

TLINE

Lossless Transmission Line Simulation

via

the Finite Difference–Time Domain Technique

Electromagnetic Simulations Laboratory
Technical Report # 31

February 16, 2005
J. Richie

Updated: January 2007

Summary

This report details the program `tline` that simulates waves on a transmission line using a one-dimensional finite-difference time-domain (FDTD) algorithm on the voltage and current. This code can be used to prepare video clips of the wave motion along the transmission line, and to provide the AC steady state information regarding up to three sections of line/load. Parallel connections such as stub tuners are not available.

Future Considerations

- Direct implementation of resistances
- The “Big” Problem
- Lossy Lines
- Reactive Loads
- Stub Matching Networks
- Multiple ($\lambda/4$) matching sections

1. Theory.

There are a number of options when considering simulation of the voltage and current on a uniform transmission line. SPICE is capable of transmission line simulation, either as an ideal (lossless) delay line [1], or using an RLC model [2]. Computer modeling and analysis of digital transmission lines is discussed in great detail in [3].

In addition, a set of two programs specifically for educational purposes have been reported. The first is capable of animating transients on transmission lines for educational purposes [4]. The second program is an interactive solver for the engineering parameters of a transmission line with stubs and loads using sinusoidal excitations [5].

In this work, the use of the finite difference method is described using the simulation of transmission lines as an example application. This application is also discussed in [6].

This section begins with the development of the finite difference equations used for the first order coupled transmission line equations. The stability and implementation of the difference equations is presented, followed by details regarding the source and boundary conditions. The post processing performed at this time is limited; however, it is described at the conclusion of this section.

2. Finite Difference Equations.

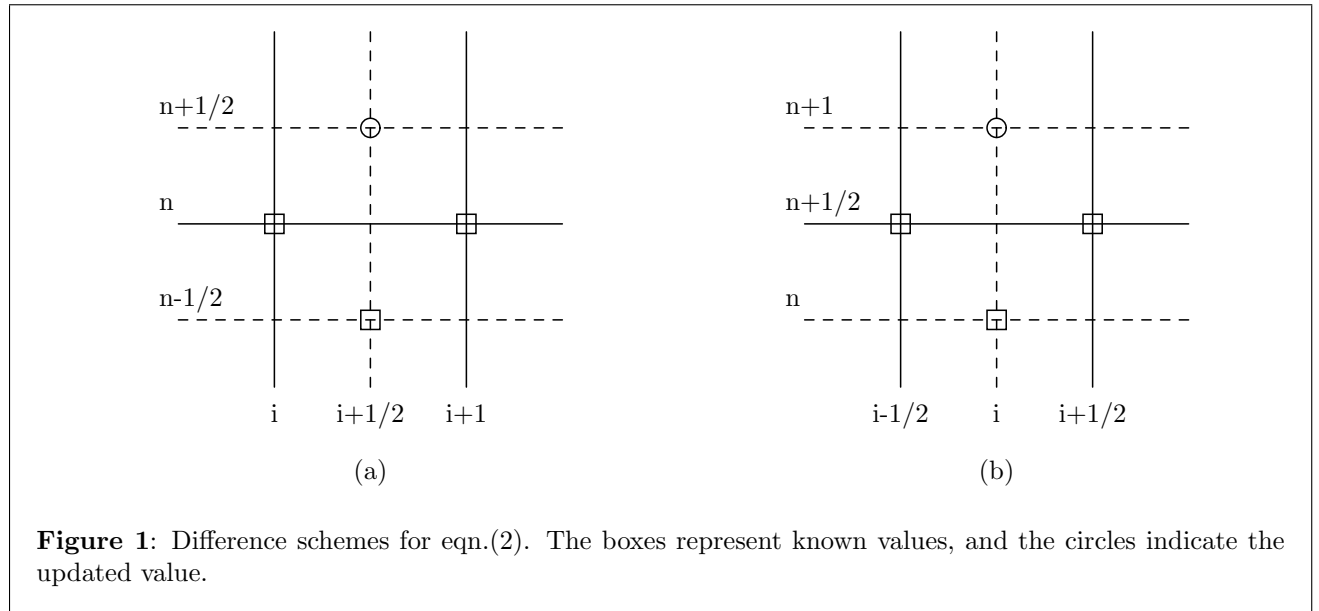
The transmission line equations can be derived from the distributed parameter model that uses R , L , G , and C :

$$-\frac{dv}{dx} = Ri + L\frac{di}{dt} \quad (1.a)$$

and

$$-\frac{di}{dx} = Gv + C\frac{dv}{dt} \quad (1.b)$$

where $v = v(x, t)$, $i = i(x, t)$ are the voltage and current, respectively, along the transmission line. We shall assume a lossless line, i.e., $R = G = 0$.



Let $x = i\Delta x = i$ and $t = n\Delta t = n$, then, the marching-on-in-time method can be used to compute the next time step for I , and a half time step later, V is computed, then another half time step later I , and so on. Using central finite differences for the derivatives in (1.a,b), we obtain [7]:

$$-\frac{V^n(i+1) - V^n(i)}{\Delta x} = -L \frac{I^{n+1/2}(i+1/2) - I^{n-1/2}(i+1/2)}{\Delta t} \quad (2.a)$$

where both sides are evaluated at $(n, i+1/2)$, and

$$-\frac{I^{n+1/2}(i+1/2) - I^{n+1/2}(i-1/2)}{\Delta x} = -C \frac{V^{n+1}(i) - V^n(i)}{\Delta t} \quad (2.b)$$

where both sides are evaluated at $(n+1/2, i)$.

In (2), eqn. (2a) is used to update $I^{n+1/2}(i+1/2)$ using data from time n and $n-1/2$, and eqn. (2b) follows (2a) to update $V^{n+1}(i)$. See Figures 1.a and 1.b to see the finite difference scheme corresponding to (2.a,b).

Eqns. (2) can be rearranged to indicate the calculation necessary to compute the next time step, given previous time steps:

$$I^{n+1/2}(i+1/2) = I^{n-1/2}(i+1/2) - \frac{\Delta t}{L\Delta x} [V^n(i+1) - V^n(i)] \quad (3.a)$$

and

$$V^{n+1}(i) = V^n(i) - \frac{\Delta t}{C\Delta x} [I^{n+1/2}(i+1/2) - I^{n+1/2}(i-1/2)] \quad (3.b)$$

In general, a “marching-on-in-time” technique is used to simulate the voltage and current. First, the voltages are updated at time $n+1$, and the currents are updated a half time-step later, at $n+1/2$. In the next section, the relationships required to perform suitable time-stepping are discussed.

3. Stability.

The stability of the calculation is important because with inappropriate time step and/or inappropriate Δx the iterations can quickly diverge to inaccurate results.

Generally, the condition to insure stability of FD-TD calculations is:

$$\Delta t \leq \frac{\Delta x}{v_{ph}} \quad (4)$$

where v_{ph} is the phase velocity of the wave along the x direction. Note that the condition when (4) is an equality is denoted as the magic time step. In general, the FD-TD technique introduces a (usually) small numerical dispersion, related to the stability condition (4).

In the code, the parameters `tfact` and `lfact` are used. `lfact` is the number of cells per wavelength ($\lambda/\Delta x$). Since the wavelength is not easily defined for waveforms in the time domain, a maximum valid frequency of 100 MHz has been chosen. Generally, 20 cells per wavelength are used.

For the lossless transmission line, and for the non-dispersive one-dimensional case in general, use of the magic time step results in no numerical dispersion. Δt is chosen initially to satisfy the magic time step. The parameter `tfact` is used to quantify how much Δt and the magic time step differ:

$$\Delta t = \frac{tfact}{lfact} \frac{\lambda}{v_{ph}} = \frac{tfact}{lfact} \frac{1}{f} \quad (5)$$

It has been found that the stability of the computations is very dependent on the relative values of dielectric constant. For a wave traveling from left to right, when the wave reaches a boundary and enters a medium with a higher dielectric constant, the results at the magic time step diverge.

The code has been written to check for divergent results, and reduce Δt by the parameter `tfact` if the fields diverge. This makes the code adaptive, in a sense. What is done in the code, is to check for excessively large values of voltage or current. If excessively large values are identified, the simulation is aborted, the time step is decreased, and the simulation begins. This process continues until the simulation is completed with no excessive values identified.

4. Implementation.

Figure 2 shows the one dimensional geometry. Values of x and the corresponding physical parameters are labelled above the horizontal axis and i values ($x = i\Delta x$) are below the axis. The problem space consists of three regions, from $i = 0$ to i_1 , from i_1 to i_2 , and from i_2 to i_{max} .

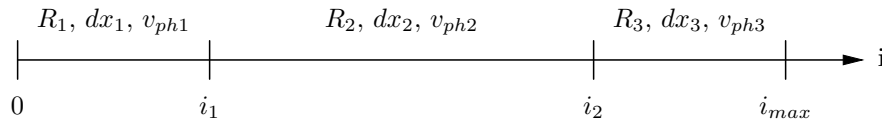


Figure 2: Geometry for three-region problems. At $i = i_{max}$, an open circuit or a short circuit can also be implemented.

