HEWLETT PACKARD

Math Pac

Owner's Manual

Series 70

Notice

Hewlett-Packard Company makes no express or implied warranty with regard to the keystroke procedures and program material offered or their merchantability or their fitness for any particular purpose. The keystroke procedures and program material are made available solely on an "as is" basis, and the entire risk as to their quality and performance is with the user. Should the keystroke procedures or program material prove defective, the user (and not Hewlett-Packard Company nor any other party) shall bear the entire cost of all necessary correction and all incidental or consequential damages. Hewlett-Packard Company shall not be liable for any incidental or consequential damages in connection with or arising out of the furnishing, use, or performance of the keystroke procedures or program material.

Math Pac

Owner's Manual

For Use With the HP-75

April 1983

00075-90106

Printed in U.S.A.

Introducing the Math Pac

The Math Pac is a set of powerful tools for solving a wide range of mathematical, scientific, and engineering problems. These tools are provided in the convenient and flexible form of BASIC keywords. Once the Math Pac is plugged into your HP-75 computer, these keywords are instantly available: no program to load, no waiting. (The Math Pac is a ROM-based LEX file, described in appendix B of the *HP-75 Owner's Manual.*) You can use these keywords in any program as often as needed; you avoid the restrictions that would apply to program calls and save the memory that subroutines would require.

The Math Pac adds the following capabilities to your HP-75.

- Advanced real- and complex-valued functions.
- Real and complex matrix operations.
- Solutions to systems of equations.
- Roots of polynomial equations and user-defined functions.
- Numerical integration.
- Finite Fourier transform.

Contents

How To Use This Manual

This manual assumes that you are generally familiar with the operation of your HP-75 computer, especially how to create, edit, store, and run programs. You should also understand the mathematical basis for the operations you will be performing. Because the keywords in the Math Pac cover such a wide range of mathematical subjects, we cannot provide much tutorial information on the mathematical concepts involved.

The keywords in the Math Pac are independent of one another, so you may deal with only the keywords that specifically interest you. Each section in this manual contains information on keywords of a particular mathematical type—real-valued functions, matrix algebra operations, and so on. All keywords described after section 3 (except FNROOT and INTEGRAL) use arrays in their operation. To become familiar with arrays you should read section 13 of the *HP-75 Owner's Manual* and the general information at the beginning of section 4 of this manual.

Within each section you will find a description of each keyword name, function, syntax, and operation in the following format.

Keyword Name

Function That the Keyword Performs

Syntax Legal data types and numeric values for use with this keyword.
Describes the values returned by this keyword and the details of the keyword's operation.

Keyword Name. This is the way the keyword will be referenced elsewhere in the manual. It is usually a mnemonic of the function that the keyword performs. In most cases the name must be embedded in a longer statement that includes arguments, parentheses, and so on; the name by itself usually isn't an acceptable BASIC statement.

Several keywords have names that are identical to names of keywords already present in your HP-75—like DISP, +, and *. The syntax in which such a name is embedded indicates which operation to perform. All operations available to you in the HP-75 itself are still available, unaffected by the presence of the Math Pac.

Syntax. This is a description of the acceptable BASIC statements in which the keyword's name can be embedded. The following conventions are used throughout the manual in describing the syntax of a keyword.

Typographical Item	Interpretation
dot matrix	Words in dot matrix type may be keyed in using either lower or upper case letters, but otherwise must be entered exactly as shown.
italic	Variables in italic type represent numeric expressions; phrases in italic type represent a parameter that is defined elsewhere.
bold	Variables in bold type represent arrays.
[ ]	Square brackets enclose optional items.
stacked items	When two items are placed one above the another, one (and only one) of them may be used.
...	An ellipsis indicates that the optional items within the brackets may be repeated indefinitely.

Legal Data Types and Numeric Values. This information describes the types and ranges of arguments for the keyword that are acceptable to the Math Pac. Use this information to avoid generating errors and to isolate the cause of those that do occur. *This is not a mathematical definition of the domain of the function that the keyword computes.*

Included in each section are a number of examples illustrating the use of the keywords in the section. Almost all of the examples are given as keyboard calculations so that you can immediately see the result of using a particular keyword. The effects of using a keyword in a program will be identical except that in a program you can access only program variables, not calculator variables.

To try an example yourself, type in the commands given in the **Input/Result** column using either upper or lower case, ending each line with a [RTN]. After you complete a command, the display of your HP-75 should look like the display shown in the **Input/Result** column following the command—provided that you have set your line width to 32 by entering width 32 [RTN]. In many cases a single command will produce a sequence of displays, shown as consecutive lines in the display figure. You can control the length of time each display remains visible with the DELAY command described in section 2 of the *HP-75 Owner's Manual.*

Some sections include additional information to help you make effective use of the more sophisticated operations. If you would like still more information, you can refer to the *HP-15C Advanced Functions Handbook.* Although the Math Pac differs from the HP-15C Advanced Programmable Scientific Calculator in its operation and capabilities, much of the information in the *HP-15C Advanced Functions Handbook* applies to the Math Pac. Such information includes techniques to increase the effectiveness of equation-solving algorithms, integration algorithms, matrix operations, system solutions, and accuracy of numerical calculations.

Section 1

Installing and Removing the Module

The Math Pac module can be plugged into any of the three ports on the front edge of the computer.

CAUTIONS

- Be sure to turn off the HP-75 (press [SHIFT] [ATTN]) before installing or removing any module. If there are any pending appointments, type `alarm off` [RTN] in EDIT mode to prevent the arrival of future appointments (which would cause the computer to turn on). If the computer is on or if it turns itself on while a module is being installed or removed, it might reset itself, causing all stored information to be lost.
- Do not place fingers, tools, or other foreign objects into any of the ports. Such actions could result in minor electrical shock hazard and interference with pacemaker devices worn by some persons. Damage to port contacts and internal circuitry could also result.

To insert the Math Pac module, orient it so that the label is right-side up, hold the computer with the keyboard facing up, and push in the module until it snaps into place. During this operation be sure to observe the precautions described above.

To remove the module, use your fingernails to grasp the lip on the bottom of the front edge of the module and pull the module straight out of the port. Install a blank module in the port to protect the contacts inside.

Section 2

Real Scalar Functions

Hyperbolic Functions

SINH — Hyperbolic Sine

```
SINH(X)
```

where X is a numeric expression, $|X| < 1151.98569368$

COSH — Hyperbolic Cosine

```
COSH(X)
```

where X is a numeric expression, $|X| < 1151.98569368$

TANH — Hyperbolic Tangent

```
TANH(X)
```

where X is a numeric expression.

ASINH — Inverse Hyperbolic Sine

```
ASINH(X)
```

where X is a numeric expression.

ACOSH — Inverse Hyperbolic Cosine

```
ACOSH(X)
```

where X is a numeric expression, $X \geqslant 1$.

ATANH — Inverse Hyperbolic Tangent

ATANH(*X*)

where *X* is a numeric expression, $-1 < X < 1$.

Logarithmic Functions

LOG2 — Base 2 Logarithm

LOG2(*X*)

where *X* is a numeric expression, $X > 0$.

$$\text{LOG2}(X) = \log_2(X) = \frac{\ln(X)}{\ln(2)}$$

LOGA — Variable Base Logarithm

LOGA(*X*,*B*)

where *X* is a numeric expression, $X > 0$, and *B* is a numeric expression, $B > 0$ and $B \neq 1$.

$$\text{LOGA}(X,B) = \log_B(X) = \frac{\ln(X)}{\ln(B)}$$

Rounding and Truncating Functions

ROUND — Round

ROUND(*X*,*N*)

where *X*, *N* are numeric expressions.

If *N* is positive, rounds *X* to *N* digits to the right of the decimal point. If *N* is negative, rounds *X* to *N* digits to the left of the decimal point.

$$\text{ROUND}(X,N) = \frac{\text{INT}(X * 10^P + .5)}{10^P}$$

where INT is the standard HP-75 function, and *P* is *N* rounded to the nearest integer.

Note: The rounding convention given above is used *only* in the ROUND keyword. In particular, the ROUND keyword rounds numbers "toward positive infinity" so that if 1.5 is rounded to the nearest integer using ROUND, the result is 2. If −1.5 is rounded to the nearest integer using ROUND, the result is −1. Anywhere else a number needs to be rounded, the Math Pac uses the same convention as the HP-75 itself: positive numbers are rounded "toward positive infinity", and negative numbers are rounded "toward negative infinity". This can only make a difference when the number to be rounded is negative and lies exactly halfway between the numbers to which it could be rounded.

TRUNCATE

Truncate

TRUNCATE(*X*,*N*)

where *X*, *N* are numeric expressions.

If *N* is positive, truncates *X* to *N* digits to the right of the decimal point. If *N* is negative, truncates *X* to *N* digits to the left of the decimal point.

$$\text{TRUNCATE}(X,N) = \frac{\text{IP}(X * 10^P)}{10^P}$$

where IP is the standard HP-75 function, *P* is *N* rounded to the nearest integer.

Factorial/Gamma Function

FACT

Combined Factorial and Gamma Functions

FACT(*X*)

where *X* is a numeric expression not equal to a negative integer,
$-254.1082426465 < X < 253.1190554375$.

If *X* equals a non-negative integer, FACT(*X*) = *X*!

In general, FACT(*X*) = $\Gamma(X + 1)$, defined for $X > -1$ as

$$\Gamma(X + 1) = \int_0^\infty t^X e^{-t} dt$$

and defined for other values of *X* by analytic continuation.

Examples

COSH, SINH, ATANH, ACOSH

Input/Result

```
cosh(0) RTN
```

Hyperbolic cosine of a numeric constant.

```
1
```

```
sinh(1/3+2^3) RTN
```

Hyperbolic sine of a numeric expression.

```
2080.1308825
```

```
x=9 RTN
atanh(1/sqr(x)) RTN
```

Inverse hyperbolic tangent of a numeric expression with a numeric variable.

```
.34657359028
```

```
acosh(cosh(200)) RTN
```

Inverse hyperbolic cosine of a numeric expression.

```
200
```

LOG2, LOGA

Input/Result

```
log2(2^17) RTN
```

Logarithm (base 2) of a numeric expression.

```
17
```

```
loga(81,3) RTN
```

Logarithm (base 3) of 81.

```
4
```

```
x=4 RTN
y=3 RTN
z=x^y RTN
loga(z,x) RTN
```

LOGA of numeric variables.

```
 3
```

ROUND, TRUNCATE

Input/Result

```
x=12345.67 RTN
truncate(x,1) RTN
```

```
 12345.6
```

TRUNCATE "blanks" all the digits rightward of the digit indicated by the second argument.

```
truncate(x,0) RTN
```

```
 12345
```

```
truncate(x,-2.1) RTN
```

```
 12300
```

```
round(x,1) RTN
```

```
 12345.7
```

ROUND rounds the first argument at the digit indicated by the second argument.

```
round(x,-1) RTN
```

```
 12350
```

If the second argument is negative, this indicates a digit to the left of the decimal point.

```
round(-13.25,1) RTN
```

```
 -13.2
```

−13.25 is midway between −13.2 and −13.3; in this case the number is rounded in the positive direction to −13.2.

Section 3

Base Conversions

Binary, Octal, and Hexadecimal Data Types

The operations in this section allow your HP-75 to recognize and manipulate numbers expressed in number systems other than decimal (base 10). These functions conform to the ANSI standards described in appendix B of the *HP-75 Owner's Manual.*

Because the HP-75 assumes that any number stored in a numeric variable or entered from the keyboard is a decimal number, you must enter and store every non-decimal number as a character string. In particular, if you store the number in a variable, the variable's name must end with "$"; if you enter the number from the keyboard, it must be enclosed in quotes.

In the tables below, *S$* will represent a binary, octal, or hexadecimal string or string expression.

- A *binary string* consists entirely of 0's and 1's, and represents a number in the base 2 number system. A *binary string expression* is a string expression whose value is a binary string.
- An *octal string* consists entirely of 0's, 1's, ..., 6's, and 7's, and represents a number in the base 8 number system. An *octal string expression* is a string expression whose value is an octal string.
- A *hexadecimal string* consists of 0's, ..., 9's, A's, ..., F's, and represents a number in the base 16 number system. (Be sure to capitalize the letters A through F, which represent the decimal values 10 through 15.) A *hexadecimal string expression* is a string expression whose value is a hexadecimal string.

Base Conversion Functions

BVAL

Binary, Octal, or Hexadecimal to Decimal Conversion

BVAL(*S$*,*N*)

where *S$* is a binary string expression whose value is not greater than 1110100011010100101001010000111111111111 (binary), and *N* is a numeric expression whose rounded integer value is 2;

or *S$* is an octal string expression whose value is not greater than 16432451207777 (octal), and *N* is a numeric expression whose rounded integer value is 8;

or *S$* is a hexadecimal string expression whose value is not greater than E8D4A50FFF (hexadecimal), and *N* is a numeric expression whose rounded integer value is 16.

Converts a string expression *S$* representing a number expressed in base *N* into the equivalent decimal number. The value of the decimal equivalent can't exceed 999,999,999,999 (decimal).

BSTR$

Decimal to Binary, Octal, or Hexadecimal Conversion

BSTR$(*X*,*N*)

where *X* is a numeric expression, $0 \leq X < 999{,}999{,}999{,}999.5$,
and *N* is a numeric expression whose rounded integer value is 2, 8, or 16.

Converts the rounded integer value of *X* (decimal) into the equivalent base *N* string.

Examples

Input/Result

```
bval("1010",2) RTN
```

```
 10
```

The decimal value of 1010 (binary).

```
b$="1111" RTN
bval(b$,2) RTN
```

```
 15
```

The decimal value of the binary string "1111".

```
bval(b$&b$,2) RTN
```

```
 255
```

The decimal value of the binary string "11111111".

```
bstr$(3,2) RTN
```

```
 11
```

The binary representation of 3 (decimal).

```
bstr$(72,8) RTN
```

```
 110
```

The octal representation of 72 (decimal).

```
bstr$(bval("AF1C8",16),2) RTN
```

```
 10101111000111001000
```

The binary representation of AF1C8 (hexadecimal).

```
bstr$(bval("14772",8)+bval
("570",8),8) RTN
```

```
 15562
```

The octal sum of 14772 (octal) and 570 (octal).

Additional Information

Three considerations determined the range of acceptable parameters for the base conversion keywords.

- The keywords give the exact answer for any integer in the range of acceptable parameters.
- The keywords are inverses of one another, so that composition in either direction is the identity transformation for integers.
- The integers from 0 through 999,999,999,999 form the largest block of consecutive non-negative integers that the HP-75 can display in integer format.

Section 4

Array Input and Output

An array is a variable that is either singly subscripted (a *vector*) or doubly subscripted (a *matrix*), with a range of values for the subscripts (*dimension*) limited only by available memory. Values for array elements are stored sequentially in memory, in *row order*:

- From left to right along each row.
- From the top row to the bottom row.

An array can be one of three data types: REAL, SHORT, or INTEGER. Operations provided by the Math Pac will *not* change the declared type of an array; when the values from a REAL array are assigned to a SHORT or INTEGER array, the values are rounded as they are stored into that array. (Arrays are described in section 13 of the *HP-75 Owner's Manual*; REAL, SHORT, and INTEGER are described in section 5.)

Recall that the upper bounds of an array's subscripts are determined by a dimensioning statement, and that the lower bound of all subscripts in a program is determined by an OPTION BASE statement:

- OPTION BASE 0 sets the lower bound to zero.
- OPTION BASE 1 sets the lower bound to one.

For calculator variables, OPTION BASE 0 is *always* in effect. However, a program variable dimensioned under OPTION BASE 1 will continue to have 1 for the lower bound of its subscripts when the program is interrupted, until the program is deallocated. Note that DIM, REAL, SHORT, and INTEGER statements executed in calculator mode will have *no effect* on program variables *even if the program is interrupted and the program variables are accessible*. This is also true for the explicit and implicit redimensioning implemented by the Math Pac.

Many array operations in the Math Pac are of the form

MAT*result array*=*operation*(*operand array(s)*)

where the operand arrays are the arguments of the operation and the result array is the array in which the results of the operation will be stored. The operation changes *only* the result array.

It is wise to dimension every array before it appears in a Math Pac statement. (Refer to section 13 in the *HP-75 Owner's Manual* for information on default dimensioning of arrays.) If an operation redimensions an array, the array must have been given at least as many elements in its original dimensioning statement as there will be in the redimensioned array, but the numbers of rows and columns need not be individually greater.

The keywords in this section can help you to:

- Change the size of an array.
- Fill an array with values.
- Display values already in an array.

Redimensioning an Array (REDIM)

The keyword REDIM allows you to rearrange an array *without* destroying the information in the array. The values are reassigned according to the new dimensions, and any extra values are inaccessible and unaffected by operations on the array until you again redimension the array. (Some statements can redimension an array before performing an operation. In these cases, the extra values will become accessible when the array is redimensioned and then will be acted on by the operation.)

For example, if you redimension the 2 × 3 array shown below to a 2 × 2 array, you can no longer access the elements 5 and 6.

$$\text{If } \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \text{ is redimensioned to } \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \quad 5 \text{ and } 6 \text{ become inaccessible.}$$

In some cases a DIM *array* statement should be followed by a REDIM *array* statement using the same parameters. This is necessary only if the array is later redimensioned, either explicitly or implicitly, and only if the program segment that dimensions and redimensions the array will be executed *more than once* under program control. Because the DIM statement is executed the first time that the program segment is executed but is skipped each subsequent time, the array won't be reset to its initial size by the DIM statement. The REDIM statement following the DIM statement and using the same parameters is redundant the first time but properly resets the array to its initial size each subsequent time. This rule also applies to REAL, SHORT, and INTEGER statements that dimension an array.

REDIM

Redimensioning

REDIM **A**(*i*) [, **B**(*j*)] ... , **C**(*k*, *l*) [, **D**(*m*, *n*)] ...

where **A**, **B** are vectors, and **C**, **D** are matrices, and *i*, *j*, *k*, *l*, *m*, *n* are numeric expressions.

Redimensions arrays and reassigns values in row order. A redimensioning subscript can be a numeric expression; its rounded integer value becomes the upper bound of the corresponding subscript.

The total number of elements in the redimensioned array can't exceed the total number of elements the array was given in a dimension statement.

Assignments

=

Simple Assignment

MAT **A**=**B**

where **A**, **B** are both vectors, or **A**, **B** are both matrices.

Redimensions **A** to be the same size as **B** and assigns to **A** the corresponding values from **B**. The total number of elements of **A** must be at least as large as the total number of elements of **B**.

=( )

Numeric Expression Assignment

MAT **A**=(*X*)

where **A** is an array and *X* is a numeric expression.

Assigns the value of *X* to every element of **A**.

ZER

Zero Array

MAT **A**=ZER[(*redimensioning subscript(s)*)]

where **A** is an array.

Assigns a value of zero to every element of **A**. If redimensioning subscripts are present, redimensions **A** just as REDIM would.

CON

Constant Array

MAT **A**=CON[(*redimensioning subscript(s)*)]

where **A** is an array.

Assigns a value of one to every element of **A**. If redimensioning subscripts are present, redimensions **A** just as REDIM would.

IDN

Identity Matrix

```
MAT A=IDN
MAT B=IDN(X,Y)
```

where **A** is a square matrix;
or **B** is a matrix, and *X*, *Y* are numeric expressions with the same rounded integer value.

For a square matrix **A**, assigns a value of one to every element on the diagonal of **A** and assigns a value of zero to every other element.

For a matrix **B**, redimensions **B** to a square matrix with the upper bound of each subscript equal to the rounded integer value of *X* and *Y*; then assigns a value of one to every element on the diagonal of **B** and assigns a value of zero to every other element.

Array Input

READ

Assign Values from Data Statements

```
MAT READ A[,B]...
```

where **A**, **B** are arrays.

Assigns values to the specified array(s) by reading from one or more `DATA` statements in the same program as the `MAT READ` statement. Operation is similar to the `READ` keyword in the HP-75. For each array, elements are assigned values in row order; if there is more than one array, they are filled in the order specified.

This keyword can be used only in a program.

INPUT

Assign Values from Keyboard Input

MAT INPUT **A**[,**B**]...

where **A**, **B** are arrays.

Assigns values to the specified array(s) by prompting with the name of an array element and then accepting a number from the keyboard as the value of that element. For each array, prompts are given for the elements in row order; if there is more than one array, they are handled in the order specified.

When the name of an array element is displayed, enter its value by typing in the number and then pressing RTN. You can enter values for several consecutive elements by separating the values with commas. When an array is filled, the remaining values are automatically entered into the next array. After you press RTN the computer will display the name of the next element to be assigned a value.

All values entered must be numbers; you cannot enter numeric variables or expressions.

Array Output

DISP

Display in Standard Format

MAT DISP **A** [, ; **B**]...[, ;]

where **A**, **B** are arrays.

Displays the values of the elements of the specified arrays. The values are displayed in row order. Each row begins on a new line; a blank line is displayed between the last row of an array and the first row of the next array.

The choice of terminator—comma or semicolon—determines the spacing between the elements of an array.

Terminator		Spacing Between Elements
;	Close:	Elements are separated by two spaces. A minus sign, if present, occupies one of the two spaces.
,	Wide:	Elements are placed in 21-column fields.

If the last array specified doesn't have a terminator, the array will be displayed with wide spacing between elements.

PRINT

Print in Standard Format

MAT PRINT **A** [, **B** ;] ... [, ;]

where **A**, **B** are arrays.

Prints the values of the elements of the specified arrays. Operation is identical to MAT DISP except that the output is sent to the PRINTER IS device. (If no PRINTER IS device is present, the output is sent to the DISPLAY IS device.)

DISP USING

Display Using Custom Format

MAT DISP USING *format string* / *statement number* ; **A** [, **B** ;] ... [, ;]

where **A**, **B**, ..., are arrays.

Displays the values of the elements of the specified arrays in a format determined by the *format string* or by the specified IMAGE statement. (Refer to section 16 of the *HP-75 Owner's Manual* for a description of DISP USING, format strings, IMAGE statements, and their results.)

The values are displayed in row order. Each row begins on a new line; a blank line is displayed between the last row of an array and the first row of the next array.

The terminators between the arrays—commas or semicolons—serve only to separate the arrays and have no effect on the display format.

PRINT USING

Print Using Custom Format

MAT PRINT USING *format string* / *statement number* ; **A** [, **B** ;] ... [, ;]

where **A**, **B** are arrays.

