DELIGHT FOR BEGINNERS

by

B. Nye and A. Tits

Memorandum No. UCB/ERL M82/55

1 July 1982

DELIGHT FOR BEGINNERS

by

Bill Nye and Andre Tits

Memorandum No. UCB/ERL M82/55

1 July 1982

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

DELIGHT For Beginners

Abstract

This writeup is meant to help new users get started on DELIGHT. DELIGHT is an interactive optimization-based computer-aided design system which was designed to provide a friendly and flexible environment for designers working in a multitude of disciplines. It not only serves the designer who wishes only to use it as a tool to quickly meet his needs, but also provides a "guru" with an ideal environment for developing and testing new algorithms.

This writeup covers:

The writeup ends with an

and

References to supplementary sources of information in the DELIGHT Reference Manual have been included. (This writeup is actually section 1d of that manual.)

1 Introduction

DELIGHT is a very powerful interactive computer-aided-design system. From the user's point of view, it is easy to learn and use, yet contains some very powerful utility commands as well as engineering design aids.

It is hard, however, for the beginner to know where to start, and how to make the best use of the multitude of DELIGHT commands and features. The purpose of this DELIGHT_FOR_BEGINNERS writeup is to point out the important features for new users, so they can get used to the main ideas of DELIGHT and start making good use of it quickly. To learn more about the goals of the DELIGHT system and its use in engineering design, see the paper "DELIGHT: An Optimization-Based Computer-Aided Design System", by W.T. Nye, E. Polak, A.L. Sangiovanni-Vincentelli and A.L. Tits, in the Proceedings of the IEEE ISCAS, Chicago, Illinois, April 1981. See also an upcoming paper in the IEEE Proceedings.

In terminal dialogue that follows, user input has been underlined or is in**boldface**, for clarity. Also, blank lines have been inserted at various places in the terminal dialogue to separate groups of statements, and will not appear on the terminal screen.

All references to supplementary information refer to the DELIGHT Reference Manual from the University of California, Berkeley. The format is as in the example "See IF_ELSE(2d)" which refers to a manual section called IF_ELSE in part d of section 2. Another example is "See LANGUAGE_ENHANCEMENTS(A4)" which refers to part 4 of appendix A.

When using DELIGHT on the UNIX operating system, there is a command, "dlman", which may be used to obtain sections of the DELIGHT Reference Manual on the terminal screen. For the above examples, they are obtained by typing "dlman IF_ELSE" or "dlman LANGUAGE_ENHANCEMENTS", respectively. Typing "dlman CONTENTS" lists all the sections of the manual which dlman knows about.

2 Getting Started

Start DELIGHT by simply typing "DELIGHT". (If this does not work, check with your local DELIGHT guru.) The "Identifier" line that may then appear on the screen indicates some property of the 'memfile' which DELIGHT is restoring from. Memfiles are binary files which contain precompiled RATTLE procedures, variables, arrays, etc. which the user will have access to. An understanding of memfiles is not needed at this point; see the manual section CONVENTIONS(1c) for more information. After the memfile has been read, the "Welcome to DELIGHT..." message appears followed by any other messages of the day. The terminal screen may appear:

```
DELIGHT
Restoring from <memfile> ...
Identifier: Standard Optimization Memfile with Matrix Macros.

                ***** Welcome to DELIGHT *****
      A General Purpose Interactive Computing System with Graphics
                                for
   Optimization-Based Computer-Aided-Design of Engineering Systems
                          Developed by the
              Optimization-Based Computer-Aided-Design Group
                          University of California
                            Berkeley, Ca. 94720.
```

The culmination of your startup of DELIGHT is the prompt "1>". This means that DELIGHT is ready to accept commands from the terminal.

3 Typing Commands

Once you've seen the prompt "1>", you can type commands, which are requests that DELIGHT do something. Try typing "date" followed by a carriage return. (As mentioned in the introduction to this writeup, in terminal examples shown here, what you should type has been underlined or is in **boldface**. Also, (1), be sure to TYPE IN ALL BLANKS exactly as shown here, and (2), be sure to TYPE IN CAPITALS everything which is shown here in capital letters.) The terminal should appear something like:

```
1> date
Date: 06/06/82  Time: 16:50:07
1>
```

If you make a mistake typing a command name, DELIGHT will tell you as in either of the following:

```
1> datee
ERROR: Command not found: "datee"
1> Date
ERROR: Command not found: "Date"
1>
```

4 Basic RATTLE Language Statements

RATTLE, for RATfor Terminal Language Environment, is the interactive programming language used in all facets of the DELIGHT system. The basic features of RATTLE are displayed here in several sections, starting first with the easiest language features. All examples shown can (and should) be typed directly into the terminal for "hands-on" experience with RATTLE and DELIGHT.

4.1 Simple Unformatted Output

The simplest way to get the value of arbitrary numeric expressions is with the "print" statement. The "print" statement may be followed by any number of expressions as in the following examples:

```
1> print 1.3
 1.300
1> print 1.3 -1.4
 1.300 -1.400
1> print  1/3  sin(3.1416/2)  2**64
 .3333  1.000  1.845e+19
1>
```

In the third example, the operator "**" stands for exponentiation. Thus, "2**64" means 2 to the power of 64.

Notice that there are exactly two expressions in the statement "print 1.3 -1.4" (notice the space between "1.3" and "-1.4") instead of the one expression "1.3-1.4". The rule by which DELIGHT makes this distinction is explained later in section 4.4, Expressions and Assignments.

4.2 Formatted Output Using "printf"

Whereas the "print" statement does not allow any control of the format of the numbers printed, the "printf" statement does. The "printf" statement requires a quoted *format control string* followed by from 0 to 6 arguments which must be in one-to-one correspondence with *conversion specifications* in the control string. The following two examples show the output of one and then two real numbers:

```
1> printf '%r/n'  1/3
 .3333
1> printf 'min=%r  max=%r/n'  -2**8  2**9
min=-2.560e+2  max= 5.120e+2
1>
```

The control string contains two types of objects: ordinary characters, which are simply copied to the output, and conversion specifications, each of which causes conversion and printing of the next successive argument on the line. Each conversion specification is introduced by the character "%" and ended by one of the conversion characters "i", "r", "c", "s", or "p". See PRINTF(3b) in the DELIGHT Reference Manual for information on the "i", "c", "s", and "p" specifications; only the "r" conversion specification is demonstrated here.

To output a real number, "%r" may be used as in the above examples and has a default of 4 significant figures. You may control this by, e.g., using "%.7r" to get 7 significant figures printed to the right of the decimal point. The "/n" means output a NEWLINE, i.e., go to the next output line, at that point in the output; notice the results of the first example below: an extra blank line has been output due to the leading "/n" in the format control string. The second example below shows that a "/" character is output by preceding it by another "/"; the "/" character is actually an *escape character* which changes the meaning of any character it precedes.

```
1> printf '/n  A=%.6r/n  B=%.2r/n'  1.0  2k/2

  A= 1.000000
  B= 1.00+3
1> printf 'Answer is 3//4/n'
Answer is 3/4
1>
```

4.3 Number Conventions for Post-attached Units

The RATTLE language supports certain metric scale factor suffixes or post-attached units which may be attached to any number. In RATTLE, a number may be an integer such as 12 or -44, a floating point number such as 3.14159, either an integer or floating point number followed by an integer exponent such as 1e-14 or 2.65e3, or either an integer or a floating point number followed by one of the following scale factors, which may be in either upper or lower case:

```
p = 10**-12     n = 10**-9     u = 10**-6     m = 10**-3
k = 10**3       me = 10**6     g = 10**9
```

Letters immediately following a number that are not scale factors are ignored, and letters immediately following a scale factor are ignored. Hence, 10, 10v, 10VOLTS, and 10hz all represent the same number, and M, mA, MSEC, and mwatts all represent the same scale factor. The "me" scale factor is for mega-something such as 20megavolts (ouch!). Note that 1000, 1000.0, 1000hz, 1e3, 1.0e3, 1k, 1khz and .001meg all represent the same number as the following example shows:

```
    1> print 1000 1000hz 1e3 1.0e3 1k 1khz .001meg
1.000e+3  1.000e+3  1.000e+3  1.000e+3  1.000e+3  1.000e+3  1.000e+3
    1>
```

4.4 Expressions and Assignments

Now we explain why "print 1.3 -1.4" contains two expressions instead of the single expression "1.3-1.4". This is due to the following RATTLE convention on where an expression ends: Expressions end at the first blank (or at the end of the line) following balanced parenthesis. For this definition, an expression with no parenthesis is considered to have balanced parenthesis and thus ends at the first blank following it. Each of the following examples contains two expressions:

```
1> print  ( 1 + 2 + 3 )  ( (3) - (2) - (1) )
 6.000  0.000
1> print max( 1, 1.2, -3, 99 )  min( 1, 2, 3)
 9.900e+1  1.000
1>
```

The functions "max" and "min" return the maximum and minimum, respectively, of any number of numeric arguments.

Assignment statements end at the NEWLINE and have no such parenthesis rule (From now on, NEWLINE will be used to indicate the character at the end of a line). Here are a few assignment statements that you can try:

```
1> x = 1 + 2 + 3
1> y = sin(x)          -          cos(x)
1> print x y
 6.000 -1.240
1> z000=x/y+2
1>
```

4.5 Continuation of Expressions and Assignments

Both expressions and assignments may be continued on the next line if they end with a character which could not possibly legally end an expression. In particular, they are continued if they end in any of the characters:

```
+   -   *   /   ,   (   |   &
```

Note below that the prompt character changes to "}" after a partial statement has been typed in but before the complete (executable) RATTLE statement has been typed. More will be said about this later.

```
1> y = 1+
1}    2
1> y = max ( 1 , 2 , 3 ,
1}    4 , 5 , 6 )
1> print y+5/
1} 2
 8.500
1>
```

4.6 If-statements

An 'if-statement' allows you to test the value of a logical expression and execute a RATTLE statement if the logical expression is TRUE. The general form of an if-statement is:

```
if logical-expression
   RATTLE-statement
else
   RATTLE-statement
```

Unlike Fortran or Ratfor, the logical expression need not be surrounded by parenthesis. The else-clause, i.e., the word "else" and the associated RATTLE statement are

optional; if not there and the logical expression is FALSE, execution just falls through to the next statement.

The first "if" below is erroneous since "if" must be followed by a logical expression (remember the balanced parenthesis rule) whereas the second two are correct statements. After the error, "reset" is typed to return to the normal "1>" prompt state.

On the "reset" command line below there is a comment which starts with the character "#". The *DELIGHT comment convention* is that anything following a "#" character up to the end of the line is considered a comment and is discarded by the RATTLE compiler in DELIGHT.

```
1> size = 50
1> if size > 0
if size > 0
           ^
ERROR(1)    assignment syntax error  [ siz   >0 ]
1} reset                # (Back to normal prompt state.)
1> if ( size > 0 )
1}    print size
1} go
 5.000e+1
1> if size>0
1}    print size
1} go
 5.000e+1
1> if size==50
1}    print size
1} else
1}    print -size
 5.000e+1
1>
```

Note above that the if-statement does not execute immediately since it is waiting for a possible else-clause; typing "go" forces it to execute. Of course, the else-clause may be given, as in the last example, and no "go" will be needed.

