NOTICES OF THE AMERICAN MATHEMATICAL SOCIETY

SEPTEMBER 1991, VOLUME 38, NUMBER 7

Providence, Rhode Island, USA

ISSN 0002-9920

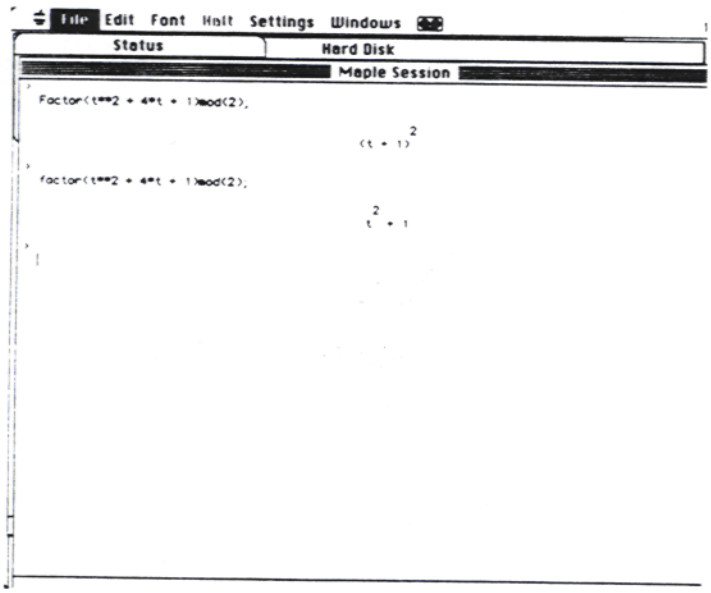
Computers and Mathematics

Edited by Keith Devlin

This month's column

One oft-repeated argument in favor of computer algebra systems is that they free the user from large amounts of tedious, messy, detailed calculations, where it is so easy to make a small slip that renders the result at best inaccurate and at worst, if real-world applications are involved, plain dangerous. But of course, just as the old pencil and paper approach carried its own risks of error, so too do today's high-tech methods. A slight mistake in typing and who knows what may result?

Take a look at the screen output from a *Maple* session shown below.



Spot the difference

At first glance this looks like a "proof" that

$$(t+1)^2 = t^2 + 1.$$

If you take a closer look you will see that there is, however, a difference between the two commands, namely one of capitalization. Readers familiar with *Maple* syntax will recognize what is going on here, but now imagine that, even though you are quite "expert" with *Maple*, you are in the middle of a long and complicated *Maple* session, streching over several screenfulls, and you inadvertantly type `factor` instead of `Factor`. *Maple* might well continue to respond obediently to everything you subsequently ask of it, and will gaily produce an answer for you. But it will, of course, be the wrong answer, and you might well be quite unaware of what was, after all, not a mathematical error but a typing mistake caused by not hitting the shift key at the right moment; a simple error in capitalization that does not cause the program to hiccup, but which renders meaningless the rest of the calculation.

This particular example is taken from the second of this month's two articles, both of which concern these all-purpose mathematics-by-computer systems that many of us are using these days. The first article comes from David Stoutemyer, who has been involved in the development of several such systems. He writes from the standpoint of the system developer, someone who knows some of the dangers and limitations involved in using such systems.

Following Stoutemyer's piece, Charles Livingston describes some work in knot theory he did with Jim Davis at Indiana University, using the computer algebra system *Maple*.

From the messages I receive, it is obvious that this column is widely read by an eager audience. So it is clearly not going to fade away for lack of readers. There are a great many people out there who are interested in the interplay between mathematics and computing. But, it can only continue if there is a steady supply of articles to print. Contributions are welcomed on all issues to do with computers and mathematics, particularly articles that deal with the use of computers in mathematical research. Just drop me a line if you have something you want to write about. My address is:

Professor Keith Devlin
Department of Mathematics
and Computer Science
Colby College
Waterville, Maine 04901

Correspondence by electronic mail is preferred, to:

kjdevlin@colby.edu

Crimes and Misdemeanors in the Computer Algebra Trade

David R. Stoutemyer*

University of Hawaii and Soft Warehouse, Inc.

As a co-author of the *Derive*[R] and *muMath*[tm] programs, a past contributor to *Reduce* and *Macsyma*[R], and an occassional user of some other computer-algebra systems, I am delighted about the rapidly increasing acceptance of computer algebra.

