P. Starič, E. Margan:

Wideband Amplifiers

Part 6:

Computer Algorithms for Analysis and Synthesis of Amplifier–Filter Systems

If you search for something long enough,
you will certainly discover something else.

(Erik's First Amendment to Murphy's Law
applied to scientific research)

How It All Began

Ever since I heard of 'electronic brains', back in the early 1960s, I was asking all sorts of people how those things work, but I never received any answer, with the exception of one, coming from a medical profession, saying that no one understood the biological brains either, so how could I expect to understand the electronic ones? Much later I discovered that an awful lot of people did not like using their brains at all, and they were doing just fine without it, thank you for asking!

In the autumn of 1974, as a student, I had a limited access to an IBM-1130 machine while attending a course on Fortran. It was not exactly a top model of punched card technology, but it could be used for many purposes, other than calculating the monthly wages of the University personnel. As a beginner, it took me nearly three months to program and run a simple $1 - \mathrm{e}^{-t/RC}$ response. I had just heard of Moore's law, but I guessed he was exaggerating, since I could do the same job with a slide rule in little less than four hours, so I expected that within my professional lifetime I would never need a computing power that sophisticated. How wrong I was!

In the spring of 1975 my father bought me an HP-29-C, a programmable pocket calculator with 'scientific' functions, sines, cosines, logs, exps, and all that jazz. And in addition to the four stack registers it had some 96 program registers of 'continuous' memory (CMOS, thus the '-C' suffix). Many of my colleagues had similar toys, too, but while most of them were playing the then very popular 'Moon Landing Simulator' (in which you typed in the amount of fuel to burn at each step and in response it displayed your speed and altitude — and a flashing display on crashing), I was busy programming the 0.5 dB tolerance frequency response of an RIAA phonograph correction network, optimized to standard E-12 R and C values. Next year in summer, I made a preamp based on those calculations and inserted it between a simple five transistor power amp and my new Transcriptor's 'Skeleton' turntable with a 'Vestigal' tonearm and a Sonus 'Blue Label' pickup. It worked perfectly and sounded beautiful.

In the late 1970s I was totally devoted to audio; however, I had good relations with the local Motorola representative, who was supplying me with the latest data sheets and application notes, so I simply could not have missed the microprocessor revolution. But I was still using digital chips in the same way as analog ones — by building the hardware, its function was determined once and for all; I never thought of it as something programmable or adaptable to different tasks.

It was only in the early 1980s, with the first PCs, that I really began to devote a substantial part of my working time to programming, and even that was more out of necessity than desire. I was working on the signal sampling section for a digital oscilloscope project, so I had to know all the interactions with the microprocessor. I was also busy drawing printed circuit boards with one of the first CAD programs, the Wintek's 'smArtwork'. Some time later I received a first demo version of the Spectrum-Software's 'MicroCAP' circuit simulator and then the 'PCMatlab' by The MathWorks, Inc.

Then, one day Peter came to my lab and asked me if I could do a little circuit simulation for him. This turned out to be the start of a long friendship and one of the results of it is now in front of you.

Contents:

List of Figures:

List of routines:

6.0 Aim and Motivation

The analysis and synthesis of electronic amplifier–filter systems using symbolic calculus is clearly very labor intensive, as we have seen in the previous parts of the book. The introduction of computers opened up the possibility of solving such problems numerically. Then, of course, the solutions can be approximated only, but it is always possible to trade calculation time for accuracy.

Unfortunately, commercially available computer programs are not well suited to the job from the system designer's point of view. Most of the so called CAD/CAE programs require actual specifications of active devices and passive components before the circuit's simulation can start. The better the program, the more complex are the device models used. Also, these programs will perform circuit analysis only, so the user is left on his own for circuit synthesis and configuration selection, as well as for modeling the influence of parasitic components, which in return depend on the particular circuit topology and devices used.

When designing a system, we usually like to start from some ideal performance goals, so we would prefer to use small and flexible algorithms which would allow us to quickly calculate and compare the required responses of several systems, before we decide what kind of system to use and the technology to realize it. As this book deals with amplifiers, the emphasis will be on the behavior of **linear, time invariant, causal systems (LTIC)**, especially in the time domain.

The algorithms developed are based mostly on analytically derived formulae and procedures which can be easily implemented as numerical routines in any computer language. Some of these formulae have already been presented and discussed in previous parts, but are repeated here in order to put the computer algorithms into an adequate perspective. The algorithms have been written for a special maths oriented programming environment, named Matlab™ [Ref. 6.1, 6.2, 6.3]. Matlab has a large set of pre-programmed mathematical, logical, and graphical functions and allows the user to create macro functions (saved as the '*.M' files) of his own. Over the years Matlab has grown in popularity and has become one of the five most favorite problem solving environments, setting standards in data analysis and control software. Also, its syntax, similar to that of the 'C' programming language, allows easy implementation in embedded systems, so we feel its use here is justified. For the sake of readability we have tried to write the routines and the examples with mainly the basic Matlab instructions, adding as much comment as possible.

(blank page)

6.1 LTIC System Description — A Short Overview

Assume $x(t)$ to be the signal presented at the input of a **linear, time invariant, causal system** (LTIC) which has n dominant energy storing (reactive) components (or briefly a n^{th}-order system). The system output in the time domain, $y(t)$, may be expressed by a **linear differential equation** with constant coefficients:

$$\sum_{i=0}^{n} b_i \, y^{(i)}(t) = \sum_{j=0}^{m} a_j \, x^{(j)}(t) \tag{6.1.1}$$

where the coefficients $\boldsymbol{a_j}$ **and** $\boldsymbol{b_i}$ **are derived from the system's time constants**, whilst $y^{(i)}$ and $x^{(j)}$ are the i^{th} and j^{th} derivatives of the output and input signal, as required by the system's order. From the theory of differential equations we know that the solution of Eq. 6.1.1, given the initial conditions $y(0)$, $y'(0)$, $y''(0)$, …, $y^{(n-1)}(0)$ is of the form:

$$y(t) = \; y_{\text{h}}(t) + y_{\text{f}}(t) \tag{6.1.2}$$

Here $y_{\text{h}}(t)$ is the solution of the **homogeneous** differential equation (in which $x(t)$ and all its derivatives are zero), whilst $y_{\text{f}}(t)$ is the **particular** solution for $x(t)$, which means that $y_{\text{f}}(t) = y_{\text{f}}\{x(t)\}$. From circuit theory we know that $\boldsymbol{y_{\text{h}}(t)}$ **represents the natural (also free, impulse, transient, or relaxation from the initially energized state) response** and $\boldsymbol{y_{\text{f}}(t)}$ **represents the forced (also particular, final, steady state) response.**

Knowing that $y_{\text{f}}(t)$ is a description of the output signal in time very distant from the initial disturbance, when the system has regained a new state of (static or dynamic) balance, we can define the **system transfer function** $\boldsymbol{F(s)}$ from $y_{\text{f}}(t)$ and $x(t)$:

$$F(s) = \frac{y_{\text{f}}(t)}{x(t)} \qquad \text{where} \qquad x(t) = \text{e}^{st} \tag{6.1.3}$$

Such an input signal has been chosen merely because it still retains its exponential form when differentiated, i.e.:

$$x^{(n)}(t) = s^n \text{e}^{st} \tag{6.1.4}$$

For the same reason $y_{\text{f}}(t)$ is expected to be of the form:

$$y_{\text{f}}(t) = A \, \text{e}^{st} \tag{6.1.5}$$

Eq. 6.1.1 may now be rewritten as:

$$(b_n s^n + b_{n-1} s^{n-1} + \cdots + b_1 s + b_0) A \text{e}^{st} = (a_m s^m + a_{m-1} s^{m-1} + \cdots + a_1 s + a_0) \text{e}^{st} \tag{6.1.6}$$

Then A must be:

$$A = \frac{a_m s^m + a_{m-1} s^{m-1} + \cdots + a_1 s + a_0}{b_n s^n + b_{n-1} s^{n-1} + \cdots + b_1 s + b_0} \tag{6.1.7}$$

Returning to Eq. 6.1.3 we can now define the system transfer function as a rational function of s:

$$F(s) = \frac{y_f(t)}{x(t)} = \frac{A\,e^{st}}{e^{st}} = \frac{a_m s^m + a_{m-1} s^{m-1} + \cdots + a_1 s + a_0}{b_n s^n + b_{n-1} s^{n-1} + \cdots + b_1 s + b_0} \tag{6.1.8}$$

Instead of $y_f(t)$, it is much easier in practice to find $F(s)$ first, since the coefficients a_i and b_i are derived from the system's time constants. The system's time domain response is then found from $F(s)$. From algebra we know that ***F(s)* can be expressed also as a function of its poles and zeros.** A n^{th}-order polynomial can be expressed as a product of terms containing its roots r_k:

$$P_n(s) = \sum_{i=0}^{n} a_i s^i = \prod_{k=1}^{n} (s - r_k) \tag{6.1.9}$$

The value of this product is zero whenever s assumes a value of a root r_k. Therefore we can rewrite Eq. 6.1.8 as:

$$F(s) = \frac{(s - z_1)(s - z_2)\ldots(s - z_{m-1})(s - z_m)}{(s - p_1)(s - p_2)\ldots(s - p_{n-1})(s - p_n)} \tag{6.1.10}$$

Here the roots of the polynomial in the numerator are the system's zeros, z_j, and the roots of the polynomial in the denominator are the system's poles, p_i.

We shall have this form in mind whenever a system is specified, because we shall always **start the design by specifying some optimum pole–zero pattern as the design goal** and then work towards the required system's time constants.

The system's time domain equivalent of $F(s)$, labeled $f(t)$, is the system's impulse response:

$$y_h(t) = f(t)\Big|_{x(t)=\delta(t)} \tag{6.1.11}$$

where $\delta(t)$ is the *Dirac*'s function (the infinitesimal time limit of the unit area impulse).

The response to an arbitrary input signal may then be found by convolving the input signal with the system's impulse response (for convolution see Part 1, Sec. 1.15; see also the VCON routine in Part 7, Sec. 7.2).

It is owed to *Oliver Heaviside* (1850–1925, [Ref. 6.4–6.8]), who pioneered the transform theory, that we solve differential equations through the use of the **Laplace transform**[1]**.**

The transform is applied to the time variable t through a single time domain integration, producing a new variable s, whose dimension is t^{-1} (frequency). In the frequency domain the n^{th}-order differential equation is reduced to an n^{th}-order polynomial, whilst the convolution is reduced to simple multiplication. Once solved (using simple algebra), the result is transformed back to the time domain.

[1] Apparently, Heaviside developed his 'operational calculus' in the 1890s independently of Laplace. Although useful and giving results in accordance with practice, his method was considered unorthodox and suspicious for quite a while and only in the mid 1930s it was realized that the theoretical basis for his work could be traced back to Laplace. Interestingly, he also develpoed the method of compensating the dominantly capacitive telegraph lines by inductive peaking, amongst many other things.

Let $F(s)$ represent the Laplace transform of $f(t)$. Then:

$$F(s) = \mathcal{L}\{f(t)\} = \int_{-\infty}^{+\infty} f(t)\, \mathrm{e}^{-st}\, dt \tag{6.1.12}$$

where $\mathcal{L}\{\}$ denotes the Laplace operator as defined by the integral (see Part 1, Sec.1.4).

Actually, the integration is usually made from $t = 0$ and not from $-\infty$, in order to preserve the response's causality (i.e., something happens only after closing the switch). This limitation is caused by the term e^{-st} which for $t < 0$ would not integrate to a finite value unless $f(t) = 0$ for $t \leq 0$. Such restriction is readily accomplished if we modulate the input signal by closing a switch at $t = 0$. Mathematically, this can be expressed by multiplying $f(t)$ by $h(t)$ — the Heaviside's unit step function. In our case this is not necessary, since for calculation of the transient response we consider such input signals which satisfy the convergence condition by definition. Also we shall always assume that the system under investigation was powered up for a time long enough to settle down, so we can safely say that all initial conditions are zero (or an additive constant at worst).

Physically, by multiplying the time domain function by e^{-st} in Eq. 6.1.12 we have canceled the rotation of the phasor e^{st} at that particular frequency (s), allowing the function to integrate to some finite value (see Part 1, Sec.1.2). At other frequencies the phasors will continue to rotate, integrating eventually to zero. By doing so for all frequencies we produce the frequency domain equivalent of $f(t)$. This same process is going on in a sweeping filter spectrum analyzer; the only difference is that in our case an infinitely narrow filter bandwidth is considered. Indeed, such bandwidth takes an infinitely long energy build up time, thus the integration must also last infinitely long and be performed in infinitely small steps.

The **inverse transform** process is defined as:

$$f(t) = \mathcal{L}^{-1}\{F(s)\} = \frac{1}{2\pi j} \int_{\sigma-j\infty}^{\sigma+j\infty} F(s)\, \mathrm{e}^{st}\, ds \tag{6.1.13}$$

where σ is an arbitrarily chosen real valued positive constant for which the inversion solution exists (this restriction is required for functions which do not decay to zero in some finite time and therefore the integral would not converge, e.g., the unit step).

Eq. 6.1.1 can now be written as:

$$y(t) = \mathcal{L}^{-1}\Big\{F(s) \cdot \mathcal{L}\{x(t)\}\Big\} \tag{6.1.14}$$

Note that for transient response calculation, $x(t)$ (the time domain input function), is either the Dirac function **(or $\delta(t)$ — the unity area impulse)** or the Heaviside function **(or $h(t)$ — the unity amplitude step).** In these two cases $\mathcal{L}\{x(t)\} = X(s)$ is either 1 (the transform of the unity area impulse), or $1/s$ (the transform of the unity amplitude step), as we have already seen in Part 1.

Eq. 6.1.14 has been used extensively in previous parts to calculate the transient responses analytically. However, for calculation of the frequency response, we are interested only in that part of the transformed function which is a function of a purely

imaginary s and therefore a special case of $F(s)$, that is $F(j\omega)$. It is thus interesting to examine the possibility of calculating the transient response using the **inverse Fourier transform** (a special case of the inverse Laplace transform) of the system frequency response. We have already seen in Part 1 that the only difference between the Laplace and Fourier transforms is that s is replaced by $j\omega$, which is the same as making σ, the real part of s, equal to zero.

Eq. 6.1.11 shows that the time domain equivalent of the system's frequency response $F(s)$ is $f(t)$ resulting from the excitation by $\delta(t)$, or, in words, the system impulse response. Since the impulse response of any system (except the conditionally stable systems, as well as the oscillating or regenerating systems) decays to zero after some finite time, we do not have to make those special precautions (as in the inverse Laplace transform) to allow the integral to converge, but instead we can use the inverse Fourier transform of $F(j\omega)$ to calculate the impulse response:

$$f(t) = \mathcal{F}^{-1}\{F(j\omega)\} \tag{6.1.15}$$

However, **the Fourier transform of the unity amplitude step does not converge**, so we shall have to use an additional procedure to calculate the step response.

It is possible to put Eq. 6.1.1 into numerical form [Ref. 6.20, 6.21]. Whilst there are ways of using Eq. 6.1.14 in numerical form [Ref. 6.22, 6.23], we shall rather concentrate on Eq. 6.1.15, since the Fast Fourier Transform algorithm (FFT, [Ref. 6.16–6.19]), which we are going to use, offers some very distinct advantages. In addition, we shall develop an algorithm based on the residue theory (Part 1, Sec. 1.9); the details are given in Sec. 6.6.

Another point to consider, known from modern filter theory, is that optimized high order systems are difficult to realize in direct form, because the ratio of the smallest to the largest time constant quickly falls below component tolerances as the system's order is increased. *Butterworth* [Ref. 6.11] has shown that optimum system performance is more easily met by a cascade of low order systems (several of second and only one third, if n is odd) separated by amplifiers. As a bonus such structures will satisfy the gain–bandwidth product requirement more easily. So in practice we shall rarely need to solve high order system equations, usually only at the system integration level.

The formulae presented above will be used as the starting point in algorithm development. We shall develop the algorithms for calculating the system poles for a desired system order, the complex frequency response, the magnitude and phase response, the group (envelope) time delay, the impulse response, the step response, and the numerical convolution. Those algorithms can, of course, be written to solve only our particular class of problems. It is wise, however, to write them to be as universally applicable as possible, in spite of losing some algorithm efficiency, to suit eventual future needs.

6.2 Algorithm Syntax And Terminology

Readers who have not used Matlab or other similar software before will probably have some difficulties in understanding the algorithm syntax and the operations implied. Here is some of the syntax and terminology used throughout Part 6:

`matrix` — An array of data, organized in m rows and n columns.
The operations involving matrices follow the standard matrix calculation rules.

`vector` — A single row or single column matrix; either m or n is equal to 1.

`scalar` — A single element matrix; both m and n are equal to 1.

`size` `length` — Matrix dimension; if `M` is an m by n matrix then `[m,n]=size(M)` returns the number of rows in `m` and the number of columns in `n`. Likewise, `max(size(M))` returns m or n, whichever is greater. For vectors `max(size(V))` is the `length`, or the total number of elements in `V`.

`submatrix` — A smaller matrix contained inside a larger one.
`A=V(k)` is the kth element of the vector `V` (`k` is the index).
`A=V(k)` is the same as `A=V(round(k))` if `k` is non-integer.
`B=M(:,k)` is the kth column of M.
`C=M(j:k,h:i)` is the matrix of hth to ith elements from the jth to kth row.

`+` `-` `*` `/` — Operations involving matrices of 'compatible' dimensions. A scalar can be added to, subtracted from, can multiply or divide a matrix of any dimension. Two matrices can be added or subtracted if they have the same dimensions. Multiplication `A*B` requires that the number of rows in `B` is equal to the number of columns in `A`. Division `A/B` requires an equal number of columns in `A` and `B`.

`.*` `./` — The dot before the operation specifies element by element multiplication and division. The matrix containing the inverse values of the elements in matrix `A` can be calculated as: `1 ./A` (note the **space** between 1 and the dot).

`^` `.^` — Powers: `A.^n` is a matrix with each element of `A` raised to the power of `n`;
`n.^A` is a matrix of `n` to the power of each element of `A`;
`A^n` is possible if `A` is a square matrix (equal number of rows and columns);
`exp(A)` is `e.^A`, where `e=2.71828...`;
`A^B`, if both `A` and `B` are matrices, is an error.

`:` — Colon. Indicates range.
`(1:5)` is a vector `[1,2,3,4,5]`; `V(1:2:N-1)` denotes all odd elements of V.
`A(:)` is all elements of `A` in a single column.

`=` — Equality (right to left assignment of the function or operation result); examples:
`[output arguments]=function(input arguments);` or:
`[out]=[in1]operation[in2];` i.e.: `y=sin(w*t+phi); c=a*b/(a+b);`

`==` — Identically equal (relation); `if x==0, do something; end`

`>` `>=` `<` `<=` — greater, greater or equal, smaller, smaller or equal;

`&` `|` `~` `~=` — logical operators: 'and', 'or', 'not', 'not equal'.

`;` — Semicolon, logical end of command line. For matrices it indicates the end of a row.

`2+3j` — Complex numbers: `2+3j` or `2+j*3` or `2+3*sqrt(-1);`
Most Matlab operations can deal with matrices containing complex elements.

`%` — Characters following `%` are ignored by Matlab. Used for comments.

(blank page)

6.3 Poles and Zeros

It is beyond the scope of this text to cover all the background of system optimization theory. Let us just mention the most important optimization criteria for each major system family. For the same bandwidth and system order n:

• the Butterworth family

is optimal in the sense of having a maximally flat pass band magnitude;

• the Bessel–Thomson family

is optimal in the sense of having a maximally flat group (envelope) delay and, consequently, a maximally steep step response with minimum overshoot;

• the Chebyshev family

is optimal in the sense of a having maximally steep pass band to stop band transition, at the expense of some specified pass band ripple;

• the Inverse Chebyshev family

is optimal in the sense of having a maximally steep stop band to pass band transition, at the expense of some specified stop band ripple;

• the Elliptic (Cauer) family

is optimal in the sense of having a maximally steep pass band to stop band transition, at the expense of some specified pass band and stop band ripple.

It must be pointed out, however, that some system families, the Bessel–Thomson family in particular, can be realized in practice more easily than others, owing to the lower ratio of the largest to the smallest system time constant for a given system order, the maximum usable ratio being limited by component tolerances. Also, low order systems can be realized more easily than high order systems. We shall have to keep these things in mind when denormalizing the system to the actual upper frequency limit and deciding the number of stages used to achieve the total amplification factor.

The trouble is that during the design process we select the system poles and zeros in accordance with certain circuit simplifications, useful for speeding up the analysis. But the implementation of the poles and zeros in an actual amplifier is more a matter of practical know–how, instead of a rigorous theory. This is particularly true if we are pushing the performance to the limits of realizability, since in these conditions the component stray reactances must be taken into account when specifying the system time constants. Luckily for the amplifier designer, for the same component and layout strays the Bessel–Thomson system yields the highest system bandwidth, with the bonus of an optimal transient response. This is also true for ‘feedback stabilized’ systems, since the large phase margin offered by this system family aids system stability; also, feedback induced Q-factor enhancement at high frequencies lowers the required imaginary versus real part ratio of the response shaping component impedances even further.

