Chapter 5

MATHEMATICAL FUNCTIONS

5.1 STANDARD FUNCTIONS

All microcomputers have a library of built-in mathematical functions, but the BBC microcomputer has a particularly comprehensive set. RND will be dealt with separately, but the remainder are listed below, with their effects.

There are two important points to remember

1. In all cases the opening brackets of the parameter or argument must follow without a space in between (for example INT(X), not INT (X)).
2. Most of the mathematical functions are calculated by approximation methods. This means that the results can not be guaranteed to be exact; there are always liable to be rounding errors. Thus you must be careful how you handle the results; for instance, never test the result of a mathematical function for equality. Either test with > or <, or test that it lies within a small range around the desired value. For example, if you want to test whether SIN (X) is 0.5, use the test

```
IF SIN(X)>0.499999 AND SIN(X)<0.500001
```

INT(X)

Gives the integer less than or equal to X. Thus INT(5.9) gives 5 and INT(-2.3) gives -3.

If you want to round to the nearest integer instead of always rounding down, use INT(X+0.5).

X must lie within the range of integer variables, between -2147483645 and +2147483645, otherwise the function will fail with the error ‘Too big’.

ABS(X)

Gives the magnitude of X, that is the value ignoring any minus sign.

SGN(X)

Gives the sign of X

```
X<0 SGN (X)=-1
X=0 SGN (X)=0
X>0 SGN (X)=1
```

Thus SGN(X)* ABS(X)=X.

SQR(X)

Gives the positive square root of X. X must be positive.

PI

Returns the value of π.

DEG(X) and RAD(Y)

DEG(X) gives the value in degrees of an angle X in radians (I degree = π/180 radians). RAD(Y) performs the reverse function. Thus RAD(DEG(X))=X (within the rounding errors of the conversion functions).

SIN(X), COS(X) and TAN(X)

These functions give the trigonometric functions sine, cosine and tangent. X must be in radians (but if it is in degrees, use SIN(RAD(X))).

ASN(X), ACS(X) and ATN(X)

These functions give the inverse trigonometric functions aresin, arccos and arctan. The result is in radians, but to get Y in degrees, use Y=DEG(ASN(X)). For ASN and ACS, X must be in the range -1<X<1.

LOG(X), LN(X) and EXP(X)

LOG gives the logarithm of X to base 10, LN(X) gives the log to base e (natural logarithm), and EXP(X) gives e to the power X. Thus LN(EXP(X))=X. The only inverse function for LOG(X) is the exponent form of a number (for example 3E5).

The ranges of X are

X>0 for LOG and LN
X<=88 for EXP

Outside these ranges, the explicit error messages ‘Log range’ and ‘Exp range’ occur.

EVAL(FUNC$)

This function attempts to treat the string FUNC$ as a mathematical expression and evaluate it.

The purpose of the function is to allow the user to input a mathematical expression, in order to plot a graph of the function for instance. The EVAL function is used in Example 9.9 to evaluate user input that may be in either decimal or hexadecimal.

DIV and MOD

These operators yield respectively the whole number part and the remainder of the division of two integer numbers. (If used with non-integer numbers, the numbers are truncated before the division, so that only the integer part of the number is used.) For example

`13 DIV 3` gives `4`
`13 MOD 3` gives `1`

Clearly B%*(A% DIV B%)+A% MOD B% = A%

AND, OR and EOR

These will be dealt with later in this chapter, but are listed here for completeness. In contrast to some microcomputers, they carry out full four byte, bitwise operations. EOR means exclusive or, and is not commonly available on other microcomputers.

Exercise 5.1

Check all the predicted relationships by direct mode commands such as

```
PRINT 3*(13 DIV 3)+13 MOD 3
```

Exercise 5.2

Write a program to INPUT as a string a mathematical expression, EXP(-X*X) for example, and use EV AL to print a table for a series of values of your expression and its parameter(s).

Exercise 5.3

Write a program for the user to enter an amount of money in pence, separate it into pounds and pence using DIV and MOD and print out the result.

5.2 RANDOM NUMBERS

The use and application of random numbers is worth considering in a little more detail. The random numbers produced by computers are more properly pseudo-random numbers, generated from the remainders after one large prime number is divided by another. To see how this works out in practice, try the following program.

Example 5.1

```
10 INPUT "Type any vaLue (1 to 13) for the seed: ",SEED
20 CONST=19
```

```
30 DVSR=13
40 FOR J=1 TO 15
50 SEED=(CONST*SEED) MOD DVSR
60 PRINT SEED
70 NEXT J
```

The choice of 13 for the divisor gives 12 possible numbers, which as you will see come fairly randomly. The sequence then starts to repeat. The ‘seed’ simply decides at which point you join the repeating sequence.