*David Stoutemyer is a professor of computer science at the University of Hawaii, Honolulu, HI 96822, and is the co-founder of Soft Warehouse, Inc., 3615 Harding Ave., Suite 505, Honolulu, HI 96816.

However, it is important for users to be aware of some of the limitations of such systems to use them wisely. First, some definitions:

A *bug* is something bad that a program does contrary to the programmers' intent—such as returning 5 for the result of $2+2$.

A *limitation* is something a program can't do that a user wishes it could. *Theoretical* limitations include undecidability results from logic and the theory of computation. The major *resource* limitation is exhaustion of computer memory due to the size of intermediate or final expressions. Another resource limitation is that a computation might take an unacceptable amount of time. *Algorithmic* limitations arise from the fact that it is impractical to coordinate all known mathematics and reasoning power in a reasonable-sized program using a reasonable-sized programmer staff for a reasonable length of time. For example, many computer-algebra systems cannot determine a closed-form antiderivative of $|x|^{-1}$, such as $\ln x \operatorname{sign} x$, even though these same systems can determine much more impressive antiderivatives. Such expertise gaps are inevitable.

Computer-algebra systems also rely on various *assumptions*, such as certain subexpressions being real or nonnegative. Unfortunately, the documentation or displayed results might not state relevant assumptions, and stated assumptions might be inapplicable in a significant portion of applications.

Experiment 1: Determine how the computer-algebra systems available to you and your colleagues simplify $\int |x|^{-1}\,dx$.

1. Bugs

Computer algebra programs tend to be large and complicated. Even when there are no known bugs in the most recent versions of these programs, most authors would admit that earlier versions had some bugs. The empirical evidence thus suggests that it is prudent to make sure that you have the current version and to act as if even the current version has bugs by checking results several ways.

Computer algebra systems don't necessarily have a higher rate of bugs per megabyte than other programs. Rather, based on past history, most programmers would agree that most large or complicated programs have bugs. Examples of large and/or complicated programs include many numerical methods packages, the compilers that compile them, the operating systems under which they run, the CPU microcode in chips on which they run, and the programs that were used to design the chips and computers. Don't despair. These systems are clearly useful despite their bugs. However, results obtained with the help of computers and programs deserve the same scrutiny as results obtained with help of slower devices such as calculators, tables, paper and pencil, or blackboard and chalk.

Be grateful when bugs are spectacular, such as producing obvious nonsense or crashing the program in a way that requires you to abandon the program and restart it or the computer. What we must guard against most is acceptance of an incorrect result because it superficially looks plausible and we are too gullible or lazy to check it.

One way to check a computer-algebra result is to derive it on more than one computer-algebra system. It is unlikely that two systems have the same bug. However, two systems might employ the same assumptions about branch selection, continuity, etc., and these assumptions might be inappropriate to your problem. Thus it is also important to read the documentation thoroughly to discover stated assumptions. Moreover, the experiments in this paper might reveal some of the unstated assumptions.

For example, you might discover that the default domain of variables is real numbers and that some of the corresponding transformations might be invalid if you substitute complex values after exploiting these transformations.

There is an additional benefit of trying more than one system: Even when the results are equivalent, you might find that different systems produce the most attractive form for various problems. You will also discover that most systems have features that are absent or weaker in others so that you will learn to exploit the best features of each system.

Many operations have an inverse, so an additional way to check a result is to use the inverse operation to see if you obtain a result that is equivalent to the input. It is unlikely that there are self-canceling bugs that transform an incorrect result back into a correct input.

As examples of such checks, invert an inverse matrix, expand a factored result, or differentiate an antiderivative.

This process often yields an expression that is not identical in form to the input: If the input is a form that would not be produced by any of the various simplification alternatives, then you might find it impossible to transform the inverted result to the same form as the input without tedious manual intervention. Consequently, it is usually best to try, instead, simplifying the difference between the input and the inverted result to 0. Computer-algebra systems are usually much better at simplifying to 0 expressions that are equivalent to 0 than they are at simplifying to a particular form an expression that is not equivalent to a rational number.