The last example above also shows the RATTLE relational operator, "==", for checking for equality of two quantities. The logical expression "A==B" is true if variable A equals variable B. Other RATTLE arithmetic, relational, and logical operators along with their precedence are shown in the following list. The upper entries have "higher

precedence" than the lower: "A+B*C" is automatically grouped as "A+(B*C)" since "*" has a higher precedence (lower numeric value) in column one of the table than "+". Operators with the same precedence value in column one are grouped left to right: "A*B/C" is automatically grouped as "(A*B)/C". As a final example, "a>=b|c!=d&e==f" is grouped as "(a>=b)|((c!=d)&(e==f))".

Precedence	Operator	Meaning
1	**	Exponentiation
2	*	Multiplication
2	/	Division
3	+	Addition
3	-	Subtraction
4	<=	Less than or equal to
4	<	Less than
4	==	Equal to
4	!=	Not equal to
4	>=	Greater than or equal to
4	>	Greater than
5	!	Logical not
6	&	Logical and
7	|	Logical or

4.7 Statement Blocks

To make a RATTLE statement such as "if", "for", "repeat", or "while" act on more than one statement, the statements must be surrounded by curly brackets. With if-statements, this allows you to program the idea: "if something is true, do this group of things". Here are a couple of if-statements which use statement blocks:

```
1> if ( size > 0 )  {
1}    print size
1}    print -size
1}    }
1} go
 5.000e+1
-5.000e+1
```

```
1> if size>0 { printf 'size = %i/n' size ; print size**2 }
1} go
size = 50
 2.500e+3
1>
```

The last example shows that the RATTLE statement of the if-statement can be on the same line as the "if", though this is not nearly as clear a programming style as the use of indentation in all the other if-statement examples above. The semicolon in the last example separates two statements as explained next.

4.8 Separation of Statements by Semicolons

Statements (or commands) may be separated by semicolons on the same line as shown in the following examples:

```
1> print 1 ; print 3/2 ; date
 1.000
 1.500
Date: 06/06/82  Time: 17:09:12
1> if 5<8 { printf 'Blah.../n' ; x=0 ; print sin(x) }
1} go
Blah...
 0.000
1>
```

4.9 Using "case1", "case2", ... to Make a Case Statement

By using the built-in 'case-' defines (see LANGUAGE_ENHANCEMENTS(A4)) which define "case1" as "if" and "case2", "case3", ... as "else if", you can make a case statement as shown in the following example:

```
1> x = 3
1> case1 ( x == 1 )  printf 'what?/n'
1} case2 ( x == 2 )  printf 'who?/n'
1} case3 ( x == 3 )  printf 'where?/n'
```

```
1} case4 ( x == 4 )  printf 'why?/n'
1} else              printf 'Illegal entry: %i/n'  x
where?
1>
```

This becomes a series of 'else-if's in which the logical conditions are evaluated one at a time until one is found which is true. Then, its associated statement is executed and the entire 'case' group is exited. If none of the logical conditions is true then the "else" statement, if one is provided, is executed. See IF_ELSE(2d) for an additional warning about using if-statements in such a case statement.

4.10 Loop Statements: While, Repeat-Until and For

These RATTLE statements, like the if-statement, take exactly one RATTLE statement as their 'body', unless several statements are surrounded in curly brackets. Consider the following four ways to add up the entries in a one-dimensional array. When typing in any of the examples in this writeup, you need not type in the comments (anything following a "#"); they are there for clarification and indeed are not underlined or in **boldface** type.

```
1> array z(10)        # Create the array.
1> for i = 1 to 10    # Initialize the array: z(1)=1,
1}    z(i) = i        # z(2)=2, z(3)=3, etc.
1>

1> sum1 = 0                          # METHOD 1
1> for i = 1 to 10
1}    sum1 = sum1 + z(i)
1>

1> sum2 = 0                          # METHOD 2
1> for ( i=1 ; i<=10 ; i=i+1 )
1}    sum2 = sum2 + z(i)
1>
```

```
1> sum3 = 0                          # METHOD 3
1> i = 1
1> while ( i <= 10 )  {
1}    sum3 = sum3 + z(i)
1}    i = i + 1
1}    }
1>

1> sum4 = 0                          # METHOD 4
1> i = 0
1> repeat  {
1}    i = i + 1
1}    sum4 = sum4 + z(i)
1}    }
1} until ( i == 10 )
1>

1> ## Now check the results.
1> printf '%i %i %i %i/n' sum1 sum2 sum3 sum4
55 55 55 55
1> ## GREAT !!!
1>
```

See LOOP_STATEMENTS(2e) for additional information about these loop statements, especially the not-so-obvious for-loop used in METHOD 2.

4.11 Breaking Out of Loops with the "Break" statement

The "break" statement allows you to leave any RATTLE loop before the normal loop termination. Execution resumes with the statement following the 'body' or last statement in the loop. In this example, the inner "j" loop normally would execute 6 times but due to the if...break statement it only executes 3 times, as seen in the output:

```
1> for i = 1 to 2  {
1}    printf '%i/n'  i
1}    for j = 1 to 6 {
1}       printf '    %i/n'  j
1}       if ( j == 3 )  break
1}       }
1}    }
```


The [illegible] formal loop

```
1
    1                    (Note, here, that j is never
    2                     printed greater than 3.)
    3
2
    1
    2
    3
1>
```

4.12 Procedures and Functions

Procedures in RATTLE are analogous to subroutines in Fortran or Ratfor. They allow you to execute as a unit a group of RATTLE statements, the 'body' of the procedure, which are compiled only once. A function is identical in structure except that it contains one or more "return" statements to specify the value to be returned as the 'function value'. Also, the function 'call' or invocation appears in an expression, as in "print 2+funval(5)".

A procedure or function can have zero or more arguments. If a function has no arguments, it can still be called in any expression by following its name with a set of empty parenthesis as in "print 5+fval() fval()/2".

The body of a procedure consists of one RATTLE statement. If more than one statement is desired in the procedure body, they must be surrounded by curly brackets, similarly to the body of loop statements. Exit from a procedure body is automatic when "hitting the bottom", i.e., after the bottom statement has been executed. To exit from any other place, a "return" statement may be used. For a function, the function value to return is the expression value following the word "return".

Here are several function and procedure examples to try, each example separated by a blank line:

```
1> function foo
1}     return 5+3
1> print foo()
 8.000
1>
```

```
1> function foo (x)
                ^
WARNING(1)  Number of arguments changed on an existing procedure
1}     return ( x + 4 )
1> y = 1
1> print foo(1)  foo(y)  foo(-4)  foo(-2*2*y)
 5.000  5.000  0.000  0.000
1>

1> procedure doit (a,b)  {
1}     if ( a == 1 )  print b
1}     else           print -b
1}     printf 'Leaving doit/n'
1}     }
1> doit(1,5)
 5.000
Leaving doit
1> doit(2,5)
-5.000
Leaving doit
1>
```

Note that when function "foo" is defined a second time above, the new function body completely supersedes the previous one.

Lets make a procedure in a file and accidently leave off the closing curly bracket. Then we type "include filename", which causes DELIGHT to read the lines from the file just as if they had been typed at the terminal, and watch what happens. For this example, the built-in UNIX-like editor of DELIGHT is being used. See section EDIT(F) in the DELIGHT reference manual for details on how to use this editor. Below, the "a" command takes you into the editor input mode while a single "." alone on a line takes you out of input mode and back into the editor command mode. The command "1z" prints one screen full of edit buffer text starting at line 1. "wq" is a combination command which writes out the edit buffer back into the file and then quits (leaves) the editor.

```
1> edit foo
Unable to open "foo"
:a
procedure foo (x)  {
   if ( x == 1 )  print x
   else           print -1/x
.
:1z
procedure foo (x)  {
   if ( x == 1 )  print x
   else           print -1/x
:wq
"foo" 3 lines
1> use foo        ## NOTE: "use" is defined to be "include".
1}
```

At this point, we notice that the prompt character is different: the "}" in the prompt sometimes indicates that a "}" is expected, as here. One can either type "reset" or supply the closing "}". If the closing curly bracket IS supplied, the procedure in the file just included will then be correctly RATTLE compiled and ready to use. This is shown here followed by the 'call' or execution of the procedure with the argument "2" and the addition of the missing "}" to the file:

```
1} }
1> foo(2)
-.5000
1> edit foo
"foo" 3 lines
:a
   }
.
:1z
procedure foo (x)  {
   if ( x == 1 )  print x
   else           print -1/x
   }
:wq
"foo" 4 lines
1>
```


around [illegible] exclusive lock here. One can either type

5 Interrupting RATTLE Execution: Hard Interrupts

DELIGHT recognizes two kinds of terminal-generated interrupts, 'hard' interrupts and 'soft' interrupts. A 'hard' interrupt is when you press the terminal's interrupt or break key twice in succession; a 'soft' interrupt is just once. (In the rest of this write-up, we say "press the break key" to mean generate an interrupt even though the means of generating interrupts on a terminal is highly machine dependent and may not actually involve the "break" key. In fact, the "break" key might abort or kill the entire DELIGHT program so watch out!)

Type in the next loop and generate a 'hard' interrupt after, say, 5 seconds by pressing the terminal's break key twice in succession.

```
1> for i = 1 to 20k
1}     j = i**3 + i**2 + i

Interrupt...
2> print i
 5.920e+2
2>
```

Notice that the prompter is "2>", indicating that you are in a suspended state. The value of variable i printed will likely be different than the one shown above (probably much larger if you're on one of those damn fast Crays!).

Interrupt the next loop also:

```
2> for k = 1 to 5k
2}     j = k

Interrupt...
3>
```

Now you are in interrupt level 3. The maximum interrupt level is 5. Execution of the "for k ..." loop may be resumed by typing "resume":

```
3> print j k
 1.319e+3  1.319e+3
3> resume
2> print j k
 5.000e+3  5.001e+3
2>
```

The "2>" prompt before "print j k" indicates that the "for k ..." loop has completed. Now, the "for i ..." loop may be resumed by typing "resume". In the following example, generate a "hard" interrupt by pressing the terminals interrupt or break key twice where shown in parenthesis:

```
2> resume        (After 3 seconds, generate "hard" interrupt)

Interrupt...
2> print i
 4.495e+3
2> ## Creeping slowly ...
2> resume        (After 3 seconds, generate "hard" interrupt)

Interrupt...
2> print i
 6.494e+3
2> ## Forget it !!!
2> reset
1> resume
Nothing to resume.
1>
```

The "reset" command has again taken you out of the interrupted state.

6 Catching 'Soft' Interrupts

A 'soft' interrupt is generated when the interrupt key is pressed just once. An executing procedure can detect such an interrupt by using "interrupt" in an if-statement test, as shown in the example below. This can be used to alter program flow or to suspend execution using the "suspend" statement (see INTERRUPT_EOF(A6)). Stopping execution with a soft interrupt thereby allows your procedure to suspend at a 'major stopping point' instead of at some arbitrary statement as when generating a hard interrupt. Of course,you may "resume" after either type of interrupt. See also INTERRUPTS(2k)

and INTERRUPT_EOF(A6) for more information.

