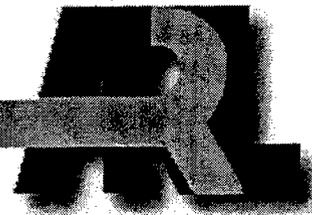


ARMY RESEARCH LABORATORY



A Lumped Circuit Model of a Compensated Pulse Generator and Rail Gun

Charles R. Hummer

ARL-TR-1990

JUNE 1999

19990722 076

Approved for public release; distribution is unlimited.

DTIC QUALITY INSPECTED 4

Pentium™ is a trademark of Intel Corporation.

Visual C++® is a registered trademark of Microsoft Corporation.

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

Army Research Laboratory

Aberdeen Proving Ground, MD 21005-5066

ARL-TR-1990

June 1999

A Lumped Circuit Model of a Compensated Pulse Generator and Rail Gun

Charles R. Hummer
Weapons & Materials Research Directorate

Approved for public release; distribution is unlimited.

Abstract

Because of its compact size, an electrical generator that uses an internal rotating mass as an energy source for electrical power is being developed to power a rail gun in a future combat system. At this stage of development, there are many proposed designs for these electric generators and many proposed designs for their possible uses: rail guns, coil guns, electromagnetic armor, etc. To study these various designs, a computer program was written to calculate the current in all parts of the electric generator, the load, the angular velocity of the rotating mass, and the velocity of the projectile from a rail gun or a coil gun. This was accomplished by modeling the electric generator and the rail gun by a circuit of inductors and resistors. This model results in a set of differential equations that are coupled with the equation of motion for the rotating mass and with the equation of motion for the projectile.

TABLE OF CONTENTS

	<u>Page</u>
LIST OF FIGURES	vii
1. INTRODUCTION	1
2. EQUIVALENT CIRCUIT MODEL	3
3. COMPUTER PROGRAM	17
4. SUBROUTINES	23
5. DISCUSSION AND RESULTS	28
REFERENCES	33
APPENDIX	
A. Computer Program	35
DISTRIBUTION LIST	59
REPORT DOCUMENTATION PAGE	61

INTENTIONALLY LEFT BLANK

LIST OF FIGURES

<u>Figure</u>		<u>Page</u>
1.	Cross Section of a Six-pole/Three-phase Compulsator	2
2.	Compulsator Circuit	2
3.	Magnetically Coupled Circuits	3
4.	Schematic of the Full Wave Rectifier Bridge to Field Coil	8
5.	Four-phase Compulsator	9
6.	Conducting Diode Equivalent Circuit	10
7.	Four-phase Load Circuit	11
8.	Four-phase Simplified Circuit	12
9.	Self-Exciting Field Current	30
10.	Rail Gun Current	30

INTENTIONALLY LEFT BLANK

A LUMPED CIRCUIT MODEL OF A COMPENSATED PULSE GENERATOR AND RAIL GUN

1. INTRODUCTION

A compensated pulse generator (compulsator) [1], is an electrical generator that uses an internal rotating mass as an energy source for electrical power. One type of compulsator was built under the Sub-Scale Focused Technology Program (SSFTP) by the Center of Electromagnetics (CEM) and the Institute of Advanced Technologies (IAT), University of Texas at Austin, Texas, as a step to develop a compulsator that will power a rail gun on a future combat system. It is speculated, however, that a compulsator could be used to power coil guns, electromagnetic armor, etc. Computer programs that can evaluate the usefulness of a compulsator for these other purposes do exist, but they are not readily available. Therefore, a program was written to do this evaluation by calculating the current in all parts of the compulsator and the load, the angular velocity of the rotating mass, and the velocity of the projectile from a rail gun or a coil gun. To keep the program simple, some features of the compulsator were not included and some assumptions were made, and the program was written for a particular type of compulsator. This program, however, could be rewritten to include these features and it could be a start for modeling other types of compulsators. The intent of this report is to provide the growing community with a documented computer program of a simple model of a compulsator. This documentation may allow members of the community to modify or expand it to satisfy their needs.

The rotating mass in the SSFTP is a titanium cylindrical shell 1.0 meter long and 0.3 meter in radius. This assembly is designed to rotate at 12,000 revolutions per minute (rpm) for an energy storage of 25 megajoules (MJ). A field winding (see Figure 1) located on the outside surface of the cylindrical shell will produce a magnetic field with six poles when a current is passed through it. Once the magnetic field is produced, voltages are induced by the rotating magnetic field in the nine armature windings that are mounted on a stationary cylindrical shell surrounding the rotor. The armature windings are electrically connected in three groups ϕ_1 , ϕ_2 , and ϕ_3 , with three windings in each group. The alternating voltage induced in each group or phase coil is shifted in phase relative to the other phase coils. Because there are six magnetic poles produced by the field winding and three phase coils, this compulsator is typed as a six-pole/three-phase generator.

The three phase coils and the field coil are connected to a circuit (see Figure 2) made of silicon-controlled rectifiers (SCRs) that control and direct the currents. The ends of the phase coils connect to a full-wave rectifier bridge whose output is connected to the field coil L_f and to a half-wave rectifier bridge whose output is connected to the load. The other ends of the phase coils are

connected to a common ground in a “Y” configuration. After the drum has been spun, a small “seed” current is started in the field coil by an auxiliary capacitor bank, not shown in Figure 2, to produce a small magnetic field. This magnetic field induces alternating currents, which are full wave rectified and directed to the field coil by the “Pos. Bus Bar” and the “Neg. Bus Bar” in the phase coils. This additional current through the field coil will continue to increase if the current gain is greater than the energy losses. Thus, the field coil current is “self-excited” from a small seed current until it reaches a larger current by using some of the rotational energy. After the field coil current reaches the desired level, the SCRs to the “Load Bus Bar” are closed to deliver current to the load.

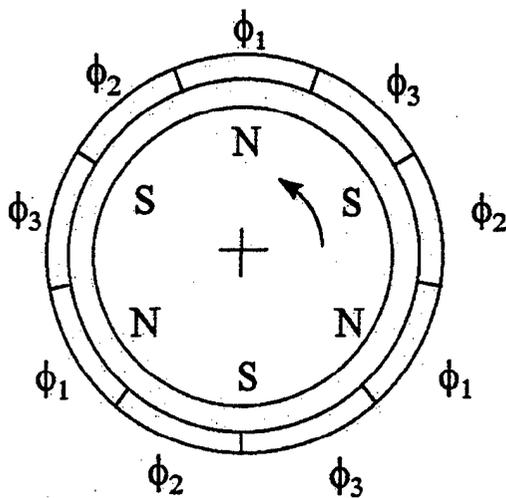


Figure 1. Cross Section of a Six-pole/Three-phase Compulsator.

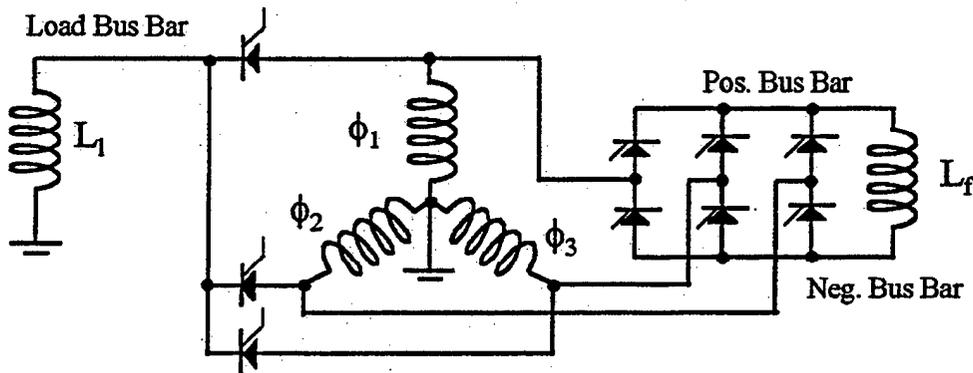


Figure 2. Compulsator Circuit.

2. EQUIVALENT CIRCUIT MODEL

The variables to be calculated are the shaft position and angular velocity, the projectile's position and velocity, the temperature of all the resistors, and all the currents. This is done by taking the time derivative of all these variables, which results in a system of first order differential equations that can be solved by numerical methods. These differential equations are introduced by considering a number (N) of simple circuits, each having a resistor and an inductor (see Figure 3). It will be shown later that each circuit represents a current loop in the compulsator circuit. The inductor and resistor in the simple circuits represent the total inductance and resistance in the current loop, and the mutual inductance between the simple circuits represents the magnetic coupling between the current loops. So let all the inductors inside the dashed box be magnetically coupled to each other, except for L_L which represents the inductance of the rail gun or the coil gun. The dots represent all the other similar circuits that are not shown. I_j is the current in each circuit with an inductor L_j and resistor R_j . To find the time derivative of the currents, consider the magnetic flux times the number of turns in each inductor, Φ_i , inside the box:

$$\Phi_i = \sum_{j=1}^N M_{i,j} I_j. \quad (1)$$

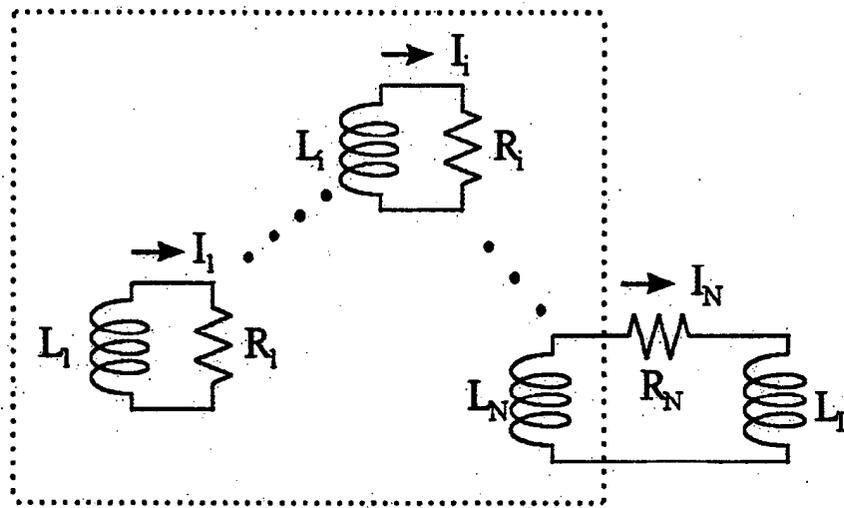


Figure 3. Magnetically Coupled Circuits.

The off-diagonal elements of the symmetrical matrix $M_{i,j}$ are the mutual inductance between the i -th and the j -th inductor, while the diagonal elements, $M_{i,i}$, are the self inductances. The electromotive force of the coil, \mathcal{E}_i , in each circuit is the electric field induced in the coil by the

changing magnetic flux $\mathcal{E}_i = -d\Phi_i/dt$. This electromotive force must equal the voltage drop across its resistor, $\mathcal{E}_i = R_i I_i$, or

$$\mathcal{E}_i = -\frac{d\Phi_i}{dt} = R_i I_i = -\sum_{j=1}^N \left(\frac{dM_{i,j}}{dt} I_j + M_{i,j} \frac{dI_j}{dt} \right). \quad (2)$$

The time derivative of the mutual inductance will not be zero because the field coil is moved by the rotor, causing the mutual inductance between it and the other inductors to change with time. Equation 2 can now be solved for the time derivative of the currents after a diagonal matrix, $R_{k,j}$, is introduced:

$$\frac{dI_i}{dt} = -\sum_{j=1, k=1}^{N, N} M_{i,k}^{-1} \left(\frac{dM_{k,j}}{dt} + R_{k,j} \right) I_j, \quad (3)$$

in which the diagonal element, $R_{k,k}$, is the resistance in the k -th circuit and all off-diagonal elements are zero. This matrix makes it possible to conveniently factor out the currents in the form of a vector and permit the addition of the matrices in the parentheses and the multiplication of the inverted matrix. It is found that Equation 3 applies to more complicated circuits, such as the compulsator, when the matrix, $R_{i,j}$, has some off-diagonal elements that are not zero. Therefore, the introduction of the matrix, $R_{i,j}$, is a generalization of the equation.

The time derivative of the shaft's angular velocity, the time derivative of the projectile's velocity (the acceleration), and the time derivative of the temperatures of the resistors can be derived from the total energy stored in the system:

$$W = \frac{1}{2} \sum_{i=1, j=1}^{N, N} I_i M_{i,j} I_j + \frac{1}{2} \mathcal{I} \omega^2 + \frac{1}{2} L_L(x) I_N^2 + \frac{1}{2} m v^2 + \sum_{i=1}^N H_i(T_i). \quad (4)$$

The first term is the total magnetic energy in the coils. The second term is the rotational energy of the rotor, in which \mathcal{I} is the moment of inertia of the rotor and ω is the angular velocity of the rotor. The third term is the magnetic energy of the launcher in which the inductance of the launcher depends on the position of the projectile x . The fourth term is the kinetic energy of the projectile, in which m is the mass and v is the velocity of the projectile. The last term is the sum of the enthalpy of the resistors:

$$H_i(T) = m_i \int_{T_a}^T C_i(T) dT, \quad (5)$$

in which $C_i(T)$ is the specific heat of the resistor at constant pressure and m_i is the mass of the resistor. The enthalpy is the energy required to raise the temperature of the resistor from ambient temperature T_a to the temperature T . The rate that this total energy changes with time, after using the circuit equations, Equation 2, is

$$\frac{dW}{dt} = -\frac{1}{2} \sum_{i=1, j=1}^{N, N} I_i \frac{dM_{i,j}(\theta)}{dt} I_j + \eta \omega \frac{d\omega}{dt} - \frac{I_N^2}{2} \frac{dL_L(x)}{dt} + mv \frac{dv}{dt} - \sum_{i=1, j=1}^{N, N} I_i R_{i,j} I_j + \sum_{i=1}^N m_i C_i(T_i) \frac{dT_i}{dt} \quad (6)$$

If the total energy of the system is to be constant, Equation 6 must be equal to zero at all times. This condition can be satisfied by identifying the terms that depend on a particular variable and setting the sum of the terms to zero. As an example, the first two terms depend on the shaft angle θ and its time derivative ω . This identification gives the equation of motion for the rotor:

$$\eta \omega \frac{d\omega}{dt} = \frac{1}{2} \sum_{i=1, j=1}^{N, N} I_i \frac{dM_{i,j}(\theta)}{dt} I_j. \quad (7)$$

Since the rotor or the field coil is the only coil that is moving, this equation could be rewritten as

$$\eta \frac{d\omega}{dt} = I_f \sum_{j=1}^N \frac{dM_{f,j}(\theta)}{d\theta} I_j, \quad (8)$$

in which I_f is the current in the field coil and $M_{f,j}(\theta)$ are the mutual inductances between the field coil and the armature coils that are carrying currents I_j . The chain rule was used to eliminate the time derivative inside the summation. The next two terms in Equation 6 depend on the position of the projectile x and its time derivative v . Setting the sum of these terms to zero gives the equation of motion for the projectile:

$$m \frac{dv}{dt} = \frac{I_N^2}{2} \frac{dL_L(x)}{dx}, \quad (9)$$

in which the chain rule was again applied to eliminate the time derivative on the right-hand side. The last two summations in Equation 6 depend on the temperatures of each individual resistor.

Equating like terms in the summations yields the rate that the temperature increases for each resistor:

$$m_i C_i(T_i) \frac{dT_i}{dt} = I_i^2 R_i(T_i), \quad (10)$$

which depends on its temperature T_i . The temperature dependence of the resistance is $R_i(T_i) = R_{0i} (1.0 - c_i [T_i - T_r])$, in which R_{0i} is the resistance of the resistor at the reference temperature T_r and c_i is the temperature coefficient. In general, the specific heat of materials depends on the temperature, but the specific heats for the resistors are assumed to be constant, $C_i(T_i) = C_i$. If the values of all the variables are given at a particular time, Equations 3, 8, 9, and 10 give the rate at which these variables are changing at the given time. The values of the variables can now be calculated for later times by numerically solving this system of first order differential equations. Bahder and Bruno [2] used this approach to find a system of differential equations for the circuit containing the rail gun, the N-th circuit in Figure 3. They replaced the rest of the circuits by a rotating "hard magnet," and numerically solved the differential equations for a single-phase alternating current flowing through the rail gun.