However, for the full class of irrational expressions allowed by most computer-algebra systems, it is probably impossible for any algorithm to guarantee simplification to 0 in a finite number of steps an arbitrary finite expression that is equivalent to 0. Thus, if the above difference does not simplify to an expression that you recognize as equivalent to 0, try substituting random numbers in the domain of interest for some or all variables and try to simplify that to 0. For irrational expressions, this usually requires approximate arithmetic and judging whether the residual is convincingly small compared to its components. A programming bug is likely to give a result that appears wrong for most random substitutions. Some design assumptions discussed below entail formula transformations that are invalid for subexpressions that are negative or have an imaginary part outside the interval $(-\pi, \pi]$, so be sure to include sets of such values if they are relevant. Other design assumptions

discussed below entail transformations that are invalid only at isolated points or curves in the complex plane, so these are unlikely to be revealed by random substitutions.

Another way to check a result is graphically. For example, Figure 1 shows a *Derive* screen in which window 1 in the upper left corner is a plot of expression 1 and window 2 in the upper right corner is a plot of expression 2. Expression 1 is a continuous antiderivative of $(2+\cos x)^{-1}$ computed by *Derive*, whereas expression 2 is a discontinuous antiderivative specialized from formula 4.3.133 in Abramowitz and Stegun [1]. As illustrated by this comparison, a plot is a quick and comprehensible way to check a result at several hundred sample points.

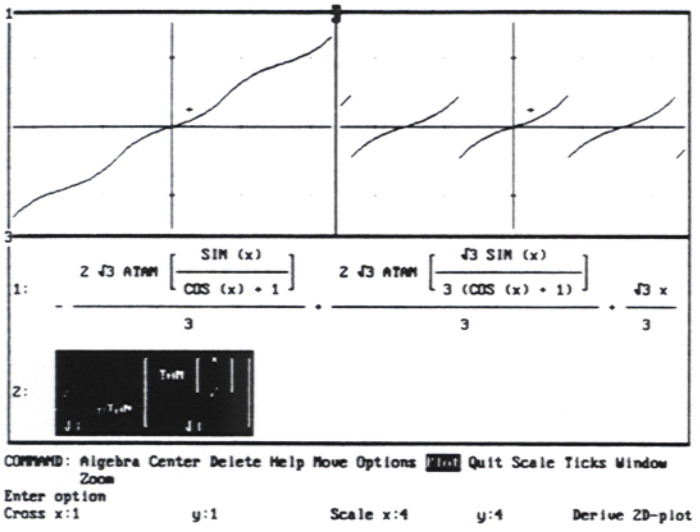


Figure 1: Alternative Antiderivatives of $(2+\cos x)^{-1}$

Experiment 2: For all accessible computer-algebra systems plot $(2+\cos x)^{-1}$ and its symbolic antiderivative to determine if the latter introduces spurious discontinuities. To see that most integral tables and textbooks are fallible too, plot an analogous antiderivative from each of the accessible tables and calculus texts. Don't be fooled if gullible plotting algorithms join discontinuities by spurious near-vertical line segments. This class of antiderivatives is discussed by Jeffrey and Rich [2], by Freese and Ortel [3], and by Kahan [4].

2. Theoretical and Practical Limitations

As implied by the preceding remark about the undecidability of recognizing 0, there are theoretical limitations to what is computable. Surprisingly, one of the theoretical difficulties is recognizing 0 in expressions composed of transcendental *constants* such as e^e or $\pi + e$. For most such compositions, it is unknown whether they are rational or irrational.

For example, suppose a computer-algebra system is requested to divide by the expression

$$e^{\pi\sqrt{163}} - 262,537,412,640,768,745.$$

Could it be 0? Six-digit and sixteen digit floating point arithmetic both yield magnitudes small enough compared to the two terms to be attributable to roundoff. Most computer-algebra systems provide arbitrary-precision approximate arithmetic. Consequently, such a system can keep increasing the precision until at, say, 42 digits it obtains a result such as

$$262537412640768744.999999999999250072597198,$$

which is strong evidence that the expression is not equivalent to 0. Arbitrary-precision *interval* arithmetic could enable the system to prove this rigorously by brute force. This well-known example has been proved irrational by more elegant means, but the point is that there is a general programmable way to prove that constant expressions are not equivalent to 0. The theoretical difficulty is that this method can never prove that an expression is equivalent to 0. If an expression really is equivalent to 0, such as $e^{i\pi}+1$, then the interval always contains 0, so the system keeps increasing the precision until it exhausts memory, thus aborting the proof.