Prints the values of the elements of the specified arrays in a format determined by the *format string* or by the specified IMAGE statement. Operation is identical to that of MAT DISP USING except that the output is sent to the PRINTER IS device. (If no PRINTER IS device is present, the output is sent to the DISPLAY IS device.)

Examples

IDN, DISP USING

Input/Result

```
clear vars RTN
dim a(2,2) RTN
mat a=idn RTN
mat disp using"x,d.dd";a RTN
```

```
 1.00 0.00 0.00
 0.00 1.00 0.00
 0.00 0.00 1.00
```

The 3 × 3 identity matrix displayed with one digit before the decimal point, two digits after the decimal point, and one space between the values displayed.

INPUT, REDIM, DISP, ZER, CON

Input/Result

```
clear vars RTN
dim b(2,2) RTN
```

Dimensions **B** to be a 3 × 3 array (remember we are in `OPTION BASE 0`).

```
mat input b RTN
```

```
B(0,0)?
```

Prompts for first element's value.

```
1,2,3,4,5,6,7 RTN
```

The values for `B(0,0)`, `B(0,1)`, . . . , `B(2,0)`.

```
B(2,1)?
```

Prompts for next value to be assigned.

```
8,9 RTN
mat disp b; RTN
```

The values for `B(2,1)` and `B(2,2)`.

```
 1  2  3
 4  5  6
 7  8  9
```

The matrix displayed in close formation.

```
redim b(1,1) RTN
mat disp b; RTN
```

Redimensions **B** to a 2 × 2 matrix.

```
 1  2
 3  4
```

The values of the redimensioned matrix.

```
mat b=zer RTN
mat disp b; RTN
```

Sets all the elements of **B** to zero.

```
 0  0
 0  0
```

The zero matrix.

```
redim b(2,2) RTN
mat disp b; RTN
```

Redimensions **B** to its original size.

```
 0  0  0
 0  5  6
 7  8  9
```

The values that were inaccesible are again accessible, unchanged by `mat b=zer`.

```
mat b=con(3,1) RTN
```

Redimensions **B** to be a 4 × 2 array and assigns the value 1 to all elements.

```
mat disp b; RTN
```

```
 1  1
 1  1
 1  1
 1  1
```

The 4 × 2 constant array.

```
redim b(2,2) RTN
mat disp b; RTN
```

The value from the original matrix is unchanged.

```
 1  1  1
 1  1  1
 1  1  9
```

MAT READ, MAT DISP

To try this example, key in the program listed below and RUN the program.

```
10 OPTION BASE 1
20 DIM A(2,3),B(3,1)
30 DATA 1,2,3,4
40 MAT READ A,B
50 MAT DISP A;B;
60 DATA 5,6,7,8,9,10,11
```

Input/Result

RUN

```
 1  2  3
 4  5  6

 7
 8
 9
```

The array **A** is given the first six values in the data statements.

And the array **B** is given the next three values. The remaining values of the DATA statement would be assigned to the next array in a MAT READ statement, if there were one.

Section 5

Matrix Algebra

Arithmetic

The keywords below perform standard arithmetic operations on arrays. Be sure that the dimensions of the operand arrays are compatible with the particular operation.

- For addition and subtraction, the operand arrays must both be vectors or both be matrices, and they must have the same number of rows and the same number of columns. In this case we will say that the arrays are *conformable for addition.*
- For multiplication, the arrays can be matrices with the number of columns of the first operand equal to the number of rows of the second operand. You can also multiply a matrix and a vector as long as the vector is the *second* operand and the number of columns of the matrix is the same as the number of rows of the vector. In either case, we will say that the arrays are *conformable for multiplication.*

The result array is automatically redimensioned (if possible) to be the correct size.

=– Negation

MAT **A**=–**B**

where **A**, **B** are both vectors or both matrices.

Redimensions **A** to be the same size as **B** and assigns to each element of **A** the negative of the corresponding element of **B**.

+ Addition

MAT **A**=**B**+**C**

where **A**, **B**, **C** are all vectors or all matrices, and **B**, **C** are conformable for addition.

Redimensions **A** to be the same size as **B** and **C**, and assigns to each element of **A** the sum of the values of the corresponding elements of **B** and **C**.

− Subtraction

MAT **A**=**B**−**C**

where **A**, **B**, **C** are all vectors or all matrices, and **B**, **C** are conformable for addition.

Redimensions **A** to be the same size as **B** and **C**, and assigns to each element of **A** the difference of the values of the corresponding elements of **B** and **C**.

* Matrix Multiplication

MAT **A**=**B***C

where **B** is a matrix, **A**, **C** are both vectors or both matrices, and **B**, **C** are conformable for multiplication.

Redimensions **A** to have the same number of rows as **B** and the same number of columns as **C**. The values of the elements of **A** are determined by the usual rules of matrix multiplication.

( )* Multiplication by a Scalar

MAT **A**=(*X*)***B**

where **A**, **B** are both vectors or both matrices, and *X* is a numeric expression.

Redimensions **A** to be the same size as **B** and assigns to each element of **A** the product of the value of *X* and the value of the corresponding element of **B**.

Operations

INV Matrix Inverse

MAT **A**=INV(**B**)

where **A** is a matrix and **B** is a square matrix.

Redimensions **A** to be the same size as **B** and assigns to **A** the values of the matrix inverse of **B**.

TRN

Matrix Transpose

```
MAT A=TRN(B)
```

where **A**, **B** are matrices.

Redimensions **A** to be the same size as the matrix transpose of **B**, and assigns to **A** the values of the matrix transpose of **B**.

CROSS

Cross Product

```
MAT A=CROSS(B,C)
```

where **B**, **C** are both vectors having three elements, and **A** is a vector.

Redimensions **A** to have exactly three elements and assigns to **A** the values of the cross product **B** × **C**.

RSUM

Row Sum

```
MAT A=RSUM(B)
```

where **A**, **B** are arrays.

If **A** is a vector, redimensions **A** to have as many elements as there are rows in **B**; if **A** is a matrix, redimensions **A** to have as many rows as **B** and to have exactly one column.

Assigns the sum of the values in each row of **B** to the corresponding element of **A**.

CSUM

Column Sum

```
MAT A=CSUM(B)
```

where **A**, **B** are arrays.

If **A** is a vector, redimensions **A** to have as many elements as there are columns in **B**; if **A** is a matrix, redimensions **A** to have as many columns as **B** and exactly one row.

Assigns the sum of the values in each column of **B** to the corresponding element of **A**.

Examples

( )*, *, INV, CSUM, RSUM

Input/Result

```
clear vars RTN
dim a(2,2),b(2,2),c(3,3) RTN
mat a=con(2,1) RTN
mat disp a; RTN
```

A is now a 3 × 2 constant matrix.

```
 1  1
 1  1
 1  1
```

```
x=4 RTN
mat c=(x^2-1)*a RTN
mat disp c; RTN
```

The scalar product of a numeric expression and an array.

```
 15  15
 15  15
 15  15
```

Note that **C** was redimensioned to be a 3 × 2 matrix.

```
mat input b RTN
```

```
B(0,0)?
```

```
1,2,3,0,1,2,0,0,1 RTN
mat disp b; RTN
```

```
 1  2  3
 0  1  2
 0  0  1
```

The matrix **B**.

```
mat c=b*a RTN
mat disp c; RTN
```

```
 6  6
 3  3
 1  1
```

The matrix product **BA**.

```
mat b=inv(b) RTN
mat disp b; RTN
```

Inverts the matrix **B**.

```
 1 -2  1
 0  1 -2
 0  0  1
```

The matrix inverse of the matrix **B**.

```
mat a=b*c RTN
```

This should "undo" the earlier multiplication by **B**.

```
mat disp a; RTN
```

```
 1  1
 1  1
 1  1
```

```
mat c=csum(a) RTN
mat disp c; RTN
```

```
 3  3
```

Sum of each column of **A**. Note that **C** was redimensioned to be a 1×2 matrix.

```
mat c=rsum(a) RTN
mat disp c; RTN
```

```
 2
 2
 2
```

Sum of each row of **A**. Note that **C** was redimensioned to be a 3×1 matrix.

Additional Information: INV

The INV keyword uses the *LU* decomposition (described in section 7), together with extended precision arithmetic, to produce an accurate inverse. Special attention is paid to matrices that are nearly singular—that is, close to a matrix which is not invertible. Consider the matrix shown below.

$$\begin{bmatrix} 1 & 3 & 0 \\ 0 & 0 & 1 \\ .666666666667 & 2 & 0 \end{bmatrix}$$

Although this matrix is very nearly singular, it can be succesfully inverted using the INV keyword:

Input/Result

```
clear vars RTN
dim a(2,2),b(2,2) RTN
mat input a RTN
```

```
A(0,0)?
```

Prompts for the first element.

```
1,3,0 RTN
```

```
A(1,0)?
```

Prompts for the next element.

```
0,0,1 RTN
```

```
A(2,0)?
```

```
.666666666667,2,0 RTN
```

A now represents the matrix given above.

```
mat b=inv(a) RTN
```

B is now the computed inverse of **A**.

```
mat b=b*a RTN
mat disp b; RTN
```

```
 1  0  0
 0  1  0
 0  0  1
```

The product of the matrix and its computed inverse is the identity.

If the matrix to be inverted is singular, then the *LU* decomposition is changed by an amount, which is usually small in comparison with roundoff error, to yield a nonsingular matrix. It is this nonsingular matrix which is then inverted.

Section 6

Real-Valued Matrix Functions

The keywords in this section are functions that use arrays as arguments and give a real number as a value. Like other HP-75 functions, they may be used alone or in combination with other functions to produce numeric expressions.

Determinants

DET — Determinant

DET(**A**)

where **A** is a square matrix.

Returns the determinant of the matrix **A**.

DETL — Determinant of Last Matrix

DETL

Returns the determinant of the last matrix that was:

- Inverted in a MAT...INV statement.
- Decomposed in a MAT...LUFACT statement (described in section 7).
- Used as the first argument of a MAT...SYS statement (described in section 8).

DETL retains its value (even if your HP-75 is turned off) until another MAT...INV, MAT...LUFACT, or MAT...SYS statement is executed.

Matrix Norms

FNORM — Frobenius (Euclidean) Norm

FNORM(**A**)

where **A** is an array.

Returns the square root of the sum of the squares of all elements of **A**.

RNORM — Infinity Norm (Row Norm)

RNORM(**A**)

where **A** is an array.

Returns the maximum value (over all rows of **A**) of the sums of the absolute values of all elements in a row.

CNORM — One-Norm (Column Norm)

CNORM(**A**)

where **A** is an array.

Returns the maximum value (over all columns of **A**) of the sums of the absolute values of all elements in a column.

SUM — Array Element Sum

SUM(**A**)

where **A** is an array.

Returns the sum of the values of all elements in **A**.

ABSUM

Array Element Absolute Value Sum

```
ABSUM(A)
```

where **A** is an array.

Returns the sum of the absolute values of all elements in **A**.

AMAX

Array Element Maximum

```
AMAX(A)
```

where **A** is an array.

Returns the value of the maximum element in **A**.

AMIN

Array Element Minimum

```
AMIN(A)
```

where **A** is an array.

Returns the value of the minimum element in **A**.

MAXAB

Array Element Maximum Absolute Value

```
MAXAB(A)
```

where **A** is an array.

Returns the value of the largest element (in absolute value) in **A**.

MINAB

Array Element Minimum Absolute Value

```
MINAB(A)
```

where **A** is an array.

Returns the value of the smallest element (in absolute value) in **A**.

Inner Product

DOT

Inner (Dot) Product

DOT(**X**,**Y**)

where **X**, **Y** are vectors with the same number of elements.

Returns **X** · **Y**, the inner product of **X** and **Y**.

Subscript Bounds

UBND

Subscript Upper Bound

UBND(**A**,*N*)

where **A** is an array and *N* is a numeric expression whose rounded integer value is 1 or 2.

Returns the upper bound of the *N*th (first or second) subscript of **A**. If **A** is a vector, UBND(A,2) = −1.

LBND

Subscript Lower Bound

LBND(**A**,*N*)

where **A** is an array and *N* is a numeric expression whose rounded integer value is 1 or 2.

Returns the option base in effect when **A** was dimensioned. If **A** is a vector, LBND(A,2) = −1.

Examples

DET, DETL, RNORM, UBND

Input/Result

```
clear vars RTN
dim a(3,3) RTN
```

```
mat input a
```

```
A(0,0)?
```

```
1,0,0,0 RTN
1,2,0,0 RTN
1,1,3,0 RTN
1,1,1,4 RTN
det(a) RTN
```

Entering the elements of a matrix one row at a time makes the data entry easier and more accurate.

```
24
```

The determinant of **A**.

```
mat a=inv(a) RTN
detl RTN
```

Computes the inverse of **A**.

```
24
```

Displays the determinant of the last matrix inverted in a MAT...INV, MAT...SYS, or MAT...LUFACT statement. In this case, it is the determinant of the matrix **A**.

```
mat a=inv(a) RTN
mat disp a; RTN
```

```
 1  0  0  0
 1  2  0  0
 1  1  3  0
 1  1  1  4
```

The original matrix **A**.

```
rnorm(a) RTN
```

```
7
```

The sum of the absolute values of the elements in the rows are (in order) 1, 3, 5, 7. The maximum of these is 7.

```
redim a(2,4) RTN
ubnd(a,1) RTN
```

```
 2
```

After redimensioning, the upper bound of the first subscript of **A** is 2 ...

```
ubnd(a,2) RTN
```

```
 4
```

And the upper bound of the second subscript is 4.

ABSUM, AMIN, DOT

Input/Result

```
clear vars RTN
dim x(4),y(4),a(2,1) RTN
mat input x,y,a RTN
```

```
X(0)?
```

The HP-75 prompts you for the first element to be assigned ...

```
1,2,3,0,0 RTN
```

```
Y(0)?
```

And when the first array is full, it begins filling the next matrix in the `MAT INPUT` statement.

```
0,0,-3,-2,-1 RTN
```

```
A(0,0)?
```

```
3,2 RTN
```

```
A(1,0)?
```

```
1,-1 RTN
```

```
A(2,0)?
```

```
-2,-3 RTN
mat disp x;y;a; RTN
```

```
 1
 2
 3
 0
 0

 0
 0
-3
-2
-1

 3   2
 1  -1
-2  -3
```

The vector **X**.

The vector **Y**.

The matrix **A**.

```
absum(x),absum(a),absum(y) RTN
```

```
 6                    12
 6
```

$6 = |1| + |2| + |3| + |0| + |0|.$
$12 = |3| + |2| + |1| + |-1| + |-2| + |-3|.$
$6 = |0| + |0| + |-3| + |-2| + |-1|.$

```
amin(a),amin(x),amin(y) RTN
```

```
-3                    0
-3
```

$-3 = \min\{3, 2, 1, -1, -2, -3\}.$
$0 = \min\{1, 2, 3, 0, 0\}.$
$-3 = \min\{0, 0, -3, -2, -1\}.$

```
dot(x,y) RTN
```

```
-9
```

The inner product of **X** and **Y**.

Additional Information: DET, DETL

The DETL keyword is most useful in direct conjunction with the MAT...INV, MAT...LUFACT, and MAT...SYS statements. In each of these, the result of the operation is less reliable when the matrix argument is very nearly singular. This condition can be detected with DETL. If DETL gives a result very close to zero, then the matrix argument in the corresponding operation is very nearly singular and the result should be interpreted accordingly.

The ⌹ET keyword computes the determinant of a matrix by first decomposing the matrix into its *LU* form. (The next section in this manual describes *LU* decomposition.) If the matrix is singular—that is, its determinant is equal to zero—it may not have an *LU* decomposition. This may cause underflow or overflow warnings to be generated, but it *will not* affect the result of the ⌹ET function.

Section 7

LU Decompositon

A number of operations in the Math Pac, including DET, SYS, and INV, use the *LU* decomposition of a matrix as an intermediary step. The keyword below gives you access to this powerful operation for your own use.

LUFACT

***LU* Decomposition**

MAT **A**=LUFACT(**B**)

where **A** is a matrix and **B** is a square matrix.

Redimensions **A** to be the same size as **B** and assigns to **A** the values of the *LU* decomposition of **B**:

- The elements in **A** above the diagonal are assigned the value of the corresponding elements in **U**.
- The elements in **A** on or below the diagonal are assigned the value of the corresponding elements in **L**.

Example

Input/Result

```
clear vars RTN
dim a(2,2) RTN
mat input a RTN
```

```
A(0,0)?
```

```
1,1,1 RTN
1,0,1 RTN
2,1,2 RTN
mat a=lufact(a) RTN
mat disp a; RTN
```

```
2    .5   1
1   -.5   0
1    .5   0
```

The **L** part of the *LU* decomposition of **A** is

$$\begin{bmatrix} 2 & & \\ 1 & -.5 & \\ 1 & .5 & 0 \end{bmatrix},$$

the **U** part of the *LU* decomposition of **A** is

$$\begin{bmatrix} & .5 & 1 \\ & & 0 \\ & & \end{bmatrix},$$

so that

$$\mathbf{L} = \begin{bmatrix} 2 & 0 & 0 \\ 1 & -.5 & 0 \\ 1 & .5 & 0 \end{bmatrix} \text{ and } \mathbf{U} = \begin{bmatrix} 1 & .5 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Additional Information

The Math Pac *LU* decomposition factors a square matrix **A** into the matrix product **LU**, where

- **L** is a lower-triangular matrix—it has values of zero for all elements above the diagonal.
- **U** is a upper-triangular matrix—it has values of zero for all elements below the diagonal—with values of one for all elements on the diagonal.

For example,

$$\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 3 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 3 & -.5 \end{bmatrix} \begin{bmatrix} 1 & .5 \\ 0 & 1 \end{bmatrix} = \mathbf{LU}.$$

Some matrices can't be factored into **LU** form. For example,

$$\mathbf{A} = \begin{bmatrix} 0 & 1 \\ 1 & 2 \end{bmatrix} \neq \mathbf{LU}$$

for *any* pair of lower- and upper-triangular matrices **L** and **U**. However, if rows are interchanged in the matrix to be decomposed, then *any non-singular* matrix can be so decomposed. Row interchanges in the matrix **A** can be represented by the matrix product **PA** for some permutation matrix **P**. Allowing for

row interchanges, the *LU* decomposition can be represented by the equation **PA** = **LU**. So, for the above example

$$\mathbf{PA} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} 0 & 1 \\ 1 & 2 \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix} = \mathbf{LU}.$$

Row interchanges can also reduce rounding errors that can occur during the calculation of the decomposition.

The Math Pac uses the Crout method with partial pivoting and extended precision arithmetic to construct the *LU* decomposition. The *LU* decomposition is returned in the form

$$\begin{bmatrix} & \mathbf{U} \\ \mathbf{L} & \end{bmatrix}.$$

For example, if the result of the `MAT A=LUFACT(B)` statement is

$$\mathbf{A} = \begin{bmatrix} 2 & 3 & 4 \\ 5 & 6 & 7 \\ 8 & 9 & 2 \end{bmatrix}, \text{ then } \mathbf{L} = \begin{bmatrix} 2 & 0 & 0 \\ 5 & 6 & 0 \\ 8 & 9 & 2 \end{bmatrix} \text{ and } \mathbf{U} = \begin{bmatrix} 1 & 3 & 4 \\ 0 & 1 & 7 \\ 0 & 0 & 1 \end{bmatrix}$$

and **PB** = **LU** for some row interchange matrix **P**.

It is not necessary to store the diagonal elements of **U** in the result matrix since the value of each of these is equal to one. The row interchanges are also recoverable in many cases because, aside from row interchanges, the first column of **L** is the same as the first column of the matrix being decomposed.

For example, if $\mathbf{B} = \begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix}$ and `MAT A=LUFACT(B)` is executed, then $\mathbf{A} = \begin{bmatrix} 2 & .5 \\ 1 & -.5 \end{bmatrix}$.

The fact that the first column of **A** is reversed from the first column of **B** indicates that the rows have been interchanged, so that

$$\mathbf{PB} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 1 & -.5 \end{bmatrix}\begin{bmatrix} 1 & .5 \\ 0 & 1 \end{bmatrix} = \mathbf{LU}.$$

In many cases, the *LU* decomposition will be correct *even if the matrix is singular*. This can be checked by remultiplying the **L** and **U** matrices and comparing the result to the original matrix. This feature gives you the ability to find the *LU* decomposition of matrices that are not square.

For example, to find the *LU* decomposition of the 3 × 4 matrix

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 4 & 9 & 16 \\ 1 & 8 & 27 & 64 \end{bmatrix},$$

we will find the decomposition of

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 4 & 9 & 16 \\ 1 & 8 & 27 & 64 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

instead. From this decomposition, the *LU* decomposition of the original matrix is easily found. The keystroke sequence below illustrates the process.

Input/Result

```
clear vars RTN
dim s(3,3) RTN
mat input s RTN
```

```
S(0,0)?
```

```
1,2,3,4,1,4,9,16,1,8,27,64,0,0,
0,0 RTN
mat s=lufact(s) RTN
mat disp s; RTN
```

```
 1   2   3   4
 1   6   4   10
 1   2  -2   4
 0   0   0   0
```

Therefore

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 6 & 0 & 0 \\ 1 & 2 & -2 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \text{and} \quad \mathbf{U} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 1 & 4 & 10 \\ 0 & 0 & 1 & 4 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Their matrix product is

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 8 & 27 & 64 \\ 1 & 4 & 9 & 16 \\ 0 & 0 & 0 & 0 \end{bmatrix},$$

so that

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 4 & 9 & 16 \\ 1 & 8 & 27 & 64 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 6 & 0 \\ 1 & 2 & -2 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 1 & 4 & 10 \\ 0 & 0 & 1 & 4 \end{bmatrix}$$

This technique works best when the matrix has fewer rows than columns. If your matrix has fewer columns than rows, find the *LU* decomposition of its *transpose* by the above technique, and take the transpose of the result.

Section 8

Solving a System of Equations

The Math Pac provides you with a quick and accurate way to solve a system of linear equations. The first step in using this capability is to translate the system of equations into a triple of arrays: the result array, the coefficient array, and the constant array. The *result array* corresponds to the variables in the equations; the *coefficient array* holds the values of the coefficients of the variables; the *constant array* holds the values of the constants in the equations. For example, if you wanted to solve the system of equations

$$11x + 12y + 13z = 1$$

$$21x + 22y + 23z = 2$$

$$31x + 32y + 33z = 3$$

then the result array would correspond to the array

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix},$$

the coefficient array would be

$$\begin{bmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \\ 31 & 32 & 33 \end{bmatrix},$$

and the constant array would be

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}.$$

If we denote the result array by **X**, the coefficient array by **A**, and the constant array by **B**, then the system of equations can be written in matrix notation as **AX=B**. This is the form assumed by the SYS keyword.

