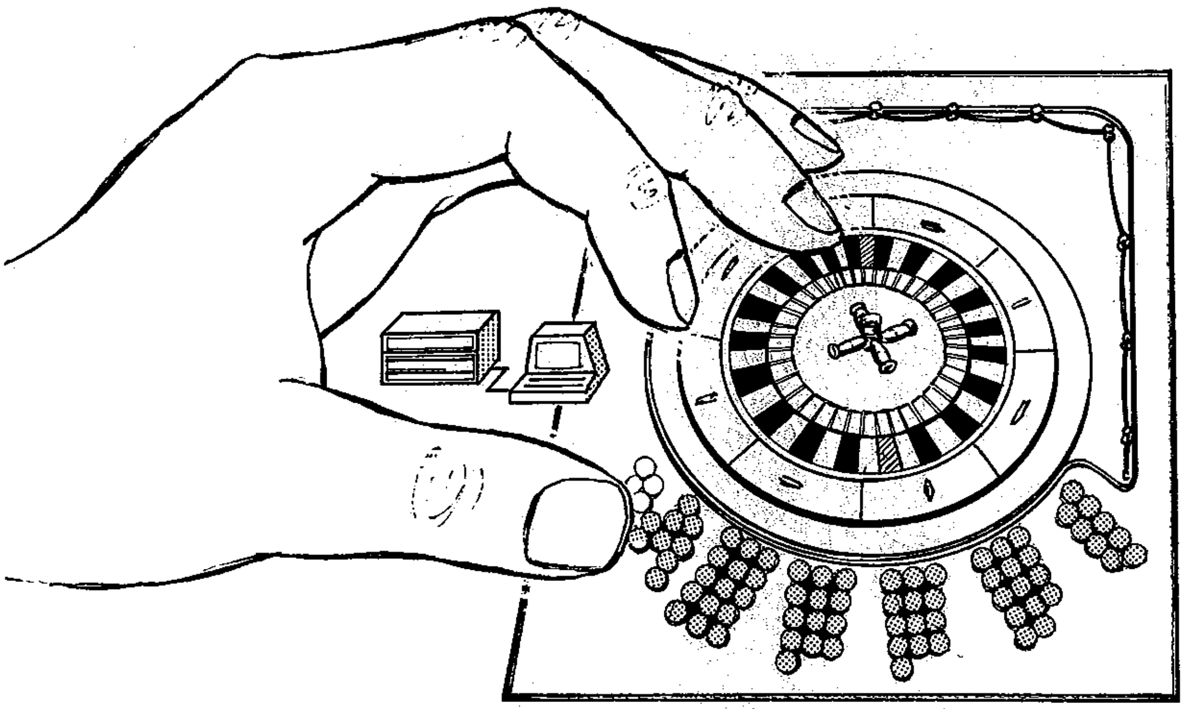
AAEC/S20

AUSTRALIAN ATOMIC ENERGY COMMISSION
RESEARCH ESTABLISHMENT
LUCAS HEIGHTS

SUMMER SCHOOL 1978

SIMULATION IN SCIENCE WITH MATHEMATICS AND COMPUTERS

Edited by
J.P. Pollard



December 1978
ISBN 0 642 59656 5

AUSTRALIAN ATOMIC ENERGY COMMISSION
RESEARCH ESTABLISHMENT
LUCAS HEIGHTS

SUMMER SCHOOL 1978

SIMULATION IN SCIENCE WITH MATHEMATICS AND COMPUTERS

Edited by

J.P. POLLARD

ABSTRACT

These notes are for a Summer School which will introduce mathematically minded year-12 High School students to simulation applied to science. The course considers the application of

1. discrete Monte Carlo techniques, and
2. continuous differential techniques

to a reactor problem - the transmission of neutrons through mild steel.

A considerable portion of the course is devoted to electronic computing using large scientific digital computers, minicomputers, microcomputers and hybrid, analogue-digital computers. These techniques are applied to the neutron transmission problem and other simulation processes.

National Library of Australia card number and ISBN 0 642 59656 5

The following descriptors have been selected from the INIS Thesaurus to describe the subject content of this report for information retrieval purposes. For further details please refer to IAEA-INIS-12 (INIS: Manual for Indexing) and IAEA-INIS-13 (INIS: Thesaurus) published in Vienna by the International Atomic Energy Agency.

[1] SIMULATION; STOCHASTIC PROCESSES

[2] DIFFERENTIAL EQUATIONS; MATHEMATICS; COMPUTER CALCULATIONS; FUNCTIONS

[3] SIMULATION; MONTE CARLO METHOD; RANDOMNESS; NEUTRONS; SHIELDING; NEUTRON FLUX

[4] NEUTRON BEAMS; NEUTRONS; ABSORPTION; SHIELDING; NEUTRON REACTIONS; SCATTERING; NUCLEI; IRON

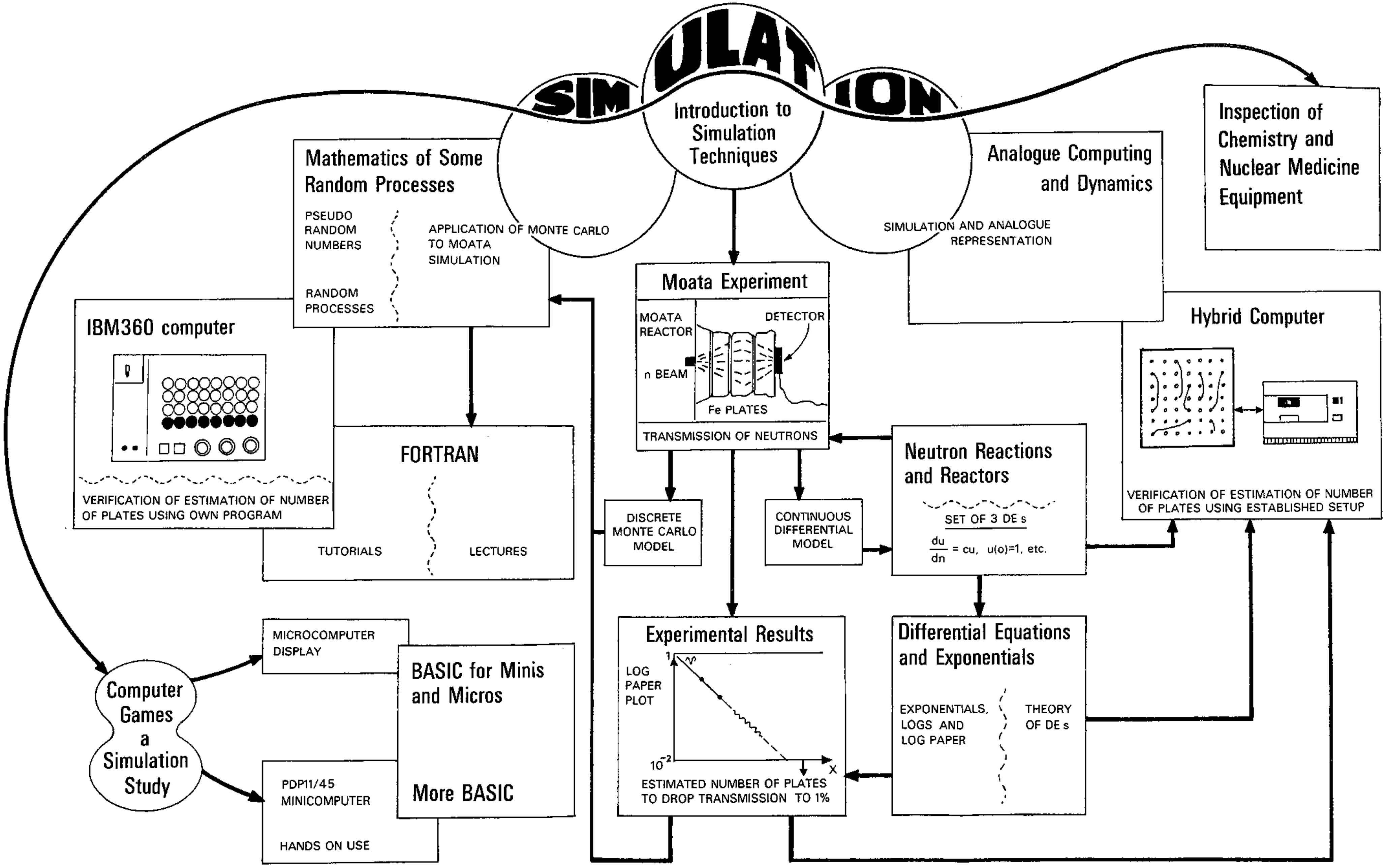
[5] AMPLIFIERS; ANALOG COMPUTERS; DIFFERENTIAL EQUATIONS; HYBRID COMPUTERS; MATHEMATICAL MODELS; POPULATION DYNAMICS; SIMULATION

[6] COMPUTER CALCULATIONS; FORTRAN; MATHEMATICS; PROGRAMMING LANGUAGES

[7] PROGRAMMING LANGUAGES; PDP COMPUTERS; MICROPROCESSORS

[8] COMPUTER CALCULATIONS; SIMULATION

CONTENTS


SIMULATION
Introduction to Simulation Techniques
Mathematics of Some Random Processes
PSEUDO RANDOM NUMBERS
RANDOM PROCESSES
APPLICATION OF MONTE CARLO TO MOATA SIMULATION
Analogue Computing and Dynamics
SIMULATION AND ANALOGUE REPRESENTATION
Inspection of Chemistry and Nuclear Medicine Equipment
IBM360 computer
VERIFICATION OF ESTIMATION OF NUMBER OF PLATES USING OWN PROGRAM
FORTRAN
TUTORIALS
LECTURES
Moata Experiment
MOATA REACTOR
n BEAM
DETECTOR
Fe PLATES
TRANSMISSION OF NEUTRONS
DISCRETE MONTE CARLO MODEL
CONTINUOUS DIFFERENTIAL MODEL
Neutron Reactions and Reactors
SET OF 3 DE s
$\frac{du}{dn} = cu, \quad u(o)=1$, etc.
Hybrid Computer
VERIFICATION OF ESTIMATION OF NUMBER OF PLATES USING ESTABLISHED SETUP
Experimental Results
LOG PAPER PLOT
1
10^{-2}
X
ESTIMATED NUMBER OF PLATES TO DROP TRANSMISSION TO 1%
Differential Equations and Exponentials
EXPONENTIALS, LOGS AND LOG PAPER
THEORY OF DE s
Computer Games a Simulation Study
MICROCOMPUTER DISPLAY
BASIC for Minis and Micros
More BASIC
PDP11/45 MINICOMPUTER
HANDS ON USE

CHAPTER 1

INTRODUCTION TO SIMULATION TECHNIQUES

Lecture by

G. DOHERTY
(University of Wollongong)

CONTENTS

1.1 GENERAL REMARKS

Simulation is an attempt to represent the behaviour of systems by some sort of model. In the process of constructing the model, we try to establish the relationships between the different interacting components of the system. If possible, we try to express these relationships in the form of mathematical equations containing quantitative as well as qualitative information. When the parameters in the equations have been determined, the model can be used to predict system effects. The simulation is usually conducted because it is either cheaper or more convenient, or perhaps both, to conduct experiments on the model than on the system itself. The use to which the simulation model will be put is one major determinant of the complexity of the model; the other is the complexity of the system itself. The following is a checklist of questions which should be asked before and during the construction of the model.

What is the purpose in building the model and conducting the simulation?

Shielding calculations like the one you will see enable reactor designers to design a radiation shield with the required characteristics without having to build and test a prototype shield for each new reactor design. Our purpose in looking at this problem is to give you a wide exposure to different computers, computing techniques and some interesting mathematics.

How much of the system must be included explicitly in a model?

To answer this question we must know the purpose of the model. In the case of shielding around a reactor, it is much more difficult to model the reactor itself *and* the shield, than to model the shield with the reactor replaced by a source of neutrons hitting the inside of the shield. Provided that we don't intend to use the model to predict the electricity output of the station, the simple model of shield plus source should suffice.

What system characteristics need we consider?

Simulation is usually concerned with a system in an initial state, some inputs, the system in a final state, and some outputs. Sometimes the states of the system are our primary concern (*e.g.* the state of the economy in an economic model), whereas in other situations the main interest may be system outputs produced in response to various inputs. The shielding problem falls into the latter category - we are not interested in the final state of the shield itself provided that its properties do not deteriorate with use.

Is our system deterministic or stochastic?

Do we think we can write down a set of equations such that if the inputs and initial state of the system are specified, the outputs and final state of the system are uniquely determined? To answer the question, we must study the system carefully, conducting experiments where possible to measure outputs over a range of inputs, and testing whether experiments are reproducible. Here the physical scientist enjoys a considerable advantage over the economist whose observations are limited to one set of inputs and one experiment. If we cannot determine output uniquely from specified input, two explanations are possible - either the system is deterministic but we cannot find the explicit relationships, or the output of the system is determined by chance so that a unique output is, in principle, unavailable. In the latter case, a simulation will usually seek to establish the relative probabilities of various outcomes by conducting a series of trials. Philosophically, it may be difficult to decide whether or not a system is deterministic.

What sort of model should we construct?

Largely, our choice of model is constrained by whether we believe the system is deterministic or stochastic. The shielding problem you will examine offers a wider choice of model than do many systems. Each neutron which impinges on the shield may eventually pass through it, or not, depending on the outcome of a series of events with particular probabilities of occurrence. In this sense the system is stochastic and may be modelled by conducting a series of trials with individual neutrons. However, the number of neutrons in an actual experiment is so large (say 10^{12}) that the outcome of the experiment is essentially deterministic because the statistical fluctuations are negligible. The behaviour of the system can be modelled successfully by a differential equation involving the average population of neutrons. You can judge for yourself which type of model is better - both have strengths and weaknesses.

What validation of the model is required?

This important phase of the process must be completed before predictions based on the model can be taken seriously. Some possible sources of error are as follows:

(i) Incorrect perceptions of system relationships. Some economic models are particularly vulnerable here because the experiment cannot be repeated with different inputs.

(ii) Incorrect measurement of system parameters. The importance of a particular system component to the simulation may not be apparent when its parameters are estimated.

(iii) Incorrect computer coding of the model. The difficulty of producing a large *correct* computer code is not widely appreciated.

1.2 A SIMPLE STOCHASTIC MODEL

Having discussed the general ideas of simulation it might be helpful to examine these ideas in the context of a fairly simple simulation. Since many simulations involve probabilities, and since I have a nodding acquaintance with this particular system, I would like to discuss, briefly, the poker machine.

1.2.1 System Description

A vintage three-wheel machine has 30 (say) faces on each wheel, each face showing one of 9, 10, J, Q, K or A. Each insertion of 20 cents (or coinage of lesser value), followed by a pull on the handle, presents one face from each wheel in the paying position. Pays range from 0 for most combinations to a $100 jackpot for AAA. To the honest player, the machine presents as a stochastic system (*i.e.* the output of money from successive inputs of 20 cents is obviously quite variable). There is a suspicion amongst players that the initial and final states of the machine are not totally independent (in the sense that what shows next in the window relates to what was there last time), but this could probably be ascribed to wishful thinking. The principal determinants of the machine behaviour are the relative abundances of the various faces on each wheel. Armed with this information (which is available to the machine owner), we should be able to produce a satisfactory model of this simple mechanical device.

1.2.2 Purpose of the Model

The model is intended to allow a player of poker machines to play our model instead.

1.2.3 Scope of the Model

The model will be required to display the result of each play and keep a cumulative record of the player's financial status.

1.2.4 Construction of the Model

Consider the first wheel. Let f_i be the faces 9, 10, J, Q, K and A. Let n_i = the number of each face on the wheel. $\sum_{i=1}^{6} n_i = 30$ the total number on each wheel.

The probability that f_i will appear is

$$p_i = \frac{n_i}{\sum_{i=1}^{6} n_i} = \frac{n_i}{30} .$$

We need a mechanism to select the face in accordance with this basic probability law. As this problem is common to many stochastic simulations we will spend a little time on its solution, and the related problem for a continuously distributed variable.

Firstly, we define a cumulative probability distribution by the relations:

$$\begin{aligned} P_0 &= 0 \\ P_i &= P_{i-1} + p_i . \end{aligned}$$

Thus

$$\begin{aligned} P_0 &= 0 \\ P_1 &= p_1 \\ P_2 &= p_1 + p_2 \\ \\ P_6 &= p_1 + p_2 + p_3 + p_4 + p_5 + p_6 = 1 \end{aligned}$$

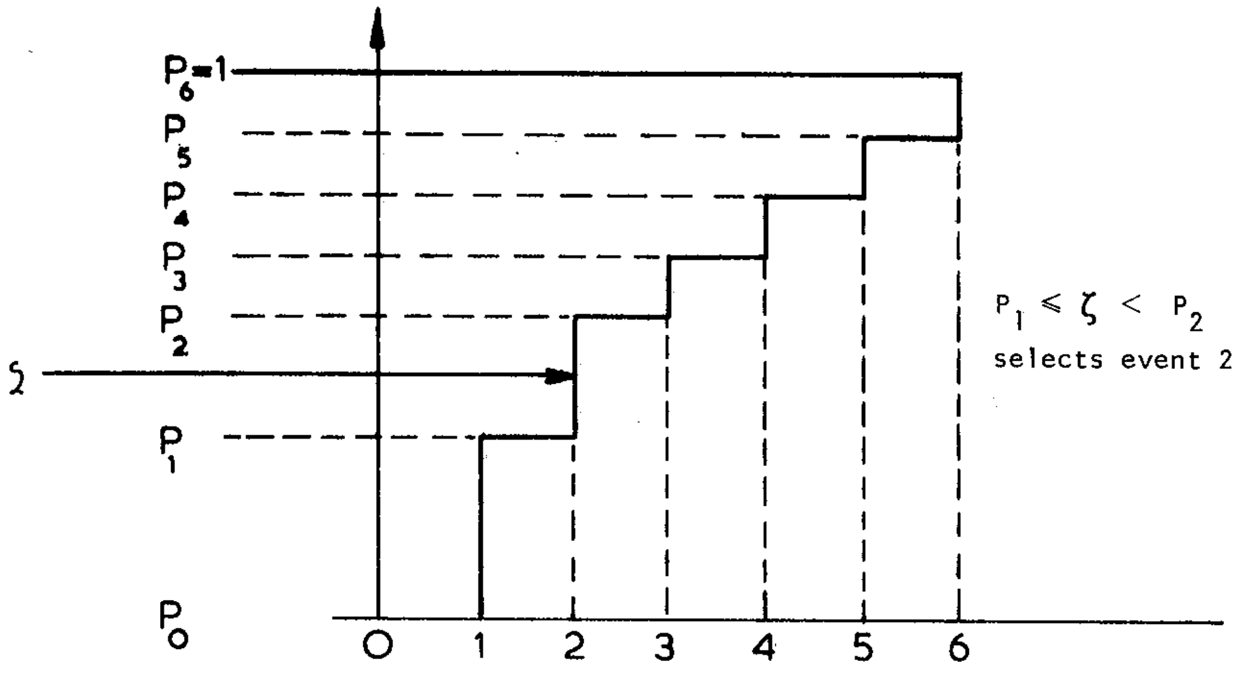
Then, at each play, we generate a random or pseudo-random number in the interval (0,1). Algorithms for performing this task are available on most computers and in most computer languages. Each number in the range (0,1) should be equally likely to be generated and various tests can be undertaken to establish the validity of the generation algorithm.

For each random number ζ we test the inequalities

$$P_{i-1} \leqslant \zeta < P_i$$

and since ζ is in (0,1) the inequality will be satisfied for one, and only one, value of i. This chooses the value f_i of the face on the wheel.

This generation of events can be pictured as follows:



Digression

The above figure applies to a discrete event, *i.e.* choosing one out of six faces. The same type of idea can be extended to choosing a continuous variable from the interval (a,b).

Let $p(x)\,dx$ = the probability that x will be chosen in the interval x to x + dx, normalised so that

$$\int_a^b p(x)dx = 1$$

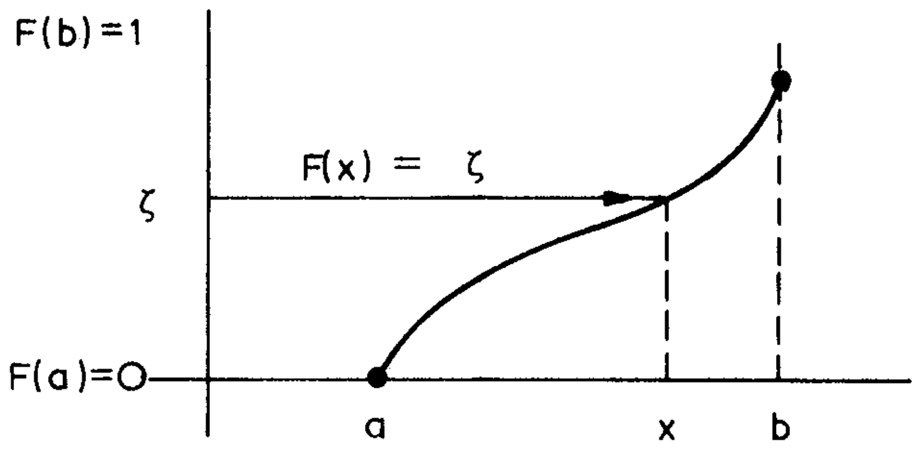
Define $\int_a^x p(x)dx = F(x)$

Choose ζ in (0,1)

then choose x from $F(x) = \zeta$.

p(x) is called a probability density function.

F(x) is a cumulative probability function.



$F(x) = \zeta$ has a unique solution provided that $p(x) > 0$, which is a sensible restriction on $p(x)$ in view of its definition.

Returning to our model construction: For each wheel, we read in the relative abundances n_i and form the cumulative probabilities P_i. Our program also needs a table of paying combinations, and the amount that each combination is to pay. Given this information, we can determine the expected payout of our machine for each unit invested. No doubt the latter information is supplied in table form by the vendor to the purchaser of such a machine. We can use it as one check on the correctness of our simulation.

Use of the model: Our simple model is ready to generate a succession of results corresponding to plays on the actual machine. With marginally more effort, and extra information on player habits, such as how much the player was prepared to lose, at what level of winnings the player would cease playing, the rate of play and total time available, we could simulate an evening's play in a few milliseconds. Further, given a lifetime playing regime, we could simulate the outcome of a lifetime's involvement in a few seconds. In one minute of computer time, on any respectable computer, we could generate sufficient possible lifetime histories to give a prospective player an educational view of his prospects. The computer bill for this exercise might not exceed the loss resulting from one minute of real play.

Validation: Stochastic models are difficult to check because the output is, by definition, unpredictable. It is often possible to check expected values of quantities against the mean values computed during the course of simulation. For example, the expected payout should correspond with the mean payout, computed over a long series of trials, to within a statistical error. Failure to obtain agreement, at the required significance level, would indicate either incorrect coding of the simulation *or* a faulty random number generating algorithm. Further checks could also be made on the expected occurrences of each face on each wheel.

1.3 SOME REAL SIMULATIONS

In September 1978, a biennial conference on simulation was held at Canberra. The titles of some of the papers are reproduced below to give an idea of the different types of simulation which were discussed. As you can see, there is quite a lot of interest in the construction of biological and economic models, as well as the more conventional areas of engineering and physical science.

SOME TITLES FROM THE 1978 SIMSIG CONFERENCE

The Development of an Ecosystem Model of South West Arm (Port Hacking, NSW)

Estimating Differential Equation Models of Water Quality in Non-tidal Rivers

Automatic Tuning of Parameter Values in a Pasture Growth Simulation Model

Hydrologic Simulation of Rainfall-Runoff in a small Urban Catchment using an ARMA Model

A Control Model of Disturbed Cell Proliferation

Simulation with the NIF Model of the Australian Economy

Economy-wide Effects of Long-run Changes in Demography, Technology and International Trade

The Use of Simulation to Analyse Exhaustible Resource Cartels

Railway Yard Design using Digital Simulation

Analysis and Simulation of Townsville Port Operations

Traffic Network Simulation - Assignment Model, TNS

The Hybrid Simulation of a Boiling Channel

Simulation of Depressurisation and Overheating of a Model Nuclear Power Reactor

Dynamic Modelling and Control of Multistage Biochemical Reactor Systems

A Study of Various Methods Available for Tuning Controllers

Digital Simulation of an Industrial Hydraulic Control System

Field Validation of a Crop/Pest Management Descriptive Model

Applications of Computer Simulation to Animal Disease Control

A Simulation Model of Temperatures in Grain Bins

Rapid Development of Predictive Pest-Moth Models

CHAPTER 2

DIFFERENTIAL EQUATIONS AND EXPONENTIALS

Lecture by

J.P. POLLARD

CONTENTS

2.1 DIFFERENTIAL EQUATIONS

In its simplest form, a differential equation (DE) problem looks like this

$$\underbrace{\boxed{\frac{dy}{dx} = f(x)}}_{\text{DE}} \quad , \quad \underbrace{\boxed{y(0) = y_0}}_{\substack{\text{initial}\\ \text{condition}}} \quad , \quad \text{say,} \qquad (2.1)$$

where y is a function of x to be determined with given value y_0, and

$f(x)$ is a given function of x.

Although y is unknown, we at least know one value as given by the initial condition and we know the slope, $\frac{dy}{dx}$, at every point. In fact what is given is sufficient for us to determine the function y. To see this, let us consider the following problem.

$$\underbrace{\boxed{\frac{dy}{dx} = 1+x}}_{\text{DE}} \quad , \quad \underbrace{\boxed{y(0) = 1}}_{\substack{\text{initial}\\ \text{condition}}} \qquad (2.2)$$

We set up a table of given information

x	$\frac{dy}{dx}$	y
0	1	1
1	2	
2	3	
3	4	
4	5	
5	6	

and then adopt an approximate graphical approach (figure 2.1) to fill in the missing values.