In practice, a more important limitation on achieving explicit exact solutions is the expressability of polynomial zeros in terms of radicals: This is often impossible for polynomials of degree exceeding 4, and the exact solution of a general quartic is so unwieldy that it is better not to request it. Even the exact solution to a general cubic is fairly obscene. User's often don't appreciate the implications for solving one polynomial equation or a system of such equations exactly. For example, it is usually impossible to determine the exact eigenvalues and eigenvectors of a matrix exceeding 4×4 and impractical for a matrix exceeding 3×3 unless you are lucky. Similarly, it is usually impossible to obtain an exact explicit antiderivative of a rational function having a random fifth degree denominator.

Another frequent limitation on achieving explicit exact solutions is the size of intermediate or final results. For example, the expanded determinant of the $n\times n$ matrix having n^2 distinct letters as entries has $n!$ terms, each with n letters, for a total of about $(n+1)!$ symbols. The inverse of that matrix has n^2 entries, each with that determinant as a denominator dividing a numerator that has $(n-1)!$ terms, each having $n-1$ letters. Thus, the inverse of the general $n\times n$ matrix is not very useful for n beyond about 3 or 4, depending on your tolerance for lengthy, boring results that convey no insight.

3. Sets of Measure Zero

When asked to determine an antiderivative of x^k with respect to x, with k being an unrestricted variable, most systems respond $x^{k+1}/(k+1)$. Although this is incorrect for $k=-1$, the implementor might claim the justification that this is a set of measure 0 among all complex, real or integer exponents. However, a glance through mathematical literature reveals that an exponent of -1 is actually quite

common. Indeed, sets of measure 0 are often the focus of an analysis—for example, the zeros of $\det(A - \lambda I)$.

Some systems might pause to ask the user whether or not $k = -1$ before returning a result appropriate to the response. This is nice, but such queries are baffling when generated indirectly by an intermediate problem that bears no obvious relation to the user's input. For example, an ordinary differential equation solver might ask whether or not 1 equals a dummy variable that the user has never seen before. Moreover, such queries can be inconvenient when you want to be away from your computer during a lengthy computation. Consequently, some systems permit you to declare domain restrictions on variables before starting a sequence of calculations, using these declarations to determine if a candidate result is valid throughout the default and declared domains of variables.

If *Derive* cannot determine from the default and declared domains of variables that expression k excludes -1, then it returns $(x^{k+1} - 1)/(k + 1)$ for the above antiderivative, following a suggestion of William Kahan's. The limit of this expression is $\ln x$ as $k \to -1$, so the expression gives the correct result for any specific numeric value of k, provided you use the `lim` function to replace k by that specific number. Although users are likely first to *substitute* -1 for k giving $0/0$, this indeterminate form suggests that they then use the limit. In contrast, substituting -1 for k in $x^{k+1}/(k + 1)$ gives $x^0/0$, which simplifies to its limit $1/0$, representing complex ∞ rather than $\ln x$.

Experiment 3: Determine how all accessible systems treat $\int x^k\,dx$ both with and without a declaration that $k \neq -1$.

The above transformation $x^0 \to 1$ raises another issue about ignoring a set of measure 0: The cancellation of polynomial greatest common divisors. Most systems either automatically or optionally transform an expression such as

$$\frac{x^3 + 2x^2 + 3x + 2}{x^3 + 4x^2 + 5x + 6}$$

to $(x + 1)/(x + 3)$, thus gratuitously removing the removable singularities at the zeros of $x^2 + x + 2$. This reduction usually gives a more meaningful and concise result, but not always. An algebraist might regard the domain as a quotient field in which the unreduced and reduced expressions are equivalent. An analyst might feel differently.

It would be nice to have the option of simplifying this example to a conditional expression of the form

$$\text{if } x^2 + x + 2 \neq 0 \text{ then } (x + 1)/(x + 3) \text{ else } 0/0.$$

However, it would be challenging to simplify thoroughly combinations of such expressions as they grow through a long sequence of calculations. After seeing results laden with complicated provisos or after not obtaining results because of exhaustion of memory or patience, optimistic users might want to suppress this mechanism on all but simple problems. Fateman [5] discusses some experimental work of this nature.

If a system transforms x^0 to 1, then, for consistency between the algebra and arithmetic, the system should also transform 0^0 to 1. Kahan [6] and Graham, Knuth, and Patashnik [7] give different persuasive reasons why 0^0 should ordinarily simplify to 1 even in a system that does only arithmetic.

