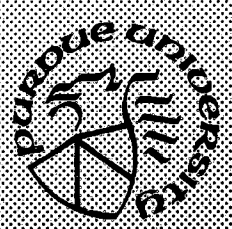
1-1-1990

TARDIS: A Numerical Simulation Package for Drive Systems

W. Suwanwisoot
Purdue University

C. M. Ong
Purdue University



TARDIS: A Numerical Simulation Package for Drive Systems

W. Suwanwisoot
C. M. Ong

TR-EE 90-3
January, 1990

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

PREFACE

TARDIS is a digital computer simulation package originally intended to simulate drive systems. Its versatility has proved to be very useful in other areas as well. It is designed to replace the functionality of analog computers so that the user who cannot afford to have one can use the program on a personal or mainframe computer . The author believes that it is one of the most efficient and accurate simulation programs of its kind at this point even though not all of its potential has been exploited to the fullest. It can handle index 0 and 1 differential-algebraic systems with discontinuities.

The author's intention in creating this package is to help researchers with a simulation tool that will eventually result in a better quality of living. PLEASE DO NOT USE THIS PACKAGE TO DESIGN WEAPONS. IF THAT IS NOT POSSIBLE, PLEASE AT LEAST MAKE IT THE VERY LAST CHOICE FOR WEAPON SIMULATION. IN ANY CASE THE USER WILL BE RESPONSIBLE FOR ANY PROPERTY DAMAGE, INJURY, OR LOSS OF LIFE AS A DIRECT OR INDIRECT RESULT OF THE USE OF THIS PACKAGE.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

TARDIS is a differential-algebraic equation solver with discontinuity handling capability. It can be used with a language translator to create a complete simulation package with user-interface. Written in C, TARDIS is intended for solving a system of differential-algebraic equations with index 0 and 1 only. The integration part of TARDIS is the variable-step, variable-order Gear algorithm with new local truncation error control. The objective of the control is to have one iteration per time step to reduce the total number of calls to the routine containing system equations.

TARDIS allows two types of discrete events: state and scheduled events. For locating state events, TARDIS uses a simple interpolation scheme which is found to be working accurately and efficiently. The scheme requires integration to the points of discontinuities to avoid locating false state events. TARDIS handles discrete events by using finite state machines. TARDIS also uses sparse matrix techniques to reduce computation for large systems.

CHAPTER 1

SIMULATION PROGRAMS FOR ELECTRIC DRIVE SYSTEMS

1.1. What are Electric Drive Systems?

The term "electric drive system" refers to a wide variety of electric machines system in industrial and non-industrial applications where position, speed, torque, or power are to be controlled to better match the load characteristics. A drive system can be as simple as an adjustable speed electric fan or as sophisticated as a computer-controlled manipulator. The power rating of drive systems ranges from a fractional horsepower to more than one million horsepower.

Both ac and dc motors are used in drive systems, though ac motors are gradually replacing dc motors in many applications because they require less maintenance and cost less. However, controls for ac motors are often much more complicated than those of dc motors, in order to achieve a response as fast as that of dc motors. Consequently, dc motors can still be found in some low-power and less expensive applications. The ac motors used in drive systems may be synchronous motors, induction motors, or reluctance motors [1] for the larger horsepower units, and permanent magnet motors or stepper motors for smaller horsepower units.

In modern electric drives, the voltage and current supplied to the electric machine is electronically regulated by power semiconductor devices. The kinds of power semiconductors devices presently in use include thyristors, diodes, gate turn-off thyristors (GTO), power MOSFET, bipolar junction transistors (BJT), insulated gate bipolar transistors (IGBT), and MOS-controlled thyristors (MCT). These fast acting devices perform the switching needed to shape the current or voltage supplied to the machine.

The switching control and/or the higher level control are usually done by computers or microprocessors. Besides controlling the output position, speed, or torque, more sophisticated control may include the minimization of power loss, as in induction motors [2]; torque pulsation reduction, as in current-fed induction motor drives [3]; harmonic elimination, as in voltage source inverter [4], etc.

The technology and the control methods are continually changing; several new devices and ideas are emerging which will further improve drives' performance. The main question still is cost effectiveness. As the ratio of cost to performance of drive and controller decreases, and control methods become more sophisticated in requiring minimal sensors, we will see greater use of drive systems.

1.2. Simulation of Drive Systems on Digital Computers

The simulation of modern drive systems on digital computers is very complicated due to the following reasons. First, the differential equations describing the behavior of the motors or the control are often nonlinear. This is usually not a problem since there are many excellent differential equation solvers that can handle such nonlinearity. The solvers may come in the form of ready-to-use application-specific package or subprograms. Second, the switching action of the power semiconductor devices or even some control parts may introduce discontinuities in the form of a change in the structure of the systems or a change in the values of the device parameters. If the differential equation solver is not specifically designed to handle discontinuities, it may be unable to handle them or very inefficient at handling them.

Existing simulation packages may be loosely divided into two categories: general-purpose and application-specific. Most general-purpose packages will require the input in the form of differential or differential-algebraic equations. Such packages are also referred to as equation-oriented packages. To use them, the user will have to derive the system equations by

hand and put them in the format required by the package. On the other hand, application-specific simulation packages provide ready-to-use modules for typical components. The user specifies the interconnections between or relationships of the components in the systems according to some rules imposed by the packages, but seldom has to deal with the system equations directly. Since the interconnection of modules is in a network-like fashion, such packages are also called network-oriented The main disadvantage of application-specific programs is that the capability of the programs will be restricted to whatever models are provided by the programs. There are also simulation packages that are in between the two categories; they let the user specify the equations for the modules and use them in the network-like fashion.

Since some of the ideas used in this research are based on the disclosed features of several existing simulation packages, a brief description of some of them is in order. The first four simulation packages, ACSL, ESL, EASY5, and PSCSP, are the general-purpose ones; the next three, SPICE2, EMTP, and ATOSEC5 are specifically for the simulation of electrical or electronic circuits. The last two, IESE and SABER, are general-purpose electrical network simulation programs whose component definitions are based on a black-box or module concept.

1.2.1. ACSL

Advanced Continuous Simulation Language (ACSL) [5] is a general-purpose simulation package that can handle time-dependent, non-linear systems of differential equations. With the MACRO preprocessor, ACSL may be tailored to any specific application but not in the network-like fashion. The user has to formulate the differential equations of the system and put them into the form required by the package. ACSL provides a wide variety of integration schemes: namely, Runge-Kutta, Adam-Moulton, and Gear algorithm [6]. It also has multi-derivative capability in that slow and fast transients can be integrated with different step sizes or algorithms.

The language used to specify models conforms with the specification laid down by the Continuous System Simulation Language (CSSL) Committee, with extension to handle discontinuities which are located by binary search. ACSL has a sorting capability which lets the user enter the equations in any order.

1.2.2. ESL

ESL [7] is another simulation program based on CSSL. Written in FORTRAN77, ESL comprises of an interpreter and a translator to FORTRAN. ESL uses interpolation to locate the discontinuities. The program accepts the system equations in the form of differential equations, which can also be grouped into subsets, of which only one will be active at a time. The user can define submodels which may contain discontinuities. ESL also provides several default submodels that can be used or modified. Unlike ACSL, ESL does not have sorting capability.

1.2.3. EASY5

EASY5 [8] is a simulation package that has a provision for switch states to simplify the modelling of discrete devices. It requires the user to enter system equations in the form of differential equations. The handling of discontinuities in EASY5 is a slightly modified version of Gear's [9] which uses step size control for output and discontinuities.

1.2.4. PSCSP

The Power Series Continuous-System Simulation Program (PSCSP) [10] takes a different direction from the other simulation packages mentioned before. The program uses semi-analytical methods based on power series expansion for integration and for locating the discontinuities. The program will translate the user's input equation into a FORTRAN subprogram. The step sizes used in the integration are often more than an order of magnitude larger than those used in fourth order Runge-Kutta due to the higher-order integration method used.

1.2.5. SPICE2

SPICE2 is a simulation program for semiconductor circuits. It has many capabilities besides transient analysis. The input to the program is a file describing the interconnections of the devices in the circuit, both active and passive devices. The user can choose either Gear or trapezoidal method for integration, but Nagel, the author of SPICE2, suggests that the trapezoidal algorithm with local truncation error control is preferred [11]. SPICE2 has no capability to handle power semiconductor switches other than modelling them in detail. Also with the models provided, it would not be a trivial problem to use SPICE2 to simulate ac machines in general.

There are several versions of SPICE on the market now. One version of SPICE called IGSPICE lets the user specify equations to describe the behavior of modules. Keyhani and Tsai have used this feature in [12] to simulate a start-up of an induction machine with saturable inductance.

1.2.6. EMTP

The Electro-Magnetic Transients Program (EMTP) [13] is designed for simulating power system components and large scale networks. The program is written mainly in FORTRAN. EMTP uses the trapezoidal method with equal step size for integration; the choice of step size is based on the user's experience with the circuits. The program has several built-in models for transmission lines, circuit breakers, surge arrestors, synchronous machine, thyristors or diodes. It does not seem to have the provisions needed to facilitate the simulation of the kinds of components found in the modern drive systems.

1.2.7. ATOSEC

The simulation program ATOSEC [14] is designed for simulating power electronic circuits where power semiconductor devices are treated as ideal switches. Representing power semiconductor devices as ideal switches makes the simulation run faster than those which use detailed representation. The input language is similar to that used by SPICE2.

ATOSEC can be used to simulate electric machines as long as they can be represented by circuit components provided by the program. The program uses the backward Euler method without any local truncation error control. As with EMTP, the user must have some idea of the circuit response in order to choose the integration step size.

1.2.8. IESE

IESE (Integrated Engineering Simulation Environment) is a graphic-oriented user interface to EMTP or SOLVER-Q [15]. A novice user can specify connections of electrical components graphically, while the more advanced user can define modules' equations. The input is then translated into the language used by EMTP or SOLVER-Q, which does the simulation.

SOLVER-Q [16] is a general-purpose symbolic simulation package for electrical networks. For transient simulation, the program DIFTOALG converts differential equations into algebraic equations using any desired numerical integration algorithm, including all implicit methods. The resulting algebraic equations are solved by a program called SOLVE. It has been reported, however, that SOLVER-Q can be about 10 times slower than EMTP for certain problems.

1.2.9. SABER

SABER [17] is a simulation package that has a powerful user interface, especially for post-processing of data after simulation. It can do many kinds of circuit analyses similar to SPICE2. The package also allows the user to restart the simulation from a previous run, a useful feature for long simulation.

There are other numerical simulation packages in the form of FORTRAN subroutines for solving differential or differential-algebraic equations - e.g., IMSL [18], ODEPAK [19], DASSL [20], etc. When presented with discontinuities, these packages perform poorly since they do not have any discontinuity handling capability other than local truncation error control, which usually reduces the integration step size to very small values.

1.3. Motivation and Objective

Although some of the simulation packages mentioned above can be used to simulate modern drive systems, they are far from providing the most efficient and accurate way to handle mixed discrete-continuous systems. Some of them have been used to simulate simple drive systems - e.g., dc drives or ac drives operated in certain modes only. But for more complicated drive systems, engineers and researchers usually resort to writing their own simulation programs for the specific application at hand in general-purpose programming languages such as FORTRAN.

The objective of this research, then, is to determine the combination of modelling and numerical methods, and the data structures to form a suitable framework for simulating electric drive systems efficiently on digital computers.

The results of this research have been incorporated into a new simulation program called TARDIS. The core of this program is the numerical part that combines several numerical techniques, including a variable-step, variable-order integration with a new local truncation error control, state machines to handle discrete components, and sparse matrix techniques, to maximize its computational efficiency, because it is known that time-domain simulation can be notoriously slow on the digital computer. With these numerical techniques incorporated, TARDIS also achieves the same capabilities as a general-purpose analog computer in terms of functionality.

TARDIS is written in C programming language, and the current version is about 4500 lines long (including some comments). Although, in theory, the program can be written in any computer language, the choice of C over other languages, including FORTRAN which has long been the workhorse for scientific computations, is due to some desirable features in C that do not exist in other languages locally available. For example, during execution TARDIS can adjust its own size according to how big the problem is by asking the operating system to give it more memory space whenever that is needed. TARDIS requests the space through several routines specifically designed for each type of internal data structure. These routines request the

space from the operating system in as small a chunk as 1 Kbyte, and hand out the space with the size needed by the calling routines. TARDIS also has its own space management routines that will reuse unwanted space. Note that if one wants to run the simulation in the standard FORTRAN 77 language which does not have any memory allocation function, one needs to declare a big enough work space. However, when dealing with sparse matrices, the memory space needed in the simulation will be known at run time. So one must guess, based on previous experience, how much memory is needed - a practice which is not all that practical.

Although both efficiency and accuracy are important, the program's emphasis is on accuracy. Thus all floating-point computations in TARDIS are done in double precision to ensure maximum accuracy, although the speed may be lower than the speed of single-precision computations on some computers. With C, TARDIS has ability to do bitwise operations directly. Moreover, if there are floating-point operations that can be done by using bitwise manipulations, TARDIS will use the bitwise version to improve the speed. TARDIS also avoids using indices to access successive elements in arrays or matrices. Whenever possible, pointers to the elements in arrays are used instead to increase speed. Registers are also used for often-used variables to improve the speed a bit further. Nevertheless, it has been noticed that such implementations resulted in only a slight improvement in speed of about 1% of overall floating-point operations. So the major factor used to ensure efficiency and accuracy in the simulation is still a careful implementation of the numerical algorithms.

The numerical algorithms used in TARDIS have been tested before being incorporated into the program to ensure that the resulting performance is comparable to or better than that of other existing simulation packages. Although not all the potential in TARDIS has been exploited, the results of the experiments show very convincingly that TARDIS can handle the simulation of most drive systems efficiently and accurately.

1.4. Report Outline

Following this introduction, Chapter 2 presents an overview of TARDIS. This chapter discusses the translators for TARDIS, the black box concept for component modules, equation formulations from electrical networks, and the description of state machines for handling discrete components.

Chapter 3 discusses the numerical integration method used in TARDIS, more specifically the variable-step, variable-order Gear algorithm with local truncation error scheme to control the step size of the integration. The rest of the chapter is devoted to how TARDIS locates discontinuities. Three examples are also given to illustrate how accurate TARDIS is in locating the discontinuities.

Chapter 4 describes the data structures and solution techniques used in TARDIS to handle sparse matrices. TARDIS uses sparse matrix techniques to reduce the computation involved in the Jacobian equation arising from the Newton-Raphson algorithm, which in turn is used to solve algebraic equations resulting from the Gear algorithm.

Chapter 5 gives several examples demonstrating the use of TARDIS. Some of the examples in this chapter are purposely selected from the past work of others to validate the capabilities of TARDIS. Representations of switches by high-and-low resistance and ideal switches are also discussed.

Chapter 6 summarizes the main contributions of the research and also discusses useful features that could be added to the current version of TARDIS.

Appendix A provides a brief explanation of the basics of Newton's divided difference.

Appendix B is a listing of the GetZero() routine which is modified from the idea of Brent's zeroin() routine for locating a zero of a function [21, 22]. For smooth functions, GetZero() uses the same number of iterations as Brent's

zeroin() does. However, for the worst case, GetZero() uses the number of iterations in the order of $O(\log_2 n)$ while zeroin() uses $O((\log_2 n)^2)$.

The rest of the appendices are the source codes for MainSystem() and MainEvent() routines which are used in sample circuits described in Chapter 5.

CHAPTER 2

OVERVIEW OF TARDIS

As is, TARDIS is a differential-algebraic equation solver that can handle discontinuities. It does not attach any physical meaning to the system equations; TARDIS treats them in a strictly mathematical sense. It is the user's responsibility to provide a correct mathematical representation of the system. One approach to making TARDIS more user-friendly is to have a translator acting as a user interface to TARDIS as illustrated in Fig. 2.1. The translator will then translate the user's input to necessary the necessary forms that can be compiled and linked with TARDIS's numerical routines. Although the user will then be required to learn the language used by the translator, this is usually preferable to writing the mathematical models directly, even though the latter gives the user more control over the codes.

2.1. Simulation Language to be used with TARDIS

The proposed simulation language used by TARDIS is a network-oriented type. Since components in the system to be simulated are from a mixture of mechanical, electrical, electronic and logic types, they pose some difficulty when connecting modules of different types together. To overcome the problem, Runge [23] proposes a simulation language called Modular Ordinary Differential Equation Language (MODEL) that integrates network modules from different types into a single framework. In MODEL, the user can define equations for modules in the form

$$\text{left hand side} \quad = \quad \text{right hand side} \tag{2.1}$$

Variables used within a module are referred to as local variables, and those which are shared among modules, global variables. The variables at the

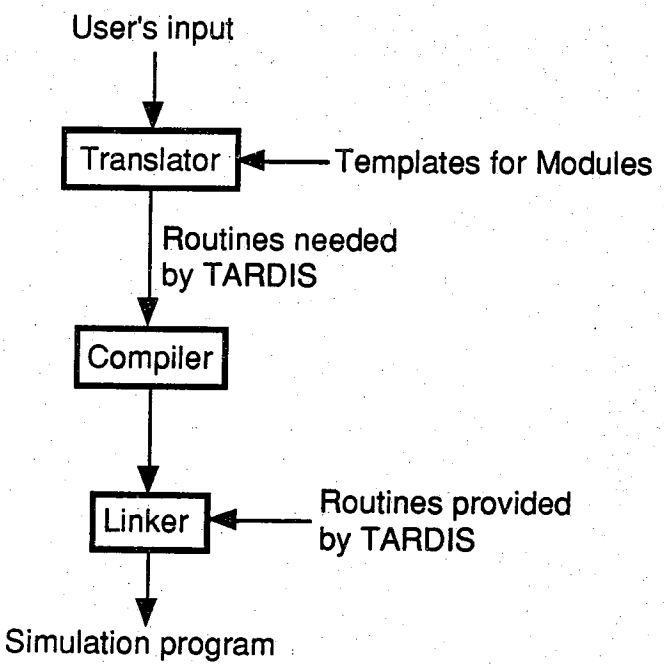


Figure. 2.1. TARDIS with a translator.

terminals of the modules (e.g., pins of integrated circuits, input and output pins of the modules) are terminal variables. A node is a common point to which the terminals of modules are connected. If the terminal variables connected to the same node are to take on the same value, the variables are called E-type terminal variables. If the sum of the terminal variables connected to the same node is to be zero, these terminal variables are called I-type variables. The equations describing a module can refer to local, terminal, and global variables, and even user-defined FORTRAN subroutines. Discontinuities are also allowed in MODEL; they are handled by the use of IF statements.

TARDIS uses a different approach to combine modules from several disciplines. (Although the translator for TARDIS has not been completed at this point, some of the ideas have been implemented in the numerical part.) TARDIS treats each module as a black box with pins or terminals to connect to the outside. The user can define modules with equations of the form

$$\mathbf{0} = \mathbf{f}(\mathbf{y}, t) \tag{2.2}$$

Associated with pins of the modules are pin variables. Similar to MODEL, the user can use pin, local, and global variables, user-defined or other predefined subprograms in these equations. Unlike MODEL, there are three types of variables used in TARDIS: electrical, non-electrical, and logic variables corresponding to electrical, non-electrical, and logic pins. Distinguishing these three variable types facilitates the simulation tasks. There are two variables associated with each electrical pin, namely, voltage and current. There is no convention imposed on the direction of the current - whether into or out of the modules. The user can choose either direction, but must be consistent throughout. For the purposes of discussion in this report, the direction of the pin currents is assumed to be into the modules.

The nodes where the pins are connected are also classified the same way: electrical, non-electrical, and logic nodes. The variables of the pins connected to the same node take on the value of the variable at the corresponding node. For electrical nodes, Kirchhoff's current law (KCL) must

also be satisfied. Usually a node has pins of the same type, but certain mixed-type connections are allowed to reduce the number of unknown variables. For example, the non-electrical pins can be connected to an electrical node; these pins are not included in the KCL equation for that node, and KCL still applies to all currents of electrical pins connected to that electrical node. The variables of the non-electrical pins connected to electrical nodes will take the value of the corresponding node voltages.

A module can have pins of different types. For example, a motor may have two electrical pins to be connected to a power supply and two non-electrical pins for torque and speed. A thyristor may have two electrical pins and one logic pin.

There are differences among the three types of variables. The electrical and non-electrical variables are unknown variables in the system equations to be solved. The logic variables are not part of the unknowns. The logic variables are to be used by a module to communicate its status to other modules. The electrical and non-electrical are represented by double-precision variables, but the size of logic variables is user-defined and may have different sizes as well. All three types of variables are global variables; they can be used inside any module or in the output routine.

Besides the above-mentioned variables, there are other double-precision global variables also: intermediate variables and parameter variables. An array of intermediate variables is used to store meaningful intermediate values that may be used by other modules, other subprograms, or the output routine. The intermediate variables are also used to avoid repeated calculations. The array of parameter variables is intended for the variables that are constant or changed occasionally. They may be used in various modules, some subprograms, or the output routine. Although the use of variables in these two arrays may be interchangeable, it is not advisable because they can make the debugging of the program difficult. Both these arrays are not part of the unknown variables to be solved.

2.2. Equation Formulations from Electrical Circuits

The formulation of equations from electrical circuits in TARDIS is not restricted to a specific method. The method used can be tableau formulation, modified nodal formulation, or even a combination of the two. For example, the equation formulation for a resistor may be done by modified nodal formulation while the equation formulation for a capacitor may be done by tableau formulation because the branch voltage of the capacitor is a state variable and is needed in the equation. Other types of formulations are also allowed, but they are less popular than these two. Note that one objective in TARDIS is to make all variables available at all times so the user may refer to them anywhere in the user-defined modules or subroutines without having to worry whether such and such a variable is available or not. Because the tableau formulation has all variables available as unknowns to be solved, this may suggest that the tableau formulation is the best for TARDIS; this, however, is not necessarily the case. By the use of intermediate variables, the user can make all the variables available without increasing the number of unknowns. For example, the modified nodal formulation for a resistor with the conductance G between nodes i and j may be written as

$$\text{KCL at node i:} \qquad 0 = G\,V_i - G\,V_j + \sum \begin{pmatrix}\text{currents leaving}\\ \text{node i}\end{pmatrix} \tag{2.3}$$

$$\text{KCL at node j:} \qquad 0 = -G\,V_i + G\,V_j + \sum \begin{pmatrix}\text{currents leaving}\\ \text{node j}\end{pmatrix} \tag{2.4}$$

As one can see, the current through the resistor above is not defined explicitly. To define the current of the resistor, the usual way is to include the current as an unknown variable. The equations now can be rewritten as

$$\text{Constitutive eq.:} \qquad 0 = G\,V_i - G\,V_j - I \tag{2.5}$$

$$\text{KCL at node i:} \qquad 0 = I + \sum \begin{pmatrix}\text{currents leaving}\\ \text{node i}\end{pmatrix} \tag{2.6}$$

$$\text{KCL at node j:} \qquad 0 = -I + \sum \begin{pmatrix}\text{currents leaving} \\ \text{node j}\end{pmatrix} \tag{2.7}$$

where I is the current passing through the resistor. However, this method is not suggested since the number of unknowns is unnecessarily increased, resulting in more computations when many resistors are defined this way. By the use of intermediate variables, the equations above may be rewritten as follows:

$$I = G\,(V_i - V_j) \tag{2.8}$$

$$\text{KCL at node i:} \qquad 0 = I + \sum \begin{pmatrix}\text{currents leaving} \\ \text{node i}\end{pmatrix} \tag{2.9}$$

$$\text{KCL at node j:} \qquad 0 = -I + \sum \begin{pmatrix}\text{currents leaving} \\ \text{node j}\end{pmatrix} \tag{2.10}$$

where I is now an intermediate variable whose value is set by Eq. (2.8). Although the number of unknowns remains the same, the order of the equations is important in the execution of the solution. The Eq. (2.8) must be executed before Eq. (2.9) and (2.10) to obtain the correct result. By itself, TARDIS does not have the equation sorting capability of a program like ACSL. The user or the translator must do the sorting for the equations that set the values for intermediate variables. These equations - like Eq. (2.8) - should then be put at the beginning of the routine, MainSystem(), which contains all the system equations.

2.3. Handling of Components Associated with Discrete Events

The simplest way to handle components associated with discrete events may be by the use of IF-ELSE statements: for example, the pseudo code for an ideal diode connecting between nodes i and j may be written as follows:

```
if (diode current is positive)
{
    v(i) - v(j) = 0;
}
else if (voltage across diode is negative)
{
    diode current = 0;
}
```

Due to the numerical integration algorithm used in TARDIS, handling of components associated with discrete events by such simple mechanism may lead to some numerical problems - i.e., the program would be unable to find the solution or would use many calls to the routine containing system equations. The reason is, when TARDIS solves the system equations by an iterative method (as part of the integration), the conditions used in IF-ELSE statements as well as the corresponding terms or equations may change from one iteration to another, the change which may result in divergence of the iterative method. Even when the iterative algorithm does converge, there can be many calls to the routine containing system equations due to the local truncation error control scheme used in the integration algorithm implemented in TARDIS.

Bodry and Foch [24] propose a framework to handle components associated with discrete events, called Petri nets which are very powerful models to describe the flow of information [25]. Petri nets are also adopted in TARDIS because they provide simple conceptual models. However, the use of Petri nets in TARDIS represents only a small portion of their real potential, and thus the term "state machines" will be used instead of the term "Petri nets". With the use of state machines, the numerical problems associated with the above simple mechanism for handling components associated with discrete events can be avoided.

In TARDIS all components associated with discrete events must be represented by state machines. The status of a state machine is indicated by a state associated with that state machine. A state machine can change its

state when certain condition is satisfied. For example, a diode may have two states: ON and OFF. The diode will change its state from ON to OFF when its forward current reverses direction, and the diode will change its state from OFF to ON when its forward voltage drop is positive. For the rest of this report, such conditions will be called transitions instead. Figure 2.2. illustrates a state machine for some discrete device. The numbers in circles indicate the status of the state machine. The paths with arrows from one circle to itself or the other circles indicate the possibilities of the next states from the current state. The bars on the paths are the transitions which must be satisfied (or fired) for the paths to be used. The transitions connected to the current state are termed active, even though they may not be fired. If there is more than one fired transition, the one with highest priority assigned by the user will be chosen. From Fig. 2.2, if the current state is 1, only T1 is an active transition. If the current state is 2, there are three active transitions, T2, T3, and T4; only one of these three transitions will be fired. Note that if the current state is 3 and T6 is fired, the next state will still be 3. If state 4 is reached, the state machine will be in this state forever since there is no other path to go.

The program codes for all state machines will be in a separate routine called MainEvent(), for purpose of efficiency. When the MainSystem() is executed, no status change in the state machines is allowed, the program codes for the state machines will not be executed. Only when the program needs the information on status changes in the state machines will the program codes for the state machines in the MainEvent() routine be executed. Communication between the two routines, MainSystem() and MainEvent(), is done through global variables and variables common to these two routines. The separation of the two routines makes TARDIS different from several other simulation packages which allow the descriptions of the discrete components and the system equations to be mixed. Although the separation of the two routines helps avoid redundant calculations, the casual user may have difficulty writing the code to describe the behavior of discrete events. This problem, however, will be alleviated with the use of a translator.

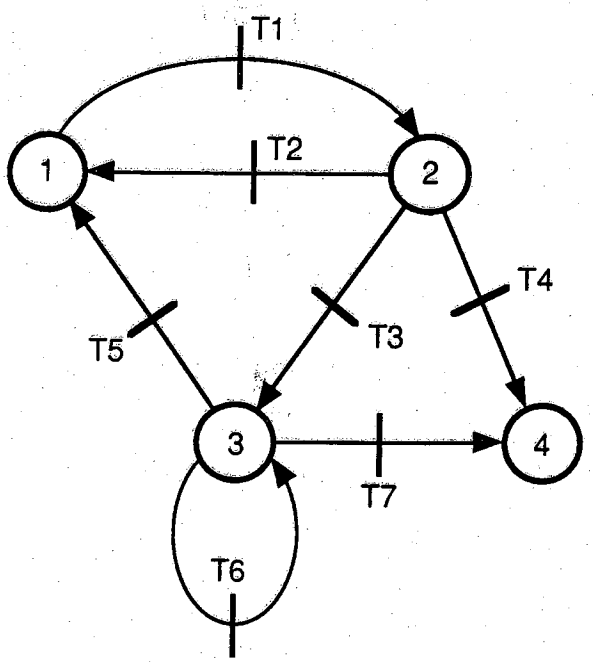


Figure 2.2. A sample state machine.

Several state machines can change their states at the same time. However, the program executions are done serially - one instruction after another. The serial operations cannot guarantee the correct status of those state machines whose transitions are dependent on the status of other state machines. It is impossible to use sorting procedure to order the codes for the state machines because the information for sorting cannot be extracted from the semantics of the codes. What is done in TARDIS to ensure the correct status of the state machines is to delay updating the status until the program codes for all state machines in MainEvent() are executed. Thus going from the current state to the next state requires a procedure rather than an assignment statement. This is also applied to logic variables; they are not allowed to take on the new values until all the information about the new values is received. In this manner, when a state machine whose transitions depend on the status of other state machines or logic variables is encountered, TARDIS ensures that the resulting status of the state machine and the values of logic variables will be correct.

There are two types of discrete events allowed in TARDIS: scheduled and conditional events. Scheduled events are events whose instants of occurrences are known in advance - e.g., sampling time that occurs periodically, thyristors' firing time that is synchronized with known voltage or current sources, etc. Conditional events are events whose instants of occurrences are not absolute but instead dependent on expressions of state variables, non-state variables, and time. These expressions can be used in the transitions of the state machines. The adopted convention in TARDIS is that when the expressions of active transitions are negative, these transitions will be fired. If possible, TARDIS will fire a transition when the values of expressions of those transitions are within certain negative bands specified by the user.

Typically in the codes for state machines, the user may make use of SWITCH statements in C to simulate the behavior of the state machines, and, if needed, IF-ELSE statements to assign priority to the transitions. For example, the pseudo code for a state machine of a diode can be as follows:

```
switch (diode's State)
{
case ON:
        if (diode's current is less than zero)
        {
                next diode's state will be OFF;
                diode's resistance is 1.e6 Ω;
                set a flag to reevaluate the Jacobian due to abrupt
                        change in diode's resistance;
        }
        break;
case OFF:
        if (voltage drop across diode is positive)
        {
                next diode's state will be ON;
                diode's resistance is 1.e-3 Ω;
                set a flag to reevaluate the Jacobian due to abrupt
                        change in diode's resistance;
        }
        break;
case STATE_INIT:
        next diode's state will be OFF;
        diode's resistance is 1.e6 Ω;
}
```

At the beginning of the simulation, all state machines will be set to the same initial state, STATE_INIT. The user can then change these states to any desired states. In the above case, the initial state of the diode is reassigned to the OFF state with a corresponding value of the diode resistance. Due to the numerical integration algorithm used in TARDIS, the Jacobian of system equations is needed. Whenever there is an abrupt change in the parameters of system equations, the user must set a flag calling for the Jacobian to be reevaluated. There is no need to set a flag at the beginning because the Jacobian will be calculated by default.

The current version of TARDIS calculates the Jacobian by taking a numerical difference. As this method of finding the Jacobian is time consuming, especially when the system has many discontinuities, the next version of TARDIS will be more selective in choosing the equations to be updated, or will let the user update the entries in the Jacobian directly.

2.4. Error Control Parameters in Simulation

Error control parameters are key input parameters that the user can specify to ensure that the simulation results are within the desired degree of accuracy. When the dynamic range of the values of the unknown variables is large - e.g., 5 mA for typical values of currents in control circuits and 1000 A for the motor currents - the same error control parameters should not be used for all variables. Although most of the application-specific simulation programs allow the user to specify error control parameters in the integration or in the iteration process, the error specification is not done on an individual basis. Faced with such a limitation, the user may resort to scaling the values of all variables to the same order of magnitude. However, in TARDIS, the user has the choice of specifying the error control parameters for each individual variable. The specification of error control parameters should be done when the modules are defined.

CHAPTER 3
NUMERICAL INTEGRATION WITH DISCONTINUITIES

3.1. Introduction to Differential-Algebraic Equations

The differential-algebraic equations (DAE) being considered can be written in the following form:

$$\mathbf{0} = \mathbf{f}(\mathbf{y}_{nst}, \mathbf{y}_{st}, \mathbf{y}'_{st}, t) \tag{3.1}$$

where $\mathbf{y}_{nst}$ are non-state variables, $\mathbf{y}_{st}$ and $\mathbf{y}'_{st}$ are state variables and their derivatives, respectively.

In general DAE's may not be equivalent to ordinary differential equations (ODE) [26]. DAE's are classified by an index system. Not all types of DAE's are numerically solvable by existing numerical algorithms. DAE's with lower index numbers can be solved, but that is not the case for higher index equations. DAE's with index 0 are actually ODE's. DAE's with index 1 are ODE's with some algebraic equations which can be symbolically or numerically reducible to ODE's. For the DAE's with an index greater than or equal to 2, one will have to do differentiation - instead of the usual integration - to get to the solution. Loosely speaking, if one has to find the n^{th} order derivative of the input to get the solution, the index number of that DAE system will be n+1, for $n \geq 1$. A more precise definition of the index of DAE's can be found in [26, 27]. Usually, systems of indices greater than 1 are those having the ability to change their state variables arbitrarily. For example, Fig. 3.1 illustrates an index-2 system whose equations are below.

$$I_L - I_S = 0 \tag{3.2}$$

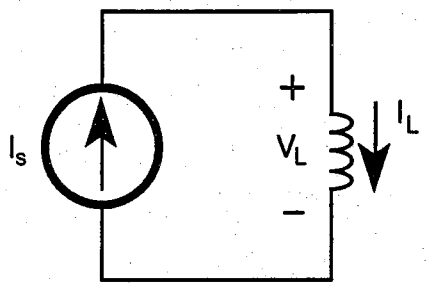


Figure 3.1. Example system with index 2.

$$V_L - L\frac{dI_L}{dt} = 0 \qquad (3.3)$$

As one can see from the equations, in order to find the voltage across the inductor, one needs to differentiate the current which is determined by the current source. There are some software package, like DASSL that attempt to handle DAE's with indices higher than 1, but for this research, the index of DAE systems will be limited to 0 and 1 only. In other words, the ability to change the values of state variables arbitrarily will not be allowed.