In this text we shall only deal with the Butterworth [Ref. 6.11] and Bessel–Thomson [Ref. 6.12, 6.13] low pass systems for calculation of poles, since these are required in wideband amplifier design. If needed, Chebyshev, inverse Chebyshev and elliptic (Cauer) functions are provided in the Matlab Signal Processing Toolbox, as well as the low pass to high pass, band pass and band stop transform algorithms. The toolbox also contains many other useful algorithms, such as RESIDUE, ROOTS, etc., which will not be considered here (see [Ref. 6.1, 6.2, 6.19]).

In order to be able to compare the performance of different systems on a fair basis we must specify some form of **system standardization:**

a) **all systems will have the pole values normalized for an upper half power angular frequency ω_h of 1 radian per second (equivalent to the cycle frequency of $f_h = 1/2\pi$ [Hz]). This leads to the use of a normalized frequency vector, implying that whenever we write either f or ω, we shall actually mean f/f_h or ω/ω_h, respectively.**

Please note that this can sometimes cause a bit of confusion, since f/f_h is the same as ω/ω_h; but $\omega = 2\pi f$, so we should keep an eye on the factor 2π, especially when denormalizing the poles to the actual system upper half power frequency.

The frequency response is calculated as a function of ω, since the poles and zeros are mapped in terms of $s = \sigma + j\omega$, where both the real and imaginary part are measured in [rad/s], but we usually plot it as a function of f (in [Hz]). If the values of the poles are normalized we can use the same normalized frequency vector to calculate and plot the frequency domain functions. Therefore to plot the magnitude and phase responses vs. frequency we shall not have to divide the frequency vector (of a length usually between 100 and 1000 elements) by 2π.

Since Matlab will not accept the symbol ω as a valid name for a variable we shall replace it by `w=2*pi*f` in our routines.

b) **all systems will have their DC gain (at $\omega = 0$) normalized to $A_0 = 1$** (throughout this text we shall consider low pass systems only). Nevertheless, we shall try to provide the correct gain treatment in the general case, in order to broaden the applicability of our algorithms.

Of course, to extract the actual system component values, as well as to scale the various frequency and time domain responses to comply with the desired upper frequency f_h, the poles and zeros will have to be **denormalized** (multiplied) by $2\pi f_h$. Also, each response will have to be scaled by the required gain factor.

I.e., for a simple current driven shunt RC system, the normalized pole, $s_{1n} = 1/(R_{1n}C_{1n}) = 1$, is first denormalized to the value of the desired bandwidth, $s_1 = s_{1n} \cdot 2\pi f_h$. From s_1 we get the new component values, $R_1C_1 = 1/(2\pi f_h)$. Finally, we multiply R_1 by the gain factor, $R = AR_1$, to obtain the desired output voltage from the available input current, that is $v_o = Ri_i$, and then reduce C_1 by the same amount, $C = C_1/A$. If C is a stray capacitance it cannot be reduced below the limit imposed by the circuit topology. Then we must work backwards by first finding the R which would give the desired bandwidth, and then determine the input current which will give the required output voltage.

6.3.1 Butterworth Systems

Butterworth systems ([Ref. 6.11], see also Part 4, Sec. 4.3) are optimal in the sense that all the derivatives of the frequency response are zero at the complex plane origin, resulting in a maximally flat magnitude response. The normalized squared magnitude response of an n^{th}-order Butterworth system is:

$$F^2(\omega) = \frac{1}{1 + (\omega^2)^n} \tag{6.3.1}$$

This can be rewritten as:

$$F(s)\,F(-s) = \frac{1}{1 + (-s^2)^n} \tag{6.3.2}$$

This is an all-pole system, since $F^2(\omega) \to \infty$ whenever:

$$1 + (\omega^2)^n = 0 \tag{6.3.3}$$

The roots of Eq. 6.3.3 are:

$$\omega = (-1)^{1/2n} \tag{6.3.4}$$

This can be solved using DeMoivre form:

$$(-1)^{1/2n} = \cos\frac{\pi + k\,2\pi}{2n} + j\sin\frac{\pi + k\,2\pi}{2n} \tag{6.3.5}$$

where $k = 0, 1, 2, \cdots, n-1$.

If, owing to the ***Hurwitz* stability requirement**, we associate the poles in the left half of the complex plane with $F(s)$, then:

$$F(s) = \frac{k_0}{\prod_{k=1}^{n}(s - s_k)} \tag{6.3.6}$$

where s_k are found from the expression of Eq. 6.3.5 in the exponential form:

$$s_k = \mathrm{e}^{j\pi\frac{1+\frac{2k-1}{n}}{2}} \qquad \text{for } k = 1, 2, 3, \ldots, n \tag{6.3.7}$$

and:

$$k_0 = \prod_{k=1}^{n}(-s_k) \tag{6.3.8}$$

In the general (non-normalized) case, $\omega_{\text{h}} = \sqrt[n]{k_0}$.

A Butterworth system is completely specified by the system order n. It is normalized so that it has its half power bandwidth limit at the unit frequency:

$$F(j)\,F(-j) = F^2(1) = \frac{1}{2} \quad \Rightarrow \quad \left|F(1)\right| = \frac{1}{\sqrt{2}} \tag{6.3.9}$$

Eq.6.3.7 and Eq.6.3.8 are implemented in the Matlab Signal Processing Toolbox function called BUTTAP (an acronym for BUTTerworth Analog Prototype):

```
function [z,p,k] = buttap(n)
%BUTTAP     Butterworth analog low pass filter prototype.
%      [z,p,k] = buttap(n) returns the zeros, poles, and gain
%      for the n-th order normalized prototype Butterworth analog
%      low pass filter. The resulting filter has n poles on the
%      unit circle in the left half plane, and no zeros.
%
%      See also BUTTER, CHEB1AP, and CHEB2AP.

%      J.N. Little and J.O. Smith 1-14-87
%      Revised 1-13-88 LS
%      (c) Copyright 1987, 1988,  by The MathWorks, Inc.

% Poles are on the unit circle in the left-half plane.
z = [];
p = exp(sqrt(-1)*(pi*(1:2:2*n-1)/(2*n) + pi/2)).';
k = real(prod(-p));
```

As an example see the complex plane layout of the poles of a 5th-order Butterworth system in Fig. 6.3.1.

For a desired attenuation $a = 1/A$ at some chosen $\omega_a > \omega_\mathrm{h}$ we can calculate the required system order:

$$n = \frac{\log_{10}(A^2 - 1)}{2 \log_{10} \dfrac{\omega_a}{\omega_\mathrm{h}}} \tag{6.3.10}$$

and round it to the first higher integer.

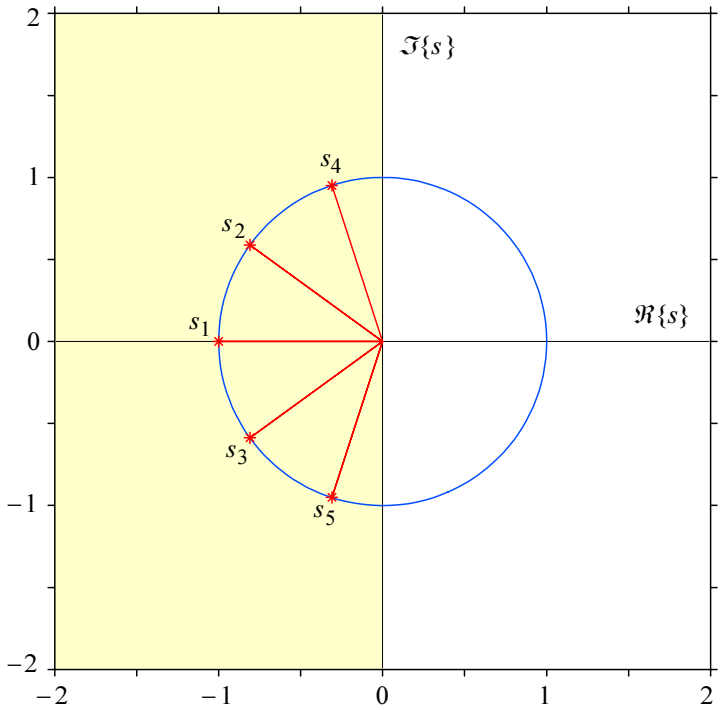


Fig. 6.3.1: The 5th-order Butterworth system poles in Cartesian coordinates of the complex plane. The left half of the figure is the s domain over which the magnitude in Fig. 6.4.1 is plotted.

6.3.2 Bessel–Thomson Systems

Bessel–Thomson systems (see [Ref. 6.12, 6.13]; see also Part 4, Sec. 4.4) are optimal in the sense that all the derivatives of the group (envelope) time delay response are zero at origin, which results in a maximally flat group delay. This means that all the relevant frequencies pass through the system with equal time delay, resulting in a transient response with a minimal overshoot. In the complex frequency plane a system with pure time delay is represented by:

$$F(s) = \mathrm{e}^{-sT} \tag{6.3.11}$$

We first normalize this by making $T = 1$. Then we expand e^{-s} as a polynomial. However, if this is done using the *Taylor* series expression for e^{x} (and if the polynomial degree exceeds 4), the resulting polynomial would not meet the ***Hurwitz* stability criterion,** because some of the poles would be in the right half of the complex plane. But there is another expression for e^{-s} which we can use:

$$\mathrm{e}^{-s} = \frac{1}{\sinh s + \cosh s} = \frac{\dfrac{1}{\sinh s}}{1 + \dfrac{\cosh s}{\sinh s}} \tag{6.3.12}$$

The Taylor series for hyperbolic sine function has even powers of s and the hyperbolic cosine has odd powers of s. When we divide these polynomials (using long division) the poles of the resulting polynomial meet the stability criterion. If we express this as a partial fraction expansion, truncated at the n^{th} fraction, an n^{th}-order Bessel–Thomson system results. This can be expressed as:

$$F(s) = \frac{c_0}{B_n(s)} \tag{6.3.13}$$

where $B_n(s)$ is an n^{th}-order Bessel polynomial:

$$B_n(s) = \sum_{k=0}^{n} c_k \, s^k \tag{6.3.14}$$

and each $B_n(s)$ satisfies one of the following relations:

$$\begin{aligned} B_0(s) &= 1 \\ B_1(s) &= s + 1 \\ B_n(s) &= (2n-1)\, B_{n-1}(s) + s^2\, B_{n-2}(s) \end{aligned} \tag{6.3.15}$$

The coefficients c_k of the resulting polynomial can be calculated as:

$$c_k = \frac{(2n-k)!}{2^{(n-k)}\, k!\, (n-k)!} \qquad \forall \;\; k = 0, 1, 2, \ldots, n-1, n \tag{6.3.16}$$

The function, which will calculate the Bessel polynomial coefficients using Eq. 6.3.16 will be called BESTAP (this stands for BESsel–Thomson Analog Prototype, but the name is also in good agreement with the best time domain response of this system family). Within this function the system poles are extracted using the ROOTS

function in Matlab. This works well up to $n = 24$; for higher orders the ratio of c_n to c_0 is so high that the computer numerical resolution (‘double precision’ or 16 significant digits) is exceeded, but this is not a severe limitation because in most circuit configurations the 1% component tolerances will limit system realizability to about $n = 13$ (assuming a 6-stage system, for which the highest reactive component value ratio is about 12:1). But if needed, we can always calculate the frequency response from the polynomial expression, using the coefficients c_k directly in Eq. 6.1.8, instead of using Eq. 6.1.10, as in the Matlab POLYVAL and FREQS routines.

Bessel–Thomson system poles are found in the left half of the complex plane on a family of ellipses, having a nearer focus at the complex plane origin and the other focus on the positive part of the real axis (see Fig. 6.3.2**)**.

The poles calculated in this way define a family of systems with equal envelope delay (normalized to 1 s). This results in a progressively larger bandwidth and smaller rise time for each higher n (see Fig. 6.5.11). In addition, two other normalizations of the Bessel–Thomson system are possible.

One is to make the asymptote of the magnitude roll off slope the same as it is for the Butterworth system of equal order (this is useful for calculating transitional Bessel to Butterworth systems, as we have seen in Part 4, Sec. 4.5.3). If ω_a is to become the half power cut off frequency of the new system:

$$\left|F(\omega_a)\right| = \frac{c_0}{2\,\omega_a^n} = \frac{1}{2} \qquad \Rightarrow \qquad \omega_a = c_0^{1/n} \tag{6.3.17}$$

In this case, with the roots of $B_n(s)$ divided by $c_0^{1/n}$, the envelope delay will be equal to $c_0^{1/n}$, instead of 1, and the system bandwidth will be smaller for each higher n.

The other is to have equal bandwidth for any n, possibly normalized to 1 rad/s, as is the Butterworth family; in this way we would be able to compare different systems on a fair basis. Unfortunately there is no simple way of matching the Bessel–Thomson system bandwidth to that of a Butterworth system of the same order. To achieve this we have to recursively multiply the poles by a correction factor proportional to the bandwidth ratio, until a satisfying approximation is reached (the values of poles modified in such a way for systems of order 2 to 10 are shown in Part 4, Table 4.5.1). The `while` loop at the end of the BESTAP routine has a tolerance of 0.0001 and it was experimentally found to match in only 8 to 12 loop iterations, depending on n; this tolerance is satisfactory for most practical purposes, but the reader can easily change it to suit his needs.

All these three normalization options (group delay, asymptote, and bandwidth) are being provided for by the BESTAP routine by entering, besides the system order n, an additional input argument in the form of a single character string:

`'n'` for 1 rad/s cutoff frequency normalization (the default),

`'t'` for unit time delay and

`'a'` for the same attenuation slope asymptote as Butterworth system of equal order.

As in the BUTTAP routine, three output variables are returned. But the number of arguments returned by BESTAP can be either 3, 2, or just 1. If all three output arguments are requested, the zeros are returned in `z`, the poles in `p`, and the non-

normalized system gain is returned in the output variable `k`. Since there are no zeros in this family of systems, an empty matrix is returned in `z`.

With just two output arguments, only `z` and `p` are returned.

When only one output argument is specified, instead of having an empty matrix returned in `z`, which would not be very useful, we have decided to return the Bessel polynomial coefficients c_k. Note that for the n^{th}-order system there are $n+1$ coefficients, from c_n to c_0. The system gain normalization is achieved by dividing the each coefficient by c_0, that is, the last one in the vector `c`, i.e., `c=c/c(n+1)`. The coefficients are scaled as for the `'t'` option (equal envelope delay); other options are then ignored. But, if necessary, we can always calculate the polynomial coefficients for those cases from the poles, by invoking the POLY routine, i.e., `c=poly(p)`.

```
function [z,p,k]=bestap(n,x)
%BESTAP  BESsel-Thomson Analog Prototype.
%        Returns the zeros z, poles p and gain k of the n-th order
%        Bessel-Thomson system. This is an all-pole system, so an
%        empty matrix is returned in z. The poles are calculated for
%        a maximally flat envelope (group) delay.
%
%        Call  : [z,p,k]=bestap(n,x);
%        where :
%                n is the system order
%                x is a single-character string, making the poles:
%                  'n' - normalized to a cutoff of 1 rad/s (default);
%                  'a' - normalized to have the same attenuation
%                        asymptote as a Butterworth system of same n;
%                  't' - scaled for a group-delay of 1s.
%                k is the non-normalized system DC gain.
%                p are the poles (length-n column vector)
%                z are the zeros (no zeros, empty matrix returned)
%        With only one output argument :
%        c=bestap(n);
%        the n+1 coefficients of the system polynomial are returned,
%        scaled as in the 't' option, ignoring other options.

% Author :  Erik Margan, 881012,    Free of copyright !

if nargin == 1          % nargin is the number of input arguments
      x='n';            % by default, normalize to 1 rad/s cutoff
end
z=[];                   % no zeros
if n == 1
      if nargout == 1
            c=[1, 1];   % first-order system coefficients
      else
            p=-1;       % first-order pole
            k=1;        % gain
            return      % end execution of this routine
      end
else
                        % find the Bessel polynomial coefficients
                        % from factorials :
                        % 0!=1 by definition, the rest is calculated
                        % by CUMPROD (CUMulative PRODuct)
      fact=[1, cumprod(1:1:2*n)];
      binp=2 .^(0:1:n));      % a vector of binary powers
      c=fact(n+1:1:2*n+1)./(binp.*fact(1:1:n+1).*fact(n+1:-1:1));
                        % c is a vector of polynomial coefficients,
                        % c(1) is at s^n, c(n+1) is at s^0
end
```

```
if nargout == 1          % nargout is the number of output arguments
      z=c;               % the coefficients of the Bessel polynomial
      return             % end execution of this routine
end

c=c/c(n+1);              % Normalize system gain to 1 at DC

if x == 'a' | x == 'A'         % | means logical OR
                               % Normalize to Butterworth asymptote
      g=c(1) .^((n:-1:0)/n);   % c(1) is the coefficient at s^n
      c=c./g;                  % Normalize gain
end

p=roots(c);                    % ROOTS extracts poles from coefficients

if x == 'n' | x == 'N'
            % Bandwidth normalization to 1 rad/s results in
            % progressively greater envelope delay for increasing n
      P=p;  % copy the poles to P
            % Reference (-3 dB point)
      y3=1/sqrt(2);
      y=abs(freqw(P,1));       % attenuation at 1 rad/s (see FREQW)
      while abs( 1 - y3/y ) > 0.0001
            P=P*(y3/y);        % Make iterative corrections
            y=abs(freqw(P,1));
      end
      p=P;  % copy P back to p
end

k=real(prod(-p));              % non-normalized system gain
```

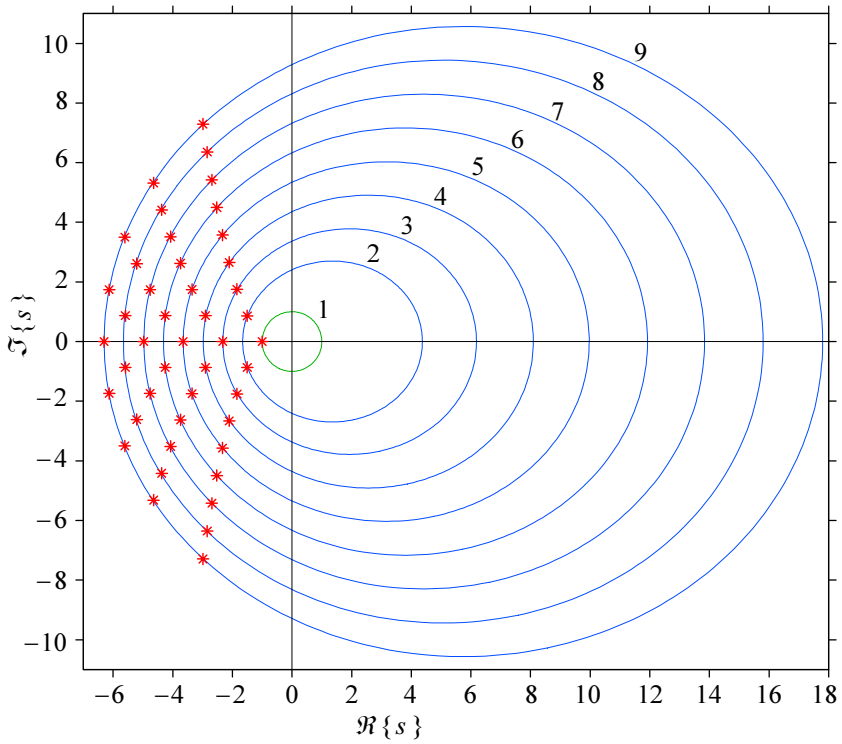


Fig. 6.3.2: Complex plane pole map for unit group delay Bessel–Thomson systems of order 2–9 (with the first-order reference).

6.4 Complex Frequency Response

In Matlab the frequency response is calculated by two routines, called POLY and FREQS, which require the polynomial coefficients as the input argument. We shall use two different routines: PATS, which calculates the product of terms containing polynomial roots at each value of s, according to Eq. 6.1.9 and FREQW, which returns the complex frequency response $F(s)$, after Eq. 6.1.10, given the zeros, poles and the normalized frequency vector. FREQW calls PATS as a subroutine.

```
function P=pats(R,s)
% PATS   Polynomial (product form) value AT S.
%        P=pats(R,s) returns the values of the product of terms,
%        containing n-th order polynomial roots R=[r1, r2,.., rn],
%        for each element of s.
%        Values are calculated according to the formula :
%
%     P(s)=(s-r1)*(s-r2)*...*(s-rn) / (((-1)^n)*(r1*r2*...*rn))
%
%        and are normalized so that P(0)=1.
%     PATS is used by FREQW. See also POLY and FREQS.

% Author :  Erik Margan, 881110, Free of copyright !

[m,n]=size(s);
P=ones(m,n);            % A matrix of all ones, same dimension as s
nr=max(size(R));        % number of elements in R
for k=1:nr
      if R(k) == 0
            P=P.*s;     % Multiply, but prevent from dividing by 0.
      else
            P=P.*(s-R(k))/(-R(k));
      end
end
```

```
function F=freqw(z,p,w)
% FREQW  returns the complex frequency response F(jw) of the system
%        described by the zeros (vector z=[z1,z2,...,zm]) and the
%        poles (vector p=[p1,p2,...,pn]).
%        Call : F=freqw(z,p,w);
%        w is the frequency vector; can be real, imaginary or complex.
%        F=freqw(p,w) assumes a system with poles only.
%        FREQW uses PATS. See also FREQS and FREQZ.

% Author :  Erik Margan, 881110, Free of copyright !

if nargin == 2          % nargin returns the number of input arguments
      w=p; p=z; z=[];   % assume a system with poles only
end
for k=1:max(size(p))
      if real(p(k)) >= 0
            disp('WARNING : This is not a Hurwitz-type system!')
      end
end
if ~any(imag(w))
      w=sqrt(-1)*w;     % if w is real, assume it to be imaginary
end
if isempty(z)
      F=1 ./pats( p, w ) ;
else
      F=pats( z, w )./pats( p, w ) ;
end
```

6.4.1 Frequency Dependent Response Magnitude

The absolute value of the complex frequency response is called **magnitude**; it is calculated as a square root of the product of $F(s)$ with its own complex conjugate:

$$|F(s)| = \sqrt{F(s)\,F^*(s)} = \sqrt{\Big(\Re\{F(s)\} + j\Im\{F(s)\}\Big)\Big(\Re\{F(s)\} - j\Im\{F(s)\}\Big)}$$

$$= \sqrt{\Big(\Re\{F(s)\}\Big)^2 + \Big(\Im\{F(s)\}\Big)^2} = M(s) \qquad (6.4.1)$$

Assuming a sinusoidal input signal, the magnitude represents the output to input ratio of the peak signal value at that particular frequency. In practice, when we talk about the system's 'frequency response', we usually mean 'the frequency dependent magnitude', $M(\omega)$. The magnitude contains no phase information.