In the following procedure, the if-statement in the for-loop tests for a soft interrupt. If one is detected, a message is printed and execution is suspended via the "suspend" statement. After you type in the procedure, type "catch()" as shown below to execute it and hit the break key after approximately 1 second:

```
1> procedure catch  {
1}   printf 'Entering catch '
1}   printf 'generate a soft interrupt after 1 second./n'
1}   for i = 1 to 20k
1}      if interrupt  {
1}         printf 'Got the interrupt when i = %i/n'  i
1}         suspend
1}         }
1}   printf 'Sorry you did not send an interrupt in time !/n'
1}   }
1>
1> catch()
Entering catch, generate an interrupt after 1 second.
Got the interrupt when i = 2077

Interrupt ...
2> reset
1>
```

7 Additional RATTLE Input and Output Features

We now show some more advanced RATTLE features for input and output. In particular, for simple scalar variables or arrays with one or two dimensions, the "printv" command may be used to label and display the scalar's value or the entire array very easily:

```
1> array y1(3), y2(2,5)
1> printv y1
Column y1(3):
  0
  0
  0
```

```
1> y2(1,1) = 1.23
1> printv y2
Matrix y2(2,5):
  1.23  0.00  0.00  0.00  0.00
  0.00  0.00  0.00  0.00  0.00
1> for i = 1 to 2
1}    for j = 1 to 5
1}       y2(i,j) = i+j
1> printv y2
Matrix y2(2,5):
  2   3   4   5   6
  3   4   5   6   7
1> printv i
Scalar i =  3.000
1>
```

Note that both arrays printed are initially identically zero; arrays are automatically initialized to zero when declared. This is explained further in ARRAYS(2g). Also note that "printv" is 'smart': it only prints the number of significant figures needed to show the 'longest' floating-point mantissa.

For a complete discussion of RATTLE input possibilities, see manual sections RATTLE_I/O(3a), READF(3c), and READF(B23). Here we only show how numbers are read.

There is a system for input called "readf" which parallels the "printf" statement. "readf" also requires a format control string followed by from 0 to 6 arguments, which correspond to the "%" conversion specifications in the control string. The slash character, "/", here, is also the "escape character" as for "printf". "/n" appearing in the control string causes the remainder of the input line to be skipped over and discarded, i.e., it causes a skip to past the next NEWLINE character. A blank appearing in the control string causes any number of blanks (or tabs) to be skipped over in the input.

Here are two examples; the first reads one number while the second reads three numbers separated by any number of blanks (note the blanks in the second control string):

```
1> readf '%r/n'  x
7.2
1> print x
 7.200
```

```
1> readf ' %r %r %r/n'  x  y  z000
    1.2     -2.3     345
1> print x y z000
 1.200 -2.300  3.450e+2
1>
```

To input numbers from a file, you may use the "input_from" and "inputf" statements. "inputf" works just like "readf" except that it reads from the file specified by "input_from". In the following example we first create a file with some numbers in it:

```
1> edit temp2
Unable to open "temp2"
:a
1 2 3
4
5
6
.
:wq
"temp2" 4 lines
1> list temp2
------------------- Begin temp2 -------------------
1 2 3
4
5
6
-------------------- End temp2 --------------------
1> procedure read_temp2  {
1}    input_from temp2          ## Beginning of input_from.
1}    array y(6)
1}    for i = 1 to 3
1}       inputf ' %r'  y(i)     ## Reading from the same line.
1}    inputf '/n'               ## Skip past the NEWLINE.
1}    inputf '%r/n'  y(4)       ## Reading from separate lines.
1}    inputf '%r/n'  y(5)
1}    inputf '%r/n'  y(6)
1}    printv y
1}    input_end                 ## End of input_from.
1}    }
```


Bypass the NEWLINE.

```
1> read_temp2()
Column y(6):
  1
  2
  3
  4
  5
  6
1> ## HEY, IT WORKED !!!
1>
```

8 Advanced Output Formatting

The first example below shows that the field width of a "%r" field may be controlled by giving the width directly after the "%" character. To left justify the number in the field, follow the "%" with the "-" character. To specify the number of significant figures that will be printed to the right of the decimal point, use ".6", for example, after the field width. Field width and left justification also work for integer fields.

```
1> printf '"%r"  (%12r)  <%-12r>/n'  1.234k  1.234k  1.234k
" 1.234e+3"  (    1.234e+3)  < 1.234e+3   >
1> printf '(%15.6r)/n'  1k/6
(    1.666667e+2)
1> printf '[%5i]  (%-5i)/n'  25  25
[   25]  (25   )
1>
```

9 Extensibility through Defines

The extensibility of RATTLE, that is, the ability to extend the language by defining new language statements and commands, has probably contributed the most to its success among advanced RATTLE users. This capability takes two forms, defines and macros, both of which can greatly enhance program readability. Defines are discussed here; see manual section MACROS(4c) for a discussion of RATTLE macros.

Defines, patterned after those in Ratfor, allow you, in the simplest usage, to substitute one piece of text for another. For example, "define (TWOPI,6.283185307)" allows you to use the value of the mathematical constant 2*pi in expressions without actually writing the whole number out. An example of using defines to create a command would be "define (erase,greras())" which allows you to type "erase" to call built-in routine greras to erase the graphics screen.

Several extensions to defines include (1) arguments, (2) literal strings, which must appear explicitly when using the define, (3) optional arguments, and (4) default values for the optional arguments. Consider the define below. It has been broken into two lines in this document contrary to the DELIGHT define conventions (see manual section DEFINES(4b)).

```
define (vector x1 y1 x2 y2 ; 'in' ''color='white' ,
                grcolp(color) ; grvect(x1,y1,x2,y2) )
```

This define, "vector", has 4 required arguments, x1, y1, x2, and y2 (which appear before the first semicolon), and 1 optional argument, color. All arguments after the semicolon are optional and may be followed by "=" and the default characters which will be substituted where their argument name appears in the definition (which follows the comma). In the above define, "in" has to be typed explicitly when using the define. The definition, appearing after the comma, contains two calls to built-in DELIGHT subroutines, grcolp and grvect. The terminal dialogue below will introduce each of these features individually.

This define could be used in any of the following ways:

```
vector 0 0 1 1
vector 0 0 1 1 in red
vector  sin(x)  1+sin(x)  cos(x)  1+cos(x)  in  orange
```

If you used the second example above in a procedure it would be, using the definition above, as if you had used:

```
grcolp('red') ; grvect(0,0,1,1)
```

The following terminal dialogue shows the creation and use of several defines, starting first with the simplest type of define and then proceeding to the various extensions mentioned above. The first group shows the simplest type of defines, the second group extends these to defines with arguments, and the third shows the literal string "over" which must be present when using the define. Also shown are several invalid uses of the define and their resulting error messages.

```
1> define (d,4)                          ## Simple defines.
1> define (a,5) define (b,6)
1> print d a b**2
 4.000  5.000  3.600e+1
1> whatis d
"d" is a define:  "4".
1>

1> define (p x,print x**2)               ## Defines with arguments.
1> p 5
 2.500e+1
1>

1> define (p x 'over' y,print x/y) ## Defines with literal strings
1> p 5
ERROR: expecting "over " for define "p"
1> p 5 over
ERROR: missing arguments after "p"
1> p 5 over 2
 2.500
1>
```

Shown next is the extension which allows optional arguments which may have default values. In the first example, "x" comes after the semicolon and is thus an optional argument. If no argument is typed after "p", "x" takes on the default value (actually, just a string substitution) of "8". In the next group of lines, a multi-line define is shown. It does not have a leading left parenthesis and it ends with the keyword "end".

```
1> define (p ; x=8 , print x)            ## Defines with optional
1>                                       ## arguments with default
1>                                       ## values.
1> p
 8.000
1> p 2
 2.000

1> define p x
   print x
   print -x
   end
```

```
1> p 5
 5.000
-5.000
1>
```

Note that each time "p" above is redefined, the new definition completely supersedes the previous definition.

For additional information on defines, see manual section DEFINE(4b).

10 Commonly Used DELIGHT Commands

The "display" command, with its two or three arguments, permits you to view all the DELIGHT symbol table entries which are of a specified class and which match an optionally specified pattern. In this pattern, there are only two 'magic' characters, star (asterisk,"*") and question mark ("?"). "*" represents a match of zero or more of any character, e.g., appearing alone it matches any symbol table entry name, while "?" matches any single character in an entry name. Thus, the command "display defines L*" would display all of the user's defines starting with the letter L. Similarly, "display variables ??x" would display all the user's variables containing three characters and ending in "x".

When starting DELIGHT from the basic optimization memfile, several global variables already exist as shown in the following dialogue. Other RATTLE variables, created during the course of this terminal session, also are listed.

```
1> display variables *

21 variables:

   Iter              =   0.00000         GLOBAL
   NXseqsize         =   2.00000e+1      GLOBAL
   Neq               =   0.00000         GLOBAL
   Nfineq            =   0.00000         GLOBAL
   Nineq             =   0.00000         GLOBAL
   Nparam            =   0.00000         GLOBAL
   Nsvbound          =   0.00000         GLOBAL
   Penalty_param     =   0.00000
   W0                =   0.00000
   Wc                =   0.00000
   i                 =   3.00000
```

```
   j                    =   6.00000
   k                    =   5.00100e+3
   size                 =   5.00000e+1
   sum1                 =   5.50000e+1
   sum2                 =   5.50000e+1
   sum3                 =   5.50000e+1
   sum4                 =   5.50000e+1
   x                    =   1.20000
   y                    =  -2.30000
   z000                 =   3.40000

1> display defines *

4 defines:

   a
   b
   d
   p

1> display arrays *

6 arrays:

   Meq                  (0)                  GLOBAL
   Mfineq               (0,0)                GLOBAL
   Mineq                (0)                  GLOBAL
   y1                   (3)
   y2                   (2,5)
   z                    (10)
1>
```

For more information on the "display" command, see manual section COMMAND_DEFINES(A3).

Another useful command is "display_time" which displays a list of procedure names, total cpu time, direct cpu time, and number of times called for RATTLE procedures and built-in routines, sorted by total cpu time (largest first). Direct cpu time is the total amount of time actually spent in a procedure but NOT in any procedure called by it. The number of items listed is specified by an optional argument which has default value 10. "clear_time" resets all the call-counts and the cpu time values to zero, i.e., the "display_time" quantities are since the last "clear_time". One of the uses of these commands is for pin-pointing the major cpu time bottlenecks in a program so that

they may be made more efficient.

```
1> display_time
  TOTAL     DIRECT    NUMBER
 SECONDS   SECONDS  OF CALLS  PROCEDURE/MACRO NAME
 5.10e+2     ----      ---     Cpu-time since last "clear_time"
 3.51e+1   3.51e+1      35     trmcom_
 2.67e+1   2.67e+1       5     exedit
 2.30e+1   0.00         65     matop
 1.74e+1   1.25e+1    4041     pbstpl_
 1.35e+1   1.16e+1     381     pbnuml_
 1.04e+1   0.00         21     mat_func_
 1.04e+1   7.00e-3       2     ego_
 1.03e+1   0.00         21     fill
 1.03e+1   0.00          6     quadprog
 9.51      0.00         17     clip
```

All the procedures and macros listed above are part of the DELIGHT system software. The total cpu times of your procedures foo, doit, read_temp2, and catch have been totally swamped by those listed above. They would appear if you had typed, for example, "display_time 10000". Alternatively, try typing the following:

```
1> clear_time
1> foo(2)
-.5000
1> for i = 1 to 5
1}    doit (i,3)
 3.000
Leaving doit
-3.000
Leaving doit
-3.000
Leaving doit
-3.000
Leaving doit
-3.000
Leaving doit
```


"display_time" displays the cumulative cpu time since the last "clear_time"

```
1> display_time
  TOTAL     DIRECT    NUMBER
 SECONDS   SECONDS  OF CALLS  PROCEDURE/MACRO NAME
 3.27        ----      ---     Cpu-time since last "clear_time"
 .167       .100         5     doit
 6.67e-2   6.67e-2        5     prinf6
 1.67e-2   1.67e-2        1     foo
```

The built-in DELIGHT routine prinf6 listed above is called by the "printf" statement in procedure "doit". These computer times are on a VAX 11/780 running Berkeley UNIX.

