WEED
A WONDERFUL EQUATION ELIMINATION DEVICE

by

CELSO JOHN FRAZAO GUIMARAES

June 1971

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS
DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.
DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.
WEED
A WONDERFUL EQUATION ELIMINATION DEVICE*

by

CELSO JOHN FRAZAO GUIMARAES

June 1971

Department of Computer Science
University of Illinois
Urbana, Illinois 61801

* This report was supported in part by the Department of Computer Science and in part by the Atomic Energy Commission under grant US AEC AT(ll-1)1469, and submitted to the graduate college in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.
ACKNOWLEDGMENT

I am indebted to the following people whose assistance was most invaluable in the preparation of this thesis. To Professor C. W. Gear for suggesting the topic and for his enlightening guidance in times of panic. To Miss Barbara Hurdle for her unmatched patience and talent in typing the manuscript. To the Graphics Group (Al·Whaley, Marty Michel, Roger Haskin, John Nickolls, and Sally Wilkins) for the fruitful "rap" sessions. And to the Department of Computer Science in general for so kindly providing all the necessary facilities.
PREFACE

This thesis is a description of WEED, an algorithm for simplifying systems of equations symbolically and classifying variables for further numerical analysis. It is an integral part of the Graphics-Oriented Simulation and Modeling System currently being developed at the University of Illinois.

Through a consistent method of elimination, WEED produces a condensed system of equations and an Equivalence Table for variables, which allow for a more efficient compilation and numerical integration than would otherwise be possible.
**TABLE OF CONTENTS**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. THE MODELING SYSTEM.</td>
<td>1</td>
</tr>
<tr>
<td>2. PRELIMINARIES AND OVERVIEW OF THE ALGORITHM.</td>
<td>3</td>
</tr>
<tr>
<td>3. INPUT SPECIFICATIONS</td>
<td>6</td>
</tr>
<tr>
<td>4. OUTPUT REQUIREMENTS</td>
<td>10</td>
</tr>
<tr>
<td>5. EQUATION CLASSIFICATION</td>
<td>14</td>
</tr>
<tr>
<td>5.1 Variable = Constant</td>
<td>16</td>
</tr>
<tr>
<td>5.2 Variable_1 = Variable_2</td>
<td>16</td>
</tr>
<tr>
<td>5.3 Variable = Expression</td>
<td>17</td>
</tr>
<tr>
<td>5.4 Expression = Expression</td>
<td>18</td>
</tr>
<tr>
<td>6. THE PROCESS ROUTINE</td>
<td>19</td>
</tr>
<tr>
<td>6.1 Constant</td>
<td>19</td>
</tr>
<tr>
<td>6.2 Global Variable or TIME</td>
<td>19</td>
</tr>
<tr>
<td>6.3 Other Variable</td>
<td>19</td>
</tr>
<tr>
<td>6.4 Binary Operation</td>
<td>20</td>
</tr>
<tr>
<td>6.5 Unary Operation</td>
<td>22</td>
</tr>
<tr>
<td>6.6 User-Defined Function</td>
<td>23</td>
</tr>
<tr>
<td>7. THE FINAL STAGES: NAME REASSIGNMENTS, EQUATION RESTRUCTURING AND OUTPUT</td>
<td>24</td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td>26</td>
</tr>
<tr>
<td>APPENDICES</td>
<td>27</td>
</tr>
</tbody>
</table>
1. THE MODELING SYSTEM

The General Simulation and Modeling System currently being developed at the University of Illinois has as its primary objective "to provide a tool for the user with which he can (graphically) construct models of a proposed system and analyze them for static, dynamic and oscillatory behavior. The user will have the capability to define elements which are models of individual items of his system (e.g. diodes, motors, water pumps) and specify their behavior by means of differential and algebraic equations. The elements will have terminals (e.g. wires, mechanical links, pipes) through which certain variables may be transmitted (e.g. voltage, current, torque, displacement, pressure, flow). The terminals from several elements can be connected together at nodes (connection points) to form a network." A network may be used as an element of a larger network, and hence in a recursive fashion fairly complex networks can be constructed quite easily. The overall network is then the model of the proposed system, and is equivalent to the system of differential and algebraic equations derived from each element and node of the model. Thus the model can be completely analyzed by solving this equivalent system of equations.

