Jim Hawkins Dartmouth BASIC Language Interpreter

V 4.0 2022

Updated 2/6/2022

PREFACE

This interpreter is strongly based, in style, to the original Dartmouth BASIC, designed by two professors at Dartmouth College, John G. Kemeny, released in 1964 at Dartmouth College and is intended as a quick and handy interpretive (write and run) language. It is extended from the original BASIC, but not so far extended as to be unrecognizable as the original BASIC as many other Good products given the BASIC name, such as Visual BASIC. The BASIC name is an acronym that stands for **Beginners' All-purpose Symbolic Instruction Code**. It consists of simple, line numbered statement or command strings of the form:

Line Number	Statement	Expression……………………….

Line Number Statement Expression or, in particular:

100 if a > b then 400
 language that can be handy for writing one's own utilities. The original set of commands and statements are implemented so an older BASIC program can be loaded. Being aware that most original BASIC programs were in all uppercase, a special load commands, **lcold** filename has been provided. So that uppercase programs as converted to lowercase as they are loaded. If it is saved, the file will be lower case. The first example at the end is a Celsius to Fahrenheit conversion table generator. It both displays the table on the monitor and writes the table into a file, which can be printed.

As for the "holy war" arguments of line numbers vs no line numbers, gotos or no gotos, long variable names; think of it this way: there are plenty of BASIC programmers who prefer the old line numbers, tolerate the short tags variable names, still like gotos can be compared to people who still like LPs over CDs or digital music files, people who prefer old typewriters to modern word processing software, old cars, especially from the 50s and 60s, are no less valid than people who want more, updated BASICs. This version does provide structured "ifs", "while" loops in addition to "for" loops, so you could write a program without using gotos. When I program in the C or C++ languages, it almost never occurs to me to use a goto. It comes naturally. But, if people still like to write "spaghetti" like code, I have no criticism as to whether it's some sort of sin. To each his/her own. I confess that this interpreter has exactly ONE goto in over 10,000 line of code. I was a convenient and quick solution to situation where trying to use a "while" was a little too messy for me and the time I would have to spend to restructure it.

```
                 *
                        *
                           *
                          *
                     *
             *
        *
       *
          *
                 *
```

OR:

```
================|
======================|
=========================|
========================|
===================|
============|
=======|
======|
=========|
================|
======================|
=========================|
========================|
===================|
============|
=======|
```

It can serve as a quick algebraic expression computer as in:

print 3.14*33/2*sin(30)*(33+22+11)

and so on. Notice that it can be typed in without a line number for immediate calculation. I wrote the expression evaluator. The precedence of operators are guided by a simple operator precedence table.

I wrote the original version to work on my DOS computer, many years ago, but the saved source code just sat there as I moved into Windows. It has been a challenging job just to port it as a console application written in the C programming language as C has evolved to meet an ANSI standard with more strict rules. In the process of testing it, I found many “bugs” and fatal errors causing program crashes. Most of them are fixed, but I know there are more. Some functions were completely rewritten in the last four months.

I plan to offer it as open source, for others to play with or expand, but not until I post some algorithms for others to use, free of any license or charge.

BEFORE GETTING STARTED

Upper case BASIC programs may be loaded using the *lcold* command. Caps are converted to lower case as it is loaded into the interpreter.

***File paths* use backslashes and disk drives, to denote file paths such as**

\ folder1\folder2\file

OR

c:\folder\file

***Desktop icon* It makes it easier to create a desktop icon by right clicking on the BASIC executable and choosing Send to > desktop icon and/or pin to Start menu**

***Improving console appearance* When you first start this program, you will undoubtedly get a tiny black console window. Click on the upper left corner icon, select and chose “size” as 24 or 28. You can even chose Font > Bold font. After making these settings, they will be “remembered” and the app will start with these settings.**

1. Conventions

This Manual: All things enclosed in [ ] are optional

expr Any algebraic expression which could be a constant, variable, math function or a combination of the aforesaid, separated by arithmetic operators as in:

```
a+b*3.14*(4.4+c2*sin(b+s)) +a(2,2)
```
See "variables' and "math functions" below.

Note that whenever an array is specified with dimension(s) containing variables, those variables must have been assigned a value through *let* or *read* commands.

Operators: + - * / % or ^ for addition, subtraction, multiplication, division, modulo, or exponentiation in. order of lowest to highest precedence. + and - have the sameprecedence and *, /, and % have the same precedence. Parenthesis () around expressions forces the contents to be higher precedence than all parts of the expression outside, those parenthesis. Note also that when the '-' is used as a unary it maintains its low precedence, hence the expression -2^2 yields 4 instead of 4. In all cases a good rule of thumb to ensure precedence is to enclose the part of high precedence in parenthesis, thereby (-2)^2 yields 4.

Operator		Meaning	X if not implemented
( )		parenthesis	
^		exponentiation	
*		Multiplication	
/		division	
%		modulo	
+		Addition	
-		subtraction	
>		Greater than	
>=		greater than or equal to	
=		equal to	
<>		not equal to	
<=		less than or equal to	
<		less thans	
not	!	Binary compliment	X
and	&&	Binary and	
or	||		
xor			X
exp			X
imp			X

,

Relationals: <, >, =, <=, > =, <>, *or, and,* *||, &&* for less than, greater than, equality, less or equal, .greater than or equal, not equal, logical *or* and logical *and*. Source Program Name The source program name is suffixed by a **.bas**

Statement A basic statement consists of a line number. (integer value between 1 and 65534 (on 32 bit machines) followed by a command; space and operand which follows the syntax governed by the command as in:

```
100 print "Hello World"
```

A statement can be typed without a line number in which case it will execute immediately. This is true for all commands, but doesn't make sense for a command such as *fo*r. Immediate execution is handy for diagnostic purposes such as *print* x. for some commands.

Strings Sequences of ASCII characters, enclosed by double quote characters or may be represented by a string variable x$, y3$, p$(j, k, l).

String Variables String variable names are followed by a dollar sign '$' as in a$, x3$, etc. They currently wile work for Let command, Read-Data statements, and print, as in:

```
2000 let a$ = “Hello “
2100 let x2$ = “World”
2200 print a$;x2$
```
or they may be added together:
```
2200 print a$ + x2$
```

Matrix Variables are named the same as regular variables, but are distinguished by the commands that operate on them. However, the variable names, as seen in the operations, are those of corresponding array variables. Hence, variable ‘a’ corresponds to a(row, column). They must be dimensioned with the dimension statement, but only the name of the array is used in matrix operations.

Another example:

```
2000 let a$ = "Jim "
2050 let b$ = "Hawkins"
2055 read x$,y$,z$
2100 data a$,b$,"is a programmer."
2200 print x$;y$;z$
```

Note that the data statement mixes 2, pre-defined string variables and a literal string.

Variables Regular variable names can be either lower case or upper case alpha (a-z) or (A-Z) with or without an integer subscript (0-9). The lower case and upper case can co-exist in a program and represent different variables. That is, ‘A’ and ‘a’ are two different variables. Arrays have the same name convention as regular variables and take the form: varname (expr1, expr2, expr3. . . .expr 10) where expr1 - expr 10 are the dimension attributes of the array and can take the form of any legal expression. Array variables are first allocated with the dim command. In the case of array names, the upper case names are not separate, but name the same variable. That is, A(20) is the same as a(20). This may be changed in the future.