Additional commonly used commands that you can try are "echo" and "noecho" for turning on and off the echo of lines from the terminal or from a file, "time" for showing the total cpu time since starting DELIGHT and the difference in cpu time since the last "time" command, "date" for printing the current date and time of day, and "history" for showing the last 22 input lines entered from the terminal. Also "whatis", and "remove" pertain to DELIGHT (symbol table) items such as defines, variables, procedures, etc.

```
1> echo
1> print 1 2 3
>> print 1 2 3
 1.000  2.000  3.000
1> time
>> time
Cpu time:  Total 16   Delta 16 (seconds)
1> date
>> date
Date: 06/06/82  Time: 18:09:12
1> noecho
>> noecho
1> history
  47 p 5
  48 p 5 over
  49 p 5 over 2
  50 define (p ; x=8 , print x)
  51 p
  52 p 2
  53    end
  54 p 5
  55 display variables *
  56 display defines *
```

```
  57 display arrays *
  58 display_time
  59 clear_time
  60 foo(2)
  61 for i = 1 to 5
  62    doit (i,3)
  63 display_time
  64 echo
  65 print 1 2 3
  66 time
  67 date
  68 noecho
1> whatis y
"y" is a variable.
1> remove y
"y" removed.
1> whatis y
"y" does not exist.
1>
```

As shown above, lines echoed due to "echo" are preceded by ">>". Also notice that blank command/statement input lines (i.e., if you just hit the RETURN key) do not appear in the history list.

To see how "echo" also shows lines being read from a file, turn on input echo and reuse the file "foo" created in section 4.12 of this writeup:

```
1> echo
1> use foo
>> procedure foo (x)  {
>>    if ( x == 1 )  print x
>>    else           print -1/x
>>    }
1> noecho
>> noecho
1>
```

Now let's take a look at some DELIGHT features to aid in debugging RATTLE procedures. We discuss the "trace" and "enter" commands.

The "trace" command is very useful for debugging DELIGHT "RUN-TIME ERROR"s. Type

in the following:

```
1> foo(0)

RUN-TIME ERROR: 1 overflow(s) or other floating point exception(s)
 0.000

Interrupt...
2> trace
Interrupted IN procedure
          foo                line 4    of file foo
2> reset
1>
```

Using the editor, you can now locate the source of the RUN-TIME ERROR: it is on or near line 4 of file foo. Obviously, it is due to the division by x with x equal to zero on line 3. The "trace" command can be used whenever an interrupt has occurred, i.e., when the interrupt level is greater than one (prompt is "2>", "3>", etc.)

Another DELIGHT feature to aid in debugging is the "enter" command. Let us create a simple procedure with three local variables, a, b, and c:

```
1> procedure foo (x) {
1}     a = 1
1}     b = 2
1}     c = 3
1}     print a*x b*x c*x
1}     }
1> foo(2)
 2.000   4.000   6.000
1>
```

After executing this procedure as above, we may "enter foo" and look at the local variables:

```
1> enter foo
e> display local variables *

3 variables:

   a                    =   1.00000
   b                    =   2.00000
   c                    =   3.00000

e> leave
1>
```

Note that after entering a procedure with the "enter" command, the prompt changes to "e>" to remind you that any variables you create or use are actually local to the entered procedure. (See manual section PROCEDURES(2h) for additional information on procedure local variables.) Later, in section 14 of this writeup, the use of "enter" to aid in debugging is more fully demonstrated.

The real purpose of the history list of input lines is not to look at the lines displayed by the "history" command but to be able to re-issue previous commands or lines easily. You may re-issue a command by typing, for example, "!foo" if the command had started with the letters "foo" or "!23" if the command had been number 23 in the "history" command output. If "!foo" is typed, DELIGHT looks up the history list, starting from the bottom (most recently typed line), and reuses the first command line it finds which begins with the letters "foo". The command to be reused is first printed on the terminal.

```
1> history 6                ## Display last 6 input lines.
  84    print a*x b*x c*x
  85    }
  86 foo(2)
  87 enter foo
  88 display local variables *
  89 leave
1> !fo
foo(2)
 2.000  4.000  6.000
1> !86
foo(2)
 2.000  4.000  6.000
1>
```

Many times, one would like to loop in a manner other than the usual arithmetic increment of some loop variable. For example, one might want to loop from 10 to 100 with 5 points per decade, logarithmically spaced between 10 and 100. The "loop" macro extends the for-loop capabilities of RATTLE by substituting a RATTLE for-loop which is generally much messier and less readable than the actual "loop" statement. For the above example, "loop x from 10 to 100 dec 5" could be used and would cause the following code to actually be RATTLE compiled:

```
by_ = 10**(1/5)
for ( x=10 ; x<=100 ; x=x*by_ )
```

Using the "oct" keyword instead of "dec" would allow you to specify the number of loop passes per octave instead of per decade. The "times" keyword causes the loop variable to be increased (or decreased) by multiplying it by the value of the specified factor. The "dec" and "times" cases are illustrated in the following simple loop statements:

```
1> loop i from 1 to 100 dec 2
1}    print i
 1.000
 3.162
 1.000e+1
 3.162e+1
 1.000e+2
1> loop x from 4 to 64 times 2
1}    printf '%i/n' x
4
8
16
32
64
1>
```

See manual section LOOP(B18) for more information on the loop macro.

11 Redirection of Output or Echo to a File

"echo_o_to filename" is used to cause a copy (an echo) of all DELIGHT output to be sent to a file as well as to the terminal screen. Similarly, "echo_io_to filename" causes both DELIGHT input and output to echo to the same file. (Only one of "echo_o_to" or "echo_io_to" should be used at a time.) Another similar command is "output_to filename" which sends all succeeding output to a file but NOT to the terminal screen. To terminate any of these, "echo_o_end", "echo_io_end", or "output_end" are used, respectively. WARNING: be sure you "end" the writing to a file before trying to list the file with the "list" command!

These commands are demonstrated below, starting with echo_io_to:

```
1> echo_io_to t1
Created file "t1"
1> date
Date: 06/06/82  Time: 20:25:33
1> print 1/3
 .3333
1> ?                # Show input, output, echo redirection.
NO input redirection.
NO output redirection.
Echo output going to "t1".
1> echo_io_end
1> list t1
-------------------- Begin t1 --------------------
>> date
Date: 06/06/82  Time: 20:25:33
>> print 1/3
 .3333
>> ?
NO input redirection.
NO output redirection.
Echo output going to "t1".
>> echo_io_end
--------------------- End t1 ---------------------
1> ?
NO input redirection.
NO output redirection.
NO echo output set.
1>
```

Now let's try an "output_to" example. Here, ONLY the output goes ONLY to the file you specify. The "?" macro, which reports on input, output, and echo redirection, always prints on the terminal, as seen below:

```
1> output_to t2
Created file "t2"
1> date
1> array B(2,3)
1> printv B
1> ?
NO input redirection.
Output redirected to "t2".
NO echo output set.
1> output_end
1> list t2
-------------------- Begin t2 --------------------
Date: 06/06/82  Time: 20:29:35
Matrix B(2,3):
  0  0  0
  0  0  0
--------------------- End t2 ---------------------
1>
```

For more information on these commands see manual sections RATTLE_I/O(3a), OUTPUT(B19), and ECHO_TO(B8a)

12 Graphics and Plotting

One of the important user features of DELIGHT is the ability to convey information very effectively using computer graphics. DELIGHT has many high and low level, terminal-independent graphics commands for creating many types of graphical displays. A prerequisite to using these commands is that you understand the notions of 'viewport' and 'world coordinate bounds'. (Only an understanding of viewports is necessary to use the "plot" or "plot3d" commands, described below.)

The phrases 'viewport' and 'world coordinate system' are taken from Neumann and Sproul, *Principles of Interactive Computer Graphics*. A viewport is a rectangular region on the terminal screen where output is sent. The "viewport" command specifies this region in a 0-0,1-1 coordinate system in which 0,0 is at the lower left corner of the screen and 1,1 is at the upper right corner, for all graphics terminals. For example, "viewport

0 0 0.1 0.1" specifies a very small square region in the lower left-hand corner of the screen while "viewport 0.45 0.45 0.55 0.55" specifies a small region in the center of the screen. By using "viewport", graphical output may be placed anywhere on the screen by simply setting the viewport coordinates.

World coordinate bounds are the bounding x,y values between which all x,y coordinates are actually passed to the graphics commands. After "world -100 -100 100 100", which sets the lower left bound to -100,-100 and the upper right bound to 100,100, the command "cursor 0 0" would place the cursor at the very center of the present viewport, ready for text output, while "vector -100 -100 100 100" would draw a diagonal completely across the present viewport. See WINDOW(B34) for a means of giving names to viewports with prescribed viewport and world coordinate bounds.

So now you are ready for the syntax of a small subset of the graphics commands described in greater detail in manual section GRAPHICS(B15):

```
box                     - Draw box around present viewport.
color colorname         - Set present color.
cursor x1 y1            - Position cursor for text output.
erase                   - Erase the entire screen.
oval                    - Inscribe "circle" in present
                          viewport.
terminal                - Set or get terminal type.
text '%i' [args]        - Output text characters at
                          present cursor position using
                          printf-like control string.
vector x1 y1 x2 y2      - Draw vector between 2 coordinates.
viewport x1 y1 x2 y2    - Set screen viewport bounds to
                          lower-left and upper-right
                          coordinates specified.
world x1 y1 x2 y2       - Set world coordinate bounds which
                          map x,y coordinates into the
                          present viewport.
```

Before trying any of these commands you must be on one of the graphics terminals supported by DELIGHT graphics. To see what terminals are supported, type "terminal choices". Each line shown is for a different terminal or plotter. You will probably be on either an HP2648a, a Tektronix 4027, or a Tektronix 4010 terminal. Typing "terminal" alone prints the present terminal type. To specify the Tektronix 4027, for example, you would type "terminal 4027". In theory, you should type "grinit" to initialize the terminal for graphics after specifying any terminal type. But presently, only the Tektronix 4027 terminal requires that "grinit" be typed.

If you absolutely cannot get to a graphics terminal you may type "terminal dumb" and get (very) low resolution graphical output. But watch out though since after each of the graphics commands below, a flush of the dumb-terminal graphics buffer is performed. To get the graphical output of several commands on the same screen, make a statement block by preceding the first command with "{" and following the last command by "}". Only after typing the closing "}" will the buffer flush. ("Terminal dumb" can also be tried on a graphics terminal for fun.)

Some graphics commands to try are:

```
1> terminal 4027
1> terminal                    # Check terminal type.
Terminal is 4027
1> color white
1> box
1> erase
1> viewport .3 .3 .7 .7
1> box
1> world  -100 -100 100 100
1> vector -100 -100 100 100
1> color green
1> vector -100 100 -25 25
1> color red
1> cursor 0 0
1> text 'ab=%i'  26/2
1> oval
1>
```

For plotting graphs, DELIGHT provides the means of plotting arbitrary expressions versus one or two parameters (variables). The "plot" command allows you to plot up to 9 y-value expressions versus a single parameter on the same labeled axis. The axis is scaled to the min and max automatically. The syntax of the plot command is:

```
plot y1_expr [y2_expr y3_expr ... y9_expr]
  vs x_variable from from_expr to to_expr [ {by   } inc_expr ]
                                          {times}
                                          {oct  }
                                          {dec  }
```

where everything after "to_expr" is optional but you may choose one of the keywords "by",

"times", "oct", or "dec". These keywords have the same meaning as they did for the "loop" statement explained earlier. If you don't give the optional increment keyword and expression, "by 1" is assumed. For "times", "oct", and "dec" the x-axis of the graph is logarithmic. The list of up to 9 expressions may be continued onto succeeding lines, i.e., there may be NEWLINE characters between expressions (see second example below).