The boundary conditions for the differential equations are dictated by the action of the SCRs, which can be modeled by switches that are opened or closed during certain conditions. Using a switch as a model for an SCR ignores the voltage drop across the SCR when it is conducting, and the reverse current that flows through it for a brief time when it is becoming non-conducting. Assume that current is flowing through an SCR in the forward or positive direction, anode to cathode. The current will flow until the current decreases to zero and tries to flow in the reverse or negative direction, cathode to anode. This action is modeled by a closed switch that opens when the current through it decreases to zero. Now assume that the SCR is not conducting and there is a reverse bias voltage across the SCR. The SCR will not conduct a current until the reverse bias voltage reaches zero and there is a forward voltage across the SCR, provided that the SCR is triggered at this time. This action is modeled by an open switch that is closed when the voltage across the switch becomes zero. Thus, the first step in analyzing the compulsator circuit is to determine which SCRs are conducting and which are not and substitute them with a open or closed switch. After this substitution, the compulsator circuit then becomes a network of inductors and resistors that can be described by a set of differential equations. The solution of the differential equations (see Equation 3), the rotor (Equation 8), and the projectile (Equation 9) are all performed until the state of an SCR changes and is replaced by an open or closed switch. When the SCRs are replaced by an open or closed switch, all the currents are kept the same. Thus, the currents are continuous at all times in all parts of the circuit, but the time

derivatives of these currents are not continuous, and there will be discontinuous voltage changes across an opening SCR when modeled by an opening switch. Snubber circuits are usually placed in parallel to the SCRs to limit the voltage spikes across them and protect them from damage. These snubber circuits are ignored in this analysis, but there is a resistor and an inductor in series with each SCR to model the resistance and inductance of the connections to the SCR.

A number of conventions were made to aid in the writing of the program. One convention is the use of a "phase" angle rather than the angular position of the rotor. To illustrate these angles, consider just one of the phase coils ϕ_1 in Figure 1, for example. Because there are six magnetic poles on the field winding, the magnetic flux through the phase coil will go through three cycles for each rotation of the rotor. One cycle is produced when an N-Pole and an S-Pole pass the phase coil. Thus, the shaft angle θ is in proportion to the "phase" angle a , $a = N_p \theta/2$, in which N_p is the number of magnetic poles. The equation of motion of the rotor Equation 8 in terms of the phase angle is then

$$\frac{4}{N_p^2} \tau \frac{d^2 a}{dt^2} = I_f \sum_{j=1} \frac{dM_{f,j}(a)}{da} I_j. \quad (11)$$

This is convenient because the emphasis is on the calculation of the currents and not on the angular position of the rotor.

The next convention is the choice of currents and their directions that will become the basic variables, and the choice of voltage loops that will generate the differential equations. Because the boundary conditions for these differential equations are determined by the currents through the SCRs, it is convenient to choose these currents. The direction of a current is selected so as to be positive when the SCR is conducting, which is the established standard for specifying a current through SCRs and diodes. Given the currents through each SCR in the compulsator, it is possible to find the current through any other part of the circuit by using Kirchhoff's law. The currents in the full wave rectifier bridge, however, are not all independent. Consider the current through the field coil as shown in Figure 4 when no current is flowing through the load.

This is the mode of operation when the current through the field coil is being self-excited to a higher current, starting from the small seed current. Although SCRs are used in the full-wave rectifier bridge, they are gated so that they act as diodes and will be referred to as such. Furthermore, the diodes that are connected to the positive bus bar will be called "the positive diodes," and the diodes that are connected to the negative bus bar will be called "the negative diodes." The current through each diode is denoted by the symbol next to it. The numerical

subscript corresponds to the number labeling the phase coil, and the superscript indicates that the diode is either a positive diode or a negative diode. Applying Kirchhoff's law to the positive bus bar in the general case when all the positive diodes are conducting, the field current is $I_f = I_1^+ + I_2^+ + I_3^+$. Applying Kirchhoff's law to the negative bus bar in the general case when all the negative diodes are conducting, the field current must also be $I_f = I_1^- + I_2^- + I_3^-$. Thus, the sum of the currents through the positive diodes must be equal to the sum of the currents through the negative diodes. Once the values of five of the currents are independently chosen, there is no choice for the sixth current. In the specific case when some of the diodes may not be conducting, the current through a nonconducting diode is zero and is not a variable. Therefore, the total number of independent currents, N , is equal to the sum of the number of conducting positive diodes, N_p , plus the number of conducting negative diodes, $N_n - 1$, after applying Kirchhoff's law, or $N = N_p + N_n - 1$.

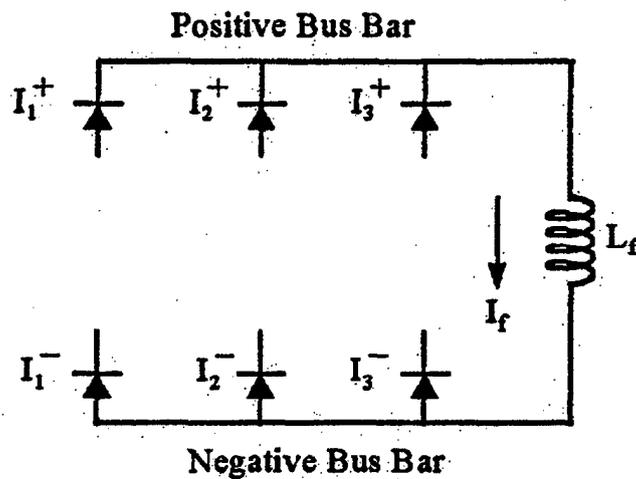


Figure 4. Schematic of the Full Wave Rectifier Bridge to Field Coil.

Now that a convention for the independent currents and their number is established, an equal number of linearly independent differential equations must be found that can be solved for the time derivative of each current (Equation 8). A differential equation is found by summing the voltages across the elements that make a loop in the circuit and setting their sum to zero. Because one loop results in one differential equation, the number of loops must be equal to the number of independent currents. Choosing the required number of loops in this circuit is a difficult subject because of the large number of possible combinations of conducting and non-conducting diodes. Each combination can produce a circuit with a number of possible loops. The following is a procedure that the program uses to choose these loops for a given combination of conducting and

non-conducting diodes. This procedure is motivated by the desire to treat compulsators with an arbitrary number of phase coils. In fact, this procedure is presented here for a compulsator with four phase coils. If this procedure were to be applied to a compulsator with three phases, some of the steps would seem to be trivial, and it would be difficult to explain why the steps are necessary. Figure 5 is a partial circuit for a four-phase compulsator, which is a possible design for the exit criteria machine (ECM) at IAT. The same notation for the diode currents is used here as in Figure 4.

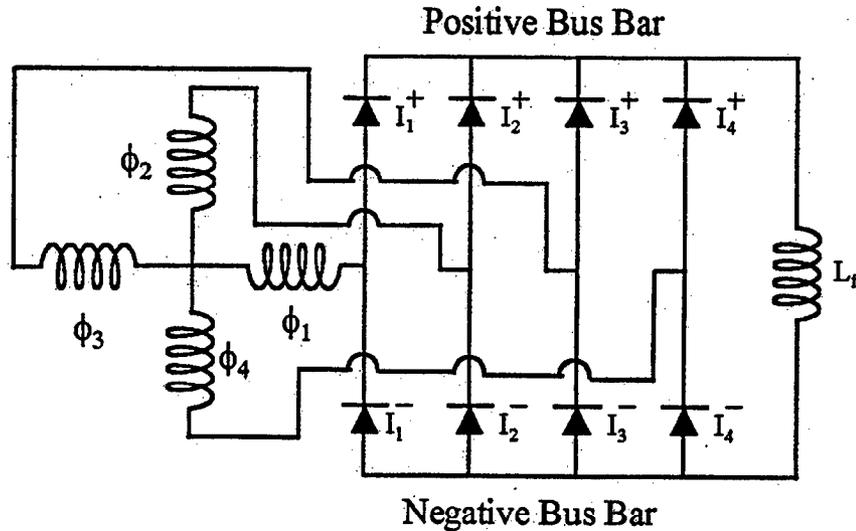


Figure 5. Four-phase Compulsator.

The circuit to the load is not considered at this time and is not shown. The load circuit is easy to analyze and can be easily included with the loop equations, as is shown later.

The number of possible loops through this circuit is reduced by the fact that not all combinations of conducting and non-conducting diodes can occur. For example, diodes I_1^+ and I_1^- cannot be conducting at the same time. Current can either be leaving the phase coil ϕ_1 and be directed to the positive bus bar through diode I_1^+ , or the current can enter the phase coil ϕ_1 from the negative bus bar through diode I_1^- , but not both at the same time. This is true for the other like pairs of diodes: I_2^+ and I_2^- , I_3^+ and I_3^- , and I_4^+ and I_4^- . It is possible, however, for both diodes in these pairs to be non-conducting at the same time, which means that no current is following through the phase coil. In addition, at least one positive diode and at least one negative diode must be conducting at all times so that a current can flow through the field coil.

Using these rules, assume that diodes I_1^+ , I_2^+ , I_3^- , and I_4^- are conducting and the rest are not conducting. After eliminating the non-conducting diodes from the circuit, it is possible to rearrange the elements into a simpler equivalent circuit (see Figure 6).

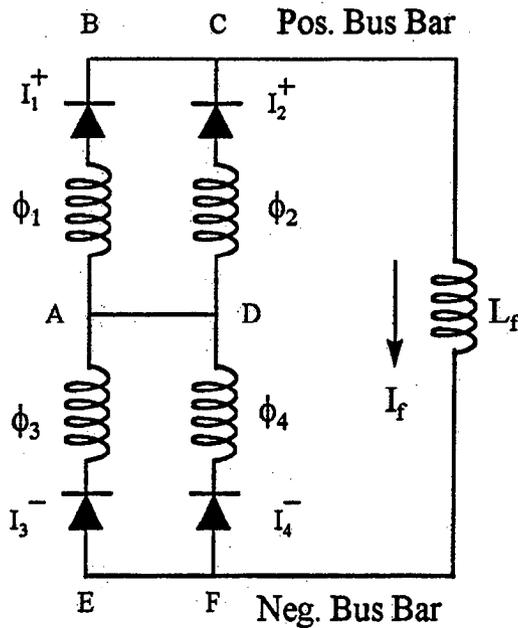


Figure 6. Conducting Diode Equivalent Circuit.

In this example, there are two conducting positive diodes, $N_p = 2$, and two conducting negative diodes, $N_n = 2$. Thus, there are three independent currents, and three loops must be constructed from this circuit. These loops are found in two sets. All the loops in the first set go through the field coil and the first negative diode on the left I_3^- and through a different positive diode. The first loop in this scheme is traced by going through the points ABCFEA, and the second loop is traced by the points ADCFEA. If the compulsator had more phases and more conducting positive diodes, each additional loop would go through one of the additional positive diode, through the field coil and through the first negative diode I_3^- for a total of N_p loops. In the second set, all the loops go through the first positive diode on the left I_1^+ , through the field coil and return through a different negative diode but exclude the first negative diode I_3^- that was used in the first set. Thus, there is only one loop in this set that is traced by going through the points ABCFDA, including diode I_4^- . If the compulsator had more phases and more conducting negative diodes, each additional loop would still go through the first positive diode I_1^+ , through the field coil, and return through one of the additional negative diode for a total of $N_n - 1$ loops. The total number of loops in both sets is $N_p + N_n - 1$, which matches the number of independent currents. This is not the only possible scheme, but the amount of effort to write the computer code to automate it was reasonable.

After the field coil current has accumulated from the small seed current during the self-excitation stage of operation, the SCRs to the load are allowed to conduct. This, of course, will

add to the number of currents or basic variables and will add to the number of loops in the circuit. Because of the nature of the circuit, half-wave rectifying bridge, the number of additional basic variables is simply the number of SCRs that are conducting to the load as shown in Figure 7.

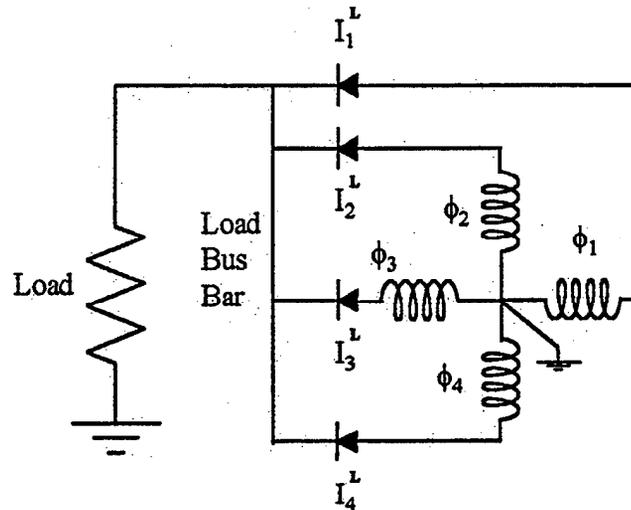


Figure 7. Four-Phase Load Circuit.

Here, the field coil and its full-wave bridge have been eliminated. Current to the load is the sum of the currents through the SCRs $I_{load} = I_1^l + I_2^l + I_3^l + I_4^l$. This current leaves the load and returns to the circuit at the common connection of the phase coils through the ground. Because there are no diodes or SCRs in this return path, there are no additional basic variables and no additional conditions on the currents. The additional loops simply go through a phase coil, its SCR, and the load. Each loop will then have a different phase coil-SCR combination and the load for a total of N_l additional loops. Thus, the total number of basic variables and loops will be $N = N_l + N_p + N_n - 1$, when the currents to the field coil are included. It is possible that the load could be connected to the phase coils by a full-wave bridge in some future compulsator. In this case, the analysis of the load circuit and the field circuit would be the same.

This concludes the discussion of all the essential details needed to write the program. What have not been discussed are the details about how the program performs the task of setting up the differential equations to be solved. The method used by the program is illustrated by considering the simplified circuit of a four-phase compulsator (see Figure 8).

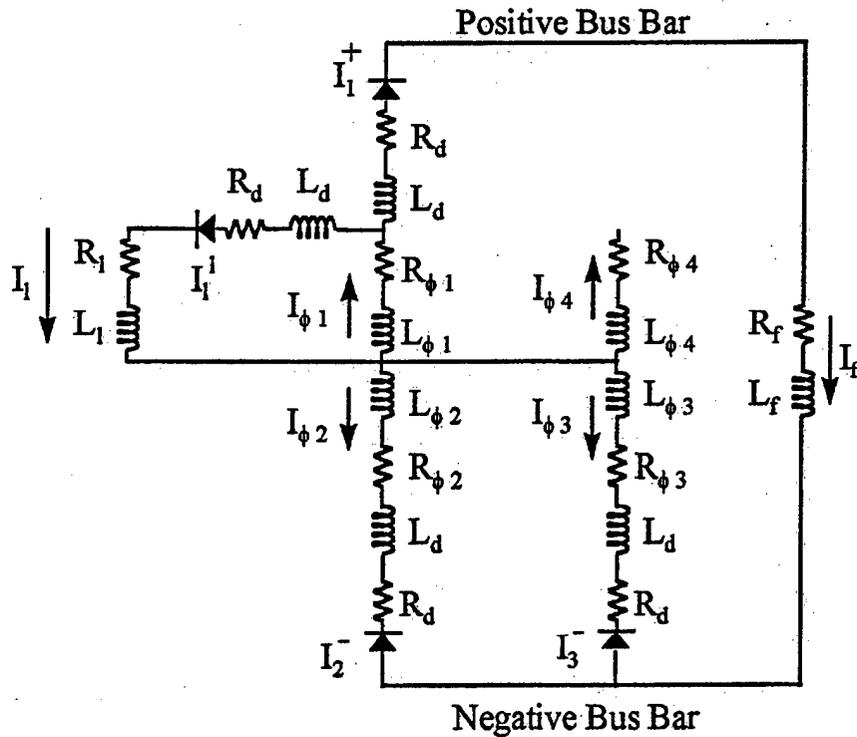


Figure 8. Four-Phase Simplified Circuit.

