The complementary roles of prepared software & programming in the learning of mathematics

David Tall

Mathematics Education Research Centre
University of Warwick
COVENTRY CV4 7AL

Introduction

This article represents the distillation of several years' experience working with two complementary ways of using the computer to enhance mathematical learning. On the one hand there is the use of prepared software, carefully error-protected and specially designed for both demonstration and exploration of mathematical concepts. On the other hand is the activity of students using short programs, modifying them to solve similar problems and writing their own to carry out mathematical algorithms.

The first activity, the use of prepared software, will be exemplified by my *Graphic Calculus* programs (Tall 1986a), which are described in greater detail in a series of six papers, in *Mathematics Teaching*, beginning with *"understanding the calculus"* (Tall 1984). The second activity will be exemplified by the use of *132 Short Programs for the Mathematics Classroom* , published by the Mathematical Association, for whom I was a member of the writing team. These programs are written in BASIC, structured using the facilities of the BBC dialect wherever possible, but many comments made here apply to other languages, in particular to Logo, wherever the programming activities are aimed at developing an understanding of the underlying mathematical concepts.

My initial idea in writing *Graphic Calculus* was that suitably prepared software could furnish mathematical experiences to aid the formation of mathematical concepts without the need for any computer programming by the pupil or the teacher. Such an approach has been shown to have significant success (Tall 1986b). However, as I observed pupils and teachers using the material, I realised that understanding would be enhanced by insight into the way the computer was carrying out the algorithms and conjectured that this could be done through simple computer programming.

On the other side of the coin, to be able to program with any degree of flexibility requires a considerable investment in time and effort. Furthermore, any computer language has idiosyncrasies which may conflict with the mathematical task in hand. Even if the student has sufficient familiarity with a language to be able to program confidently, it is one thing to be able to write a program to carry out a simple specific task, it is quite another to create an environment that allows flexible exploration. Thus writing short programs to carry out a mathematical process, may well be followed by the use of appropriate prepared software for more extended investigations.

In this paper I shall therefore advocate a compromise. The most economical method of learning specific new mathematical concepts may be through prepared software, but it is best complemented by simple programming to gain further insight into the nature of the mathematical algorithms.

Strengths and weaknesses of short programs

It is remarkable how powerful very simple programs can be. For instance a single function definition in BASIC such as

```
1000 DEF FNf(x)=SIN(x)
```

allows one to print out values of the function so that

```
PRINT FNf(PI/2)
```

will (on the BBC computer) give the value 1.

The same techniques work if SIN(x) is replaced by a more complicated expression, so that a function definition will print out the values of any given formula without needing to go through the sequence of intermediate calculations that would be needed to evaluate the expression using a calculator.

A function definition may be used to investigate zeros of the function. For instance, if FNf(x)=SIN(x) one might note that the value of FNf(3) is positive and FNf(4) is negative, suggesting that, all being well, there may be a zero between x=3 and x=4. Initially intelligent guesswork may allow one to home in onto a more accurate estimate and this exploration could move on to more structured methods. For instance one may look at a range of values with the command:

```
FOR x=3 TO 4 STEP 0.1 : PRINT x,FNf(x) : NEXT
```

to give the sequence of values (to two decimal places):

```
3.00          0.14
3.10          0.04
3.20         –0.06
3.30         –0.16
etc.
```

which suggests a root between 3.1 and 3.2. Further repetitions with successively smaller steps, such as:

```
FOR x=3.1 TO 3.2 STEP 0.01 : PRINT x, FNf(x) : NEXT
```

will allow a closer approximation to be found. Such exploration greatly enhances the understanding of more sophisticated techniques which follow.

The initial difficulty of this approach in practice lies not the mathematics, but in the vagaries of the computer formatting of numbers. For instance, on the BBC computer, the enigmatic command @%=&2020A is needed to give two decimal places. If this command is not issued, the first list of pairs of values x,FNf(x) given above is displayed as the sequence:

```
30.141120008
3.14.15806628E–2
3.2–5.83741427E–2
3.3–0.157745693
etc.
```

Here the default number formatting runs one number into the next. The two lists should be compared very carefully to try to divine what the second one means !

However, there is more than cosmetic difficulty involved in using the computer. Even if the computing environment were made more friendly by the introduction of new keywords, such as **DECIMAL N** to give N decimal places, there would still be the need to come to terms with the particular manner in which a computer stores and handles numbers.