Here are the results from figure 2.1:

x	$\frac{dy}{dx}$	y
0	1	1
1	2	2.5
2	3	5
3	4	8.5
4	5	13
5	6	18.5

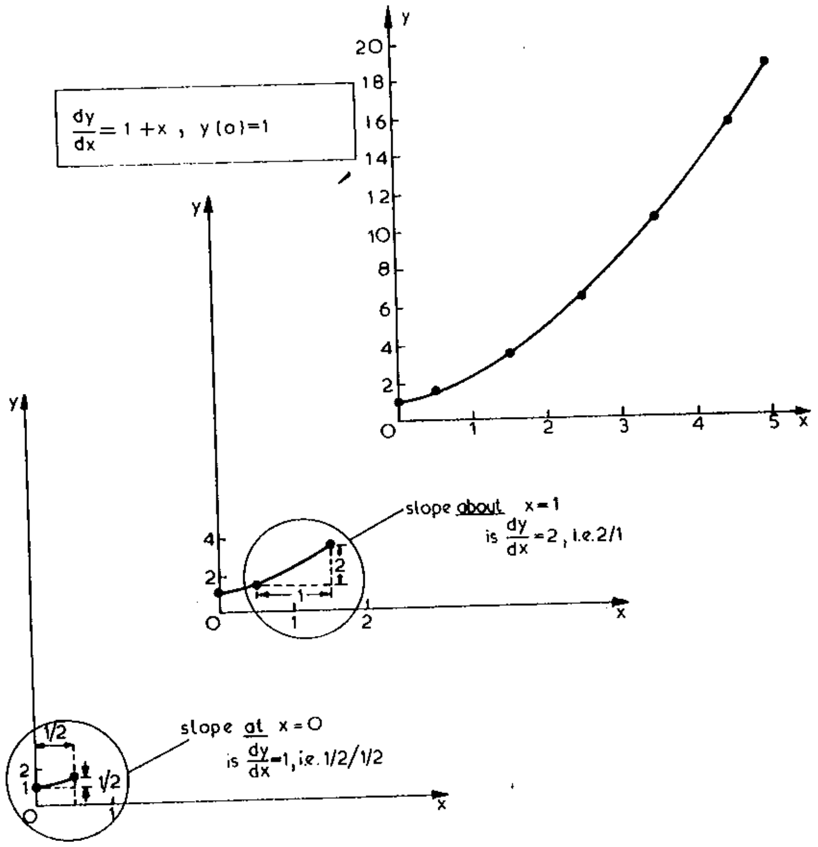


Figure 2.1 An approximate graphical approach

Hey wait a bit, you say, I can get these results analytically - watch!

Now $\qquad \frac{dy}{dx} = 1+x$,

$\times\, dx \rightarrow \qquad dy = (1+x)dx$,

$\int \rightarrow \qquad \int_1^y dy = \int_0^x (1+x)dx$,

$\uparrow \qquad \uparrow$
y is 1 when x is 0 - the initial condition

$\therefore \qquad [y]_1^y = [x + \frac{1}{2}x^2]_0^x$ - did you follow that ?

remember $\qquad \int x^n dx = \frac{x^{n+1}}{n+1}$ $\qquad$ (2.3)

and finally

$$y = 1 + x + \frac{1}{2}x^2 \quad , \qquad (2.4)$$

which gives exactly the same numerical results as the table. (Lucky!)

We have thus solved the DE given by equation 2.2 in two ways - numerically (or rather, graphically) and analytically. Encouraged by this, we tackle the general form given by equation 2.1. The same analytic attack gives

$$\int_{y_0}^{y} dy = \int_0^x f(x)dx \quad ,$$

i.e.
$$y = y_0 + \int_0^x f(x)dx \quad . \tag{2.5}$$

As a further example, say $y_0 = 1$ and $f(x) = 1 + x + \frac{1}{2}x^2$,

then
$$y = 1 + \int_0^x (1 + x + \frac{1}{2}x^2)dx$$

$$= 1 + [x + \frac{1}{2}x^2 + \frac{1}{6}x^3]_0^x$$

which we can write as

$$y = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} \quad , \tag{2.6}$$

where $2! = 1 \times 2 = 2; \quad 3! = 1 \times 2 \times 3 = 6; \ldots;$

$$n! = 1 \times 2 \times 3 \times \ldots \times (n-1) \times n \tag{2.7}$$

is the factorial function.

Before we get carried away with the ease of analytically solving DEs, let us introduce a variation of the problem that looks deceptively simple, namely

$$\underset{\text{DE}}{\boxed{\frac{dy}{dx} = y}} \quad , \quad \underset{\text{initial condition}}{\boxed{y(0) = 1}} \tag{2.8}$$

Easy, you say;

$$y = 1 + \frac{1}{2}y^2 \quad .$$

WRONG! Let us tackle this problem a bit more slowly. We do not yet know y on the right hand side of the DE and so, as a starter, we use

$$y = 1 \qquad \text{for all } x \tag{2.9}$$

which is at least consistent with the initial condition. Then

$$\frac{dy}{dx} = 1 \quad , \quad y(0) = 1 \tag{2.10}$$

has the solution

$$y = 1 + x \tag{2.11}$$

by direct integration. Let us use equation 2.11 on the right so that we get the DE

$$\frac{dy}{dx} = 1 + x \quad , \quad y(0) = 1 \quad . \tag{2.12}$$

We have already solved this as

$$y = 1 + x + \frac{1}{2} x^2 \quad . \qquad (2.13)$$

We use this on the right to get the DE

$$\frac{dy}{dx} = 1 + x + \frac{1}{2} x^2 \quad , \quad y(0) = 1 \qquad (2.14)$$

and again we have already solved this as

$$y = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} \quad . \qquad (2.15)$$

If this process is repeated we obtain

$$y = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \ldots + \frac{x^n}{n!} + \ldots \qquad (2.16)$$

which fortunately has terms towards the end that get smaller and smaller no matter how big x happens to be since n! for large n is a whopper (see the table).

Pardon my figuring out.

```
~~01 RCL0~~
~~02 1~~
~~03 +~~
~~04 STO0~~
~~05 fPAUSE~~
~~06 X~~
~~07 GTO00~~
```

n	n!
0	1
1	1
2	2
3	6
4	24
5	120
6	720
7	5040
8	40320
9	362880
10	3628800
15	1.3076744×10^{12}
20	2.4329020×10^{18}

```
~~10 REM N!~~
~~20 M=1~~
~~30 FOR N=1 TO 20~~
~~40 M=M*N~~
~~50 PRINT N;"!=";M~~
~~60 NEXT N~~
~~70 END~~
```

Equation 2.16 then gives us a convergent series expansion which satisfies the DE (equation 2.8). In fact, the value of y for x = 1 is simply obtained by adding up the reciprocals of entries in the above table (to n = 10),

$$y(1) = 1 + 1 + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \ldots + \frac{1}{10!} \qquad (2.17)$$

$$= 2.718282 \quad ;$$

this is an important number called e, *i.e.*

$$\boxed{e = 2.718282} \qquad (2.18)$$

which is as highly regarded in the mathematical world as the number

$$\pi = 3.141593 \tag{2.19}$$

Digression

Having met π as a really special number, you may not like to see a competitor enter the field. Well don't worry! π and e are cousins through the strange but beautiful relationship

$$e^{\pi\sqrt{-1}} = -1 \text{ , (derived by Euler).} \tag{2.20}$$

2.2 EXPONENTIALS

We get an idea of the importance of e when we find that there is a special mathematical function called the exponential function, which is e raised to any power,

$$y = e^x \quad . \tag{2.21}$$

Let us see if we can differentiate this expression. Here we go ...

$$\frac{dy}{dx} = \lim_{\delta x \to 0} \left[\frac{y(x+\delta x)-y(x)}{\delta x}\right] \quad \text{— remember first principles}$$

$$= \lim_{\delta x \to 0} \left[\frac{e^{x+\delta x}-e^x}{\delta x}\right]$$

$$= \lim_{\delta x \to 0} \left[\frac{e^x e^{\delta x}-e^x}{\delta x}\right] \quad \left\{ \begin{array}{l}\text{did you follow this bit} \\ e^{x+\delta x} = e^x e^{\delta x} \text{ from a property} \\ \text{of powers?}\end{array}\right.$$

$$= e^x \lim_{\delta x \to 0} \left[\frac{e^{\delta x}-1}{\delta x}\right] \quad \text{— taking out } e^x \text{ as a common factor.}$$

Now $\dfrac{e^{1/16}-1}{1/16} = 1.032$ and $\dfrac{e^{1/64}-1}{1/64} = 1.008$ from which we infer, correctly, that

$$\lim_{\delta x \to 0} \left[\frac{e^{\delta x}-1}{\delta x}\right] = 1 \tag{2.22}$$

then

$$\frac{dy}{dx} = e^x = y \quad . \tag{2.23}$$

Isn't this our DE 2.8? Yes!

Well who would believe it; there are two solutions to our DE (2.8):

$$y = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \ldots + \frac{x^n}{n!} + \ldots \tag{2.16}$$

and

$$y = e^x \quad . \tag{2.21}$$

This is taking things 'two' far. Still the battle is yet to be 'one'. Referring to an old maths text book (sorry I can't give the reference as the cover has fallen off), we find that the solution of our DE is unique. Aha! you mean

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \ldots + \frac{x^n}{n!} + \ldots \qquad (2.24)$$

yes indeed!

Recapitulating –

the solution of the DE

$$\frac{dy}{dx} = y \;, \quad y(0) = 1 \qquad (2.8)$$

is $$y = e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \ldots + \frac{x^n}{n!} + \ldots \qquad (2.25)$$

and $\frac{de^x}{dx} = e^x$, *i.e.* e^x is its own derivative. (2.26)

Of almost equal importance is the integral results obtained from the above. We have, by direct integration,

$$\int_0^x \frac{dy}{dx}\,dx = \int_0^x y(x)dx \quad ;$$

but

$$\int_0^x \frac{dy}{dx}\,dx = \int_0^x dy = [y(x)]_0^x = y(x) - y(0) = y(x) - 1 \quad ,$$

hence

$$\int_0^x y(x)dx = y(x)-1 \;,$$

or $\int_0^x e^x dx = e^x - 1$ (definite integral) ,

or $\int e^x dx = e^x$ (indefinite integral) ,

i.e. e^x is its own integral. (2.27)

A quick look at figure 2.2 will help us to visualise the properties 2.26 and 2.27.

Normally, a computer manufacturer supplies a package of routines for mathematical calculations and we would then use the expression

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

to return the exponential of X in the storage location Y. (If you have access to BASIC on a micro that doesn't have this package, refer to Appendix 2A.)

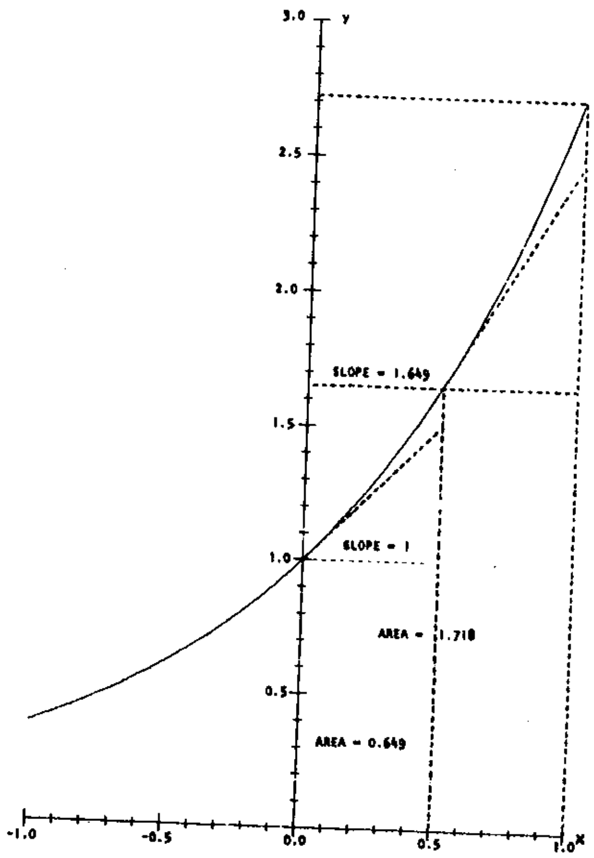


Figure 2.2 The function e^x

2.3 NATURAL LOGARITHMS

The natural logarithm, denoted $\log_e y$ or ℓn y, inverts the relationship given by the exponential

$$y = e^x \ , \tag{2.28}$$

to give $\quad \ell n\ y = x \quad ,$ (2.29)

just as the ordinary logarithm, denoted log y, inverts the relationship given by the antilog expression

$$y = 10^z \ , \tag{2.30}$$

to give $\quad \log y = z \quad .$ (2.31)

From our DE (2.8), rearrangement gives

$$\frac{dy}{y} = dx$$

and then $\displaystyle\int\frac{dy}{y} = \int dx = x = \ell n\ y \quad ,$

hence the important result

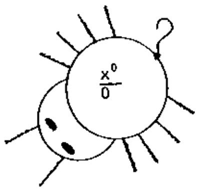
$$\boxed{\int \frac{dy}{y} = \ln y} \quad . \tag{2.32}$$

(Have you ever tried to apply the integration formula

$$\int x^n dx = \frac{x^{n+1}}{n+1}$$

when n = -1

$$\int x^{-1} dx \; (= \int \frac{dx}{x}) =$$



Now we know better

$$\int x^n dx = \begin{cases} \frac{x^{n+1}}{n+1} \; , & n \neq -1 \\ \ln x \; , & n = -1 \quad .) \end{cases} \tag{2.33}$$

On a computer, we use one of the following expressions to return a natural logarithm:

```
    X=ALOG(Y)      - with the FORTRAN language
or  X=LOG(Y)       - with the BASIC language  .
```

A plot of $y = e^x$ is shown in figure 2.3.

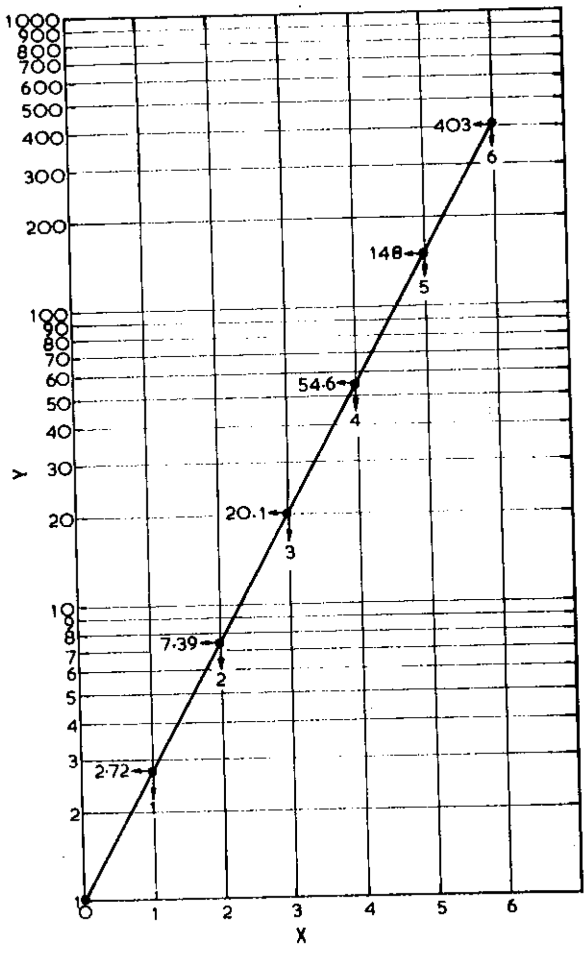


Figure 2.3 I am log paper showing a plot of $y=e^x$

2.4 BACK TO DIFFERENTIAL EQUATIONS

How would we solve the problem

$$\underset{\text{DE}}{\boxed{\frac{du}{dx} = -cu}} \quad , \quad \underset{\text{initial condition}}{\boxed{u(0) = 1}} \quad ? \tag{2.34}$$

Easy: $\dfrac{du}{u} = -c\,dx$

hence $$\int_{\underset{\substack{\uparrow\\ \text{u is 1 when}}}{1}}^{u} \frac{du}{u} = -c \int_{\underset{\substack{\uparrow\\ \text{x is 0}}}{0}}^{x} dx$$

u is 1 when x is 0 - the initial condition,

$\therefore \quad [\ell n\ u]_1^u = -c[x]_0^x$,

$\ell n\ u - \ell n\ 1 = -c(x-0)$,

$\ell n\ u = -cx$, since $\ell n\ 1=0$,

and finally $$u = e^{-cx} \quad . \tag{2.35}$$

(Also equations 2.34 and 2.35 show us that

$$\frac{de^{-cx}}{dx} = -c\ e^{-cx} \quad . \quad) \tag{2.36}$$

How do we go about solving the twin set:

$$\boxed{\begin{aligned} \frac{du}{dx} &= -cu \\ \frac{dv}{dx} &= gu-hv \end{aligned}} \quad \begin{aligned} , \\ , \end{aligned} \quad \boxed{\begin{aligned} u(0) &= 1 \\ v(0) &= 1 \end{aligned}} \quad ? \tag{2.37}$$

Well, we know that the solution of the first equation is simply

$$u = e^{-cx}$$

hence $$\frac{dv}{dx} = ge^{-cx}-hv \quad . \tag{2.38}$$

Hindsight (maybe this was the name of the old maths textbook with the cover missing) suggests the following manipulations:

$\dfrac{dv}{dx} + hv = ge^{-cx}$; then we multiply by e^{hx} (why? Why not!)

$e^{hx}\dfrac{dv}{dx} + he^{hx}v = g\ e^{(h-c)x}$ (and we will assume that $h \neq c$) ,

but the chain rule tells us that

$$\frac{d}{dx}(e^{hx}v) = e^{hx}(\frac{dv}{dx}) + (\frac{de^{hx}}{dx})\,v$$

$$= e^{hx}\frac{dv}{dx} + h\,e^{hx}\,v \quad \text{(it is becoming clear now)}$$

$$\therefore \quad \frac{d}{dx}(e^{hx}v) = g\,e^{(h-c)x} \quad .$$

Integrating the above we get

$$\int_0^x \frac{d}{dx}(e^{hx}v)\,dx = g\int_0^x e^{(h-c)x}dx$$

$$\left(= g\int_0^x e^{[(h-c)x]}\frac{d[(h-c)x]}{h-c}\right),$$

$$\therefore \quad [e^{hx}v]_0^x = g\left[\frac{e^{(h-c)x}}{h-c}\right]_0^x$$

and then $e^{hx}v - 1 = \frac{g}{h-c}(e^{(h-c)x}-1)$.

The final manipulations are of the 'home straight' variety:

$$e^{hx}v = (1 - \frac{g}{h-c}) + \frac{g}{h-c}e^{(h-c)x}$$

$$= \frac{h-c-g}{h-c} + \frac{g}{h-c}e^{(h-c)x} ,$$

$$\therefore \quad v = \left(\frac{h-c-g}{h-c}\right)e^{-hx} + \left(\frac{g}{h-c}\right)e^{-cx} , \qquad (2.39)$$

which is the desired solution.

For this Summer School, the equations are even more involved and so we bring in computers to help us solve them (numerically rather than analytically).

APPENDIX 2A

SOME BASIC FUNCTIONS FOR MICROS IF NOT PROVIDED

(i) $x' = \sqrt{x}$

Input argument variable X

Uses variables X,Y,Z

Output variable X

Calling procedure

```
     GOSUB 31005  -  normal
or   GOSUB 31010  -  if √|x| required
or   GOSUB 31015  -  if x known to be non-negative
or   GOSUB 31020  -  above and y ≃ √x is available as a guess
or   GOSUB 31025  -  above and x known to be non-zero.
```

BASIC coding based on Newton-Raphson method.

```
31000    REM X=SQR(X)
31005    IF X<0 THEN PRINT "SQR ERR";X
31010    X=ABS(X)
31015    Y=X
31020    IF X=0 THEN RETURN
31025    Z=1E38:GOTO 31035
31030    Z=Y
31035    Y=(Y+X/Y)/2: IF Y<Z THEN 31030
31040    X=Y:RETURN
```

(ii) $x' = e^x$

Input argument variable X

Uses variables V,W,X,Y,Z

Output variable X

Calling procedure

```
     GOSUB 31105
```

BASIC coding based on square root method of a previous Summer School (reference on next page)*

```
31100    REM X=EXP(X)
31105    IF X>87 THEN PRINT "EXP OVER"; X : X=1E38 : RETURN
31110    IF X<-87 THEN PRINT "EXP UNDER"; X : X=0 : RETURN
31115    V=INT(X+.5):W=(X-V)/12 : X=1+3*W*W : GOSUB 31015
```

```
31120    X=(2*W+X)/(1-W) : X=X*X*X*X
31125    IF V=0 THEN RETURN
31130    Y=2.718282
31135    IF V<0 THEN V=-V : Y=1/Y
31140    FOR W=1 TO V : X=X*Y : NEXT W : RETURN
```

(iii) $x' = \ell n\ x$

Input	argument variable X
Uses	variables V,W,X,Y,Z
Output	variable X
Calling	procedure

GOSUB 31205

BASIC coding based on square root method of previous Summer School*

```
31200    REM X=LOG(X)
31205    IF X<=0 THEN PRINT "LOG ERR"; X:X=-1E38 : RETURN
31210    V=1 : Y=X
31215    W=X : GOSUB 31025
31220    IF W<.8 THEN V=V*2 : GOTO 31215
31225    IF W>1.3 THEN V=V*2 : GOTO 31215
31230    X=6*(W-1)*V/(W+4*X+1) : RETURN
```

(iv) $x' = x^y$

Input	argument variables X and Y
Uses	variables U,V,W,X,Y,Z
Output	variable X
Calling	procedure

GOSUB 31255

BASIC coding using earlier routines

```
31250    REM X=X↑Y
31255    U=Y : GOSUB 31205 : X=U*X : GOSUB 31105 : RETURN
```

* Newton, P.J.F. ed. [1977] - Down but never out - the mathematics and computation of exponentials arising in the fields of physics, chemistry, biology...AAEC/S19.

CHAPTER 3

MATHEMATICS OF SOME RANDOM PROCESSES

Lecture by

B.E. CLANCY

CONTENTS

3.1 THE MONTE CARLO METHOD AND RANDOM NUMBERS

Many simulation studies use what is called the Monte Carlo method. Here, the average properties of a large group are estimated by constructing possible life histories for the individual members of the group. The construction is done from a knowledge of the forces to which the individuals will be subjected during life. In the course of an individual's life history, there will be occasions when apparently chance situations occur, the outcome of which cannot be predicted. The life history construction procedure must, however, make this prediction; in a computer simulation, this is done by choosing a *random number* from some distribution and letting the value of that number decide the outcome.

The simplest situation for random number generation - and the basis of all other situations - arises when we have to select a number R in the range $0 < R < 1$. In a perfect selection procedure, no number (in the range 0 to 1) should be more likely to be chosen than any other number in the range; this should be true irrespective of how many numbers have been chosen before, and the size of one number chosen should not depend on the size of the previous number selected. We formalise these intuitive ideas with the language of mathematics and describe the way in which values for the random variate R occur in terms of a probability density function (p.d.f), $f(x)$. We say $f(x)$ is a probability density function for the random variate R, if

- (i) $f(x)$ is never negative, and
- (ii) the probability that any selected R will lie between x_1 and x_2 is equal to the area under the curve $y = f(x)$ between x_1 and x_2

These can be stated more formally

$$\left.\begin{aligned} &\text{(i)} \quad f(x) \geqslant 0 \text{ for all } x \\ &\text{(ii)} \quad \int_{x_1}^{x_2} f(x)dx = P(x_1 < R < x_2) \end{aligned}\right\} \quad 3.1$$

The second statement or equation is sometimes stated in the form

- (iii) the probability that any selected R lies in the range x, $x+dx$ is $f(x)dx$ as $dx \to 0$

or

$$f(x)dx = P(x < R < x+dx) \qquad 3.2$$

For our basic random number generation problem, the probability distribution function $f(x)$ is defined by

$$\left.\begin{array}{lll} f(x) = 0 & & x \leqslant 0 \\ f(x) = 1 & & 0 < x < 1 \\ f(x) = 0 & & 1 \leqslant x \end{array}\right\} \quad 3.3$$

and R is then said to be uniformly distributed on the open interval (0,1); this is sometimes abbreviated to $R = U(0,1)$.

3.2 PSEUDO-RANDOM NUMBERS

Computer generation of random numbers for this basic distribution is simply not possible - in part, for the reason that only a finite set of all possible real numbers in the range 0,1 can be represented on a digital computer. What has to be done is to accept a *second best* situation where, at request, the computer generates numbers in the range (0,1) which, to all intents and purposes, seem to have been chosen according to the probability density function of equations (3.3), such numbers being called *pseudo-random*.

A number of techniques for generating a sequence of pseudo-random numbers have been tried in the past. Nowadays, the usual technique is the congruential method; this method generates numbers which seem to be chosen at random from the fractions

$$\frac{1}{m}, \frac{2}{m}, \frac{3}{m}, \frac{4}{m}, \ldots \quad \frac{m-1}{1},$$

where m is a fairly large integer. What has to be done is to generate a sequence of numerators for the fractions. In the congruential method, we choose an integer multiplier $j < m$, a starting integer $i_1 < m$ and compute i_2, i_3 ..., successively by the procedure

$$i_2 \equiv j.i_1 (\text{mod } m)$$
$$i_3 \equiv j.i_2 (\text{mod } m)$$

Here $p \equiv q(\text{mod } m)$ merely means that p is the remainder when q is divided by m. For example, if the modulus m is 7, the multiplier j is 5 and the starter i_1 is 3, the sequence becomes

3	(the starter)	
1	(5 x 3 = 15	= 2 x 7 + 1)
5	(5 x 1 = 5	= 0 x 7 + 5)
4	(5 x 5 = 25	= 3 x 7 + 4)
6		
2		
3		

and the sequence then recycles.

We could thus generate the sequence of pseudo-random numbers 3/7, 1/7, 5/7, 4/7, 6/7, 2/7 with period 6. There are not really enough of these, so the random number generator which is provided for you on the IBM360 computer uses

modulus $m = 2^{48}$ (=281474976710656)

multiplier j = 452807053

and starter i_1 depending on the time of day when the sequence is started.

3.3 RANDOM NUMBERS OTHER THAN R = U(0,1)

With an established procedure for selecting values for our basic uniformly distributed variate R = U(0,1) it is simple to select values for other uniformly distributed variates. For example, if we need values for a variate uniformly distributed on the interval (3,4), we would generate R, as before, and add 3.0 to the result; if we wanted our variate uniformly distributed on the interval (0,2) we would generate R and multiply by 2.0 . In the simulation problem you will be studying, a variate μ will be needed which is uniformly distributed on the interval (-1,1); and here it is clear that we generate R, multiply by 2.0 and subtract 1.0 from the result to get $\mu = U(-1,1)$.

Generating values for random variates not uniformly distributed is a little more complicated. Rather clever methods have been developed for some distributions but, on this occasion, we will consider a general method for generating a value for a random variate X with a probability density function f(x). If the p.d.f. is such that values for X less than 'a' are impossible, then the equation

$$\int_a^X f(x)dx = 1-R \ [=U(0,1)] \qquad 3.4$$

is the key to the generation procedure. If we can carry out the integration and solve the resulting equation for X, the solution gives us a recipe for generating the required values.

An important example which you will need later arises when the p.d.f. is defined by

$$f(x) = 0 \qquad x \leqslant 0$$
$$f(x) = Se^{-Sx} \qquad x > 0$$

with S as a known positive constant.

