PROGRAMS FOR FAST NUMERICAL INVERSION OF LAPLACE TRANSFORMS IN MATLAB LANGUAGE ENVIRONMENT

Lubomír Brančík *

Institute of Theoretical and Experimental Electrical Engineering
Faculty of Electrical Engineering and Computer Science
Brno University of Technology

Abstract

The paper deals with methods for a fast numerical inversion of Laplace transforms developed to run in Matlab language environment. The methods are based on the application of fast Fourier transformation followed by so-called ε–algorithm to speed up the convergence of infinite complex Fourier series. Especially in conjuction with capabilities of the Matlab language created programs seem to be very fast besides they are accurate enough as weel. Examples of the application for a time-domain analysis of transmission lines are presented after respective versions of procedures are explained.

1. Introduction

The methods for numerical inversion of Laplace transforms (*NILT*) rank among ones which are widely used for time-domain simulations e.g. an analysis of transient phenomena in systems containing elements with distributed parameters. From many developed methods those based on *FFT* and ε–algorithm application seem to be convenient from point of view both desired speed and accuracy [1, 2]. Particularly utilizing capabilities of Matlab language the numerical methods under consideration show being very fast to invert not only simple Laplace transforms but also rather complicated systems containing e.g. multiconductor transmission lines. This is enabled due to many operations in Matlab language can run in parallel on multidimensional arrays without necessity to use outer loop structures which leads to essential saving CPU time. In this paper the method elaborated in [1, 2] is summarized and some generalized program versions never published before are also presented. After explaining respective version of a numerical procedure examples of its application are given.

2. Theoretical foundations

On principle to find the original $f(t)$ to a Laplace transform $F(s)$ the definition formula of *ILT* is considered like

$$f(t) = \frac{1}{2\pi j} \int_{c-j\infty}^{c+j\infty} F(s) e^{st} ds \ , \qquad (1)$$

under basic assumption $|f(t)| \le K e^{\alpha t}$, K real positive, α as an exponential order of the real function $f(t)$, $t \ge 0$, and $F(s)$ defined for $\mathrm{Re}[s] > \alpha$.

The task is to evaluate the eq. 1 numerically not only accurate enough but also very fast on a whole given interval $\langle 0; t_m \rangle$. The way of the solution was proposed in [1] where the *FFT* algorithm is applied in order to ensure the high speed of computation, and the method has been improved considerably from point of view getting more accuracy by using ε–algorithm in [2].

Following these results an approximate formula in a discrete form can be written as

$$\tilde{f}^k = C^k \{2\,\mathrm{Re}[\sum_{n=0}^{\infty} F_n E_n^k] - F_0\} \ , \qquad (2)$$

for $k = 0,1,\cdots,N-1$, with

$$\tilde{f}^k = \tilde{f}(kT) \ , \quad C^k = \frac{\Omega}{2\pi} e^{ckT} \ , \quad F_n = F(c - jn\Omega) \ , \quad E_n^k = e^{-jkTn\Omega} \ , \qquad (3)$$

where T and $\Omega = 2\pi/(NT)$ are sampling periods in original and transform domains, respectively. It can be proven the eq. 2 corresponds to a Fourier series approximation of the original $f(t)$ when the error can theoretically be controlled on the interval $t \in \langle 0; NT \rangle$. Practically, however, to suppress an increased error at the end of this interval the required maximum time is supposed to be $t_m = (M-1)T$, with $M = N/2$ as the number of resultant computed points, which leads to the condition of choosing $\Omega = \pi(1 - 1/M)/t_m$. The coefficient c can be setted using the formula

$$c \approx \alpha - \frac{\Omega}{2\pi} \ln E_r \ , \qquad (4)$$

where E_r denotes a desired relative error. To minimize the error towards this theoretical value the infinite sum in eq. 2 must be evaluated as much accurately as possible. The solution then consists of three steps. Firstly this sum is evaluated using only N first terms when a *FFT* algorithm can be applied supposing $N = 2^m$, m integer. Secondly after truncating the result of the *FFT* operation to have only a length M the ε–algorithm is applied to give a precision to the resultant sum. The ε–algorithm uses only a few additional terms above those N used by the *FFT* algorithm, however, the sum becomes as if were evaluated using greatly more terms. Finally the result of the ε–algorithm application is substituted into the eq. 2 to finish the computation. Expressing these operations in vector form the eq. 2 can be rewritten as

$$\tilde{\mathbf{f}}^M = \mathbf{C}^M \circ \{2\,\mathrm{Re}[\mathrm{E}\{FFT(\mathbf{F}^N)\}] - \mathbf{F}_0^M\} \qquad (5)$$

where particular vectors of upper indexed lengths are created according to the eq. 3, namely for $k = 0,1,\cdots,M-1$, $n = 0,1,\cdots,N-1$. Especially $\mathbf{F}_0^M$ is the M–element constant vector of values c. The $\mathrm{E}\{.\}$ designates an operator of the ε–algorithm [3] (including the operation of $N \to M$ vector length reduction), and the symbol $\circ$ means Hadamard product of matrices (in Matlab language called as element-by-element product). The principle of the ε–algorithm can be explained by means of a lozenge diagram in Fig.1.

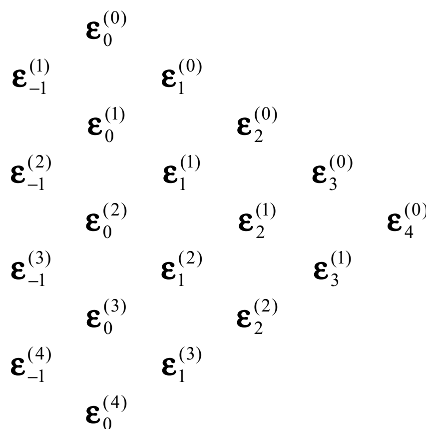

Fig.1 The ε-algorithm lozenge diagram

The first column is formed with $\boldsymbol{\varepsilon}_{-1}^{(s)} = \mathbf{0}^M$, $s = 1,2,3\cdots$, where $\mathbf{0}^M$ means an M-element zero vector. The second column represents partial sums computed recurrently as

$$\boldsymbol{\varepsilon}_0^{(s+1)} = \boldsymbol{\varepsilon}_0^{(s)} + F_{N+s}\mathbf{E}_{N+s}^M , \qquad s = 0,1,2\cdots, \tag{6}$$

