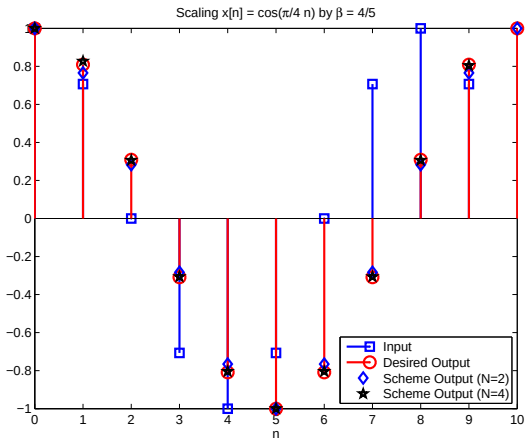

Fig. 5. Output for the scaling of $\cos(\pi/4n)$ by $\beta = 4/5$.

Figure 6 shows the RMS error when the input $\cos(\pi/4n)$ is scaled by $\beta \in [0.5, 1.5]$ using different order Lagrange interpolators. Given range for β corresponds to the range of sampling rate change from $2/3$ (downsampling) to 2 (upsampling). This range is beyond the main motivation for the presented scheme; but we can note that the scheme presents a good performance for a fairly large range of β values for the frequency of $\omega = \pi/4$. It should be noted that the case of $\beta = 1$ corresponds to the case of no sampling rate change for which no error is incurred. In addition, the RMS value for $\cos(\pi/4\beta n)$ (desired output), which is $1/\sqrt{2}$, is also indicated as signal RMS in Figure 6 for comparison purposes.

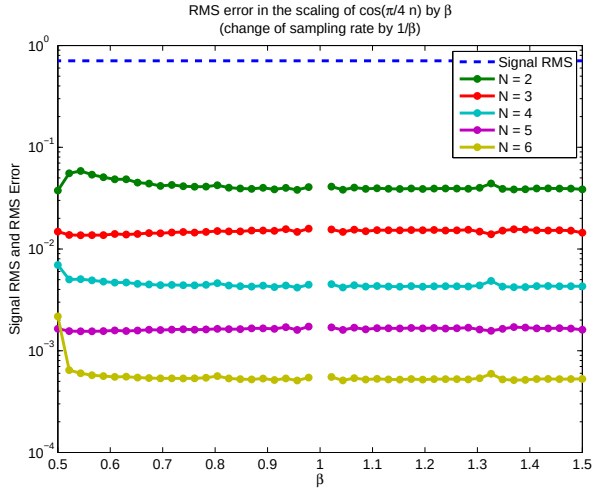

Fig. 6. RMS error for the scaling of $\cos(\pi/4n)$ by β.

Figure 7 shows the error for $\beta = 0.9$ when the frequency of the input $\cos(\omega n)$ is varied. As $\omega \to \pi$, the polynomial interpolation ceases to be useful; since rapid oscillations can not be faithfully approximated by a small degree polynomial. It can be noted from Figure 7 that the presented scheme works well for $|\omega| \leq \pi/2$. Hence, it is possible to conclude that the suggested method can be applied for a wide range of β, not only for $\beta \approx 1$, if the input is at least two times oversampled. If this condition is not satisfied, the designer implementing a gentle rate change may choose to upsample the input by 2 and then apply the suggested method.

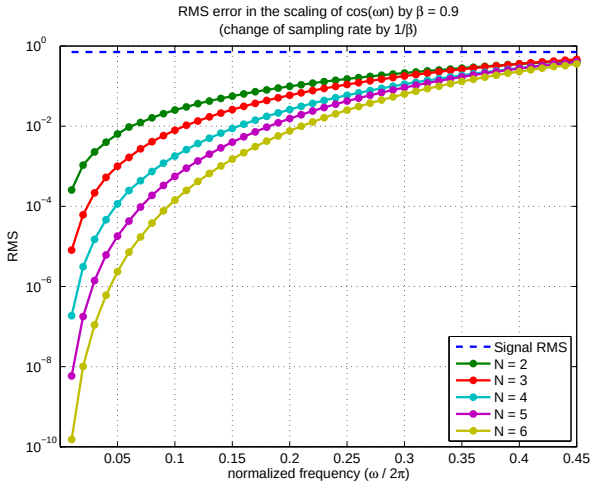

Fig. 7. RMS error for the scaling of $\cos(\omega n)$ by $\beta = 0.9$.

V. Conclusions

A scheme to digitally scale the sampled input or to change its sampling rate is suggested. The scheme requires very few operations (N multiplications for Nth order interpolation) per output sample and yields a satisfactory performance for a wide range of input frequency and scaling factor. We believe that the main application of the presented scheme would be in the change of sampling rate around unity, such as $\beta = 24/25$, for which the conventional technique of upsampling followed by downsampling is not a feasible approach.

References

[1] R. Istepanian and M. Stojanovic, *Underwater Acoustic Digital Signal Processing and Communication Systems*. USA: Springer Press, 2002.

[2] K. Rajamani, Y.-S. Lai, and C. Furrow, "An efficient algorithm for sample rate conversion from CD to DAT," *IEEE Signal Processing Lett.*, vol. 7, no. 10, pp. 288–290, Oct. 2000.

