AAEC/S8

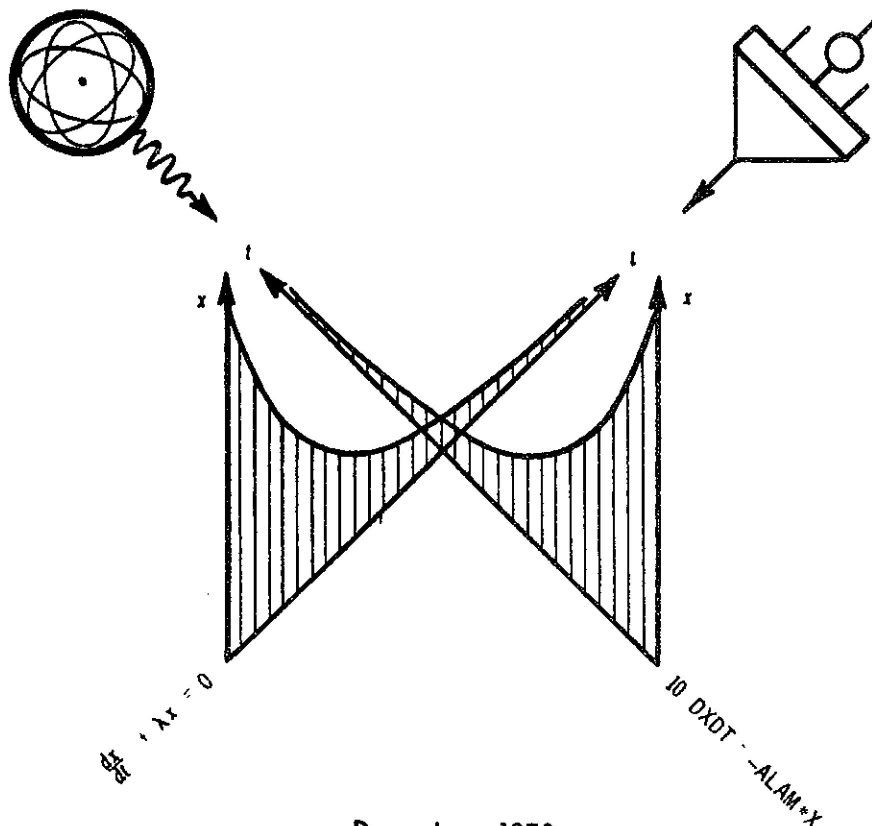
AUSTRALIAN ATOMIC ENERGY COMMISSION
RESEARCH ESTABLISHMENT
LUCAS HEIGHTS

NUMERICAL MATHEMATICS AND FORTRAN

REACTOR PHYSICS, MATHEMATICS AND COMPUTERS

SUMMER SCHOOL FOR TEACHERS

Lecture by J. POLLARD



December, 1972

NUMERICAL MATHEMATICS AND FORTRAN

by

J. POLLARD

ABSTRACT

In these notes the computer is introduced via a real problem, although in miniature, and we see how a computer becomes part of a mathematician's problem solving kit. To communicate with a computer we need a language and FORTRAN, a time-proven language, is introduced.

CONTENTS

		page
1.	ELECTRONIC DIGITAL COMPUTERS	1
2.	MATHEMATICAL PROBLEM SOLVING	2
	2.1 Problem Identification	2
	2.2 Mathematical Description	3
3.	FORTRAN	4
	3.1 Punching Detail	4
	3.2 Number Representation	5
	3.3 Variables	6
2.	MATHEMATICAL PROBLEM SOLVING continued	7
	2.3 Numerical Analysis for Job (A)	8
	2.4 Computer Programming	11
	2.5 Programme Checkout	11
	2.6 Production Runs	11
	2.7 Interpretation	12
3.	FORTRAN continued	12
	3.4 Input and Output, etc.	12
	3.5 Operations and Expressions	15
	3.6 Arithmetic Statements	17
	3.7 Mathematical Functions	18
	3.8 Transfer of Control	19
	3.9 Do Loops	21
	3.10 Arrays of Variables	23
4.	PRACTICE EXAMPLES	25
5.	ANSWERS AND TYPICAL CODING	27
6.	FURTHER READING	29
7.	ACKNOWLEDGEMENTS	29
APPENDIX 1	Some analysis for those who can stand it	
APPENDIX 2	Punching convention(s)	
APPENDIX 3	Control cards	
FIGURE 1	A(n im)possible reactor response to a power surge	
FIGURE 2	A possible reactor response to a power surge	
FIGURE 3	A possible reactor response to a (negative) power surge	
FIGURE 4	A graphical illustration of the slide rule calculations	

1. ELECTRONIC DIGITAL COMPUTERS

Electronic digital computers are devices capable of making elementary decisions, such as 'is A>B?', as well as carrying out arithmetic operations such as addition, subtraction, multiplication and division at high speed. The great speed of computation, typically a few microseconds per operation (1 microsecond = 10^{-6} sec), is only realizable because both numbers (data) and instructions (programme) may be accessed from the main (core) memory in a few microseconds. The machine does not wait on us to enter the next instruction, rather we enter a complete set of instructions, called a programme, to solve the problem we are tackling. Our programme and data ('input') are normally punched on 80 column computer cards using a card punch machine not connected to the computer ('off-line') and the deck of cards (a 'job') is entered through the card reader connected to the computer ('on-line'). (Normally our job joins a queue of jobs prepared by others and we must wait a few minutes, or a few hours if the machine is really busy.) Results of our calculation ('output') are usually printed by the fast printer on-line to the computer, although many different devices ('computer hardware') may be connected to the machine. Normally we do not write the programme in the language of the machine,(which differs from model to model) and typically appears like this to a binary machine.

0100000110010000000000000010100 11

but rather we use a computer independent, general language, such as FORTRAN. Smart programmes, called compilers, are normally provided by the machine manufacturer to translate a general language to machine instructions. Compilers for several different general languages may be provided with a machine - these are called 'computer software'.

The range of devices attached to the computer varies with the particular application. It is possible to employ the computer to control the overall operation of a plant, for example a nuclear reactor power station. Continuing with this example, we could have output from thermocouples, flow meters, etc., connected to the computer and if, say, the power output (in the form of heat) from the nuclear reactor varied outside certain limits, motors could be turned on by the computer to adjust valves, control devices, etc. This type of application is at present expanding. The use of computers in the commercial world to process accounts, salaries and stock inventories is possibly familiar to you. The application we will have in mind is mathematical problem solving which embraces a variety of professional disciplines.

Computers are used in mathematical problem solving because of their ability (1) to operate at high speed, (2) to produce accurate reliable results, (3) to store large quantities of information, and (4) to carry out long and complicated sequences of operations without human intervention. Typical of the computers of today

(1) we may multiply two numbers in a few microseconds,

(2) the result may be accurate to 7 (single precision) or 16 (double precision) decimal places,

(3) we may store 10^5 (single precision) numbers in the main (core) memory of the machine and perhaps 10^7 numbers on fast access disks attached to the machine with a further 10^9 numbers, or more, on magnetic tapes stored within the vicinity of the machine,

and (4) we may confidently carry out computations lasting several hours.

2. MATHEMATICAL PROBLEM SOLVING

Several well defined steps go to make up mathematical problem solving with a computer. Here we are content to briefly look at each step in turn so that we get some feel for the way computers are used by scientists and engineers.

2.1 Problem Identification

Considering a nuclear reactor, for example, the first question that perhaps comes to the mind of a non-reactor specialist is 'will it blow up?' Even reactor specialists consider this question during preliminary design assessments of a possible reactor, but of course only the safe reactors are ever built. These reactor specialists need to study the physical processes occurring in a reactor in order to make precise the vague term 'blow-up'. This is the sort of thing involved in problem identification.

Continuing with the above question (and we will do this throughout this work), physicists analysing the reactor may be able to provide an equation for the power produced from fission in the fuel (the process that makes the reactor go) as a function of time, say $p(t)$. Then if we have the result depicted in Figure 1, the reactor will blow up, if we have the result depicted in Figure 2 the reactor will return to normal power provided the power surge does not melt the fuel, and if we have the result depicted in Figure 3 the reactor will shut down. We have thus obtained a quantity $p(t)$ which helps make precise our

originally vague idea.