The user interacts with the Modeling System through a drawing program which builds data structures representing each element and network. From these, Item and Global Analyses create new data structures which no longer contain graphical information. The equations are parsed into tree structures, assigned values are substituted in the equations, and all external names are replaced with new internal pointers. WEED then simplifies the equations, classifies them according to the types of operations and variables they contain, and weeds out those which are no
longer needed. Variables are organized in an Equivalence Table according to their original definition, and placed into different sets depending on how they are used in the equations. New internal names are then assigned to all the variables, and the equations are updated and restructured into a compact format. The final stage then compiles the condensed system of equations, and solves for transient, steady-state or oscillatory response.
2. PRELIMINARIES AND OVERVIEW OF THE ALGORITHM

Variables contained in the system of equations are not all of the same nature. Some refer to terminals in the network, and are identified as I-variables (e.g. electrical current). Others are associated with nodes and are classed as E-variables (e.g. voltage). Local variables (also called internal variables) are those which are specific to an instance of element in the network. Global variables, on the other hand, are common to groups of elements (e.g. ambient temperature). The special global variable TIME is permanently assigned. Finally, parameters are those variables associated with a given type of element, whose value is specified with each instance of that element in the network (e.g. resistance of a resistor).

In addition to keeping track of the above categories, WEED places all variables into four mutually exclusive sets, according to their usage in the equations:

a. Set $S_1$—all global variables and variables defined by $S_1$-equations (of the form $V = \text{function (globals and constants only)}$).

b. Set $S_2$—variables defined by $S_2$-equations (of the form $V = \text{function (globals, constants, and time)}$), and the special variable TIME.

c. Set $L$—variables involved only in linear expressions.

d. Set $M$—variables involved in non-linear or differential expressions. As an example, given the equations
\[ E_0 = 3E_1 + G_0(E_2 + E_3) - 1/E_4 + (E_5 + 2) \cdot T - E_6E_7 + E_8 \]
\[ I_0 = 3 - G_0/2 \]
\[ I_1 = I_0 \cdot \sin(\cdot T) \]
\[ I_2 = (I_1 + 3)/(I_1 - I_0) \]

where the \( E \) are \( E \)-variables, the \( I \) are \( I \)-variables, the \( G \) are globals and \( T \) is time, \( E_0 \) through \( E_3 \) would belong in set \( L \); \( E_4 \) through \( E_8 \) in set \( M \); \( I_0 \) in set \( S_1 \); and \( I_1 \) and \( I_2 \) in set \( S_2 \). Note that since global values are fixed throughout any given execution of the Modeling System, global variables are treated as constants insofar as variable and expression classification is concerned.

An additional set \( D_Y \) of new variables is created for the output of general equations, and is discussed in section 4.

A number of passes are made through the system of equations, each time advantage being taken of simplifications made in previous passes. As each equation is processed, all expressions which are only functions of constants are evaluated immediately and replaced by the result. Expressions of the form

\[ E + 0, O * E, E * 0, O / E, E / 1, l * E, E * 0, E * 1, \]
\[ l * * E, \text{ and } \frac{d}{dt} (S_1 \text{ or } S_2 \text{-expression}) \]

are simplified as they are detected, and unary minuses are eliminated wherever possible. Equations which become trivial relations, such as

\[ \text{variable} = \text{constant} \]

or

\[ \text{variable} = \pm \text{variable} \]

are eliminated by setting appropriate pointers in the Equivalence Table (described in section 4), and the remaining equations are placed...
into three sets: $S_1$-equations, $S_2$-equations, and general equations (those which are neither $S_1$ nor $S_2$).

The reason for arranging all the equations and variables into different sets is that each set can receive special treatment in the numerical analysis phase, thereby increasing performance and reducing cost. For instance, variables in set $S_1$ need only be evaluated once per simulation; those in set $S_2$ once per time step; and derivatives of variables in set $L$ need not be evaluated at all.

The equations are reprocessed until there are no more changes in the Equivalence Table (up to a maximum of three passes). All variables are then renumbered so that the members of each set are contiguous in memory—that is, the numbers 0 to $n_{S_1}$ refer to the members of set $S_1$; $n_{S_1}$ to $n_{S_1} + n_{S_2}$ to the members of set $S_2$; $n_{S_1} + n_{S_2}$ to $n_{S_1} + n_{S_2} + n_{L}$ to the members of set $L$; and $n_{S_1} + n_{S_2} + n_{L}$ to the members of set $M$. The new names are substituted in the equations, and these in turn are compressed and output in their three different sets to the numerical integration routines. A concise flow chart of the WEED algorithm is given in Appendix A.
3. INPUT SPECIFICATIONS

The input to WEED consists of a set of Input Tables and a single list of equations, as shown in Figure 1.