For instance, the computer would seem to be an ideal environment to look at limiting arguments and calculate numerical derivatives. The function definition

```
2000 DEF FNg(x,h)=(FNf(x+h)–FNf(x))/h
```

in BBC BASIC can be used to investigate the numerical gradient g(x,h) of the earlier function definition from (x,f(x)) to (x+h,f(x+h)) as h gets small. The gradient at $x=\pi/3$ for f(x)=sin(x) may be considered by taking h to be successively 1/10, 1/100, etc, using the command:

```
FOR n=1 TO 10 : PRINT FNg(PI/3,1/10↑n) : NEXT
```

On the BBC computer this gives the sequence:

```
0.455901884
0.495661539
0.499954913
0.499980524
0.499654561
0.500585884
0.465661287
0.232830644
0
```

Could *any* student faced only with this information be expected to believe that, as $h\to 0$, so g(x,h)$\to$1/2? Why does the sequence fail to approach 1/2 and why does it end up being zero ?

This is due to the errors in numerical representation and calculation. Suppose that the expression d=f(x+h)–f(x) is calculated with an error $\pm e$. Then upon dividing by h, even if there were no more errors in calculation, the result would be

$$\frac{d\pm e}{h} ,$$

which is the correct answer d/h, plus or minus an error e/h. If the computer is giving an answer to nine decimal places, then the error e is of the order of size 10^{-9}. For a small value of h, say $h=10^{-8}$, the error e/h in the answer is then about 1/10, which means that even the accuracy of the first decimal place is in doubt! As a practical rule of thumb, the numerical derivative is likely to be most accurate when h starts its decimal expansion about halfway along the number of decimal places displayed, i.e. for 9 decimal places, h should be around $1/10^4$ or $1/10^5$.

When h is much smaller, say $h=1/10^{10}$, a new phenomenon occurs. The top of the expression,

$$\frac{f(x+h)-f(x)}{h}$$

is the difference between two numbers which are so close, that they are represented within the computer as being equal, so their difference is zero. But the bottom of the expression, h, is represented in exponent form (in this case as 1E–10), which is non-zero, and so the calculated value of the whole expression is zero. Thus the attempt to calculate the numerical limit of the gradient as h gets small, first tends towards the true limit, until the maximum accuracy is reached (around $h=1/10^4$ or $1/10^5$), then accuracy deteriorates until the result is either displayed as zero, or becomes undefined.

The notion of a numerical gradient can still be very useful, provided the value of h is not taken to be too small. For example, in *132 Short Programs*, there is a useful program to draw the numerical derivative. Here it is in a structured form:

```
  10 MODE 1
  20 INPUT "f(x)=" f$
  30 PROCdrawaxes
  40 PROCdrawgraph
  50 PROCplotgradient
  60 END
  70
  80 DEFPROCdrawaxes
  90 VDU 29,640;512; : GCOL 0,3
 110 MOVE –500,0 : DRAW 500,0 : MOVE 0,–500 : DRAW 0,500
 120 VDU5 : MOVE–500,–10 : PRINT "–5" : MOVE 500,–10 : PRINT "5" :
VDU4
 130 ENDPROC
 140
 150 DEFPROCdrawgraph
 160 GCOL 0,2 : MOVE 100*(–5),100*FNf(–5)
 170 FOR x=–5 TO 5 STEP 0.2 : DRAW 100*x, 100*FNf(x) : NEXT x
 180 ENDPROC
 190
 200 DEFPROCplotgradient
 210 h=0.001
 220 GCOL 0,3
 230 FOR x=–5 TO 5 STEP 0.2 : PLOT69,100*x,100*FNg(x) : NEXT x
 240 ENDPROC
 250
 260 DEF FNf(x)=EVALf$
 270 DEF FNg(x)=(FNf(x+h)–FNf(x))/h
```

The main part of the program (lines 10-60) selects a four colour graphic mode on the BBC computer (MODE 1), requests the input of a function, then goes through three procedures to draw the axes, draw the graph and plot the gradient. These procedures are separated out after the end of the program, so that the structure can be more clearly seen. The main program uses meaningful names whilst the computer jargon is relegated to the procedures.