2.2 Mathematical Description

Following a temporary disturbance to a nuclear reactor, the equation for $p(t)$ is too complicated for us to consider here, however making many simplifying assumptions we arrive at the first order (non-linear) differential equation

$$\frac{dp}{dt} = -(p-1-be^{-t})p, \quad p(o) = 1, \quad (e = 2.718282), \qquad \ldots(1)$$

where b represents a parameter measuring the disturbance and we require the solution $p(t)$ for $0 \leq t \leq 10$, say, for particular values of b. (To impart some reality to the problem we may take the unit of power, p, to be 1000 Megawatts (10^9 watts) and the unit of time, t, to be seconds.) Of particular interest is the quantity.

$$h(b) = \max_{0 \leq t \leq 10} p(t) \;, \qquad \ldots(2)$$

the maximum power obtained following the disturbance b, since our reactor fuel may start to melt if this quantity becomes too large, say

$$h(b) > 1.2 \;. \qquad \ldots(3)$$

Returning to the differential equation we note that

$$\left.\frac{dp}{dt}\right|_{t=o} = b \;, \qquad \ldots(4)$$

hence initially the power will rise if $b>0$, however

$$\text{if} \quad p>1+be^{-t}, \quad \text{then} \quad \frac{dp}{dt} < 0 \qquad \ldots(5)$$

and the power will drop. The reactor thus won't blow up, but its fuel may melt. Having satisfied ourselves with the question we originally proposed we have been led to consider a new problem. (This is the way of most scientific problem solving.) We are not so vague now, as we have already developed the ideas. The question 'will the reactor fuel melt?' is equivalent to 'is condition (3) satisfied?' All we need do is

(a) solve equation (1) for a given value of b,

(b) calculate h(b) from equation (2),

and (c) apply condition (3)

- but how do we proceed? As you may guess we intend to solve equation (1) using a computer. We will then need some more mathematics (Numerical Analysis) and a suitable computer language (FORTRAN). As I guess you may have had enough maths we will postpone the remainder of Section 2 until later. Now for training in FORTRAN.

3. FORTRAN

Although FORTRAN is a general language available on most computers, differences do exist between different versions. We will stick to the simplest form of FORTRAN IV and we will only introduce enough of the language to enable us to write (compose) simple programmes for the IBM 360 computer. As we aspire to write bigger and better programmes we must dig out a book most suited to our needs (see Section 6 - further reading).

3.1 Punching Detail

Punching the cards of a FORTRAN programme is simplified if we legibly write our programmes on specially ruled FORTRAN coding forms as sketched

1 5	6	7 FORTRAN STATEMENT	72
C		J.POLLARD - REACTOR POWER SURGE JOB - JAN 71	
C		POWER IN 1000MEGAWATTS, TIME IN SECONDS	

Each line of the form, which contains only one statement, represents a new card. Coding normally proceeds from column 7 to column 72 although a statement number may be added in columns 1 to 5, for example

```
   20 TIME = 0.
```

From column 7 on, blanks may be used to make the statement more readable - these blanks do not change the meaning given to the statement. In addition, comments, which are printed along with the rest of the statements but are otherwise ignored by the compiler, may be interspersed throughout the deck to label certain features of a section of coding. A comment is a card with a 'C' in column 1 as indicated in the sketch above. Normally we should start our job with a comment

(or two) to identify the deck of cards. Comments certainly help us pick up the threads of a programme at a later date.

The character set permitted with the FORTRAN code, and available on almost all computers, consists of

(i) 26 capital letters of the alphabet,
(ii) 10 numbers 0 to 9,
(iii) 9 special characters + - * / . , ( ) =
and (iv) a blank (usually written b if the presence of a blank is to be emphasised).

3.2 Number Representation

There are three modes of numbers most commonly used with the IBM 360 ....

(1) fixed point numbers, which are signed integers ($\leqq$ 9 digits) written without a decimal point, for example
-123, 12, -99999, 0, 357 (the plus sign is not required),

(2) floating point single precision numbers, which are signed decimal numbers ($\leqq$ 7 digits) written with a decimal point either given with, or without, an exponent, for example
-123., 1.23E+2, 0.9876543E-12, 0., -57.000,
where 1.23E+2 = 1.23×10^2, etc. The magnitude $|A|$ of a floating point number, A, must lie within the range $10^{-78} < |A| < 10^{75}$, or be zero,

and (3) floating point double precision numbers which are similar to (2) but either (i) written with a D to designate the exponent or (ii) have between 8 and 16 digits
-123.D0,1.23D+2, 0.987654321D-12,0.D0,-57.000D0,2.7182818285, 3.1415926536
(We will make no use of double precision numbers in this work although they are important in mathematical problem solving when, for example, number of nearly equal size, say with 5 figures the same, are to be subtracted.)

Numbers may be used as

(a) statement numbers or indices (unsigned fixed point numbers)

```
e.g. 21 CONTINUE
     ↑
       X(1) = XYZ
         ↑
```

(b) constants in a programme

```
e.g. DTIME = 0.1
              ↑
```

and (c) input data which form the last cards of the deck

```
e.g. 0.5
     ↑
C    HAVE A BREAK
C    ********************************
```

3.3 Variables

FORTRAN variables are labelled with up to 6 character names (special characters are not permitted) of which the first character must be alphabetic, for example,

TIME

T

I5

A3B

Each variable designates a unique portion (word) of main (core) memory and reference to a variable implies reference to the number which appears in the memory assigned to the variable. If we move a number into TIME (of course we mean into the memory set aside for this purpose) then we replace any number previously resident there, however if we move a number from TIME we only take a copy, as the number remains intact. Thus

```
|          ||TIME = 0.
```

puts a floating point (single precision) zero into the memory assigned to TIME, and

```
|          ||    X = TIME
```

puts a copy of the latest number available in TIME into the memory set aside for X without destroying the contents of TIME.

Variables are designated either

(1) fixed point, or
(2) (single precision) floating point

depending on the first letter of their name.

(1) Fixed point variables start
I,J,K,L,M, or N.

For example JOB, IN2 and NUM and they contain fixed point numbers. We may thus have

```
|      ||  NUM = -1
```

which assigns the fixed point number -1 to the memory set aside for NUM.

(2) Floating point variables start with any letter except one from the list (1) above. For example POWER, TIME and H3. We may thus have

```
|      ||DTIME = 0.1
```

which puts the floating point number 0.1 into DTIME.

```
C    NOW BACK TO SECTION 2
C    AND MORE MATHS
```

2. <u>MATHEMATICAL PROBLEM SOLVING</u> continued

You no doubt remember that we had obtained a differential equation 2.2(1) for reactor power, p, as a function of time, t, following a disturbance, b, to normal reactor operation. The question we want answered is

(A) 'will the reactor fuel melt if $b = 0.5$?'

given that we may check our theory and method of solution against the (imagined) experimentally measured result

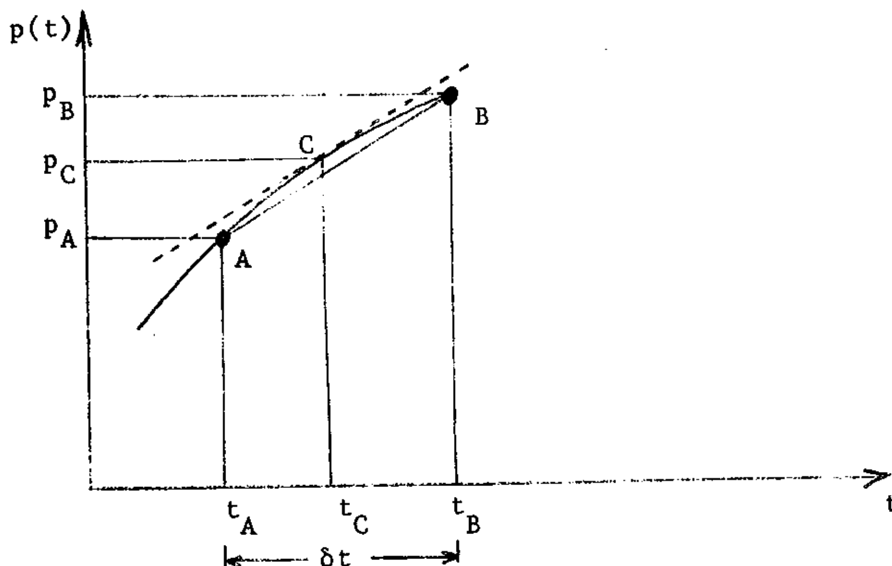
(B) $h(0.3) = 1.12 \pm 0.01$.