We can calculate the magnitude by any of the following Matlab basic functions:

```
M=sqrt( (real(F)).^2 + (imag(F)).^2 );    % or :
M=sqrt( F .* conj(F) );                   % or :
M=sqrt( F .* ( F' ) );                    % or :
M=abs(F);                                 % abs --> absolute value
```

We shall use the ABS command, not just because it is easy to type in, but because it executes much faster when there is a large amount of data to process.

In order to acquire a better understanding of what we are doing, let us write an example for a 5th-order Butterworth system. In the Matlab command window we write:

```
[z,p]=buttap(5);↵      % Note: a command line is executed by "ENTER"
```

If we now type:

```
z ↵        answer:          []

p ↵        answer:          -0.3090 + 0.9511i
                            -0.3090 - 0.9511i
                            -0.8090 + 0.5878i
                            -0.8090 - 0.5878i
                            -1.0000 + 0.0000i
```

Since there are no zeros an empty matrix (shown by square brackets) is returned in `z`. A 5-element column vector with complex conjugate pole values is returned in `p`. Let us plot these poles in the complex plane using Cartesian coordinates:

```
plot( real(p), imag(p), '*' ), axis([-2,2,-2,2]); ↵
                                  % see the result in Fig.6.3.1.
```

and the result would look as in Fig. 6.3.1 (for clarity, the distance from the origin and the unit circle are also shown there, both needing extra 'plot' operations, not written in the example above). From now on we shall not write the ENTER character explicitly.

The system magnitude as a function of the complex frequency s has a very interesting 3D shape and it is instructive to have a closer look at it:

```
[z,p]=buttap(5);                % 5th-order Butterworth poles
r=(-2:1/20:0);                  % real frequency vector
w=(-2:1/20:2);                  % imaginary frequency vector
[x,y]=meshgrid(r,w);            % make the complex domain grid
M=abs(1 ./pats(p,x+j*y));       % magnitude due to poles in x+j*y domain
for m=1:max(size(M))
     n=find(M(m,:)>12 );        % find magnitude > 12
     M(m,n)=12;                 % limit magnitude to 12 for plot
end
waterfall(x',y',M')             % waterfall plot of magnitude in 3-D
                                % prime(') aligns plots along jw-axis
axis([-2,0,-2,2,0,12])          % set axes limits
view(50,25)                     % view(azimuth,elevation) set view angle
                                % add axes labels (Matlab-V format):
xlabel( '{\it\sigma} = \Re\{{\its}\}', 'FontSize', 10 )
ylabel( '{\itj\omega} = {\itj}\Im\{{\its}\}', 'FontSize', 10 )
zlabel( '{\itM}({\its}) = |{\itF}({\its})|', 'FontSize', 10 )
                                % see the result in Fig.6.4.1.
```

Fig. 6.4.1 has been created using the Matlab WATERFALL function and shows the 3D magnitude of the 5th-order Butterworth system over a limited s domain in the complex plane. The s domain here is the same as the left half of Fig. 6.3.1. Over the poles, the magnitude would extend to infinity, so we have had to limit the height of the plot in order to show the low level features in more detail.

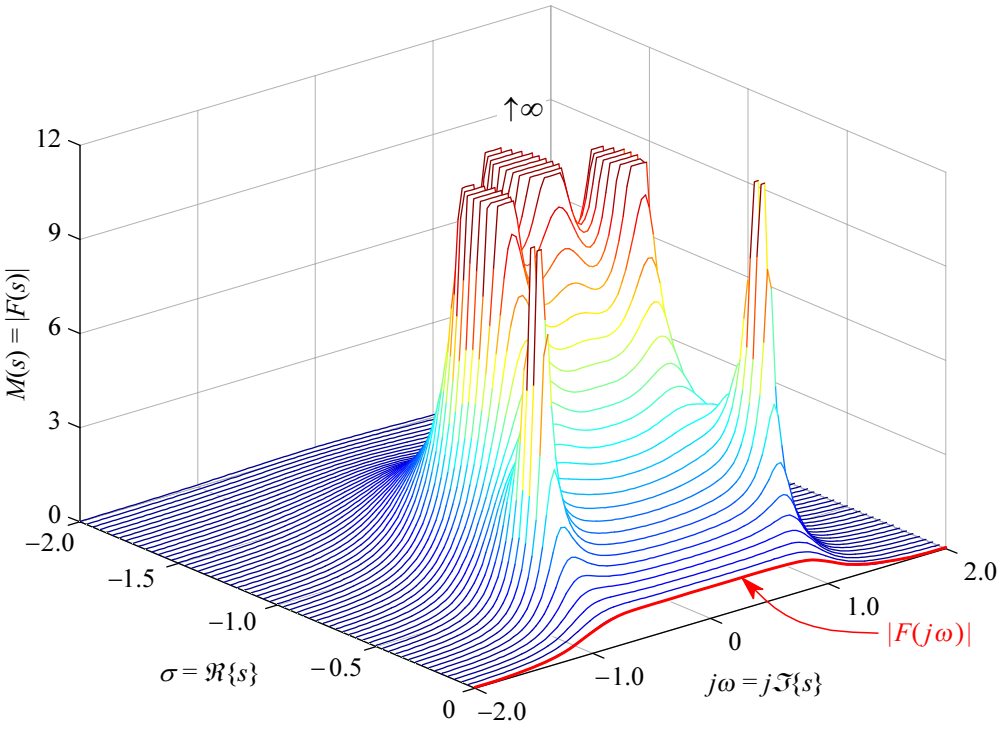


Fig. 6.4.1: The 5th-order Butterworth system magnitude, plotted over the same s-domain as the shaded left half of Fig. 6.3.1. The surface represents $|F(s)|$, but limited in height in order to reveal the low level details. Its shape above the $j\omega$ axis is $|F(j\omega)| = M(\omega)$.

Now, we have intentionally limited the s domain to just the left half of the complex plane (where the real part is either zero or negative). This highlights the shape of the plot along the imaginary axis, which is — guess what ? — $M(\omega)$.

Looking at those lines parallel to the imaginary axis we can see what would happen if the poles were moved closer to that axis: the magnitude would exhibit a progressively pronounced peak. Such is the consequence of lowering the real part of the poles. Since the negative real part is associated with energy dissipative (resistive) components, it is clear that its role is to suppress resonance. But when we design an oscillator we need to compensate any energy lost in the parasitic resistances of the reactive components by an active regeneration ('negative resistance' or a positive real part) in order to set the system poles (usually just one pair for oscillators) exactly on the imaginary axis.

What is interesting to note is the mirror like symmetry about the real axis, owed to the complex conjugate nature of the Laplace space. Here we see at work the concept of 'negative frequency', which will be discussed latter in Sec. 6.5, dealing with the Fourier transform inversion. This symmetry property will allow us to greatly improve the inverse transform algorithm efficiency.

It is also instructive to see the complex frequency response $F(j\omega)$ in 3D:

```
w=(-3:0.01:3);          % 601 frequencies, -3 to 3, in 0.01 increment
F=freqw(z,p,w);         % 601 points of complex frequency response
plot3(w,real(F),imag(F))% 3D plot of the Im and Re part of F(jw)
view(65,15);            % view angle, azimuth 65deg., elevation 15deg.
                        % see the result in Fig.6.4.2.
```

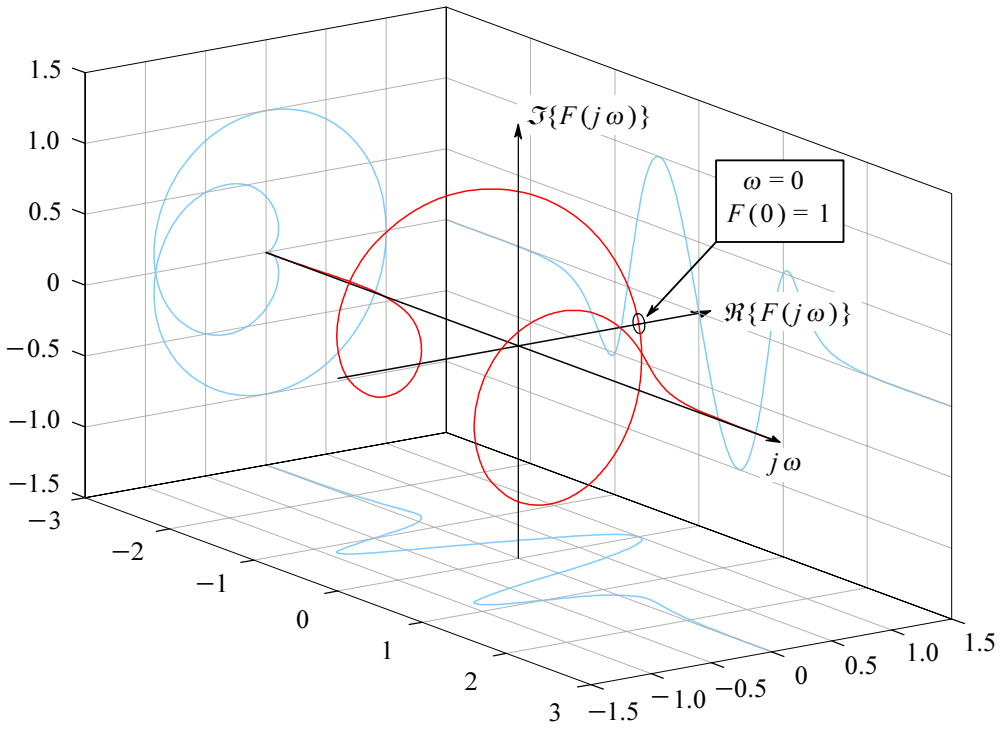


Fig. 6.4.2: The complex 3D plot of $F(j\omega)$. The response phasor rotates clockwise, going from negative to positive frequency. The distance from the frequency axis is the magnitude. The circle on the real axis marks the DC response point. The Nyquist plot (see Fig. 6.4.3) usually shows only the $\omega > 0$ part, viewing in the $-j\omega$ direction. The three projections are plotted to help those readers who do not have access to Matlab to visualize the shape.

Fig. 6.4.2, which has been created using the Matlab PLOT3 function, shows $F(j\omega)$ with the phase angle twisting about the $j\omega$ axis and the magnitude as the distance from the $j\omega$ axis. The circle marker denotes the point where $F(j\omega)$ crosses the real axis at zero frequency — the DC system gain normalized to 1.

Whilst the Fig. 6.4.1 waterfall plot shape was relatively easy to interpret and 'feel', the 3D curve shape is somewhat less clear. In Matlab one can use the `view(azimuth,elevation)` command to see the graph from different viewing angles. In Fig. 6.4.2, `view(65,15)` was used. In more recent versions of Matlab the user can even select the viewing point by the 'mouse'. To help the imagination of readers without access to Matlab we have also plotted the three 'shadows'.

Regarding the symmetry, $F(j\omega)$ is not a mirror image of $F(-j\omega)$ — unlike $M(\omega)$ — because the phasor preserves its sense of rotation (clockwise, negative by definition for any system with poles on the left) throughout the $j\omega$ axis. But if folded about the real axis the shape would match.

As a result of such symmetry $F(j\omega)$ can be plotted using only the $\omega \geq 0$ part of the axis, without any loss of information. The *Nyquist* plot [Ref. 6.9] shows both the magnitude and the phase angle on the same graph:

```
w=(0:0.01:3);           % 301 frequencies, 0 to 3, increment 0.01
F=freqw(z,p,w);         % 301 points of complex frequency response
axis('square');         % the plot axes will have a 1:1 aspect ratio
plot(real(F),imag(F))   % plot the imaginary part versus real part
```

The result should look like Fig. 6.4.3. The view is as if we look at the Fig. 6.4.2 in the opposite direction of the $j\omega$ axis (from $+\infty$ towards the origin).

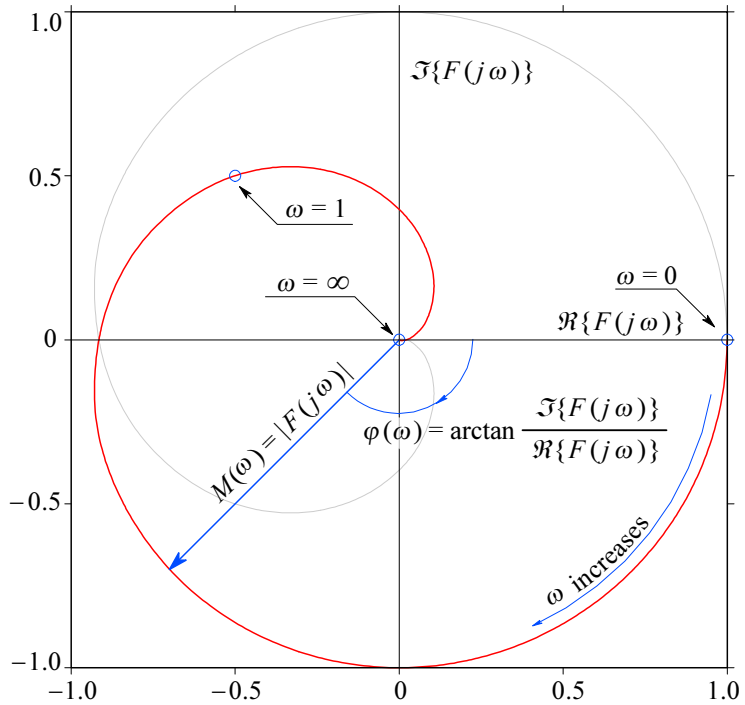


Fig. 6.4.3: The Nyquist plot of the 5th-order Butterworth system frequency response. The frequency axis is reduced to a single point projection at the origin and is parametrically incremented with the phase angle, from the DC point on the real axis, to the half power bandwidth point at `[-0.5, 0.5*j]` and to infinity at the origin.

However, in a Nyquist plot it is difficult to see the frequency dependence of the magnitude and phase, unless we intentionally mark a few chosen frequencies on the plot, as was done in Fig. 6.4.3 in order to make the orientation easier. Thus it has become a standard practice to make separate plots of magnitude and phase as functions of frequency, as introduced by *Bode* [Ref. 6.10]. Again, exploiting the symmetry, we can plot only the $\omega \geq 0$ part without loosing any information:

```
                        % following the previous example:
M=abs(F);               % 301 points of magnitude
plot(w,M)               % display M versus w, see Fig.6.4.4.
```

This should look like Fig. 6.4.4. The special point on the graph is the magnitude at the unit frequency — its value is $1/\sqrt{2}$, or 0.707, and since power is proportional to the magnitude squared this is the system's half power cut off frequency.

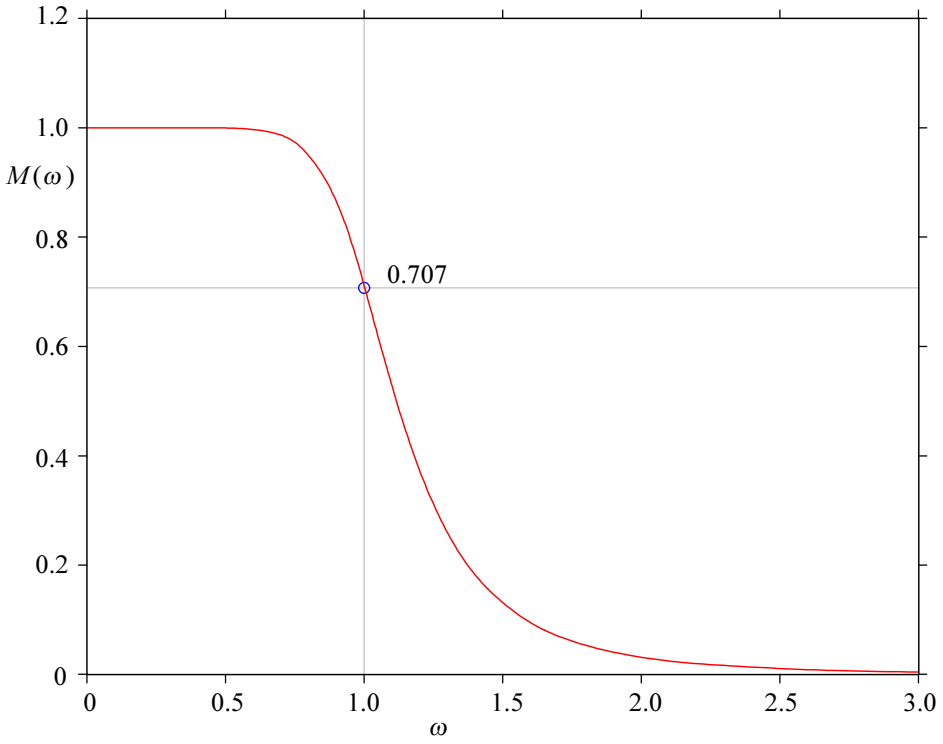


Fig. 6.4.4: The magnitude vs. frequency plot in a linear scale. The characteristic point is the half power bandwidth.

It has also become a standard practice to enhance the stop band detail by using either the $\log M$ vs. $\log \omega$ or the semilog dB(M) vs. $\log \omega$ plot scale (Fig. 6.4.5):

```
w=logspace(-1,1,301);   % frequency, 301 points, equally spaced in
                        % log-scale from 0.1 to 10
F=freqw(z,p,w);         % 301 points of complex frequency response
M=abs(F);               % 301 points of magnitude
semilogx(w,20*log10(M))% display M in dB versus log-scaled w
                        % see the result in Fig.6.4.5.
```

By using the log-log scale or a linear dB vs. log frequency scale we can quickly estimate the system order, since the slope is simply (for all pole systems) n times the first-order system slope (n×20 dB per frequency decade).

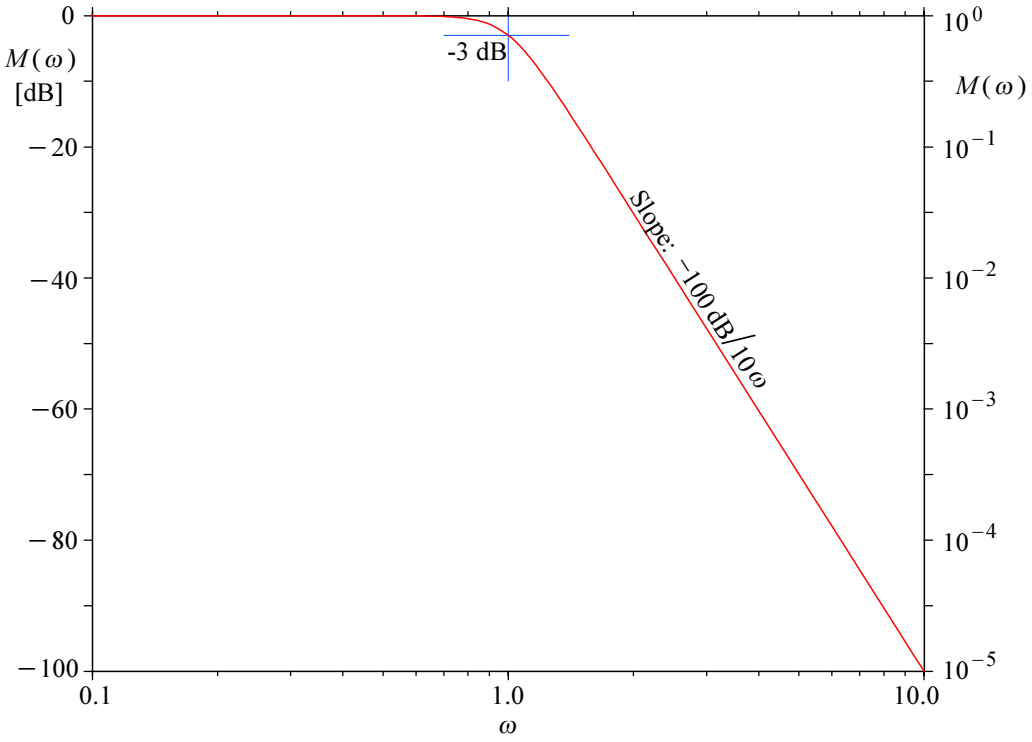


Fig. 6.4.5: Bode plot of the 5th-order Butterworth system magnitude, as in Fig. 6.4.4, but in a linear dB vs. log frequency scale. In such a scale, all-pole systems have an asymptotically linear attenuation slope, proportional to the system order (a factor of 10^n or n×20 dB/10ω). The marked −3dB reference point is the same half power cut off frequency point as in Fig. 6.4.4.

