Gerald's Column
by Gerald Fitton

I have received a lot of correspondence about my Illustrative Tables article. Nearly all has been complimentary however I have been sent some good suggestions for improvement which I shall share with you. Secondly I have been sent an alternative Minesweeper game. My major topic for this month is an Introduction to Modulo Arithmetic & Algebra. I shall introduce you to 'One Way Functions' and demonstrate a Modulo function which can be used as the basis of a simple encryption system. As an experiment—let me know what you think—I am including on the Archive monthly disc, files which are in Lotus and Excel 4 format so that, whatever spreadsheet you use, you will be able to Load my example files.

Data is . . .

A couple of months ago I wrote about Illustrative Tables. Robert Beech has written to me taking me to task for using Data as a singular noun in sentences such as: "Reference Data is obtained . . ." I hadn't really thought about it but I guess that I was using Data as a collective noun rather like the word Information. Sentences such as: "Information is . ." might refer to many items of information but the word is still treated as singular.

Robert says: "The Financial Times and other good quality publications correctly use a plural verb after 'data'. In education this is insisted on." and "Scientific American is always careful to use a plural verb after 'data'." also "Scientific Style and Format – The CBE Manual for Authors, Editors and Publishers (Cambridge University Press) stipulates 'data' as plural." I have to agree—all my dictionaries indicate that Data is plural and Datum is its singular.

Thousands or Millions

My worked example (of Illustrative Tables) was the two part statement: "The number of licensed vehicles almost doubled between 1961 and 1981. The increase was mainly due to the growth in the number of private vehicles." I included two tables. The first, difficult to read, is that shown below.

Table 1 Motor vehicles currently licensed:
by type of vehicle and year.

United Kingdom thousands

Type of Vehicle	1961	1966	1971	1976	1981
Private cars and vehicles	6114	9747	12361	14373	15632
Motorcycles, etc	1842	1430	1033	1235	1386
Public transport vehicles	94	96	108	115	112
Goods vehicles	1490	1611	1660	1796	1771
Agricultural vehicles	481	478	450	414	373
Other vehicles	206	260	247	300	340

Original Source: Department of Transport

Colin Singleton (yes—it's that man again!) wrote to me suggesting that the second table could be improved by using millions as the units rather than thousands. I have to agree that values such as 6.1 million are much more easily understood than 6100 thousand. Indeed his point is so obvious (now) that I don't know why I didn't think of it myself!

The modified Table 2 is shown below:

Table 2 Private and other vehicles currently licensed: by year (1961 to 1981) and type of vehicle.

United Kingdom millions

Year	Type of Vehicle		All Vehicles
	Private	Other	
1961	6.1	4.1	10.2
1966	9.7	3.9	13.6
1971	12.4	3.5	15.9
1976	14.4	3.9	18.2
1981	15.6	4.0	19.6
Average	11.6	3.9	15.5

Original Source: Department of Transport

Rows or Columns

After some (well deserved?) words of praise John Harrison says:

"I take issue with you on one thing though. You used an invalid example to support your assertion that it is easier to compare numbers aligned vertically rather than horizontally. You should have shown two arrays, one like the top of the RH column and the other with the x and y axes swapped."

I agree that an extra pair of tables would have been a better demonstration of this rule than including my example as part of the body text of the article. I would like you to have another look at the two tables above and compare the first row of Table 1 (Private cars and vans) with the first column of Table 2 (Private). I know that the numbers in Table 2 have been rounded but I assure you that even if they were not they would be easier to compare than the numbers in Table 1.

John continues:

"The example . . . had the additional factor that the numbers to be compared were separated by other numbers and because of the page layout, the string went onto the next line and so it was not even horizontal."

I agree that in the Archive magazine the line wrapped around at an unfortunate place and so didn't help my example. In spite of John's criticism of the format of the example I'm sure that he does agree with the rule: "Put numbers which have to be compared in columns and not in rows" otherwise he would have said so.

Mine Hunt

As well as writing to me about “Data is...” Robert Beech has commented on the MineSweeper game which I included about three months ago. Robert sent me a MineSweeper game written by Paul LeBeau <paul_lebeau@equinox.gen.nz>. It is more flexible than the Resultz version that I included.

You will find Paul’s MineSweeper game on the Archive monthly disc. If you do use and enjoy his game then he asks you to make a donation to your own favourite charity. He retains the copyright whilst making it freely available.

Modulo Algebra

Now I shall treat you to my Introduction to Modulo Algebra. I shall try to capture your interest by describing its use as a simple encryption tool and I shall ask you to try to ‘crack the code’ (just to prove that it isn’t too hard to break).

