7.0. Introduction

In Part 6 we have developed a few numerical algorithms that will serve us as the basis of system analysis and synthesis. We have shown how simple it is to implement the analytical expressions, related to the various aspects of system performance, into compact, fast-executing computer code, which reduces the tedious mathematics to pure routine. Of course, a major contribution to this easyness was provided by the programming environment, a high-level, math-oriented language called Matlab™ (Ref. [7.1]).

As wide-band amplifier designers, we want to be able to accurately predict its performance, particularily in the time-domain. With the algorithms developed, we now have the essential tools to revisit the circuits presented in Part 1 - 5 , possibly gainig a better insight of how to put them to use in our new designs eventually.

But the main goal of Part 7 is to put the algorithms in a wider perspective. Here we intentionally use the term "system", in order to emphasize the fact that there is a high degree of integration in modern electronics design, which forces us to abandon the old paradigm of adding up separately optimized subsystems into the final product ; instead, the design process should be concieved to optimize the total system performance from the start. As more and more digital processing power is being built-in into modern products, the analogue interface with the real world needs to be given adequate treatment on the system level.

7.1. Using Convolution : Response to Arbitrary Input Waveforms

The algorithms that we have developed in Part 6 gave us the system time-domain response to two special cases of input signal : the unit-area impulse and the unit-amplitude step. Here we will consider the response to any type of input signal, provided that its application will not exceed neither the input nor the output system capabilities. In technical literature this is known as the BIBO-condition[1]. And, of course, we are still within the constraints of our initial LTIC-conditions[2].

As we have seen in Part 1, Sec. 1.14, the system time-domain response to an arbitrary input signal can be calculated in two ways :

a) either by transforming the input signal to complex-frequency domain, multiplying it by the system transfer function and transforming the result back to time-domain, or
b) directly in time-domain by the convolution integral.

A short reminder of how the convolution integral was defined and the transcription from differential to difference form is in order here. Let $x(t)$ be the time-domain signal, presented to the input of a system being characterized by its impulse-response $h(t)$. The system output can then be calculated by convolving $x(t)$ with $h(t)$:

$$y(t) = \int_{t_0}^{t_1} h(\tau - t)\, x(t)\, dt \tag{7.1.1}$$

where τ is a time-constant, it's value chosen so that $h(t)$ is time-reversed. Usually, it is sufficient to make τ large enough to allow the system impulse-response, $h(t)$, to completely relax and reach the steady-state again (not just the first zero-crossing point !).

If $x(t)$ was applied to the system at t_0, then this can be the lower limit of integration. Of course, the time scale can be safely renormalized so that $t_0 = 0$. The upper integration limit, labeled t_1, can be wherever needed, depending on how much of the input and output signal we are interested in.

Now, dt in Eq. 7.1.1 is implicitly approaching zero and so there would be an infinite number of samples between t_0 and t_1. Since our computers have a limited amount of memory (and we have a limited amount of time !), we must make a compromise between the sampling rate and the available memory length. If M is the number of memory bytes reserved for $x(t)$, then the required sampling time interval will be :

$$\Delta t = (t_1 - t_0)/M \tag{7.1.2}$$

[1] Bounded input - bounded output. Interestingly, the inverse of BIBO is in wide-spread use in the computer world. In fact, any digital computer is a GIGO-type device (= garbage in - garbage out).
[2] Linearity, time-invariance and causality.

So, if Δt replaces dt, the integral in Eq. 7.1.1 transforms into a sum of M elements, $x(t)$ and $y(t)$ become vectors `x(n)` and `y(n)`, where `n` is the index of a signal sample location in memory, and $h(\tau - t)$ becomes `h(m-n)`, with `m=length(h)`, resulting in:

$$\mathtt{y(n)} = \sum_{\mathtt{n=1}}^{\mathtt{M}} \mathtt{h(m-n)*x(n)} \qquad (7.1.3)$$

Here Δt is implicitly set to 1, since the difference between two adjacent memory locations is unit integer. Good book-keeping practice, however, recommends the construction of a separate time-scale vector, with values from t_0 to t_1, in Δt increments between adjacent values. All other vectors are then plotted against it, as we have seen it done in Part 6.

