Introduction

This book is about chaos, a relatively new science, in which the laboratory equipment consists of computer software, the results are often graphical displays of stunning complexity and the subjects of the experiments are numbers themselves. Indeed, many people have already suggested that the term 'Experimental Mathematics' be applied to the whole area of chaos theory and fractals. In this section of the book, I want to look briefly at how this book can be used, and at some of the techniques that we'll be employing.

I suppose that before going any further we should bring in a definition; unfortunately, chaos doesn't lend itself to rapid definitions, so I'll be giving a fuller one later in the book. However, for the time being let's just say that a chaotic system is a deterministic system with great sensitivity to initial conditions. In other words, the behaviour of such a system is predictable by the use of suitable mathematical equations *but varies greatly depending upon what the starting conditions are.* Thus a model of the world's weather on a computer (Chapter 4, The Lorenz equations) can give staggeringly different results when the value of an input parameter is varied by, say, 1 part in 1000. Intuition tells us that small changes in input should give proportional changes at the output; chaos shows us that this isn't necessarily the case.

This book is aimed at anyone with access to a computer running Turbo Pascal or BBC BASIC, although the programs in the book are presented in such a way that they can be duplicated in virtually any language you care to choose, provided that the language has the means of drawing graphical images on a computer display. The programs in the book are designed as starting points for further experimentation, and I'll be giving hints of further things to try in each chapter. It's hoped that the listings will give you a starting point for your own programs.

Although the subject is based in maths, this book is designed as a 'how to do it' book, so I've kept the mathematical content to a minimum. For those of you wishing to follow up some of the mathematics, I've included a substantial list of further reading and computer software. In addition to the computer-based work, I've included in Chapter 11 guidelines for some

experimental work in chaotic systems that you might find in the real world. As to actually using this book, the newcomer might like to start at Chapter 1 and work on through; those with some knowledge of chaos might prefer to get right on with the programs featured. Finally, if there's enough interest, who knows; there might be a book called *Advanced Chaos* ...

The computer

To get the best out of this book you need a computer of some sort. I have concentrated on the IBM PC and clones and the BBC Model B. In order to save space, from here on I'll refer to the BBC Model B as the BBC Micro, and to the IBM PC and its clones as a PC. The main points about any computer used for chaos experiments are as follows.

Graphics

The computer needs some medium-to-high-resolution graphics capability to display the images generated. If you have a PC, then the only problem you'll encounter is if you've got a machine fitted with a text-only display adapter. Most graphics systems fitted to computers are colour, but this doesn't matter too much; very attractive monochrome images can be created.

More important than colour is the resolution of the graphics display. This is the number of individually addressable points available on the screen. I would say that a minimum requirement for the work featured in this book is about 200 horizontal by 200 vertical points (*pixels*) (200×200). You may find that some authors suggest a higher resolution graphics screen than you have on your machine; if this is the case, then my advice would be to try out the programs on your computer unless the author says the program definitely *won't* work. After all, experimentation is the spice of chaos!

Even if your machine cannot support graphics, some chaos work can be done with a text-only display, by printing out the values rather than graphing them. An example of this is given in Chapter 4 when the Lorenz equations are discussed.

The BBC and IBM PC are two machines capable of supporting good graphics. Here follows a brief description of the graphics modes available on the BBC and the common graphics standards for the PC.

BBC graphics modes

On the BBC Microcomputer, there is a trade-off between screen resolution and memory available for programs. Table 1 shows some common BBC

graphics modes. Modes 0 to 2 take up 20k of memory, and this leaves only around 5k or so for programs when the BBC has a disc drive fitted. Modes 4 and 5 take up 10k of memory.

Mode	Colours	Resolution
0	2 colours	640×256 pixels
1	4 colours	320×256 pixels
2	16 colours	160×256 pixels
4	2 colours	320×256 pixels
5	4 colours	160×256 pixels

Table 1. BBC Micro screen modes.

The other BBC screen modes are not really suitable for use in most of the programs in this book, but can be pressed into service for things like the Life program in Chapter 10.