3.2. System Stiffness

Loosely speaking, stiffness usually means having fast and slow transients together in the system. Actually, the stiffness does not depend on the system characteristics alone but also on the initial conditions, accuracy requirement [28], and duration of integration. Since the stiffness of the system to be simulated may not be known in advance, it is prudent to use an integration algorithm suitable for stiff systems just in case that the system is stiff. It is possible to implement two integration algorithms - one for stiff systems and the other for non-stiff systems - together in the same program, as is done in many existing software. In that case, the user may decide which one to choose, or the selection of algorithms can be made automatic according to

how the system responds. In TARDIS, only the integration algorithms for stiff systems will be considered for the time being.

3.3. Gear Backward Differentiation Formulae

The implicit integration algorithms can generally handle stiff systems better than the explicit ones [11]. There are several implicit algorithms, among which the ones most commonly used are the trapezoidal, Adams, and Gear algorithms [29, 30]. The Adams algorithm is usually used for smooth or non-stiff systems. Both the trapezoidal and the Gear algorithms can be found implemented in several simulation programs for electrical or electronic circuits. Of the three, the Gear algorithm is the best implicit algorithm for handling stiff systems.

TARDIS uses a variable-step, variable-order Gear algorithm implemented by using Newton's divided difference as described in [31]. The implementation is also repeated here with a slight change in notation to avoid nested subscripts. The following description is for the state variable part and it assumes that we have a single first-order implicit differential equation only. So the following y will replace $\mathbf{y}_{st}$. If there is more than one state variables, the following discussion will apply to every one of them.

Let y_i, y_{i-1}, y_{i-2}, ..., y_{i-k} be k+1 solution values of a differential equation

$$0 = f(y, y', t) \tag{3.4}$$

at t_i, t_{i-1}, t_{i-2}, ..., t_{i-k} respectively. To find y_{i+1} at t_{i+1}, the algorithm replaces y′ of Eq. (3.4) with

$$y'_{i+1} = \beta\, y_{i+1} + S \tag{3.5}$$

where β depends on t_i, t_{i-1}, t_{i-2}, ..., t_{i-k} and S depends on y_i, y_{i-1}, y_{i-2}, ..., y_{i-k} and t_i, t_{i-1}, t_{i-2}, ..., t_{i-k}, respectively. The resulting Eq. (3.6) after the

substitution is algebraic which will be solved by the Newton-Raphson algorithm (NR).

$$0 = g(y_{i+1}, t_{i+1}) \tag{3.6}$$

The NR algorithm will be described in detail along with sparse matrix techniques in the next chapter.

If there are non-state variables, the Eq. (3.6) will become a system of algebraic equations with both non-state and state variables as unknowns. In the course of calculating β and S, the predictor y^p_{i+1} which is the polynomially extrapolated value of the previous k+1 solutions at t_{i+1} will be found also. This y^p_{i+1} will be used as an initial guess to the NR method which should make the iterations converge faster. The steps in finding β, S, and y^p_{i+1} are as follows:

1. Find the divided differences $y[t_i, t_{i-1}, ..., t_{i-j}]$ for j = 1, 2, 3, ..., k. (The divided differences can be found in Appendix A.)

2. Calculate the coefficients α_j, β_j, and L_j as follows:

$$L_j = t_{i+1} - t_{i-j+1} \qquad \text{for } j = 1, 2, ..., k$$
$$\alpha_1 = L_1,$$
$$\alpha_j = \alpha_{j-1} L_j \qquad \text{for } j = 2, 3, 4, ..., k$$
$$\beta_1 = \frac{1}{L_1},$$
$$\beta_j = \beta_{j-1} + \frac{1}{L_j} \qquad \text{for } j = 2, 3, 4, ..., k$$

3. Find S and y^p_{i+1} by

$$y^{p_0}_{i+1} = y(t_i)$$
$$y^{p_j}_{i+1} = y^{p_{j-1}}_{i+1} + \alpha_j\, y[t_i, t_{i-1}, \ldots, t_{i-j}] \qquad \text{for } j = 1, 2, 3, ..., k$$

$$S_1 = \frac{y_{i+1}^{p_0}}{L_1}$$

$$S_j = S_{j-1} + \frac{y_{i+1}^{p_{j-1}}}{L_j} \qquad \text{for } j = 2, 3, 4, \ldots, k-1.$$

Fig. 3.2 illustrates how some variables are defined in the implementation of Gear algorithm.

The order of integration k can be as high as 6, which still be a stiffly stable algorithm [11]. When the order is higher, the step size tends to be larger also, and this usually results in fewer steps in integration, but more overhead computation. So some software packages that use this algorithm - such as DASSL [20], LSODAR [32], and IVPAG from IMSL [18] - limit the integration order to 5. These programs, except DASSL, also incorporate the Adams algorithm with order of up to 12 for smooth systems. However, in TARDIS, an arbitrary limit of integration order of 6 is set since it does not incorporate the Adams method. This is done so that, for smooth systems the integration of order 6 can reduce some computations by using larger time steps than those obtained when the order is limited to 5.

y_{i+1}^{p} is used not only to start the NR method but also to estimate the local truncation error (LTE), a measure on which the adjustments of integration order and step size of the integration are based. There are various formulae for calculating the current local truncation error (LTE_{curr}). Brayton (reference from Vlach and Singhal in [33]) compared them and concluded that the correct formula should be

$$LTE_{curr} = \frac{\left| y_{i+1} - y_{i+1}^{p} \right|}{\beta \, L_{k+1}} \,. \qquad (3.7)$$

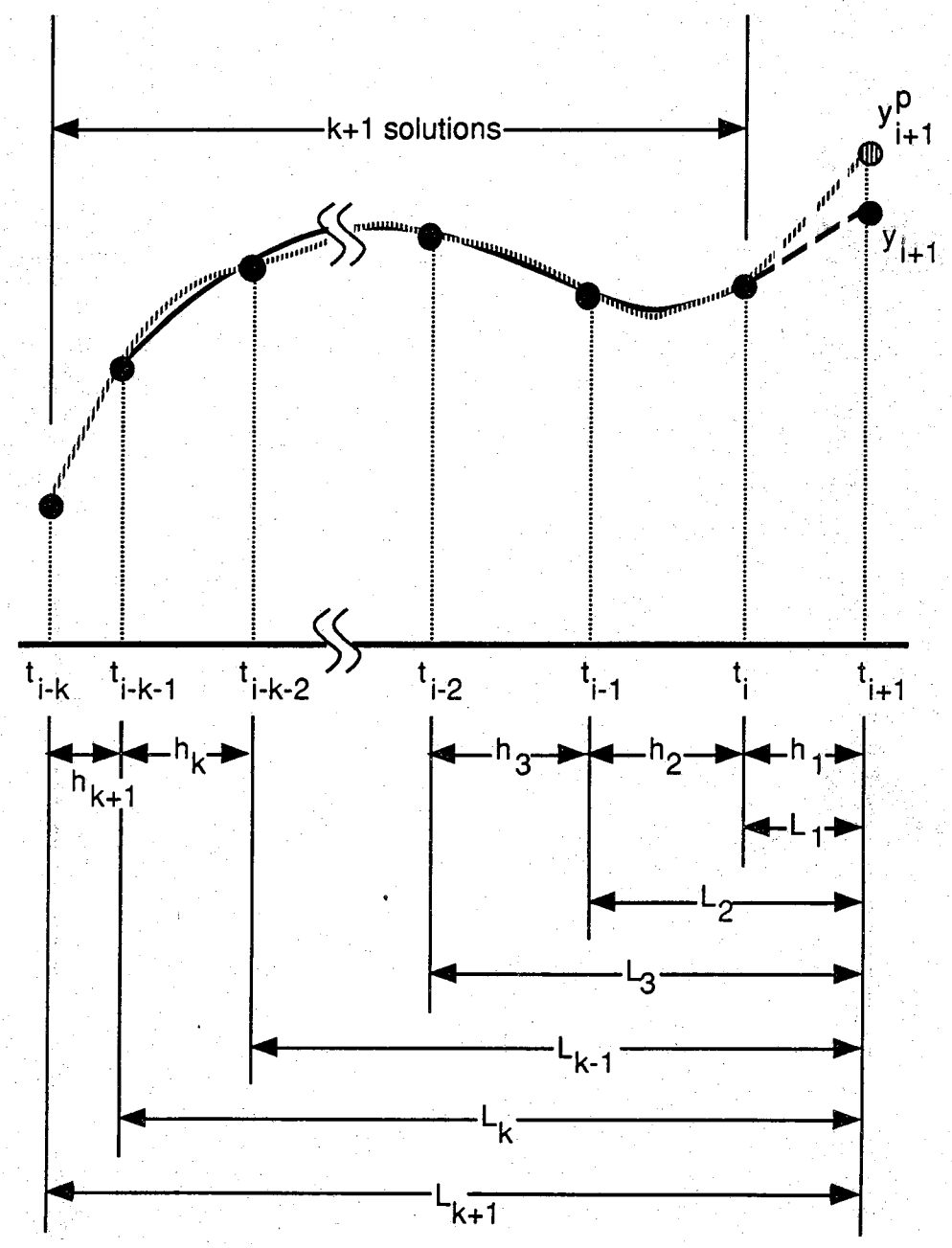


Figure 3.2. Illustration of Gear algorithm.

The above formula is also used by Zein, et al. [34]. To use this information to adjust the step size of integration, one may compare this LTE_{curr} to the allowable local truncation error (LTE_{allow}) at this step. Brayton [35], Van Bokhoven[31], and Vlach and Singhal [16] calculate this LTE_{allow} from the specified truncation error per unit time κ, which is a ratio of global truncation error to duration of simulation, as

$$LTE_{allow} = \kappa\, h_{allow} \tag{3.8}$$

where h_{allow} is the allowable step size.

Since the local truncation error of the k^{th} order integration is proportional to h^{k+1} - i.e.,

$$\frac{LTE_{allow}}{LTE_{curr}} = \left(\frac{h_{allow}}{h_{curr}}\right)^{k+1}, \tag{3.9}$$

the allowable step size can be found by

$$h_{allow} = h_{curr}\left(\frac{\kappa\, h_{curr}}{LTE_{curr}}\right)^{\frac{1}{k}}. \tag{3.10}$$

This h_{allow} will be used as the next step size. If there is more than one state variables, the value of h_{allow} will be calculated for each of them. Then the next step size will be taken from the smallest values of h_{allow}:-

$$h_{next} = \min\left(h_{allow}\right) \tag{3.11}$$

To adjust the order of integration, one also uses the Eq. (3.7) to calculate LTE of different orders by using y^{p}_{i+1}, β, and L_k of different orders: - usually one order lower and one order higher. Then the order of integration will be decided upon the maximum of h_{allow} of these three different orders.

However, the use of the truncation error per unit time may sometimes cause the step size calculated from the Eq. (3.10) to be very small, especially when simulating stiff systems or running the simulation for a long duration. Thus, in TARDIS, the LTE_{allow} will not be calculated from the Eq. (3.8).

Most of the software packages, such as IVPAG from IMSL, ACSL [5], ODEPACK [36], and DASSL[20], will try to control LTE by letting the user specify some kind of error tolerance and the routine will use that information to calculate the anticipated LTE_{allow}. One approach is as follow:

$$LTE_{allow} = \varepsilon_{rel} |y_{i+1}| + \varepsilon_{abs} \tag{3.12}$$

where ε_{rel} and ε_{abs} are relative and absolute error tolerance specified by the user respectively. Using the Eq. (3.10) to calculate LTE_{allow} will adjust the step size according to the responses of the system on a local basis - not on a global basis as in the scheme used before. This scheme makes the integration step size adapt to the system response better and still gives acceptable accuracy for both fast and slow responses. For smooth systems, the Eq. (3.9) and (3.10) will give comparable accuracy if both schemes use a comparable number of steps in integration.

By using the Eq. (3.10), the allowable step size can be calculated from

$$h_{allow} = h_{curr} \left(\frac{\varepsilon_{rel} |y_{i+1}| + \varepsilon_{abs}}{LTE_{curr}} \right)^{\frac{1}{k+1}} \tag{3.13}$$

When starting the integration there is no information about previous values so one cannot use the Eq. (3.7) to find out the LTE at the first point. What is usually done is to take the first step, after the discontinuities or from the beginning of simulation, small enough so that the LTE should be acceptable. For example, IVPAG of IMSL arrives at the default initial step size by dividing the output interval by a factor of 1000. However, this is not good enough

when dealing with discontinuities because the instants of their occurrences are not known in advance, and whatever first step size is chosen may be too big to locate the discontinuities accurately. So the LTE at the initial point is needed to be able to adjust the initial step size accordingly. It is proposed that the initial LTE can be calculated from Eq. (3.7) with y^p_{i+1} derived from the forward Euler formula; that is

$$y^p_{i+1} = y_i + L_1 y'_i \tag{3.14}$$

Note that using the LTE information to adjust the first step size will defeat the purpose of relying on the stiff method implemented in the program, because the stiff method is not employed to skip very fast transients. Note that both the Adams and Gear algorithms start out with the first-order backward Euler formula. It would be an interesting subject for a comparative study to see which algorithm is more efficient and accurate.

Using the LTE information to change the order, as mentioned earlier, may sometimes lead to spurious change of order. To reduce such spurious order changes, the order is reduced only if

$$h_{allow(k-1)} > h_{allow(k)} > h_{allow(k+1)}.$$

Similarly, the order of integration will be increased only if

$$h_{allow(k-1)} < h_{allow(k)} < h_{allow(k+1)}.$$

These conditions are similar to the ones used in DASSL by Petzold [20].

In TARDIS, if h_{allow} is less than .75 h_{curr}, then the current step of integration will be repeated, since this may indicate that the current step size could give an unacceptable LTE. The factor .75 is chosen to prevent the algorithm from hunting - that is if the LTE after changing the step size were still too big, the integration would otherwise have to be repeated several times. To further

ameliorate this problem, one can reduce the step size h_{allow} even further. The ability to adjust the current step size - not just the next step size - ensures that the location of discontinuities can be determined accurately.

Since LTE calculation is just an approximation, it may not be sufficiently reliable for the purpose at hand. One may want to make the calculation of the next time step more conservative by reducing it by some means; for example, by multiplying the right-hand-side of Eq. (3.10) by some factor less than 1. Even then h_{allow} may be too big compared to h_{curr}. If the next step size is too big, it is very likely that the next step will not pass the monitoring of LTE, and the next integration will have to be repeated. To minimize the likelihood of this happening, one may a priori set the limit of the ratio $\frac{h_{allow}}{h_{curr}}$ to some number. If the ratio is higher than the limit, h_{allow} will take the value determined by this limit. Choosing too large a value of the limit will not solve the problem, but if the limit is too small, the integration will take more steps than necessary.

Nevertheless, it has been noticed that the speed of the implemented algorithm is slower than that of IVPAG of IMSL due to the fact that TARDIS typically uses two or more iterations per time step in NR algorithm while IVPAG usually uses one iteration per time step. This observation provides an incentive to find a better LTE control scheme.

Let us consider how one can obtain one iteration per time step on the average. One way to achieve this is to find values of h_{allow} small enough to guarantee that the predicted value of state variables are close to the corrector values within the tolerance of the NR algorithm. However, many combinations of parameters in the LTE control scheme as described above had been tried but were not successful in reducing the number of iterations per time step in sample tests. The experimentation finally leads to another LTE control scheme, in which:

$$h_{allow} = \left(.75 \frac{\text{Error Tol. for NR algo.}}{\left| y_{i+1} - y^{p}_{i+1} \right|} \right)^{\frac{.7}{k+1}} \tag{3.15}$$

where

$$\text{Error Tol. for NR algo.} = \varepsilon_{rel} \left| y_{i+1} \right| + \varepsilon_{abs} \tag{3.16}$$

and ε_{rel} and ε_{abs} are relative and absolute error tolerance, respectively, for stopping criterion of NR algorithm. Note that the new scheme uses the same ε_{rel} and ε_{abs} for error control in integration and for the stopping criterion of the NR algorithm. The factor of .75 is used to reduce the ratio of the expected error in the NR algorithm to the correction made to the predictors at the current step. The factor of .7 in the exponent is used to account for fact that the Eq. (3.14) is a very roughly estimate of the relationship between local truncation error and step size. The two factors given are by no means optimal but they have so far given satisfactory results in all the test problems.

A relevant question at this point is whether finding the new step size from Eq. (3.14) is reliable or not. The answer seems to be yes since the equation tends to select a step size smaller than the one calculated by the previous LTE control scheme, and TARDIS is able to achieve one iteration per time step, when using the same error criterion for the NR algorithm. This way one can use each call to the routine containing the system equations to advance in time and stop the iteration process, while the previous LTE control will be likely to use one call to the routine to advance in time and another call to stop the iteration process. Moreover, by choosing appropriate parameters in Eq. (3.14), one can make the new step size not too conservative also.

It has been observed that there are spurious changes of order of integration even with the Eq. (3.14) implemented. Gear [37] suggests using some factors to multiply the step size before selecting the order of integration. So the following implementation has been added to the LTE control.

The order will be reduced if

$$\frac{h_{allow(k-1)}}{1.1} > h_{allow(k)} > \frac{h_{allow(k+1)}}{1.21} ,$$

and the order will be increased only if

$$\frac{h_{allow(k+1)}}{1.095} < h_{allow(k)} < \frac{h_{allow(k+1)}}{1.2} .$$

Again, the 4 factors above are not guaranteed to be optimal in all cases but they seem to work just fine with the sample tests. With the Eq. (3.14) and the implementation above to reduce the spurious order change, the ratio of number of iterations to total steps of TARDIS is roughly 1.5 on the average. For most cases, the speed up is double when compared to the previous LTE control scheme in Eq. (3.13).

It is also observed that the LTE control parameters ε_{rel} and ε_{abs} are not quite reliable. For example, when these parameters are reduced, one would expect less error and more iterations or time steps needed. When the parameters are increased, the reverse should happen. Test results on TARDIS do not fully support such reasoning. When the two parameters are slightly reduced, one may not obtain higher accuracy even though TARDIS uses more calls. At other times, TARDIS may give higher accuracy with fewer calls when the parameters are slightly reduced. This behavior happens with both LTE control schemes and should deserve further investigation. It may be of interest to note that IMSL's IVPAG uses one parameter to control the accuracy of the solutions, instead of the usual two parameters used in Eq. (3.12).

3.4. Comparison between Gear and Trapezoidal Algorithms

Let us compare the Gear method implemented in TARDIS and the trapezoidal algorithm, which is one of the most widely used integration

algorithms for a number of simulation packages, on the following 2^{nd} order system

$$y'_1 = -y_2 \tag{3.17}$$

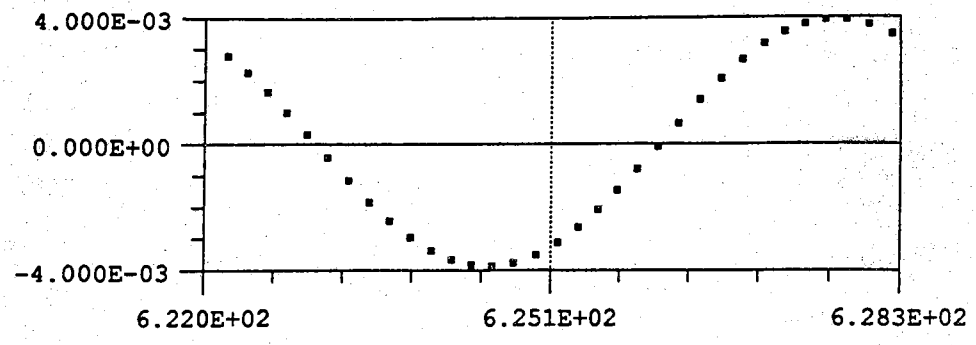
$$y'_2 = y_1 \tag{3.18}$$

with the initial condition $y_1(0) = 1$ and $y_2(0) = 0$. The exact solutions are

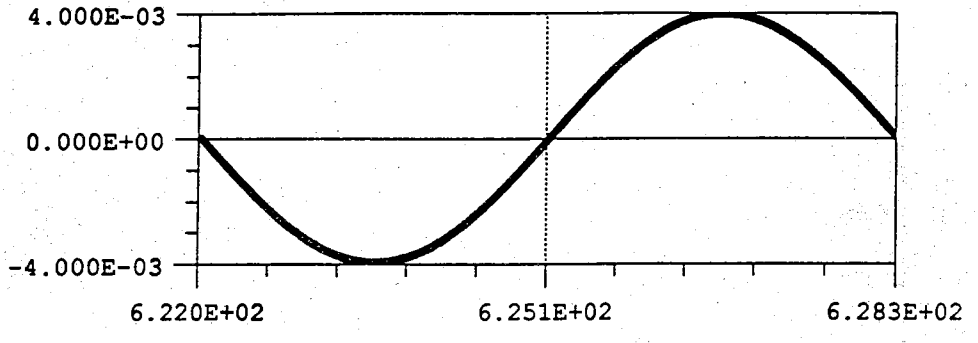
$$y_1 = \cos(t) \tag{3.19}$$

$$y_2 = \sin(t) \tag{3.20}$$

Let us integrate this system using both algorithms for 100 cycles and adjust the step size of the trapezoidal algorithm or the error tolerances of the Gear algorithm to give the same magnitude of global error of y_1 at 100^{th} cycle (compared with the exact solution of y_1) to compare the computation requirements of both algorithms. For the trapezoidal algorithm, let us use equal step size equivalent to .5 degree, and for the Gear method, let us use a variable-step, variable-order method as described in the previous section with $\varepsilon_{rel} = \varepsilon_{abs} = .9e\text{-}7$ and a maximum order of 6. Figure 3.3 shows the errors of y_1 of both algorithms at the final cycle. Designed specifically for this smooth system, the program for the trapezoidal algorithm is written to be as efficient as possible - that is using the intermediate variables to store values that are used more than once. The trapezoidal algorithm uses 72,000 steps while the Gear algorithm gives 3,423 output points and 3,431 calls to the routine containing the above differential equations. The total simulation times on Mac IIcx are 11 s and 13 s for the trapezoidal and Gear algorithms respectively. Although the trapezoidal algorithm uses many more steps than the Gear algorithm, the trapezoidal algorithm is faster than the Gear algorithm in this case. However, the execution time cannot be used as an indication that the trapezoidal method is more efficient than the Gear method, since the trapezoidal program is being optimized for this specific case. Nevertheless, from this test one may conclude that, to achieve the same accuracy, using the trapezoidal algorithm will require many more



a) TARDIS (34 points)



b) Trapezoidal method (720 points)

Figure 3.3. Errors of y_1 of 2^{nd} order system.

a) TARDIS (34 points)

b) Trapezoidal method (720 points)

output points and thus many more calls to the routine containing system equations than those of the Gear method.

For stiff systems, the comparison of TARDIS and equal step-size trapezoidal algorithm will not be justified since the step size for the trapezoidal algorithm will be very small throughout the simulation to avoid numerical oscillations. A variable step-size trapezoidal algorithm, such as the one used in SPICE2, will be needed for a fair comparison. However, the direct comparison between SPICE2 and TARDIS cannot be justified because there is overhead in SPICE2 to process the user's input which will be added to the execution time. Although, in designing SPICE2, Nagel points out that the trapezoidal method is more efficient than the Gear method, the implementations of Gear in SPICE2 and in TARDIS are very different, mainly in the local truncation error control scheme and the calculation of β and S in Eq. (3.5). This can be a subject for comparative study on the efficiency and accuracy of both algorithms.

3.5. Handling of State Machines

Before starting the simulation, all initial conditions of state variables and all states of state machines must be set. With these initial values of state variable and state machines, TARDIS solves the system equations for the values of non-state variables and the derivatives of state variables at the initial time. If TARDIS detects some fired transitions, TARDIS will fire those transitions, change the states of the state machines, and solve for the new values of non-state variables and the derivatives of state variables. This process is repeated until there is no more fired transition at the initial point. Figure 3.4 shows the flow chart of TARDIS at the beginning of the simulation.

After no fired transition is detected, the integration process begins. With the derivatives of state variables being replaced by the values obtained from Eq. (3.5), TARDIS will solve the system equations for the values of state and non-state variables. After a step is completed, TARDIS goes through all active transitions to see whether any of them can be fired. If none is detected, the

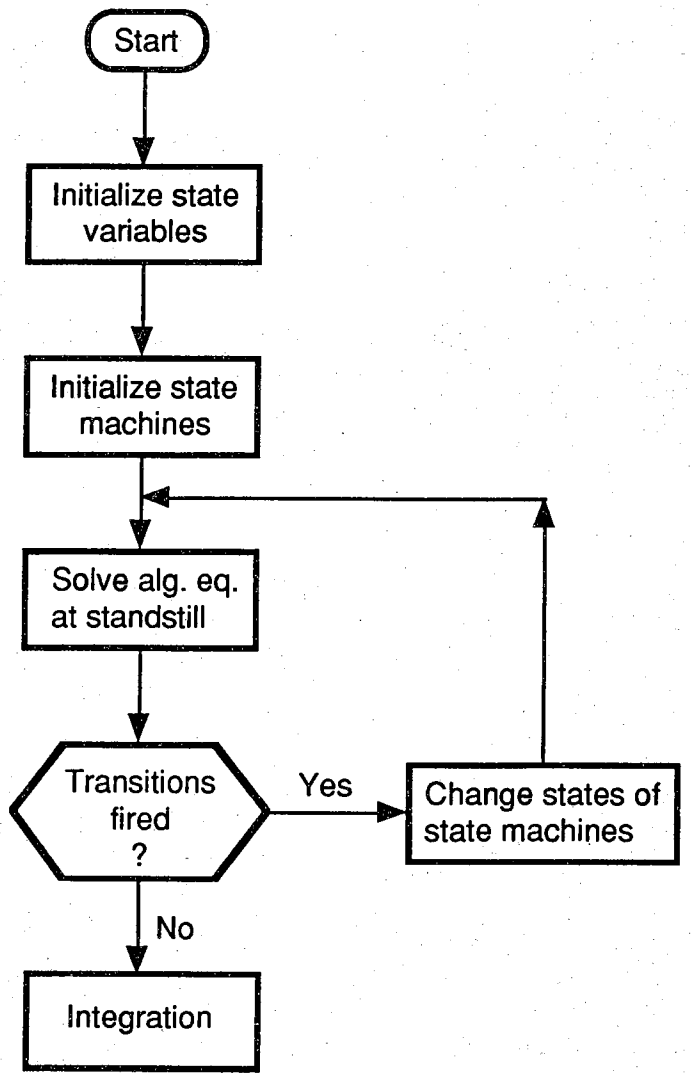


Figure 3.4. Flow chart at the start of the simulation.

next step of integration will be carried out. If the occurrences of some discrete events are detected within the current time step, TARDIS will treat the discrete events according to their types - scheduled or conditional.

For a scheduled event, the integration process will be carried out to the instant where the scheduled event occurs because the instant is known in advance. Then all active transitions will be checked to find out the fired transitions. If no fired transition is detected, the integration will continue. If there are fired transitions, the corresponding state machines will change their states. Then the system equations will be solved for the values of non-state variables and the derivatives of state variables. This process will be repeated until there is no more fired transition. Note that the state variables take on their current values; they are known variables at this instant.

For a conditional event, the current integration step will actually go beyond the instant where the event occurs since the instant is not known in advance. There are several methods proposed by various researchers to locate the instant of the conditional event, which will be mentioned briefly in the next section. After the instant of the event is found, repeated solving of the system equations and changing states of the state machines will be carried out until there is no fired transition. Then the integration will resume again.

Changes in the states of the state machines may cause changes in input signals to the system or in system parameters. Whenever a change that cannot be locally expressed by polynomials, such as an abrupt change, occurs, previous values of state variables should be discarded and the integration order should be reset to 1. This is because the integration algorithm used is based on interpolating polynomials. If the integration is allowed to be carried on without resetting the order, it is very likely that the next several integration steps will repeatedly fail, and the step size will eventually be reduced to a very small value before the integration process can continue. For abrupt changes in the system parameters, the Jacobian matrix used in the NR method must be updated accordingly also.

3.6. Locating Zeros of Switching Functions

There are several methods proposed by various researchers to handle discontinuities in the integration. Some of them try to handle the discontinuities by the integration routines directly, and others use interpolating polynomials to locate the discontinuities. Carver [38, 39] uses another set of differential equations which are derived from switching functions which are to be solved with the system differential equations and also uses inverse interpolation to locate discontinuities. This seems to add more computation to the simulation. Gear [9] uses the step size control to locate discontinuities. Ellison [40] and Birta, et al. [41] use the values of switching functions and their derivatives in interpolating polynomials to locate discontinuities with the Runge-Kutta integration algorithm.

There are also some integration routines that have a root-finding capability. One such routine is LSODAR [32, 36]. To use this package, the user has to specify switching functions in another routine. The locations of zeros are detected by the sign change of the specified functions. However, this is slightly different from what has been implemented in TARDIS due to TARDIS's greater complexity. Instead of detecting the sign change, TARDIS detects the negative values of expressions used for transitions. The normal values of these expressions are positive. Whenever they are negative, TARDIS will try to locate the zero-crossing points. Then the integration will be carried out to the minimum values of all zero-crossing points detected.

For locating the zero-crossing points, TARDIS uses the simple interpolating polynomials. The values of switching functions of active transitions will be stored in terms of divided differences for interpolation. The order of interpolation will be equal to or less than the integration order. If the order of integration is reset, the order of interpolating polynomials is also reset. At any time, the order of interpolating polynomials will be less than or equal to the integration order. Locating negative-going-zero-crossing is done by a routine modified from Brent's [22]. Brent's routine alternately uses the three methods depending on how the function whose zero is to be found behaves: linear interpolation, inverse quadratic interpolation, and binary search.

However, TARDIS uses inverse quadratic interpolation and binary search to locate the zero-crossing point, except at the beginning where the routine starts off with linear interpolation. The criterion for switching between the two methods is shown in Fig. 3.5. If the middle point, 2a, is in the area of the two inverse parabolas which have the slope at one end equal to infinity, the inverse quadratic will be used to find a zero. Otherwise, the bisection method will be used (point 2b in Fig. 3.5).

For smooth functions, the implemented routine, GetZero(), and Brent's routine perform similarly, but for the worst case, GetZero() will use $3 \log_2 \left| \frac{t_a - t_b}{\zeta} \right|$ function evaluations, where t_a and t_b are the end points, and ζ is a specified error tolerance, while Brent's routine will use $(\log_2 \left| \frac{t_a - t_b}{\zeta} \right|)^2$ function evaluations. The source code of the routine is listed in Appendix B. In the simulation, the function whose zero is to be found is usually smooth from one time step to another so the routine needs only 1-2 function calls to locate a zero-crossing point.

Locating the zero-crossing points using interpolating polynomials is not always reliable. Sometimes, it may predict a false zero-crossing point. This usually happens in a very short period of time: one or two time steps just before the real zero-crossing point occurs. Thus one cannot take the zero-crossing point predicted by the interpolating polynomial right away. TARDIS avoids this problem by reintegrating to that point to confirm whether that point is a real zero-crossing point. This, however, can make the step size very small. So the integration part of TARDIS is designed to expect this very small step size on such occasions.

To avoid numerical problems, some "band of certainty" which is suggested by Birta, et al [41] in locating discontinuities is also used. In TARDIS, this band must be on the negative side and will be called a negative band. As to how large this negative band ought to be, that depends on the switching function and how accurate the user wants it to be. As illustrated in Fig. 3.6, TARDIS will try to locate a zero-crossing point which gives a magnitude of the

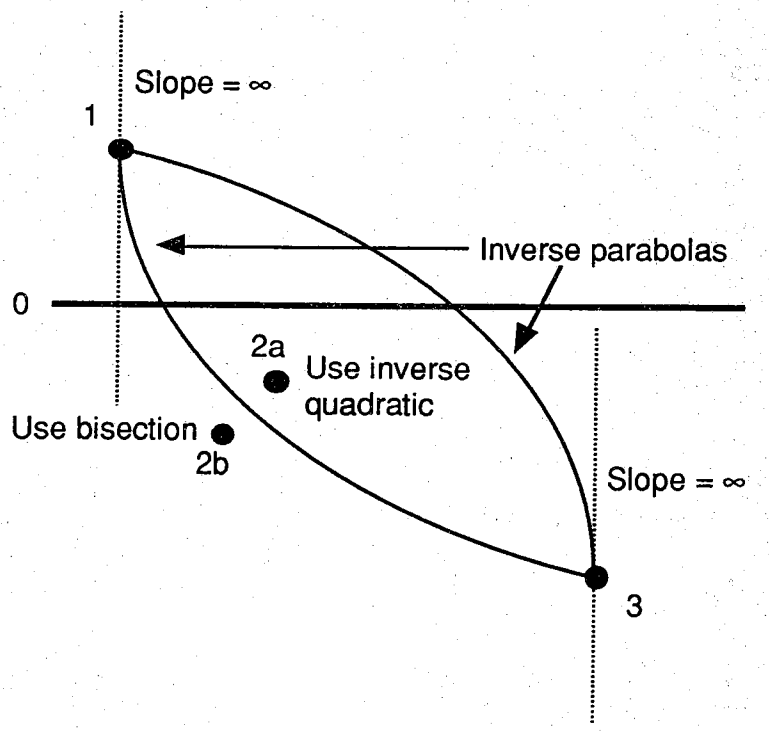


Figure 3.5. Criterion for switching methods in locating a zero in TARDIS.

switching function less than that of the band but as close to it as possible (down to the machine precision). The reason is that when there is more than one fired transition in the current step, the switching functions at the zero-crossing point will have the values within the band as shown by the gray line in Fig. 3.6.