PROCdrawaxes starts with a VDU29 command, which is the BBC computerese to select the origin at the centre of the screen (with screen coordinates (640,512)) then selects graphic colour 3 (white). Line 110 draws the axes and line 120 shows the scale by printing –5 and 5 at the extremes of the x-axis (where VDU5 is the command to print at the graphics cursor and VDU4 returns to normal text output). PROCdrawgraph selects graphic colour 2 (yellow), moves to the point (–5,f(–5)) on the graph then successively joins up to (x,f(x)) as x increases in steps of 0.2 from –5 to 5. The multiplicative factor 100 simply converts the x&y ranges –5 to 5 into screen coordinates from –500 to 500 in each direction. PROCplotgradient is the same kind of thing, except it selects colour 3 (red) and uses PLOT69, which is the BBC computer's terminology for plotting single points. Note that the definition of the function FNf(x) uses the EVAL command to *evaluate* the value of the string f$, (an extremely useful facility in BBC BASIC).

This program is indeed very powerful. For instance, if it is RUN and the function f(*x*) is typed in as

f(x)=**SIN(x)**

then the sine curve will be drawn and the set of points representing the gradient is clearly in the shape of the cosine curve (figure 1).

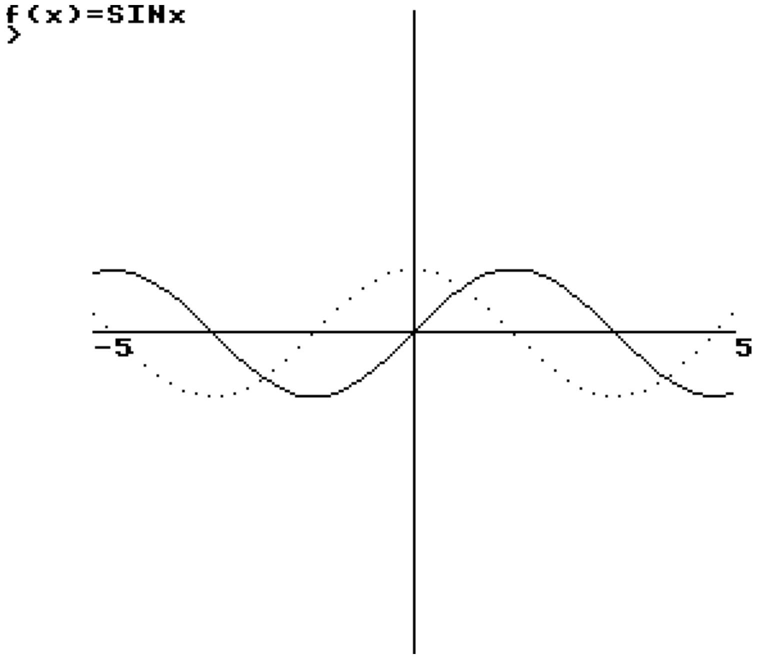


Figure 1

Further investigations with this program will draw very evocative numerical gradients for graphs such as f(*x*)=*x*^2, f(*x*)=*x*^3, which will begin to give some insight into the formula for the derivative of x^n. However, f(*x*)=*x*^4 provides a surprise (figure 2).

What are the funny near vertical lines drawn as part of the first graph? The answer lies in the way the computer represents the coordinates in memory. Each coordinate is held as a two byte number and, as each byte is 8 binary digits, that means that the coordinate is held in memory as a 16 digit binary number. However, the most significant digit is used to represent the sign of the number (1 for positive, 0 for negative). This leaves 15 bytes for the number itself, a maximum of

$2^{15}-1=32767$.

If the arithmetic takes the result slightly above this value, the most significant bit is set and the number is considered as being *negative*. As the *y*-coordinate on-screen is initially 100*(–5)^4=62500 which exceeds 32767, the graph starts off erroneously as "negative" and then switches to large and positive, hence the near vertical part at the beginning (and also at the end).