where $\mathbf{E}_n^M$ is an M–element vector created according to the exponential term E_n^k in eq. 3, for $k = 0,1,\cdots,M-1$ and a given n, and the initial $\boldsymbol{\varepsilon}_0^{(0)}$ term is the result of the *FFT* operation truncated to have the length M. The leftover columns are computed using the formula like

$$\boldsymbol{\varepsilon}_{r+1}^{(s)} = \boldsymbol{\varepsilon}_{r-1}^{(s+1)} + [\boldsymbol{\varepsilon}_r^{(s+1)} - \boldsymbol{\varepsilon}_r^{(s)}]^{-1} , \quad r,s = 0,1,2\cdots, \tag{7}$$

when the inversion operation is supposed to run component-wise. Thus the sequence of successive approximations $\boldsymbol{\varepsilon}_0^{(0)}$, $\boldsymbol{\varepsilon}_2^{(0)}$, $\boldsymbol{\varepsilon}_4^{(0)},\cdots$ converges usually much more quickly than the original sequence of partial sums. Supposing to start a computation using $2P+1$ partial sums the $\boldsymbol{\varepsilon}_{2P}^{(0)}$ term is the required result of the ε-algorithm. However, this algorithm is under a numerical instability if P is chosen too big. From many experiments it seems $P = 2$ or 3 are good choices. The first case is completely shown in Fig.1. This lozenge diagram will further be utilized to explain generalized versions of the *NILT* procedure, see later.

3. Matlab language implementation – basic version

The Matlab functions intended to invert transforms defined by subfunctions whose names are typed in the form of strings are presented. These subfunctions can be defined at the end of the body of respective *NILT* M-file function or in separate M-files under necessary names.

```
%*********************** NILT–FUNCTION DEFINITION  (basic version) *********************%
  function [ft,t]=nilt(F,tm);
  alfa=0; M=256; P=2;
  N=2*M; wyn=2*P+1;
  t=linspace(0,tm,M);
  NT=2*tm*N/(N-2); omega=2*pi/NT;
  c=alfa+25/NT; s=c-i*omega*(0:N+wyn-2);
  Fsc=feval(F,s);
  ft=fft(Fsc(1:N)); ft=ft(1:M);
  for n=N:N+wyn-2
     ft(n-N+2,:)=Fsc(n+1)*exp(-i*n*omega*t);
  end
  ft1=cumsum(ft); ft2=zeros(wyn-1,M);
  for l=1:wyn-2
     ft=ft2+1./diff(ft1);
     ft2=ft1(2:wyn-l,:); ft1=ft;
  end
  ft=ft2+1./diff(ft1); ft=2*real(ft)-Fsc(1);
  ft=exp(c*t)/NT.*ft; ft(1)=2*ft(1);
  plot(t,ft); xlabel('t'); ylabel('f(t)'); grid on;

%***************************** F1–SUBFUNCTION DEFINITION *****************************%
  function f=F1(s)                  %   subfunction F1 is called like this: nilt('F1',tm);
     f=F(s);                        %   F(s) transform evaluation
%*************************************************************************************%
```

In the first line of the body of the function the exponential order α, the number of points to be plotted $M = 2^m$, m integer, and P parameter of the ε-algorithm can be changed if necessary.

The eq. 4 is incorporated into the function to ensure the value $\ln E_r = -25$ which means the relative error about 10^{-11} could be expected if the ε-algorithm is efficient enough. In [2] there have been tested relative errors for several common transforms with known originals and confirmed that such an error can really be achieved with the exception of a vicinity of discontinuities and the origin of the interval. In the body of a subfunction it is suitable to use the operations .*, ./ and .^ for a multiplication, division and power, respectively, i.e. those running on element-by-element basis, to make the evaluation as fast as possible. All common built-in functions run by this way automatically. Othervise a loop structure would have to be used to cycle through all the complex frequency elements of the vector variable s leading to much slower computation.

3.1. Examples of the application

The presented program will be used for the time-domain simulation of lossy transmission lines. Suppose the operator model of the linear system with a transmission line in Fig.2.

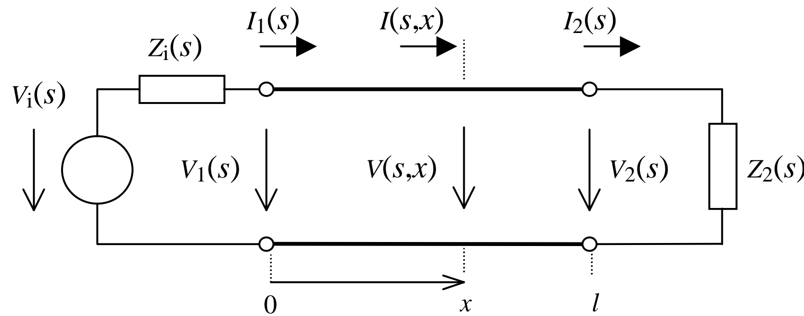

Fig. 2 Linear system with lossy transmission line

The line is uniform of the length l and is described with per-unit-length primary parameters R_0, L_0, G_0 and C_0. The voltage and the current can be expressed in operator forms as

$$V(s,x) = V_i(s)\frac{Z_v(s)}{Z_i(s)+Z_v(s)}\cdot\frac{e^{-\gamma(s)x}+\rho_2(s)e^{-\gamma(s)[2l-x]}}{1-\rho_1(s)\rho_2(s)e^{-2\gamma(s)l}} \quad , \tag{8}$$

$$I(s,x) = V_i(s)\frac{1}{Z_i(s)+Z_v(s)}\cdot\frac{e^{-\gamma(s)x}-\rho_2(s)e^{-\gamma(s)[2l-x]}}{1-\rho_1(s)\rho_2(s)e^{-2\gamma(s)l}} \quad , \tag{9}$$

where

$$Z_v(s) = \sqrt{Z(s)/Y(s)} \quad (10) \quad , \qquad \gamma(s) = \sqrt{Z(s)Y(s)} \quad (11)$$

are the characteristic impedance and the propagation constant, respectively, with

$$Z(s) = R_0 + sL_0 \quad (12) \quad , \qquad Y(s) = G_0 + sC_0 \quad (13)$$

as the series impedance and the shunting admittance, respectively, and

$$\rho_1(s) = \frac{Z_i(s)-Z_v(s)}{Z_i(s)+Z_v(s)} \quad (14) \quad , \qquad \rho_2(s) = \frac{Z_2(s)-Z_v(s)}{Z_2(s)+Z_v(s)} \quad (15)$$

designate the reflection coefficients at the near and far ends of the line, respectively.