Three separate regions have been chosen for all simulations. Two regions can easily be incorporated by using the same parameters for two of the three regions. Three regions were chosen to allow a wide variety of simulations, such as:

- transmission line and a load resistance;
- source with an impedance R_g , a transmission line, and a load;
- transmission line, quarter-wave transformer, and a load;
- transmission line, half-wave window, and a load;

For each region, the user enters values (for each section of line) for the impedance, phase velocity, and length (in meters).

Generally, the values of i_1 , i_2 , and i_{max} are determined from Δx , the spatial increment. Then, to insure stability, Δt is chosen using (4), where Δt is the same in all regions. The spatial increment must be chosen to provide sufficient accuracy, usually $\Delta x \leq \lambda/20$, where λ is the smallest wavelength. The smallest wavelength is related to the maximum frequency, assumed to be 100 MHz in the simulations. Then, the minimum wavelength is determined using:

$$\lambda_{min} = \frac{v_{ph}}{f_{max}} = 20\Delta x_{max} \quad (6)$$

where f_{max} is 100 MHz.

The value of Δt is computed using (5), where the largest phase velocity is used. Δt is constant for all i . The value of Δt is decreased if divergent results are obtained (see section on stability). Once Δt is found, Δx in each region can be computed. The length (in meters) for each region is then used to determine i_1 , i_2 , and i_{max} .

This implementation of FD-TD does not enable a full-wave analysis; results for multiple frequencies can not be gleaned from a single simulation. Therefore, only the present time step and one previous time step must be stored in memory. This results in less possible information, while allowing the software to run using less computational resources.

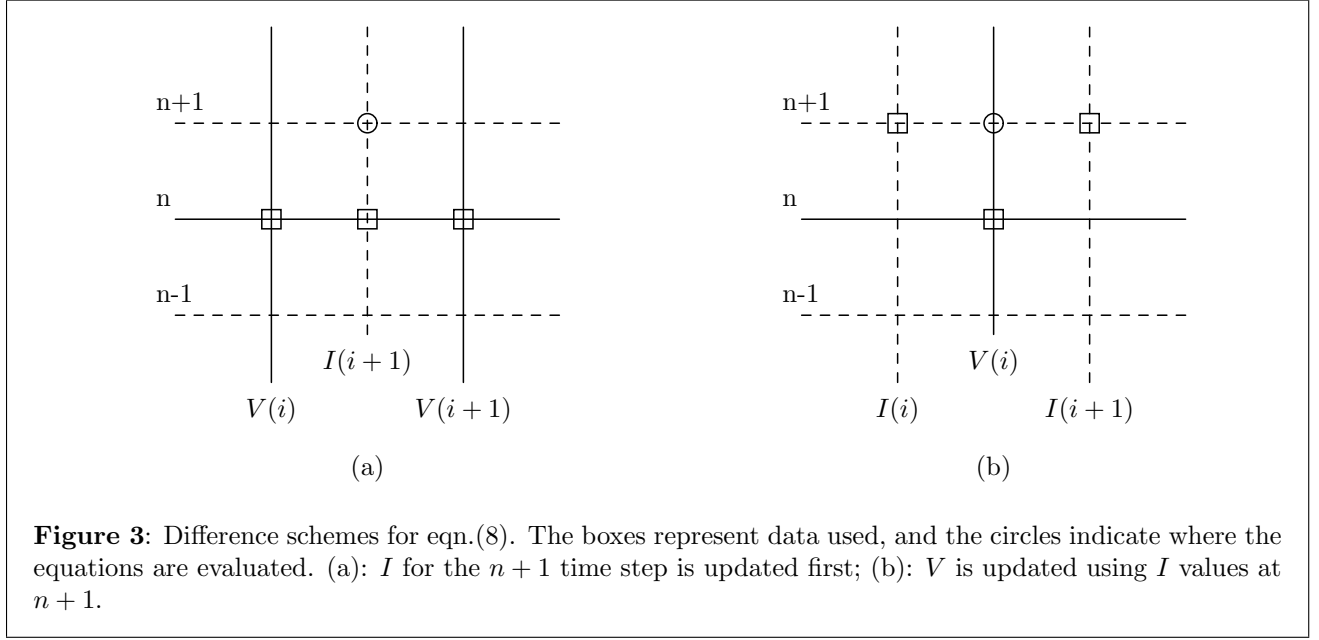


Figure 3: Difference schemes for eqn.(8). The boxes represent data used, and the circles indicate where the equations are evaluated. (a): I for the $n+1$ time step is updated first; (b): V is updated using I values at $n+1$.

5. Iteration Process.

The iteration process is based on (3), repeated here:

$$I^{n+1/2}(i+1/2) = I^{n-1/2}(i+1/2) - r_e [V^n(i+1) - V^n(i)] \quad (7.a)$$

$$V^{n+1}(i) = V^n(i) - r_h [I^{n+1/2}(i+1/2) - I^{n+1/2}(i-1/2)] \quad (7.b)$$

where

$$r_e = \frac{\Delta t}{L\Delta x} \quad r_h = \frac{\Delta t}{C\Delta x} \quad (8)$$

Since r_e and r_h depend on the region, arrays are used to store r_e and r_h for every value of i .

The half-steps in the iteration process are inconvenient. It is necessary to adjust the half-steps in the current. For I , $I^{n+1/2}(i+1/2)$ becomes $I^{n+1}(i+1)$. Then, (7) becomes:

$$I^{n+1}(i+1) = I^n(i+1) - r_e [V^n(i+1) - V^n(i)] \quad (8.a)$$

$$V^{n+1}(i) = V^n(i) - r_h [I^{n+1}(i+1) - I^{n+1}(i)] \quad (8.b)$$

The iteration molecule for I is shown in Figure 3a. Once all I values are updated at time $n+1$, the iteration molecule for V shown in Figure 3b is used to update all V values.

6. Source.

The source of the wave on the transmission line has been implemented using two possibilities: a sinusoid for all time, and a Gaussian pulse. The sinusoid begins at $t = 0$, and is a sine curve with user-input frequency.

The Gaussian pulse shape, S_g , is given by:

$$S_g(n) = e^{-(n-N_o)^2/N_d^2} \quad (13)$$

where n is the present time step, N_o is the time where the pulse is at its peak, and N_d is the variance (describing the width) of the pulse.

The gaussian pulse must begin sufficiently small at $t = 0$, since FD-TD methods do not accurately model abrupt changes in the source. The Gaussian pulse is implemented with a value for N_o that is at least three times N_d [8]. This insures that no numerical ringing is introduced in the results.

In the finite difference-time domain method, the source can be considered “hard” or “soft”. A hard source implies that the value of the voltage or current is set to precisely the value of the source. In the soft source case, the voltage or current at the source is added to the value obtained in the time-stepping algorithm. By utilizing a soft source, the reflected wave is allowed to propagate through the source. A hard source is only recommended if no reflection back to the source can be guaranteed [9].

7. Boundary Conditions.

In all numerical techniques in electromagnetics that utilize partial differential equations (rather than integral equations), the boundary of the solution space must be prepared to avoid numerical reflection at the edges of the problem. Here, we incorporate Mur's implementation of absorbing boundary conditions (ABC's), derived from the Engquist-Majda equations [10],[11].

The Engquist-Majda equations are "one-way wave equations". One-way wave equations allow propagation in only 'some' directions and can thus be used to set up the absorption of waves at boundaries. The equations are found by factoring the partial differential equation operator and were implemented by Mur in Cartesian coordinates.

Consider the scalar wave equation in Cartesian coordinates:

$$\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} - \frac{1}{c^2} \frac{\partial^2 U}{\partial t^2} = 0 \quad (14)$$

and let us define

$$L = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} - \frac{1}{c^2} \frac{\partial^2}{\partial t^2} = D_x^2 + D_y^2 - \frac{1}{c^2} D_t^2 \quad (15)$$

thus, we have $LU = 0$.

To factor the operator, L , we state without proof that:

$$LU = L^+ L^- U = 0 \quad (16)$$

where:

$$L^- = D_x - \frac{D_t}{c} \sqrt{1 - S^2} \quad S = \frac{D_y}{(D_t/c)} \quad (17)$$

Note that L^+ is the same except a $+$ sign is used before the radical. In addition, the D_x is placed in front when factoring the operator so that a condition for an $x = c$ boundary is derived. The operator L^- is used at the smallest x (usually $x = 0$) for waves propagating in the $-x$ direction, and L^+ is used at $x = x_{max}$ for $+x$ propagating waves.

The relation $L^- U = 0$ will absorb a plane wave propagating toward the boundary $x = 0$ for any angle α , as shown in Figure 4. However, the operator L^- , with the radical, is a "pseudodifferential operator", and is not a local operator. It is difficult to incorporate L^- directly.