In this case, there is current in the load, which means that the stage of operation when this circuit configuration occurred was after the self-excitation of the field and some time after current to the load has been started. The phase coil, ϕ_4 , is shown even though it is not connected to anything. This means that all three diodes connected to it, two in the full-wave bridge and one to the load, are not conducting and no current is flowing through it at this time. At some later time, the voltage across this phase coil will become equal to the voltage on the positive bus bar, the negative bus bar, or the load when the appropriate diodes become conducting and change the configuration of the circuit. Thus, it is necessary to calculate the voltage output of any open phase coil and to test its value.

The method that is used by the program to set up the differential equations is based on matrix algebra. One matrix embodies Kirchoff's law, which produces a list of currents in a given order when it is multiplied with a list of the basic variables or diode currents. Another matrix performs the summation of the voltages around the required number of loops. Using these matrices and others, one can construct a matrix that will give a list of the time derivative of the diode currents when it is multiplied by a list of the diode currents as in Equation 3. Indeed, the steps leading to Equation 3 are equivalent to the steps taken here. The only difference is that the matrix for the Kirchoff's law and the matrix for the summation of the voltage around the circuit

loops are included here. This is the most complicated part of the program. The other parts of the program are simple manipulation, sorting, and bookkeeping of the data. Thus, this part of the program is discussed by using the circuit in Figure 8 as an example and presenting all the matrices in detail.

The matrix for Kirchhoff's law is constructed in two steps. This first step is to list the diode currents that are not zero in a specific order. This order starts with the first phase coil that is conducting to the positive bus bar, to the load, or both (ϕ_1 in Figure 8). In this case, there are two conducting diodes, I_1^+ and I_1^l . Of these two currents, the one conducting to the positive bus bar I_1^+ appears first, followed by the current to the load I_1^l . If either one of these currents were zero, it would be eliminated from the list, leaving the other as the first on the list. As an example, if I_1^+ were zero or not conducting, it would be eliminated and I_1^l would be the first entry. Conversely, if I_1^l were zero, it would be eliminated and I_1^+ would be the first entry. The next two possible entries would be the next phase coil that is conducting to the positive bus bar or the load. The source of the next entry could have been ϕ_4 if one or both of its diodes, I_4^+ or I_4^l , were conducting. Thus, the next two entries could have been I_4^+ , followed by I_4^l if both diodes were conducting, or just I_4^+ if it was the only conducting diode, or just I_4^l if it was the only conducting diode. The next set of entries is simply the negative diodes listed from left to right in Figure 8, I_2^- and I_3^- . Using these four diode currents, the current through any other part of the circuit can be found. Note that these currents are not independent and one of them must eventually be eliminated, but for now, assume that they are independent. Using Kirchhoff's law, a list of currents through the rest of the circuit is found by multiplying a matrix with the list of diode currents (see Equation 12).

The list of currents on the left is also in a specific order. First are the currents through the phase coils, followed by the current through the field coil. The next set of entries is the same as the list of currents on the right of the matrix. The last entry is the load current, if any diode is conducting to the load. The load current will not be in the list when there is no load current. Given the conducting state of all the diodes, the program constructs this matrix. There is one complication: the direction for a positive current through a phase coil is directed away from the common connecting point that must be adopted so that the sign of the mutual inductance between the phase coils is to be consistent with the current directions. If a positive current is flowing through the phase coil ϕ_1 , for example, the magnetic flux through another coil will increase when the mutual inductance between them is positive or will decrease when it is negative. Likewise, a negative current through ϕ_1 will decrease the magnetic flux through another coil when the coil's mutual inductance is positive and will increase the magnetic flux when it is negative. Thus, the

direction of the positive current must be specified when the mutual inductance is measured. The next step is to eliminate one of the diode currents. The convention adopted here is to eliminate the first negative diode, I_2^- , and use the other diode currents as the basic variables.

$$\begin{pmatrix} I_{\phi 1} \\ I_{\phi 2} \\ I_{\phi 3} \\ I_f \\ I_1^+ \\ I_1^l \\ I_2^- \\ I_3^- \\ I_l \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} \times \begin{pmatrix} I_1^+ \\ I_1^l \\ I_2^- \\ I_3^- \end{pmatrix}. \quad (12)$$

I_2^- is eliminated by using the following matrix equation:

$$\begin{pmatrix} I_1^+ \\ I_1^l \\ I_2^- \\ I_3^- \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} I_1^+ \\ I_1^l \\ I_3^- \end{pmatrix}. \quad (13)$$

Multiplying the two matrices together, the currents through the rest of the circuit in terms of the basic variables are

$$\begin{pmatrix} I_{\phi 1} \\ I_{\phi 2} \\ I_{\phi 3} \\ I_f \\ I_1^+ \\ I_1^l \\ I_2^- \\ I_3^- \\ I_l \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 \\ -1 & 0 & 1 \\ 0 & 0 & -1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \times \begin{pmatrix} I_1^+ \\ I_1^l \\ I_3^- \end{pmatrix}. \quad (14)$$

This relationship also holds for the time derivative of the currents.

Now that the current and the time derivative of the currents are known in all parts of the circuit, it is now possible to calculate the voltage drop across any coil. Each inductor in Figure 8 is in series with a resistor that represents the resistance of the windings of the coil. Therefore, the voltage drop across any coil is $V = \dot{\Phi} + RI$ when the voltage is measured in the direction of a positive current. If the coil is magnetically coupled to other coils, such as the phase coils, the time derivative of the magnetic flux in the coil is found by taking the time derivative of Equation 1 to give the voltage drop across the coil as

$$V_i = R_i I_i + \sum_{j=1}^N (M_{i,j} \dot{I}_j + \dot{M}_{i,j} I_j), \quad (15)$$

in which \dot{I}_j is the time derivative of a current and $\dot{M}_{i,j}$ is the time derivative of the mutual inductance. If the coil is not magnetically coupled to another coil, its voltage drop would simply be $V = RI + L\dot{I} + \dot{L}I$ in which L is the self-inductance of the coil. Normally, the self-inductance of a coil will not change with time, but the rail gun and other loads may have an inductance that will change with time. Explicitly, Equation 15 for the circuit in Figure 8 is shown in Equation 16 in which the second 9x9 matrix is the sum of the matrix, $\dot{M}_{i,j}$, and the resistance matrix, $R_{i,j}$, which has the resistance of the coils on its diagonal. These two matrices fit well together because most of the diagonal elements of $\dot{M}_{i,j}$ are zero and the resistance matrix $R_{i,j}$ is diagonal. In Equation 16, v is the velocity of the projectile, L' is the inductance gradient of the rail gun, ω is the angular velocity of the phase angle, and the M 's are the derivatives of the mutual inductance between the phase coils and the field coil with respect to the phase angle.

The next step is to construct a matrix that will sum the voltage drops around the loops in the circuit and make the sum be zero:

$$\begin{pmatrix} 1 & -1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & -1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} V_{\phi 1} \\ V_{\phi 2} \\ V_{\phi 3} \\ V_f \\ V_I^+ \\ V_I^l \\ V_2^- \\ V_3^- \\ V_l \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}. \quad (17)$$

$$\begin{pmatrix} V_{\phi 1} \\ V_{\phi 2} \\ V_{\phi 3} \\ V_f \\ V_1^+ \\ V_1^l \\ V_2^- \\ V_3^- \\ V_l \end{pmatrix} = \begin{pmatrix} L_{\phi 1} & M_{1,2} & M_{1,3} & M_{1,f} & 0 & 0 & 0 & 0 & 0 \\ M_{2,1} & L_{\phi 2} & M_{2,3} & M_{2,f} & 0 & 0 & 0 & 0 & 0 \\ M_{3,1} & M_{3,2} & L_{\phi 3} & M_{3,f} & 0 & 0 & 0 & 0 & 0 \\ M_{f,1} & M_{f,2} & M_{f,3} & L_f & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & L_d & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & L_d & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & L_d & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & L_d & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & L_l \end{pmatrix} \times \begin{pmatrix} \dot{I}_{\phi 1} \\ \dot{I}_{\phi 2} \\ \dot{I}_{\phi 3} \\ \dot{I}_f \\ \dot{I}_1^+ \\ \dot{I}_1^l \\ \dot{I}_2^- \\ \dot{I}_3^- \\ \dot{I}_l \end{pmatrix} + \begin{pmatrix} R_{\phi 1} & 0 & 0 & \omega M'_{1,f} & 0 & 0 & 0 & 0 & 0 \\ 0 & R_{\phi 2} & 0 & \omega M'_{2,f} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & R_{\phi 3} & \omega M'_{3,f} & 0 & 0 & 0 & 0 & 0 \\ \omega M'_{f,1} & \omega M'_{f,2} & \omega M'_{f,3} & R_f & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & R_d & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & R_d & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & R_d & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & R_d & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & R_l + vL' \end{pmatrix} \times \begin{pmatrix} I_{\phi 1} \\ I_{\phi 2} \\ I_{\phi 3} \\ I_f \\ I_1^+ \\ I_1^l \\ I_2^- \\ I_3^- \\ I_l \end{pmatrix} \quad (16)$$

The loops are the ones that were described before. The first row of the matrix sums the voltages in the loop that goes through the first phase coil, ϕ_1 , the first positive diode, I_1^+ , the field coil, I_f , the first negative diode, I_2^- , and its phase coil, ϕ_2 . The second row of the matrix sums the voltages in the loop that goes through the first phase coil, ϕ_1 , the first positive diode, I_1^+ , the field coil, I_f , the second negative diode, I_3^- , and its phase coil, ϕ_3 . The last row of the matrix sums the loop that goes through the first phase coil, ϕ_1 , the first diode to the load, I_1^l , and the load, I_l .

The final step is to solve the above equations for the time derivatives of the basic variables. Instead of using the explicit matrices and vectors, the above equations are rewritten in the usual matrix notation. Starting with Equation 17 for the sums of the voltage drops around the loops, we have $\mathcal{L}_{n,i} V_i = 0$, in which $\mathcal{L}_{n,i}$ is the matrix that performs the summations. Substitute this into Equation 15 in matrix notation, $V_i = M_{i,m} \dot{q}_m + (R_{i,m} + \dot{M}_{i,m}) q_m$, and the equation becomes $\mathcal{L}_{n,i} M_{i,m} \dot{q}_m + \mathcal{L}_{n,i} (R_{i,m} + \dot{M}_{i,m}) q_m = 0$, in which q_m is the list of all the currents. Now let $q_m = \mathcal{K}_{mj} I_j$ and $\dot{q}_m = \mathcal{K}_{mj} \dot{I}_j$ in which \mathcal{K}_{mj} is the Kirchhoff matrix in Equation 14 and I_j is the list of the independent basic variables. With this substitution, it is now possible to solve for \dot{I}_j to find an equation that is equivalent to Equation 3:

$$\dot{I}_j = -(\mathcal{L}_{j,i} M_{i,m} \mathcal{K}_{m,n})^{-1} (\mathcal{L}_{n,i} [R_{i,m} + \dot{M}_{i,m}] \mathcal{K}_{m,k}) I_k. \quad (18)$$

This equation, the equation of motion of the rotor (Equation 13), the equation of motion of the projectile (Equation 9), and the enthalpy equation for the resistors (Equation 10) are all used by a fourth order Runge-Kutta method [2] to calculate the values of all the basic variables at some future time. These results are tested for a change in the conducting state of a diode. If a diode has become conducting or non-conducting, then a new Kirchhoff matrix, a new loop matrix, and an new list of independent currents are constructed and used until the conducting state of a diode again changes.

3. COMPUTER PROGRAM

The discussion of the program that is presented in Appendix A starts with a "header" section where the important variables, constants, functions, and structures are either declared or defined. This is followed by a discussion of the "main" section. Subroutines that are important for reporting the results are also discussed. A user should be able to edit these sections and use the program for his or her purposes after reading this part. This approach, however, will omit a discussion of some very important subroutines because they do not appear explicitly in this part

of the program, but they will be covered in the next section. The program to be discussed is listed in the appendix and is referred to by line number or by quoting the statement. The line numbers are for reference only and are not part of the C Language. All the parameters in the header section are taken directly or derived from Kitzmiller.[3]

All the parameters that describe a compulsator are defined at the beginning of the program. “6:#define PHASE 3” defines the number of phases and “7:#define POLES 6” defines the number of poles of the field coil. “9:#define MAX 11” is a maximum number that is used for dimensioning arrays which is equal to the number of SCRs plus one for the field current and plus one for the load current or three times the number of phases plus two. “10:#define TREF 20.0” is the reference temperature in degrees Celsius. Statement 11 defines a structure tagged “resistor” which holds the parameters for a resistor. “ r_o ” is the resistance at the reference temperature “TREF”. “c” is the temperature coefficient for the resistance. “cp” is the heat capacity or the specific heat of the resistor. “mass” is the mass of the resistor. Kitzmiller [3] does not give the masses for the resistors, but he does give the resistivity, r_o , the cross-sectional area, A, the density, s, and the resistance at ambient temperature, r_o , that could be used to find the mass

$$m = \frac{\sigma A^2 r_o}{\rho_o} \quad (19)$$

Next is a list of all the resistive elements in the compulsator, starting with Statement 12 which defines a structure with the parameters for the field coil resistance. Statement 13 defines the common parameters for each resistor in series with an SCR that is connected to the positive bus bar “Pos. Bus Bar” in Figure 2. Statement 14 lists the common parameters for the resistors in series with the SCRs that are connected to the negative bus bar “Neg. Bus Bar,” and Statement 15 contains the common parameters for the resistors in series with the SCRs that are connected to the “Load Bus Bar.” The values for “ r_o s” of the resistors were taken from Kitzmiller [3], but the rest of the parameters in Statements 13 through 15 were derived by assuming that the connections between the compulsator and the SCRs are made of aluminum, which establishes the temperature coefficient c and the specific heat cp. The masses are a rough guess. Fortunately, these masses are not critical to the final results as long as the temperature rise in these resistors is small compared to that of the field coil and the armature coils. Statement 16 stores the parameters for the resistance of the phase coils in an array.

The two-dimensional array in Statement 17 defines the self and mutual inductances of the armature. The diagonal elements are the self-inductance of a phase coil while the off-diagonal elements are the mutual inductances between the pairs of phase coils. As an example, the self

inductance of the phase coil, ϕ_1 , identified by the starting index 0, is the element "mut[0][0]" in the array or 1.16e-06 H, and the self inductance of ϕ_2 is the element "mut[1][1]" in the array or 1.12e-06 H. Thus, the mutual inductance between these two phase coils is the element "mut[0][1]" or, equivalently, "mut[1][0]" which is -4.72e-07 H. The next array in Statement 18 contains the maximum mutual inductance between the field coil and each of the phase coils. If the field coil and the phase coils had perfect symmetry, the phase shift between each phase would be 120° of the phase angle. Because a real compulsator is not perfectly symmetrical, the phase shifts are slightly different. The phase shift angles are tabulated in the array "os[]" in Statement 19 where the angles are in radians. The first entry in this array is always zero, since this phase is taken to be the reference phase from which all other phases are measured. The next entry is then the phase shift for the next coil, etc. The next four statements, 20 through 23, are the inductances of the field coil "hfield," the inductance to the SCRs that are connected to the positive bus bar "hpscr," the inductance to the SCRs that are connected to the negative bus bar "hnsr," and the inductance to the SCRs that are connected to the load bus bar "hlscr." Statement 24 is the moment of inertia of the rotor scaled to the phase angle (Equation 13). This concludes the section that contains all the constants for the compulsator.