For detecting short-lived discontinuities, such as the one shown in Fig. 3.7, the signs of the slopes at the end points are also used whenever such information is available. In TARDIS the slopes are approximated from the interpolating polynomials which are calculated and stored from one time step to another. When the slopes at the end points have different signs, TARDIS will first approximate the location of the minimum point using linear interpolation of slopes (see point 1 in Fig 3.7), and then use the interpolating polynomial to find out whether the value at that point is lower than the negative band or not. If it is, TARDIS will use the information at the minimum point and previous points to locate the zero-crossing point (see point 2 in Fig. 3.7).

3.7. Test Examples on Integration with Discontinuities

The following three tests are taken from Birta, et al. [41]. In [41] Birta, et al. use the Runge-Kutta (RK) formula for integration with local truncation error control; thus without further details, it is not possible to compare the efficiency of the two methods. The purpose of these examples here, therefore, is just to show that what has been implemented in TARDIS is reliable. For completeness, however, numbers of calls to the routines containing systems equations by both methods are listed. Note that all three test systems are not stiff, and if the new Jacobian matrix is needed, TARDIS will require n+1 calls to the routine that contains n system equations to approximate the Jacobian. This is a very expensive way to find the Jacobian, especially in medium and large systems. The current version of TARDIS requires calculation of the Jacobian at each discontinuity regardless of its type. This inefficiency will be stiff, and if the new Jacobian matrix is needed, TARDIS will require n+1 calls to the routine that contains n system equations to approximate the Jacobian.

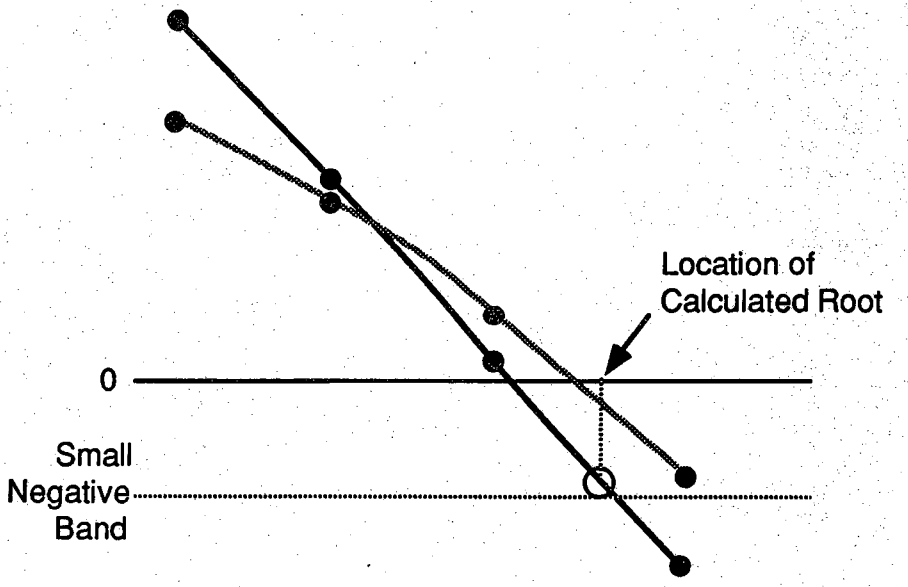


Figure 3.6. Locating a zero-crossing point.

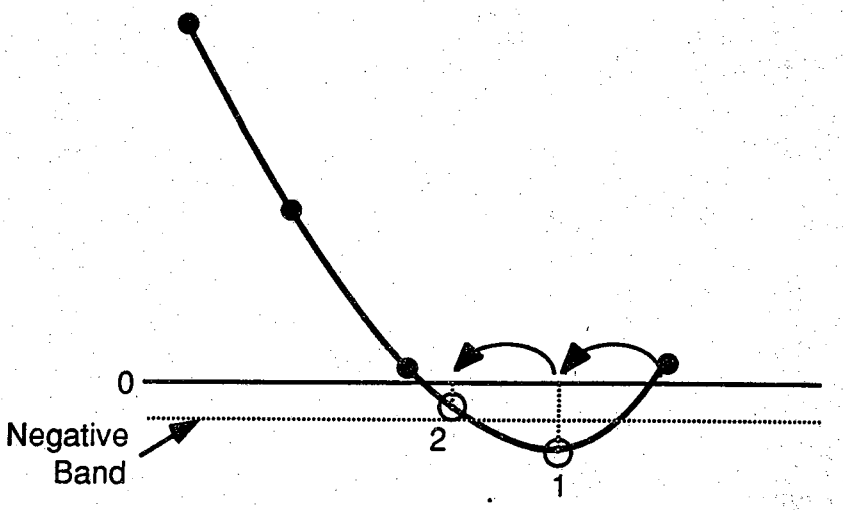


Figure 3.7. Short-lived discontinuity.

This is a very expensive way to find the Jacobian, especially in medium and large systems. The current version of TARDIS requires calculation of the Jacobian at each discontinuity regardless of its type. This inefficiency will be corrected in the next version of TARDIS. The data from TARDIS are obtained using Mac IIcx with THINK C version 4, and all calculations are done using double precision.

Example 1 The system differential equations are as follows:

$$y'_1 = \omega y_2 \tag{3.19}$$

$$y'_2 = -\omega y_1 \tag{3.20}$$

with $y_1(0) = 0$, $y_2(0) = 1$, $0 \le t \le 3$, and $\omega = \pi$. The switching function is

$$\phi = y_1 - A t \tag{3.21}$$

Assume that the locations where ϕ changes sign are to be found in the time interval above. The critical value of A is .4033006 where there will be two discontinuities in the interval of interest. If A is greater than the critical value, there will be only one discontinuity. However, if A is less than the critical value, there will be three discontinuities. Table 3.1 compares the results of TARDIS, those of Birta, et al., and exact values given by Birta, et al. Note that the parameters used in TARDIS are $\varepsilon_{rel} = \varepsilon_{abs}$ = .9e-7, the negative band is 1.e-10, and the maximum order of integration is 6 on Max IIcx. The exact time is also from Birta, et al.

Again, one should not conclude that the implementation of Gear method in TARDIS is more efficient than Runge-Kutta even though for each value of A the number of calls to the routine containing the system equations by TARDIS is less than those used by Runge-Kutta. Much more information is needed. Nevertheless, one can see from the instants of discontinuities just how reliable TARDIS is for this problem. Incidentally, EASY5 [8] with the Adams method also uses fewer calls with the same accuracy for this problem.

Table 3.1. Comparison of the results of test example 1.

A	Exact time (s)	Birta, et al.	TARDIS	RHS calls for RK	Calls in TARDIS
.35	.898206 2.29733 2.62827	.898206 2.29733 2.62828	.8982060 2.297335 2.628273	344	302
.40	.884843 2.41850 2.50000	.882843 2.41849 2.50001	.8848427 2.418501 2.499998	330	293
.41	.882196	.882197	.8821963	288	203
.45	.871693	.871693	.8716927	295	199

Let us push TARDIS more to the limit by trying the value of A closer to the critical value. If A is .4033, with the old values of ε_{rel} and ε_{abs}, TARDIS will miss two discontinuities completely. However, if one decreases ε_{rel} and ε_{abs} to 1.e-10, TARDIS will be able to detect all three discontinuities.

Example 2 The system differential equations are

$$y'_1 = y_2 \tag{3.22}$$

$$y'_2 = u(t) - .2\, y_2 - y_1 \tag{3.23}$$

from $0 \le t \le 30$. The input function u(t) takes the value either 0 or 1 whenever the switching function

$$\phi = y_1 - 1 \tag{3.24}$$

changes sign. Initially u(0) is 1. The results of this system are shown in Table 3.2. The same parameters used in TARDIS in the first test are also used in this test. The exact time is from Birta, et al. In this case, EASY5 [8] uses twice the number of calls used by Birta, et al. for the same accuracy.

Example 3 The last example is a 3rd order system.

$$y'_1 = \alpha_1 y_1 \tag{3.25}$$

$$y'_2 = \alpha_2 y_2 \tag{3.26}$$

$$y'_3 = y_1 + y_2 \tag{3.27}$$

with $y_1(0) = y_2(0) = 0.5$ and $y_3 = 0$. α_1 and α_2 will assume the values either 2 or -1 alternately. Initially α_1 is 2 and α_2 is -1. The switching function alternates between

$$\phi_1 = 1 - y_1 \tag{3.28}$$

Table 3.2. Comparison of the results of test example 2.

Exact time (s)	Birta, et al.	TARDIS	RHS calls for RK	Calls in TARDIS
1.679382	1.679382	1.679382	71	61
3.037086	3.037087	3.037085	127	126
6.194505	6.194505	6.194505	225	209
7.186908	7.186914	7.186906	274	263
10.34433	10.34433	10.34433	365	341
11.05712	11.05713	11.05711	407	390
14.21454	14.21455	14.21453	498	470
14.72415	14.72418	14.72415	533	519
17.88157	17.88159	17.88157	617	596
18.24636	18.24639	18.24634	645	642
21.40378	21.40380	21.40377	729	719
21.66569	21.66573	21.66567	757	762
24.82311	24.82314	24.82309	834	836
25.01178	25.01183	25.01176	862	877
28.16920	28.16924	28.16918	939	951
28.30551	28.30556	28.30547	974	1002

and

$$\phi_2 = 1 + y_2 \tag{3.29}$$

Initially ϕ_1 is in effect.

The solutions y_1 and y_2 are in the form ae^{bt}. The value of b alternates between 2 and -1. As time goes on, the switching frequency increases. Table 3.3 shows the comparison of the results from Birta, et al., and TARDIS; the exact values shown in the table are also from Birta, et al.

From these three examples, one can see that TARDIS is quite accurate in locating zeros. The performance of the present version of TARDIS can even be further improved by using user's defined Jacobian or providing a way for the user to update the Jacobian directly without numerical differencing.

Table 3.3. Comparison of the results of test example 3.

Exact time (s)	Birta, et al.	TARDIS	RHS calls for RK	Calls in TARDIS
.3465736	.3465740	.3465735	50	55
.8664340	.8664352	.8664338	92	121
1.126364	1.1263665	1.126364	113	177
1.2563298	1.2563314	1.256329	148	226
1.321312	1.321314	1.321311	190	276
1.353803	1.353805	1.353803	239	314
1.370049	1.370051	1.370048	288	347
1.378172	1.378174	1.378171	337	377
1.382233	1.382235	1.382232	386	408
1.384264	1.384266	1.384263	435	437
1.385279	1.385281	1.385279	484	465
1.385787	1.385789	1.385786	533	490
1.386041	1.386043	1.386040	582	514
1.386167	1.386170	1.386167	631	536
1.386231	1.386233	1.386230	680	560
1.386263	1.386265	1.386262	729	583
1.386278	1.386281	1.386278	778	605
1.386286	1.386289	1.386286	827	627
1.386290	1.386293	1.386290	876	649
1.386292	1.386295	1.386292	925	669

CHAPTER 4

SOLVING NON-LINEAR ALGEBRAIC EQUATIONS

4.1. Newton-Raphson Algorithm

There are quite a few methods that can be used to solve nonlinear algebraic equations. However, none seems to be as reliable as the Newton-Raphson (NR) algorithm, which is widely used in several general-purpose and application-specific simulation programs. Likewise, TARDIS uses the NR algorithm to solve nonlinear equations.

Given the algebraic equations of the form

$$\mathbf{g(y)} = \mathbf{0} \tag{4.1}$$

and the initial values for the variables in **y**, the NR algorithm solves the Eq (4.1) for **y** as follows:

$$\mathbf{J}\,\Delta\mathbf{y}^j = -\mathbf{g} \tag{4.2}$$

$$\mathbf{y}^{j+1} = \mathbf{y}^j + \Delta\mathbf{y}^j \tag{4.3}$$

where **J** is a Jacobian of **g** evaluated at $\mathbf{y}^j$.

If the initial values are close to the solution, the convergence will be fast. At the beginning of the simulation, TARDIS uses default zero initial values or user-supplied initial values to start the NR algorithm. After the state machines change their states, the initial values of **y** (the unknowns which include the non-state variables and the derivatives of state variables only) of the system equations at that point are taken from the values before the changes in the state machines. During the integration, however, the initial values of state

variables are the predicted values of the Gear algorithm. Since TARDIS also stores previous values of non-state variables for interpolations, these data can be used to provide the predicted values of the non-state variables. In sample test problems, the use of such extrapolated values of non-state variables does help the NR algorithm to converge faster, resulting in fewer computations overall.

Solving the Jacobian equation (4.2) is usually done by LU decomposition - instead of matrix inversion - to reduce the computations and preserve sparsity. Although, during the integration, the LU decomposition changes according to the new order and step size of integration, it is common practice in many simulation programs to reduce computations further by repeatedly using the same LU decomposition over a period of simulation time, as long as the convergence of the NR algorithm can be achieved in a few iterations. But with the variable-order, variable-step-size Gear algorithm, it is very unlikely that the same LU decomposition can be used for several steps without some modifications to reflect the change in the order or step size of the integration.

Petzold uses the following scheme in DASSL [20] to speed up the NR convergence.

$$\mathbf{y}^{j+1} = \mathbf{y}^{j} + c\,\Delta\mathbf{y}^{j} \tag{4.4}$$

and c is calculated from

$$c = \frac{2}{1 + \beta_{curr}/\beta_{old}} \tag{4.5}$$

where β_{curr} is the current value of β, and β_{old} is the value of β where the Jacobian was last evaluated (see equation (3.5) for β). This scheme had been tried in TARDIS, but it was found that, for the kinds of problems of interest, the computations required actually increased. Thus the scheme above is not suitable for TARDIS, and another scheme is needed.

During integration, it was observed that the value of β is very large compared to other terms in the system equations most of the time. Consequently, when the LU decomposition is performed, the entries involved with β are often chosen as pivots. This observation led us to implement a scheme for updating the LU decomposition directly based on the ratio of β_{curr} to β_{old}. Pivots involved with β will be modified as follows:

$$pivot_{new} = pivot_{old} \frac{\beta_{curr}}{\beta_{old}} \tag{4.6}$$

Pivots that are not involved with β will not be updated. This scheme seems to extend the use of the same Jacobian for several more time steps in the sample tests.

A straightforward implementation of the NR algorithm may not be suitable for certain problems: for example, problems with exponential functions which can make the algorithm diverge. When faced with this type of problem, TARDIS lets the user specify any limit function that will be applied on some or all variables in $\Delta\mathbf{y}$. By default there is no limit function.

There are several criteria that can be used to test the convergence of the NR algorithm [42]. The default convergence criterion used in TARDIS is

$$|\Delta y_i| < \varepsilon_{rel}|y_i| + \varepsilon_{abs} \tag{4.7}$$

where ε_{rel} and ε_{abs} are relative and absolute error tolerances. The stopping criterion is implemented in a separate routine so that the user can change it if needed. By default, the same ε_{rel} and ε_{abs} are applied to all variables in $\mathbf{y}$.

In the current version, TARDIS calculates the Jacobian $\mathbf{J}$ by numerical approximation. Thus

$$\frac{\partial f_j}{\partial y_i} \approx \frac{f_j(y_1, \ldots, y_i + \sigma_i, \ldots, y_n) - f_j(y_1, \ldots, y_i, \ldots, y_n)}{\sigma_i} \tag{4.8}$$

The value of σ_i is critical to avoid numerical cancellation and to achieve fast convergence for NR algorithm. Stoer and Bulirsch [43] suggests that σ_i should be such that the difference of f_j in Eq. (4.8) is about the half the machine precision of f_j itself. However, for TARDIS there is no a priori knowledge of the types of functions, so the value of y_i will be perturbed by half the machine precision instead. This approach seems to be working well for the sample problems tested. As in the case of the stopping criterion, the user can write a routine to replace the default routine for perturbing the value of y_i provided in TARDIS.

4.2. Solving the Jacobian Equation

Instead of matrix inversion, the LU decomposition technique is employed in solving the Jacobian equation (4.2) to reduce computations and preserve sparsity. Note that the LU decomposition is equivalent to Gaussian elimination, but the LU decomposition is prefered because the decomposition can be reused for different problems in which only the right-hand-side vector in the Jacobian equation (4.2) changes, whereas, with the Gaussian elimination, the whole elimination process must be repeated if the right-hand-side vector changes. There are several equivalent methods for LU decomposition [44, 45, 46], and there are also incomplete LU decomposition methods which need some iterations [47, 48]. TARDIS provides just the Crout algorithm for LU decomposition [46, 49] at this time. The decomposition of **J** will give **L** and **U** matrices such that

$$\mathbf{J} = \mathbf{L}\,\mathbf{U} \tag{4.9}$$

where **L** and **U** matrices have the following forms

$$\mathbf{L} = \begin{bmatrix} \times & 0 & 0 & 0 \\ \times & \times & 0 & 0 \\ \times & \times & \times & 0 \\ \times & \times & \times & \times \end{bmatrix} \tag{4.10}$$

and

$$\mathbf{U} = \begin{bmatrix} 1 & \times & \times & \times \\ 0 & 1 & \times & \times \\ 0 & 0 & 1 & \times \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.11}$$

The entries of **L** and **U** matrices can be found as follows: The first step is

$$L_{j1} = J_{j1} \quad \text{for } j = 1, \ldots, n \tag{4.12}$$

$$U_{1j} = J_{1j} / L_{11} \quad \text{for } j = 2, \ldots, n \tag{4.13}$$

For $i = 2, \ldots, n$, the i^{th} step is

$$L_{ij} = J_{ij} - \sum_{k=1}^{i-1} L_{ik} U_{kj} \quad \text{for } j = i, \ldots, n \tag{4.14}$$

$$U_{ji} = (J_{ji} - \sum_{k=1}^{i-1} L_{jk} U_{ki}) / L_{ii} \quad \text{for } j = i+1, \ldots, n \tag{4.15}$$

To conserve memory space, entries in the original matrix **J** can be replaced one by one by the entries obtained during the decomposition one-by-one. Since the diagonals of **U** are always 1, there is no need to reserve any space for them.

The result after the i^{th} step is a submatrix of the size $(i-1)\times(i-1)$ to be processed the same way as the previous step - i.e.,

$$J_{ij} = J_{ij} / J_{ii} \quad \text{for } j = i+1, \ldots, n \tag{4.16}$$

$$J_{jk} = J_{jk} - J_{ji} J_{ik} \quad \begin{matrix} \text{for } j = i+1, \ldots, n \\ \text{for } k = i+1, \ldots, n \end{matrix} \tag{4.17}$$

From Eq. (4.16), it is clear that the pivot J_{ii} should not be zero, a condition which cannot be guaranteed for general matrices. To avoid a division by zero, other nonzero entries in the rows must be chosen as pivots. Note that

the pivots are used to divide all other entries in the same rows to get the entries in the **U** matrix; the above pivoting scheme is called row pivoting. Choosing the biggest entries in the rows for row pivoting (or columns for column pivoting) of submatrices is called partial pivoting. It has the effect of limiting the growth of values in the decomposition process, which usually results in better numerical stability. Note that large values of entries involved in partial pivoting are more common. But large values do not necessarily imply large backward error in the solution [50]. The user should take this into consideration when choosing a threshold to determine which entries are to be chosen as pivots. The threshold pivoting strategy used in TARDIS will be described later in Section 4.3.2.

There is another LU decomposition method called the Doolittle algorithm [46] in which the column pivoting scheme is used. Column pivoting is in fact more common than the row pivoting [51] described above. The value of the norm of residue **r** which is defined as

$$\mathbf{r} = \mathbf{J}\,\Delta\mathbf{y} - (-\mathbf{g}), \tag{4.18}$$

will be smaller in column pivoting than in row pivoting [51]. However, TARDIS uses the row pivoting scheme because it is more compatible with the LU decomposition and the data structures of sparse matrices. Nevertheless, the stopping criterion of the NR algorithm will guarantee that $\Delta\mathbf{y}$ must be within some specified bound before the NR algorithm stops.
The solution of the Jacobian equation (4.2) using **L** and **U** matrices can be divided into two steps: forward elimination and back substitution. In the forward elimination step, a temporary vector solution $\mathbf{y}_{tmp}$ is found from

$$\mathbf{L}\,\mathbf{y}_{tmp} = -\mathbf{g} \tag{4.18}$$

The expanded form of Eq. (4.18) is

$$y_{1,tmp} = -g_1 / L_{11}, \tag{4.19}$$

$$y_{2,tmp} = (-g_2 - L_{21}\,y_{1,tmp}) / L_{22}, \tag{4.20}$$

$$y_{n,tmp} = (-g_n - \sum_{i=1}^{n-1} L_{ni}\, y_{i,tmp}) / L_{nn}, \qquad (4.21)$$

where n is the number of equations. The desired solution, $\Delta\mathbf{y}$, is calculated in the back substitution step from the following equation

$$\mathbf{U}\,\Delta\mathbf{y} = \mathbf{y}_{tmp} \qquad (4.22)$$

The expanded form of Eq. (4.22) is

$$\Delta y_n = y_{n,tmp} \qquad (4.23)$$

$$\Delta y_{n-1} = y_{n-1,tmp} - U_{(n-1),n}\,\Delta y_n \qquad (4.24)$$

$$\Delta y_k = y_{k,tmp} - \sum_{i=n}^{k-1} U_{ki}\,\Delta y_i \qquad (4.25)$$

As one can see, the above forward elimination and back substitution steps can be repeated for different right-hand-side vectors without going through the LU decomposition process again.

4.3. Sparse Matrix Techniques for LU Decomposition

Sparse matrix techniques have been implemented in TARDIS mainly to reduce computations, especially in large systems. Symbolic LU decomposition is not appropriate for TARDIS because the structure of the system equations is not known. The structure of the system equations depends on the order in which the equations are written. Although the equations for the electrical components may be written so that their corresponding parts in the Jacobian are symmetric or nearly symmetric in structure, the equations from other non-electrical components may not be written as such. Moreover, with the symbolic LU decomposition, it is not possible to consider the value of the pivots. If the pivots chosen by symbolic

decomposition happen to be much smaller than other terms, the entries in **L** and **U** may increase to very large values. To avoid such numerical problems, the method used in TARDIS decomposes the Jacobian matrix based on numerical values while trying to preserve sparsity at the same time. This method will be explained later.

4.3.1. Data Structures for Sparse Jacobian

The data structure chosen for sparse Jacobian must be suitable for the LU decomposition. In the LU decomposition, elements of the original matrix must be readily accessible by both rows and columns. The data structure used in TARDIS for sparse matrices is modified from that of Horowitz and Sahni [52] which uses circular linked lists. For the LU decomposition, however, there is no need to use circular linked lists, so simple linked lists are used instead. Figure 4.1 shows the data structure for one entry of the matrix. Each entry has five fields: value of entry, row number, column number, pointer to the next entry in the same row, and pointer to the next

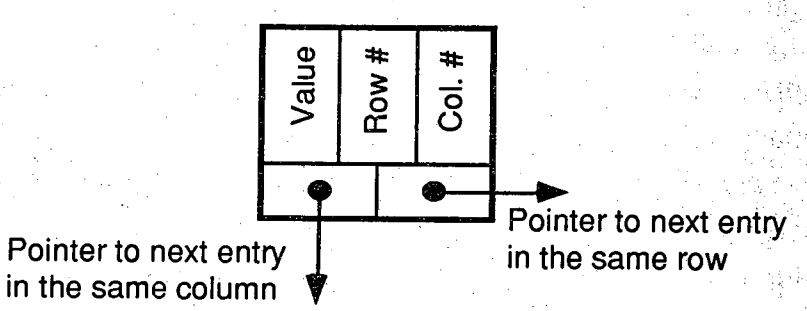


Figure 4.1. Data structure for individual entry.

$$\begin{bmatrix} 2 & 0 & 7 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 9 & 0 & 5 \\ -4 & 0 & 0 & 3 \end{bmatrix}$$

Figure 4.2. Sample matrix.

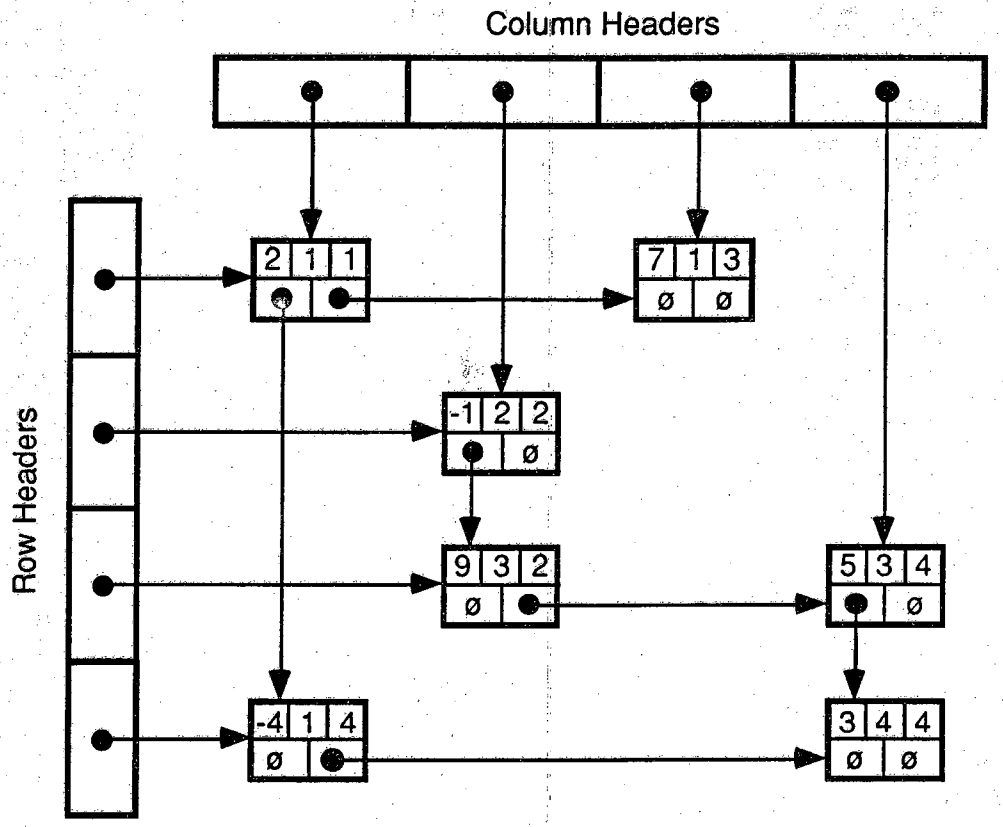


Figure 4.3. Row and column linked lists for sparse matrix.

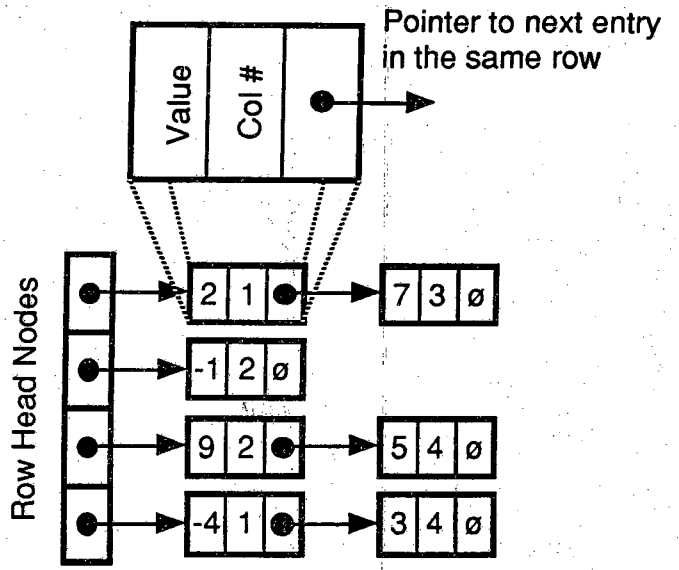
entry in the same column. Figure 4.3 illustrates how the structures are combined to represent a sample matrix in Fig. 4.2. The NULL pointer, ø, indicates that there are no more elements in the same row or column. Every row- and column-linked list begins with a head node. The search for elements in the matrix starts from these head nodes. This data structure allows orderly sequential access to all elements in the matrix in both rows- and column-directions.

Using this data structure described above, the LU decomposition is about five times slower than that with symbolic LU decomposition on a medium-size finite-element problem with about 1700 equations. A significant portion of time is spent on arranging elements in the linked lists in ascending order. Since the LU decomposition does not require the ordering of column linked lists, separating the column linked lists from the row linked list as shown in Fig. 4.4 may improve the speed by not updating the column-linked lists without ordering (new information about fill-ins can be added to the beginning of column linked lists). The information about columns which is no longer needed after the pivots corresponding to those columns are selected can be put back in the free-storage pool for reuse. Note that after the LU decomposition is done, all the elements in column linked lists are returned to the free-storage pool. The double-precison values of the entries are stored in the row linked lists only. The speed improvement with these separated data structures is found to be about a factor of two. It is possible not to order entries in row linked lists also, and this is left for future investigation to see whether further speed improvement can be made.

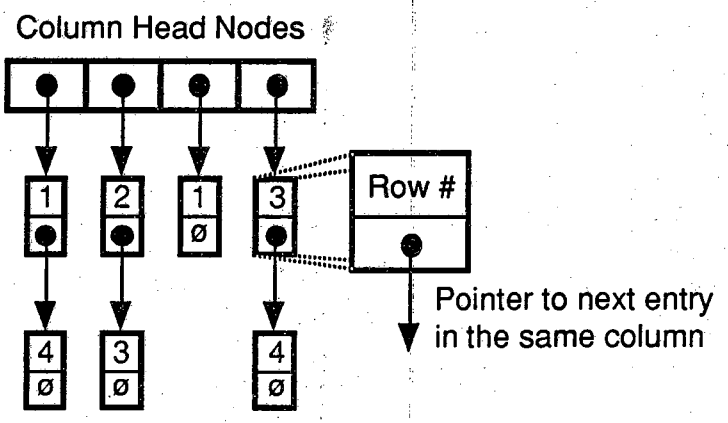
4.3.2. Markowitz Strategy with Threshold Pivoting

There are several strategies for reordering equations to minimize the number of fill-ins. Nagel [53] and Duff, et al. [54] have, however, reported that none seems to perform better than the Markowitz strategy. The strategy of Markowitz [48] is to select the pivot from the entry with the lowest Markowitz count,

$$(r_k - 1)\,(c_k - 1), \tag{4.25}$$



a) Row linked lists.



b) Column linked lists.

Figure 4.4. Separate row and column linked lists.

a) Row linked lists.
b) Column linked lists.

where r_k and c_k are respectively the numbers of row and column of nonzero entries of the k^{th} candidate for pivot. However, to ensure numerical stability in the LU decomposition, entries that have large values should be chosen as the pivots. To compromise between sparsity and numerical stability as suggested by Duff, et al. [55], TARDIS uses the Markowitz strategy with threshold pivoting. In this pivoting scheme, those entries having values larger than some threshold relative to the maximum value in the same row are considered as candidates for the pivots - i.e.,

$$J_{pv,k} \geq \mu \max_j |J_{ij}| \tag{4.26}$$

where $J_{pv,k}$ is the k^{th} candidate for the pivot of the i^{th} row, and μ is a constant between 0 and 1. When 1 is used for μ, the pivoting scheme becomes partial pivoting. Duff, et al., suggest a value of 0.1 for μ from their extensive testing. In TARDIS, the user may specify the value of μ between 0 and 1, or use the default value of 0.1 provided.

The Markowitz strategy requires the storage of numbers of nonzero entries in the rows and columns. Duff, et al. suggest storing them in separate doubly linked lists, one for rows and the other for columns, for easy updating [55]. The row and column linked lists are alternately scanned in the order of the increasing number of nonzero entries. The limit on the number of rows and columns to be scanned can be specified by the user. The default limit is 3. Whenever there are several candidates for pivots that have the same minimum Markowitz counts, TARDIS will choose the entry with the largest value to be the pivot, as suggested by Osterby and Zlatev [56].

CHAPTER 5
SAMPLE SYSTEMS

5.1. Modelling of Switches in Electrical Circuits

The switches in electrical circuits are usually handled in two different ways: ideal switches and high-and-low resistance switches. For an ideal switch, when it is ON, there is no resistance, and thus the voltage drop across it is zero. When the switch is OFF the current passing through it is zero. The system equations of an electrical circuit having its switches modelled as ideal switches are usually varied depending on the states of the switches. The numbers of state variables and non-state variables are also varied. Simulating such system is very tricky. One technique that can be done is to find the equations of all possible combinations of switch states. For a complicated system, such combinations can be prohibitively large. Another technique is to use a tensor approach which uses a connection matrix to relate independent variables of the connected subcircuit [57]. Note that it is possible for an electrical circuit to have several unconnected subcircuits, in which case the simulation must be done separately on each subcircuit. What is usually done, however, is to have components with high impedances connecting all unconnected subcircuits to a reference node.

For a high-and-low resistance switch, the on state is represented by a low resistance while the OFF state is represented by a high resistance. With such modelling of switches, the structure of the system equations of an electrical circuit containing switches is fixed, but the parameters in the equations can vary depending on the states of the switches. The number of state variables and the number of non-state variables are also fixed.

In terms of simulation, both switch models require about the same amount of computations when they are handled properly. Using the ideal switches may have the advantage that the number of state variables may be reduced somewhat for some periods of the simulation. But there is an overhead in formulating the equations from the changing circuit topology.

For TARDIS, any model of switches will work just fine as long as the system equations are posed correctly. Note that TARDIS expects fixed numbers of state variables and non-state variables. Using ideal switches in TARDIS would be more tricky than using high-and-low resistance switches. Therefore, switches in all sample tests in this chapter are modelled by high-and-low resistance.