We cannot answer the question until we find out how to solve equation (1). We are thus led to consider a branch of mathematics oriented to numerical solution of equations.

2.3 Numerical Analysis for Job (A)

Here we will only consider one aspect of one part of Numerical Analysis - that which is concerned with solution of differential equations and we will follow a line which introduces ideas as quickly as possible.

Say, by some means or other, we know the solution we seek up to the point A, as sketched, and we wish to extend the solution to the point B corresponding to the given time t_B.



From the sketch we note that

the slope of the chord AB = the slope of the curve at some point C between A and B

$$\frac{p_B - p_A}{\delta t} = \left.\frac{dp}{dt}\right|_C = \left(\frac{dp}{dt} \text{ evaluated at } t = t_C\right) . \qquad \ldots(1)$$

We may assume that the slope of the curve at C is some sort of average of the slopes at A and B

$$\left.\frac{dp}{dt}\right|_C = \left.\frac{dp}{dt}\right|_A (1-\theta) + \left.\frac{dp}{dt}\right|_B \theta , \quad 0 \leq \theta \leq 1, \qquad \ldots(2)$$

$$\left(\theta=0 \rightarrow \left.\frac{dp}{dt}\right|_C = \left.\frac{dp}{dt}\right|_A , \; \theta=\frac{1}{2} \rightarrow \left.\frac{dp}{dt}\right|_C = \frac{1}{2}\left(\left.\frac{dp}{dt}\right|_A + \left.\frac{dp}{dt}\right|_B\right) , \theta=1 \rightarrow \left.\frac{dp}{dt}\right|_C = \left.\frac{dp}{dt}\right|_B\right)$$

then equation (1) becomes

$$p_B = p_A + \left[\frac{dp}{dt}\bigg|_A (1-\theta) + \frac{dp}{dt}\bigg|_B \theta\right] \delta t \; . \qquad \ldots(3)$$

Now we know $\frac{dp}{dt}\Big|_A$ from the differential equation we are seeking to solve (equation 2.2(1)),

$$\frac{dp}{dt}\bigg|_A = -\left(p_A - 1 - be^{-t_A}\right) p_A, \qquad \ldots(4)$$

and if we take $\theta=0$ in equation (3) we get a 'predicted' value for p_B,

$$p_B' = p_A + \frac{dp}{dt}\bigg|_A \delta t \; , \qquad \ldots(5)$$

which we may use to estimate $\frac{dp}{dt}\Big|_B$,

$$\frac{dp}{dt}\bigg|_B = -\left(p_B' - 1 - be^{-t_B}\right) p_B' \; . \qquad \ldots(6)$$

Choosing $\theta=\frac{1}{2}$, corresponding to taking the usual average of the two slopes, and substituting the values given by equations (4) and (6) into equation (3) then gives us a 'corrected' estimate of p_B.

Let us see how we use the method just discussed - the so-called Heun method. We may start the method at $t_A = 0$ since we are given $p(o) = 1$. As an illustration we take $b = 0.3$ and $\delta t = 0.1$ and we apply the method using arithmetic performed with a slide rule. We repeat here the equations we will use in the form of a computational procedure.

(C) $\delta t = 0.1$

$t_A = 0$

$p_A = 1$

$b = 0.3$

$$\left.\frac{dp}{dt}\right|_A = -\left(p_A - 1 - be^{-t_A}\right) p_A \quad \ldots(4)$$

$$t_B = t_A + \delta t$$

$$p'_B = p_A + \left.\frac{dp}{dt}\right|_A \delta t \quad \ldots(5)$$

$$\left.\frac{dp}{dt}\right|_B = -\left(p'_B - 1 - be^{-t_B}\right) p'_B \quad \ldots(6)$$

$$p_B = p_A + \frac{1}{2}\left(\left.\frac{dp}{dt}\right|_A + \left.\frac{dp}{dt}\right|_B\right)\delta t \quad \ldots(3)$$

$$t_A = t_B$$

if $t_A \geqq 10$ finish

$$p_A = p_B$$

go to (4)

(D)

t_A	p_A	$1+0.3e^{-t_A}$	$\left.\frac{dp}{dt}\right\|_A$	t_B	p'_B	$1+0.3e^{-t_B}$	$\left.\frac{dp}{dt}\right\|_B$	$\frac{1}{2}\left(\left.\frac{dp}{dt}\right\|_A + \left.\frac{dp}{dt}\right\|_B\right)$	p_B
0	1	1.3	0.3	0.1	1.03	1.272	0.250	0.275	1.0275
0.1	1.0275	1.272	0.252	0.2	1.0527	1.246	0.203	0.227	1.0502
0.2	1.0502								

A graph of the above results is given in Figure 4.

Numerical Analysis is also much concerned with the study of

(1) the error incurred by using an approximation such as the Heun method

and (2) the effect of roundoff on the calculation, for example referring to the calculation above

$$p_A = \underline{1.0275X}$$

may have an error in the 6th significant figure (indicated by the X) but

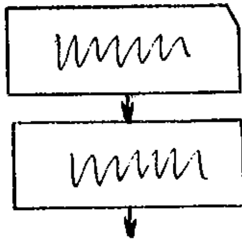
$$p_A - 1 = \underline{0.0275X}$$

would then have an error in the 4th significant figure.

We must leave the subject here, but if you want to dig deeper see Section 6 - further reading.

2.4 Computer Programming

Before we start to write a computer programme we set out the steps we want to use in the form of a computational procedure, as we did in the previous section (C). Some people put boxes around the steps and join them to produce what is called a flow diagram ....



If this helps you by all means proceed this way, although I must add that I never draw a flow diagram as such.

We then write a programme in FORTRAN to solve the problem following the steps we have set out in our computational procedure (or flow diagram).

2.5 Programme Checkout

'Do not kick the beast-
-It did its best.
Did you thihk to test,
Your programme first?'

We should consider every programme to be wrong, that is to contain mistakes ('bugs'), until it is proved to be otherwise. This attitude to computers is a must if we are to use them with any confidence. Removing any mistakes is easy, but finding them ('debugging') is a different matter. In answering question (A) ('will the reactor fuel melt?') we should first choose b=0.3 and check our results against table (D) and then experimental result (B). Until agreement is satisfactory there is no point proceeding.

2.6 Production Runs

When we are satisfied with our programme we run all of the cases we wish to study. Here we only have one production run in mind, corresponding to b=0.5, however normally we would run lots of cases (b=0.45, 0.475, etc.) to get a feel of how sensitive our answer is to the temporary disturbance, b.

2.7 Interpretation

If the reactor fuel melts, equation 2.2(1) may no longer hold. 'Oh, will the reactor blow up?' And so it goes with mathematical problem solving.

```
C       HAVE A REST
C       ********************************
```

3. FORTRAN continued

We have so far only been introduced to constants and variables which is hardly sufficient repertoire from which to compose a programme - so on with the job.

3.4 Input and Output, etc.

Input to a programme may consist of numbers punched on cards, for example (each line denoting a card)

```
0.45
0.475
0.5
0.525
0.55
```

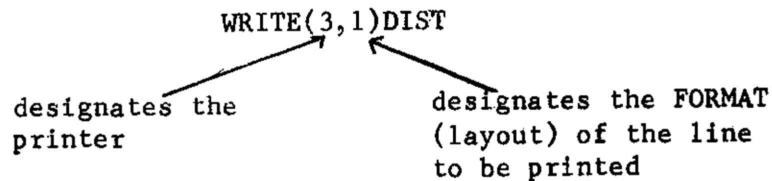
and the cards would follow the last statement of the programme. For example in our nuclear reactor problem if we wanted to study several cases for b=0.45, 0.475, etc., we could enter one value using

```
READ(1,1)DIST
```

designates the card reader (first 1) — designates the FORMAT (layout) of the card, (second 1)

where DIST is the variable name associated with the disturbance, b. The first time we have the READ statement we would get the equivalent of DIST=0.45, however if we used the above statement we would have to change the card to DIST=0.475 to run the second case and this would mean resubmitting the job to be run a second time on the computer. (This would be particularly upsetting if we had to wait for 50 other jobs to be processed before we could get our second run.) As we will see we can arrange for the same READ statement to be encountered over and over again in the flow of the programme as it runs on the machine - the second time we would get the result DIST=0.475, then DIST=0.5, etc., until all data are used.