Arrays can have another array as one of its values!

```
2000 DIM A(10),B(10)
2200 LET B(2) = 9
2300 LET B(1) = 8
2400 LET A(B(1)) = 22
2500 LET A(B(2)) = 33
2600 PRINT A(B(1));" ";A(B(2))
2700 DIM W(10),X(10),Y(10),Z(10,10,10)
3000 LET W(1) = 4
3100 LET X(1) = 5
```

```
3150 LET Y(1) = 6
3200 LET Z(W(1),X(1),Y(1)) = 340
3300 PRINT Z(W(1),X(1),Y(1))
3400 DIM P(10)
3500 DIM Q(10,10)
3600 LET Q(5,5) = 9
3650 REM Multiple dimension arrays nested inside other arrays illegal at this time
3700 LET P(Q(5,5)) = 22
3800 PRINT P(Q(5,5))
```

COMMANDS AND STATEMENTS

ascii [flag]	**Where flag = d, o, x or a** Prints out ASCII value table. Added for my convenience. Not a normal part of BASIC. Flag d (default) decimal, o is octal, x is hex and a is all three.
auto [start line#] [,increment]	Turns on auto line number and increment] Defaults to 10,10 Terminate with a decimal point or **period '.'**
bye or **q**	Exit the interpreter. The 'q' command will give you the chance to confirm that you want to quit. Bye will just quit.
call <expression> "<overlay name>",<line number>	Call overlay and execute it. Allows programs to be segmented into overlays, which are initially not loaded. It loads the overlay file or program, if not already loaded, then executes i. Overlays must end in return. First line needs to be a rem <overlay name> Overlay must be in quotes. A string variable can be used, instead. When compared to a regular for loop of 500,000,000 iterations, the *call* is only 5% slower roughly. The loop around *call* is about 37 seconds and the ordinary loop is 35 seconds.

```
4700 call "square_f",10000
4800 rem


10000 rem square_f
10100 let y = 0
10200 let u = 0
10300 let s = 2
10400 let u = sum((1/h)*sin(h*a),h,o,1,n)
10800 let y = 10*u+15
.
.
```

Commented [J1]:

11000 return	
loadcall <expression> “<overlay name>”,<line number>	Same as call, except, previous overlay(s) are not cleared, allowing combining of overlays.
com [mon]	Preserve variables for subsequent run. Issue of the run command otherwise de-allocates all variables. NOT IMPLEMENTED
con [line#]	Continue normal execution from single step mode. See *step* command
data (expr), (expr), (expr)	The *data* statement is a string ,of defined constants or expressions referred to by the "read" statement. Unlike most BASIC interpreters, the data is stored only in the form of text strings which allows the read statement to evaluate expressions as well as constants.
Types of variables, my be mixed string and numeric 1000 read w, a$, x, b$ 2000 data 3.3,”Laurel, Stan”,4.33,”Hardy, Oliver” Quotes may be left off of strings if there are no commas or spaces.	
del [ete] lownum [,highnum]	Delete line-number specified if only lownum given. Delete all lines between lownum and highnum if both are specified. See the undo command. If the lownum doesn’t exist, it starts the deletion of the next higher line number.

Commented [J2]:

dim <variable name1>(expr1,expr2....,expr10), <variable name2>(expr1 expr2........,expr10),.......	Allocate space and define the dimensional characteristics of subscripted variable. Keep in mind that the memory allocated is the dimensions multiplied together along with information tables used to form the array. Say, an array is x(10,10), the memory used will be 10X10 = 100 along with the header information. The "size" or "mem" commands can tell you how much memory is used and how much is available. The *expunge* command will free up variable space taken up but arrays.
dim <string variable name>$(d1,d2,d3)	Dimension variable array up to three dimensions. Naming convention is the same as for numeric variables, except with a $ added, as in: **dim x1$(10,10)** A dim statement can have multiple arrays and types separated by commas as in: **String array and numerical array arguments may be mixed.** 1000 dim x(10,10),a$(5,5),y(5,1), etc.
end	Define logical end of program. Causes termination of current run. The same as "stop", except that stop displays the line number where it stopped, whereas "end" just stops the program. *End* is more desirable when you are writing output data to a file and you don't want interpreter messages to appear in the output file.
expunge	Force all variable space, including subscripted variables to be freed. Or deallocate used variable space.
files	Displays the currently open file or loaded BASIC program file, (if any) and any files open for read, read, write or append. The highest number of files allowed to be open in any mode or combination of modes is 8. read, write or append. The highest number of files allowed to be open in any mode or combination of modes is 8.

free[up] <array variable name>	Frees up individual array chunks previously allocated by the dim statement..
gosub line# or variable with line number	*Goto* a subroutine, resume from following statement after *return* encountered.
goto line# or variable with line number	Force execution to continue starting at the line# specified.
If (expr1) relational (expr2) then line#	Redirect program flow to line# if expr 1 is related to expr2 by the specified relational. The then in the then statement can be optionally replaced with goto or gosub

Other forms of
 	
if
	if (expr1) relational (expr2) then var = (expr) (form 1)
	if (expr1) relational (expr2) [(boolian) (expr3) relational (expr4)] then [line number]
	if-then-else-endif When no line number is given, the “structured” if is assumed.

input [#filedes] ["prompt message";] 1var1[,var2,var3.....]	Input assigns the number values typed at the input prompt, to the variables or array variables specified in the command line such as: 1000 input a, b, c(2,2,2). An optional prompt message can be added, informing the user what data is expected. Note that if fewer numbers are entered at the input prompt '?' than the operand or command line variables, the input command will display a message telling the user that more values are specified or type the letter 'q' to quit. It will continue to ask until the number of comma separated variables like, 20, 3.14, 98.6 is equal to or more than the number of variable given in the input argument. If extra entries are made, they will simply be ignored. that the user fix the input file and the BASIC program will rewind the file and stop. Values can also be read from a file as designated by #filedes. Message prompts are disabled when reading from a file.
Input a$, b$, x$, p$(1,2,3)	input can also assign string values to string variables or arrays, string values
let variable = <expr>	Assign the value of numeric or string expression to an appropriate type of variable.
list or (l) [lownum [,highnum]]	List the text in working storage. If lownum is given then only that number is listed., if lownum and highnum are specified, then a listing is displayed between the given statement numbers.
load [program name]	Same as the *old* command, except working storage is not cleared. This allows adding other code to be added, providing that there are no conflicting line numbers. (unless that is what you want)

mov startnum, endnum, newnum [,increm]	The *mov* command causes the lines, beginning with startnum and ending with endnum to be moved (ie. renumbered) to the line beginning with newnum and incremented by increm. The default value for incrern is 10. All references to the moved lines are updated. The' user is responsible to see that line numbers associated with moved lines do not conflict with existing lines which will cause loss of program text. *mov* is similar to *renum* (see below) except that only the specified lines are renumbered.
n	The *n* command lists 20 lines at a time. To continue listing, type <ENTER>. To exit, type ‘q’.
new	Clear program working storage for new program to be typed.
old [program name]	open [program name] Clear user space and load program. If old is typed with no argument it will prompt the user for a program name if not defined or load the last defined program name.
lcold	loads and converts a upper case BASIC program to all lower.

on (expr) goto line#, line#	Is a selective *goto* with multiple line number targets. The target branched-to depends on the value of expr which is truncated. Control is passed to the first line# specified after *goto* if the value of the expression is 1. Control passes to the second line# if the value is 2, the third if 3 and so on.
on (expr) gosub line#, line#.......	Same action as *on-goto*, except action is that of *gosub*.
on (expr) let var=num2, [var=num1, var=num3,…….]	Chose which variable assignment
on (expr) call <“overlay name”>,<start line number><“overlay name”>,<start line number>	On call selects one of the overlays to be loaded and executed. Overlay must be started with a rem statement with the overlay name, The name must match the filename without the .bas extension.
optrace [1]	Debug tool Prints a trace of the commands and statements executer. Typing optrace with no argument turns it off. The number can be any number. It’s only purpose is to turn it on.

x

pause	Causes execution to be suspended until a "newline", or "return" is typed. This is useful for programs which need to be continuously in run, but need to allow a time for user action i.e. unit insertion. **Typing the letter 'q'** while paused, will stop execution or quit running the BASIC program.
print [#fildes] (expr's, quoted strings or tab operators)	The print statement is a limited format display statement in which expressions are evaluated and displayed along with quoted literals. The **tab**(expr) operator causes the print head to move to the absolute column position computed by expr provided the current head position is smaller. The specifiers must be separated by one or more commas or semicolons. **hex**(expr) prints the hex converted value of expr, but, unlike **oct**(expr), hex() is not a math function, but only a print function, because hex numbers in this interpreter cannot be handled by the (double) values of this interpreter. Note that the results of base conversions are REPRESENTATIONS of the converted print in floating point.