IBM graphics adapters

PCs don't have a range of built-in graphics modes like the BBC; instead, they can be equipped with an add-in card and suitable display to suit the requirements of the user. Once upon a time, all PCs came with *no* display

Adapter	Resolution	Colours	BIOS Mode
CGA	40×25 and 80×25	16 colour text	
	320×200 pixels	4 colours	4 and 5
	640×200 pixels	2 colours	6
EGA	as CGA plus:		
	320×200 pixels	16 colours	13
	640×200 pixels	16 colours	14
	640×350 pixels	16 colours	15
VGA	as EGA plus:		
	640×480 pixels	2 colours	17
	640×480 pixels	16 colours	18
	320×200 pixels	256 colours	19

Table 2. Common IBM PC screen modes.

system at all; a shock to those of us who were more used to home computers! The most common graphics modes supported on IBM machines are shown in Table 2.

The VGA adapter is the latest available on most PC systems. The BIOS mode referred to above is a reference to how the IBM PC operating system BIOS (the program built into a computer that looks after such things as keyboards and displays) turns a particular graphics mode on and off. If you're interested in this sort of thing, I suggest you look at the listings in the Appendix for examples of use of BIOS calls, and look at some of the books listed in the Bibliography. Although it looks like VGA owners are spoilt for choice, it should be pointed out that not all programming languages and pieces of software can actually use all of the modes available unless direct BIOS calls are used.

Memory

The memory of a computer is the amount of space available in it for the storage of computer programs and other data needed for the correct running of a computer program. On BBC Microcomputers, the amount of memory available for the computer program depends upon the resolution of graphics used. On PCs, the amount of memory available is usually 512–640k (1k = 1024 bytes = 1024 characters) irrespective of the graphics mode in use. Some PCs are equipped with more memory than this, but it isn't usually much use to chaos experimenters unless some special programming techniques are employed, so I won't discuss it here. In the topics covered in this book, memory is used for:

1. Storing the program instructions needed.
2. Storing any data needed for the program to run properly.
3. Storing data used by the program or produced by the program.

In general terms, the larger the amount of memory available the quicker the program will run and the less likely the programs are to 'fall over' and fail to work properly.

Processing power

The processing power of a computer depends principally upon the type of central processing unit the computer uses. This is the chip at the heart of the computer that does all the work.

The BBC Microcomputer uses a 6502 chip, quite an old-timer in the microprocessor world. It's also not a terribly powerful chip, and users of the BBC Micro who need a lot of clout behind their programs tend to use second processors—add-on boxes that contain a further CPU that does the

actual processing in programs, leaving the BBC's own 6502 to look after just the keyboard and screen.

Life is more complicated with the IBM PC. There are three CPU chip families in wide circulation.

8088/8086: These chips are quite old, but are still to be found in many machines, such as the Amstrad PC1512, Toshiba 1200 and many other low-cost clones. Machines using these chips are often called PC machines, as the original IBM PC used one of these processors.

80286: This is a more powerful processor, which also can handle more computer memory than the 8086/8088 processor. Machines using this processor are often called AT clones.

80386/80386SX: These processors are very powerful indeed, and PC clones using these machines represent the most powerful machines in common use. The SX chip is a slightly cut-down version of the 80386, but for our purposes the SX chip is quite adequate.

As well as the processor type, the speed at which it operates is also quite important. For example, the BBC 6502 trundles along at 1MHz (1 million steps per second). This isn't the same as 1 million calculations every second. The 8086/8088 processors usually operate in the 4MHz to 8MHz range, 80286s up to 12MHz and 386SX/386 processors anywhere between 15 and 33MHz. You can get 20MHz 80286 chips which will outpace some of the slower 80386SX chips, but do not offer some of the more subtle advantages of the 80386SX. However, for chaos and fractals work, these chips are fine. Some machines are equipped with a 'Turbo' mode, in which a machine that normally runs at, say, 4MHz can, at the push of a button, run at 7 or 8MHz. The reason for having two modes is that some software requires that the computer be running at an official speed—in this case, the 4MHz speed—and may not function correctly if the higher speed is used.

A useful adjunct to the CPU for any computer that's doing a lot of numerical work is a special chip called a *maths coprocessor.* In the usual run of things, the CPU has to do all the arithmetic, and while it's doing this it can't do anything else. Mathematical operations are not usually the forte of computer chips; they're fine at doing simple arithmetic, but to do complex mathematics they have to execute programs of instructions and this takes time. A coprocessor is designed to do the complicated mathematical operations under the control of the CPU, and it will then pass the results back to the CPU when the calculations are completed. Such a device will make vast improvements in speed to the overall running of mathematically oriented programs, provided that the language used to write the programs can support the maths coprocessor. There is no dedicated maths coprocessor available for the 6502 chip, but for the others, the processor number

needed is obtained by replacing the 6 in the part name with a 7. So the 80286 requires an 80287 maths coprocessor.