In general case the time-domain solution cannot be expressed in an analytical form, therefore, the only way is to use a numerical method. Consider the line has a length $l = 1$ and per-unit-length primary parameters in normalized forms $R_0 = 0.5$, $L_0 = 4$, $G_0 = 0.1$, $C_0 = 1$. The terminating impedances are $Z_i(s) = 0$, $Z_2(s) = 10$, the input voltage has the form of

a sin square puls $v_i(t) = \sin^2(\pi t)$, $0 \le t \le 1$, $v_i(t) = 0$ otherwise, with the Laplace transform

$$V_i(s) = \frac{2\pi^2(1-e^{-s})}{s(s^2+4\pi^2)} \quad . \tag{16}$$

Then, for example, in order to find the time-domain solution for the voltage in the middle of the transmission line, i.e. to find $v(t, l/2)$, a called subfunction can be of the form as follows

```
%****************************** Vx–SUBFUNCTION DEFINITION ******************************%
function f=Vx(s)
  l=1; x=l/2;
  Ro=0.5; Lo=4; Go=0.1; Co=1;
  Vi=2*pi^2*(1-exp(-s))./s./(s.^2+4*pi^2);
  Zi=0; Z2=10;
  Z=Ro+s*Lo; Y=Go+s*Co;
  Zv=sqrt(Z./Y); gam=sqrt(Z.*Y);
  ro1=(Zi-Zv)./(Zi+Zv); ro2=(Z2-Zv)./(Z2+Zv);
  Vx=Vi.*Zv./(Zi+Zv).*(exp(-gam*x)+ro2.*exp(-gam*(2*l-x)))./(1-ro1.*ro2.*exp(-2*gam*l));
  f=Vx;
%******************************************************************************************%
```

To compute and plot this waveform e.g. on the interval $t \in (0;12\rangle$ the main *NILT* function is called with the input parameters like this: nilt('Vx',12). Similarly a subfunction for the current computation can be created replacing Vx for Ix according to eq. 9. The results are in Fig.3, the CPU times were under 50*ms* on PC with Pentium II 266MHz processor.

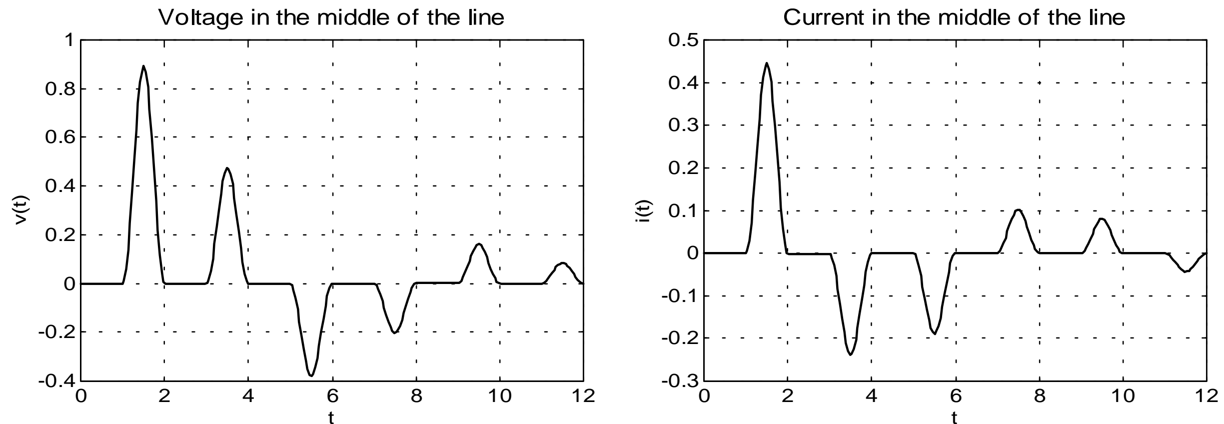

Fig.3 Voltage and current waveforms in the middle of the line

The presented functions can easily be modified by various ways. For example, in order to simplify entering desired *x*–coordinate another input parameter can be introduced, and so on.

4. Generalized *NILT* function – vector version

In the following the above described basic version of *NILT* function will be generalized to be able to invert Laplace transforms in vector form. It can advantageously be utilized in the cases like above presented when calling main *NILT* function need not be repeated, and when respective transforms are evaluated in advance. The own inversion process runs then in parallel for all the precalculated transforms having been formed into a vector. Suppose the transform $F(s)$ in the definition formula 1 is replaced with a column vector like

$$\mathbf{F}^J(s) = [{}^1F(s), {}^2F(s), \cdots {}^JF(s)]^T \quad . \tag{17}$$

Then an approximate formula in a matrix form corresponding to the eq. 5 can be written as

$$\widetilde{\mathbf{f}}^{J\times M} = \mathbf{C}^{J\times M} \circ \{2\,\mathrm{Re}[\mathrm{E}\{\underset{<2>}{FFT}(\mathbf{F}^{J\times N})\}] - \mathbf{F}_0^{J\times M}\} \tag{18}$$

where all the terms are matrices of upper indexed dimensions computed according to eq. 3, but formed for all the components ${}^{j}F(s)$, $j = 1,2,\ldots,J$, of the vector $\mathbf{F}^J(s)$. The subscript $<2>$ means the *FFT* operation runs along the 2nd dimension (columns) but in parallel for all the rows. The ε–algorithm runs by the same way as explained earlier, but then all the terms in the lozenge diagram in Fig.1 are $J\times M$ matrices. Thus $\boldsymbol{\varepsilon}_{-1}^{(s)} = \mathbf{0}^{J\times M}$, $s = 1,2,3\cdots$, zero matrices form the first column of the lozenge diagram, the second column is formed with partial sums computed recurrently as follows

$$\boldsymbol{\varepsilon}_0^{(s+1)} = \boldsymbol{\varepsilon}_0^{(s)} + \mathbf{F}_{N+s}^{J} \otimes \mathbf{E}_{N+s}^{M}\ , \qquad s = 0,1,2\cdots, \tag{19}$$

where $\mathbf{F}_n^J$ is the *J*-element column vector in eq. 17 evaluated for a given *n* according to eq. 3, $\mathbf{E}_n^M$ is the *M*–element row vector created as explained in par. 2, and the initial $\boldsymbol{\varepsilon}_0^{(0)}$ term is the result of the *FFT* operation truncated to have the size $J\times M$. Finally a symbol $\otimes$ designates so-called Kronecker tensor product of matrices.