The operator is approximated by performing a Taylor series expansion on the radical in (17). One term and two term approximations have been reported in the literature. In the one-term approximation,

$$\sqrt{1 - S^2} \rightarrow 1 \quad (18)$$

which results in:

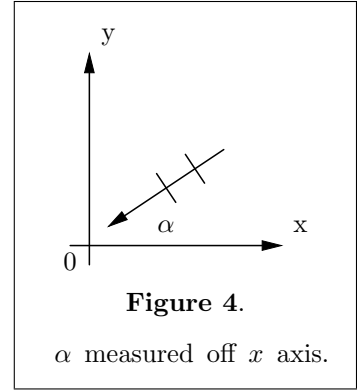
$$L^- = D_x - \frac{D_t}{c} \quad \frac{\partial U}{\partial x} - \frac{1}{c} \frac{\partial U}{\partial t} = 0 \quad (19)$$

For the two term approximation,

$$\sqrt{1 - S^2} \rightarrow 1 - \frac{1}{2} S^2 \quad (20)$$

which results in:

$$L^- = \frac{\partial^2}{\partial x \partial t} - \frac{1}{c} \frac{\partial^2}{\partial^2 t} + \frac{c}{2} \frac{\partial^2}{\partial y^2}$$



$$L^-U = \frac{\partial^2 U}{\partial x \partial t} - \frac{1}{c} \frac{\partial^2 U}{\partial^2 t} + \frac{c}{2} \frac{\partial^2 U}{\partial y^2} \quad (21)$$

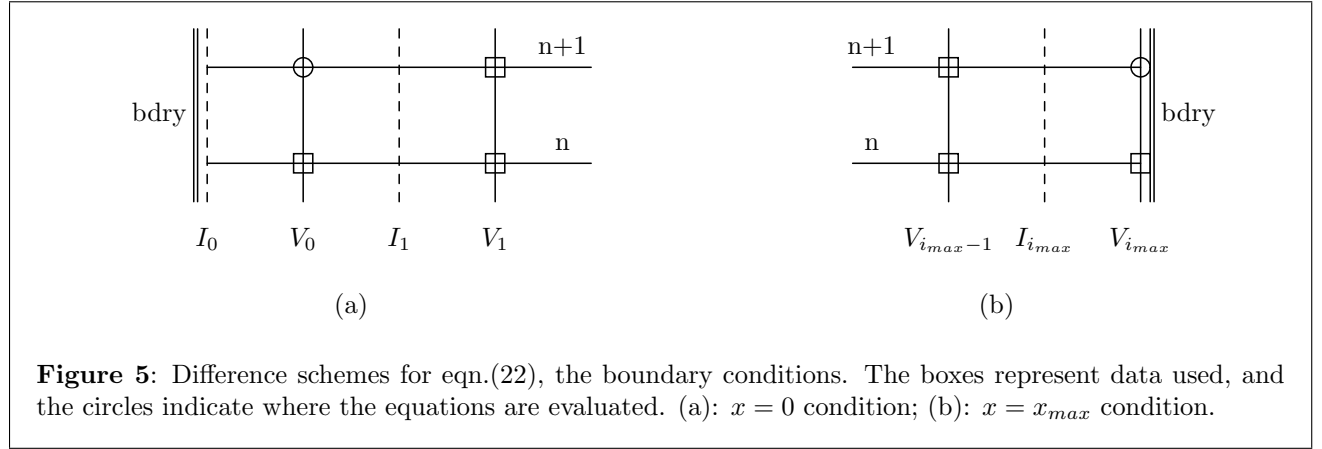
For the transmission line implementation here, these equations are applied to V and I at the grid outer boundaries.

The Mur finite difference scheme is an implementation of the ABC's derived previously. The two-term case is very useful for two-dimensional simulations, and the one-term case can be derived by removing the y -derivative terms. The one-term approximation is sufficient for one-dimensional problems.

Presently, the code incorporates the following difference scheme. At $x = 0$,

$$V_0^{n+1} = V_1^n + \frac{v_{ph}\Delta t - \Delta x}{v_{ph}\Delta t + \Delta x} [V_1^{n+1} - V_0^n] \quad (22)$$

where it should be noted that all values of V_i^{n+1} have been computed except the boundary value, V_0^{n+1} . At $x = x_{max}$, the only change is related to the subscripts for the indices, i.e., the edge value goes from 0 to i_{max} and the 1 becomes $i_{max} - 1$.



Consider the boundary at $x = 0$, shown in Figure 5a. In simulation, I_0 is zero, always. The value of I_0 is not needed, and is never updated. The boundary condition is applied to V_0^{n+1} , where most of the V and I values at $n + 1$ have been computed. The iteration molecule for (22) is shown in Figure 5a. The boundary and iteration molecule for (22) at $x = x_{max}$ is shown in Figure 5b. In iteration, $I_{i_{max}}^n$ and $V_{i_{max}-1}^n$ are the last values updated. The boundary condition is applied to $V_{i_{max}}$, using (22) as shown in Figure 5b.

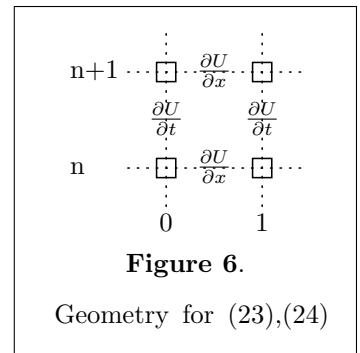
To derive (22), one must do a finite difference approximation to the differential equation in (19). This is accomplished by using

$$\frac{\partial U}{\partial t} = \frac{1}{2} \frac{\partial U}{\partial t} \Big|_{i=0} + \frac{1}{2} \frac{\partial U}{\partial t} \Big|_{i=1} \quad (23)$$

and

$$\frac{\partial U}{\partial x} = \frac{1}{2} \frac{\partial U}{\partial x} \Big|_n + \frac{1}{2} \frac{\partial U}{\partial x} \Big|_{n+1} \quad (24)$$

and applying central differences. Figure 6 illustrates the finite difference scheme for the derivatives in (23) and (24). In Figure 6, the derivatives are evaluated using data from the grid points in boxes either vertically or horizontally displaced from the label of the derivative.



8. Post Processing.

Presently, the post-processing that is done consists of writing snapshots of the V and I arrays to files along with a gnuplot script that incorporates these snapshots into a video clip. Extreme values of the V and I arrays are automatically tracked so that a common scale is used for each gnuplot graph. The gnuplot graphs can be generated as encapsulated postscript files and displayed in sequence using the animation utility of ImageMagick (image manipulation software under the FSF license).

In addition, the code computes the number of time steps it will take for the wave to travel from one end of the problem space to the other. This time is multiplied by 50 to determine the final iteration time step. The final time step is then multiplied by 47/50. When the wave has traveled the length of the problem space 47 times, the maximum absolute values at each cell are recorded. This is done by comparing the present value during each iteration with the maximum value recorded at that location.

The maximum values are then used to compute and print the voltage standing wave ratio on each section of the line. A data file is also written that records the maximum value of V and I at each location.

9. Results and Discussion.

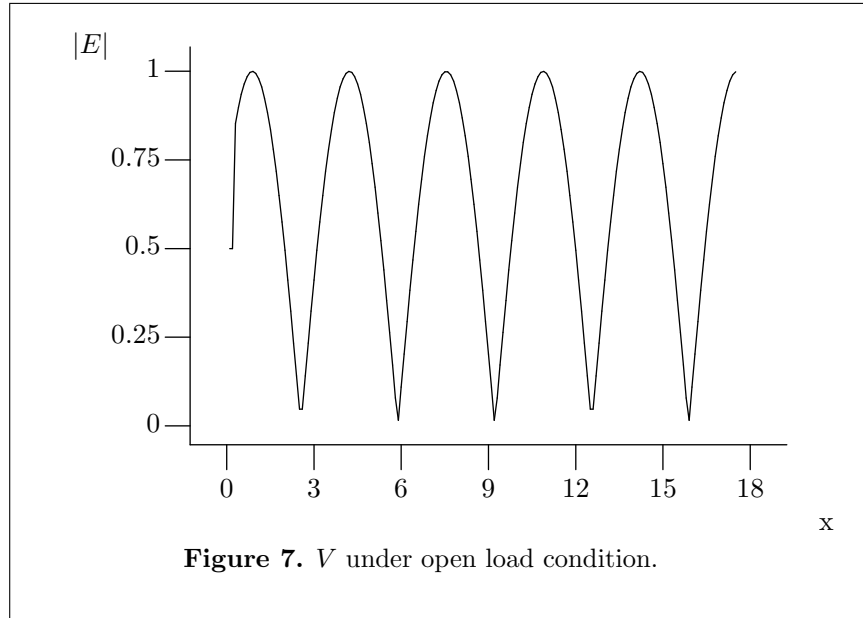
In this section, the code is evaluated. The tests described here are:

- Open circuit load on a single transmission line;
- Standing wave ratio (VSWR) on a line that is terminated in an unmatched (real) impedance;
- Time step (`tfact`) considerations and dispersion;
- Two examples of quarter-wave transformers:
 - where the load impedance is larger than the line impedance;
 - where the line impedance is larger than the load impedance;

In all simulations, the cell size corresponds to 20 cells per wavelength.