Disc drives

The computer disc drive system is simply used for rapid storage and retrieval of data. It's not essential for chaos work; you can use cassette tapes for program and data storage on the BBC Micro, but it makes life easier! Programs can be retrieved more quickly from disc than from tape. On the PCs, there isn't an effective tape system available, and all clones need at least one disc drive, as described below. Furthermore, on PCs many computer languages use the disc as temporary storage whilst you are writing programs, so the disc drive effectively becomes an extension of your computer's memory.

BBC disc systems

There are two main systems on the BBC Micro. These are the 40-track disc, capable of holding 100,000 characters of data (100k) and the 80-track disc, which holds 200k. Figure 1 shows a typical *floppy* disc. The disc fits into a disc *drive* which allows data to be read from and written to the disc as a series of magnetic imprints on the disc surface. The discs are reusable and if you need to use another disc, you simply take the current disc out of the drive, put it in a storage bag, and stick in another one.

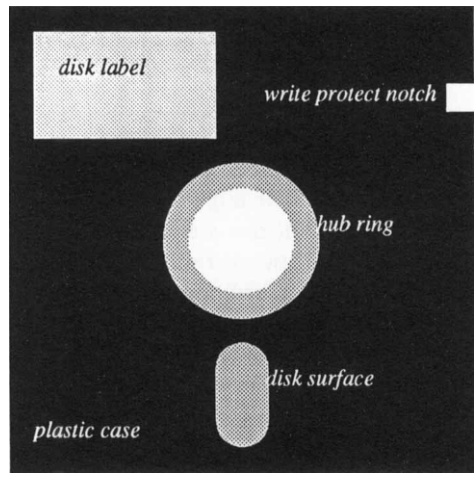

Figure 1. A typical floppy disc.

Floppy discs come in two physical sizes (5.25" and 3.5") and the amount of data that can be stored on them depends upon how the computer *formats* the disc to receive information.

PC disc systems

On PCs there are a variety of floppy disc formats to deal with:

3.5" There are two types of 3.5" disc, called *low density* and *high density*. Low density discs can hold about 720k of data, and high density discs hold about 1400k of data.

5.25" Again, high and low density discs are available, with low density discs holding 360k of data and high density discs holding 1200k of data.

Hard discs

All PC clones will come with at least one floppy disc drive. Some machines have two, but a more common configuration is to have one floppy disc drive and one hard disc. A hard disc is a *non-removable* storage medium; that is, when it fills up you have to erase some old data before you can save new material on to the disc. It's not easy to remove the disc and stick a new one in, but the disc is reusable, and many users move old data from their hard disc on to floppy discs, and then erase the old data from the hard disc to make space for new material. Hard discs can hold large amounts of data—anywhere between 20 million and several hundred million bytes—and are also much faster than floppy discs. A hard disc is well worth the extra cost as it usually makes programming easier and faster and is particularly valuable for storing and recovering screen image files or other large amounts of data.

Interfaces

Computer interfaces are the ways in which the computer talks to other devices in the outside world. Most computers come with at least an interface to allow the computer to send information to a printer to produce hard-copy results from programs. This is usually what is known as a *Centronics* interface, which is a standard means of communication between computers and printers. A further means of communication between computers and peripheral devices is via an *RS232* interface, which allows the computer to communicate with plotters, which can draw graphs in many colours, or allows data transfer between two different computers, even if they are of different makes. If you have an additional piece of equipment called a *modem*, you can communicate with other computers over the telephone line via the RS232 interface.

If you go on to experiment with chaotic systems in the real world, rather than various computer models, then you will require additional interfaces on your computer to convert things happening in the real world into electrical signals that the computer can understand. The BBC Micro is equipped with some of these interfaces, in the form of a *User Port* and *Analogue to Digital Converter* (ADC). The User Port allows simple electrical on or off signals to be read by the computer, whilst the ADC allows continuously varying voltages to be recorded on a computer. Quantities such as light level or temperature can thus be recorded on a computer via a suitable electronic circuit to convert, say, light levels into voltages.