The corresponding Matlab function contains the third parameter 'pl' typed in the form of the string to choose a method of plotting – either into a single figure as multiple plot ('pl1') or into necessary number of separate figures as individual plots ('pl2') .

```
%********************** NILTV–FUNCTION DEFINITION  (vector version) *********************%
function [ft,t]=niltv(F,tm,pl);
global ft t;
alfa=0; M=256; P=2;
N=2*M; wyn=2*P+1;
t=linspace(0,tm,M);
NT=2*tm*N/(N-2); omega=2*pi/NT;
c=alfa+25/NT; s=c-i*omega*(0:N+wyn-2);
Fsc=feval(F,s);
ft=fft(Fsc(:,1:N),[],2); ft=ft(:,1:M);
for n=N:N+wyn-2
   ft(:,:,n-N+2)=kron(Fsc(:,n+1),exp(-i*n*omega*t));
end
ft1=cumsum(ft,3); ft2=zeros(size(Fsc,1),M,wyn-1);
for l=1:wyn-2
   ft=ft2+1./diff(ft1,1,3);
   ft2=ft1(:,:,2:wyn-l); ft1=ft;
end
ft=ft2+1./diff(ft1,1,3);
ft=2*real(ft)-repmat(Fsc(:,1),[1,M]);
ft=repmat(exp(c*t)/NT,[size(Fsc,1),1]).*ft; ft(:,1)=2*ft(:,1);
feval(pl);

function pl1                                    %    multiple plotting into a single figure
   global ft t;
   plot(t,ft); xlabel('t'); ylabel('f(t)'); grid on;

function pl2                                    %    plotting into separate figures
   global ft t;
   for k=1:size(ft,1)
      figure;
      plot(t,ft(k,:)); xlabel('t'); ylabel('f(t)'); grid on;
   end
%**************************************************************************************%
```

```
%***************************** Fv1–SUBFUNCTION  DEFINITION ****************************%
function f=Fv1(s)
   f(1,:)=F1(s);                %     vector transform is evaluated component-wise
   f(2,:)=F2(s);
   f(3,:)=F3(s);
      . . . .
%***************************************************************************************************%
```

The vector transform evaluation can be performed not only component-wise as shown above but also by putting together vectors resulted from some matrix operations when complex frequency components are cycled in a loop structure, and finally also as a result of a single function evaluation in the case its arguments were replaced with those prepared by meshgrid function.

Thus to get e.g. the voltage waveforms at the near and far ends of the line together in a single figure the main *NILT* function is called like this: niltv('V12',12,'pl1') and the subfunction 'V12' can be of the form as follows

```
%***************************** V12–SUBFUNCTION DEFINITION  ****************************%
function f=V12(s)
   l=1;
   Ro=0.5; Lo=4; Go=0.1; Co=1;
   Vi=2*pi^2*(1-exp(-s))./s./(s.^2+4*pi^2);
   Zi=0; Z2=10;
   Z=Ro+s*Lo; Y=Go+s*Co;
   Zv=sqrt(Z./Y); gam=sqrt(Z.*Y);
   ro1=(Zi-Zv)./(Zi+Zv); ro2=(Z2-Zv)./(Z2+Zv);
   K=Vi.*Zv./(Zi+Zv)./(1-ro1.*ro2.*exp(-2*gam*l));
   x=0;
   V1=K.*(exp(-gam*x)+ro2.*exp(-gam*(2*l-x)));
   x=l;
   V2=K.*(exp(-gam*x)+ro2.*exp(-gam*(2*l-x)));
   f(1,:)=V1;
   f(2,:)=V2;
%***************************************************************************************************%
```

Similarly the results can be obtained for the currents at both ends of the line, see Fig.4.

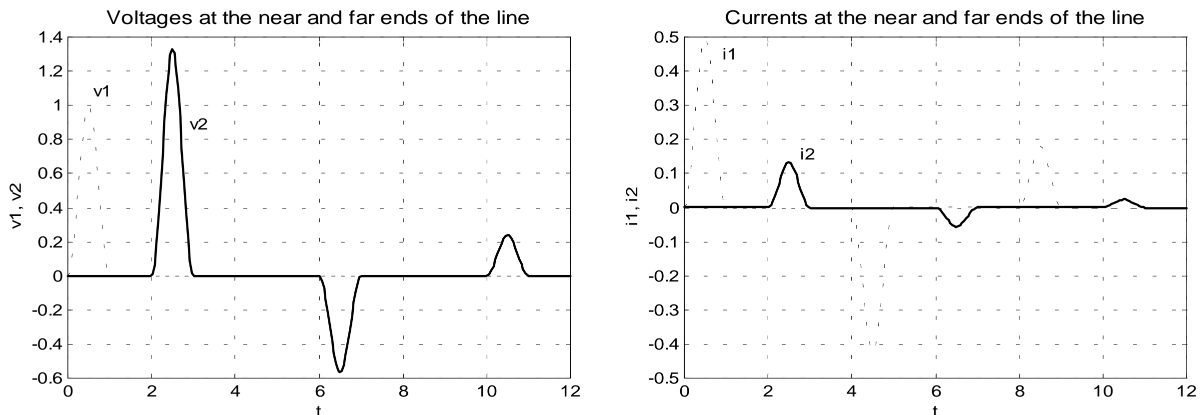

Fig.4 Voltages and currents at the near and far ends of the line

Finally the vector version of *NILT* function can easily be used to compute voltage or current wave propagations along the line to display them three-dimensionally. For this purpose another plot 'pl3' function and a called subfunction 'Vx3' can be created as follows

```
%*********************************** PLOT FUNCTION pl3 ************************************%
function pl3
   global ft t x;                              %    x must also be global in Vx3
   m=length(t); tgr=[1:m/64:m,m];              %    only 65 time points is used for plotting
   mesh(t(tgr),x,ft(:,tgr));
   xlabel('t'); ylabel('x'); zlabel('f(t,x)');

%**************************** Vx3–SUBFUNCTION DEFINITION *****************************%
function f=Vx3(s)
   global x;                                   %    x must also be global in pl3
   l=1;  Ro=0.5; Lo=4; Co=1; Go=0.1;
   x=linspace(0,l,65);                         %    65 coordinate points is used for plotting
   [S,X]=meshgrid(s,x);
   Zi=0; Z2=10;
   Vi=2*pi^2*(1-exp(-S))./S./(S.^2+4*pi^2);
   Z=Ro+S*Lo; Y=Go+S*Co;
   Zv=sqrt(Z./Y); gam=sqrt(Z.*Y);
   ro1=(Zi-Zv)./(Zi+Zv); ro2=(Z2-Zv)./(Z2+Zv);
   Vx=Vi.*Zv./(Zi+Zv).*(exp(-gam.*X)+ro2.*exp(-gam.*(2*l-X)))./(1-ro1.*ro2.*exp(-2*gam*l));
   f=Vx;
%*******************************************************************************************%
```

Here a different approach of programming is shown when meshgrid function is applied to get the result without using any loop structure although that could have also been used. The main vector *NILT* function is called like this: niltv('Vx3',12,'pl3'), when the CPU time was about 2 seconds. Similarly 3D graph can be got for the current wave propagation, see Fig.5.

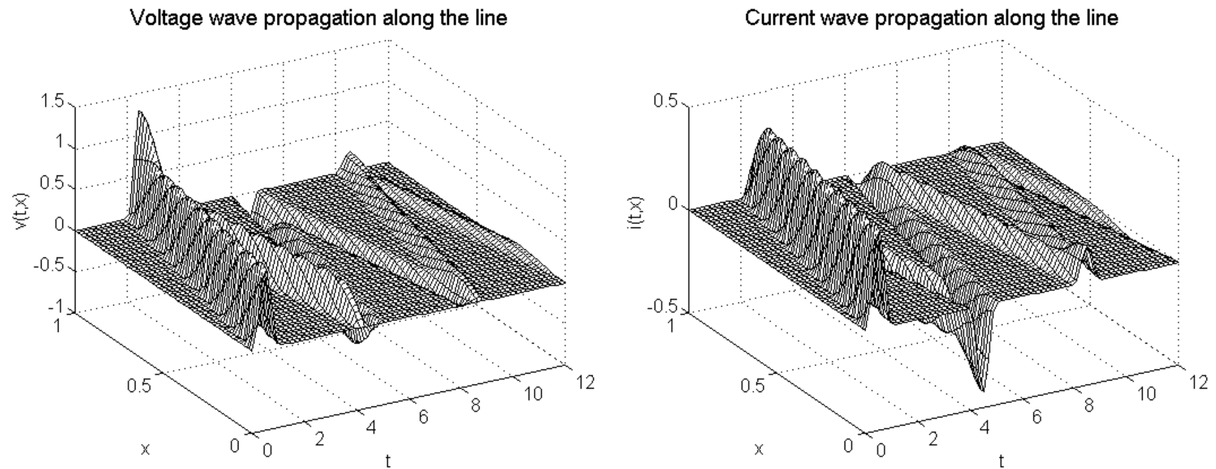

Fig.5 Voltage and current wave propagations along the line

5. Generalized *NILT* function – matrix version

In the cases when e.g. multiconductor lines are simulated there is a need to invert transforms of a matrix form like

$$\mathbf{F}^{J\times L}(s)=\begin{bmatrix} {}^{11}F(s) & {}^{12}F(s) & \cdots & {}^{1L}F(s) \\ {}^{21}F(s) & {}^{22}F(s) & \cdots & {}^{2L}F(s) \\ \vdots & \vdots & & \vdots \\ {}^{J1}F(s) & {}^{J2}F(s) & \cdots & {}^{JL}F(s) \end{bmatrix} . \tag{20}$$

The approximate formula can now be expressed using three-dimensional arrays like

$$\widetilde{\mathbf{f}}^{J\times M\times L} = \mathbf{C}^{J\times M\times L} \circ \{2\,\mathrm{Re}[\,\mathrm{E}\{\underset{<2>}{FFT}(\mathbf{F}^{J\times N\times L})\}] - \mathbf{F}_0^{J\times M\times L}\}\,. \tag{21}$$

Here *FFT* operation runs again along the second dimension as the transform matrix to be inverted is situated along the first and the third dimensions, but this is not the only way to arrange the solution. While in the vector version of the *NILT* procedure the ε–algorithm runs into the third dimension here the fourth one will be used. This enable necessary operations to be able to run in parallel over remaining dimensions and thus to save the CPU time in maximum measure. Without any further mathematical description the main *NILT* Matlab function and a called subfunction are presented below.

```
%*********************** NILTM-FUNCTION DEFINITION  (matrix version) ********************%
function [ft,t,x]=niltm(F,tm,pl);
global ft t x;
alfa=0; M=256; P=2;
N=2*M; wyn=2*P+1;
t=linspace(0,tm,M);
NT=2*tm*N/(N-2); omega=2*pi/NT;
c=alfa+25/NT; s=c-i*omega*(0:N+wyn-2);
Fsc=feval(F,s);
ft=fft(Fsc(:,1:N,:),[],2); ft=ft(:,1:M,:);
dim1=size(Fsc,1); dim3=size(Fsc,3);
for n=N:N+wyn-2
   ft(:,:,:,n-N+2)=reshape(kron(Fsc(:,n+1,:),exp(-i*n*omega*t)),dim1,M,dim3);
end
ft1=cumsum(ft,4); ft2=zeros(dim1,M,dim3,wyn-1);
for l=1:wyn-2
   ft=ft2+1./diff(ft1,1,4);
   ft2=ft1(:,:,:,2:wyn-l); ft1=ft;
end
ft=ft2+1./diff(ft1,1,4);
ft=2*real(ft)-repmat(real(Fsc(:,1,:)),[1,M,1]);
ft=repmat(exp(c*t)/NT,[dim1,1,dim3]).*ft; ft(:,1,:)=2*ft(:,1,:);
feval(pl);
%*******************************************************************************************%
```

```
%**************************** Fm1–SUBFUNCTION  DEFINITION ****************************%
function f=Fm1(s)
   f(1,:,1)=F11(s)             %    vector transform is evaluated component-wise
   f(1,:,2)=F12(s)
         . . .
   f(2,:,1)=F21(s)
   f(2,:,2)=F22(s)
         . . .
%*******************************************************************************************%
```

Again more likely some other method of the evaluation of the called subfunction will often be used, e.g. the method based on using meshgrid function or putting together matrices while the complex frequency components are cycled in a loop structure. The main *NILT* function must be joined with an appropriate plot function like e.g. 'pl4' function shown below.