Equation 3.4 is now

$$\int_0^X Se^{-Sx}dx = 1-R \quad .$$

Since $-Se^{-Sx}$ is the derivative (with respect to x) of the exponential function e^{-Sx}, we carry out the integration to get

$$[-e^{-Sx}]_0^X = 1-R$$

$$-e^{-SX}+1 = 1-R$$

$$\therefore \quad e^{-SX} = R$$

$$X = -\frac{\log_e(R)}{S} \quad . \qquad 3.5$$

In a few words then, to generate values of X for this p.d.f. $f(x) = Se^{-Sx}$, we generate a value R from U(0,1), take its logarithm to base e and multiply by -1/S. Because R is less than 1, its logarithm will be negative and multiplication by -1/S will give a positive result.

3.4 THE PROBLEM TO BE SIMULATED

The simulation problem with which you will be concerned during much of the school arises from a need to design a suitable shield, *i.e.* one in which the radiation dose beyond the shield is reduced to 1 per cent of that with no shield. Experiments done with one, two and three steel slabs will suggest a suitable thickness - number of slabs - to achieve this, but in the nuclear area it is always essential to calculate the resulting system using our knowledge of the physical processes taking place. We do this by simulating the physical system - as we know it - on the computer.

In real life an enormous number of neutrons will enter the shield every second, and no two of these will behave in exactly the same way. However, they all travel through the same environment, they are at each stage of their history subjected to the same laws (probabilistic laws) and large groups of them behave in roughly the same way. You will have seen that of those neutrons entering a single slab shield about 50 per cent will get through. We simulate the average behaviour of the neutrons by taking a group, constructing possible histories for each of them and asserting that this group will be representative of the whole population of neutrons. The laws which the neutrons obey in the shield are as follows:

- All neutrons travel in straight lines until they collide with an atomic nucleus in the shield.
- The probability of travelling a distance r without colliding is e^{-Sr} where S is the total cross section.
- The probability density function f(r) for collision in r,r+dr is $f(r) = Se^{-Sr}$.
- When a neutron collides, it has a fixed probability c of being captured and hence a probability 1-c of not being captured.
- If not captured, the neutron is scattered, loses all knowledge of its previous direction and can start off again in any direction.

In figure 3.1, we have a section of the shield, with the horizontal coordinate x representing the distance into the shield, which itself lies between x = 0 and x = T. The line PQ represents the track of a neutron which had a collision at P, was scattered and then travelled the distance r to Q before its next collision.

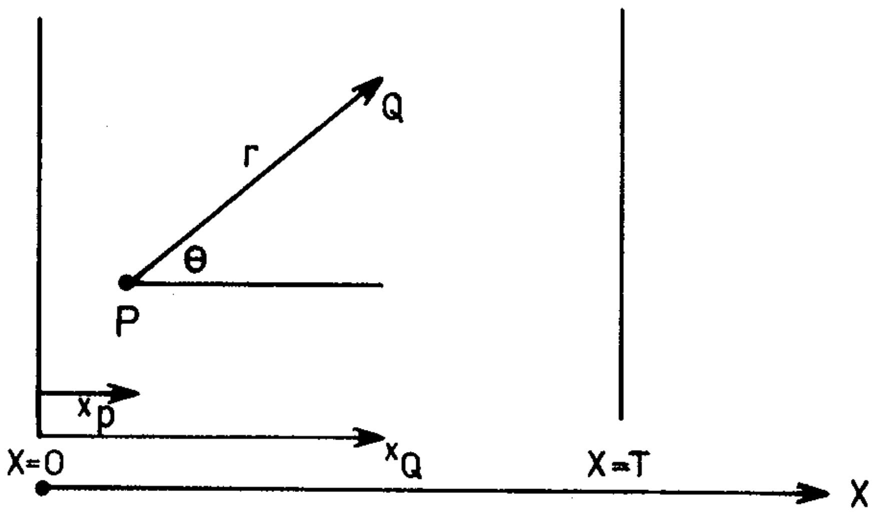


Figure 3.1

At the point P, the position of the neutron is given by $x = x_P$, and its position at Q is $x = x_Q$. The direction of the track PQ is described by the angle θ or simply by μ the cosine of this angle. Since θ can be anywhere between zero and π (0° and 180°), μ will be between +1 and -1. From the usual triangle relationships, we see that

$$x_Q = x_P + r.\mu \quad .$$

3.5 GENERATION OF A LIFE HISTORY

To generate a life history for a neutron we proceed as follows:

1.	Set position coordinate x to zero	(neutrons enter shield from left)
2.	Set direction cosine μ to +1	(neutrons start with angle $\theta=0$)
3.	Generate a distance r to the next collision	(select r from p.d.f. Se^{-Sr}) where S is the total cross section
4.	Change position coordinate x to $x + r.\mu$	
5.	Test to see whether the neutron is still in the shield If neutron is not in shield, terminate history and go to 10	(if x is now < 0 neutron was reflected; if x is now $> T$ neutron has been transmitted) (increase appropriate counter by one)
	OTHERWISE	
6.	Neutron has collided in shield so tally collision	(increase collision counter by one)
7.	Decide whether the collision will result in scattering or capture of the neutron. If decision is capture, terminate history and go to 10,	(select R from U(0,1) and say capture if $R < c$, where $c = \frac{\text{absorption cross section}}{\text{total cross section}}$ If captured, increase capture counter by one)
	OTHERWISE	
8.	Neutron has been scattered, so select new direction cosine μ	(increase scatter counter by one, select μ from the p.d.f. $f(\mu) = 1/2$ for $-1 < \mu < +1$ $f(\mu) = 0$ otherwise *i.e.* select from U(-1,1) because the neutron can be scattered into any new direction)
	AND THEN	
9.	Go back to 3 to continue the history	
10.	The history of this neutron is finished.	

In the marginal comments, various integer counters have been mentioned — one each for collisions, scatters, captures, reflections and transmissions. At the beginning of your program you will set these counters to zero. Some of these will change during the life history of the particular neutron. Whether any changes occur during an individual life history is of little interest - what does matter is their values after a *large* number of histories have been generated, because they provide part of the answer to the problem we have to solve. For instance if, after generating life histories for 100 neutrons, the transmission counter is at 34, then your program has estimated that 34 per cent of the neutrons would be transmitted through the shield. The values in the various counters also provide some useful checks on the logic of your program. Before your program can be trusted to give the right answer, the following should be checked:

$$\text{scatters} + \text{captures} = \text{collisions}$$

$$\frac{\text{captures}}{\text{collisions}} \cong c = \frac{\text{scattering cross section}}{\text{total cross section}}$$

$$\text{reflections} + \text{transmissions} + \text{captures} = \text{the total number of neutrons which entered shield.}$$

3.6 ESTIMATION OF FLUX OUTSIDE THE SHIELD

You will appreciate that the neutrons penetrating the shield cannot in real life be counted like sheep passing through a race; instead an instrument is used to detect some of the neutrons. It turns out that if neutrons leave the shield uniformly over its area and in all possible directions, a single counter will see and detect less of the neutrons which come out at right angles to the shield plane than those that come out at other angles. Also it sees more of those coming out in directions almost parallel to the shield than those at intermediate angles. The detector is a device for estimating neutron flux. Your simulation program must allow for this fact and one simple addition must be made to the procedure for generating a neutron history. One additional counter is necessary - set to zero at the beginning of the program like the others. When the program decides (at step 5 in Section 3.5) that a neutron has been transmitted through the shield, we increase this counter (a flux counter) by the *real* number $1.0/\mu$, where μ is the direction cosine of the neutron track as it leaves the shield. At the end of many histories, the flux counter will contain a *real* number which should be proportional to the neutron flux measured by the physical detector.

3.7 THE SIMULATION PROGRAM

On the basis of what has gone before, the structure of your simulation program is relatively simple; essentially it has six steps

1.	Define values for	total cross section of shield material, capture cross section of shield material, thickness of shield, and number of neutrons which will enter the shield.
2.	Calculate	c = ratio of capture cross section to total cross section, and incident flux at inside of shield = number of neutrons which will enter shield.

3. Set all counters to zero.
4. Generate history for neutron 1. 2. as described in Section 3.5

   etc., until all histories have been generated.
5. Print out values for all counters.
6. Calculate and print out ratio of transmitted flux to incident flux.

Having written and run your program, find the errors and start again. *Good luck*!

CHAPTER 4

NEUTRON REACTIONS AND REACTORS

Lecture by

J.W. CONNOLLY

CONTENTS

4.1 INTRODUCTION

The experiment you will perform during this Summer School will be to measure the attenuation of a beam of neutrons passing through an iron slab. To understand the physical significance of the equations you will use to analyse this experiment, it is necessary to introduce some simple ideas about neutrons and their interaction with nuclei.

4.2 NEUTRONS

For our purposes, we can consider neutrons as spheres of mass 1.66 x 10^{-24} g (~ 1 atomic mass unit) and possessing kinetic energy $1/2\ mv^2$ by virtue of their velocity v. (We will see later that v is a vector quantity and that the direction of neutron travel is important as well as speed.)

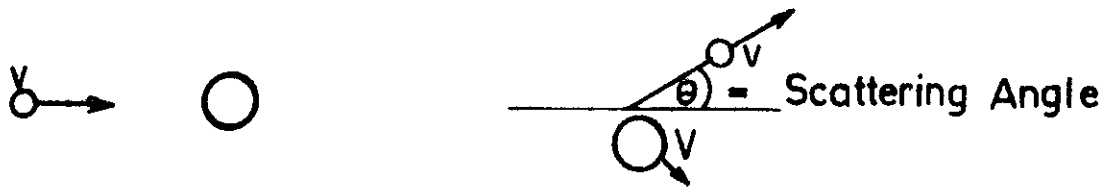
Neutrons carry no electric charge and thus are not subject to electrostatic repulsive forces such as exist between protons and nuclei. Thus neutron-nuclei reactions are possible at low neutron energies.

4.3 NUCLEI

We can consider the nuclei of atoms to be made up of Z protons and N neutrons. The mass number A is equal to N+Z and, since the masses of both neutron and proton are close to unity, on the atomic mass scale, the mass number is the integer closest to the atomic weight of the nucleus. The atomic number Z determines the number of electrons about the neutral atom and thus the chemistry of a particular atom. Nuclei having the same Z but different A are called isotopes, and although their chemical behaviour is identical their nuclear properties are not. We denote isotope X as $^{A}X_{Z}$, *i.e.* the two boron isotopes $^{10}B_5$ and $^{11}B_5$ contain 5 protons each but 5 and 6 neutrons respectively.

4.4 NEUTRON-NUCLEI REACTIONS

Neutrons interact with nuclei in several ways. The simplest is *elastic scattering* in which a neutron collides with a nucleus (which we can consider to be stationary) in much the same way as billiard balls do - *i.e.* the kinetic energy of the system is preserved, the neutron giving some kinetic energy to the nucleus and moving off in a different direction with a reduced kinetic energy:



However, as a result of a collision the neutron may enter the nucleus and give up some of its kinetic energy to increasing the *internal* energy of the nucleus before the neutron escapes with a lower energy. This process is called *inelastic scattering*. The nucleus loses the excess internal energy by the emission of gamma rays.

If the neutron does enter a nucleus, it may or may not be re-emitted. In this case there are several possibilities. The new nucleus might be stable, such as in the reaction

$$^{56}Fe_{26} + n \rightarrow {}^{57}Fe_{26} ,$$

or it might be unstable (radioactive), decaying to a stable nucleus by the emission of charged particles and gamma rays. The decay may be essentially instantaneous, as in the reaction

$$^{10}B_5 + n \rightarrow {}^{7}Li_3 + {}^{4}He_2 \text{ (α particle)} ,$$

or it may take place with a characteristic half-life ($T_{\frac{1}{2}}$) as in

$$^{23}Na_{11} + n \rightarrow {}^{24}Na_{11} \rightarrow {}^{24}Mg_{12} + \beta + \gamma \qquad T_{\frac{1}{2}} = 15 \text{ h.}$$

We call these reactions *neutron capture* or *absorption*. (In the special case of the production of a radioactive atom of measurable half-life, the term *activation* is sometimes used to describe the reaction.)

A capture reaction of particular importance is called *fission*. Certain heavy elements become so unstable upon absorbing a neutron that instead of returning to stability by the emission of a few particles, the nucleus breaks up into two new (and lighter) nuclei and several neutrons. The new nuclei are called *fission products*; these are usually very unstable and form stable nuclei by the emission of β particles and gamma rays. The neutrons released can cause further fissions and it is the production of neutrons in fission that makes a chain reaction possible.

Such a chain reaction, although of scientific interest, is of more importance because each fission event is accompanied by the release of a large amount of energy. This energy release appears because the mass of the fission products and the liberated neutrons is somewhat less than that of the original nucleus plus the neutron. The energy equivalent of this mass difference is given by

$$E = mc^2$$

and mostly takes the form of kinetic energy of the fission products, subsequently appearing as heat as they collide with nuclei in their vicinity. About 200 x 10^6 electron volts of energy is released per fission; compare this with about 10 eV for chemical reactions per molecule. (The *energy* unit, the electron volt (eV), is defined as follows. The volt is the potential difference requiring an energy of 1 joule to transfer 1 coulomb of charge across it. If we define the electron volt as the energy required to transfer one electron (charge = 1.6 x 10^{-19} coulomb) across a potential difference of one volt, then 1 eV = 1.6 x 10^{-19} joule.)

4.5 RATES OF NEUTRON-NUCLEI REACTION

Neutron physics is primarily concerned with understanding the manner and rate of neutron reactions with nuclei. Suppose we consider a material in which there are N nuclei per cm^3. (Since nuclear radii are of the order of 10^{-12} cm and atomic radii are of the order of 10^{-8} cm, the material is mostly 'empty' space.) Into this material we imagine the introduction of a uniform distribution of n neutrons per cm^3 initially at rest. At time t = 0 we give each neutron a speed v but allow the direction of travel to be completely random, *i.e.* just as many neutrons travel in one direction as another. After one second, how many neutrons will have collided with nuclei?

Intuitively, we might expect the collision rate to depend on n, v, N and the size of the nuclei and neutrons. If we suppose the neutrons to be much smaller than the nuclei, and the nuclei to have a cross sectional area A, then in one second the relative velocity of the neutrons and nuclei will define a volume NAv of nuclei space per cm^3 of material. Since there are n neutrons per cm^3, the number of nuclei-neutron collisions will be NA nv per second in each cubic centimetre of material. We call the quantity nv (dimensions of cm^2 s^{-1}) neutron flux and NA (dimensions L^{-1}) the macroscopic cross section of the material. The number of collisions per cm^3 per second is called the reaction rate. Writing NA as Σ, then

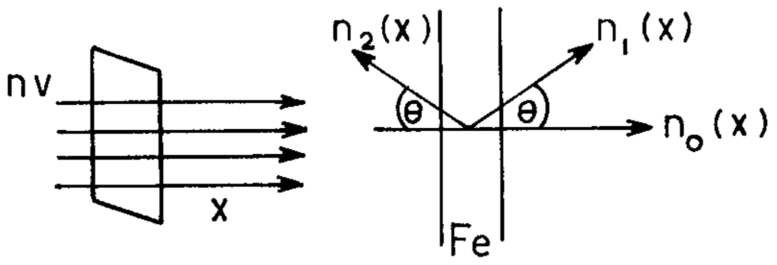
$$\text{Reaction rate} = \Sigma\, nv \quad .$$

Thus Σ is the probability of a neutron-nucleus collision per cm of neutron travel, and is the sum of the individual probabilities that a collision will result in elastic scatter, inelastic scatter, absorption or fission

$$\Sigma = \Sigma_s + \Sigma_{in} + \Sigma_a + \Sigma_f \quad .$$

4.6 THE NEUTRON ATTENUATION EXPERIMENT

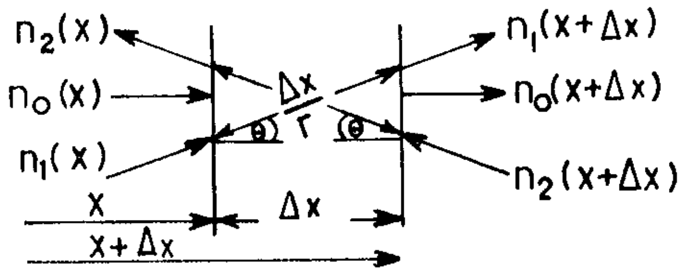
In this experiment you will measure the decrease in neutron density as neutrons travel through increasing thicknesses of iron. Initially the neutrons are all travelling in the same direction - in what is called a beam. The beam flux then is simply the number of neutrons crossing a square centimetre of area perpendicular to the direction of neutron velocity.



Inside the iron slab, however, neutrons will disappear in absorption reactions and have their direction of travel changed by scattering reactions ($\Sigma_f = 0$ for iron). We will assume that, on average, neutrons are scattered either forwards or backwards at the angle θ so that their track length through the iron slab is increased by $1/\cos\theta = 1/\mu$. Thus in the iron slab the neutron flux has three components:

- Neutrons that have not suffered a collision and are moving to the right, that is in the original beam direction. At position x within the slab, we denote the flux of these neutrons by $n_0(x)v$.
- Neutrons that have been scattered in the forward direction at the average angle θ; these are denoted by $n_1(x)v$.
- Neutrons that have been scattered to the left at the average angle θ; these are denoted by $n_2(x)v$.

We now examine a small volume of unit cross sectional area and thickness Δx within the iron slab.



The neutron densities n_0, n_1 and n_2 will be different at x and x+Δx because of scattering and absorption collisions within the volume Δx. These changes form the differential equations you will use to analyse the experiment.

First let us examine the change in the flux of uncollided neutrons. The uncollided flux at $(x+\Delta x)$ is simply that at x less the number of collisions occurring in the element Δx. That is

$$n_0(x+\Delta x)v = n_0(x)v - n_0(x)\Sigma\ \Delta x\ v$$

or

$$\left.\frac{n_0(x+\Delta x)-n_0(x)}{\Delta x}\right|_{\Delta x\to 0} = \frac{dn_0}{dx} = -\Sigma\ n_0(x) \quad . \qquad 4.1$$

The change in the flux of neutrons travelling to the right at the angle θ is a little more complicated. Because the neutrons pass through the unit area at an angle θ, the flux is changed by the factor μ and the volume element by the factor $1/\mu$. The flux at $x+\Delta x$ of the neutrons is equal to the flux at x minus the total number of collisions in the volume element $\Delta x/\mu$ plus the number of collisions resulting in neutrons being scattered to the right at the angle θ.

That is

$$n_1(x+\Delta x)v\ \mu = n_1(x)v\ \mu - n_1(x)v\ \mu\ \Sigma\ \frac{\Delta x}{\mu}$$

$$+\frac{1}{2}\Sigma_s\left[n_1(x)v\ \mu\ \frac{\Delta x}{\mu} + n_2\frac{(x+\Delta x)}{\mu}v\ \mu\ \Delta x + n_0 v\Delta x\right] ,$$

or

$$\left.\mu[n_1\frac{(x+\Delta x)-n_1(x)]}{\Delta x}\right|_{\Delta x\to 0} = \mu\frac{dn_1}{dx}$$

$$= \frac{1}{2}\Sigma_s\ n_0(x) + \left[\frac{\Sigma s}{2} - \Sigma\right]n_1(x) + \frac{1}{2}\Sigma_s\ n_2(x) \ . \qquad 4.2$$

By similar reasoning, we can obtain the change in the density of neutrons moving to the left at the angle θ

$$\frac{dn_2}{dx} = -\mu\ n_0(x)\frac{\Sigma s}{2} - \frac{\Sigma s}{2}\ n_1(x) + \left[\frac{\Sigma s}{2} - \Sigma\right]n_2(x) \ . \qquad 4.3$$

If we now make the substitutions

$$a = \Sigma_a \qquad b = \frac{1}{2}\Sigma_s \qquad c = \Sigma = a+2b$$

$$u = n_0 \qquad v = n_0+n_1 \qquad w = n_2$$

Equations 4.1, 4.2 and 4.3 become

$$\frac{du}{dx} = -cu \qquad 4.4$$

$$\mu \frac{dv}{dx} = c(1-\mu)u - (a+b)v + bw \qquad 4.5$$

$$\mu \frac{dw}{dx} = -bv + (a+b)\ w \qquad 4.6$$

These equations will be solved with the following values of the constants which are appropriate to the mild steel plates to be used for the Summer School experiment:

$$\mu = 0.577$$

$$a = \Sigma_a = 0.184 \text{ cm}^{-1}, \quad b = \frac{1}{2}\Sigma_s = 0.494 \text{ cm}^{-1},$$

and $c = \Sigma = \Sigma_a + \Sigma_s = a+2b = 1.172 \text{ cm}^{-1}.$

CHAPTER 5

ANALOGUE COMPUTING AND DYNAMICS

Lecture by

C.P. GILBERT

CONTENTS

5.1 INTRODUCTION

5.1.1 Dynamic Systems

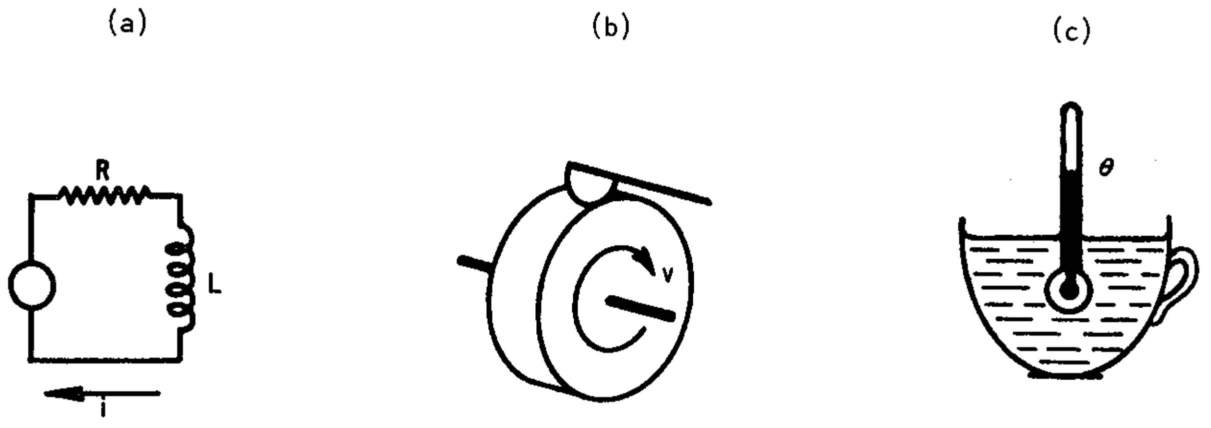
Many advances in science and engineering are possible only because of our ability to use mathematical equations to describe the behaviour of complicated systems.

Here we are mainly concerned with what are known as *dynamic* systems, *i.e.* those that vary with time, which are usually described using differential equations. An example of a dynamic situation is the movement of a ball bouncing on an uneven surface and, if necessary, equations could be formulated to describe this motion.

Having derived methods of representing dynamic systems, it will be shown that they can be used for other types of differential equations.

5.1.2 Analogues

When dynamic systems are examined in detail, one important property emerges. Many electrical, mechanical, biological and other systems can be described by equations of the same *form*, although the actual numbers involved may be different in each case. Figure 5.1 shows simple examples



$$\frac{di}{dt} = -\frac{R}{L}\,i \qquad \frac{dv}{dt} = -\frac{C}{I}\,v \qquad \frac{d\theta}{dt} = -\frac{H}{M}\,\theta$$

Figure 5.1 (a) Current i in an inductive circuit.
(b) Velocity v of a flywheel with a brake
(c) Temperature θ of a cup of hot water

of three systems, each with energy decay. The current i in an inductive circuit, the velocity v of a flywheel with a brake, and the temperature θ of a cup of hot water which is cooling down, can all be described by equations of the form:

$$\frac{dx}{dt} = -Kx \quad . \qquad (5.1)$$

Systems which resemble each other in this way are called *analogues* of one another and, if each is given an equivalent disturbance, they will all behave in exactly the same manner, although at different speeds.

Thus although the three systems are different in physical form, their *dynamic* properties are identical. There is then the possibility that we can examine one of them (that happens to be convenient) in order to find out how the others behave. As we shall see, this can assist us with the solution of very complicated sets of differential equations.

5.1.3 Simulation

One way to examine the dynamic behaviour of a nuclear reactor, say, would be to do the following experiment. A disturbance would be purposely injected, and the reactor power and temperature would be measured as they varied with time. Unfortunately, with a *full-size* power reactor the experiment would be slow, very expensive and possibly dangerous. However, if we could find some sort of analogue of the reactor (*i.e.* another physical system, or *model*, having the same dynamic behaviour), then it would be much simpler, safer and cheaper to do the same experiment on the analogue. This idea has been found to be so successful in some applications that instead of looking for convenient analogues in a haphazard way, special pieces of equipment have been built solely for this purpose.

These *analogue computers*, as they are called, consist of a number of units which can be put together like building blocks to form analogues of different systems; accurate measurements can then be made on the resulting model. The process of doing an experiment on a computer model instead of on the real system is known as *simulation*.

5.1.4 Computing Operations

As will become clear, addition and integration are the most important processes that an analogue computer has to perform, and a number of methods are possible.