Similarly if we want to write (print) numbers on the line printer we could use



Every time the above statement is encountered as the programme runs on the machine the latest number stored in DIST is printed on a fresh line.

Summarising input and output ('I/O') we have

READ } (unit, format) list
WRITE }

where (i) unit = 1 designates the card reader
= 2 " " " punch
= 3 " " line printer
= etc. " disk storage

(ii)format= statement number of a coded statement describing the layout of data and we will only use the statement

```
1 FORMAT(4E15.5)
```

(iii) list = a list of required data, for example we may have

```
WRITE(3,1)TIME,POWER,HPOWER,DIST
```

When the above FORMAT is used to describe input layout, say we use the statement

```
READ(1,1)A,B,C
```

then 3 numbers may be punched on each card

A punched in columns 1 to 15 (the 1st field)
B " " " 16 to 30 (the 2nd field)
C " " " 31 to 45 (the 3rd field)

Within each field a number may be punched anywhere (blanks are interpreted as zeros in this application) except that an exponent, if given, must end on the last column of the field. Thus we might have as data

```
1               15,16              30,31              45
| 0.5             | -1.2             |            0.01 |
        1.23E-3   |      -4.598E12   |    0.           |
         -1.E-1   |                  |    1.23     E-3 |
                  |field of all blanks|    blanks      |
                  interpreted as 0.     interpreted as zeros
```

We have nearly learnt enough FORTRAN to write a simple programme. Before we do, however, we must appreciate that all of the programme is compiled before our job runs. We designate the end of our programme (or programme segment - sub-routine or function) with the statement

```
END
```

which tells the compiler that all of the programme (or segment) has been processed. Our job is then set up to run ('link edited') and then at last run. The job starts on the first executable statement (comments are ignored) and continues onto the statements that follow one after the other until the programme flow is diverted with a transfer of control statement, for example

```
GO TO 20
```

which sends the programme back, or on, to the statement numbered 20.

We can now write a programme to list (print) a deck of cards (punched according to our standard FORMAT) with 3 numbers punched on each card with the last two numbers to be interchanged. We could write...

```
C          J.POLLARD      CARD   COPYING   JOB     JAN 71
      1 FORMAT(4E15.5)
     10 READ(1,1)A,B,C
        WRITE(3,1)A,C,B
        GO TO 10
        END
0.1              0.2              0.3
0.4              0.5              0.6
0.7              0.8              0.9
```

See Appendix 3 for other cards that you must supply to keep the 'operating system' going. By operating system is meant a control programme that supervises the entire business of compiling, linking, running of your job, then the next persons job, etc. In principle the system should throw off my job when (if!) my programme goes haywire without the next job in line being fouled up. After all, the next job could be yours!

In our example we finish after reading through all of the punched data cards - the system will make sure we do not start to read through the next person's job.

3.5 Operations and Expressions

Five arithmetic operations are available with FORTRAN and each is indicated in the following fashion...

(1) addition + e.g. A + B

(2) subtraction - e.g. X - Y

(3) multiplication * e.g. 2 * I (2 times I)

(4) division / e.g. C/D ($\frac{C}{D}$)

(fixed point division results in truncation i.e. 5/3 = 1)

(5) exponentiation ** e.g. W**3 (W^3)

Brackets are permitted as in everyday algebra toenable expressions to be given a unique meaning, for example,

(A+B) * (C-D)

(A+B) ** 2 ($(A+B)^2$)

A/(C*B) ($\frac{A}{C \text{ times } B}$).

Brackets are also necessary to prevent two operation symbols from appearing one next to another, for example,

X * - Y must be coded X*(-Y)

to indicate correctly to the compiler the intention of the expression. To simplify the coding of expressions the compiler uses a hierarchy for grouping a sequence of operations. The hierarchy is, from highest to lowest,

(1) **
(2) * and /
and (3) + and -

Thus the expression

```
X+(Y/A) - (3.*U) + P*(S**4)
```

may be abbreviated to

```
X + Y/A - 3.*U + P*S**4
```

For a sequence of operations of the same hierarchy the compiler groups the sequence from the left. The expression

```
1./A*B
```

is thus calculated as

```
(1./A)*B  i.e. B/A
```

not 1./(A*B)

An expression should consist of variables or constants all of the same mode, i.e. the mode should not be mixed. An exception to this rule is that the exponent of a floating point variable or constant may be a fixed point variable or constant. The following are permitted forms of exponentiation...

```
(1) V**2
(2) (-V)**0.45
(3) V**(-2)
(4) V**A
(5) V**(-I)
(6) I**3
(7) 10.**2
```

Form (7) above would not normally be used as the computer would be required to compute 10^2 - this constant would be available for direct use if it were written 1.E+2, or simply 100.

Trouble would arise in the calculation implied in the form (2), unless V were negative, as otherwise the result would not be real. In situations such as this, an error message is printed as the programme runs.

3.6 Arithmetic Statements

Arithmetic statements have the general form 'variable = expression',

e.g. A = (B+C)/E

The equal sign should be interpreted as 'has stored in it the result of the calculation' and is not to be used in the everyday algebraic sense. The above statement thus reads A has stored in it the result of the calculation (B+C)/E. Using this interpretation a statement such as

```
X = X*0.1
```

does not imply that X = 0, but rather states that the old value of X is reduced to 1/10th of its value to form a new value of the variable still called X. A statement such as

```
A*E = B+C
```

has no meaning, as A*E is an expression not a variable. When the result of a calculation is stored in a variable, or at least the magnetic core assigned to this variable by the compiler, the previous contents are lost. Use of variables on the right of an arithmetic statement does not result in any change of their value. As an example consider the two arithmetic statements

```
A = 10.
A = A+A/10.
```

On completion of the second statement A would have the value 11.

The mode of the variable on the left need not be the same as the mode of the expression on the right. The compiler will arrange for a change of mode to be carried out before the result of the calculation is stored away. Thus A = I+1 is permitted. On account of the truncation of fixed point division, i.e. the decimal part is lost, care should be used with this type of arithmetic.

For example

```
      A = 5/3
```

would store the result 1. in A

and

```
      A = 5./3.
```

would store the result 1.666667 in A.

This truncation may be used to advantage as shown in the example

```
      J = I-2*(I/2)
```

where J will be 1 if I is odd and 0 if I is even.

A statement number may be used to provide a reference to a particular statement to enable the programme to return to this point if required. Statement numbers need not be given, and if given they do not have to be given in any particular order - they must however be unique. Remember that when a statement is punched, the first character should not appear in columns 1 to 5, which are reserved for a statement number, or column 6, which should be left blank. Emphasising these points, the above example may be written

```
  219 J = I-2*(I/2)
```

3.7 Mathematical Functions

Certain widely used mathematical functions are available as part of the normal FORTRAN system. To use one of the mathematical functions it is necessary to follow the function name by the argument enclosed in brackets. The result is returned as if the function name designated a variable of the programme. For example, the name for the exponential function is EXP so that to calculate

$$y = e^x$$

we may write

```
      Y = EXP(X)
```

or say we wish to calculate

$$v = ae^{x}+be^{y}$$

we may write

```
V = A*EXP(X)+B*EXP(Y)
```

In the above examples we have taken the argument to be a variable, but we may take an arithmetic expression, say

$$w = e^{ax+by}$$

then we may write

```
W = EXP(A*X+B*Y).
```

A list of the most frequently used functions follows.

Mathematical function	Function name (argument)
Square root, $\sqrt{x}$	SQRT(X)
Exponential, e^{x}	EXP(X)
Natural logarithm, $\log_e x$ (or $\ln x$)	ALOG(X)
Sine of an angle in radians, $\sin x$	SIN(X)
Cosine of an angle in radians, $\cos x$	COS(X)
Arctangent (the angle is given in radians), $\tan^{-1}x$	ATAN(X)
Absolute value (floating point), $\lvert x \rvert$	ABS(X)

Other functions are available and still others may be written in FORTRAN and submitted to be compiled along with the main part of the programme. We will not pursue the details however.