Print command Print Zones (fields separated by commas)

```
|    zone 1    |     zone 2    |     zone 3    |    zone 4     |    zone 5    |
 ----------------------------------------------------------------------------
|    1-15      |     16-30     |      31-45    |    46-60      |    61-72     |
```

The statement print a,b,c,d,e will cause the values of the first five variables
to be printed, one number per zone. If more than five variables appearin a PRINT
statement, the sixth value is printed on the next line in zone 1, the seventh, zone 2,
and so on.

Text example:
100 print "FIRST COLUMN","THE SECOND COLUMN","THIRD COLUMN"

```
 |    zone 1    |     zone 2    |     zone 3    |    zone 4     |    zone 5    |
 ----------------------------------------------------------------------------
 |    1-15      |     16-30     |     31-45     |    46-60      |    61-72     |
  FIRST COLUMN   THE SECOND COLUMN              THIRD COLUMN   FOURTH COLUMN
```

print using [#<file number>]<"format string"> [,exprl,expr2 .... expr10]	Where the format string or string variable representing the format allows you to specify how the text and numbers are supposed to be printed as follows: print using "##.# degrees Fahrenheit converts to ##.# degrees Celsius.",98.6,37.0 Adding $$ in front of the format string causes dollar signs to be printed in a column before the number. Adding ** in front of the format string causes the blank fill to be replaced with asterisks '*' Adding $** in front of the format string enables both the dollar signs and the asterisk filling.
Adding $$ in front of the format string causes dollar signs to be printed in a column before the number. Adding ** in front of the format string causes the blank fill to be replaced with asterisks '*' Adding $** in front of the format string enables both the dollar signs and the asterisk filling. Examples: print using "Total: $$#####.## dollars",33.45 Total: $ 33.45 print using "**#####.##",33.45 Total: ***33.45 print using "$**#####.## dollars",33.45 Total: $***33.45	
	randomize Causes **rnd()** f to start at an "unpredictable" value. The seed represents the number of seconds that have elapsed since midnight Jan 1, 1970, which is continually changing.
read varl,var2,var3, var4$, var5$	The read statement causes data from the *data* statement to be assigned to each variable in the list from the constant. Strings and variables need to match the variable type being read.
rem	The remark statement causes no operation in BASIC but may be followed by any string of characters for the purpose of commenting a program. They are of paramount importance in making a program readable by human beings and are, therefore, strongly recommended. As in:

rem //////// This is a comment /////////
// does the same thing as *rem*, but added to make comments less cumbersome looking as in:
// ///////////// This is a comment ///////////
Note that a space must follow the double slash, otherwise the interpreter will see the whole field as a BASIC command.

ren[um] startnum [, increm]	The renumber command causes the statement numbers and all references to them (such as ifs gotos, gosubs, etc.) to be renumbered starting at startnum and incremented by increm. If startnum and/or increm are omitted, the default values are 10 and 10 respectively.
restore [line number]	Restores the data pointer to the first field of the first data. If a line number is specified, the data pointer is restored to the data line at that particular line number.
return	Return from subroutine called by gosub statement.
run [program name]	Run basic program specified. If no argument is given, run attempts to execute whatever is currently in working storage.
s[ub] line#/old-string/new-string/	Substitute in line line# the new-string for the old-string. The last delimiter is optional, unless new-string is null in which case it is desired that oldstring be removed.
sing [line#] or **step [line#]**	Enter the single step mode starting at the line# specified or at the first line of the program if no line# is specified. In single step mode an instruction is executed and then the prompt " is displayed. At this time the user may enter any command (i.e. print) or hit the "return" key to execute the next instruction. See the con command. Entering the letter 'q' while exit single step mode.

size or mem	Causes amount of working storage used and remaining. It also displays the number of assigned regular variables and the amount of space occupied by all arrays. Variables do not occupy the same space as the program and array variables are allocated and stored in “heap” storage as allocated by the C language calloc() call. Calloc() fills the data space with zeros.
stop	Stop execution of program, giving the line# of the stop command. The end command does the same thing but without the message.
save [program name]	Save the contents of working storage specified by program name. If no program name is given, it referenced file-name is used. If no file name was referenced, the user is prompted for a name. The default load and save folder is the one that this executable resides in. Otherwise, a complete path or one relative to the folder that this executable resides in as in: *save* filename, *old* or *load* filename or save c:\<folder>\filename
time	displays time DDD MMM, DD, YYYY HH:MM AM/PM such as: Sat Oct 28, 2022 12:40 PM
undo	Undo last s command or single line deletion
v	prints the current version number and date. Mainly used by me to remind me of what version I have loaded, when I am comparing to versions.

STRUCTURED FLOW COMMANDS

Use of these commands and statements could eliminate the use of gotos.	
for- next	Cause code enclosed by this combination to be executed under the conditions specified in the *for* statement as in: *for <variable> = <expression1> to <expression2>.* The step directive, tells the *for* loop how much to step for each iteration as in: **for x = 1 to 360 step 2**.
if-then-else-endif or **if-then-endif**	Allows more control The file the test in if directs to two different possibilities, condition met or condition not met. Gotos or on gotos are not allows within a structured if-then-else scope.
while	A new addition to this interpreter is the *while-endwhile* loop. Either numerical or string expressions may be used. In the case of numerical expressions, all of the relational operators may be used. With strings, only >, <, <> and = are available. A logic operator such as "or", or "and" may be used to include two expressions. You may also use && or ||. **NOTE: An infinite loop may be gracefully stopped by typing CTRL-C. CTRL-BREAK** terminates the interpreter**.**
break	can be used to break out a loop like for-next or while-endwhile before the final condition is met
continue	forces continuation of for loop like for-next or while-endwhile before loop condition is met

MATRIX COMMANDS, FUNCTIONS AND OPERATORS

MATRIX COMMANDS	
mat print [#n]<matrix variable>[,][;]	prints a matrix in row, column form. If the matrix name is followed by a semicolon, the matrix is displayed closely spaced. If it is followed by a comma, the matrix is displayed in larger spacing. With no delimiter, it is printed somewhere in between. [#n] is an optional output file slot.
mat read <mat variable name>	Can fill an entire matrix from a data line in one, simple statement.
100 dim a(3,3) 200 mat read a 300 data 4,3,6,4,3,7,5,3,7	
mat input <mat variable name>	Allows input of individual matrix members. Implemented for one-at-a-time entry instead of multiple entry by commas. Input prompt includes array member being assigned.
MATRIX ALGEBRA	
mat <mat variable> = <mat variable 1> + < mat variable 2>	**Matrix addition**. Assigns the sum of the two right matrix variables to the left matrix variable. All matrices must be the same dimensions.
mat <matrix var> = <mat var 1> - < mat var 2>	**Matrix subtraction**. Assigns the difference of the two right matrix variables to the left matrix variable. All matrices must be the same dimensions.
mat <mat var> = <matrix var 1> * < mat var 2>	**Matrix multiplication**. Assigns the product of the two right matrix variables to the left matrix variable. The row dimension of the first right matrix variable must be the same as the column dimension of the second right matrix variable. Swapping the two right matrices causes different results. That is A*B != B*A

mat <mat var> = <matrix var 1> cross < mat var 2>	**Matrix cross product**
mat <mat var> = <matrix var 1> dot < mat var 2>	**Returns dot product of two matrices**
mat <left matrix variable> = (<scaler>)* <right matrix variable 1>	Matrix multiplied by a scaler number or variable. The scaler can be an algebraic equation, but without parentheses in the equation. As in: mat a = (2)*b or mat a = (6/2 + 2).