The next section contains constants and functions that model the load of the compulsator. In this case, the load is a 3.0-m-long rail gun launching a 0.32-kg projectile. The "mass" in Statement 25 is double the mass of the projectile. The function defined by Statements 26 through 28 is the load inductance as a function of the position of the projectile in the rail gun. Any stray inductance between the output of the compulsator and the breach of the rail gun is included in this function. The next function "dhdx" in Statements 29 through 31 is the derivative of "hload." Statements 32 through 34 are the load resistance as a function of the position of the projectile and include the resistance between the output of the compulsator and the breach of the rail gun. All the constants and all the functions that model the compulsator and the rail gun are now defined. Other compulsators, rail guns, or loads can be modeled by editing the Statements 6 through 34. No further editing is needed. In some rail guns, a circuit is placed across the muzzle to limit the voltage at the muzzle or to recover some of the energy stored in the rail gun after the projectile exits. Some of these circuits may be modeled by these functions, but if the circuit contains a capacitor then the program must be modified.

Statements 49 through 113 define various structures and variables that are used by procedures that are discussed in a later section. The structures between Statements 50 and 55 are important for printing the results of the calculations and are discussed here. One of the features of the C Language is the ability to group variables that have some common feature into one

structure. As an example, Statement 50 groups all the mechanical variables into a structure called "mechanical." These variables are the phase angle "th," the time derivative of the phase angle or the angular frequency "w," the position of the projectile "x," and the velocity of the projectile "v." The next set or structure of variables, Statement 51, is the temperatures of all the resistive elements: the SCRs to the positive bus bar "prec," the SCRs to the negative bus bar "nrec," the SCRs to the load "lrec," the phase coils "phs," and the field coil "field." All the variables in this structure are indexed except for "field." The index of these variables refers to an phase coil labeled with the index. As an example, prec[0], nrec[0], and lrec[0] are the temperatures of the SCRs that are connected to the phase coil ϕ_1 whose temperature is stored in phs[0]. The "config" structure in Statement 52 contains logical variables that give the conducting state of each SCR which is "TRUE" if the SCR is conducting or "FALSE" if the SCR is not conducting. The names of the variables in this structure are the same as the ones in the "thermal" structure. Thus, the like variables in both of these structures refer to the same SCR when the indices of all these variables are the same. The variable "load" in Statement 52 is special. When this variable is FALSE, all the SCRs connected to the load "lrec" are forbidden to become conductive even when the conditions for becoming conductive are satisfied. Thus, the load is disconnected from the compulsator and the compulsator is in the self-excitation mode. When it is set to TRUE, all the load SCRs that have a forward voltage are immediately closed. Afterward, any nonconducting load SCR is allowed to become conductive when its forward voltage becomes positive. Thus, the compulsator is in the launch mode when the projectile is being accelerated down the rail gun. This variable is returned to FALSE at some later time when any SCR that was conducting while "load" was TRUE remains conducting and will continue to conduct until its current reverses and becomes nonconductive. Once the SCR becomes nonconductive, it is forbidden to become conductive again. Thus, the current to the load is being shut off as the conducting SCRs become nonconducting and are not being gated back on. Statement 53 defines a structure for the currents in the compulsator using the same variable names used in the previous structures. "prec[PHASE]" are the currents through the SCRs that are connected to the positive bus bar, "nrec[PHASE]" are the currents through the SCRs connected to the negative bus bar, and "lrec[PHASE]" are the currents through the SCRs connected to the load. The field coil current and the load current are included for convenience. All these structures are then united into a single structure, Statement 54, which is defined as the state of the compulsator. The final structure in the header section, Statement 55, contains the voltages at various points in the circuit relative to the ground point in Figure 2: the voltage across the phase coil "vphs," the voltage of the positive bus bar "vplus," the voltage of the negative bus bar "vminus," and the voltage across the load "vload," which includes the resistance and inductance of connections between the output

terminals and the breach of the rail gun. These voltages are used to determine if a nonconducting SCR becomes conducting.

This completes the discussion of the header section of the program. The main part of the program, Statements 56 through 151, organizes the computation by calling the procedures in proper order and analyzing the results. These procedures are discussed only in general terms in this section, along with the discussion of the statements. Statements 78 through 97 prompt the user to enter information for the simulation, starting with the name of the output file. Next is the initial current in the field coil "seed" which seeds the magnetic field for self-excitation and the initial revolutions per minute of the rotor. The ambient temperature "tamb" and time step between calculations "tstep" are entered next. "tclose" is the time when the SCRs are permitted to conduct to the load. "topen" is the time when the conducting SCRs to the load are forbidden to conduct again after they have become nonconducting. "tend" is the maximum time that the calculation is allowed to continue. Because "tstep" is usually much smaller than "tend," it takes a large number of steps to reach "tend." If the results of the calculations were to be written to a file at each step, the resulting file could be large and contain too much information to be useful. The "Report Skip = " asks for the number of steps that will skip the writing of the results to the file.

The first procedure to be called is "initiate" in Statement 101 and is defined in Statements 154 through 198 which fill the state structure named "yo" with the initial values of all the variables at the time the seed current through the field coil has been established. This procedure does not model the discharge of the auxiliary capacitor bank into the field coil. Instead, the final seed current is taken as a given value. To determine the values for the rest of the variables, the voltages of the phase coils are found by assuming that there is no current in them, but there is a steady current in the field coil. These voltages are searched for a maximum voltage magnitude between a pair of phase coils, Statements 180 through 188. Once the pair of phase coils are found, the phase angle is set so that the voltage difference across the phase coils is at its peak value. The SCRs of the positive phase coil, Statement 189, and the negative phase coil, Statement 192, are made conductive. The current through the conducting diodes is set to be equal to the seed current in Statements 190 and 193. A procedure is then called, statement 198, that sets up the "bookkeeping" for the calculations. The time derivatives of all the variables in the structure "yo" are calculated by calling the "deriv" procedure in Statement 102 and the results are stored in the "dyo" structure which primes the numerical method for solving the differential equations. The initial conditions are written to the output file by the "report" procedure.

The "advance" procedure in Statement 105 uses the fourth order Runge-Kutta method "rk4" to advance the values in both structures to a later time specified by "tstep." The procedure tests for a change in the conducting state of the SCRs. If there is a change, the procedure performs a sub-step to the time when the change occurs. It then analyzes the conditions of each SCR and changes its conducting state accordingly and advances the solution to the end of the time step. The values of all the variables at the end of the time step are returned in the same structures. This procedure is repeatedly called until the time for the closing of the load SCRs.

The "dump" procedure in Statement 111 writes all the values in the structure to a special file for diagnosis, or the data may be used to pick up the simulation for later times. One possible use is to dump the data just before a load is connected to the compulsator. The data can then be used to simulate various rail gun or other loads without recalculating the self-excitation mode of the compulsator. This program does not have this feature.

Statements 113 through 142 are a work in progress and are subject to change. When the SCRs to the load are closed, there are significant changes in the time derivatives of the currents, accompanied by a change in the voltages across all the coils. Some combination of the nonconducting SCRs will become conductive under these new conditions. The problem is finding the correct combination. The strategy used here is to close all the load SCRs that could conduct to the load and to keep the conducting state of the rest of the SCRs the same. Using this configuration, the currents and voltages are calculated for a time step later, Statement 124, when the state of the SCRs is examined by the "check" procedure in Statement 132 and a new combination is suggested. The currents are then recalculated, using the original currents and the new combination, and checked again. This is repeated until the currents and the voltages are consistent with the conducting state of the SCRs. If a consistent configuration is not found within three attempts, the program will signal a "reconfiguration error" in Statement 127 and will terminate. Reconfiguration errors may occur with this strategy. When they do occur, it is common practice to change the closing time of the SCRs "tclose" by at least a time step or change the time step and run the program again. Other more robust strategies are now being considered and tested. Statements 144 through 151 cover the time that the compulsator is driving the load. When the time "t" is greater than the opening time "topen," Statement 145, the load SCRs are not allowed to become conductive by setting "yo->load" to "FALSE." The program again has some difficulties when the last conducting SCR to the load becomes nonconducting. This sudden transition from a conducting state to a nonconducting state again causes a large change in the time derivatives of the currents, accompanied by large changes in voltages. Finding the correct

conducting states of the SCRs after this time is not as important because this is usually the end of the simulation.

The "report" procedure, Statements 152 through 153, can be modified to produce an output file of any of the variables in the structure "s" in any format or to do other calculations. In this case, the procedure simply reports the time, the field and the load currents.

4. SUBROUTINES

The discussions in the previous section of some of the subroutines were limited to the necessary details to edit the program for other compulsators and rail guns and how to report the results in a desirable format. These subroutines are presented again with further details, and the details of other subroutines not explicitly used in the main program are also given here. This should allow one to modify them to include features that were ignored: the presence of compensating windings, reverse recovery current of the SCRs, a gating schedule for the SCRs, etc.

The first procedure to be discussed is "setup," which must be called as soon as the conducting states of the diodes have been determined or when there is any change in the conducting states of the diodes. Given the conducting state of the SCRs in the form of the "config" structure, "setup" first constructs various arrays of indices, Statements 199 through 293, that are used to construct the Kirchhoff matrix and the matrix that sums the voltage drops in the circuit, "loop." These arrays are also used for various bookkeeping tasks. One task is to retrieve the currents stored in a "state" structure and to form a list of the basic variables of the differential equations. Another task is to do the reverse, i.e., store a list of the basic variables or their time derivatives back into the "state" structure. As an example, the values for these arrays and indices in the following discussion are for the state of conduction of the SCRs in Figure 8. "nt_d" is the total number of conducting diodes or SCRs, which is four, and "np_d" is the number of diodes conducting to the positive bus bar and to the load, which is two. The array "diode" is a map between an assigned number for a conducting diode (the index of the array) and the number of the phase coil that it is connected to (the value of its element). Because the numbering in the C language starts with zero, the array element $\text{diode}[0] = 0$ means that diode #0 is connected to phase coil #1 in Figure 8. Therefore, $\text{diode}[0] = 0$, $\text{diode}[1] = 0$, $\text{diode}[2] = 1$, and $\text{diode}[3] = 2$ are referring to diodes I_1^+ , I_1^- , I_2^- , and I_3^- , respectively, in Figure 8. Associated with this array are two other arrays, "b_field" and "b_load." A "TRUE" value in the "b_field" array means that the diode is conducting to or from the field coil. A "TRUE" value in the "b_load" array means that the diode is conducting to the load. Thus, the "b_field" array has the

following entries: $b_field[0] = TRUE$, $b_field[1] = FALSE$, $b_field[2] = TRUE$, and $b_field[3] = TRUE$. The “ b_load ” array is the complement of the “ b_field ” array: $b_load[0] = FALSE$, $b_load[1] = TRUE$, $b_load[2] = FALSE$, and $b_load[3] = FALSE$. The “ $index$ ” array is a list of the phase coil numbers that are conducting a current. Any phase coil with no current will not be listed. Thus, $index[0] = 0$, $index[1] = 1$, and $index[2] = 2$. “ nt_phs ” is the total number of phase coils that are carrying a current, which is three in this example. “ np_phs ” is the number of phase coils that are either conducting current to the load or to the positive bus bar, which means that $np_phs = 1$.

Statements 234 through 239 test for a phase coil having a load diode and a negative diode, the one connected to the negative bus bar, which are conducting at the same time. This condition occurs when the current through an inductive load is decreasing, which may cause the voltage across the load to be less than the voltage of the negative bus bar, resulting in a forward voltage bias on a load diode. Because this program assumes that all the SCRs are gated to act as diodes, it is assumed that the load diode will act as a diode, even though it is actually an SCR and could be nonconducting during these circumstances.

“Setup” then constructs the Kirchhoff’s matrix, “ kir ” in Statements 241 through 264, and the loop matrix, “ $loop$ ” in Statements 265 through 293 that are declared in Statement 39. Equation 14 shows an example of the Kirchhoff matrix, and Equation 17 shows an example of the “ $loop$ ” matrix.

The “advance” procedure, Statements 294 through 344, advances the values of the state variables stored in the structure “ y ” and their time derivatives “ dy ” by a time step “ ts .” The first procedure that is called uses a Runge-Kutta method, “ $rk4$,” in Statement 308 to calculate the new state variables “ yn ” from the old state variables “ y ,” its derivatives “ dy ,” and a time step “ ts .” “ $rk4$ ” does not return the time derivative of the new state variables, but they are calculated by calling “ $deriv$ ” in statement 309. The next procedure to be called is “ $check$ ” in the conditional part of statement 310, which tests for a change in the conducting state of the diodes. If it detects a change in the conducting state of a diode, it returns a “ $TRUE$ ” value, and the body of the “ if ” statement will then be executed. This procedure also returns a new configuration for the diodes “ c_n ” and an estimate of the time “ dt ” when the change of the configuration had occurred in units of the time step “ ts .” A new set of state variables and their derivatives is found for this approximate time, Statements 315 and 316, which are copied to the original structures, Statements 317 and 318. The new configuration that was found when the “ $check$ ” procedure was called in Statement 411 is adopted, and a new set of state variables and their derivatives is found

for a full time step after the approximate time, Statements 319 through 332, and the configuration is once again checked in Statement 333. If necessary, Statements 323 through 333 will be repeated at most three times or until the "check" subroutine does not report a change in the conducting states of the diodes. If these statements are repeated three times, it is assumed that the conducting states of the diodes cannot be found. The program then dumps the data and terminates. This repetition covers the possibility that the conducting state of more than one diode may have changed during this time step. Even though the "check" procedure tests for a change in the conducting state of all the diodes, it does so under the assumption that the conducting state of all the diodes does not change. All this procedure does is to test if a diode has a reverse current or a forward voltage bias. It does not consider the fact that a change in the conducting state of a diode will cause a discontinuous change in the current derivatives and a discontinuous change in the voltages, which may change the test conditions. Furthermore, if more than one diode changes its conducting state at different times during the time step, only the earliest time is used. In spite of this fact, a consistent configuration can be found by repeating the calculation and testing if the configuration needs changing. Once a new consistent configuration is found, it is adopted (Statements 334 through 340), and the solution is continued to the end of the present time step (Statements 341 and 342). The new set of state variables and their derivatives is then copied to the original structures before returning to the main program.

To test for a change in the conducting state of the diodes, the "check" procedure, Statements 623 through 678, first calls the "volts" procedure in Statement 629. This procedure calculates the voltages across each phase coil, the positive bus bar, the negative bus bar, and the load at the beginning of the time step. It is called again in Statement 630 to calculate these voltages at the end of the time step. After some variables are initialized, Statements 631 through 633, the diodes on each phase coil are tested, Statements 638 through 676. The original conducting state of the diodes for a given phase coil is copied to another in Statements 635 through 637 which is returned at the end of the procedure. The first diode to be examined is the one connected to the positive bus bar, Statements 638 through 643. If it is conducting, it is then tested for a negative current statement 638. If the current is negative, the "test" flag is set to "TRUE," signaling that a change of any kind has been detected. The approximate time for a zero current through the diode is found by linearly interpolating the diode current at the beginning of the time step "so->a.prec[i]" and the current at the end of the time step "sn->a.prec[i], Statement 640. The state of the diode is set to "FALSE," Statement 641, and its current is set to zero, Statement 642. This estimated time is tested for a minimum. If it is less than the minimum time so far, it becomes the new minimum, Statement 643. If this diode was not conducting, it is tested for a forward bias voltage, Statement 644. If it is forward biased, the "test" flag is set to

“TRUE” in Statement 645. The approximate time that the voltage across the diode is zero is estimated by linearly interpolating the voltages of the phase coil “vo.phs[i]” and the positive bus bar “vo.plus” at the beginning of the time step, and the voltages of the phase coil “vn.phs[i]” and the positive bus bar “vn.vplus” at the end of the time step. The conducting state of the diode is set to “TRUE” and its current is set to zero. If the estimated time is less than the minimum time until this point, it becomes the new minimum. The conducting state of the diode to the negative bus bar and the conducting state of the diode to the load are examined and recorded in a similar manner. At the end, the structure “cn” has all the changes in the conducting states of the diodes and is accessible to the calling program, as well as the minimum time when a change had occurred. The “check” procedure returns the value of “test” which is “TRUE” when there is any change or “FALSE” when there is no change in the conducting state of a diode.