10. Open Circuit Load.

Figure 7 shows the magnitude of the voltage along a transmission line that is open circuited at the load end. The line is 18 m long, has a characteristic impedance (R_o) of $50\ \Omega$, and the $v_{ph} = 200 \times 10^6$. The values of R_o and v_{ph} are chosen to be similar to RG58U cable. The signal is at a frequency of 30 MHz.



In Figure 7, the source is at the left end, and the open circuit load is at the right end. The VSWR along the line is reported by `tline` as between 20 and 60. Certainly, the open circuit condition is easily seen in Figure 7 by the maximum in the voltage at the load end.

11. VSWR and Dispersion.

Figure 8 shows the Voltage Standing Wave Ratio (VSWR) along a transmission line with characteristic impedance (R_o) of $50\ \Omega$, and the $v_{ph} = 200 \times 10^6$. The values of R_o and v_{ph} are chosen to be similar to RG58U cable. The value of R_L is varied from 5 to $300\ \Omega$. The signal is at a frequency of 30 MHz.

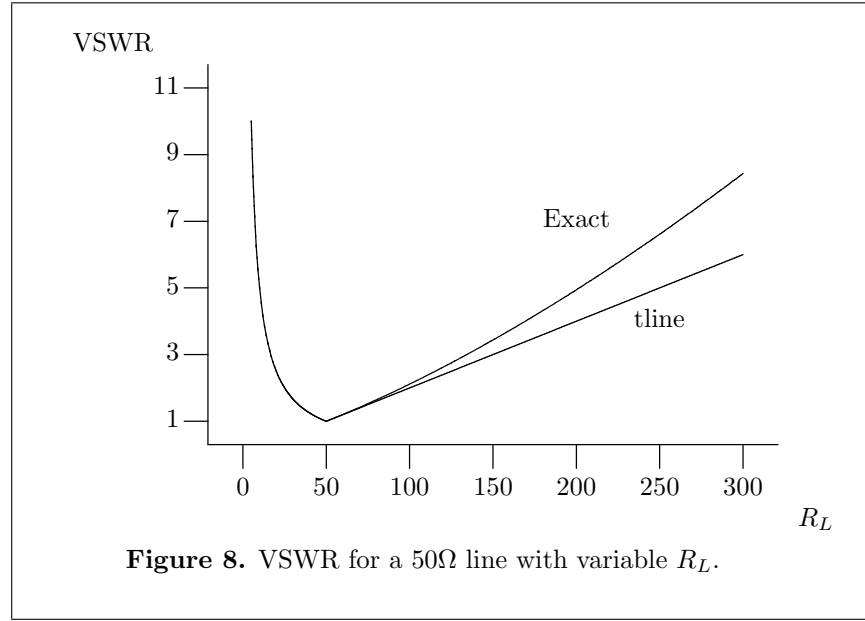
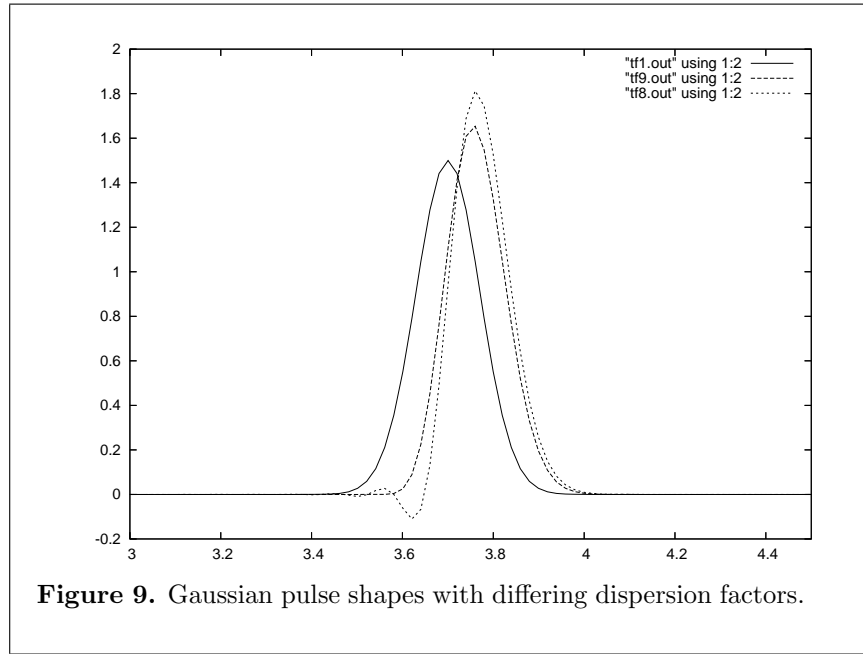


Figure 8. VSWR for a 50Ω line with variable R_L .

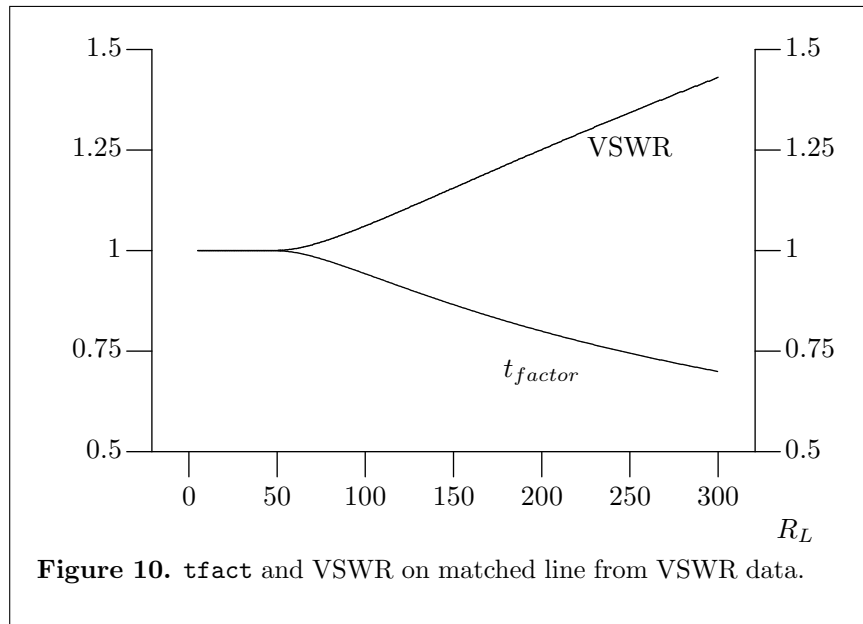
Figure 8 demonstrates that the VSWR calculated does reach 1 at $50\ \Omega$, and the simulation results match the exact value for $R_L < R_o$. As R_L increases past R_o , the simulation results do not agree with theory.

The reason for the discrepancy is related to the value of **tfact**. Recall that the software tests for divergence and lowers the value of **tfact** until the simulation has successfully completed, i.e., steady state has been reached with no divergent values of V or I . Simulations with $R_L < R_o$ do not require any changes to **tfact**.

However, when $R_L > R_o$, **tfact** is reduced. When **tfact** is reduced from the magic time step, dispersion is introduced. The dispersion results in changes to the pulse shape. For example, Figure 9 shows the pulse shape of a Gaussian pulse under three different **tfact** values (or equivalently, for three different dispersion rates).

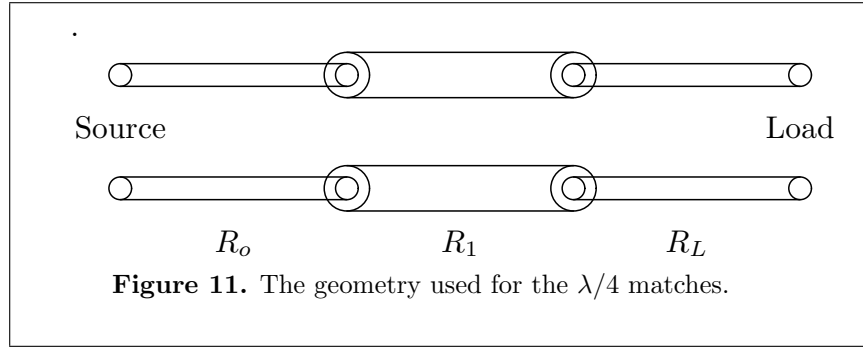


In Figure 10, the value of t_{fact} is plotted vs. R_L from the VSWR simulations. Also shown in Figure 10 is the value of VSWR on the load section of line, which should have a VSWR of one in this case. As the value of t_{fact} is forced smaller to avoid divergent results, the value of VSWR on the matched section rises.