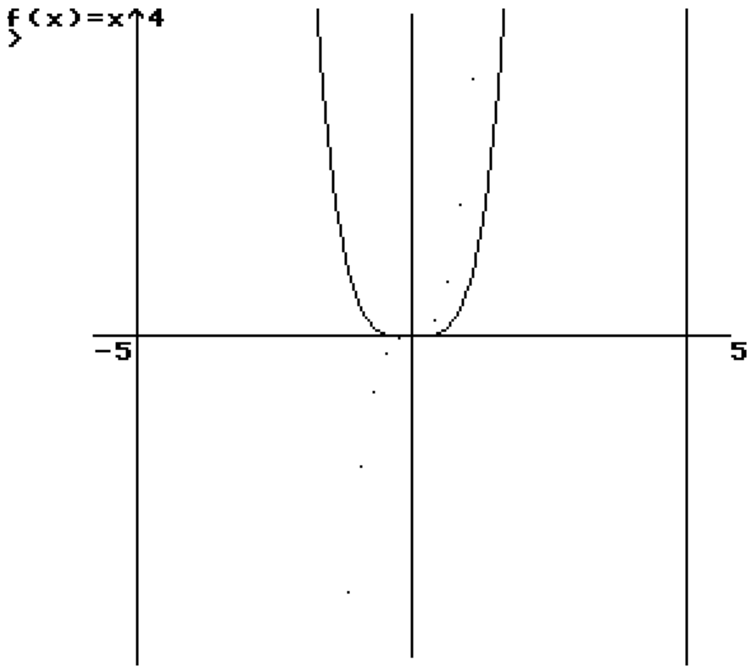


Figure 2

If the program is used for investigating slightly more complicated functions, it breaks down. For instance the function f(x)=1/x is not defined for x=0 and the program may crash when the graph reaches this point. By changing the step in lines 170 and 230 from 0.2 to, say, 0.2001, then the calculation at the origin will be avoided so that no crash occurs but then the "plot and join" method of drawing the graph will erroneously join the two separate parts of the curve on either side of the origin.

The function $1/x^2$ causes a more serious crash as the values for x near the origin get "too big" for the computers graphic coordinates. Other functions, such as the square root SQR, or the natural logarithm LN, are not defined everywhere in the given range, so the program fails even more comprehensively.

Of course, it is possible to patch up the programs, for instance one could truncate coordinate values so that they do not cause problems, by selecting a suitable maximum value M and replacing any larger value by M. To take care of positive and negative values one could use a function such as:

```
3000 DEF FNc(t): IF t<–M THEN =–M ELSE IF t>M THEN = M ELSE =t
```

Taking M to be a bit less than 32767 and replacing all y-values, such as `100*FNf(x)`, by the truncated values `FNt(100*FNf(x))` would give suitable screen coordinates.

But are such technicalities sensible, or even necessary, when one is using the program just to illustrate a specific mathematical point? It would be my contention that, without a considerable earlier investment in programming, it would be unwise to go to such lengths in the mathematics curriculum.

A recent survey for the Mathematical Association Working Party on "Using the Computer in the Secondary Mathematics Classroom" revealed that the typical secondary school in a sample of 52 schools had between 10 and 20 computers available, of which only 6% were reserved specifically for mathematical use. This works out at roughly 1 computer per 1,000 pupils for mathematics! As the school year amounts to a total of less than 1,000 hours, if all the computers assigned to mathematics were used continuously for mathematical programming, then each secondary pupil would have about an hour a year...

Perhaps computers could be used that are set aside for other purposes (for example the 50% or more in the computing laboratory). Even with these facilities added, an approach based on individual children having sufficient experience of programming is currently far from universally practicable.

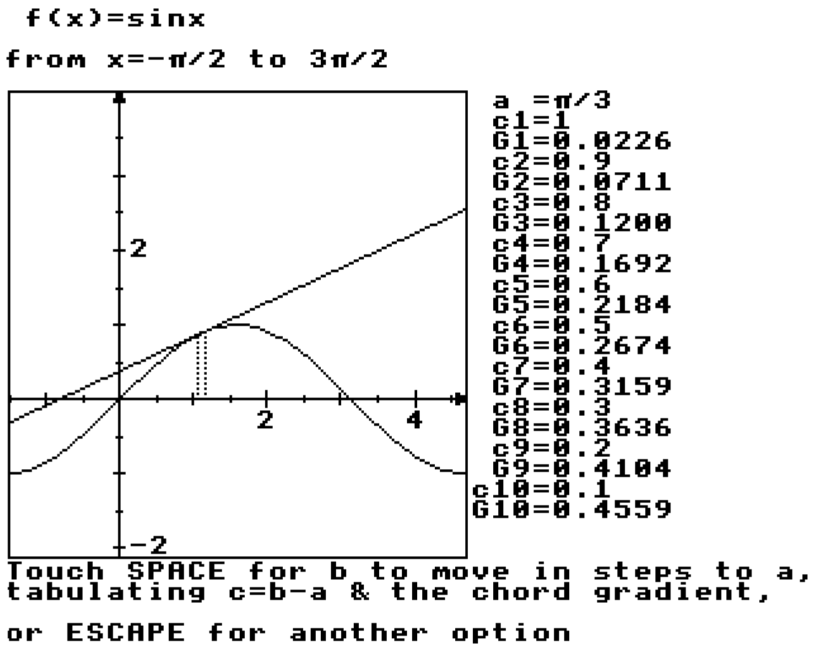
Limited computer resources may be used more effectively to investigate mathematical concepts through the use of specially prepared software designed for both classroom demonstration and discussion, and later for groups of pupils using it in small groups on a circuit system.