Figure 5.2(a) shows how *addition* can be performed using a liquid; if the contents of the smaller containers are emptied into a sufficiently large container, the final volume of liquid is the sum of the initial volumes:

$$V = v_1 + v_2 + v_3 + v_4 + v_5 \quad .$$

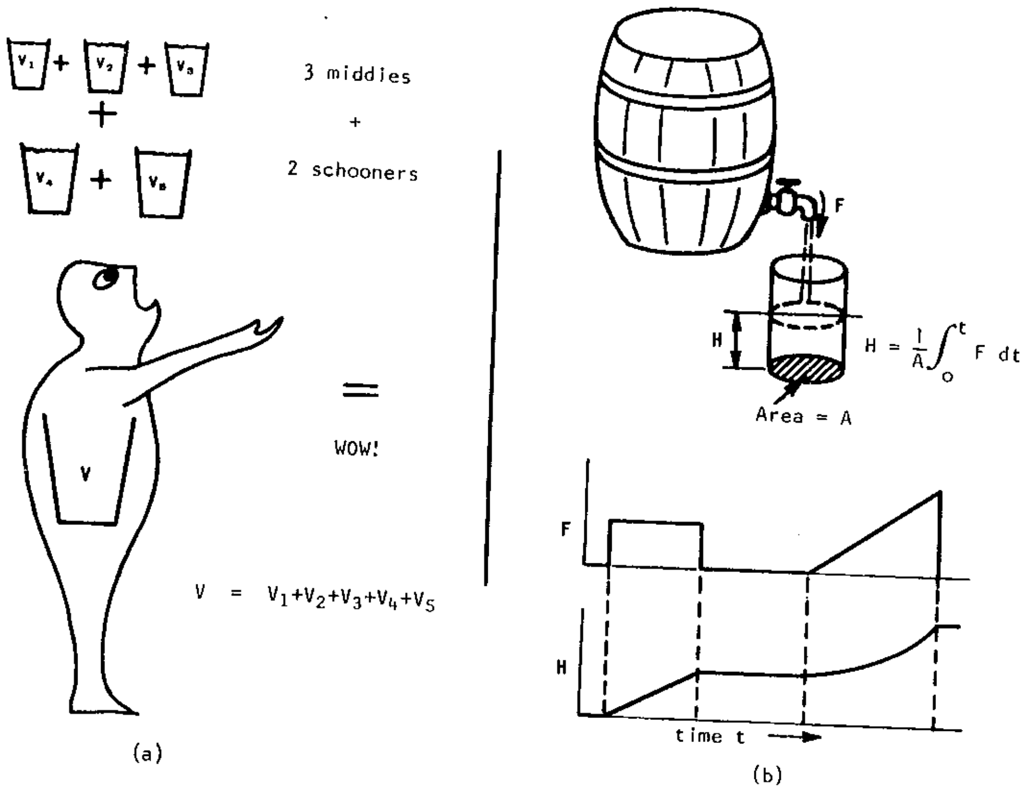


Figure 5.2 Liquid analogues for (a) addition, using volumes, and (b) integration

The more important process of *integration* can be achieved as shown in figure 5.2(b). The height H of the fluid in a container of base area A is the integral, with respect to time, of the fluid flow F (volume/second) as determined by the tap:

$$H = \frac{1}{A}\int_0^t F\,dt \quad .$$

These simple analogues would be of little use in practice; they are slow, unsuitable for interconnection, and the variables are difficult to measure accurately. However, there are much better analogue processes available, the most useful of them using an electronic amplifier as described in the following section.

5.2 ELECTRONIC ANALOGUE COMPUTER COMPONENTS

There are only three analogue components that we need to understand at this stage, the first being an electrical component the others being based on the use of electronic 'operational' amplifiers.

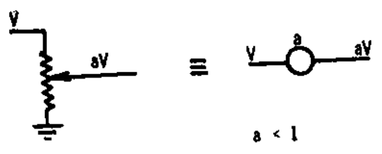
The *potentiometer* is simply a means of reducing the size of a voltage by an amount which can be set very accurately (0.01 per cent). The circuit and conventional representation are shown in figure 5.3(a) in which the input voltage V is reduced to aV at the output, with $0 \leqslant a \leqslant 1.0$. The potentiometer is the only adjustable element in an analogue computer.

The second component to be considered is the *adder*, whose output is the algebraic sum of a number of input voltages. The output voltage is inverted, *i.e.* a positive input gives a negative output and *vice versa*. Each individual input may be increased by a factor of ten if desired, but unless defined otherwise this gain is assumed to be unity.

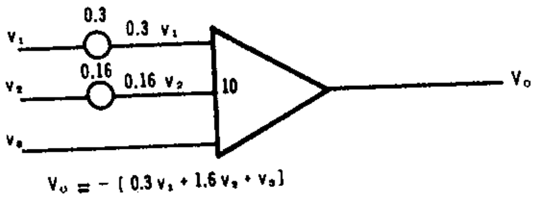
Figure 5.3(b) shows the conventional representation of an adder which, with the potentiometers, gives the relationship

$$V_o = - [0.3 v_1 + 1.6 v_2 + v_3] \tag{5.2}$$

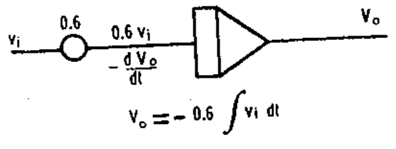
Note that all voltages are measured with respect to earth although the earth connection is not shown. The voltages may remain at a fixed level for many seconds, or may vary at any frequency up to about a MHz.



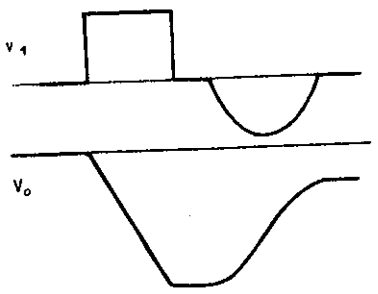
(a) A potentiometer



(b) An adder, using an 'operational' amplifier



(c) An integrator, using an 'operational' amplifier



(d) Integrator wave forms

Figure 5.3 The conventional diagrams for the main computing components

Finally, the most important component is the *integrator*, shown in figure 5.3(c). Like the adder, it inverts and may increase the input voltage by a factor of ten if desired. For the circuit shown, the performance is defined by:

$$V_o = -0.6 \int_o^t v_1 \, dt$$

or $$-\frac{dV_o}{dt} = 0.6 \; v_1 \quad . \qquad (5.3)$$

Thus a constant value of v_1 causes the output voltage to change at a constant rate (figure 5.3(d)), a sine input gives a cosine output and so on. More than one input voltage can be applied, the output then being minus the integral of the sum of the inputs.

Accuracies better than 0.1 per cent can be obtained without difficulty using analogue components of the type discussed.

5.3 ELECTRONIC ANALOGUE COMPUTERS

5.3.1 General

An electronic analogue computer consists of a number of components suitable for addition, integration, multiplication and a range of other functions; facilities are provided which permit the interconnection and switching of the computing circuits, and which allow accurate measurements to be made on them. The *problem variables* in which we are interested (flux, velocity, concentration, temperature or force, for instance) are all represented in the computer by *voltages*. These voltages may vary quite slowly, and can then be read on a voltmeter, or they may change so quickly that an oscilloscope is required to observe them.

A medium-size machine might have about 100 operational amplifiers, including perhaps 30 integrators, and could thus perform 30 integrations at the same time. Also, most analogue computers have a number of logic elements such as gates, monostables, *etc*. This *patchable* logic can be interconnected to suit any given problem.

5.3.2 Equation Solution

Consider the circuit of figure 5.4. The extra input on top of the integrator is inverted and supplies a *fixed* voltage b to the output as an *initial condition* before the integration starts, but has no other effect. When switch A is closed, this *defines* the instant which the computer regards as t = 0, and at this time the output V = b. Because of the potentiometer, the integrator input is λV, and so the circuit obeys the equation

$$-\frac{dV}{dt} = \lambda V \qquad \text{or} \qquad \frac{dV}{dt} = -\lambda V \quad . \qquad (5.4)$$

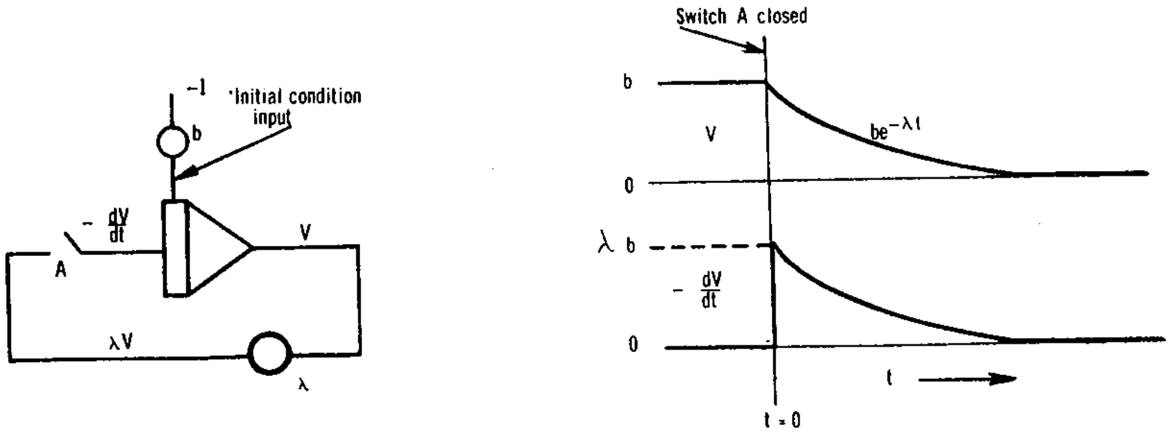


Figure 5.4 A circuit for the solution of $\frac{dV}{dt} = -\lambda V$ and typical waveforms. The input via b only provides the starting voltage

This is of the same form as equation 5.1, which describes the systems of figure 5.1, and so the circuit of figure 5.4 is simply one more analogue, having the same dynamic properties as the other three systems. As you know from a previous lecture, the solution to equation 5.4 is an exponential: if we let $V = ke^{-\lambda t}$, where k is an unknown constant, then differentiating we get

$$\frac{dV}{dt} = -\lambda k e^{-\lambda t} = -\lambda V \quad ,$$

which is the same as equation 5.4. This demonstrates that $V = ke^{-\lambda t}$ is *a* solution of equation 5.4. Since we have made V = b at t = 0, substitution shows that k = b, and so *the* solution is $V = be^{-\lambda t}$.

The circuit of figure 5.4 'solves' equation 5.4 by producing a voltage proportional to $be^{-\lambda t}$ each time switch A is closed. V starts off positive and, *via* the integrator, forces itself to get smaller. As it does so, its rate of change also gets smaller, which is precisely what equation 5.4 tells us in a more compact way.

Switches such as A and many other controls required by the computer are usually omitted from the computing circuit diagram - their presence is assumed.

Summarising, if we should wish to examine one of the systems of figure 5.1, possibly with a very complicated series of disturbances, the simplest and most accurate way of doing the experiment would be to apply a voltage representing the disturbances to the circuit of figure 5.4.

5.4 PROBLEM SOLUTION

To obtain the above solution, we started with a computer circuit and *analysed* its behaviour. The usual process is the other way round - we are given a set of equations and have to *design* a circuit which will solve them, resulting in the process illustrated in figure 5.5. The equivalence between the physical system and the analogue circuit is very marked, and examination of the behaviour of the latter, used as a working model, provides considerable insight into the operation of the original system. In fact, one major advantage of simulation is that it may provide a means of learning and, in some cases, simulators behave so much like the original system that they are used to train operators. Such training simulators are widely used in the nuclear and aircraft industries and, although originally they used analogue computers, they are now more frequently based on digital computers.

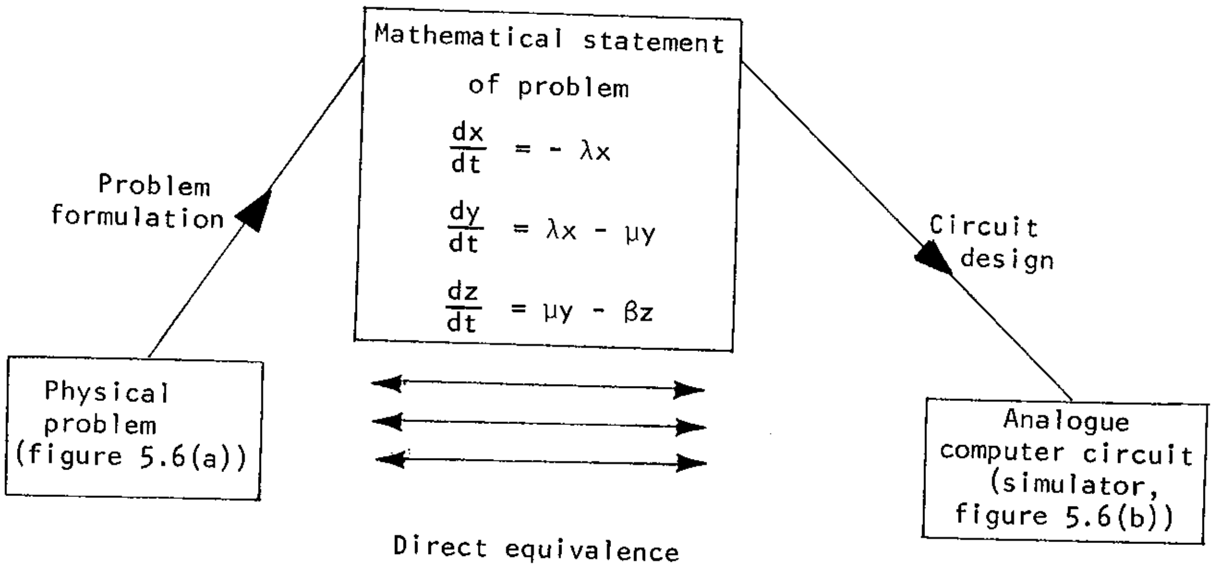

Mathematical statement of problem
$\frac{dx}{dt} = -\lambda x$
$\frac{dy}{dt} = \lambda x - \mu y$
$\frac{dz}{dt} = \mu y - \beta z$
Problem formulation
Circuit design
Physical problem (figure 5.6(a))
Analogue computer circuit (simulator, figure 5.6(b))
Direct equivalence

Figure 5.5 Pictorial representation of the analogue method of problem solving

As an example of the process of figure 5.5, consider the series of tanks in figure 5.6(a), each having a drain hole through which it leaks into the next tank. The depth of water in the first tank is x, and the surface moves (or the depth changes) with velocity dx/dt, which depends on the flow of water in and out. For the first tank the inflow is zero, although the experiment starts (the plug is pulled out) with an initial depth x_o. The outflow depends on the size of the hole, denoted by λ, and the depth (*i.e.* pressure) of water. Thus

velocity of surface = inflow − outflow

i.e. $$\frac{dx}{dt} = 0 - \lambda x \quad ,$$

or $$\frac{dx}{dt} = -\lambda x \qquad (5.5)$$

with the initial condition $x = x_o$ at $t = 0$. By comparison with equation 5.4, we know that x will fall exponentially.

However, the second tank, which starts empty, has an inflow from tank 1 as well as the normal outflow; its rate of change of depth is thus

$$\frac{dy}{dt} = \lambda x - \mu y \quad , \qquad (5.6)$$

with $y = 0$ at $t = 0$. Similarly for the third tank

$$\frac{dz}{dt} = \mu y - \beta z \quad , \qquad (5.7)$$

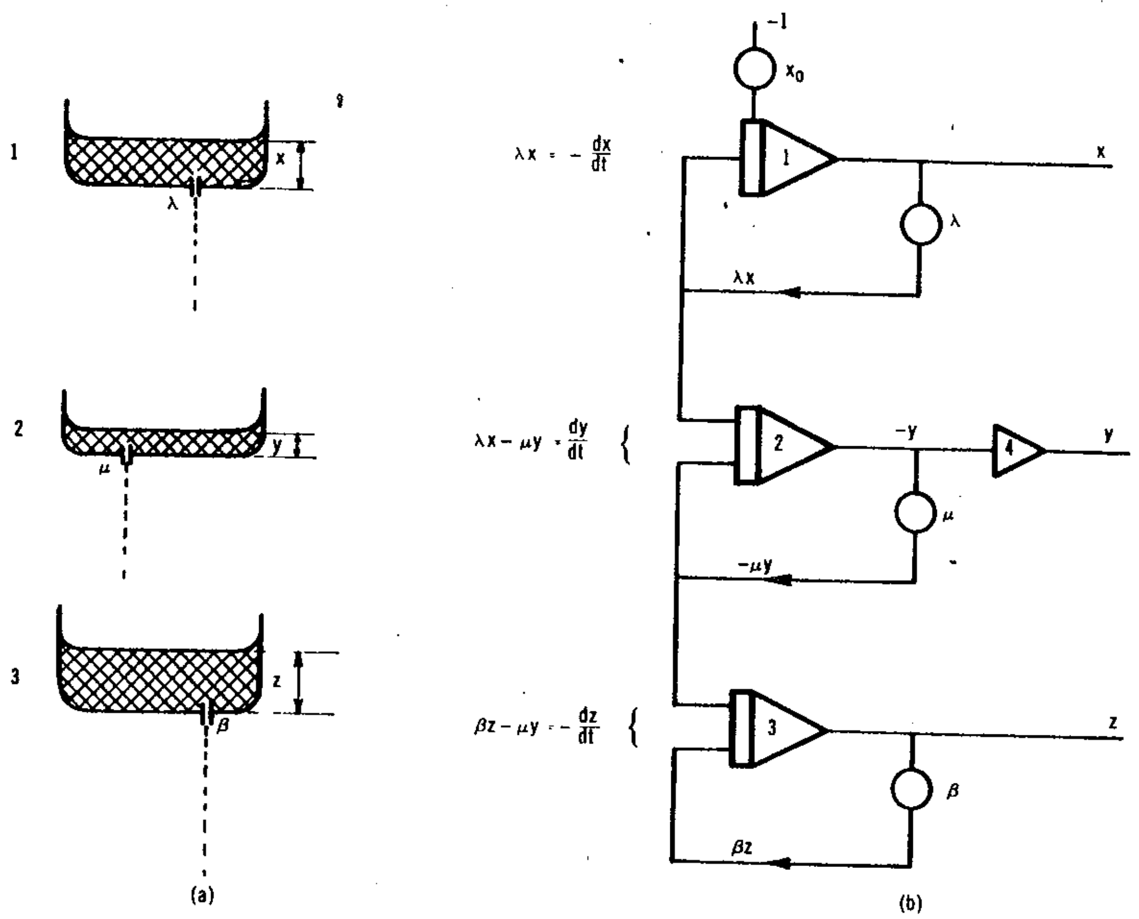
with $z = 0$ at $t = 0$, and one could go on indefinitely.

This system of tanks gives a very clear idea of how one radioactive material decays into another, which itself decays (at a different rate) into a third, because the equations describing that situation are identical to equations 5.5 to 5.7, *i.e.* the two systems are analogous.

Our simple exponential solution only fits the first tank, the change in depth in the others being complicated by their varying inflow. However, we have successfully *formulated* the problem of figure 5.6(a), and can now go on to design the computer circuit.

From figure 5.4, we know that we can solve equation 5.5, using integrator 1 of figure 5.6(b) to represent tank 1. The potentiometer introduces λ, the size of the drain hole (or the decay constant of a radionuclide).

Consider now equation 5.6. Let us assume that a signal representing $-\mu y$ is available; then with the existing λx signal we can make up dy/dt. This is integrated (and inverted) in integrator 2 to give $-y$, and so we can supply the wanted $-\mu y$ signal from the potentiometer. The circuit of integrator 3 solving equation 5.7 can be found in exactly the same way except that the signs are all reversed, and so we have developed an analogue computer circuit to simulate the water levels in the three



1
x
λ
2
y
μ
3
z
β
(a)
-1
x_0
$\lambda x = -\frac{dx}{dt}$
1
x
λ
λx
$\lambda x - \mu y = \frac{dy}{dt}$
2
-y
4
y
μ
-μy
$\beta z - \mu y = -\frac{dz}{dt}$
3
z
β
βz
(b)

Figure 5.6 (a) A problem in hydraulics
(b) A simulator to represent the problem

tanks. An inverting amplifier (4) allows y to be viewed the right way up.

Notice from the demonstration that all the computing operations occur simultaneously (in parallel) not in sequence as in a digital computer, and that the solution arises at a definite speed, *i.e.* the same speed as the levels in the tanks in our case. By using smaller capacitors in the integrators, the problem can be solved at least 1000 times faster and, if many integrations are involved, the overall operation is much faster than can be achieved by a digital computer. However, this high speed cannot be properly utilised by a human operator.

5.5 HYBRID COMPUTERS

A fairly recent development is the *hybrid computer*, which consists of an analogue computer, a general purpose digital computer, and an interface. The latter provides the facilities required for the two machines to cooperate effectively (figure 5.7).

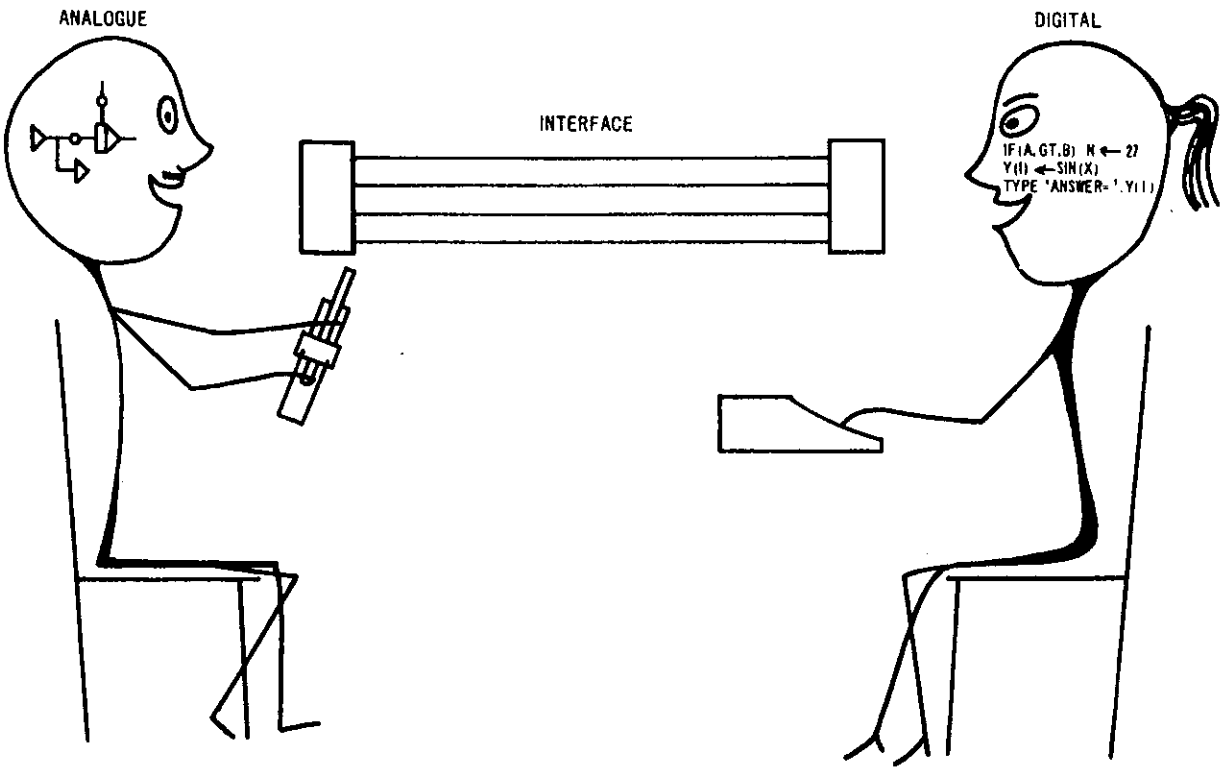


Figure 5.7 A hybrid computer

The analogue computer allows high speed, parallel computation. The digital computer can be programmed to:

- Perform the scaling calculations, and check the connections of the analogue circuit and the settings of the potentiometers.
- Act as a very high speed operator which readjusts the computer before each solution, as determined by the preceding solution.
- Perform parts of the computation which the analogue computer finds difficult.

One might also express the same idea by saying that the analogue computer becomes one of the peripherals upon which the digital computer can call when required.

As an example, suppose we wanted to find the value of λ for an experimental result thought to be an exponential. The operator could use the circuit of figure 5.4 and, by comparing the output with the wanted curve, adjust the potentiometer to get a better fit.

After a number of trial and error solutions he could get a reasonable match and the value of λ would be given by the potentiometer setting. This process would be tedious and inaccurate when performed manually. However, with the digital section of the hybrid computer performing the comparison and resetting the analogue section, many trial solutions would be performed in one second, and an accurate result could be obtained very quickly.

Another application of a hybrid computer is described in the next section.

5.6 SIMULATION OF THE SHIELDING EXPERIMENT

5.6.1 Basic Representation

In Chapter 3, equations were introduced to describe the bulk behaviour of a large number of neutrons as they passed through a series of iron shielding plates. The equations are restated:

$$\frac{du}{dx} = -cu \quad , \tag{5.8}$$

$$\mu\frac{dv}{dx} = c(1-\mu)u - (a+b)v + bw \quad , \tag{5.9}$$

$$\mu\frac{dw}{dx} = -bv + (a+b)w \quad , \tag{5.10}$$

with $u(0) = v(0) = 1$ (*i.e.* normalised to unity), and $w(X) = 0$. X is the total thickness of the plates, and we wish to find v at X, *i.e.* the fraction of the neutrons which eventually arrives at the detector (figure 5.8).

Superficially, these equations are very similar to equations 5.5 to 5.7 for the water tanks, and it would be convenient if it were possible to solve them using the same sort of approach.

The most obvious difference between the two sets of equations is that the independent variable in the tank equations is time t (as it is for the analogue circuits) whereas for the neutrons, the independent variable is distance x. We are already familiar with the idea of a computer voltage representing the depth of water or, in this case, the fraction of neutrons, so we can consider letting computer *time* represent problem *distance*, the relationship being, say,

$$1 \text{ second} \equiv 1 \text{ centimetre.}$$

We know in practice that time can only change steadily at a fixed rate and, provided that we are satisfied with x changing in the same way, there is no difficulty. One method is to start the computer at

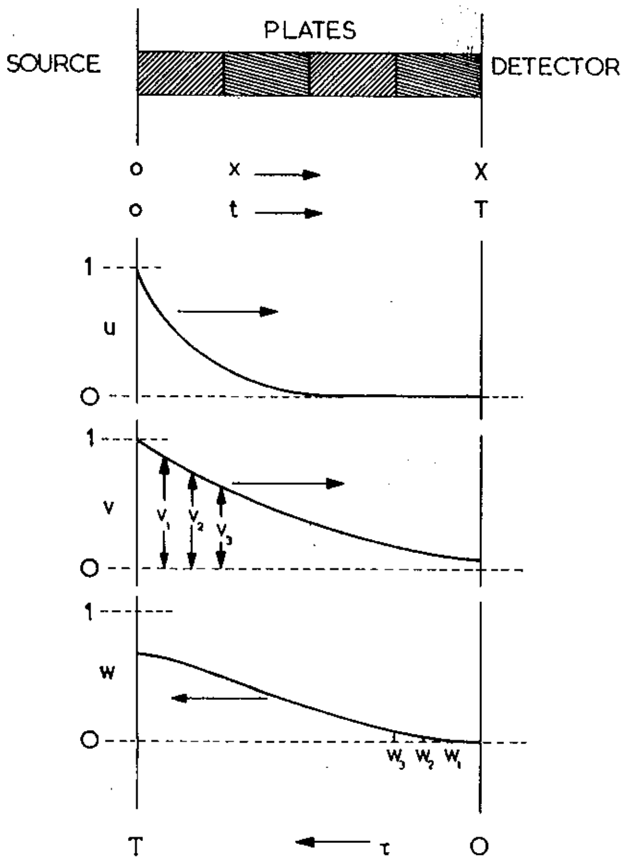


Figure 5.8 Values of u - number of source neutrons moving to the right; v - total number of neutrons moving to the right; and w - number of neutrons moving to the left (backwards). Ten samples such as v_1, v_2, v_3,.... are taken for each plate, but for clarity they are shown more widely spaced.

t = 0 at the neutron source (x = 0) and sweep steadily through the plates at a fixed speed of 1 cm/sec until t = T at the detector (x = X) as in figure 5.8. Unlike the Monte Carlo calculations, the variables would be available for all values of t, although in this case we are mainly interested in the value of v at the end of the run when t = T.