In **Part 1 - 4** we have learned that analytically solving the convolution integral can be a time-consuming task even for a skilled mathematician. Sometimes, even if $x(t)$ and $h(t)$ are analytical functions, their product need not be elementary integrable in the general case. But taking the $\mathcal{L}$-transform route may sometimes be equally difficult. Fortunately, numerical computation of the convolution integral, following Eq. 7.1.3, can be programmed easily:

```
function y=vcon(h,x)
%VCON   Convolution, step-by-step example. See also CONV and FILTER.
%
%    Call :      y=vcon(h,x);
%
%    where:      x(t) --> the input signal
%                h(t) --> the system impulse response
%                y(t) --> the system response to x(t) by convolving
%                         h(t) with x(t).
%    If  length(x)=nx  and  length(h)=nh, then length(y)=nx+nh-1.

%    Erik Margan, 890416,  Free of copyright !

                        % force h to be the shorter vector :
if length(h) > length(x)
     xx=x; x=h; h=xx;  % exchange x and h via xx;
     clear xx
end
nh=length(h);
nx=length(x);
h=h(:).';  % organize x and h as single-row vectors ;
x=x(:).';

y=zeros(2,nx+nh-1);     % form a (2)-by-(nx+nh-1) matrix y, all zeros
y(1,1:nx)=h(1)*x;       % first row : multiply x by the first element of h
for k=2:nh
                        % second row : multiply and shift (insert 0):
     y(2, k-1:nx+k-1)=[0, h(k)*x];
                        % sum the two rows column-wise and
                        % put result back into first row :
     y(1,:)=sum(y);
end                     % repeat for all remaining elements of h ;
y=y(1,:);               % the result is the first row only ;
```

To get a more clear view of what the VCON routine is doing, let's write a short numerical example, using a 6-sample input signal and a 3-sample system impulse-response and displaying every intermediate result of the matrix y in VCON:

```
x=[0 1 3 5 6 6];    h=[1 3 -1];    vcon(x,h);

% initialization - all zeros, 2 rows, 6+3-1 columns :
    0     0     0     0     0     0     0     0
    0     0     0     0     0     0     0     0
% step 1 : multiply x by the first sample of h, h(1)=1 and
% insert it into the first row :
    0     1     3     5     6     6     0     0
    0     0     0     0     0     0     0     0
% step 2 : multiply x by the second sample of h, h(2)=3,
% shift it one place to the right by adding a leading zero and
% insert it into the second row :
    0     1     3     5     6     6     0     0
    0     0     3     9    15    18    18     0
% step 3 : sum both rows vertically and put the result in the first row
    0     1     6    14    21    24    18     0
    0     0     3     9    15    18    18     0
% iterate steps 2 and 3, each iteration using the next sample of h :
    0     1     6    14    21    24    18     0
    0     0     0    -1    -3    -5    -6    -6

    0     1     6    13    18    19    12    -6
    0     0     0    -1    -3    -5    -6    -6
% after 2 iterations (as h is only 3 samples long)
% the result is the first row of y :
    0     1     6    13    18    19    12    -6
% actually, the result is only the first 6 elements :
    0     1     6    13    18    19
% since there are only 6 elements in x, the process assumes the rest
% to be zeros. So the remaining two elements of the result represent
% the relaxation from the last value (19) to zero by the integration
% of the system impulse response h.

% Basically, the process above does the following :
% ( note the reversed sequence of h )

                0  1  3  5  6  6
          -1  3  1
(↓*) ==>        0                 (→+) ==> 0

                0  1  3  5  6  6
             -1  3  1
(↓*) ==>        0  1              (→+) ==> 1

                0  1  3  5  6  6
               -1  3  1
(↓*) ==>        0  3  3           (→+) ==> 6

                0  1  3  5  6  6
                  -1  3  1
(↓*) ==>        0 -1  9  5        (→+) ==> 13

% ......... etc.
```

For convolution, Matlab has a function named CONV, which uses a built-in FILTER command to run substantially faster, but in this way the process remains hidden to the user; however, the final result is the same as with VCON.

Another property of Matlab is the matrix indexing, which starts with 1 (see the rug of the sum symbol in Eq. 7.1.3), in contrast to most programming languages, which use memory "pointers", base address + offset, thus the offset of the first element of an array is 0.