The Strengths and Weaknesses of Prepared Software

The deficiencies in short programs mentioned in the previous section can be avoided in more sophisticated software by careful programming. For example, options may be offered which format the numbers to a specific number of decimal places at the choice of the user, and graph-drawing procedures can have routines to truncate the screen coordinates, deal carefully with asymptotes and introduce error-handling routines to cope when the functions become undefined. The interface can be made more friendly by allowing input in familiar mathematical notation instead of the kind of notation required in computer languages. There may be sophisticated options that are designed to encourage exploration and help to give insight into the mathematical concepts.

For example, the program *Gradient* in *Graphic Calculus*, deals with the problem of the limit by having a default number format that shows four decimal places and then using a cleverly chosen limiting process. The chord is drawn through two points a,b, then recalculated as b moves in steps towards a, displaying the difference between a and b, and the numerical chord gradient. Each screenful of chord gradients starts a distance $c=b–a$ from a, then moves in ten equal steps $c/10$, ending up a distance $c/10$ from a (figure 3).

If the points a,b begin a reasonable distance apart, say $b=a+1$, then one can see the limiting process in action. The default of four decimal places usually means that the chord gradient “reaches” the numerical limit for a screenful of values, staying there possibly for another screen or so before the numerical errors cause the chord gradient to stray off course. In this way the student can get a feeling both how the chord gradient tends to a specific limit, yet pressing the values numerically too close will lead to errors. It is a relatively easy matter to accept that, when one divides two small numbers with given errors, the resulting error in the quotient can be quite large. In this way the student can be led to believe that the formal expression really does tend to a specific limit, and be given an insight into how the limiting process works numerically.



```
f(x)=sinx
from x=-π/2 to 3π/2
                                   a =π/3
                                   c1=1
                                   G1=0.0226
                                   c2=0.9
                                   G2=0.0711
                                   c3=0.8
                                   G3=0.1200
                                   c4=0.7
                                   G4=0.1692
                                   c5=0.6
                                   G5=0.2184
                                   c6=0.5
                                   G6=0.2674
                                   c7=0.4
                                   G7=0.3159
                                   c8=0.3
                                   G8=0.3636
                                   c9=0.2
                                   G9=0.4104
                                  c10=0.1
                                  G10=0.4559
Touch SPACE for b to move in steps to a,
tabulating c=b-a & the chord gradient,
or ESCAPE for another option
```

Figure 3

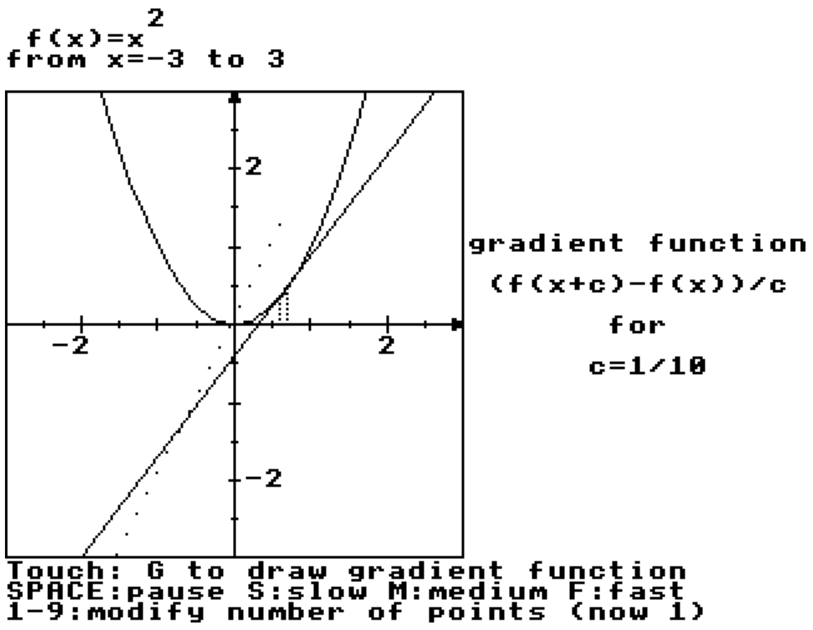
Figure 4 shows the first six screensfull of chord-gradients (the first taken from figure 3). The first column begins with a=π/3 and b=π/3+1, and successive entries down the column exhibit the gradient of the chord for fixed a to moving b, where the distance from a to b is first 1, then 9/10, then 8/10, ... , down to 1/10. The second column begins a=π/3, b=π/3+1/10, and the successive differences between a and b are 1/10, 9/100, 8/100, ... , down to 1/100. As b moves towards a, successive columns show the gradient of the chord moving towards the value 0.5000.