If you want numbers between 0 and 1, this can simply be achieved by dividing the seed by 13.

With much larger prime numbers than 19 and 13, large sequences of random numbers can be produced. Try for example 4637 in place of 19 and 2546887 in place of 13 and replace line 50 by

```
50 SEED=CONST*SEED-INT(CONST*SEED/DVSR)*DVSR
```

since MOD will no longer work for such large numbers.

There are two alternative possibilities that you might need when using a random number generator. On the one hand, when using a program repeatedly, or developing a new program, you might want to use the same sequence each time, so that you ‘know’ the answer or can check other factors in your program. On the other hand, you may want to be sure of getting a different sequence each time, as in the card game we shall look at later.

BBC BASIC has a random number generator called by the function RND(X). This function has a number of different modes, depending on the value of X supplied. (As so often, X should be an integer, and real numbers are truncated.) These are summarized as follows

X absent	Random integer number between $+23^2$ and -23^2
X>1	Random integer number between 1 and X
X=1	Random fractional number between 0 and 1
X=0	Repeats the last number generated by RND if it was RND(1)
X<0	Returns the value X and reseeds the generator with a seed based on X

This choice caters for almost all eventualities. An effectively infinite range of numbers can be produced by RND or RND(1), depending on whether you want integer or fractional numbers. (To alter the range of real numbers, say to 0 to 100, use 100*RND(1). RND(100) would give an integer between 1 and 100.)

A random value from a set of integers up to N can be produced by RND(N). Thus to simulate the throw of a dice, you could use RND(6).

To get a repeatable sequence of numbers, start your program with X=RND(-1) (or any negative value which takes your fancy). To get a guaranteed non-repeatable sequence needs just a little more effort, since the generator always starts with the same seed when first switched on. The easiest solution is to reseed with a random negative number, which can quite simply be produced from TIME, which is busily ticking away 100 times a second.

Example 5.2

A simple application to simulate the throw of two dice is given in the program below.

```
10 CLS
20 X=RND(-TIME): REM RANDOMIZE THE START POINT
30 INPUT "Number of throws: ",N
40 PRINT
50 FOR J=1 TO N
60 T1=RND(6): T2=RND(6)
70 PRINT "Thows are ";T1;" and ";T2;". Totat ";T1+T2
80 NEXT J
```

Example 5.3

A simple program to test the statistical randomness of a sequence of numbers is shown below. Using RND(1), the mean should be 0.5, and the standard deviation should be SQR(0.08333/(N-1)).

```
 10 CLS
 20 INPUT "Hw many numbers are to be tested? ",N
 30 X=RND(–TIME)
 40 SUM=0: SUMDEVSQ=0
 50 FOR J=1 TO N
 60 X=RND(1)
 70 SUM=SUM+X
 80 SUMDEVSQ=SUMDEVSQ+(0.5-X)*(0.5-X)
 90 NEXT J
100 PRINT '"Mean is ";SUMIN;" (nominal 0.5)"
110 PRINT '"Standard deviation is ";SQR(SUMDEVSQ/ (N*(N-1)))'
    "(nominal ";SQR(0.08333/(N-1)); ")"
```