12. Quarter-Wave Transformers.

The code essentially uses three sections of transmission line to perform the simulations. On the left end of the lines a source is placed, and the line is matched by a generator impedance so that the waves do not reflect off the source. On the right end of the lines, the user can choose an open circuit, a short circuit, or a matched condition. Figure 11 shows the 3 sections.



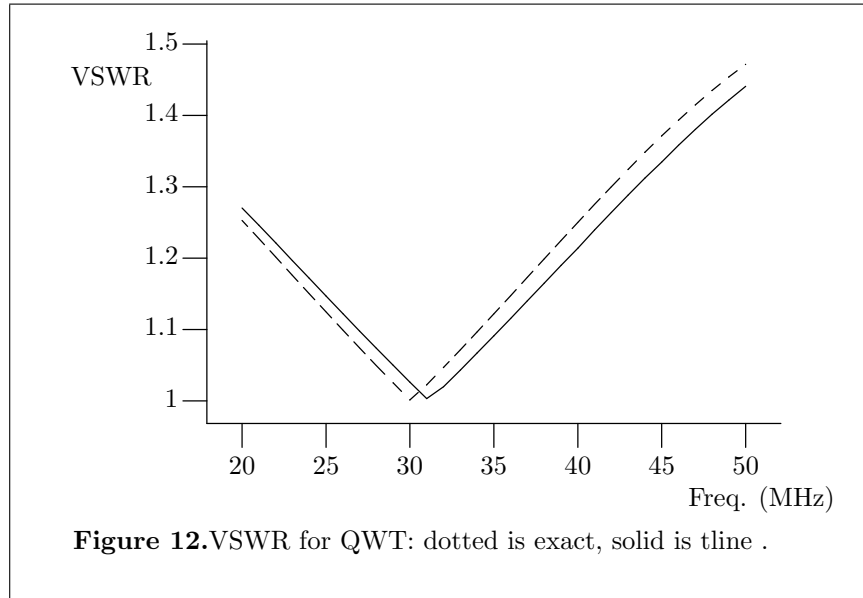
First Example: Load Impedance Smaller Than Characteristic Impedance

Here, $R_L = 32\Omega$, $R_o = 50\Omega$, all phase velocities are 2×10^8 m/s, the matching section, R_1 has an impedance of

$$R_1 = \sqrt{32 \times 50} = 40\Omega$$

and, since $\lambda = 6.666$ m on the line at 30 MHz, the quarter wave section is 1.666 m long.

The simulation was performed at a variety of frequencies from 20 to 50 MHz. The VSWR on the first section of line (connected to the source) would ideally have VSWR=1 at 30 MHz, and VSWR would increase as the frequency deviates from 30 MHz. The results of the simulation are shown in Figure 12.



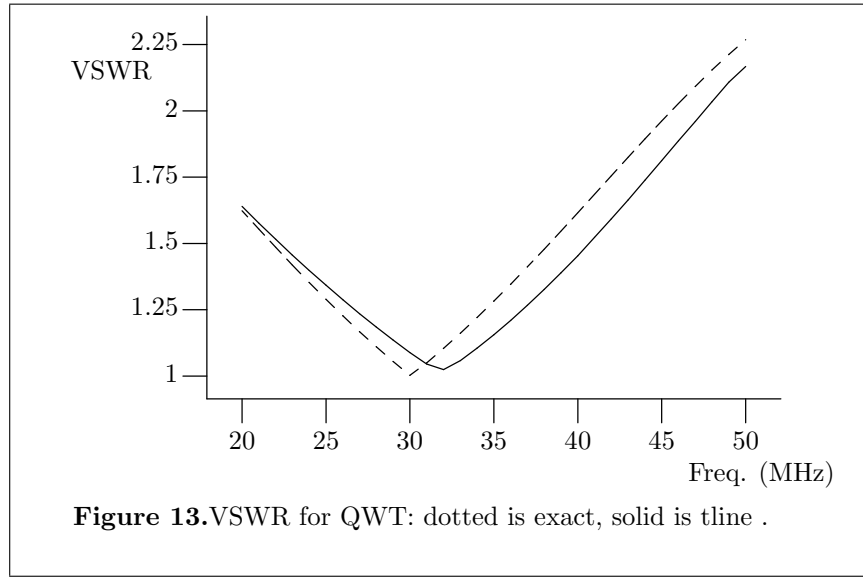
Second Example: Load Impedance Larger Than Characteristic Impedance

Here, $R_L = 128\Omega$, $R_o = 50\Omega$, all phase velocities are 2×10^8 m/s, the matching section, R_1 has an impedance of

$$R_1 = \sqrt{128 \times 50} = 80\Omega$$

and, since $\lambda = 6.666$ m on the line at 30 MHz, the quarter wave section is 1.666 m long.

The simulation was performed at a variety of frequencies from 20 to 50 MHz. The VSWR on the first section of line (connected to the source) would ideally have VSWR=1 at 30 MHz, and VSWR would increase as the frequency deviates from 30 MHz. The results of the simulation are shown in Figure 13.



Discussion

In both examples, the tline simulation results are slightly shifted in frequency compared to the exact VSWR values. This is believed to be due to the discretization of the line. Discretizing the line causes the length of the line to be slightly different than the actually required line length.

In the first example, no dispersion is encountered because the impedances are from larger to smaller as one moves from the source to the load. However, in the second example, the effect of dispersion could be significant. Careful inspection of Figure 13 reveals a slight curvature in the tline simulation results for frequencies above 30 MHz. The effect of dispersion in the second example is very small.

13. Code Description.

The code is listed and described here. First, the makefile is provided, followed by header files and main for the code. Each function that is listed in the prototypes of main is a separate file and follow main in the order of the prototype list.

Major sections of the code listing are:

- Makefile to illustrate how the code is compiled and how the documentation is assembled;
- Header Files which hold global variables and `#define` statements;
- Main to illustrate an overview of the code;
- Solver that computes the fields in time;
- Post Processing to illustrate how the engineering parameters or video clips are assembled;

An index is also provided at the very end of this report.

The documentation is assembled using a set of files, including the web file, `tline.w`, and a file `tline.jrd`, holding a variety of files that are parsed out of `tline.jrd` as needed, such as commands to build the graphics in the documentation. The `mfile.*` files are used to typeset the makefile.

The `libcpp.a` library [12] is also used for file manipulation (opening, closing, etc.). The `libcpp.a` library has been developed as part of the software used in the Electromagnetic Simulations Laboratory (ESL) at Marquette University.

14. Makefile.

Here is the makefile for the code and the documentation.

```
#
# Makefile for tline
#
CC=g++ -g
LOCAL=/home/richiej/bin/local
INCLS=$(LOCAL)/include
LIBS=$(LOCAL)/lib
#
OBJS= main.o inputs.o setup.o iterat.o vswr.o print.o scripts.o
#
        all: tline tline.ps
#
        tline: $(OBJS)
                $(CC) -L$(LIBS) -o tline $(OBJS) -lcpp -lm
#
# compiling the OBJ's
#
        main.o: main.cc
                $(CC) -I$(INCLS) -c main.cc
        inputs.o: inputs.cc
                $(CC) -I$(INCLS) -c inputs.cc
        setup.o: setup.cc
                $(CC) -I$(INCLS) -c setup.cc
        iterat.o: iterat.cc
                $(CC) -I$(INCLS) -c iterat.cc
        vswr.o: vswr.cc
                $(CC) -I$(INCLS) -c vswr.cc
        print.o: print.cc
                $(CC) -I$(INCLS) -c print.cc
        scripts.o: scripts.cc
                $(CC) -I$(INCLS) -c scripts.cc
#
# Section to create documentation
#
        tline.ps: tline.dvi
                dvips -t letter -f tline.dvi >tline.ps
        tline.dvi: tline.tex makefile.tex tline.w alpha.tex 2pa.tex
                2pb.tex geo.tex 8pa.tex 8pb.tex bc1.tex bc2.tex
                upic.tex $(HOME)/admin/lib/jr.bib results.jrd
                tex tline; bibtex tline; tex tline ; tex tline
        tline.tex: tline.w
                cweave tline.w;
#
# graphics
#
        2pa.tex: 2pa.pic
                gpic -t 2pa.pic >2pa.tex
```

```

2pa.pic: tline.jrd
    awk -F "|" '$$1=="2pa" print $$2' tline.jrd > 2pa.pic
2pb.tex: 2pb.pic
    gpics -t 2pb.pic >2pb.tex
2pb.pic: tline.jrd
    awk -F "|" '$$1=="2pb" print $$2' tline.jrd > 2pb.pic
geo.tex: geo.pic
    gpics -t geo.pic >geo.tex
geo.pic: tline.jrd
    awk -F "|" '$$1=="geo" print $$2' tline.jrd > geo.pic
8pa.tex: 8pa.pic
    gpics -t 8pa.pic >8pa.tex
8pa.pic: tline.jrd
    awk -F "|" '$$1=="8pa" print $$2' tline.jrd > 8pa.pic
8pb.tex: 8pb.pic
    gpics -t 8pb.pic >8pb.tex
8pb.pic: tline.jrd
    awk -F "|" '$$1=="8pb" print $$2' tline.jrd > 8pb.pic
alpha.tex: alpha.pic
    gpics -t alpha.pic > alpha.tex
alpha.pic: tline.jrd
    awk -F "|" '$$1=="alpha" print $$2' tline.jrd >alpha.pic
bc1.tex: bc1.pic
    gpics -t bc1.pic >bc1.tex
bc1.pic: tline.jrd
    awk -F "|" '$$1=="bc1" print $$2' tline.jrd > bc1.pic
bc2.tex: bc2.pic
    gpics -t bc2.pic >bc2.tex
bc2.pic: tline.jrd
    awk -F "|" '$$1=="bc2" print $$2' tline.jrd > bc2.pic
upic.tex: upic.pic
    gpics -t upic.pic >upic.tex
upic.pic: tline.jrd
    awk -F "|" '$$1=="upic" print $$2' tline.jrd > upic.pic

#
# Makefile typeset
#
    makefile.tex: makefile
        sed -f mfile.sed makefile>makefile.tmp;
        awk -f mfile.awk -F "|" makefile.tmp >makefile.tex

#
# Admin Section
#

    move: tline
        mv tline $(HOME)/sims/fd1d
    clean:
        rm -f *.o tline *.out * *.tex *.dvi
        *.idx *.log *.scn *.toc *.jpg *.eps
        *.tmp *.pic *.ps tline.tibx jr.ref INDEX
        *.blg *.aux *.bbl
    tar:
        tar -cf tline3.tar