SYS

System Solution

MAT**X**=SYS(**A**,**B**) where **A** is a square matrix, **X**, **B** are both vectors or both matrices, and **A**, **B** are conformable for multiplication.
Redimensions **X** to be the same size as **B** and assigns to **X** the values that satisfy the matrix equation **AX=B**.

Example

To solve the system of equations given in the introduction, namely,

$$11x + 12y + 13z = 1$$

$$21x + 22y + 23z = 2$$

$$31x + 32y + 33z = 3$$

we could use the following keystrokes.

Input/Result

```
clear vars RTN
```

```
dim x(2),b(2),a(2,2) RTN
```

X will represent $\begin{bmatrix} x \\ y \\ z \end{bmatrix}$.

```
mat input b RTN
```

```
B(0)?
```

```
1,2,3 RTN
```

B will be $\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$.

```
mat input a RTN
```

```
A(0,0)?
```

11,12,13,21,22,23,31,32,33 RTN **A** will be $\begin{bmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \\ 31 & 32 & 33 \end{bmatrix}$.

```
mat x=sys(a,b) RTN
mat disp x; RTN
```

```
 .160424051041
-2.08481020818E-2
-3.95759489591E-2
```

$= x$.
$= y$.
$= z$.

Solving the Steady-State Heat Equation

A rectangular plate, with a length-to-width ratio of 6 to 5, has its edges held at a constant temperature of 0.* The plate also has a number of internal heat sources or sinks with the result that these points are held at constant temperatures, perhaps different from 0. Find the steady-state heat distribution throughout the plate.

The Model

The plate will be modeled by a rectangular lattice of points, any number of which can be designated as sources or sinks of heat. The temperature of a lattice point at location (i, j) will be denoted by $T(i, j)$ $(1 \leqslant i \leqslant 5,\ 1 \leqslant j \leqslant 6)$.

* The equations are independent of the temperature scale used, so that this only represents the zero of *this* temperature scale.

The system of equations that models the steady-state heat distribution is derived from the "average of nearest neighbors" technique:

$$T(i, j) = \begin{cases} \text{The given value if } (i, j) \text{ corresponds to a source or a sink.} \\ \dfrac{T(i + 1, j) + T(i - 1, j) + T(i, j + 1) + T(i, j - 1)}{4} \text{ otherwise, with the} \\ \qquad \text{convention that } T(m, n) = 0 \text{ if } (m, n) \text{ represents a location outside the lattice.} \end{cases}$$

For example, if (4, 2) is not the location of a source or sink, then

$$T(4, 2) = \frac{T(5, 2) + T(3, 2) + T(4, 3) + T(4, 1)}{4}.$$

We can redimension the 5×6 array **T** with elements $T(i, j)$ to be a 30×1 array **X** with elements $X(k, 1)$ by the formula

$$X(k, 1) = T(i, j) \text{ for } k = 6(i - 1) + j.$$

The correspondence between indexes can be defined equivalently by

$$i = \text{int}\,\frac{k - 1}{6} + 1 \text{ and } j = \text{mod } (k - 1, 6) + 1.$$

Now the system of equations can be written in the form $\mathbf{X} = \mathbf{CX} + \mathbf{B}$, where **C** is the 30×30 matrix given by

$$c_{kn} = \begin{cases} 0 \quad \text{if } (i, j) \text{ corresponds to a source or sink,} \\ \qquad \text{where } i = \text{int}\,\dfrac{k - 1}{6} + 1 \text{ and } j = \text{mod } (k - 1, 6) + 1. \\ ¼ \quad \text{if } (i', j') \text{ is a nearest neighbor to } (i, j), \\ \qquad \text{where } i' = \text{int}\,\dfrac{n - 1}{6} + 1 \text{ and } j' = \text{mod } (n - 1, 6) + 1. \\ 0 \quad \text{otherwise.} \end{cases}$$

and **B** is the 30×1 matrix given by

$$b_{k1} = \begin{cases} \text{The given value if } k \text{ corresponds to a source or sink.} \\ 0 \quad \text{otherwise.} \end{cases}$$

The system of equations can finally be written as the matrix equation $\mathbf{AX} = \mathbf{B}$ where $\mathbf{A} = (\mathbf{I} - \mathbf{C})$ and **I** is the identity matrix. This is the form required by the `SYS` keyword and the form we will use to solve the equations.

The Program

```
10 OPTION BASE 1
20 SHORT A(30,30)

30 DIM B(5,6)

40 MAT B=ZER @ MAT A=IDN
50 INPUT"Number of sources/sinks?";N
60 FOR L=1 TO N
70 INPUT"I,J,T(I,J)";I,J,B(I,J)

80 NEXT L
90 DISP "Solving; please wait"
100 REDIM B(30,1)

110 FOR K=1 TO 30


120 IF B(K,1)<>0 THEN GOTO 190









130 I=INT((K-1)/6)+1
140 J=MOD(K-1,6)+1
150 IF J<6 THEN A(K,K+1)=-.25

160 IF J>1 THEN A(K,K-1)=-.25
170 IF I<5 THEN A(K,K+6)=-.25
```

Since **A** will consist of 0's, 1's and $-\frac{1}{4}$'s, we can use this data type to save memory.

B is the source/sink array and will be used to store the final results.

Gets the locations and temperatures at the sources and sinks.

Rearranges **B** as a one-column array in preparation for using the system solver.

Begins construction of the matrix **A**. *K* corresponds to that row of **A** currently under construction.

We check whether a location is a source/sink by checking whether the assigned temperature is non-zero.* If it is a source/sink, we go on to the next location, leaving the entire row of **A** unchanged from the corresponding row of the identity matrix. If it isn't a source/sink, we continue the computation of the elements of the *K*th row of **A**.

Computes the (*I*,*J*) position from *K*.

If the nearest neighbor isn't off the edge, assigns $-\frac{1}{4}$ to the corresponding element of **A**.

* If a source is supposed to have zero as a given temperature, *do not* enter zero for its value. Instead, enter a very small but non-zero number; 1E-40 will work.

```
180 IF I>1 THEN A(K,K-6)=-.25
190 NEXT K
200 MAT B=SYS(A,B)
210 REDIM B(5,6)
220 MAT DISP USING"x,dd.d";B
```

Solves the equations and stores the results in **B**.

Arranges the results in lattice form.

This will display the results in a compact lattice form. If your results have more than two digits to the left of the decimal point, this display will be inadequate.

Using the Program

Suppose there is one source located at position (2, 3) that maintains a 10-degree temperature difference. We would run the program and when prompted with `number of sources/sinks?` we would respond with `1`. When prompted with `I,J,T(I,J)?` we would respond with `2,3,10`. The program would then display

```
Solving; please wait
```

```
 1.0  2.2   3.6  2.3  1.2   .5
 1.6  4.2  10.0  4.4  2.1   .9
 1.4  3.0   4.6  3.2  1.8   .8
  .9  1.7   2.3  1.9  1.2   .6
  .4   .8   1.0   .9   .6   .3
```

This is the lattice of temperatures in the plate under these conditions.

Additional Information

The `SYS` keyword solves the matrix equation $\mathbf{AX} = \mathbf{B}$ for $\mathbf{X}$ in several stages. First, the *LU* decomposition of $\mathbf{A}$ is found to give $\mathbf{PA} = \mathbf{LU}$. (*LU* decomposition is described in section 7.)

Using $\mathbf{PA} = \mathbf{LU}$, the equivalent problem is to solve $\mathbf{LUX} = \mathbf{PB}$ for $\mathbf{X}$. This is done by solving $\mathbf{LY} = \mathbf{PB}$ for $\mathbf{Y}$ (*forward substitution*) and then solving $\mathbf{UX} = \mathbf{Y}$ for $\mathbf{X}$ (*backward substitution*). This value for $\mathbf{X}$ is used as a first approximation to the desired solution in a process of iterative refinement, which produces the final result.

In many cases, the Math Pac will arrive at a correct solution even if the coefficient array is singular (so that the formula $\mathbf{X} = \mathbf{A}^{-1}\mathbf{B}$ is invalid). This feature allows you to use `SYS` to solve under- and over-determined systems of equations.

For an under-determined system (more variables than equations), the coefficient array will have fewer rows than columns. To find a solution using SYS:

- Append enough rows of zeros to the bottom of your coefficient array to make it square.
- Append corresponding rows of zeros to the constant array.

You can now use these arrays with the SYS keyword to find a solution to the original system.

For an over-determined system (more equations than variables), the coefficient array will have fewer columns than rows. To find a solution using SYS:

- Append enough columns of zeros on the right of your coefficient array to make it square.
- Be sure that your result array is dimensioned to have at least as many rows as the new coefficient array has columns.
- Add enough zeros on the bottom of your constant array to ensure conformability.

You can now use these arrays with the SYS keyword to find a solution to the original system. Only those elements in the result array that correspond to your original variables will be meaningful.

For both under- and over-determined systems the coefficient array is singular, so you should check the results returned by SYS to see if they satisfy the original equation.

The SYS keyword can also be used for inverting a square matrix **A**. MAT X=SYS(A,B) will return the inverse of **A** if **X**, **A**, **B** are all dimensioned to exactly the same size and if **B** is chosen to be the identity matrix. This technique is more accurate and generally faster than MAT X=INV(A), but it requires more memory for its operation. (Refer to appendix B for information about memory requirements).

Section 9

Complex Variables

The keywords in this section enable you to perform algebraic operations on complex numbers in a simple and efficient way. The HP-75 Math Pac can interpret any array with exactly two elements as a complex number. In particular, a 1×2 matrix, a 2×1 matrix, and a two-element vector can all represent complex numbers. If an array **Z** represents a complex number z, then the value of the first element of **Z** is the real part of z and the value of the second element of **Z** is the imaginary part of z. For example, the arrays

$$[1\ 2] \text{ and } \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

both represent the complex number $1 + 2(-1)^{1/2} = 1 + 2i$. Throughout this section we will refer to an array with exactly two elements as a *complex scalar*.

The operand arrays for these keywords must be complex scalars. However, you need not ensure that the result array is a complex scalar. If it is not, it will be automatically redimensioned to have exactly two elements. The result array must, therefore, have been given at least two elements in its original dimensioning statement. If the result array is doubly subscripted, it will be redimensioned to be a 1×2 matrix. If it is singly subscripted, it will be redimensioned to have exactly two elements. This feature allows two-element arrays to be used interchangeably for complex operations.

Polar/Rectangular Conversions

Since the Math Pac assumes rectangular (Re + iIm) form for all complex numbers, two operations are provided to change a pair of numbers representing the magnitude (R) and angle (θ) of a complex number into the real and imaginary parts of that complex number, and vice versa.

CPTOR

Polar to Rectangular Conversion

```
MAT Z=CPTOR(A)
```

where **A** is an array with two elements and **Z** is an array.

Redimensions **Z** to be a complex scalar; then assigns to the first element of **Z** the real part, and to the second element of **Z** the imaginary part, of the complex number $R \exp(i\theta)$, where R is the value of the first element of **A** and θ is the value of the second element of **A**.

θ will be interpreted as degrees or radians, according to the `OPTION ANGLE` in effect.

CRTOP

Rectangular to Polar Conversion

```
MAT A=CRTOP(Z)
```

where **Z** is a complex scalar and **A** is an array.

Redimensions **A** to be a complex scalar; then assigns to the first element of **A** the magnitude, and to the second element of **A** the angle, of the complex number $x + iy$, where x is the value of the first element of **Z** and y is the value of the second element of **Z**.

The angle will be given in degrees ($-180 < \theta \leq 180$) or in radians ($-\pi < \theta \leq \pi$) according to the `OPTION ANGLE` in effect.

Complex Arithmetic Operations

CONJ

Complex Conjugation

```
MAT Z=CONJ(W)
```

where **W** is a complex scalar and **Z** is an array.

Redimensions **Z** to be a complex scalar and assigns to the first element of **Z** the value of the first element of **W** and assigns to the second element of **Z** the negative of the value of the second element of **W**.

CADD

Complex Addition

```
MAT Z=CADD(W,U)
```

where **W**, **U** are complex scalars and **Z** is an array.

Redimensions **Z** to be a complex scalar and assigns **Z** the values corresponding to the complex number **W** + **U**.

CSUB

Complex Subtraction

```
MAT Z=CSUB(W,U)
```

where **W**, **U** are complex scalars and **Z** is an array.

Redimensions **Z** to be a complex scalar and assigns **Z** the values corresponding to the complex number **W** − **U**.

CMULT

Complex Multiplication

```
MAT Z=CMULT(W,U)
```

where **W**, **U** are complex scalars and **Z** is an array.

Redimensions **Z** to be a complex scalar and assigns to **Z** the values corresponding to the complex number **W*U**.

CDIV

Complex Division

```
MAT Z=CDIV(W,U)
```

where **W**, **U** are complex scalars, **U** ≠ (0, 0), and **Z** is an array.

Redimensions **Z** to be a complex scalar and assigns to **Z** the values corresponding to the complex number **W**/**U**.

CRECP

Complex Reciprocal

`MAT Z=CRECP(W)` where **W** is a complex scalar, **W** ≠ (0, 0), and **Z** is an array.
Redimensions **Z** to be a complex scalar and assigns to **Z** the values corresponding to the complex number 1/**W**.

Examples

CPTOR, CRTOP

Input/Result

```
clear vars RTN
option angle degrees RTN
```

The HP-75 will now use degree measure for the angle in CRTOP and CPTOR conversions.

```
dim w(1),u(1,0),z(2,3) RTN
```

W and **U** are dimensioned as two-element arrays and so are both complex scalars. It will be possible to redimension **Z** as a complex scalar, since it has more than two elements.

```
mat input w RTN
```

```
W(0)?
```

```
10,90 RTN
```

W will be used to represent a vector with magnitude 10 and angle 90 degrees.

```
mat z=cptor(w) RTN
mat disp z; RTN
```

```
 0   10
```

Z is the complex number $0 + 10i$, which is the rectangular representation of **W**.

```
mat input u RTN
```

```
U(0,0)?
```

```
3,-4 RTN
```

U represents the complex number $3 - 4i$.

```
mat z=crtop(u) RTN
```

```
mat disp z; RTN
```

```
 5 -53.1301023542
```

Z now represents the magnitude (5) and the direction (−53.1301023542 degrees) of the vector **U**.

```
option angle radians RTN
mat z=crtop(u) RTN
mat disp z; RTN
```

```
 5 -.927295218002
```

In this case, the direction is given in radians.

CADD, CMULT, CRECP

Input/Result

```
clear vars RTN
dim z(2,3),w(0,1),u(1) RTN
mat input u RTN
```

```
U(0)?
```

```
1,1 RTN
```

U represents the complex number $1 + i$.

```
mat w=crecp(u) RTN
```

W represents $\frac{1}{1+i}$ ...

```
mat disp w; RTN
```

```
 .5 -.5
```

Which equals $.5 - .5i$.

```
mat z=cadd(w,u) RTN
```

Z represents $\frac{1}{1+i} + (1 + i)$ ...

```
mat disp z; RTN
```

```
 1.5   .5
```

Which equals $1.5 + .5i$.

```
mat z=cmult(u,z) RTN
```

Z represents $(1 + i)\left(\frac{1}{1 + i} + (1 + i)\right)$ ...

```
mat disp z; RTN
```

```
 1   2
```

Which equals $1 + 2i$.

Section 10

Complex Functions

Many useful functions are defined for complex as well as real arguments. The keywords below give you access to a number of these functions. Since the values of these functions are, in general, complex numbers, their syntax is closer to that of array operations than to their real-valued counterparts. Although the result array need not be a complex scalar for these keywords, it must have been given at least two elements in its original dimensioning statement.

These keywords will produce error (or warning) messages if the conditions listed in their descriptions are not satisfied. They will also produce error or warning messages if either the real or the imaginary part of the function's value cannot be represented in the range [−9.99999999999E499, −1E−499], [1E-499, 9.99999999999E499] or 0. The two-dimensional nature of these functions precludes giving more simple bounds for the arguments that will avoid all such error messages. In addition, if either the real or imaginary part of the value for any of the functions CSIN, CCOS, CSINH, CCOSH, or CPOWER is too large to be represented by the computer and so produces a num too large message and returns a value of ±9.99999999999E499, it is quite likely that the other part of the value returned is inaccurate.

Simple Transcendental Functions

All keywords in this section involve trigonometric functions and always take their arguments to be in radian measure, *even if* OPTION ANGLE DEGREES *is in effect.*

CEXP — Complex Exponential

MAT **Z**=CEXP(**W**)

where **W** is a complex scalar and **Z** is an array.

Redimensions **Z** to be a complex scalar and assigns to **Z** the values of the complex exponential of **W**. If **W** represents the complex number $x + iy$, then **Z** will represent the complex number

$$\exp(x + iy) = e^x(\cos y + i \sin y).$$

CSIN

Complex Sine

```
MAT Z=CSIN(W)
```

where **W** is a complex scalar, $|\text{Im}\ (\mathbf{W})| < 2300.28250791$, and **Z** is an array.

Redimensions **Z** to be a complex scalar and assigns to **Z** the values of the complex sine of **W**. If **W** represents the complex number $x + iy$, then **Z** will represent the complex number

$$\sin\ (x + iy) = \sin x\ \cosh y + i\ \cos x\ \sinh y.$$

CCOS

Complex Cosine

```
MAT Z=CCOS(W)
```

where **W** is a complex scalar, $|\text{Im}\ (\mathbf{W})| < 2300.28250791$, and **Z** is an array.

Redimensions **Z** to be a complex scalar and assigns to **Z** the values of the complex cosine of **W**. If **W** represents the complex number $x + iy$, then **Z** will represent the complex number

$$\cos\ (x + iy) = \cos x\ \cosh y - i\ \sin x\ \sinh y.$$

CTAN

Complex Tangent

```
MAT Z=CTAN(W)
```

where **W** is a complex scalar and **Z** is an array.

Redimensions **Z** to be a complex scalar and assigns to **Z** the values of the complex tangent of **W**. If **W** represents the complex number $x + iy$, then **Z** will represent the complex number

$$\tan\ (x + iy) = \frac{\sin\ (x + iy)}{\cos\ (x + iy)} = \frac{\sin x\ \cos x + i\ \sinh y\ \cosh y}{\sinh^2 y + \cos^2 x}$$

CSINH — Complex Hyperbolic Sine

```
MAT Z=CSINH(W)
```

where **W** is a complex scalar, |Re (**W**)| < 2300.28250791, and **Z** is an array.

Redimensions **Z** to be a complex scalar and assigns to **Z** the values of the complex hyperbolic sine of **W**. If **W** represents the complex number $x + iy$, then **Z** will represent the complex number

$$\sinh (x + iy) = (-i) \sin (-y + ix).$$

CCOSH — Complex Hyperbolic Cosine

```
MAT Z=CCOSH(W)
```

where **W** is a complex scalar, |Re (**W**)| < 2300.28250791, and **Z** is an array.

Redimensions **Z** to be a complex scalar and assigns to **Z** the values of the complex hyperbolic cosine of **W**. If **W** represents the complex number $x + iy$, then **Z** will represent the complex number

$$\cosh (x + iy) = \cos (-y + ix).$$

CTANH — Complex Hyperbolic Tangent

```
MAT Z=CTANH(W)
```

where **W** is a complex scalar and **Z** is an array.

Redimensions **Z** to be a complex scalar and assigns to **Z** the values of the complex hyperbolic tangent of **W**. If **W** represents the complex number $x + iy$, then **Z** will represent the complex number

$$\tanh (x + iy) = (-i) \tan (-y + ix).$$

Inverse Functions

The keywords in this section give you the ability to compute the principal values of a number of complex inverse functions. A description of the principal branches and values chosen for these inverse functions is given in "Additional Information" at the end of this section.

Although the result array need not be a complex scalar for these keywords, it must have been given at least two elements in its original dimensioning statement.

CSQR — Complex Square Root

MAT **Z**=CSQR(**W**)

where **W** is a complex scalar and **Z** is an array.

Redimensions **Z** to be a complex scalar and assigns to **Z** the complex principal value of the square root of **W**.

CPOWER — Complex Power

MAT **V**=CPOWER(**Z**,**W**)

where **Z**, **W** are complex scalars, **Z** $\neq$ (0, 0) if Re (**W**) $\leq$ 0, and **V** is an array.

Redimensions **V** to be a complex scalar and assigns to **V** the complex principal value of $\mathbf{Z}^{\mathbf{W}}$. If **Z** and **W** represent complex numbers z and w respectively, then **V** represents the complex number exp (w ln z).

CLOG — Complex Logarithm

MAT **Z**=CLOG(**W**)

where **W** is a complex scalar, **W** $\neq$ (0, 0), and **Z** is an array.

Redimensions **Z** to be a complex scalar and assigns to **Z** the complex principal value of the logarithm of **W**. If **W** represents the complex number

$$R\ (\cos\theta + i \sin\theta)$$

where $-\pi < \theta \leq \pi$ (radian measure), then **Z** represents the complex number

$$\ln R + i\,\theta.$$

CASIN — Complex Inverse Sine

MAT **Z**=CASIN(**W**)

where **W** is a complex scalar and **Z** is an array.

Redimensions **Z** to be a complex scalar and assigns to **Z** the complex principal value of the inverse sine of **W**.

CACOS

Complex Inverse Cosine

```
MAT Z=CACOS(W)
```

where **W** is a complex scalar and **Z** is an array.

Redimensions **Z** to be a complex scalar and assigns to **Z** the complex principal value of the inverse cosine of **W**.

CATN

Complex Inverse Tangent

```
MAT Z=CATN(W)
```

where **W** is a complex scalar, **W** $\neq$ (0, 1) or (0, -1), and **Z** is an array.

Redimensions **Z** to be a complex scalar and assigns to **Z** the complex principal value of the inverse tangent of **W**.

CASINH

Complex Inverse Hyperbolic Sine

```
MAT Z=CASINH(W)
```

where **W** is a complex scalar and **Z** is an array.

Redimensions **Z** to be a complex scalar and assigns to **Z** the complex principal value of the inverse hyperbolic sine of **W**.

CACOSH

Complex Inverse Hyperbolic Cosine

```
MAT Z=CACOSH(W)
```

where **W** is a complex scalar and **Z** is an array.

Redimensions **Z** to be a complex scalar and assigns to **Z** the complex principal value of the inverse hyperbolic cosine of **W**.