Plots generated by the "plot" command may be targeted for black/white or color terminals. For black and white terminals, the curves drawn can have little triangle and square identifiers placed on them to help you identify which curve goes with which y-variable expression. This feature is obtained by typing "use <gpidents>". (The triangular brackets surrounding the filename mean "look for this file in a standard place in the file system". For more on this, see "File openhdtl and <file_name> Conventions" in CONVENTIONS(1c).) To have the curves drawn in color, type "use <gpcolors>" (on the HP2648a, colors are simulated using various dashed lines and intensities).

Try the following simple plot commands:

```
1> viewport 0 0 1 1
1> use <gpcolors>     ## Use this for COLOR terminals.      ##
1> use <gpidents>     ## Use this for NON-COLOR terminals. ##
1> plot sin(x) vs x from 0 to TWOPI*2 by TWOPI/50
------ Compiling plot loop ------
1> plot sin(x) .5*sin(2*x) .2*sin(5*x)
1} vs x from 0 to TWOPI by TWOPI/100
------ Compiling plot loop ------
1> plot (1/sqrt( freq**2 + 1 )) vs freq from .01 to 100 dec 20
------ Compiling plot loop ------
1>
```

The output plots for these commands are shown in figures 12.1, 12.2, and 12.3 for the HP2648a black and white terminal (copies made on the HP2631G printer). "use <gpidents>" was selected above. Note that the x-axis of the plot is logarithmic on figure 12.3 due to the "dec" increment keyword above.

For graphing a single expression vs two parameters on a 3-dimensional plot, the "plot3d" command is used. Its syntax is:

```
plot3d z_expr vs x from x_start to x_stop by x_increment
              vs y from y_start to y_stop by y_increment
```

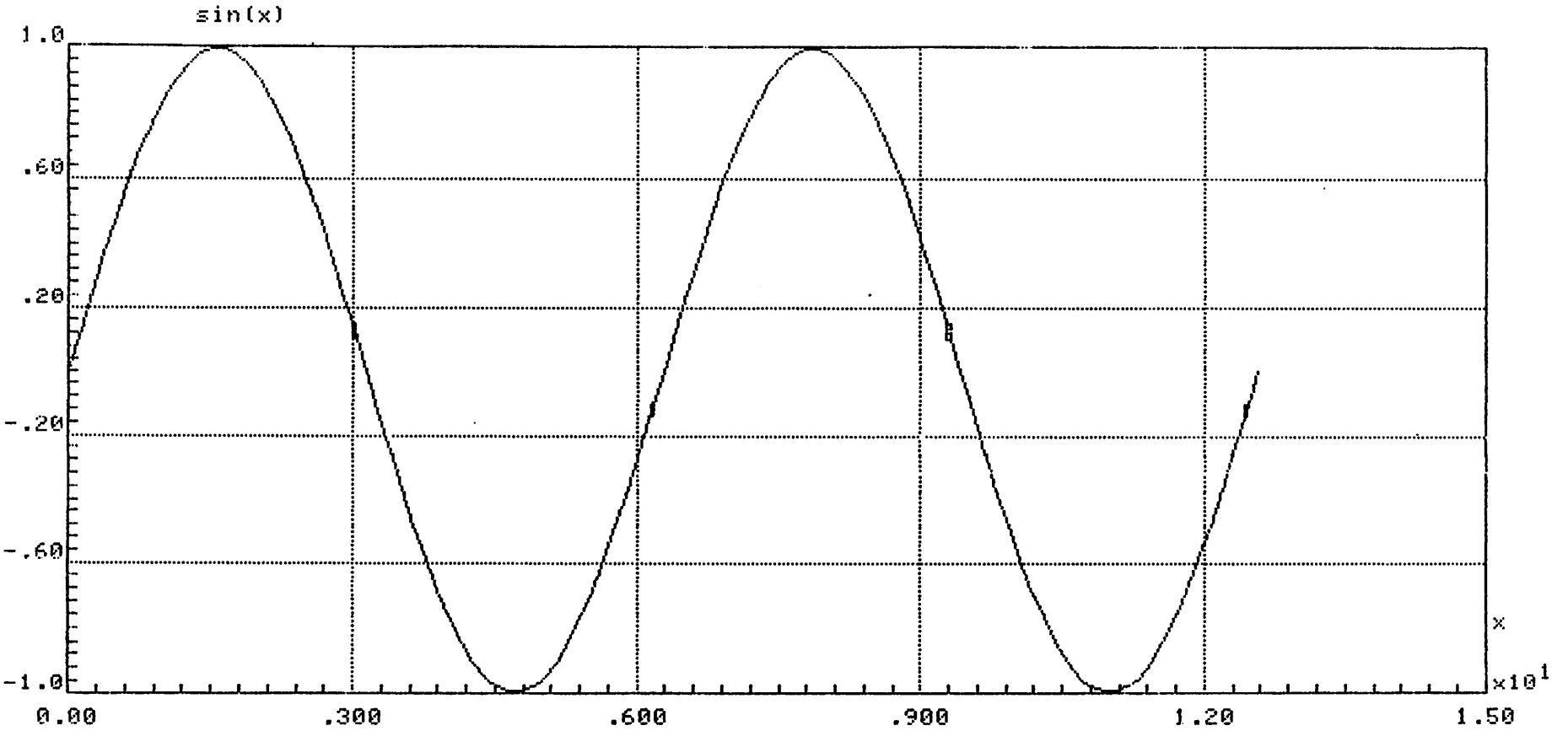


Figure 12.1: "plot sin(x) vs x from 0 to TWOPI*2 by TWOPI/50"

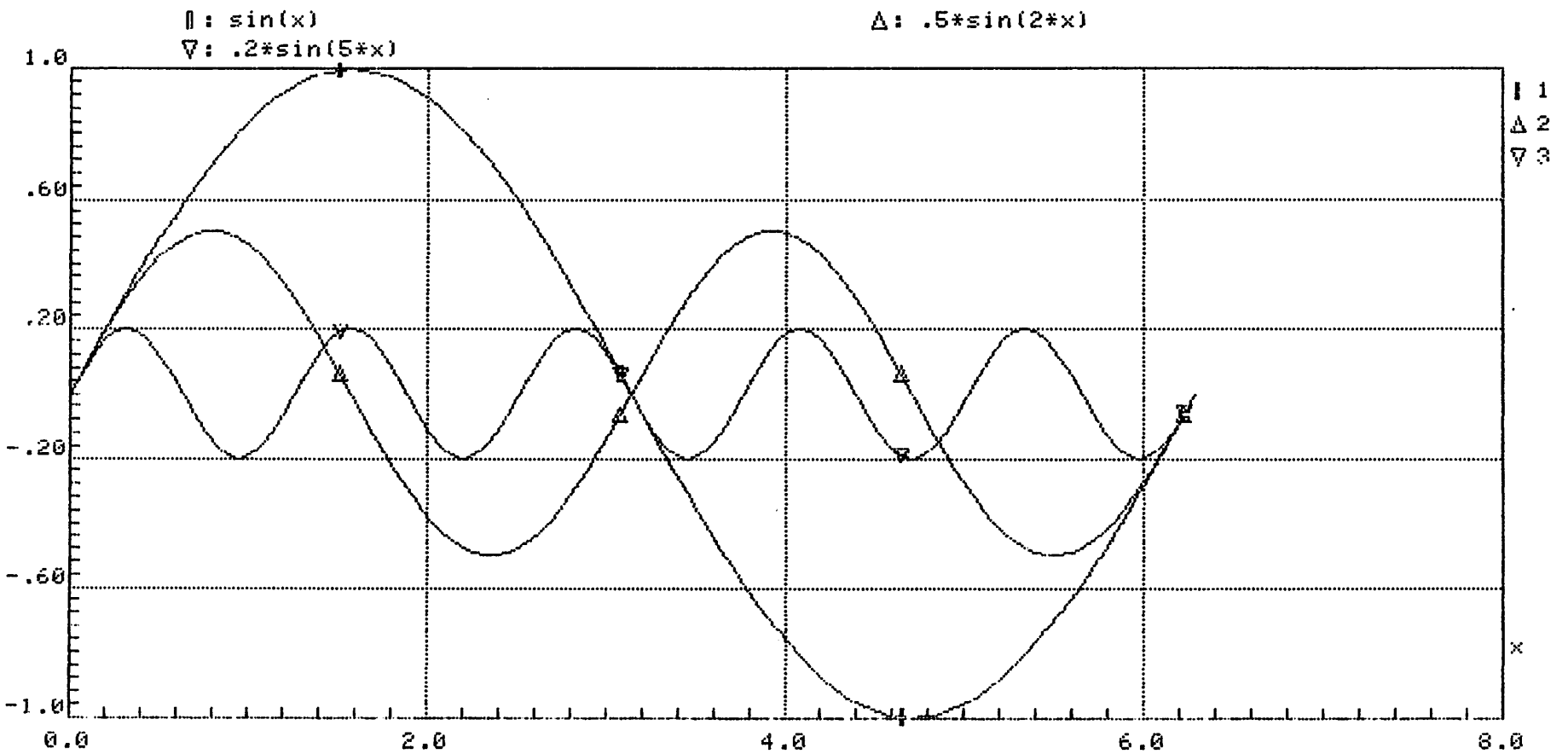


Figure 12.2. "plot sin(x) .5*sin(2*x) .2*sin(5*x) vs x from 0 to TWOPI by TWOPI/100"