[3] A. Tarczynski, W. Kozinski, and G. Cain, "Sampling rate conversion using fractional-sample delay," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, Apr. 1994, pp. 285–288.

[4] F. Harris, "Performance and design of Farrow filter used for arbitrary resampling," in *13th Int. Conf. on Digital Signal Processing Proceedings,*, July 1997, pp. 595–599.

[5] M. Blok, "Fractional Delay Filter Design for Sample Rate Conversion," in *Proc. Federated Conference on Computer Science and Information Systems*, Sept. 2012, pp. 701–706.

[6] C. Candan, "An Efficient Filtering Structure for Lagrange Interpolation," *IEEE Signal Processing Lett.*, vol. 14, no. 1, pp. 17–19, Jan. 2007.

[7] ——. (2013) Changing Signal Scale or Sampling Rate "Gently" By Fractional Delay Filtering, MATLAB Code. [Online]. Available: http://www.eee.metu.edu.tr/∼ccandan/pub.htm

[8] F. B. Hildebrand, *Introduction to numerical analysis*. New York: McGraw-Hill, 1974.

MATLAB CODES:

The following is the MATLAB function prepared for the implementation of the suggested scheme. (MATLAB codes can also be downloaded from the author's webpage, http://www.eee.metu.edu.tr/~ccandan/pub.htm)

```
function out = digscale(s,beta,Npoint)
% function out = digscale(s,beta,Npoint)
% Scales the input vector s by beta.
%
% The operation is equivalent of out(t) = s(beta t) in discrete time
% The operation can be interpretted as the sampling rate
%
% s      : input
% beta   : scaling coefficient (beta = 1 - alpha)
% Npoint : Number of points to be used Lagrange interpolation
%
% out    : output
%
% Cagatay Candan
% Jan. 2013
%

if Npoint ≤ 0, disp ('Npoint should be at least 2'); return; end;

alpha = 1 - beta; % alpha > 0 --> Higher Sampling Rate or sig. expands in time
                  % alpha < 0 --> Lower  Sampling Rate or sig. compresses

N = length(s);

if alpha>0,   %Upsampling

    %Construct A and B (state model)
    A = diag(ones(1,Npoint-1),-1);
    b = zeros(Npoint,1); b(1)=1;

    %Construct C1 for the output of differencing blocks
    C1 = zeros(Npoint,Npoint);
    dum = 1; C1(1,:)=[dum zeros(1,Npoint-1)];
    for dumind = 1:Npoint-1,
        dum = conv([1 -1],dum);
        C1(dumind+1,:)=[dum zeros(1,Npoint-1-dumind)];
    end;

    %Delay (Compensate Anti-Causal Part of Response)
    delay = floor((Npoint-2)/2);
    x = zeros(Npoint,1); %Default Buffer Contents
    for input_index = 1: delay, %Fill Buffers "delay" number of times
        x = A*x + b*s(input_index); %State Model for buffer contents
    end;

    %
    output_index = 0; D = -alpha;
    for input_index=delay+1:N,
        D = D + alpha;
        output_index = output_index + 1;

        if D > 1 + 100*eps, %Insert Extra Sample
            [coefvec,Dvec2]=construct_multipliers(Npoint,alpha,output_index-1);
            out(output_index) = coefvec'*out_diff;  %Extra Sample without buffer update
%            coefvec'*C1;

            D = D + alpha - 1;
            output_index = output_index + 1;
        end;

        x = A*x + b*s(input_index); %State Model for buffer contents
        out_diff = C1*x; %Output of differencing blocks

        [coefvec,Dvec2]=construct_multipliers(Npoint,alpha,output_index-1);
%        coefvec'*C1,

        out(output_index) = coefvec'*out_diff;


    end;
```

```
71
72  else %if alpha<0  %Downsampling
73
74      %Construct A and B (state model)
75      A = diag(ones(1,Npoint-1),-1);
76      b = zeros(Npoint,1); b(1)=1;
77
78      %Construct C1 for the output of differencing blocks
79      C1 = zeros(Npoint,Npoint);
80      dum = 1; C1(1,:)=[dum zeros(1,Npoint-1)];
81      for dumind = 1:Npoint-1,
82          dum = conv([1 -1],dum);
83          C1(dumind+1,:)=[dum zeros(1,Npoint-1-dumind)];
84      end;
85
86      %Delay (Compensate Anti-Causal Part of Response)
87      delay = floor((Npoint-2)/2)+2;
88      x = zeros(Npoint,1); %Default Buffer Contents
89      for input_index = 1: delay, %Fill Buffers "delay" number of times
90          x = A*x + b*s(input_index); %State Model for buffer contents
91      end;
92
93      %
94
95      output_index = 1;
96      out(1) = s(1); D = 0;
97      for input_index=delay+1:N,
98          D = D - alpha;
99
100         x = A*x + b*s(input_index); %State Model for buffer contents
101         out_diff = C1*x; %Output of differencing blocks
102
103         if D > 1  - 100*eps, %Skip Sample
104             D = D - 1 + alpha;
105             continue,
106         else
107               [coefvec,Dvec2]=construct_multipliers(Npoint,alpha,output_index);
108 %              coefvec'*C1,
109               output_index = output_index + 1;
110               out(output_index) = coefvec'*out_diff;
111         end;
112
113
114     end;%end of else
115 end; %end of if alpha
116
117 %%%%%%%%
118 function [coefvec,Dvec2] = construct_multipliers(Npoint,alpha,n)
119 % Calculates the multipliers shown with d_i[n] in the manuscript
120 %
121 % coefvec : multipliers d_i[n], 0 ≤ i ≤ Npoint -1
122 % Dvec2   : generated to verify the system
123 %         :  fliplr(Dvec2) should be identical to Dvec
124 %         %  calculated in the main function
125 %
126 % The output of this function is not used in the implementation given.
127 % The function is presented to verify the results of manuscript.
128
129 if (alpha > 0), %upsampling
130     if rem(Npoint,2) == 0,  %Npoint : Even
131         D =  - ((Npoint - 2)/2 + rem(alpha*n,1+1500*eps));
132     else  %Npoint : Odd
133         D =  - ((Npoint - 3)/2 + rem(alpha*n,1+1500*eps));
134     end;
135 else %alpha <0, downsampling
136     if rem(Npoint,2) == 0,  %Npoint : Even
137         D =  - ((Npoint - 2)/2 + rem((1+alpha)*n,1+1500*eps));
138     else  %Npoint : Odd
139         D =  - ((Npoint - 3)/2 + rem((1+alpha)*n,1+1500*eps));
140     end;
141 end;
142
143 coefvec = zeros(Npoint,1);
144 coefvec(1) = 1; diffblock{1}=1;
145 for ii=1:Npoint-1,
146     coefvec(ii+1) = coefvec(ii)/ii * (D + ii-1);
147     diffblock{ii+1} = conv([1 -1],diffblock{ii});
```

```
148 end;
149
150 Dvec2 = zeros(1,Npoint);
151 for ii=1:Npoint,
152     Dvec2 = Dvec2 + ...
153         coefvec(ii)*[diffblock{ii} zeros(1, Npoint-length(diffblock{ii}))];
154 end;
```