Modulo Arithmetic is sometimes called ‘Clock Arithmetic’. A question which you might be asked is: “What time will it be 1000 hours from now?” Of course it depends on what the time is now so, for the purpose of this question, let’s suppose that it is midnight. To discover the answer we must divide 1000 by 24 and extract the remainder. The important thing for you to note about this calculation is that the number of times 24 ‘goes into’ 1000 is totally unimportant; what you are trying to discover is what’s left over!

The mathematical function for finding remainders can be found in many spreadsheets. In most the syntax of the function and its two arguments is mod(number,modulus). To find the answer to the question: “What time will it be 1000 hours from now (midnight)?” you enter mod(1000,24). The answer, 16, is returned meaning sixteen hours after midnight.

In BASIC the syntax is <number>MOD<modulus>. So PRINT 1000MOD24 returns 16.

Modulo Algebra has so many interesting uses that I could spend months telling you about them! However, I want to grab your attention so I shall restrain my enthusiasm and concentrate on just one application, encryption.

Before I start let me acknowledge that it was a book called “In Code” by Sarah Flannery which inspired me to use encryption as my example. Sarah is a teenager born in 1982. Her writing is highly readable and her enthusiasm for mathematics is infectious.

Letters or Numbers

The first step in nearly all modern encryption systems is to convert the text of a message into numbers. This is done because it is easier to use mathematical functions to encrypt numbers than it would be, for example, to use lookup tables of characters or phrases.

The words in this article are stored in my computer as numbers. Not only that, nearly all computers encode text using the same or a very similar encoding system. For simplicity I shall use this (nearly) standard text to numbers code in my worked example.

Using this (nearly) standard text to number conversion, characters are stored in an 8-bit code sometimes mistakenly called ASCII—the American Standard Code for Information Interchange originated as a seven bit code. With eight bits we can distinguish 256 separate characters; these are labelled 0 to 255 inclusive. Those codes labelled 0 to 31 are often described as 'Control Characters'. The remaining 224 characters are the 'Printable Characters' which we use in our messages. The coding is nearly standard; there are just a few characters which differ from one character set to another.

In BASIC there is a function ASC() which returns the 8-bit code number of the character which is the argument of the function. For example ASC("A") returns 61. When code 61 is sent to a printer the letter A is printed.

For my simple example I shall encrypt the simple text message "Gerald Fitton". First I shall encode these thirteen characters (don't forget that <space> is a 'Printable Character') using the 8-bit code built into my RiscPC which I shall (incorrectly) refer to as ASCII.

The ASCII code for G is 71 and the ASCII code for a lower case e is 101. In BASIC the function CHR$ can be used to convert these ASCII codes into characters. For example the BASIC command PRINT CHR$(61) will return an upper case "A".

Encryption

Of course, converting the Printable Characters of "Gerald Fitton" into thirteen numbers between 0 and 255 is not much of an encryption system. Indeed, as I have indicated, both the encode and decode functions (ASC and CHR$ in BASIC) are readily available; what is more they are simple to apply. To make the encoded message much more difficult to decode we need to do something more complicated than convert the characters into an 8-bit numerical code! However, complication is not the only consideration as we shall see.

If you want someone to send you a message in code then they must know how to encode their message and you must know how to decode it. Over millennia encryption systems became more and more complex but generally they all had one thing in common. This was that once you knew how to encode a message then it was relatively easy to deduce a decoding technique. Both the encoding and decoding algorithms had to remain secret. Since you might want many people to send you secret messages the possibility of a 'leak' increased with every new recruit to the service.

The system which I am about to describe is different. It is a relatively recent development, only a few decades old. In this more recent system the full details of the method of encoding is made public. By this I mean that anyone can encode a message because the method, a mathematical function, is made available to everyone. To make the encoding specific to the intended recipient it is necessary to enter an 'address'; each potential recipient has a unique 'address'. This 'address' is in the form of a number (or a set of numbers). It is published—so everyone has access to it. This number (or set of numbers) is used as a modifier of the mathematical function so that the encoded form of the message is unique to the recipient.

Today we use computers to encode seriously secret messages. Using the recent system I am describing everyone uses the same published computer program for encryption. They enter their message and the 'address' of the recipient and the program does the rest!

The recipient has their own version of the program but they hold their own ‘address’ in a different form from the public version. When the recipient enters the incoming message and their own version of their ‘address’ the computer program decodes the secret message.