Example 5.4

Finally, the more elaborate program below shows how a realistic computer version of the card game pontoon can be written with the aid of RND. The key line is line 1010 where the procedure selects a card using X(N)=RND(13).

```
10 CLS
20 DATA ACE,TWO,THREE,FOUR,FIVE,SIX,SEVEN
30 DATA EIGHT,NINE,TEN,JACK,QUEEN,KING
40 X=RND(-TIME)
50 DIM CARD$(13),SCORE(2),PONTOON(2),ACE(2),X(10),Y(10)
60 FOR J=1 TO 13: READ CARD$(J): NEXT J
70 WINNINGS=0
```

```
  80 PLAYER=1
  90 SCORE(PLAYER)=0: PONTOON(PLAYER)=0: ACE(PLAYER)=0
 100 WHO$="Your": WHICH$=" first"
 110 N=1: PROC_deal(1): PROC_result(WHO$,WHICH$,CARD$(X(N)),
     PONTOON(PLAYER),SCORE(PLAYER),ACE(PLAYER))
 120 INPUT "Place your bet, in pence: ",BET
 130 WHICH$=" second"
 140 N=2: PROC_deal(2): PROC_result(WHO$,WHICH$,CARD$(X(N)),
     PONTOON(PLAYER),SCORE(PLAYER),ACE(PLAYER))
 150 IF PONTOON(PLAYER)=0 THEN PROC_twist
 160 IF SCORE(PLAYER)>21 AND PONTOON(PLAYER)=0 THEN PRINT
     "BUST": WIN=-1: GOTO 290
 170 IF ACE(PLAYER)>0 AND SCORE (PLAYER)<-11 THEN
     SCORE(PLAYER)=SCORE(PLAYER)+10
 180 IF N>=5 THEN PRINT "FIVE CARDS": SCORE(PLAYER)=40
 190 PLAYER=2: SCORE(PLAYER)=0: PONTOON(PLAYER)=0:ACE(PLAYER)=0
 200 WHO$="My": WHICH$=" first"
 210 N=1: PROC_deal(1): PROC_result(WHO$,WHICH$,CARD$(X(N)),
     PONTOON(PLAYER),SCORE(PLAYER),ACE(PLAYER))
 220 WHICH$=" second"
 230 N=2: PROC_deal(2): PROC_result(WHO$,WHICH$,CARD$(X(N)),
     PONTOON(PLAYER),SCORE(PLAYER),ACE(PLAYER))
 240 IF PONTOON(PLAYER)=0 THEN PROC_dealer
 250 IF SCORE(PLAYER)>21 AND PONTOON(PLAYER)=0 THEN PRINT
     "BUST": WIN=1: GOTO 290
 260 IF ACE(PLAYER)>0 AND SCORE(PLAYER)<=11 THEN
     SCORE(PLAYER)=SCORE(PLAYER)+10
 270 IF N>=5 THEN PRINT "FIVE CARDS": SCORE(PLAYER)=40
 280 IF SCORE(2)>=SCORE(1) THEN WIN=-1: ELSE WIN=1
 290 IF WIN=ASN1 THEN PRINT "YOU LOSE": ELSE PRINT "YOU WIN"
 300 WINNINGS=WINNINGS+BET*WIN
 310 IF WINNINGS<0 THEN PRINT "You owe me ";ELSE PRINT "I owe
     you ";
 320 PRINT ABS(WINNINGS);" pence"
 330 INPUT "Would you like to play again (Y/N)",A$
 340 IF A$="Y" THEN GOTO 80
 350 IF WINNINGS<0 THEN PRINT "Please pass my winnings into
     the grill"
 360 END
1000 DEF PROC_deal(N)
1010 X(N)=RND(13)
1020 IF X(N)=1 THEN ACE(PLAYER)=N
1030 Y(N)=X(N): IF Y(N)>10 THEN Y(N)=10
1040 SCORE(PLAYER)=SCORE(PLAYER)+Y(N)
1050 IF ACE(PLAYER)>0 AND N=2 AND SCORE(PLAYER)=11 THEN
     PONTOON(PLAYER)=1: SCORE(PLAYER)=30
1060 REM PONTOON
1070 ENDPROC
```

```
2000 DEF PROC_result(WHO$,WHICH$,CARD$,PONTOONFLAG,SCORE,ACEFLAG)
2010 PRINT 'WHO$;WHICH$;" card is: ";CARD$
2020 IF PONTOONFLAG>0 THEN PRINT "PONTOON": ENDPROC
2030 IF N>1 THEN PROC_score(SCORE,ACEFLAG)
2040 ENDPROC
3000 DEF PROC_score(SCORE,ACEFLAG)
3010 PRINT "Total: ";SCORE;
3020 IF ACEFLAG>0 THEN PRINT " or ";SCORE+10;
3030 PRINT
3040 ENDPROC
4000 DEF PROC_twist
4010 INPUT "Twist or stick (T/S)",A$
4020 IF A$="S" THEN ENDPROC
4030 N=N+1: PROC_deal(N)
4040 PRINT '"Your next card is: ",CARD$(X(N))
4050 IF SCORE(PLAYER)>21 THEN ENDPROC
4060 PROC_score(SCORE(PLAYER),ACE(PLAYER))
4070 PROC_twist
4080 ENDPROC
5000 DEF PROC_dealer
5010 IF SCORE(PLAYER)>17 OR N>=5 OR (ACE(PLAYER)>0
     AND (SCORE(PLAYER)<12 AND SCORE(PLAYER)>9)) THEN ENDPROC
5020 PRINT "I will twist"
5030 N=N+1
5040 PROC_deal(N)
5050 PRINT '"My next card is: ",CARD$(X(N))
5060 IF SCORE(PLAYER)>21 THEN ENDPROC
5070 PROC_score(SCORE(PLAYER),ACE(PLAYER))
5080 PROC_dealer
5090 ENDPROC
```

Note the use of recursion in the procedures PROC_twist and PROC_dealer. Most of the procedures have parameters because this is good practice, but it does make the program slightly longer.