The “deriv” procedure, Statements 462 through 604, calculates the time derivatives of all the state variables that have been a major subject of this report, and in which many of the equations are encoded. Statements 477 through 483 calculate the derivatives of the mechanical variables, and Statements 485 through 489 calculate the rate at which the temperature of the resistors increases with time. The rest of the procedure calculates the derivatives of the currents. Starting with Statement 490, the currents that will be the basic variables are recalled from the state structure and organized into a vector “di.” After various matrices are set to zero, Statements 495 through 604, the matrices that were illustrated in the previous sections are constructed. The first matrix to be constructed is “rmf,” Statements 513 through 534, which contains the resistances of all the elements, the products of the angular frequency, and the gradient of the mutual inductance between the phase coils and the field coil and the product of the velocity of the projectile and the inductance gradient of the load. An example of this matrix is the second 9x9 matrix in Equation 16. This matrix is then multiplied on the right by the Kirchhoff’s matrix “kir” and on the left by the “loop” matrix. The resulting matrix “r” is the $\sum_{n,i} (R_{i,m} + \dot{M}_{i,m}) \mathcal{Z}_{m,k}$ term in Equation 18. A matrix of the inductances of the elements “temp” is constructed in Statements 544 through 564. An example of this matrix is the first 9x9 matrix in Equation 16. It, too, is multiplied on the right by “kir” and on the left by “loop.” The inverse of the resulting matrix “h” is found by calling a matrix inverting procedure “minv” to give the $(\sum_{j,i} M_{i,m} \mathcal{Z}_{m,n})^{-1}$ term in Equation 18. The matrix “h,” the matrix “r,” and the vector of the basic variables “di” are multiplied together in Statements 575 through 580 to yield the time derivative of the basic variables “di_dot.” This vector, “di_dot,” is multiplied by the “kir,” Statements 588 through 591, to generate a vector of the derivatives of all the currents in the circuit. The values in this vector are then stored into the appropriate locations in the structure

“ds,” Statements 592 through 604. This results in a structure of the same type as the data structure “s,” but values stored in the returned structure “ds” contain the time derivative of the variables.

Using these variables and their time derivatives at a given time, “rk4” calculates the variables at a later time as specified by the time step. This procedure uses the Euler method, which is most elementary method for numerically solving first order differential equations of the form $y' = f(t,y)$, in which y' is the derivative of y with respect to t . The Euler method is simply $y(t+h) = y(t) + h F(t,y(t))$ in which h is the time step. This method is encoded in Statements 474 through 536. Because the Euler method is only accurate to the first order of the step size, it is seldom used by itself. It is a basis, however, for other methods that are accurate to higher orders. One method is the fourth order Runge-Kutta which is accurate to the fourth order. There are other Runge-Kutta methods that are accurate to higher orders, but the fourth order Runge-Kutta has proved to be economical and practical in practice. This Runge-Kutta [4], Statements 375 through 423, uses the Euler method, Statements 424 through 442, to find the derivatives of y at four points:

$$\begin{aligned}
 k_a &= F(t_n, y_n) \\
 k_b &= F\left(t_n + \frac{h}{2}, y_n + \frac{h}{2} k_a\right) \\
 k_c &= F\left(t_n + \frac{h}{2}, y_n + \frac{h}{2} k_b\right) \\
 k_d &= F(t_n + h, y_n + h k_c)
 \end{aligned} \tag{20}$$

A weighed average of these derivatives is used to estimate the value of y_{n+1} at the time $t_n + h$:

$$y_{n+1} = y_n + \frac{h}{6}(k_a + 2k_b + 2k_c + k_d) \tag{21}$$

The “volts” procedure, Statements 605 through 622, calculates the voltages at various points in the circuit. Statements 612 through 615 find the voltages across each phase coil with respect to the central ground point, even when a phase coil may not be carrying a current. Statement 616 calculates the voltage across the load. The voltage of the positive bus bar is found by first searching for a diode that is conducting a current to it, calculating the voltage drop across the diode, and subtracting it from the voltage across its phase coil (Statements 617 through 619). A similar procedure is used to determine the voltage of the negative bus bar (Statements 620 through 622). The “check” procedure uses these results to test for a forward bias voltage on a nonconducting diode.

5. DISCUSSION AND RESULTS

Although this program produces results, it is a work in progress in which some areas can be improved. One area of improvement is the testing for the conducting state of the SCRs, especially when the load current just starts and when the load current ends. Although the present strategy works for this compulsator, it may need improvement for compulsators that have more phase coils. When there are more phase coils, there are more SCRs, which means there is an increased chance that two or more SCRs are changing state at the same time or very close to the same time. The present strategy may not resolve this situation. Another possible area for improvement is to use a Runge-Kutta routine that changes the step size and tests the quality of the solution. When these routines detect that the variables are slowly changing during a time step by some criteria, they will increase the next step size, and they will also decrease the next step size when the variables are changing too much. The routines are usually faster than the ones that keep the same time step, because they quickly step over regions where the variables are not changing significantly, while the fixed step size routine will simply step through these regions. These routines that do vary the step size assume that the transition from the regions where the variables are changing slowly to the regions where the variables are changing quickly or vice versa is continuous and detectable. The currents in the compulsator, however, do not have this property. Because the currents vary smoothly when the conducting state of the diodes does not change, these routines may continue to increase the step size during this time. By the time a diode does change its conducting state and the currents are changed very quickly at some unforeseen time, the step size may be large. These routines may then spend a lot of effort in decreasing the step size and testing when the currents have suddenly started to quickly change. This effort may cancel the advantage of changing the step size. Thus, a fixed time step is used in this program. Other Runge-Kutta routines test the quality of the solution, which is desirable. Because this program does not have an intrinsic test for the quality of the solution, it should be repeated several times with decreasing step sizes and the results examined. If the results do not significantly change when the step size is decreased, then the quality of the results is good.

Because the SCRs are modeled by an open or closed switch, the characteristics of the SCRs are ignored. These characteristics are the voltage drop across the SCRs when they are conducting, and the recovery characteristics when the current through them is starting to reverse. If it is assumed that the characteristics of the SCRs should not have a significant effect on the performance of the compulsator, then the reason for including them would be to study other aspects of the compulsator: voltage transits, snubber circuits, etc. Including these characteristics may solve the configuration problem in this program because they will make the discontinuous

change of an SCR in a conducting state to a nonconducting state into a continuous transition. By taking small time steps, this transition may allow "rk4" to follow a solution through this period and determine the conducting state of all the other SCRs.

If a compulsator is powering a rail gun, it may be desirable to shape or control the current pulse so that the muzzle velocity of the projectile is at a maximum without over-stressing the rail gun. By gating the SCRs to the load at different times, it is possible to change the shape of the current pulse. This program does not have this pulse-shaping capability, but it could be modified to use a gating schedule for the load SCRs and find the resulting current pulse.

The program was validated in two ways. First, the total energy of the system, Equation 4, was calculated in a previous version, and it was observed that the total energy was conserved. Second, the results of the program agree very well with the results reported in Kitzmiller [3]. As an example, one of the results is duplicated with the following prompted input:

```
Output File: results.out
Field current (a) = 6000.0
Initial rpm = 10000.0
Temperature (C) = 20.0
Time step = 5.0e-06
Close Time = 0.0756
Open Time = 0.0796
End Time = 0.0820
Report Skip = 20
Report Skip = 20
```

The program run time is about 13 seconds on a 100-MHz Pentium™ when compiled with Visual C++® 5.0 when the "release" configuration was selected. The field coil current during the self-excitation is compared to the results of Kitzmiller [3] (the squares in Figure 9) after 4 ms was subtracted from Kitzmiller's time scale.

Four milliseconds is the approximate time for the capacitor bank to establish the seed current in the field coil which was included in Kitzmiller's graphs. The rail gun currents are compared in Figure 10 for the time after the current starts.

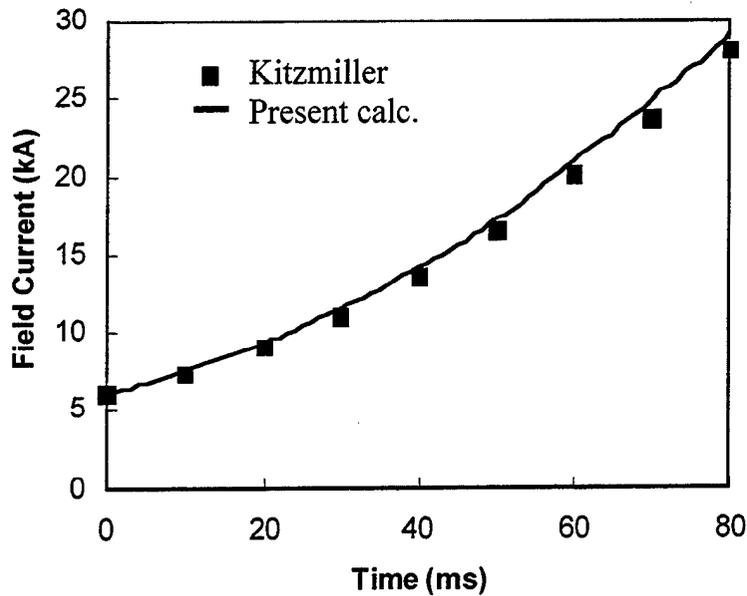


Figure 9. Self-Exciting Field Current.

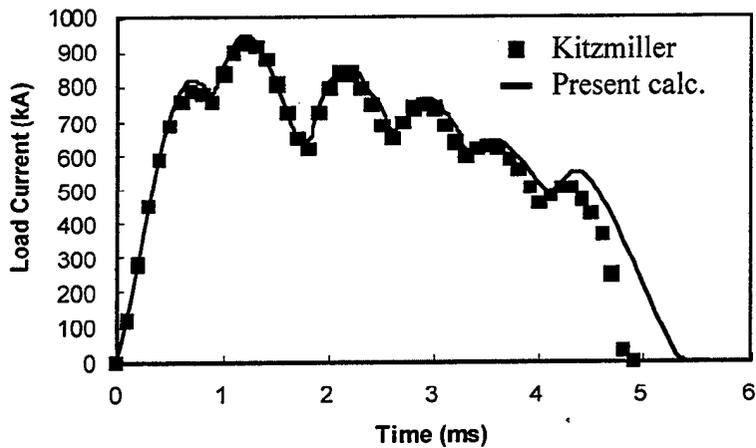


Figure 10. Rail Gun Current.

The closing time was varied until there was a good agreement. This is necessary because the initial conditions of the two calculations are not the same. In the present calculation, the initial conditions were chosen so that an initial phase angle can be determined. Kitzmiller, however, use the initial conditions for a model of the auxiliary capacitor bank that establishes the seed current in the field coil. Thus, this liberty was taken to produce this agreement. The

present calculation gives the final velocity of the projectile as 1248 m/s, while Kitzmiller reports a final velocity of 1200 m/s. Another point of comparison is the field current. Kitzmiller finds that the field current had reached 28 kilo-amperes (kA) in 80 ms. The present calculation finds that the field current had reached 28 kA in about 76 ms. This difference may be attributable to the different ways in which the two programs solve differential equations. Still, this program has nearly reproduced the results of other cases given in Kitzmiller's report, showing that the two programs are close equivalents. This program is now being used to study the SSFTP compulsator as a power supply for coil guns and electromagnetic armor. The results of this study will be given in future reports.

INTENTIONALLY LEFT BLANK

REFERENCES

1. Walls, W.A., M.L. Spann, S.B. Pratap, and J.R. Kitzmiller, "Rotating Machine Development at the University of Texas," *8th IEEE International Pulsed Power Conference*, San Diego, CA, editors R. White and K. Prestwitch, pp 533-536, 1991.
2. Bhader, T.B, and J.D. Bruno, "Transient Response of an Electromagnetic Rail Gun: A Pedagogical Model," ARL-TR-1663, May 1998.
3. Kitzmiller, J., "FTP—Subscale System Performance Predictions," Internal memorandum, CEM Univ. of Texas at Austin.
4. Press, W.H., B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, "Numerical Recipes: The Art of Scientific Computing," Cambridge University Press, Cambridge, 1986.

INTENTIONALLY LEFT BLANK

APPENDIX A
COMPUTER PROGRAM

INTENTIONALLY LEFT BLANK

COMPUTER PROGRAM

```
1: #include <stdio.h>
2: #include <math.h>
3: #include <stdlib.h>
4: FILE *fp;
5: #define PI 3.141592654
   /* Version: This version keeps track of the      */
   /* temperatures of all the resistors.           */
   /* --Start of parameters & function block-----*/
   // SSFTP compulsator driving a rail-gun
   // Number of phases
6: #define PHASE 3
   // Number of poles
7: #define POLE 6
8: // MAX = 3*PHASE + 2
9: #define MAX 11
   // Reference temperature for the resistors
10: #define TREF 20.0
11: struct resistor {
       double ro;      // Resistance at TREF
       double c;      // Temperature coefficient
       double cp;     // Heat capacity (J/g/C)
       double mass;   // Resistors mass (g)
   };
12: const struct resistor rfield = { 0.078, 0.0025,
                                   0.963, 8.44e+04 };
13: const struct resistor rpscr = { 1.27e-04, 0.0045,
                                   0.906, 3.00e+03 };
14: const struct resistor rnscr = { 1.22e-03, 0.0045,
                                   0.906, 3.00e+03 };
15: const struct resistor rlscr = { 1.27e-04, 0.0045,
                                   0.906, 3.00e+03 };
16: const struct resistor rphs[PHASE] =
   { { 4.46e-04, 0.004, 0.963, 3.41e+04},
     { 3.71e-04, 0.004, 0.963, 2.83e+04},
     { 3.41e-04, 0.004, 0.963, 2.60e+04} };
   //Inductance table for the stator
17: const double mut[PHASE][PHASE] =
   { { 1.16e-06, -4.72e-07, -4.35e-07 },
     { -4.72e-07, 1.12e-06, -4.74e-07 },
     { -4.35e-07, -4.74e-07, 1.07e-06 } };
   //Mutual inductance phase coil and field coil
18: const double mf[] = {2.92e-05, 2.68e-05, 2.46e-05 };
   //Angle offset
19: const double os[] = { 0.0, 1.9635, 3.9270 };
20: const double hfield = 2.28e-03; //Field coil
21: const double hpscr = 1.94e-07; //Diode inductance
22: const double hnscr = 2.60e-06; //Diode inductance
23: const double hlscr = 1.94e-07; //Diode inductance
24: const double moi = 2.83333;    //4*Mom. of inertia
                                   //(kg*m*)/POLE/POLE
25: const double mass = 0.640;    //2 * Mass (kg)
```

```

    // The load inductance as a function of a position
26: double hload ( const double x ) {
27:     if ( x < 3.0 ) return 3.6e-07*x + 2.5e-07;
28:     return 13.3e-07;
    }
    // The gradient of the load inductance
29: double dhdx ( const double x ) {
30:     if ( x < 3.0 ) return 3.6e-07;
31:     return 0.0;
    }
    // The rail-gun resistance
32: double rload ( const double x ) {
33:     if ( x < 3.0 ) return 8.33e-05*x + 2.38e-04;
34:     return 4.879e-04;
    }
    /*-----End of parameter & function block-----*/
35: typedef short int BOOL;
36: #define TRUE 1
37: #define FALSE 0
    // Structure for a matrix of real values
38: struct matrix {
        double v[MAX][MAX];
        int r; //Row index dimension
        int c; //Column index dimension
    } rmf, temp, h, r;
    // Structure for a matrix of integer values
39: struct i_matrix {
        short int v[MAX][MAX];
        short int r; // Row index dimension
        short int c; // Column index dimension
    } kir, loop;
40: short int index[PHASE]; // Phase index
41: short int np_phs; // Number of pos. phases
42: short int nt_phs; // Total number phases
43: short int diode[MAX]; // Diode number
44: BOOL b_load[MAX]; // Diode conducting to load
45: BOOL b_field[MAX]; // Diode conducting to field
46: BOOL load_flag; // A diode conducts to load
47: short int np_d; // Number of pos. diodes
48: short int nt_d; // Total conducting diodes
    // Structure for a real vector
49: struct rvec {
        double v[MAX];
        int n;
    };
50: struct mechanic {
        double w; // Angular frequency
        double th; // Phase angle
        double v; // Projectile velocity
        double x; // Projectile position
    };
51: struct thermal {
        double prec[PHASE]; // Pos. diode temp
        double nrec[PHASE]; // Neg. diode temp