CATANH — Complex Inverse Hyperbolic Tangent

```
MAT Z=CATANH(W)
```

where **W** is a complex scalar, **W** ≠ (1, 0) or (−1, 0), and **Z** is an array.

Redimensions **Z** to be a complex scalar and assigns to **Z** the complex principal value of the inverse hyperbolic tangent of **W**.

Roots of a Complex Number

This keyword allows you to easily and accurately determine the set of all Nth roots of a complex number, where N is a positive integer. The roots are returned in an $N \times 2$ array where each row representing a complex root, with the real part of the root in the first column and the imaginary part of the root in the second column. Successive roots are in order of increasing argument (angle). The result array must have been given at least $2N$ elements in its original dimensioning statement.

CROOT — Roots of a Complex Number

```
MAT R=CROOT(Z,N)
```

where **R** is a matrix, **Z** is a complex scalar, and *N* is a numeric expression whose rounded integer value is positive.

Redimensions **R** to be an $P \times 2$ array (where P is the rounded integer value of N) and assigns to **R** all the values of $\mathbf{Z}^{1/P}$.

Examples

CSIN, CTAN, CCOSH, CACOSH

Input/Result

```
clear vars RTN
dim z(1),w(2,2) RTN
```

Z is dimensioned to be a complex scalar. **W** is just an array.

```
mat input z RTN
```

```
Z(0)?
```

Input/Result

```
21,-2 RTN
```

Z now represents 21 − 2*i*.

```
mat w=csin(z) RTN
mat disp w; RTN
```

```
 3.14766223822   1.98653756813
```

The complex sine of 21 − 2*i*.

```
mat w=ctan(z) RTN
mat disp w; RTN
```

```
-3.40609899325E-2
-1.01418411496
```

The complex tangent of 21 − 2*i*.

```
mat w=cacosh(z) RTN
mat disp w; RTN
```

```
 3.74163218344
-9.50579293965E-2
```

The principal value of the inverse hyperbolic cosine of 21 − 2*i*.

```
mat w=ccosh(w) RTN
mat disp w; RTN
```

```
 21  -2
```

The hyperbolic cosine of the principal value of the inverse hyperbolic cosine of 21 − 2*i*.

CSQR, CLOG

Input/Result

```
clear vars RTN
default on RTN
```

The values we will use will produce error messages and stop the operation unless we choose the DEFAULT ON option.

```
dim z1(1),z2(0,1),r(1,0) RTN
```

All of these are dimensioned to be complex scalars.

```
mat input z1,z2 RTN
```

```
Z1(0)?
```

```
-1,1e-499 RTN
```

Z1 represents $-1 + \text{EPS}i$.

```
Z2(0,0)?
```

```
-1,-1e-499 RTN
mat r=csqr(z1)@mat disp r; RTN
```

And **Z2** represents $-1 - \text{EPS}i$.

```
WARNING:num too small
 0   1
```

The warning indicates that the result is so close to the imaginary axis that its real part is less than 1.E-499 and so cannot be shown as anything but zero, even though it is nonzero. The principal value of the square root of $-1 + \text{EPS}i$ is thus very close to $0 + i$.

```
mat r=csqr(z2)@mat disp r; RTN
```

```
WARNING:num too small
 0  -1
```

The warning here occurs for the same reason as the previous warning. The result this time is very close to $0 - i$. The jump between this value and that of the previous example is the direct consequence of the branch cut along the negative real axis for the complex square root function.

```
mat r=clog(z1)@mat disp r; RTN
```

```
WARNING:num too small
 0   3.14159265359
```

```
mat r=clog(z2)@mat disp r; RTN
```

```
WARNING:num too small
 0  -3.14159265359
```

Again, the jump in value from πi to $-\pi i$ when the argument changes from $-1 + \text{EPS}i$ to $-1 - \text{EPS}i$ is a direct result of the branch cut, this time for the complex logarithm function.

Additional Information

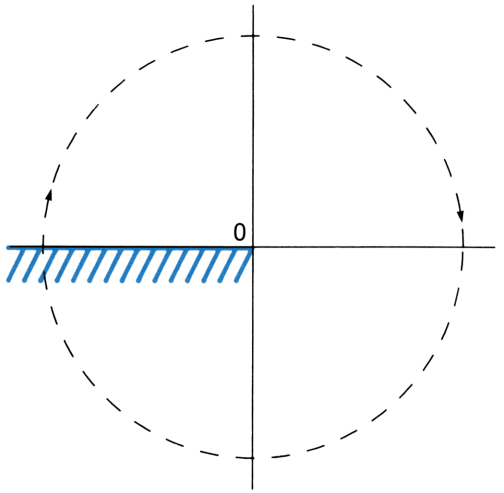
In general, the inverse of a function $f(z)$—denoted $f^{-1}(z)$—has more than one value for any argument z. For example, $\cos^{-1}z$ has infinitely many values for each argument. However, the Math Pac calculates the single *principal value*, which lies in the part of the range defined as the *principal branch* of the inverse function $f^{-1}(z)$. In this discussion, uppercase letters will denote a single-valued inverse function (like $\text{COS}^{-1}z$) to distinguish it from its multivalued inverse ($\cos^{-1}z$).

For some inverse functions, the definitions of the principal branches are not universally agreed upon. The branches used by the Math Pac were carefully chosen. They are all analytic in the regions where their real-valued counterparts are defined; that is, the branch cut occurs where the real-valued inverse is undefined. In addition, most of the important symmetries are preserved. For example,

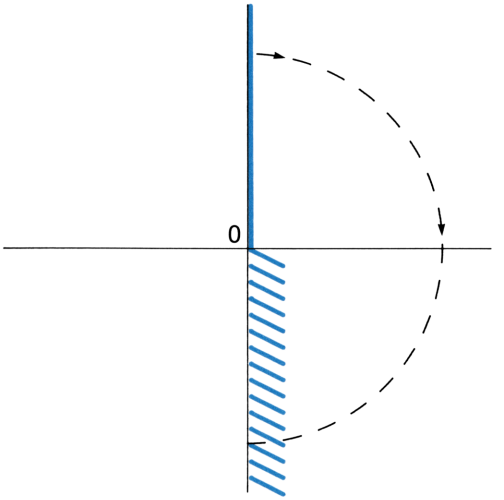
$$\mathrm{SIN}^{-1}(-z) = -\mathrm{SIN}^{-1}(z) \text{ for all } z.$$

The illustrations that follow show the principal branches of the inverse functions that the Math Pac calculates. The left-hand graph in each figure represents the cut domain of the inverse function; the right-hand graph shows the range of the principal branch. The blue and the black lines in the left-hand graph are mapped, under the inverse function, to the corresponding blue and black lines in the right-hand graph.

$$\sqrt{z} = \sqrt{r}\, e^{i\theta/2} \text{ for } -\pi < \theta \leq \pi$$

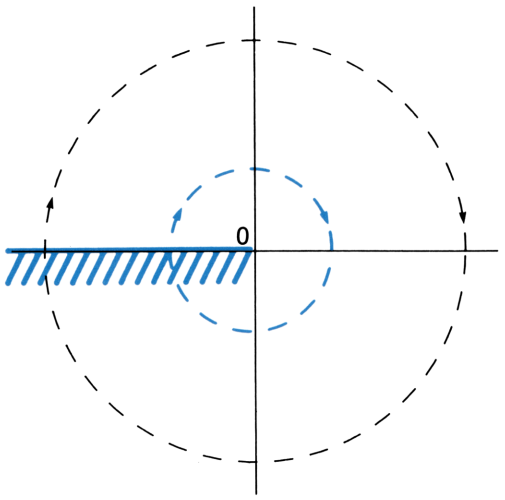


z

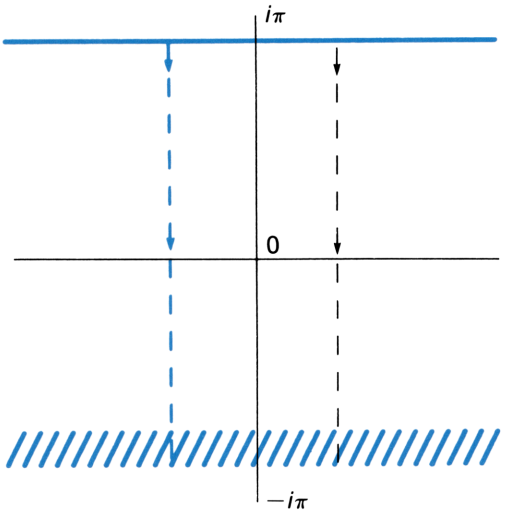


$w = \sqrt{z}$

LN(z) = ln r + $i\theta$ for $-\pi < \theta \leq \pi$

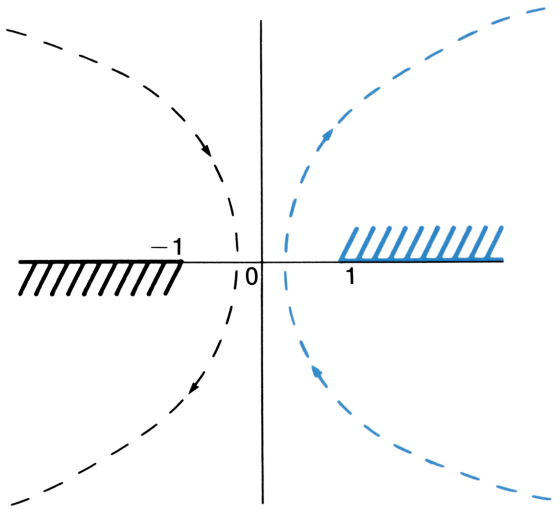


z

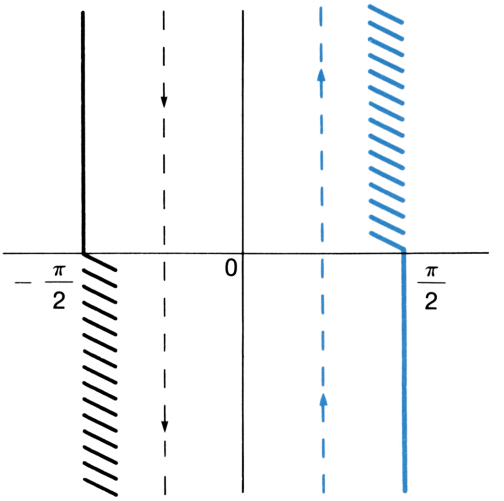


w = LN(z)

$\sin^{-1}(z) = -i \ln [iz + (1 - z^2)^{1/2}]$

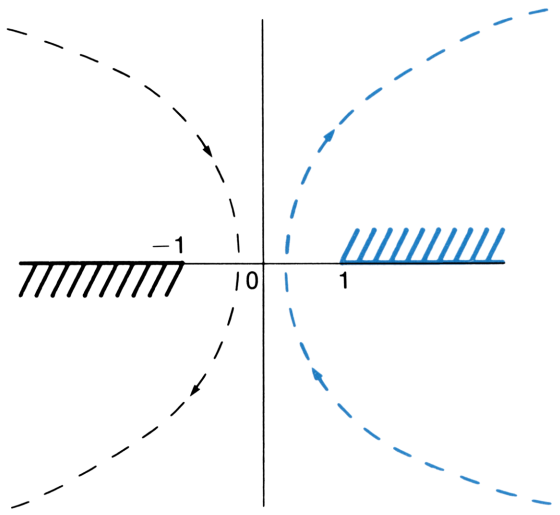


z

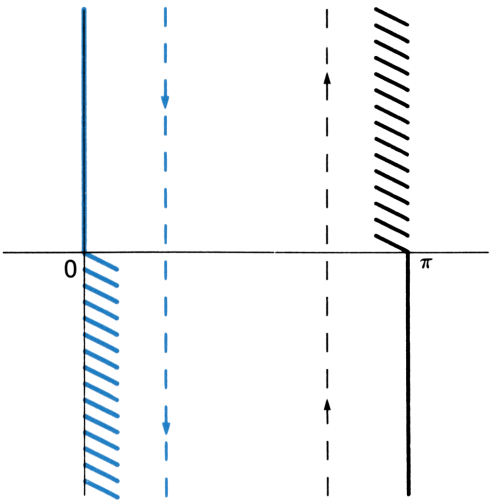


w = SIN^{-1}(z)

$$\cos^{-1}(z) = -i \ln [z + (z^2 - 1)^{1/2}]$$

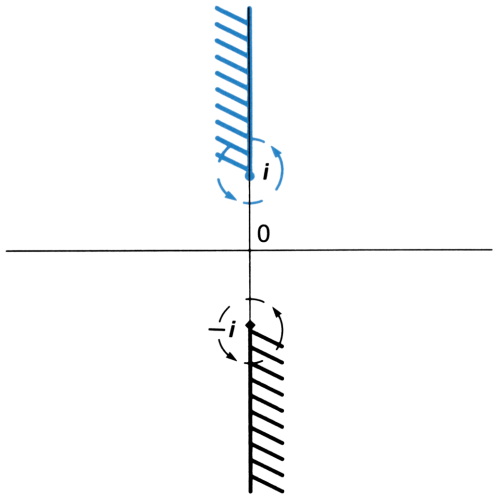


z

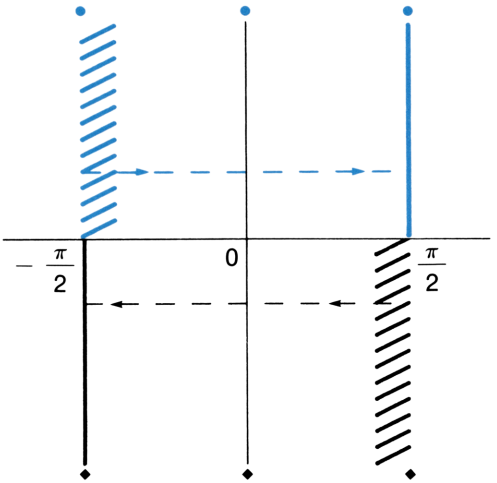


$w = \mathrm{COS}^{-1}(z)$

$$\tan^{-1}(z) = \frac{i}{2} \ln \frac{i + z}{i - z}$$

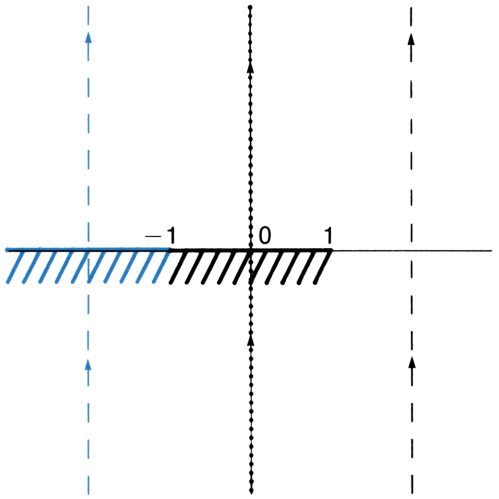


z

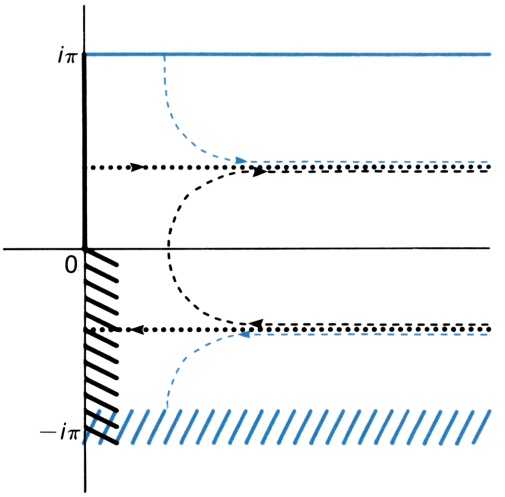


$w = \mathrm{TAN}^{-1}(z)$

$$\cosh^{-1}(z) = \ln[z + (z^2 - 1)^{1/2}]$$



z



$w = \text{COSH}^{-1}(z)$

The principal branches in the last four graphs above are obtained from the equations shown, but don't necessarily use the principal branches of $\ln z$ and $\sqrt{z}$.

The remaining inverse functions may be determined from the illustrations above and the following equations:

$$\text{SINH}^{-1}(z) = -i\ \text{SIN}^{-1}(iz)$$

$$\text{TANH}^{-1}(z) = -i\ \text{TAN}^{-1}(iz)$$

$$w^z = \exp\ (z\ \text{LN}\ w)$$

To determine *all* values of the inverse functions, use the expressions below to derive these values from the principal values calculated by the Math Pac. In these expressions, $k = 0, \pm 1, \pm 2$, and so on.

$$\sqrt{z} = \pm\text{SQR}(z) \qquad \ln(z) = \text{LN}(z) + 2\pi ik \qquad w^z = w^z\ e^{2\pi ikz}$$

$$\sin^{-1}(z) = (-1)^k\ \text{SIN}^{-1}(z) + \pi k \qquad \sinh^{-1}(z) = (-1)^k\ \text{SINH}^{-1}(z) + \pi ik$$

$$\cos^{-1}(z) = \pm\text{COS}^{-1}(z) + 2\pi k \qquad \cosh^{-1}(z) = \pm\text{COSH}^{-1}(z) + 2\pi ik$$

$$\tan^{-1}(z) = \text{TAN}^{-1}(z) + \pi k \qquad \tanh^{-1}(z) = \text{TANH}^{-1}(z) + \pi ik$$

Section 11

Complex Matrix Operations

The keywords in this section perform complex operations on arrays with complex values. The form in which complex numbers are stored in an array is similar to the form they are stored in complex scalars. The Math Pac can interpret any array with an even number of columns as an array with complex values. The first column of the array will represent the real part of the complex array's first column, the second column will represent the imaginary part, and so on. For example, the 2×6 matrix

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 10 & 11 & 12 \end{bmatrix}$$

will represent the complex 2×3 matrix

$$\begin{bmatrix} 1+2i & 3+4i & 5+6i \\ 7+8i & 9+10i & 11+12i \end{bmatrix}.$$

We will say that an array is a *complex array* if it is doubly subscripted and has an even number of columns.

The operations of addition, subtraction, and negation are identical for real- and complex-valued arrays, so these operations are not included in this section. You can use the array addition, array subtraction, and array negation operations discussed in section 5 in exactly the same manner for both complex and real arrays.

CMMULT

Complex Matrix Multiplication

MAT **A**=CMMULT(**B**,**C**)

where **B**, **C** are complex matrices such that there are twice as many columns in **B** as there are rows in **C**, and **A** is a matrix.

Redimensions **A** to have the same number of rows as **B** and the same number of columns as **C**, and assigns to **A** the values of the complex matrix product **BC** according to the usual rules of complex matrix multiplication.

CTRN — Complex Conjugate Transpose

```
MAT A=CTRN(B)
```

where **B** is a complex matrix and **A** is a matrix.

Redimensions **A** to have half as many rows as **B** has columns and twice as many columns as **B** has rows—if **B** is an $N \times 2P$ matrix, **A** will be a $P \times 2N$ matrix. **A** will be assigned the values of the complex conjugate transpose of the complex matrix represented by **B**.

CINV — Complex Matrix Inverse

```
MAT A=CINV(B)
```

where **B** is a *square* complex matrix (twice as many columns as rows) and **A** is a matrix.

Redimensions **A** to be exactly the same size as **B** and assigns to **A** the values of the matrix inverse of the complex matrix represented by **B**.

CDET — Complex Determinant

```
MAT Z=CDET(A)
```

where **A** is a *square* complex matrix (twice as many columns as rows) and **Z** is an array.

Redimensions **Z** to be a complex scalar and assigns to **Z** the complex value of the determinant of the complex matrix represented by **A**.

CIDN — Complex Identity Matrix

```
MAT A=CIDN
```

where **A** is a *square* complex matrix (twice as many columns as rows).

Assigns to **A** the values of the complex identity matrix. **A** is not redimensioned.

CSYS

Complex System Solution

MAT **Z**=CSYS(**A**,**B**)

where **A** is a *square* complex matrix (twice as many columns as rows), **B** is a complex matrix with the same number of rows as **A**, and **Z** is a matrix.

Redimensions **Z** to be exactly the same size as **B** and assigns to **Z** the complex values that solve the complex matrix equation

$$\mathbf{AZ} = \mathbf{B}.$$

Examples

CTRN, CIDN

Input/Result

```
clear vars RTN
dim a(1,5),b(3,6) RTN
```

Dimensions **A** to be a 2 × 6 array and **B** a 4 × 7 array.

```
mat input a RTN
```

```
A(0,0)?
```

```
1,2,3,4,5,6,7,8,9,10,11,12 RTN
```

A now represents the complex 2 × 3 matrix

$$\begin{bmatrix} 1+2i & 3+4i & 5+6i \\ 7+8i & 9+10i & 11+12i \end{bmatrix}.$$

```
mat b=ctrn(a) RTN
mat disp b; RTN
```

```
 1 -2  7  -8
 3 -4  9 -10
 5 -6 11 -12
```

B now represents the complex conjugate transpose of **A**.

```
redim a(1,3) RTN
mat a=cidn RTN
mat disp a; RTN
```

```
 1   0   0   0
 0   0   1   0
```

The 2 × 2 complex identity matrix:

$$\begin{bmatrix} 1+0i & 0+0i \\ 0+0i & 1+0i \end{bmatrix}$$

CINV, CMMULT

Input/Result

```
clear vars RTN
dim a(2,5),b(2,5) RTN
mat input a RTN
```

```
A(0,0)?
```

```
1,1,0,0,0,0,1,1,2,2,0,0,1,1,
2,2,3,3 RTN
mat disp a; RTN
```

```
 1   1   0   0   0   0
 1   1   2   2   0   0
 1   1   2   2   3   3
```

A represents the complex 3 × 3 matrix

$$\begin{bmatrix} 1+i & 0+0i & 0+0i \\ 1+i & 2+2i & 0+0i \\ 1+i & 2+2i & 3+3i \end{bmatrix}$$

```
mat b=cinv(a) RTN
mat disp b; RTN
```

```
 .5 -.5   0   0   0   0
-.25  .25  .25 -.25  0   0
0   0 -.166666666666
 .166666666667  .166666666666
-.166666666667
```

B represents the complex 3 × 3 matrix

$$\begin{bmatrix} 1/2 - i/2 & 0+0i & 0+0i \\ -1/4 + i/4 & 1/4 - i/4 & 0+0i \\ 0+0i & -1/6 + i/6 & 1/6 - i/6 \end{bmatrix}$$

```
mat b=cmmult(b,a) RTN
```

```
mat disp b; RTN

1   0   0   0   0   0
0   0   1   0   0   0
0   0   0   0   .999999999999
-.000000000003
```

CSYS

You can use CSYS to solve a system of equations with several choices of constants *all at once*. For example, to solve the systems

$$(2 + 3i)\ z_1 + (.7 - i)\ z_2 = 2 + 21i$$

$$(4 - 1.3i)\ z_1 + (4 + 0i)\ z_2 = 1 + 3i$$

and

$$(2 + 3i)\ u_1 + (.7 - i)\ u_2 = 0$$

$$(4 - 1.3i)\ u_1 + (4 + 0i)\ u_2 = -3i$$

and

$$(2 + 3i)\ w_1 + (.7 - i)\ w_2 = 9 - .22i$$

$$(4 - 1.3i)\ w_1 + (4 + 0i)\ w_2 = -3.5 + i$$

we could write the entire system as the complex matrix equation $\mathbf{AX} = \mathbf{B}$ where

$$\mathbf{A} = \begin{bmatrix} 2 + 3i & .7 - i \\ 4 - 1.3i & 4 + 0i \end{bmatrix}, \quad \mathbf{X} = \begin{bmatrix} z_1 & u_1 & w_1 \\ z_2 & u_2 & w_2 \end{bmatrix},$$

$$\text{and } \mathbf{B} = \begin{bmatrix} 2 + 21i & 0 + 0i & 9 - .22i \\ 1 + 3i & 0 - 3i & -3.5 + i \end{bmatrix}.$$

This is the form that CSYS accepts, and the one we will use to solve the system.