To make this change, t is formally substituted for x in equations 5.8 to 5.10; the opportunity is also taken to divide the second two equations by μ and then to simplify the coefficients. These changes result in the equations:

$$\frac{du}{dt} = -cu \quad , \tag{5.11}$$

$$\frac{dv}{dt} = eu - fv + gw \quad , \tag{5.12}$$

$$\frac{dw}{dt} = -gv + fw \quad , \tag{5.13}$$

where $e = c(1-\mu)/\mu$,

$f = (a+b)/\mu$,

and $g = b/\mu$.

We must also remember the initial conditions

$$u(0) = v(0) = 1 \text{ at } t = 0$$

and the terminal condition

$$w(T) = 0 \text{ at } t = T.$$

5.6.2 Can Time Go Backwards?

The equations now look even more like the tank equations, and it is disappointing to find that they still cannot be solved in the same way. The most important difficulty is that in equation 5.13 the fw term has the 'wrong' sign.

It might be thought that this could be dealt with simply by using an adder to invert the signal (like adder 4 in figure 5.6), but physical reasoning will bring out the difficulty.

In an equation similar to 5.7, considering only the z terms we have

$$\frac{dz}{dt} = \ldots . - \beta z \ldots .$$

which has the same sort of behaviour as the original equation 5.1. From Section 5.4, we recognise that the βz term represents the drainage from the third tank of figure 5.6(a): the greater the depth z, the faster the water runs out, *i.e.* the larger the rate of change in depth in the negative sense.

Consider now

$$\frac{dz}{dt} = \ldots . + \beta z \ldots .$$

which has the same form as equation 5.13. The rate of change of depth is now positive, *i.e.* the tank fills up. Further, the greater the depth of water, the faster the water pours in through the drain! Even if the tank contained only a single drop (molecule?) of water to start with, before long a 'drain' of this sort would force the tank to overflow. For convenience we will call this curious device a negtank*.

* A system of this sort is said to be unstable, and it might be thought that such things could not arise naturally. It is true that they cannot behave in this way for very long, but systems with these characteristics do arise in some circumstances, as discussed in section 5.7.

The problem is that one can never get a tank completely empty; or, in more practical terms, an analogue integrator which is connected to represent a negtank can never have its initial condition made exactly equal to zero, nor its input signal correct within fractions of a micro-volt, and so such a circuit tends to overload long before the computation is complete. A demonstration will illustrate the difficulty (which is not confined to analogue computers).

Let us think about this unexpected behaviour. If we had a normal tank and made a film of the water draining out, the film when projected would show water running out of the drain at a rate determined by the depth in the tank. If now the film was run *backwards*, it would show the tank starting empty and filling up at a rate which increased as the water got deeper, just like a negtank!

Thus one way of thinking about the negtank's curious behaviour is that it is the same as a normal tank working backwards in time.

Now we know that we cannot really make time run backwards but, as you will see, we can easily solve some equations *as though* time was going in reverse. In the physical system under discussion (figure 5.8), time going backwards simply means that equation 5.10 is solved for w *from* the detector (x = X) *to* the source (x = 0) and, since w in fact represents the neutrons moving in this direction, it is an entirely natural thing to do.

It would appear possible then to represent things like negtanks with time running backwards. Thus, starting with equation 5.10, we substitute $-\tau$ for x and, making the same simplifications as before, we end up with:

$$\frac{dw}{d\tau} = gv - fw \qquad (5.14)$$

in which the fw term has the required sign. Not only can this equation be solved easily, but also the initial condition is known. Thus at

$$\tau = 0,\ x = X \text{ and } w(X) = 0$$

and so

$$w(0) = 0 \text{ for } \tau = 0 \quad \text{(figure 5.8).}$$

Summarising then, we can solve equations 5.11 and 5.12 forwards, and equation 5.14 'backwards'. Unfortunately, the w signal found from equation 5.14 cannot be fed directly to equation 5.12 because it is generated in the wrong direction, and v from equation 5.12 cannot be fed directly to equation 5.14 for the same reason.

5.6.3 Function Storage and Replay

This final problem cannot be overcome in a simple analogue computer, but a straightforward solution is possible using the storage facilities of a hybrid computer. In this, the appropriate analogue signals are *sampled* ten times as we integrate through each plate, and these values are *stored* in the memory of the digital section of the hybrid. At some later time, these values are *replayed* to the analogue circuits in the reverse order to that in which they were stored (figure 5.9).

The equations finally solved are derived from equations 5.11, 5.12 and 5.14. Where necessary, v and w have been replaced by v' and w' which are the stored and replayed versions of v and w respectively.

$$\frac{du}{dt} = -cu \quad , \tag{5.15}$$

$$\frac{dv}{dt} = eu - fv + gw' \quad , \tag{5.16}$$

$$\frac{dw}{d\tau} = gv' - fw \quad , \tag{5.17}$$

with $u(0) = v(0) = 1$ at $t = 0$

and $w(0) = 0$ at $\tau = 0$.

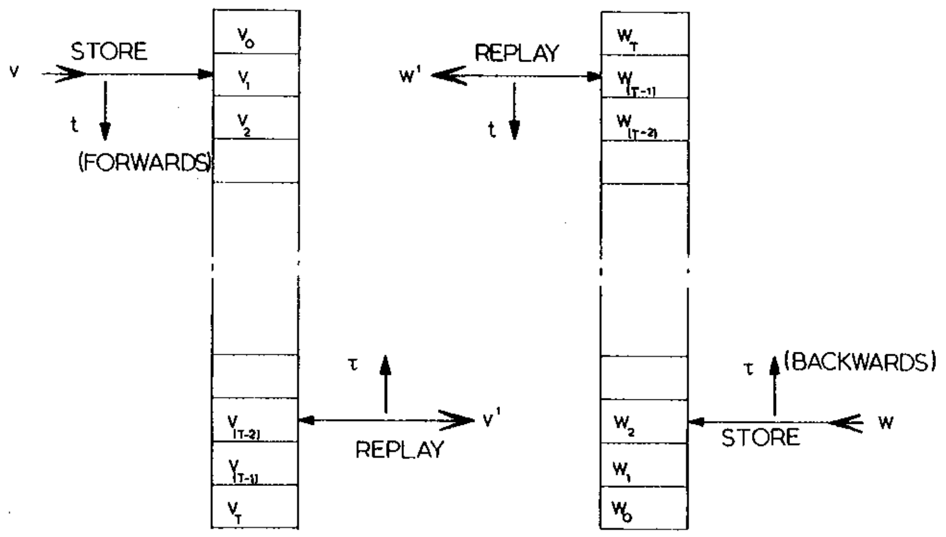


Figure 5.9 Function storage: the series of values shown in Figure 5.8 are stored in the digital computer. During a forward stroke ($0 < t < T$) v is stored and w' is replayed. During a backward stroke ($0 < \tau < T$) v' is replayed and w is stored

The computer circuit is shown in figure 5.10, with integrators 0 and 2 solving equations 5.15 and 5.16 respectively, and integrator 5 solving equation 5.17

Instead of simply running once, the computer alternates between running forwards from 0 to T with only integrators 0 and 2 operating, and backwards from 0 to T with only integrator 5 operating. The individual analogue operations are exactly as described previously, and the digital computer is programmed to manage the timing, switching, storing and replaying processes.

On the first forward run (figure 5.11), equations 5.15 and 5.16 are solved with respect to t, thus generating u and v, the latter being stored. w' is replayed from the digital computer but, since its correct value is not yet known (it is set to zero initially), the values of v are certain to be in error. Next the machine runs backwards, solving equation 5.17 with respect to τ. The stored values of v are replayed in reverse as v' and, although they are in error to some extent, integrator 5 generates values for w which are stored.

On the next forward run, this stored signal is replayed in reverse as w', and allows a more accurate solution for v to be stored. This in turn enables a better solution for w to be found in the next backward run, and so on. After about 12 cycles of iteration the functions converge, *i.e.* there is no change from one run to the next, indicating that the correct solution has been obtained.

All that remains is to measure the value of v at t = T in order to find the fraction of neutrons arriving at the detector. If this fraction is too high, an additional plate must be considered and so a larger X, and hence T, must be used in the next set of solutions.

Summarising, this is a fairly complicated technique for solving the original equations, although once set up it is simple to use, as the demonstration shows. It would be used in practice only if it was important to know the neutron density throughout the iron shielding plates, not just the fraction of neutrons getting to the detector.

5.7 EXPONENTIALS AND EXPERIENCE

5.7.1 Increasing Exponentials

In the previous section, we noted the curious behaviour of a negtank. Although we did not pursue the matter, it is easy to show that its output was proportional to e^{kt}, an increasing exponential of the

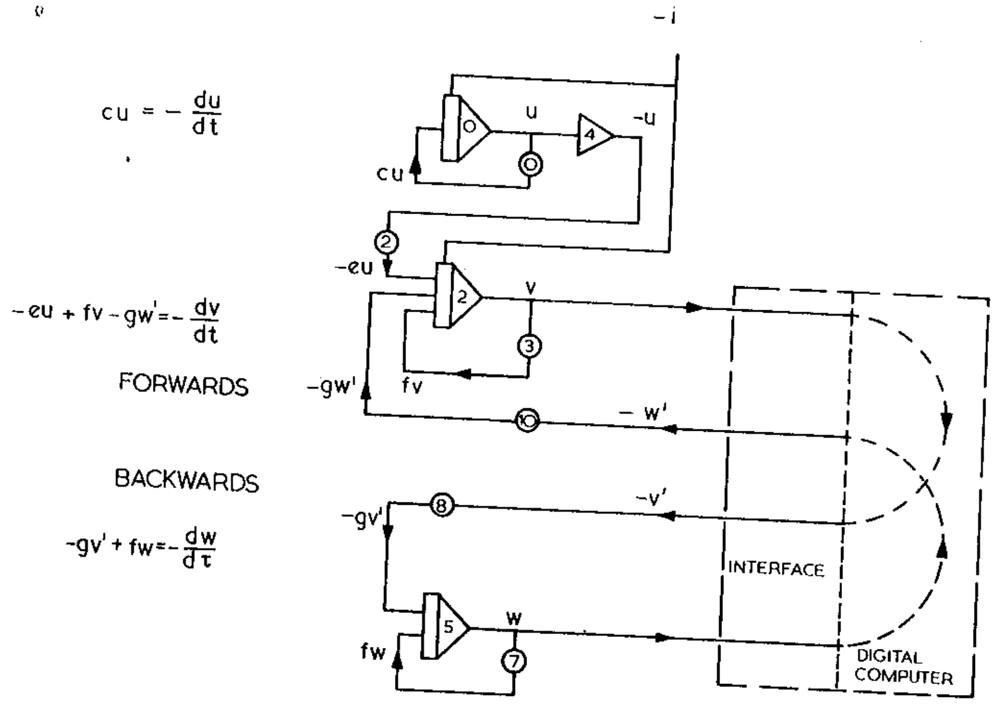


Figure 5.10 The analogue circuit for the solution of equations 5.15 - 5.17. During a forward stroke only integrators 0 and 2 are used. During a backward stroke only integrator 5 operates

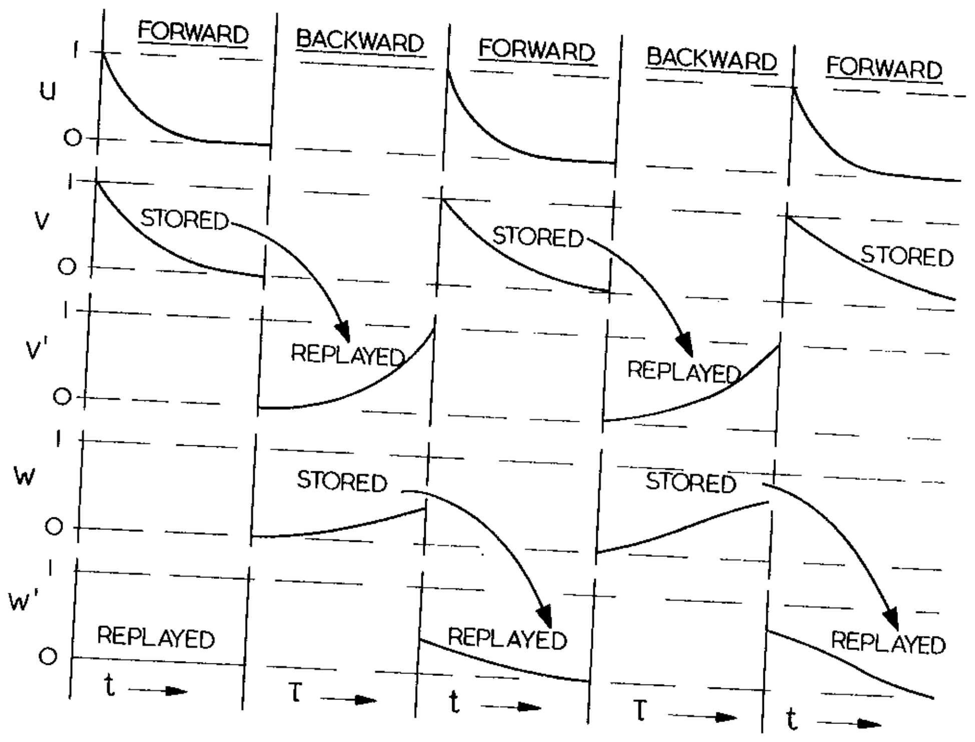


Figure 5.11 The first few iterations of a solution from the circuit of Figure 5.10

type shown in figure 5.12, which can become infinitely large if it continues for a long enough period.

In practice, such a response cannot continue indefinitely; it is always terminated in some way. Thus the negtank overflowed and the amplifiers reached their voltage limit.

Compound interest on a sum of money gives exponential growth; 5 per cent compound interest leads to a 'doubling' time of about 14 years, although in this case the transient is usually terminated by the withdrawal of the money from the bank.

5.7.2 Populations

Consider a colony of 100 grubs. Given sufficient food and space, an average of 20 eggs are produced by each grub per month, of which 10 eggs hatch out and produce grubs which survive to the egg laying stage. Remembering that the original grubs die at the end of the month, the grub population increases by a factor of ten each month (figure 5.12)

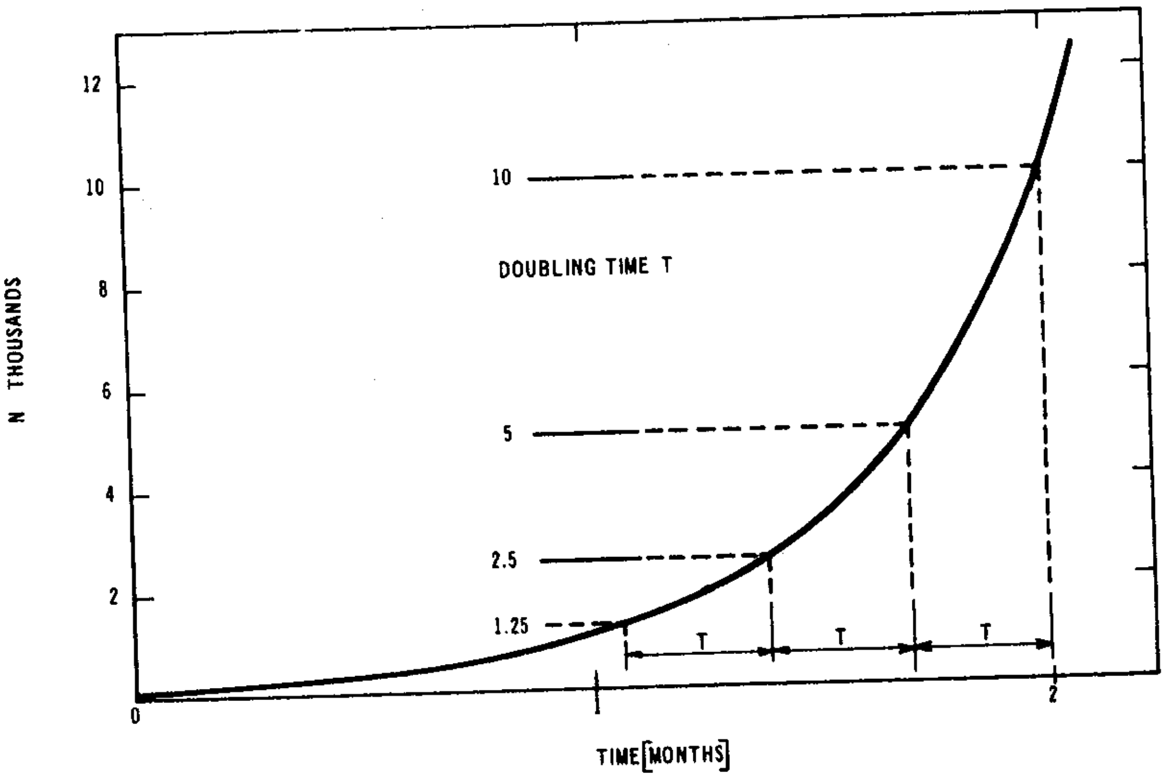


Figure 5.12 The exponential $N=100e^{2\cdot3t}$ showing the growth of a population of grubs

Months	0	1	2	6	12	n
Population N	10^2	10^3	10^4	10^8	10^{14}	$10^{(n+2)}$

(If the grubs are each 1 millimetre long, 10^{14} of them placed end-to-end would stretch for 10^8 kilometres! The moon is only 4×10^5 km away.)

The population curve can be fitted exactly by the equation $N = 100\, e^{2\cdot3t}$, where t is in months, and so the original differential equation must have been

$$\frac{dN}{dt} = 2.3\,N \quad .$$

Thus the grub population is expanding exponentially with a doubling time of about nine days.

The most frightening thing about such an increase is its insidious speed. If you only observe the past (which is usually all that we can do), it is difficult to realise just how quickly things will change in the future, and any delay can turn a difficult situation into an impossible one. Fortunately for us most populations run out of food or space, are subject to unfavourable weather changes, or are eaten by some other animal before their numbers become too alarming.

5.7.3 The Really Super Important Problem

The one population which has been encouraged to expand unchecked is the human population. For over 300 years it has been growing *more* than exponentially, *i.e.* initially with a doubling time of 250 years, but now, at a level of over 3500 million, with a doubling time of only 33 years.

Not only this, but the human race is using up unrenewable resources (oil, coal, metals, *etc.*) and generating pollution at rates which are also growing more than exponentially, both because of the rising number of people and because of a rising material standard of living. Clearly this type of growth cannot continue very much longer or there will be no room left, insufficient food and few raw materials.

In view of what we know of exponentials, it is clear that the human race must manage its affairs better in the future by finding ways of limiting both the population and its usage of the world's resources to levels that our planet can sustain. Unless this is done, nature will apply one of her own traditional methods of limitation — famine and disease, probably preceded and accompanied by war.

Also we know that every delay in coming to grips with the problem makes matters worse — in fact a delay of only about 30 years doubles the size of the problem.

The past four or five generations have worked hard to provide food and better material standards of living, and so have helped to increase the population and accelerate the use of natural resources. The present generation is continuing to do this, owing to sheer inertia and bewilderment, but at least it has realised that a serious problem exists. It will be the responsibility of the next generation to start dealing with these formidable difficulties.

CHAPTER 6

F O R T R A N

Lecture by

J.M. BARRY

CONTENTS

6.1 INTRODUCTION

Each digital computer is capable of obeying a number of basic instructions. These instructions vary for different computers but they have many attributes in common:

(i) The ability to perform the four arithmetic operations (+, -, ×, ÷).

(ii) The ability to perform logical operations (is $A \geqslant B$?).

(iii) The ability to perform 'housekeeping' instructions (*e.g.* moving numbers from core store to registers where arithmetic and logical operations may be performed on them).

For a programmer to communicate with the computer at this most fundamental level, it would be necessary for him to develop programs in the basic machine language of the computer at his disposal. In the early days of computing, it was necessary for scientists and mathematicians to concern themselves with the intricacies of binary coding. The long delays and inconvenience of this form of man-machine communication accelerated the growth of programming languages that the problem solver could use more readily. Many languages (FORTRAN, BASIC, COBOL, ALGOL, PLI, APL, ACL, PASCAL, *etc.*) have been developed for scientific, commercial and other applications. FORTRAN is chosen as the main vehicle for problem solving at this Summer School because throughout the world it is the most accepted scientific computing language. There are no computers that obey programs written in FORTRAN directly. It is necessary for programs in 'high level' languages such as FORTRAN to be translated into an appropriate set of machine language instructions. This process is known as *compilation.*

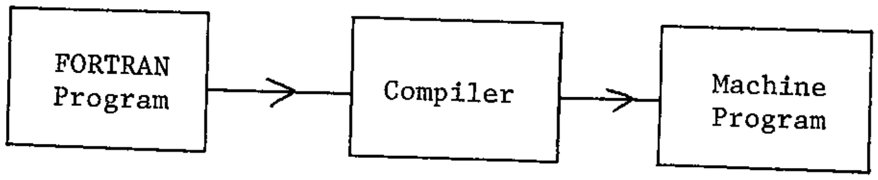


Figure 6.1 - Compilation of a FORTRAN program

The FORTRAN source statements are translated to a set of machine language instructions by a FORTRAN compiler (figure 6.1). The compiler is itself a program (usually supplied by the machine manufacturer) that first checks to ensure the FORTRAN statements obey the 'rules' of the language (syntax analysis), and then supplies a set of machine instructions that will implement what the programmer has specified. When the compilation process is completed, the machine instructions generated may be *executed*. The finer details of this process and the way it is implemented on the IBM360 will not be our concern at this Summer School as we are interested primarily in using the computer as a tool for mathematical problem solving.

6.2 OVERVIEW OF FORTRAN PROGRAMMING

Let us first consider the steps involved in solving a sample problem, and the FORTRAN program that could be developed to carry them out. When this is done we shall examine the various FORTRAN statements in closer detail.

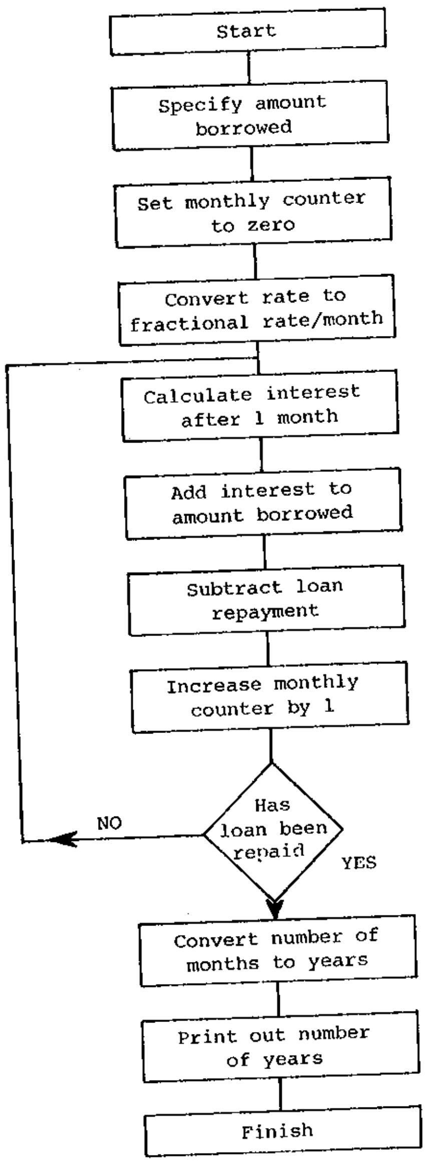

Start
Specify amount borrowed
Set monthly counter to zero
Convert rate to fractional rate/month
Calculate interest after 1 month
Add interest to amount borrowed
Subtract loan repayment
Increase monthly counter by 1
Has loan been repaid
NO
YES
Convert number of months to years
Print out number of years
Finish

Figure 6.2 Flow chart for compound interest problem

Problem If $18000 is borrowed at a rate of 11% (monthly reducible) and repayments of $300 each month are made, then how many years will it take to repay the loan?

Before we can program a digital computer to solve a problem, it is necessary for us to be able to detail logically the steps that are needed to solve the problem, in much the same way as we would if we were going to tackle the problem with a desk calculating machine, slide rule, or pen and paper. Some people find it helpful to draw a flowchart (figure 6.2) showing the steps involved, whereas others prefer to visualise all the steps in their mind.

From this flow chart the following program can be coded. At this point we will not concern ourselves with the formal rules for coding but just look at the end product (figure 6.3).

```
1   5 6 7                                                           7273   80
C        PRØGRAM BY J.M. BARRY TØ DETERMINE THE NUMBER ØF
C        YEARS NECESSARY TØ REPAY A LØAN.
C        THE PRINCIPAL BØRRØWED, INTEREST RATE AND MØNTHLY REPAYMENT
C        ARE TØ BE READ FRØM A PUNCHED DATA CARD.
         READ,PRINC,RATE,PAYMNT
         MNCNTR=0
         FRATEM=RATE/(100.*12.)
       1 ADPRIN=PRINC*FRATEM
         PRINC=PRINC+ADPRIN
         PRINC=PRINC-PAYMNT
         MNCNTR=MNCNTR+1
         IF(PRINC.GT.0.) GØ TØ 1
         YEARS=MNCNTR/12.
         PRINT,YEARS
         STØP
         END
```

The data card sufficient for this problem would be

```
1                                                                          80
18000.        11.        300.
```

Figure 6.3 - Sample program for compound interest problem

6.3 PUNCHING OF CARDS

To assist in the punching of cards, programmers usually use a standard coding sheet (as shown in the following example) representing the 80 columns available on a punched card. Each line of the sheet represents a new card which may contain only one statement.

1	5	6	7	72 80
C			J. SMITH STATEMENT EXAMPLE	THIS
C			THE ABØVE IS A CØMMENT	SECTION
			X=A+B	NOT
	50		Y=9.-C	USED
			P R INC = RATE * PRINC/100. + PRINC	IN
			SUM = A+B+C+D+E+F+G+H+	FORTRAN
		1	Ø+P+Q+R	PROGRAMS

Statements can be punched from columns 7 to 72. To assist the programmer to recall aspects of a program, a comment card (denoted by a C in column 1) may be placed anywhere within the punched deck. These are ignored by the FORTRAN compiler. Normally we shall commence our programs with a comment card to assist with the identification of the program.

Columns 1 to 5 inclusive can be used if desired to assign a statement label in the form of a number in the range 1 to 99999 (there is no need to choose labels in ascending order).

Blanks may be inserted within a statement to make it more readable, and may be considered as being removed in the compilation process. If a statement is too long to fit on one card, it is continued from column 7 of a subsequent card provided that column 6 of this card contains a continuation character (any character other than a blank or zero will suffice as a continuation character).