```

```

        *.cc *.h makefile mfile.* tline.w tline.jrd
pack: clean tar
      gzip -9 tline3.tar;
      mv tline3.tar.gz /home/richiej/tar/pack/tline3.tgz

```

15. Header file. Filename: fd1d.h

This is the header file, **fd1d.h**. This file contains all the global variables. The file is included in all code files except **main**.

```

extern double V[2][5005];    /* electric field */
extern double I[2][5005];    /* magnetic field */
extern double re[5005], rh[5005]; /* major iteration parameters */
;
; /* miscellaneous iteration parameters */
;
extern double eta0, Vp1, Vp2, Vp3, dt, dx1, dx2, dx3, R1, R2, R3;
extern double l1, l2, l3, freq, tstart, tstop, Vmax, Imax, Vmin, Imin;
extern int k1, k2, kmax, Nmax, OpenShortFlag, TimeFreqFlag, SourceFlag;

```

16. Header file. Filename: std-defs.h

This is the header file, **std-defs.h**. This file contains a few definitions (**#define** statements) that are convenient to make the code more easily read. The file is included in all code files.

```

#define COMPLEXstd::complex < double >
#define CIN std::cin
#define COUT std::cout

```

17. Main. Filename: `main.cc`

This is the main function, or entry point to the code. Functions listed here are listed in later sections of the documentation.

The main function consists of a large `do` loop. This loop is used to allow for re-initiation of the solution if divergent results are detected.

```
#include <stdio.h>
#include <iostream>
#include <stdlib.h>
#include <math.h>
#include "cpp-fcns.h"
#include "std-def.h"    /*
```

Prototypes

```
*/
void inputs(void);
void setup(double lfact, double tfact);
void iterate(int n);
void vswr(double *e);
void printout(int n, int count);
void scripts(int count);    /*
```

Global Variables

```
*/
double V[2][5005];    /* electric field */
double I[2][5005];    /* magnetic field */
double re[5005], rh[5005];    /* major iteration parameters */
;
;    /* miscellaneous iteration parameters */
;
double eta0, Vp1, Vp2, Vp3, dt, dx1, dx2, dx3, R1, R2, R3;
double l1, l2, l3, freq, tstart, tstop, Vmax, Imax, Vmin, Imin;
int k1, k2, kmax, Nmax, OpenShortFlag, TimeFreqFlag, SourceFlag;    /*
```

Main/Entry point for software

```
*/
int main()
{
    int i, k, n, count, Nhold;
    double tfact, lfact, tfSave, time;
    double Emax[5005], Hmax[5005];
    char fileOut[NAME_SIZE];
    std::fstream fout;    /*
```

input section

```
*/
    inputs();
    lfact = 20.;
```

```

    tfact = 1.0;
    COUT << "#_tfact_=" << tfact << "\n\n";    /*

    . . . . . Beginning of do loop

    */
    do {
        tfSave = tfact;    /*

set up parameters

    */
        setup(lfact, tfact);    /*

set up max and min for plots

    */
        Vmax = -999;
        Vmin = 999;
        Imax = -999;
        Imin = 999;    /*

set up iterations

    */
        n = 0;
        count = 11;
        Nhold = int(double(Nmax) * double(47./50.));
        COUT << "#_Begin_holding_max_at_n_=" << Nhold << "\n";
        for (k = 0; k < 5005; k++) {
            Emax[k] = 0.;
            Hmax[k] = 0.;
        }    /*

Iteration Loop

    */
        for (n = 0; n < Nmax; n++) {
            iterate(n);    /*

test for divergence

    */
        for (k = 1; k < kmax; k++) {
            if (fabs(V[0][k]) > 1000) {
                COUT << "#_Results_diverged.\n\n";
                tfact = tfact - 0.001;
                COUT << "#_tfact_=" << tfact << "\n\n";
                n = Nmax + 2;    /* to break off simulations */
            }
        }
        if (TimeFreqFlag == 1) {
            time = (double) n * dt;
            if ((¬(n % 3)) ∧ (time > tstart) ∧ (time < tstop)) {
                ;    /* printout is used to get video clips */
                printout(n, count);
                count++;
            }
        }
    }
}

```

```

    }
  }
  if (TimeFreqFlag == 0) { /*
record max values after reaching steady state

  */
    if (n > Nhold) {
      for (k = 1; k < kmax; k++) {
        if (fabs(V[0][k]) > Emax[k]) Emax[k] = fabs(V[0][k]);
        if (fabs(I[0][k]) > Hmax[k]) Hmax[k] = fabs(I[0][k]);
      }
    }
  }
} while (tfact != tfSave); /*

. . . . . End of do loop

*/
if (TimeFreqFlag == 1) scripts(count); /* scripts is used to get video clips */
if (TimeFreqFlag == 0) {
  ; /*
here, print out the max values

  */
  sprintf(fileOut, "VImax.out");
  OpenOutputFile(fileOut, fout);
  for (k = 1; k < kmax; k++) {
    fout << k << "░░" << Emax[k] << "░░" << Hmax[k] << "\n";
  }
  fout.close();
  vswr(Emax);
}
return 0;
}

```

18. Inputs. Filename: inputs.cc

This portion of the code is used to obtain the input information from the user. The variables that are obtained are all global variables (listed in `fd1d.h`). In addition, the length of one wavelength at 30 MHz for each section of line is printed to aid in assembling sections of line that are specific wavelengths, such as a quarter wave section of line. The data is obtained using `getinputd` and `getinputi`, from the `libc++.a` library [12].

```
#include <iostream>
#include "cpp-fcns.h"
#include <math.h>
#include "fd1d.h"
#include "std-def.h"
void inputs(void)
{
    double pi, mu0, eps0, v1, v2, v3, f, c;
    pi = 4. * atan(1.);
    mu0 = 4. * pi * 1. * 10-7;
    eps0 = 8.854 * 10-12;
    TimeFreqFlag = getinputi("#_Enter_1_for_time_domain,_0_for_frequency_domain_results");
    if (TimeFreqFlag == 1) SourceFlag = getinputi("#_Gaussian_(1)_or_sinusoidal_source_(0)");
    if (TimeFreqFlag == 0) SourceFlag = 0;
    COUT << "#_Enter_the_Characteristic_Impedances_(real):\n\n";
    R1 = getinputd("#_R_0");
    R2 = getinputd("#_R_1");
    R3 = getinputd("#_R_L");
    COUT << "#_Enter_the_phase_velocities:\n\n";
    Vp1 = getinputd("#_Vp1");
    Vp2 = getinputd("#_Vp2");
    Vp3 = getinputd("#_Vp3");
    COUT << "\n#_Wavelength_(in_meters_at_30_MHz)_for_each_section:\n\n";
    f = 30. * 106;
    COUT << "#_Section_1:_ << Vp1/f << "\n";
    COUT << "#_Section_2:_ << Vp2/f << "\n";
    COUT << "#_Section_3:_ << Vp3/f << "\n";
    COUT << "\n#_Enter_the_line_lengths_(in_meters):\n\n";
    l1 = getinputd("#_length_1");
    l2 = getinputd("#_length_2");
    l3 = getinputd("#_length_3");
    if (SourceFlag == 0) {
        freq = getinputd("\n#_Enter_the_center_frequency_(for_sine_source,_in_MHz)");
        freq = freq * 1. * 106;
    }
    if (TimeFreqFlag == 1) {
        COUT << "\n#_T_1_is_ << l1/Vp1 << "\n";
        COUT << "#_T_2_is_ << l2/Vp2 << "\n";
        COUT << "#_T_3_is_ << l3/Vp3 << "\n";
        tstart = getinputd("#_input_start_time");
        tstop = getinputd("#_input_stop_time");
    }
}
```

```

COUT << "#_TERMINATION:\n";
;
COUT << "#_Enter_1_for_open_circuit,\n";
COUT << "#_Enter_-1_for_short_circuit,\n";
OpenShortFlag = getinputi("#_Enter_0_for_neither");
COUT << "\n";
return;
}

```

19. Solver.

In this major section of the documentation, the code is listed that performs the task of computing the fields. The iterations are performed in a marching-on-in-time fashion.