0.0226	0.4559	0.4957	0.4996	0.5000	0.5000
0.0711	0.4604	0.4961	0.4996	0.5000	0.5000
0.1200	0.4648	0.4965	0.4997	0.5000	0.5000
0.1692	0.4693	0.4970	0.4997	0.5000	0.5000
0.2184	0.4737	0.4974	0.4997	0.5000	0.5000
0.2674	0.4781	0.4978	0.4998	0.5000	0.5000
0.3159	0.4825	0.4983	0.4998	0.5000	0.4999
0.3636	0.4869	0.4987	0.4999	0.5000	0.4999
0.4104	0.4913	0.4991	0.4999	0.5000	0.5000
0.4559	0.4957	0.4996	0.5000	0.5000	0.5000

Figure 4

There are a couple of small errors in the sixth column, but these occur when b–a are 0.000004 and 0.000003, which is already so small that numerical errors would be expected to interfere. By changing the number of decimal places in the display, it is possible to see how the values get close to the limiting value before numerical error causes the result to stray off course. At the same time one may look at the formalism in simple cases, for instance the gradient function for f(x)=x^2, and show how the algebra gives the theoretical limit to which the numerical limiting process is an approximation.

Gradient also includes a much more valuable routine, now becoming well-known, which draws the extended chord through (x,f(x)), (x+c,f(x+c)) and plots the value of the chord gradient as a point, repeating the process for increasing values of x, thus building up a picture of the numerical gradient of the graph as a “gradient function”. (Figure 5.)



```
Touch: G to draw gradient function
SPACE:pause S:slow M:medium F:fast
1-9:modify number of points (now 1)
```

Figure 5

There is a facility in the program to superimpose a graph with a formula specified by the user, who can therefore conjecture the gradient function empirically and test it out practically before going through any formal manipulation to obtain the derivative. This kind of facility can easily be employed by a teacher with a single microcomputer for demonstration and class discussion, before passing the initiative to the pupils for exploration in small groups whilst the rest carry out paper and pencil exercises. The program can equally easily be used for pupil exploration from the beginning, when an appropriate number of microcomputers are available, as Jorj Kowzun describes in the materials accompanying the M.E.P. Secondary Mathematics Pack (Waddingham and Wigley 1985).

The value of prepared software is that it can represent a philosophy of approach which is based on research into how pupils actually learn. For instance, *Graphic Calculus* is the result of ten years work, first studying pupils' cognitive difficulties in handling the limit concept, then into a consideration of various approaches that might make the initial introduction of the ideas more intuitive (Tall 1986b). The result is a decision to base the idea of derivative not on the notion of a limit of the gradient of the tangent, but to visualize it as the *gradient of the graph itself.* This is done through first exploring a program *Magnify* that allows any graph (given in terms of standard formulae) to be drawn and then any portion of the graph may be magnified onscreen. Most standard functions have the property that small portions of the graph magnify to "look straight". Pupils soon learn to glance along the graph to see its changing gradient and form a dynamic mental image of the gradient function. In this way the ideas of the calculus take on a real meaning over and above the usual mechanical process of formal differentiation. Likewise the idea of integration is approached through seeing dynamic pictures of areas being calculated, and the solution of first-order differential equations is visualized as knowing the gradient dy/dx at any point (x,y), and simply sketching the solution curves which follow the given gradient directions (Tall 1986c, 1986d).

The philosophy of learning behind *Graphic Calculus* is therefore embodied in the programs and takes the learner from pre-calculus concepts through to fundamental insights in differentiation, integration and differential equations. But the programs are also written flexibly to allow individual teachers and pupils to use them in their own way. However, if used without any regard to the features of the computer environment, the classical analyst and the unguided learner may both find some peculiarities.