5.2. Sample Test Circuits

Presented in this chapter are simulations of four sample systems to show how the various ideas discussed earlier will perform when implemented together within TARDIS. As in other simulation programs, TARDIS has some limitations, which the user is advised to take notes. The following suggestions are offered to avoid numerical problems that may arise.

1. When dealing with switches, avoid extreme values of ON and OFF resistance. Choose some reasonable values that will have no significant effect on the simulation. Using extreme values can make the program reduce the integration step size to very small values or even abort when the Jacobian matrix becomes singular.

2. Choose a small enough value of the negative band tolerance so that the residue will not affect the accuracy of the simulation. But too small a value may lead to numerical problem. Do not choose zero for this tolerance.

3. If equations can be reduced by mere inspection, the user is encouraged to reduce them and thus speed up the simulation.

4. If it is known in advance that dynamics which occur very fast are not of interest, the user is advised not to include the corresponding terms or equations in the simulation since TARDIS will not hesitate to reduce the step size down to a very small value to accommodate such fast transients.

5. Do not eliminate the number of state variables by merely zeroing their coefficients; they should be eliminated explicitly. For example, in $V = L\frac{di}{dt}$, the state variable i cannot be eliminated by zeroing the L. A very small value of L may result in a very small integration step size. The best way to get rid of some state variables from the simulation is not to include them in the equations.

In the next sections, the following sample systems will be examined:

1. simple R-L circuit with one diode,
2. single-phase full-bridge with dc motor,
3. high-frequency inverter, and
4. induction motor with current source inverter.

The complexity of these sample systems increases in the order that they are presented.

5.2.1. Simple R-L Circuit with One Diode

Although the circuit diagram shown in Fig. 5.1 looks like a simple circuit, it is not easy to simulate the behavior of diode correctly without reintegrating to the discontinuity points, as mentioned in Chapter 3. The MainSystem() and MainEvent() routines for this system are shown in Appendix C.

The state transition diagram describing a diode's behavior is shown in Fig. 5.2. Since the voltage-current relationship of the diode is simply $V_D = R_D I_D$, the current and voltage always have the same sign. Detecting the value of

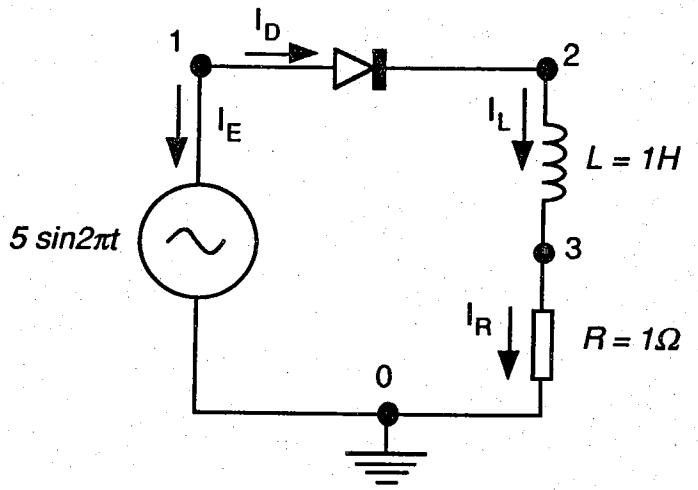


Figure 5.1. Circuit diagram of an R-L circuit with one diode.

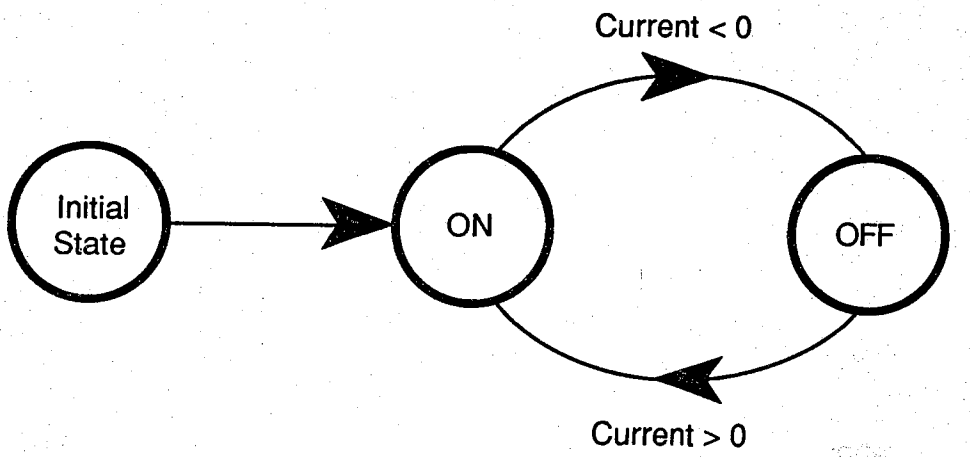


Figure 5.2. State transition diagram of a diode.

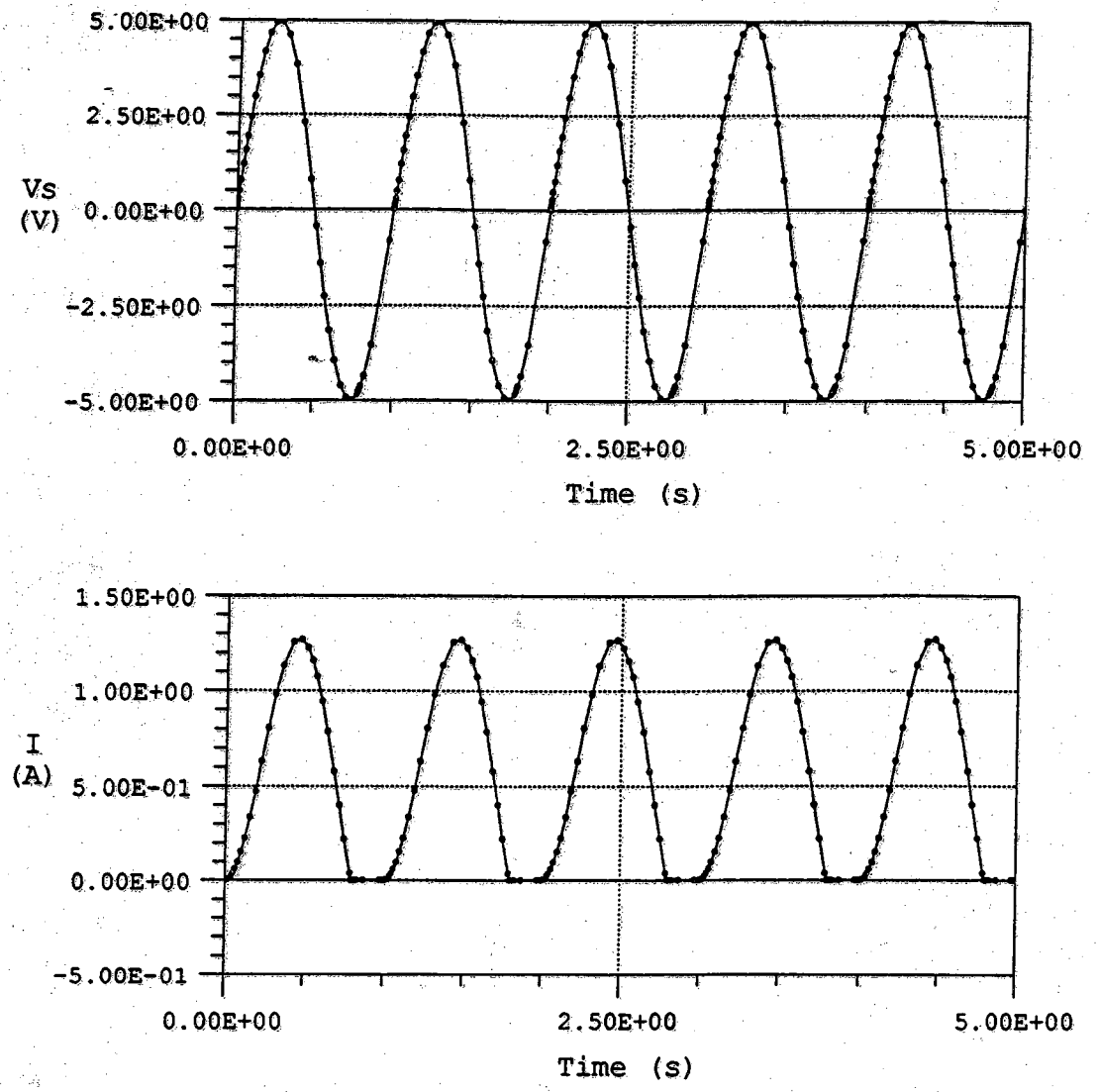


Figure 5.3. Output of simple R-L circuit with one diode.

current passing through the diode is equivalent to detecting the value of voltage across it. Better accuracy is obtained by tracking the diode voltage for the transition from OFF to ON and the diode current for transition from ON to OFF. The output points of the results obtained from TARDIS are plotted in Fig. 5.3. These points are close together right after the discontinuities because the order of integration is reset to one and the step size begins with some small value.

For this circuit, if a zero-crossing point of the diode's switching function, obtained from the root-finding routine, is used without reintegrating to such point to confirm the existence of the discontinuity, a false ON state of the diode may occur at that point.

If a simple mechanism is used to handle the discontinuities of the diode's operation, such as the IF-ELSE statement used in the following pseudo code

if ($I_L > 0.e0$)
    $R_D = 1.e\text{-}4$;
else
    $R_D = 1.e7$;
$0 = 5 \sin(2\pi t) - I_L(Rd+1) - \frac{dI_L}{dt}$;

where R_D is the diode resistance, then there will be output points close together both before and after the discontinuities, and the instants where the diode starts conduction will be less consistent than those obtained by using the state machine, as shown in Fig. 5.4. The reason is, when the integration process encounter a discontinuity the first time around when integrating beyond the discontinuity point, the local truncation error detected is large due to an abrupt change in the diode's resistance. As a result, the current integration step has to be repeatedly carried out with a smaller and smaller step size until the local truncation error criterion is satisfied, and the number of iterations required by the Newton-Raphson algorithm also increases because the diode's resistance beyond the discontinuity point does not

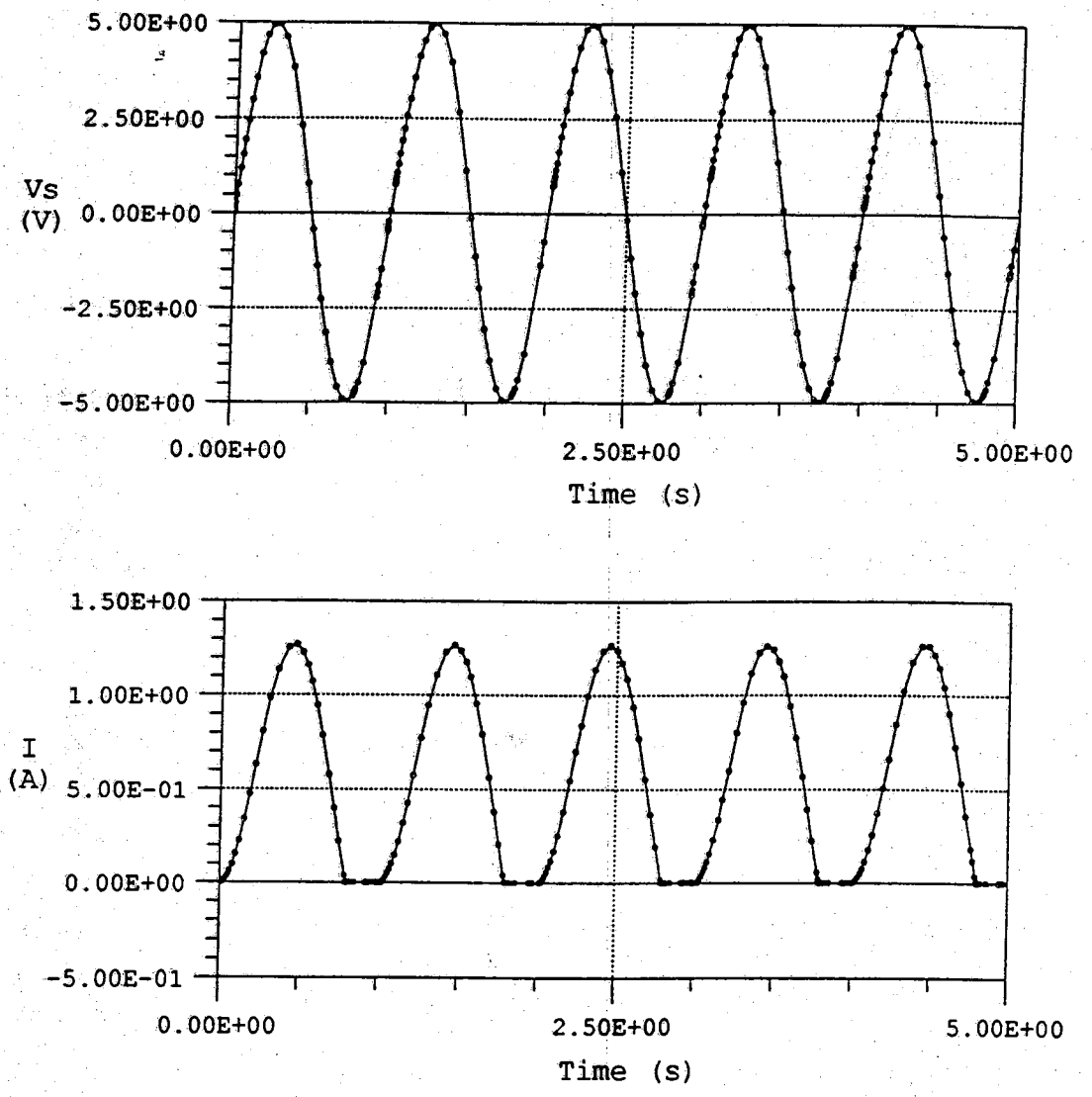


Figure 5.4. Output of the circuit in Fig. 5.1 using IF statements.

correspond to that in the Jacobian which is evaluated before the discontinuity. Thus the performance using the above form of coding to handle the diode's operation is much poorer than the state machine implementation used in TARDIS.

For more complicated systems, the use of similar coding, like the one above, to handle components associated with discrete events may cause the Newton-Raphson to diverge, no matter how small the integration step size might be.

5.2.2. Single-Phase Full Bridge with DC Motor

The single-phase drive system shown in Fig. 5.5 is taken from [38]. The purpose of this simulation is to compare the results with those given in [38] which were obtained from an analog computer. The "mainsys.c" file for this system is given in Appendix D.

The machine is a 240 V, 5 hp dc shunt motor. The rated motor current is 16.2 A and the rated speed is 1220 rpm. The simulated condition is with the motor running at a constant speed of 1318 rpm (138 rad/s), carrying the rated load torque. The voltages and currents of the dc motor and the voltage source are shown in Fig. 5.6. For this particular operating condition, there are intervals in which the motor current is zero. Such discontinuous conduction of the bridge occurs when the back electromotive force (e.m.f.) of the motor is higher than the source voltage. Note that as simulated the commutation inductance of the voltage source is not zero. If it is, the problem reduces to one with ideal ac voltage supply to the bridge - a simpler problem.

5.2.3. High-Frequency Inverter

The circuit of this system is shown in Fig. 5.7. The purpose of this sample test is to show the comparison of the modelling of a capacitor and an inductor. Connected between nodes 3 and 4 is a capacitor in series with the resistor of 0.1 Ω; these elements are modelled as follows:

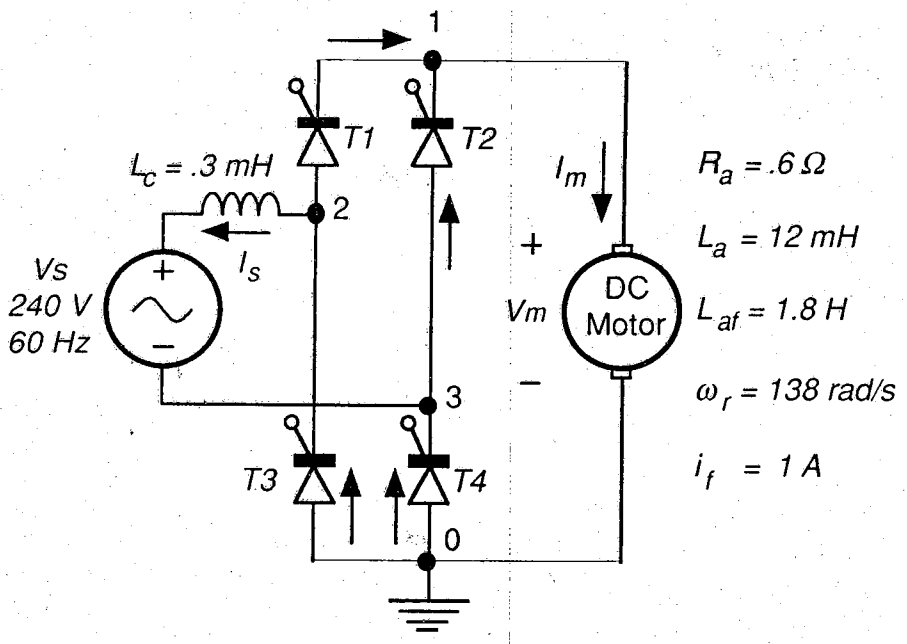


Figure 5.5. Single-phase full-bridge rectifier with dc motor.

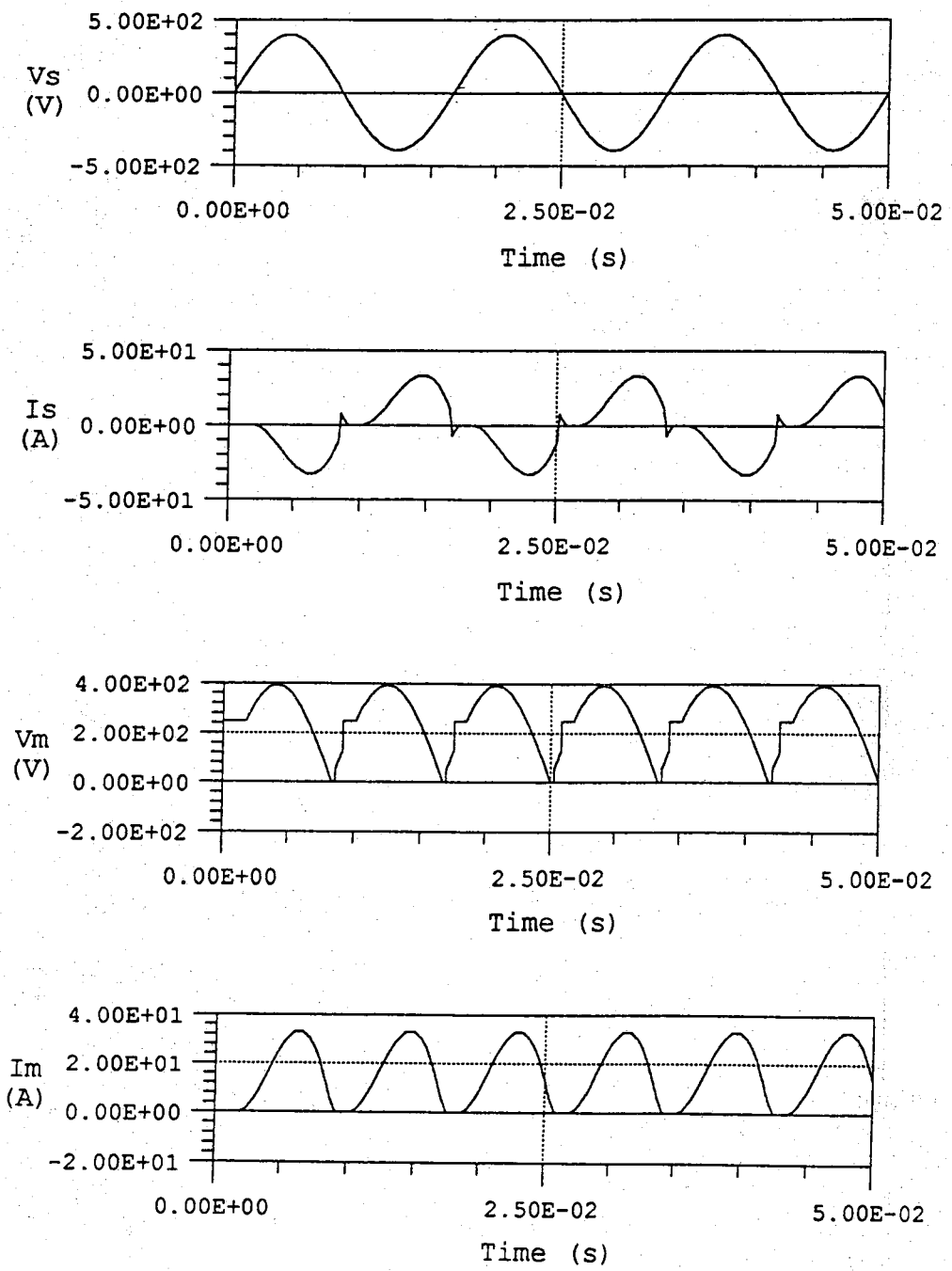


Figure 5.6. Output of the simulation of the circuit in Fig. 5.5.

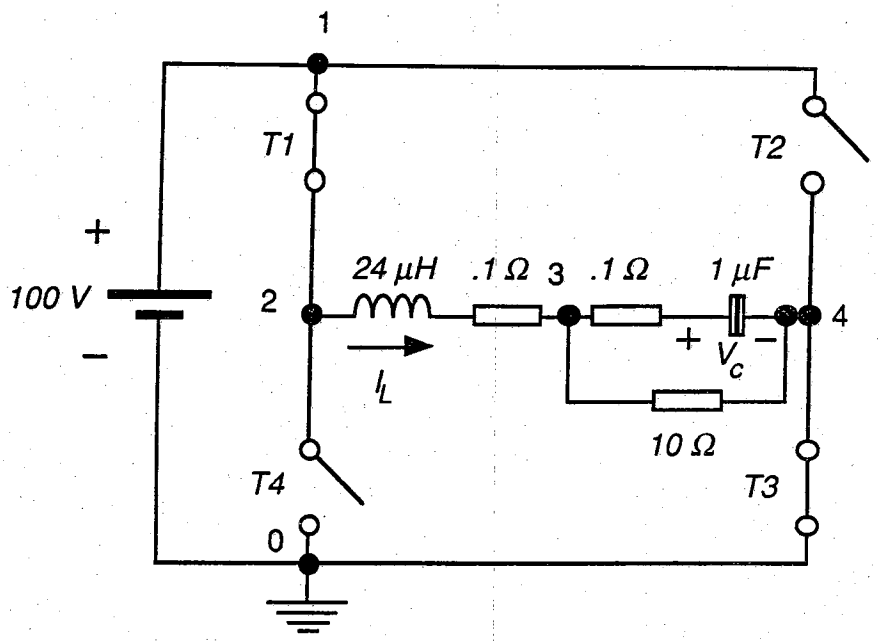


Figure 5.7. High frequency inverter circuit.

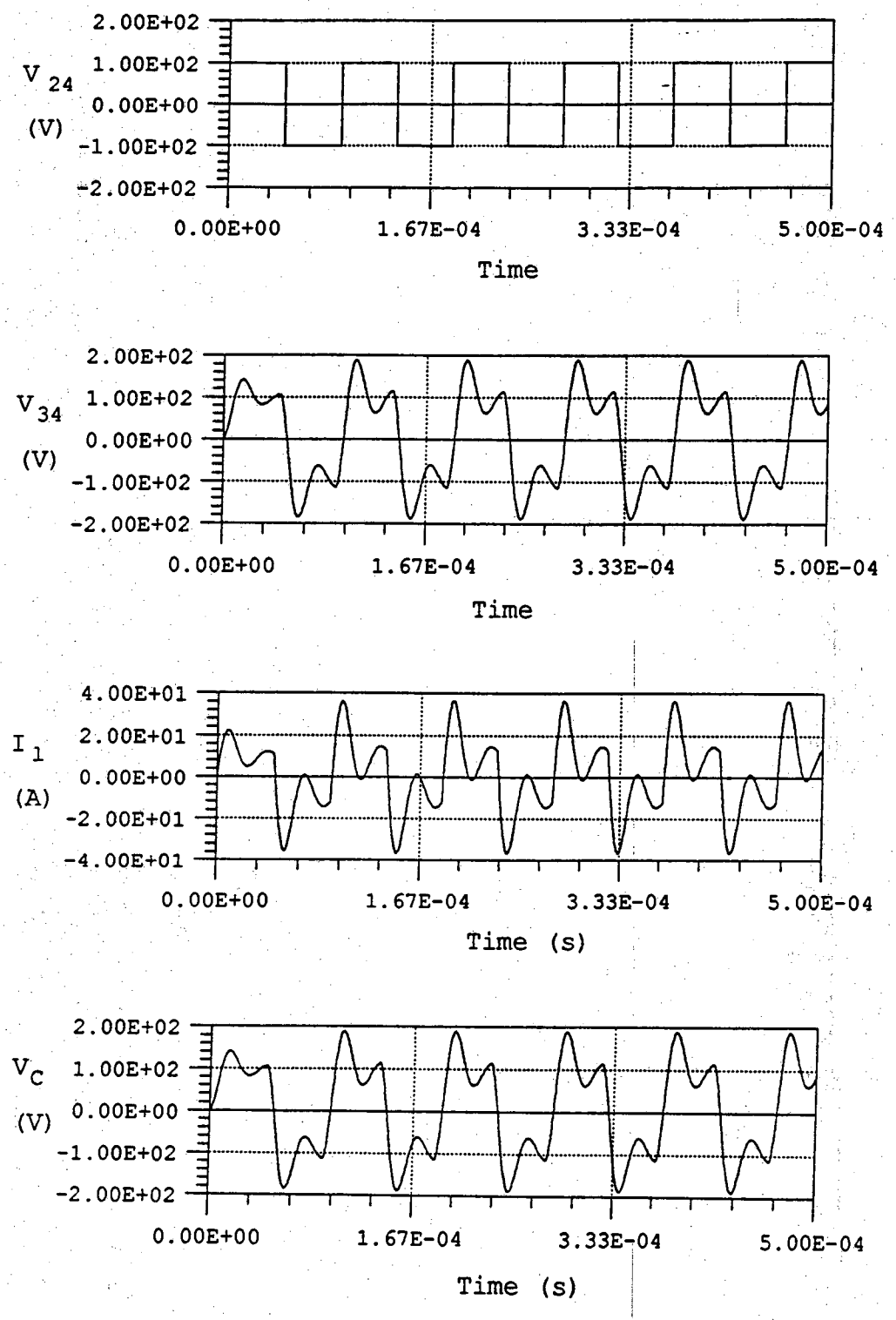


Figure 5.8. Output of high frequency inverter as shown in Fig. 5.7.

$$0 = V(3) - V(4) - (0.1\, I_C + V_C) \quad (5.1)$$

$$0 = I_C - C\frac{dV_C}{dt} \quad (5.2)$$

As one can see from the above equations, V_C is a state variable while V(3), V(4), and I_C are non-state variables. In general, adding a capacitor to the circuit will add one state variable (V_C) and one non-state variable (I_C) to the system equations. Adding an inductor to the circuit will, on the other hand, add only one state variable; for example, the equation for the inductor of 24 μH in series with the resistance 0.1 Ω can be written as

$$0 = V(2) - V(3) - (L\frac{dI_L}{dt} + 0.1\, I_L) \quad (5.3)$$

When operated at the switching frequency of 10.83 KHz with zero initial conditions for both the capacitor voltage and the inductor current, the waveforms of the circuit are as shown in Fig. 5.8.

5.2.4. Induction Machine with Current Source Inverter

The power circuit of a current-fed induction motor drive is shown in Fig 5.9, and the block diagram of the control part is shown in Fig. 5.10. The parameters of the machines and the control scheme are the same as those given in [39]. The initial values of the motor speed and dc-link current are zero. For the simulation results shown, the input speed was set at 900 rpm or 50 % of the rated speed. The load torque was assumed to be proportional to the speed; thus at 50 % of the rated speed the load torque is 50 % of the rated value. The induction machine equations are nonlinear due to the nonlinearities in the magnetization characteristics, in the speed voltage terms, and in the torque term. Furthermore, some control elements can be nonlinear or can cause discontinuities. The "mainsys.c" file containing program codes of the system equations and state machines is listed in Appendix F.

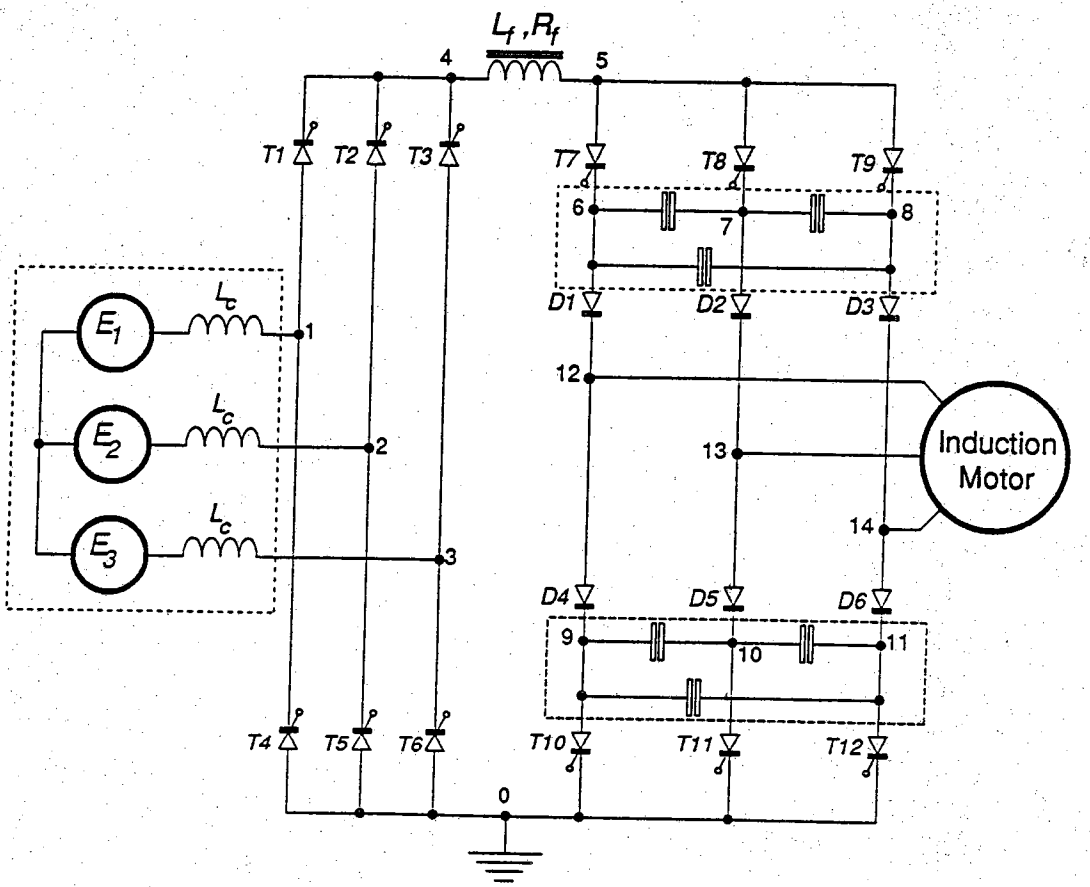


Figure 5.9. Induction motor with current source inverter.

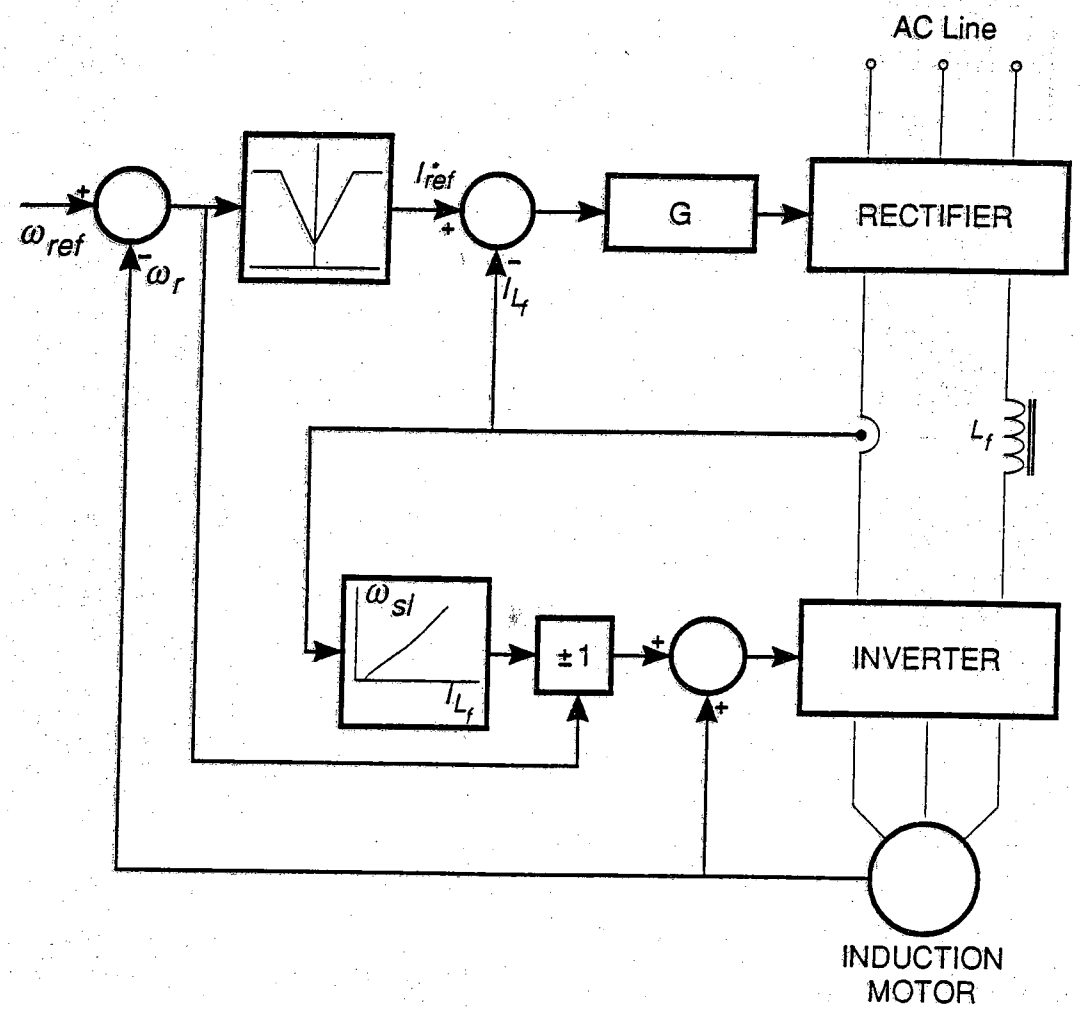


Figure 5.10. Control scheme of induction motor system in Fig. 5.9.