```

```

        double lrec[PHASE]; // Load diode temp
        double phs[PHASE]; // Phase coil temp
        double field;      // Field coil temp
    };
52: struct config {
    // Conducting state of the pos. diodes.
    BOOL prec[PHASE];
    // Conducting state of the neg. diodes.
    BOOL nrec[PHASE];
    // Conducting state of the load diodes.
    BOOL lrec[PHASE];
    BOOL load;
}cf; // The configuration
53: struct current {
    // All diode currents are positive
    double prec[PHASE]; // Pos. diode current
    double nrec[PHASE]; // Neg. diode current
    double lrec[PHASE]; // Load diode current
    double phs[PHASE]; // Phase coil current
    double field;      // Field coil current
    double load;       // Load current
};
54: struct state {
    struct mechanic m;
    struct thermal tc;
    struct current a;
    struct config *c; // Active configuration
};
55: struct voltage {
    // Voltage across each phase coil
    double vphs[PHASE];
    // Voltage of the positive busbar.
    double vplus;
    // Voltage of the negative busbar
    double vminus;
    // Voltage across the load.
    double vload;
};
/* ----- main () -----*/
56: void main () {
57: void initiate ( struct state *, double,
                  double, double );
58: void deriv ( const struct state *, struct state * );
59: void setup ( const struct config * );
60: void advance ( struct state *, struct state *,
                 double );
61: void report ( const struct state * , const double );
62: void copy ( struct state *, const struct state *);
63: void rk4 ( struct state *, const struct state *,
             const struct state *, const double );
64: BOOL check ( const struct state *,
                const struct state *, struct state *,
                const struct state *, struct config *,
                double * );

```

```

65: void dump ( const double t, const struct state * );
66: double seed;      //Seed field current (amps)
67: double tamb;     //Ambient Temperature
68: double rpm;      //Initial rpm
69: char fname[40];  //Output file name
70: double t, tstep, tclose, topen, tend;
71: double dt;
72: int i, skip, rc;
73: struct state yo; // State variables
74: struct state dyo; // and derivatives
75: struct state yn;
76: struct state dyn;
77: struct config *pc_o;
78: struct config c_n;
79:     printf ("Output File: ");
80:     scanf ("%s", fname);
81:     fp = fopen ( fname, "w");
82:     printf ("Field current (a) = ");
83:     scanf ("%le", &seed);
84:     printf ("Initial rpm = ");
85:     scanf ("%le", &rpm);
86:     printf ("Temperature (C) = ");
87:     scanf ("%le", &tamb);
88:     printf ("Time step = ");
89:     scanf ("%le", &tstep);
90:     printf ("Close Time = ");
91:     scanf ("%le", &tclose);
92:     printf ("Open Time = ");
93:     scanf ("%le", &topen);
94:     printf ("End Time = ");
95:     scanf ("%le", &tend);
96:     printf ("Report Skip = ");
97:     scanf ("%d", &skip);
98:     rc = 1;
99:     yo.c = &cf;      // The original configuration
100:    t = 0.0;
101:    initiate ( &yo, tamb, seed, rpm );
102:    deriv ( &yo, &dyo );
103:    report ( &yo, t );
104:    while (t < tclose ) {
105:        advance ( &yo, &dyo, tstep );
106:        t += tstep;
107:        if ( rc == skip ) {
108:            report ( &yo, t );
109:            rc = 0;
110:        }
111:        rc++;
112:    }
113:    dump ( t, &yo );
114:    if ( t < tend ) {
115:        for ( i = 0; i < PHASE; i++) {
116:            c_n.prec[i] = yo.c->prec[i];
117:            c_n.nrec[i] = yo.c->nrec[i];
118:            c_n.lrec[i] = yo.c->lrec[i];

```

```

    }
117:     c_n.load = TRUE;
118:     pc_o = yo.c;
119:     yo.c = &c_n;
120:     i = 0;
121:     do {
122:         setup ( &c_n );
123:         deriv ( &yo, &dyo );
124:         rk4 ( &yn, &yo, &dyo, tstep );
125:         deriv ( &yn, &dyn );
126:         if ( i == 3 ) {
127:             printf ("Reconfiguration Error\n");
128:             yo.c = pc_o;
129:             dump ( t, &yo );
130:             exit( 0 );
        }
131:         i++;
132:     } while ( check ( &yo, &dyo, &yn,
                       &dyn, &c_n, &dt ) );

133:     yo.c = pc_o;
134:     dyo.c = pc_o;
135:     for ( i = 0; i < PHASE; i++ ) {
136:         yo.c->prec[i] = c_n.prec[i];
137:         yo.c->nrec[i] = c_n.nrec[i];
138:         yo.c->lrec[i] = c_n.lrec[i];
    }

139:     yo.c->load = c_n.load;
140:     report ( &yo, t );
141:     printf ("Report Skip = ");
142:     scanf ("%d", &skip);
    }

143:     rc = 1;
144:     while ( t < tend && load_flag ) {
145:         if ( t > topen ) yo.c->load = FALSE;
146:         advance ( &yo, &dyo, tstep );
147:         t += tstep;
148:         if ( rc == skip ) {
149:             report ( &yo, t );
150:             rc = 0;
        }
151:         rc++;
    }
} // End of main ()

152: void report ( const struct state *s,
               const double time ) {
153:     fprintf ( fp, " %8.3f %8.3f %8.3f\n",
               time*1000.0, s->a.field/1000.0,
               s->a.load/1000.0 );
    }
// Initiates struct state *t
154: void initiate (struct state *t, double ta,
                 double af, double rpm ) {
155: void setup ( const struct config *pc );
156: double mag, arg, magmax;

```

```

157: double xa, ya, dx, dy;
158: int i, j, imax, jmax;
159:     t->m.th = 0.0;
160:     t->m.w = PI*POLE*rpm/60.0;
161:     t->m.v = 0.0;
162:     t->m.x = 0.0;
163:     t->tc.field = ta;
164:     t->c->load = FALSE;
165:     magmax = -1.0;
166:     for ( i = 0; i < PHASE; i++ ) {
167:         t->c->prec[i] = FALSE;
168:         t->c->nrec[i] = FALSE;
169:         t->c->lrec[i] = FALSE;
170:         t->a.prec[i] = 0.0;
171:         t->a.nrec[i] = 0.0;
172:         t->a.lrec[i] = 0.0;
173:         t->a.phs[i] = 0.0;
174:         t->tc.prec[i] = ta;
175:         t->tc.nrec[i] = ta;
176:         t->tc.lrec[i] = ta;
177:         t->tc.phs[i] = ta;
178:         xa = mf[i]*cos( os[i] );
179:         ya = mf[i]*sin( os[i] );
180:         for ( j = i+1; j < PHASE; j++ ) {
181:             dx = xa - mf[j]*cos( os[j] );
182:             dy = ya - mf[j]*sin( os[j] );
183:             mag = sqrt( dx*dx + dy*dy );
184:             if ( mag > magmax ) {
185:                 imax = i;
186:                 jmax = j;
187:                 magmax = mag;
188:                 arg = atan2 ( dy, dx );
189:             }
190:         }
191:     }
192:     t->c->prec[jmax] = TRUE;
193:     t->a.prec[jmax] = af;
194:     t->a.phs[jmax] = af;
195:     t->c->nrec[imax] = TRUE;
196:     t->a.nrec[imax] = af;
197:     t->a.phs[imax] = -af;
198:     t->m.th = -arg;
199:     t->a.load = 0.0;
200:     t->a.field = af;
201:     setup ( t->c );
202: } // End of initiate ()
203: void setup ( const struct config *c ) {
204: int i, j;
205:     // Zero out the matrices
206:     for ( i = 0; i < MAX; i++ )
207:         for ( j = 0; j < MAX; j++ ) {
208:             kir.v[i][j] = 0;
209:             loop.v[i][j] = 0;
210:         }

```

```

205:     kir.r = 0;
206:     kir.c = 0;
207:     loop.r = 0;
208:     loop.c = 0;
209:     nt_d = 0;
210:     nt_phs = 0;
211:     load_flag = FALSE;
      // Diodes on the positive phases
212:     for (i = 0; i < PHASE; i++) {
213:         if ( !c->nrec[i] &&
              (c->prec[i] || c->lrec[i]) ) {
214:             if ( c->prec[i] ) {
215:                 diode[nt_d] = nt_phs;
216:                 b_field[nt_d] = TRUE;
217:                 b_load[nt_d] = FALSE;
218:                 nt_d++;
                }
219:             if ( c->lrec[i] ) {
220:                 load_flag = TRUE;
221:                 diode[nt_d] = nt_phs;
222:                 b_field[nt_d] = FALSE;
223:                 b_load[nt_d] = TRUE;
224:                 nt_d++;
                }
225:             index[nt_phs++] = i;
            }
        }
226:     np_d = nt_d;
227:     np_phs = nt_phs;
      // Diodes on the negative phases
228:     for (i = 0; i < PHASE; i++)
229:         if ( c->nrec[i] ) {
230:             diode[nt_d] = nt_phs;
231:             b_field[nt_d] = TRUE;
232:             b_load[nt_d] = FALSE;
233:             nt_d++;
234:             if ( c->lrec[i] ) {
235:                 load_flag = TRUE;
236:                 diode[nt_d] = nt_phs;
237:                 b_field[nt_d] = FALSE;
238:                 b_load[nt_d] = TRUE;
239:                 nt_d++;
                }
240:             index[nt_phs++] = i;
        }
      // Construct the Kirchhoff matrix
241:     for (j = 0; j < np_d; j++) {
242:         kir.v[diode[j]][j] = 1;
          // Field current
243:         if ( b_field[j] ) kir.v[nt_phs][j] = 1;
          // Diode identity
244:         kir.v[nt_phs + 1 + j][j] = 1;
245:         if ( b_load[j] )
            kir.v[nt_phs + nt_d + 1][j] = 1;

```

```

}
246:   for ( j = np_d; j < nt_d; j++ ) {
247:       if ( b_field[j] ) kir.v[diode[j]][j] = -1;
248:       else kir.v[diode[j]][j] = 1;
           // Diode identitiy
249:       kir.v[nt_phs + 1 + j][j] = 1;
250:       if ( b_load[j] )
           kir.v[nt_phs + nt_d + 1][j] = 1;
}
251: kir.r = nt_phs + nt_d + 1;
252: if ( load_flag ) kir.r++;
253: kir.c = nt_d;
           // Eliminate the first negative diode current
254: for ( i = 0; i < np_d; i++ )
255:     if ( b_field[i] ) {
256:         kir.v[np_phs][i] += -1;
257:         kir.v[nt_phs + np_d + 1][i] += 1;
}
258: for ( i = np_d; i < nt_d; i++ )
259:     if ( b_field[i] ) {
260:         kir.v[np_phs][i] += 1;
261:         kir.v[nt_phs + np_d + 1][i] += -1;
}
           // Eliminate the zero column from the matrix
262: kir.c--;
263: for ( i = 0; i < kir.r; i++)
264:     for ( j = np_d; j < kir.c; j++)
           kir.v[i][j] = kir.v[i][j+1];
// Construct the loop matrix
           // The loops through the pos. phase currents
265: for (i = 0; i < np_d; i++) {
           // Loop through the phase coil,
           //field coil & first neg.
266:     if ( b_field[i] ) {
           // The phase coil
267:         loop.v[loop.r][diode[i]] = 1;
           // The field coil column
268:         loop.v[loop.r][nt_phs] = 1;
           // The pos. field diode
269:         loop.v[loop.r][nt_phs + i + 1] = 1;
           // The first neg. phase
270:         loop.v[loop.r][np_phs] = -1;
           // The neg. field diode
271:         loop.v[loop.r][nt_phs + np_d + 1] = 1;
272:         loop.r++;
}
}
           // Loop through the first pos.
           // field coil & rest of neg. phase
273: j = 0;
           //Find first pos. phase to field
274: while ( !b_field[j] ) j++;
           //Index to the pos. phase.
275: j = diode[j];

```

```

276:     for ( i = np_d + 1; i < nt_d; i++) {
277:         if ( b_field[i] ) {
                // First phase coil
278:         loop.v[loop.r][j] = 1;
                // Field coil
279:         loop.v[loop.r][nt_phs] = 1;
                // First pos. field diode
280:         loop.v[loop.r][nt_phs + j + 1] = 1;
                // Neg. phase coil
281:         loop.v[loop.r][diode[i]] = -1;
                // Neg. field diode
282:         loop.v[loop.r][nt_phs + i + 1] = 1;
283:         loop.r++;
                }
        }
284:     loop.c = nt_phs + nt_d + 1;
                // Loop through the load
285:     if ( load_flag ) {
286:         for ( i = 0; i < nt_d; i++) {
287:             if ( b_load[i] ) {
288:                 loop.v[loop.r][diode[i]] = 1;
289:                 loop.v[loop.r][nt_phs + i + 1] = 1;
290:                 loop.v[loop.r][loop.c] = 1;
291:                 loop.r++;
                }
            }
292:         loop.c++;
        }
293:     return;
} // End of setup()
// Advance the state structures by a time step
294: void advance ( struct state *y, struct state *dy,
                double ts ) {
295: void rk4 ( struct state *, const struct state *,
                const struct state *, const double );
296: void deriv ( const struct state *, struct state * );
297: void copy ( struct state *, const struct state * );
298: void setup ( const struct config * );
299: BOOL check ( const struct state *,
                const struct state *,
                struct state *, const struct state *,
                struct config *, double * );
300: void dump ( const double , const struct state * );
301: struct state yn;
302: struct state dyn;
303: struct config c_n;
304: struct config *co; // Original config.
305: double dt;
306: int i, cnt;
307:     co = y->c;
308:     rk4 ( &yn, y, dy, ts );
309:     deriv ( &yn, &dyn );
310:     if ( check ( y, dy, &yn, &dyn, &c_n, &dt ) ) {
                // Advance to the approx. time of

```

```

        // change using the original configuration
311:     if ( dt < 0.0 || dt > 1.0 ) {
312:         printf ("dt = %e out of bounds\n", dt);
313:         dump ( dt, y );
314:         exit( 0 );
    }
315:     rk4 ( &yn, y, dy, ts*dt );
316:     deriv ( &yn, &dyn );
    //The new becomes the original
317:     copy ( y, &yn );
318:     copy ( dy, &dyn );
    // Try advancing a full time step
    // using the new config.
319:     y->c = &c_n;
320:     dy->c = &c_n;
321:     cnt = 0;
322:     do {
323:         setup ( &c_n );
324:         deriv ( y, dy );
325:         rk4 ( &yn, y, dy, ts );
326:         deriv ( &yn, &dyn );
327:         if ( cnt == 3 ) {
328:             printf ("Reconfiguration Error\n");
329:             y->c = co;
330:             dump ( 0.0, y );
331:             exit( 0 );
        }
332:         cnt++;
333:     } while ( check (y, dy, &yn,
        &dyn, &c_n, &dt) );

334:     y->c = co;
335:     dy->c = co;
336:     y->c->load = c_n.load;
337:     for (i = 0; i < PHASE; i++) {
338:         y->c->prec[i] = c_n.prec[i];
339:         y->c->nrec[i] = c_n.nrec[i];
340:         y->c->lrec[i] = c_n.lrec[i];
    }
    // Advance to the end of the time step
341:     rk4 ( &yn, y, dy, ts*(1.0-dt) );
342:     deriv ( &yn, &dyn );
    }

343:     copy ( y, &yn ); //The new becomes the original
344:     copy ( dy, &dyn );
    } // End of advance ()
    // Generates a dump file for the state structure
345: void dump ( const double t,
    const struct state *s ) {
346: FILE *fdmp;
347: int i;
348:     fdmp = fopen ("DUMP.OUT", "w");
349:     fprintf (fdmp, " %24.16e\n", t);
350:     fprintf (fdmp, " %24.16e\n", s->m.w);
351:     fprintf (fdmp, " %24.16e\n", s->m.th);