Let's now use this routine in some real-life examples. Suppose we have a gated sine-wave generator connected to the same 5th-order Butterworth system which we inspected in detail in Part 6. Also, let the the Butterworth system half-power bandwidth be 1 kHz, the generator frequency 1.5 kHz and we turn on the gate at signal zero-crossing. From the frequency-response calculations, we know that the forced-response amplitude would be:

```
Aout=Ain*abs(freqw(z,p,1.5));
```

where `z,p` are the zeros and poles of the normalized Butterworth system.

But how will the system respond to the signal turn-on transient ?

We can simulate this, using the algorithms we have developed in Part 6:

```
fh=1000;                % system half-power bandwidth, 1kHz
fs=1500;                % input signal frequency, 1.5kHz
t=(0:1:300)/(50*fh);    % time vector, 20us delta-t, 6ms range
nt=length(t);

[z,p]=buttap(5);        % 5th-order Butterworth system
p=2*pi*fh*p;            % denormalized system poles
h=atdr(z,p,t,'n');      % system impulse-response

d=25;                   % switch-on delay, 25 time-samples
                        % make the input signal :
x=[zeros(1,d), sin(2*pi*fs*t(1:nt-d))];

y=vcon(h,x);            % convolve x with h ;

A=nt/(2*pi*fh*max(t)); % denormalize amplitude of h for plot
                        % plot input, system i.r. and convolution result
plot( t*fh, x, '-g', ...
      t*fh, [zeros(1,d), h(1:nt-d)*A],'-r', ...
      t*fh, y(1:nt), '-b')
xlabel('Time [ms]')
```

The convolution result, compared to the input signal and the system impulse-response is shown in Fig. 7.1.1.

Note that we have plotted only `nt` samples of the convolution result, however the actual length of `y` is `nx+nh-1`, or one sample less that the sum of the input and system response lengths. The first `nx=nt` samples of `y` represent the system response to `x` and the remaining `nh-1` points are the consequence of the system relaxation: since there are no more signal samples in `x` after the last point `nx`, the convolution assumes them to be zero.

The system relaxation is therefor the step-response from the last signal value to zero. So, if we are interested only in the system response to the input signal, we simply limit the response vector to the same length as was the input signal. Also, in the general case, the length of the system impulse response vector, `nh`, does not have to be equal to the input signal vector length, `nx`. In practice, we often make `nh << nx`, but `h` should be large enough to allow the system to relax to a level very close to zero, as only then the sum of all elements of `h` will not differ much from the system gain.

There is, however, an important difference in the plot and the calculation, which must be explained. The impulse-response that we have got from Butterworth system poles was normalized to represent a unity-gain system, since we want to see the frequency-dependence on the output amplitude by comparing the input and output signals. Thus our composite system should either have a gain of unity, or the output should be normalized to the input in some other way (i.e. if the gain was known, we could have divided the output signal by the gain, or multiplied the input signal). But the unity-gain normalized impulse-response would be too small in amplitude, compared to the input signal, so we plot it denormalized.

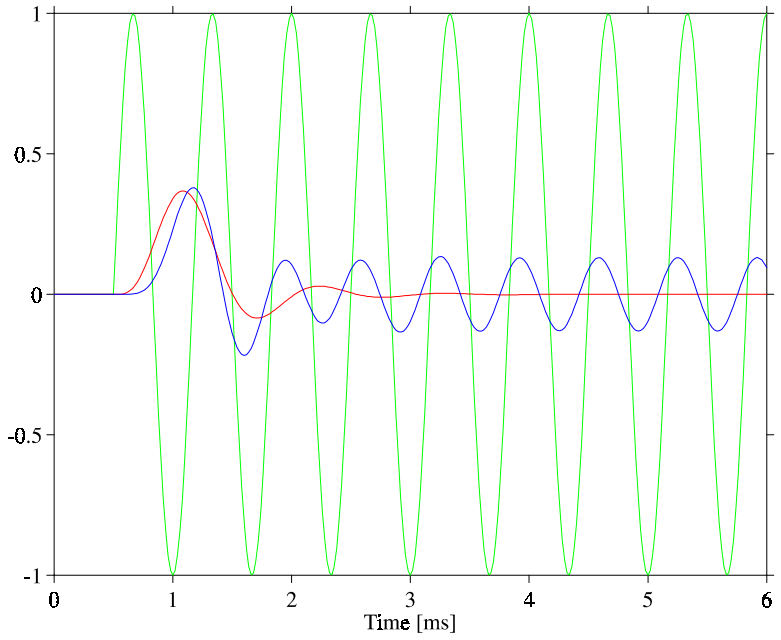

Fig. 7.1.1 : Convolution example : response (blue) to a sine wave (green) switched-on into a 5th-order Butterworth system, whose impulse-response is shown (red) denormalized (instead of unity-gain) and delayed by the same switch-on time. Shown in such relation, it is now clear that the system responds by phase-shifting and amplitude modulating the first few wave periods, as indicated by the impulse-response, reaching finally the forced ("steady-state") response.