Figure 1. Input to WEED
The Name Tables contain the actual eight-character variable names provided by the user. In the equations, however, these variables are represented by the displacement of their name from the beginning of their respective Name Tables. The same holds true for constants, where the Constant Table contains the double-precision floating-point constant values.

The equations are in tree-structured form, and consist of a series of "nodes" of information connected by pointers, as illustrated in Figure 2. The first node of each equation consists of a pointer to the next equation in the list, as a displacement from the current equation. The remaining nodes are clusters of halfwords (usually two or three) contiguous in memory. The first HW of each node is an op-code which indicates the type of node. If the node represents an operation, the remaining HW-fields contain pointers to its respective operands. If the node is a variable or constant, the second HW contains the internal name associated with it. Table 1 describes the different op-codes and associated operands.

---

1 E- and I-type variables do not have alphabetic names associated with them, and are numbered sequentially, starting from 0.

2 The 'user-specified function' node is a variable-length node, including fields for function type (second HW) and number of pointers (third HW).

3 A 'function' node contains a field specifying the type of function (second HW).
\[ E_0 = -\sin(P_\theta \cdot t) \]
\[ I_3 = 1 + G_1 \]

<table>
<thead>
<tr>
<th>address</th>
<th>in memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000 0028</td>
</tr>
<tr>
<td>4</td>
<td>0007 00A 000E</td>
</tr>
<tr>
<td>A</td>
<td>0004 0000</td>
</tr>
<tr>
<td>E</td>
<td>0011 0012</td>
</tr>
<tr>
<td>12</td>
<td>0012 0006 0018</td>
</tr>
<tr>
<td>18</td>
<td>000A 001E 0022</td>
</tr>
<tr>
<td>1E</td>
<td>0002 0008</td>
</tr>
<tr>
<td>22</td>
<td>0000 0000</td>
</tr>
<tr>
<td>28</td>
<td>0000 0024</td>
</tr>
<tr>
<td>2C</td>
<td>0007 00A 000E</td>
</tr>
<tr>
<td>32</td>
<td>0005 0003</td>
</tr>
<tr>
<td>36</td>
<td>0008 0014 0018</td>
</tr>
<tr>
<td>3C</td>
<td>0006 0008</td>
</tr>
<tr>
<td>40</td>
<td>0001 0008</td>
</tr>
</tbody>
</table>

Figure 2. Equation Format
<table>
<thead>
<tr>
<th>Node Type</th>
<th>Op</th>
<th>Operands</th>
<th>Halfwords</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>X'0'</td>
<td>X'0'</td>
<td>2</td>
</tr>
<tr>
<td>Global Variable</td>
<td>1</td>
<td>Displ. in Global Name Table</td>
<td>2</td>
</tr>
<tr>
<td>Parameter</td>
<td>2</td>
<td>Displ. in Parm Name Table</td>
<td>2</td>
</tr>
<tr>
<td>Internal Variable</td>
<td>3</td>
<td>Displ. in Int. Var. Name Table</td>
<td>2</td>
</tr>
<tr>
<td>E-Variable</td>
<td>4</td>
<td>Numeric Name</td>
<td>2</td>
</tr>
<tr>
<td>I-Variable</td>
<td>5</td>
<td>Numeric Name</td>
<td>2</td>
</tr>
<tr>
<td>Constant</td>
<td>6</td>
<td>Displ. in Constant Table</td>
<td>2</td>
</tr>
<tr>
<td>Equals (=)</td>
<td>7</td>
<td>Left-pointer; Right-pointer</td>
<td>3</td>
</tr>
<tr>
<td>Addition (+)</td>
<td>8</td>
<td>L-ptr; R-ptr</td>
<td>3</td>
</tr>
<tr>
<td>Subtraction (-)</td>
<td>9</td>
<td>L-ptr; R-ptr</td>
<td>3</td>
</tr>
<tr>
<td>Multiplication (*)</td>
<td>A</td>
<td>L-ptr; R-ptr</td>
<td>3</td>
</tr>
<tr>
<td>Division (/)</td>
<td>B</td>
<td>L-ptr; R-ptr</td>
<td>3</td>
</tr>
<tr>
<td>Assignment (+)</td>
<td>C</td>
<td>L-ptr; R-ptr</td>
<td>3</td>
</tr>
<tr>
<td>Power (**))</td>
<td>D</td>
<td>L-ptr; R-ptr</td>
<td>3</td>
</tr>
<tr>
<td>Max</td>
<td>E</td>
<td>L-ptr; R-ptr</td>
<td>3</td>
</tr>
<tr>
<td>Min</td>
<td>F</td>
<td>L-ptr; R-ptr</td>
<td>3</td>
</tr>
<tr>
<td>Differentiation</td>
<td>10</td>
<td>Operand-ptr</td>
<td>2</td>
</tr>
<tr>
<td>Unary</td>
<td>11</td>
<td>Operand-ptr</td>
<td>2</td>
</tr>
<tr>
<td>Function</td>
<td>12</td>
<td>Fn ID#; Operand-ptr</td>
<td>3</td>
</tr>
<tr>
<td>User-spec fn</td>
<td>13</td>
<td>Fn ID#; #-operands; operand-ptrs</td>
<td>≥3</td>
</tr>
</tbody>
</table>