This section holds the code for setup and iterate, two major functions in the code. In the future, the function iterate should also be broken to include functions just for the source and just for the boundary conditions.

20. setup. Filename: `setup.cc`

This section of the code sets up the simulations as described in part C, Implementation, of the Theory section.

```

#include <stdlib.h>
#include <iostream>
#include <math.h>
#include "fd1d.h"
#include "std-def.h"
#define pi M_PI
void setup(double lfact, double tfact)
{
    int i, n, k;
    double fmax, c, lambda, tmp, eLrge, induct, cap, Nreal;
    fmax = 100. * 106; /*
set up dx's and dt. First, base dt on largest  $v_{ph}$  and maximum frequency of 100 MHz
*/
    eLrge = Vp1;
    if (Vp2 > eLrge) eLrge = Vp2;
    if (Vp3 > eLrge) eLrge = Vp3;
    lambda = eLrge / fmax;
    tmp = lambda / lfact;
    dt = tfact * tmp / eLrge; /*
find dx1 and number of time steps to travel to end of length 1
*/
    lambda = Vp1 / fmax;
    dx1 = lambda / lfact;
    Nreal = l1 / (Vp1 * dt); /*
find dx2 and number of time steps to travel to end of length 2
*/
    lambda = Vp2 / fmax;
    dx2 = lambda / lfact;
    Nreal += l2 / (Vp2 * dt); /*
find dx3 and number of time steps to travel to end of length 3
*/
    lambda = Vp3 / fmax;
    dx3 = lambda / lfact;
    Nreal += l3 / (Vp3 * dt); /*
assume steady state reached after 50 traverses of line
*/
    Nmax = int(50. * Nreal); /*
print summary information so far
*/

```

```

COUT << "#_Nmax=_ " << Nmax << "\n";
COUT << "#_dt=_ " << dt << "\n";
COUT << "#_dx1=_ " << dx1 << "\n";
COUT << "#_dx2=_ " << dx2 << "\n";
COUT << "#_dx3=_ " << dx3 << "\n"; /*

```

here, set k values

```

*/
k1 = int(l1/dx1);
k2 = int(l2/dx2 + k1);
kmax = int(l3/dx3 + k2);
if (kmax > 5000) {
    COUT << "\n\nnkmax_too_large\n\n";
    exit(1);
}
COUT << "#_k1=_ " << k1 << ",_k2=_ " << k2 << ",_kmax=_ " << kmax << "\n"; /*

```

fill the re and rh arrays

```

*/
induct = R1/Vp1;
cap = 1./(R1 * Vp1);
COUT << "inductance/capacitance_for_sectionA:_ " << induct << "_ " << cap << "\n";
for (i = 0; i < k1; i++) {
    re[i] = dt/(induct * dx1);
    rh[i] = dt/(cap * dx1);
}
induct = R2/Vp2;
cap = 1./(R2 * Vp2);
COUT << "inductance/capacitance_for_sectionB:_ " << induct << "_ " << cap << "\n";
for (i = k1; i < k2; i++) {
    re[i] = dt/(induct * dx2);
    rh[i] = dt/(cap * dx2);
}
induct = R3/Vp3;
cap = 1./(R3 * Vp3);
COUT << "inductance/capacitance_for_sectionC:_ " << induct << "_ " << cap << "\n";
for (i = k2; i < kmax; i++) {
    re[i] = dt/(induct * dx3);
    rh[i] = dt/(cap * dx3);
} /* eta for source */
eta0 = R1; /*

```

initialize V and I arrays

```

*/
for (k = 0; k < kmax; k++) {
    V[0][k] = 0.;
    I[0][k] = 0.;
    V[1][k] = 0.;
    I[1][k] = 0.;
}
}

```

21. iterate. Filename: `iterat.cc`

Here is the iteration process, as written in Section 5, the Iteration Process. The source and boundary condition implementation are separate functions to allow various scenarios.

```

#include <math.h>
#include "fd1d.h"
#include "std-def.h"
#define pi M_PI
void iterate(int n)
{
    int k;
    double No, Nd;
    No = 15.;
    Nd = 5.;
    ; /*

update I values

    */
    for (k = 1; k < kmax; k++) {
        I[1][k + 1] = I[0][k + 1] - re[k] * (V[0][k + 1] - V[0][k]);
        if ((k == 2)) /* Source is added here */
        {
            if (SourceFlag == 0) {}
            I[1][k + 1] += 0.01 * sin(2. * pi * freq * (double) n * dt);
            if (SourceFlag == 1) I[1][k + 1] += 0.03 * exp(-(n - No) * (n - No) / (Nd * Nd));
        }
    }
    if (OpenShortFlag == 1) I[1][kmax - 1] = 0.; /* to implement an open circuit */
    ; /*

update V values

    */
    for (k = 1; k < kmax; k++) {
        V[1][k] = V[0][k] - rh[k] * (I[1][k + 1] - I[1][k]);
        if ((k == 2)) /* Source is added here, can generator resistance be included? */
        {
            if (SourceFlag == 0) V[1][k] += 0.01 * eta0 * sin(2. * pi * freq * (double) n * dt);
            if (SourceFlag == 1) V[1][k] += 0.03 * eta0 * exp(-(n - No) * (n - No) / (Nd * Nd));
        }
    }
    ; /*

apply boundary conditions

    */
    V[1][kmax] = V[0][kmax - 1] + ((Vp3 * dt - dx3) / (Vp3 * dt + dx3)) * (V[1][kmax - 1] - V[0][kmax]);
    V[1][1] = V[0][2] + ((Vp1 * dt - dx1) / (Vp1 * dt + dx1)) * (V[1][2] - V[0][1]);
    ; /*

test for short circuit boundary condition at load end

    */

```

```

    if (OpenShortFlag  $\equiv -1$ ) V[1][kmax - 1] = 0.;    /*

V[1][kmax-1]=75.*I[1][kmax-1];  implements (correctly) a load resistor of 75 ohms.  This
can be implemented in the future to allow true load resistances.

    */
    ;    /*

update values

    */
    for (k = 1; k < kmax; k++) {
        V[0][k] = V[1][k];
        I[0][k] = I[1][k];
    }
}

```

22. Post Processing.

The post processing that is available depends on whether a time domain or a frequency domain solution is requested. For time domain simulations, snapshots are printed and a gnuplot script is generated, suitable for the creation of a video clip.

For frequency domain simulations, the VSWR on each section of the line is printed to standard output. In addition, the maximum values of V and I are printed to a file for each location along the transmission line.

23. VSWR. Filename: `vswr.cc`

This portion of the code computes the voltage standing wave ratio (VSWR) on each section of the line.

```
#include <iostream>
#include "fd1d.h"
#include "std-def.h"
void vswr(double *e)
{
    int i, k, index[4];
    double max, min, val;
    index[0] = 3;
    index[1] = k1;
    index[2] = k2;
    index[3] = kmax;
    for (i = 0; i < 3; i++) {
        max = 0.;
        min = 999.;
        for (k = index[i] + 1; k < index[i + 1] - 1; k++) {
            val = *(e + k);
            if (val > max) max = val;
            if (val < min) min = val;
        }
        COUT << "\n\n#_-----\n";
        COUT << "\n#_Section_" << i + 1 << "\n\n";
        COUT << "#_Maximum_Voltage_is_" << max << ",_Minimum_Voltage_is_" << min << "\n\n";
        if (min > 0) COUT << "#_VSWR_" << max/min << "\n\n";
        if (min == 0) COUT << "#_VSWR_is_infinite\n\n";
    }
    return;
}
```