How can we check that our routine works correctly ?

Apart from entering some simple number sequences as in the previous example, we could do this by entering an input signal for which the result we have already calculated in a

different way, i.e. the system step-response (see Fig. 6.5.1, Part 6). If we enter a unit-step, instead of the sine-wave, we should get the step response from convolution :

```
% continuing from above :

x=[zeros(1:d) ones(1:nt-d)];                % input unit-step function
y=vcon(h,x);                                % convolution

plot( t*fh, x, '-g', ...
      t*fh, [zeros(1,d), h(1:nt-d)*A],'-r', ...
      t*fh, y(1:nt), '-b')
xlabel('Time [ms]')
```

Neglecting the initial 25 sample time-delay, the step response of Fig. 7.1.2 should be identical to that of Fig. 6.5.1, Part 6.

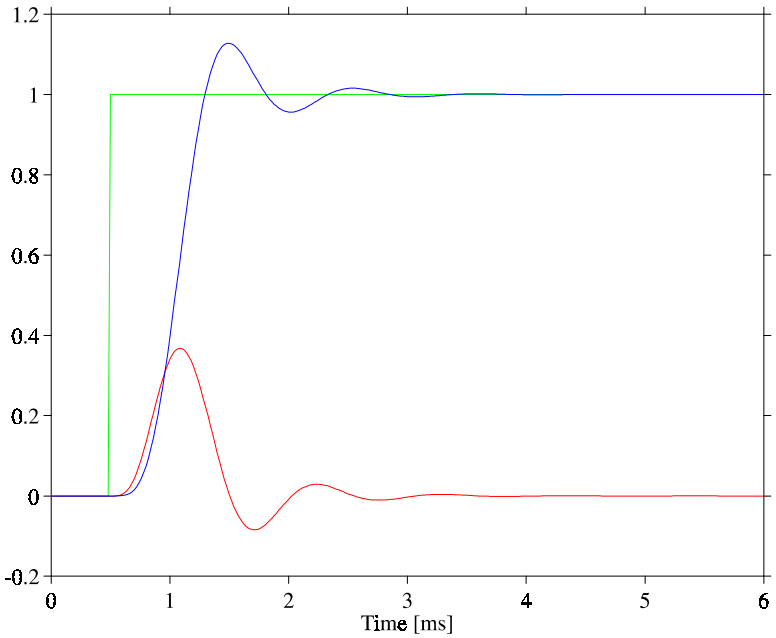

Fig. 7.1.2 : Checking convolution: response to a unit-step of the 5th-order Butterworth system, whose impulse-response (normalized to its peak value and delayed by the same switch-on time) is also shown, along with the input step signal. The step-response, apart from the 25-sample switch-on time delay and the time-scale, is identical to the one shown in Part 6, Fig.6.1.11.

We can now revisit the convolution integral example given in Part 1, Sec. 1.14, shown in Fig. 1.14.1 . There was a unit-step input signal, fed into a two-pole Bessel-Thomson system, its output in turn being fed into a two-pole Butterworth system. The commands below simulate the final result of Fig. 1.14.1 . But this time, let's use the frequency-to-time domain transform of the TRESP routine (Part 6). See the result in Fig. 7.1.3 and compare it to Fig. 1.14.1g .

```
[z1,p1]=bestap(2,'t'); % Bessel-Thomson 2nd-order system poles
[z2,p2]=buttap(2);     % Butterworth 2nd-order system poles
N=256;                 % number of samples
m=4;                   % set the bandwidth factor
w=(0:1:N-1)/m;         % frequency vector, w(m+1)=1 ;
F1=freqw(p1,w);        % Bessel-Thomson system frequency response
F2=freqw(p2,w);        % Butterworth system frequency response

[S1,t]=tresp(F1,w,'s');% step-response of the Bessel-Thomson system,
                       % the normalized time vector is same for both.
I2=tresp(F2,w,'u');    % unity-gain normalized Butterworth inpulse-resp.;
d=max(find(t<=15));    % limit the plot to first 15 time units
I2=I2(1:d);            % limit the I2 vector length to d

     % convolution of Bessel-Thomson system step-response with
     % the first d points of the Butterworth impulse response :
y=vcon(I2,S1);

A=N/(2*pi*m*max(t));   % amplitude denormalization for I2
     % plot first d points of all three responses vs. time :
plot( t(1:d), S1(1:d), '-r',...
      t(1:d), I2(1:d)*A, '-g',...
      t(1:d),  y(1:d), '-b' )
xlabel('Time [s]')
```