Table 1. Internal Codes for Equation Nodes
4. OUTPUT REQUIREMENTS

The output consists of a Table Block and an Equation Block, as depicted in Figure 3. The Table Block consists of the Constant Table (including possible additions created in the WEED step), the input Name Tables (for use in cross-referencing and diagnostics), and the Equivalence Table (EQV), which contains all the variables ordered according to network type, with their respective new internal names (variables found to be equivalent will, of course, have the same assigned name). The Table Block also contains other necessary data, such as pointers to the various tables and to the Equation Block.

Upon entry to WEED, the variables are arranged in the EQV in order of ascending numeric names for each field in the table, as shown in Figure 4. As the equations are processed, different types are assigned to the variables, and appropriate pointers are set for those which are found to be equivalent to other variables or to constants. During internal name reassignment, all variables are marked 'assigned' (except for those equivalent to constants), so that upon output, only three types will be present in the EQV: assigned (+V), assigned (-V), and constants.
Figure 3. Output From _____
Figure 4. Equivalence Table

The Equation Block consists of three sub-blocks corresponding to the three different sets of equations. The equations take a form similar to the input equations, with two differences:

1. Operation nodes incorporate operands which are variables, their pointer fields being replaced by the corresponding new variable names. The remaining pointers to other nodes take on a negative value, in order to differentiate from variable names.
2. General equations are put in the form \( 0 = \text{expression} \), and the 0 is replaced by a new variable of the set \( \text{DY} \), where the \( \text{DY} \)'s are numbered starting at \( n_{S_1} + n_{S_2} + n_L + n_M \).

Figure 5 illustrates the modifications performed on the example of Figure 2.

\[
\begin{align*}
\text{DY}_0 &= E_0 + \sin(P_1 \cdot T) \\
\text{I}_3 &= 1 + G_1
\end{align*}
\]

<table>
<thead>
<tr>
<th>Address</th>
<th>In Memory</th>
<th>var</th>
<th>new name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000 001C</td>
<td>DY_0</td>
<td>123</td>
</tr>
<tr>
<td>4</td>
<td>0007 0123 FFF6</td>
<td>E_0</td>
<td>69</td>
</tr>
<tr>
<td>A</td>
<td>0008 0069 FFF0</td>
<td>P_1</td>
<td>33</td>
</tr>
<tr>
<td>10</td>
<td>0012 0006 FFEA</td>
<td>T</td>
<td>17</td>
</tr>
<tr>
<td>16</td>
<td>000A 0033 0017</td>
<td>I_3</td>
<td>269</td>
</tr>
<tr>
<td></td>
<td></td>
<td>G_1</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 5. Output Equations
5. EQUATION CLASSIFICATION

For this and the following sections, the flow charts provided in the Appendices should be useful as an aid to understanding the WEED Algorithm.

Upon entry, WEED goes through an initialization phase, in which it GETMAIN's an area for the output Table Block, copies the Name Tables and Constant Table from the input area, sets pointers to the different tables, and zeroes out the Equivalence Table. It is then ready for the Equation Classification stage.