Input/Result

```
clear vars RTN
dim a(1,3),x(1,5),b(1,5) RTN
mat input a,b RTN
```

```
A(0,0)?
```

```
2,3,.7,-1 RTN
```

```
A(1,0)?
```

```
4,-1.3, 4,0 RTN
```

Don't forget to enter both the real and imaginary parts of each complex number, even if the value is zero.

```
B(0,0)?
```

```
2,21,0,0,9,-.22 RTN
```

```
B(1,0)?
```

```
1,3,0,-3,-3.5,1 RTN
mat x=csys(a,b) RTN
mat disp using "x,dd.d";x RTN
```

```
 4.4   1.4    .2   -.1    .5   -2.0
-4.6    .7   -.1   -.6   -.7    2.4
```

The solution of the matrix equation gives the solution of all three systems:

$z_1 = 4.4 + 1.4i$
$z_2 = -4.6 + .7i$

$u_1 = .2 - .1i$
$u_2 = -.1 - .6i$

$w_1 = .5 - 2.0i$
$w_2 = -.7 + 2.4i$.

Additional Information

By combining REDIM and real array operations with the operations of this section, you have at your disposal all of the common operations on complex matrices. As already mentioned, addition, subtraction, and negation of arrays are identical operations for real and complex arrays.

Scalar Multiple of a Complex Array

If you multiply an array **B** by a real scalar x using

MAT **A** = (x) * **B**,

the result is correct whether **B** represents a real or a complex array. For multiplying a complex array by a complex scalar, use the following procedure.

If **B** is an $N \times 2P$ array representing an $N \times P$ complex matrix, and **Z** is a 1×2 array representing a complex scalar:

1. Redimension **B** to be an $NP \times 2$ array. This makes **B** into a complex column vector.
2. Multiply **B** on the right by **Z** using complex matrix multiplication. (You must use *complex* matrix multiplication, not real matrix multiplication.)
3. Redimension the result of the multiplication to be an $N \times 2P$ array. The result array is now the complex product of **B** with the complex scalar **Z**. Remember to redimension **B** if you want it in its original form.

The following example demonstrates this procedure.

Input/Result

```
clear vars RTN
```

```
dim b(2,3),a(2,3),z(0,1) RTN
```

A and **B** are dimensioned to be 3×4 arrays; **Z** is a 1×2 array.

```
mat input b RTN
```

```
B(0,0)?
```

```
1,0,0,2,0,2,1,0,0,2,0,2 RTN
```

B now represents the complex matrix

$$\begin{bmatrix} 1+0i & 0+2i \\ 0+2i & 1+0i \\ 0+2i & 0+2i \end{bmatrix}.$$

```
redim b(5,1) RTN
```

B is redimensioned as a complex column vector.

```
mat disp b; RTN
```

```
 1   0
 0   2
 0   2
 1   0
 0   2
 0   2
```

```
mat input z RTN
```

```
Z(0,0)?
```

```
0,1 RTN
```

Z represents $0 + i$.

```
mat a=cmmult(b,z) RTN
```

A = **B** i.

```
redim a(2,3),b(2,3) RTN
mat disp a; RTN
```

```
 0   1  -2   0
-2   0   0   1
-2   0  -2   0
```

Complex Conjugate of a Complex Array

You can use a similar technique to find the complex conjugate of a complex array. For example, if **B** is an $N \times 2P$ array representing an $N \times P$ complex matrix, you can find its complex conjugate as follows.

1. Redimension **B** to be an $NP \times 2$ array and multiply **B** on the right by the 2×2 array

$$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

   using real array multiplication.

2. Redimension the result to be an $N \times 2P$ array.

The result will then be the complex conjugate of the original. Be sure to redimension **B** if you want it in its original form.

Note that combining this complex conjugation with the complex conjugate transpose operation gives you the complex transpose of a complex matrix.

Complex Form of a Real Array

As a final example of these operations, note that you can use the following procedure to create a complex matrix with zero imaginary part and the same real part as a given real matrix. The resulting matrix then represents the same matrix, but can be used in complex array operations.

If **B** is a real $N \times P$ array you wish to put in complex form:

1. Dimension the array in which you wish to store the result to be $N \times 2P$ and assign it the values of the zero array.
2. Assign the result array the values of **B**. This also redimensions the result to be $N \times P$, and has no effect on the inaccessible zero values.
3. Redimension the result array to be $2 \times NP$. The result array now consists of two rows, the first row contains the values of the **B** and the second row contains only zeros.
4. Take the (real) transpose of the result array.
5. Redimension the result array to be $N \times 2P$. The result array now has the values of **B** alternating with zeros.

The following program will convert a 4×3 real array to its corresponding complex array using the above procedure.

```
10 OPTION BASE 1
20 DIM B(4,3),A(4,6)
30 MAT A=ZER
40 MAT INPUT B
50 MAT A=B
60 REDIM A(2,12)
70 MAT A=TRN(A)
80 REDIM A(4,6)
90 MAT DISP A;
```

To create the complex form of the real matrix

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix},$$

type in the program and use the following keystrokes.

Input/Result

RUN

```
B(1,1)?
```

1,2,3,4,5,6,7,8,9,10,11,12 RTN

```
 1   0   2   0   3   0
 4   0   5   0   6   0
 7   0   8   0   9   0
 10   0   11   0   12   0
```

The complex form of the given real matrix.

Section 12

Finding Roots of Polynomials

The keyword in this section finds *all* solutions—both real and complex—of $P(x) = 0$, where P is a polynomial of your choice with real coefficients. If P is a polynomial of degree n there will be n (not necessarily distinct) solutions of this equation, so this keyword resembles an array operation in its format.

To use this keyword to find the solutions of the equation $P(x) = 0$, where

$$P(x) = a_n\, x^n + a_{n-1}\, x^{n-1} + \ldots + a_1\, x + a_0,$$

first store the coefficients a_n, a_{n-1}, ..., a_0 in an array with $n + 1$ elements in all. They should be stored in the order indicated above, with the coefficient of the highest power first and the constant term last. Aside from the total number of elements in the array, which indicates to the Math Pac the degree of the polynomial, the dimensions of the array are irrelevant. For example, the arrays

$$[6,\ 5,\ 4,\ 3,\ 2,\ 1],\quad \begin{bmatrix} 6 & 5 & 4 \\ 3 & 2 & 1 \end{bmatrix},\ \text{and}\quad \begin{bmatrix} 6 & 5 \\ 4 & 3 \\ 2 & 1 \end{bmatrix}$$

all can represent the polynomial

$$6x^5 + 5x^4 + 4x^3 + 3x^2 + 2x + 1.$$

The array in which you wish the roots to be stored must be doubly subscripted and must have been given at least $2n$ elements in its original dimensioning statement. The degree of the polynomial you can find the roots of is limited only by the amount of memory you have available.

PROOT

Roots of a Polynomial

MAT **R**=PROOT(**P**)

where **P** is an array with at least two elements, and **R** is a matrix.

Redimensions **R** to be an $N \times 2$ array (where **P** has a total of $N + 1$ elements) and assigns to **R** the (complex) values of the solutions of the equation $P(x) = 0$ (where P is the polynomial of degree N whose coefficients are the values of the elements of **P**). The first column of **R** will contain the real parts of the roots and the second column will contain the imaginary parts.

Example

Input/Result

```
clear vars RTN
```

```
dim s(6),w(5,1) RTN
```

S will contain the seven coefficients of a sixth degree polynomial, and **W** will contain its six complex roots.

```
mat input s RTN
```

```
S(0)?
```

```
5,-45,225,-425,170,370,
-500 RTN
```

S now represents the polynomial

$$5x^6 - 45x^5 + 225x^4 - 425x^3 + 170x^2 + 370x - 500.$$

```
mat w=proot(s) RTN
mat disp w; RTN
```

```
 1  -1
 1   1
-1   0
 2   0
 3  -4
 3   4
```

The roots of this polynomial are $1 - i$, $1 + i$, $-1 + 0i$, $2 + 0i$, $3 - 4i$, and $3 + 4i$.

Additional Information

The Math Pac uses a modified version of Laguerre's method together with extended precision arithmetic and a sophisticated scaling and deflation (polynomial division) procedure to find the roots of polynomials. Ordinarily, an array with $n + 1$ elements represents a polynomial of degree n, and should therefore have n roots. However, if the leading coefficient happens to be zero (so that the polynomial is actually of degree $n - 1$) this method will calculate that the polynomial has a root at complex infinity, and so will report (9.99999999999E499, 9.99999999999E499) as a root. This will normally produce an error message; if the DEFAULT ON option is in effect, the Math Pac will display a warning message and then correctly find the roots of the lower degree polynomial.

There are several methods of gauging the accuracy of the calculated roots. The first method is to calculate the value of the polynomial at the alleged root, and compare this value with zero. Although quite straightforward in theory, this has a number of drawbacks in practice. It may easily happen that the the root calculated is the closest machine-representable number to a true root, but because the polynomial has such a large value for its derivative at this root, the value of the polynomial at the calculated root is very large. A simple example of this phenomenon is given by the polynomial $1E20x^2 - 2E20$. A true root is $\sqrt{2}$; a calculated root is 1.41421356237, which is the machine-representable number closest to $\sqrt{2}$. However, the value of the polynomial at this approximation to the square root of 2 is $-1{,}000{,}000{,}000$, a number which *seems* very far from zero.

Another drawback of the above method is that because of the limited precision available in *any* numerical calculation, the roundoff errors that occur in the calculation of the polynomial's value may completely eliminate the significance of the difference between the calculated value and zero. This is especially true when the polynomial is of large degree, has coefficients widely varying in size, or has roots of high multiplicity.

A second method of gauging the accuracy of the calculated roots is to attempt to reconstruct the polynomial from these roots. If the reconstructed polynomial closely resembles the original, the roots are then judged to be accurate. This technique is less sensitive to the problems that affect the polynomial evaluation method. Of course, this method does not give information on the accuracy of an individual root.

The program given below asks you for a polynomial and then calculates the roots of the polynomial and reconstructs the polynomial from these roots. If you wish, the program continues and calculates the value of the polynomial at a root, or any other real or complex point you choose.

To compute the reconstructed polynomial, this program starts with the polynomial 1, and then successively multiplies the polynomial by the linear factors $(x - r)$, where r is a calculated real root, or by the quadratic $x^2 - (r_1 + r_2)x + (r_1\ r_2)$ where r_1 and r_2 are a pair of complex conjugate roots.

To compute the value of the polynomial at a complex point z, the program uses synthetic division (synthetic substitution) of the polynomial by the linear binomial $(x - z)$ and the fact that the remainder of such a division is the value of the polynomial at the point z. This method of computation has the

advantage of avoiding much of the roundoff error that would occur in a more straightforward calculation.

```
10 OPTION BASE 1
20 DIM P(51,1),Q(52,2),C(52),T(52),R(50,2)
```

P will contain the coefficients of the original polynomial. **Q** will contain a complex copy of the coefficients used in the synthetic division. **C** will contain the reconstructed coefficients. **T** is used as a temporary storage for intermediate steps in the reconstruction. **R** will contain the calculated roots.

```
30 DELAY 1
```

Throughout the program we will lengthen the delay before something is to be displayed, and shorten it during a calculation.

```
40 DISP "What degree is the polynomial? (It
   must be less that 51)"
50 INPUT D
```

D is the degree of the polynomial.

```
60 REDIM P(D+1,1),Q(D+2,2),C(D+2),
   T(D+2),R(D,2)
```

Redimensions the variables to the appropriate sizes for a polynomial of degree *D*.

```
70 DISP "Enter the coeff.s of the polynomial"
80 DELAY 0
90 MAT INPUT P
```

P will now contain the coefficients.

```
100 A1=P(1,1)
```

The reconstructed polynomial will always have leading coefficients equal to 1. We will scale the reconstructed polynomial by A_1 to make the leading coefficients match. Note that this will not work if $A_1 = 0$.

```
110 MAT R=PROOT(P)
```

Calculates the roots and stores them in **R**.

```
120 DELAY 1
130 DISP "The roots are"
140 MAT DISP R;
```

Displays the calculated roots.

```
150 DELAY 0
160 REM*******
```

We now begin the process of reconstructing the polynomial from the roots found.

```
170 MAT C=ZER
180 C(2)=1
```

C now represents the polynomial 1.

```
190 MAT T=C
200 F=0
```

This just gives **T** some values for initialization.

F is a "flag": if F equals zero, this will indicate that the previous root was real; if F equals one, this will indicate that the previous root was complex. Since a root and its complex conjugate will be consecutive on our list of roots, when we find a complex root, we will multiply by the quadratic and then ignore the next root on the list.

```
210 FOR J=1 TO D
```

J represents the number of the root we are currently working with.

```
220 IF R(J,2)=0 THEN GOSUB 320 ELSE
    GOSUB 380
```

If the current root is real, we will multiply by the linear factor (subroutine starting at 320). If the current root is complex, we will multiply by the corresponding quadratic factor, if it hasn't already been done (subroutine starting at 380).

```
230 NEXT J
240 MAT C=(A1)*C
```

Scales the reconstructed polynomial by the leading coefficient of the original.

```
250 DELAY 1
260 DISP "The reconstructed polynomial is"
270 FOR K=2 TO D+2
280 DISP C(K);"*X^";D-K+2;"+";
290 NEXT K
```

Displays the polynomial in standard form.

The leading coefficient is `C(2)`, not `C(1)`. `C(1)` will always be zero.

```
300 INPUT "Do you wish to evaluate the poly.?
    (N will stop the program)","Y";U$
310 IF UPRC$(U$)="N" THEN STOP ELSE
GOTO 470
320 REM *******
```

This begins the subroutine to multiply the polynomial by a linear factor.

```
330 FOR L=3 TO D+2
340 T(L)=C(L)-R(J,1)*C(L-1)
```

During this calculation, **T** stores the results of the multiplication. After it's done, the result is again stored in **C**.

```
350 NEXT L
360 MAT C=T
370 RETURN
```

```
380 REM *******
390 IF F=1 THEN LET F=0 @ RETURN ELSE
    LET F=1
400 LET A=2*R(J,1)
410 LET B=R(J,1)^2+R(J,2)^2
420 FOR L=3 TO D+2
430 T(L)=C(L)-A*C(L-1)+B*C(L-2)
440 NEXT L
450 MAT C=T
460 RETURN
470 REM *******
480 DIM X(2),Z(2),W(2)
490 INPUT "Evaluate at a root, or some other
    value? (R for root)"," ";U$
500 IF UPRC$(U$)="R" THEN GOSUB
    710@GOTO 550
510 DELAY 1
520 DISP "Enter the real and imaginary parts of
    the value"
530 DELAY 0
540 MAT INPUT X
550 REM *******
560 DELAY 0
570 MAT Q=ZER
580 MAT Q=P
590 REDIM Q(2,D+1)
600 MAT Q=TRN(Q)
```

This begins the subroutine to multiply the polynomial by a quadratic factor.

If the flag equals one, we have already used the quadratic corresponding to this root and so we clear the flag and go on to the next root. If the flag doesn't equal one, we set the flag and continue the process.

1, *A*, and *B* are the coefficients of the quadratic factor.

T stores the results during the multiplication. The results are again stored in **C** when we are done.

This begins the polynomial evaluation routine.

X, **Z**, and **W** will be used as complex scalars.

The subroutine starting at 710 looks up the value of the root.

X contains the value of the point at which the polynomial will be evaluated, either from the MAT INPUT or the from the lookup of the root.

This section assigns to **Q** the values of the complex form of **P**.

```
610 FOR L=2 TO D+1
620 Z(1)=Q(L,1) @ Z(2)=Q(L,2)

630 W(1)=Q(L-1,1) @ W(2)=Q(L-1,2)
640 MAT W=CMULT(W,X)

650 MAT Z=CADD(Z,W)
660 Q(L,1)=Z(1) @ Q(L,2)=Z(2)
670 NEXT L
680 DELAY 1
690 DISP "The value of the polynomial is ";
    Q(D+1,1);"+i*";Q(D+1,2)

700 GOTO 300
710 REM ******

720 DISP "Which root? (1,...,";D;")"
730 INPUT J
740 X(1)=R(J,1) @ X(2)=R(J,2)
750 RETURN
```

The values of **Q** are converted to complex scalars so that the complex arithmetic operations can be used.

This calculates the next term in the synthetic division.

The value of the polynomial is the last (remainder) term.

Gets another point to use in the evaluation.

This begins the subroutine to look up the value of a root.

J is the number of the root.

If we wanted to find and evaluate the roots of the polynomial

$$x^6 + x^5 + x^4 + x^3 + x^2 + x + 1,$$

we would run the program using the following keystrokes.

Input/Result

RUN

```
What degree is the polynomial?
(It must be less than 51)
?
```

6 RTN

```
Enter the coeff.s of the polyno
mial
P(1,1)?
```

1,1,1,1,1,1,1 RTN

```
The roots are
-.222520933956   .974927912182
-.222520933956 -.974927912182
-.900968867902 -.433883739118
-.900968867902   .433883739118
 .623489801859 -.781831482468
 .623489801859   .781831482468
The reconstructed polynomial is
1 *X^ 6 + .99999999999 *X^ 5 +
1 *X^ 4 + .99999999998 *X^ 3 +
1 *X^ 2 + .99999999999 *X^ 1 +
1 *X^ 0 +

Do you wish to evaluate the poly
.? (N will stop the program)Y
```

y RTN

Any response but "N" or "n" will be interpreted as "yes."

y RTN

```
Evaluate at a root or some othe
r value? (R for root)
```

r RTN

Any reponse but "R" or "r" will be interpreted as "some other value."

```
Which root? (1,..,6 )
?
```

1 RTN

```
The value of the polynomial is
0 +i* 7.524E-13
Do you wish to evaluate the pol
y.?(N will stop the program)Y
```

The value of the polynomial at the first computed root.

Y RTN

```
Evaluate at a root, or some oth
er value? ( R for root)
```

other RTN

```
Enter the real and imaginary pa
rts of the value
X(1)?
```

-.2, .9

```
The value of the polynomial is
.222523 +i* .185814
Do you wish to evaluate the poly
.? (N will stop the program)Y
```

n RTN

About the Algorithm

The Math Pac uses Laguerre's method to find the roots of the polynomial, one root at a time, by computing a sequence of approximations Z_1, Z_2, . . . , to a root using the formula $Z_{k+1} = Z_k + S_k$, where S_k (called the Laguerre step) is given by the formula

$$\frac{-n\ P(Z_k)}{P'(Z_k) \pm [(n-1)^2\ (P'(Z_k))^2 - n\ (n-1)\ P(Z_k)\ P''(Z_k)]^{1/2}}$$

where P, P', P'' are the polynomial and its first and second derivatives, n is the degree of the polynomial, and the sign in the denominator is chosen to give the Laguerre step of smaller magnitude. Polynomials of degree 1 or 2 are solved using linear factorization or the quadratic formula. Laguerre's method is cubically convergent to simple zeros and linearly convergent to zeros of multiplicity greater than one.

The operation of PROOT is global, in the sense that you are not required to supply an initial guess. PROOT always attempts to begin its search for a root at the origin of the complex plane. An annulus that contains the root of smallest magnitude is determined, and the intial step is rejected if it would lead out of this region. If the initial step is rejected, a spiral search is begun from the inner radius to the outer radius of the annulus, and continues until an acceptable initial guess is found. Once the iteration process has begun, a circle known to contain the root is computed around each Z_k. The Laguerre step is modified if it leads outside this circle, or if the value of the polynomial does not decrease. The roots are thus generally found in order of increasing magnitude, which minimizes the roundoff errors resulting from deflation.

PROOT uses a sophisticated technique to determine when an approximation Z_k should be accepted as a root. As the polynomial is being evaluated at Z_k, a bound for the roundoff error for the evaluation is also being computed. If the polynomial value is less than this bound, Z_k is accepted as a root. Z_k can also be accepted as a root if the value of the polynomial is decreasing but the size of the Laguerre step has become negligible. Before an approximation Z_k is used in an evaluation, its imaginary part is set to zero if this part is small compared to the step size. This improves performance, since real-number evaluations are faster than complex evaluations. If the Laguerre step size has become negligible but the polynomial is not decreasing, then the message PROOT failure is reported and the computation stops. This is expected never to occur in practice.

As the polynomial is being evaluated, the coefficients of the quotient polynomial (by either a linear or quadratic factor corresponding to the Z_k) are also computed. When an approximation Z_k is accepted as a root, this quotient polynomial becomes the polynomial whose roots are sought, and the process begins again.

Multiple Zeros

No polynomial rootfinder, including PROOT, can consistently locate zeros of high multiplicity with arbitrary accuracy. The general rule-of-thumb for PROOT is that for multiple or nearly-multiple zeros, resolution of the root is approximately $12/K$ significant digits, where K is the multiplicity of the root.

Accuracy

PROOT's criterion for accuracy is that the coefficients of the polynomial reconstructed from the calculated roots should closely resemble the original coefficients.