MATRIX MATH FUNCTIONS	
mat i = inv(a)	Create an inverted version of matrix a on matrix i. Works for matrices from 2X2 to 8X8!
mat a = trn(b)	Create a transposed version of matrix b in matrix a
mat c = con(a,b)	Create a 1's filled matrix c of dimension a x b, where a and b are variables
mat c = zer(a,b)	Create a zero filled matrix c of dimensions a x b, where a and b are variables
mat I = idn(n)	Create an nXn identity matrix.

```
100 dim a(3,3)
200 mat read a
300 data 4,3,6,4,3,7,5,3,7
or
300 data 4,3,6
400 data 4,3,7
500 data 5,3,7
```

Commented [J3]:

```
1000 dim x(2,4),a(2,2)
1100 let a = 22
1200 let b = 33
1300 let a(1,2) = 3.14
1400 mat  input x
1500 mat  print x

% run
x(1,1) ? 1
x(1,2) ? a(1,2)
x(1,3) ? 3
x(1,4) ? a
x(2,1) ? b
x(2,2) ? 22
x(2,3) ? 33
x(2,4) ? 44

   1     3.14    3    22
  33    22       33   44
```

COMPLEX NUMBERS

Complex numbers live in single dimension arrays. It contains 2 values, first is the real number portion and the second is the imaginary number. So, if you want to allocate or "create" a complex number, you need only use the dim command. A complex number with the name of x would be defined by dim x(1,2), which is 1 row and 2 columns. The first column x(1,1) represents the real portion and the second column represents the imaginary portion. So. X(1,1)=3 and x(1,2) = 4 is 3 + 4i.

COMPLEX NUMBER COMMANDS	
complex print [#n]<complex variable>[,][;]	prints a complex number in a+bi form #n is an optional file slot if desired with an open file for output.
complex read <variable name>	Not yet implemented
complex input <variable>[,<variable>,<variable>..]	Requests input of real and imaginary values of M<variable name>
COMPLEX NUMBER ALGEBRA	
complex <variable> = < variable 1> + < variable 2>	**Sum**
complex <var> = < var 1> - < var 2>	**Difference**
complex <var> = < var 1> * < var 2>	**Product**
complex <var> = < var 1> / < var 2>	**Quotient**
complex <left matrix variable> = (<scaler>)* <right matrix variable 1>	Complex number multiplied by a scaler number or variable.
COMPLEX FUNCTIONS	

complex p=polar(c)	Converts complex number ‘c’ to polar form in ‘p’. Polar variable contains two array elements. Magnitude and angle.

USER DEFINED FUNCTIONS
def fna(<variable1>[,<variable2>,<variable3>….])) = <equation containing the input variable(s)>

The first two letters must always be “fn” followed by any letter in the alphabet from a-z. That gives you 26 possible user defined functions. The alphabetic variable passed to the function in the definitions must be the same letter variable used throughout the expression. Why? Because in the interpreter code, ‘x’ and the value of ‘x’ will be stored with its associated value. The equation is solved by the expression evaluator using the same variable name. It’s equivalent to doing a *let ‘x’ = 10*, then solving the expression as a function of ‘x’. The variables used by the user defined function are like local variables, private to the definition, using separate variable table for storage. This makes it possible to set a normal variable that is not disturbed by the function defined variables. The equation or formula itself is stored in dynamically allocated memory. The function is internally flagged as “defined.” User defined functions should be among the first lines of the program, like dimension statements -or- at least before they are used.

```
2000 def fnx(x) = x ^2 + 2* x - 20
2100 def fnx(q) = sin(2*q) + cos(45+q)

run
20632.14
```

Usage: Once defined, the function can be used in the BASIC program in the same way that all other mathematical functions are used. Functions must be defined before (lower line number) their use. A particular function, like fna(), may be redefined at a later (higher line number) by the same function. The old one is deleted and memory freed. A defined function can have any number of arguments.

Nesting of defined functions is also possible and the same variable name may be reused as each instance of the function has its own private variable as in:

```
1000 def fna(x) = 2*x
1100 def fnb(y) = 2*y
1200 def fnc(z) = 2*fnb(2*fna(z))
1300 let a = fnc(5)
1400 print a
```

```
run
80.0
```

fna() returns 5, which is passed into fnb(), multiplying it by 3, yielding 15.

Fourier series defined in fnf to 6 odd harmonics then asterisk plot of square wave

```
1000 rem // Solve and plot fourier series summation out to 6th odd harmonic
1100 rem // for square wave using user defined function
1200 def fnf(x) = sin(x)+sin(3*x)/3+sin(5*x)/5+sin(7*x)/7+sin(9*x)/9+sin(11*x)/11+sin(13*x)/13
1300 for a = 0 to 720 step 40
1400 let f = 10+10*fnf(a)
1500 gosub 1800
1600 next
1700 end
1800 for p = 0 to f
1900 print " ";
2000 next
2100 print "*"
2200 return
```

FILE COMMANDS	
The file commands: append, *openin*, and *openout* are followed by one or more file-names separated by commas. Files are assigned to designators (integer values between 1 and 8 inclusive) in the order that they are open. All commands such as print and input which refer to a file use the designator number preceded by a character to refer to that file. 100 print #1"hello world" or 100 input #3a (x,y)	
append file1,file2 .... file4l	If file exists open for output cause new data to be appended. If file does not exist, the named file is created.
openin “file1 or string variable”[,”flle2” ....” file4”]	Open file for input.,. File must exist. Arguments can either be literal quoted string or string variable assigned by *let* or *read* commands.
openout “file1 or string variable”[,”flle2” ....” file4”]	Create, named file(s) and open for output. If named files exist, the old. data is destroyed.
closef #fildes	Close file associated with file designator.
closeall	Close all files input and output.
files	Displays the currently open file or loaded BASIC program file, (if any) and any files open for read, read, write or append. The highest number of files allowed to be open in any mode or combination of modes is 8.

-

STRING FUNCTIONS	
fnum[ber]$(<format string>, <number>)	Format number according to format string pattern. This can be used in the *print* command giving it the flexibility to mix in formatting like the *print using* command. Works well in zone spaced expressions.

Use of fnum$ in print to generate formatted columns in zones.

```
1000 print "first column","second column","third column"
1100 print "Larry ";fnum$("##.##",33.3),"Moe ";fnum$("###.##",444.4),"Curly ";44.4
1200 print "      ";fnum$("##.##",33.3),"    ";fnum$("###.##",4.4),"      ";4.4
1300 let x$ = "##.##"
1400 let y$ = "###.##"
1500 print "Larry ";fnum$(x$,33.3),"Moe ";fnum$(y$,444.4),"Curly ";44.4
```

```
first column   second column  third column
Larry 33.30    Moe 444.40     Curly 44.4
      33.30          4.40           4.4
Larry 33.30    Moe 444.40     Curly 44.4
```

left$(<string literal>, n)	return first n characters of the given string
right$(<string literal>, n)	return last n characters of the given string
mid$(<string>,p,n) or **ext[ract]$**	extract substring from string at position ‘p’ with ‘n’ characters
str$(<numeric expression>)	converts numeric expression to string
string$(n, <char>)	repeat char ‘n’ times
loc$(<string>)	change all uppercase to lowercase
upc$(<string>)	change all lowercase to uppercase
chr$(<single character>)	convert <int> convert integer to ASCII character
time$([n])	Returns Day Date and Time of day. Format: DAY MON DD HH:MM:SS YYYYwhere n is optional: n = 0, 1, 2

time$

where n = 0 or no arg DAY MON DD HH:MM:SS YYYY (23 Hour)
n = 1   DAY MON DD, YYYY HH:MM (24 Hour)
n = 2   DAY MON DD, YYYY HH:MM AM/PM

MATH FUNCTIONS	
abs(expr)	Absolute value.
asc{“<char>”)	Return numeric value of ASCII character
atn(expr)	Arc-tangent.
asin(expr)	Arc sin
acos(expr)	Arc cosine
cos(expr)	Cosine.
exp(expr)	Natural exponential.
fact(expr)	Factorial or gamma function
Int(expr)	Integerize, or truncate fractional part of result of expr.
	100 input s 200 for x = 1 to 3 step s

	300 print tab(10*fact(x));"*" 400 next x
log(expr)	Natural log.
rnd(n)	Return pseudo random number between 0 and 32767. The **randomize** command seeds the random number with a continually changing time. See: **randomize**
sgn(n)	Return 1 with the same sign as n sgn(-4.3) = -1, sgn(100) = 1
s**in(expr)**	Sine.
sqr(expr)	Square root.
sum(<expression>,<loop variable>, <mode>,<start number>, <end number>)	Series summing function. 1200 print sum("1/fact(x)",”x”,”a”, 0, 6) Using this rather than a BASIC “for” loop is about twice as fast. It would be useful when you have millions of computations like in games or particle acceleration.
prod(“<expression>”,<loop variable>, <mode>,<start number>, <end number>)	Series product function. 1200 print prod(x , x, a, 0, 6) Gives 6 factorial Quotes are optional around non numerics.
timestamp()	Circular count in seconds up to 99999 seconds before going back to 0
det(<matrix>)	is a function that returns the determinant of an array or matrix. The array or matrix must be square, rows = columns.
len(<string>)	return length of string literal or variable.
pos(<string>,<key character>[,offset])	Returns position ‘p’ of character <key character> within string <string> with optional starting point <offset>
diff(<expression>,<variable used in expression>,<variable point>)	Returns the numeric value of the derivative of the expression at point <variable point>
gcd(<number1>,<number2>)	Returns greatest common denominator.
avg(<number>,<number>,<number>,…)	Returns average of given numbers.
vlength(<array name>)	Returns length of vector

.