WEED takes each equation in turn, replaces the '=' in the first node by a '-', and calls upon the PROCESS Routine (described in section 6) to determine the types of expressions on the left and righthand sides of the equation. PROCESS travels recursively down the tree, performing simplifications as it goes along, and returns two pairs of numbers representing the left and right expression types. Every expression has two numbers associated with it: one (referred to in the program as EXPTP, LX or RX) which gives an indication of the structure and complexity of the expression; and another (referred to as S1S2EXP, S1S2LX, or S1S2RX) which indicates whether the expression is type $S_1$, $S_2$, or neither. Table 2 lists the possible types and respective codes.
Table 2. Expression Types

Once WEED receives the left and right expression types from PROCESS, it restores the '=' and tries to put the equation in a standard format, according to the following priorities:

1. \( S_1 \) or \( S_2 \)-expressions on the right side. (If the left side is also an \( S_1 \) or \( S_2 \)-expression an error is indicated, the equation is eliminated and the next equation is fetched.)

2. Single variable on the left side.

3. Doublet (\( C+V \) or \( +V \pm V \)) on the left side.

After the appropriate switches are made, if a doublet (or any expression of the form \( E+\overline{E} \)) is found on the left, it is split up, the left branch of the doublet being transposed to the right side of the equation.\(^1\) At this stage, the equation will necessarily be in one of the following forms:

\(^1\) Any constants in a doublet are always kept on the left. Hence in a split-up, the constant always gets moved to the right side of the equation.
variable = constant
variable = ± variable
variable = expression
expression = expression

We can now describe the individual steps taken in each case.

5.1 Variable = Constant

Set variable type to 'constant' (X'Cl') in the EQV. Place a pointer to the constant (as a displacement from the beginning of the Constant Table) in the EQV value field of the variable. Eliminate the equation, and set EQVFLAG to 1, indicating a change made to the EQV.

5.2 Variable₁ = Variable₂

The two variable types determine the action to be taken, according to the following table:

<table>
<thead>
<tr>
<th>Variable₁</th>
<th>S₁</th>
<th>S₂</th>
<th>L</th>
<th>M</th>
<th>C</th>
<th>+V</th>
<th>-V</th>
</tr>
</thead>
<tbody>
<tr>
<td>S₁</td>
<td>E</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S₂</td>
<td>E</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>E</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+V</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-V</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
E:  Error.  Relay message to the user and delete the equation.

\( \Rightarrow \)  Switch the variables, so that \( \text{variable}_1 \) becomes \( \text{variable}_2 \) and vice versa. Refer to the table.

\( \Rightarrow \)  Follow the pointer (or chain of pointers)\(^2\) beginning at \( \text{variable}_1 \). The last variable in the chain becomes the new \( \text{variable}_1 \). Return to the table.

OK:  Set the type of \( \text{variable}_1 \) to +V or -V according to the equation which defines it. If \( \text{variable}_2 \) points to another variable, follow the pointer chain.\(^2\) Place a pointer to the new \( \text{variable}_2 \) in the value field of \( \text{variable}_1 \).\(^3\) Eliminate the equation and set EQVFLAG to 1.

5.3  Variable = Expression

If the expression is neither a \( \text{S}_1 \) nor \( \text{S}_2 \)-expression, go to section 5.4 (Expression = Expression). Set the variable type to \( \text{S}_1 \) or \( \text{S}_2 \) accordingly. If the equation has already been placed in the proper equation set, proceed to the next equation. Otherwise, mark the equation by placing a 1 or 2 in the first byte of its link field, and set EQVFLAG to 1.

\(^2\) In order to avoid following long chains in future searches through the EQV, a direct pointer is set from the variable being traced to the last variable in the chain.

\(^3\) In the case where \( \text{variable}_2 \) points to a constant, \( \text{variable}_1 \) is made to point to the same constant (or its complement) and its type is set to 'constant.'
5.4 Expression = Expression

If the equation is in any of the following forms, simplify
it, retest it for further simplifications and reprocess it:

\[
\begin{align*}
E_1/E_2 &= 0 \quad \Rightarrow E_1 = 0 \\
E_1C &= 0 \quad \Rightarrow E_1 = 0 \quad \text{for } C \neq 0 \\
CE_1 &= 0 \quad \Rightarrow E_1 = 0 \quad \text{for } C \neq 0 \\
E_1E_2 &= 0 \quad \Rightarrow E_1 = 0 \\
E_1 + E_2 &= 0 \quad \Rightarrow E_1 = -E_2 \\
E_1/E_2 &= E_3 \quad \Rightarrow E_1 = E_2E_3 \\
E_3 &= E_1/E_2 \quad \Rightarrow E_1 = E_2E_3 \\
E_1/E_2 &= E_3/E_4 \quad \Rightarrow E_1E_4 = E_2E_3 \\
\end{align*}
\]