Experiment 4: Determine how all accessible calculators, numeric and symbolic programs treat x^0 and 0^0.

David Jeffrey pointed out to me another example of ignoring a set of measure 0 in solving an equation such as $cx = 0$ for x. Most systems return only $x = 0$, but if declarations don't exclude $c = 0$, then another solution is $c = 0$. Since we requested the values of x that satisfy the equation, we could express the solution set somewhat awkwardly as

$$x = \{\text{if } c = 0 \text{ then } @ \text{ else } 0\}$$

where @ is a unique new variable designating "anything". In contrast, there is less need to worry about the case $c = 0$ in solving $cx = 1$ for x, because the solution $x = 1/c$ contains a manifest indication of a limit solution at $x = 1/0 =$ complex ∞. It is the invisible failures of a formula that are most dangerous.

A similar issue arises in reducing to row-echelon form matrices having nonnumeric entries. What do we do when the only remaining pivot choices are symbolic expressions that don't exclude 0? For example,

$$\text{Row Echelon}\begin{pmatrix} 1 & 2 \\ k & 2 \end{pmatrix} = \begin{cases} \begin{pmatrix} 1 & 2 \\ 0 & 0 \end{pmatrix} & \text{if } k = 1, \\ \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} & \text{if } k \neq 1. \end{cases}$$

Corless, Jeffrey, and Nerenberg [8] devised a generalization of the LU decomposition that addresses this.

Experiment 5: Determine how all accessible computer-algebra systems solve $cx = 0$ and row reduce the above matrix example.

Most systems permit substitution of ∞, $-\infty$, or complex ∞ for a variable. Thus, to avoid canceling removable singularities at ∞, these systems could conservatively simplify $x - x$ to a conditional expression such as

$$\text{if } |x| \neq \infty \text{ then } 0 \text{ else } x - x$$

Although taking $x = \ln y$ makes $x - x \to 0$ and $y/y \to 1$ equivalent misdemeanors, it is safe to say that a system that didn't simplify $x - x$ to 0 would not be widely appreciated. However, a system that permits infinity as a constant should at least resist incorrect transformation of expressions involving that constant.

Experiment 6: Determine how all accessible systems document and automatically simplify $\infty - \infty$, ∞/∞, 0^∞ and 1^∞.

As suggested by the previous sentence, a system that documents assumptions is partially exonerated. The guilt

varies with the ease of locating such warnings in the documentation.

4. Verification Not Included

In order to construct solutions, humans and computer-algebra systems sometimes take steps that might introduce spurious solutions, so each of the candidates thus generated should be verified.

For example, squaring both sides of the equation $\sqrt{x} = 1 - x$ generates a related quadratic having two solutions, only one of which satisfies the original equation. Unfortunately, it is very hard to verify such candidates in general. For example, the cubic and quartic formulas can generate incredibly large and messy nested radicals or inverse trigonometric expressions, and when they are substituted into the original equation, no computer-algebra system can currently simplify the difference in the two sides to 0. For this reason, many computer-algebra systems make no attempt at such verification, leaving it to the user. Unfortunately, some of these systems do not state that warning in the documentation or display a warning whenever such a dangerous transformation is done. If there is a prominent warning, then perhaps we could categorize the weakness as an evasion of responsibility rather than a crime. It is better if a system makes an attempt at verification and issues a warning whenever a candidate could be neither verified nor rejected.

Experiment 7: Determine which accessible systems correctly solve the above example. Determine if those that return an incorrect solution warn the user either during the solution or in the documentation.

As another example, if a computer-algebra system can determine an exact antiderivative valid throughout an open interval, then the easiest way to determine a corresponding definite integral over that interval is to compute the difference in the limits of the antiderivative as the integration variable approaches the endpoints from within the interval. Unfortunately, it is impossible to guarantee finding and assessing the integrability of all internal singularities in the integrand even when there are no extra nonnumeric parameters in the integrand, and it takes a lot of sophisticated code to decide about a worthwhile percentage of examples that arise in practice. Again, many computer-algebra systems make no attempt at such verification, leaving it to the user, and unfortunately, some of the systems do not state that warning in the documentation and do not issue a warning whenever attempted verification is indecisive.