4. Modes of Operation

4.1 Editor or Idle Mode

When the BASIC interpreter is invoked with no argument, a prompt " appears meaning that the interpreter is waiting for the user to enter something from the keyboard. BASIC is then said to be in the Editor or Idle mode.

Editing is accomplished as it is in any, BASIC language interpreter in that lines are entered by typing a line-number followed by the statement and removed or deleted by merely typing the line-number.

Listing is accomplished with the list or ‘l’ command (explained under "Standard Commands"). In addition to the above, it is possible to list single lines by typing the <ENTER> key in which case the program is listed one line-at-a-time, starting at the first. When the last one is reached, the sequence starts at the first line again. At any time it is also possible to type the symbol to "backup" a line-at-a-time. Other editing facilities are s delete, and reseq/renum also explained under "Standard Commands".

4.2 Run Mode

If the run command is typed and a program is currently in user storage, the program begins execution, starting with the first line of the program, then executing each line in order of line numbered sequence. The sequence of execution is altered by program flow control statements like jr, for-next or any statement, containing a *goto*.

4.3 Immediate Execution Mode

Immediate execution is accomplished by typing a command without preceding it with a line number. Although this is possible with all commands, it doesn't always make sense. For example, using commands that control program flow in immediate mode is unlikely and often disastrous.

Immediate mode is designed so that the user may get immediate action as in the command *run* or *print* a. Some commands are almost always used in immediate mode such as *q, delete, expunge, load, list, old, renum, save, etc.*

4.4 Single Step Single step mode is entered with the sing command and exited with, the con command. During this mode,' one may find "BUGS" in the program by observing the program flow or sequence or examining the values of variables at given points in the program to see if they have the expected values. See *sing* or *con* under the "Standard Commands" section of this paper.

5. Interruption of program ctrl-c or ctrl-break good for termination of infinite loops.

6 Error Messages

Diagnostic error messages are issued by the interpreter which indicate syntax errors , system failure, illegal commands or expressions, etc.

6.1 Standard Error Messages

NUMBER	**MESSAGE TEXT**
0	REFERS TO A NON-EXISTING LINE NUMBER
1	UNRECOGNIZABLE OPERATION
2	CANNOT OPEN FILE
3	ILLEGAL VARIABLE NAME
4	BAD FILENAME
5	WORKING STORAGE AREA EMPTY
6	RUNS NESTED TOO DEEPLY
7	UNASSIGNED VARIABLE

8	EXPRESSION SYNTAX
9	BAD' KEYWORD IN STATEMENT
10	IMPROPER OR NO. RELATIONAL OPERATOR
11	UNBALANCED QUOTES
12	FILE EDITING NOT PERMITT ED IN SINGLE STEP MODE
13	MISSING OR ILLEGAL DELIMITER
14	GOSUB- WITH NO RETURN
15	IS. FATAL
16	UNBALANCED PARENTHESIS
17	UNKNOWN MATH FUNCTION.
18	NEXT WITH NO OR WRONG FOR IN PROGRESS
19	CANNOT PROCESS, IMAGINARY NUMBER
20	WHAT?
21	BAD' DIMENSION SYNTAX
22	TOO MANY DIMENSIONS
23	REDUNDANT DIM STATEMENT
24	NOT ENOUGH WORKING STORAGE SPACE
25	VARIABLE NOT DIMENSIONED
26	WRONG NUM ' OF DIMS
27	ONE OR MORE DIMS LARGER THAN ASSIGNED
28.	NEG. OR ZERO DIMENSION. ILLEGAL
29	DIVIDE BY ZERO
30	BAD TAR SPEC. INPRINT
32	BAD FILE DECLARE SYNTAX
33	OUT OF DATA
34	FILE-NAME TOO, LONG
35	FILE DES. USED UP
36	FILE NOT OPEN FOR OUTPUT
37	FILE NOT OPEN FOR INPUT
38	EXPRESSION YIELDS AN IMPOSSIBLE VALUE
39	PRINTF: ARG COUNT MISMATCH
40	PRINTF: MORE THAN 10 ARGS
41	LINE TOO LONG FOR STRIP PRINTER
42	MOV REQUIRES 3 LINE #'s, SPACING IS OPTIONAL
43	BAD NAME OR LINE NUMBER AT BEGINNING OF SUBROUTINE
49	POSSIBLE EMPTY LINE IN INPUT FILE
50	NON-STRING IN STRING ASSIGNMENT
51	NUMERIC IN STRING EXPRESSION
52	INVALID STRING OPERATOR
53	CANNOT COMPARE STRING WITH NUM. TYPES
54	UNKNOWN STRING FUNCTION
55	OUT OF STRING RANGE
56	BREAK OR CONTINUE WITH NO 'WHILE' IN PROGRESS
57	FACTORIAL NEG. OR TOO LARGE
58	SEEK: SYNTAX ERR
59	SEEK FAILED
60	ILLEGAL SEEK MODE
61	REWIND: SYNTAX ERR

62	OPERATOR OR FUNCTION TYPE NOT VALID
63	INITIAL VALUE IN 'FOR' IS > FINAL WITH POSITIVE STEP
64	INITIAL VALUE IN 'FOR' IS < FINAL WITH NEGATIVE STEP
65	ARRAY MUST BE 2 DIMENSIONAL
66	MAT COMMAND MUST HAVE AN ARGUMENT
67	DIMENSIONS MUST BE THE SAME FOR MATRIX ADDITION OR SUBTRACION
68	DETERMINANT REQUIRES A SQUARE MATRIX
69	DEF FUNCTION WITH NO EXPRESSION
70	ILLEGAL DEFINED FUNCTION NAME
71	TOO MANY ARGUMENTS FOR FUNCTION
72	WRONG NUMBER OF ARGUMENTS IN FUNCTION CALL
73	TOO MANY ARGUMENTS TO USER DEFINED FUNCTION
74	NO ARRAY SPECIFIED
75	"MISSING ENDWHILE STATEMENT" /* 75 */"NUMERIC IN STRING EXPRESSION",

EXAMPLE BASIC PROGRAMS

These can be cut and pasted into a Windows notepad file, saved as a .txt then loaded into BASIC. Making sure that there are no intervening or trailing blank lines.