The character set available within the FORTRAN system consists of

(i) 26 capital letters A,B,C,...,Z ;

(ii) 10 numerals 0,1,2,...,9;

(iii) 10 special characters +,-,*(multiplication),/(division) ,.,',(,),=,$; and

(iv) a blank (usually written ƀ if its presence is to be emphasised for punching).

6.4 ARITHMETIC CONSTANTS

We will treat three different types of constants sufficient for handling data (numbers) in most scientific problems:

(i) *INTEGER* (or fixed point) constants.

- a whole number without a decimal point whose absolute value is $\leqslant 2^{31} - 1 = (2147483647)$.

Valid integer constants   0   -5   +357   7005192

Invalid integer constants 27. 5,132   9812735997

(ii) *REAL* (or floating point) single precision constants

- up to seven decimal digits with a decimal point, with or without an exponent. The absolute magnitude is approximately 10^{-78} to 10^{75}.

Valid real single precision constants

+0.   7.91   5.3E+2(=5.3 x 10^2)

5.3E2(5.3 x 10^2)   -.051E-03 (-.051 x 10^{-3})

Invalid real single precision constants   1   3,471.2   1.E

(iii) *REAL* (or floating point) double precision constants

- similar to (ii) but up to 16 digits are possible with a D exponent being necessary instead of E. Double precision constants will not be necessary for the Summer School problems.

Distinctions are carefully drawn between the three types of constants for electronic rather than mathematical reasons. The electronic 'hardware' necessary for *INTEGER* arithmetic operations is less sophisticated and consequently for most machines is faster than that used for *REAL* arithmetic. By performing those operations that require no decimal point in integer mode, considerable time savings can be made.

6.5 VARIABLES

A variable is a symbolic name used to identify a data item that will occupy a location (one word) of core storage. The actual address of this location is assigned by the compilation process. If we move a number into a variable it will replace the previous contents of that location.

```
TIME=0.
```

This places zero in the location reserved for TIME. When a transfer is made from a location, the previous contents remain unaltered.

```
X=TIME.
```

This assigns the contents of the location reserved for TIME to that reserved for X without altering the contents of the location associated with TIME.

The '=' operation should be interpreted as the assignment of the result of the right hand expression to the left hand location. Consequently, an expression such as

```
A=A+1.
```

does not yield any algebraic result but rather is interpreted as increasing the old value associated with A by 1. to give a new result also called A. The old value of A is, of course, lost.

Variable names may have up to six characters (special characters are not permitted) the first of which must be alphabetic such as

```
TIME , X3B , I5 , T .
```

Variables like constants take an *INTEGER* or *REAL* form. Unless the programmer provides specifications to the contrary, all variables commencing with I,J,K,L,M or N are *INTEGER* variables, whereas the remainder are single precision *REAL* variables.

Variables may also be subscripted in FORTRAN. Such variables may be used to represent vectors or matrices which you probably have encountered in your mathematics courses:

V(3) is the FORTRAN representation of the vector component v_3 ,

A(3,4) is the FORTRAN representation of the matrix element a_{34}.

(Further consideration of *SUBSCRIPTED* variables will be delayed until section 6.12.)

6.6 INPUT AND OUTPUT

One way of assigning values to variables is through the direct use of an arithmetic expression:

```
X=6.3
```

Should one wish to alter the data on which the program is to operate without changing the program itself (a most frequent requirement), then a READ statement is needed.

The READ statements initiate the reading of data cards (which are physically separated from the program cards) and result in the transfer of numbers punched on these cards to variables in the READ lists.

$$\text{READ, }\underbrace{\text{PRINC,RATE,PAYMNT}}_{\text{list of variables to be read}}$$

Numbers are read from a punched card and stored in the three variable locations PRINC, RATE and PAYMNT. (Any numbers remaining from an earlier card are ignored.) The data can be supplied in what is called free format. This means that no specific columns of the card are required for the various data items. It is sufficient to leave at least one blank column or have a comma between each number in order to define it.

The output statement PRINT allows free format output and functions in a similar manner. For example, the output statement

```
PRINT, 'NØ ØF YEARS =', YEARS
```

would display on the printer, output of the form

```
NØ ØF YEARS =        0.2631400E+02
```

As you can see, free format input and output (I/O) has certain advantages in that the FORTRAN rules for its use are easily understood. There is not a great degree of control over the layout, however, and this is particularly unfortunate when preparing printed output, particularly if one wishes to make it as pleasing to the eye as possible. In addition, free format as described here is only available for the particular FORTRAN compiler (WATFIV) used at the Summer School. Appendix 6B contains examples of format-controlled I/O as a guide for the more advanced reader.

6.7 ARITHMETIC OPERATIONS AND EXPRESSIONS

Five arithmetic operations are available to FORTRAN users:

(i)	addition	+	*e.g.* A+B
(ii)	subtraction	-	*e.g.* A-B
(iii)	multiplication	*	*e.g.* A*B
(iv)	division	/	*e.g.* A/B
(v)	exponentiation	**	*e.g.* A**3 (A^3)

Expressions may be enclosed within parentheses as in normal algebra:

$(a+b)\ (c+d)$	(A+B)*(C+D)
$(a+b)^2$	(A+B)**2
$\frac{a}{bc}$	A/(B*C)

Parentheses are necessary to prevent two operations from appearing next to each other (should such a combination be possible)

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

The sequence of operations in expressions is determined from the following hierarchy and is consistent with normal mathematics:

(i) **

(ii) */ left to right precedence

(iii) +- left to right precedence.

Consequently, the expression

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

could have been correctly abbreviated to

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

The integer variables or constants deserve special mention. Division of one integer by another results in the truncation of any fractional remainder.

```
I=9
K=I/2
```

would result in K taking the value 4. This property can often be exploited to the programmer's advantage when the testing for even integers;

```
K=I-I/2*2
```

would assign 1 to K if I is odd, and 0 if I is even.

Expressions should consist of variables or constants all in the same mode (*i.e.* all *REAL* or all *INTEGER*). (This is required by standard FORTRAN. Most compilers allow mixed mode as an extension. This is the case with WATFIV. The user of mixed mode arithmetic must, however, be aware of the way in which it is designed to function, or else be ensnared. For simplicity it is best avoided.)

There is one exception to this rule in that the exponent of a *REAL* variable or constant may be *INTEGER*. The following are permitted forms of exponentiation:

```
V**2            V**A
(-V)**.45       V**(-I)
V**(-2)         I**3
```

The mode of a variable on the left hand side of an arithmetic assignment need not be the same as that of the expression on the right.

```
A=I+1
```

The compiler will arrange for the right hand side to be evaluated in *INTEGER* mode and the result to be converted to the *REAL* mode before it is stored away. Because of truncation in *INTEGER* division, great care should be exercised in using this type of arithmetic.

6.8 SUPPLIED MATHEMATICAL FUNCTIONS

As there are a number of special mathematical functions or operations that are common to many problems, the FORTRAN compiler provides these as part of the normal system. To calculate the exponential function $x=e^t$, all we need do is code

```
X=EXP(T)
```

To use a supplied mathematical function, it is only necessary to follow the function name by an argument enclosed in parentheses. The result will be returned as though the function name itself designated a variable in the program. The argument may be a variable, constant or arithmetic expression

```
A=EXP(A-C)+SQRT(15.)
```

A list of frequently required functions follows:

Mathematical Function	Function Name (Argument)
square root,	SQRT(X)
exponential, e^x	EXP(X)
natural logarithm, log x (or *ℓn* x)	ALØG(X)
sine of an angle (in radians), sin x	SIN(X)
cosine of an angle (in radians), cos x	CØS(X)
tangent of an angle (in radians), tan x	TAN(X)
arctangent (result in radians), $\tan^{-1}x$	ATAN(X)
absolute value (real numbers), $\lvert x\rvert$	ABS(X)

Functions other than those supplied through the compiler are often necessary, so FORTRAN allows a programmer to name and define his own special functions (section 6.13).

In the Summer School, frequent use is made of random numbers. Details on what constitutes a series of pseudo-random numbers (these are the ones we actually produce on a computer) will be given in other lectures, however, we should note that random number routines are far from trivial. Considerable mathematical and computer literature has been devoted to this subject. At this stage, however, it is important to show how one may generate them as part of a FORTRAN program. Because the random number generator is not a required component of a standard FORTRAN system, and because random number generators are frequently supplied by individual users, they are not included in the above list of functions. There are two random number functions available for the Summer School. These are

RAND(X), and

RND(X)

The first function generates a sequence of random numbers starting each time with the same number, so that the sequence generated can be repeated at a later time. It is a little like having purchased a printed book of random numbers (Yes! Such things are available.) and commencing each time at the start. This can be of use when developing and testing a new program, in which case a constant environment may be of assistance. The second function produces a random sequence of random numbers. You may find this fits in more nicely with your basic intuition. The second generator is much faster than the first; consequently, it will enable you to handle many more neutrons during the computer time allowed for the Summer School problem. Just how many neutrons are sufficient will be open to discussion later; for the moment we quote the maxim 'enough is enough'.

Both generators can be invoked in a similar manner

```
Y=RND(X)
```

This results in a random number being assigned to the variable Y. This number lies in the range

$$0 < y < 1 \quad .$$

Unlike all the other functions, the argument X serves no meaningful purpose. It is not used by the random number generator and is only there because FORTRAN (remember it is the Boss) says that all functions must have at least one argument. Failure to code such a dummy argument would cause the FORTRAN compiler to reject your program because it was not properly attired.

It is important to observe that the random number generator is the most frequently written program of all time. Take particular care that your programs are not unintentional attempts to make random number producers (see section 6.14).

6.9 TRANSFER OF CONTROL

Execution of a program will commence at the first executable statement and proceed through subsequent instructions in order, unless a transfer of control statement is encountered. The simplest means of transfer of control is through an 'unconditional GØ TØ' statement.

```
56 READ,X
   PRINT,X
   GØ TØ 56
```

This section of program would cause cards to be read and printed with no escape mechanism until the supply of punched data cards was exhausted, in which case an error condition would cause the program to fail. Clearly such a statement alone would be of limited use.

There is an extension of this statement, known as the 'computed GØ TØ', which gives a little more choice in the statement to which the branch is to be made:

```
GØ TØ (71,56,1,9),I
```

If I=1 control passes to statement 71
If I=2 control passes to statement 56
If I=3 control passes to statement 1
If I=4 control passes to statement 9

For any other value of I, control would pass to the next sequential statement in the program.

Note that if I is a random integer produced by

```
I=10*RND(X)+1
```

then a computed GØ TØ

```
GØ TØ (1,2,3,4,5,6,7,8,9,10),I
```

could be used to select 1 of 10 options with equal probability.

The most useful form of the transfer of control statement is the 'logical IF' as demonstrated in our first sample program:

```
IF(PRINC.GT.O.)GØ TØ 1
```

If PRINC is greater than zero, then control will pass to the statement labelled 1. The logical IF statement can be considered to be of the form

IF (logical expression) restricted executable statement

The logical expression can take one of two values only, .TRUE. or .FALSE. In a logical IF, the statement appended will be executed only if the logical expression returns a .TRUE. result. When it is .FALSE. the appended statement is ignored and control will pass to the next statement:

```
      IF(A.LT.B)GØ TØ 56
      PRINT,B
   56 A=B*C
```

If A < B, then A will be recalculated as the product of B and C. For A ⩾ B, the value of B will be printed first.

Although the appended statement is frequently a 'GØ TØ' statement, it may be any executable statement other than another 'logical IF' or a 'DØ' statement (section 6.10).

```
      IF(A.LT.0.)A=-A
```

This would be sufficient to replace A with its absolute value although the coding would be somewhat slower than if using the alternative statement

```
      A=ABS(A)
```

Logical expressions are most frequently formed by two arithmetic expressions and a relational operator:

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	$a \leqslant b$

e.g.

```
      IF(A+B.LE.C+SQRT(X**2*Y**2))A=1.
```

Frequently, we wish to carry out more than one logical test at a time. This can be done by combining logical expressions with one of the following logical operators:

.AND. both expressions must be .TRUE. to return a .TRUE. result

.ØR. result is .TRUE. if either expression is .TRUE.

```
      READ A,B,C
      IF(A+B.GE.C.AND.A+C.GE.B.AND.B+C.GE.A)PRINT,'TRIANGLE',A,B,C
      STØP
      END
```

The above program will read three values for A,B and C respectively from a punched data card (not shown here) and will test whether the values A,B and C are capable of being the lengths of the sides of a triangle. As before, if the combined logical expression is .FALSE., then control will pass to the next statement. IF it is .TRUE., it will first print out the message and specified data.

6.10 LOOPS

We frequently find it necessary to repeat a section of code a given number of times. Suppose our problem is to find the sum of the first 20 integers, *i.e.* 1+2+...+20. Ignoring any appeal to mathematical analysis, the following code would be sufficient:

```
      .
      .
      .
      ISUM=0
      I=1
    5 ISUM=ISUM+I
      I=I+1
      IF(I.LE.20)GØ TØ 5
      .
      .
      .
```

In this example, ISUM is chosen as a variable name to accumulate the sum of the integers (integer variables start with I,J,K,L,M or N). It is first necessary to initialise this to zero and the counter (I) to 1. Two statements are then necessary to increase the counter and test it to determine whether the loop is complete, and to transfer control back if it is not. Because scientific programming is often repetitive in this way, FORTRAN supplies a 'DØ' statement to allow operations such as the above to be quickly coded as

```
      ISUM=0
      DØ  5 I=1,20
    5 ISUM=ISUM+I
```

The DØ statement specifies the last statement in the series of statements to be repeated (5), an *INTEGER* variable to act as the counter (I), and two *INTEGER* constants or variables to act as the initial and final values for which the loop is executed.

```
          DØ  2 J=N,M
```

will cause all statements up to and including label 2 to be repeated (M-N+1) times. It is necessary for N and M to have previously been assigned values, either as the left hand side of an arithmetic assignment, or through a READ command. FORTRAN requires that N ⩾ 1; while M ⩾ N. When a DØ loop is completed, the DØ variable (J in the above example) is regarded as being undefined.

It is at times necessary to nest one DØ loop inside another. Suppose we have 100 punched data cards with one number on each card, and that our aim is to find the average of each group of 10 and print out that average. The following is a complete program capable of doing this:

```
C      PRØGRAM BY J. SMITH
C      TØ READ 100 NUMBERS AND
C      FIND AND PRINT THE AVERAGE ØF EACH GRØUP ØF 10
       DØ  1 I=1,10
       SUM=0.
       DØ  2 J=1,10
       READ,X
     2 SUM=SUM+X
       AVG=SUM/10.
     1 PRINT,'AVERAGE FØR GRØUP ØF 10=',AVG
       STØP
       END
----------------------------------------------------
      18.51  }
      4235   }
    -6.7     }   data cards
     .       }
     .       }
     .       }
```

The loops function so that the inner counter will vary the most rapidly, *i.e.*

I	1	1	1	...	1	2	2	2	...	2	...	10	10
J	1	2	3	...	10	1	2	3	...	10	...	9	10

The last statement in a DØ loop can be any executable statement other than a transfer of control or another DØ statement. A dummy statement CØNTINUE, which does not perform any machine function, is provided as a way around this restriction:

```
         DØ 27 I=1,N
          .
          .
         IF(X.GT.27.35)GØ TØ 95
      27 CØNTINUE
          .
          .
      95 X=X+7.
          .
          .
```

6.11 STØP AND END STATEMENTS

The STØP and END statements serve two different purposes.

(i) The END statement provides an indication to the compiler that all the FORTRAN statements that precede it form a complete and separate program or subprogram in their own right.

(ii) The STØP statement is translated by the FORTRAN compiler as part of the machine program to be executed. When the program is executed and the STØP statement encountered, execution of it will cease and the computer will switch to the next waiting job.

6.12 ARRAYS OF VARIABLES

Many mathematical operations require the use of vectors and matrices. FORTRAN supplies a means of handling 1,2,3 or higher dimensional arrays. For the simplest array (the 1-dimensional vector), the ith element of the vector v (v_i) is represented in FORTRAN as V(I). Elements of an array or vector are capable of being used in FORTRAN in the same way that ordinary variables are employed;

```
      V(I)=0.          the ith element of V is set to zero
      A=V(I)+C(J)-D(3)
      V(I-1)=V(3*I-7)
```

The subscripts used to refer to vector or array elements must be *INTEGER* and greater than zero. They may be constants, variables or expressions. The FORTRAN compiler reserves one location (word) for non-subscripted variables to be stored. As subscripted variables take one location for each array element, it is necessary for the programmer to specify to the compiler the maximum number of elements associated with each array. This is done through a non-executable statement, the 'DIMENSIØN' statement that must precede the first use of the array it is defining:

```
      DIMENSIØN V(15)
      DØ  1 I=1,8
    1 V(2*I-1)=0.
```

The DIMENSIØN statement would tell the compiler that V is a vector (1-dimensional) array requiring 15 storage locations. The supplied statements would set all the odd components of V to zero. The next example demonstrates how a vector may be used to calculate the mean and standard deviation of a set of 10 numbers. These numbers are read from 10 cards (*i.e.* one number per card):

$$\bar{X} = \frac{\sum_{i=1}^{10} x_i}{10}$$

$$\text{Standard deviation} = \sqrt{\frac{\sum_{i=1}^{10} (X_i - \bar{X})^2}{9}}$$

```
C     J. SMITH CALCULATE MEAN AND STANDARD DEVIATIØN
C     ØF 10 NUMBERS
      DIMENSIØN X(10)
      DØ  1 I=1,10
    1 READ,X(I)
      SUM=0.
      DØ  2 I=1,10
    2 SUM=SUM+X(I)
      AVG=SUM/10.
      SUMSQ=0.
      DØ  3 I=1,10
    3 SUMSQ=SUMSQ+(X(I)-AVG)**2
      SDEV=SQRT(SUMSQ/9.)
      PRINT, 'MEAN AND STANDARD DEVIATIØN =', AVG,SDEV
      STØP
      END
```

Here we use the vector X to store ten numbers before finding the mean and standard deviation. Before employing vectors in a program, *make sure they are really necessary*. In a previous example (section 6.10), the mean of a set of numbers was required. There was no need in that case to retain the ten numbers because the sum accumulated when each number was read from a punched card. When the standard deviation is sought, the numbers must be retained at least up to the point where the mean is determined.

The next example demonstrates a program that computes the vector sum $\underset{\sim}{s}$ of two vectors $\underset{\sim}{u}$ and $\underset{\sim}{v}$:

$$\underset{\sim}{s} = \underset{\sim}{u} + \underset{\sim}{v} .$$

For $\underset{\sim}{u} = (3,5,2)$

and $\underset{\sim}{v} = (4,2,7)$

then $\underset{\sim}{s} = (3+4,\ 5+2,\ 2+7)$

$= (7,7,9)$

Mathematically we say that the i^{th} component of $\underset{\sim}{s}$ is formed as

$$s_i = u_i + v_i \qquad 1 \leqslant i \leqslant 3$$

The program will read the three pairs of data from separate punched cards as shown

$\underset{\sim}{u}$	$\underset{\sim}{v}$
3.	4.
5.	2.
2.	7.

into two vector arrays (U and V), compute the vector sum in S and print out each component of S on a separate line.

```
      DIMENSIØN S(3),U(3),V(3)
C     FIRST READ IN THE DATA
      DØ  1 I=1,3
    1 READ,U(I),V(I)
C     NØW FØRM THE VECTØR SUM
      DØ  2 I=1,3
    2 S(I)=U(I)+V(I)
```

```
C     WRITE ØUT HEADING AND RESULTS
      PRINT,'VECTØR S'
      DØ  3 I=1,3
    3 PRINT,S(I)
      STØP
      END
```

(In this example, it would have been possible to perform the vector addition operation without the use of subscripted variables - how? See appendix 6A for a solution. Such an operation, however, is frequently a small part of a much larger program where it is necessary to store the data in subscripted variables.)

When arrays of higher order than the 1-dimensional vector treated so far are needed, the DIMENSIØN statement informs the compiler of the number of dimensions (*i.e.* the number of subscripts) and the total storage for the array.

```
      DIMENSIØN A(5,5)
```

This informs the compiler that A is a matrix (2-dimensional array) requiring 25 locations for storage:

```
      DIMENSIØN A(5,5),B(5,5),C(5,5)
      .
      .
      .
      DØ  1 I=1,5
      DØ  1 J=1,5
    1 C(I,J)=A(I,J)+B(I,J)
      .
      .
      .
```

In this case, two matrices A and B are summed and the result is stored in a new matrix C.

6.13 SUBPROGRAMS

We have met (section 6.8) the special mathematical functions supplied through the FORTRAN compiler. The user is able to supply two types of subprograms of his own when necessary:

- FUNCTIØN subprogram.
- SUBRØUTINE subprogram.

Need for subprograms arises

(i) when the same mathematical function or procedure is required at many points in a program;

(ii) in larger programs, where it pays to write and test sections of the code independently; and

(iii) when more than one person is responsible for developing the code.

The FUNCTIØN subprogram returns a single value as its result and is usually used to perform mathematical operations similar to $\sqrt{\ }$, or function evaluation. The user supplied function is best demonstrated by an example. Suppose we wish to evaluate a cubic polynomial for various values of x:

$$f(x) = 1 + 1.5x + 3.2x^2 + 6x^3 ,$$

which for speed of computation is best written as

$$f(x) = 1 + x\ (1.5 + x\ (3.2 + 6x)) .$$

Then we might use the coding ...

```
    .                                 )
    .                                 |
    .                                 |
Y = F(X)+6.                           |
    .                                 |
    .                                 |
    .                                 |
Z = F(X-1.)                           >   Main or calling program
    .                                 |
    .                                 |
    .                                 |
STØP                                  |
END                                   )
FUNCTIØN F(A)                           )
F = 1.+A*(1.5+A*(3.2+6.*A))             > FUNCTIØN subprogram
RETURN                                  |
END                                     )
```

In the main program, the function is invoked by naming the function and enclosing in parentheses a constant, variable, or expression for which the cubic polynomial is to be evaluated. The FUNCTIØN subprogram is defined through the use of the 'FUNCTIØN' statement and an appropriate name 'F' (in this case) by which the function is to be known. An argument list corresponding to that in the main program is also required. The argument names in the function are only dummy ones and need not be the same as those in the main program (all the other variables and labels are local to the function and are in no way associated with labels or variables in the main program). When the above function is invoked twice by the main program, the values X and X-1. respectively are transferred into the location set aside for A. The function *must* return one value through the assignment of an arithmetic expression to the function name as in

```
F = 1.+A*(1.5+A*(3.2+6.*A))
```

The 'RETURN' is a transfer of control from the function back to the main program from where control was originally passed. The END statement is once again a signal to the compiler that this is the end of a logically independent set of FORTRAN statements.

The SUBRØUTINE subprogram is the more powerful version of a subprogram and usually performs more involved operations than those for which the FUNCTIØN is designed. Typical tasks for which subroutines are used would include finding the roots of equations, multiplication or inversion of matrices, and solving sets of linear equations. Unlike the function subprogram, the subroutine is not restricted to returning one result as part of an arithmetic expression. The subroutine and the main program communicate through the argument list only. The following code shows the use of a subroutine QUAD to determine real roots of a quadratic equation $ax^2 + bx + c = 0$. The coefficients of the equation to be solved are supplied as arguments to the subroutine, whereas the subroutine is responsible for returning the two roots and is an indication as to whether real roots were possible:

```
1 READ,C1,C2,C3
  CALL QUAD(C1,C2,C3,X1,X2,IER)
  IF(IER.EQ.0)PRINT, 'RØØTS ØF QUADRATIC ARE',X1,X2
  IF(IER.NE.0)PRINT, 'NØ REAL RØØTS EXIST'
  GØ TØ 1
  END
  SUBRØUTINE QUAD(A,B,C,R1,R2,K)
  DISC=B*B-4.*A*C
  IF(DISC.LT.0) GØ TØ 5
  DISC=SQRT(DISC)
  R1=(-B+DISC)/(2.*A)
  R2=(-B-DISC)/(2.*A)
  K=0
```

```
      RETURN
    5 K=1
      RETURN
      END
```

The subroutine is invoked, through a 'CALL' statement, by naming the subroutine and supplying a list of variables through which values are to be transferred to and from the subroutine. The main program passes the three coefficients of the quadratic while the subroutine will return the roots in X1 and X2 and an indication (K=1 or 0) to the sign of the discriminant. Once again the code within the subroutine is independent of the calling program.

6.14 <u>ERRORS IN PROGRAMMING</u>

The FORTRAN compiler will inform us in no uncertain terms of any syntactical errors we make in coding a program. Such errors are easy to detect and correct. The computer is a totally obedient servant; provided that we ask it to perform a task in the language it understands, it will obey us without question. Therefore, the hardest errors to identify are the ones we make in specifying the logic or steps involved in solving our problem. Reversing to British justice, all programs should be considered guilty (of containing bugs) until proven innocent ('debugged').

Too often the poor computer is blamed for an error in the program that should have been found and removed by the programmer when he was debugging his code.

garbage in   implies   garbage out

This adage is certainly true but the programmer and, in particular, the scientific programmer may find it difficult to recognise the output of a program for what it is. It is advisable to test programs thoroughly before placing any confidence in their output. This is often done by comparing the computed solution with a known mathematical or physical solution. When agreement is satisfactory we may then proceed to use our program for all the cases we are interested in.

There are three ways in which the problems of the scientific programmer are different to those of the more commercially oriented programmer. As most commercial tasks are well defined, errors in the computer output are directly due to the program or incorrect data on which it operated. The scientific problem solver is solving a mathematical model of some real physical system. When this model was developed, many assumptions (and probably simplifications) were made. Just how valid were these and are they the source of errors? Were the errors caused by the type of numerical technique chosen to solve the model? Or were the errors due to the coding of these techniques?

6.15 A RANDOM METHOD FOR DETERMINING π

If one had a circular dart board mounted on a square background, as shown in figure 6.4, then it would be possible to determine experimentally an approximation for π. When a dart is thrown randomly to land

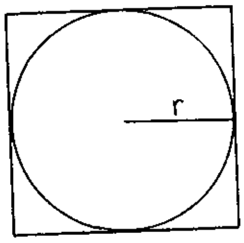


Figure 6.4 Dart board

in the square it may land within the circle or outside it. The probability of it landing within a certain section is proportional to the area of that section:

Relative proportion of darts landing in the circle

$$= \frac{\text{area of circle}}{\text{area of square}}$$

$$= \frac{\pi r^2}{(2r)^2}$$

$$= \frac{\pi}{4}$$

$$\therefore \pi = 4 \times \text{relative proportion of darts landing in the circle.}$$

Consequently, by randomly throwing darts and measuring the relative frequency of those falling within the circle, π can be determined directly. Instead of throwing darts, a computer can be employed to do

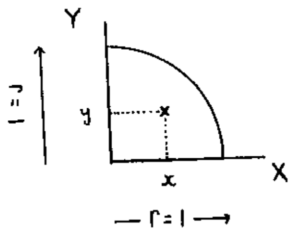


Figure 6.5 Quadrant simplification of dart board

this by direct simulation. The process can be simplified as shown in figure 6.5 by taking a quadrant of a circle of unit radius.
By selecting two random numbers, using our random number generator, we may let these two numbers (say x and y) represent the coordinates of the point where the dart lands. This point may be within the circle or outside it. If we measure the distance d of the point from the origin

$$d = \sqrt{x^2+y^2}$$

we can determine if it lies in the circle or not depending upon whether $d \leqslant 1$ or $d > 1$ (or, equivalently, whether $d^2 \leqslant 1$ or $d^2 > 1$).