6.4.2 Frequency Dependent Phase Shift

The **phase** response is calculated from the complex frequency response as the arctangent of the ratio of the imaginary versus real part of $F(s)$:

$$\varphi(\omega) = \arctan \frac{\Im\{F(s)\}}{\Re\{F(s)\}} \tag{6.4.2}$$

Using the previously calculated $F(j\omega)$ we can write this as:

```
phi=atan(imag(F)./real(F));          % 301 samples of phase of F(jw)
```

Note that the Matlab arctangent function is called `atan`. However, Matlab also has a built in command named ANGLE, using the same Eq. 6.4.2, so:

```
phi=angle(F);                 % phase response, modulo 2pi ;
semilogx(w,phi);              % show phi in radians vs. log-scaled w ;
```

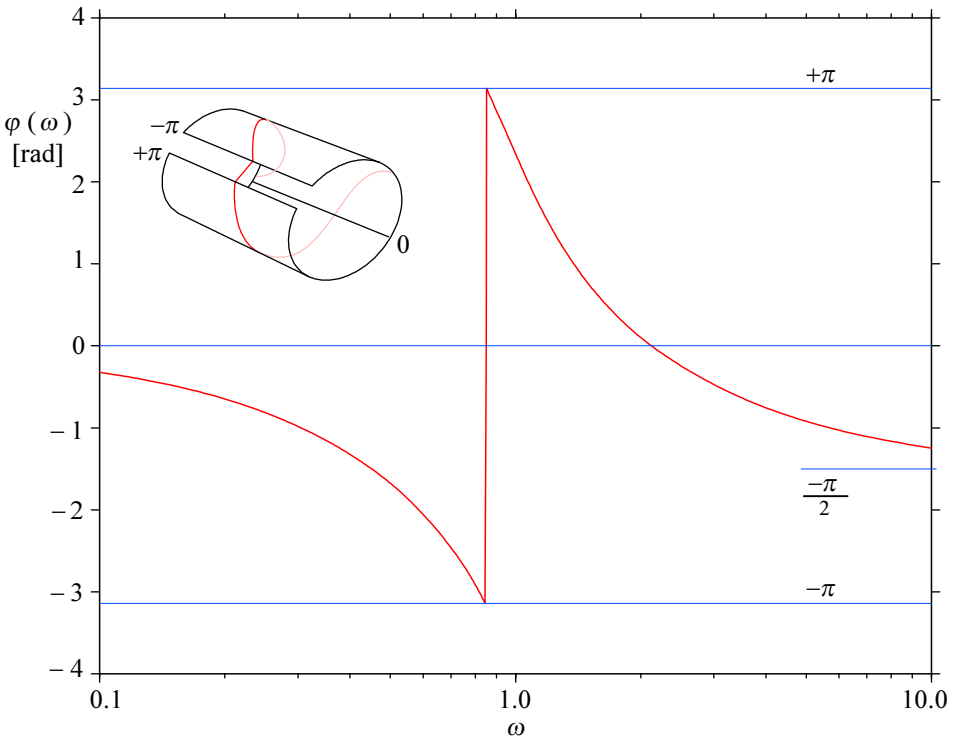


Fig. 6.4.6: The phase angle vs. frequency plot of the 5th-order Butterworth system. The circularity of trigonometric functions, defined within the range $\pm\pi$ radians, is the cause of the discontinuous phase vs. frequency relationship.

Clearly this is a circular function of modulo 2π radians. For systems of 3rd or greater order the phase will rotate by more than 2π, so there will be jumps from $-\pi$ to $+\pi$ in the phase graph, as in Fig. 6.4.6, and we must 'unwrap' it (roll the 'cylinder' along the φ axis) in order to get a continuous function. Matlab has the ADDTWOPI (older) and UNWRAP (newer) routine, but both perform irregularly for $\varphi > 4\pi$, especially for systems with zeros. The following EPHD routine works correctly.

```
function q=ephd(phi)
% EPHD      Eliminate PHase Discontinuities.
%     Outperforms UNWRAP and ADDTWOPI for systems with zeros.
%     Use :       q=ephd(phi);
%     where :
%     phi --> input phase vector in radians ( range: -pi>=phi>=pi );
%     q   --> output phase vector, "unwrapped";
%     If phi is a matrix, unwrapping is performed down each column.

% Author :  Erik Margan, 890505, Free of copyright !

[r,c]=size(phi);
if min(r,c) == 1
      phi=phi(:);       % column-wise orientation
      c=1;
end
q=diff(phi);            % differentiate to detect discontinuities
      % compensate for one element lost in diff and round the steps :
q=[zeros(1:c); pi*round(q/pi)];
q=cumsum(q);            % integrate back by cumulatively summing
q=phi-q;                % subtract the correcting values
if r == 1
      q=q.';            % restore orientation
end
```

The 'trick' used in the EPHD routine is to first differentiate the phase, in order to find where the discontinuities are and determine how large they are, then normalize them by dividing by π, round this to integers, multiply back by π, integrate back to obtain the corrections, and subtract the corrections from the original phase vector.

Following our 5th-order Butterworth example, we can now write:

```
alpha=ephd(phi);              % unwrapped ;
semilogx(w,180*alpha/pi)      % show alpha in degrees vs. log-scaled w;
```

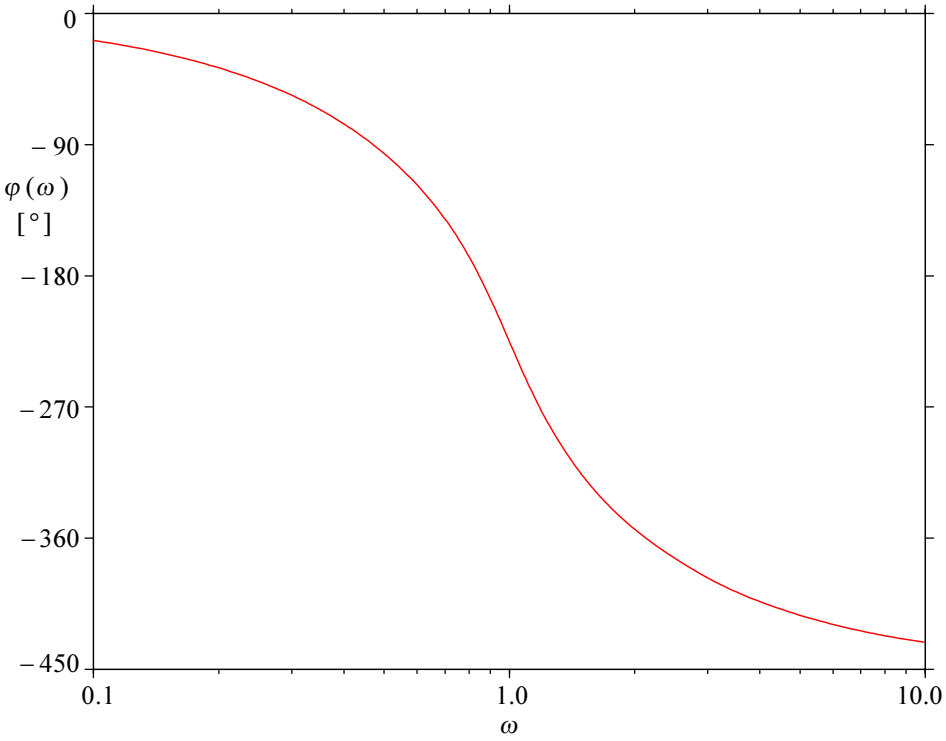


Fig. 6.4.7: Bode plot of 'unwrapped' phase, in a linear degree vs. log frequency scale.

Plotting the phase in linear degrees vs. log scaled frequency reveals an interesting fact: the system exhibits a 90° phase shift for each pole, 450° total phase shift for the 5th-order system. Also, the phase shift at the cut off frequency ω_h is exactly one half of the total phase shift (at $\omega \gg \omega_h$).

Another important fact is that stable systems (those with poles on the left half of the complex plane) will always exhibit a **negative** phase shift, whatever the system configuration (low pass or high pass, inverting or non-inverting). If you ever see a phase graph with a positive slope, first inspect what is the system gain in that frequency region. If it is 0.1 or higher that is a cause for major concern (that is, if your intention was not to build an oscillator!).

6.4.3 Frequency Dependent Envelope Delay

The **envelope delay** is defined as the phase versus frequency derivative:

$$\tau_{\mathrm{d}}(\omega) = \frac{d\varphi(\omega)}{d\omega} \tag{6.4.3}$$

(note: φ must be in radians!). Now it becomes evident why we have had to 'unwrap' the circular phase function: each 2π discontinuity would, when differentiated, produce a very high, sharp spike in the envelope delay.

Numerical differentiation can be performed by simply taking the difference of each pair of adjacent elements for both the phase and the frequency vector:

```
dphi=phi(2:1:300)-phi(1:1:299);
dw=w(2:1:300)-w(1:1:299);
```

But Matlab has a built in command called DIFF, so let us use it:

```
tau=diff(phi)./(diff(w));
```

By doing so we run into an additional problem. Numerical differentiation assigns a value to each difference of two adjacent elements, so if we started from N elements the differentiation will return $N-1$ differences. Since each difference is assigned to the interval between two samples, instead of the samples themselves, this results in a half interval delay when the result is displayed against the original frequency vector `w`.

If we have low density data we should compensate this by redefining `w`. For a linearly scaled frequency vector we would simply take the algebraic mean, `w=(w(2:1:N)-w(1:1:N-1))/2`. But for a log-scaled frequency vector, a geometric mean (the square root of a product) is needed, as in the example below:

```
w1=sqrt(w(1:1:299).*w(2:1:300));
semilogx(w1,tau)                    % see the result in Fig.6.4.8.
```

Note that the values in variable `tau` are negative, reflecting the fact that the system output is delayed in time. Since we call this response a 'delay' by definition, we could use the absolute value. However, we prefer to keep the negative sign, because it also reflects the sense of the phase rotation (see Fig. 6.4.3 and 6.4.7). An upward rotating phase (or a counter-clockwise rotation in the Bode plot of the complex frequency response) would imply a positive time delay or output before input and, consequently, an unstable or oscillatory system.

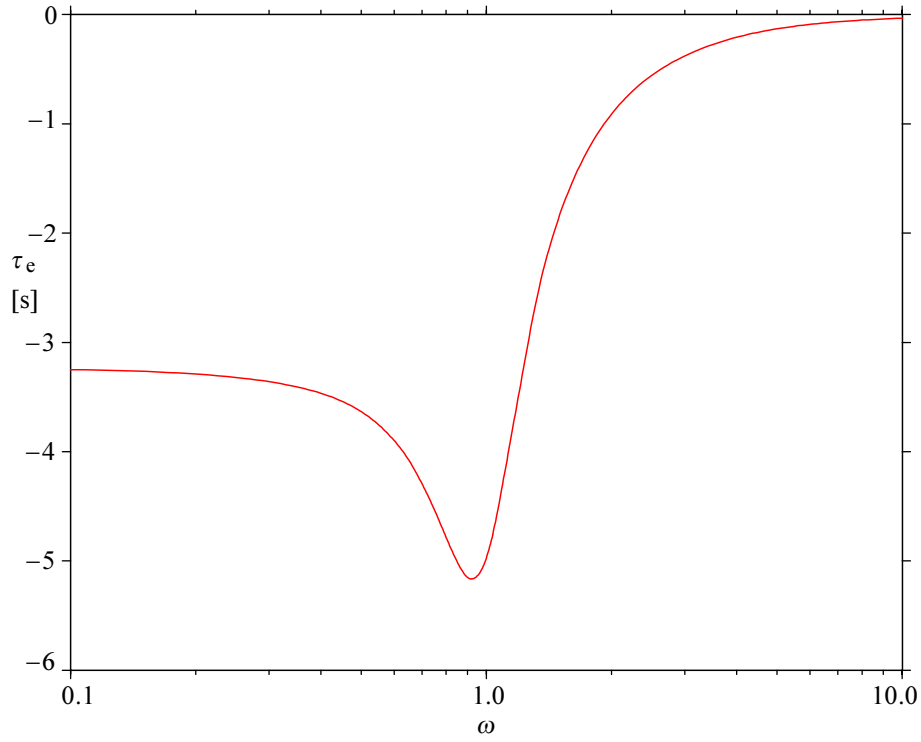


Fig. 6.4.8: The envelope (group) delay vs. frequency of the 5th-order Butterworth system. The delay is the largest for frequencies where the phase has the greatest slope.

So far we have derived the phase and group delay functions from the complex response to imaginary frequency. There are times, however, when we would like to save either processing time or memory requirement (as in embedded instrumentation applications). It is then advantageous to calculate the phase or the group delay directly from the system poles (and zeros, if any) and the frequency vector.

The phase influence of a single pole p_k can be calculated as:

$$\varphi_k(\omega, p_k) = \arctan \frac{\omega - \Im\{p_k\}}{\Re\{p_k\}} \qquad (6.4.4.)$$

The influence of a zero is calculated in the same way, but with a negative sign. The total system phase shift is equal to the sum of all particular phase shifts of poles and zeros:

$$\varphi(\omega) = \sum_{k=1}^{n} \varphi_k(\omega, p_k) - \sum_{i=1}^{m} \varphi_i(\omega, z_i) \qquad (6.4.5.)$$

But owing to the inherent complex conjugate symmetry of poles and zeros, only half of them need to be calculated and the result is then doubled. If the system order is odd the real pole is summed just once, the same is true for any real zero. This, of course, requires some sorting procedure of the system poles and zeros, but sorting is performed much quicker than multiplication with ω, which is usually a lengthy vector. If we are interested in getting data for a single frequency, or just two or three characteristic points, then it might be faster to skip sorting and calculate with all poles and zeros. In Matlab poles and zeros are already returned sorted. See the PHASE routine, in which Eq. 6.4.4 and 6.4.5 were implemented.

Note also that with the PHASE routine we obtain the 'unwrapped' phase directly and we do not have to recourse to the EPHD routine.

```
function phi=phase(z,p,w)
% PHASE  returns the phase angle of the system specified by the zeros
%        z and poles p for the frequencies in vector w :
%
%        Call :   phi=phase(z,p,w);
%
%     Instead of using angle(freqw(z,p,w)) which returns the phase
%     in the range +/-pi, this routine returns the "unwrapped" result.
%     See also FREQW, ANGLE, EPHD and GDLY.

% Author: Erik Margan, 890327, Last rev.: 980925, Free of copyright !

if nargin == 2
      w = p ;
      p = z ;
      z = [];           % A system with poles only.
end
if any( real( p ) > 0 )
      disp('WARNING :  This is not a Hurwitz-type system !' )
end
n = max( size( p ) ) ;
m = max( size( z ) ) ;
% find w orientation to return the result in the same form.
[ r, c ] = size( w ) ;
if c == 1
      w = w(:).' ;      % make it a row vector.
end
% calculate phase angle for each pole and zero and sum it columnwise.
phi(1,:) = atan( ( w - imag( p(1) ) ) / real( p(1) ) ) ;
for k = 2 : n
      phi(2,:) = atan( ( w - imag( p(k) ) ) / real( p(k) ) ) ;
      phi(1,:) = sum( phi ) ;
end
if m > 0
      for k = 1 : m
            phi(2,:) = atan( ( imag( z(k) ) - w ) / real( z(k) ) ) ;
            phi(1,:) = sum( phi ) ;
      end
end
phi( 2, : ) = [] ;      % result is in phi(1,:)
if c == 1
      phi = phi(:) ;    % restore the form same as w.
end
```

A similar procedure can be applied to the group delay. The influence of a single pole p_k is calculated as:

$$\tau_k(\omega, p_k) = \frac{\Re\{p_k\}}{\Re\{p_k\}^2 + \left(\Im\{p_k\} - \omega\right)^2} \qquad (6.4.6.)$$

As for the phase, the total system group delay is a sum of all delays for each pole and zero:

$$\tau_{\mathrm{d}}(\omega) = \sum_{k=1}^{n} \tau_k(\omega, p_k) - \sum_{i=1}^{m} \tau_i(\omega, z_i) \qquad (6.4.7.)$$

Again, owing to the complex conjugate symmetry, only half of the complex poles and zeros need to be taken into account and the result doubled, and any delay of an eventual real pole or zero is then added to it. The GDLY (Group DeLaY) routine implements Eq. 6.4.6 and 6.4.7.

```
function tau=gdly(z,p,w)
% GDLY returns the group (envelope) time delay for a system defined
%      by zeros z and poles z, at the chosen frequencies w.
%
%      Call :    tau=gdly(Z,P,w);
%
%      Although the group delay is defined as a positive time lag,
%      by which the system response lags the input, this routine
%      returns a negative value, since this reflects the sense of
%      phase rotation with frequency.
%
%      See also FREQW, PATS, ABS, ANGLE, PHASE.

% Author: Erik Margan, 890414, Last rev.: 980925, Free of copyright !

if nargin == 2
      w=p;
      p=z;
      z=[]; % system has poles only.
end
if any( real( p ) > 0 )
      disp( 'WARNING :  This is not a Hurwitz type system !' )
end
n=max(size(p));
m=max(size(z));
[r,c]=size(w);
if c == 1
      w=w(:).' ;  % make it a row vector.
end
tau(1,:) = real(p(1)) ./(real(p(1))^2 + (w-imag(p(1))).^2);
for k = 2 : n
      tau(2,:) = real(p(k)) ./(real(p(k))^2 + (w-imag(p(k))).^2);
      tau(1,:) = sum( tau ) ;
end
if m > 0
      for k = 1 : m
            tau(2,:)=-real(Z(k)) ./(real(Z(k))^2 + (w-imag(Z(k))).^2);
            tau(1,:) = sum( tau ) ;
      end
end
tau(2,:) = [] ;
if c == 1
      tau = tau(:) ;
end
```

6.5 Transient Response by Fourier Transform

There are several methods for time domain response calculation. Three of these that are interesting from the system designer's point of view, including the FFT method, were compared for efficiency and accuracy in [Ref. 6.23]. Besides the high execution speed, the main advantage of the FFT method is that we do not even have to know the exact mathematical expression for the system frequency response, but only the graph data (i.e. if we have measured the frequency and phase response of a system). Although the method was described in detail in [Ref. 6.23] we shall repeat here the most important steps, to allow the reader to follow the algorithm development.

There are **five difficulties** associated with the discrete Fourier transform that we shall have to solve:

a) the inability to transform some interesting functions (e.g., the unit step);

b) the correct treatment of the DC level in low pass systems;

c) preserving accuracy with as little spectral information input as possible;

d) find to what extent our result is an approximation owed to finite spectral density;

e) equally important, estimate the error owed to finite spectral length.

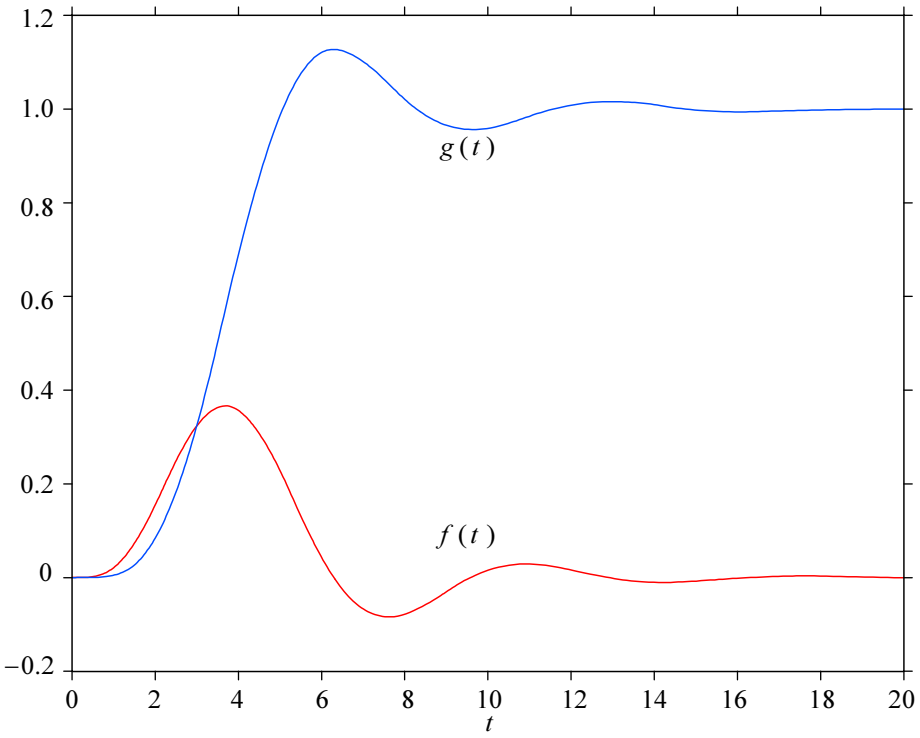


Fig. 6.5.1: The impulse and step response of the 5th-order Butterworth system. The impulse amplitude has been normalized to represent the response to an ideal, infinitely narrow, infinite amplitude input impulse. The impulse response reaches the peak value at the time equal to the envelope delay value at DC; this delay is also the half amplitude delay of the step response. The step response first crosses the final value at the time equal to the envelope delay maximum. Also the step response peak value is reached when the impulse response crosses the zero level for the first time. If the impulse response is normalized to have the area (the sum of all samples) equal to the system DC gain, the step response would be simply a time integral of it.