When a version of the program *Gradient* was reviewed by a teacher who did not have any notes, a number of apparent “bugs” arose. First the teacher tried to get the program to draw graphs under extreme circumstances to severely test the drawing routines. One graph attempted was $f(x)=x^{21}$ over the range $x=-10000$ to $x=100000$. Not being sure of the appropriate y-range, the teacher took the offered option for the computer to scale the picture. It responded with the error message “no graph”. The first “bug” had been found...

The problem here is that BBC BASIC only represents numbers in the approximate range $\pm 10^{38}$. The value of $(-10000)^{21}$ is 10^{84}, which is out of the available number range, and $100000^{21}=10^{105}$ is even more extreme. In fact, x^{21} is only defined in BBC BASIC for x in the approximate range –66 to +66, and this is only about 0.1% of the range requested by the teacher. To scale the graph the program actually calculates the function at 40 intermediate points. No wonder it failed to find a graph defined only in a thousandth part of the range!

Then the teacher tried to draw the graph of x^n for various values of n, including $n=100$, which fails to give a graph to the left of the origin. The reason for this was not obvious to me, until I tried the command:

```
FOR n=1 TO 100 : PRINT n, (–1)↑n : NEXT
```

This quite happily printed n and $(-1)^n$ up to $n=63$ but crashed at $n=64$ with the error message “log range”. What appears to be happening is that BBC BASIC calculates x^n as a product when n is a positive integer less than 64, but otherwise uses the standard formula

$$x^n = e^{\ln(x)n}$$

to calculate x^n using the natural logarithm and the exponential function. This formula issues the “log range” error when x is negative.

There were other criticisms of the program, but the one that seemed to cause the most problems was the claim that the program did not carry out the limiting process properly. The numerical gradient

$$\frac{f(x+c)-f(x)}{c}$$

appeared to be drawn alright for some values of c, but the complaint was that when c was taken to be small, the wrong picture was drawn. In particular the teacher tried the value $c=1/10^{10}$.

We saw earlier that in this case $f(x+c)$ and $f(x)$ are so close to nine decimal places that the computer usually records their difference as being zero. Only when x itself is small, so that $f(x+c)$ and $f(x)$ are represented in exponent notation as being different is a non-zero value given. The resulting graph of the numerical gradient is then zero away from the origin, with a tiny portion roughly correct near the origin !

The implication of this discussion is that these programs can be used to give great insight into the mathematical processes, provided that they are used sensibly. But if matters are taken to an extreme, then difficulties may occur. The only way that these difficulties can be understood is to have some idea of how the algorithms in the program are carried out, and this implies some understanding of programming.

Comparisons of programming & prepared software

Both programming by pupils and the use of prepared sofware can greatly enhance the development of mathematical understanding. Programming gives individual initiative to the pupil and a great sense of personal achievement in problem-solving, whilst good educational software can be designed to represent a philosophy of approach that helps the student gain insight into the mathematical processes through exploring the available facilities. But programming demands an overhead in time and effort to develop the expertise necessary for personal initiative in an environment which may have idiosyncrasies that cause difficulties to the beginner. And educational software needs to be designed carefully and used sensibly in a way which does not push it beyond its natural limitations.

In the future better programming environments will be designed that are more suitable for building and testing mathematical concepts, for the moment the available resources suggest the use of prepared software for teacher demonstration, class discussion and pupil exploration (not necessarily in that order), with a leavening of programming to gain an understanding of how the computer works and how it can be made to perform mathematical processes.

References

Mathematical Association 1985: *132 Short Programs for the Mathematics Classroom.* Stanley Thorne.

Tall D.O. 1985: ‘Understanding the Calculus’, *Mathematics Teaching,* 110, 49-53.

Tall D.O. 1986a: *Graphic Calculus I, II, III,* Glentop Publishers, London.

Tall D.O. 1986b: *Building and Testing a Cognitive Approach to the Calculus using Interactive Computer Graphics* , Ph.D. thesis, Warwick University.

Tall D.O. 1986c: ‘A graphical approach to integration and the fundamental theorem’, *Mathematics Teaching,* 113, 48-51.

Tall D.O. 1986d: ‘Lies, Damned Lies ... and Differential Equations’, *Mathematics Teaching ,* 115, 54-57.

Waddingham J. & Wigley (editors) 1985 : *Secondary Mathematics with Micros : Inservice Pack,* M.E.P.