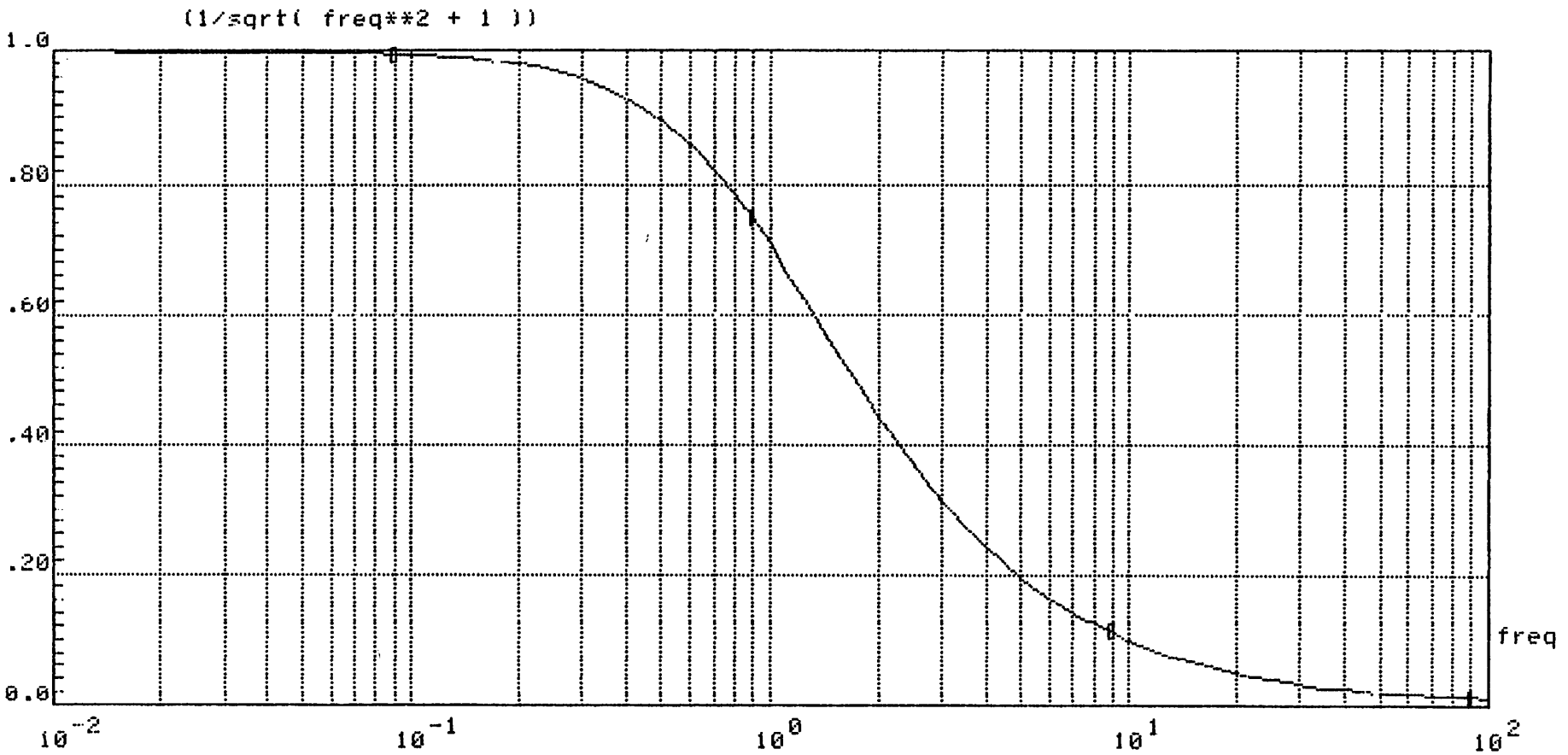


Figure 12.3. "plot (1/sqrt(freq**2 + 1)) vs freq from .01 to 100 dec 20"

where the command actually has to be typed on the same line unless an argument is continued onto the next line by following a "(" with a NEWLINE as in the file listing of the second example below. Here are two examples to try for this command. The first one is typed in directly (here, shifted back to make it fit on the page) and the second one comes from a standard demo file in the DELIGHT system (note the triangular brackets surrounding the filename "<P3milkdrop>"). Beware, the second example requires approximately 1 minute of CPU time to execute (on a VAX 11/780 running Berkeley UNIX).

```
1> plot3d sin(x)*cos(y) vs x from 0 to 9 by .5 vs y from 0 to 6 by 5
    Size:  21 x 13
    1> list <P3milkdrop>
    ------------------- Begin <P3milkdrop> -------------------
    ##============
    ## P3milkdrop - 3-d plot milk drop (finite element partial
    ##============ differential equation solution, ha, ha).

    plot3d exp(exp(-x**2-y**2)*(exp(cos(x**2+y**2)**20)+8*sin(
       x**2+y**2)**20+2*sin(2*(x**2+y**2))**8))*(
       1)  vs x from -1.5 to 1.5 by .05 vs y from -1.5 to 1.5 by .05
    -------------------- End <P3milkdrop> --------------------
    1> use <P3milkdrop>
    Size:  61 x 61
    1>
```

The output plots for these commands are shown in figures 12.4 and 12.5 for the HP2648a black and white terminal.

Other files which you may "use" that contain demonstration plot and plot3d commands include <P3cylin>, <P3wedge>, <Plowpass>, <rosedemo>, and <roseloop>. Special thanks to Aristotle Arapostathis at Berkeley for creating many of the expressions in these files which produce such "good-looking" 3-dimensional plots. WARNING: some of the plot commands in these files may take roughly a minute of CPU time to execute. File <roseloop> is very entertaining (see manual section ROSE(B26)).

Both the plot and plot3d commands can be interrupted by hitting the "break" key once, i.e., by generating a 'soft' interrupt.

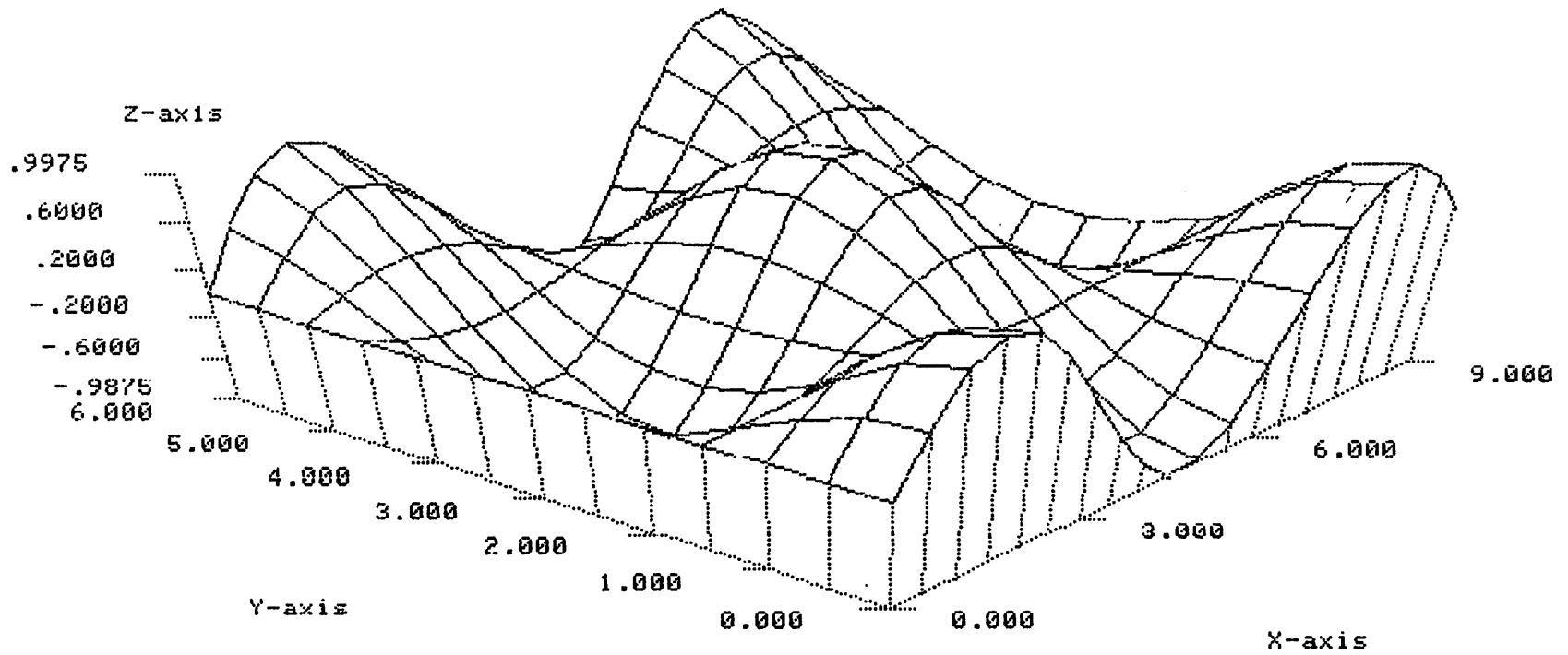


Figure 12.4: "plot3d sin(x)*cos(y) vs x from 0 to 9 by .5 vs y from 0 to 6 by .5"

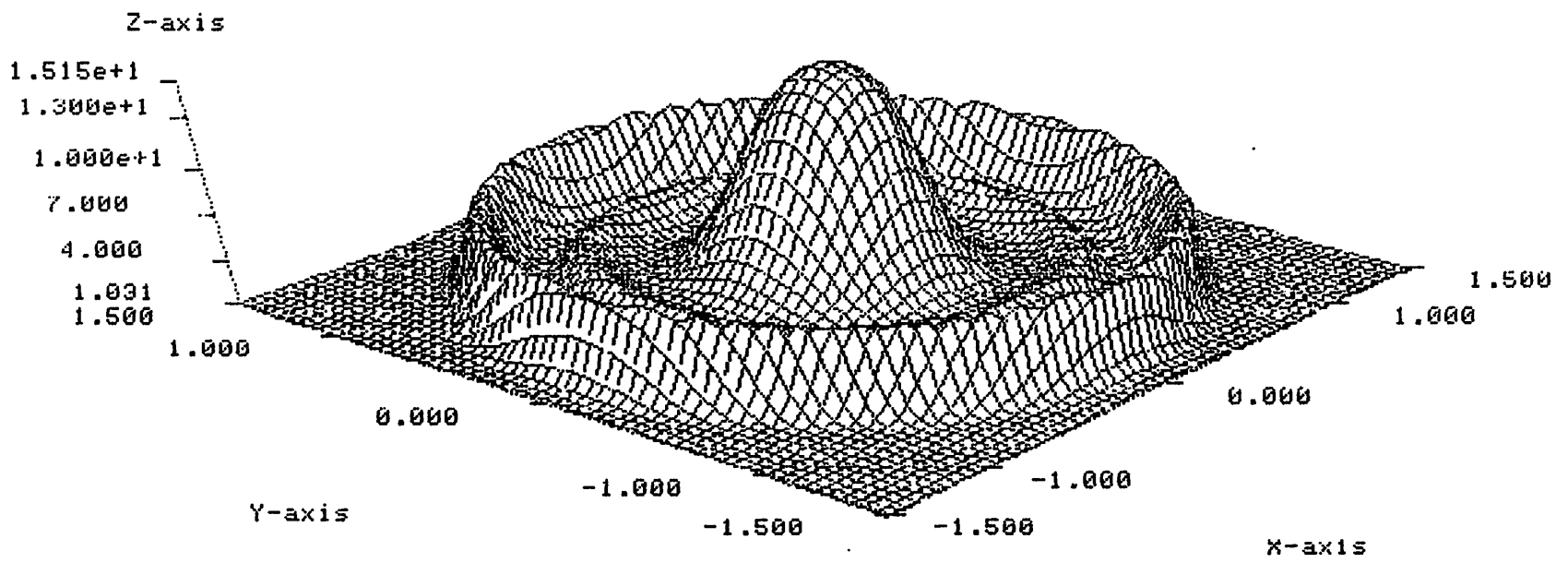


Figure 12.5: Graphical output from "use <P3milkdrop>" command.

13 Matop and Other Matrix Macros.

Presently, there is a huge arsenal of numerical analysis software available to DELIGHT users. Many of these pertain to matrix linear algebra and are from the LINPACK and Harwell subroutine libraries. Since DELIGHT does not yet allow full matrix expressions, a set of macros have been written to perform many matrix-related operations in a simple way. These macros allow your RATTLE procedures to be very readable and self-documenting. The macros also automatically (and hidden from the user) create all the necessary work arrays and inputs for the calls to the built-in FORTRAN library routines.