For this particular system, special considerations have to be given to the Y-connected 3-phase voltage source and two Δ-connected capacitor banks in the inverter circuit. The three inductor currents in the Y-connected source are not independent of each other, and neither are the three capacitor voltages of each Δ-connected capacitor bank. The three inductors in the voltage source should be considered together as one single unit with only two state variables; the same applies to the three capacitors of the Δ-connected bank.

Since the startup of the drive takes a long time, relative to the switching operations, to preserve some of the details due to the switching the results of the startup run are divided up into three separate sections: the interval from 0 s to .5 s in Fig. 5.11, the interval from .5 s to 1 s in Fig. 5.12, and the interval from 5.5 s to 5.6 s in Fig. 5.13. As one can see, there are many discontinuities in the current waveforms; these are also reflected in the magnetic flux and torque of the motor. The firing of the thyristors in the rectifier is synchronized to the frequency of the ac voltage source, and that of the thyristors in the inverter is synchronized to the motor speed which increases from a zero value to the desired value at 900 rpm.

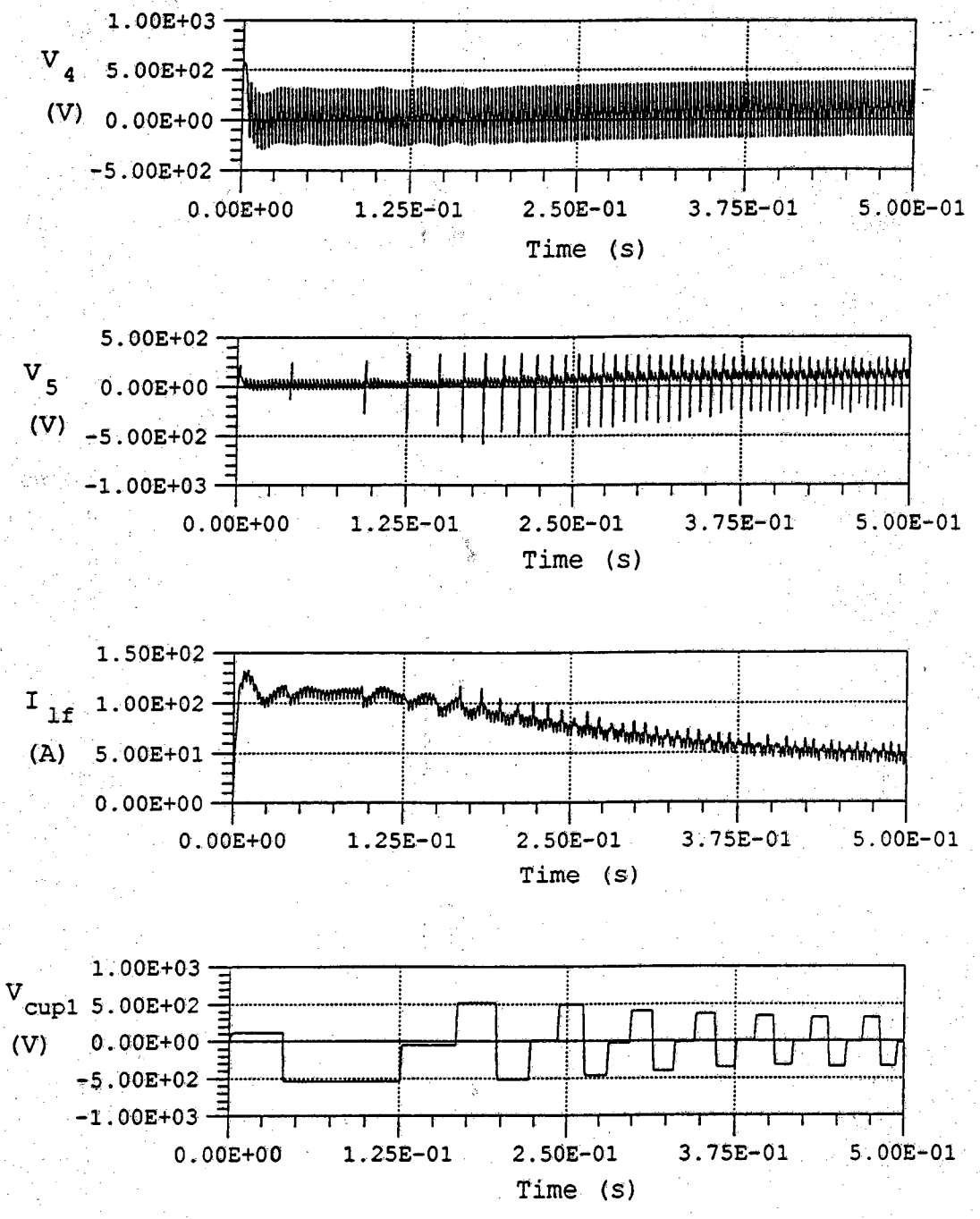


Figure 5.11. Operations of induction motor system from 0 s to .5 s.

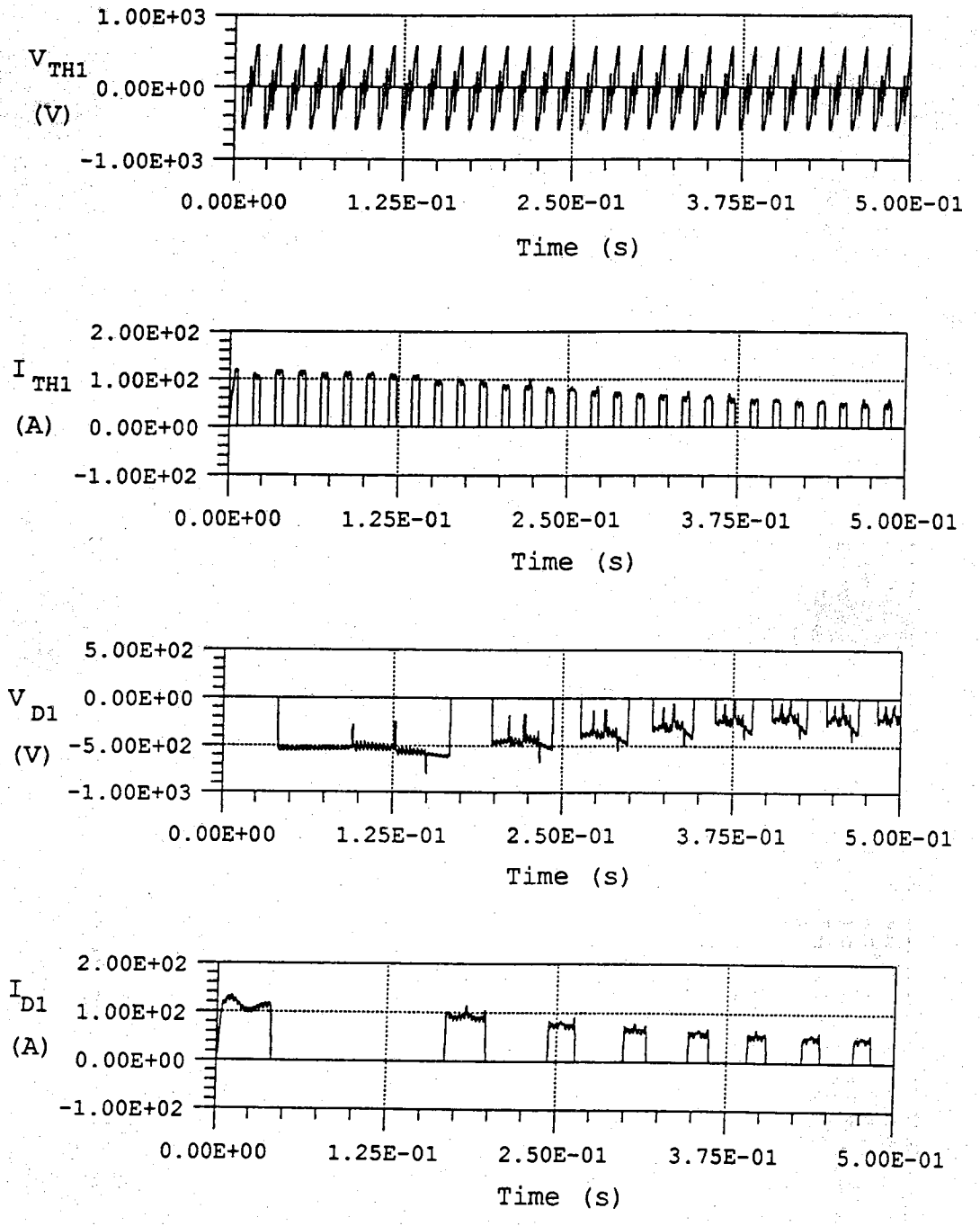


Figure 5.11. Continued.

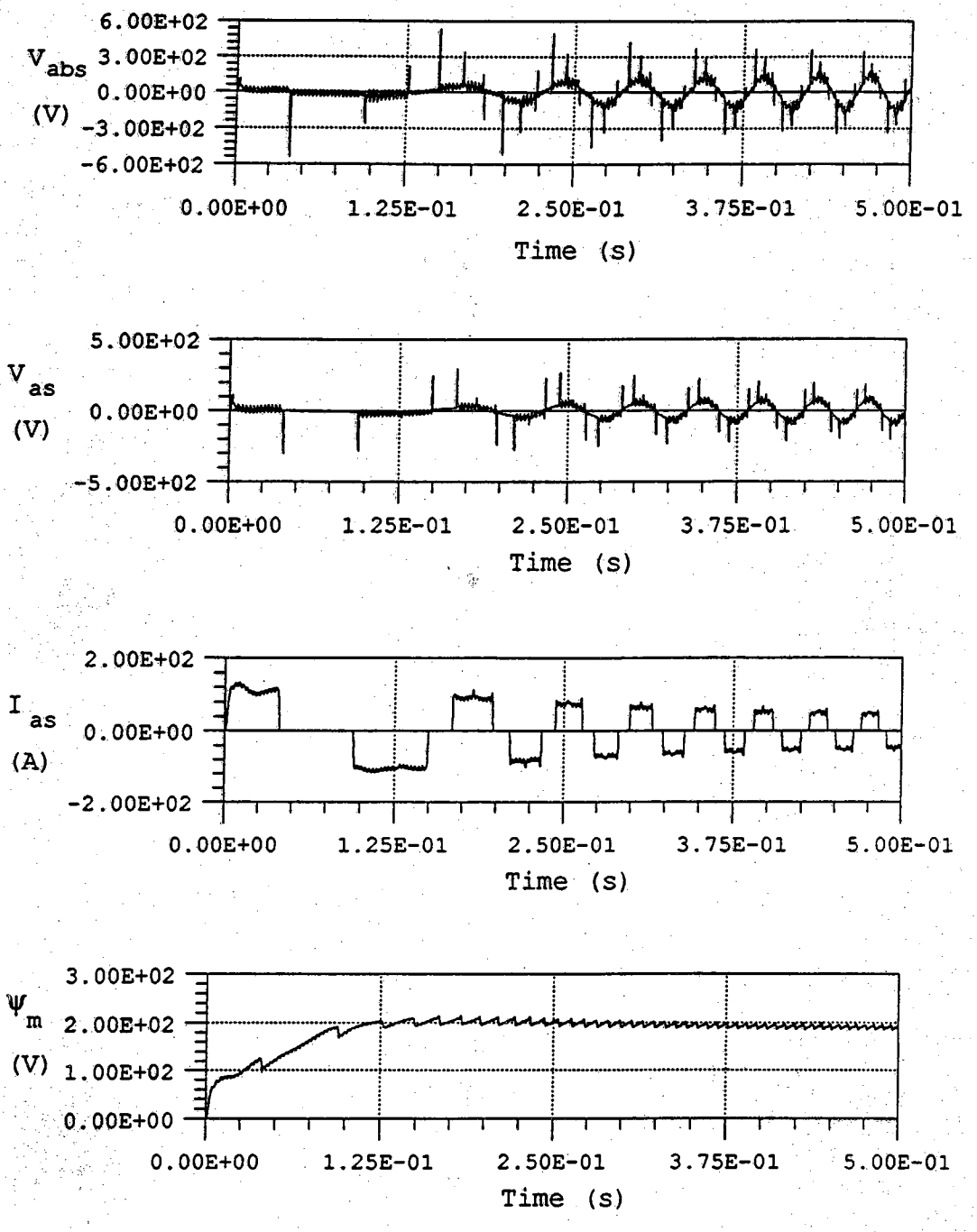


Figure 5.11. Continued.

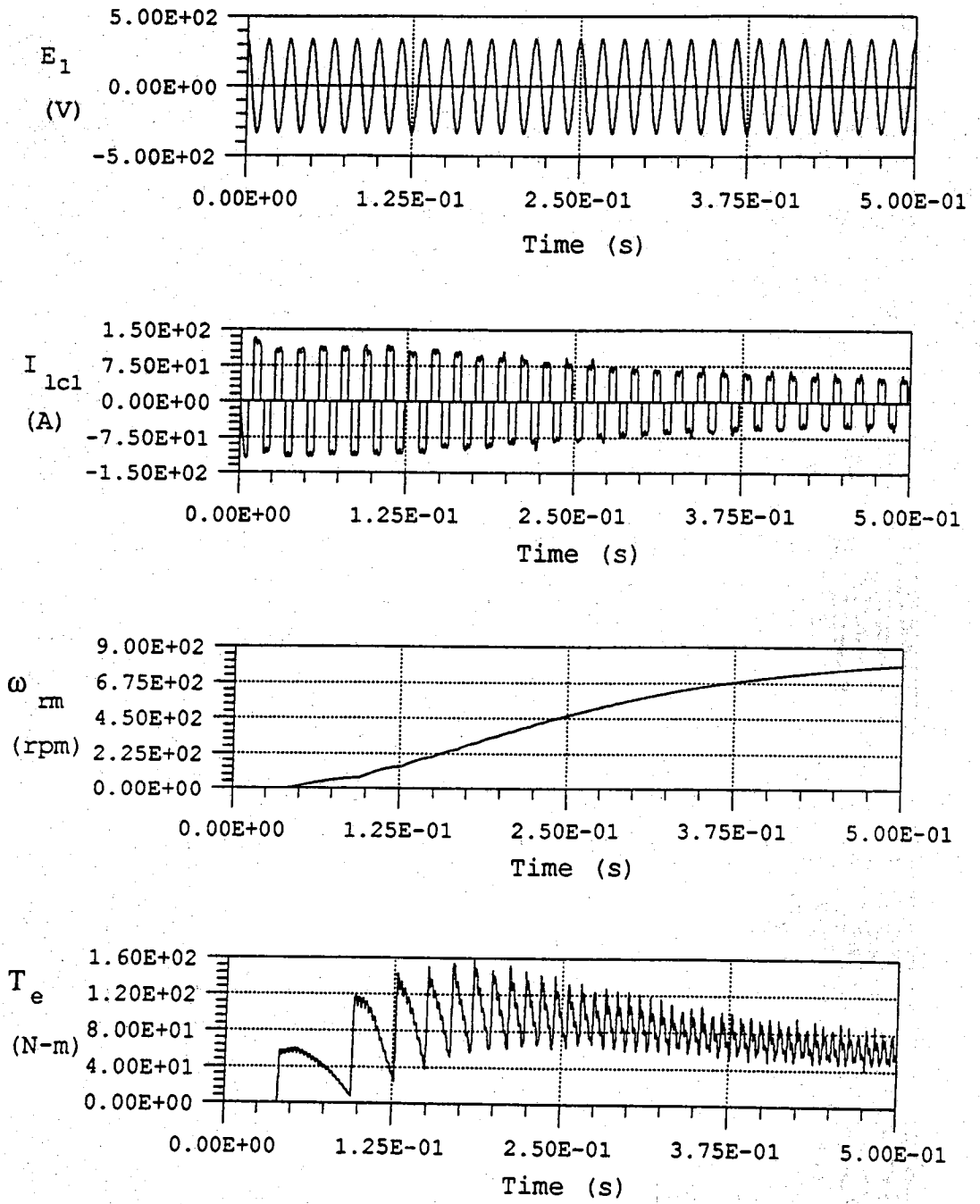


Figure 5.11. Continued.

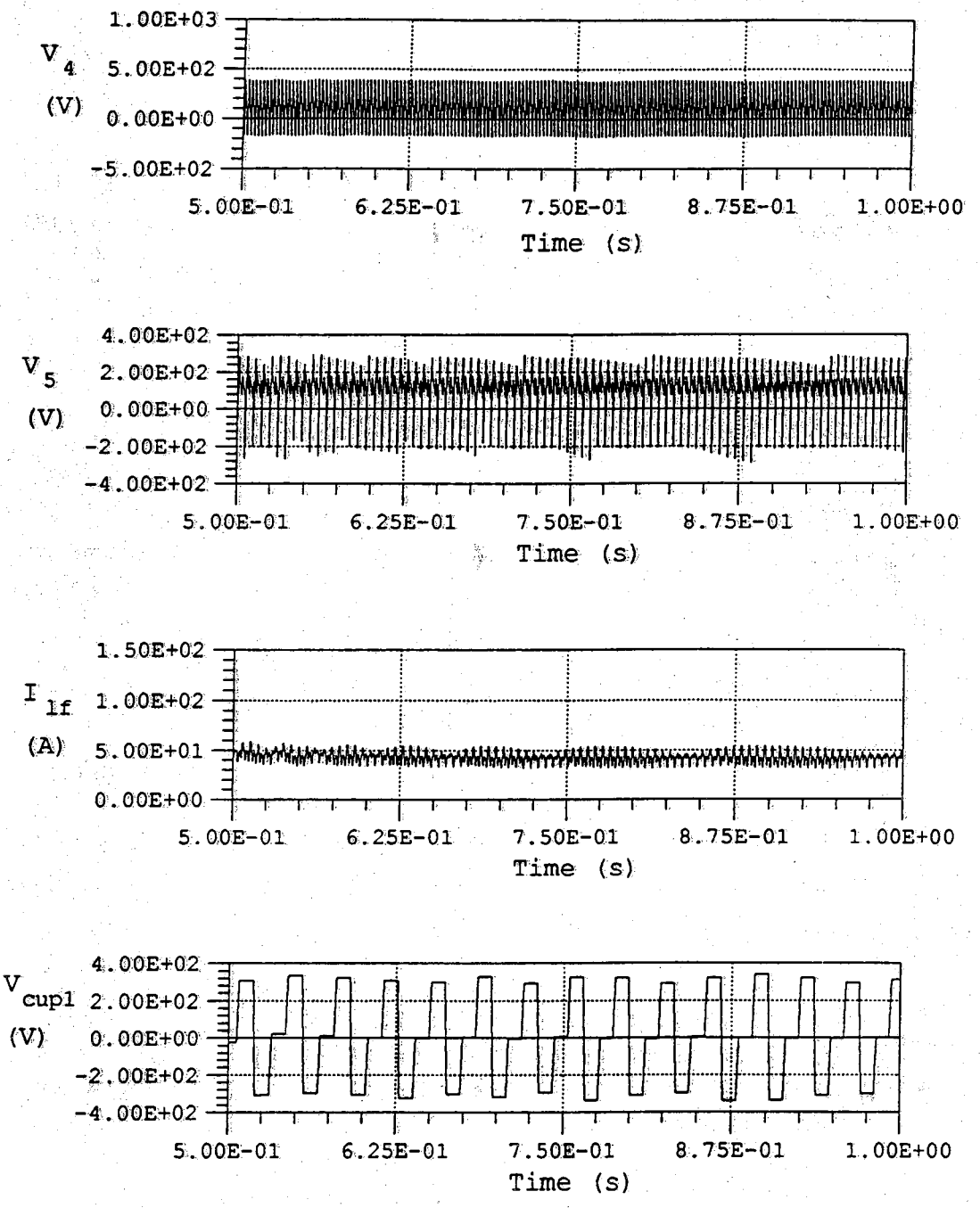


Figure 5.12. Operations of induction motor system from .5 s to 1 s.

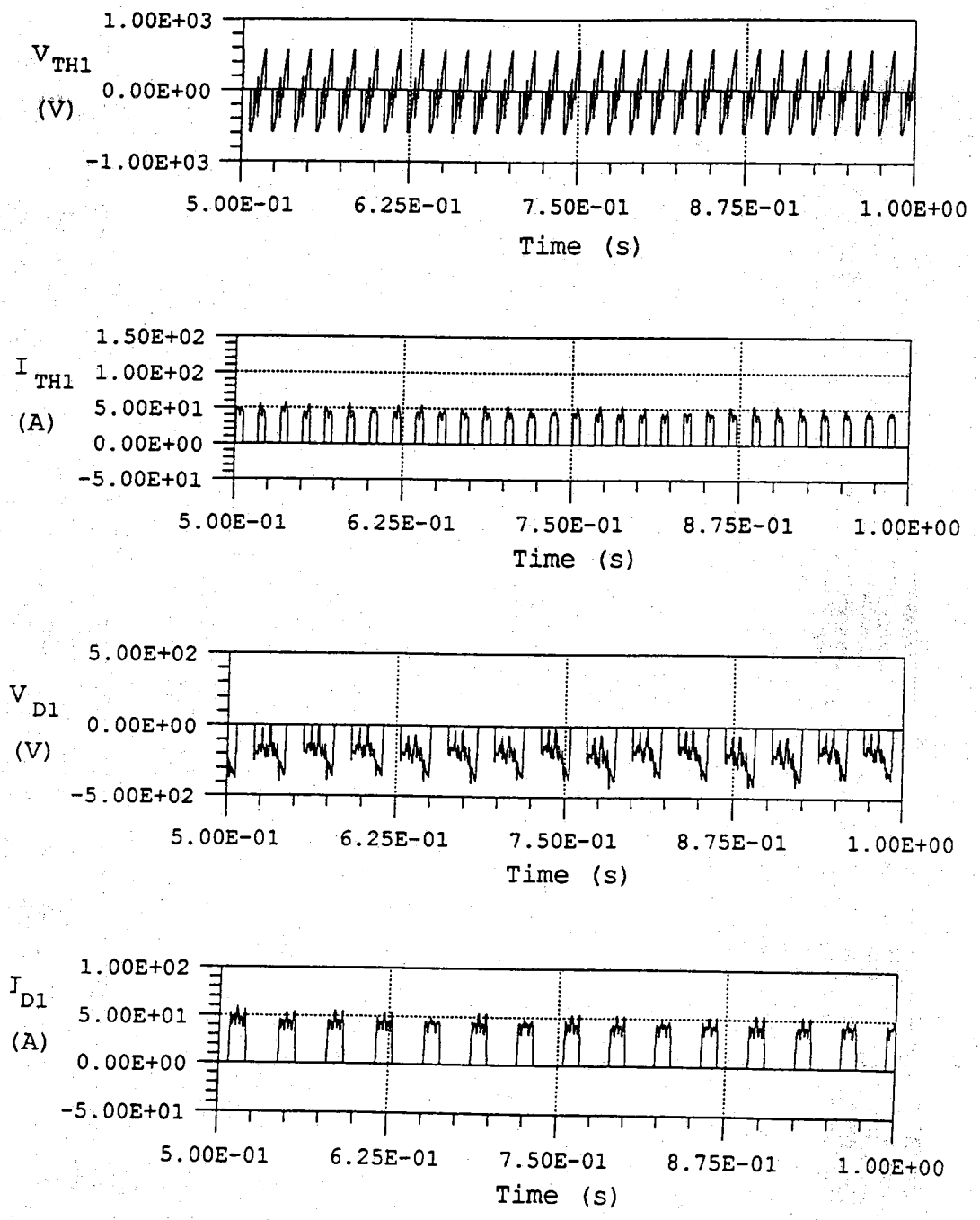


Figure 5.12. Continued.

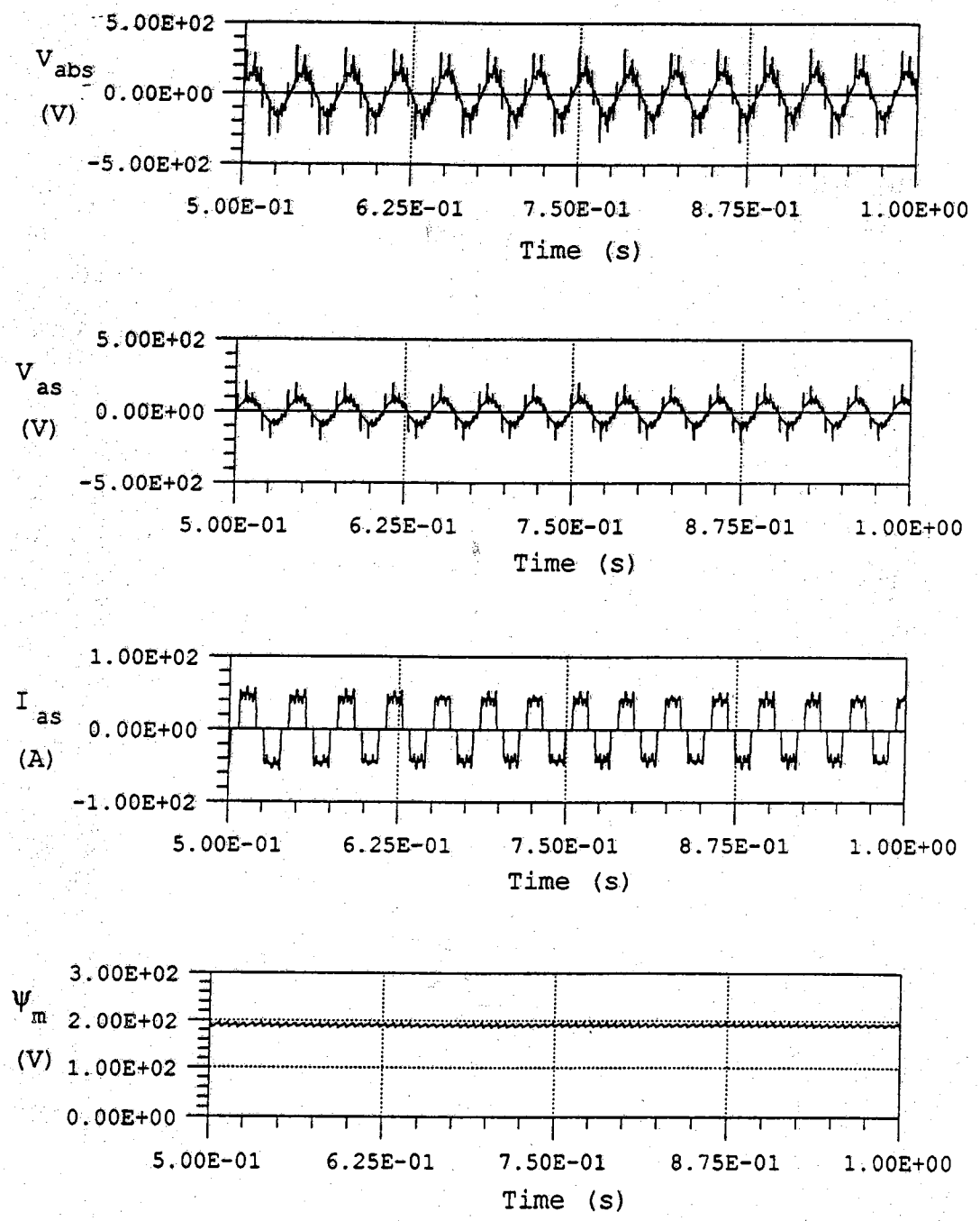


Figure 5.12. Continued.

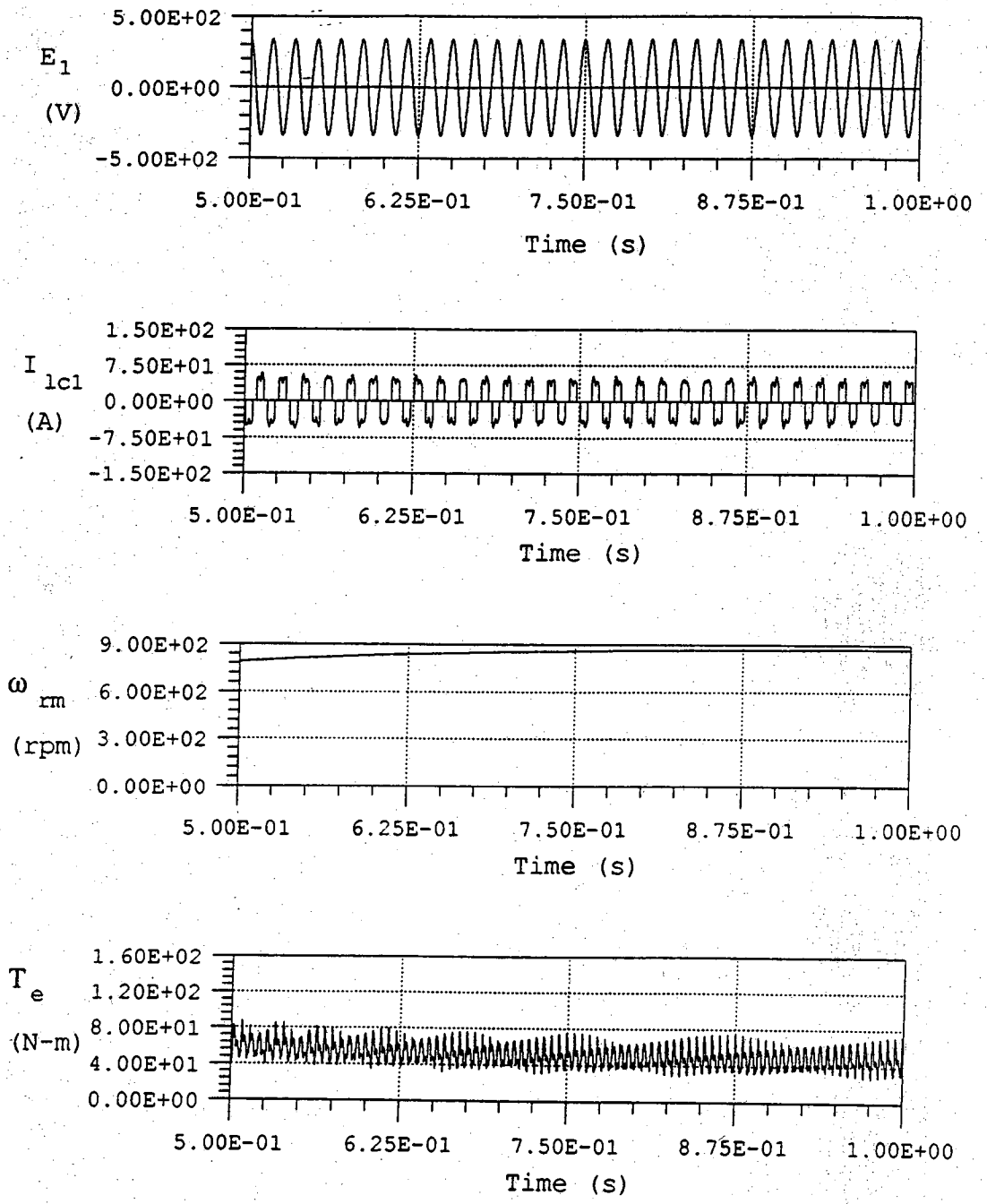


Figure 5.12. Continued.