5.3 LOGICAL EXPRESSIONS

5.3.1 Booleans

When a conditional test is made in an IF...THEN statement, what is happening is that a relationship is being evaluated as either TRUE or FALSE. A relationship which can have just the two possible values TRUE or FALSE is called a *Boolean*. More accurately, the IF statement has the general form

```
IF <Boolean expression> THEN...
```

On the BBC microcomputer, the Boolean values TRUE and FALSE actually exist as pseudo-variables (they can be read, but not altered). Thus it is quite legal (though rather pointless) to write

```
IF TRUE THEN...
```

In this case the statements following THEN would always be obeyed. A more useful application is with REPEAT...UNTIL, where again a Boolean is evaluated.

```
REPEAT...UNTIL FALSE
```

will create an endless loop, which can sometimes be useful. A loop that is to be repeated until a certain condition is fulfilled midway through the loop could have the structure

```
FOUND= FALSE
REPEAT
  ...
  ...
  ...
IF <condition> THEN FOUND=TRUE
  ...
  ...
  ...
UNTIL FOUND
```

Just as with arithmetic, we can have Boolean expressions as well as simple Booleans. There is a complete algebra for Boolean expressions. We will study this in full after reviewing the types of conditional test that are possible.

5.3.2 Conditional tests

The normal form of Boolean is a relationship, which involves testing two quantities with a *relational operator*, such as > (greater than). The full list of possible operators is

>	greater than
<	less than
=	equals
<>	not equals
>=	greater than or equals
<=	less than or equals

The double symbols must be in the order shown (e.g. <>, not ><) and *must not* be separated by a space.

The computer will evaluate the relationship as TRUE or FALSE, and assign a numerical value to the outcome (-1 for TRUE or 0 for FALSE). A relationship can thus also be used within a normal mathematical expression. Note that the equals sign has two quite separate functions in BASIC. It is used in an assignment statement, which strictly has the form

```
LET <variable>=<expression>
```

The LET is almost invariably omitted, but it does serve to emphasize that this is an assignment.

The second use of the equals sign is in a relational test. The computer finds no ambiguity in this, since it will treat = as an assignment where it is logical to do so, and a relational test otherwise.

Actually, a test for equality is not always a good idea, as was pointed out earlier when discussing functions. Never test real variables for equality if either has been evaluated in an expression, rather than having a directly assigned value (as the control variable of a FOR loop, for example).

Exercise 5.4

Check the fact that relationships have numerical values by direct mode statements such as

```
PRINT (1=2)
PRINT 15*(7<>3)
PRINT 7+(9<=10)
```

The fact that BASIC treats TRUE and FALSE as numbers means that it is also possible to write IF statements of the form

```
IF A% THEN...
```

A% will evaluate as TRUE if it has any value other than zero. Thus the above is a shorter equivalent to

```
IF A%<>0 THEN...
```

(Beware, however, that IF NOT A%... is not equivalent to IF A%=0...)

Check out these statements by lines such as

```
IF 10 THEN PRINT "TRUE" ELSE PRINT "FALSE"
```

The fact that a relational expression takes a value can be utilized to advantage to produce more compact programs and avoid IF...THEN statements.

For example, suppose we want to set a bus fare to 10p, 50p or 25p depending on whether the age of the traveller is under 15, adult or over 65. The obvious way to do this is

```
100 FARE=10
110 IF AGE>=15 THEN FARE=50
120 IF AGE>=65 THEN FARE=25
```

However, since a relational expression is -1 if TRUE and 0 if FALSE, this can be written much more compactly (though at the expense of clarity) as

```
100 FARE=10-40*(AGE>=15)+25*(AGE>=65)
```

Exercise 5.5

Rewrite lines 170 and 260 of the pontoon program, Example 5.4, using the above method.

Relational operators work with strings and string variables as well as numbers. The meaning of = and <> is fairly obvious, and useful in situations such as YES/NO tests. For example

```
100 INPUT "Do you want to continue? ",YN$
110 IF YN$="YES" THEN PR0C_continue ELSE IF YN$<>"NO" THEN
    GOTO 100
```

Testing for inequalities is also possible, however, and carries out an alphabetical (or more strictly, lexicographical) test, ordering strings as they would appear in a dictionary. Thus

"FRED" is less than "JIM"
"BET" is less than "BETTER"
"10" is less than "20" (but "20" is less than "3")

Actually, what is done is a successive comparison of the ASCII values of the characters until a difference (if any) is found.