The following is a partial list of the matrix operations explained more fully in the MATRIX_MACROS(C) manual section:

```
Usage                                          Operation
-------------------------------------------------------------
matop A = B'                                   Transpose of a matrix.
matop A = B + C                                Addition of matrices.
matop Ainv = inv(A)                            Inverse of a matrix.
matop LAMBDA,EV = sym_eigen(A)                 Eigenvalues/eigenvectors
                                               of a symmetric matrix.
clip A = B(3:4,2:6)                            Clipping out a submatrix.
lineq A*x = b                                  Solving linear equations.
linprog z=argmin{c'*x | A*x=b,x>=0}            Linear programming.
det(A)     (in any expression)                 Determinant of a matrix.
||v||      (in any expression)                 L2 norm of a vector.
<<x,y>>    (in any expression)                 Inner product of 2 vectors.
```

Try typing in the following matrix operations. "readmatrix" is a command which prompts for the rows of a one or two-dimensional array.

```
1> array B(2,3)
1> readmatrix B
1: 1 2 3
2: -2 -3 -4
1> printv B
Matrix B(2,3):
  1  2  3
 -2 -3 -4
```

```
1> matop Bt = B'
1> printv Bt
Matrix Bt(3,2):
  1 -2
  2 -3
  3 -4
1> clip Btop = B(1:2,1:2)
1> printv Btop
Matrix Btop(2,2):
  1  2
 -2 -3
1> matop Binv = inv(B)

Mismatched dimensions in inv ... arg(s): Binv B

Interrupt...
2> reset
1> matop Btopinv = inv(Btop)
1> printv Btopinv
Matrix Btopinv(2,2):
 -3 -2
  2  1
1> print det(Btop)
 1.000
1> array v(5)
1> readmatrix v
: 1 1 1 1 1
1> printv v
Column v(5):
  1
  1
  1
  1
  1
1> print  ||v||  sqrt(5)
 2.236  2.236
1> array w(5)
1> readmatrix w
: 2 3 0 0 0
```

```
1> printv w
Column w(5):
  2
  3
  0
  0
  0
1> print <<v,w>>
 5.000
1>
```

Now lets try to get the eigenvalues and eigenvectors of a symmetric matrix, and check if the inverse of the eigenvector matrix times the modal matrix times the eigenvector matrix gives the original matrix (whew!):

```
1> array C(2,2)
1> readmatrix C
1: 2 4
2: 4 3
1> matop L,EV = sym_eigen(C)
1> printv L
Column L(2):
  6.53113
 -1.53113
1> printv EV
Matrix EV(2,2):
   .8618026   .7496782
   .7496782 -.6618026
1> array MODAL(2,2)
1> MODAL(1,1)=L(1)
1> MODAL(2,2)=L(2)
1> printv MODAL
Matrix MODAL(2,2):
  6.53113  0.00000
  0.00000 -1.53113
1> matop EVinv = inv(EV)
1> matop temp = MODAL * EV   ## COMPUTE Cnew=EVinv*MODAL*EV
1> matop Cnew = EVinv * temp
1> printv Cnew
Matrix Cnew(2,2):
  2  4
  4  3
```

```
1> printv C
Matrix C(2,2):
  2  4
  4  3
1> ## --- IT WORKED ... C AND Cnew ARE THE SAME !!! ---
1>
```

A subset of the available matrix operations in DELIGHT has been implemented for complex matrices. See the MATRIX_MACROS(C) section of the DELIGHT reference manual for more information.

14 A Comprehensive Example: Newton Raphson Iteration with Graphics.

In the standard .directory, there are four files which contain procedures which demonstrate a Newton Raphson solution of the 2 by 2 system of nonlinear equations, y=x**2 and x=y**2. Included in these files is a graphics procedure for drawing the 2-dimensional axis and the parabolas, and for plotting the Newton Raphson iteration progress on the same axis.

The four files are:

DEMOnewA	-	"All"-file: including this simply includes all of the other files automatically.
DEMOnewF	-	The function and Jacobian RATTLE procedures.
DEMOnewG	-	The graphics procedure for drawing the axis and the parabolas.
DEMOnewP	-	The actual Newton Raphson main procedure.

To run this demonstration, first of all, you must be on a real graphics terminal; you may try "terminal dumb" but the graphical output is flushed after each iteration and the results are pretty bad. Next, you must RATTLE compile all of the above files by 'including' file DEMOnewA which may be accomplished with "use <DEMOnewA>". The triangular brackets surrounding the filename mean "look for this file in a standard place in the file system".

Let's include <DEMOnewA> and list the files. The array_sequence line in file <DEMOnewF> declares that we wish to store the last 20 values of a 2-long vector called "X". The second component of the previous iteration of X could be accessed, for example, with "X[Iter-1](2)" in any expression. See manual section ARRAY_SEQUENCE(B3) for

more on array sequences.

In the file listings below you will probably see a few RATTLE statements which have not been introduced in this DELIGHT_FOR_BEGINNERS writeup. Hopefully, these features are somewhat self-explanatory but if not, more information can be found by looking up the keywords in section ENTRY_INDEX(5) of the DELIGHT reference manual.

```
1> use <DEMOnewA>
including <gpcolors>    (120sec)
including <DEMOnewF>    (123sec)
including <DEMOnewG>    (127sec)
including <DEMOnewP>    (131sec)
Usage: PGnewton X[0](1) X[0](2) color

1> list <DEMOnewA>
------------------- Begin <DEMOnewA> -------------------
##===========
## DEMOnewA - First file (All) to include for Newton Raphson demo
##===========

include_and_print <gpcolors>   # Use colors instead of
                               # identifiers for "plot".
include_and_print <DEMOnewF>   # Function and Jacobian.
include_and_print <DEMOnewG>   # Graphics procedure.
include_and_print <DEMOnewP>   # Newton-Raphson procedure.
-------------------- End <DEMOnewA> --------------------

1> list <DEMOnewF>
------------------- Begin <DEMOnewF> -------------------
##===========
## DEMOnewF - Function and Jacobian for Newton Raphson example.
##===========

array_sequence X[20](2)     # Keep 20 past values of 2 long vecb

procedure func (x,funval) { # Function value for Newton Raphson.

   array x(2), funval(2)

   funval(1) = x(1)**2 - x(2)
   funval(2) = x(2)**2 - x(1)
   }
```

```
procedure jacobian (x,J)  { # Jacobian matrix for Newton Raphson

   array x(2), J(2,2)

   J(1,1) = 2*x(1)
   J(1,2) = -1
   J(2,1) = -1
   J(2,2) = 2*x(2)
   }
--------------------- End <DEMOnewF> --------------------

1> list <DEMOnewG>
-------------------- Begin <DEMOnewG> --------------------
##===========
## DEMOnewG - Graphics for demo Newton Raphson example.
##=========== Draws axis and 2 parabolas: y=x**2 and x=y**2.

procedure Gnewton_axis (bound)  {

   world -bound -bound bound bound

   color light                # Draw axis.
   vector -bound 0 bound 0
   vector 0 -bound 0 bound

   x = -sqrt(bound)           # Draw y=x**2 parabola.

   move x x**2

   while ( x <= sqrt(bound)+.0001 )  {
      clip_draw x x**2
      x = x + bound/30
      }

   y = -sqrt(bound)           # Draw x=y**2 parabola.
   clip_move y**2 y

   while ( y <= sqrt(bound)+.0001 )  {
      clip_draw y**2 y
      y = y + bound/30
      }

   grend ()                   # End graphics mode.
   }
```

```
---------- End <DEMOnewG> ----------

1> list <DEMOnewP>
---------- Begin <DEMOnewP> ----------
##===========
## DEMOnewP - Example Procedure (with Graphics) to
##=========== demonstrate Newton-Raphson loop in RATTLE.
##
## USAGE:     PGnewton X1initialguess X2initialguess color
## EXAMPLE:   PGnewton  0.2  1  green

define (PGnewton x1 x2 ; ''color='white' , PGnewton_(x1,x2,color)
printf 'Usage: PGnewton X[0](1) X[0](2) color/n'

PGnewton_bound_ = 2.5

procedure PGnewton_ (x1,x2,color_pstr)  {

   array J(2,2), f(2)
   import PGnewton_bound_
   # X is global.

   X[0](1) = x1
   X[0](2) = x2

   Gnewton_axis (PGnewton_bound_)    # Draw the axis and parabola
   color_v  color_pstr               # Set the requested color.

   move     X[0](1) X[0](2)          # Place "o" on initial point
   cursorel X[0](1) X[0](2) -.5 -.5
   text 'o'

   Iter = 0

   repeat  {

      func ( X[Iter] , f )           # Get function value.

      jacobian ( X[Iter] , J )       # Get Jacobian.

      printf 'Iter=%2i  X = %-10r %-10r' Iter X[Iter](1) X[Iter
      printf '  ||f|| = %r/n'  ||f||

      if ( interrupt )  {
```

```
            printf '/nPnewton interrupted at Iter = %i .../n'  Iter
            suspend NO   # Don't print "Interrupt ..."
            }

         restore_position
         clip_draw X[Iter](1) X[Iter](2)

         matop Jinv      = inv (J)
         matop deltaX    = Jinv * f
         matop X[Iter+1] = X[Iter] - deltaX

         Iter = Iter + 1
         }
      until ( ||f|| <= 1.0e-14 )
      }
---------- End <DEMOnewP> ----------

1> terminal                    # Check terminal type.
Terminal is 4027
1> PGnewton .9 0 red
Iter= 0  X =  .9000      0.000       ||f|| =  1.211
Iter= 1  X =  0.000     -.8100       ||f|| =  1.042
Iter= 2  X = -.6561      0.000       ||f|| =  .7847
Iter= 3  X =  0.000     -.4305       ||f|| =  .4687
Iter= 4  X = -.1853      0.000       ||f|| =  .1885
Iter= 5  X =  0.000     -3.434e-2    ||f|| =  3.436e-2
Iter= 6  X = -1.179e-3   0.000       ||f|| =  1.179e-3
Iter= 7  X =  0.000     -1.390e-6    ||f|| =  1.390e-6
Iter= 8  X = -1.932e-12  0.000       ||f|| =  1.932e-12
Iter= 9  X =  0.000     -3.734e-24   ||f|| =  0.000
1> PGnewton .5 .5
Iter= 0  X =  .5000      .5000       ||f|| =  .3536

RUN-TIME ERROR: Singular matrix in inv ... arg(s): J

Interrupt...
2> trace
Interrupted IN procedure
          errprocmess_     No-Trace in file <Merrmess>
Called by invproc_         No-Trace in file <Minvproc>
Called by PGnewton_        line 48  of file <DEMOnewP>
```