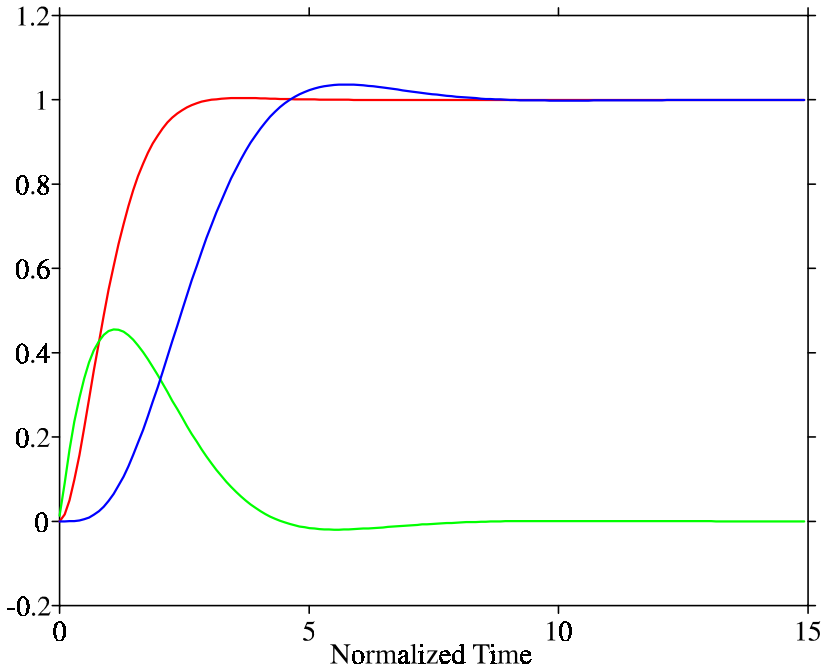

Fig. 7.1.3 : Convolution example of Part 1, Sec. 1.14. : a Bessel-Thomson 2-pole system step-response (red), fed to the 2-pole Butterworth system, its impulse-response (green, shown here denormalized in amplitude) was used for convolution, resulting in the output step-response (blue). Compare this with Fig. 1.14.1(G).

The VCON function is a lengthy process. On a 12 MHz AT-286 PC, which I was using for first experiments back in 1986-7, it took more than 40 s to complete the example shown above, but even with today's fast computers there is still a noticeable delay, even if the slow VCON is replaced by the Matlab original CONV function.

In some cases, particularly with long signal sequences (N>1000), it could be interesting to take the Fourier-transform route, numerically.

Here is an example, using a signal recorded by a nuclear magnetic resonance imaging (MRI) system. The signal is noisy and there is some interference from another source, which we will try to clean by a 5th-order Bessel-Thomson unity-gain 1 MHz cut-off filter :

```
load R.dat              % load the recorded signal from a file "R.dat"
N=length(R);            % total vector length, N=2048 samples
Tr=102.4e-6;            % record length 102.4 us
dt=Tr/N;                % samppling time interval, 50 ns
t=dt*(0:1:N-1);         % time vector reconstruction

% plot the first 1200 samples of the recorded signal, see Fig.7.1.4a
plot(t(1:1200,R(1:1200),'-g')
xlabel('Time [\mus]')  % input signal, fist 60us

G=fft(R);               % G is spectrum of R, Fast-Fourier-Transformed
G=G(1:N/2);             % use only up to the Nyquist freq. ( 10 MHz )
f=(1:1:N/2)/dt;         % frequency vector reconstructed

[z,p]=bestap(5,'n');    % 5rd-order Bessel filter poles
p=p*2*pi*1e+6;          % half-power bandwidth is 1 MHz
F=freqw(z,p,2*pi*f);    % filter frequency response

% multiplication in frequency equals convolution in time :
Y=F.*G;                 % output spectrum

x=max(find(f<=8e+6));  % plot spectrum up to 8 MHz
M=max(abs(G));          % normalize the spectrum to its peak value
plot( f(1:x), abs(F(1:x)), '-r', ...
      f(1:x), abs(G(1:x))/M, '-g', ...
      f(1:x), abs(Y(1:x))/M, '-b' )
xlabel('Frequency [MHz]'), ylabel('Normalized Magnitude')
% see Fig.7.1.4b

y=2*(real(fft(conj(Y)))-1)/(N/2);  % return to time domain

a=max(find(t<=5e-5));
b=min(find(t>=20e-6));
plot( t(a:b), g(a:b), '-g', t(a:b), y(a:b), '-b' )
xlabel('Time [\mus]')  % see Fig.7.1.4c
```