Fahrenheit to Celsius table 0 to 101 c

```
2000 openout temptable.txt
2100 prec 2
2200 dim f1(51)
2300 dim c1(51)
2400 dim f2(51)
2500 dim c2(51)
2600 for n = 1 to 51
2700 let c = n-1
2800 let f1(n) = 1.8*c+32
2900 let c1(n) = c
3000 next
3100 for n = 1 to 51
3200 let c = n+50
3300 let f2(n) = 1.8*c+32
3400 let c2(n) = c
3500 next
3600 printf "\n  F      C            F      C\n"
3700 printf "--------------------------------------------------\n"
3800 printf #1"\n      F      C            F      C\n"
3900 printf #1"--------------------------------------------------\n"
4000 for n = 1 to 51
4100 printf "    %3.2f   %3.2f   |      %3.2f   %3.2f\n",f1(n),c1(n),f2(n),c2(n)
4200 printf #1" %3.2f   %3.2f   |      %3.2f   %3.2f\n",f1(n),c1(n),f2(n),c2(n)
4300 next
4400 closef #1
```

```
1000 rem //         3 X 3 Determinant solution
2000 rem //    Solution to 3 variable simultaneous equations
2100 rem
2200 rem //                a11X + a12Y + a13Z = b1
2300 rem  //               a21X + a22Y + a23Z = b2
2400 rem //                a31X + a32Y + a33Z = b3
2500 rem
2600 print "Input values for b1, b2, b3 - like: 10,20,-30"
2700 input b1,b2,b3
2800 dim a(3,3)
2900 read a(1,1),a(1,2),a(1,3),a(2,1),a(2,2),a(2,3),a(3,1),a(3,2),a(3,3)
3000 gosub 6200
3100 let d = a1-a2+a3
3200 print "d = ";d
3300 restore
3400 read a(1,1),a(1,2),a(1,3),a(2,1),a(2,2),a(2,3),a(3,1),a(3,2),a(3,3)
3500 read a(1,1),a(2,1),a(3,1)
3600 gosub 6200
3700 let d1 = a1-a2+a3
3800 print "d1 = ";d1
3900 restore
4000 read a(1,1),a(1,2),a(1,3),a(2,1),a(2,2),a(2,3),a(3,1),a(3,2),a(3,3)
4100 read a(1,2),a(2,2),a(3,2)
4200 gosub 6200
4300 let d2 = a1-a2+a3
4400 print "d2 = ";d2
4500 restore
4600 read a(1,1),a(1,2),a(1,3),a(2,1),a(2,2),a(2,3),a(3,1),a(3,2),a(3,3)
4700 read a(1,3),a(2,3),a(3,3)
4800 gosub 6200
4900 let d3 = a1-a2+a3
5000 print "d3 = ";d3
5100 restore
5200 data 6,5,9
5300 data 2,0,1
5400 data 3,4,0
5500 data b1,b2,b3
5600 let x = d1/d
5700 let y = d2/d
5800 let z = d3/d
5900 print "x = ";x;" y = ";y;" z = ";z
6000 end
```

MAXIMUM FILE OPEN TEST WITH DEMONSTRATION OF FILES COMMAND
CLOSES TWO FILES THEN OPENS TWO FOR INPUT

```
90 rem //    Tests file open for read or write up to the maximum of 8 files
95 rem //    Opens 8 files, closes two, then opens two for input
96 rem //    Demonstrates the files command to show how the file table
97 rem //    is filled each time files are open or closed
98 rem //
100 openout aout.txt,bout.txt,cout.txt,dout.txt,eout.txt,fout.txt,gout.txt,hout.txt
200 print #1"Test 1"
300 print #2"Test 2"
400 print #3"Test 3"
500 print #4"Test 4"
600 print #5"Test 5"
700 print #6"Test 6"
800 print #7"Test 7"
900 print #8"test 8"
950 print "All file slots open for output."
1000 files
1100 closef #2
1200 closef #7
1250 print "Slot 2 and 7 closed."
1300 files
1400 openin test1.txt,test2.txt
1500 print "All input and output files closed.  Only current BASIC file, currently open, if any."
1550 print "Two files open for input to fill empty slots 2 and 7."
1600 files
1700 closeall
1800 files
```

TEST AND DEMO OF PRINTF COMMAND

```
2000 let p = 3.1415926535
2200 let e = 2.71828
2300 let t = 98.6
2400 let a = 222.95
2500 printf "p = $%3.2f e = $%3.2f\tt = $%3.2f\n", p,e,t
2600 printf "p = $%3.3f e = $%3.3f\tt = $%3.3f\n", p,e,t
2700 printf "p = $%3.2f e = $%3.2f\tt = $%3.4f\n", p,e,t
2800 printf "p = $%3.2f e = $%3.2f\tt = $%3.5f\n", p,e,t
2900 printf "p = $%3.2f a = $%3.2f\tt = $%3.5f\n", p,a,t
```

TEST AND DEMO OF NESTED IF-THEN-ELSE WITH TAB INDENTS

```
1100 print "Input numbers for a,b,c,d";
1200 input a,b,c,d
1300 if a > b then
1400    if c > d then
1500            print "a > b and c > d"
1600    else
1700            print "a > b and c < d"
1800    endif
1900 else
2000    if c > d then
2100            print "a < b and c > d"
2200    else
2300            print "a < b and c < d"
2400    endif
2500 endif
2600 goto 1100
```

FOR-NEXT-BREAK-CONTINUE

```
1000 for x = 1 to 10
1500     if x > 5 then
1600             break
1650     else
1700             print x
1710     endif
1800 next
1900 print "breaked at ";x
2000 for x = 1 to 10
2100     if x > 5 then
2200             continue
2300     else
2400             print x
2500     endif
2600 next
2700 print "continued to ";x
```

BASIC PROGRAM TO GENERATE SIN, SQUARE, SAWTOOTH WAVES GIVEN THE NUMBER OF HARMONICS THE FOURIER SERIES CALCULATRS OUT TO

```
90 expunge
100 print "Input number of Harmonics";
200 input n
300 if n = -1 then 1700
400 print "Select waveform: 1 for squarewave, 2 for sawtooth, 3 for triangle";
500 input w
600 if w <= 3 then 1200
700 print "Number ";w;" is out of range, input 1 - 3"
800 goto 400
900 rem //////////////////////////////////
1000 rem ///Print a square wave in asterisks
1100 rem //////////////////////////////////
1200 let a2 = 720
1300 for a = 1 to a2 step 40
1400 on w gosub 1800,3700,4700
1500 next
1600 goto 100
1700 stop
1800 rem square_f
1900 let y = 0
2000 let u = 0
2100 let s = 2
2200 for h = 1 to n step 2
2300 let u = u+((1/h)*sin(h*a))
2400 next
2500 rem Multiply Series with amplitude of 10
2600 rem Add a :carrier of 15 so that negative values
2700 rem don't get clipped
2800 let y = 10*u+15
2900 gosub 3100
3000 return
3100 rem plot_star
3200 for x = 1 to y
3300 print " ";
3400 next
3500 print "*"
3600 return
3700 rem sawtooth_f
3800 let y = 0
3900 let u = 0
4000 let s = 2
```

```
4100 for h = 1 to n step 1
4200 let u = u+(1/h)*sin(h*a+(h%2*180))
4300 next
4400 let y = 10*u+15
4500 gosub 3100
4600 return
4700 rem triangle_f
4800 let y = 0
4900 let u = 0
5000 let s = 2
5100 for h = 1 to n step 2
5200 let u = u+(n/fact(h))*cos(h*a)
5300 next
5400 let y = 10*u/h+20
5500 gosub 3100
5600 return
```