A flow chart to describe the steps involved is shown in figure 6.6 for a sample of 1000 darts. A FORTRAN program sufficient to produce an approximation for π would be

```
      N=1000
      NDART=0
      NCIRC=0
    1 X=RND(X)
      Y=RND(X)
      D2=X*X+Y*Y
      IF(D2.GT.1.)GØ TØ 2
      NCIRC=NCIRC+1
    2 NDART=NDART+1
      IF(NDART.LT.N)GØ TØ 1
      PIE=4.*NCIRC/NDART
      PRINT,'APPRØXIMATIØN ØF PI = ',PIE
      STØP
      END
```

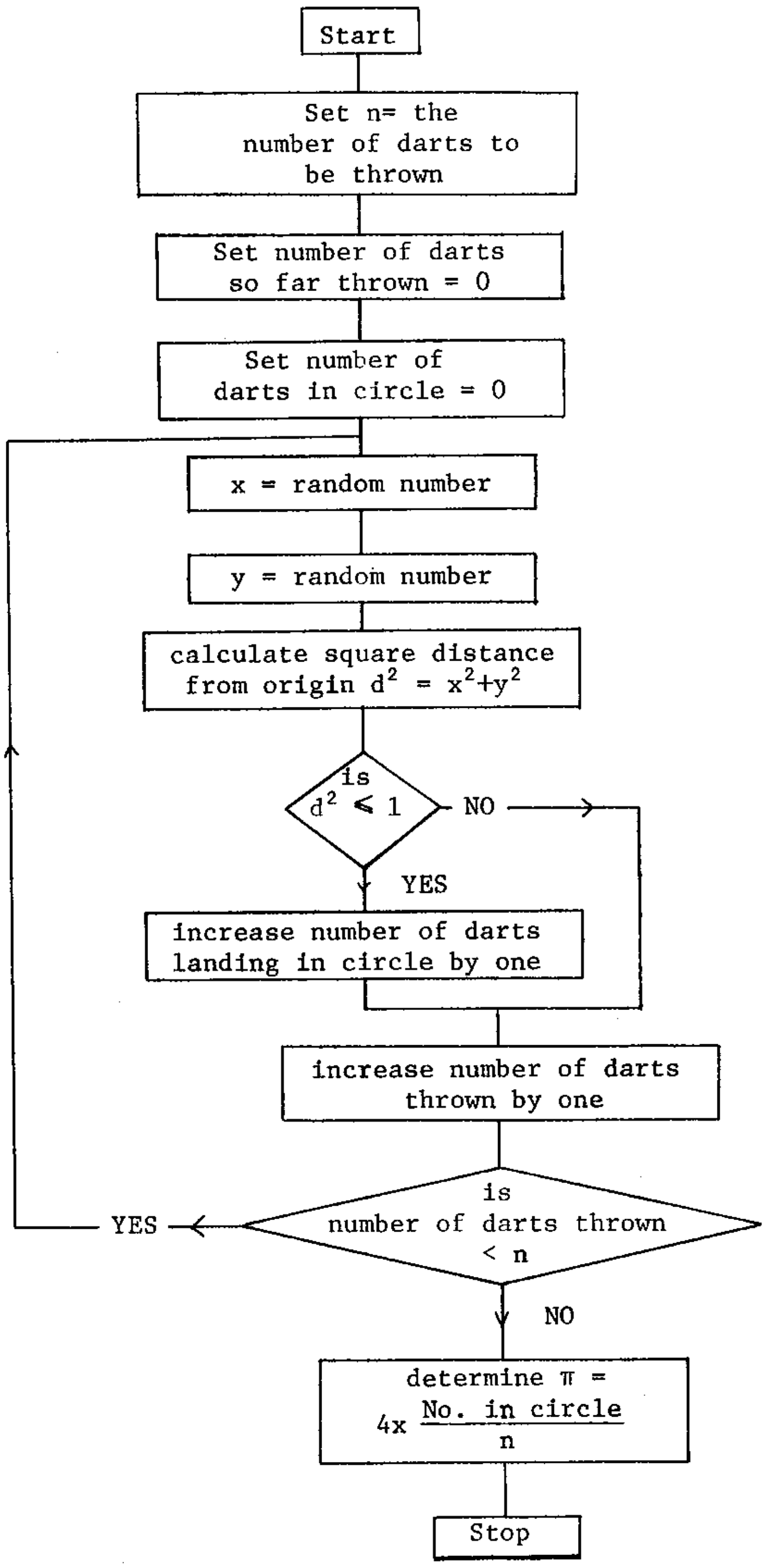

Start
Set n= the number of darts to be thrown
Set number of darts so far thrown = 0
Set number of darts in circle = 0
x = random number
y = random number
calculate square distance from origin $d^2 = x^2+y^2$
is $d^2 \leqslant 1$
NO
YES
increase number of darts landing in circle by one
increase number of darts thrown by one
is number of darts thrown < n
YES
NO
determine $\pi = 4x \frac{\text{No. in circle}}{n}$
Stop

Figure 6.6 Flow chart for π problem

6.16 PRACTICE EXAMPLES

Before you attempt to code the simulation problem described in chapter 3, try these practice examples. The answers to the questions are given in section 6.17, but don't be too hasty to seek these out until you have had a go yourself.

Q1. (a) In the list below, which items are variables or constants?

(b) What is the mode (integer or real) 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) BØS, (12) A*B, (13) 5,312.6, (14) BLØT

Q2. Write each of the following algebraic formulae as a FORTRAN statement to calculate y. Use any convenient real names for the variables, which will be assumed to have been assigned values by previous steps of the program.

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

(2) $y = \frac{\sqrt{b^2-4ac}}{2a}$

(3) $y-x = a-\pi y$ (π=3.141592)

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

(4) I=A

(5) I=A/2.

(6) U=A/2.

Q3. Write the necessary statements of portion of a program 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 program and that the values do not require special consideration in calculating the expressions:

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

(2) $y = e^x$

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

(4) $v = \tan x$

(5) $c = \ell n \left|\dfrac{1}{1+a^3}\right|$

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

Q4. Write a FORTRAN statement that will assign a random number to a variable named y, such that $0 < y < 1$.

Q5. Write a FORTRAN statement that will assign a random number to y, such that $0 < y < 10$.

Q6. Write a FORTRAN statement that will assign a random number to y, such that $-5 < y < 5$.

Q7. Write a FORTRAN program to print out ten random numbers such that each number satisfies the conditions of Q5.

Q8. Write a FORTRAN statement that uses the FORTRAN random number generator and will produce a random integer from the set (0,1, 2,...,10).

Q9. Bill Smith travels to work 5 days each week by bus. Bill is an extremely methodical person who arrives at the bus stop at precisely 8.00 a.m. The government bus service is also extremely punctual and its vehicles call at Bill's stop at 8.01, 8.06 and 8.11 a.m. Each of these is capable of getting Bill to work on time. His employer although somewhat flexible and tolerant will dismiss him should he arrive at work late more often than once a month (one month = four working weeks) averaged over the period of employment. Can Bill reasonably expect to retain his job in the long term if the number of passengers each bus can pick up at his stop varies randomly between 0 and 9, and if he could find any random number of people up to 10 in front of him in the queue? Ignore any appeal to probability theory and write a FORTRAN program using the random number generator RND to simulate the bus stop situation and help estimate Bill's employment security with his present firm. Run the daily simulation for 1000 such mornings and print out the number of days per month Bill is late. Before you write any FORTRAN code, you might like to draw a flow chart to make sure you clearly understand the steps involved.

Q10. Write a program that will

(1) Read the four coefficients of a cubic polynomial $f(x)=a+bx+cx^2+dx^3$ from a punched card.

(2) Read the estimate x_0 of a root of the equation $f(x)=0$ from a second card.

(3) Improve the estimate of the root by the Newton-Raphson method

i.e. $$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

(4) The process can be considered to have converged if

$$\frac{|x_{n+1} - x_n|}{|x_n|} < 0.001$$

(5) Print out the improved estimate of the root.

(6) Allow only five iterations. If convergence has not been achieved, print a message warning of this.

(7) Repeat from (1).

Q11. (For advanced students only)

Read in a set of ten numbers punched one per card. Write a code that will sort these numbers in descending order.

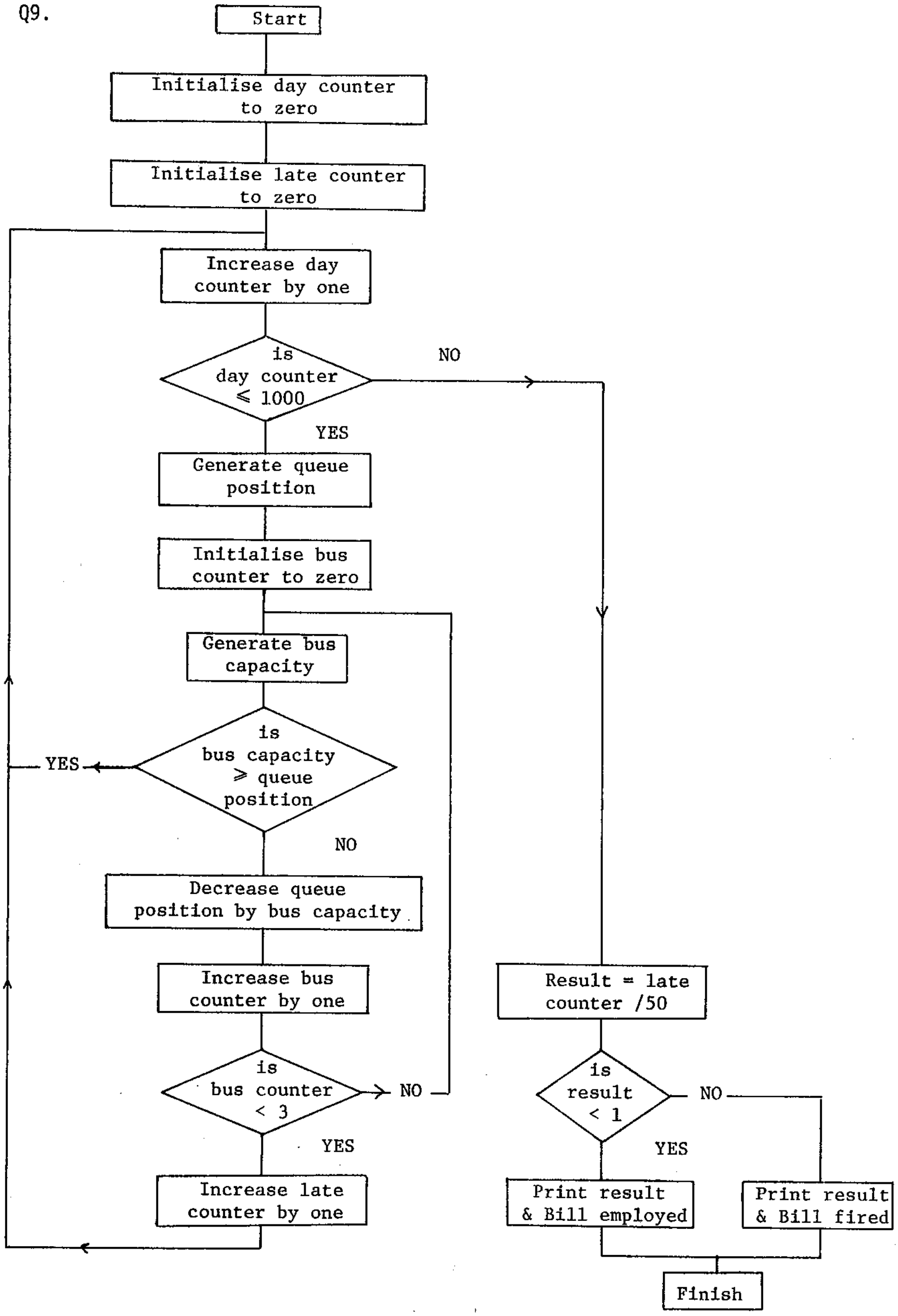
6.17 ANSWERS AND TYPICAL CODING

Q1. (1) real constant, (2) real variable, (3) integer variable, (4) integer constant, (5) real constant, (6) invalid variable name as more than 6 characters, (7) real variable, (8) invalid variable name as first character is not alphabetic, (9) invalid variable name as e is a lower case letter, (10) integer constant, (11) real variable, (12) invalid since an expression is not a variable, (13) invalid as comma is not permitted, (14) real variable.

Q2.
```
(1)     Y = 0.5*(B+C)
(2)     Y = SQRT(B*B-4.*A*C)/(2.*A)
(3)     Y = (X+A)/4.141592
(4)     I = 3
(5)     I = 1
(6)     U = 1.5
```

```
Q3.  (1)        S = SQRT(X*X+Y*Y+Z*Z)
     (2)        Y=EXP(X)
     (3)        W1 = EXP(X)
                W2 = 1./W1
                U = (W1-W2)/(W1+W2)
     (4)        V = TAN(X)
     (5)        C = -ALØG(ABS(1.+A*A*A))
     (6)        W1 = A*X
                Y = (EXP(W1)+EXP(-SQRT(W1)))*0.3333333

Q4.        Y = RND(X)

Q5.        Y = RND(X)*10.

Q6.        Y = 5.-RND(X)*10.

Q7.        DØ 1 I=1,10
           Y=RND(X)*10.
         1 PRINT,Y

Q8.        I=RND(X)*11.
```

Q9.


Start
Initialise day counter to zero
Initialise late counter to zero
Increase day counter by one
is day counter ⩽ 1000
NO
YES
Generate queue position
Initialise bus counter to zero
Generate bus capacity
is bus capacity ⩾ queue position
YES
NO
Decrease queue position by bus capacity
Increase bus counter by one
is bus counter < 3
NO
YES
Increase late counter by one
Result = late counter /50
is result < 1
NO
YES
Print result & Bill employed
Print result & Bill fired
Finish

```
      NDAY=0
      NLATE=0
    1 NDAY=NDAY+1
      IF(NDAY.GT.1000)GØ TØ 1000
      NQ=RND(X)*23+1
      NBUS=0
    2 NBSCAP=RND(X)*10.
      IF(NBSCAP.GE.NQ)GØ TØ 1
      NQ=NQ-NBSCAP
      NBUS=NBUS+1
      IF(NBUS.LT.3)GØ TØ 2
      NLATE=NLATE+1
      GØ TØ 1
 1000 DLATE=NLATE/50.
      IF(DLATE.LT.1.)GØ TØ 3
      PRINT,'BILL FIRED',DLATE
      GØ TØ 4
    3 PRINT,'BILL EMPLØYED',DLATE
    4 STØP
      END
```

Q10.

```
C       J. SMITH
C       RØØT ØF CUBIC EQUATIØN
C       READ CØEFFICIENTS FRØM ØNE PUNCHED CARD
      9 READ,A,B,C,D
C       READ IN ESTIMATE FØR RØØT FROM NEXT CARD
        READ,XN
C       LØØP TØ IMPRØVE ESTIMATE
        DØ  1 I=1,5
        XNP1=XN-(A+XN*(B+XN*(C+D*XN)))/(B+XN*(2.*C+XN*3.*D))
C       NØW ASK IS PRØCESS CØNVERGING
        IF(ABS((XNP1-XN)/XN).LT..001)GØ TØ 2
      1 XN=XNP1
C       RØØT HAS NØT BEEN FØUND
        PRINT,'RØØT NØT FØUND WITHIN 5 ITERATIØNS'
C       TRY ANØTHER SET ØF CØEFFICIENTS
        GØ TØ 9
C       RØØT LØCATED WITHIN PRESCRIBED BØUNDS
      2 PRINT,'RØØT ØF CUBIC=',XNP1
C       TRY ANØTHER SET ØF CØEFFICIENTS
        GØ TØ 9
        END
```

Sample data cards:

```
        5.6     3.7       2.       -46.
        1.2
```

Q11.

```
C       J. SMITH
C       SØRTING PRØBLEM
C       NØTE THIS IS THE SIMPLEST TØ CØDE - BUT NØT THE FASTEST
        DIMENSIØN X(10)
C       SET UP LØØP TØ READ 10 NUMBERS
        DØ  1 I=1,10
      1 READ,X(I)
        DØ  2 I=1,9
        N=I+1
        DØ  2 J=N,10
        IF(X(J).LE.X(I))GØ TØ 2
        TEMP=X(J)
        X(J)=X(I)
        X(I)=TEMP
      2 CØNTINUE
         .
         .
         .
```

APPENDIX 6A

SOLUTION TO THE VECTOR SUMMATION PROBLEM OF SECTION 6.12

```
C      VECTØR SUM PRØBLEM.
C      THIS TIME THE PRØBLEM IS SØLVED WITHØUT USING
C      SUBSCRIPTED VARIABLES IN ØRDER TØ DEMØNSTRATE THE CAUTIØN
C      THAT SHØULD BE EMPLØYED BEFØRE INTRØDUCING THEM UNNECESSARILY.
       PRINT,'VECTØR S'
       DØ 1 I=1,3
       READ,U,V
       S=U+V
     1 PRINT,S
       STØP
       END
```

APPENDIX 6B
FORMAT CONTROL OF I/O OPERATIONS

Far greater control of your input and output operations may be obtained by using FØRMAT-controlled READ and WRITE operations. Because these take a considerable time to master, most students are advised to stay with the simpler forms of I/O control treated earlier in the Summer School. This appendix is included to give a limited idea of what is possible. The general form of the I/O statement is

```
READ(n,m)  list of variables
WRITE(n,m) list of variables,
```

where n is an integer representing the device type for which the input/output operation is to occur. At the AAEC Research Establishment

n=1 for the card reader

n=2 for the card punch

n=3 for the printer,

m is an integer constant and represents the statement number of a FØRMAT statement. The FØRMAT statement is used to edit the transfer of data. A few examples will demonstrate the ideas involved.

(1)
```
      READ(1,127)N,X
  127 FØRMAT(I5,F10.2)
```

This will read one punched card and obtain from it two numbers. The first number must be of integer form and may occupy the first five columns of the card. Numbers with less than five digits should be punched with leading blanks so that the number concludes in column 5. The second number may span columns six to fifteen and would normally have a decimal point punched. (If you fail to punch the decimal point the computer gives you one in the default position specified by the F10.2 FØRMAT code, *viz.* two digits from the right hand side.)

(2)
```
      X=527.1392
      I=96
      WRITE(3,100)I,X
  100 FØRMAT(I5,F12.5)
```

This will produce the output

```
   96   527.13920
```

The following FØRMAT statements would have produced the results indicated.

```
100 FØRMAT(I3,5X,F9.4)                 96       527.1392
100 FØRMAT(I1,5X,F5.1)                *      527.1
100 FØRMAT(I5,5X,F6.2)                   96      527.14
100 FØRMAT(I5,5X,E11.4)                  96       0.5271E+03
105 FØRMAT(' I=',I3,' X=',F6.2)       I= 96 X=527.14
```

CHAPTER 7

BASIC FOR MINIS AND MICROS

Lecture by

J.P. POLLARD

CONTENTS

7.1 INTRODUCTION

At the present time, the possible use of microcomputers in the high school has aroused much interest. Some schools already have such a machine; others are contemplating getting them. Well, you may ask, what is a microcomputer? How does it differ from a large scientific computer such as the IBM360/65 to be used at this Summer School? The real answer to these questions will come when we see one for ourselves. Just now, we will be content to note that the most important feature is probably the price. A reasonable machine can be obtained for about $1000 to $2000. Almost all microcomputers available accept the BASIC language – some nothing else – hence our interest in that language at this Summer School.

BASIC was originally developed for beginners at programming by Dartmouth College, New Hampshire, USA in 1964. Since then, many different versions ('dialects' of BASIC) have arisen. If we were to restrict ourselves to the subset common to them all, we might find that the only type of statement available would be

```
        10 REM  THIS IS IT
```

- a remark made to help us find things in a program when listed but otherwise ignored by the machine.

Here, we will meet a useful subset of so-called BASIC+, that will at least run on a PDP11/45 minicomputer connected to various terminals at Lucas Heights and which will also run on some microcomputers.

Unlike FORTRAN, BASIC is an interactive language (usually) hence the main output of data is likely to come from a terminal (monitor, teletype or whatever) rather than punched cards. We may thus make up our minds as we go. Depending on what happens, we may readily change the program and data to meet contingencies not originally envisaged.

7.2 GETTING THROUGH

Naturally, the computer system will not want to speak to us unless we are bonafide. We must therefore find a terminal and sign on in an acceptable way. During the week of the Summer School, we are given two 'passwords':

1. initials: SSK (for Summer School Kids), and
2. ID: AM290060

which will get us through to BASIC. In the following stinted dialogue between us and a computer (not actually the PDP11/45) the underlined

items are *machine responses* and ↓ denotes the carriage RETURN or ENTER key.
Here we go...

(1) Push (not hit) space bar

(2) ID: AM290060↓ (this won't print in case someone else is watching)

(3) $2 ↓

(4) login:SSK↓ (upper or lower case letters here select the mode for the run)

(5) % BASICP ↓

...

(6) ready *(A MICROCOMPUTER WILL PROBABLY START HERE)*

```
our      ⎧ 10 REM THIS IS WHERE BASIC GOES - AT LAST!
BASIC    ⎨ ...
program  ⎩ ...
```

7.3 LET'S RUN

Here is a simple program to prepare a table of 21 values of $y = e^x$ for x = -10,-9,...,0,...,9,10. A (WATFIV) FORTRAN version is shown for comparison.

BASIC	FORTRAN
┌ line numbers determine normal program 'flow'	┌statement numbers not necessary every line
10 REM PROG. TO TAB. EXP(X)	C PROG. TO TAB. EXP(X)
20 X=-10.	X=-10.
30 FOR I=1 TO 21	DO 70 I=1,21
40 Y=EXP(X)	Y=EXP(X)
50 PRINT X,Y	PRINT,X,Y
60 X=X+1.	X=X+1.
70 NEXT I	70 CONTINUE
80 STOP	STOP
90 END	END
└ normally initially 10 apart for later insertions	
We type in the prog. (in any order of line nos.)	We punch the prog.
We type LIST to check statements.	We list the cards to check them.
We type RUN to start prog.	We submit to run as a batch job.

We note the close similarity to FORTRAN (or at least its WATFIV dialect) and the subtle differences (no comma after PRINT for example). The FOR-NEXT loop is similar to the DO-CONTINUE (or whatever terminating statement) loop. A counter (any variable in BASIC, not just I,J,...,N type fixed point variables of FORTRAN) is incremented, usually in steps of 1, until the terminating value is reached. We may however directly specify negative values. For example, we may modify our BASIC program thus:

delete statement 20 by typing

```
     20  and nothing else,
```

change statement 30 by entering

```
     30 FOR X=-10 TO 10
```

delete statement 60 with

```
     60
```

and *change* 70 to

```
     70 NEXT X
```

The typed command LIST would then give

```
     10   REM PROG. TO TAB. EXP(X)
     30   FOR X=-10 TO 10
     40   Y=EXP(X)
     50   PRINT X,Y
     70   NEXT X
     80   STOP
     90   END
```

With a slight modification to the program, we may start with the highest value...

change statement 30 by typing

```
     30   FOR X=10 TO -10 STEP-1
```

As for FORTRAN, the loop limits may actually be specified by variables rather than constants. Some BASICs permit non-integer values (like STEP 0.5), and indeed our BASIC+ does this whereas others just don't; they work with the integers equal to or just below the numbers.

Like FORTRAN, sensibly nested loops are permitted, *e.g.*

```
     100   FOR I=1 TO 10
     110   FOR J=1 TO I
      :    (coding which does not change the loop variables - I
      :     and J here)
```

```
200  NEXT J
210  NEXT I
```

Note - the biggest line number permitted is usually 32767.

7.4 WHAT'S IN A NAME?

Normally BASIC has only two types of variables:

(1) Numeric variables with names

A,B,C,...,Z and

A0,A1,A2,...A9; B0,B1,B2,...B9; ...Z0,Z1,Z2,...,Z9

- the numeric variable I7 could contain one of the numbers; *e.g.*

0, 27, -2183, 1.63, 3.141592, 7.65E+19, 1.139261E-2

(in fact any positive or negative number between about 1.E-38 to 1.E38, or 0).

(2) String variables, with names

A$, B$, C$,...,Z$ and

A0$,...,Z9$

- the string variable A$ could contain up to a line of text characters, *e.g.* "JOHN", "YES", "NO(S) LEFT", "**MARY'S GO**"

Note - text is enclosed between double quote marks.

Assignment of variables is of the style

```
A1=12.5
C3=A1
A1$="*"
B$=A1$
```

(and some older style BASICs want you to let them know that the statements are assignments thus,

```
LET  A1=12.5
LET  C3=A1,
```
etc.

but please forget you were ever told about this).

Most versions of BASIC permit indexed arrays, *e.g.* A(112), but we will not be concerned with such things here.

7.5 OPERATIONS TO REMEMBER

The BASIC expression

```
Y=(1+X*X)/(1-X↑N)
```

calculates

$$y=(1+x^2)/(1-x^n)$$

and differs only from the FORTRAN counterpart in the use of ↑ (or ^ for some keyboards) for exponentiation (raising to a power). The example illustrates all you need to remember.

7.6 GO TO AND OTHER DEVIATIONS

The regular flow of BASIC when running is from the smallest line number to the next in order, no matter in which order the statements are typed. We have seen a FOR-NEXT combination to establish a loop back and forth. Here are other statements that cause deviations to the regular flow:

(1) GOTO line number (alternatively GO TO line number)
e.g. GOTO 10

(2) $$\text{IF } \&_1 \left\{\begin{array}{c} = \\ > \\ < \\ >= \\ <= \\ <> \end{array}\right\} \&_2 \quad \text{THEN} \quad \text{statement} \left[\begin{array}{l} \text{equals} \\ \text{greater than} \\ \text{less than} \\ \text{greater than or equal} \\ \text{less than or equal} \\ \text{not equal} \end{array}\right.$$

where $\&_1$ and $\&_2$ are arbitrary *expressions*
and *statement* is almost any BASIC statement, *e.g.*

```
IF A<B-1   THEN Z=1
```

except that (for some BASICs) another IF cannot follow and *only the line number* must be given *for a GOTO*. Thus

```
IF X1>0  THEN  200
```

will transfer control to line 200 if X1 is positive. (Some newer style BASICs do not insist on the THEN as part of the IF - but we will.)