6.5.1 Impulse Response, Using FFT

The basic idea behind this method is that the Fourier transform is a special case of the more general Laplace transform and the Dirac impulse function is a special type of signal for which the Fourier transform solution always exists. Comparing Eq. 1.3.8 and Eq. 1.4.3 and taking in account that $s = \sigma + j\omega$, we see:

$$\mathcal{L}\{f(t)\} = \mathcal{F}\{f(t)\,\mathrm{e}^{-\sigma t}\} \tag{6.5.1}$$

Since the complex plane variable s is composed of two independent parts (real and imaginary), then $F(s)$ may be treated as a function of two variables, σ and ω. This can be most easily understood by looking at Fig. 6.4.1, in which the complex frequency response (magnitude) of a 5-pole Butterworth function is plotted as a 3D function over the Laplace plane.

In that particular case we had:

$$F(s) = \frac{-s_1 s_2 s_3 s_4 s_5}{(s - s_1)(s - s_2)(s - s_3)(s - s_4)(s - s_5)} \tag{6.5.2}$$

where $s_{1\text{–}5}$ have the same values as in the example at the beginning of Sec. 6.4.1.

When the value of s in Eq. 6.5.2 becomes close to the value of one of the poles, s_i, the magnitude $|F(s)|$ then increases until becoming infinitely large for $s = s_\mathrm{i}$.

Let us now introduce a new variable p such that:

$$p = s\,\Big|_{\sigma\,=\,0} \qquad \text{or:} \qquad p = j\omega \tag{6.5.3}$$

This has the effect of slicing the $|F(s)|$ surface along the imaginary axis, as we did in Fig. 6.4.1, revealing the curve on the surface along the cut, which is $|F(j\omega)|$, or in words: the magnitude $M(\omega)$ of the complex frequency response. As we have indicated in Fig. 6.4.5, we usually show it in a log-log scaled plot. However, for transient response calculation a linear frequency scale is appropriate (as in Fig. 6.4.2), since we need the result of the inverse transform in linear time scale increments.

Now that we have established the connection between the Laplace transformed transfer function and its frequency response we have another point to consider: conventionally, the Fourier transform is used to calculate waveform spectra, so we need to establish the relationship between a frequency response and a spectrum. Also we must explore the effect of taking **discrete values (sampling)** of the time domain and frequency domain functions, and see to what extent we **approximate** our **results** by taking **finite length vectors of finite density sampled data.** Those readers who would like to embed the inverse transform in a microprocessor controlled instrument will have to pay attention to **amplitude quantization (finite word length)** as well, but in Matlab this is not an issue.

We have examined the Dirac function $\delta(t)$ and its spectrum in Part 1, Sec. 1.6.6. Note that the spectral components are separated by $\Delta\omega = 2\pi/T$, where T is the impulse repetition period. If we allow $T \to \infty$ then $\Delta\omega \to 0$. Under these conditions we

can hardly speak of discrete spectral components because the spectrum has become very dense; we rather speak of **spectral density**. Also, instead of individual components' magnitude we speak of **spectral envelope** which for $\delta(t)$ is essentially flat.

However, if we do not have an infinitely dense spectrum, then $\Delta\omega$ is small but not 0, and this merely means that the impulse repeats after a finite period $T = 2\pi/\Delta\omega$ (this is the mathematical equivalent of testing a system by an impulse of a duration much shorter than the smallest system time constant and of a repetition period much larger than the largest system time constant).

Now let us take such an impulse and present it to a system having a selective frequency response. Fig. 6.5.2 shows the results both in the time domain and the frequency domain (magnitude). The time domain response is obviously the system impulse response, and its equivalent in the frequency domain is a spectrum, whose density is equal to the input spectral density, but with the spectral envelope shaped by the system frequency response. The conclusion is that we only have to sample the frequency response at some finite number of frequencies and perform a discrete Fourier transform inversion to obtain the impulse response.

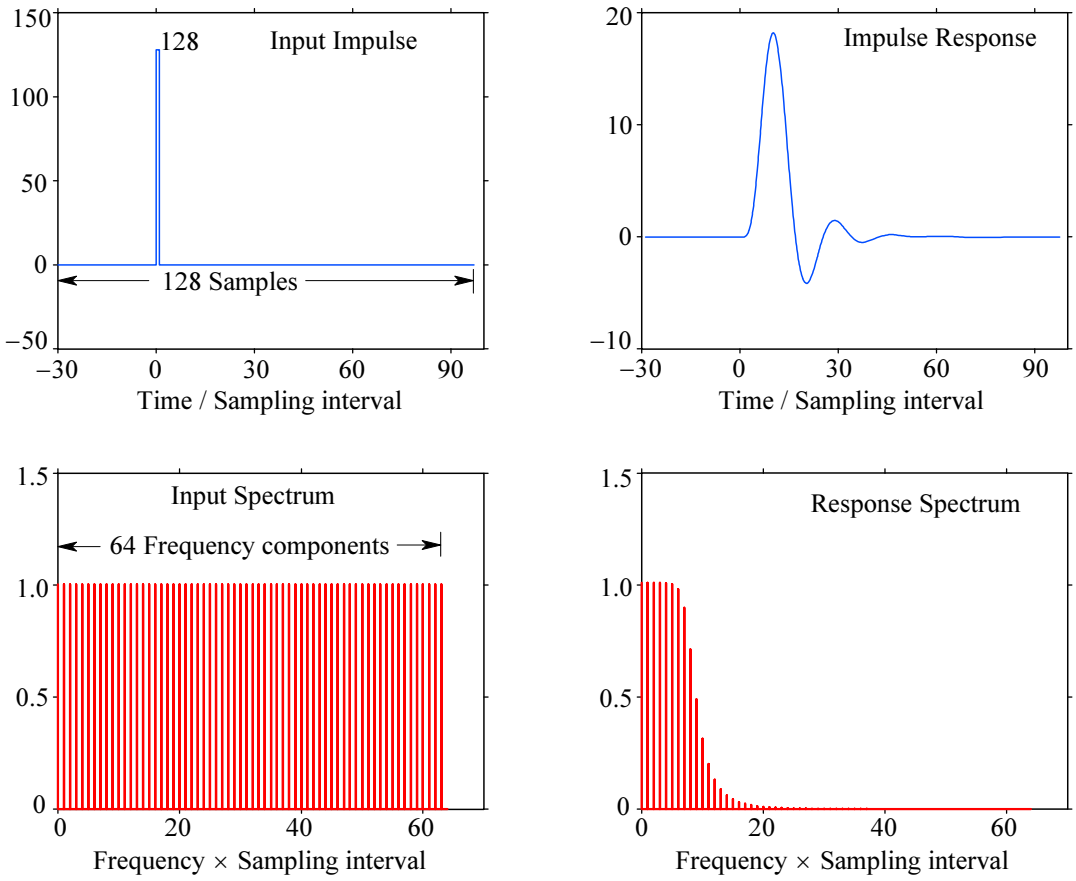


Fig. 6.5.2: Time domain and frequency domain representation of a 5-pole Butterworth system impulse response. The spectral envelope (only the magnitude is shown here) of the output is shaped by the system frequency response, whilst the spectral density remains unchanged. From this fact we conclude that the time domain response can be found from a system frequency response using inverse Fourier transform. The horizontal scale is the number of samples (128 in the time domain and 64 in the frequency domain — see the text for the explanation).

If we know the magnitude and phase response of a system at some finite number of equally spaced frequency points, then each point represents:

$$F_i = M_i \cos(\omega_i t - \varphi_i) \tag{6.5.4}$$

As the contribution of frequencies components which are attenuated by more than, say, 60 dB can be neglected, we do not have to take into account an infinitely large number of frequencies, and the fact that we do not have an infinitely dense spectrum merely means that the input impulse repeats in time. By applying the superposition theorem, the output is then equal to the sum of all the separate frequency components.

Thus for each time point the computer must perform the addition:

$$f(t_k) = \sum_{i(\omega_{\min})}^{i(\omega_{\max})} M_i \cos(\omega_i t_k - \varphi_i) \tag{6.5.5}$$

Eq. 6.5.5 is the **discrete Fourier transform**, with the exponential part expressed in trigonometric form. However, if we were to plot the response calculated after Eq. 6.5.5, we could see that the time axis is reversed, and from the theory of Fourier transform properties (symmetry property, [Ref. 6.14, 6.15, 6.18]), we know that the application of two successive Fourier transforms returns the original function but with the sign of the independent variable reversed:

$$\mathcal{F}\Big\{\mathcal{F}\{f(t)\}\Big\} = \mathcal{F}\{F(j\omega)\} = f(-t) \tag{6.5.6}$$

or more generally:

$$f(t) \underset{\mathcal{F}^{-1}}{\overset{\mathcal{F}}{\rightleftarrows}} F(j\omega) \underset{\mathcal{F}^{-1}}{\overset{\mathcal{F}}{\rightleftarrows}} f(-t) \underset{\mathcal{F}^{-1}}{\overset{\mathcal{F}}{\rightleftarrows}} F(-j\omega) \underset{\mathcal{F}^{-1}}{\overset{\mathcal{F}}{\rightleftarrows}} f(t) \tag{6.5.7}$$

The main drawback in using Eq. 6.5.5 is the high total number of operations, because there are three input data vectors of equal length (ω, M, φ) and each contributes to every time point result. It seems that greater efficiency might be obtained by using the input frequency response data in the complex form, with the frequency vector represented by the index of the $F(j\omega)$ vector.

Now $F(j\omega)$ in its complex form is a two sided spectrum, as was shown in Fig. 6.4.3, and we are often faced with only a single sided spectrum. It can be shown that a real valued $f(t)$ will always have $F(j\omega)$ symmetrical about the real axis σ. Thus:

$$F_N(j\omega) = F_P^*(-j\omega) \tag{6.5.8}$$

F_N and F_P are the $\omega < 0$ and $\omega > 0$ parts of $F(j\omega)$, with their inverse transforms labeled $f_N(t)$ and $f_P(t)$. Note that F_P^* is the complex conjugate of F_P.

This symmetry property follows from the definition of the **negative frequency concept**: instead of having a single phasor rotating counter-clockwise (positive by definition) in the complex plane, we can always have two half amplitude phasors rotating in opposite directions at the same frequency (as we have already seen drawn in Part 1, Fig. 1.1.1; for vector analysis see Fig. 6.5.3**)**. We can therefore conclude that the

inherent conjugate symmetry of the complex plane allows us to define 'negative frequency' as a clockwise rotating, half amplitude phasor, being the complex conjugate of the usual counter-clockwise (positive by definition) rotating (but now also half amplitude) phasor. And this is not just a fancy way of making simple things complex, but is rather a direct consequence of our dislike of sine–cosine representation and the preference for the complex exponential form, which is much simpler to handle analytically.

One interesting aspect of the negative frequency concept is the ***Shannon* sampling theorem**: for a continuous signal, sampled with a frequency f_s, all the information is contained within the frequency range between 0 and $f_s/2$, because the spectrum from $f_s/2$ to f_s is a mirror image, so the spectrum is symmetrical about $f_s/2$, the *Nyquist* frequency. Therefore a frequency equal to f_s can not be distinguished from a DC level, and any frequency from $f_s/2$ to f_s can not be distinguished from $f_s - f$.

But please, also note that this 'negative frequency' does not necessarily imply 'negative time', since the negative time is defined as the time before some arbitrary instant $t = 0$ at which the signal was applied. In contrast, the negative frequency response is just one half of the full description of the $t \geq 0$ signal.

However, those readers who are going explore the properties of the *Hilbert transform* will learn that this same concept can be extended to the $t < 0$ signal region, but this is beyond the scope of this text.

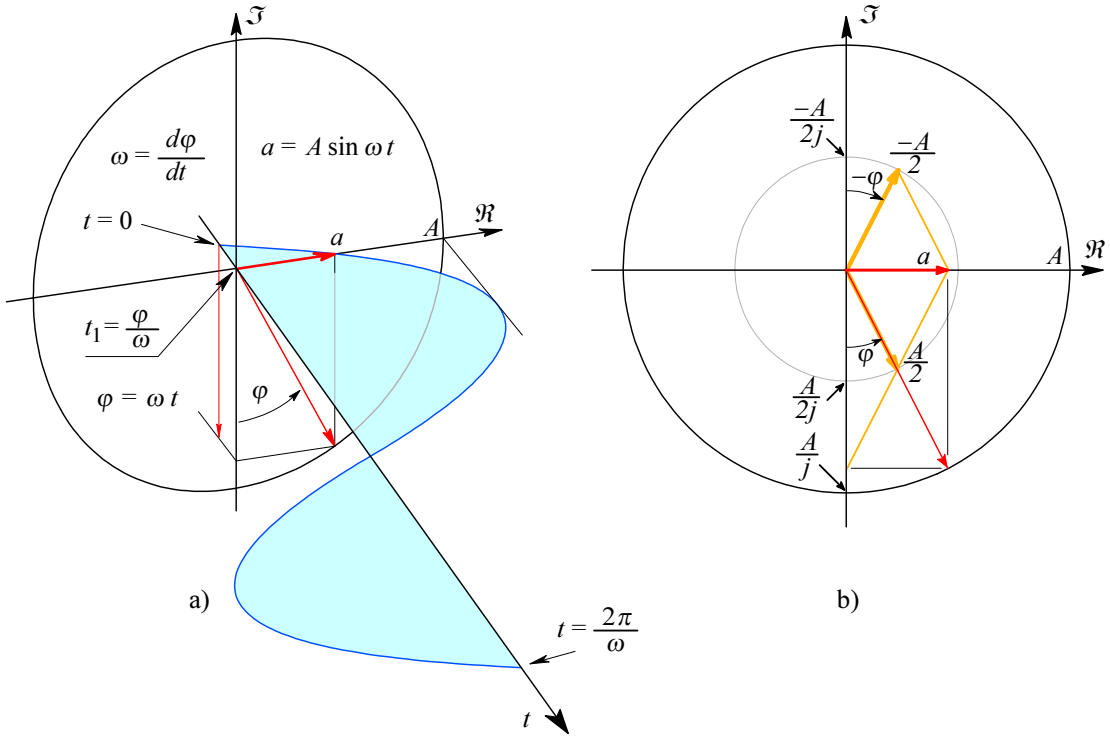


Fig. 6.5.3: As in Part 1, Fig. 1.1.1, but from a slightly different perspective: **a)** the real signal instantaneous amplitude $a(t) = A \sin\varphi$, where $\varphi = \omega t$; **b)** the real part of the instantaneous signal phasor, $\Re\{\vec{0A}\} = \vec{0a} = A \sin\varphi$, can be decomposed into two half amplitude, oppositely rotating, complex conjugate phasors, $-(A/2j)\sin\varphi + (A/2j)\sin(-\varphi)$. The second term has rotated by $-\varphi = -\omega t$ and, since t is obviously positive (see the **a)** graph), the negative sign is attributed to ω; thus, **clockwise rotation is interpreted as a 'negative frequency'**.

Eq. 6.5.8 can thus be used to give:

$$F(j\omega) = F_P(j\omega) + F_P^*(-j\omega) \tag{6.5.9}$$

but from Eq. 6.5.7 we have:

$$\mathcal{F}^{-1}\Big\{F_P^*(-j\omega)\Big\} = f_P^*(t) \tag{6.5.10}$$

hence using Eq. 6.5.9 and Eq. 6.5.10 and taking into account the cancellation of imaginary parts, we obtain:

$$f(t) = f_P(t) + f_P^*(t) = 2\,\Re\Big\{f_P(t)\Big\} \tag{6.5.11}$$

Eq. 6.5.11 means that if $F_P(j\omega)$ is the $\omega \geq 0$ part of the Fourier transformed real valued function $f(t)$, its Fourier transform inversion $f_P(t)$ will be a complex function whose real part is equal to $f(t)/2$. Summing the complex conjugate pair results in a doubled real valued $f(t)$. So by Eq. 6.5.10 and Eq. 6.5.11 **we can calculate the system impulse response from just one half of its complex frequency response using the forward Fourier transform** (not inverse!):

$$f(t) = 2\,\Re\left\{\left[\mathcal{F}\Big\{F_P^*(j\omega)\Big\}\right]^*\right\} \tag{6.5.12}$$

Note that the second (outer) complex conjugate is here only to satisfy the mathematical consistency — in the actual algorithm it can be safely omitted, since only the real part is required.

As the operator $\mathcal{F}\{\}$ in Eq. 6.5.12 implies integration we must use the **discrete Fourier transform (DFT)** for computation. The DFT can be defined by decomposing the Fourier transform integral into a finite sum of N elements:

$$F(k) = \frac{1}{N}\sum_{i=0}^{N-1} f(i)\,\mathrm{e}^{-j\frac{2\pi k}{N}} \tag{6.5.13}$$

That means going again through a large number of operations, comparable to Eq. 6.5.5. Instead we can apply the **Fast Fourier Transform (FFT)** algorithm and, owing to its excellent efficiency, save the computer a great deal of work.

Cooley and *Tukey* [Ref. 6.16] have shown that if $N = 2^B$ and B integer, there is a smart way to use Eq. 6.5.13, owing to the periodical nature of the Fourier transform.

If Eq. 6.5.13 is expressed in a matrix form then the matrix which represents its exponential part can be divided into its even and odd part, and the even part can be assigned to $N/2$ elements. The remaining part can then also be divided as before and the same process can then be repeated over and over again, so that we end up with a number ($\log_2 N$) of individual sub-matrices. Furthermore, it can be shown that these sub-matrices contain only two non-zero elements, one of which is always unity (1 or j).

Therefore multiplying by each of the factor matrices requires only N complex multiplications.

Finally (or firstly, depending on whether we are using the 'decimation in frequency' or the 'decimation in time' algorithm), we rearrange the data, by writing the position of each matrix element in a binary form, and reordering the matrix it by reversing the binary digits (this operation is often referred to as the 'reshuffling').

The total number of multiplications is thus $N \log_2 N$ instead of the N^2 required to multiply in one step. Other operations are simple and fast to execute (addition, change of sign, change of order). Thus in the case of $B = 10$, $N = 1024$, and $N^2 = 1048576$ whilst $N \log_2 N = 10240$, so a reduction of the required number of multiplications by **two orders of magnitude** has been achieved.

Matlab has a command named 'FFT' which uses the 'radix-2' type of algorithm and we shall use it as it is. Those readers who would like to implement the FFT algorithm for themselves can find the detailed treatment in [Ref. 6.16, 6.17 and 6.19].

A property of the FFT algorithm is that it returns the spectrum of a real valued signal as folded about the Nyquist frequency (one half of the frequency at which the signal was sampled). As we have seen in Fig. 6.5.2, if we have taken 128 signal samples, the FFT returns the first 64 spectral components from $\omega = 0, 1, 2, \ldots, 63$ but then the remaining 64 components, which are the complex conjugate of the first ones, are in the reversed order.

This is in contrast to what we were used to in the analytical work, as we expect the complex conjugate part to be on the $\omega < 0$ side of the spectrum. On the other hand, this is equivalent, since the 128 samples taken in the signal time domain window are implicitly assumed to repeat, and consequently the spectrum is also repeating every 128 samples. So if we use the standard inverse FFT procedure we must take into account all 128 spectral components to obtain only 64 samples of the signal back. However, note that the Eq. 6.5.12 **requires only a single–sided spectrum of N points to return N points of the impulse response**. This is, clearly, an **additional two–fold improvement** in algorithm efficiency.

6.5.2 Windowing

For calculating the transient response a further reduction in the number of operations is possible through the use of **'windowing'**. Windowing means multiplying the system response by a suitable window function. We shall use the windowing in the frequency domain, the reason for this we have already discussed when we were considering to what extent DFT is an approximation. Like many others before us, in particular the authors of various window functions, we, too, have found out that the accuracy improves if the influence of higher frequencies is reduced.

Since the frequency response of a high order system (third-order or greater) falls off quickly above the cut off frequency, we can take just $N = 256$ frequency samples, and after inverse FFT we still obtain a time domain response with an accuracy equal to or better than the vertical resolution of the VGA type of graphics (1/400 or 0.25%). And as the sample density (number of points) of the transient wavefront increases with the number of stop band frequency samples, it is clear that the smaller the contribution of higher frequencies, the greater is the accuracy.

But, in order to achieve a comparable accuracy with 1st- and 2nd-order systems we would have to use $N_1 = 4096$ and $N_2 = 1024$ frequency samples, respectively. Thus since we would like to minimize the length of the frequency vector, low order systems need to be artificially rolled off at the high end. This can be done by multiplying the frequency response by a suitable window function, element by element, as shown in Fig. 6.5.4. The window function used in Fig. 6.5.4 (and also in the TRESP routine) is a real valued Hamming type of window (note that we need only its right hand half, since we use a single-sided spectrum; the other half is implicitly used owing to Eq. 6.5.12).

```
W=0.54-0.46*cos(2*pi*(N+1:1:2*N)/(2*N)); % right half Hamming window
```

The physical effect of applying the Hamming window is similar to the implementation of an additional non-dominant pole of high order at about $12\,\omega_h$ and a zero at about $25\,\omega_h$. Multiplication in frequency domain is equivalent to convolution in time domain, and vice-versa; however, note that the window function is **real**, thus no phase distortion occurs!

After extensive experimentation with different types of windows, it was found that the Hamming window yields the lowest error, most notably at the first few points of the impulse response. This is understandable, since the FFT of this window has lowest spectral side lobes and the same is true for the inverse transform. For the same reason the final value error of the 1st- and 2nd-order system will also be reduced.