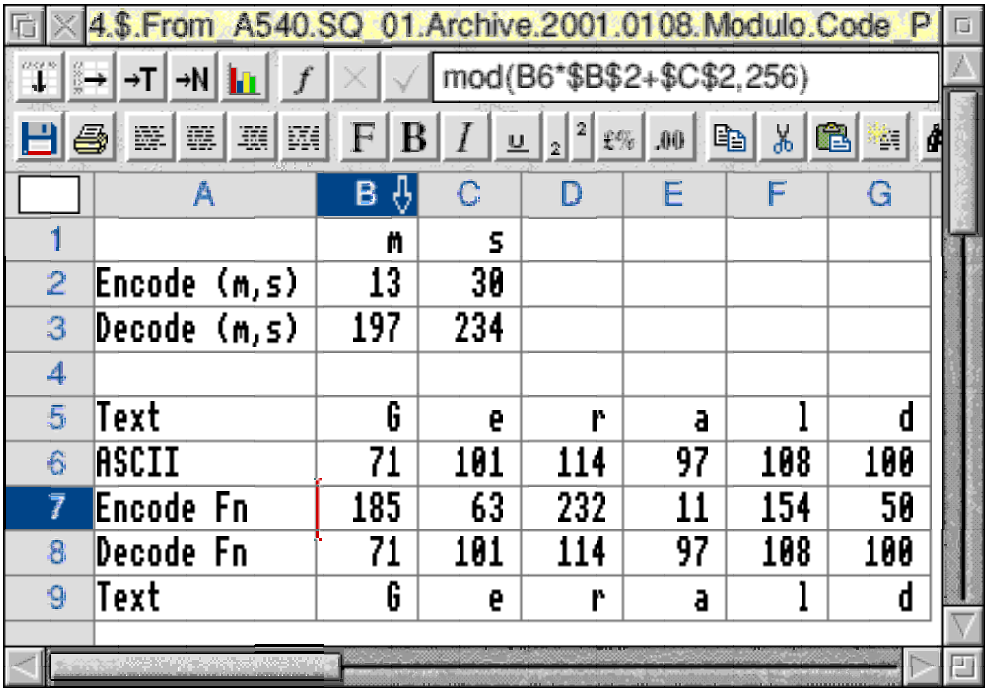
One Way Functions

So what is different about this recent system of encryption? The answer is “One way mathematical functions”. When I was at school most of my Algebra lessons were spent finding the ‘inverse’ of mathematical functions and using the inverse function to solve equations. For example a function such as $y = 2x + 3$ has the solution $x = (y – 3)/2$. This function would be rather useless to convert the coded text, represented by x, to a set of new codes, y, using $y = 2x + 3$ because it is all too easy to invert the equation and hence decode the message. A function which can not be inverted at all is called a ‘One way function’.

A truly ‘One way function’ would be of no use either because it would be impossible to provide the recipient of the encoded message with the means of decoding it. What we want is a function which is a ‘Nearly one way function’. By this I mean that the inverse function does exist but that it is difficult to find it even when you know the encoding function, the ‘address’ and a message in plain and encoded form. By “difficult” I mean that it is practically impossible even using powerful computers to find the inverse function.

Modulo Algebra is the mathematical tool which can be used to provide ‘Nearly one way functions’ of varying complexity. This month I shall give you a simple example which illustrates how it works. I have used a spreadsheet rather than BASIC because that will allow most of you to see what is happening more clearly. The example file is on the Archive monthly disc in PipeDream, Fireworkz, Eureka, Excel and Lotus formats.

Encoding



	A	B	C	D	E	F	G
1		m	s				
2	Encode (m,s)	13	30				
3	Decode (m,s)	197	234				
4							
5	Text	G	e	r	a	l	d
6	ASCII	71	101	114	97	108	100
7	Encode Fn	185	63	232	11	154	50
8	Decode Fn	71	101	114	97	108	100
9	Text	G	e	r	a	l	d

The screenshot above shows part of the spreadsheet in PipeDream format. This spreadsheet encodes “Gerald” in row 5 to the numbers in row 7 of the spreadsheet.

What I have done in row 7 is to multiply the ASCII code from row 6 by a fixed ‘multiplier’, m, and then add another number called the ‘shift’, s, before applying the modulo function using modulus 256. In the example the numbers m and s used for encoding are 13 and 30—you will see them in row 2 of the spreadsheet. The use of 256 as the modulus guarantees that the answer will be a number in the range 0 to 255 inclusive. Let’s check one of these. The upper case G (in B5) has the ASCII code 71. Multiply 71 by the multiplier 13 to get 923; add the shift of 30 to get 953; divide by 256 and find the remainder, 185. This ASCII code could be converted to a character but I haven’t done this.

The encoding function in cell B7 is mod(B6*$B$2+$C$2,256). It is replicated to the end of row 7—all the numbers returned are in the range 0 to 255. Using the x and y notation the encoding function is y = mod(13x+30,256). Finding the inverse of this function is a little beyond A level Algebra (do they still do Algebra at A Level?) so that the algorithm for decoding the message is not easily deduced even with all this information.