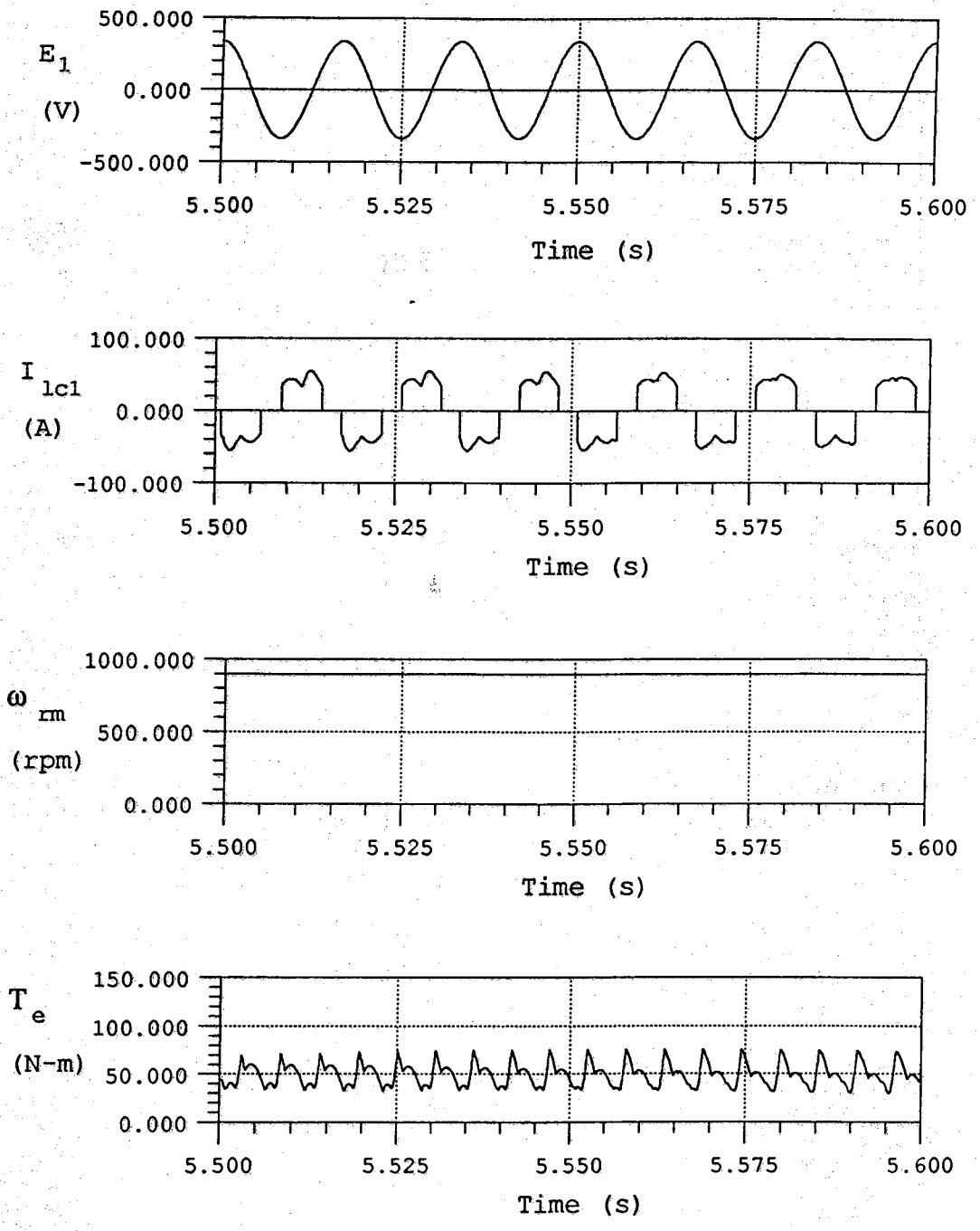


Figure 5.13. Operations of induction motor system from 5.5 s to 5.6 s.

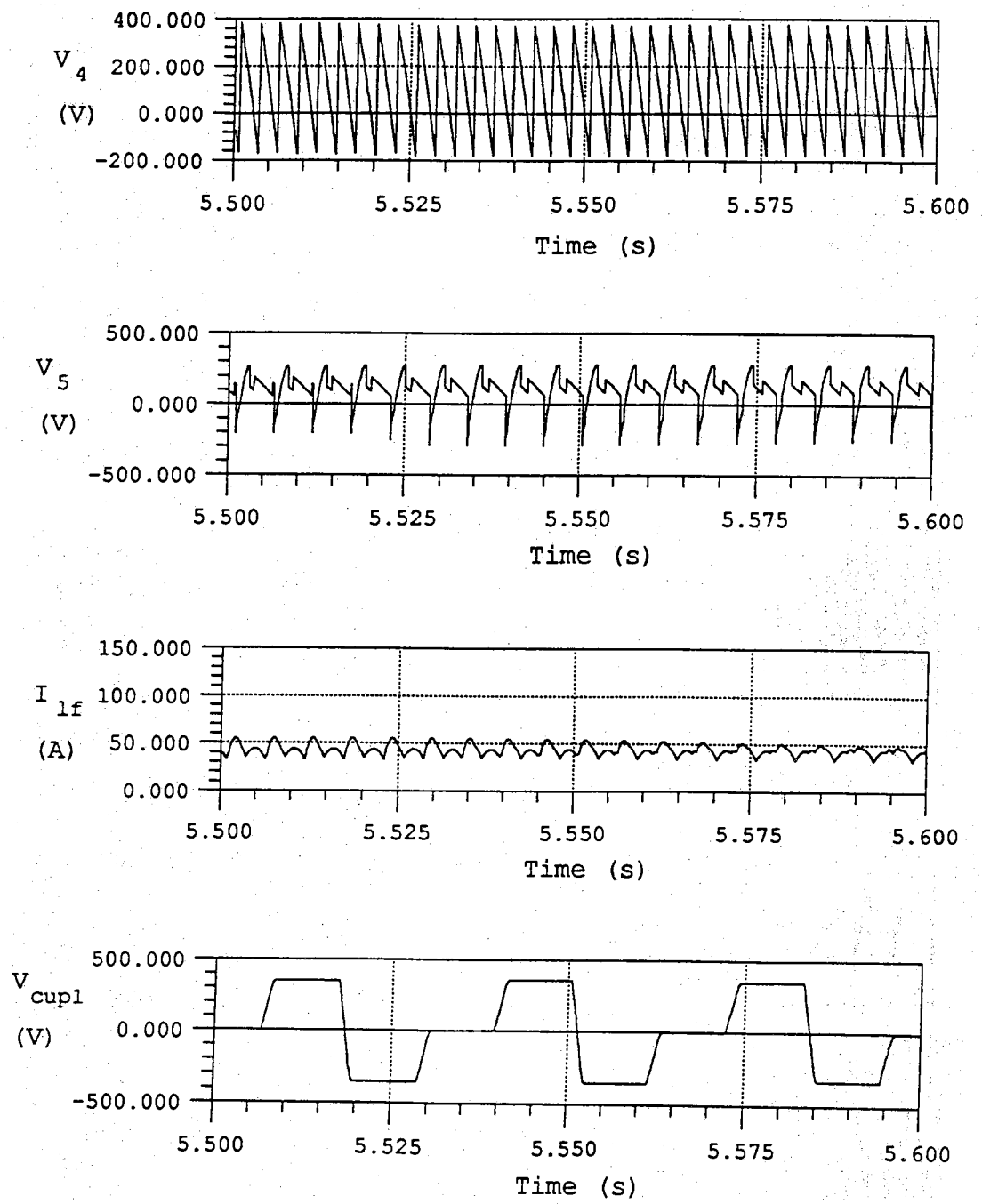


Figure 5.13. Continued.

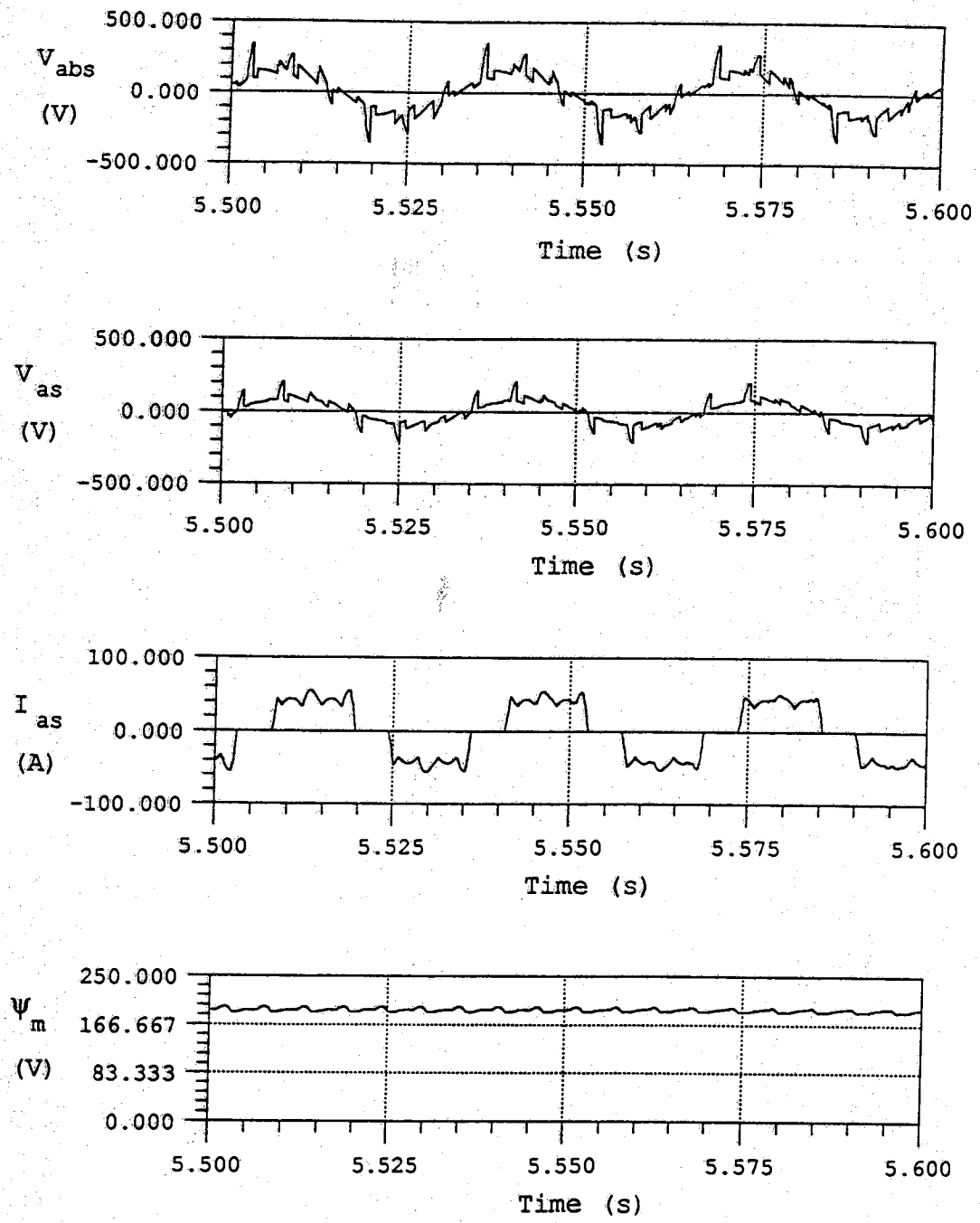


Figure 5.13. Continued.

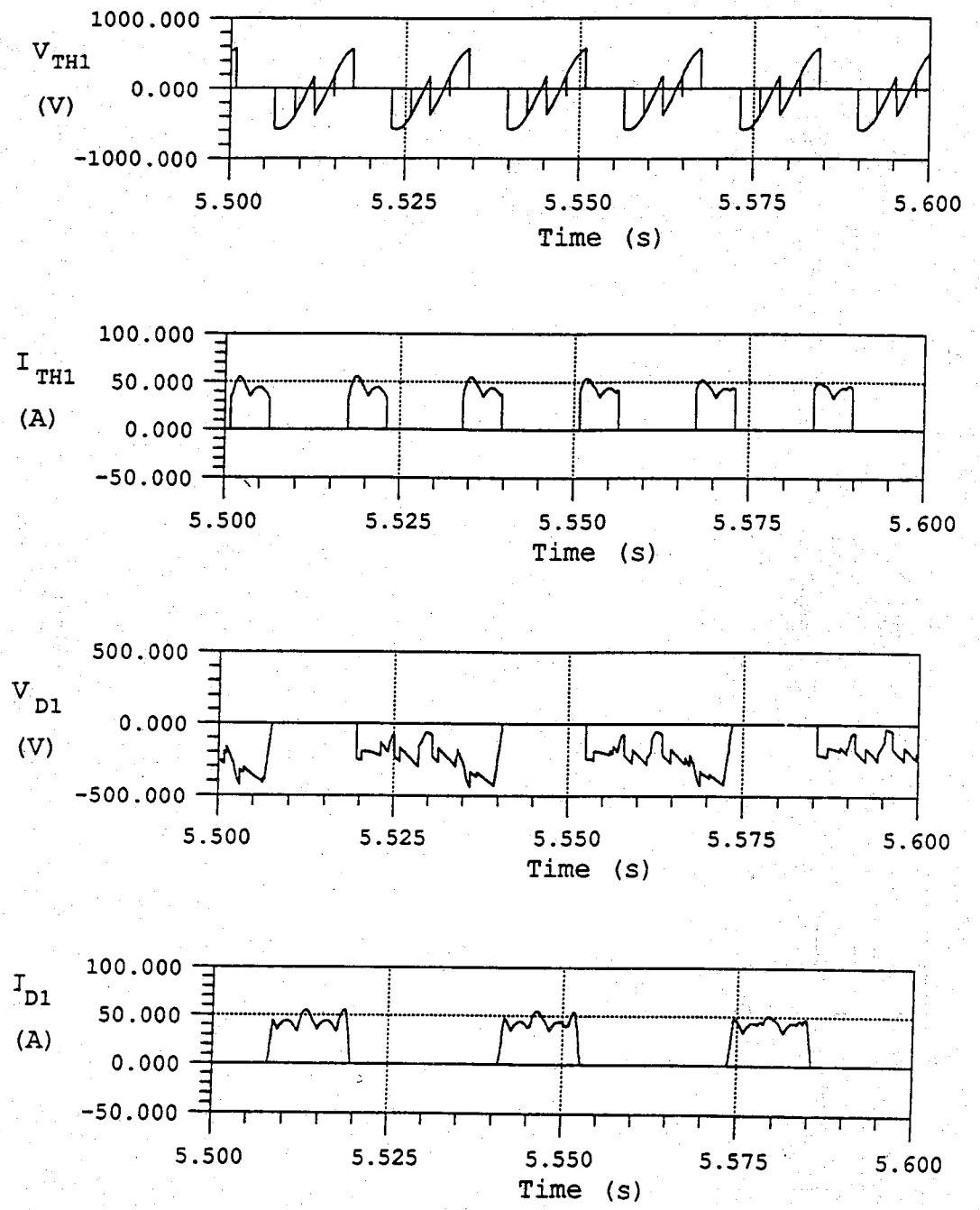


Figure 5.13. Continued.

CHAPTER 6
CONCLUSION AND RECOMMENDATIONS

6.1. Conclusion

In this research a study has been performed to look into the modelling of components used in drive systems, the framework for handling components associated with discrete events, the data structures for sparse matrices, and numerical techniques for integration and LU decomposition, for an accurate and efficient simulation of any drive system on a digital computer. The research results in several contributions in the area of transient simulation. The first contribution of this research is in a new scheme for local truncation error control that selects the step size such that the Newton-Raphson algorithm converges in one iteration on most occasions. With the usual local truncation error control scheme, the Newton-Raphson algorithm usually needs two iterations: one for correcting the predictor values and the other to stop the iteration process. The new scheme selects the step size that makes the differences between the predictor values and the values obtained from the first iteration fall within the bounds for stopping the Newton-Raphson iterative process. Although this scheme generates slightly more output points than the usual scheme does, the new scheme can reduce the number of calls to the routine containing system equations, since each call to the routine is used effectively to advance time in the simulation - not wasted in the stopping criterion of the Newton-Raphson algorithm. The new scheme is also more reliable than the usual scheme because the new scheme uses a smaller step size than does the usual scheme with the same stopping criterion for the Newton-Raphson algorithm. By choosing appropriate error control parameters in the scheme, it is possible to avoid too small a step size. The computations involved in the new scheme are also fewer than those of the usual scheme.

The second contribution is in the use of state machines to represent the components associated with discrete events. Although the idea of state machines has been used to simulate the behavior of switches, the use of state machines in TARDIS is extensively applied to all discrete components - not just switches. Using the state machines to simulate the behavior of discrete components simplifies the modelling of discrete components, and also contributes to better convergence characteristic for the Newton-Raphson algorithm when solving equations involving discrete parameters, because the state machines will allow those parameters to be changed only after the convergence of iteration process is achieved, not during the iteration process.

Keeping the system equations in a separate routine from that containing the state machines improves the speed, because, when the states of the state machines are to be changed, TARDIS does not have to go through the system equations. Yet changes in the state machines are effected on the system equations in the other routine by the use of variables common to both routines. In TARDIS the simultaneous changes in the state machines are simulated by updating the status of all the state machines after the program codes for all the state machines have been executed.

The third contribution is in the method of locating the zeros of switching functions and confirming the existence of zeros by integrating to the instants in question. Simple polynomial interpolation will work fine when locating a zero that occurs away from a minimum point. There is no need to use more elaborate algorithms for this kind of zero crossing. But for zeros that occur near the minimum points, information on the first derivatives is found to be helpful in locating such zeros. In any case, the existence of a zero should be confirmed by actually integrating to the instants found from the interpolation, to avoid a false location of zero.

The fourth contribution is in the data structures for the sparse Jacobian and the LU decomposition method used. Using separate data structures for the rows and columns is found to be faster than using a structure which has the row and column combined; some time is saved by not ordering the

information in the separate column structures, ordering which is unavoidable when the combined data structure is used.

Representing the tangible result of this research, TARDIS is essentially a versatile simulation program that has a variable-step, variable structure integration algorithm with root-finding capability, state machines to handle components associated with discrete events, and sparse matrix techniques. As such features have never before been incorporated into a single simulation package, TARDIS is unique in this sense.

6.2. Recommendations for Future Work

There are several areas that still need to be investigated to improve the speed and usefulness of the package: for example, the handling of components with time-delay representation such as transmission lines. Most of the time-delay models are usually based on equal step-size integration. It is possible to approximate the behavior of such devices by the chord method, but then how efficient and accurate will the simulation be? Another area to be investigated is the impact of the Adams method on this kind of simulation, because there seems to be no advantage in using the Gear method in this package when it does not skip the fast transients.

As is, TARDIS may not be able to simulate certain systems on some computers because the implementation of the Gear method using divided difference can cause a floating-point overflow in some internal variables. There are several possible ways to overcome this. Further investigation on the trade-off between efficiency and accuracy has to be done.

Several error detection methods should be added to the package to make it easier to debug the program. TARDIS skips some error detection for efficiency, but it is found that some of them should be implemented to prevent the program from crashing. Also if one is not careful, TARDIS can get stuck in an infinite loop at standstill where the state machines never stop changing their states.

Instead of using the limit functions in the modified Newton-Raphson algorithm for certain types of nonlinear functions which make the algorithm difficult to converge, one can use the state machines to model these nonlinear functions. As an example, a nonlinear function will be divided into several segments, each associated with a state in a state machine. Both ends of a segment will be extended by straight lines having the same slopes at the end points of the segment. The more segments there are, the easier the convergence of the Newton-Raphson algorithm is, and the more computations are involved. The comparison of computational efficiency of the segmentation of nonlinear function and that of the limit functions should be a very interesting topic to be investigated.

Although one can write system equations out in order to use the programs, it is a very cumbersome and error prone process. It is better to have an input language so that the user can specify the system equations in terms of modules. The language part will have to be completed to make it easier for someone else to use the package.

LIST OF REFERENCES

LIST OF REFERENCES

[1] Lipo, Thomas A., "Recent progress in the development of solid-state motor drives," IEEE Trans. Power Electronics, vol. 3, no. 2, April, 1988, pp. 105-117.

[2] Kirschen, Daniel S., Donald W. Novotny, and Warin Suwanwisoot, "Minimizing induction motor losses by excitation control in variable frequency drives," *IEEE Trans. on Ind. Appl.,* vol. IA-20, no. 5, September/October 1984, pp. 1244-1250.

[3] Lipo, T. A., "Analysis and control of torque pulsations in current fed induction motor drives," *Adjustable Speed AC Drive Systems*, Bimal K. Bose, Ed., IEEE Press, New York, 1980, pp. 244-251.

[4] Patel, Hasmukh S. and Richard G. Hoft, "Generalized techniques of harmonic elimination and voltage control in thyristor inverters: part i - harmonic elimination," *Adjustable Speed AC Drive Systems*, Bimal K. Bose, Ed., IEEE Press, New York, 1980, pp. 110-117.

[5] Mitchell, Edward E., "Advanced continuous simulation language (ACSL): an update," *IMACS World Congress on System Simulation and Scientific Computation,* Montreal, Canada, vol. 1, August, 1982, pp. 462-464.

[6] *Advanced Continuous Simulation Language (ACSL) Reference Manual*, Mitchell and Gauthierr Associates, Concord, Mass., 1987.

[7] Crosbie, Roy E. and J. L. Hay, "Description and processing of discontinuities with the ESL simulation language," *Proceedings of the Conference on Continuous System Simulation Languages,* Ed. Francois E. Cellier, San Diego, Society for Computer Simulation, California, 1986, pp. 30-35.

[8] Ummel, Brian R., "Simplified modeling of discontinuous phenomena using EASY5 switch states," *Proceedings of the 1986 Summer Computer Simulation Conference,* Eds. Roy Crosbie and Paul Luker, July, 1986, pp. 99-104.

[9] Gear, C. W., "Efficient step size control for output and discontinuity," *Trans. of the Society for Computer Simulation,* vol. 1, no. 1, 1984, pp. 27-31.

[10] Halin, H.J. and H. Benz, "Continuous-system simulation with PSCSP a new simulation program based upon semianalysis methods," *IMACS World Congress on System Simulation and Scientific Computation,* Montreal, Canada, vol. 1, August, 1982, pp. 358-360.

[11] Nagel, Laurence W. SPICE2: *A Computer Program to Simulate Semiconductor Circuits,* Memorandum No. UCB/ERL M520, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA, May 9, 1975, pp. 160-233.

[12] Keyhani, A. and H. Tsai, "IGSPICE simulation of induction machines with saturable inductances," *IEEE PES Summer Meeting*, 1988.

[13] *Electromagnetic Transients Program (EMTP) Application Guide,* S. F. Mauser and T. E. McDermott, Principal Investigators, EL-4650, Research Project 2149-1, Westinghouse Company.

[14] Rajagopalan, Venkatachari, *Computer-Aided Analysis of Power Electronic Systems,* Dekker, 1987.

[15] Alvarado, F. L., R. H. Lasseter, and Y. Liu, "An integrated engineering simulation environment," *Power Industry Computer Application Conference Record,* 1987, pp. 213-221.

[16] Alvarado, Fernando L. and Yenren Liu, "General purpose symbolic simulation tools for electric networks," *Power Industry Computer Application Conference Record*, 1987, pp. 222-229.

[17] Smith, David W., Scott A. Majdecki, and Doug Johnson, "Interactive control of analog system simulation," *VLSI Systems Design,* July, 1987, pp. 46-54.

[18] *MATH/LIBRARY FORTRAN subroutines for Mathematical Applications,* IMSL Inc., U.S.A., 1987, pp. 640-651.

[19] Hindmarsh, Alan C., "Toward a systematized collection of ODE solvers," *IMACS World Congress on System Simulation and Scientific Computation,* Montreal, Canada, vol. 1, August, 1982, pp. 427-429.

[20] Petzold, Linda, "A Description of DASSL: A differential/algebraic system solver,", *IMACS World Congress on System Simulation and Scientific Computation,* Montreal, Canada, vol. 1, August 1982, pp. 430-432.

[21] Forsythe, George, Michael A. Malcolm, and Cleve B. Moler, *Computer Methods for Mathematical Computations,* Prentice-Hall, Englewood Cliffs, NJ, 1977, pp. 161-166.

[22] Brent, Richard P., *Algorithms for Minimization without Derivatives,* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1973, pp. 47-60, 187-191.

[23] Runge, T. F. "A universal language for network simulation," *Numerical Methods for Differential Equations and Simulations,* Eds. A. W. Bennett and R. Vichnevetsky, IMACS, North-Holland, Amsterdam, 1978, pp. 169-175.

[24] Bordry, F. and H. Foch, "Computer-aided analysis of power-electronic systems," *IEEE Power Electronics Specialists Conference Record,* 1985, pp. 516-522.

[25] Peterson, James L., "Petri nets," *Comput. Surveys,* vol. 9, no. 3., September, 1977, pp. 223-252.

[26] Petzold, Linda, "Differential/algebraic equations are not ODE's," *SIAM J. Sci. Stat. Comput.,* vol. 3, no. 3, September, 1982, pp. 367-384.

[27] Mattsson, Sven Erik, "On modelling and differential/algebraic systems," *Simulation,* January, 1989, pp. 24-32.

[28] Enright, W. H., "The comparison of numerical methods for stiff ODEs," *Stiff Computation,* Richard C. Aiken, ed., Oxford University Press, New York, 1985, pp. 175-180.

[29] Chua, Leon O. and Pen-Min Lin, *Computer-Aided Analysis of Electronic Circuits,* Prentice-Hall, Englewood Cliffs, New Jersey, 1975, pp. 480-534.

[30] Vlach, Jiri and Kishore Singhal, *Computer Methods for Circuit Analysis and Design,* Van Nostrand Reinhold Company, New York 1983, pp. 364-394.

[31] Van Bokhoven, W. M. G., "Linear implicit differentiation formulas of variable step and order," *IEEE Trans. Circuits and Syst.,* vol. CAS-22, no. 2, February, 1975, pp. 109-115.

[32] Petzold, Linda and Alan C. Hindmarsh, LSODAR code from ODEPACK.

[33] Vlach, Jiri and Kishore Singhal, *Computer Methods for Circuit Analysis and Design,* Van Nostrand Reinhold Company, New York 1983, pp. 364-394.

[34] Zein, D. A., C. W. Ho, and A. J. Gruodis, "A new interactive circuit design program in APL," *International Symposium on Circuits and Systems,* 1980, pp. 913-917.

[35] Brayton, Robert K., Fred G. Gustavson and Gary D. Hachtel, "New efficient algorithm for solving differential-algebraic systems using implicit backward differentiation formulas," *Proc. IEEE,* vol. 60, no. 1, Jan. 1972, pp. 98-108.

[36] Hindmarsh, Alan, "The ODEPACK solvers," *Stiff Computation,* Richard C. Aiken, ed., Oxford University Press, New York, 1985, pp. 167-174.

[37] Gear, C. W., "The automatic integration of ordinary differential equations," *Commun. ACM.,* Ed. W. P. Timlake, vol. 14, no. 3, March, 1971, pp. 176-179.

[38] Carver, M. B. and S. R. MacEwen, "Numerical analysis of a system described by implicitly-defined ordinary differential equations containing numerous discontinuities," *Appl. Math. Modelling,* vol. 2, December, 1978, pp. 280-286.

[39] Carver, M. B., "Efficient integration over discontinuities in ordinary differential equations," *Numerical Methods for Differential Equations and Simulation,* Ed. A. W. Jennett and R. Vichnevetsky, *1978 IMACS,* North-Holland Publishing Company, pp. 51-56.

[40] Ellison, D., "Efficient automatic integration of ordinary differential equations with discontinuities," *Math. and Comp. in Simulation,* vol. XXIII, 1981, pp. 12-20.

[41] Birta, L. G., T. I. Ören, and D. L. Kettenis, "A robust procedure for discontinuity handling in continuous system simulation," *Trans. of the Society for Computer Simulation,* vol. 2, no. 3, 1985, pp. 189-205.

[42] Rice, John R., *Numerical Methods, Software, and Analysis: IMSL Reference Edition,* McGraw-Hill Book Company, New York, 1983. pp. 226-227.

[43] Stoer, J and R. Bulirsch, *Introduction to Numerical Analysis,* Springer-Verlag, New York, 1980, p. 267.

[44] Nagel, Laurence W., *SPICE2: A Computer Program to Simulate Semiconductor Circuits,* Memorandum No. UCB/ERL M520, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA, May 9, 1975, pp. 88-90.

[45] Stoer, J and R. Bulirsch, *Introduction to Numerical Analysis,* Springer-Verlag, New York, 1980, pp. 159-168.

[46] Duff, I. S., A. M. Erisman, and J. K. Reid, *Direct Methods for Sparse Matrices,* Oxford UP, Oxford, 1989, pp. 46-53.

[47] Duff, I. S., A. M. Erisman, and J. K. Reid, *Direct Methods for Sparse Matrices,* Oxford UP, Oxford, 1989, pp. 195-197.

[48] Markowitz, Harry, "The elimination form of the inverse and its application to linear programming," *Management Science, Journal of the Institute of Management Sciences,* vol. 3, no. 3, April, 1957, pp. 255-269.

[49] Chua, Leon O. and P. M. Lin, *Computer Aided Analysis of Electronic Circuits: Algorithms & Computational Techniques,* Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1975, pp. 181-185.

[50] Higham, Nicholas and Desmond J. Higham, "Large growth factors in Gaussian elimination with pivoting," *SIAM J. Matrix Anal. Appl.,* vol. 10, no. 2, April, 1989, pp. 155-164.

[51] Skeel, Robert D., "Effect of equillibration on residual size for partial pivoting," *SIAM J. Numer. Anal.,* vol. 18, no. 3, June, 1981, pp. 449-454.

[52] Horowitz, Ellis and Sartaj Sahni, *Fundamentals of Data Structures,* Computer Science Press, Inc., Maryland, 1982, pp. 134-140.

[53] Nagel, Laurence W., *SPICE2: A Computer Program to Simulate Semiconductor Circuits,* Memorandum No. UCB/ERL M520, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA, May 9, 1975, pp. 97-104.

[54] Duff, I. S., A. M. Erisman, and J. K. Reid, *Direct Methods for Sparse Matrices,* Oxford UP, Oxford, 1989, p. 135.

[55] Duff, I. S., A. M. Erisman, and J. K. Reid, *Direct Methods for Sparse Matrices,* Oxford UP, Oxford, 1989, pp. 178-183.

[56] Osterby, Ole and Zahari Zlatev, *Direct Methods for Sparse Matrices,* Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1983, pp. 48-49.

[57] Ong, C. M., C. T. Liu, and C. N. Lu, "Generation of connection matrices for digital computer simulation of converter circuits using the tensor approach," *IEEE Trans. Power Systems*, vol. PWRS-2, no. 4, November, 1987, pp. 906-912.

[58] Krause, Paul C., *Analysis of Electric Machiney,* McGraw-Hill Book Company, 1986, pp. 101-102.

[59] Cornell, Edward P. and Thomas A. Lipo, "Modeling and design of controlled current induction motor drive systems," *Adjustable Speed AC Drive Systems,* Ed. Bimal K. Bose, IEEE Press, New York, 1980, pp. 234-243.

[60] Scheid, Francis, *Theory and Problem of Numerical Analysis, Schaum's outline Series in Mathematics,* McGraw-Hill Book Company, New York, 1968, pp. 58-59.

[62] Stoer, J. and R. Bulirsch, *Introduction to Numerical Analysis,* Translated by R. Bartels, W. Gautschi, and C. Witzgall, Springer-Verlag, New York, 1983, pp. 43-49.

[63] Kunz, Kaiser S., *Numerical Analysis,* McGraw-Hill Book Company, Inc., 1957, p. 101.

APPENDICES

APPENDICES

Appendix A - Newton's Divided Differences

The following describes the basics of the divided differences [60, 61] which are used in the Gear algorithm as described in Chapter 3. A first divided difference is defined by

$$y[t_i, t_{i-1}] = \frac{y[t_i] - y[t_{i-1}]}{t_i - t_{i-1}} \tag{3.1}$$

The second order divided difference is then

$$y[t_i, t_{i-1}, t_{i-2}] = \frac{y[t_i, t_{i-1}] - y[t_{i-1}, t_{i-2}]}{t_i - t_{i-2}} \tag{3.2}$$

The n^{th} order divided difference can be written as

$$y[t_i, t_{i-1}, \ldots, t_{i-n}] = \frac{y[t_i, t_{i-1}, \ldots, t_{i-n-1}] - y[t_{i-1}, t_{i-2}, \ldots, t_{i-n}]}{t_i - t_{i-n}} \tag{3.3}$$

Note that

$$\frac{dy}{dt} = y[t_i, t_i] \tag{3.4}$$

and

$$\frac{1}{n!}\frac{d^n y}{dt^n} = y[t_i, t_i, t_i, \ldots t_i] \tag{3.5}$$

with n+1 terms of t_i in the right-hand side argument [62].