```

```

352:     fprintf (fdmp, " %24.16e\n", s->m.v);
353:     fprintf (fdmp, " %24.16e\n", s->m.x);
354:     for (i = 0; i < PHASE; i++) {
355:         fprintf (fdmp, " %24.16e", s->tc.prec[i]);
356:         fprintf (fdmp, " %24.16e", s->tc.nrec[i]);
357:         fprintf (fdmp, " %24.16e", s->tc.lrec[i]);
358:         fprintf (fdmp, " %24.16e\n", s->tc.phs[i]);
    }
359:     fprintf (fdmp, " %24.16e\n", s->tc.field);
360:     for (i = 0; i < PHASE; i++) {
361:         fprintf (fdmp, " %24.16e", s->a.prec[i]);
362:         fprintf (fdmp, " %24.16e", s->a.nrec[i]);
363:         fprintf (fdmp, " %24.16e", s->a.lrec[i]);
364:         fprintf (fdmp, " %24.16e\n", s->a.phs[i]);
    }
365:     fprintf (fdmp, " %24.16e\n", s->a.field);
366:     fprintf (fdmp, " %24.16e\n", s->a.load);
367:     for (i = 0; i < PHASE; i++)
        fprintf (fdmp, " %1d", s->c->prec[i]);
368:     fprintf (fdmp, "\n");
369:     for (i = 0; i < PHASE; i++)
        fprintf (fdmp, " %1d", s->c->nrec[i]);
370:     fprintf (fdmp, "\n");
371:     for (i = 0; i < PHASE; i++)
        fprintf (fdmp, " %1d", s->c->lrec[i]);
372:     fprintf (fdmp, "\n");
373:     fprintf (fdmp, " %1d\n", s->c->load );
374:     fclose (fdmp);
    } // End of dump
    // The fourth-order Runge-Kutta
375: void rk4 ( struct state *sn, const struct state *so,
            const struct state *dso,
            const double h) {
376: struct state dym;
377: struct state yt;
378: struct state dyt;
379: void deriv ( const struct state *, struct state * );
380: void euler ( struct state *, const struct state *,
            const struct state *, const double );
381: double hh, hs;
382: int i;
383:     hh = h/2.0;
384:     hs = h/6.0;
385:     sn->c = so->c;
        // First step
386:     euler ( &yt, so, dso, hh );
        // Second step
387:     deriv ( &yt, &dyt );
388:     euler ( &yt, so, &dyt, hh );
        // Third step
389:     deriv ( &yt, &dym );
390:     euler ( &yt, so, &dym, h );
        // Add the derivatives
391:     dym.m.x += dyt.m.x;

```

```

392:     dym.m.v += dyt.m.v;
393:     dym.m.w += dyt.m.w;
394:     dym.m.th += dyt.m.th;
395:     dym.a.load += dyt.a.load;
396:     dym.a.field += dyt.a.field;
397:     dym.tc.field += dyt.tc.field;
398:     for (i = 0; i < PHASE; i++) {
399:         dym.tc.prec[i] += dyt.tc.prec[i];
400:         dym.tc.nrec[i] += dyt.tc.nrec[i];
401:         dym.tc.lrec[i] += dyt.tc.lrec[i];
402:         dym.tc.phs[i] += dyt.tc.phs[i];
403:         dym.a.prec[i] += dyt.a.prec[i];
404:         dym.a.nrec[i] += dyt.a.nrec[i];
405:         dym.a.lrec[i] += dyt.a.lrec[i];
406:         dym.a.phs[i] += dyt.a.phs[i];
    }
    // Fourth step
407:     deriv ( &yt, &dyt );
408:     sn->m.x = so->m.x +
        hs*(dso->m.x + dyt.m.x + 2.0*dym.m.x);
409:     sn->m.v = so->m.v +
        hs*(dso->m.v + dyt.m.v + 2.0*dym.m.v);
410:     sn->m.w = so->m.w +
        hs*(dso->m.w + dyt.m.w + 2.0*dym.m.w);
411:     sn->m.th = so->m.th +
        hs*(dso->m.th + dyt.m.th + 2.0*dym.m.th);
412:     sn->a.load = so->a.load
        + hs*(dso->a.load + dyt.a.load
        + 2.0*dym.a.load);
413:     sn->a.field = so->a.field
        + hs*(dso->a.field + dyt.a.field
        + 2.0*dym.a.field);
414:     sn->tc.field = so->tc.field
        + hs*(dso->tc.field + dyt.tc.field
        + 2.0*dym.tc.field);
415:     for (i = 0; i < PHASE; i++) {
416:         sn->tc.prec[i] = so->tc.prec[i]
            + hs*(dso->tc.prec[i] + dyt.tc.prec[i]
            + 2.0*dym.tc.prec[i]);
417:         sn->tc.nrec[i] = so->tc.nrec[i]
            + hs*(dso->tc.nrec[i] + dyt.tc.nrec[i]
            + 2.0*dym.tc.nrec[i]);
418:         sn->tc.lrec[i] = so->tc.lrec[i]
            + hs*(dso->tc.lrec[i] + dyt.tc.lrec[i]
            + 2.0*dym.tc.lrec[i]);
419:         sn->tc.phs[i] = so->tc.phs[i]
            + hs*(dso->tc.phs[i] + dyt.tc.phs[i]
            + 2.0*dym.tc.phs[i]);
420:         sn->a.prec[i] = so->a.prec[i]
            + hs*(dso->a.prec[i] + dyt.a.prec[i]
            + 2.0*dym.a.prec[i]);
421:         sn->a.nrec[i] = so->a.nrec[i]
            + hs*(dso->a.nrec[i] + dyt.a.nrec[i]
            + 2.0*dym.a.nrec[i]);

```

```

422:         sn->a.lrec[i] = so->a.lrec[i]
           + hs*(dso->a.lrec[i] + dyt.a.lrec[i]
           + 2.0*dym.a.lrec[i]);
423:         sn->a.phs[i] = so->a.phs[i]
           + hs*(dso->a.phs[i] + dyt.a.phs[i]
           + 2.0*dym.a.phs[i]);
    }
    } // End of rk4()
    // Euler method advances the state by a time step
424: void euler (struct state *sn,
               const struct state *so,
               const struct state *dso,
               const double h ) {
425: int i;
426:     sn->c = so->c;
           // The mechanical part
427:     sn->m.x = so->m.x + h * dso->m.x;
428:     sn->m.v = so->m.v + h * dso->m.v;
429:     sn->m.th = so->m.th + h * dso->m.th;
430:     sn->m.w = so->m.w + h * dso->m.w;
431:     sn->tc.field = so->tc.field + h * dso->tc.field;
432:     sn->a.field = so->a.field + h * dso->a.field;
433:     sn->a.load = so->a.load + h * dso->a.load;
434:     for (i = 0; i < PHASE; i++) {
           // The temperatures
435:         sn->tc.prec[i] = so->tc.prec[i]
           + h * dso->tc.prec[i];
436:         sn->tc.nrec[i] = so->tc.nrec[i]
           + h * dso->tc.nrec[i];
437:         sn->tc.lrec[i] = so->tc.lrec[i]
           + h * dso->tc.lrec[i];
438:         sn->tc.phs[i] = so->tc.phs[i]
           + h * dso->tc.phs[i];
           // The currents
439:         sn->a.prec[i] = so->a.prec[i]
           + h * dso->a.prec[i];
440:         sn->a.nrec[i] = so->a.nrec[i]
           + h * dso->a.nrec[i];
441:         sn->a.lrec[i] = so->a.lrec[i]
           + h * dso->a.lrec[i];
442:         sn->a.phs[i] = so->a.phs[i]
           + h * dso->a.phs[i];
    }
    } // End of euler()
    // Copies the source structure over to the target
443: void copy ( struct state *st,
               const struct state *ss ) {
444: int i;
445:     st->c = ss->c;
           // The mechanical part
446:     st->m.x = ss->m.x;
447:     st->m.v = ss->m.v;
448:     st->m.th = ss->m.th;
449:     st->m.w = ss->m.w;

```

```

450:     st->tc.field = ss->tc.field;
451:     st->a.field = ss->a.field;
452:     st->a.load = ss->a.load;
453:     for ( i = 0; i < PHASE; i++) {
        // The temperature part
454:         st->tc.prec[i] = ss->tc.prec[i];
455:         st->tc.nrec[i] = ss->tc.nrec[i];
456:         st->tc.lrec[i] = ss->tc.lrec[i];
457:         st->tc.phs[i] = ss->tc.phs[i];
        // The currents
458:         st->a.prec[i] = ss->a.prec[i];
459:         st->a.nrec[i] = ss->a.nrec[i];
460:         st->a.lrec[i] = ss->a.lrec[i];
461:         st->a.phs[i] = ss->a.phs[i];
    }
    }// End of copy()
    // Returns the derivative of a state
462: void deriv ( const struct state *s,
               struct state *ds ) {
463: double hload ( const double x );
464: double dhdx ( const double x );
465: double rload ( const double x );
466: double mfi ( const double , const int );
467: double dmfi ( const double , const int );
468: double dTdt ( const struct resistor *r,
               const double );
469: double rt ( const struct resistor *r,
             const double );
470: BOOL minv ( struct matrix * );
471: int i, j, k, m, n;
472: double sum;
473: struct rvec di;      // List of diode currents
474: struct rvec di_dot; // Derivative of diode currents
475: struct rvec amp_dot; // Vector of all the i-dots
476:     ds->c = s->c;     // Both have the same config.

        // The mechanical variables
477:     ds->m.x = s->m.v; // Projectile velocity
        // Projectile acceleration
478:     ds->m.v = s->a.load * s->a.load
            * dhdx ( s->m.x )/mass;
479:     ds->m.th = s->m.w; // Angular velocity
480:     sum = 0.0;
481:     for ( i = 0; i < PHASE; i++)
482:         sum += s->a.phs[i] * dmfi( s->m.th, i );
        // Angular acceleration
483:     ds->m.w = sum * s->a.field /moi;
        // The thermal variables and
        // zero out the current section
484:     ds->tc.field = s->a.field * s->a.field *
            dTdt (&rfield, s->tc.field );
485:     for ( i = 0; i < PHASE; i++ ) {
486:         ds->tc.prec[i] = s->a.prec[i]*s->a.prec[i]
            * dTdt (&rpscr, s->tc.prec[i] );

```

```

487:         ds->tc.nrec[i] = s->a.nrec[i]*s->a.nrec[i]
                * dTdt (&rnsr, s->tc.nrec[i] );
488:         ds->tc.lrec[i] = s->a.lrec[i]*s->a.lrec[i]
                * dTdt (&rlscr, s->tc.lrec[i] );
489:         ds->tc.phs[i] = s->a.phs[i]*s->a.phs[i]
                * dTdt (&rphs[i], s->tc.phs[i]);
    }
    // The currents
490:     di.n = 0;
491:     for (i = 0; i < np_d; i++) {
492:         j = index[ diode[i] ];
493:         if ( b_field[i] )
                di.v[di.n++] = s->a.prec[j];
494:         if ( b_load[i] )
                di.v[di.n++] = s->a.lrec[j];
    }
    // Skips the first negative diode
495:     for (i = np_d + 1; i < nt_d; i++) {
496:         j = index[ diode[i] ];
497:         if ( b_field[i] )
                di.v[di.n++] = s->a.nrec[j];
498:         if ( b_load[i] )
                di.v[di.n++] = s->a.lrec[j];
    }
    // Zero out the matrices
499:     for (i = 0; i < MAX; i++)
500:         for (j = 0; j < MAX; j++) {
501:             rmf.v[i][j] = 0.0;
502:             temp.v[i][j] = 0.0;
503:             h.v[i][j] = 0.0;
504:             r.v[i][j] = 0.0;
        }
505:     rmf.r = 0;
506:     rmf.c = 0;
507:     temp.r = 0;
508:     temp.c = 0;
509:     h.r = 0;
510:     h.c = 0;
511:     r.r = 0;
512:     r.c = 0;
    // Construct the matrix of the resistances
    // The phase coil resistances
513:     for (i = 0; i < nt_phs; i++) {
514:         j = index[i];
515:         rmf.v[i][i] = rt( rphs + j, s->tc.phs[j] );
516:         rmf.v[i][nt_phs] = s->m.w * dmfi(s->m.th,j);
517:         rmf.v[nt_phs][i] = rmf.v[i][nt_phs];
    }
    // The field coil resistance
518:     rmf.v[nt_phs][nt_phs] = rt(&rfield,s->tc.field);
519:     for (i = 0; i < np_d; i++) {
520:         j = index[ diode[i] ];
521:         k = nt_phs + i + 1;
        // The positive diode resistances

```

```

522:         if ( b_field[i] )
                    rmf.v[k][k] = rt(&rpscr,s->tc.prec[j]);
523:         if ( b_load[i] )
                    rmf.v[k][k] = rt(&rlscr,s->tc.lrec[j]);
    }
524:     for (i = np_d; i < nt_d; i++) {
525:         j = index[ diode[i] ];
526:         k = nt_phs + i + 1;
        // The negative diode resistances
527:         if ( b_field[i] )
                    rmf.v[k][k] = rt(&nscr,s->tc.nrec[j]);
528:         if ( b_load[i] )
                    rmf.v[k][k] = rt(&rlscr,s->tc.lrec[j]);
    }

529:     rmf.r = nt_phs + nt_d + 1;
530:     rmf.c = rmf.r;
531:     if ( load_flag ) {
532:         rmf.v[rmf.r][rmf.c] = rload (s->m.x)
                    + s->m.v*dhdx( s->m.x );
533:         rmf.r++;
534:         rmf.c++;
    }

    // Multiply loop * rmf * kir
535:     for (i = 0; i < kir.c; i++)
536:         for (j = 0; j < loop.r; j++) {
537:             for (m = 0; m < rmf.r; m++) {
538:                 sum = 0.0;
539:                 for (n = 0; n < kir.r; n++)
540:                     sum += rmf.v[m][n]*kir.v[n][j];
541:                 r.v[i][j] += loop.v[i][m] * sum;
            }
        }

542:     r.r = loop.r;
543:     r.c = kir.c;
    // Construct the matrix of the inductances
544:     for (i = 0; i < nt_phs; i++) {
545:         temp.v[i][nt_phs] = mfi(s->m.th,index[i]);
546:         temp.v[nt_phs][i] = temp.v[i][nt_phs];
547:         for (j = 0; j < nt_phs; j++)
548:             temp.v[i][j] = mut[index[i]][index[j]];
    }

549:     temp.v[nt_phs][nt_phs] = hfield;
550:     for (i = 0; i < np_d; i++) {
551:         j = nt_phs + i + 1;
552:         if( b_field[i] ) temp.v[j][j] = hpscr;
553:         if (b_load[i] ) temp.v[j][j] = hlscr;
    }

554:     for (i = np_d; i < nt_d; i++) {
555:         j = nt_phs + i + 1;
556:         if( b_field[i] ) temp.v[j][j] = hnscr;
557:         if (b_load[i] ) temp.v[j][j] = hlscr;
    }
558:
559:     temp.r = nt_phs + nt_d + 1;
560:     temp.c = nt_phs + nt_d + 1;