Note that the first point of the 1st-order impulse response will be in error anyway, because we have started from a spectrum of finite density, in which the distance between adjacent spectral components was equal to $1/N$. The time domain equivalent of this is an impulse whose amplitude is finite (N) and its width is $1/N$ (so that the impulse area is equal to 1). This means that the response rise time is not infinitely short, so its first point will be smaller than expected, the error being proportional to $1/N$.

If the system order and type could be precisely identified, this error might be corrected by forcing the first point of the 1st-order impulse response to a value equal to the sampling time period, multiplied by the system DC gain, as has been done in the TRESP routine.

In Sec. 6.5.6 we give a more detailed error analysis for the 1st-, 2nd- and 3rd-order system for both windowed and non-windowed spectrum.

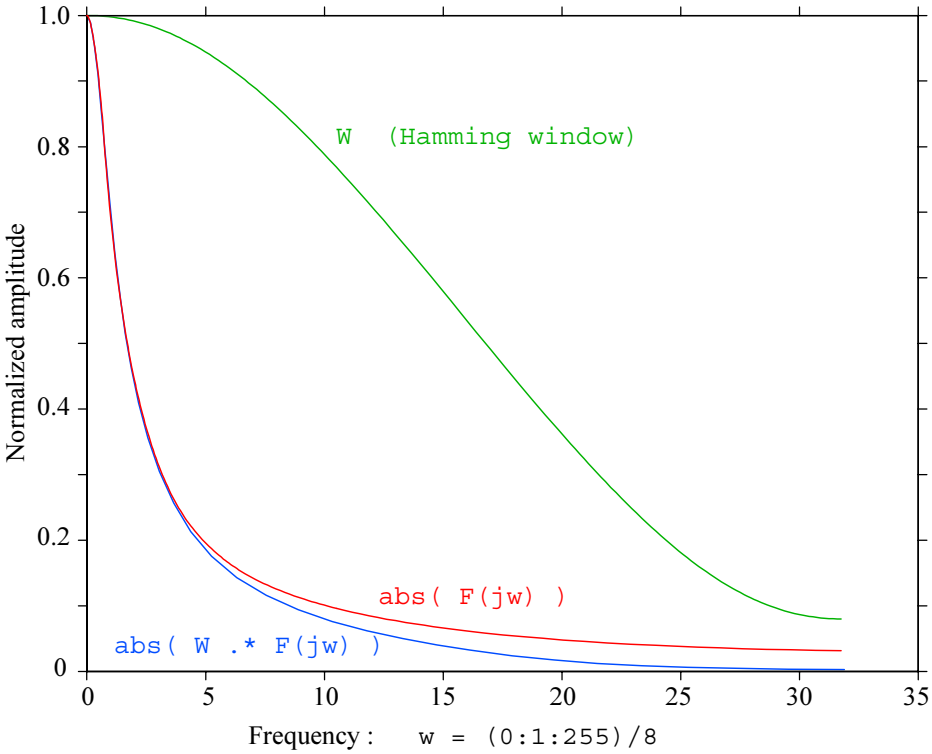


Fig. 6.5.4: Windowing example. The 1st-order frequency response (only the magnitude is shown on plot) is multiplied elemet by element by the Hamming type of window function in order to reduce the influence of high frequencies and improve the impulse response calculation accuracy. Note that the window function is real only, affecting equally the system real and imaginary part, thus the phase information is preserved and only the magnitude is corected.

6.5.3 Amplitude Normalization

The impulse response, as returned by Eq. 6.5.12, has the amplitude such that the sum of all the N values is equal to N times the system gain. Also, if we are dealing with a low pass system and the first point of its frequency response (the DC component) is $F(0)$, then the impulse response will be shifted vertically by $F(0)$ (the DC component is added). Thus we must first cancel this 'DC offset' by subtracting $F(0)$ and then normalize the amplitude by dividing by N:

$$f_{\mathrm{n}}(t) = \frac{f(t) - F(0)}{N} \tag{6.5.14}$$

By default, the TRESP routine (see below) returns the impulse amplitude in the same way, representing a unity gain system's response. Optionally, we can denormalize it as if the response was caused by an ideal, infinitely high impulse; then the 1st-order response starts the exponential decay from a value very close to one, as it should. If the system's half power bandwidth, ω_{h}, is found at the `m+1` element of the frequency response vector, the amplitude denormalization factor will be:

$$A = \frac{N}{2\pi\,\mathrm{m}} \tag{6.5.15}$$

The 2π factor comes as a bit of surprise here. See Sec. 6.5.5 about time scale normalization for an explanation.

The term `m` can be entered explicitly, as a parameter. But it can also be derived from the frequency vector by finding the index at which it is equal to 1, or it can be found by examining the magnitude and finding the index of the point, nearest to the half power bandwidth value (in both of cases the index must be decremented by 1).

Another problem can be encountered with high order systems, which exhibit a high degree of ringing, e.g., Chebyshev systems of order 8 or greater. If `m<8`, some additional ringing is introduced into the time domain response. This ringing results from the time frame implicitly repeating with a period $T = 2\pi/\Delta\omega$, where $\Delta\omega$ describes the finite spectral density of input data. If we have specified the system cut off frequency too near to the origin of the frequency vector it would cause a time scale expansion. Thus overlapping of adjacent responses will introduce distortion if the impulse response has not decayed to zero by the end of the period T. Therefore the choice of placing the cut off frequency relative to the frequency vector is a compromise between the pass band and stop band description. In Matlab, the frequency vector of `N` linearly spaced frequencies, normalized to 1 at its `m+1` element, can be written as:

```
N=256;      m=8;          w=(0:1:N-1)/m;
```

The variable `m` specifies the normalized frequency unit. The transient response of both Butterworth and Bessel systems can be calculated with good accuracy by using a frequency vector normalized to 1 at its 5th sample (`m=4`). But by placing the cutoff frequency at the 9th sample (`m=8`) of a frequency vector of length 256, an acceptably low error will be achieved even for a 10th-order Chebyshev system. For higher order, high ringing systems one will probably need to increase the frequency vector to 512 or 1024 elements in order to prevent time window overlapping.

6.5.4 Step Response

Up to this point we have obtained the impulse response. The step response is not available straight from the Fourier transform (if the unit step is integrated, the integral will diverge to $+\infty$; this is why we prefer the more general Laplace transform for analytical work). However, from signal analysis theory we know that the response to an arbitrary input signal can be found by convolving it with the system's impulse response (for convolution, see Part 1, Sec. 1.15 and Fig. 1.15.1; see Part 7, Sec. 7.2 for the numerical algorithm). With the unit step as the input signal the convolution is reduced to a simple time domain integration of the system impulse response.

But this integration must give us the final step response value equal, or at least very close, to the system DC gain. So we must use the impulse response normalized to obtain the sum of all its elements equal to the system's gain. Numerical integration can be done by cumulative summing. This means that the first element is transferred unchanged, the second element is the sum of the first two, the third of the first three, and so on, up to the last element which is the sum of all elements. The CUMSUM command in Matlab will return in `Y` a cumulative sum of vector `X` like this:

```
                                % the CUMulative-SUM example :
Y(1)=X(1);                      % transfer the first sample unchanged
for k=2:max(size(X))
     Y(:,k)=Y(k-1)+X(k);        % next sample is the sum of all previous
end
```

However, inside the Matlab command interpreter, this `for`-loop executes considerably slower, owing to vector remapping, thus we shall use the built in CUMSUM command in the TRESP routine.

An interesting problem arises with the first-order system, as well as any system with zeros, since the first sample of the impulse response will be non-zero, therefore the step response will also start from above zero, which is not so in the real world. We can solve this problem by artificially adding a zero valued first element to the impulse response vector, but we then have to be careful not to increment the total number of elements by one. That is because we want to keep the sum of the original impulse response vector, divided by the number of elements, equal to the system gain (the increased number of elements affects the normalization).

Also, as we have seen in the envelope delay derived from the phase response, the numerical differentiation, owing to a finite number of elements taken into account, results in a one half sample shift. The numerical integration, being an inverse process, does the same thing, even if does not increase the number of elements by one. The integration shift is also in the opposite direction.

Both these problems will be treated in the next section dealing with time scale normalization.

6.5.5 Time Scale Normalization

When we plot a time domain response we want to be able to correlate it to the system's envelope delay, so we need to specify the time scale normalization factor. This factor depends on how many samples of the frequency response were used in the FFT and which sample was at the system's cut off frequency.

We have already seen that a Fourier transformed signal has a spectrum periodic in frequency, the period being $\omega_s = 2\pi/\Delta t$, thus inversely proportional to the sampling period. Also the spectral density $\Delta\omega$ reflects the overall time domain repetition period, $T_R = 2\pi/\Delta\omega$, which represents the size of the time domain window. Note that when we specify the single-sided spectrum, we usually do it from 0 up to the Nyquist frequency $\omega_N = \omega_s/2 = N\Delta\omega$.

Now our frequency vector has been defined as: `w=(0:1:N-1)/m`. Obviously, at its `m+1` element `w(m+1)=1`, which is the numerical equivalent of the normalized cut off frequency $\omega_h = 1$. Also, $\Delta\omega$ can be found as `w(2)-w(1)`, but `w(1)=0` and `w(2)=1/m`, so we can say that $\Delta\omega = \omega_h/m$. Since the Nyquist frequency is $\omega_N = N\Delta\omega$, and the sampling frequency is twice that, $\omega_s = 2N\Delta\omega$, then the sampling time interval is $\Delta t = 1/\omega_s = 1/2N\Delta\omega = m/2N\omega_h$.

But remember that Eq. 6.5.12 allows us to improve our algorithm, obtaining N time samples from N frequency samples, a result which we would otherwise get from $2N$ frequency samples. So, we have $\Delta t = m/N\omega_h$. Also remember that we have calculated the frequency response using a normalized frequency vector, ω/ω_h, and the term $\omega_h = 1$ was effectively replaced by $f_h = 1$, loosing 2π. Our sampling interval should therefore be:

$$\Delta t = \frac{2\pi\, m}{N} \tag{6.5.16}$$

You may have noted that this is exactly the inverse of the amplitude denormalization factor, Eq. 6.5.15, $\Delta t = 1/A$. This is not just a strange coincidence! Remember Fig. 6.5.2: the amplitude and the width of the input impulse were set so that $A\Delta t = N$, with N being also the time domain vector's length, so if $\Delta t = 1$ then $A = N$. For the unity gain system the response must contain the same amount of energy as the input, thus the sum of all the response values must also be equal to N. Of course, N is a matter of choice, but once its value has been chosen it is a constant. So it is only the system bandwidth, set by the factor m, that will determine the relationship between the response amplitude and its time scale.

Therefore, after Eq. 6.5.16:

```
dt=2*pi*m/N;            % delta-t
t=dt*(0:1:N-1);         % normalized length-N time vector
```

We may check the time normalization easily by calculating the impulse response of a first-order system and comparing it to the well known reference, the analytical first-order low pass RC system impulse response, which is just:

$$f_r(t) = e^{-t/RC} \tag{6.5.17}$$

Now, normalizing the time scale means showing it in increments of the system time constant ($RC, 2RC, 3RC, \ldots$). Thus we simply set $RC = 1$. For the starting sample at $t = 0$, the response $f(0) = 1$, so in order to obtain the response of a unity gain system excited by a finite amplitude impulse we must denormalize the amplitude (see Eq. 6.5.15) by $1/A$:

$$f_r(t) = \frac{1}{A} e^{-t} \tag{6.5.18}$$

```
z=[];                                % no zeros,
p=-1;                                % just a single real pole
N=256;                               % total number of samples
m=8;                                 % samples in the frequency unit
w=(0:1:N-1)/m;                       % the frequency vector
dt=2*pi*m/N;                         % sampling time interval = 1/A
t=dt*(0:1:N-1);                      % the time vector
F=freqw(z,p,w);                      % the frequency response
In=(2*real(fft(conj(F)))-1)/N;       % the impulse response
Ir=dt*exp(-t);                       % 1st-order ref., denormalized
plot( t, Ir, t, In )
title('Ideal vs. windowed response'), xlabel('Time')
plot( t(1:30), Ir(1:30), t(1:30), In(1:30) )
title('Zoom first 30 samples')
```

In the above example (see the plot in Fig. 6.5.5), we see that the final values of normalized impulse response `In` are not approaching zero, and by zooming on the first 30 points we can also see that the first point is too low and the rest somewhat lower than the reference. Windowing can correct this:

```
W=0.54-0.46*cos(2*pi*(N+1:2*N)/(2*N));   % right half Hamming window
Iw=(2*real(fft(conj(F.*W)))-1)/N;        % impulse, windowed fr.resp.
plot( t, Ir, t, Iw ), xlabel('Time')
title('Ideal vs. windowed response')
plot( t(1:30), Ir(1:30), t(1:30), Iw(1:30) )
title('Zoom first 30 samples')
```

This plot fits the reference much better. But the first point is still far too low. From the amplitude denormalization factor, by which the reference was multiplied, we know that the correct value of the first point should be $1/A = \Delta t$. So we may force the first point to this value, but, by doing so, we would alter the sum of all values by `N*(dt-I(1))`. In order to obtain the correct final value of the step response, the impulse response requires the correction of all points by `1/(1+(dt-I(1))/N)`, as in the following example:

```
% the following correction is valid for 1st-order system only !!!
er1=dt-I(1);                         % the first point error
Iw(1)=dt;                            % correct first-point amplitude
% note that with this we have altered the sum of all values by er1,
% so we should modify all the values by :
Iw=Iw*(1/(1+er1/N));
Ir=Ir*(1/(1+er1/N));
% the same could also be achieved by :  Ix=Ix/sum(Ix);
plot( t(1:30), Ir(1:30), t(1:30), Iw(1:30) ), title('Zoom first 30')
plot( t, (Iw-Ir) ), title('Impulse response error plot')
```

Likewise we can compare the calculated step response. Our reference is then:

$$f_r(t) = 1 - e^{-t} \tag{6.5.19}$$

But if the first-order impulse response is numerically integrated the value of the first sample of the step response will be equal to the value of the first sample of the impulse response instead of zero, as it should be in the case of a low pass LTIC system.

Also, there is an additional problem resulting from numerical integration, which manifests itself as a one half sample time delay. Remember what we have observed when we derived the envelope delay from the phase: numerical differentiation has assigned each result point to each difference pair of the original data, so that the resulting vector was effectively shifted left in (log-scaled) frequency by a (geometrical) mean of two adjacent frequency points, $\sqrt{\omega_n\,\omega_{n+1}}$. Because we work with a linear scale, the shift is the arithmentic mean, $\Delta\omega/2$. Since the numerical integration is the inverse process of differentiation, the signal is shifted right. However, whilst the differentiation vector had one sample less, the numerical integration returns the same number of samples, not one more.

So, in order to see the actual shape of the error, we have to compensate for this shift of one half sample. We can do this by artificially adding a leading zero to the impulse response vector, then cumulatively sum the resulting $N+1$ elements and finally take the mean value of this and the version shifted by one sample, as in the example below which is uses the vectors from above (see the result in Fig. 6.5.6):

```
Sr=1-exp(-t);                  % 1st-order step response reference
Sw=cumsum([0, Iw]);            % step resp. from impulse + leading zero
     % compensate the one half sample delay by taking the mean :
Sw=(Sw(1:N)+Sw(2:N+1))/2;
Sw(1)=0;                       % correct the first value
plot( t, Sr, t, Sw ), title('Ideal vs. windowed step response')
plot( t(1:50), Sr(1:50), t(1:50), Sw(1:50) )
title('Zoom first 50 samples')
plot( t, (Sw-Sr) ), title('Step response error plot')
```

This compensation will lower the algorithm's efficiency somewhat, but considering the embedded applications numerical addition is fast and division by 2 can be done by shifting bits one binary place to the right, so the operation can still be performed solely in integer arithmetics.

For the second-order system we can use the reference response which was calculated in Part 2, Eq. 2.2.37 (see the error plots in Fig. 6.5.7 and 6.5.8.):

```
[z,p]=buttap(2);                    % 2nd-order Butterworth system
% the following variables are the same as before:
N=256; m=8; w=(0:1:N-1)/m; dt=2*pi*m/N; t=dt*(0:1:N-1);
F=freqw(z,p,w);                     % complex frequency response
W=0.54-0.46*cos(2*pi*(N+1:2*N)/(2*N));  % a right-half Hamming window
Iw=(2*real(fft(conj(F.*W)))-1)/N;   % impulse resp., windowed fr.resp.
Sw=cumsum([0,Iw]);                  % numerical integration, step r.
Sw=(Sw(1:N)+Sw(2:N+1))/2;           % compensate half sample t-delay
T=sqrt(2);                          % 2nd-order response constants,
theta=pi/4;                         % see example: Part 2, Eq.2.2.39.
Sr=1-T*exp(-t/T).*sin(t/T+theta);   % the 2nd-order response reference
plot(t,Sr,t,Sw), title('Ideal vs. windowed step response')
plot(t(1:60),Sr(1:60),t(1:60),Sw(1:60)), title('Zoom samples 1-60')
plot( t, (Sw-Sr) ), title('Step response error plot')
```

Note that the TRESP routine allows us to enter the actual denormalized frequency vector, in which case all (but the first one) of its elements might be greater than 1. The normalized frequency unit `m` is then found from the frequency response, by checking which sample is closest to `abs(F(1))/sqrt(2)`, and then decremented by 1 to compensate for the frequency vector starting from 0. But in the case of a denormalized frequency vector we should also denormalize the time scale, by dividing the sampling interval by the actual upper cut off frequency, which is `w(m+1)`.

To continue with our 5th-order Butterworth example, we can now calculate the impulse and step response by using the TRESP routine in which we have included all the above corrections:

```
[z,p]=buttap(5);        % the 5th-order Butterworth system poles
w=(0:1:255)/8;          % form a linearly spaced frequency vector
F=freqw(z,p,w);         % the frequency response at w
[I,t]=tresp(F,w,'i');   % I : ideal impulse, t : normalized time
S=tresp(F,w,'s');       % S : step response ( time same as for I )
plot(t(1:100),I(1:100),t(1:100),S(1:100))
                        % plot 100 points of I and S vs. t
```

The results should look just like Fig. 6.5.1. Here is the TRESP routine:

```
function [y,t]=tresp(F,w,r,g)
%TRESP   Transient RESPonse, using Fast Fourier Transform algorithm.
%    Call : [y,t]=tresp(F,w,r,g);
%    where:
%     F --> complex-frequency response, length-N vector, N=2^B, B=int.
%     w --> can be the related frequency vector of F, or it
%           can be the normalized frequency unit index, or it
%           can be zero and the n.f.u. index is found from F.
%     r --> a character, selects the response type returned in y:
%           - 'u' is the unity area impulse response (the default)
%           - 'i' is the ideal impulse response
%           - 's' is the step response
%     g --> an optional input argument: plot the response graph.
%     y --> the selected system response.
%     t --> the normalized time scale vector.

% Author :  Erik Margan, 880414, Last rev. 000310, Free of copyright!

% ----------- Preparation and checking the input data ------------
if nargin < 3
     r='u';             % select the default response if not specified
end
G=abs(F(1));            % find system DC gain
N=length(F);            % find number of input frequency samples
v=length(w);            % get the length of w
if v == 1
     m=w;               % w is the normalized frequency unit or zero
elseif v == N           % find the normalized frequency unit
     m=find(abs(w-1)==min(abs(w-1)))-1;
     if isempty(m)
          m=0;          % not found, try from the half power bandwidth
     end
else
     error('F and w are expected to be of equal length !');
end
if m == 0               % find the normalized frequency unit index
     m=max(find(abs(F)>=G/sqrt(2)))-1;
end
```

```
% check magnitude slope between the 2nd and 3rd octave above cutoff
M=abs(diff(20*log10(abs(F(1+4*m*[1,2])))));
x=3;                    % system is 3rd-order or higher (>=18dB/2f)
if M < 15
      x=2;              % probably a 2nd-order system (12dB/2f)
elseif M < 9
      x=1;              % probably a 1st-order system (6dB/2f)
end

% ----------- Form the window function --------------------------
if x < 3
      W=0.54-0.46*cos(2*pi*(N+1:2*N)/(2*N)); % right half Hamming
      F=W.*F;           % frequency response windowed
end

% ----------- Normalize the time-scale --------------------------
A=2*pi*m;               % amplitude denormalization factor
dt=A/N;                 % calculate delta-t
if v == N
      dt=dt/w(m+1);     % correct for the actual frequency unit
end
t=dt*(0:1:N-1);         % form the normalized time scale

% ----------- Calculate the impulse response --------------------
y=2*real(fft(conj(F)))-G;     % calculate iFFT and null DC offset
if x == 1
      er1=A*G-y(1);     % fix the 1st-point error for 1st-order system
      y(1)=A*G;
end
if r == 'u' | r == 'U' | r == 's' | r == 'S'
      y=y/N;            % normalize area to G
end

% ----------- Calculate the step response -----------------------
if r == 's' | r == 'S'
      if x == 1
            y=y*(1/(1+er1/N)); % correct 1st-point error
      end
      y=cumsum([0, y]); % integrate to get the step response
      if x > 1
            y=(y(1:N)+y(2:N+1))/2;  % compensate half sample t-delay
            y(1)=0;
      else
            y=y(1:N);
      end
end

% ----------- Normalize the amplitude to ideal ------------------
if r == 'i' | r == 'I'
      y=y/A;            % denormalize impulse amplitude
end

% ----------- Plot the graph ------------------------------------
if nargin == 4
      plot( t, y, '-r' ), xlabel('Time')
      if r == 'i' | r == 'I' | r == 'u' | r == 'U'
            title('Impulse response')
      else
            title('Step response')
      end
end
```

6.5.6 Calculation Errors

Whilst the amount of error in low order system impulse responses might seem small, it would integrate to an unacceptably high level in the step responses if the input data were not windowed. In Fig. 6.5.5 to 6.5.10 we have computed the difference between analytically and numerically calculated impulse and step responses of Butterworth systems, using both the normal and windowed frequency response for the numerical method. Fig. 6.5.5 and 6.5.6 show the impulse and the step response of the 1st-order system, Fig. 6.5.7 and 6.5.8 show the 2nd-order system and Fig. 6.5.9 and 6.5.10 the 3rd-order system. The error plots, shown within each response plot, were magnified 10 or 100 times to reveal the details.