Appendix B - Source Code for GetZero()

```
#include <stdio.h>
#include <math.h>
#include <values.h>

#define  DEBUG
#undef   DEBUG

extern  double          GetZero(),  my_func();
extern  double          my_pow();

#ifdef   sun
extern  double          dbl_scalbn();
#endif

main()
{
   double a,  b,  result;
   int        count;

   a = -1.e0, b = 1.e0;
   fprintf(stderr,"Enter a and b: ");
   (void) scanf("%lf %lf", &a, &b);
    printf("%22.15e\n", result = GetZero(a,b,my_func,1.e-10,&count));
   printf("count =%d, ", count);
    printf("f =%22.15e\n", my_func(result));
}
double  my_func(x)
double   x;
{
   double rvalue;
/*
   double n = 5.e0;
   double a,  b;

   a = 0.e0;
   b = 1.e-4;
   rvalue = pow(x,n) + b;
   rvalue = exp(40.e0*x-27.631e0) - 1.e-12 - 1.e-1;
   rvalue = x >= 0.e0? pow(x,5.e-1): -pow(-x,5.e-3);
   rvalue = my_pow(x,7) + 28.e0*my_pow(x,4) - 480.e0;
*/
   rvalue = pow(x,9.e0);
   return rvalue;
}
double  my_pow(x,n)
double   x;
int      n;
{
   double return_value = x;

  while (--n)
     return_value *= x;
  return return_value;
```

```
}
double  GetZero(a,c,f,tol,cnt_addr)
double   a, c, (*f)(), tol;
int      *cnt_addr;
{
   double fa, fb, fc, fd, a_b, b_c, a_c, a_d, b_d, fa_fb, fb_fc, fa_fc;
   double dvdf10, dvdf11, dvdf2, dtmp, mag_a_b;
   double c_d, b, d, slope_inv, old_a_b1, old_a_b2;
   double eps4, eta2;
   int    sign_fa, sign_fb, sign_fc, sign_fd, invq_cnt;
   double sc_fa, sc_fb, sc_fc;

   *cnt_addr = 0;
   eps4 = 4.4e-16; /* Should be set according to the hardware. */
   eta2 = 2.2e-16; /* 2 * eps4 */
   /*
    * Initial checking.
    */
   if ((*cnt_addr)++, !(fa = (*f)(a)))
     return a;
   if ((*cnt_addr)++, !(fc = (*f)(c)))
    return c;
   if ((sign_fa = signbit(fa)) == (sign_fc = signbit(fc)))
   {
      printf("Error: fa and fb cannot have the same sign.\n");
      printf("fa = %22.15e, fb = %22.15e\n", fa, fc);
      return MAXDOUBLE;
   }
   if (fabs(fa) > fabs(fc))    /* Swap a and c if |fa| > |fc|. */
   {
     dtmp = a, a = c, c = dtmp;
     dtmp = fa, fa = fc, fc = dtmp;
      sign_fa = !sign_fa, sign_fc = !sign_fc;
   }
   a_c = a - c;                          /* |fa| < |fc|. */
   tol = eps4 * fabs(a) + eta2;
   if (fabs(a_c) < tol)
     return a;
   /*
    * Start with linear interpolation.
    */
   sc_fa = fa / fc;
   fa_fc = sc_fa - 1.e0;
   slope_inv = a_c / fa_fc;
#ifdef   DEBUG
   printf("Linear interpolation.   |a-c| = %22.15e\n", fabs(a_c));
   printf("a =%22.15e, c =%22.15e\n", a, c);
   printf("fa=%22.15e, fc=%22.15e\nslope_inv %22.15e\n", fa, fc, slope_inv);
#endif
   a_b = sc_fa * slope_inv;
   if (fabs(a_b) < tol)
      a_b = copysign(tol,a_c);
   b = a - a_b;
   if ((*cnt_addr)++, !(fb = (*f)(b)))
     return b;
   sign_fb = signbit(fb);
   b_c = b - c;
   if (sign_fb != sign_fc)     /* Is sign(fb) != sign(fc)? */
   {                      /* Switch a and c. */
```

```
      dtmp = a, a = c, c = dtmp;
      dtmp = fa, fa = fc, fc = dtmp;
       sign_fa = !sign_fa, sign_fc = !sign_fc;
      a_c = -a_c;
      dtmp = a_b;
      a_b = -b_c;
      b_c = -dtmp;
    }
    /*
     * Big loop; switch between inverse quadratic and bisection methods.
     * Knowns are: a_b, a_c.
     */
    invq_cnt = 0;
    while (fb && ((mag_a_b = fabs(a_b)) > (tol = eps4*fabs(b) + eta2)))
    {
#ifdef   DEBUG
       printf("After processed.   |a-b| = %22.15e, tol = %22.15e\n", mag_a_b,
tol);
      printf("b =%22.15e, fb =%22.15e\n", b, fb);
       printf("a_b=%22.15e, a_c=%22.15e, b_c=%22.15e\n\n", a_b, a_c, b_c);
#endif
     /* a b-> c
      */
      if ((fabs(fb) > fabs(fc)) || (mag_a_b < ldexp(tol,1)))
        goto bisect;
      if (fabs(fc) > fabs(fa))
     {
        sc_fa = fa / fc;
        sc_fb = fb / fc;
        sc_fc = 1.e0;
     }
     else
     {
        sc_fa = 1.e0;
        sc_fb = fb / fa;
        sc_fc = fc / fa;
     }
      fa_fc = sc_fa - sc_fc;
      slope_inv = a_c / fa_fc;
      if ((fa_fb = sc_fa - sc_fb) && (fb_fc = sc_fb - sc_fc))
     {
        dvdf10 = a_b / fa_fb;
        dvdf11 = b_c / fb_fc;
        dtmp = dvdf10 - dvdf11;
#ifdef   DEBUG
        printf("dtmp = %22.15e\n\n", dtmp);
#endif
        if (fabs(dtmp) < fabs(slope_inv))
       {
         /*
           * Inverse quadratic with a <-d-> b-> c.
           */
          switch (invq_cnt)
         {
         case 2:
             if (mag_a_b > ldexp(old_a_b2,-1))
              goto bisect;
         case 1:
            old_a_b2 = old_a_b1;
```

```
            case 0:
               old_a_b1 = mag_a_b;
               if (invq_cnt < 2)
                 invq_cnt++;
            }
#ifdef   DEBUG
              printf("Inverse quadratic.   |a-b| = %22.15e\n", mag_a_b);
             printf("a =%22.15e, b =%22.15e, c =%22.15e\n", a, b, c);
              printf("fa=%22.15e, fb=%22.15e, fc=%22.15e\nslope_inv %22.15e\n",
               fa, fb, fc, slope_inv);
#endif
            dvdf2 = dtmp / fa_fc;
             b_d = sc_fb*(dvdf10 - sc_fa*dvdf2);
            if (fabs(b_d) < tol)
           {
               b_d = copysign(tol,-a_b);
             d = b - b_d;
             a_d = a - d;
           }
           else
           {
              d = b - b_d;
               if (fabs(a_d = a - d) < tol)
             {
                 a_d = copysign(tol,a_b);
               d = a - a_d;
               b_d = b - d;
             }
           }
             fd = (*f)(d); (*cnt_addr)++;
#ifdef   DEBUG
             printf("d = %22.15e, fd = %22.15e\n", d, fd);
             printf("a-d = %22.15e, b-d = %22.15e\n", a_d, b_d);
#endif
            if (!(fd))
              return d;
            sign_fd = signbit(fd);
            if (sign_fd == sign_fc)
           {
#ifdef   DEBUG
              printf("a d-> b-> c\n");
              printf("a b-> c     <== replace\n");
#endif
             c = b; fc = fb;
             b = d; fb = fd;
              a_c = a_b; a_b = a_d; b_c = -b_d;
           }
           else
           {
#ifdef   DEBUG
              printf("a <-d b-> c\n");
#endif
              if (mag_a_b > fabs(c_d = c - d))
             {
#ifdef   DEBUG
                 printf("     a b-> c<== replace\n");
#endif
               a = d; fa = fd;
                a_b = -b_d; a_c = -c_d;
```

```
            }
            else
            {
#ifdef   DEBUG
                printf("c <-b a     <== replace\n");
#endif
                c = a; fc = fa; sign_fc = sign_fa;
                a = b; fa = fb; sign_fa = sign_fb;
                b = d; fb = fd; sign_fb = sign_fd;
                a_c = -a_b; b_c = -a_d; a_b = b_d;
            }
          }
           continue;
        }
      }
       /* Bisection.
       */
bisect:
#ifdef   DEBUG
       printf("Bisection.   |a-b| = %22.15e\n", mag_a_b);
       printf("a =%22.15e, b =%22.15e, c =%22.15e\n", a, b, c);
         printf("fa=%22.15e,  fb=%22.15e,  fc=%22.15e\nslope_inv  %22.15e\n",  fa,
fb,
          fc,  slope_inv);
#endif
      invq_cnt = 0;
     c = b; fc = fb;
#ifdef   DEBUG
       printf("After shifting in bisection.\n");
       printf("a =%22.15e, c =%22.15e\n", a, c);
       printf("fa=%22.15e,  fc=%22.15e\n", fa, fc);
#endif
     a_c = a - c;
      a_b = b_c = ldexp(a_c,-1);
     b = c + b_c;
      fb = (*f)(b); (*cnt_addr)++; sign_fb = signbit(fb);
      if (sign_fb != sign_fc)   /* a <- b c */
     {                          /* c <- b a<== replace */
       dtmp = a, a = c, c = dtmp;
       dtmp = fa, fa = fc, fc = dtmp;
        sign_fa = !sign_fa, sign_fc = !sign_fc;
       a_c = -a_c;
       a_b = -b_c; /* |a-b| == |b-c| */
       b_c = a_b;
     }
   }       /* End big loop. */
#ifdef   DEBUG
    printf("After  processed.    |a-b| = %22.15e,  tol  =  %22.15e\n",  mag_a_b,
tol);
   printf("b =%22.15e,  fb =%22.15e\n", b,  fb);
    printf("a_b=%22.15e,  a_c=%22.15e,  b_c=%22.15e\n\n", a_b,  a_c,  b_c);
#endif
   if (fabs(fa) > fabs(fb))
     return b;
   return a;
}
```

Appendix C - "mainsys.c" file for test circuit 1

```
/*****************************************************************************
  * This is a simple R-L circuit with a diode using a state machine.
  *****************************************************************************/
#include "compdep.h"
#include <math.h>
#include <stdio.h>

#include "alloc.h"
#include "dbl.h"
#include "default.h"
#include "disc.h"
#include "intg.h"
#include "machdep.h"
#include "mainsys.h"
#include "msg.h"

#define  DEBUG_event
#undef   DEBUG_event
#define  ResetIntgOrd          (rs_order = 1)
#define  RecomputeJac          (comp_jac = 1)
#define  ResetEventOrd(x)      ((x)->order = 0)

#define  MONITOR(ste_ptr,expr,tol)  monitor_ste(ste_ptr,expr,tol)
#define  SCHEDULE(sch_ptr,expr)          schedule(sch_ptr,expr,t)

#define  SCH_ALLOC()    (ECALLOC(1,SCH_t))

#define  STATE_INIT     -1    /* Initially all state will be set to -1. */

#define  L         1.e0
#define  R         1.e0
#define  Vm        5.e0

#define  ON              1     /* States of a diode. */
#define  OFF             0

#define  DIODE           0     /* State machine number. */

#define  I23             yst[0]
#define  dI23            dyst[0]

#define  Rd        dbl_parm[0]

#define  I230            0.e0  /* Initial condition. */
#define  tol             1.e-10
#define  ron             1.e-4
#define  roff            1.e7

static  char        *local_fname = "mainsys.c";

static  double           dbl_interm[1],
                         dbl_parm[1];

static  double      pi2;
static  FILE        *outfile2;
```

```
int                 numst = 1,
                    numnst = 0,
                    numeq = 1,
                    NumStMach = 1,
                    NumLnode = 0;
long                count_mainsys = 0,
                    count_output = 0;
/*****************************************************************************
 * Diode() simulates behavior of a diode using a finite state machine.
 *****************************************************************************/
static void Diode(yst,current,resistanceAddr,initState,indexStMach,
        name,stetmpAddr)
double  *yst, current, *resistanceAddr;
int     initState, indexStMach;
char    *name;
STE_t   **stetmpAddr;
{
   switch (StMach[indexStMach])
  {
  case OFF:
     if (MONITOR(*stetmpAddr, -current, tol))
    {
        NextState(indexStMach, ON);
       RecomputeJac;
       *resistanceAddr = ron;
    }
     break;
  case ON:
     if (MONITOR(*stetmpAddr, current, tol))
    {
        NextState(indexStMach, OFF);
       RecomputeJac;
       *resistanceAddr = roff;
    }
     break;
  case STATE_INIT:
     *stetmpAddr = STE_alloc(1);
      NextState(indexStMach, initState);
     *resistanceAddr = (initState == OFF? roff: ron);
    break;
  default:
     fprintf(stderr,"No state %d for %s\n", StMach[indexStMach], name);
    break;
  }
#ifdef  DEBUG_event
   fprintf(outfile,"%s->order is %d, %s->state is %d\n",
        name, (*stetempAddr)->order, name, StMach[indexStMach]);
   fprintf(outfile,"fire_type is %d, fire_h = %g\n", (*stetempAddr)-
>fire_type,
        (*stetempAddr)->deltat);
   printvec((*stetempAddr)->data, (int) (MAX_ORD+1));
   printvec((*stetempAddr)->dvdf, (int) MAX_ORD);
#endif
}
/*****************************************************************************
 * MainSystem() calculates f given y and t.
 *****************************************************************************/
void MainSystem(f, yst, t)
double      *f;
```

```
 register  double   *yst;
 double      t;
 {
    register double *ynst,  *dyst;

    count_mainsys++;
    ynst  =  yst  +  numst;
    dyst  =  yst  +  numeq;

    f[0]  =  -  L*dI23  +  (Vm*sin(pi2*t)  -  I23*(Rd+R));
 }
 /*****************************************************************************
  * MainEvent() calculates the expression for events.
  *****************************************************************************/
 void  MainEvent(yst,  t)
 register  double   *yst;
 double      t;
 {
    { static STE_t  *tmp;

        Diode(yst,I23,&Rd,ON,DIODE,"DIODE",&tmp);
    }
 }
 /*****************************************************************************
  * usr_init() sets up initial conditions and file pointers.
  *
  * Another way to initialize data is to read data from a file and this
  * gives more flexiblity to modify initial data.  This file should provide
  * both names and their numerical values.
  *****************************************************************************/
 void  MainInit(yst,  t)
 register  double   *yst;
 double      t;
 {
    int    i;

    pi2  =  atan2(0.e0,-1.e0)  *  2.e0;
    I23  =  I230;
    TSTART  =  0.e0;
    TFINAL  =  5.e0;
    HPRINT  =  1.e-3;

    outfile2  =  fopen("out2",  "w");
    event_init();
    for  (i  =  0;  i  <  NumStMach;  i++)
       StMach[i]  =  STATE_INIT;
 }
 /*****************************************************************************
  * output() prints output at each time step of the size HPRINT.
  *****************************************************************************/
 void  output(yst,  t)
 register  double   *yst;
 double      t;
 {
    count_output++;
    fprintf(outfile2,  "%e\t%e\t%d\t%e\t%e\n",  t,  *h,  prev_ord,  Vm*sin(pi2*t),
 *yst);
 }
 /*****************************************************************************
```

```
 * DumpUserVar() dumps user's variables, StMach, Lnode, dbl_parm, and
 * dbl_interm.
 ****************************************************************************/
void  DumpUserVar(dumpStream)
FILE     *dumpStream;
{
   int    errorFlag;

   errorFlag = 0;
   /* You can write your dump routine here too.
    */
  /* MainSystem variables.
    */
   errorFlag |= DumpArray((char *) dbl_parm, (size_t) sizeof(*dbl_parm),
         (size_t) NO_OF_ELM(dbl_parm), dumpStream, "dbl_parm");
   errorFlag |= DumpArray((char *) dbl_interm, (size_t) sizeof(*dbl_interm),
         (size_t) NO_OF_ELM(dbl_interm), dumpStream, "dbl_interm");
   if (errorFlag)
     exit(1);
}
/****************************************************************************
 * RestoreUserVar() restores user's variables, StMach, Lnode, dbl_parm, and
 * dbl_interm.
 ****************************************************************************/
void  RestoreUserVar(restoreStream)
FILE     *restoreStream;
{
   int    errorFlag;

   errorFlag = 0;
   /* You can write your restore routine here too.
    */
   /* MainSystem variables.
    */
   errorFlag |= RestoreArray((char *) dbl_parm, (size_t) sizeof(*dbl_parm),
         (size_t) NO_OF_ELM(dbl_parm), restoreStream, "dbl_parm");
   errorFlag |= RestoreArray((char *) dbl_interm, (size_t)
sizeof(*dbl_interm),
         (size_t) NO_OF_ELM(dbl_interm), restoreStream, "dbl_interm");
   if (errorFlag)
     exit(1);
}
```

Appendix D - MainSystem() and MainEvent() for Test Circuit 2

Note that DumpUserVar() and RestoreUserVar() are the same as those in test circuit 1. The #define and #include statements at the beginning are also the same.

```
/*****************************************************************************
 * This system is a full-bridge rectifier with a dc motor running at
 * constant speed.
 *****************************************************************************/

#define  il yst[0]
#define  im yst[1]

#define  v1        ynst[0]
#define  v2        ynst[1]
#define  v3        ynst[2]
#define  it1       ynst[3]
#define  it2       ynst[4]
#define  it3       ynst[5]
#define  it4       ynst[6]

#define  dil       dyst[0]
#define  dim       dyst[1]

#define  rt1       dbl_parm[0]
#define  rt2       dbl_parm[1]
#define  rt3       dbl_parm[2]
#define  rt4       dbl_parm[3]

#define  pulse_width     dbl_parm[4]
#define  off_time        dbl_parm[5]
#define  tol             dbl_parm[6]
#define  Vm              dbl_parm[7]

#define  ron       1.e-3
#define  roff      1.e5

#define  il0       0.e0  /* Initial condition. */
#define  im0       0.e0

#define  Lc        .3e-3
#define  Ra        .6e0
#define  Laa       .012e0
#define  Laf       1.8e0
#define  Wr        138.e0      /* CHANGE SPEED HERE. */
#define  If        1.e0

#define  OFF             0     /* States of state machines. */
#define  ON              1
#define  TRIG            2
```

```
#define  THY1            0
#define  THY2            1
#define  THY3            2
#define  THY4            3
#define  GATE1CON        4
#define  GATE2CON        5

#define  LOGIC_NODE0     0
#define  LOGIC_NODE1     1

static  char *local_fname  =  "mainsys.c";
static  FILE *outfile2;
static  double      dbl_parm[8],
                    dbl_interm[1];
static  double      pi_120,  pi_over_6;

int                 numst  =  2,
                    numnst  =  7,
                    numeq  =  9,
                    NumStMach  =  6,
                    NumLnode  =  2;
/*****************************************************************************
 * Thyristor() simulates behavior of a thyristor using a finite state
 * machine.
 *****************************************************************************/
static  void  Thyristor(yst,current,resistanceAddr,initState,indexStMach,
          indexLnode,name,stetmpAddr)
double    *yst,  current,  *resistanceAddr;
int       initState,  indexStMach,  indexLnode;
char      *name;
STE_t     **stetmpAddr;
{
   double *ynst,  *dyst;

   ynst  =  yst  +  numst;
   dyst  =  yst  +  numeq;

    switch  (StMach[indexStMach])
   {
   case  OFF:
      if  (*((int  *)  Lnode[indexLnode].val)  ==  TRIG)
         NextState(indexStMach,  TRIG);
     break;
   case  TRIG:
      if  (MONITOR(*stetmpAddr,  -current,  tol))
     {
         NextState(indexStMach,  ON);
        RecomputeJac;
        *resistanceAddr  =  ron;
     }
      else  if  (*((int  *)  Lnode[indexLnode].val)  ==  OFF)
     {
         ResetEventOrd(*stetmpAddr);
         NextState(indexStMach,  OFF);
     }
     break;
  case  ON:
      if  (MONITOR(*stetmpAddr,  current,  tol))
     {
```

```
            NextState(indexStMach, OFF);
            RecomputeJac;
            *resistanceAddr = roff;
        }
        break;
    case STATE_INIT:
        *stetmpAddr = STE_alloc(1);
        NextState(indexStMach, initState);
        *resistanceAddr = (initState == OFF? roff: ron);
        break;
    default:
        fprintf(stderr,"No state %d for %s\n", StMach[indexStMach], name);
        break;
    }
#ifdef   DEBUG_event
    fprintf(outfile,"%s->order is %d, %s->state is %d\n",
            name, (*stetmpAddr)->order, name, StMach[indexStMach]);
    fprintf(outfile,"fire_type is %d, fire_h = %g\n", (*stetmpAddr)-
>fire_type,
            (*stetmpAddr)->deltat);
    printvec((*stetmpAddr)->data, (int) (MAX_ORD+1));
    printvec((*stetmpAddr)->dvdf, (int) MAX_ORD);
#endif
}
void MainSystem(f, yst, t)
double      *f;
register double   *yst;
double      t;
{
    double *ynst, *dyst;

    ynst = yst + numst;
    dyst = yst + numeq;
    /*
     * Calculate intermediate variables.
     */
    { /* Do KCL.
       * Don't forget to translate currents of two terminal devices into
       * single variables.
       *
       * Look into how one will define internal KCL for general multi-terminal
       * devices.
       */
      double  *Enode = f;

      *Enode++ = it1 + it2 - im;
      *Enode++ = it3 - il - it1;
      *Enode = il - it2 + it4;
    }
    /* Equations.
     * Comments on components and modules can be copyable.
     * One may define symbols for copyable comments.
     *
     * Begin with index 3.  (3 equations has been specified before.)
     */
    f[3] = v2 - v3 - Lc * dil - Vm * sin(pi_120*t);
    f[4] = v2 - v1 - rt1 * it1;
    f[5] = v3 - v1 - rt2 * it2;
    f[6] = - v2 - rt3 * it3;
```

```
   f[7] = - v3 - rt4 * it4;
   f[8] = v1 - Ra * im - Laa * dim - Laf * Wr * If;
}
/*****************************************************************************
 * MainEvent() contains all state machines.
 *
 * MainEvent() may be called repeatedly until all states are initialized
 * before integration.
 *****************************************************************************/
void MainEvent(yst, t)
register double   *yst;
double      t;
{
   double *ynst, *dyst;

   ynst = yst + numst;
   dyst = yst + numeq;
#ifdef   DEBUG_event
    fprintf(outfile,"event_h  is\n");
    printvec(event_h,MAX_ORD);
    fprintf(outfile,"event_al  is\n");
    printvec(event_al,MAX_ORD);
#endif
   { static STE_t  *tmp;

       Thyristor(yst,it1,&rt1,OFF,THY1,LOGIC_NODE0,"THY1",&tmp);
   }
   { static STE_t  *tmp;

       Thyristor(yst,it2,&rt2,OFF,THY2,LOGIC_NODE1,"THY2",&tmp);
   }
   { static STE_t  *tmp;

       Thyristor(yst,it3,&rt3,OFF,THY3,LOGIC_NODE1,"THY3",&tmp);
   }
   { static STE_t  *tmp;

       Thyristor(yst,it4,&rt4,OFF,THY4,LOGIC_NODE0,"THY4",&tmp);
   }
    { /* next_event_time, off_time and pulse_width are parameters.
      */
     static double next_time;
     static SCH_t  *tmp;
     int           state_tmp;

      switch (StMach[GATE1CON]) /* State of gate signal at Lnode 1. */
     {
     case TRIG:
        if (SCHEDULE(tmp,next_time))
      {
         state_tmp = OFF;
         NextState(GATE1CON,  state_tmp);
          NextLogic(LOGIC_NODE0, (char *) &state_tmp, GATE1CON);
         next_time  +=  off_time;
      }
      break;
     case OFF:
        if (SCHEDULE(tmp,next_time))
      {
```

```
            state_tmp = TRIG;
            NextState(GATE1CON,  state_tmp);
              NextLogic(LOGIC_NODE0, (char *) &state_tmp, GATE1CON);
            next_time  +=  pulse_width;
          }
          break;
        case STATE_INIT:
          tmp = SCH_ALLOC();
          state_tmp = TRIG;
           NextState(GATE1CON, state_tmp);
           NextLogic(LOGIC_NODE0, (char *) &state_tmp, GATE1CON);
          next_time = pulse_width;       /* Time for next event. */
          break;
        default:
           fprintf(stderr,"No state %d for %s\n", StMach[GATE1CON], "GATE1CON");
          /* Print stop at line %d in file %s at time = %e\n
          */
          break;
      }
#ifdef   DEBUG_event
    fprintf(outfile,"State of control GATE1CON is %d, Lnode[0] = %d\n",
           StMach[GATE1CON], *((int  *)  Lnode[0].val));
#endif
    }
    { /* next_event_time, off_time and pulse_width are parameters.
       */
      static double next_time;
      static SCH_t  *tmp;
      int           state_tmp;

       switch (StMach[GATE2CON]) /* State of gate signal at Lnode 2. */
     {
      case TRIG:
          if (SCHEDULE(tmp,next_time))
        {
           state_tmp = OFF;
           NextState(GATE2CON,  state_tmp);
             NextLogic(LOGIC_NODE1, (char *) &state_tmp, GATE2CON);
           next_time  +=  off_time;
        }
        break;
      case OFF:
          if (SCHEDULE(tmp,next_time))
        {
           state_tmp = TRIG;
           NextState(GATE2CON,  state_tmp);
             NextLogic(LOGIC_NODE1, (char *) &state_tmp, GATE2CON);
           next_time  +=  pulse_width;
        }
        break;
      case STATE_INIT:
        tmp = SCH_ALLOC();
        state_tmp = OFF;
         NextState(GATE2CON, state_tmp);
         NextLogic(LOGIC_NODE1, (char *) &state_tmp, GATE2CON);
        next_time = 1.e0/120.e0;       /* Time for next event. */
        break;
      default:
         fprintf(stderr,"No state %d for %s\n", StMach[GATE2CON], "GATE2CON");
```

```
            /* Print stop at line %d in file %s at time = %e\n
            */
           break;
        }
   #ifdef   DEBUG_event
      fprintf(outfile,"State of control GATE1CON is %d, Lnode[0] = %d\n",
            StMach[GATE2CON], *((int *) Lnode[1].val));
   #endif
     }
   }
   void MainInit(yst)
   double   *yst;
   {
      double *ynst, *dyst;
      double pi;
      int       i;
      short int *p_StMach;

      ynst = yst + numst;
      dyst = yst + numeq;

      TSTART = 0.e0;  /* Control parameters for integration. */
      TFINAL = 5.e-2;
      HPRINT = 1.e-4;

      outfile2 = fopen("out2", "w");

      il = il0; /* Initial value of a state variable. */

      pi = atan2(0.e0, -1.e0);
      pi_120 = 120.e0*pi;
      pi_over_6 = pi / 6.e0;
      pulse_width = 1.e0/120.e0;
      off_time = 1.e0/60.e0 - pulse_width;
      tol = 1.e-8;
      Vm = 280.e0 * sqrt(2.e0);
      /*
        * State machine initialization.
       */
      event_init();
      Lnode[0].size = sizeof(int);
      Lnode[1].size = sizeof(int);
      for (i = NumStMach, p_StMach = StMach; i--;)
         *p_StMach++ = STATE_INIT;
      for (i = 0; i < NumLnode; i++)
      {
         Lnode[i].val = ECALLOC(Lnode[i].size, char);
         NextLnode[i] = ECALLOC(Lnode[i].size, char);
      }
   }
   void output(yst,t)
   double   *yst;
   double   t;
   {
      double *ynst;

      ynst = yst + numst;

   #ifdef   DEBUG_event
```

```
    fputc('\n', outfile);
#endif
    fprintf(outfile,"%12.5e ", t);
    fprintf(outfile,"%12.5e %1d ", h[0], prev_ord);
    fprintf(outfile,"%d%d%d%d %12.5e ", StMach[0], StMach[1], StMach[2],
StMach[3], v1);
   printvec(yst, 2);
    fprintf(outfile2,"%e\t%e\t%e\t%e\t%e\n", t, Um*sin(pi_120*t), il, v1, im);
#ifdef   DEBUG_event
    fputc('\n', outfile);
    fflush(outfile);
#endif
}
```

Appendix E - MainSystem() and MainEvent() for Test Circuit 3

Note that DumpUserVar() and RestoreUserVar() are the same as those in test circuit 1. The #define and #include statements at the beginning are also the same.

```
/*************************************************************************
 * This system is from Ghazy.  It is a high-frequency converter.
 *************************************************************************/
#define  il yst[0]
#define  vc yst[1]
#define  v1 ynst[0]
#define  v2 ynst[1]
#define  v3 ynst[2]
#define  v4 ynst[3]

#define  it1      ynst[4]
#define  it2      ynst[5]
#define  it3      ynst[6]
#define  it4      ynst[7]
#define  ie ynst[8]
#define  ic ynst[9]
#define  ir ynst[10]

#define  dil      dyst[0]
#define  dvc      dyst[1]

#define  rt1      dbl_parm[0]
#define  rt2      dbl_parm[1]
#define  rt3      dbl_parm[2]
#define  rt4      dbl_parm[3]

#define  pulse_width    dbl_parm[4]
#define  tol            dbl_parm[5]
#define  f0       dbl_parm[6]
#define  fs       dbl_parm[7]

#define  ron      1.e-3
#define  roff     1.e6

#define  Em       100.e0
#define  L        24.e-6
#define  C        1.e-6
#define  Rl       .1e0
#define  Rc       .1e0
#define  R0       10.e0

#define  il0      0.e0  /* Initial condition. */
#define  vc0      0.e0

static  char *local_fname  =  "mainsys.c";
static  FILE *outfile2;

static  double     dbl_parm[8],      /* Where to initialize this parameter
array.  */
```

```
            dbl_interm[1];
int         numst  = 2,
            numnst  = 11,
            numeq  = 13,
            NumStMach  = 1,
            NumLnode  = 0;
void MainSystem(f, yst, t)
double      *f;
register double   *yst;
double      t;
{
   double *ynst, *dyst;

   ynst = yst + numst;
   dyst = yst + numeq;
   /*
    * Calculate intermediate variables.
    */

   { /* Do KCL.
      * Don't forget to translate currents of two terminal devices into
      * single variables.
      *
      * Look into how one will define internal KCL for general multi-terminal
      * devices.
      */
     double  *Enode = f;

     *Enode++ = ie + it1 + it3;
     *Enode++ = it1 - it4 - il;
     *Enode++ = il - ic - ir;
     *Enode   = it3 + ic + ir - it2;
   }
   /* Equations.
    * Comments on components and modules can be copyable.
    * One may define symbols for copyable comments.
    *
    * Begin with index 4.  (4 equations has been specified before.)
    */
   f[4] = v1 - Em; /* Voltage source. */
   f[5] = (v1 - v2) - it1 * rt1;
   f[6] = (v1 - v4) - it3 * rt3;
   f[7] = v2 - it4 * rt4;
   f[8] = v4 - it2 * rt2;
   f[9] = v2 - v3 - Rl * il - L * dil;
   f[10] = v3 - v4 - vc - ic * Rc;
   f[11] = ic - C * dvc;
   f[12] = v3 - v4 - ir * R0;
}
/*****************************************************************************
 * MainEvent() contains all state machines.
 *
 * MainEvent() may be called repeatedly until all states are initialized
 * before integration.
 *****************************************************************************/
void MainEvent(yst, t)
register double   *yst;
double      t;
{
```

```
#define  StateMach      switch

#define  ONE_TWO        1
#define  THREE_FOUR     0

   double *ynst,  *dyst;

   ynst  = yst  + numst;
   dyst  = yst  + numeq;
#ifdef   DEBUG_event
    fprintf(outfile,"event_h  is\n");
    printvec(event_h,MAX_ORD);
    fprintf(outfile,"event_al  is\n");
    printvec(event_al,MAX_ORD);
#endif
  {
     static SCH_t  *tmp;
     static double next_time;

     switch (StMach[0])
     {
     case ONE_TWO: /* Thyristors 1 and 2 are on. */
        if (SCHEDULE(tmp, next_time))
      {
          NextState(0, THREE_FOUR);
         rt1 = rt2 = roff;
         rt3 = rt4 = ron;
          next_time += pulse_width;
         RecomputeJac;
      }
       break;
     case THREE_FOUR:
       if (SCHEDULE(tmp, next_time))
      {
          NextState(0, ONE_TWO);
         rt1 = rt2 = ron;
         rt3 = rt4 = roff;
          next_time += pulse_width;
         RecomputeJac;
      }
       break;
     case STATE_INIT:
       tmp = SCH_ALLOC();
        NextState(0, ONE_TWO);
      rt1 = rt2 = ron;
      rt3 = rt4 = roff;
       next_time = pulse_width;
       RecomputeJac;
      break;
     default:
        fprintf(stderr,"No state %d for %s\n", StMach[0], "control");
      /* Print stop at line %d in file %s at time = %e\n
      */
       break;
    }
  }
}
void  MainInit(yst)
double   *yst;
```

```
{
    double *ynst,  *dyst;
    int        i;
    short int *p_StMach;

    ynst = yst + numst;
    dyst = yst + numeq;

    TSTART = 0.e0;  /* Control parameters for integration. */
    TFINAL = .5e-3;
    HPRINT = 1.e-4;

    il = il0; /* Initial value of a state variable. */
    vc = vc0;

    outfile2 = fopen("out2", "w");

    f0 = 1.e0 / (2.e0 * atan2(0.e0,-1.e0) * sqrt(L * C));
    fs = f0 / 3.e0;
    pulse_width = .5e0 / fs;
    tol = 1.e-10;
    /*
      * State machine initialization.
     */
    event_init();
    for (i = NumStMach, p_StMach = StMach; i--;)
       *p_StMach++ = STATE_INIT;
    for (i = 0; i < NumLnode; i++)
    {
       Lnode[i].val = ECALLOC(Lnode[i].size, char);
       NextLnode[i] = ECALLOC(Lnode[i].size, char);
    }
}
void output(yst,t)
double   *yst;
double   t;
{
   double *ynst;

   ynst = yst + numst;

#ifdef   DEBUG_event
   fputc('\n', outfile);
#endif
   fprintf(outfile,"%12.5e ", t);
   fprintf(outfile,"%12.5e %1d ", h[0], prev_ord);
   fprintf(outfile,"%12.5e ", v3 - v4);
   printvec(yst, 2);
   fprintf(outfile2, "%e\t%e\t%e\t%e\t%e\n", t, v2-v4, v3-v4, il, vc);
#ifdef   DEBUG_event
   fputc('\n', outfile);
   fflush(outfile);
#endif
}
```