3.8 Transfer of Control

Execution of a programme normally proceeds from the first statement, to the second statement, etc. Programmes which do not require sets of instructions to be repeated over and over again are seldom worth coding. As we have seen it is possible to transfer out of the sequence of one instruction followed by the next using statements which transfer control to any statement labelled with a statement number punched in columns 1 to 5 of the source card.

The simplest transfer of control always causes a transfer and is called an 'unconditional GO TO'. For example

```
GO TO 10
```

will cause the statement numbered 10 to be executed immediately after this statement.

An extension of this type of statement is the 'computed GO TO' which transfers control depending on the value of a fixed point variable. For example

```
GO TO (12,1,1,973), I
```

In this example (1) if I = 1 then control is transferred to statement 12
(2) if I = 2 " " " " " " 1
(3) if I = 3 " " " " " " 1
(4) if I = 4 " " " " " " 973
(5) if I is any other integer then control is transferred to the next statement in sequence.

Certainly the most important statement in controlling the logical flow of a programme is the 'logical IF'. The form of the statement is

IF (logical expression) any executable statement

for example

IF(A.EQ.B) GO TO 10 (if A is equal to B, go to 10).

If the logical expression is true then the appended executable statement is carried out, otherwise the next statement in line is executed. The possible forms of logical expression are made up of the following relational operators

a.EQ.b	a is equal to b	$a=b$
a.NE.b	a is not equal to b	$a\neq b$
a.GT.b	a is greater than b	$a>b$
a.GE.b	a is greater than or equal to b	$a\geqslant b$
a.LT.b	a is less than b	$a<b$
a.LE.b	a is less than or equal to b	$a\leqslant b$

where 'a' and 'b' are arithmetic expressions, for example 'a' might be SIN(X)/X. In addition we have the following possible logical operators

this.OR.that   either this is true <u>or</u> that is true
this.AND.that   both this is true <u>and</u> that is true

where 'this' and 'that' are logical expressions, for example 'this' might be (SIN(X)/X) GT.Y As a complete example, say

if $0.999 \leq A \leq 1.001$

then we want to set A=1. We could use the coding

```
IF(A.GE.0.999.AND.A.LE.1.001) A=1.
```

Here if the logical expression is true then the appended statement is executed and then the next statement in line follows as the appended statement is not a transfer of control.

```
C
C       WE HAVE NOW BEEN INTRODUCED TO ENOUGH OF THE FORTRAN LANGUAGE TO WRITE A
C       PROGRAMME FOR THE REACTOR POWER SURGE JOB
C       ********************************
```

3.9 <u>Do Loops</u>

Summation is often required to be carried out when using a computer as it forms the basis of the many mathematical procedures including numerical integration. The DO statement, for example,

```
DO 2 I = 12, 20, 3
```

sets I = 12 when this statement is encountered and carries out all statements down to and including statement 2. The fixed point loop index, I, then has 3 added to it, and provided $I \not> 20$ the loop repeats from the statement after the DO, otherwise the statement after 2 will be executed. The unsigned fixed point numbers in the above example 12, 20, 3 may be replaced by fixed point variables to give for example

```
DO 99 JOB = 21, NEXT, INCR
```

and the last argument may be left out if an increment of 1 is required. Thus

```
DO 2 I = 1,N
```

sets a loop to repeat N times. The last statement of a 'DO loop' numbered 2 in the above example, must be an executable statement other than a transfer of control. An example better illustrates the use of this powerful type of statement.

Consider a programme to calculate the average of 1000 floating point numbers punched one to a card.

```
C             J. POLLARD          DO LOOP    JOB     JAN 71
            1 FORMAT(4E15.5)
            2 SUM=0.
              DO 3 I=1,1000
              READ(1,1)X
            3 SUM=SUM+X
              AVE=SUM/1000.
              WRITE(3,1)AVE
              GO TO 2
              END
0.1234 }
0.5678 }      data
```

see Appendix 3 for control cards to be inserted.

A nest of 'DO loops' is permitted one inside the other, almost without limit, provided the loops do not overlap each other. Returning to the previous example, let us suppose we wish to calculate the average of the X-s, 1-100, 1-200, ..., 1-1000. The programme may be modified to give ...

```
C             J. POLLARD          DO LOOP    JOB     JAN 71
            1 FORMAT(4E15.5)
            2 SUM=0.
              DO 3 I=100,1000,100
              DO 4 J=1,100
              READ(1,1)X
            4 SUM=SUM+X
```

```
      W=I
      AVE=SUM/W
    3 WRITE(3,1)AVE
      GO TO 2
      END
data
```

don't forget those control cards (Appendix 3).

Since the last statement of a 'DO loop' cannot be a transfer of control, a dummy statement, not performing any machine operation, is permitted in FORTRAN. This statement is normally only used when a 'DO loop' is not required to be carried out for certain values of a variable. The statement is written, for example,

```
   11 CONTINUE
```

As an example consider the previous problem where the average is to treat negative numbers as being zero. This could be coded

```
C       J. POLLARD          CONTINUE      EXAMPLE     JAN 71
      1 FORMAT(4E15.5)
      2 SUM=0.
        DO 3 I = 1,1000
        READ(1,1)X
        IF(X.LE.0.) GO TO 3
        SUM=SUM+X
      3 CONTINUE
C       ETC.
```

The CONTINUE serves simply as a terminal instruction of a 'DO loop' in this case, although it may be used anywhere in a programme in general.

3.10 Arrays of Variables

Many mathematical operations may be conveniently expressed in terms of matrices. To facilitate coding of these operations 1, 2 and higher dimensional arrays are permitted in the FORTRAN system. Taking the simplest array, the 1

dimensional array or vector, the Ith element of the vector V is represented as V(I), for example

```
V(I) = 0.
```

would set the Ith element zero (not the entire array). In Section 3.3 we saw that the compiler assigned a word of core storage to each variable. How many words does it assign to an array? To make this assignment precise, the compiler requires a non executable statement (i.e., one which is not carried out when the programme is being executed) which specifies the maximum number of elements to be associated with any array. For example if the array V is required to have no more than 15 elements this information would be supplied to the compiler using the statement

```
DIMENSION V(15)
```

which must precede the first use of the vector array V.

Subscripts associated with arrays may consist of fixed point expressions. For example, we may have the subscript 2 * IND + 1, or 5 * J - 3. The simplest example is a constant, 6, say. These features are best illustrated with an example of portion of a programme to set the odd elements of the array V to zero.

```
      DIMENSION V(15)
      DO 1 I = 1,8
    1 V(2*I-1) = 0.
```

or better still this could be coded

```
      DO 1 I = 1,15,2
    1 V(I) = 0.
```

as this does not require the fixed point multiplication of the previous method.

The more involved arrays must also be preceded by a DIMENSION before they are used. For example to set all elements of a 2 dimensional array A to the values of the corresponding elements of another array B previously defined.

```
      DIMENSION A(3,100),B(3,100)
C     ETC.
      DO 1 I = 1,3
      DO 1 J = 1,100
    1 A (I,J) = B(I,J)
```

Similarly a 3 dimensional array C would require storage to be assigned by the compiler using, say

```
      DIMENSION C(2,3,4)
```

and a typical element would be set zero thus

```
      C(I,J,K) = 0.
C     THE FORTRAN LANGUAGE HAS MOSTLY BEEN MET NOW AND WE WONT'T BE BOTHERED
C     WITH THE REST HERE, BUT PRACTICE IS A MUST
C     SO ...
C     ********************************
```

4. PRACTICE EXAMPLES

Answers and typical coding for the practice examples are given in Section 5. But do not be too hasty to GO TO 5. Try to do the examples yourself as you will need the practice before you tackle the main job ((A) 'will the reactor fuel melt if b=0.5?').

(1)/ (a) In the list below, which items are variables or constants?
(b) What is the mode (fixed or floating point) of each variable or constant in the list?
(c) Are any invalid?