Experiment 8: Determine how the systems available to you and your colleagues document and treat

$$\int_{-3}^{2} x^{-2}\,dx \text{ and } \int_{a}^{b} (x-c)^{-2}\,dx.$$

A growing community of computer scientists believe that hardware and software should routinely verify every operation including those that should not generate spurious solutions. For example, every multiply would be checked by a divide. This would increase the program size and computing time, but the rapid increase in computer power should increase acceptance of this idea.

5. Branch Abuse and Sins of Omission

It is desirable for simplification to commute with substitution or limits. In other words, it is desirable for a transformed expression to give the same result as the original when numbers in the domain of interest are substituted for variables. Failing that, it is desirable that the limit of the original and transformed expressions should be equivalent as each variable therein approaches any number in the domain of interest. An earlier section described common violations of the first goal or both at sets of "measure zero".

This section describes commonly employed computer-algebra transformations that violate both goals for at least half the real numbers or for almost all complex numbers. If we designate the measure-0 offenses as misdemeanors, then these must be felonies. They all derive from a cavalier treatment of multiply-branched expressions.

Fractional powers, logarithms, and inverse trigonometric functions return only a single number on numeric calculators and numeric computer programs. Calculators and programs that do not support complex arithmetic might use the real branch for a fractional power of a negative number if the denominator if the exponent is odd. Otherwise, it is increasingly common to use the principal branch: $-\pi < \text{phase}(z) \le \pi$, $\pi < \mathcal{F}(\ln z) \le \pi$, $z^{m/n} = (z^{1/n})^m$, and $\text{phase}(z^{1/n}) = \text{phase}(z)/n$, etc.

Based on experience with calculators and numeric programs that return a single branch, most users expect and want the same behavior for numeric subexpressions on computer-algebra systems. For example, users expect $32^{1/5}$ to simplify to 2 rather than the set $\{2e^{2n\pi i/5} \mid n = 0, 1, 2, 3, 4\}$. Similarly, they expect $\ln(1)$ to simplify to 0 rather than the infinite set $\{2n\pi i \mid n \in \mathbf{Z}\}$. Built-in functions for solving equations might construct the full set from the principal branch when needed, and users are free to do so too if they wish.

A more compact way to represent all branches is to leave $\ln(1)$ as is and to transform $32^{1/5}$ only to $2 \times 1^{1/5}$, with both results representing all branches. There is merit to this approach, but it requires bravely forcing a massive attitude adjustment on users. I can imagine all the angry phone calls and letters asking why $\ln(1)$ doesn't simplify to 0 and $1^{1/5}$ doesn't simplify to 1.

Carrying the all-branch idea even further in the algebraic case, a computer-algebra system could use an implicit representation such as `ZerosOf` $(z^5 + z + 1)$ to represent a set of algebraic number associates that aren't expressible as radicals or nested radicals. Some systems have add-on algebraic number packages of this nature. However, users must take special care to prevent the mechanism from being sabotaged by the one-branch automatic simplification of the underlying systems. Moreover, these systems generally require users to anticipate and specify all of the algebraic extensions in advance as a single minimal polynomial rather

an let the extensions automatically accumulate separately only as necessary. For consistency, these systems also generally require the user to express even radicals such as the set $\pm\sqrt{2}$ in this implicit notation as `ZerosOf` (z^2-2). Note how I used $\pm\sqrt{2}$ to avoid the ambiguity about whether $\sqrt{2}$ designates all or one particular branch. No wonder confusion abounds!

We train students to seek *explicit* solutions as the holy grail, so such unnecessarily implicit results are doomed to poor acceptance. The desire for explicit solutions is so strong that most users prefer necessarily *approximate explicit* results to necessarily *implicit exact* results. Thus, it is doubtful that many users understand or appreciate these add-on algebraic number packages.

If fractional powers, logarithms, and inverse trigonometric functions return one particular branch for numeric arguments, then automatic transformations for nonnumeric arguments should also be valid for that particular branch. Otherwise, substitution and limits give different results when variables become numbers in the original versus transformed expressions.

For brevity, each felony described in this section is merely one example of a whole class of felonies.