(3) ON & GOTO $n_1, n_2, n_3, \ldots$
where & is an expression (*e.g.* I, B+1, *etc.*)
and if $1 \leqslant \& < 2$ control is transferred to line number n_1,
$2 \leqslant \& < 3$ control is transferred to line number n_2,
$3 \leqslant \& < 4$ control is transferred to line number n_3, *etc.*

In the BASIC program that we will use, if & cannot find an appropriate n, (*e.g.* if & = 0), then an error condition arises.

(4) STOP
- does just that.

(5) END
- similar to STOP but some BASICs can only tolerate one of these and then normally only as the final statement. Being FORTRAN users, we will be happy to finish a BASIC program with END.

7.7 BUILT-INS

Like FORTRAN, BASIC has a suite of built-in functions. These are as follows:

(1) $\text{ABS(X)}=\begin{cases}+X & \text{if } X\geqslant 0\\ -X & \text{if } X<0\end{cases}$ *e.g.* ABS(-2.7)=2.7

(2) INT(X)= integer, equal to, or just smaller than X, *e.g.* INT(3.14)=3 *but* INT(-3.14)=-4

(3) $\text{SQR(X)}=\begin{cases}\sqrt{X} & \text{if } X\geqslant 0\\ \text{error otherwise}\end{cases}$

(4) EXP(X)= e^X

(5) $\text{LOG(X)}=\begin{cases}\ell n\ X \text{ if } X>0\\ \text{error otherwise}\end{cases}$

(6) SIN(X)= sin x - note that x must be in radians (1 radian = 180/π degrees)

(7) COS(X)= cos x

(8) TAN(X)= tan x

(9) ATN(X)= $\tan^{-1}X$ (*i.e.* arctan X)

(10) RND(X)= a pseudo-random number, bigger than 0 and smaller than 1

(an argument is to be supplied but is ignored)

(11) RANDOMIZE with no argument

- makes sure that successive runs of the same program give different random numbers for the first use of RND(X) - otherwise a chess playing program would always open 'pawn to queen 4'.

7.8 LET'S PUT IT TOGETHER

We now put together a simple BASIC program to calculate the highest common factor (HCF) of two positive integers M and N using Euclid's algorithm (method):

```
10  REM HCF PROG
20  M=85
30  N=204
40  GOTO 80
50  PRINT H
60  STOP
70  REM
80  REM START OF EUCLID ROUTINE
```

```
 90  H=N
100  N=M-N*INT(M/N)
110  M=H
120  IF N<>0 THEN 80
130  GOTO 50
140  END
```

Follow the program flow yourself to see what is happening. Our variables change thus:

	initially	1st pass	2nd pass	3rd pass	4th pass	
H	-	204	85	34	17	← answer
N	204	85	34	17	0	← finish ∴
M	85	204	85	34	17	

Surely we don't have to keep changing lines 20 and 30 if we want to calculate the HCF of other numbers? No!

7.9 I/O U

We have seen how to output a number on the terminal with, for example,

```
PRINT U
```

Surely to input a number from a terminal we would use

```
INPUT U
```

- yes!

The ambitious user of BASIC might also want to input a string of characters (perhaps the name of the person running the program). We would simply use, for example,

```
INPUT A$
```

and later we would output with

```
PRINT A$
```

Our HCF program may be modified to input M and N thus:

```
20  INPUT M,N    (and at the appropriate RUN stage we enter,
                 for example, 85,204)

30      and nothing else (to delete the line)

60  GOTO 20      (to always return to input two more values
                 for M and N)
```

The input described above is for interactive use where we may change our minds depending on the printed results achieved so far. (In a sense we are part of a loop with the computer.) However, say we only want to enter occasionally varying data such as the constants μ, a, b, c of our Summer School project. We enter such (essentially fixed) data with a statement consisting of a line number followed by the control word DATA, *e.g.*

```
1000 DATA 0.577,0.184,0.494,1.172
```

which may appear anywhere in the program (although being FORTRAN users we would probably put it towards the end, *i.e.* give it a high line number). As with FORTRAN, the data are not entered into the computation until a READ is issued, *e.g.*

```
READ U
```

which will read the next word of data from a DATA statement. If previous READs have exhausted all items of one DATA statement, then further input will come from the next DATA statement in the program (if one exists). Thus we could equally well have used for our example

```
1000  REM MU
1010  DATA 0.577
1020  REM A,B,C
1030  DATA 0.184,0.494,1.172
```

which we could enter into the program with, for example, the statement

```
READ U,A,B,C
```

When we RUN a job, the READ *pointer* begins from the first word of the first DATA statement and then every successive READ moves the pointer down the DATA list. We cannot *jump over* unwanted data (except with *dummy* READ statements); however, we can always restore the pointer to the beginning with the statement

```
RESTORE
```

7.10 ROUTINE MATTERS

When a portion of BASIC code forms a routine to do a specific job we would like to be able to jump to the routine and return to the line that follows (*cf.* a FORTRAN subroutine). The need becomes more apparent when different parts of the main coding require *calls* to the one routine, for then the return is to be made to different parts of the program. We might as well modify our HCF program to include a subroutine for the Euclid algorithm. We type

```
40  GOSUB 80
130 RETURN
```

Get the idea?

7.11 PRETTY PRINT

If we run our HCF program we find it is a bit terse, for it says

```
?
```

when we are to supply input. In BASIC (and WATFIV, FORTRAN) we can easily arrange to print descriptive information (without the labour entailed by string variables). We simply enclose the information between double quotes (single quotes for WATFIV) as part (or all) of the PRINT statement. Our HCF program could thus be further modified:

```
15  PRINT "ENTER VALUES FOR M AND N"
50  PRINT "THE HCF IS";H
60  GOTO 15
```

No, the typist did this correctly - a semicolon implies a short skip after the preceding number or characters, whereas a comma implies a long skip so that precise columns of printout are maintained. Try it for yourself! You might even be bold enough to try

```
15  PRINT "ENTER VALUES FOR M AND N";
```

A feature for establishing printed output at specific print positions is the TAB(X) function. The printed columns are usually numbered 0,1,2,...,71 and TAB(X) for X=3.141592, for example, would set the subsequent print to begin at column number 3 (the INT part of X). We might then decide to place our result for the HCF of two numbers in the middle of the line thus:

```
50  PRINT TAB(25);"THE HCF IS";H
```

With a little pondering you might also see how rough graphical output, say for the function Y=EXP(X), may be obtained using

```
PRINT TAB(Y);"*"
```

We are hardly likely to remember all the changes that have been made to our HCF program. Let us do another

```
LIST

 10  REM HCF PROG
 15  PRINT "ENTER VALUES FOR M AND N";
 20  INPUT M,N
 40  GOSUB 80
 50  PRINT TAB(25);"THE HCF IS";H
 60  GOTO 15
 70  REM
 80  REM START OF EUCLID ROUTINE
 90  H=N
```

```
100  N=M-N*INT(M/N)
110  M=H
120  IF N<>0 THEN 80
130  RETURN
140  END
```

7.12 HAVE A BREAK

Say that in our HCF program we had mistakenly typed line 100 as

```
100  N=M
```

and we had requested the job to run

```
RUN
```

We would gain the (wrong) impression that the machine was slow for we would have no more response out of it. But no - we are running a program that is caught up in a never ending loop. How can we (or rather the program) escape? Simply this - we just depress the two keys

$\boxed{\text{CTRL}}$ $\boxed{\text{Z}}$ (actually we hold the first key down and then depress the second).

Thank goodness! Imagine the computer bill if we had had no way to break out of that loop!

A similar feature holds for interrupting a LIST:

$\boxed{\text{CTRL}}$ $\boxed{\text{S}}$ gives a temporary interruption to the LIST and

$\boxed{\text{CTRL}}$ $\boxed{\text{Q}}$ resumes the LIST.

We may, however, select to LIST only a portion of our program from the outset. For example, if we only want to list the EUCLID ROUTINE of our HCF program, we would use

```
LIST  80-130
```

7.13 SAVE IT

Having gone to a lot of effort to type in a program, we would obviously like to save it for later use. For the minicomputer setup used at this Summer School, the program would be saved on disk, whereas for a microcomputer setup the program would be saved on an ordinary audio-cassette. Here we go:

```
SAVE  int.HCFJOB
```

where int are *your* 3 initials used throughout the School (not SSK) and HCFJOB (or the like - a name having up to 8 characters) denotes the job. Thus I might use

```
SAVE  JPP.TESTPRGM
```

Should the job already have been saved you would need to say, for example,

```
          REPLACE   JPP.TESTPRGM
```

7.14 ANY OLD JOBS

Of course saving a job would not be of much use if we could not reload it (usually at some other time). The command is simply this, for example,

```
          OLD   JPP.TESTPRGM
```

which we would presumably then want to

```
          RUN
```

or we could combine the two operations into one

```
          RUN   JPP.TESTPRGM
```

7.15 SIGNING OFF

When we have finished a session (<u>and perhaps SAVEd our job</u>), we will terminate with the following signing off process:

(1) BYE - end of connection to BASIC+

(2) $\boxed{\text{CTRL}}$ $\boxed{\text{D}}$ - end of connection to PDP11/45

(3) $\boxed{\text{CTRL}}$ $\boxed{\text{P}}$ - end of connection to the computer system.

7.16 WHAT COULD BE MORE BASIC?

Besides other BASIC statements which will not concern us (for it is here that we encounter most differences from microcomputers) we can use the machine as a calculator. If we leave out a line number, then the instruction is carried out immediately. As a simple example we could calculate $3e^2$ using the coding

```
          PRINT 3*EXP(2)
```

which would return the result immediately the carriage return was depressed. Similarly, the line

```
          X=3*EXP(2):PRINT X
```

would achieve an equivalent result (except that, in addition, the answer would be stored in X) where ':' separates different statements on the one line. (\ is an alternative statement separator for some versions of BASIC.)

7.17 TRY THIS FOR YOURSELF

During the Summer School, you will be given the opportunity to 'have a go' for yourself. Three jobs are detailed here - everyone should try the first one; a few might get through to the third one.

(1) Write a BASIC program to display pseudo-random numbers. Print 10 such numbers and stop. How could you get a different set of 10 numbers for different runs?

(2) Write a BASIC program to simulate the roll of a dice 1000 times. Report the number of throws that achieve a 1. Use the RANDOMIZE statement and run the job 3 times to get some *feel* for the dispersion of the results about the expected mean of 167.
Note that this is almost equivalent to the problem of estimating the number of source neutrons of our Summer School project that are absorbed on their first collision when many iron plates are present. Why?

(3) If you have finished the Summer School project, write a BASIC program for the game MRMIND described in Appendix 7A.

Note Appendix 7B gives a list of BASIC statements that may help you to get started.

APPENDIX 7A
FOR THOSE WHO HAVE FINISHED THE SUMMER SCHOOL PROJECT: MONTE CARLO SIMULATION

7A.1 INTRODUCTION

The study of (1) neutrons moving through absorbing material; (2) queuing problems of people at a supermarket exit; (3) preparation of high school timetables; and (4) fun and games can be assisted by simulation of the processes. The simulation consists of studying hosts of different possibilities pseudo-randomly chosen on a computer. Since you will undoubtedly find some games to play at a computer terminal, the idea here is to give you some idea of *what happens after you hit carriage-return at the terminal.*

A basic requirement of Monte Carlo simulation is to generate pseudo-random numbers r_1, r_2, r_3, ... from a given starting value to r_0. Since we can write down the outcome, the sequence is reproducible (provided that we stay with the one computer) but, to all intents and purposes, we can consider the set of numbers $\{r_0, r_1, r_2, \ldots, r_n\}$ to be random and uniformly distributed throughout the interval (0,1). With BASIC we obtain a random number in the following way:

```
RANDOMIZE    -  initial statement at beginning of the program
  :             to yield a 'random' starting number.
R=RND(0)     -  later on, then R is a random number (and,
                since we began with RANDOMIZE, the BASIC
                system returns to us its next random number
                rather than the first of a reproducible
                sequence).
```

Then I=INT(10*R)

is a random integer (digit) 0,1,2,3,4,5,6,7,8 or 9 since INT takes the integer part of 10 times R. Our next use of the RND function will then return a different number, and so on. After, say, 1000 calculations of I we might produce the results.

I	No. of times
0	103
1	104
2	98
3	103
4	93
5	100
6	98
7	92
8	101
9	108

7A.2 ON WITH THE JOB

Use these ideas to write a BASIC version of

MISTER MIND

(Whereas *master mind* uses assorted colours, *mister mind* uses integer digits 0 to 9. The game is an adaptation of BAGLES taken from the DEC manual, 101 BASIC COMPUTER GAMES, 1975, p.22.)

In brief, the computer is required to obtain 3 random integer digits (same as 'I' earlier) imagined to be the leading, middle and trailing digits of a '3 digit number' between 000 and 999. A player is permitted to have 20 goes at attempting to guess the digits. After every go, the program is to return a clue in the form of two numbers

D = the number of correctly positioned digits, and

M = the number of matches of the player's digits with the machine's digits, including any already counted in D.

A simple program would input player's digits one at a time. A more complicated program would input a three-digit number and internally strip off the digits.

As an example, say the sought after number is 346 and the opponent's number is 640; then we print

D=1 (*i.e.* 1 correct digit, the 4)

and M=2 (*i.e.* 2 matched digits, the 6 and the 4).

Or if the sought after number is 355 and the opponent's number is 505 then we have

D=1 (*i.e.* the last 5)

and M=4 (*i.e.* the first 5 matches the last two and the last 5 does the same).

APPENDIX 7B

A LIST OF BASIC STATEMENTS VIA AN EXAMPLE: SOLUTION OF QUADRATIC EQUATIONS

```
ID:     AM290060
$2                                                                          see chapter

                                                                            }getting through
login: ssk                                                                  }
ssk                         Note - depending on the mode of reply           }7.2
                            of the password SSK (upper or lower
% basicp                    case) so that mode will be maintained
                            for the run.  However most BASICs work
UNSW Basic+                 with capitals only.

Ready

old jpp.quad                                                                old 7.4

Ready

list                                                                        list 7.3
jpp22-Nov-7807:08 AM
10 rem soln of quadratic eqns                                               rem 7.1
20 print "soln of quadratic eqns"                                           print 7.3,7.9,7.11
30 read n                                                                   read 7.9
40 print "first";n;" examples"                                              ;with print 7.11
50 for i=1 to n                                                             for 7.3
60 read a,b,c
70 gosub 900                                                                gosub 7.10
80 next i                                                                   next 7.3
90 rem over to user
100 print  : print "please suppy name";                                     : mult.stmts.per line 7.16
110 input a$                                                                input 7.9 & string
120 print "now your turn ";a$                                               variables($) 7.4
130 print
140 print a$;" please enter coeffs in a*x*x+b*x+c=0"
150 print "a,b,c=";
160 input a,b,c
170 gosub 900
180 go to 130                                                               goto 7.6
900 rem print problem then solve
910 print "soln of";a;"*x*x+";b;"*x+";c;"=0 is..."
920 gosub 1000
930 return
1000 rem routine for extracting roots (real or complex)
1010 if a=0 then print "that's not a quadratic" : stop                      if 7.6,stop 7.6
1020 d=b*b-4*a*c                                                            numeric variables 7.4
1030 k=1                                                                    & operations 7.5
1040 b$="    "                                                              string assignment 7.4
1050 if d<0 then d=abs(d) : k=2 : b$="+/-i"                                 functions 7.7
1060 w=2*a            (note - all this applies if d negative)
1070 x1=-b/w
1080 x2=sqr(d)/w
1090 on k go to 1100,1130                                                   on goto 7.6
1100 w=x1
1110 x1=x1+x2
1120 x2=w-x2
1130 print "x1&x2=";x1;b$;x2
1140 return                                                                 return 7.10
2000 rem test data n then (a,b,c),...
2010 data 2                                                                 data 7.9
2020 data 1,-3,2
2030 data 7,1,2
32767 end                                                                   end 7.6

Ready
```

run 7.3 & 7.14

```
run
jpp22-Nov-7807:09 AM
soln of quadratic eqns
first 2  examples
soln of 1 *x*x+-3 *x+ 2 =0 is...
x1&x2= 2       1
soln of 7 *x*x+ 1 *x+ 2 =0 is...
x1&x2=-.714286E-1 +/-i .529728

please suppy name? john
now your turn john

john please enter coeffs in a*x*x+b*x+c=0
a,b,c=? 1,0,1
soln of 1 *x*x+ 0 *x+ 1 =0 is...
x1&x2= 0 +/-i 1

john please enter coeffs in a*x*x+b*x+c=0
a,b,c=? 12,8,2
soln of 12 *x*x+ 8 *x+ 2 =0 is...
x1&x2=-.333333 +/-i .235702

john please enter coeffs in a*x*x+b*x+c=0
a,b,c=? 8,4,1
soln of 8 *x*x+ 4 *x+ 1 =0 is...
x1&x2=-.25 +/-i .25

john please enter coeffs in a*x*x+b*x+c=0
a,b,c=? 4,8,1
soln of 4 *x*x+ 8 *x+ 1 =0 is...
x1&x2=-.133975      -1.86603

john please enter coeffs in a*x*x+b*x+c=0
a,b,c=?
Ready

bye
%
connect  time      2:33.00
user cpu time         1.74
sys  cpu time         4.16
$
END-SESSION
```

CTRL Z break 7.12

$\left.\begin{array}{l}\text{BYE}\\ \text{CTRL D}\\ \\ \\ \text{CTRL P}\end{array}\right\}$ signing off 7.15

CHAPTER 8

COMPUTER GAMES - A SIMULATION STUDY

Lecture by

E. CLAYTON

CONTENTS

Just as eating against one's will is injurious to health, so study without a liking for it spoils the memory and it retains nothing.
Leonardo da Vinci : Notebooks

8.1 WHY

The most obvious reason for playing computer games can be given in one word - FUN. However, those of us who have problems of conscience about enjoying ourselves ('life wasn't meant to be easy') must find some deeper reason. Calculators are now being accepted into our schools, and soon the controversy that erupted over their introduction will arise again as small computers are installed. There appears to be two schools of thought. Firstly, there is the notion that these devices will be used only as a crutch; something that allows us to do hard sums (divisions for example). This usage may result in a generation of non-numerate people; people who can do arithmetic, but to whom mathematics is a mystery. The other opinion is that we can use calculators and computers as imagination stimulators. We can use them not only to do hard sums but also to explore the exciting world of mathematics. If this was the only reason we could give for playing computer games, their use would be fully justified. Of course, I favour the latter school of thought and in this lecture I hope to show you why.

8.2 HOW

The *how* of computer games is a function of our imagination and ingenuity. Many games devised for computers are simulations of games that we play elsewhere! Given a visual display unit (VDU), we can play a game of billiards, lose a fortune playing a poker machine, win a game of chess or even play a round of golf. Setting up a game can have many phases depending on the degree of sophistication. The best way to illustrate such a process is to go through a fairly well known game, namely billiards.

Let us imagine that we are going to set up a game of billiards on the VDU. First, we must decide what is strictly relevant to our game. Obviously we can dispense with the table legs - our table will be represented by a rectangle on the screen. What then is necessary? We need some mechanism to simulate collisions. Here we are lucky because classical mechanics gives us all the appropriate mathematics. As a matter of fact, the equations governing our case are known as *billiard ball collisions*. Possibly our ancestors were really trying to improve their billiard play instead of solving the problems of the universe as we have been led to believe.

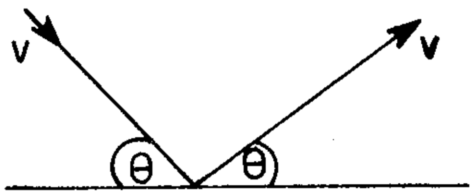
Rather than go through the mathematics let us just say that our collisions will be represented by a function F:

$$F(\phi, V_1, V_2) \equiv f(\theta, v_1, v_2) \quad ,$$

such that two billiard balls, travelling with velocities v_1 and v_2 at some angle θ to each other will collide and then travel with velocities V_1 and V_2 at some angle ϕ to each other.

Next, we need pockets in the table. That is very easily done. Six circles are drawn in the appropriate place on the rectangle. Once our balls have collided and are heading off in whatever direction, all we have to do is determine whether or not they will finish up in a pocket. I should add that one advantage of our simulation is that if a ball goes into a pocket, it stays there - something that may not always happen in the real case. If we get one in a pocket, our program calculates our score and displays it. Sounds very easy doesn't it?

Now we need to put cushions on our table. Fortunately this is once again no problem. A ball coming in with velocity v at an angle θ to the cushion bounces off at an angle θ and velocity v



We are all set up to go! Now follows a period of feverish coding as we translate this rather broad analysis into a series of statements in a computer program. We will gloss over the frustrations that arise during coding and go straight to the finished product - a shiny new billiards program.

We have a shot and watch as the first collision occurs. The balls collide exactly as our equations predicted, bounce off a cushion exactly as we had predicted, then bounce off another cushion, and another and another and continue on ... quite happily rolling along until we stop the program, realising that something is wrong.

Some head scratching reveals the very obvious solution. There must be friction on our hypothetical table otherwise the balls will continue to roll along until infinity comes. Our game has failed in a very important aspect in all simulation - *the need to take account of all relevant physical processes.* Once we code friction into our model all will be well. However, there is one further point; even if you were to run this game on a desert island, a crowd of spectators will soon appear, so make certain you provide supper and light refreshments for your *kibitzers**

8.3 WORLD GAMES

There are games in which you set up a closed world, give it logical rules to work by and then see how it develops. I think the most famous of these is John Conway's *Life Game* described in *Scientific American* (October 1970). In this game, counters on a large chess board die or reproduce according to some simple genetic rules. These were carefully laid down so that

- there should be no initial pattern for which there is a simple proof that the population can grow without limit;
- there should be initial patterns that apparently grow without limit; and
- there should be simple initial patterns that grow and change for a considerable period of time before coming to an end in three possible ways: fading away completely (from overcrowding or from becoming too sparse); settling into a stable configuration that remains unchanged; or entering an oscillating phase in which they repeat an endless cycle of two or more periods.

Each point or cell of the board has eight neighbouring cells. Conway's genetic laws are very simple:

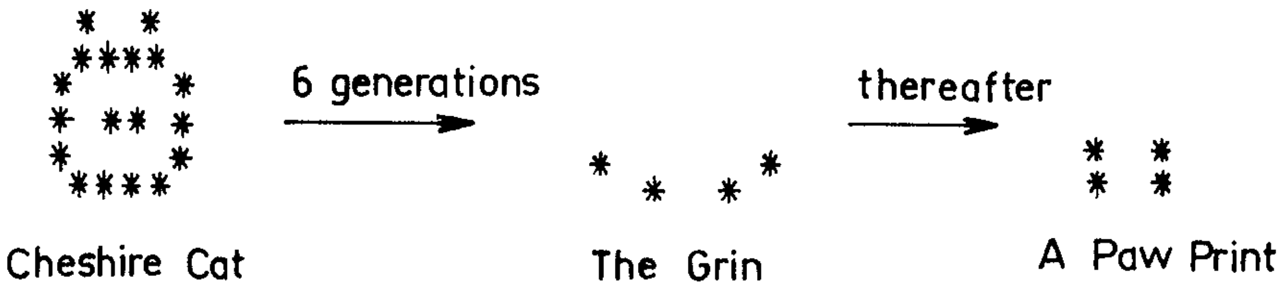
- Survivals Each counter with two or three neighbouring counters survives for the next generation. A counter is an occupied cell.
- Deaths Each counter with four or more neighbours dies (is removed) from overpopulation. Every counter with one neighbour or none dies from isolation.

**Kibitzer:* Meddlesome person; one who gives advice gratuitously; one who watches a game of cards from behind the players; meddlesome looker on [*The Concise Oxford Dictionary*]

- Births Each empty cell adjacent to exactly three neighbours is a birth cell. A counter is placed in it at the next move.

All births and deaths occur simultaneously. Together they form a single generation of the complete life history of the initial configuration.

Although these rules may seem rather dry, this game gives amazing and sometimes quite beautiful patterns. One I like is the Cheshire Cat, which slowly disappears leaving only its grin and finally a paw print.



8.4 CALCULATOR GAMES

Showing lack of imagination, I hadn't really considered playing games on an ordinary pocket calculator. Imagine my surprise and pleasure when I received *Games with the Pocket Calculator*. This book contains 24 games that can be played on a small pocket calculator. These games range from Blackjack (Pontoon) to NIM. My favourite is Defect Detective which runs something like this.

One player thinks up a single malfunction in a calculator, *e.g.* 2.5 is added to every result of a calculation. Other players take turns in supplying a computation problem to the calculator, and from the result try to guess what is wrong.

$$6 + 7 = 15.5$$
$$2 + 2 = 6.5$$
$$10 \div 5 = 4.5$$
$$0.5 + 1 = 4$$
$$2 - 2 = 2.5$$

This gives the clue 2.5 is being added onto every correct answer; 13 + 2.5, 4 + 2.5, 2 + 2.5, 1.5 + 2.5, 0 + 2.5.

Given the mean streak that exists in all of us, we can think of some absolutely fiendish malfunctions to give our calculator. How about reversing the order of digits in the answer. 3 x 8 = 42, even repeating the first digit of the input. 3 + 15 = (33 + 15) = 48, or doing this only if a certain number, say 3, appears (8 + 4 = 12; 8 + 3 = (8 + 33) = 41). Very nasty players might take the exponential of the answer 3 ÷ 2 = (exp(1.5)) = 4.481689!! Any malfunction you can think up will provide headaches and, I hope, a lot of laughs for all concerned.

8.5 ODDS AND ENDS

It is impossible in a short lecture to even scratch the surface of a tiny portion of the world of games and puzzles. The theory of many games has now reached the stage of a 30 page dissertation on the theory of noughts and crosses complete with such strategical concepts as 'semi-threat or threat, double threat and combined threat'. I hope that your imagination has been stirred and that you will go out and play games forever.

8.6 BOOKMANSHIP

The Master Book of Mathematical Recreations. Fred Schuh [1968]. Dover Publications, New York.

Games with the Pocket Calculator. Sivasailam Thiagarajam & Harold D. Stolovitch [1976]. Dymax, Menlo Park, California.

What to Do After You Hit Return or P.C.C's First Book of Computer Games. [1977]. Menlo Park, California.

101 Basic Computer Games. [1973] Digital Equipment Corporation, Maynard, Massachusetts.

The Best of Creative Computing. Vols. 1 and 2, Edited by David Ahl. [1976 and 1977].

NOTES

NOTES

NOTES

NOTES