List (1) 1., (2) ABC, (3) I4, (4) 14, (5) -0.0001E-10, (6) INKSTAIN, (7) FIVE, (8) 6IX, (9) e, (10) 0, (11) BOS, (12) A*B

(2)/ (a) Write each of the following algebraic formulae as a FORTRAN statement to calculate y. Use any convenient floating point names for the variables, which will be assumed to have been assigned values by previous steps of the programme.

(1) $y = \frac{1}{2}(b+c)$

(2) $y = (a+b)^2/3$

(3) $y = \left(\frac{1}{a} + \frac{1}{b} + \frac{1}{c}\right)^{-1}$

(4) $y = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!}$ $\quad (n! = 1 \times 2 \times \ldots \times (n-1) \times n)$

(5) $y-x = a - \pi y$ $\quad (\pi = 3.141592)$

(6) $\sqrt{y} = u$

(7) $x = \frac{1}{y} + \frac{1}{b} + \frac{1}{c}$

(b) What values would be stored in the variable on the left of the following arithmetic statements given that A=3. ?

(8) I = A

(9) I = A/2.

(10) U = A/2.

(3)/ Write the necessary statements of portion of a programme to calculate the variables given by the following expressions. Use any convenient names for the variables. You may assume that variables on the right have been assigned values by previous steps of the programme and that the values do not require special consideration in calculating the expressions - for example $a \neq 0$ in (1).

(1) $x = \frac{1}{2a}\left(-b + \sqrt{b^2 - 4ac}\right)$

(2) $s = \sqrt{x^2 + y^2 + z^2}$

(3) $u = \tanh x = \dfrac{\frac{1}{2}\left(e^x - e^{-x}\right)}{\frac{1}{2}\left(e^x + e^{-x}\right)}$

(4) $v = \tan x$

(5) $h = 1 - \frac{x^2}{2!} + \frac{x^4}{4!}$

(6) $y = 1 - e^{-x} - \frac{1}{5} e^{-2x} - \frac{1}{25} e^{-3x}$

(7) $c = \ln \left| \frac{1}{1 + a^3} \right|$

(8) $g = \left(\frac{\pi}{xy}\right)^{1/2} \sin\left(\frac{xy}{\pi}\right)$

(9) $y = \left(e^{ax} + e^{-\sqrt{ax}}\right)/3$

(10) $z = \frac{-\tan^{-1}(x/a)}{1 + u/a}$

(4)/ Write a computer programme, including all necessary control cards, to calculate the (real) roots of the equation

$$x^2 + bx + c = 0$$

given a deck of cards punched with b and c in the standard format. The results should be printed b, c, x_1 and x_2. (The roots are not real if $b^2 - 4c < 0$, in which case we should only print b and c.)

(5)/ Job (A) 'will the reactor fuel melt if b=0.5?'

5. ANSWERS AND TYPICAL CODING

(1)/

List (1) floating point constant, (2) floating point variable, (3) fixed point variable, (4) fixed point constant, (5) floating point constant, (6) invalid variable name as more than 6 characters, (7) floating point variable, (8) invalid variable name as first character is not alphabetic, (9) invalid variable name as e is a lower case letter, (10) fixed point constant zero if you thought this was number 0 or floating point variable if you thought this was letter O - see Appendix 2 for designation of number 0 and letter O to avoid punching errors, (11) floating point variable even if you thought the middle character was number 0 and (12) invalid since an expression is not a variable.

(2)/

```
(1)  Y  =  0.5*(B+C)
(2)  Y  =  0.3333333*(A+B)*(A+B)
(3)  Y  =  1./(1./A+1./B+1./C)
(4)  Y  =  1.+X*(1.+X*(0.5+0.1666667*X) )
(5)  Y  =  (X+A)/4.141592
(6)  Y  =  U*U
(7)  Y  =  1./(X-1./B-1./C)
(8)  I  =  3
(9)  I  =  1
```

(10) U = 1.5

(3)/

```
(1)  X  =  (-B+SQRT(B*B-4.*A*C) )/(2.*A)
(2)  S  =  SQRT(X*X+Y*Y+Z*Z)
(3)  W1 = EXP(X)
     W2 = 1./W1
     U  =  (W1-W2)/(W1+W2)
(4)  V  =  TAN(X)
(5)  W1 =  X*X
     H  =  1.-W1*(0.5-0.04166667*W1)
(6)  W1 =  EXP(-X)
     Y  =  1.-W1*(1.+W1*(0.2+0.04*W1) )
(7)  C  =  -ALOG(ABS(1.+A*A*A) )
(8)  W1 =  X*Y/3.141592
     G  =  SQRT(1./W1)*SIN(W1)
(9)  W1 =  A*X
     Y  =  (EXP(W1)+EXP(-SQRT(W1) ) )*0.3333333
(10) Z  = -ATAN(X/A)/(1.+U/A)
```

(4)/

Control cards as per Appendix 3

```
C     J.POLLARD   QUAD EQN EXAMPLE   JAN 71
      1 FORMAT(4E15.5)
      2 READ(1,1)B,C
        D=B*B-4.*C
        IF(D.LT.0.)GO TO 3
        D=SQRT(D)
        X1=0.5*(-B+D)
        X2=0.5*(-B-D)
        WRITE(3,1)B,C,X1,X2
        GO TO 2
      3 WRITE(3,1)B,C
        GO TO 2
        END
```

Control cards as per Appendix 3

Data cards, e.g.

	0.5	2.
cols.	1 to 15	16 to 30

(5)/ Output from a programme written to solve the power surge problem gives the results recorded below.

b	h(b)
0.3	1.1135
0.4	1.1527
0.5	1.1926
1.0	1.4037

We note that h(0.5) is close to the limit 1.2 - although slightly less. Our mathematical equation thus suggests that the fuel will not melt. For such a borderline case, however, we would need to know how well the mathematical equation describes the physical situation before a definite answer could be given. Fortunately in an actual reactor system the disturbances can never be as drastic as studied here.

6. FURTHER READING

When regularly involved in programming a computer we should become familiar with the manufacturer's FORTRAN manual, as differences do exist from one machine to another. For casual use we may have recourse to one of the many books available on FORTRAN. It is my belief, however, that the way to learn FORTRAN is to tackle real problems (even though in miniature) and not all books follow this line. One book that is useful, even though the FORTRAN is somewhat out of date, is

McCracken, D.D. and Dorn, W.S., 1964. (Student Edition). Numerical Methods and FORTRAN Programming. John Wiley and Sons, New York.

For those interested in numerical methods I could recommend

Pollard, J.P., 1967. Numerical Computing. Science Press, Sydney.

7. ACKNOWLEDGEMENTS

I wish to thank Br. Berchmans (De la Salle College, Cronulla), Mr. B. Clancy (AAEC, Lucas Heights), Dr. A. Dalton (AAEC, Lucas Heights) and Mr. J. Gilkes (RAN College, Jervis Bay) for their help in preparing these notes.

APPENDIX 1

SOME ANALYSIS FOR THOSE WHO CAN STAND IT

Given the differential equation for the power $p(t)$ of a reactor following a temporary disturbance b,

$$\frac{dp}{dt} = -\left(p-1-be^{-t}\right)p , \qquad p(o) = 1 ,$$

we ask 'what can we do analytically?' Well, mathematical hindsight suggests we change the dependent variable to q given by

$$p = e^q$$

then

$$\frac{dp}{dt} = e^q \frac{dq}{dt} = p\frac{dq}{dt}$$

and the differential equation becomes

$$\frac{dq}{dt} = -\left(e^q-1-be^{-t}\right) , \qquad q(o) = 0 \quad (\text{since } e^o = 1) .$$

This equation looks worse than the equation we were given, but let us suppose that we know that the power surge is not large so that we may take the first few terms in an expansion of e^q

$$e^q \simeq 1 + q .$$

We then obtain

$$\frac{dq}{dt} = -\left(q - be^{-t}\right) , \qquad q(o) = 0 .$$

A solution of the above differential equation is then an approximate solution of the given differential equation (to an extent that we will not pursue) provided $q(t)$ remains small (so that $e^q \simeq 1 + q$).