PROOT's performance with isolated zeros is illustrated by the 100th degree polynomial $x^{100} - 1$. When PROOT is used to find the roots of this polynomial, all but eight of the roots are found to 12-digit accuracy. Of these eight, all but two are accurate to 11 digits, with the 12th digit of either the real or imaginary part off by 1. The other two calculated roots are 3.27172623763E−14 $\pm i$ instead of 0 $\pm i$.

The polynomial $(x + 1)^{20}$, which has -1 as a root of multiplicity 20, was solved by PROOT to yield calculated roots of:

$-.999954866562 + 0i$
$-.985568935304 + 0i$
$-.676467025812 + 0i$
$-.746641243182 + .203801767293i$
$-.746641243182 - .203801767293i$
$-1.04166040212 + .334892343643i$
$-1.04166040212 - .334892343643i$
$-.827370927334 + .278237315935i$
$-.827370927334 - .278237315935i$
$-.92857985345 + .323524811701i$

$-.92857985345 - .323524811701i$
$-1.35739089743 + 0i$
$-1.33261156263 + .128152487571i$
$-1.33261156263 - .128152487571i$
$-.694494465769 + .107674717679i$
$-.694494465769 - .107674717679i$
$-1.15757200375 + .307382598202i$
$-1.15757200375 - .307382598202i$
$-1.26137867921 + .23658285644i$
$-1.26137867921 - .23658285644i$

The computed roots are inaccurate due to the high multiplicity of the true root. From the formula given previously you would expect no correct digits, or perhaps one, but note that the first pair of computed roots are more accurate than this. When a polynomial is reconstructed from these roots, its coefficients resemble the coefficients of the original polynomial to 11 or more digits.

Section 13

Solving $f(x) = 0$

You can use the keywords in this section to help you determine the solutions of an equation in one real variable. The first step in using this capability is to rewrite the equation to be solved in the form $f(x) = 0$. Even this form, however, is not explicit enough to be used by your Math Pac. You must write a user-defined function FN*function name* such that FN*function name*(x) calculates $f(x)$. (Refer to section 13 of the *HP-75 Owner's Manual* for information about user-defined functions.)

The keyword FNROOT can be used anywhere inside the program that contains the definition of the function (except inside the definition itself) to find the values of x for which $f(x)$ is zero.

The keyword FNGUESS is provided as an aid in interpreting the results of the FNROOT keyword. Since they are numeric valued, FNGUESS and FNROOT can be used alone or in combination with other functions and variables to form numeric expressions.

FNROOT

Function Root

FNROOT(*A*, *B*, FN*function name*(*X*))

where *A*, *B* are numeric expressions (not necessarily distinct), FN*function name* is a user-defined numeric function, and *X* is a numeric variable.

Returns the first value found (starting with guesses *A* and *B*) that is one of the following.

1) An exact root of the specified function.
2) An approximation to a root of the specified function, correct to 12 digits.
3) An approximation to a local minimum of the absolute value of the specified function.
4) In a region where the specified function is constant.
5) ±9.99999999999E499 if the search for a root led beyond the range of representable numbers.

X is a *dummy* variable—its inclusion here doesn't affect the use of this variable name in any other context.

This keyword can be used only in a program.

FNGUESS

Previous Estimate of Function Root

FNGUESS
Returns the next-to-last value tried as a solution in the most recent FNROOT statement. FNGUESS retains its value, even if your HP-75 is turned off, until FNROOT is again executed.

To help you distinguish among the five possibilities outlined above for FNROOT, you should *always* include a statement in your program that calculates and stores and/or checks the value of the specified function at the point found by FNROOT. Examples of such statements are

```
LET Z=FNF(FNROOT(A,B,FNF(X)))  and
DISP FNROOT(A,B,FNF(X)) @ DISP FNF(RES)
```

where FNF is the specified function.

By checking the values of FNF at the points returned by FNROOT and FNGUESS, you can interpret the result of FNROOT as follows.

- If FNF (*result of* FNROOT) = 0, the result of FNROOT is an exact root and the result of FNGUESS will be a number close to the root.
- If the result of FNROOT and the result of FNGUESS differ only in the twelfth significant digit, these two numbers surround the exact root.
- If the result of FNROOT and the result of FNGUESS differ but FNF (*result of* FNROOT) and FNF (*result of* FNGUESS) are equal, these results lie in a region where FNF is constant.

Example: Solving log (x) = e/x

To solve $\log(x) = e/x$, we first write the equation in the form $f(x) = 0$. This can be done by subtracting e/x from both sides of the equation, yielding $\log(x) - e/x = 0$. We can rewrite this in the equivalent but slightly more convenient form $x \log(x) - e = 0$. Since the left-hand side of this equation is undefined for $x \leq 0$, and we can't guarantee that the search for a root will not venture into this region, we will consider instead the equation $|x| \log |x| - e = 0$. This equation has exactly the same positive solution(s) as the first equation, but this equation makes sense for both positive and negative (but non-zero) numbers. The program below includes a user-defined function that computes the left-hand side of this equation, and uses FNROOT to find a solution of the equation.

```
10 DEF FNF(X)

20 FNF=ABS(X)*LOG(ABS(X))-EXP(1)
30 END DEF
40 INPUT A,B
50 R=FNROOT(A,B,FNF(X))
60 DISP "The value found (R) is";R
70 DISP "FNF(R)=";FNF(R)
80 DISP "FNGUESS=";FNGUESS
```

This user-defined function computes the left-hand side of the equation.

These will be the initial guesses.

To use the program we must decide on initial guesses. Although the initial guesses need not be in increasing order, or even distinct, a choice of initial guesses that surround a root will produce results more quickly in general. Noting that if $|X| < 1$ then FNF (X) will be negative and if $|X|$ is large (say, 100) then FNF (X) will be positive, we can choose .5 and 100 for our initial guesses.

Key in the program and [RUN] it, and when prompted with ? respond with .5,100 [RTN], which supplies the initial guesses. The computer will then display

```
The value found (R) is
2.71828182846
FNF(R)= 0
FNGUESS= 2.71828182832
```

Since FNF(R)= 0, the value given is an exact root for FNF.

Additional Information

Choosing Initial Estimates

When you use FNROOT to find roots of equations, the initial estimates determine where the search for a root will begin. If the two estimates surround an odd number of roots (signified by their function values having opposite signs), then FNROOT will find a root between the estimates quite rapidly. If the function values at the two estimates do not differ in sign, then FNROOT must search for a region where a root lies. Selecting initial estimates as near a root as possible will speed up this search. If you merely want to explore the behavior of the function near the initial estimates (such as to determine if there are any roots or extreme points nearby), then specify any estimates you like.

Another thing to consider is the range in which the equation is meaningful. In solving $f(x) = 0$, the variable x may only have a limited range in which it is conceptually meaningful as a solution. In this case, it is reasonable to choose initial estimates within this range. Frequently an equation that is applicable to a real problem has, in addition to the desired solution, other roots that are physically mean-

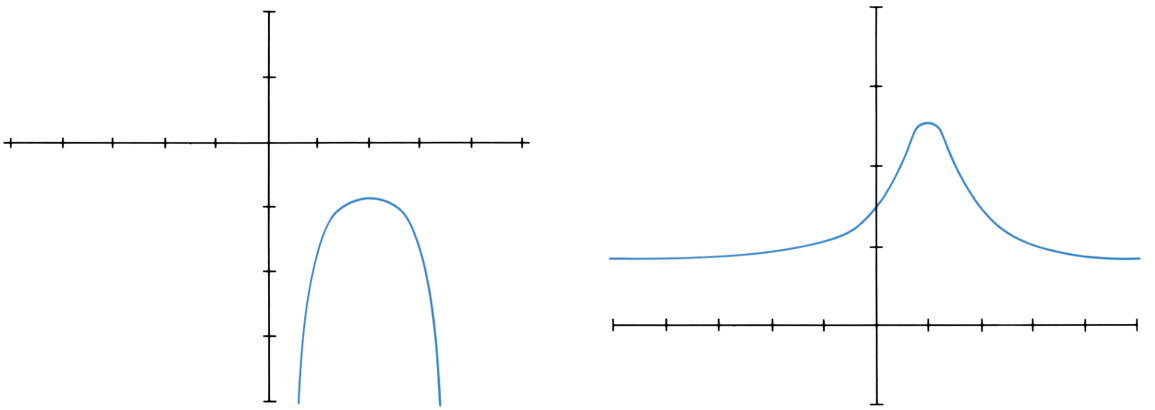
ingless. These usually occur because the equation being analyzed is appropriate only between certain limits of the variable. You should recognize this restriction and interpret the results accordingly.

Interpreting Results

When using FNROOT, always evaluate the function at the value returned, as described above. This enables you to interpret the results. There are two possibilities: the value of the function at the value returned by FNROOT is close to 0; or the value of the function at the value returned by FNROOT is not close to 0. It is up to you to decide how close is close enough to consider the value a root.

If the function value is too large, then the information returned by the keyword FNGUESS, together with information already considered, is sufficient to determine the general behavior of the function in the region. For example, suppose that FNROOT is used to find a root of a function—say, FNF(X)—and the value returned is *r*. If |FNF(r)| is too large to consider *r* a root, then there are several possibilities.

If FNF(r) and FNF(FNGUESS) have the same sign, then *r* is either an approximation to a local minimum of |FNF(X)| or in a region where the graph of FNF(X) is horizontal.



In these two cases, FNROOT sees no tendency of FNF(X) to decrease in absolute value, and so to cross the *x*-axis. It will then try to approximate a local extreme point, if any. This approximation can be resolved to further precision by further executions of FNROOT, using *r* and FNGUESS as initial estimates. Repeated execution of FNROOT in this manner will tend to convergence to the extreme point in many cases. The idea is that FNROOT can be used to find local extreme points, or the information about where the extreme points are can be used to re-direct the search elsewhere, in hope of finding a root. Here is an example program which can be used to find the local minimum and root of $f(x) = |x - 1|^{1/2}$. Note that $x = 1$ is both a root and local minimum, which makes it a difficult root to find. This program takes advantage of the way FNROOT finds minima to find the root.

```
10 DEF FNF(X) = SQR(ABS(X-1))
20 R=FNROOT(5,9,FNF(X))
30 FOR I = 1 TO 20

40 R=FNROOT(R,FNGUESS,FNF(X))
50 NEXT I
60 DISP "Root or minimum at";R
```

This is the user-defined function.

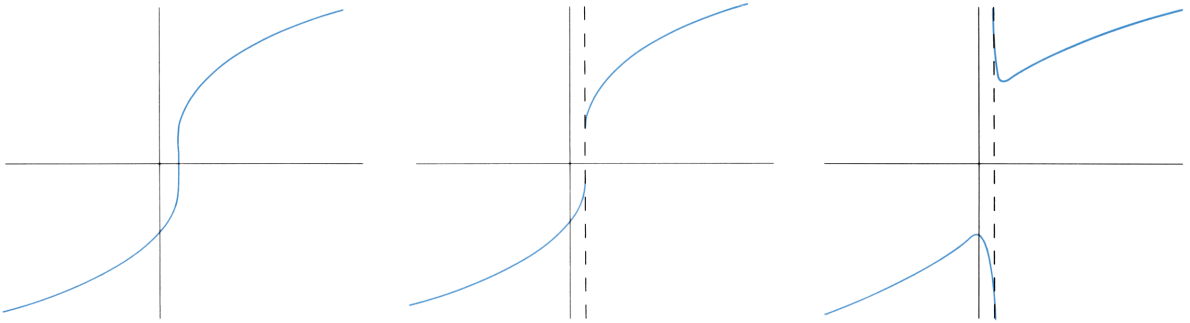
Tries to find a root.

Iterates 20 times to resolve the minimum to greater accuracy.

To execute this program, key it in and [RUN] it. The display will then show:

```
Root or minimum at    1
```

When |FNF(r)| is too large to consider r a root, another possibility is that FNF(r) and FNF(FNGUESS) have different signs. In this case it would appear that there is a root between, because for the function to change signs it should cross the x-axis. Typically, when FNROOT finds two guesses on opposite sides of the x-axis, it only stops after it has resolved them to two consecutive machine numbers. In this case there is no machine representable number between r and FNGUESS. Thus, the behavior of the function cannot be determined between r and FNGUESS. To interpret such results, you should be aware of these situations.



In case 1, r and FNGUESS are the best approximations to the root which are representable on the machine. Case 2 looks exactly the same to FNROOT, but there is no root—there is a jump discontinuity instead. In case 3 there is a pole, which can look like a root if a guess on each side of the pole is found. FNROOT returns information in FNGUESS and the root to help you isolate situations where convergence is to a pole or jump discontinuity.

Decreasing Execution Time

The exponent range of your HP-75 is ±499. This allows for sensitive observation of the behavior of a function, even very close to a root. FNROOT takes advantage of this dynamic range by not accepting a guess as a root until the function value underflows, is zero, or two consecutive machine representable numbers that bracket a root are found. The cost of this precision is that, occasionally, it may take quite a while to obtain such precision. If this high degree of sensitivity is not required, then you may wish to set a smaller tolerance. For example, you may only need to know a place where the function is less than 1E-20. This is accomplished in your function subprogram by checking the value of the function before assigning it to the function variable and setting the function variable to zero if the computed value is smaller than the desired tolerance. For example, suppose you wanted to find any roots of $f(x) = x^4$, and $|f(x)| \leq 1E-32$ is acceptable as a root. Here is a program you can use.

```
10 DEF FNF(X)
20 F=X^4
30 IF F<=1E-32 THEN FNF=0 ELSE
FNF=F

40 END DEF
50 DISP FNROOT(2,3,FNF(X))
60 DISP FNF(RES)
```

Multiline function definition of $f(x) = x^4$.

Checks error tolerance and sets the function value accordingly.

Computes and displays the root.

Displays the function value at the root.

To execute this program, key it in, and press [RUN]. In the display will appear:

```
8.30442501969E-9
0
```

In this example, if this *tolerance technique* were not used, execution would last much longer. This is because the computed function will not underflow until x is very small, since the root is at zero and the distribution of machine-representable numbers is very dense close to zero. So FNROOT has a lot of guesses to try before finding one it can accept as a root.

An alternate approach to decreasing execution time is to translate the function so that the root is not so near zero, compute the root of the translated function, then translate the root back. This will decrease the time to find roots of certain functions with roots close to zero, but will generally decrease the accuracy of the roots found. Here is a sample program for $f(x) = x^4$.

```
10 DEF FNF(X) = (X-1)^4
20 R= FNROOT(3,4,FNF(X))
30 DISP R-1

40 DISP FNF(R)
```

This is x^4 translated by 1.

Computes the root.

Translates the root back and displays the root and function value.

Finally, there is a technique that may improve the speed and accuracy of FNROOT. Any equation is typically one of an infinite family of equivalent equations with the same roots. However, some may be easier to solve than others. For example, the two equations $f(x) = 0$ and $\exp(f(x)) - 1 = 0$ have the same real roots, but one is almost always easier to solve. When $f(x) = x^4 - 6x - 1$, the first is easier; but when $f(x) = \ln(x^4 - 6x - 1)$, then the second is easier. While FNROOT has been designed to provide accurate results for a wide range of problems, it is worthwhile to be aware of such possibilities.

Section 14

Numerical Integration

The keywords in this section enable you to evaluate the integral of a function between definite limits. Before you can calculate the integral of a function $f(x)$ you must write a user-defined function that calculates the values of $f(x)$. (For information about user-defined functions, refer to section 13 of the *HP-75 Owner's Manual.*)

You can then use the keyword INTEGRAL to calculate the integral of the user-defined function. You can use INTEGRAL anywhere within the program in which the user-defined function is defined except within the *definition* of the user-defined function.

The keywords IBOUND and IVALUE give you additional flexibility in the evaluation of the integrals. INTEGRAL, IBOUND, and IVALUE are numeric-valued, so they can be used alone or in combination with other functions and variables to form numeric expressions.

INTEGRAL

Definite Integral

INTEGRAL(*A*, *B*, *E*, FN*function name*(*X*))

where *A*, *B*, *E* are numeric expressions, FN*function name* is a user-defined numeric function, and *X* is a numeric variable.

Returns an approximation to the integral from *A* to *B* of FN*function name*. The relative error *E* (rounded to the range 1E−12 ≤ *E* ≤ 1) indicates the accuracy of FN*function name* and is used to calculate the acceptable error in the approximation to the integral.

INTEGRAL generates a sequence of increasingly accurate approximations to the definite integral. If three successive approximations are within the acceptable error of each other—the first is close to the second and the second is close to the third—they are considered to have converged and the third approximation is returned as the value of the definite integral. If a total of 16 approximations are calculated without converging, the sixteenth approximation is returned.

X is a *dummy* variable—its inclusion here doesn't affect the use of this variable name in any other context.

This keyword can be used only in a program.

IVALUE

Last Result of INTEGRAL

IVALUE

Returns the last approximation computed by the INTEGRAL keyword. If the [ATTN] key was pressed or the operation of INTEGRAL was otherwise interrupted, then IVALUE returns the value of the current approximation to the integral. Otherwise, IVALUE returns the same value that INTEGRAL last returned.

IVALUE retains its value (even if your HP-75 is turned off) until another INTEGRAL is computed.

IBOUND

Error Approximation for INTEGRAL

IBOUND

Returns the final error estimate for the definite integral most recently computed by INTEGRAL.

- A positive value for IBOUND means that the approximations converged.
- A negative value for IBOUND means that the approximations didn't converge completely, so that the value returned by INTEGRAL may not be within the acceptable error of the actual value.

Like IVALUE, IBOUND retains its value (even if the HP-75 is turned off) until another INTEGRAL is computed. Unlike IVALUE, the value of IBOUND has no relation to the current approximation to the integral if the operation of INTEGRAL is interrupted.

The operation of INTEGRAL and IBOUND can be described more precisely as follows.

1. Based on a relative error of E for the specified function, the computer calculates an error tolerance for the integral of the specified function. If $f(X)$ is the "true" function that FNF approximates, then choose E such that

$$E \geqslant \frac{|\text{FNF}\ (X) - f(X)|}{|\text{FNF}\ (X)|}$$

   for all X in the interval of integration. Your input for E is rounded to the range $1\text{E}{-}12 \leqslant E \leqslant 1$. For example, if FNF is derived from experimental data with N significant digits, let E equal 10^{-N}.
2. The computer calculates a sequence of approximations I_k to the integral of the specified function. The difference between successive approximations is compared to the error tolerance for the integral.
3. A value for the integral is returned when:
   - The approximations I_k have converged. Convergence is determined using J_k, defined as the kth approximation to the integral of $10^{(\text{int}\ (\log|\text{FNF}|))}$ over the same interval of integration. J_k represents the error inherent in the computation of I_k.

The approximations I_k are judged to have converged to I_n if

$$|I_k - I_{k-1}| \leq E\ J_k$$

for $k = n - 1$ and for $k = n$. The value of I_n is then returned by INTEGRAL; a positive value for the error estimate will be returned by IBOUND.

or when

- The computer has evaluated I_1 through I_{16} but the convergence criterion is still not met. I_{16} is then returned by INTEGRAL; a negative value for the error estimate will be returned IBOUND.

Examples

INTEGRAL, IBOUND, IVALUE

To find the integral from 0 to 1 of the function

$$f(x) = \exp\ (x^3 + 4x^2 + x + 1)$$

you can use the following program.

```
10 DEF FNF(X)=EXP(X^3+4*X^2+X+1)
20 INPUT E

30 DISP "Integrating; please wait"
40 X=INTEGRAL(0,1,E,FNF(W))
50 BEEP
60 DISP "The value of the integral is"; X
70 DISP "The approx. error is"
80 DISP IBOUND
```

The user-defined function FNF.

Gets the relative error we expect in FNF as compared with f.

Remember that W is a dummy variable.

After you key in the program, run it using the following keystrokes.

Input/Result

RUN

```
?
```

The prompt to enter the relative error of the function.

1E-5 RTN

Although our function is accurate to one part in 10^{12}, we can say that it is less accurate (in this case, one part in 10^5) so that the computation will finish more quickly.

```
Integrating; please wait
```

The integral will take about a minute to be computed.

```
The value of the integral is
104.291097528
The approx. error is
3.42880788095E-4
```

The value of the integral is 104.2911 $\pm 3.4 \times 10^{-4}$.

ivalue RTN

```
104.291097528
```

IVALUE gives the value of the last computed integral.

INTEGRAL, IBOUND

You can use INTEGRAL to compute the amount of heat required to heat one gram of gas at a constant volume from one temperature to another. The amount of heat needed, Q, is given by the formula

$$Q = \int_{T1}^{T2} C(T)\, dT\ ,$$

where $C(T)$ is the specific heat of the gas as a function of temperature, $T1$ is the starting temperature, and $T2$ is the final temperature.

If $C(T) = a + bT$, where a and b are experimentally determined to be $a = 1.023\text{E}-2$ and $b = 2.384\text{E}-2$ with four significant digits, then we can compute the relative error of $C(T)$ to be approximately 5E−4. The program below prompts you for the initial and final temperature in degrees Kelvin and then computes the heat needed to raise the temperature of the gas from the initial to the final temperature.

```
10 DEF FNC(T) = .01023+.02384*T

20 INPUT "Initial and final temp.s in degrees
   Kelvin?";T1,T2
30 DISP "Integrating"
40 Q=INTEGRAL(T1,T2,.0005,FNC(T))
50 DISP "The amount of heat needed
   is";Q;"+-";IBOUND
```

The user-defined function that calculates the specific heat.

Computes the integral.

Displays the answer and the approximate error.

To find the heat needed to raise the temperature from 300°K to 310°K, type in the program and use the following keystrokes.

Input/Result

RUN

```
Initial and final temp.s in degr
ees Kelvin?
```

300,310 RTN

```
Integrating
```

```
The amount of heat needed is
72.8143 +- .005
```

Additional Information

The INTEGRAL keyword has been designed to obtain accurate results rapidly for a wide range of problems. Without some help from the user, however, no numerical integration scheme can successfully integrate all functions representable by the computer. This section includes information about numerical integration in general, the algorithm used by INTEGRAL, and ways to handle more difficult problems.

Overview of Numerical Integration

Numerical integration schemes generally sample the function to be integrated at a number of points in the interval of integration. The calculated integral is simply a weighted average of the function's values at these sample points. Since a definite integral is really an average value of a function over an *infinite* number of points, numerical integration can produce accurate results only when the points sampled are truly representative of the function's behavior.

If the sample points are close together and the function does not change rapidly between two consecutive sample points, then the numerical integration will give reliable results. On the other hand, numerical integration will not produce good answers on a function whose values vary wildly over a domain that is small in comparison with the region of integration. Other errors that can affect the result of a numerical integration include the round-off errors typical of any floating point computation and errors in the procedure that computes the function to be integrated.