The following script generates Figure 5 and is given to illustrate the usage of the function digscale.m

```
N=10; alpha = 1/5;
omega = 2*pi*1/8;
n=0:N-1;
s=cos(omega*n);

Npoint=2; out2 = digscale(s,1-alpha,Npoint);
Npoint=4; out3 = digscale(s,1-alpha,Npoint);

nout = -1 + (1:length(out2)); sout = cos(omega*(1-alpha)*nout);

h = figure(1); set(h,'Defaultaxesfontsize',12); set(h,'Defaulttextfontsize',12);

stem(0:length(s)-1,s,'s','markersize',10,'linewidth',2), hold all;
stem(0:length(sout)-1,sout,'r','markersize',11,'linewidth',2),
plot(0:length(out2)-1,out2,'bd','markersize',8,'linewidth',2),
plot(0:length(out3)-1,out3,'kp','markersize',8,'linewidth',2),

title(['Scaling x[n] = cos(\pi/4 n) by \beta = 4/5']);
legend('Input','Desired Output','Scheme Output (N=2)', 'Scheme Output (N=4)','location','SouthEast');
xlabel('n');
hold off;
```

The following script generates Figure 7 and is given to illustrate the usage of the function digscale.m

```
N=500;
select=20:119;

freqvec=linspace(0.01,0.45,45);
Npointvec=2:6;
alpha = 0.1;

beta = 1 - alpha;
n=0:N-1;

h = figure(1); set(h,'Defaultaxesfontsize',12); set(h,'Defaulttextfontsize',12);
set(gca,'xtick',0.05:0.05:0.45)

rmssignal = 1/sqrt(2);
semilogy(freqvec,rmssignal*ones(1,length(freqvec)),'--','linewidth',2); drawnow; hold all;

for Npoint = Npointvec,
    error = [];freqind = 0;
    for freq = freqvec,
        omega = 2*pi*freq;
        s=cos(omega*n);

        out = digscale(s,beta,Npoint);
        nout = -1 + (1:length(out));
        s2 = cos(omega*(1-alpha)*nout);
        e = s2-out;
        e = e(select);
        freqind = freqind + 1;
        error(freqind) = sqrt(mean(e.^2));

    end;
    semilogy(freqvec,error,'.-','linewidth',2,'markersize',20);  drawnow;
end;

title(['RMS error in the scaling of cos(\omegan) by \beta = ' num2str(beta) char (10) ...
       '(change of sampling rate by 1/\beta)']);

xlabel('normalized frequency (\omega / 2\pi)');
ylabel('RMS'); ;
dumstring={'Signal RMS', 'N = 2'  ,  'N = 3'  ...
         , 'N = 4'  ,  'N = 5' ,   'N = 6'};
legend(dumstring,'Location','SouthEast');
set(gca,'xtick',0.05:0.05:0.45)
dum = axis; axis([0 0.45 dum(3:4)]);
grid on
hold off;
```