Continuing with the analysis, we multiply the equation above by e^t and we obtain

$$\frac{dq}{dt} e^t + qe^t = b.$$

Now we (may) know that

$$\frac{d}{dt}(qe^{t}) = \frac{dq}{dt}e^{t} + qe^{t} ,$$

hence the differential equation reduces to

$$\frac{d}{dt}(qe^{t}) = b ,$$

which integrates to give

$$qe^{t} - 0 = bt \qquad (\text{since } q(o) = 0)$$

We thus obtain the approximate solution

$$q(t) = bte^{-t} .$$

In order to find the maximum power surge h(b) we seek the maximum value for p(t) and hence q(t). Now

$$\frac{dq}{dt} = b(e^{-t} - te^{-t})$$

and $$\frac{dq}{dt} = 0 \quad \text{when } t = 1$$

hence $$\max q(t) = be^{-1} = 0.368\ b ,$$

which gives us the result

$$h(b) = e^{0.368\ b} \quad (\text{since } p = e^{q}).$$

Again we must emphasize that the above is only approximate. Nevertheless we find that

$$h(0.3) = 1.117,$$

which is in good agreement with the 'experimental' result cited earlier

$$\left(h(0.3) = 1.12 \pm 0.01\right) ,$$

and $h(0.5) = 1.202.$

This is so close to our critical value $h = 1.2$ that before we accept the result we would have to pursue detailed analysis of the effect of the approximation $e^q \simeq 1+q$ on our final answer. Alternatively we may pursue a numerical approach since we know that simple analysis cannot give us a sufficiently accurate result for the problem in hand.

APPENDIX 2

PUNCHING CONVENTION(S)

General When submitting coding or data to be punched by a card punch operator we must clearly distinguish between several characters. We will use the following convention ...

the letter	O	is written	O
the number	0	is written	0
the letter	I	is written	I
the number	1	is written	|
the letter	Z	is written	Z
the number	2	is written	2

Unfortunately in scientific work everybody has their own convention. The 360 recommended practice of crossing the number 0 thus Ø is never used by people with lots of data to prepare. Since I come into the category mentioned, I am a non-conformist.

FORTRAN (BUFF40) IBM 360/40

Special fast compilers such as the BUFF40 compiler available on the IBM 360/40 have a few restrictions regarding embedded blanks in statements. For example

```
DO1I = 1,10    is not permitted
```

rather we should use

```
DO | I = 1,10
```

```
Also    IF(A.EQ.B)GO TO 15    is not permitted
but     IF(A.EQ.B) GO TO 15   is acceptable
```

APPENDIX 3

CONTROL CARDS

Special control cards are required in a job to identify the job, the part of the deck which is coding (FORTRAN) and the part of the deck which is data. These cards supply information to a control programme (operating system, O/S) which enables many jobs to be run one after the other (or several at once with some systems) without intervention of the computer operator. Each machine has its own system (or systems). We will look at cards required by two systems in this appendix. The control cards are punched starting in column 1.

(1) BUFF40 system available on the IBM 360/40, IBM Test and Education Centre, Bradfield Highway, Sydney

A complete job is set up as indicated below.

```
(a) $COMPILE
(b) C         USER        JOB
      1 FORMAT(4E15.5)
    etc.
(c) $EXECUTE
    0.1          0.2       0.3
    etc.
```

(a) The $COMPILE card calls in the FORTRAN compiler - FORTRAN follows.

(b) The first FORTRAN statement must identify us (USER) and the task we are tackling (JOB) so that output may be returned to us.

(c) The $EXECUTE card causes the job to be run. Data would normally follow.

(2) O/S available on the IBM 360/50, AAEC Research Establishment, Lucas Heights, Sydney

A complete job is set up as indicated below. Here the system is more complicated for our simple requirement than (1), however when we want to save programmes and data on disk etc., the versatility of O/S becomes apparent.

```
(a)  { //
     { //JOB          USER
(b)  { //A       EXEC       FORTGCLG
     { //FORT.SYSIN    DD     *
       C         A PROG. TO CALC ...
          1 FORMAT(4E15.5)
       etc.
```

```
(c)     /*
(d)     //GO.SYSIN DD *
        0.1        0.2        0.3
        etc.
(e)   { /*
      { //
```

(a) The first card (//) is the first of a Job and the second card, supplied by the computer section, identifies the JOB and the USER.

(b) The next cards identify the compiler required (FORTGCLG, although FORTHCLG is available as well) and where the compiler input is to be found (//FORT.SYSIN DD * tells the O/S that the FORTRAN follows).

(c) The card punched /* indicates the end of the input to the compiler.

(d) Any data to follow the FORTRAN is then indicated

(e) and the data must end with the terminating card (/*). The last card (//) is the last of a Job.