```

```

561:   if ( load_flag ) {
562:       temp.v[temp.r][temp.c] = hload ( s->m.x );
563:       temp.r++;
564:       temp.c++;
    }
    // Multiply loop * temp * kir
565:   for ( i = 0; i < kir.c; i++)
566:       for ( j = 0; j < loop.r; j++) {
567:           for ( m = 0; m < temp.r; m++) {
568:               sum = 0.0;
569:               for ( n = 0; n < kir.r; n++)
570:                   sum += temp.v[m][n]*kir.v[n][j];
571:               h.v[i][j] += loop.v[i][m] * sum;
            }
        }
572:   h.r = loop.r;
573:   h.c = kir.c;
574:   minv ( &h );
575:   for ( di_dot.n = 0; di_dot.n < h.r; di_dot.n++) {
576:       di_dot.v[di_dot.n] = 0.0;
577:       for ( m = 0; m < h.c; m++) {
578:           sum = 0.0;
579:           for ( j = 0; j < r.c; j++)
580:               sum += r.v[m][j]*di.v[j];
           di_dot.v[di_dot.n] -=
               h.v[di_dot.n][m]*sum;
        }
    }
581:   ds->a.load = 0.0;
582:   ds->a.field = 0.0;
583:   for ( i = 0; i < PHASE; i++) {
584:       ds->a.prec[i] = 0.0;
585:       ds->a.nrec[i] = 0.0;
586:       ds->a.lrec[i] = 0.0;
587:       ds->a.phs[i] = 0.0;
    }
    // Put all the derivatives and currents
    // into the structure
    // Multiply the vector di_dot by the kir matrix
588:   for ( amp_dot.n = 0; amp_dot.n < kir.r;
        amp_dot.n++) {
589:       amp_dot.v[amp_dot.n] = 0.0;
590:       for ( j = 0; j < kir.c; j++ )
591:           amp_dot.v[amp_dot.n] +=
               kir.v[amp_dot.n][j]*di_dot.v[j];
    }
    // amp_dot.v is now a list of all the i-dots
592:   for ( i = 0; i < nt_phs; i++ ) {
593:       j = index[i];
594:       ds->a.phs[j] = amp_dot.v[i];
    }
595:   ds->a.field = amp_dot.v[nt_phs];
596:   for ( i = 0; i < np_d; i++ ) {
597:       j = index[ diode[i] ];

```

```

598:         if ( b_field[i] )
           ds->a.prec[j] = amp_dot.v[nt_phs + i + 1];
599:         if ( b_load[i] )
           ds->a.lrec[j] = amp_dot.v[nt_phs + i + 1];
        }
600:     for (i = np_d; i < nt_d; i++ ) {
601:         j = index[ diode[i] ];
602:         if ( b_field[i] )
           ds->a.nrec[j] = amp_dot.v[nt_phs + i + 1];
603:         if ( b_load[i] )
           ds->a.lrec[j] = amp_dot.v[nt_phs + i + 1];
        }
604:     if ( load_flag )
        ds->a.load = amp_dot.v[nt_phs + nt_d + 1];
    } // End of deriv()
605: void volts (struct voltage *volt,
              const struct state *s,
              const struct state *ds){
606: double mfi ( const double th, const int i );
607: double dmfi ( const double th, const int i );
608: double rt (const struct resistor *, const double t);
609: int i, j;
610: double d;
611:     d = s->a.field * s->m.w;
612:     for (i = 0; i < PHASE; i++) {
613:         volt->vphs[i] = -d*dmfi(s->m.th, i)
           - rt(rpshs + i, s->tc.phs[i])*s->a.phs[i]
           - ds->a.field * mfi(s->m.th, i);
614:         for (j = 0; j < PHASE; j++)
615:             volt->vphs[i] -= mut[i][j]*ds->a.phs[j];
        }
616:     volt->vload = s->a.load*rload(s->m.x)
        + ds->a.load*hload(s->m.x)
        + s->m.v*dhdx(s->m.x)*s->a.load;
    // Search for a phase coil
    // conducting to the pos. busbar
617:     j = 0;
618:     while ( !s->c->prec[j] && j < PHASE ) j++;
619:     volt->vplus = volt->vphs[j]
        - rt(&rpscr, s->tc.prec[j]) * s->a.prec[j]
        - hpscr * ds->a.prec[j];
    // Search for a phase coil
    // conducting to the neg. busbar
620:     j = 0;
621:     while ( !s->c->nrec[j] && j < PHASE ) j++;
622:     volt->vminus = volt->vphs[j]
        + rt(&rnscr, s->tc.nrec[j]) * s->a.nrec[j]
        + hnscr * ds->a.nrec[j];
    } // End of volts ()
    // Routine checks the new results for a change in
    // the configuration. Returns TRUE if there is a
    // change and a dt for an approximate time when
    // the change occurs.
623: BOOL check ( const struct state *so,

```

```

        const struct state *dso,
        struct state *sn,
        const struct state *dsn,
        struct config *cn, double *dt ) {
624: void volts ( struct voltage *v,
        const struct state *s,
        const struct state *ds );

625: BOOL test;
626: struct voltage vo, vn;
627: double tmin, t, dv;
628: int i;
629:     volts ( &vo, so, dso );
630:     volts ( &vn, sn, dsn );
631:     tmin = 1.0;
632:     test = FALSE;    // Assume no change in config.
633:     cn->load = so->c->load;
634:     for ( i = 0; i < PHASE; i++ ) {
635:         cn->prec[i] = so->c->prec[i];
636:         cn->nrec[i] = so->c->nrec[i];
637:         cn->lrec[i] = so->c->lrec[i];
638:         if ( so->c->prec[i] &&
            ( sn->a.prec[i] < 0.0 ) ) {
            // Diode has a reverse current
639:             test = TRUE;
640:             t = so->a.prec[i] /
                (so->a.prec[i] - sn->a.prec[i]);
            // Diode not conducting
641:             cn->prec[i] = FALSE;
642:             sn->a.prec[i] = 0.0;
643:             if ( t < tmin ) tmin = t;
        }
644:         if ( !so->c->prec[i] &&
            ( vn.vphs[i] > vn.vplus ) ) {
            // Nonconducting diode is forward biased
645:             test = TRUE;
646:             dv = vo.vplus - vo.vphs[i];
647:             t = dv / ( vn.vphs[i] - vn.vplus + dv );
648:             cn->prec[i] = TRUE; // Conducting diode
649:             sn->a.prec[i] = 0.0;
650:             if ( t < tmin ) tmin = t;
        }
651:         if ( so->c->nrec[i] &&
            ( sn->a.nrec[i] < 0.0 ) ) {
            // A diode has a reverse current
652:             test = TRUE;
653:             t = so->a.nrec[i] /
                (so->a.nrec[i] - sn->a.nrec[i]);
654:             cn->nrec[i] = FALSE;
655:             sn->a.nrec[i] = 0.0;
656:             if ( t < tmin ) tmin = t;
        }
657:         if ( !so->c->nrec[i] &&
            ( vn.vphs[i] < vn.vminus ) ) {
            // Nonconducting diode is forward biased

```

```

658:         test = TRUE;
659:         dv = vo.vminus - vo.vphs[i];
660:         t = dv/( vn.vphs[i] - vn.vminus + dv );
           // Diode becomes conducting
661:         cn->nrec[i] = TRUE;
662:         sn->a.nrec[i] = 0.0;
663:         if ( t < tmin ) tmin = t;
           }
664:     if ( so->c->lrec[i]
           && (sn->a.lrec[i] < 0.0) ) {
           // Load diode has a reverse current
665:         test = TRUE;
666:         t = so->a.lrec[i]
           / (so->a.lrec[i] - sn->a.lrec[i]);
           // Diode is no longer conducting
667:         cn->lrec[i] = FALSE;
668:         sn->a.lrec[i] = 0.0;
669:         if ( t < tmin ) tmin = t;
           }
670:     if ( so->c->load &&
           !so->c->lrec[i]&&(vn.vphs[i]>vn.vload)){
           // A nonconducting diode has a forward
           // voltage bias. Section is skipped
           // when the load is not connected.
671:         test = TRUE;
672:         dv = vo.vload - vo.vphs[i];
673:         t = dv/( vn.vphs[i] - vn.vload + dv );
674:         cn->lrec[i] = TRUE;
675:         sn->a.lrec[i] = 0.0;
676:         if ( t < tmin ) tmin = t;
           }
           }
677:     *dt = tmin;
678:     return test;
           } // End of check ()
679: double mfi ( const double th, const int i ) {
680:     return mf[i]*sin(th + os[i]);
           } // End of mfi()
681: double dmfi ( const double th, const int i ) {
682:     return mf[i]*cos(th + os[i]);
           } // End of dmfi ()
           // The increase rate of the resistor's temperature
683: double dTdt ( const struct resistor *rs,
           const double temp ) {
684:     return ( rs->ro*( 1.0 + rs->c*(temp-TREF) )
           / rs->mass / rs->cp );
           } // End of dTdt()
           // The resistance at the temperature
685: double rt ( const struct resistor *rs,
           const double temp ) {
686:     return ( rs->ro*(1.0 + rs->c*(temp-TREF) ) );
           } // End of rt()
           // Inverts the matrix
687: BOOL minv ( struct matrix *a ) {

```

```

688: int ik[MAX],jk[MAX];
689: double amax,save;
690: double fabs();
691: int i,j,k;
692:     if ( a->r != a->c ) return FALSE;
693:     for (k = 0; k < a->c; k++) {
        /*Find the largest element of the matrix.*/
694:     amax = 0.0;
695:     for (i = k; i < a->c; i++)
696:         for (j = k; j < a->c; j++)
697:             if (fabs(amax) < fabs(a->v[i][j])) {
698:                 amax = a->v[i][j];
699:                 ik[k] = i;
700:                 jk[k] = j;
        }
        /*Switch rows and columns to put amax on diagonal.*/
701:     i = ik[k];
702:     if (i != k)
703:         for (j = 0; j < a->c; j++) {
704:             save = a->v[k][j];
705:             a->v[k][j] = a->v[i][j];
706:             a->v[i][j] = -save;
        }
707:     j = jk[k];
708:     if (j != k)
709:         for (i = 0; i < a->c; i++) {
710:             save = a->v[i][k];
711:             a->v[i][k] = a->v[i][j];
712:             a->v[i][j] = -save;
        }
        /*Accumulate elements of the inverse matrix.*/
713:     for (i = 0; i < a->c; i++)
714:         if (i!=k) a->v[i][k] = -a->v[i][k]/amax;
715:     for (i = 0; i < a->c; i++)
716:         for (j = 0; j < a->c; j++)
717:             if ((i!=k) && (j!=k))
718:                 a->v[i][j] =
                    a->v[i][j]+a->v[i][k]
                    *a->v[k][j];
719:     for (j = 0; j < a->c; j++)
720:         if (j!=k) a->v[k][j] = a->v[k][j]/amax;
721:     a->v[k][k] = 1.0/amax;
    }
    /*Restore ordering of matrix.*/
722:     for (k = a->c - 1; k > -1; k--) {
723:         j = ik[k];
724:         if (j>k) for (i = 0; i < a->c; i++) {
725:             save = a->v[i][k];
726:             a->v[i][k] = -a->v[i][j];
727:             a->v[i][j] = save;
        }
728:     i = jk[k];
729:     if (i>k) for (j = 0; j < a->c; j++) {

```

```
730:         save = a->v[k][j];
731:         a->v[k][j] = -a->v[i][j];
732:         a->v[i][j] = save;
        }
    }
733:     return TRUE;
} // End of minv()
```

<u>NO. OF COPIES</u>	<u>ORGANIZATION</u>	<u>NO. OF COPIES</u>	<u>ORGANIZATION</u>
2	ADMINISTRATOR DEFENSE TECHNICAL INFO CENTER ATTN DTIC OCP 8725 JOHN J KINGMAN RD STE 0944 FT BELVOIR VA 22060-6218	1	SAIC ATTN KEITH A JAMISON 1247-B N EGLIN PKWY SHALIMAR FL 32579
1	DIRECTOR US ARMY RESEARCH LABORATORY ATTN AMSRL CS AL TA REC MGMT 2800 POWDER MILL RD ADELPHI MD 20783-1197	1	SAIC ATTN W RIENSTRA 8200 N MOPAC EXPRESSWAY STE 150 AUSTIN TX 78759
1	DIRECTOR US ARMY RESEARCH LABORATORY ATTN AMSRL CI LL TECH LIB 2800 POWDER MILL RD ADELPHI MD 207830-1197	1	SAIC ATTN J GULLY 1410 SPRING HILL ROAD STE 400 MCLEAN VA 22102
1	DIRECTOR US ARMY RESEARCH LABORATORY ATTN AMSRL DD J J ROCCHIO 2800 POWDER MILL RD ADELPHI MD 20783-1197	1	SAIC ATTN G CHRYSOMALLIS 3800 WEST 80TH STREET STE 1090 BLOOMINGTON MN 55431
2	DIRECTOR US ARMY RESEARCH LABORATORY ATTN AMSRL SE EP T B BAHDER J D BRUNO BLDG 203 2800 POWDER MILL RD ADELPHI MD 20783-1197	5	DARPA ATTN DR M FREEMAN DR D RADACK DR D FIELDS B KASPAR L STOTTS 3701 N FAIRFAX DR ARLINGTON VA 22203-1714
1	DIRECTOR MICOM RDEC ATTN AMSMI RD W C MCCORKLE REDSTONE ARSENAL AL 35898-5240	2	THE UNIV OF TEXAS AT AUSTIN J J PICKLE RSRCH CAMPUS ATTN MAIL CODE R7000 A WALLS MAIL CODE R7000 S PRATAP CENTER FOR ELECTROMECHANICS AUSTIN TX 78712
3	THE UNIV OF TEXAS AT AUSTIN INST FOR ADVANCED TECHNOLOGY ATTN DR H FAIR DR I MCNAB DR D ECCLESHALL 4030-2 W BRAKER LANE AUSTIN TX 78759-5329	1	PM-TMAS ATTN AMSTA AR FSP E D LADD BLDG B354 PICATINNY ARSENAL NJ 07806-5000
1	U BUFFALO DEPT EE ATTN J SARJEANT BOX 601900 BONNER HALL ROOM 312 BUFFALO NY 14260-1900		<u>ABERDEEN PROVING GROUND</u>
		2	DIRECTOR US ARMY RESEARCH LABORATORY ATTN AMSRL CI LP (TECH LIB) BLDG 305 APG AA
		1	DIR USARL ATTN AMSRL WM B E M SCHMIDT BLDG 390A
		2	DIR USARL ATTN AMSRL WM BC P WEINACHT A ZIELINSKI BLDG 390

NO. OF
COPIES ORGANIZATION

- 1 DIR USARL
 ATTN AMSRL WM BE G KATULKA
 BLDG 390A
- 1 DIR USARL
 AMSRL WM M A B TANNER
 BLDG 4600
- 3 DIR USARL
 AMSRL WM MB R BOSSOLI
 S CORNELISON F PIERCE
 BLDG 120
- 19 DIR USARL
 ATTN AMSRL WM TE P BERNING
 J CORRERI C HUMMER (5 CYS)
 D DANIEL L KECSKES
 T KOTTKE K MAHAN
 M MCNEIR A NIILER
 J POWELL A PRAKASH
 S ROGERS H SINGH
 C STUMPFEL G THOMSON
 BLDG 120

ABSTRACT ONLY

- 1 DIRECTOR
 US ARMY RESEARCH LABORATORY
 ATTN AMSRL CS AL TP TECH PUB BR
 2800 POWDER MILL RD
 ADELPHI MD 20783-1197

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE June 1999	3. REPORT TYPE AND DATES COVERED Final	
4. TITLE AND SUBTITLE A Lumped Circuit Model of a Compensated Pulse Generator and Rail Gun			5. FUNDING NUMBERS PR: 61104BH62	
6. AUTHOR(S) Hummer, C.R. (ARL)				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) U.S. Army Research Laboratory Weapons Technology Directorate Aberdeen Proving Ground, MD 21010-5066			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) U.S. Army Research Laboratory Weapons Technology Directorate Aberdeen Proving Ground, MD 21010-5066			10. SPONSORING/MONITORING AGENCY REPORT NUMBER ARL-TR-1990	
11. SUPPLEMENTARY NOTES				
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) Because of its compact size, an electrical generator that uses an internal rotating mass as an energy source for electrical power is being developed to power a rail gun in a future combat system. At this stage of development, there are many proposed designs for these electric generators and many proposed designs for their possible uses: rail guns, coil guns, electromagnetic armor, etc. To study these various designs, a computer program was written to calculate the current in all parts of the electric generator, the load, the angular velocity of the rotating mass, and the velocity of the projectile from a rail gun or a coil gun. This was accomplished by modeling the electric generator and the rail gun by a circuit of inductors and resistors. This model results in a set of differential equations that are coupled with the equation of motion for the rotating mass and with the equation of motion for the projectile.				
14. SUBJECT TERMS pulse generator rail gun			15. NUMBER OF PAGES 70	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT	