The design of time-domain simulation tools: the computational engine approach

Jean Mahseredjian
Institut de Recherche d'Hydro-Québec (IREQ)
1800 Montée Ste-Julie
Varennes, Québec, Canada J3X 1S1

Fernando Alvarado
University of Wisconsin-Madison
Electrical & Computer Engineering
1415 Johnson Drive, Madison, WI 53706, USA

Summary: The design of large computer programs for time-domain circuit simulation, such as the EMTP (Electromagnetic Transients Program), is traditionally based on coding using a conventional computer language: Fortran, C or Pascal. The programming language of the currently available EMTP version is FORTRAN-77. This paper presents new design ideas suitable for the re-development of such programs using high level tools. A transient analysis numerical *simulator* is created connecting an *input processor* to a separate set of solution blocks programmed in the *computational engine* frame of MATLAB.

Keywords: EMTP, MATLAB, time-domain network solution, high level programming

1. INTRODUCTION

The conventional concept of a time-domain circuit (power networks and electrical/electronic circuits) *simulator* is simple: a large set of algebraic-differential equations is first transformed into a discrete algebraic equivalent and then solved over the requested interval $[0, t_{max}]$. The actual solution is available only at discrete time-points $(0, t_1, t_2, t_3, \ldots, t_{max})$ where a fixed (as is the case in the EMTP[1]) or variable time-step Δt_i is used. Time-domain simulation usually requires proper initialization. Steady-state solution for a linear single frequency system status is normally a part of a conventional time-domain circuit *simulator*. Further sophistication is estimation of steady-state conditions from a harmonic steady-state module[2]. The calculation of some other initial conditions, specifically in control circuits, requires user intervention or implementation of specialized algorithms.

In a conventional *simulator* design, everything is based on line-by-line coding. Every component is implemented this way, as is the solution algorithm and any details of the overall solution process. Moreover, old-fashioned programming techniques inhibit modularity and are geared towards memory conservation. This is the case of the large EMTP code. The low level design methodology explains the low renewal and enhancement rate of large *simulator* codes. The closed architecture of these codes makes them difficult to maintain and modify, it is also prohibitive to experiment with modern algorithmic ideas.

Many network solution and modelling methods are simple to visualize and support mathematically, but their translation into an actual large scale working code is complex. Commonly used programming languages are ill-suited to human abilities for dealing with complexity. Software built using such languages is often inadequate. Some other languages such as ADA and C++, provide several powerful features for the formulation of appropriate abstractions [3] for the desired application. But programming is always easier if a specialized language is already available for the creation of similar applications. Large and specialized programs such as EMTP, should in fact possess their own dedicated *computational engine* where the programmer can build and compose with high level constructs. Programming a new *computational engine* from scratch is a major effort. There is also the risk that the specialization of such an engine may become less powerful for interconnecting applications.

This paper proposes to use an already available and widely used general purpose program as a *computational engine*: MATLAB [4]. This paper presents the creation of MATEMTP: a transient analysis program prototype in MATLAB M-files. It is based on a new formulation of the main system of network equations, shown to eliminate several topological data restrictions and capable of handling completely arbitrary switch interconnections, provided an otherwise consistent problem.

2. SOLUTION METHOD

2.a Basic principles

The problem formulation and solution must benefit from the high level functionalities of MATLAB. The recent implementation of sparse matrix manipulation capabilities into MATLAB and the reliability of the built-in solution methods for linear equations, stimulate algorithmic ideas based on matrix computations.

MATEMTP uses matrices and vectors, as much as possible, for coding and solving network equations, closely following the underlying mathematics of network theory. The core code operates by defining a larger and more general matrix to represent network equations than is customary. This matrix is described next.

2.b Network equations and component models

The main system (core code) of network equations must be defined before programming the individual component models. The following sparse formulation is used in MATEMTP:

$$\begin{bmatrix} Y_n & V_a^t & S_a^t \\ V_a & 0_{V_s} & 0_{V_sS} \\ S_a & 0_{V_sS}^t & S_0 \end{bmatrix} \begin{bmatrix} V_n \\ I_{V_s} \\ I_S \end{bmatrix} = \begin{bmatrix} I_n \\ V_s \\ 0 \end{bmatrix} \qquad (1)$$

Y_n is the standard $n \times n$ nodal admittance matrix, V_a is the $nV_s \times n$ node incidence matrix of voltage sources, S_a is the $n_S \times n$ node incidence matrix of closed switches, 0_{V_s} is an $nV_s \times nV_s$ null matrix, 0_{V_sS} is an $nV_s \times n_S$ null matrix, S_0 is an $n_S \times n_S$ sparse binary matrix used to nullify open switch currents, V_n is the vector of unknown node voltages, I_{V_s} holds the unknown voltage source currents, I_S is for unknown switch currents, I_n holds the nodal current injections and V_s stands for voltage sources. This system is used in both steady-state and time-domain solutions. The node incidence switch matrix S_a is modified to avoid the reformulation of Y_n in a varying topology condition and remains fixed in steady-state.

Equation (1) is less restrictive than the standard EMTP nodal analysis and expands on modified nodal analysis by including explicitly the switch equations. Contrary to EMTP, MATEMTP can model voltage sources not connected to ground, floating switch nodes and branch to branch relations. All switch currents are automatically calculated and the explicit switch matrix S_a usage simplifies the detection of illegal switch loops.

There are 4 basic model types: current source, voltage source, ideal switch and branch. The goal is to insert model equations into the main system (1).

The steady-state solution is a frequency domain solution. Its objective is to initialize the time-domain solution when steady-state conditions exist before transient analysis. MATEMTP can handle a simple fundamental frequency initialization or the prediction of the actual harmonic steady-state as proposed in [2]. Component models must provide their frequency domain equations to be inserted during the steady-state solution.

The time-domain solution is based on the discretization of the component models. Although Euler Backward and trapezoidal integration are programmed for the default set of models, other integration methods are freely used in individual model files, as long as compliance with core code requests exists. In addition to handling discontinuities [5][6], Euler Backward integration is useful for startup from manual initial conditions where only capacitor voltages and inductor currents can be defined.

3. THE MATEMTP CODE

The objective is to program MATEMTP using only MATLAB M-files[4]. These files include several standard MATLAB statements and may also refer to other M-files. An M-file is an ASCII script or function file. Since these files are run directly in the MATLAB environment and there is no requested compilation stage, MATEMTP inherits an open source code.

In inexperienced hands, the large number of available MATLAB building functions and constructs, cannot inhibit the creation of an inefficient and cryptic program. Some experience is needed for programming solutions with a minimal number of code lines and for minimal CPU time. The key to minimal CPU time is the vectorization of the solution algorithms.

The design of MATEMTP avoids loops and if-then-else statements and uses vectorized algorithms as much as possible. The main program structure shown in Fig. 1 relies on the *input processor* to interconnect the MATEMTP M-files. The *input processor* is a C program that can actually decode and validate standard EMTP data files [1] and create the MATEMTP *case.m* file. This file is a processed file of network data created from the external *case data* format.

The *model selector* selects the models needed for *case.m* and connects them to the main program. All models are programmed in separate M-files that simply obey to a set of predefined core code requests. A typical request for a branch model is "*provide admittance matrix*" or "*update history*". The creation of any new model is as simple as programming a new M-file which is automatically recognized and inserted into the appropriate code location by the *model selector.*

The *organizer* gathers the solution M-files according to selected options and overall solution needs. It appears that MATEMTP applies a data dependent interconnection of individual code modules and avoids testing for selected models and options and minimizes the number of code initialization procedures.

Programming within a *computational engine* frame such as MATLAB has several advantages. A major difference from conventional coding is the small number of code lines needed to express an elaborate algorithm and sparse matrix manipulations. The complete MATEMTP (all M-files without component models) requires only 500 lines of code. It must be recalled that a large number of simulation data manipulation and presentation functions is also provided in MATLAB, in addition to graphical user interface building blocks.

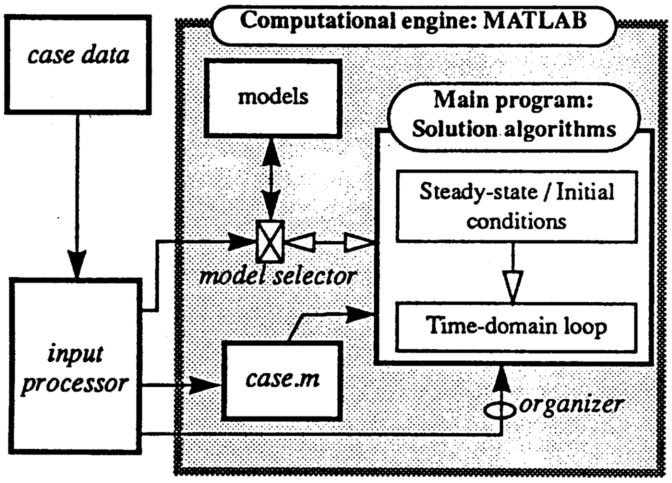
case data
Computational engine: MATLAB
models
Main program: Solution algorithms
Steady-state / Initial conditions
Time-domain loop
model selector
input processor
case.m
organizer

Figure 1: MATEMTP main structure

Here is a valid sequence of files called in by the *organizer* (see Fig. 1) for solving a typical case *case.m*:

1❐ *matemtp.m*: program startup and request for data case

2❐ *case.m*: the actual case file, any name can be used

3❐ *start.m*: initial setups for models, initial conditions, initialization of the time-domain solution

4❐ *timeloop.m*: the time-domain loop for the simulation interval

The *start.m* script file simply initializes all variables to zero (or manually supplied initial conditions) or provides automatic frequency domain initialization for any subnetwork where active sources exist at $t < 0$.

Appendix A shows a section of code called in from *start.m* for linear harmonic initialization. In a similar fashion, the script file *timeloop.m* calls *step1.m* at each time-point. Differences with *steady1.m* are: the need to account for switch position changes and the MATLAB LU decomposition function is now used for solving (1). There is no need to reapply LU when no switch position change is detected.

Since all component models appear hidden to MATEMTP, the *model selector* can only communicate through built-in generic function files (communication files) for the 4 basic component model types: *mcursou.m*, *mvolsou.m*, *mbranch.m* and *mswitch.m*. A single file *mglobal.m* is used to transfer data from workspace to model functions. Calling a MATLAB function is faster and more modular than a script M-file.

The simple test circuit of Fig. 2 is used to demonstrate some of the above outlined functionalities. The contents of *test1iwh.m* (this is now *case.m*) are listed in Appendix B.

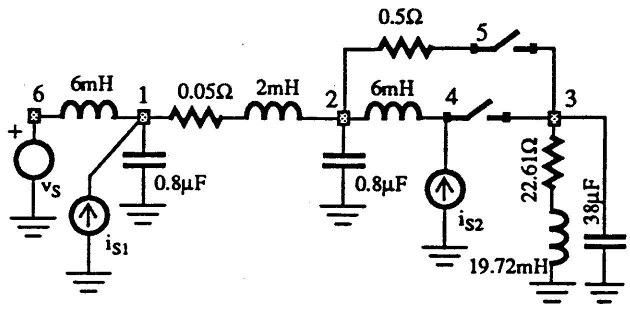

Figure 2: Test case *test1iwh.m*

It appears that this test case is using RLC branch models, two ordinary switch models, a sinusoidal voltage source model and two sinusoidal current source models. The names of those model M-files are available in an *input processor* library and that is why the *model selector* sends the following communication files:

mglobal.m: (called in from *matemtp.m*)

```
    gvsine;%sinusoidal voltage source data
    gisine;%sinusoidal current source data
    rlcglob;%RLC model data
    sw0glob; %ordinary switch model data
```

mvolsou.m:

```
    function mvolsou(ido)
    vsine(ido); %sinusoidal voltage source
```

mcursou.m:

```
    function mcursou(ido)
    isine(ido); %sinusoidal current source
```

mbranch.m:

```
    function mbranch(ido)
    rlcmod(ido); %RLC model
```

mswitch.m:

```
    function mswitch.m(ido)
    sw0(ido); %ordinary switch model
```

As an example of model data connection file, here are the contents of *gvsine.m*:

```
global Vadj Vsinein Vmag Vstart Vstop Vphi
    Vw;
```

Another possibility would have been to let each model handle its own data, by subdividing *case.m* into separate model data M-files.

Any given model can perform a standardized number of model tasks according to the *ido* parameter. Here is an extract of code from *rlcmod.m* when this model responds to admittance matrix creation in steady-state and time-domain for *ido=2* and *ido=5* respectively:

```
if ido == 2  %add into Yn for ss
Yn=Yn+RLCadj'*sparse(1:nRLC,1:nRLC,
1./(RLCR+jz*(w*RLCL - RLCC/w ) ) )*RLCadj;
elseif ido == 5   %Yn in time-step
GRLC=sparse(1:nRLC,1:nRLC,
1./(RLCR+(2/Dt).*RLCL + (Dt/2).*RLCC));
GHRLC=(2/Dt).*RLCL+RLCR-(Dt/2).*RLCC;
 Yn=Yn+RLCadj'*GRLC*RLCadj;
end
```

Here is the code for *ido=5* for a single phase transmission line model:

```
Yn=Yn+(Tadj>0)'*sparse(1:nT,1:nT,
1.0/TZc)*(Tadj>0);
Yn=Yn+(Tadj<0)'*sparse(1:nT,1:nT,
1.0/TZc)*(Tadj<0);
```

The switch status detection code (for *ido=7)* is a good example of code sophistication:

```
p=pulse(Sclose,Sopen);
inBand=(~p.*Sstatus).*(abs(IS)<Seps);
changedSigns=
(~p.*Sstatus).*((IS.*Slast)<=0);
newSstatus=
unitstep(p+(~(inBand+changedSigns)));
if max(abs(newSstatus-Sstatus))
   reBuild=1;
   Sactive=bkrtreel(newSstatus, Sadj,n,nS);
end;
Sstatus=newSstatus;
Slast=IS;
```

4. TEST CASES

4.a Case 1

The circuit diagram of this test case is shown in Fig. 2. This test case is unfair for the EMTP, since the standard EMTP cannot handle steady-state initialization with different source frequencies in the same subnetwork and according to *testliwh.m* (see Appendix B) the harmonic current sources i_{S1} and i_{S2} are connected for $t < 0$.

The EMTP found voltage waveform, shown in Fig. 3, starts with wrong initial conditions (both current sources set to start at $t = 0$) and will enter almost perfect steady-state only after 8 seconds of simulation time. Since the study is performed in steady-state conditions, the switch closing times must be readjusted. EMTP uses Euler backward [6][7] integration to eliminate the numerical oscillations caused by the startup of i_{S2}.

MATEMTP can solve this case directly through its initialization algorithm of Appendix A. Fig. 4 shows that MATEMTP is in harmonic steady-state at $t = 0$. The superposed waveform is found from shifting the EMTP solution backwards by 7.95 seconds. The differences are almost imperceptible.

The *eratio* (defined as total MATEMTP elapsed execution time over EMTP execution time) of $\cong 1/3$ is in favor of MATEMTP because EMTP needs a much longer simulation time.

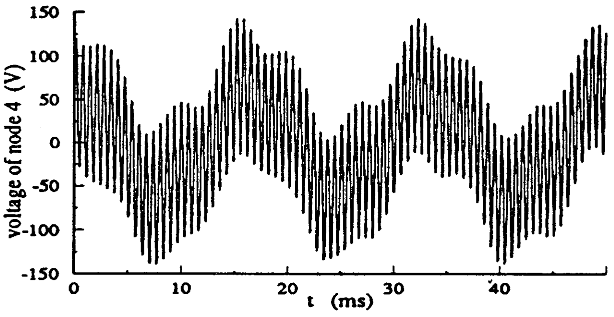

Figure 3: EMTP simulation, Case 1, wrong initial conditions

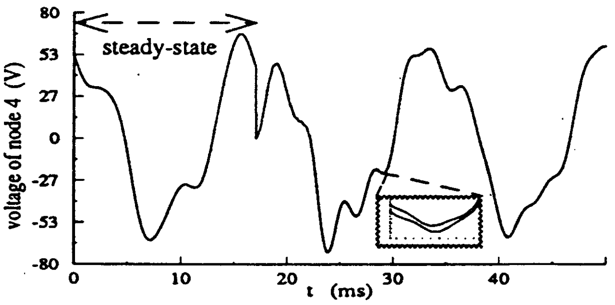

Figure 4: MATEMTP and shifted EMTP solutions, Case 1

4.b Case 2

When current sources i_{S1} and i_{S2} in *testliwh.m* are hypothetically set to operate at 60Hz, then (see Fig. 5) both MATEMTP and EMTP can start in steady-state immediately. Now the *eratio* of $\cong 2.5$ is more indicative. Considering that EMTP suffers from its large number of always present options and models and has a large overhead for its input and output operations, this number indicates that MATEMTP is slow compared to a compiler based code, but not that slow!

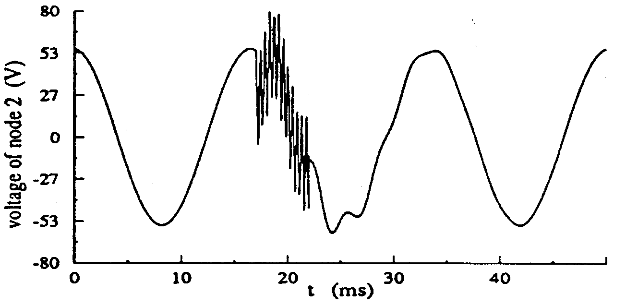

Figure 5: MATEMTP and EMTP solutions, Case 2

4.c Case 3

The circuit diagram of this case is shown in Fig. 6. This case is used to pinpoint the MATEMTP algorithmic advantages in handling switches. Typical applications are in power electronics, power system control or model logic.

All switches are initially open and according to shown closing times, the closing of switch 3 creates a switch loop. It is not easy for the EMTP code to handle such loops and EMTP will actually halt the simulation at 6μs and print an error message. The detection of a switch loop is simple in MATEMTP, since such a loop will create a linearly dependent row in the switch matrix $\mathbf{S_a}$ of equation (1). The loop is currently broken by preserving the maximum number of nodes. Other criteria based on possible usage of switch current as a control variable, can be easily implemented.

Another difficulty for EMTP algorithms are the floating subnetworks created by the resistor and the node interconnecting switches 1 and 2, before the closing of appropriate switches. EMTP identifies such subnetworks during its Gaussian elimination process and adds a large resistor to ground for the existing zero pivot nodes. MATEMTP simply relies on its *computational engine* LU factorization (for sparse matrices) used for solving equation (1). It is true that the resistor creates an infinite condition submatrix in $\mathbf{Y_n}$, but the L and U matrices of such a submatrix exist and the forward backward substitution is correctly applied for any otherwise consistent right hand side.

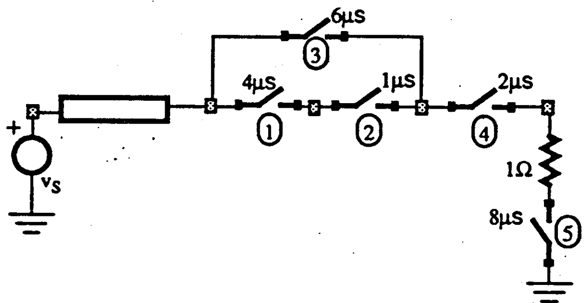

Figure 6: Test case with a possible switch loop

5. DISCUSSION

The initial programming of MATEMTP applied extreme modularity through a large number of script and function files. The *input processor* was only a simple file format translator, all models were always present in the code and a large number of logical statements was required for data initialization and selection of appropriate solution options. Such programming contributed to the creation of an extremely slow program. Drastic performance improvement came from the creation of the ready-to-run structure of Fig. 1 in addition to replacing most of the script M-files by function M-files and using a single script M-file for repetitive core code operations such as the time-step loop. Preallocating vectors in which results are stored is also an important factor in the overall MATEMTP speed. It is obvious that blind usage of dynamic memory simplifies programming but places a heavy burden on the MATLAB interpreter.

A detailed analysis of MATEMTP CPU usage in the time-step loop for the typical case of Fig. 2, shows the following disposition: less than 15% for LU factorization and triangular solution, close to 60% for updating the right hand side of equation (1) and the remaining is for individual model updates. It appears that applying vectorized algebraic functions is time consuming. A promising possibility is the replacement of such functions by compiled C language MEX-files [8], but this should be applied only at the last stage of programming and contradicts the high level programming objective, although the core code remains unchanged. And what if an automatic C-code generator was available for any MATLAB M-file? Automatic creation of ready to link object files would be sufficient.

CONCLUSIONS

This paper presented the programming of a time-domain network simulator in the *computational engine* frame of MATLAB: MATEMTP. It is based on the concept of an external *input processor* that gathers MATEMTP solution and model M-files according to the simulated network case.

Although the computational speed is found to be acceptable for the tested cases, MATEMTP being an interpreter, remains slower than a standard compiler based program. Several speed improvement methods have been proposed and the performance of such methods must be tested in a large network case.

MATEMTP has been also implicitly used for testing a new and less restrictive formulation of main network equations.

MATEMTP is a powerful prototyping tool suitable for the standard EMTP redevelopment phase.

APPENDIX A

MATEMTP linear initialization module: *steadyl.m*

The following is a listing of *steadyl.m*:

```
t0=cputime;
Wall=[];
mcursou(5);  %put all current source ws in Wall
```

```
mvolsou(5);  %put all voltage source ws in Wall
Wall=sort(Wall);
Vn_init=zeros(n,1); %initialize n node voltages
IVs=zeros(nVs,1);   %initialize IVs
IS=zeros(nS,1);     %initialize IS
nfreq=size(Wall,1); %the number of ws to do
ifreq=1;
wdone=[];
while ifreq <= nfreq
 w=Wall(ifreq);
 if w ~= wdone
   steady1; %see code below
   wdone=w;
   mbranch(3); %accumulate steady-state at t=0
 end
 ifreq=ifreq+1;
end
Vn=Vn_init;  %solution at t=0
sscputime=sscputime+cputime-t0;
```

The following lines are from *steady1.m* :

```
%This is the ss module step for the frequency w
Yn=sparse(n,n); %Build Yn
mbranch(2); %contribution to Yn by branch models
Ytmp=[Yn Vadj'; Vadj sparse(nVs,nVs)];
%
if nS ~= 0 %insert active switches
    Stmp=sparse(1:nS,1:nS,Sactive)*
         [Sadj sparse(nS,nVs)];
    Sz  =sparse(1:nS,1:nS,~Sactive);
    Yaug=[Ytmp Stmp'; Stmp Sz ];
else
    Yaug=Ytmp;
end;
% Contruct the steady state RHS
In=zeros(n,1);      %n is the number of nodes
mcursou(3); %current sources in In for w
mvolsou(3); %voltage sources in Vs for w
Itmp=[In; Vs];
Iaug=[Itmp; zeros(nS,1)]; %account for switches
%
Vaug=Yaug\Iaug; %compute unknown phasors
Vn=Vaug(1:n);     % Nodal phasor voltages
Vn_init=real(Vn)+Vn_init;  %at t=0 accumulate
if nVs ~= 0
 IVs=real(Vaug(n+1:n+nVs))+IVs;
end
if nS ~= 0
 IS=real(Vaug(n+nVs+1:n+nVs+nS))+IS;
 Slast=IS;
end
```

APPENDIX B

MATEMTP data file for the test case of Fig. 2

The following is a listing of *test1iwh.m* :

```
Dt=50e-06;
tmax=1000;        %this is 50ms
n=6; %number of nodes
BUS=['BUS1  ';'BUS12 ';'BUS13L';'BUS13S';
'BUS1S ';'SRC   ';]; %bus names
%
%RLC model
RLCadj=sparse(8,6); %node incidence
RLCadj(1,6)=1; RLCadj(1,1)=-1;
RLCadj(2,1)=1; RLCadj(2,2)=-1;
RLCadj(3,1)=1; RLCadj(4,2)=1;
RLCadj(5,2)=1; RLCadj(5,4)=-1;
RLCadj(6,3)=1; RLCadj(7,2)=1; RLCadj(7,5)=-1;
RLCadj(8,3)=1;
RLCout=sparse(8,1);
RLCout(5)=1; %current output request
RLCR=[0;0.05;0;0;0;22.61;0.5;0;];
RLCL=[0.006;0.002;0;0;0.006;0.01972;0;0;];
RLCC=[0;0;8e-07;8e-07; 0;0;0; 38e-06];
%
%Sine current source model
Isineadj=sparse(2,6);
Isineadj(1,4)=1; Isineadj(2,1)=1;
Imagn=[1.001000; 0.10000; ];
Iphi=[10.000000; 5.000000; ];
Istart=[-1.0; -1.0; ]; Istop=[Inf; Inf; ];
Iw=2*pi*[180.000000; 360.000000; ];
%
%Ordinary switch model, allow open at 0 crossing
Sadj=sparse(2,6);
Sadj(1,3)=-1; Sadj(1,4)=1; Sadj(2,3)=-1;
Sadj(2,5)=1;
Sclose=[17.E-3; 22.E-3; ];
Seps=[0; 0; ]; Sopen=[Inf; Inf; ]; %never open
%
%Sine voltage source model
Vadj=sparse(1,6);
Vadj(1,6)=1; Vsinein=[1;]; %row number in Vs
Vmag=[56.340000; ];
Vphi=[0.000000; ]; Vstart=[-1.000000; ];
Vstop=[Inf; ]; Vw=2*pi*[60.000000; ];
```

REFERENCES

[1] Electric Power Research Institute, EMTP Development Coordination Group, EPRI EL-6412-L: Electromagnetic Transients Program Rule Book, Version 2

[2] X. Lombard, J. Mahseredjian, S. Lefebvre and C. Kieny: Implementation of a new harmonic initialization method in the EMTP. IEEE Trans. on Power Systems, Summer Meeting 94, paper 94 SM 438-2 PWRD

[3] G. Bray and D. Pokrass: Understanding Ada, A Software Engineering Approach. John Wiley & Sons, 1985

[4] MATLAB, High-Performance Numeric Computation and Visualization Software. The MathWorks, Inc. MATLAB User's guide, August 1992

[5] B. Kullicke: Simulation program Netomac, Difference conductance method for continuous and discontinuous systems. Siemens Research and Development Reports, Vol. 10, pp. 299-302, 1981, no. 5

[6] J. R. Marti and J. Lin: Suppression of numerical oscillations in the EMTP. IEEE Trans. on Power Systems, Vol. 4, No. 2, 1989, pp. 739-747

[7] J. Mahseredjian: The EMTP SUN and CRAY UNIX versions. Rapport IREQ-93-065, March 1993, Hydro-Québec

[8] MATLAB, High-Performance Numeric Computation and Visualization Software. The MathWorks, Inc. External Interface guide, January 1993