Decoding

I hope that you will agree that the use of the modulo function makes it difficult to decode the message even if you know the encoding function. With ordinary arithmetic the inverse of the function y = 13x+30 can be derived easily as x = (y − 30)/13. In Modulo Algebra there is no division available so this method of finding the inverse function does not apply.

What is not generally known except by mathematicians with a knowledge of Modulo Algebra is that if the encoding multiplier is chosen carefully then a unique inverse function can be found. Indeed the unique inverse function is also of the form x = mod(m*y+s,256) but the multiplier and the shift values are different from those used to encode the message.

The function in cell B8 is mod(B7*$B$3+$C$3,256) and it is replicated to the end of row 8. The decoding values used for m and s in my example are m = 197 and s = 234. This corresponds to the decoding function x = mod(197y+234,256). For example starting with the value 185 (the encoded G) the value of x is mod(197*185+234,256). This decoding function returns the original 71, the ASCII code for G.

Nearly One Way Functions

My example is a simple one but demonstrates a principle.

I can give you the formula y = mod(mx+s,256) and the two values m and s which define the function you must use to encode your message to me. I can make the formula, the multiplier, 13, and the shift, 30, known to everyone. If I keep secret the two values, 197 and 234, which I use to decode the message then I have a certain degree of security.

Anyone can encode a message to me but only I can decode it. Well that’s not quite true because my simple example can be cracked fairly quickly by anyone who knows enough about Modulo Algebra to be able to invert the original function and hence find the two keys (my m and s) that I am trying to keep secret.

The encoding method I have used is too simple for serious use. It is the principle that I am trying to get across to you. I make an encoding algorithm available to anyone who wants to communicate with me secretly and I keep secret the key values which allow me to decode the message. The encoding function can be made more sophisticated and hence more secure. I have used only multiplication and addition but we could try (another day) the effect of raising numbers to powers.

We have to be able to find a pair of functions one of which is the inverse of the other but we have to do this in such a way that it is almost impossible to derive one from the other. I shall postpone describing a method of finding such a pair of functions until another day because I am running out of space and because there is something I want to say now.

Key Space

With the number 256 as the modulus the multiplier, m, can be any odd number between 1 and 255 inclusive. Even numbers don't work—try it and you'll see why. The shift can be any number between 0 and 255 inclusive. That means that this particular coding system using a modulus of 256, a multiplier m and a shift s can give 256*256/2 = 32768 different codes. Thirtytwo thousand codes is not a lot and if everyone wanted a unique code (and they do) then we should soon run out of codes. The numbers used to encode and decode are called the 'Keys' and the set of all Keys is called the 'Key Space'. A Key Space containing only 32768 Keys (well only half this because they come in pairs) is too small for practical use. In any case, with only a small Key Space (even) your RiscPC computer will rattle through all the possibilities in a few minutes. A real encoding system has to have a much larger Key Space to deter proponents of this Trial and Error technique.

A Puzzle

I have constructed a spreadsheet which finds the inverse of every multiplier. For example I can tell you that the first seven pairs are (1,1), (3,171), (5,205), (7,183), (9,57), (11,163) and (13,197). By 'pair' I mean that either one of the numbers of the pair can be used to encode and the other number is the decoding multiplier. These pairs are independent of the value chosen for the shift. The decoding shift depends on both the encoding multiplier and the encoding shift. For example the decoding key corresponding to (7,30) is (183,142).

So my puzzle to you is this: "What is the decoding multiplier corresponding to 127?" When you have done that one try to find the inverse multiplier for 99. As a hint I suggest that you try a zero shift because the inverse of a zero shift is always a zero shift. Then find out the decoding shift corresponding to s = 30 for both multipliers, 127 and 99. Hint for finding the decode shift value: What does ASCII code zero map to?

If you have either PipeDream, Fireworkz, Eureka, Excel or Lotus then, just for you, I have included a version of the file shown in the screenshot on the Archive monthly disc. I haven't included a Schema version because I couldn't find the mod(,) function—is there one? Enter zero in cells C2 and C3. Enter 127 in cell B2. Then experiment with values in B3 until the text line decodes correctly, I assure you it won't take long. If you are feeling very mathematical then try to find an algorithm which returns the decode multiplier when you enter the encode multiplier. It can be done and, hint, it has something to do with an extension of Euclid's method for finding the highest common factor of two numbers!

Finally

You can write to me at the address given in Paul's Fact File. I do give help with spreadsheets including Eureka, Schema, Fireworkz and PipeDream. I can even load files into Excel and Save them in a format which can be loaded into a RISC OS spreadsheet.

If you do need some help then please let me have an example file which you have created; it saves me so much work and reduces the chances of misunderstanding your problem. Return postage and a self addressed label will be appreciated.