Appendix F - MainSystem() and MainEvent() for Test Circuit 4

Note that DumpUserVar() and RestoreUserVar() are the same as those in test circuit 1. The #define and #include statements at the beginning are also the same. The routines Diodes() and Thyristors() can be taken from those in Appendices 1 and 2 respectively.

```
/*****************************************************************************
  * This system is an induction machine with current source inverter.
   ****************************************************************************/
#define  ilc1          yst[0]
#define  ilc2          yst[1]
#define  ilf           yst[2]
#define  vcup1         yst[3]
#define  vcup2         yst[4]
#define  vclow1        yst[5]
#define  vclow2        yst[6]
#define  psi_qs        yst[7]
#define  psi_ds        yst[8]
#define  psi_qr        yst[9]
#define  psi_dr        yst[10]
#define  theta         yst[11]
#define  wr            yst[12]
#define  y_inv         yst[13]
#define  y_ctrl        yst[14]
#define  angle         yst[15]

#define  ilc3          ynst[0]
#define  v1            ynst[1]
#define  v2            ynst[2]
#define  v3            ynst[3]
#define  v4            ynst[4]
#define  v5            ynst[5]
#define  v6            ynst[6]
#define  v7            ynst[7]
#define  v8            ynst[8]
#define  v9            ynst[9]
#define  v10           ynst[10]
#define  v11           ynst[11]
#define  v12           ynst[12]
#define  v13           ynst[13]
#define  v14           ynst[14]
#define  vm1           ynst[15]
#define  vm2           ynst[16]
#define  im1           ynst[17]
#define  im2           ynst[18]
#define  im3           ynst[19]
#define  iup1          ynst[20]
#define  iup2          ynst[21]
#define  iup3          ynst[22]
#define  ilow1         ynst[23]
#define  ilow2         ynst[24]
#define  ilow3         ynst[25]
```

```
#define  psi_mq          ynst[26]
#define  psi_md          ynst[27]

#define  dilc1           dyst[0]
#define  dilc2           dyst[1]
#define  dilf            dyst[2]
#define  dvcup1          dyst[3]
#define  dvcup2          dyst[4]
#define  dvclow1         dyst[5]
#define  dvclow2         dyst[6]
#define  dpsi_qs         dyst[7]
#define  dpsi_ds         dyst[8]
#define  dpsi_qr         dyst[9]
#define  dpsi_dr         dyst[10]
#define  dtheta          dyst[11]
#define  dwr             dyst[12]
#define  dy_inv          dyst[13]
#define  dy_ctrl         dyst[14]
#define  dangle          dyst[15]

#define  vm3             dbl_interm[0]
#define  vpeak           dbl_interm[1]
#define  vqs             dbl_interm[2]
#define  vds             dbl_interm[3]
#define  psi_m           dbl_interm[4]
#define  f_psi_m         dbl_interm[5]
#define  wrm             dbl_interm[6]
#define  sin_theta       dbl_interm[7]
#define  cos_theta       dbl_interm[8]
#define  sin_theta_pi6   dbl_interm[9]
#define  cos_theta_pi6   dbl_interm[10]
#define  pi120t          dbl_interm[11]
#define  T_elec          dbl_interm[12]
#define  iqs             dbl_interm[13]
#define  ids             dbl_interm[14]
#define  inverter_freq   dbl_interm[15]
#define  fire_angle      dbl_interm[16]
#define  w_error         dbl_interm[17]
#define  i_error         dbl_interm[18]
#define  iref            dbl_interm[19]
#define  iref_star       dbl_interm[20]
#define  w_slip          dbl_interm[21]
#define  T_load          dbl_interm[22]
#define  E1              dbl_interm[23]
#define  E2              dbl_interm[24]
#define  E3              dbl_interm[25]
#define  it1             dbl_interm[26]
#define  it2             dbl_interm[27]
#define  it3             dbl_interm[28]
#define  it4             dbl_interm[29]
#define  it5             dbl_interm[30]
#define  it6             dbl_interm[31]
#define  it7             dbl_interm[32]
#define  it8             dbl_interm[33]
#define  it9             dbl_interm[34]
#define  it10            dbl_interm[35]
#define  it11            dbl_interm[36]
#define  it12            dbl_interm[37]
#define  id1             dbl_interm[38]
```

```
#define  id2            dbl_interm[39]
#define  id3            dbl_interm[40]
#define  id4            dbl_interm[41]
#define  id5            dbl_interm[42]
#define  id6            dbl_interm[43]

#define  rt1            dbl_parm[0]
#define  rt2            dbl_parm[1]
#define  rt3            dbl_parm[2]
#define  rt4            dbl_parm[3]
#define  rt5            dbl_parm[4]
#define  rt6            dbl_parm[5]
#define  rt7            dbl_parm[6]
#define  rt8            dbl_parm[7]
#define  rt9            dbl_parm[8]
#define  rt10           dbl_parm[9]
#define  rt11           dbl_parm[10]
#define  rt12           dbl_parm[11]
#define  rd1            dbl_parm[12]
#define  rd2            dbl_parm[13]
#define  rd3            dbl_parm[14]
#define  rd4            dbl_parm[15]
#define  rd5            dbl_parm[16]
#define  rd6            dbl_parm[17]
#define  tol            dbl_parm[18]
#define  pulse_width    dbl_parm[19]
#define  sign_w_slip    dbl_parm[20]
#define  sign_load      dbl_parm[21]
#define  sign_inv       dbl_parm[22]
#define  tbase          dbl_parm[23]
#define  rect_next_time dbl_parm[24]
#define  inv_next_time  dbl_parm[25]

#define  ron            1.e-3
#define  roff           1.e6

#define  lc       .2e-3       /* Commutating inductor has
                               * been chosen with less than
                               * 50 times the capacity of machine. */
#define  rf       .091e0
#define  lf       14.589e-3   /* Smoothing inductor. */
#define  c        80.e-6            /* Inverter capacitor. */

#define  TAU            10.e-3      /* Current control. */
#define  CURRENT_GAIN   1.e0

#define  ilf_max        (115.18e0)
#define  ilf_min        (25.6699e0)
#define  rpm_ref        900.e0
#define  ksp            (2.e0*(ilf_max  -  ilf_min)/377.e0)
#define  kc       1.e0        /* 1.e0 */

#define rs                .0788e0          /* Induction motor parameters. */
#define xls               .2118e0
#define xm_unsat          9.23e0
#define xlr               .4628e0
#define rr                .0408e0
#define  xm_star            (1.e0/(1.e0/xm_unsat+1.e0/xls+1.e0/xlr))
#define J                 .31e0
```

```
#define pole_pair          2.e0
#define w                  wr
#define DEAD_BAND          1.e-2

#define  ilc10             0.e0  /* Initial conditions. */
#define  ilc20             0.e0
#define  ilf0              0.e0
#define  vcup10            0.e0
#define  vcup20            0.e0
#define  vclow10           0.e0
#define  vclow20           0.e0
#define  psi_qs0           0.e0
#define  psi_ds0           0.e0
#define  psi_qr0           0.e0
#define  psi_dr0           0.e0
#define  theta0            0.e0
#define  wr0               0.e0
#define  y_inv0            .1e0
#define  y_ctrl0           0.e0

         /* The followings are indices to StMach. */
#define  THY1      0     /* Thyristors indices are used as Lnode indices also.
*/
#define  THY2      1
#define  THY3      2
#define  THY4      3
#define  THY5      4
#define  THY6      5
#define  THY7      6
#define  THY8      7
#define  THY9      8
#define  THY10     9
#define  THY11     10
#define  THY12     11
#define  D1 12
#define  D2 13
#define  D3 14
#define  D4 15
#define  D5 16
#define  D6 17
#define  INVERTER  18    /* Control part. */
#define  INV_CTRL 19
#define  RECTIFIER       20
#define  RECT_CTRL       21
#define  CURRENT_LIMIT   22
#define  ANGLE_LIMIT     23
#define  ABS_VALUE       24
#define  BLOCK2          25
#define  LOAD            26

#define  OFF             0
#define  ON              1
#define  TRIG            2
#define  DOWN            3
#define  START_PULSE     4
#define  UP              5
#define  IDLE            6
#define  TRACK           7
#define  UNDER_LIMIT     0
```

```
#define  LIMIT          1
#define  NEG            1
#define  POS            2
#define  DEAD_ZONE      0
#define  ALIVE_ZONE     1

static  char        *local_fname  =  "mainsys.c";

static  double          dbl_interm[44],
                        dbl_parm[26];

static  double          sin_pi_over_6,  cos_pi_over_6,  pi_2_over_3,
                        wb,  two_over_sqrt3,  pi2,  rpm_to_we;
static  int         rect_seq_index,  inv_seq_index;
static  FILE        *timeFilePtr,  *motorFilePtr,  *controlFilePtr,
                    *deviceFilePtr,  *supplyFilePtr;

static  double      f_sat();
static  void        Thyristor(),  Diode();
int                 numst  =  16,
                    numnst  =  28,
                    numeq  =  44,
                    NumStMach  =  27,
                    NumLnode  =  12;
long                count_mainsys  =  0,
                    count_output  =  0;
void MainSystem(f,  yst,  t)    /* Main system to be simulated. */
double      *f;
register  double    *yst;
double      t;
{
   double *ynst,  *dyst,  *pf;

   ynst  =  yst  +  numst;
   dyst  =  yst  +  numeq;
   pf  =  f;
    count_mainsys++;
   /*
     *  Calculate  intermediate  variables.
    */
   {  /*  12  thyristors.
       */
     it1  =  (v1  -  v4)  /  rt1;
     it2  =  (v2  -  v4)  /  rt2;
     it3  =  (v3  -  v4)  /  rt3;
     it4  =  (0.e0   -  v1)  /  rt4;
     it5  =  (0.e0   -  v2)  /  rt5;
     it6  =  (0.e0   -  v3)  /  rt6;
     it7  =  (v5  -  v6)  /  rt7;
     it8  =  (v5  -  v7)  /  rt8;
     it9  =  (v5  -  v8)  /  rt9;
     it10  =  (v9  -  0.e0)  /  rt10;
     it11  =  (v10  -  0.e0)  /  rt11;
     it12  =  (v11  -  0.e0)  /  rt12;
  }
  {  /*  6  Diodes.
       */
     id1  =  (v6  -  v12)  /  rd1;
     id2  =  (v7  -  v13)  /  rd2;
```

```
    id3 = (v8 - v14) / rd3;
    id4 = (v12 - v9) / rd4;
    id5 = (v13 - v10) / rd5;
    id6 = (v14 - v11) / rd6;
}
vm3 = -(vm1 + vm2);
pi120t = wb*t;
sin_theta = sin(theta);
cos_theta = cos(theta);
sin_theta_pi6 = sin_theta*cos_pi_over_6 - cos_theta*sin_pi_over_6;
cos_theta_pi6 = cos_theta*cos_pi_over_6 + sin_theta*sin_pi_over_6;
E1 = vpeak*cos(pi120t);
E2 = vpeak*cos(pi120t-pi_2_over_3);
E3 = - (E1 + E2);
iqs = two_over_sqrt3 * (cos_theta_pi6*im1 + sin_theta*im2);
ids = two_over_sqrt3 * (sin_theta_pi6*im1 - cos_theta*im2);
vqs = two_over_sqrt3 * (cos_theta_pi6*vm1 + sin_theta*vm2);
vds = two_over_sqrt3 * (sin_theta_pi6*vm1 - cos_theta*vm2);
f_psi_m = f_sat(psi_m = dbl_hypot(psi_mq, psi_md));
T_load = sign_load * 98.865e0/377.e0*wr;
T_elec = 1.5e0*pole_pair*(psi_ds*iqs - psi_qs*ids)/wb;
{ /* Do KCL.
     * Don't forget to translate currents of two terminal devices into
     * single variables.
     *
     * Look into how one will define internal KCL for general multi-terminal
     * devices.
     */
    *pf++ = ilc1 + it1 - it4;          /* Node 1 */
    *pf++ = ilc2 + it2 - it5;          /* Node 2 */
    *pf++ = ilc3 + it3 - it6;          /* Node 3 */
    *pf++ = ilf - it1 - it2 - it3;     /* Node 4 */
    *pf++ = it7 + it8 + it9 - ilf;     /* Node 5 */
    *pf++ = id1 + iup1 - it7;          /* Node 6 */
    *pf++ = id2 + iup2 - it8;          /* Node 7 */
    *pf++ = id3 + iup3 - it9;          /* Node 8 */
    *pf++ = it10 + ilow1 - id4;              /* Node 9 */
    *pf++ = it11 + ilow2 - id5;              /* Node 10 */
    *pf++ = it12 + ilow3 - id6;              /* Node 11 */
    *pf++ = im1 + id4 - id1;           /* Node 12 */
    *pf++ = im2 + id5 - id2;           /* Node 13 */
    *pf++ = im3 + id6 - id3;           /* Node 14 */
}
/* Equations.
  * Comments on components and modules can be copyable.
  * One may define symbols for copyable comments.
 */
{ /* Voltage source.
    */
    *pf++ = v1 - v2 + lc*(dilc2 - dilc1) + E2 - E1;
    *pf++ = v1 - v3 + lc*(-2.e0*dilc1 - dilc2) + E3 - E1;
    *pf++ = ilc1 + ilc2 + ilc3;
}
{ /* Big inductor.
    */
    *pf++ = v4 - v5 - rf * ilf - lf * dilf;
}
{ /* Upper bridge capacitors.
    */
```

```
    *pf++ = v6 - v7 - vcup1;
    *pf++ = v7 - v8 - vcup2;
    *pf++ = - iup1 + c * (2.e0*dvcup1 + dvcup2);
    *pf++ = - iup2 + c * (dvcup2 - dvcup1);
    *pf++ = iup1 + iup2 + iup3;
  }
  { /* Lower bridge capacitors.
    */
    *pf++ = v9 - v10 - vclow1;
    *pf++ = v10 - v11 - vclow2;
    *pf++ = - ilow1 + c * (2.e0*dvclow1 + dvclow2);
    *pf++ = - ilow2 + c * (dvclow2 - dvclow1);
    *pf++ = ilow1 + ilow2 + ilow3;
  }
  { /* Induction motor.
    */
    double  dtmp_q,  dtmp_d;

    *pf++ = v12 - v13 - vm1 + vm2;
    *pf++ = v12 - v14 - vm1 + vm3;
    *pf++ = im1 + im2 + im3;
    *pf++ = iqs - (psi_qs - psi_mq) / xls;
    *pf++ = ids - (psi_ds - psi_md) / xls;
    *pf++ = -vqs + rs*iqs + (dpsi_qs + w*psi_ds)/wb;
    *pf++ = -vds + rs*ids + (dpsi_ds - w*psi_qs)/wb;
     *pf++ = rr*(psi_qr-psi_mq)/xlr + (dpsi_qr + (w-wr)*psi_dr)/wb;
     *pf++ = rr*(psi_dr-psi_md)/xlr + (dpsi_dr - (w-wr)*psi_qr)/wb;
    if (psi_m)
    {
       dtmp_q = psi_mq / psi_m;
       dtmp_d = psi_md / psi_m;
    }
    else
       dtmp_q = dtmp_d = 0.e0;
    *pf++ = psi_mq - (xm_star/xls * psi_qs + xm_star/xlr * psi_qr
              - xm_star/xm_unsat * dtmp_q * f_psi_m);
    *pf++ = psi_md - (xm_star/xls * psi_ds + xm_star/xlr * psi_dr
              - xm_star/xm_unsat * dtmp_d * f_psi_m);
    *pf++ = dtheta - w;
    *pf++ = T_elec - T_load - J*dwr/pole_pair;
  }
  { /* Equations for control part.
    */
    double  w_ref;

    w_ref = rpm_ref * rpm_to_we;
    w_error = w_ref - wr;
    *pf++ = dy_ctrl - w_error;
    iref = ksp * (kc * y_ctrl + w_error) + ilf_min;
    switch (StMach[CURRENT_LIMIT])
    {
    case UNDER_LIMIT:
      iref_star = iref;
      break;
    case LIMIT:
      iref_star = ilf_max;
      break;
    default:
       fprintf(stderr,"No state %d for %s\n", StMach[CURRENT_LIMIT],
```

```
            "CURRENT_LIMIT");
          break;
        }
         i_error = iref_star - ilf;
         *pf++ = dangle + (angle - CURRENT_GAIN*i_error)/TAU;
         switch (StMach[BLOCK2])
        {
         case UNDER_LIMIT:
           w_slip = 0.e0;
          break;
         case LIMIT:
            { /* Find slip frequency from ilf using interpolation.
              */
             static double   a[] = {-4.88e0,  .22e0,  -1.3e-3,  5.276e-6};
            int              i;

             w_slip = a[3];
             for (i = 2; i >= 0; i--)
                w_slip = w_slip*ilf + a[i];
           }
          break;
        default:
           fprintf(stderr,"No state %d for %s\n", StMach[BLOCK2],
            "BLOCK2");
          break;
       }
        inverter_freq = (wr + sign_w_slip*w_slip) / pi2;
        { /* Inverter controller.
          */
          *pf++ = dy_inv - sign_inv * inverter_freq;
       }
     }       /* End of equations for control part. */
}
/*****************************************************************************
  * MainEvent() contains all state machines.
  *
  * MainEvent() may be called repeatedly until all states are initialized
  * before integration.
   *****************************************************************************/
void  MainEvent(yst,  t)
register  double   *yst;
double      t;
{
   double *ynst,  *dyst;

   ynst  = yst  + numst;
   dyst  = yst  + numeq;
#ifdef   DEBUG_event
    fprintf(outfile,"event_h  is\n");
    printvec(event_h,MAX_ORD);
    fprintf(outfile,"event_al  is\n");
    printvec(event_al,MAX_ORD);
#endif
   { static STE_t  *tmp;

       Thyristor(yst,it1,&rt1,ON,THY1,THY1,"THY1",&tmp);
   }
   { static STE_t  *tmp;
```

```
        Thyristor(yst,it2,&rt2,OFF,THY2,THY2,"THY2",&tmp);
}
{ static STE_t  *tmp;

        Thyristor(yst,it3,&rt3,OFF,THY3,THY3,"THY3",&tmp);
}
{ static STE_t  *tmp;

        Thyristor(yst,it4,&rt4,OFF,THY4,THY4,"THY4",&tmp);
}
{ static STE_t  *tmp;

        Thyristor(yst,it5,&rt5,OFF,THY5,THY5,"THY5",&tmp);
}
{ static STE_t  *tmp;

        Thyristor(yst,it6,&rt6,ON,THY6,THY6,"THY6",&tmp);
}
{ static STE_t  *tmp;

        Thyristor(yst,it7,&rt7,ON,THY7,THY7,"THY7",&tmp);
}
{ static STE_t  *tmp;

        Thyristor(yst,it8,&rt8,OFF,THY8,THY8,"THY8",&tmp);
}
{ static STE_t  *tmp;

        Thyristor(yst,it9,&rt9,OFF,THY9,THY9,"THY9",&tmp);
}
{ static STE_t  *tmp;

        Thyristor(yst,it10,&rt10,OFF,THY10,THY10,"THY10",&tmp);
}
{ static STE_t  *tmp;

        Thyristor(yst,it11,&rt11,OFF,THY11,THY11,"THY11",&tmp);
}
{ static STE_t  *tmp;

        Thyristor(yst,it12,&rt12,ON,THY12,THY12,"THY12",&tmp);
}
{ static STE_t  *tmp;

    Diode(yst,id1,&rd1,OFF,D1,"D1",&tmp);
}
{ static STE_t  *tmp;

    Diode(yst,id2,&rd2,OFF,D2;"D2",&tmp);
}
{ static STE_t  *tmp;

    Diode(yst,id3,&rd3,OFF,D3,"D3",&tmp);
}
{ static STE_t  *tmp;

    Diode(yst,id4,&rd4,OFF,D4,"D4",&tmp);
}
{ static STE_t  *tmp;
```

```
        Diode(yst,id5,&rd5,OFF,D5,"D5",&tmp);
    }
    { static STE_t  *tmp;

        Diode(yst,id6,&rd6,OFF,D6,"D6",&tmp);
    }
    { /* Load part.
       */
      static STE_t          *tmp;

       switch (StMach[LOAD])
      {
       case ALIVE_ZONE:
          if (MONITOR(tmp, fabs(wr) - DEAD_BAND, tol))
        {
            NextState(LOAD, DEAD_ZONE);
           sign_load = 0.e0;
           RecomputeJac;
        }
         break;
       case DEAD_ZONE:
          if (MONITOR(tmp, DEAD_BAND - fabs(wr), tol))
        {
            NextState(LOAD, ALIVE_ZONE);
            sign_load = copysign(1.e0, wr);
           RecomputeJac;
        }
         break;
       case STATE_INIT:
         tmp = STE_alloc(1);
          NextState(LOAD, DEAD_ZONE);
         sign_load = 0.e0;
         break;
       default:
          fprintf(stderr,"No state %d for %s\n", StMach[LOAD], "LOAD");
         break;
      }
#ifdef   DEBUG_event
    fprintf(outfile,"LOAD->order is %d, LOAD=>state is %d, and LOAD->data
are\n",
          tmp->order, StMach[LOAD]);
    fprintf(outfile,"fire_type is %d, fire_h = %g\n", tmp->fire_type,
          tmp->deltat);
    printvec(tmp->data, (int) (MAX_ORD+1));
    printvec(tmp->dvdf, (int) MAX_ORD);
#endif
   }
   { /* Control Part. */
     { /* Gain part.
        */
       { /* Absolute value.
         */
         static STE_t     *ste_tmp;

          switch (StMach[ABS_VALUE])
        {
         case POS:
             if (MONITOR(ste_tmp, w_error, tol))
```

```
            {
                NextState(ABS_VALUE, NEG);
                sign_w_slip = -1.e0;
                RecomputeJac;
            }
            break;
          case NEG:
            if (MONITOR(ste_tmp, -w_error, tol))
            {
                NextState(ABS_VALUE, POS);
                sign_w_slip = 1.e0;
                RecomputeJac;
            }
            break;
          case STATE_INIT:
            ste_tmp = STE_alloc(1);
            NextState(ABS_VALUE, POS);
            w_error = rpm_ref * rpm_to_we - wr;   /* w_error is set here. */
            sign_w_slip = 1.e0;
            break;
          default:
            fprintf(stderr,"No state %d for %s\n", StMach[ABS_VALUE],
            "ABS_VALUE");
            break;
        }
#ifdef   DEBUG_event
    fprintf(outfile,"StMach[ABS_VALUE] = %d\n", StMach[ABS_VALUE]);
#endif
      }
      { /* Maximum value.
        */
        static STE_t    *ste_tmp;

        switch (StMach[CURRENT_LIMIT])
        {
          case UNDER_LIMIT:
            if (MONITOR(ste_tmp, ilf_max - iref, tol))
            {
                NextState(CURRENT_LIMIT, LIMIT);
                RecomputeJac;
            }
            break;
          case LIMIT:
            if (MONITOR(ste_tmp, -(ilf_max - iref), tol))
            {
                NextState(CURRENT_LIMIT, UNDER_LIMIT);
                RecomputeJac;
            }
            break;
          case STATE_INIT:
            ste_tmp = STE_alloc(1);
            NextState(CURRENT_LIMIT, UNDER_LIMIT);
            /* w_error must be set before. */
            iref = ksp * (kc * y_ctrl + w_error) + ilf_min;
            iref_star = (iref >= ilf_max)? ilf_max: iref;
            break;
          default:
            fprintf(stderr,"No state %d for %s\n", StMach[CURRENT_LIMIT],
            "CURRENT_LIMIT");
```

```
            break;
          }
#ifdef   DEBUG_event
   fprintf(outfile,"StMach[CURRENT_LIMIT] = %d\n", StMach[CURRENT_LIMIT]);
#endif
       }     /* End of maximum value. */
       { /* BLOCK2.
         */
         static STE_t    *ste_tmp;

         switch (StMach[BLOCK2])
        {
         case UNDER_LIMIT:
            if (MONITOR(ste_tmp, ilf_min - ilf, tol))
         {
              NextState(BLOCK2, LIMIT);
             RecomputeJac;
         }
          break;
        case LIMIT:
            if (MONITOR(ste_tmp, ilf - ilf_min, tol))
         {
              NextState(BLOCK2, UNDER_LIMIT);
             RecomputeJac;
         }
          break;
        case STATE_INIT:
           ste_tmp = STE_alloc(1);
           NextState(BLOCK2, UNDER_LIMIT);
          break;
        default:
            fprintf(stderr,"No state %d for %s\n", StMach[BLOCK2],
          "BLOCK2");
          break;
        }
#ifdef   DEBUG_event
   fprintf(outfile,"StMach[BLOCK2] = %d\n", StMach[BLOCK2]);
#endif
      }     /* End of BLOCK2. */
    }       /* End of gain part. */
     { /* Firing angle limiter.
       */
       static STE_t      *tmp;

        switch (StMach[ANGLE_LIMIT])
      {
       case UNDER_LIMIT:
         fire_angle = angle;
          if (MONITOR(tmp, 90.e0 - fabs(angle), tol))
       {
            NextState(ANGLE_LIMIT, LIMIT);
           ResetIntgOrd;
       }
        break;
      case LIMIT:
          fire_angle = copysign(90.e0, fire_angle);
         if (MONITOR(tmp, fabs(angle) - 90.e0, tol))
       {
            NextState(ANGLE_LIMIT, UNDER_LIMIT);
```

```
            ResetIntgOrd;
          }
          break;
        case STATE_INIT:
          tmp = STE_alloc(1);
          NextState(ANGLE_LIMIT, UNDER_LIMIT);
          fire_angle = angle;
          break;
        default:
            fprintf(stderr,"No state %d for %s\n", StMach[ANGLE_LIMIT],
             "ANGLE_LIMIT");
          break;
       }
#ifdef   DEBUG_event
    fprintf(outfile,"StMach[ANGLE_LIMIT] = %d\n", StMach[ANGLE_LIMIT]);
#endif
     }
      { /* Rectifier has 2 parts. */
        { /* Generate timing signal synchronizing with 60 Hz.
          */
          static STE_t    *ste_rect;

          /* For rectifier, thyristors THY1 and THY6 should be on
           * initially and THY2 is to be fired next.
          */
          switch (StMach[RECTIFIER])
         {
          case TRACK:
              if (MONITOR(ste_rect, 21.6e3*(tbase-t) - fire_angle, tol))
           {
               NextState(RECT_CTRL, START_PULSE);
              tbase += 1.e0/360.e0;
           }
           break;
          case STATE_INIT:
            ste_rect = STE_alloc(1);
           tbase = 1.e0/144.e0;
            NextState(RECTIFIER, TRACK);
           i_error = iref_star - ilf;
           break;
          default:
              fprintf(stderr,"No state %d for %s\n", StMach[RECTIFIER],
             "RECTIFIER");
           break;
         }
#ifdef   DEBUG_event
    fprintf(outfile,"StMach[RECTIFIER] = %d\n", StMach[RECTIFIER]);
#endif
       }     /* End of 1st part of rectifier control (generating timing
              * signal). */
        { /* 2nd part of rectifier control.
          */
          static int      seq[] = {THY1, THY6, THY2, THY4, THY3, THY5};
          static SCH_t    *sch_rect;
         int       tmp_logic;

          switch (StMach[RECT_CTRL])
         {
          case START_PULSE:
```

```
            tmp_logic = TRIG;
             NextState(RECT_CTRL, tmp_logic);
            rect_next_time = t + pulse_width;
             NextLogic(seq[rect_seq_index], (char *) &tmp_logic, RECT_CTRL);
           break;
         case TRIG:
             if (SCHEDULE(sch_rect, rect_next_time))
          {
             tmp_logic = OFF;
              NextState(RECT_CTRL, tmp_logic);
               NextLogic(seq[rect_seq_index], (char *) &tmp_logic, RECT_CTRL);
             if (5 > rect_seq_index)
               rect_seq_index++;
            else
               rect_seq_index = 0;
          }
           break;
         case OFF:
           break;
         case STATE_INIT:
            sch_rect = SCH_ALLOC();
            NextState(RECT_CTRL, OFF);
           rect_seq_index = 2;
            rect_next_time = t + pulse_width;
           break;
         default:
             fprintf(stderr,"No state %d for %s\n", StMach[RECT_CTRL],
            "RECT_CTRL");
           break;
        }   /* End of switch (StMach[RECT_CTRL]). */
#ifdef   DEBUG_event
   fprintf(outfile,"StMach[RECT_CTRL] = %d\n", StMach[RECT_CTRL]);
#endif
      }     /* End of 2nd part of rectifier control. */
    }       /* End of rectifier control. */
     { /* Inverter has 2 parts.
        */
       { /* This part generates ramp function.
         */
         static STE_t    *ste_inv;

          switch (StMach[INVERTER])
        {
         case UP:
             if (MONITOR(ste_inv, 1.e0/6.e0 - y_inv, tol))
          {
             sign_inv = -1.e0;
              NextState(INVERTER, DOWN);
               NextState(INV_CTRL, START_PULSE);
             RecomputeJac;
          }
           break;
         case DOWN:
            if (MONITOR(ste_inv, y_inv, tol))
          {
             sign_inv = 1.e0;
              NextState(INVERTER, UP);
               NextState(INV_CTRL, START_PULSE);
             RecomputeJac;
```

```
            }
             break;
            case STATE_INIT:
              ste_inv = STE_alloc(1);
              NextState(INVERTER, UP);
              sign_inv = 1.e0;
             break;
           default:
               fprintf(stderr,"No state %d for %s\n", StMach[INVERTER],
"INVERTER");
            break;
          }
#ifdef   DEBUG_event
   fprintf(outfile,"StMach[INVERTER] = %d\n", StMach[INVERTER]);
#endif
       }
        { /* This part sends pulses to thyristors.
          */
         static int      seq[] = {THY7, THY12, THY8, THY10, THY9, THY11};
         static SCH_t    *sch_inv;
        int       tmp_logic;

          switch (StMach[INV_CTRL])
        {
         case START_PULSE:
           tmp_logic = TRIG;
            NextState(INV_CTRL, tmp_logic);
           inv_next_time = t + pulse_width;
             NextLogic(seq[inv_seq_index], (char *) &tmp_logic, INV_CTRL);
          break;
        case TRIG:
            if (SCHEDULE(sch_inv, inv_next_time))
         {
             NextState(INV_CTRL, OFF);
            tmp_logic = OFF;
              NextLogic(seq[inv_seq_index], (char *) &tmp_logic, INV_CTRL);
            if (5 > inv_seq_index)
              inv_seq_index++;
           else
              inv_seq_index = 0;
         }
          break;
        case OFF:
          break;
        case STATE_INIT:
           sch_inv = SCH_ALLOC();
           NextState(INV_CTRL, OFF);
          inv_seq_index = 2;
           inv_next_time = t + pulse_width;
          break;
        default:
            fprintf(stderr,"No state %d for %s\n", StMach[INV_CTRL],
"INV_CTRL");
          break;
       }   /* End of switch (StMach[INV_CTRL]). */
#ifdef   DEBUG_event
   fprintf(outfile,"StMach[INV_CTRL] = %d\n", StMach[INV_CTRL]);
#endif
      }     /* End of 2nd part of inverter. */
```

```
    }       /* End of inverter part. */
  }         /* End of control part. */
} /* End of MainEvent(). */
void  MainInit(yst)
double   *yst;
{
   double *ynst,  *dyst;
   double pi;
   int       i;

   ynst = yst + numst;
   dyst = yst + numeq;

   TSTART = 0.e0;  /* Control parameters for integration. */
   TFINAL = .2e0;
   HPRINT = 1.e-4;

   pi = atan2(0.e0, -1.e0);
   wb = 120.e0*pi;
   sin_pi_over_6 = sin(pi / 6.e0);
   cos_pi_over_6 = cos(pi / 6.e0);
   pi_2_over_3 = pi / 1.5e0;
   two_over_sqrt3 = 2.e0/sqrt(3.e0);
   vpeak = 240.e0 * sqrt(2.e0);
   pi2 = 2.e0 * pi;
   rpm_to_we = pi2 / 60.e0 * pole_pair;
   pulse_width = 1.e-3;
   tol = 1.e-8;

   outfile = fopen("out", "w");
   warnFilePtr = fopen("warning", "w");
   timeFilePtr = fopen("time", "w");
   motorFilePtr = fopen("motor", "w");
   controlFilePtr = fopen("control", "w");
   deviceFilePtr = fopen("device", "w");
   supplyFilePtr = fopen("supply", "w");

   ilc1    = ilc10;       /* Initial conditions setup. */
   ilc2    = ilc20;
   ilf     = ilf0;
   vcup1   = vcup10;
   vcup2   = vcup20;
   vclow1  = vclow10;
   vclow2  = vclow20;
   psi_qs  = psi_qs0;
   psi_ds  = psi_ds0;
   psi_qr  = psi_qr0;
   psi_dr  = psi_dr0;
   theta   = theta0;
   wr      = wr0;
   y_inv   = y_inv0;
   y_ctrl  = y_ctrl0;
   /*
    * State machine initialization.
    */
   event_init();
   for (i = 0; i < NumStMach; i++)
      StMach[i] = STATE_INIT;
   for (i = 0; i < NumLnode; i++)
```

```
    {
        Lnode[i].size = sizeof(int);
        Lnode[i].val  = ECALLOC(Lnode[i].size, char);
        NextLnode[i]  = ECALLOC(Lnode[i].size, char);
    }
}
void output(yst,t)
double   *yst;
double   t;
{
    double *ynst;
    int       i;

    count_output++;
    if ((h[0] < 1.e-5) && prev_ord)
      return;
    ynst = yst + numst;

#ifdef   DEBUG_event
    fputc('\n', outfile);
#endif
    fprintf(outfile,"%12.5e ", t);
    fprintf(outfile,"%12.5e %1d ", h[0], prev_ord);
    for (i = 0; i < 6; i++)
        fprintf(outfile, "%d", StMach[i]);
    fputc(' ', outfile);
    for (; i < 12; i++)
        fprintf(outfile, "%d", StMach[i]);
    fputc(' ', outfile);
    for (; i < 18; i++)
        fprintf(outfile, "%d", StMach[i]);
    fputc(' ', outfile);
    for (; i < NumStMach; i++)
        fprintf(outfile, "%d", StMach[i]);
    fprintf(outfile, " ");
    for (i = 0; i < NumLnode; i++)
        fprintf(outfile, "%d", *((int *) Lnode[i].val));
    fputc('\n', outfile);
    fprintf(timeFilePtr, "%e\n", t);
    fprintf(motorFilePtr, "%e\t%e\t%e\t%e\t%e\t%e\n", v12-v13, vm1, im1,
psi_m,
          wr/rpm_to_we, T_elec);
    fprintf(controlFilePtr, "%e\t%e\t%e\t%e\t%e\n", v4, v5, ilf, fire_angle,
          vcup1);
    fprintf(deviceFilePtr, "%e\t%e\t%e\t%e\t%e\t%e\n", v1-v4, it1, v6-v12,
          id1, v5-v6, it7);
    fprintf(supplyFilePtr, "%e\t%e\n", E1, ilc1);
#ifdef   DEBUG_event
    fputc('\n', outfile);
    fflush(outfile);
#endif
}
#define  sat_coeff1     (-0.39286e3)      /* Saturation coefficients. */
#define  sat_coeff2     .21147e1
/*****************************************************************************
 *  f_sat() calculates f(psi_m) as described in Prof. Krause's book.
 *****************************************************************************/
static double f_sat(x)
double   x;
```

```
{
    return  x  >  186.85e0?  20.423e0*x-3806.38e0:  exp(sat_coeff1+sat_coeff2*x);
}
```