CALCULATE STANDARD DEVIATION AND LINEAR REGRESSION FROM DATA POINTS

```
1000 PRINT "FINDS AVERAGES OF TWO SETS OF NUMBERS, STANDARD"
1100 PRINT "DEVIATION FOR A GIVEN NUMBER OF X,Y PAIRS."
1200 PRINT "LISTS SUMS OF X, X SQUARED, Y, Y SQUARED AND"
1300 PRINT "SUM OF XY, AS STORED IN A SET OF REGISTERS 'R(N)'."
1400 PRINT "CALCULATES SLOPE AND INTERCEPT FOR LINEAR DIGRESSION."
1500 PRINT
1600 PRINT "FIRST ASKS WHETHER A NEW SET OF X,Y PAIRS IS TO BE"
1700 PRINT "ENTERED OR A CHANGE IS DESIRED IN ONE OF THE PAIRS"
1800 PRINT "ALREADY ENTERED."
1900 PRINT "THEN, EACH OF THE 'N' X,Y PAIRS IS REQUESTED.  WHEN"
2000 PRINT "THE LAST PAIR IS ENTERED, THE RESULT IS DISPLAYED."
2100 PRINT
2200 PRINT "IF 'C' IS CHOSEN, IT ASKS WHETHER YOU WANT A NUMBERED"
2300 PRINT "LIST OF PAIRS.  THEN ASKS WHICH ENTRY YOU WANT TO CHANGE."
2400 PRINT "THE MODIFIED LIST IS USED TO RE-CALCULATE."
2500 PRINT
2600 COMMON
2700 PRINT "ENTER 'N' FOR NEW VALUES OR 'C' TO CHANGE A VALUE";
2800 INPUT W$
2900 IF W$ = "N" THEN 3200
3000 IF W$ = "C" GOSUB 7400
3100 GOTO 4700
3200 EXPUNGE
3300 PRINT "INPUT NUMBER OF PAIRS:";
3400 INPUT N
3500 LET N2 = N
3600 DIM X(N)
3700 DIM Y(N)
3800 DIM R(5)
3900 PRECISION 11
4000 FOR C = 1 TO N
4100     PRINT "INPUT: X";C;",Y";C
4200     INPUT X,Y
4300     LET X(C) = X
4400     LET Y(C) = Y
4500 NEXT
4600 PRECISION 11
4700 FOR R = 1 TO 5
4800     LET R(R) = 0
4900 NEXT
5000 GOSUB 6600
5100 GOSUB 5400
```

```
5200 GOSUB 8600
5300 END
5400 FOR R = 1 TO 5
5500     PRINT "R(";R;") = ";R(R)
5600 NEXT
5700 PRINT
5800 PRINT "AVERAGE FOR X = ";R(1)/N
5900 PRINT "AVERAGE FOR Y = ";R(3)/N
6000 LET S1 = SQRT((N*R(2)-R(1)^2)/(N*(N-1)))
6100 LET S2 = SQRT((N*R(4)-R(3)^2)/(N*(N-1)))
6200 PRINT
6300 PRINT "STANDARD DEVIATION FOR X = ";S1
6400 PRINT "STANDARD DEVIATION FOR Y = ";S2
6500 RETURN
6600 FOR C = 1 TO N
6700     LET R(1) = R(1)+X(C)
6800     LET R(2) = R(2)+X(C)^2
6900     LET R(3) = R(3)+Y(C)
7000     LET R(4) = R(4)+Y(C)^2
7100     LET R(5) = R(5)+X(C)*Y(C)
7200 NEXT
7300 RETURN
7400 PRINT "LIST CURRENT VALUES? Y/N";
7500 INPUT L$
7600 IF L$ = "Y" GOSUB 8200
7700 PRINT "CHANGE VALUES FOR ELEMENT";
7800 INPUT N2
7900 PRINT "NEW VALUES FOR X AND Y:";
8000 INPUT X(N2),Y(N2)
8100 RETURN
8200 FOR C = 1 TO N
8300     PRINT "X(";C;") = ";X(C);" Y(";C;") = ";Y(C)
8400 NEXT
8500 RETURN
8600 PRINT
8700 PRINT "LINEAR REGRESSION SLOPE 'A' AND INTERCEPT 'B'"
8800 LET A = (N*R(5)-R(1)*R(3))/(N*R(2)-R(1)^2)
8900 LET B = (R(3)*R(2)-R(1)*R(5))/(N*R(2)-R(1)^2)
9000 PRINT "SLOPE A = ";A;" INTERCEPT B = ";B
9100 RETURN
```

MATRIX TEST AND DEMO

```
1000 print "Choose a matrix test."
1050 print " Type 1 for 2X2,"
1100 print "      2 for 3X3,"
1150 print "      3 for 1X2 * 2X2,"
1200 print "      4 for scaler multiply,"
1250 print "      5 for matrix add 3X3,"
1300 print "      6 for 3X3 'print,' test,"
1350 print "      7 for transpose,"
1400 print "      8 for matrix inverse 3X3,"
1450 print "      9 for MXM Identity Matrix:"
1500 input t
1550 on t gosub 1700,2650,3600,4600,5300,5950,6650,7150,7750
1600 expunge
1650 goto 1000
1700 dim a(2,2),b(2,2),c(2,2)
1750 restore
1800 mat  read b
1850 mat  read c
1900 data 1,-2,0,3
1950 data -3,0,1,2
2000 mat a = b*c
2050 print
2100 print "B"
2150 print
2200 mat  print b
2250 print
2300 print "C"
2350 print
2400 mat  print c
2450 print "A"
2500 print
2550 mat  print a
2600 return
2650 dim a(3,3),b(3,3),c(3,3)
2700 restore 2850
2750 mat  read b
2800 mat  read c
2850 data 1,2,0,0,1,1,2,0,1
2900 data 1,1,2,2,1,1,1,2,1
2950 mat a = b*c
3000 print
3050 print "B"
```

```
3100 print
3150 mat  print b
3200 print
3250 print "C"
3300 print
3350 mat  print c
3400 print "A"
3450 print
3500 mat  print a
3550 return
3600 dim a(1,2),b(1,2),c(2,2)
3650 restore 3850
3700 mat  read b
3750 mat  read c
3800 rem
3850 data -1,0
3900 data 0,2,-3,1
3950 mat a = b*c
4000 print
4050 print "B"
4100 print
4150 mat  print b
4200 print
4250 print "C"
4300 print
4350 mat  print c
4400 print "A"
4450 print
4500 mat  print a
4550 return
4600 dim a(3,3)
4650 dim b(3,3)
4700 print "Input a scaler multiplier";
4750 input m
4800 restore 5200
4850 mat  read b
4900 read b(3,1),b(3,2),b(3,3)
4950 mat  read b
5000 mat A = (m)*B
5050 mat  print b
5100 print
5150 mat  print a
5200 data 1,2,3,4,5,6,7,8,9
5250 return
```

```
5300 dim a(3,3),b(3,3),c(3,3)
5350 restore 5800
5400 mat  read b
5450 mat  read c
5500 mat a = b+c
5550 mat  print b,
5600 print
5650 mat  print c,
5700 print
5750 mat  print a,
5800 data 1,2,3,4,5,6,7,8,9
5850 data 9,8,7,6,5,4,3,2,1
5900 return
5950 dim a(3,3)
6000 mat  read a
6050 mat  print a
6100 print
6150 restore 6450
6200 mat  read a
6250 mat  print a,
6300 data 6,4,3
6350 data 2,6,4
6400 data 3,4,0
6450 data -3.1415,222.22,33.5
6500 data 196.5,197.8,199.1
6550 data 7.4,2.5,-3.3
6600 return
6650 print "Transpose"
6700 dim a(4,4),b(4,4)
6750 restore 7050
6800 mat  read b
6850 mat a = trn(b)
6900 mat  print b
6950 print
7000 mat  print a
7050 data 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16
7100 return
7150 print "Input size of matrix < 6:";
7200 input s2
7250 dim a(s2,s2),b(s2,s2)
7300 restore 7600
7350 mat  read b
7400 mat a = inv(b)
7450 mat  print b
```

```
7500 print
7550 mat  print a
7600 data 6,5,4,8,20,40,50,30,2,100,40,75,45,33.3,78
7650 data 100,340,201,659,3.4,5.5,6.6,3.3,83.33
7700 return
7750 print "Input size of identity matrix:";
7800 input s
7850 print
7900 print "Identity"
7950 dim a(s,s)
8000 mat a = idn(s,s)
8050 print
8100 mat  print a
8150 return
```

Types of Matrices

There are many different types of matrices in linear algebra. All types of matrices are differentiated based on their elements, order, and certain set of conditions. The word "Matrices" is the plural form of a matrix and is the less commonly used to denote matrices. In this article, let's learn about some of the commonly used types of matrices, their definition with examples.

What are Different Types of Matrices?