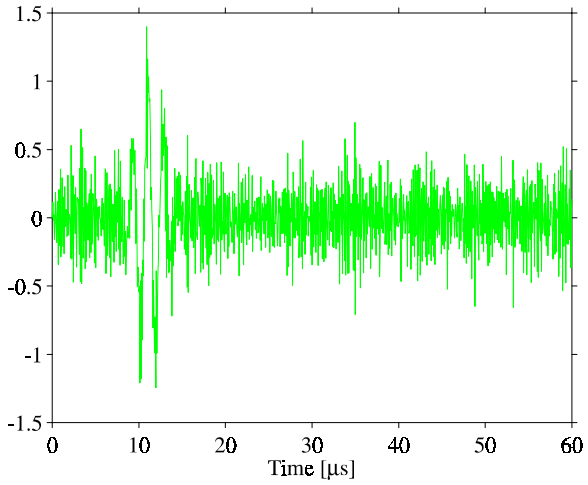

Fig. 7.1.4a : Input signal for the example of convolution in spectral-domain

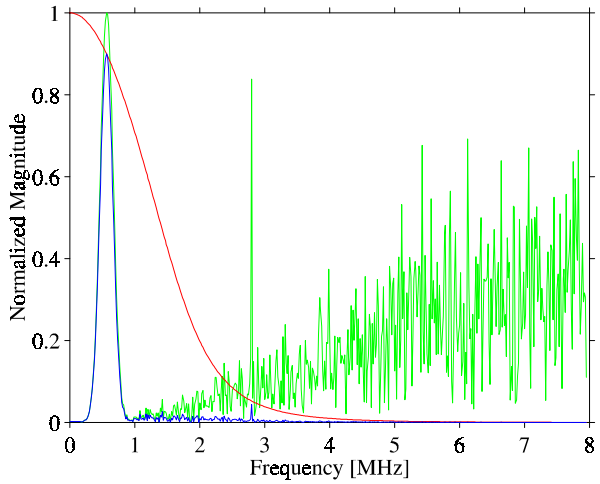

Fig. 7.1.4b : The spectrum of the signal in Fig. 7.4.1 (green) is multiplied with the system frequency-response (red) to produce the output spectrum (blue). Along with the modulated signal at 560kHz, there is a 2.8MHz interference and a high level of white noise (raising with frequency).

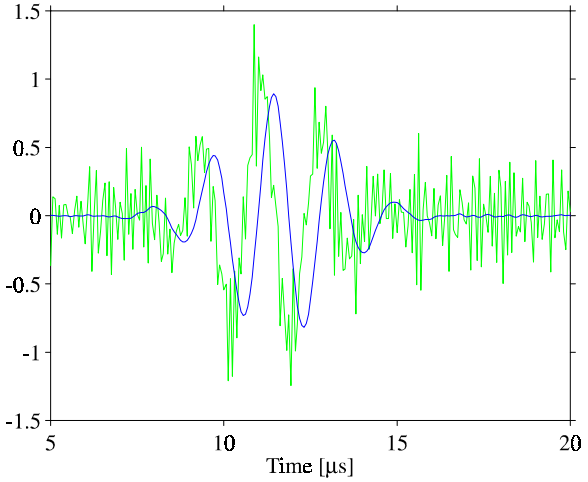

Fig. 7.1.4c : The output spectrum returned to time-domain (blue), compared with input (green), in expanded time scale. Note the change in amplitude, the noise level and the envelope-delay time-shift, with very little change in phase.