For example, to be able to compute both three-dimensional graphs in Fig.5 in parallel the called subfunction can have the form as the 'Vlx3' function has. The main *NILT* function is then called like this: niltm('Vlx3',12,'pl4'). For this example the CPU time was about 3 seconds.

```
%*********************************** PLOT FUNCTION pl4 ************************************%
function pl4
   global ft t x;                               %    x must also be global in VIx3
   m=length(t); tgr=[1:m/64:m,m];               %    65 coordinate points is used for plotting
   for k=1:size(ft,3)
      figure;
      mesh(t(tgr),x,ft(:,tgr,k));
      xlabel('t'); ylabel('x'); zlabel(strcat('f',num2str(k),'(t,x)'));
   end
%*******************************************************************************************%

%***************************** VIx3–SUBFUNCTION DEFINITION *****************************%
function f=VIx3(s)
   global x;                                    %    x must also be global in pl4
   l=1;  Ro=0.5; Lo=4; Co=1; Go=0.1;
   x=linspace(0,l,65);
   [S,X]=meshgrid(s,x);
   Zi=0; Z2=10;
   Vi=2*pi^2*(1-exp(-S))./S./(S.^2+4*pi^2);
   Z=Ro+S*Lo; Y=Go+S*Co;
   Zv=sqrt(Z./Y); gam=sqrt(Z.*Y);
   ro1=(Zi-Zv)./(Zi+Zv); ro2=(Z2-Zv)./(Z2+Zv);
   K=Vi./(Zi+Zv)./(1-ro1.*ro2.*exp(-2*gam*l));
   Vx=K.*Zv.*(exp(-gam.*X)+ro2.*exp(-gam.*(2*l-X)));
   Ix=K.*(exp(-gam.*X)-ro2.*exp(-gam.*(2*l-X)));
   f(:,:,1)=Vx;
   f(:,:,2)=Ix;
%*******************************************************************************************%
```

5.1. Application to multiconductor transmission line analysis

The last generalized matrix version of the *NILT* function can successfully be used for the simulation of transient phenomena on multiconductor transmission lines (*MTL*) [4]. Here only an example of its application for the case of an uniform *MTL* with zero initial conditions will be presented although more general cases can be taken into account, see e.g. [5].

Consider a (3+1)-conductor transmission line system according to Fig.6 [6].

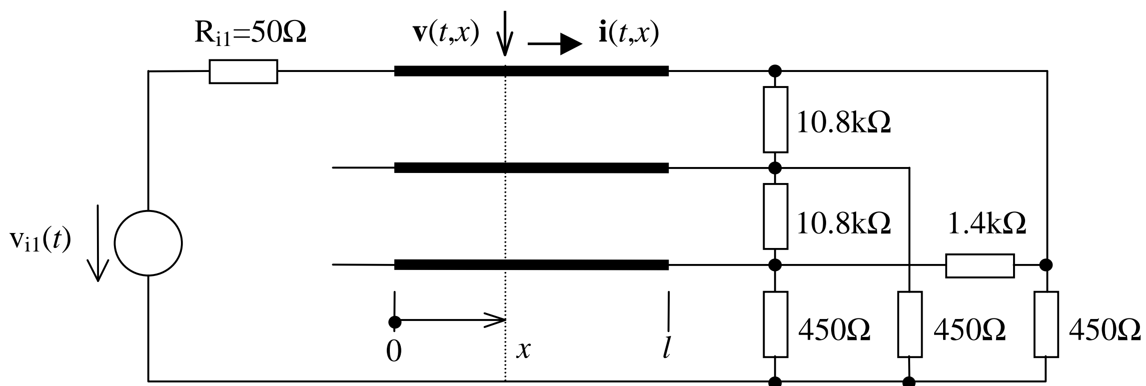

Fig.6 The (3+1)-conductor transmission line system

The line has the length $l = 0.70m$, the per-unit-length matrices are of the forms as shown below [6]

$$\mathbf{L}_0 = \begin{bmatrix} 2.4 & 0.69 & 0.64 \\ 0.69 & 2.36 & 0.69 \\ 0.64 & 0.69 & 2.4 \end{bmatrix} \frac{\mu H}{m} \quad , \quad \mathbf{R}_0 = \begin{bmatrix} 41.67 & 0 & 0 \\ 0 & 41.67 & 0 \\ 0 & 0 & 41.67 \end{bmatrix} \frac{\Omega}{m},$$

$$\mathbf{C}_0 = \begin{bmatrix} 21 & -12 & -4 \\ -12 & 26 & -12 \\ -4 & -12 & 21 \end{bmatrix} \frac{pF}{m} \quad , \quad \mathbf{G}_0 = \begin{bmatrix} 0.6 & 0 & 0 \\ 0 & 0.6 & 0 \\ 0 & 0 & 0.6 \end{bmatrix} \frac{mS}{m} . \tag{22}$$

The first wire is driven with the unit step voltage with the leading edge $0.1ns$. As remaining wires at the left end of the line are open a multiport model based on generalized Norton equivalents can be used to incorporate boundary conditions.

Generally for $(n+1)$-conductor transmission line if zero initial conditions are considered the Laplace transforms of $n \times 1$ voltage and current vectors $\mathbf{V}(s,x)$ and $\mathbf{I}(s,x)$ at a distance x from the left end of the line can be expressed in a compact matrix form as [4]

$$\begin{bmatrix} \mathbf{V}(s,x) \\ \mathbf{I}(s,x) \end{bmatrix} = \begin{bmatrix} \mathbf{\Phi}_{11}(s,x) & \mathbf{\Phi}_{12}(s,x) \\ \mathbf{\Phi}_{21}(s,x) & \mathbf{\Phi}_{22}(s,x) \end{bmatrix} \cdot \begin{bmatrix} \mathbf{V}_{\mathrm{L}}(s) \\ \mathbf{I}_{\mathrm{L}}(s) \end{bmatrix} , \tag{23}$$

where $\mathbf{\Phi}_{\mathrm{ij}}(s,x)$, $\mathrm{i,j} = 1,2$, are $n \times n$ square submatrices of a transition matrix $\mathbf{\Phi}(s,x)$ computed through a matrix exponential function like

$$\mathbf{\Phi}(s,x) = \exp\left(\begin{bmatrix} \mathbf{0} & -\mathbf{Z}(s) \\ -\mathbf{Y}(s) & \mathbf{0} \end{bmatrix} \cdot x \right), \tag{24}$$

where $\mathbf{0}$ means $n \times n$ zero matrix, and

$$\mathbf{Z}(s) = \mathbf{R}_0 + s\mathbf{L}_0 \tag{25}$$

$$\mathbf{Y}(s) = \mathbf{G}_0 + s\mathbf{C}_0 \tag{26}$$

are $n \times n$ series impedance and shunting admittance matrices, respectively. Using generalized Norton equivalents the formulae for vectors $\mathbf{V}_{\mathrm{L}}(s) = \mathbf{V}(s,0)$ and $\mathbf{I}_{\mathrm{L}}(s) = \mathbf{I}(s,0)$ at the left end of the line can be written like [4]

$$\begin{aligned} \mathbf{V}_{\mathrm{L}}(s) = {} & \{[\mathbf{\Phi}_{22}(s) - \mathbf{Y}_{\mathrm{iR}}(s)\mathbf{\Phi}_{12}(s)]\mathbf{Y}_{\mathrm{iL}}(s) + \mathbf{Y}_{\mathrm{iR}}(s)\mathbf{\Phi}_{11}(s) - \mathbf{\Phi}_{21}(s)\}^{-1} \\ & \cdot \{[\mathbf{\Phi}_{22}(s) - \mathbf{Y}_{\mathrm{iR}}(s)\mathbf{\Phi}_{12}(s)]\mathbf{I}_{\mathrm{iL}}(s) + \mathbf{I}_{\mathrm{iR}}(s)\} \end{aligned} , \tag{27}$$

$$\mathbf{I}_{\mathrm{L}}(s) = \mathbf{I}_{\mathrm{iL}}(s) - \mathbf{Y}_{\mathrm{iL}}(s)\mathbf{V}_{\mathrm{L}}(s) , \tag{28}$$

where $\mathbf{I}_{\mathrm{iL}}(s)$ and $\mathbf{I}_{\mathrm{iR}}(s)$ are $n \times 1$ internal current vectors, and $\mathbf{Y}_{\mathrm{iL}}(s)$ and $\mathbf{Y}_{\mathrm{iR}}(s)$ are internal $n \times n$ admittance matrices of Norton equivalents at the left and right ends of the line, respectively. Finally the $\mathbf{\Phi}_{\mathrm{ij}}(s)$, $\mathrm{i,j} = 1,2$, are $n \times n$ square submatrices of the whole transition matrix $\mathbf{\Phi}(s) = \mathbf{\Phi}(s,l)$.

In practical computations it is more convenient to use a recurrent formula instead of eq. 24 to compute necessary transition matrices. For a uniform transmission line this can be of the form

$$\mathbf{\Phi}(s,x_k) = \mathbf{\Phi}(s,\Delta x) \cdot \mathbf{\Phi}(s,x_{k-1}) , \tag{29}$$

for $k = 1,2 \cdots, m$, with $x_0 = 0$ and $x_m = l$, where m is the number of line elements of equal lengths $\Delta x = x_k - x_{k-1}$. Then $\mathbf{\Phi}(s,x_0) = \mathbf{E}$ is the $2n \times 2n$ identity matrix and the matrix exponential function according to eq. 24 is used to compute the matrix $\mathbf{\Phi}(s,\Delta x)$ only once. Denoting $\mathbf{W}(s,x) = [\mathbf{V}(s,x) \quad \mathbf{I}(s,x)]^T$ the eq. 23 can then be rewritten into a recurrent form

$$\mathbf{W}(s,x_k) = \mathbf{\Phi}(s,\Delta x) \cdot \mathbf{W}(s,x_{k-1}) , \tag{30}$$

when $\mathbf{W}(s,x_0)=[\mathbf{V}_\mathrm{L}(s) \quad \mathbf{I}_\mathrm{L}(s)]^T$. Thus the x_k coordinates can straight be chosen as these intended for plotting.

In the example above $n=3$ and the Norton parameters have the forms as follows

$$\mathbf{I}_\mathrm{iL}(s)=\begin{bmatrix}2\cdot10^8\cdot(1-\exp(-10^{-10}s))/s^2\\0\\0\end{bmatrix}, \quad \mathbf{I}_\mathrm{iR}(s)=\begin{bmatrix}0\\0\\0\end{bmatrix}, \quad \mathbf{Y}_\mathrm{iL}(s)=\begin{bmatrix}0.02&0&0\\0&0&0\\0&0&0\end{bmatrix}$$

$$\mathbf{Y}_\mathrm{iR}(s)=\begin{bmatrix}1/10800+1/1400+1/450 & -1/10800 & -1/1400\\ -1/10800 & 2/10800+1/450 & -1/10800\\ -1/1400 & -1/10800 & 1/10800+1/1400+1/450\end{bmatrix} \tag{30}$$

A subfunction that will be called by the main matrix version of the *NILT* function can be of the form as follows

```
%**************************** Mult3–SUBFUNCTION DEFINITION ****************************%
function f=Mult3(s)
   global x;                                    %     x must also be global in pl4
   l=0.70;
   Lo=[2.4,0.69,0.64;0.69,2.36,0.69;0.64,0.69,2.4]*1e-6;
   Ro=[41.67,0,0;0,41.67,0;0,0,41.67];
   Co=[21,-12,-4;-12,26,-12;-4,-12,21]*1e-12;
   Go=[0.6,0,0;0,0.6,0;0,0,0.6]*1e-3;
   YiL=[0.02,0,0;0,0,0;0,0,0];
   YiR=[1/10800+1/1400+1/450,-1/10800,-1/1400;-1/10800,2/10800+1/450,-1/10800; . . .
         -1/1400,-1/10800,1/10800+1/1400+1/450];
   n=3; n1=1:n; n2=n+1:2*n;
   x=linspace(0,l,65); dx=x(2);                 %     65 coordinate points is used for plotting
   f=zeros(65,length(s),2*n);                   %     predefining 3D array to speed up calculation
   for j=1:length(s)
      Z=Ro+s(j)*Lo; Y=Go+s(j)*Co;
      IiL=[2e8*(1-exp(-1e-10*s(j)))/s(j)^2;0;0]; IiR=[0;0;0];
      MZY=[zeros(n),-Z;-Y,zeros(n)];
      Fi=expm(MZY*l);
      K=Fi(n2,n2)-YiR*Fi(n1,n2);
      VL=(K*YiL+YiR*Fi(n1,n1)-Fi(n2,n1))\(K*IiL+IiR); IL=IiL-YiL*VL;
      f(1,j,:)=[VL;IL];
      Fidx=expm(MZY*dx);
      for k=2:length(x)
         f(k,j,:)=Fidx*squeeze(f(k-1,j,:));
      end
   end
%**************************************************************************************%
```

In this case the transform evaluation is performed in a loop structure where respective complex frequencies are cycled and the resultant three-dimensional array is assembled from vectors computed using above presented matrix equations. The main *NILT* function is then called like this: niltm('Mult3',2e-8,'pl4'), and the results are in Fig.7. The CPU time was about 30 seconds.

It should be noticed a function like above was integrated into programs for time-domain simulations of multiconductor transmission lines enabling to solve not only such a simple case but also e.g. nonuniform *MTL* under nonzero initial conditions, see e.g. [5].

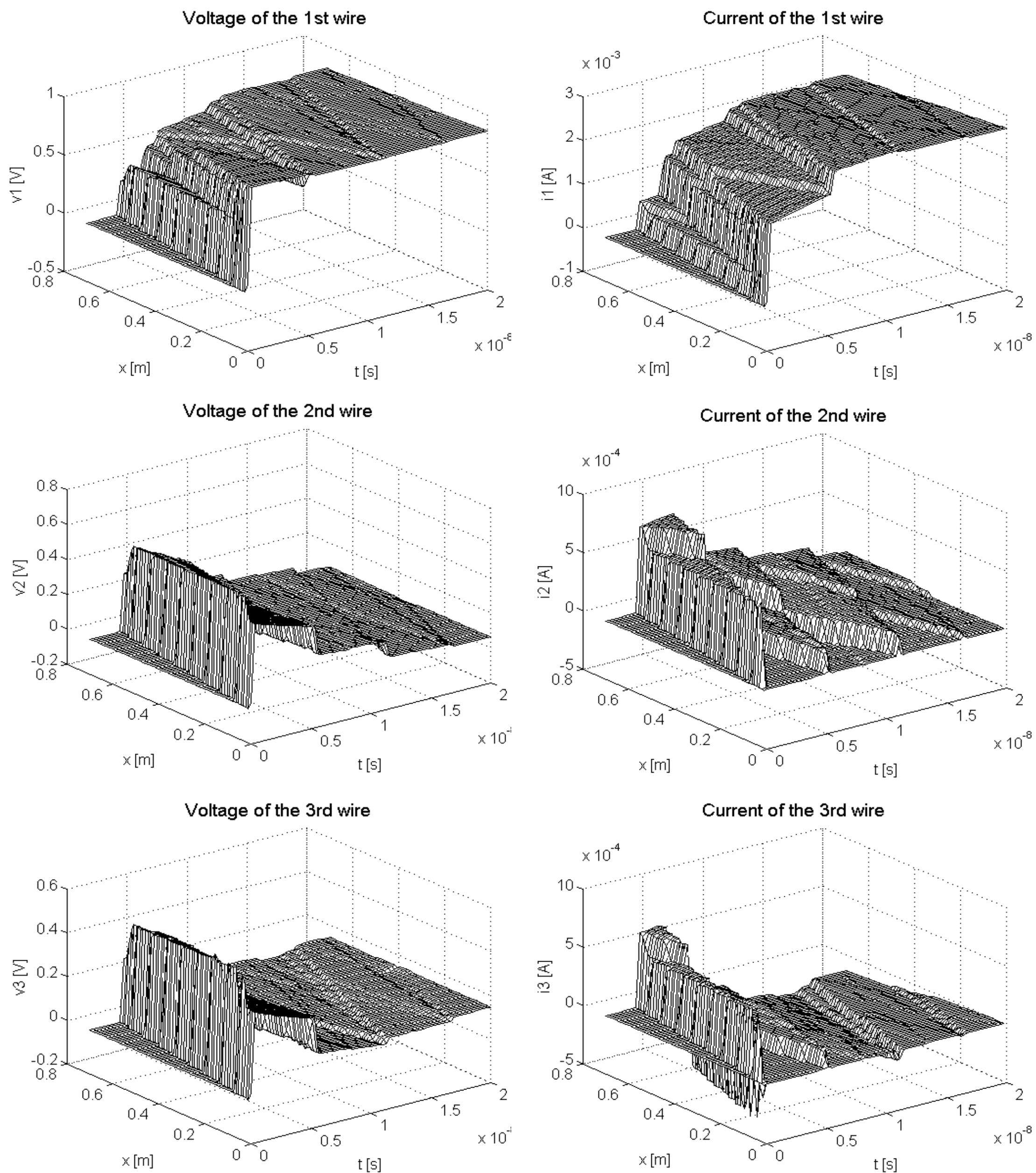

Fig.7 Voltage and current wave propagations along the *MTL*

References

[1] Brančík, L.: The Fast Computing Method of Numerical Inversion of Laplace Transforms Using FFT Algorithm. In: Proc. of 5th EDS ´98 Int. Conf., Brno, Czech Republic, June 1998, pp. 97-100.

[2] Brančík, L.: An Improvement of *FFT*-based Numerical *ILT* Procedure by Application of ε-algorithm. In: Sborník přednášek Moderní směry výuky elektrotechniky a elektroniky STO-7, Brno, září 1999, str. 196-199.

[3] Macdonald, J. R.: Accelerated convergence, divergence, iteration, extrapolation, and curve fitting. J. Appl. Phys., 10, 1964, pp. 3034-3041.

[4] Paul, C. R.: Analysis of Multiconductor Transmission Lines. John Wiley & Sons, New York, 1994.

[5] Brančík, L.: Time-Domain Simulation of Nonuniform Multiconductor Transmission Lines under Nonzero Initial Conditions Using Matlab Language. Proc. of European Conference on Circuit Theory and Design ECCTD’99, Vol. 2, Stresa, Italy, August-September 1999, pp.1135-1138.

[6] Chang, E.C., Kang, S.M: Transient Simulation of Lossy Coupled Transmission Lines Using Iterative Linear Least Square Fitting and Piecewise Recursive Convolution. IEEE Transactions on Circuits and Systems – I: Fundamental Theory and Applications, Vol. 43, No. 11, November 1996, pp. 923-932.

This work was supported by grants GAČR No. 102/98/0130 and No. 102/98/0782.

* Ing. Lubomír Brančík, CSc., Purkyňova 118, 612 00, Brno, Czech Republic