5.3.3 Boolean operators

It is possible to combine relationships with special operators called logical or Boolean operators. The available Boolean operators are AND, OR, EOR and NOT.

AND and OR perform just the logical operations that you would expect. EOR is the Exclusive OR operator, and is less common. Thus

`A AND B`	is TRUE if A and B are both TRUE
`A OR B`	is TRUE if either A or B is TRUE
`A EOR B`	is TRUE if either A or B but *not both* is TRUE

NOT is rather different, being a unary operator which reverses the value of a relationship. (A unary operator is one which operates on a single quantity; the others are binary operators, and act on two quantities.) Thus

NOT TRUE is FALSE
NOT FALSE is TRUE

5.3.4 Order of priority of operators

The rules for combining operators are the basis of Boolean algebra. The mathematical and Boolean operators are listed below in order of execution in a compound expression. Operators on the same line have equal priority, and where these are combined they are evaluated from the left.

`()`
`NOT` (and + - as unary operators, as in 7*-6)
`^`
`* /`
`> < <> >= <= =`
`AND`
`OR EOR`

As with simple conditional tests, the Boolean operators find their most obvious use in IF statements. A pair of tests might be combined with a Boolean operator, for instance

```
IF X<1 OR X>4 THEN PRINT "Error"
```

or

```
IF NOT(R$="Y" OR R$="N") THEN PRINT "Error"
```

Exercise 5.6

Check the following statements, by using them as the <condition> in IF <condition> THEN PRINT "TRUE" ELSE PRINT "FALSE"

```
NOT - 1 =0     NOT 0=-1

0 AND -1=-=0   -1 AND -1=-1

0 OR -1=-1     0 OR 0=0

0 EOR -1=-1    -1 EOR -1=0
```

Exercise 5.7

Try a number of assignments in direct mode such as

1. `X=NOT((2+5)>7`
   `PRINT X` gives -1 (TRUE)
2. `Z=NOT((3+2)>=4)`
   `PRINT Z` gives 0 (FALSE)
3. `Y=6=3 AND (2<>3 OR 1<2)`
   `PRINT Y` gives 0 (FALSE)

Try example 3 again without the brackets, and then with brackets around 6=3 AND 2<>3.

Note that Y=6=3 makes perfectly good sense to the BBC microcomputer, since 6=3 is evaluated first as a relational expression.

The most useful application of Boolean expressions is within the IF...THEN statement, which tests the truth of a Boolean expression.

For example, examine line 5010 of the pontoon program, Example 5.4, where the computer decides whether to twist. Note also the use of brackets to ensure the desired logic.

Exercise 5.8

Extract line 5010 from Example 5.4 as a simple program

```
10 INPUT "Score ",SC
20 INPUT "Ace ",ACE
30 INPUT "N",N
40 IF SC>17 OR N>=5 OR (ACE>0 AND (SC<12 AND SC>9)) THEN PRINT
   "STICK" ELSE PRINT "TWIST"
```

Run this using different values of Score, Ace and N to check the logic of the relationship.

5.3.5 Bitwise action of the Boolean operators

This last subsection assumes some familiarity with binary arithmetic. Readers who are not familiar with the binary number system may prefer to skip it.

Originally the Boolean operators referred only to single-bit quantities, but by convention they are now extended to apply to all the bits of a number, and the same operators are used to describe the process of carrying out the operation bit-by-bit on all the bits of two binary numbers of the same length.

On the BBC microcomputer, all the Boolean operators actually perform bitwise operations on 32-bit binary numbers (that is, the operators act on the full 32 bits of integer variables). Thus 3 OR 6, which is written in binary as

```
011 OR 110
```

gives 111, or decimal 7.

127 AND 159, or in binary

```
01111111 AND 10011111
```

gives 00011111, or decimal 31.

21 EOR 24, or in binary

```
10101 EOR 11000
```

gives 01101, or decimal 13.

NOT gives a full 32-bit complement of a number, so

```
NOT 111 gives 111...11000
```

Readers familiar with the two's complement representation of signed numbers will recognize this as -8 (that is, NOT 7=-8).

This also explains the apparently curious use of -1, rather than 1, for TRUE. TRUE is NOT FALSE, and to 32 bits,

```
NOT 0=111...11111
```

which in two's complement is -1.

Exercise 5.9

Try evaluating some bitwise logical expressions, by printing in direct mode expressions such as those given above. For example

```
PRINT 3 AND 6

PRINT NOT 7
```