5.1. Fractional-Power Felonies and Sins

Most computer-algebra systems automatically distribute fractional exponents over products or conversely collect such exponents without verifying that it is valid throughout the declared or default domains of contained variables when using the one particular branch used for numeric bases. As a more specific example, many systems employ one of the following two transformations even when both u and v could be negative:

$$(uv)^{1/2} \rightleftharpoons u^{1/2}v^{1/2} \tag{1}$$

For example,

$$((-1)(-1))^{1/2} \to 1^{1/2} \to 1,$$

whereas

$$(-1)^{1/2}(-1)^{1/2} \to i^2 \to -1.$$

Some systems shift guilt to the user by making these transformations optional, with the default being to avoid the transformations. This can be done with a control variable. For example, a variable named `TransformFractionalPowers` could have a default value of `false`, with optional settings of `collect` or `distribute`. Another way to provide such optional transformations is by extra transformation functions with names such as `CollectFractionalPowers` and `DistributeFractionalPowers`.

Either way, the default is then a sin of omission because the safe default is to exploit such transformations when they are valid. Otherwise, you can end up with bulky divisors such as

$$(|x|y)^{1/2} - |x|^{1/2}y^{1/2}$$

that are equivalent to 0.

Moreover, the choice between total distribution or collection of fractional exponents and no distribution or collection is too extreme. As the default, the transformations should be exploited where valid, and only there. For example, if x and y can be negative, then the default simplification can validly transform

$$\frac{(4xy)^{1/2}}{2x^{1/2}}$$

to $(xy)^{1/2}/x^{1/2}$, but not all the way to $y^{1/2}$.

There can be an override mechanism to force transformations even when the program cannot determine validity. If the default simplification is good, then users will rarely be tempted to gamble by overriding the default. In *Derive*, the override mechanism is to set the branch selection to `Any` rather than the default value `Principal`. There is also a `Real` choice that causes $(-1)^{1/3}$ to simplify to -1 rather than to the principal branch $\sqrt{3}/2 + i/2$.

Experiment 9: For the systems available to you and your colleagues, use automatic or optional simplification to validly transform $(xy|z|^2)^{1/2}/(x^{1/2}|z|)$ only to $(xy)^{1/2}/x^{1/2}$. Does each system treat `sqrt`(xy) the same as $(xy)^{1/2}$?

Many systems also automatically employ the transformation

$$(u^2)^{1/2} \to u \tag{2}$$

even when the phase of u could be outside the interval $(-\pi/2, \pi/2]$. For example, $((-1)^2)^{1/2} \to 1^{1/2} \to 1 \neq -1$. However, if the declared or default domains of variables make expression u real, then the left side can be simplified to u, which can further simplify to u if u is nonnegative. Although transformation (2) is a mathematically special case of transformation (1) in which $u = v$, the transformations are usually syntax based, hence separately programmed. Thus it is possible for a system to be guilty of one felony but not the other.

Experiment 10: For all accessible systems, determine if $(z^2)^{1/2}$ simplifies to z when z is declared real, then complex, then positive.

Another felony is the transformation

$$(1/u)^{1/2} \to 1/u^{1/2}$$

even when u could be negative. For example, $(1/-1)^{1/2} \to (-1)^{1/2} \to i$, whereas $1/(-1)^{1/2} \to 1/i \to -i$.

Experiment 11: For all accessible systems, see if $(1/x)^{1/2} - 1/x^{1/2}$ improperly transforms to 0.

Another related felony is to employ one of the transformations

$$(\exp z)^{1/2} \rightleftharpoons \exp(z/2)$$

even when z could have an imaginary part outside the interval $(-\pi, \pi]$. For example, $\exp^{1/2}(2\pi i) \to 1^{1/2} \to 1$, whereas $\exp(\pi i) \to -1$.

Experiment 12: For all accessible systems, see if you can correctly simplify $(\exp(iy))^{1/2} - \exp((iy)/2) +$

$(\exp x)^{1/2} - \exp(x/2)$ by cancelling only the last two terms, with x and y declared real. Beware that different systems use different notations for the imaginary unit and the base of the natural logarithms. If in doubt, use $(-1)^{1/2}$ and the exp function and beware of possible case sensitivity. Beware also of mixed notations: Does each system treat $\exp u - e^u + i - (-1)^{1/2}$ the same as $e^u - e^u + i - i$?

5.2. Logarithm Felonies and Sins

Another common felony is to exploit either of the transformations

$$\ln(uv) = \ln u + \ln v$$

even when both u and v could be negative. For example, $\ln((-1)(-1)) \rightarrow \ln 1 \rightarrow 0$, whereas $\ln(-1) + \ln(-1) \rightarrow \pi i + \pi i \rightarrow 2\pi i$.