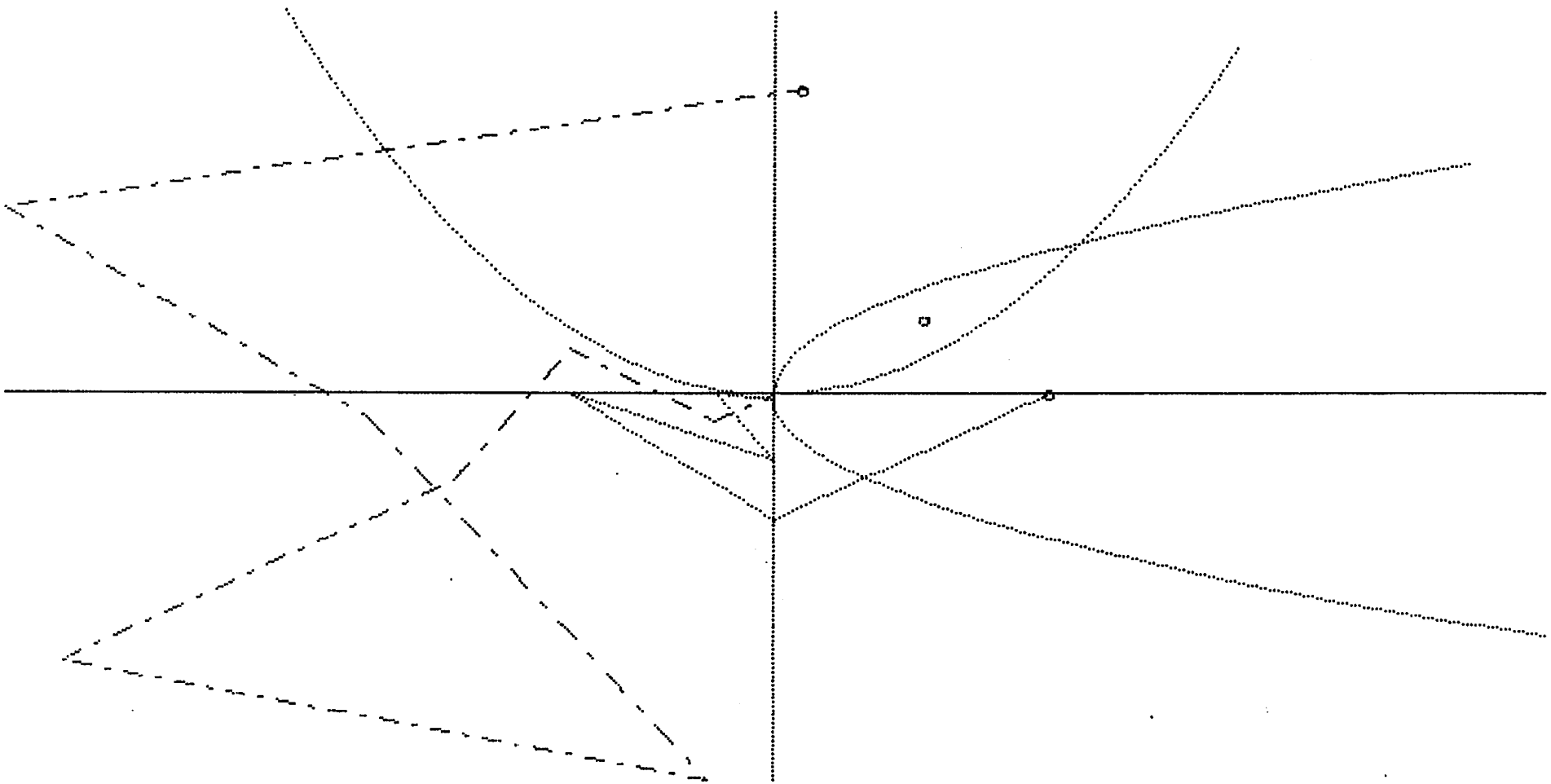
```
2> enter PGnewton_
e> display local arrays

4 items:

   J                (2,2)
   Jinv             (2,2)
   deltaX           (2)
   f                (2)

e> printv J
Matrix J(2,2):
  1 -1
 -1  1
e> print det(J)
 0.000
e> reset
1> PGnewton .1 2 green
Iter= 0  X =  .1000      2.000      ||f|| =  4.378
Iter= 1  X = -2.020e+1  -4.050      ||f|| =  4.137e+2
Iter= 2  X = -1.008e+1  -.7805      ||f|| =  1.030e+2
Iter= 3  X = -5.186      2.932      ||f|| =  2.764e+1
Iter= 4  X = -2.690      1.007      ||f|| =  7.247
Iter= 5  X = -1.317     -.1502      ||f|| =  2.312
Iter= 6  X =  2.388     -8.024      ||f|| =  6.350e+1
Iter= 7  X =  .3490     -4.034      ||f|| =  1.645e+1
Iter= 8  X = -2.305     -1.731      ||f|| =  8.818
Iter= 9  X = -1.029     -.5682      ||f|| =  2.116
Iter=10  X = -.6580      .2950      ||f|| =  .7577
Iter=11  X = -.1928     -.1793      ||f|| =  .3121
Iter=12  X = -2.184e-2  -2.874e-2   ||f|| =  3.698e-2
Iter=13  X = -8.005e-4  -4.419e-4   ||f|| =  9.149e-4
Iter=14  X = -1.947e-7  -6.405e-7   ||f|| =  6.694e-7
Iter=15  X = -4.102e-13 -3.793e-14  ||f|| =  4.120e-13
Iter=16  X = -1.439e-27 -1.683e-25  ||f|| =  0.000
```

Notice that in each of these Newton Raphson runs, once X gets close enough to the 0,0 solution, the norm of f, ||f|| starts decreasing quadratically as expected. The output graphics for these commands is shown in figure 14.1 for the HP2648a black and white terminal.



14.1: Graphical output from "PGnewton" commands in Newton Raphson example.

15 The Optimization Subsystem

15.1 Introduction

As a last step before becoming a full member of the DELIGHT user community, you have to get a taste of the optimization-based design features of DELIGHT. As a matter of fact, optimal design of various types of engineering systems was the motivation for developing the DELIGHT system. By trying the simple example below, you will get a general idea of how to use optimization in DELIGHT. You will find more details in sections OPTIMIZATION_INTRO(E1), OPTIMIZATION_COMMANDS(E2) and OPTIMIZATION_ALGOS(E3) of the DELIGHT Reference Manual.

Unless you want to write your own algorithm - in which case you will need to use the Reference Manual - all you have to do in order to solve an optimization problem is to give a description, in a suitable format, of the problem you want to solve, and then, using the available commands, perform an interactive run of this problem with an algorithm chosen from the optimization library. Subsections 15.2 and 15.3 describe these two operations on a simple example.

15.2 Formulating an Optimization Problem

The DELIGHT formulation of an optimization problem consists of a set of files whose filenames consist of the name of the problem followed by a capital letter, which indicates the function of the file. Type in the following example in which an unconstrained optimization problem is formulated by creating three files: testS (setup), testC (cost function) and testP (initial parameter data):

```
1> edit testS
Unable to open "testS"
:a
Nparam = 2      ## Optimize with respect to 2 design parameters.
.
:wq
"testS" 1 lines
1> edit testP
Unable to open "testP"
:a
X[0](1) = 1.0       ## Initial guess at iteration 0.
X[0](2) = 1.0
.
:wq
"testP" 2 lines
```

```
1> edit testC
Unable to open "testC"
:a
function cost(x)  {
   array x(2)
   return ( x(1)**2 + 2*x(2)**2 )
   }

procedure gradcost (x,g)  {
   array x(2), g(2)
   g(1) = 2*x(1)
   g(2) = 4*x(2)
   }
.
:wq
"testC" 10 lines
```

Note that, in file testC, we have provided a procedure "gradcost" to compute the gradient of the cost; this procedure is not really necessary since DELIGHT, by default, computes gradient partial derivatives by finite differences.

For a more complex optimization problem, the "S" files would also give the number of constraints of various types (e.g., equality, inequality, functional, singular value) as well as the values of problem related parameters. Additional files would have to be provided for the various types of constraints. All of these are explained further in manual section OPTIMIZATION_INTRO(E1).

15.3 Running an Optimization Process

Once your problem has been formulated, you have to choose an algorithm from the algorithm library (see OPTIMIZATION_ALGOS(E3)). You then couple problem and algorithm together using the "solve" command. The "testgrad" command allows you to check if the formulas given for the various gradients are correct by comparing them with values obtained by finite differences. The meanings of the commands "identify", "choices" and "substitute" should be clear from the examples below. More information can be found in manual section OPTIMIZATION_COMMANDS(E2). Try the following:

```
1> solve test using Agradnt
including testS        (8sec)
including testP        (12sec)
Unknowns are in X(2), 20 past values stored.
----------------------------------------
including testC        (14sec)
including <Agradnt>    (15sec)
including <Sarmijo>    (18sec)
PARAMETER:  Alpha = .5 : slope of Armijo line
PARAMETER:  Beta = .5 : trial stepsize along h is Beta**k
including <Dgradnt>    (25sec)
including <Mdirstep>   (28sec)
including <Ostate>     (34sec)
1> identify
PROBLEM:     test
ALGO:        Agradnt
MAIN-LOOP:   Mdirstep
  stepsize :  Sarmijo
  direction : Dgradnt
  output :    Ostate
1> testgrad
WHAT(ParNum)  From grad*   From Perturbation
Gradcost(1)    2.000          2.001
Gradcost(2)    4.000          4.001
1> choices
SUB-BLOCK CHOICE NAMES    PROCEDURE AND ARGUMENT USAGE
   stepsize               stepsize(x,h)
   direction              direction(x,h)
   output                 Type 'output'
1> choices stepsize
stepsize CHOICES        DESCRIPTION
Sarmijo       plain Armijo stepsize rule for unconstrained min.
Sexact        pseudo-exact line-search (improved grid search)
1> choices output
output CHOICES          DESCRIPTION
 Ostate       current iterate, cost and gradient
 Oshort       just the cost and the norm of its gradient
1> substitute Oshort
Substituting Oshort for output ...
1>
```

The "display_parameters" command displays all algorithm related parameters; these parameters are not local to any procedure and their value can be changed before starting

(or in the middle of) execution. Execution is controlled by the "run" command and output, which is now generated by a procedure in algorithm library file <Oshort>, is displayed at each iteration. A soft interrupt can be generated to suspend execution immediately after the output is displayed. Finally, the command "initprob" resets the design parameters to their initial values by re-including the 'P' file containing the initial guess. Try the following:

```
1> display_parameters
PARAMETER SOURCE FILE VALUE  DESCRIPTION
Alpha     <Sarmijo>   .5000  slope of Armijo line
Beta      <Sarmijo>   .5000  trial stepsize along h is Beta**k
1> Alpha = .9
1> run 3
Iter = 0   cost = 3.000       ||gradcost|| = 4.472
Iter = 1   cost = 2.410       ||gradcost|| = 3.971
Iter = 2   cost = 1.945       ||gradcost|| = 3.531
Iter = 3   cost = 1.577       ||gradcost|| = 3.146

Interrupt...
2> Alpha = .6
2> run            ## Runs indefinitely.
                  ## PRESS BREAK KEY AFTER SEVERAL ITERATIONS.
Iter = 4   cost = .6063       ||gradcost|| = 1.823
Iter = 5   cost = 9.548e-2    ||gradcost|| = .6180
Iter = 6   cost = 2.387e-2    ||gradcost|| = .3090
Iter = 7   cost = 5.967e-3    ||gradcost|| = .1545
Iter = 8   cost = 1.492e-3    ||gradcost|| = 7.725e-2
Iter = 9   cost = 3.729e-4    ||gradcost|| = 3.862e-2
Iter = 10  cost = 9.324e-5    ||gradcost|| = 1.931e-2

Interrupt...
2> reset
1> initprob
including testP       (53sec)
1> output
Iter = 0   cost = 3.000       ||gradcost|| = 4.472
1>
```

Epilogue

Well, you now have a key which opens the door to the charms of DELIGHT, DEsign Laboratory with Interaction and Graphics for a Happier Tomorrow. Good luck!

Oh yes, there is one more thing to know ... how to end the enchantment:

```
1> quit
Goodbye Bill, It is 20:45:49, Date: 06/06/82
```

Acknowledgements

For their comments and constructive (or destructive) criticisms, we give special thanks to Dave Riley, Andrew Heunis, Danny Stimler, Polly Siegel, Li Guanguan, and Aristotle Arapostathis. Also, thanks to John Kaye for reviewing the first two sentences and pointing out a missing comma.

This work was supported by the National Science Foundation under grants CEE-81-05790 and ECS-79-13148, by the Air Force Office of Scientific Research (AFOSR) United States Air Force Contract No. F49620-79-C-0178, and by a grant from the Semiconductor Products Division of the Harris Corporation.