If from the first test it is found that the equation fits none
of the above categories, place it in the set of general equations (by
putting a 3 in the first byte of its link field) and proceed to the
next equation.
6. THE PROCESS ROUTINE

Given a pointer to the top node of an expression, PROCESS will simplify the expression and determine its type. The types of lower level expressions (i.e. operand expressions) are used in determining the type of the current expression. Thus pairs of numbers (EXPTP, S1S2EXP) representing each lower level expression are transmitted recursively, from the bottom up.

The top node of each expression determines the subsequent steps to be taken, as follows:

6.1 Constant
Determine whether the constant is zero or non-zero, and set EXPTP to 0 or 1 accordingly. Set S1S2EXP to 1. Return.

6.2 Global Variable or TIME
Set EQV type to S₁ if node is a global, or to S₂ if it is TIME. Go to 5.3.

6.3 Other Variable
If unused so far, make the variable type L. If type L or M, set S1S2EXP to 0. If type S₁ or S₂ set S1S2EXP to 1 or 2, respectively. If type +V or -V, follow pointer chain and repeat 5.3. If type 'constant,' replace node in tree with a corresponding constant node, and go to 5.1. Set EXPTP to 2 (single variable) and return.
6.4 Binary Operation

Process the left and right expressions, remembering their respective types. S1S2EXP for the current expression is determined from the following table:

<table>
<thead>
<tr>
<th>S1S2EXP</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Simplify unary minuses according to the rules of algebra, placing any remaining unary minus above the current node (in most cases, unary minuses can be successively bubbled up in this manner, so that upon reaching the top of the equation at most one unary minus remains).

If the operation is addition or subtraction, eliminate any zeros\(^1\) on either side of the expression. Add or subtract constants appearing in constant nodes or doublet expressions \((C+V)\), keeping the resultant constant on the left branch (either of the current node or of the left doublet). Determine EXPTP from the left and right expression types according to the table below, and return.

---

\(^1\) If the top node of the equation is being processed (with '=' replaced by '-') zeros are preserved.
If the operation is multiplication or division, eliminate any 1's or -1's on either side of the expression. If either side is 0, make the current node a 0. If both sides are constants, evaluate the expression. From S1S2LX and S1S2RX determine whether the expression is non-linear under multiplication or division, as follows:

<table>
<thead>
<tr>
<th>CALL</th>
<th>0</th>
<th>0</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>NONLIN</td>
<td>*</td>
<td>,</td>
<td>/</td>
</tr>
</tbody>
</table>

If it is non-linear, call NONLIN (a routine which sets all L-type variables in the subtree to M-type and marks the top node of the expression as non-linear by making its op-code negative. This way, if a higher level expression is also found to be non-linear, the present subtree will not have to be rescanned). Finally, determine EXPTP from the table below, and return.

---

2 Division by zero causes an error message and automatic elimination of the equation.
For the case of a power operation, if both operands are constants, perform the exponentiation ($C_1^{C_2}$ evaluated as $e^{C_2 \ln C_1}$), and set EXPTP to 0 or 1 depending on the result. Perform the following simplifications if possible:

- $E^0 + 1$ (EXPTP + 1)
- $E^1 + E$ (EXPTP + LX)
- $0^E + 0$ (EXPTP + 0)
- $1^E + 1$ (EXPTP + 1)

If no simplification is possible, call NONLIN and set EXPTP to 5 (general expression). Return.

If the operation is max or min evaluate the expression if both operands are constants. Otherwise, call NONLIN and make EXPTP = 5. Return.

### 6.5 Unary Operation

Process the left operand. $S1S2EXP$ is passed from operand to operator unchanged (i.e. $S1S2EXP + S1S2LX$).
If the operation is differentiation, move any unary minus upwards. If the operand is an $S_1$-expression, replace the current expression with the constant '0' and set EXPTP to 0. Otherwise, call NONLIN and make EXPTP = 5. Return.

For a unary minus operation, if the operand starts with a unary minus, delete both unary minuses. If the operand is a constant, complement it and delete the unary minus. Copy EXPTP from the operand (i.e. EXPTP + LX) and return.

If the operation is function, evaluate it if the operand is a constant, replacing the expression with the result, and setting EXPTP to 0 or 1. Otherwise, call NONLIN and set EXPTP to 5. Return.

6.6 User-Defined Function

Process all the operand subexpressions, call NONLIN, set S1S2EXP to 0 and EXPTP to 5. Return.
7. THE FINAL STAGES: NAME REASSIGNMENTS, EQUATION RESTRUCTURING AND OUTPUT

Having processed the entire system of equations, WEED counts the numbers of independent $S_1$, $S_2$, $L$ and $M$ variables in the EQV and initializes each of four registers with the first numeric name to be assigned to variables in each of the four sets. As the EQV is scanned, each time a new variable is encountered its value field is overwritten with the contents of its associated register, its type set to 'assigned(+V)' and the register incremented by 1. Whenever a variable type $+V$ or $-V$ is encountered, the new name of its equivalent variable is copied in its value field, and its type set to 'assigned(+V)' or 'assigned(-V)', accordingly. 'Constant' type variables remain unchanged. As a final result, all variables will have been numbered sequentially, with all $S_1$-variables first, $S_2$-variables next, and so on.

At this point, the equations are modified so that the new variable names are contained within operation nodes. As the equations are processed, a cumulative byte count is kept for each of the three equation sets, so that an appropriate amount of core can be later requested for the output Equation Block. Each tree is traversed recursively by means of the TRAVEL Routine (MODFORM mode). As each variable is detected, it is incorporated in its parent node by over-writing the respective pointer field. An exception if made for the case where the variable is 'assigned(-V)', in which case the old variable is transformed into a unary-node whose operand field contains the new variable name. Constants are not incorporated in parent nodes, and all remaining node-pointer values are made negative.
Once all the equations have been reformatted, WEED issues a GETMAIN for the output Equation Block and calls upon the PUTOUT Routine to supervise the output operation. PUTOUT looks only at \( S_1 \)-equations to begin with, and passes them one by one to TRAVEL (TRANSFER mode), which recursively moves individual nodes to consecutive memory locations in the output area. Since the equations are squeezed into a compact form, the displacement relationships between the nodes are altered, and hence new pointer values must be set as the nodes are moved. New link pointers to the next equation can be set as soon as each equation transfer is completed.

PUTOUT repeats this process for \( S_2 \) and General Equations, and returns to WEED after setting pointers to the three sets of equations in the Table Block. Finally, the entire input area is FREEMAINed, the compiler is called, and WE ARE DONE.
BIBLIOGRAPHY


APPENDICES
GET SPACE FOR OUTPUT TABLE BLOCK, COPY CONSTANT & NAME TABLES, SET POINTERS, ZERO OUT EQUIVALENCE TABLE
Cycles + 0

Cycles + Cycles + 1
EQVFlag = 0
GET INPUT EQUATION PTR

LAST EQUATION?

EQVFlag = 0?
Cycles >= 37?

GET NEXT EQUATION, CLASSIFY IT OR ELIMINATE IT

EQV Modified?

EQVFlag + 1

SCAN EQV AND COUNT #Sl, S2, L & M VARS
ASSIGN NEW INTERNAL NAMES

SUBSTITUTE VARIABLE NAMES IN ALL EQUATIONS & MODIFY FORMAT FOR OUTPUT

GET SPACE FOR OUTPUT EQUATION BLOCK. MOVE Sl-EQNS TO Sl BLOCK
S2-EQNS TO S2 BLOCK
GEN EQNS TO GEN BLOCK
SET PTRS IN TABLE BLOCK FREEMAIN INPUT AREA

THE WEED ALGORITHM

LEAVE
THE PROCESS ROUTINE
PROCESS ROUTINE
(BINARY OPERATIONS)
EVALUATE FUNCTION

OPERATION

USE FUNCTION

OPERATION

IS OPERATED A CONST

RESULT. UPDATE

OPERATION

CALL NONLIN

EXPT + 5

return

CALL NONLIN

EXPT = 5

BUBBLE

UNARY-

(operation (if any)

N

EXPT = 5

EXPT = 0

CALL NONLIN

PROCESS ROUTINE

(UNARY OPERATIONS)