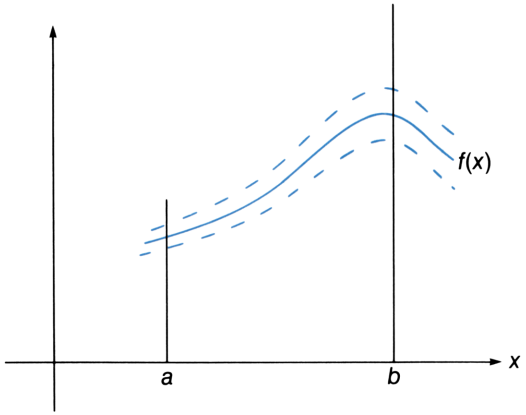
Handling Numerical Error

The INTEGRAL keyword requires specification of an error tolerance E for its operation. This error tolerance is taken to be the *relative error* of the user-defined function as compared with the "true"

function to be integrated. The error tolerance is used to define a ribbon around the user-defined function and the "true" function should then lie inside this ribbon. If the "true" function is $f(x)$ and the computed function is $FNF(x)$, then

$$FNF(x) - \text{Error } (x) \leqslant f(x) \leqslant FNF(x) + \text{Error } (x)$$

where Error (x) is half the width of the ribbon at x.



We can then conclude that

$$\int_a^b f(x)\,dx \approx \int_a^b FNF(x)\,dx \pm \int_a^b \text{Error } (x)\,dx$$

where the third integral is just half the area of the ribbon—that is, integrating the user-defined function instead of the actual function can introduce an error no greater than half of the area of the ribbon. INTEGRAL estimates this error while computing the integral; IBOUND gives you access to the estimate.

Choosing the Error Tolerance

The accuracy of the computed function depends on three factors:

- The accuracy of empirical constants in the function.
- The degree to which the function may accurately describe a physical situation.
- The round-off error introduced when the function is computed.

Functions like cos (x − sin$x$) are purely mathematical functions. This means that the functions contain no empirical constants, and neither the variables nor the limits of integration represent any actual physical quantities. For such functions you can specify as small an error tolerance as desired, provided that the function is calculated within that error tolerance (despite round-off) by the BASIC function. Of course, due to the trade-off between accuracy and computation time, you may choose not to specify the smallest possible error tolerance. Any specified error tolerance is rounded to the range [1E−12, 1].

When the integrand relates to an actual physical situation, there are additional considerations. In these cases, you must ask yourself whether the accuracy you would like in the computed integral is justified by the accuracy of the integrand. For example, if the function contains empirical constants which approximate the actual constants to three digits, then it may not make sense to specify an error tolerance smaller than 1E−3.

An equally important consideration, however, is that nearly every function relating to a physical situation is inherently inaccurate because it is only a mathematical model of an actual process or event. A mathematical model is typically an approximation that ignores the effects of factors judged to be insignificant in comparison with the factors in the model.

For example, the equation $s = s' - (.5)gt^2$, which gives the height s of a falling body when dropped from an initial height s', ignores the variation with altitude of g, the acceleration due to gravity. Mathematical descriptions of the physical world can provide results of only limited accuracy. If you calculated an integral with an accuracy greater than your model can support, then you would not be justified in using the calculated value to its full (apparent) accuracy. It makes sense to supply an error tolerance that reflects any inaccuracies in the function, or the INTEGRAL keyword will waste time computing to a level of accuracy that may be meaningless. Further, the value returned by IBOUND may not be significant.

If $f(x)$ is a function relating to a physical situation, its inaccuracy due to round-off is typically very small compared to the inaccuracy in modelling the situation. If $f(x)$ is a purely mathematical function, then its accuracy is limited only by round-off error. Precisely determining the relative error in the computation of such a function generally requires a complicated analysis. In practice, its effects are determined through experience rather than analysis.

Handling Difficult Integrals

Integrating on Subintervals. A function whose values change substantially with small changes in its argument will likely require many more points than one whose values change only slightly. This is because the behavior of the function must be adequately represented by the sampling. If a function is changing more rapidly in some subintervals of the interval of integration than in others, you can subdivide the interval and integrate the function separately on the smaller intervals. Then the integral over the whole interval is the sum of the integrals over all the subintervals, and the error of the integral is the sum of the errors of the integrals over the subintervals.

The algorithm used by INTEGRAL makes a reasonable decision during execution of how many points to sample, based on the behavior of the specified integrand on a particular interval. When the interval of integration is split up, each subinterval can be handled according to the function's behavior on that subinterval alone. This results in greater speed and precision.

For example, to integrate $f(x) = (x^2 + 1E{-}12)^{1/2}$ from $x = -3$ to $x = 5$ using an error tolerance of $1E{-}12$, it speeds up execution to subdivide the interval at $x = 0$, where $f(x)$ has a sharp bend in its graph. Because $f(x)$ is very smooth on the subintervals $(-3, 0)$ and $(0, 5)$, the integrals over these subintervals can be evaluated quickly.

$$\int_{-3}^{5} f(x)\ dx = \int_{-3}^{0} f(x)\ dx + \int_{0}^{5} f(x)\ dx$$

The following program computes this integral on the two subintervals and then combines the results.

```
10 DEF FNF(X) = SQR(X*X+.000000000001)
20 I=INTEGRAL(-3,0,.000000000001,FNF(X))
30 E=IBOUND
40 DISP "The value of the integral is"
50 DISP
   I+INTEGRAL(0,5,.000000000001,FNF(X))
60 DISP "The approximate error is
70 DISP E + IBOUND
```

We will use X*X rather than X^2 because X*X is more accurate. An analogous situation generally occurs for any *integer* power of a variable.

Integrate over the first subinterval.

Save the error to add in.

The sum of the first and second integrals.

Compute the relative error by adding the two errors together.

You can run this program by keying it in and then pressing RUN. The following will then appear in the display.

```
The value of the integral is
17
The approximate error is
5.95809523809E-12
```

When the interval is subdivided, INTEGRAL computes the answer in a few seconds. Without subdividing the interval, execution may take a long time.

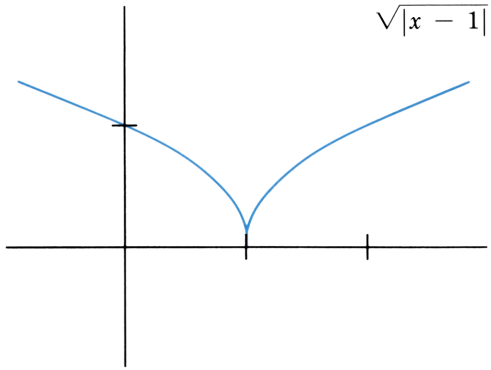
Subdividing the interval of integration is also useful for functions with a singularity in the interval. The singularity may consist of one or more points where the function is undefined or has a sharp corner point.

For example, the integral

$$\int_0^2 \frac{dx}{(x-1)^2} \text{ may be split into } \int_0^1 \frac{dx}{(x-1)^2} + \int_1^2 \frac{dx}{(x-1)^2}$$

to avoid evaluating the function at $x = 1$, where it is undefined. You can now integrate the function on each subinterval because $x = 1$ is an endpoint of each subinterval, and INTEGRAL does not sample at an endpoint.

Similarly, the function $\sqrt{|x-1|}$ has a sharp corner point at $x = 1$.

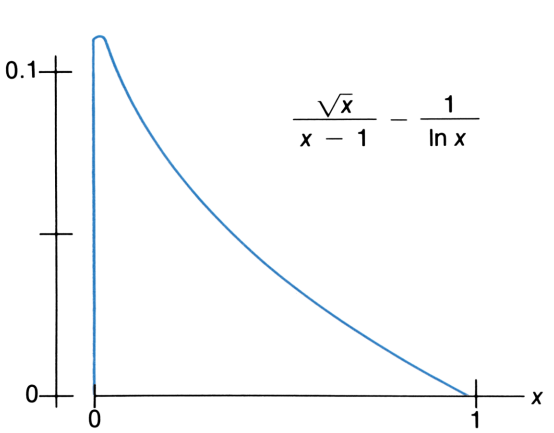


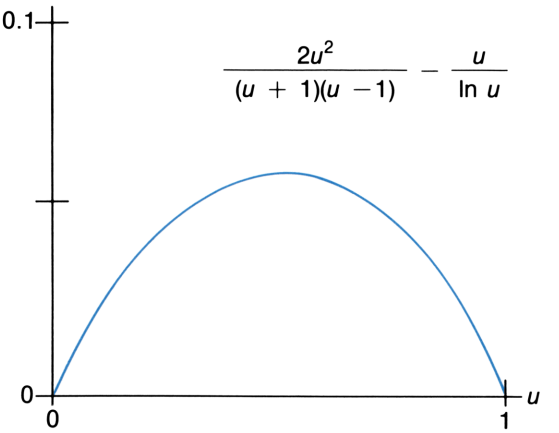
Suppose you need to integrate this function from 0 to 2. You can increase the speed and accuracy of the computation by integrating separately on the subintervals (0, 1) and (1, 2), because the function is smooth on each of these subintervals.

Transformation of Variables. A second method of handling difficult integrands is by transforming the variable. When the variable in a definite integral is transformed, the resulting definite integral may be easier to compute numerically. Consider the integral

$$\int_0^1 \left(\frac{\sqrt{x}}{x-1} - \frac{1}{\ln x} \right) dx.$$

The derivative of the integrand approaches infinity as x approaches 0, as shown on the left below. The substitution $x = u^2$ stretches out the x-axis and causes the function to be better behaved, as shown on the right.



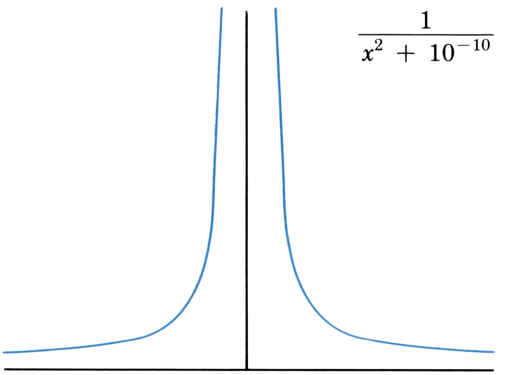


You can now evaluate the integral that results from this substitution:

$$\int_0^1 \left(\frac{2u^2}{(u+1)\,(u-1)} - \frac{u}{\ln u} \right) du.$$

(Do not replace $(u + 1)\,(u - 1)$ with $u^2 - 1$; as u approaches 1, $u^2 - 1$ loses half of its digits to round-off, yielding a final result that is too large.)

As a second case requiring substitution, consider the following function. Its graph has a long tail stretching out much farther than the main body (where most of the area is).



Although a very thin tail may be truncated without greatly degrading accuracy, this function has too wide a tail to ignore when calculating

$$\int_{-t}^{t} \frac{dx}{x^2 + 10^{-10}}$$

if t is large. In general, the compressing substitution $x = b \tan u$ maps the entire real line into $(-\pi/2, \pi/2)$ and maps subsets of the real line into subsets of $(-\pi/2, \pi/2)$. For $b = 1\text{E}{-5}$ the substitution becomes $x = 1\text{E}{-5} \tan u$ and the integral becomes

$$10^5 \int_{\tan^{-1}(-t/b)}^{\tan^{-1}(t/b)} du,$$

which is readily computed for very large t.

This compressing substitution is also a standard way to deal with infinite intervals. For example,

$$\int_{-\infty}^{\infty} \frac{dx}{x^2 + 10^{-10}} = 10^5 \int_{-\pi/2}^{\pi/2} du \ .$$

In some cases the tail can be chopped off. Consider the function $\exp(-x^2)$. This functions underflows (that is, gives a result of zero in machine arithmetic) for $x > 34$. Thus,

$$\int_0^{\infty} e^{-x^2} dx \cong \int_0^{34} e^{-x^2} dx \ .$$

Therefore, when dealing with infinite integrals you can cut off the tail if it is insignificant, but you should use a compressing substitution if it is not.

About the Algorithm

The Math Pac uses a Romberg method for accumulating the value of an integral. Several refinements make it more effective. Instead of equally spaced samples, which can introduce a kind of resonance or aliasing that produces misleading results when the integrand is periodic, INTEGRAL uses samples that are spaced nonuniformly. Their spacing can be demonstrated by substituting

$$x = \frac{3}{2} u - \frac{1}{2} u^3 \text{ into } \int_a^b f(x)dx$$

and then spacing u uniformly. Besides suppressing resonance, the substitution has two additional benefits. First, no sample need be taken from either endpoint of the interval of integration unless the interval is so small that points in the interval round to an endpoint. As a result, an integral like

$$\int_0^1 \frac{\sin x}{x} dx$$

will not be interrupted by division by zero at an endpoint. Second, INTEGRAL can integrate functions whose slope is infinite at an endpoint. Such functions are encountered when calculating the area enclosed by a smooth closed curve like $x^2 + f^2(x) = R$.

In addition, INTEGRAL uses extended precision. Internally, sums are accumulated in 16-digit numbers. This allows thousands of samples to be accumulated, if necessary, without losing any more significance to round-off than is lost within your function subroutine.

During the computation, INTEGRAL generates a sequence of iterates that are increasingly accurate estimates of the actual value of the integral. It also estimates the width of the error ribbon at each iterate. INTEGRAL stops only after three successive iterates are within the computed error of each other or after 16 iterations have been performed without this criterion being met.

In the latter case the function will have been sampled at 65,535 points. The value returned by IBOUND will be the negative of the computed error to signify that the returned value of the INTEGRAL is likely not within the error tolerance of the actual value. Typically, you should then split up the interval of integration into smaller subintervals and integrate the function over each of the subintervals. The integral over the original interval will then be the sum of the integrals over the subintervals. In this way, up to 65,535 points can be sampled on each subinterval, thus computing the integral to greater precision.

In summary, INTEGRAL has been designed to return reliable results rapidly and in a convenient, easy-to-use fashion. The above theoretical considerations discuss problems with numerical integration in general. The INTEGRAL keyword is capable of handling even difficult integrals with their aid.

Section 15

Finite Fourier Transform

The finite Fourier transform is a key step in solving many problems in mathematics, physics, and engineering, such as problems in signal processing and differential equations.

Given a set of N complex data points $Z_0, Z_1, \ldots, Z_{N-1}$, the finite Fourier transform will return another set of N complex values $W_0, W_1, \ldots, W_{N-1}$, such that for $k = 0, 1, \ldots, N-1$,

$$Z_k = \sum_{j=0}^{N-1} W_j \left(\cos \frac{2\pi kj}{N} + i \sin \frac{2\pi kj}{N} \right) .$$

The W's then represent the complex amplitudes of the various frequency components of the signal represented by the data points. The values for the W's are given by the formula

$$W_j = 1/N \sum_{k=0}^{N-1} Z_k \left(\cos \frac{-2\pi kj}{N} + i \sin \frac{-2\pi kj}{N} \right) .$$

This formula holds for any number of data points. The Math Pac uses the Cooley-Tukey algorithm and the internal language of the HP-75 to achieve excellent speed and accuracy in the calculation of the finite Fourier transform. This requires, however, that N be an integral power of 2; for example, 2, 4, 8, 16, 32, 64, and 128 are all acceptable values for the number of complex data points.

To use the finite Fourier transform, store your complex data points $Z_0, \ldots, Z_{N-1}$ as successive rows of an $N \times 2$ array with Z_0 in the first row, Z_1 in the second row, and so on. Store these values in the usual complex form: real parts in the first column, imaginary parts in the second column. The results of the finite Fourier transform $W_0, \ldots, W_{N-1}$ will be returned with the complex values stored in successive rows of an $N \times 2$ array—the same form as the data points.

The number of data points you can use is limited only by the amount of available memory and by the requirement that the number of data points be a non-negative integral power of 2.

FOUR — Finite Fourier Transform

MAT **W** = FOUR (**Z**)

where **Z** is a $N \times 2$ matrix, N a non-negative integer power of 2, and **W** is a matrix.

Redimensions **W** to be exactly the same size as **Z** and assigns to **W** the complex values of the finite Fourier transform of the data points represented by **Z**.

Example

Input/Result

```
clear vars RTN
dim z(15,1),w(15,1) RTN
mat z=con RTN
```

Z and **W** are 16 × 2 arrays.

Z now represents the complex column vector, each of whose values is $1 + 1i$. **Z** could be the sampling of a complex constant function, for example.

```
mat w=four(z) RTN
mat disp w;
```

```
 1   1
 0   0
 0   0
 0   0
 0   0
 0   0
 0   0
 0   0
 0   0
 0   0
 0   0
 0   0
 0   0
 0   0
 0   0
 0   0
```

This is the finite Fourier transform of the constant function. The only frequency that occurs is the zero frequency—all rows but the first are zero.

Additional Information

Relation Between the Finite and Continuous Fourier Transform

The finite Fourier transform is most often used as an approximation to the continuous (infinite) Fourier transform. To understand in what sense it is an approximation, and to understand the effects of various choices to be made in using this approximation, it is most useful to have the direct relationship between the continuous and finite transforms.

If $Z(x)$ is a complex valued function, its continuous Fourier transform is defined to be

$$W(f) = \int_{-\infty}^{\infty} Z(x) \exp(-2\pi i f x)\, dx.$$

If we have a set of N complex data points $Z_0, Z_1, \ldots, Z_{N-1}$ given by sampling the function Z at N equally spaced points

$$Z_k = Z(x_0 + k\Delta x) \text{ for } k = 0, 1, \ldots, N-1,$$

and then find the finite Fourier transform $W_0, W_1, \ldots, W_{N-1}$ of this data set, we can relate these values to the values of the continuous Fourier transform $W(f)$ as follows. For $k = 0, 1, \ldots, N-1$,

$$W_k = (r/N)\ \tilde{W}(k/\Delta x) \text{ where } r = \exp(-2\pi i x_0).$$

$\tilde{W}$ is a "smeared" version of the true continuous Fourier transform W. To get $\tilde{W}$ from W, you must average W in two important but very different ways. The first type of averaging that occurs can be described by defining a new function $A(f)$ intermediate between W and $\tilde{W}$.

$$A(f) = \sum_{k=-\infty}^{\infty} W(f + k/\Delta x)$$

This says that the value of A at a point f is equal to the sum of the values of W at all points that are integer multiples of the limiting frequency $1/\Delta x$ away from f. In particular, if W consists of a small bump centered at the origin, then A will consist of an infinite sequence of bumps spaced $1/\Delta x$ units apart. This is the aspect of the finite Fourier transform that gives rise to *aliasing*: any frequency that occurs in W (that is, W has a non-zero value there) will give rise to a non-zero value for A (and also $\tilde{W}$) somewhere in the interval $[0, 1/\Delta x]$ *no matter what the original frequency was*. For this reason, you should choose Δx small enough so that $1/\Delta x$ is larger than the distance between the largest and smallest f's that you suspect will occur in W. Since most functions occuring in actual situations (and *all* real-valued functions) have continuous Fourier transforms which are roughly symmetric about the origin, if a frequency f_0 occurs in W, it is likely that $-f_0$ also occurs in W. For the finite Fourier transform to contain both frequencies without aliasing, $1/\Delta x$ must be larger than $2f_0$. If we define the largest frequency occuring in W as Δf, we can express the no-aliasing requirement as $\Delta f \Delta x < \frac{1}{2}$.

The second type of averaging that occurs when going between W and $\tilde{W}$ is much more local in nature than the first. It results in a loss of frequency resolution in $\tilde{W}$ as compared with W; more precisely,

$$\tilde{W}(f) = (N\Delta x) \int_{-\infty}^{\infty} \text{sinc}\ (gN\Delta x)\ A(f-g)\ dg$$

$$\text{where sinc } (a) = \begin{cases} 1 \text{ if } a = 0, \\ \dfrac{\sin(\pi a)}{\pi a} \text{ otherwise.} \end{cases}$$

Since sinc $(gN\Delta x)$ consists primarily of a bump with width inversely proportional to $N\Delta x$, $\tilde{W}$ is more blurred (compared to W) for smaller values of $N\Delta x$. This is not a serious problem unless W has a large value at a frequency that is not a multiple of the fundamental frequency $N/\Delta x$. In this case, the "side lobes" of the sinc function become evident in $\tilde{W}$. This can be reduced somewhat by multiplying the data values Z_k by a smoothing function $G(k)$ before taking the finite Fourier transform. This results in

an averaging function that has smaller side lobes than the sinc function. One example of such a function is the Hanning function $G(k) = (½)(1 - \cos(2\pi k/N))$.

Inverse Finite Fourier Transform

Many applications of the finite Fourier transform involve taking the transform of a set of data points, operating on the transformed values (for example, increasing or decreasing the amplitudes), and then retransforming the data using the inverse Fourier transform defined by

$$Z_k = \sum_{j=0}^{N-1} W_j \left(\cos \frac{2\pi kj}{N} + i \sin \frac{2\pi kj}{N} \right) .$$

You can also use the FOUR keyword to compute the inverse finite Fourier transform in a simple way. If **W** is an $N \times 2$ array for which you wish to take the inverse Fourier transform:

1. Multiply **W** on the right by the 2×2 array $\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$ using real array multiplication.
2. Take the finite Fourier transform of the result.
3. Multiply the result array of the finite Fourier transform by the 2×2 array given in step 1.
4. Scalar-multiply the result by N. This will produce the inverse finite Fourier transform of the original array.

This application of the finite Fourier transform and the procedure for obtaining the inverse finite Fourier transform are illustrated in the example below.

Example

Suppose we want to find the steady state solution $Z(x)$ of the inhomogeneous differential equation

$$Z''(x) + 3Z'(x) + 12Z(x) = P(x)$$

where $P(x)$ is a function for which we have sampling data. If we denote the (continuous) Fourier transform of any function Q by $\tilde{Q}$, by taking the Fourier transform of the above equation we arrive at

$$-f^2\tilde{Z}(f) + 3if\tilde{Z}(f) + 12\tilde{Z}(f) = \tilde{P}(f).$$

Solving this equation algebraically, we obtain

$$\tilde{Z}(f) = \frac{\tilde{P}(f)}{(-f^2 + 12) + 3if} .$$

If we can get a good approximation to $\tilde{P}$, we can easily calculate the right side of this equation. From this result we can obtain the solution to the original equation by taking the inverse Fourier transform.

For simplicity, we will assume that the equation has been scaled so that $P(x)$ has unit period, and that the highest frequency component of P is (approximately) 30 times the fundamental frequency. Sampling P 64 times in one period will then suffice to avoid aliasing.