24. Printout. Filename: `print.cc`

This is the code that prints the data to a sequence of files that can be used to develop computer video clips.

```
#include <stdio.h>
#include <iostream>
#include "fd1d.h"
#include "std-def.h"
#include "cpp-fcns.h"
void printout(int n,int count)
{
    int k;
    double xval;
    char fileOut[NAME_SIZE];
    std::fstream fout;
    sprintf(fileOut,"%d.out",count);
    OpenOutputFile(fileOut,fout);
    for (k = 0; k < kmax; k++) {
        if (k < k1) xval = dx1 * (double) k;
        if ((k ≥ k1) ∧ (k < k2)) xval = dx1 * (k1 - 1.) + (k - (k1 - 1.)) * dx2;
        if (k ≥ k2) xval = dx1 * (k1 - 1.) + dx2 * (k2 - k1) + (k - (k2 - 1.)) * dx3;
        fout << xval << "░░" << V[0][k] << "░░" << I[0][k] << "\n";
        if (V[0][k] > Vmax) Vmax = V[0][k];
        if (V[0][k] < Vmin) Vmin = V[0][k];
        if (I[0][k] > Imax) Imax = I[0][k];
        if (I[0][k] < Imin) Imin = I[0][k];
    }
    fout.close();
}
```

25. scripts. Filename: `scripts.cc`

This is the code that prints gnuplot files that can be used to animate the data from the simulation.

```
#include <stdio.h>
#include <iostream>
#include "cpp-fcns.h"
#include "fd1d.h"
#include "std-def.h"
void scripts(int count)
{
    int k;
    char vfile[NAME_SIZE], ifile[NAME_SIZE], vfilg[NAME_SIZE], ifilg[NAME_SIZE];
    std::fstream vfout, ifout, vfjpg, ifjpg;

    sprintf(vfile, "v.plt");
    OpenOutputFile(vfile, vfout);
    sprintf(ifile, "i.plt");
    OpenOutputFile(ifile, ifout);
    sprintf(vfilg, "v-jpg.plt");
    OpenOutputFile(vfilg, vfjpg);
    sprintf(ifilg, "i-jpg.plt");
    OpenOutputFile(ifilg, ifjpg);
    Vmax = Vmax * 1.05;
    Vmin = Vmin * 1.05;
    Imax = Imax * 1.05;
    Imin = Imin * 1.05;
    vfout << "set_yrange[" << Vmin << ":" << Vmax << "]\n";
    vfout << "#set_nokey\n";
    vfjpg << "set_yrange[" << Vmin << ":" << Vmax << "]\n";
    vfjpg << "set_term_postscript_eps_monochrome\n";
    ifout << "set_yrange[" << Imin << ":" << Imax << "]\n";
    ifout << "#set_nokey\n";
    ifjpg << "set_yrange[" << Imin << ":" << Imax << "]\n";
    ifjpg << "set_term_postscript_eps_monochrome\n";
    for (k = 11; k < count; k++) {
        vfout << "plot\" << k << ".out\"_using_1:2_with_lines\n";
        vfout << "pause_1\n";
        ifout << "plot\" << k << ".out\"_using_1:3_with_lines\n";
        ifout << "pause_1\n";
        vfjpg << "set_output\" << k << "v.eps\n";
        vfjpg << "plot\" << k << ".out\"_using_1:2_with_lines\n";
        ifjpg << "set_output\" << k << "i.eps\n";
        ifjpg << "plot\" << k << ".out\"_using_1:3_with_lines\n";
    }
    vfout.close();
    ifout.close();
}
```

26. References.

- [1] P. W. Tuinenga, *Spice: A Guide to Circuit Simulation and Analysis Using PSpice*. Englewood Cliffs, New Jersey: Prentice Hall, 3 ed., 1995.
- [2] T. Quarles, A. R. Newton, D. O. Pederson, and A. Sangiovanni-Vincentelli, *SPICE3 Version 3f3 User's Manual*. University of California, Berkeley, CA, 1993.
- [3] K. D. Granzow, *Digital Transmission Lines: Computer Modelling and Analysis*. New York, NY: Oxford University Press, 1998.
- [4] C. W. Trueman, "Animating transmission-line transients with bounce," *IEEE Trans. Educ.*, vol. 46, pp. 115–123, Feb. 2003.
- [5] C. W. Trueman, "Interactive transmission line computer program for undergraduate teaching," *IEEE Trans. Educ.*, vol. 43, pp. 1–14, Feb. 2000.
- [6] D. B. Davidson, *Computational Electromagnetics for RF and Microwave Engineering*. New York, NY: Cambridge University Press, 2005.
- [7] M. N. O. Sadiku, *Numerical Techniques in Electromagnetics*. New York, NY: CRC Press, second ed., 2000.
- [8] A. Taflove, *Computational Electrodynamics: The Finite-Difference Time-Domain Method*. Boston, Massachusetts: Artech House, 1995.
- [9] W. L. Stutzman and G. A. Thiele, *Antenna Theory and Design*. New York, NY: John Wiley Sons, second ed., 1998.
- [10] B. Engquist and A. Majda, "Absorbing boundary conditions for the numerical simulation of waves," *Mathematics of Computation*, vol. 31, pp. 629–651, 1977.
- [11] G. Mur, "Absorbing boundary conditions for the finite-difference approximation of the time-domain electromagnetic field equations," *IEEE Trans. Electromagn. Compat.*, vol. 23, pp. 377–382, 1981.
- [12] J. E. Richie, "LCPP: A library for input and output operations," Tech. Rep. 15, Marquette University, Electromagnetic Simulations Laboratory, Milwaukee, WI, Mar. 2004.

27. Index.

atan: 18.
c: 18, 20.
cap: 20.
cin: 16.
CIN: 16.
close: 17, 24, 25.
COMPLEX: 16.
complex: 16.
count: 17, 24, 25.
cout: 16.
COUT: 16, 17, 18, 20, 23.
dt: 15, 17, 20, 21.
dx1: 15, 17, 20, 21, 24.
dx2: 15, 17, 20, 24.
dx3: 15, 17, 20, 21, 24.
e: 17, 23.
eLrge: 20.
Emax: 17.
eps0: 18.
eta0: 15, 17, 20, 21.
exit: 20.
exp: 21.
f: 18.
fabs: 17.
fileOut: 17, 24.
fmax: 20.
fout: 17, 24.
freq: 15, 17, 18, 21.
fstream: 17, 24, 25.
getinputd: 18.
getinputi: 18.
Hmax: 17.
I: 15, 17.
i: 17, 20, 23.
ifile: 25.
ifilg: 25.
ifjpg: 25.
ifout: 25.
Imax: 15, 17, 24, 25.
Imin: 15, 17, 24, 25.
index: 23.
induct: 20.
inputs: 17, 18.
iterate: 17, 21.
k: 17, 20, 21, 23, 24, 25.
kmax: 15, 17, 20, 21, 23, 24.
k1: 15, 17, 20, 23, 24.
k2: 15, 17, 20, 23, 24.
lambda: 20.
lfact: 17, 20.
l1: 15, 17, 18, 20.
l2: 15, 17, 18, 20.
l3: 15, 17, 18, 20.
M_PI: 20, 21.
main: 17.
max: 23.
min: 23.
mu0: 18.
n: 17, 20, 21, 24.
NAME_SIZE: 17, 24, 25.
Nd: 21.
Nhold: 17.
Nmax: 15, 17, 20.
No: 21.
Nreal: 20.
OpenOutputFile: 17, 24, 25.
OpenShortFlag: 15, 17, 18, 21.
pi: 18, 20, 21.
printout: 17, 24.
re: 15, 17, 20, 21.
rh: 15, 17, 20, 21.
R1: 15, 17, 18, 20.
R2: 15, 17, 18, 20.
R3: 15, 17, 18, 20.
scripts: 17, 25.
setup: 17, 20.
sin: 21.
SourceFlag: 15, 17, 18, 21.
sprintf: 17, 24, 25.
std: 16, 17, 24, 25.
tfact: 17, 20.
tfSave: 17.
time: 17.
TimeFreqFlag: 15, 17, 18.
tmp: 20.
tstart: 15, 17, 18.
tstop: 15, 17, 18.
V: 15, 17.
val: 23.
vfile: 25.
vfilg: 25.
vfjpg: 25.
vfout: 25.
Vmax: 15, 17, 24, 25.
Vmin: 15, 17, 24, 25.
Vp1: 15, 17, 18, 20, 21.
Vp2: 15, 17, 18, 20.
Vp3: 15, 17, 18, 20, 21.

vswr: 17, 23.

v1: 18.

v2: 18.

v3: 18.

xval: 24.

TLINE

	Section	Page
Theory	1	2
Finite Difference Equations	2	2
Stability	3	4
Implementation	4	4
Iteration Process	5	6
Source	6	7
Boundary Conditions	7	8
Post Processing	8	10
Results and Discussion	9	11
Open Circuit Load	10	11
VSWR and Dispersion	11	12
Quarter-Wave Transformers	12	14
Code Description	13	16
Makefile	14	17
Header file	15	19
Header file	16	19
Main	17	20
Inputs	18	23
Solver	19	24
setup	20	25
iterate	21	27
Post Processing	22	28
VSWR	23	29
Printout	24	30
scripts	25	31
References	26	32
Index	27	33