Programming languages

Without suitable programs, computers are simply hi-tech door stops. These programs—sequences of instructions for the computer to follow to get a particular result— are written in particular computer languages. The CPU of a computer only understands one language. This is called machine code, and machine code programs are incredibly fast because the CPU can look at instructions in a machine code program and know exactly what to do without any further processing. However, machine code programs are unintelligible to humans; after all, a machine code program looks like a long list of numbers. Languages such as BASIC or Pascal, called *high level languages*, make more sense to people but cannot be executed directly by the CPU. The programs in these two languages need further processing before they can be executed by the CPU of the computer. There are two ways in which this can be done—*interpretation* and *compilation.*

Interpreted languages

In interpreted languages, like the BASIC on the BBC Micro or the popular GWBASIC for the IBM PC, the set of instructions that is written in the language is examined by a language interpreter and a series of machine code instructions are executed for each instruction in the high level language program. Thus a single high level instruction may give rise to several thousand machine code instructions, which are pre-programmed in to the language interpreter. If we had a simple loop in a program, like:

```
FOR I=1 TO 10
PRINT "HELLO WORLD"
NEXT I
```

then the statement *PRINT "HELLO WORLD"* would be reinterpreted ten times! This, combined with the fact that the machine code statements

executed when this line is interpreted may not be as efficient as they could be, leads to interpreted languages being a little slow.

Compiled languages

In a compiled language, such as Turbo Pascal or Turbo BASIC on the IBM PC, the set of instructions making up the high level program is examined just *once* by a language *compiler*, which ultimately produces a machine code program for direct execution by the processor of the computer. This program can then be run on other machines that *don't* have the language compiler, and it also runs very quickly. Although the resultant machine code program might not be as efficient as it would be if it were written from the start in machine code, it's still much faster (often ten to twenty times) than a similar program in an interpreted language because the conversion of high level code into machine code instructions is done *once* only, at compilation time, rather than every time the high level instruction is executed.

Which languages are best?

Provided that you can access graphics facilities from a particular language, any language can be used to do experiments in chaos and fractals. My choice of Pascal and BASIC for this book was based on the following criteria.

Availability: It's likely that most readers would have access to one or both of these languages.

Algorithm Description: The algorithm of a computer program is the method used to describe how the task the program performs is done. These two languages are quite well suited to describing how things are done, if good programming practice is followed, so writing the programs in these languages allows programmers in other languages to see easily how the algorithm works.

Familiarity: I know how to program in these languages, so, why not use them. In addition, the languages are widely available and fairly cheap.

Ease of handling of floating point numbers.

However, there's nothing to stop users of other languages writing versions of the software listed in this book in those languages. Assembler language (a human version of machine code) has the advantage of speed but can entail the programmer in writing code for handling floating point arithmetic. In addition, writing assembler code is often a more long-winded process that writing programs in higher level languages. The C language has proved to be quite popular for some chaos experimenters, as it offers

Type	Range			Significant digits	Bytes
real	2.9E-39	to	1.7E+38	11–12	6
int	-32768	to	32767	5	2
long	-2147483648	to	2147483647	10	4

Table 3. Numerical resolution in Turbo Pascal

the advantages of assembler language (fast, efficient programs) with some of the ease of use of higher level languages. However, these two languages fall down when writing a book like this one in that the algorithms used in programs are not as clear as when using BASIC or Pascal; often further documentation of the code is needed before programmers can translate C or Assembler over to other languages.

Some programmers combine the best of both worlds by using assembler code for relatively simple, time-consuming tasks and integrate these routines with the main body of the program in a high level language. It's not a good idea to try and write floating point routines in assembler yourself; the ones provided with most computer languages are quite well written and to improve on them would take a lot of effort.

On the whole, therefore, which language you use is very much a personal choice. However, there are some points to take into account when considering a language, and to be borne in mind when using the language in chaos work. Many of the details about a language will be found in the programming guide for that language, but do bear these points in mind.

Numerical resolution

Computers store numbers, be they integers or floating point numbers, in a finite number of bytes of memory. For example, a computer may store integers in two bytes; this would allow positive numbers between 0 and 65535 to be stored. Increasing the number of bytes allocated to each integer allows a wider range of integers to be stored. The situation is the same with floating point numbers. For example, Turbo Pascal has the numerical resolution shown in Table 3.