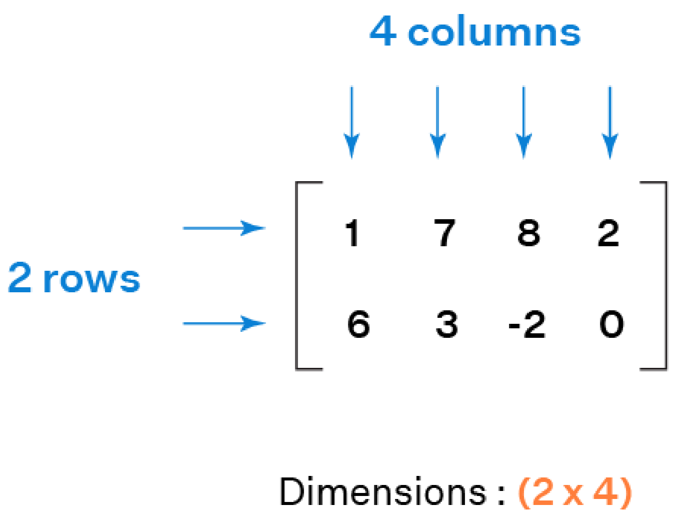
This article describes some of the important types of matrices that are used in mathematics, engineering, and science. Here is the list of the most commonly used types of matrices in linear algebra:

- Row Matrix
- Column Matrix
- Singleton Matrix
- Rectangular Matrix
- Square Matrix
- Identity Matrices
- Matrix of ones
- Zero Matrix
- Diagonal Matrix

We can use these different types of matrices to organize data by age group, person, company, month, and so on. We can then use this information to make decisions and solve a lot of math problems.

Identifying Types of Matrices Based on Dimension

Matrices are in all sorts of sizes, but usually, their shapes remain the same. The size of a matrix is called its dimension which is the total number of rows and columns in a given matrix. In the below-given image, we can see how the dimension of a matrix is calculated.



In this section, let's learn to identify the types of matrices based on their dimension:

Row and Column Matrix

Matrices with only one row and any number of columns are known as row matrices and matrices with one column and any number of rows are called column matrices. Let's look at two examples below:

Row Matrix	Column Matrix
A=[1024]	B=||⌊325||⌋
There is only one row, so A is a row matrix.	There is only one column, so B is a column matrix.

Rectangular and Square Matrix

Any matrix that does not have an equal number of rows and columns is called a rectangular matrix and a rectangular matrix can be denoted by $[B]_{m\times n}$

. Any matrix that has an equal number of rows and columns is called a square matrix and a square matrix can be denoted by $[B]_{n\times n}$

. Let's look at the examples below:

Rectangular Matrix	Square Matrix
B=⎢⎢⎣2−13505271−1−29⎥⎥⎦	

C=⎢⎢⎣2−130521−1−2⎥⎥⎦

There are three rows and four columns in this matrix, so B is a rectangular matrix.	There are three rows and three columns in this matrix, so C is a square matrix.

Constant Matrices

Constant matrices are matrices in which all the elements are constants for any given dimension/size of the matrix. The matrix elements are denoted by b_{ij}

. Let's look at these types of matrices whose elements are always constant.

Identity Matrix	Matrix of Ones	Zero Matrix
The identity matrix is a square diagonal matrix, in which all entries on the main diagonal are equal to 1, and the rest of the elements are equal to 0. It is denoted by I.	Any matrix in which all the elements are equal to 1 is called a matrix of ones.	Any matrix in which all the elements are equal to 0 is called a zero matrix.
I=⎢⎢⎣100010001⎥⎥⎦		

C=⎢⎢⎣111111111⎥⎥⎦

D=⎢⎢⎣000000000⎥⎥⎦

Other Types of Matrices

Apart from the most commonly used matrices, there are other types of matrices that are used in advanced mathematics and computer technologies. Following are some of the other types of matrices:

Singular and Non-singular Matrix

Any square matrix whose determinant is equal to 0 is called a singular matrix and any matrix whose determinant is not equal to 0 is called a non-singular matrix. Determinant of a matrix can be found by using determinant formula. Let's look at two examples below:

Singular Matrix	**Non-singular Matrix**
C = ⎡⎢⎣111111111⎤⎥⎦	

D = ⎡⎢⎣211121111⎤⎥⎦

|C| = ∣∣∣111111111∣∣∣

|D| = ∣∣∣211121111∣∣∣

|C| = 1×∣∣∣1111∣∣∣−1×∣∣∣1111∣∣∣+1×∣∣∣1111∣∣∣

|D| = 2×∣∣∣2111∣∣∣−1×∣∣∣1111∣∣∣+1×∣∣∣1211∣∣∣

=1×(1×1-1×1)-1×(1×1-1×1)+1×(1×1-1×1)	=2×(2×1-1×1)-1×(1×1-1×1)+1×(1×1-2×1)

=1×(1-1)-1×(1-1)+1×(1-1) =1×(0)-1×(0)+1×(0) =0+0+0 =0	=2×(2-1)-1×(1-1)+1×(1-2) =2×(1)-1×(0)+1×(-1) =2-0-1 =1
Here, |C| = 0, so C is a singular matrix	Here, |D| ≠ 0, so D is a non-singular matrix

Diagonal Matrix

A square matrix in which all the elements are 0 except for those elements that are in the diagonal is called a diagonal matrix. Let's take a look at the examples of different kinds of diagonal matrices: A scalar matrix is a special type of square diagonal matrix, where all the diagonal elements are equal.

Diagonal Matrix	Scalar Matrix
B = ⎡⎢⎢⎢⎣1000050000200004⎤⎥⎥⎥⎦	
C = ⎡⎢⎣300030003⎤⎥⎦	
Here, we can see that except for the diagonal elements, all the other elements are zero. Hence, B is a diagonal matrix.	Here, we can see that the diagonal elements are equal and all the other elements are zero. Hence, C is a scalar matrix.

Upper and Lower Triangular Matrix

An upper triangular matrix is a square matrix where all the elements that are present below the diagonal elements are 0. A lower triangular matrix is a square matrix where all the elements that are present above the diagonal elements are 0. Let's look at the examples below:

Upper Triangular Matrix	Lower Triangular Matrix
B = ⎡⎢⎣321045006⎤⎥⎦	

C = ⎢⎢⎣300410279⎥⎥⎦

Here, we can see that all the elements that are present below the main diagonal are 0. Hence, B is an upper triangular matrix.	Here, we can see that all the elements that are present above the main diagonal are 0. Hence, B is an upper triangular matrix.

Symmetric and Skew Symmetric Matrix

A square matrix D of size n×n is considered to be symmetric if and only if D^T= D. A square matrix F of size n×n is considered to be skew-symmetric if and only if F^T= - F. Let's consider the examples of two matrices D and F:

Symmetric Matrix	**Skew-symmetric Matrix**
D = ⎢⎢⎣236345659⎥⎥⎦	

D^T = ⎡⎢⎣236345659⎤⎥⎦

F = [03−30]

F^T = [0−330]

-F = [0−330]

Here, D = D^T. Hence, D is a symmetric matric.	Here, F^T = -F. Hence F is a skew-symmetric matrix.

Boolean Matrix

A matrix is considered to be a boolean matrix when all its elements are either 1s and 0s. Let's consider the example of the matrix B to understand this better:

B = ⎡⎢⎣010101011⎤⎥⎦

Stochastic Matrices

A stochastic matrix is a type of matrix whose all entries represent probability. A square matrix C is considered to be left stochastic when all of its entries are non-negative and when the entries in each column sum to 1. Similarly, a matrix with all its entries as non-negative such that entries in each row sum to 1 is called a right stochastic matrix. Consider the example of the matrix C here:

C = ⎡⎢⎣0.30.40.50.30.40.30.40.20.2⎤⎥⎦

Orthogonal Matrix

A square matrix B is considered to be an orthogonal matrix, when B × B^T = I, where I is an identity matrix and B^T is the transpose of matrix B. Take an example of the matrix B:

B =[0110]

B^T=[0110]

B × B^T = [0110]×[0110]

= [0+10+00+01+0]

= [1001]

Here, we can see that B × B^T = I. Hence, B is an orthogonal matrix.

WISH LIST for 2022

Add step ability in while loop
Generalized logic expressions for if and while if a > b and c < d or e < f, etc.
mat redim (redimension)
mat a = eigen(b)
Graphics
Port to Windows
Port to Linux