FIGURE 1. A(N IM)POSSIBLE REACTOR RESPONSE TO A POWER SURGE

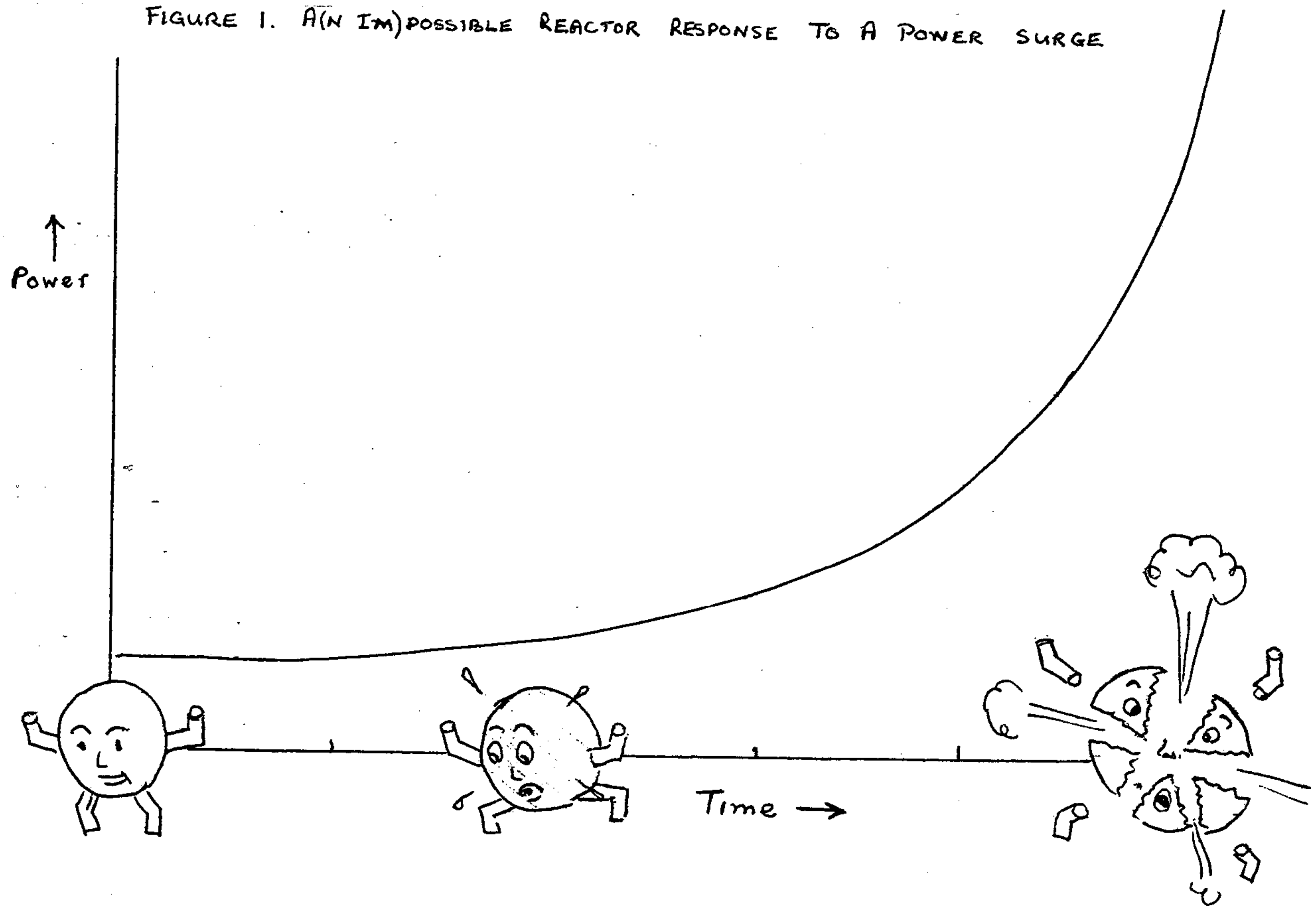


FIGURE 2 A POSSIBLE REACTOR RESPONSE TO A POWER SURGE

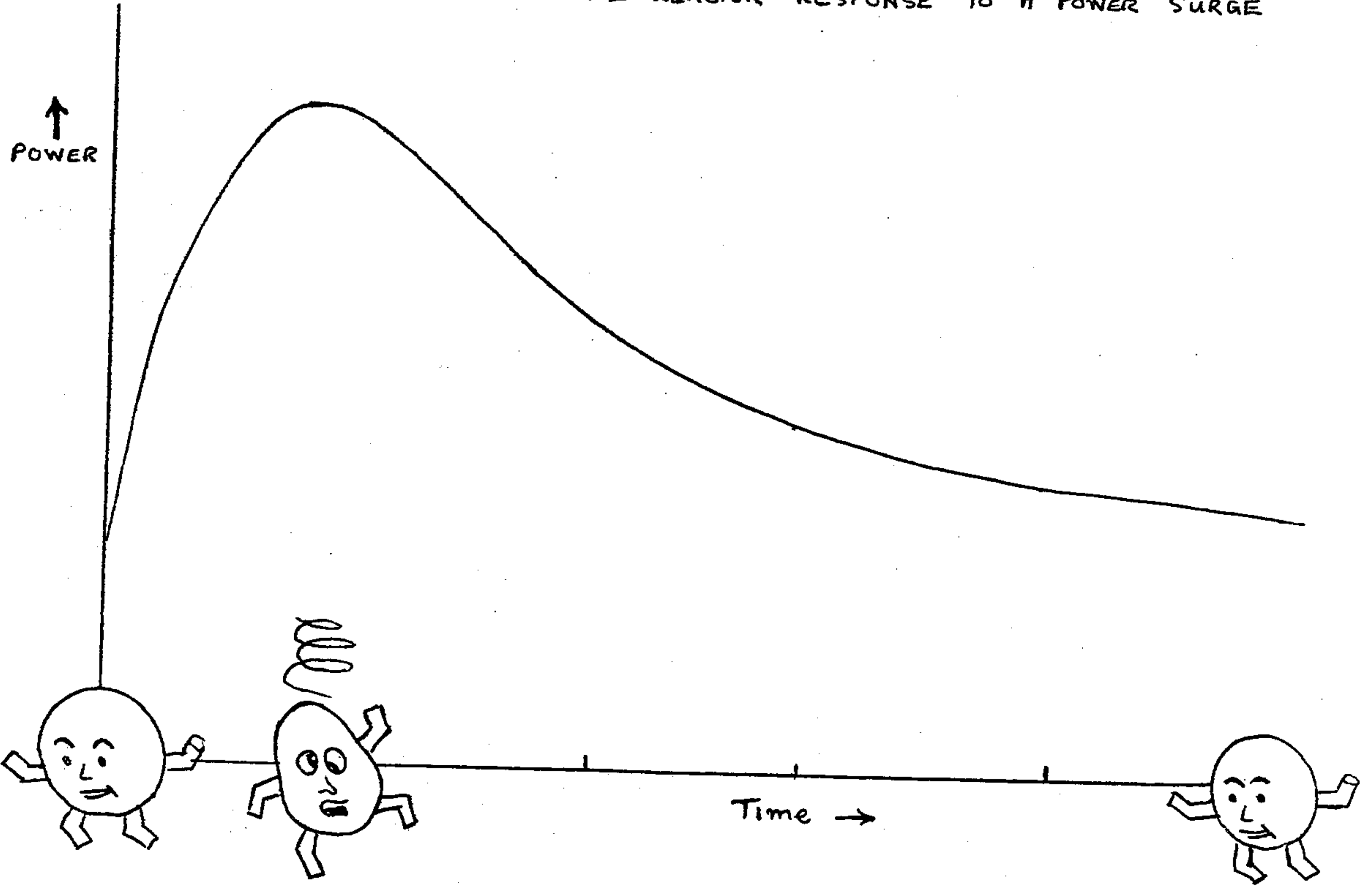


FIGURE 3 A POSSIBLE REACTOR RESPONSE TO A (NEGATIVE) POWER SURGE

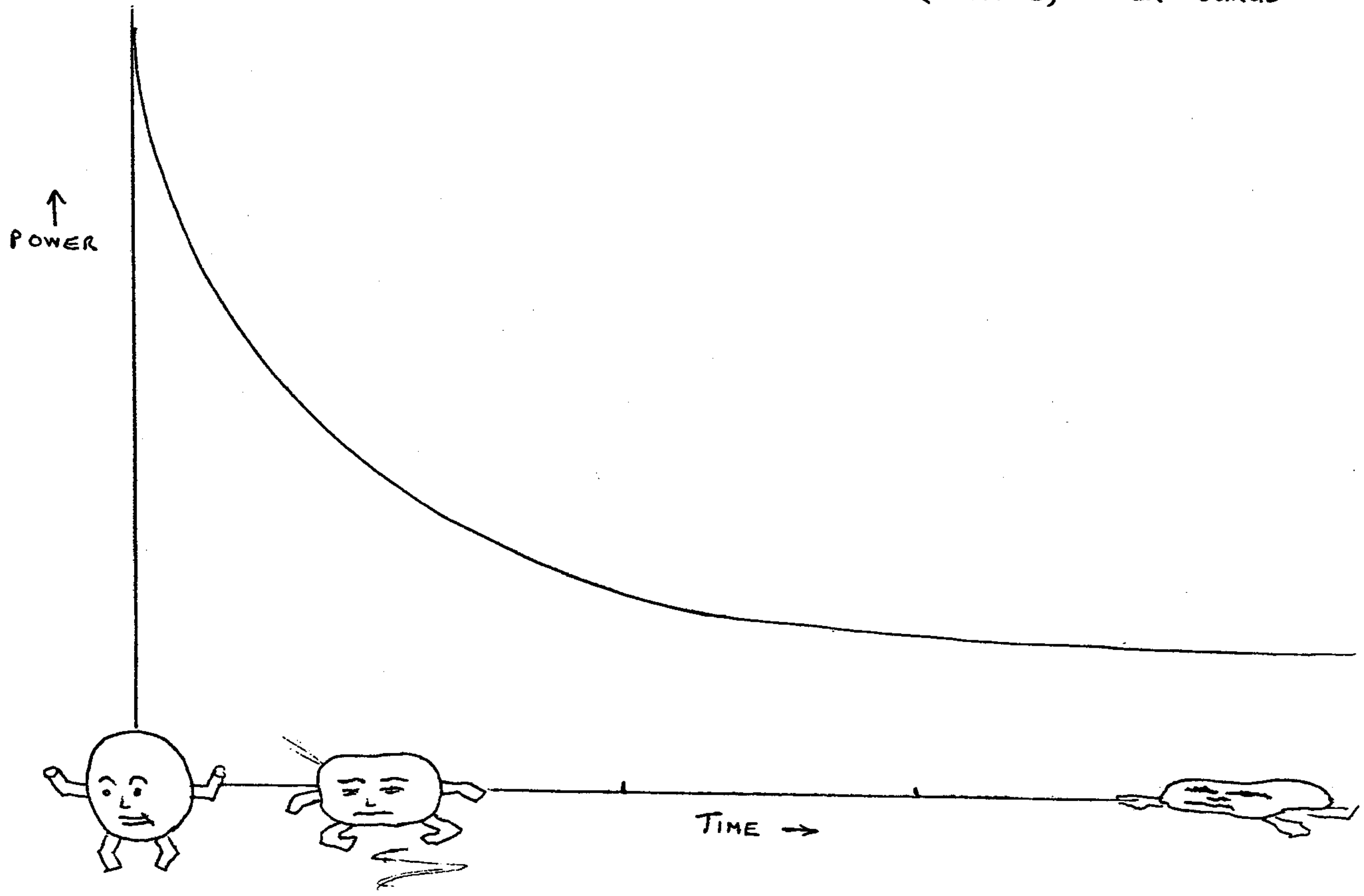


Figure 4. A graphical illustration of the slide rule calculations