Rather than prompt the user for 64 complex data points representing the sampling of P, the program below uses a relatively simple function for P, although you could use values from any other source equally well.

```
10 OPTION BASE 1
20 DIM P(64,2),Q(64,2),Z(64,2)
30 DIM C(2,2), T(2), S(2)
40 C(1,1)=1@C(2,2)=-1@C(1,2),C(2,1)=0
50 DISP "Working; please wait"
60 FOR I=1 TO 64
70 P(I,1)=6000*COS(3*PI*I/32)
   *SIN(7.5*PI*I/32)*COS(5.5*PI*I/32)
80 P(I,2)=4000*COS(13*PI*I/32) +
   3500*SIN(11*PI*I/32)
90 NEXT I
100 MAT Q=FOUR(P)
110 FOR F=-31 TO 32
120 J=MOD(F,64)+1
130 T(1)=-F^2+12@T(2)=3*F
140 S(1)=Q(J,1)@S(2)=Q(J,2)
150 MAT S=CDIV(S,T)
```

P will contain the data points representing the sampling of P. **Q** will represent $\tilde{P}$ and eventually $\tilde{P}/(-f^2 + 12 + 3if)$. **Z** will represent the solution to the differential equation.

C will be used in the inverse transformation; **T** and **S** are dimensioned to be complex scalars for use in the complex division.

C is now the matrix $\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$.

This is the sampling routine that assigns to **P** the values of the complex-valued function represented by the right-hand sides of lines 70 and 80, sampled at 64 equally spaced points.

Q now represents $\tilde{P}$.

F represents the frequency variable and spans the full range of frequencies, positive and negative, that we expect to occur in $\tilde{P}$.

J represents the number of the row in the **Q** array where the amplitude of the frequency F is stored.

T will be the denominator and

S the numerator in the complex fraction.

```
160 Q(J,1)=S(1)@Q(J,2)=S(2)
170 NEXT F
180 MAT Q=Q*C

190 MAT Z=FOUR(Q)
200 MAT Z=Z*C
210 MAT Z=(64)*Z
220 DISP "THE VALUES ARE"
230 MAT DISP USING "X,DDDD.D" ; Z
```

Q now represents $\tilde{P}/(-f^2 + 3if + 12)$.

This is the procedure that assigns to **Z** the values of the inverse finite Fourier transform of **Q**.

The values displayed will represent the complex values of the steady state solution of the differential equation, sampled at 64 equally spaced points in one period.

Fourier Sine/Cosine Series

There is another transform closely related to the finite Fourier transform that is applicable when the data points Z_k are purely real (that is, their imaginary parts are equal to zero). This is the Fourier series transformation, which takes a set of $2N$ (real) data points $Z_0, Z_1, \ldots, Z_{2N-1}$ and returns a set of $N + 1$ real values $A_0, A_1, \ldots, A_N, B_1, \ldots, B_N$ with the property that

$$Z_k = \frac{A_0}{2} + \sum_{j=1}^{N} A_j \cos \frac{2\pi jk}{2N} + B_j \sin \frac{2\pi jk}{2N} .$$

If $W_0, W_1, \ldots, W_{2N-1}$ are the complex values of the finite Fourier transform of the real data points $Z_0, \ldots, Z_{2N-1}$, then the Fourier series values are given by

$$A_j = 2\mathrm{Re}(W_j) \qquad \text{for } j = 0, \ldots, N,$$

$$B_j = -2\mathrm{Im}(W_j) \qquad \text{for } j = 1, \ldots, N.$$

Appendix A

Owner's Information

CAUTIONS

Do not place fingers, tools, or other objects into the plug-in ports. Damage to plug-in module contacts and the computer's internal circuitry may result.

Turn off the computer (press [SHIFT] [ATTN]) before installing or removing a plug-in module.

If a module jams when inserted into a port, it may be upside down. Attempting to force it further may result in damage to the computer or the module.

Handle the plug-in modules very carefully while they are out of the computer. Do not insert any objects in the module connecter socket. Always keep a blank module in the computer's port when a module is not installed. Failure to observe these cautions may result in damage to the module or the computer.

Limited One-Year Warranty

What We Will Do

The Math Pac is warranted by Hewlett-Packard against defects in materials and workmanship affecting electronic and mechanical performance, but not software content, for one year from the date of original purchase. If you sell your unit or give it as a gift, the warranty is transferred to the new owner and remains in effect for the original one-year period. During the warranty period, we will repair or, at our option, replace at no charge a product that proves to be defective, provided you return the product, shipping prepaid, to a Hewlett-Packard service center.

What Is Not Covered

This warranty does not apply if the product has been damaged by accident or misuse or as the result of service or modification by other than an authorized Hewlett-Packard service center.

No other express warranty is given. The repair or replacement of a product is your exclusive remedy. **ANY OTHER IMPLIED WARRANTY OF MERCHANTABILITY OR FITNESS IS LIMITED TO THE ONE-YEAR DURATION OF THIS WRITTEN WARRANTY.** Some states, provinces,

or countries don't allow limitations on how long an implied warranty lasts, so the above limitation may not apply to you. **IN NO EVENT SHALL HEWLETT-PACKARD COMPANY BE LIABLE FOR CONSEQUENTIAL DAMAGES.** Some states, provinces, or countries do not allow the exclusion or limitation of incidental or consequential damages, so the above limitation may not apply to you.

This warranty gives you specific legal rights, and you may also have other rights which may vary from state to state, province to province, or country to country.

Warranty for Consumer Transactions in the United Kingdom

This warranty shall not apply to consumer transactions and shall not affect the statutory rights of a consumer. In relation to such transactions, the rights and obligations of Seller and Buyer shall be determined by statute.

Obligation To Make Changes

Products are sold on the basis of specifications applicable at the time of manufacture. Hewlett-Packard shall have no obligation to modify or update products once sold.

Warranty Information

If you have any questions concerning this warranty, please contact an authorized Hewlett-Packard dealer or a Hewlett-Packard sales and service office. Should you be unable to contact them, please contact:

- In the United States:

Hewlett-Packard Company
Portable Computer Division
1000 N.E. Circle Blvd.
Corvallis, OR 97330
Telephone: (503) 758-1010
Toll-Free Number: (800) 547-3400
(except in Oregon, Hawaii, and Alaska)

- In Europe:

Hewlett-Packard S.A.
150, route du Nant-d'Avril
P.O. Box CH-1217 Meyrin 2
Geneva
Switzerland
Telephone: (022) 83 81 11

Note: Do not send products to this address for repair.

- In other countries:

Hewlett-Packard Intercontinental
3495 Deer Creek Rd.
Palo Alto, CA 94304
U.S.A.
Telephone: (415) 857-1501

Note: Do not send products to this address for repair.

Service

Service Centers

Hewlett-Packard maintains service centers in most major countries throughout the world. You may have your product repaired at a Hewlett-Packard service center any time it needs service, whether the unit is under warranty or not. There is a charge for repairs after the one-year warranty period.

Hewlett-Packard computer products normally are repaired and reshipped within five (5) working days of receipt at any service center. This is an average time and could vary depending on the time of year and work load at the service center. The total time you are without you product will depend largely on the shipping time.

Obtaining Repair Service in the United States

The Hewlett-Packard United States Service Center for battery-powered computational devices is located in Corvallis, Oregon:

Hewlett-Packard Company
Service Department
P.O. Box 999
Corvallis, OR 97339, U.S.A.

or

1030 N.E. Circle Blvd.
Corvallis, OR 97330, U.S.A.

Telephone: (503) 757-2000

Obtaining Repair Service in Europe

Service centers are maintained at the following locations. For countries not listed, contact the dealer where you purchased your unit.

AUSTRIA
HEWLETT-PACKARD Ges.m.b.H.
Kleinrechner-Service
Wagramerstrasse-Lieblgasse 1
A-1220 Wien (Vienna)
Telephone: (0222) 23 65 11

BELGIUM
HEWLETT-PACKARD BELGIUM SA/NV
Woluwedal 100
B-1200 Brussels
Telephone: (02) 762 32 00

DENMARK
HEWLETT-PACKARD A/S
Datavej 52
DK-3460 Birkerod (Copenhagen)
Telephone: (02) 81 66 40

EASTERN EUROPE
Refer to the address listed under Austria.

FINLAND
HEWLETT-PACKARD OY
Revontulentie 7
SF-02100 Espoo 10 (Helsinki)
Telephone: (90) 455 02 11

FRANCE
HEWLETT-PACKARD FRANCE
Division Informatique Personnelle
S.A.V. Calculateurs de Poche
F-91947 Les Ulis Cedex
Telephone: (6) 907 78 25

GERMANY
HEWLETT-PACKARD GmbH
Kleinrechner-Service
Vertriebszentrale
Berner Strasse 117
Postfach 560 140
D-6000 Frankfurt 56
Telephone: (611) 50041

ITALY
HEWLETT-PACKARD ITALIANA S.P.A.
Casella postale 3645 (Milano)
Via G. Di Vittorio, 9
I-20063 Cernusco Sul Naviglio (Milan)
Telephone: (2) 90 36 91

NETHERLANDS
HEWLETT-PACKARD NEDERLAND B.V.
Van Heuven Goedhartlaan 121
N-1181 KK Amstelveen (Amsterdam)
P.O. Box 667
Telephone: (020) 472021

NORWAY
HEWLETT-PACKARD NORGE A/S
P.O. Box 34
Oesterndalen 18
N-1345 Oesteraas (Oslo)
Telephone: (2) 17 11 80

SPAIN
HEWLETT-PACKARD ESPANOLA S.A.
Calle Jerez 3
E-Madrid 16
Telephone: (1) 458 2600

SWEDEN
HEWLETT-PACKARD SVERIGE AB
Skalholtsgatan 9, Kista
Box 19
S-163 93 Spanga (Stockholm)
Telephone: (08) 750 20 00

SWITZERLAND
HEWLETT-PACKARD (SCHWEIZ) AG
Kleinrechner-Service
Allmend 2
CH-8967 Widen
Telephone: (057) 31 21 11

UNITED KINGDOM
HEWLETT-PACKARD Ltd
King Street Lane
GB-Winnersh, Wokingham
Berkshire RG11 5AR
Telephone: (0734) 784 774

International Service Information

Not all Hewlett-Packard service centers offer service for all models of HP products. However, if you bought your product from an authorized Hewlett-Packard dealer, you can be sure that service is available in the country where you bought it.

If you happen to be outside of the country where you bought your unit, you can contact the local Hewlett-Packard service center to see if service is available for it. If service is unavailable, please ship the unit to the address listed above under Obtaining Repair Service in the United States. A list of service centers for other countries can be obtained by writing to that address.

All shipping, reimportation arrangements, and customs costs are your responsibility.

Service Repair Charge

There is a standard repair charge for out-of-warranty repairs. The repair charges include all labor and materials. In the United States, the full charge is subject to the customer's local sales tax.

Computer products damaged by accident or misuse are not covered by the fixed repair charge. In these cases, repair charges will be individually determined based on time and materials.

Service Warranty

Any out-of-warranty repairs are warranted against defects in materials and workmanship for a period of 90 days from date of service.

Shipping Instructions

Should your product require service, return it with the following items:

- A completed Service Card, including a description of the problem.
- A sales receipt or other documentary proof of purchase date if the one-year warranty has not expired.

The product, the Service Card, a brief description of the problem, and (if required) the proof of purchase date should be packaged in adequate protective packaging to prevent in-transit damage. Such damage is not covered by the one-year limited warranty; Hewlett-Packard suggests that you insure the shipment to the service center. The packaged product should be shipped to the nearest Hewlett-Packard designated collection point or service center. Contact your dealer for assistance.

Whether the product is under warranty or not, it is your responsibility to pay shipping charges for delivery to the Hewlett-Packard service center.

After warranty repairs are completed, the service center returns the product with postage prepaid. On out-of-warranty repairs in the United States and some other countries, the product is returned C.O.D. (covering shipping costs and the service charge).

Further Information

Service contracts are not available. Computer products circuitry and design are proprietary to Hewlett-Packard, and service manuals are not available to customers. Should other problems or questions arise regarding repairs, please call your nearest Hewlett-Packard service center.

Technical Assistance

The keystroke procedures and program material in this manual are supplied with the assumption that the user has a working knowledge of the concepts and terminology used. Hewlett-Packard's technical support is limited to explanations of operating procedures used in the manual and verification of answers given in the examples. Should you need further assistance, you may write to:

Hewlett-Packard Company
Portable Computer Division
Customer Support
1000 N.E. Circle Blvd.
Corvallis, OR 97330

Appendix B

Memory Requirements

The Math Pac reserves 52 bytes of read/write memory for its own uses. In addition to this 52-byte "overhead" and the memory required to dimension the arrays and variables you use with the keywords (described in appendix D of the *HP-75 Owner's Manual*), certain Math Pac operations use additional memory during their operation. After the operation is completed, the memory is again available for your use. The tables below provide you with the memory requirements for those keywords whose operation requires additional temporary memory.

Item	Memory Required During Operation
BSTR$	BSTR$(M,N) requires: • One byte if $M = 0$. • INT(LOGA(M,N)) + 1 bytes otherwise. This is the number of digits needed to represent M (decimal) in base N.
*	Requires additional memory only if an operand array is used for the result array. If **A** is an $M \times N$ matrix and **B** is $R \times S$ matrix: • MAT A=A*A requires $T \cdot M^2$ bytes. • MAT A=A*B requires $T \cdot M \cdot S$ bytes. • MAT A=B*A requires $T \cdot R \cdot N$ bytes. where $T = \begin{cases} 8 \text{ if } \mathbf{A} \text{ is REAL.} \\ 4 \text{ if } \mathbf{A} \text{ is SHORT.} \\ 3 \text{ if } \mathbf{A} \text{ is INTEGER.} \end{cases}$
DET	If **A** is an $N \times N$ matrix, DET(A) requires $2N(4N + 1)$ bytes.
INV	If **A** is an $N \times N$ matrix, MAT B=INV(A) requires: • $4N$ bytes if **B** is REAL. • $4N(2N + 1)$ bytes if **B** is SHORT or INTEGER.
SYS	If **A** is an $N \times N$ matrix and **B** is an $N \times P$ matrix, MAT C=SYS(A,B) requires $4N(2N + 4P + 1)$ bytes.

Item	Memory Required During Operation
LUFACT	If **A** is an $N \times N$ matrix, `MAT B=LUFACT(A)` requires: • $2N$ bytes if **B** is `REAL`. • $2N$ ($4N + 1$) if **B** is `SHORT` or `INTEGER`.
CMMULT	Same as `*`.
CDET	If **A** is an $N \times 2N$ array, `MAT Z=CDET(A)` requires $16N^2$ bytes.
CINV	If **A** is an $N \times 2N$ array, `MAT B=CINV(A)` requires $8N$ ($4N + 1$) bytes.
CSYS	If **A** is an $N \times 2N$ array and **B** is an $N \times 2P$ array, `MAT C=CSYS(A,B)` requires $8N$ ($4N + 4P + 1$) bytes.
PROOT	If **P** is an array with $N + 1$ elements representing a polynomial of degree N, `MAT R=PROOT(P)` requires $22N + 267$ bytes.
FOUR	If **A** is an $N \times 2$ array, `MAT B=FOUR(A)` requires: • No additional memory if **B** is `REAL`. • $16N$ bytes if **B** is `SHORT` or `INTEGER`.
INTEGRAL	`INTEGRAL(A,B,E,FNF(X))` requires 333 bytes.
FNROOT	`FNROOT(A,B,FNF(X))` requires 87 bytes.

Appendix C

Error Conditions

Number	Error Message and Condition
1	num too small • |*Result*| < 1E−499.
2	num too large • |*Result*| > 9.99999999999E499. • MAT U=INV(V), MAT U=CINV(V), MAT U=LUFACT(V), MAT U=SYS(V,W), MAT U=CSYS(V,W), DET(V), MAT U=CDET(V). The matrix **V** is singular (that is, its determinant is zero) and the *LU* decomposition of **V** requires division of a non-zero number by zero. This does not always indicate that the results of the operation are invalid. In particular, the results of DET and CDET will be valid. The results of the other operations should be checked when this error occurs.
11	arg out of range • ACOSH(X): $X < 1$. • ATANH(X): $\|X\| > 1$. • LOGA(X,B): $B = 1$. • MAT Z=CDIV(W,V), MAT Z=CRECP(V): **V** = (0, 0). • MAT Z=CPOWER(W,V): **W** = (0, 0) and Re(**V**) ≤ 0. • BSTR$(M,N): $M \geq$ 999,999,999,999.5. • BVAL(B$,N): (*value*) > 999,999,999,999.
12	LOG(0) • LOG2(X): $X = 0$. • LOGA(X,B): $X = 0$ or $B = 0$. • MAT Z=CLOG(W): **W** = (0, 0).

Number	Error Message and Condition
13	`LOG(neg number)` • `LOG2(X)`: $X < 0$. • `LOGA(X,B)`: $X < 0$ or $B < 0$.
89	`bad parameter` • `BVAL(B$,N)`, `BSTR$(M,N)`: rounded integer value of *N* not equal to 2, 8, or 16. • `BVAL(B$,N)`: *B$* not a valid number in base *N*. • `BSTR$(M,N)`: $M < 0$. • `MAT A=IDN(`*redimensioning subscript(s)*`)`, `MAT A=CON(`*redimensioning subscript(s)*`)`, `MAT A=ZER(`*redimensioning subscript(s)*`)`, `REDIM A(`*redimensioning subscript(s)*`)`: rounded integer value of one or both subscripts is less than the option base in effect. • `UBND(A,N)`, `LBND(A,N)`: rounded integer value of *N* not equal to 1 or 2. • `MAT R=CROOT(P,N)`: rounded integer value of *N* not positive.
201	`result dimension` • `MAT A=CON(i,j)`, `MAT A=ZER(i,j)`, `MAT A=IDN(i,i)`, `REDIM A(i,j)`: **A** singly subscripted. • `MAT A=CON(i)`, `MAT A=ZER(i)`, `REDIM A(i)`: **A** doubly subscripted. • `MAT A=`*operation (operand array(s))*: number of subscripts of **A** not the same as the number of subscripts required for the result of the operation.
202	`result size` • `REDIM A(`*redimensioning subscript(s)*`)`, `MAT A=CON(`*redimensioning subscript(s)*`)`, `MAT A=ZER(`*redimensioning subscript(s)*`)`, `MAT A=IDN(`*redimensioning subscript(s)*`)`: number of elements in the redimensioned array greater than the total number of elements given to it in a dimensioning statement. • `MAT A=`*operation*`(`*operand array(s)*`)`: total number of elements in **A** (as given in its original dimensioning statement) less than the number of elements needed to store the results of the operation.

Number	Error Message and Condition
203	`conformability` • `MAT A=B+C`, `MAT A=B-C`: **B** and **C** not *conformable for addition* (the number of rows are unequal or the number of columns are unequal). • `MAT A=B*C`, `MAT X=SYS(B,C)`: **B** and **C** not *conformable for multiplication* (the number of columns of **B** is not equal to the number of rows of **C**). • `DOT(A,B)`: number of elements of **A** not equal to the number of elements of **B**. • `MAT R=CMMULT(A,B)`, `MAT X=CSYS(A,B)`: number of columns of **A** not equal to twice the number of rows of **B**.
204	`not square` • `DET(A)`, `MAT X=SYS(A,B)`, `MAT B=INV(A)`, `MAT B=LUFACT(A)`, `MAT A=IDN`: number of rows of **A** not equal to the number of columns. • `MAT A=IDN(i,j)`: $i \neq j$. • `MAT R=CINV(A)`, `MAT R=CSYS(A,B)`, `MAT A=CIDN`, `MAT B=CDET(A)`: number of columns of **A** not equal to twice the number of rows.
205	`not vector` • `MAT X=CROSS(A,B)`, `DOT(A,B)`: **A** or **B** not singly subscripted.
206	`not 3-vector` • `MAT X=CROSS(A,B)`: **A** or **B** not three dimensional.
207	`operand dimension` • `MAT A=IDN(i)`: only one redimensioning subscript specified. • `DET(B)`, `MAT A=CDET(B)`, `MAT X=SYS(B,C)`, `MAT A=LUFACT(B)`, `MAT A=TRN(B)`, `MAT A=CTRN(B)`, `MAT A=CINV(B)`, `MAT A=INV(B)`, `MAT A=FOUR(B)`: **B** not doubly subscripted. • `MAT R=CMMULT(A,B)`, `MAT R=CSYS(A,B)`: **A** or **B** not doubly subscripted.

Number	Error Message and Condition
208	`operand size` • `MAT R=`*complex function*`(Z)`: **Z** not a complex scalar. • `MAT R=`*complex function*`(Z,W)`: **Z** or **W** not a complex scalar. • `MAT R=CROOT(Z,N)`: **Z** not a complex scalar. • `MAT R=PROOT(P)`: **P** contains exactly one element (and so represents a polynomial of degree zero). • `MAT R=CDET(A)`, `MAT R=CINV(A)`, `MAT R=CTRN(A)`: **A** doesn't have an even number of columns. • `MAT R=CMMULT(A,B)`, `MAT X=CSYS(A,B)`: **A** or **B** doesn't have an even number of columns. • `MAT A=FOUR(B)`: **B** is not an *N* × 2 array with *N* a non-negative integer power of 2.
209	`PROOT failure` • `PROOT` cannot find a root of the specified polynomial.
210	`nesting error` • `FNROOT(A,B,FNF(X))`: user-defined function `FNF` uses the `FNROOT` keyword in its definition. • `INTEGRAL(A,B,E,FNF(X))`: user-defined function `FNF` uses the `INTEGRAL` keyword in its definition.

Keyword Index

Keyword	Page	Description
RNORM	40	Infinity norm (row norm) of an array.
ROUND	14	Round.
RSUM	35	Row sum of an array.
SINH	13	Hyperbolic sine.
SUM	40	Sum of array elements.
SYS	54	System solution.
TANH	13	Hyperbolic tangent.
TRN	35	Transpose of a matrix.
TRUNCATE	15	Truncate.
UBND	42	Upper bound of array subscripts.
ZER	25	Zero array.
=	25	Simple assignment.
=()	25	Numeric-expression assignment.
=-	33	Array negation.
+	33	Array addition.
-	34	Array subtraction.
*	34	Array multiplication.
()*	34	Scalar-array multiplication.

Portable Computer Division
1000 N.E. Circle Blvd., Corvallis, OR 97330, U.S.A.

00075-90106 English

Printed in U.S.A. 4/83