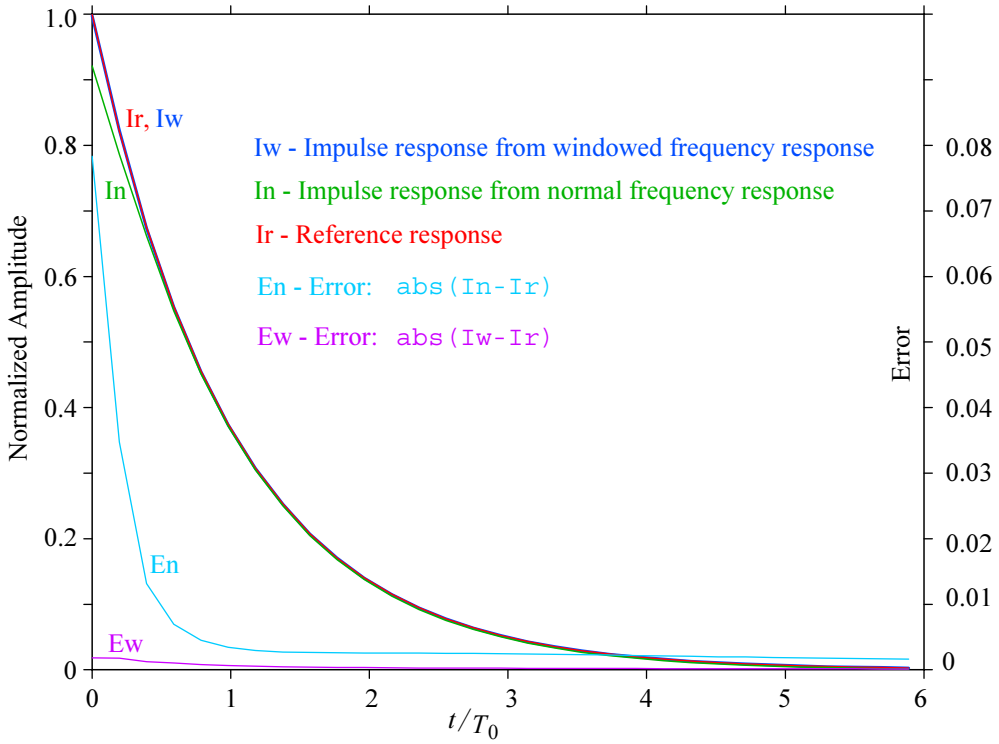
The initial value of the 1st-order system's impulse response, calculated from a non-windowed frequency response, is about 0.08% and falls off quickly with time. Nevertheless, it is about 10× higher than for the impulse response calculated from a windowed response. If the frequency response is not windowed, the impulse responses error eventually integrates to almost 4% of the final value in the step response.

The error plots of the second and higher order systems are much smaller, but they exhibit some decaying oscillations, independently of windowing. This oscillating error is inherent in the FFT method and it is owed to the Gibbs' phenomenon (see Part 1, Sec. 1.2). It can be easily shown that the frequency response of a rectangular time domain window follows a $(\sin x)/x$ curve, and the equivalent holds for the time response of the frequency domain rectangular window (remember Eq. 6.5.7). Since we have deliberately truncated the system's frequency response at the N^{th} sample, we can think of it as being a product of an infinitely long (but finite density) spectrum with a rectangular frequency window. This results in a convolution of the system impulse response with the $(\sin x)/x$ function in the time domain, hence the form of the error in Fig. 6.5.7 to 6.5.10.

This error can be lowered by taking the transform of a longer frequency response vector, but can never be eliminated. For example, if `N=2048` and we specify the frequency vector as: `w=(0:1:N-1)/8`, the error will be 8 times lower than in the case of a frequency vector with `N=256`, but the calculation will last more than 23 times longer (the number of multiplications is proportional to $N\log_2 N$ or 11 times, in addition to 8 times the number of sums and other operations).

As we have stated at the beginning, Sec. 6.0, our aim is to make quick comparisons of the performance of several systems, and on the basis of those decide which system suits us best. Since the resolution of the computer VGA type of graphics is more than adequate for this purpose, and the response error in the case of `N=256` can be seen only if compared with the exact response (and even so as only a one pixel difference), the extra time and memory requirements do not justify the improvement.

A better way of calculating the response more accurately and also directly from system poles and zeros is described in the next section.



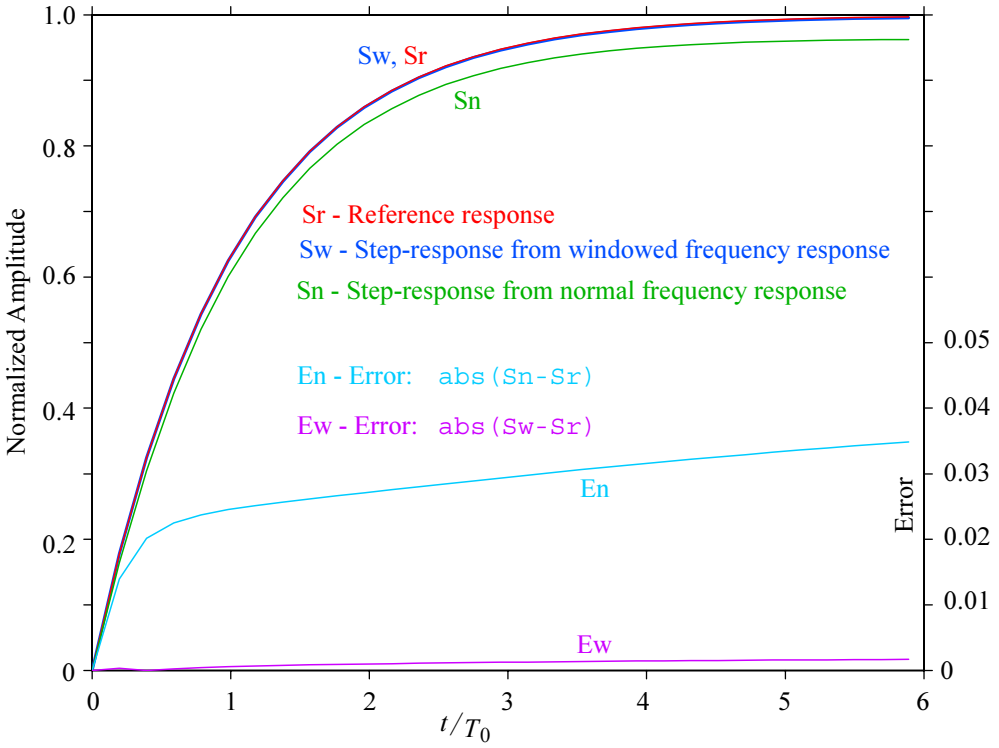
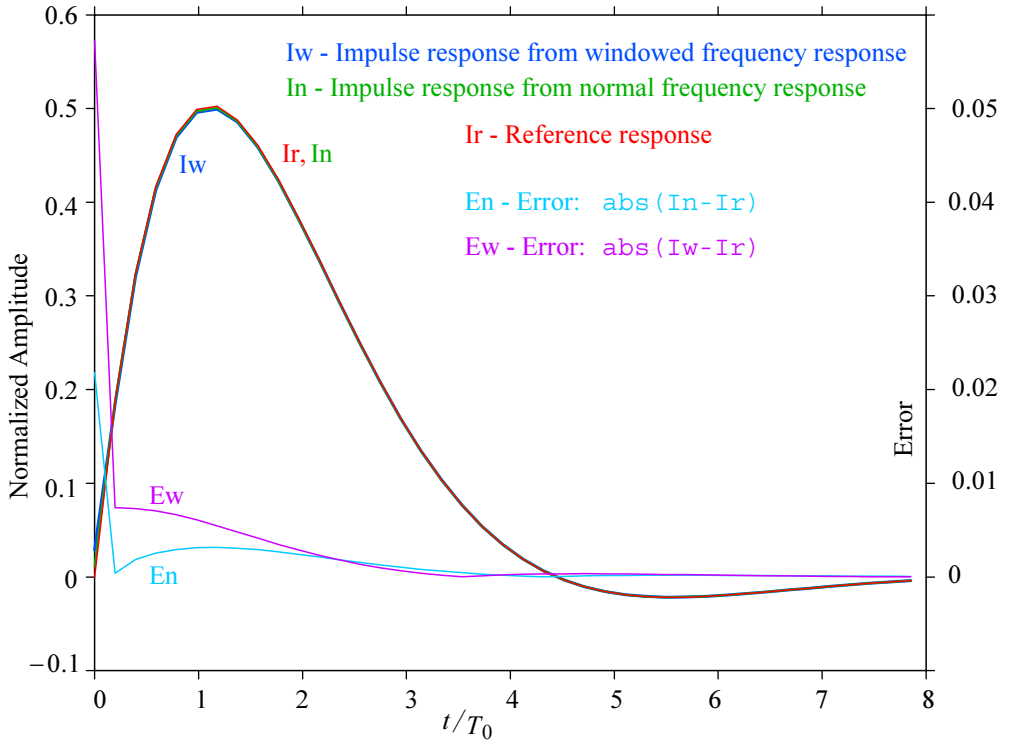


Fig. 6.5.5 and 6.5.6: The first 30 points of a 256 sample long 1st-order impulse and step response vs. the analytically calculated references. The error plots En and Ew are enlarged 10×. Although the impulse response, calculated from the normal frequency response, has a relatively small error, it integrates to an unacceptably high value (4%) in the step response. In contrast, by windowing the frequency response, both time domain errors are much lower, the step response final value is in error by les than 0.2%.



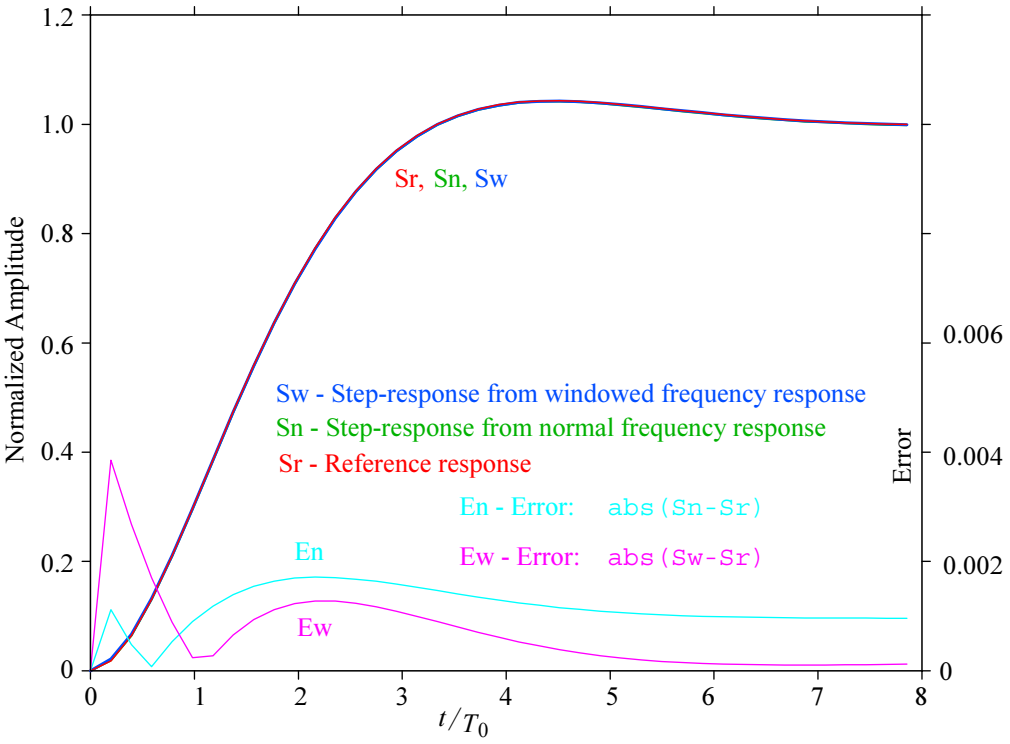
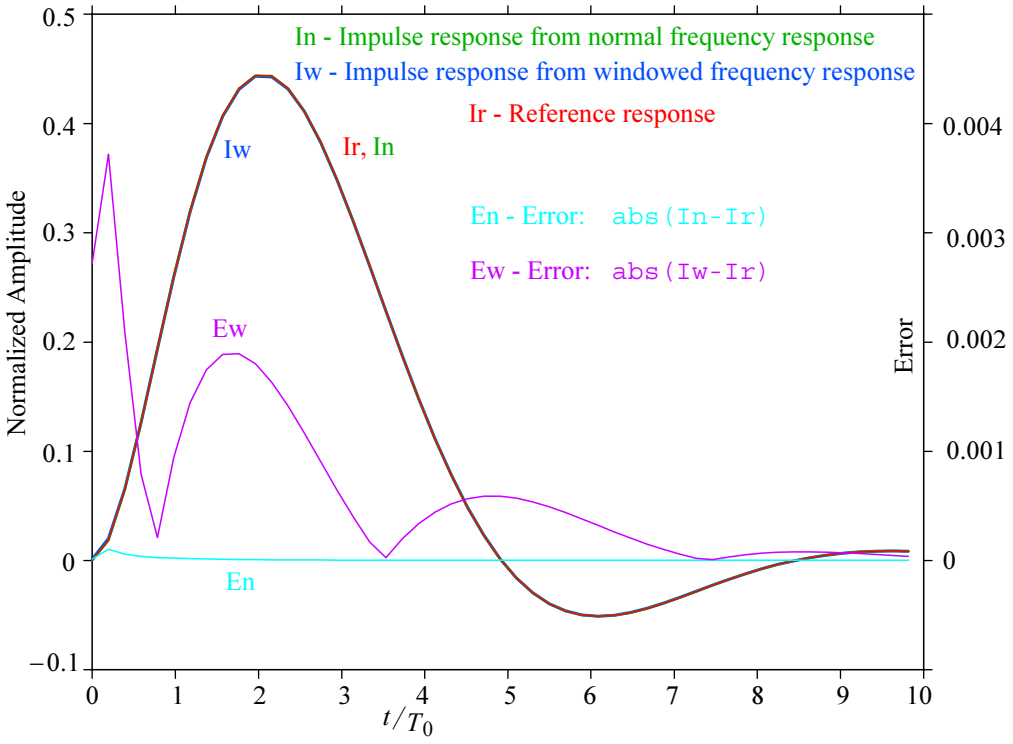


Fig. 6.5.7 and 6.5.8: As in Fig. 6.5.5 and 6.5.6, but with 40 samples of a 2nd-order Butterworth system. The impulse response error for the windowing procedure is higher at the beginning, but falls off more quickly, therefore the step response final value error is still much lower (note that the step response error plots are enlarged 100×). The oscillations in error plots, owed to the Gibbs' effect, also begin to show.



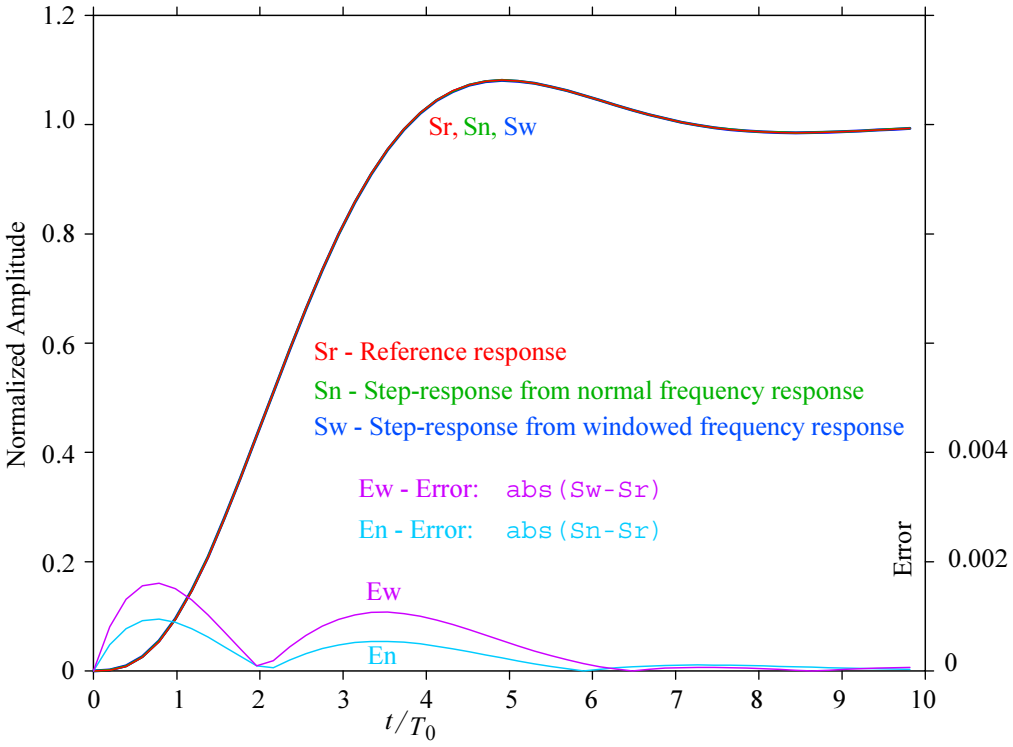


Fig. 6.5.9 and 6.5.10: As in Fig. 6.5.5–8, but with 50 samples of a 3rd-order Butterworth system. Windowing does not help any longer and produces even greater error. The dominant error is now owed to the Gibbs' effect.

6.5.7 Code Execution Efficiency

The TRESP routine executes surprisingly fast. Back in 1987, when these Matlab routines were developed and the first version of this text was written, I was using a 12 MHz PC with an i286-type processor, an i287 math coprocessor, and EGA type of graphics (640×400 resolution, 16 colors). To produce the 10 responses of Fig.6.5.11, starting from the system order, finding the system coefficients, extracting the poles, calculating the complex frequency response, running FFT to obtain the impulse response, integrating for the step response and finally plotting it all on a screen, that old PC worked less than 12 seconds. Today (March 2000), a 500 MHz Pentium-III based processor does it in a few tens of milliseconds (before you can release the ENTER key, once you have pressed it; although, it takes a lot more time for Matlab working under Windows to open the graph window). And note that we are talking about floating point, 'double precision' arithmetic! Nevertheless, being able to make fast calculations has become even more important for embedded instrumentation applications, which require real time data processing and adaptive algorithms.

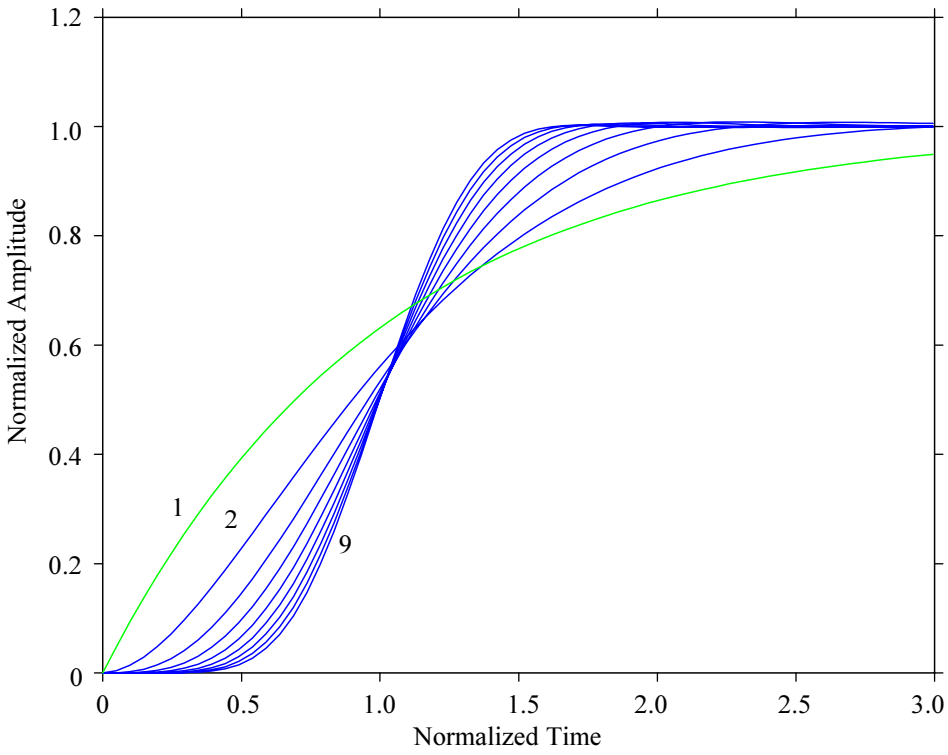


Fig. 6.5.11: Step responses of Bessel–Thomson systems (normalized to equal envelope delay), of order 2 to 9, including the 1st-order step response for comparison. Note the half amplitude delay approaching 1 and the bandwidth improving (shorter rise time) as the system order increases. The TRESP algorithm execution speed was tested by creating this figure.

(blank page)

6.6 Transient Response From Residues

The method of calculating the transient response by FFT, presented in Sec. 6.5, has several advantages over other algorithms. The most important ones are high execution speed, the possibility of computing from either a calculated complex frequency response or from a measured magnitude–phase relationship, and the use of the same FFT algorithm to work both time–to–frequency and frequency–to–time.

Its main disadvantage is the error resulting from the Gibbs' effect, which distorts the most interesting part of the time domain response. This error, although small, can sometimes prevent the system designer from resolving or identifying the cause of possible second-order effects that are spoiling the measured or simulated system performance to which the desired ideal response is being compared. In such a case the designer must have a firm reference, which should not be an approximation in any sense.

The algorithm, presented in this section, with the name ATDR (an acronym of 'Analytical Time Domain Response'), calculates the impulse and step responses by following the same analytical method, that has been used extensively in the previous parts of this book. The routine calculates the residues at each system transfer function pole, and then calculates the final response at specified time instants. However, the residues are not calculated by an actual infinitesimally limiting process, so it is not possible to apply this routine in the most general case (e.g., it fails for systems with coincident poles), but this restriction is not severe, since all of the optimized system families are covered properly. Readers who would like to implement a rigorously universal procedure can obtain the residues calculated by the somewhat longer RESIDUE routine in Matlab.

In contrast to the FFT method, whose execution time is independent of system complexity, this method works more slowly for each additional pole or zero.

A nice feature of this method is that the user has a direct control over the time vector: the response is calculated at exactly those time instants which were specified by the user. This may be important when making comparison with a measured response of an actual system prototype.

As we have seen in numerous examples, solved in the previous parts, a general expression for a residue at a pole p_k of an n^{th}-order system specified by Eq. 6.1.10 can be written like this:

$$r_k = \lim_{p \to p_k} (p - p_k) \, \frac{\prod_{i=1}^{n} (-p_i)}{\prod_{i=1}^{n} (p - p_i)} \cdot \frac{\prod_{j=1}^{m} (p - z_j)}{\prod_{j=1}^{m} (-z_j)} \, \mathrm{e}^{p_k t} \qquad (6.6.1)$$

Here n is the number of poles, m is the number of zeros, p is a vector whose elements are the system's poles p_i, p_k is the k^{th} pole for which the residue r_k is calculated, z_j are the zeros, whilst i and j are the indices.

The terms $(p - p_k)$ cancel for each $i = k$ before limiting. If we now make $p = p_k$, without using the limiting process, the general applicability of Eq. 6.6.1 is lost, but for all optimized system families (no coincident poles!) this will still be valid.

In the ATDR routine we form a vector `Z` of length n, containing the products over the index j of $(p_k - z_j)$ — one (k^{th}) element of `Z` for each residue — and divide these by the product of all transfer function zeros (if there are any).

Next, we form a matrix `D` of n by n elements, each element being a product of $(p - p_i)$. The elements on the diagonal of `D` will all have zero value (on the diagonal $p = p_i$), and since we need the products of the remaining terms, we must eliminate them to avoid multiplying by zero. This results in a `D` of $(n - 1)$ rows by n columns matrix.

In Matlab most matrix operations are designed to perform on columns, producing a single-row vector of results. The same is true for the PROD ('product') command, so `prod(D)` returns a row of n products performed over each column of `D`. This row is returned in `D`, then `Z` is divided by `D`, element by element. The resultant vector is multiplied by the product of all poles to produce the correct scaling factors of the residues, returned in the vector `P`.

If a step response is required, `P` must be divided by the poles p, element by element. Finally, each element of `P` multiplies a vector of (complex) exponential functions of the time vector multiplied by the k^{th} pole.

All the residue values at the same time point are summed (in rows) and each sum is then an element of the real valued result vector (the complex parts cancel to better than 10^{-14} and are neglected). In the case of the step response all values must be increased by 1, the value of the residue of the additional pole at the complex plane's origin resulting from the input step function transform operator $1/s$.

For the impulse response case there are two options: either the result is left as it is, representing a response (implicitly) normalized in amplitude to the response of the same system excited by the ideal (infinite amplitude, infinitely narrow) input impulse, or it can be normalized to represent a unity gain system by dividing each response value by the sum of all values. This is desirable when calculating convolution, etc., but then we have to specify the time vector as sufficiently long (in comparison with the dominant system time constants), to allow the impulse response to decay to a value close enough to zero, thus avoiding a system gain error.

Here is how the now familiar 5$^{\text{th}}$-order Butterworth system responses can be calculated using the ATDR routine:

```
[z,p]=buttap(5);        % 5th-order Butterworth zeros and poles
t=(0:1:300)/15;         % 301-point t-vector, 15 samples in one t-unit
I=atdr(z,p,t,'i');      % ideal impulse response
S=atdr(z,p,t,'s');      % step response
plot(t,I,t,S)           % plot I and S against t
```

The resulting plot should look the same as in Fig. 6.5.1 (but with a much better accuracy!).

```
function y=atdr(z,p,t,q)
%ATDR Analytical Time Domain Response by simplified residue calculus
%     (does not work for systems with multiple poles).
%     y=atdr(z,p,t) or
%     y=atdr(z,p,t,'n') returns the normalized impulse response of a
%                       unity gain system, specified by zeros z and
%                       poles p in time t.
%     y=atdr(z,p,t,'i') returns the impulse response, denormalized to
%                       the ieal impulse input.
%     y=atdr(z,p,t,'s') returns the step response of the system.
%
%     Specify the time as :  t=(0:1:N-1)/T, where N is the number of
%     desired time domain samples and T is the number of samples in
%     the time scale unit, i.e.:  t=(0:1:200)/10

% Author :  Erik Margan, 891008, Free of copyright !

if nargin==3
      q='n' ;     % by default, return the unity gain impuse response
end
n=max(size(p));   % find the number of poles
for k=1:n         % test for repeating poles
      P=p;
      P(k)=[];    % exclude the pole currently tested
      if all(abs(P-p(k)))==0  % is there another such pole ?
            error('ATDR cannot handle systems with repeating poles!')
      end
end
dc=1;             % set low pass system flag
if isempty(z)
      Z=1;        % no zeros
else
      % zeros
      if any( abs(z) <1e-6 )
            dc=0; % HP or BP system, clear dc flag
      end
      if all( abs( real( z ) ) < 1e-6 )
            z = j * imag( z ) ;     % all zeros on imaginary axis
      end
      Z=ones(size(p)) ;
      if dc
            for k=1:n
                  Z(:,k)=prod(p(k)-z)/prod(-z);
            end
      else
            for k = 1:np
                  for h = 1:nz
                        if z(h) == 0
                              Z(k,:) = Z(k,:)*p(k) ;
                        else
                              Z(k,:) = Z(k,:)*(p(k)+z(h))/z(h) ;
                        end
                  end
            end
      end
      Z=Z(:);     % column-wise orientation
end
if n == 1
      D=1;        % single pole case
else
      for k = 1:n
            d=p(k)-p;   % difference, column orientation
            d(k)=[];    % k-th element = 0, eliminate it
            D(:,k)=d;   % k-th column of D
```

```
      end
      if n > 2
            D=(prod(D)); % make column-wise product if D is a matrix
      end
      D=D.';      % column-wise orientation
end
P=prod(-p)*Z./D;  % impulse residues
if q == 's'
      P=P./p;     % if step response is required, divide by p
end
t=t(:).';         % time vector, row orientation
y=P(1)*exp(p(1)*t);     % response, first row
for k = 2:n
      y=[y; P(k)*exp(p(k)*t)];      % next row
      y=sum(y);   % sum column-wise, return a single row
end
y=real(y);        % result is real only (imaginary parts cancel)
if (q == 's') & ( isempty(z) | dc == 1 )
      y=y+1;      % if step resp., add 1 for the pole at 0+j0
end
if ( q == 'i' | q == 'n' ) & ( dc == 0 )
      y=-diff([0, y]);  % impulse response of a high pass system
end
if q == 'n'
      y=y/abs(sum(y));  % normalize impulse resp. to unity gain
end
```

6.7 A Simple Application Example

The algorithms which we have developed allow us now to make a quick comparison of the performance of two equal bandwidth 5th-order systems, a Butterworth and a Bessel–Thomson system. We shall compare the pole loci, the magnitude and the step response. Let us first calculate and plot the poles:

```
[z1,p1]=buttap(5);      % a 5-th order Butterworth zeros and poles
[z2,p2]=bestap(5,'n'); % a 5-th order Bessel system zeros and poles
p1=2*pi*1000*p1;        % denormalize the poles to 1kHz
p2=2*pi*1000*p2;
                        % plot the imag-vs.-real part of poles
plot( real(p1),imag(p1),'*r', real(p2),imag(p2),'*b' )
axis equal square ;     % set axes aspect ratio 1:1
```

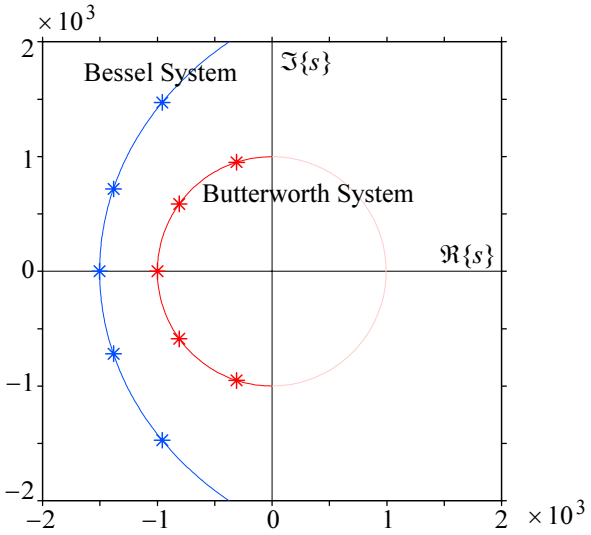


Fig. 6.7.1: The Butterworth poles (on the unit cycle) and the Bessel–Thomson poles (on the fitted ellipse). Note that for the same bandwidth (1 kHz) the values of Bessel–Thomson poles are much larger, but with a lower ratio of the imaginary to the real part.

Let us calculate and plot the frequency responses:

```
f=logspace(2,4,401);    % a log-spaced frequency vector 10^2 - 10^4 Hz
F1=freqw(z1,p1,2*pi*f);% Butterworth frequency response
F2=freqw(z2,p2,2*pi*f);% Bessel-Thomson frequency response
                        % plot the dB magnitude vs. log-frequency :
semilogx(f/1000,20*log10(abs(F1)),'-r',f/1000,20*log10(abs(F2)),'-b')
ylabel('Magnitude'), xlabel('f [kHz]')
```

The frequency response plots are shown in Fig. 6.7.2. Note the equal pass band (−3 dB point) and equal slope at high frequencies. However, the Butterworth system atenuation is an order of magnitude (20 dB) better.

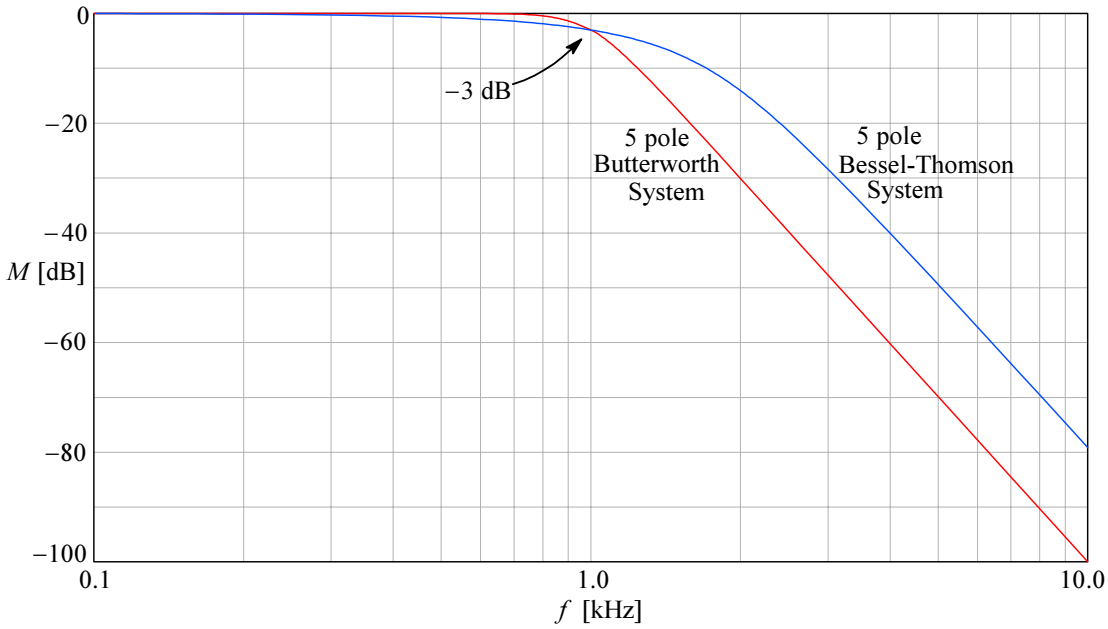


Fig. 6.7.2: Frequency responses of the Butterworth and Bessel–Thomson system. For an equal cut off frequency ($f_h = 1$ kHz), the Butterworth system stop band attenuation is about an order of magnitude (10× or 20 dB) better than that of the Bessel–Thomson.

Using the same poles and the ATDR routine, we compare the step responses:

```
t=(0:1e-5:3);           % the 3ms time vector, 100 samples/ms.
y1=atdr(z1,p1,t,'s');   % Butterworth step response
y2=atdr(z2,p2,t,'s');   % Bessel-Thomson step response
plot(t*1000,y1,'-r',t*1000,y2,'-b'), xlabel('t [ms]') % see Fig.6.7.3
```

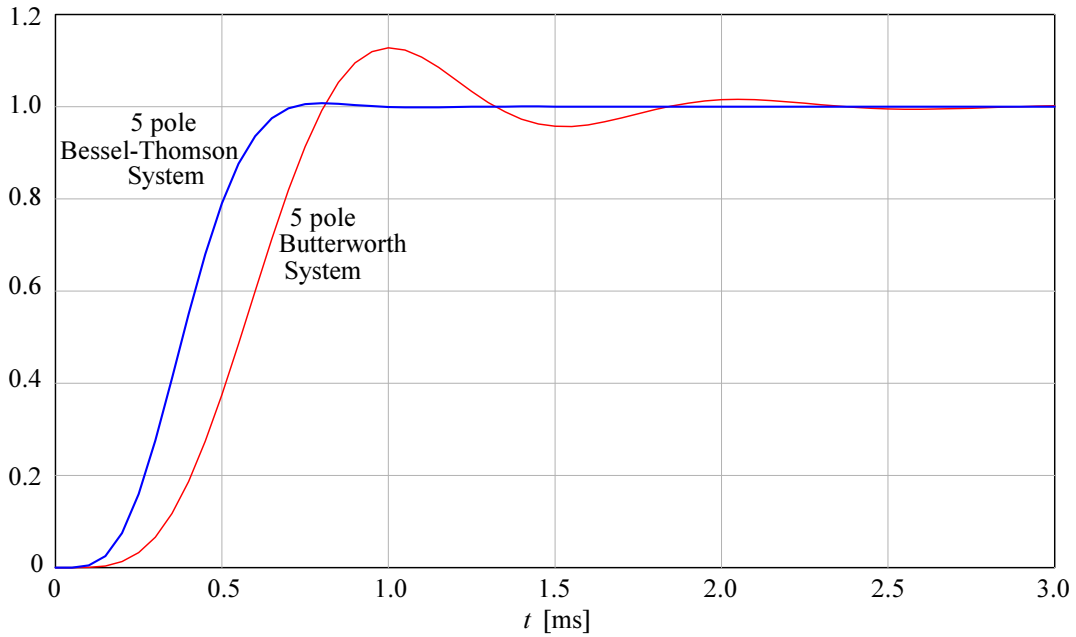


Fig. 6.7.3: Step responses of the Butterworth and Bessel–Thomson system. For the same cut off frequency (1 kHz) the Bessel–Thomson system's delay is smaller; the overshoot is only 0.4% and there is no ringing, so settling down to 0.1% occurs within the first 1 ms. Although the rise times are nearly equal, the Butterworth system is a poor choice if time domain performance is required, since it settles down to 0.1% only after some 5 ms (but Chebyshev and Elliptic filter systems are even much worse in this respect).

Résumé of Part 6

The algorithms shown are small, simple, easy to use, and fast in execution. They are ideal for starting the system's design from scratch, to specify the design goals, as well as to provide a reference with which a realized prototype can be compared.

We have shown how the system performance can easily be evaluated by using the routines developed for Matlab, the prediction of system time domain response in particular. We also hope that the development and application examples of these routines offer a deeper insight on how the system should be designed as a whole.

Still, the reader as the future system's designer is being let down at the most demanding task of finding the circuitry and hardware that will perform as required, and engineering experience is the only help here. This book should help to understand how it might be possible to push the bandwidth up, smooth the transient, and reduce the settling time. But there are also many other important parameters which must be carefully considered when designing an amplifier, such as noise, linearity, electrical and thermal stability, output power, slew rate limiting, the time it takes to recover from overdrive, etc.

However, these parameters (with the exception of electrical stability) are in most cases independent of the system pole and zero locations, but are strongly influenced by the circuit's topology and by the type of active devices used for the realization.

Once the design goals have been set and the circuit configuration selected, performance verification and iterative finalization can then be done using one of the many CAD/CAE programs available on the market.

To see the numerical convolution routine and calculation examples and an actual amplifier–filter system design example calculated using the algorithms developed so far, please turn to Part 7.

(blank page)

References:

[6.1] *J.N. Little*, *C.B. Moller*, PC–MATLAB For Students (containing disks with Matlab program), Prentice–Hall, 1989

[6.2] MATLAB–V For Students (containing CD with Matlab program), Prentice–Hall, 1998

[6.3] The MathWorks, Inc., <http://www.mathworks.com/>

[6.4] *Oliver Heaviside*, Electromagnetic Theory, Chelsea Pub. Co., 3rd edition, 1971, ISBN: 082840237X

[6.5] *Oliver Heaviside*, Electrical Papers Edition Volume 1, American Mathematical Society; January 1970, ASIN: 0821828401

[6.6] *W. A. Atherton*, Pioneers: Oliver Heaviside — Champion of inductance, Wireless World, August 1987, pp. 789–790

[6.7] *Paul J. Nahin*, Oliver Heaviside: The Life, Work, and Times of an Electrical Genius of the Victorian Age, Johns Hopkins Univ. Pr., October 2002, ISBN: 0801869099

[6.8] *Douglas H. Moore*, Heaviside Operational Calculus; An Elementary Foundation, American Elsevier Pub. Co., ASIN: 0444000909

[6.9] *H. Nyquist*, <http://www.ieee.org/organizations/history_center/legacies/nyquist.html>

[6.10] *H.W. Bode*, <http://www.ieee.org/organizations/history_center/legacies/bode.html>

[6.11] *S. Butterworth*, On the Theory of Filter–Amplifiers, Experimental Wireless & The Wireless Engineer, Vol. 7, 1930, pp.536–541

[6.12] *W. E. Thomson*, Networks With Maximally Flat Delay, Wireless Engineer, Vol. 29, October 1952, pp.256–263

[6.13] *L. Storch*, Synthesis of Constant–Time–Delay Ladder Networks Using Bessel Polynomials, Proceedings of the I.R.E., Vol. 42, 1954, pp.1666–1675

[6.14] *O. Föllinger*, Laplace– und Fourier–Transformation, AEG Telefunken (Abt. Verlag), 1982

[6.15] *M. O'Flynn*, *E. Moriarty*, Linear Systems: Time–Domain and Transform Analysis, J. Wiley & Sons, 1987

[6.16] *J.W. Cooley & J.W. Tukey*, An Algorithm for the Machine Calculation of Complex Fourier Series, Math. of Computation, Vol. 19, No. 90, April 1965, pp. 297–301

[6.17] Special Issue on the Fast Fourier Transform, IEEE Transactions on Audio & Electroacoustics, Vol. AU-15, June 1967

[6.18] *E.O. Brigham*, The Fast Fourier Transform, Prentice–Hall, 1974

[6.19] *A.V. Oppenheim*, *R.W. Schafer*, Digital Signal Processing, Prentice–Hall, 1975.

[6.20] *R.I. Ross*, Evaluating the Transient Response of a Network Function, Proceedings of the IEEE, May 1967, pp.693–694.

[6.21] *R.I. Ross*, Iterative Transient Response Calculation Procedures that have Low Storage Requirement, presented at the Second International Symposium on Network Theory, Herceg-Novi, BiH, 1972

[6.22] *J. Vlach*, *K. Singhal*, Computer Methods for Circuit Analysis and Design, Van Nostrand Reinhold, 1983

[6.23] *E. Margan*, Calculating the Transient Response, Elekrotehniški vestnik, Ljubljana, ELVEA2 58, 1991, 1, pp.11–22