The significant digits part is important, especially for the real number. Consider the number 1.6666666666666666666; clearly, if we have to fit this in to eleven significant digits, we'll end up with 1.6666666667, and we have a rounding error in the last position. With maths coprocessors, a resolution of up to twenty significant digits is possible, but then at that point we'll still have some rounding errors if we try and specify a higher resolution than the language can support. With integers, it's often possible to write routines that allow integer arithmetic to be done on integers with several hundred

significant digits if required, but extending the floating point resolution by writing your own routines is *not* to be undertaken lightly!

The significance of this to chaos programs lies in the simple definition we gave above; a chaotic system has extreme sensitivity to initial conditions. If the resolution of our system is approached, we lose accuracy and results obtained may actually be ***artifacts*** caused by this loss of accuracy. These artifacts may give rise to interesting effects themselves, but we need to be aware of their existence when operating at the extremes of numerical resolution.

Numerical accuracy

There's also the possibility of errors of accuracy in programming in some versions of languages. The classical error is something like

$$1 + 1 = 1.999999999$$

This is a rounding error in the arithmetic functions in the language, and when it occurs we can often fudge things to put it right. However, it can still cause us problems. Further accuracy problems might be caused with trigonometrical functions, such as sin and cos, as there are a variety of methods by which values can be computed. A further example is the popular method of arriving at a square root of a number by raising it to the power of a half; this will often give a different result to using the square root function of the same language. Now, the answers obtained will be correct for most practical purposes, but if were to use these values as seeds for some chaotic system, the difference in the fourth or fifth place after the decimal point might cause problems and create artifacts.

Numerical overflow

The largest or smallest number that can be represented in a computer language depends upon the number of bytes used for storage of numbers, whether it's a real number or integer, and so on. It's very easy in chaos work, especially when dealing with things like *iterative functions* (Chapter 2) to generate numbers that overflow the computer's ability to cope. This usually generates a fatal error causing execution of the program to stop.

Types of computer

Although I'll be providing listings for only the BBC Micro and the IBM PC in this book, there is nothing to stop you using other computers, if they

have the minimum requirements in terms of graphics mentioned above. However, some computers may not have the processing power needed to complete graphics displays as quickly as the Beeb or the PC. Indeed, some displays take many hours to create on even powerful 386SX computers, and literally *days* to produce on the BBC micro. However, there's nothing to stop you trying out the listings with any computer you've got, but be aware that you'll have to change the listings slightly in many cases to suit the computer used. For the record, the computers used in the writing of this book were as follows:

Packard Bell 386SX: A 386SX-based PC with high-resolution VGA graphics VGA, supporting a 40Mb hard disc and all PC floppy formats. Programs in Turbo Pascal and GWBASIC.

Amstrad PC1512: An 8086-based PC clone, 512k of memory and lower resolution graphics (CGA) than the 386SX. Programs in Turbo Pascal and GWBASIC.

BBC Model B: A 6502-based machine, 32k of memory and low-to-medium-resolution graphics. Programs in BBC BASIC.

Processing of screen images

As already mentioned, many experiments in chaos and fractals generate graphical images. Combine this with the fact that these images often take a long time to assemble and it is not surprising that you need to think about ways of preserving graphical information so that the images can be *quickly* recreated at any time in the future without having to rerun the program that generated them in the first place. Alternatively, you may wish to send images to other people, store them for archive purposes or use them as the basis of other graphics images after further processing. Before starting with any work, it would be useful to examine briefly the various methods of doing this.

Saving video memory

The video memory is the section of the computer's memory that stores the information that makes up the screen image. The easiest way to store an image is thus to simply save the video memory holding the image to disc. Saving it to tape is probably *not* a good idea, as video images tend to be big!

On the BBC Microcomputer, the **SAVE* option from the BBC operating system can be used to save the area of memory used by the graphics

mode in current use. The memory taken up by each screen mode can be calculated from information shown in the BBC User Guide or Advanced User Guide. With the PC, this isn't as straightforward and usually some facilities offered by the language need to be used. For example, in the Turbo Pascal library in the Appendix, I use the *PutImage* and *GetImage* functions to store and recall graphics images from disc.

Although this method of storing images is the simplest, it has some disadvantages. These are as follows:

1. The images are very large. For example, they can be up to twenty kilobytes long on the BBC Microcomputer and a few hundred kilobytes long on PC VGA systems. Thus a few images can take up a large amount of disc space.

