Hidden Algorithms and the Drawing of Discontinuous Functions

David Tall (Warwick University, U.K)
& Bernard Winkelmann (Universität Bielefeld, Germany)

As computer software is increasingly used in mathematics, we are having to face the problem that we may not fully understand precisely what is going on. There is a genuine cause for concern that the hidden algorithms used by the software are deceiving us. We should therefore carefully distinguish different ways we may claim to "understand" an algorithm.

We will illustrate these difficulties by considering the problems encountered in designing graph-plotting programs. We will demonstrate a novel way of drawing graphs that (usually) shows a truer picture than most graph-plotters and use this example to suggest a categorization of different modes of insight into the operation of computer algorithms.

Difficulties with graph plotters

There are many function plotting programs on the market, but none of them actually plot the true graph of a function. Essentially they either plot individual points, which is very time-consuming, or they do the best of a difficult job by plotting points and joining them up with straight lines (or as near straight as the pixels on the screen allow). Each program has its own special ways of handling problems which may occur, sometimes they succeed, sometimes they fail. In normal circumstances they may give a good representation of the graph of the function but, pressed to extremes, they may not give the true picture. Figure 1 is a plot of the graph of y=sin(1/x) drawn using the FGP program[1].

Figure 1

Near the origin the graph oscillates infinitely often in the range –1 to 1 and the plotting routine fails to pick it up. Some graph plotters (such as Supergraph[2]) have the option of changing the step-length, so by taking a longer time, a more precise picture may be drawn. But there are other swings and roundabouts where Supergraph fails and FGP is decidely superior. For example, in drawing graphs involving known discontinuous functions, such as the "integer part function" INT (where INT(x) is the largest integer which does not exceed x) or the signum function SGN (where SGN(0)=0, SGN(x)=–1 for x negative and +1 for x positive), FGP scans the input string and cunningly tells the program to switch to point-plotting only, without joining. The result is that it draws a graph such as $y=1/\text{int}(1/x)$ rather well (figure 2), though it takes about 40 seconds to complete the picture.

Figure 2

Supergraph fails this test badly, joining up the separate parts of the graph, though it does notice that the function becomes undefined at $x=1$ and marks it with a vertical line of dashes (figure 3). A routine to cope with this difficulty (which does not occur with the standard functions in the school curriculum) has been omitted in the tiny BBC memory in favour of other options such as polar and parametric curves, and dealing more carefully with the peculiarities of more typical "school mathematics" functions such as the semi-circle $y=\sqrt{(1-x^2)}$.

Figure 3

An honest(?) plotting algorithm

It seems a straightforward matter to represent a graph on a VDU, given the limitation of the number of pixels. It is simply a matter of choosing a small enough step length and plotting individual points. The only problem is one of time. If the user is allowed to input the function, how does the program know how small a step to take? A simple method is to pass over the graph several times, the first time plotting the value at the mid-point only, then successively halving the step-length and plotting intermediate points. This doubles the density of points plotted on each pass and takes hardly any longer than plotting points with a fixed step. It has the advantage of giving the user the option to quit when the graph has been filled out sufficiently.

The algorithm to do this is extremely short. If the *x*-range is from *xl* to *xh*, the routine begins with a step equal to *xh*–*xl* then, for each pass across the screen, it starts at *x*=*xl*+*step*/2, calculating the value of the function, which we'll call *y*=FNf(*x*), scaling so that the point (*x*,*y*) is plotted as (FNX(*x*), FNY(*y*)) on-screen. The value of *x* is then increased by the step value, and the process repeated until *x* exceeds *xh*, when the step-length is halved and the procedure is repeated. This can be done in any appropriate language, but it is better if the language has suitable error trapping, to cope with places where the function is undefined. Though it looks ugly in BASIC, it is actually extremely efficient. The main algorithm looks something like this:

```
100   step=xh–xl
110   x=xl+step/2
120   ON ERROR GOTO 140
130   y=FNf(x): PLOT FNX(x),FNY(y)
140   x=x+step: IF x<xh THEN 130
150   step=step/2 : GOTO 110
```

(The *x*- and *y*-ranges must be specified, and the functions FNf, FNX, FNY need defining in a manner appropriate for the particular version of the language. Other details, such as the way that the on error routine works, or the keyword for PLOT may need modifying. On the BBC computer the command to plot the point *x*,*y* is PLOT69,*x*,*y*.) The routine works by calculating and plotting the function in line 130; if an error occurs in calculating the function, the routine jumps to line 140 and continues to attempt to plot later points.

This routine is rather successful and draws some nice pictures. For example, figure 4 is the graph of *y*=int(*x*) from x=–5 to 5 after 5 passes, and figure 5 shows the result after 10 passes, clearly filling out the necessary pixels on the graph without joining across the jumps in the value. In this case the drawing routine on the BBC takes less than a second for the first 5 passes, and gives a satisfactory picture, no different from figure 5 after 8 passes, in less than five seconds.

Figure 4

Figure 5

In extreme cases, where the curve is very long, a good picture may take a long time. For example, figure 6 shows the routine drawing $y=\sin(1/x)$, with an added line in the program to print the time in seconds after each pass. Fourteen passes have already occurred, taking nearly nine minutes, and the picture was dumped in the middle of the fifteenth pass, some nine minutes later. Very close inspection may just show that the points are filled in more densely to the left of the origin on the latest pass, but it is not that clear. It was not until after the middle of the eighteenth pass, nearly two and a half hours later that the gaps in the screen image of the graph were filled in. By this time the central part of the graph was just a solid block on the screen and could not be used to read off values in any practical way. The chief purpose of this kind of drawing is to give the user a clear qualitative impression of the graph.

Figure 6

Generalising the algorithm

The algorithm generalises easily to plotting parametric and polar curves. In the case of parametric curves where x,y are given by formulae $x=x(t)$, $y=y(t)$, it is simply a matter of successively bisecting the parametric range from $t=tl$ to $t=th$, and calculating x, y by functions, say $x=\mathrm{FN}x(t)$, $y=\mathrm{FN}y(t)$, to plot the point (x,y) on the screen as before. Likewise a polar plot $r=r(a)$ for radius r, angle a, can be drawn using an a-range $a=tl$ to $a=th$ and calculating $x=r(a)*\cos a$, $y=r(a)*\sin a$, to plot (x,y).

Figure 7 shows the graph $r=a^{1/2}-\mathrm{int}(a^{1/2})$ drawn using

FNr(*a*)=*a*^(1/2)–INT(*a*^(1/2))

FNx(*a*)=FNr(*a*)*COS(*a*)

FNy(*a*)=FNr(*a*)*SIN(*a*)

and modifying the main routine to:

```
100   step=ah–al
110   a=al+step/2
120   ON ERROR GOTO 140
130   x=FNx(a):y=FNy(a): PLOT FNX(x),FNY(y)
140   a=a+step: IF a<ah THEN 130
150   step=step/2 : GOTO 110
```

Figure 7

The picture has been dumped to the printer halfway through a pass, and careful inspection will show that the earlier whorls moving out from the origin have more points plotted than the later ones. In this way one may imagine the extra points dynamically filling in successive gaps.

Making a more friendly version

If one uses the program to explore, it soon becomes apparent that it would help to surround it with more helpful routines. For example, one might wish to write the input routines to allow evaluation of input, allowing expressions such as 2*PI, or one may wish to allow automatic scaling or friendly zooming-in routines to help explore interesting parts of the curve under magnification. We did this by introducing the routine into Supergraph. The advanced version of this program allows the number of steps in drawing to be specified, so we broadened the routine so that when a negative number $-n$ is used, the program interprets the command as a DOTplot with n passes. The input n=0 is interpreted as a potentially infinite number of passes, terminated by touching the ESCAPE key, though this is a luxury, because ‘infinity’ in this case is about twenty. Any larger number takes such an interminable time!

The routines were also added to the SuperZoom version of Supergraph which allows zooming in on cartesian graphs. The effects were striking. The graph of y=1/int(1/x) looks fairly ‘continuous’ at the origin, and zooming in gives the same kind of picture. (Why?) But if one zooms in at a nearby point in the form 1/n, say 1/10000, then an appropriate magnification reveals the jump that occurs at each such point. (Figure 8.)

Figure 8

Does it always tell the truth?

Now we have an algorithm, albeit a sometimes slow one, which seems to be straightforward to understand, can we be sure that it always tells the truth? Looking through it, everything seems to be in order. The function seems to be calculated correctly and plotted on the screen satisfactorily wherever it is defined. The only weakness seems to be in the coarse pixel plotting.

If we believe the algorithm without being sceptical we deceive ourselves. There remain a number of ways in which the program may give an unsatisfactory picture. For instance, there may be internal vagaries in the way in which the computer carries out its calculations, or there may be peculiarities in the way it does its plotting.

As an example, drawing the graph of $y=x^n$ for large values of n can give a computer apoplexy. The BBC computer copes with n up to 63, drawing the graph for x negative and positive, but for greater values of n it only draws the part of the graph where x is positive. It seems that BBC BASIC calculates x^n for positive integer n as if it were a repeated product, but for n greater than 63 it reverts to computing it using the formula exp(n*ln(x)). The latter fails for x negative. Thus if any unwary student investigates the shape of the graph of x^n, there may be surprises that were unforseen.

Computer plotting routines also have certain unexpected properties. For instance, many current micro computers, including two recommended for use in British schools (the BBC and the RML380Z), hold the screen coordinates as two byte integers. Each coordinate is therefore written as a 16-digit number in binary notation. The most significant bit, however, is used to indicate the sign. Therefore an integer up to $2^{15}–1$ is represented satisfactorily, but negative numbers between $–2^{15}+1$ and $–1$ inclusive are represented internally by adding 2^{15}. The crazy result is that if a number is calculated to lie in the range 2^{15} to $2^{16}–1$, the computer represents it in the same way as a negative number. Thus points way off the top of the computer screen are considered as being below it. Fortunately these will be offscreen and will not affect individual point plotting, but if one attempts to join up the points it can lead to the lines crossing the screen...

Another, slightly strange, feature of plotting onscreen occurs because the graph coordinates are held as integers, and therefore rounding must occur. This happens by taking the integer part of the calculated screen coordinate. Thus theoretical points just above a pixel boundary will be plotted within the pixel, but points just below will be rounded down to the pixel below. Some graphs which oscillate just above or below the axis (such as $y=x^2\sin(1/x)$ near the origin) may exhibit the phenomenon that when they get within a pixel of the axis, the oscillation may not show up in the same way above and below.

Understanding algorithms

We now come to the question:

> What insight into the underlying algorithms and procedures should the user of a program have, in order to understand and interpret the output of the program?

In the specific case of graph plotters, the question refers to the understanding necessary to interpret the drawings of the graphs. But the question is a much more general one. When working with computers and software, a full insight is never possible and may not even be desirable. Even the programmer relies on the belief that the commands in his program are interpreted in a sensible way, the machine-code buff must trust the circuits of the microprocessor, the hardware guru must rely on the integrity of the physics and chemistry of the silicon circuits, and even the scientist who understands the necessary properties of the silicon may now be working on such a specific level that it is difficult to relate this knowledge through the chains of relationships that link it to the broader issues of the higher level languages. (A similar impossibility of a thorough understanding is even true in pure mathematics, but that's another story!)

Of course, the user should be able to understand why some function plotters give connected graphs, even if the function to be drawn is discontinuous and, in a complementary fashion, why the DOTplot routine gives a connected graph for continuous functions if it runs long enough, even though it 'obviously' is only plotting discrete points. (Are you sure of this? Try some hard examples, e.g. f(*x*)=SGN(*x*)*(ABS(*x*)^(1/3)) which has a vertical tangent at the origin, provided one takes f(0)=0. Or, if a multiline function definition facility is available, it may be defined as

```
DEF FNf(x)
IF x<0 THEN = (–x)^(1/3)
IF x>0 THEN = x^(1/3)
 IF x=0 THEN =0
```

For this function, does the DOTplot routine always fill in to ‘look continuous’ near the origin?)

In order to discuss the kinds of insight into the function of programs we need some terminology, a simple language to describe insights into mathematical algorithms on computers or, more generally, into programs that represent mathematical processes in some sense. Since full insight is not possible, we regard these (to a certain degree) as black boxes which have to be illuminated through some kind of insight. We distinguish three different modes: specific insight, analogous insight and external insight, which are in a certain sense independent of each other. Each of these may also be realized at different levels of sophistication, from an overall global level down to more detailed specifics within the machine.

External Insight

Sometimes we can describe exactly the (mathematical) result of an algorithm, even when we have no idea how the algorithm is actually executed. For example, students may know what the square-root function on a hand-held calculator (or in a higher level language on a computer) does, even if they do not know any algorithm for effectively finding it for themselves: it finds the best approximation (in computer terms) for a number which, when squared, gives the original number back again. From this knowledge students can give valid interpretations of results using and relying on square-roots.

Here it is interesting to note that the numerical inaccuracies of most computer calculations make it difficult to describe precisely the mathematical result of more complicated procedures: the calculator or computer almost never gives the exact square-root of its input. If it did, it would be so much easier to handle.

Computer algebra software (such as muMATH^{TM}) is strikingly different in this respect, in that it deals generally with the formulae, making only appropriate algebraic simplifications, unless specifically requested for a numerical result. In this case external insight is often sufficient. Modern computer algebras can handle the algorithms of symbolic differentiation and integration. Differentiation is fairly straightforward. It simply uses the rules of differentiation to break the formula down in the standard manner. But integration is non-trivial. Early computer algebras used to take the usual problem-solving way of looking for integrations by parts, suitable substitutions and so on. These early programs seemed surprisingly stupid because they could not spot a simple substitution that might be obvious to the naked eye of someone with a little experience. It was often possible to get such symbolic manipulators to differentiate a function and then find that it was not clever enough to know the integral of the derivative... But now there is the Risch algorithm that guarantees to give the integral as a formula, if there is one. This algorithm works quite differently from the usual techniques of integration taught in schools and universities but, even if we have no idea how it works, we can interpret the results since we know (in theory) how to differentiate to check the accuracy of the result. To do so does not give us any more understanding of how the algorithm works, but it does give us confidence in the output of the algorithm in this specific case.

A more familiar example may be the multiplication of two (not too big...) whole numbers on calculators or computers. In this case we normally have additional insights since we not only know the mathematical properties of the the product, but also some efficient algorithm for calculating it for ourselves. Once more we are in a position to check the veracity of the computer algorithm in specific cases, though we can never be sure that it will always work. For instance, we should expect that 2**a* will always give the same result as *a*+*a*, but on the computer this may not happen. (On the BBC computer, if one uses integer arithmetic, with A%=2^31–1, then one gets different answers for 2*A% and A%+A%... (A ‘feature’ of the language, which is a polite term for a known ‘bug’.))

For an understanding of the graphs of functions as represented on a computer screen, an external insight – say on the basis of an understanding of the mathematical notion of a graph as a subset of the pairs of real numbers – is normally insufficient because the many discretizing, rounding, approximating and physical drawing processes involved make notions such as connectedness, or giving at most one value for any given argument, behave differently on the screen from the formal theory.

External insight into an algorithm is extremely valuable in mathematics and programming, often rendering a more specific insight less essential. There is less reason to worry about the black box character of, say, a BASIC command when the result of its action can be easily described. Only if such an insight is difficult to obtain because the mathematics is too complicated or not well understood, is more specific insight required. Deplorably, this is the case with most numerical algorithms, and consequently with graph plotters which use numerical algorithms to compute their pictorial output.

Analogue Insight

As mentioned earlier, we are often able to execute algorithms ourselves which are similar in essence to those of the machine. For example, we know how to multiply whole numbers, so we have some kind of understanding of the machine algorithm, even if it uses an essentially different algorithm internally. Since we have performed graph-plotting by hand ourselves, by calculating tables of values then drawing and connecting the points, we know of the short-comings and possible pit-falls of the method, even if at critical points the computer internally uses more complicated algorithms in its attempts to draw the graph.

In solving differential equations, we may know the simple algorithm taking a short step along the tangent, then recalculating and repeating the process. Or we may know the ‘improved Euler method’ which gets a better approximation to the solution curve by averaging out the tangent directions at the beginning and end of a short tangential step. In this case we are able to gain some insight into the possible pitfalls of more complicated methods, such as the ‘Fourth order Runge-Kutta’, which performs a more subtle averaging process in an attempt to get a better approximation to the solution.

It is analogue insight which is obtained when we study simple forms of algorithms in short programs in order to have a better understanding of the working of more sophisticated software tools which operate such algorithms in a more robust setting. For example, one might study the algorithms for calculating areas under a graph, prior to using a well-protected program which not only calculates the areas, but handles difficulties, such as the function becoming undefined in part of the range.

Specific Insight

Specific insight occurs when the user understands both the algorithm and the manner in which it is implemented in the program. As we have seen earlier this is usually understood at the level of programming in a specific language, with the internal detail of the workings of the computer taken for granted. It is this mode of insight that is desired when one gets a student to write short programs to understand how the algorithms actually work in practice, though understanding at the language level may sometimes be thwarted by specific ‘features’ of the language.

Conclusion

Looking deeply at the operations of the computer suggests that we must develop a healthy scepticism over what is happening in the software and deep within the inner workings of the machine. It is not wise to believe that computers always work in a totally predictable and understandable way, because features of the software implementation are always likely to produce odd results in extreme (or even not so extreme) cases. Even though specific insight into the working of computer algorithms may seem desirable, analogous insight at least gives a partial understanding of the algorithmic behaviour whilst external insight allows a valuable check on the veracity of the outcome. Software developers have a duty to inform the users of their programs of undesirable ‘features’, and software users should always be on their guard.

Notes

1. The FGP program is available from several sources for both BBC computer and the RML 380Z, 480Z. It is part of the ITMA "Micros in the Mathematics Classroom" available from Longman and also of the "Secondary Mathematics with Micros" pack available from AUCBE, Endymion Rd, Hatfield, Herfordshire AL10 8AU.

2. "Supergraph" is available for the BBC and BBC Master computers. It consists of a suite of programs for plotting cartesian, polar, parametric curves and implicit functions in two dimensions, together with surfaces in three dimensions. It is available from Glentop Publishers, Standfast House, Bath Place, Barnet High Street, LONDON EN5 1ED or, at a 40% discount, from Rivendell Software, 21 Laburnum Avenue, Kenilworth CV8 2DR.