2. These images cannot be loaded in to other software packages than those that can handle screen memory images. This isn't too big a problem on BBC and other small computers, as the method of storing screen images as memory dumps is fairly widespread, but on the PC most graphics packages use a more efficient method of storing images. If the images are stored for use in programs that you've written and distributed, though, the mode of storage is up to you.

Using a standard file format for storage

One option adopted by some PC programmers is to write functions in their own software to save screen images in a format that is a standard, such as the PCX file format used by PC Paintbrush and other packages. Popular standards include:

PCX: This is the standard format used by PC Paintbrush, and the file format has been made available by the software house concerned, ZSoft, as an aid to programmers wishing to develop software compatible with this format. PCX is supported by PC Paintbrush, Microsoft Paint, Aldus Pagemaker, WordPerfect 5, Word 5.0 and a variety of other packages.

GIF: The GIF standard was designed by CompuServe, a computer bulletin board provider. It allows great compression of data but is not supported by vast numbers of programs. There are shareware converters available, and the format is becoming more popular.

TIFF: This format is used by many of the applications listed under PCX, with the addition of Logitech's Paintshow graphics software.

The advantage of writing files in these applications is that they can be read back immediately by other applications that use the same format.

For example, I have saved an image of the Mandelbrot set as a PCX file using a screen grabber, loaded it in to WordPerfect and then printed the whole thing out as a combination of text and graphics. The disadvantage is that programming some of these standards, even when the information is available about how the format is put together, can be a little difficult.

Using a screen grabber

A happy medium is to use a screen-grabbing utility, such as FRIEZE, CATCH (Paintshow) or GRAB (WordPerfect). These are resident programs on the PC which can be invoked by a couple of keystrokes and allow you to save the current screen display, ***however it was created***, to a file in one of the standard formats. These applications usually come with a graphics package or other software capable of processing graphics and will support at least the graphics format used by that application. Using such software offers the best of all possible worlds; someone else has done the hard work for you, and the output from your programs can be captured, when the software is installed, without you having to modify any existing software.

Run length coding

This is a method of screen compression and storage which is fairly simple to implement, if a little cumbersome in some cases. The idea is simply to store graphics information as sequences of bytes representing the colour of a pixel and the number of subsequent pixels that are the same colour. As soon as a change in the colour is found then the process is repeated for the next colour and run of bytes. As an example, if we started with pixel 0 in the top left corner of the screen, and said that pixel numbers increased from left to right and from top to bottom of the screen, then a simple RLC system could start with the colour of pixel 0 in 1 byte, followed by 2 bytes holding the number of pixels following that have that colour. If a run of more than 65535 pixels of the same colour was obtained, then the fourth byte would have the same colour value as the first one, and the following two bytes would have the number of pixels of that colour still remaining. RLC files are thus built up of clusters of bytes, depending upon how many bytes are used to store the run length. The RLC method of screen compression is most effective when the image consists of larger areas of the same colour, and gives least compression for highly detailed images.

Once a screen image has been filed in some way, it can be recoloured, stretched, shrunk and generally messed about with using whatever graphics facilities you've got. Then it can be printed to give hard copy.

Getting a hard copy

To transfer the screen image on to paper can be quite tricky. The easiest way is to output the data to a printer.

Printer dumps

The PC comes with a function called *GRAPHICS* that transfers the screen data to a printer in some cases, but for much of the time it will not work with high-resolution graphics images. On the BBC Micro, you will certainly have to write or buy a screen dump utility. Alternatively, most graphics programs offer a way of printing out your work, so why not use that if you've got it, by screen grabbing the image, loading it in to the graphics package, then printing it from there.

Screen dump utilities are generally written for one specific printer or a selected few. If you've got an odd printer, you may need to search around a little for a suitable utility. Use of plotters or colour printers gives the same problems; this is why using a graphics package rather than a screen dump utility is often a good idea, as these packages usually support a wide range of output devices.

Screen photography

An alternative to making a printout is to take a photograph from the screen. This is fairly simple providing that some simple rules are followed. The camera used should be some sort of SLR, with a facility for tripod mounting and a timed exposure of several seconds. This is because the photograph will be taken over a period of a few seconds to avoid any risk of dark bands on the screen caused by the shutter of the camera operating half way through the screen being drawn. Any shutter speed of less than 1 second might not be a good idea, especially with the relatively slow film I use. As to film, I tend to use ISO 50 or ISO 100 colour transparency film, as this gives the most versatile finished result. For example, I can put the resulting slide in a projector and show it like that, have it made into a print or even take it to my local printers who'll put it into a colour photocopier and provide me with an A4 colour photocopy of the slide. Once you find a film you're happy with, stick with it. For the record, the slides used in this book were taken on Fuji film with a Canon EOS 600. The steps involved are as follows.

1. Load the camera with the film, then go outside and take a couple of shots of anything that's about, provided that it's a good, contrasty subject. The reason for this is to prevent the automatic equipment that cuts film up into frames deciding that the middle of your first

exposure, which was a little bit dark, is the start of the first frame. The first couple of shots give a reference point for the equipment to prevent this happening.

2. Set up the computer in a darkened room, the darker the better, so as to avoid any unwanted reflections or glare on the computer screen. I tend to take screen shots at night, curtains drawn, lights off. It's a good idea to set up the camera *before* turning all the lights out!

3. Set up the camera with a 50mm lens set to $f5.6$ or $f8$ on a tripod so that the field of view is totally occupied by the computer screen. Ensure that an imaginary line drawn through the lens to the screen would strike the screen at ninety degrees. Turn off any automatic focus the camera has, and focus the camera on the screen. To facilitate this, I have a small BASIC program which, on running, splits the screen into four quarters that are coloured black and white to give an effect like four squares of a chess board. I focus on the intersection of these. Don't select a smaller aperture as the resultant shallow depth of field may prevent the edges of the monitor screen, which in most cases are slightly further away than the centre of the screen, from being in focus.

4. Once focusing is done, load up the image that you want to photograph and adjust the brightness and contrast controls to give a clear image. If your camera has through-the-lens metering, then setting exposure is fairly straightforward. Set the exposure as indicated when focusing the camera on an average part of the image—not total black or total white. If you use a separate light meter, take a reading from an average area of the image. The exposure time will probably be in the range 0.5 to a few seconds. Set the exposure time. If you can't get a time in this range, adjust the brightness of the monitor or open the aperture of the camera. If it's too bright use a neutral density filter.

5. Take the exposure. In order to reduce shake, use a remote cable. If your camera doesn't have the exposure time needed as a setting, use the Bulb setting and a stop watch.

6. Take two *bracketing* exposures, at 0.75 times the indicated time and 1.25 times the indicated time. You may wish to have more bracketing shots at wider time settings. The only drawback with slide film is that good results do depend rather a lot on getting the exposure correct. After a few films, you'll get the hang of things and will probably be able to dispense with bracketing shots.

Expect the first couple of films to be a bit variable in quality, but after that you should have no problems at all. I've found that a photographic

record of my better results is rather useful, and certainly less trouble to show others than having to get the computer set up!

Data export

Although we've examined the different ways of transferring screen images between programs, some programs produce data in a numeric form, such as the differential equations shown in Chapter 3. It's occasionally useful to be able to output such data for processing via another program. On the BBC, there doesn't appear to be a widely accepted standard file transfer format except for ASCII, where information is placed in a file as plain text, numbers being represented as a series of digits. However, on the PC there are a couple of standard ways of passing textual or numeric data between programs. These include:

Comma Separated Value files are files of data with each record of data separated from another by a new line and each piece of data within a record separated by a comma. For example, a series of data points and times might be passed as:

```
Time   Data

900,   1234   Record #1
901,   1236   Record #2
902,   1345   Record #3
```

Fixed Fields where data is sent as a series of strings separated by new lines. The same data as used above would be passed as:

```
09001234   Record #1
09011236   Record #2
09021345   Record #3
```

and the software receiving the data would be told to expect the data part of the record to start at character 5.

Language versions used in this book

I've used the following languages in this book. For the Pascal listings I used Turbo Pascal version 4; earlier versions do not support *Units* but will still run the programs with some changes to take the lack of Units into account, typically by including external files with the *$I* compiler directive. In addition, some of the graphics commands, such as *PutPixel*, have different names. See the *Extend* unit in the Appendix for additional functions.

For the BBC BASIC programs I used version 1.2 on a rather venerable BBC Model B. The BASIC listings will also run with the various BBC BASICs available on the Master and Electron computers, as well as versions of BBC BASIC that are available for the PC.