Some systems avoid some logarithm felonies by requiring users first to write their own corresponding rewrite rule. This cleverly shifts blame for a naive transformation to the user. However, such systems are even more guilty of a sin of omission: An experienced system implementor has a better chance of implementing a valid rule than does a naive user, and we have seen that such transformations are vital to good simplification.

Experiment 13: For all accessible systems use appropriate automatic simplification, control variables, functions, or rewrite rules to simplify validly $\ln(xy/z|) - \ln|z| - \ln x$ only to $\ln(xy) - \ln x$. Beware of possible case sensitivity and that the natural logarithm is spelled log on some systems.

Another felony is to employ either of the transformations

$$\ln(u^2) = 2\ln u$$

when u could be negative. For example, $\ln((-1)^2) \rightarrow \ln 1 \rightarrow 0$, whereas $2\ln(-1) \rightarrow 2\pi i$.

Experiment 14: For all accessible systems use appropriate automatic simplification, control variables, functions or rewrite rules to simplify validly $\ln(x^2) - 2\ln x + \ln(|y|^2) - 2\ln|y|$ by canceling only the last two terms.

Another felony is to employ either of the transformations

$$\ln(1/u) = \ln u$$

when u could be negative. For example, $\ln(1/-1) \rightarrow \ln(-1) \rightarrow \pi i$, whereas $\ln(-1) \rightarrow -\pi i$.

Experiment 15: For all accessible systems, use appropriate automatic simplification, control variables, functions, or rewrite rules to simplify validly $\ln(1/x) + \ln x + \ln(1/|y|) + \ln|y|$ by canceling only the last two terms.

Another felony is to employ the transformation

$$\ln(\exp z) \rightarrow z$$

when z could have an imaginary part outside the interval $(-\pi, \pi]$. For example, $\ln \exp(3\pi i) \rightarrow \ln(-1) \rightarrow \pi i \neq 3\pi i$.

Experiment 16: For all accessible systems use appropriate automatic simplification, control variables, functions, or rewrite rules to simplify validly $\ln(\exp x)\ln(\exp(iy))$ only to $x\ln(\exp(iy))$, where x and y are declared real.

5.3. Trigonometric Felonies and Sins

Not many computer-algebra systems provide either automatic or optional half-angle simplification. Given the usual principal-branch implementation of square roots, valid half-angle transformations for all real x are

$$\sin(x/2) = \text{sign}(\sin(x/2))((1-\cos x)/2)^{1/2}$$
$$\cos(x/2) = \text{sign}(\cos(x/2))((1+\cos x)/2)^{1/2}$$
$$\tan(x/2) = \sin x/(1+\cos x)$$
$$\cot(x/2) = (1+\cos x)/\sin x$$

The sign factors severely restrict the applicability of the first two identities.

Experiment 17: For all accessible systems, use appropriate automatic simplification, control variables, funtions, or rewrite rules to simplify validly $\sin(x/2)/(1 + \tan x \tan(x/2))$ only to the equivalent $\sin(x/2)\cos x$.

5.4. Inverse Trigonometric Felonies and Sins

A common inverse trigonometric felony is to transform $\text{atan}(\tan u)$ to u even when u could be outside the domain $[-\pi/2, \pi/2]$, and similarly for the other inverse trigonometric functions outside appropriate domains. For example, $\text{atan}\tan\pi \rightarrow \text{atan}\,0 \rightarrow 0 \neq \pi$.

Experiment 18: For all accessible systems, use appropriate automatic simplification, control variables, functions, or rewrite rules to simplify validly $\text{atan}\tan x + \text{atan}\tan\sin y$ only to $\text{atan}\tan x + \sin y$ for unrestricted real x and y. Beware of possible case sensitivity and that atan is spelled arctan on some systems.

6. Concluding Remarks

You might be alarmed at the results if you make the recommended experiments. Good. The goal here is to inspire caution. These systems can be extraordinarily useful if users are aware of the underlying assumptions and of their responsibility to verify results. The same warnings apply to most software and hardware. For example, Kahan [9] describes surprises that should inspire similar caution for use of numerical calculators and programs. Manual computation is even more dangerous now that such skill is becoming a lost art.

There is not much that implementors can do about the theoretical limitations described in section 2, and no one has yet implemented a system that avoids all of the other above limitations. Some of them will be difficult to overcome. However, we are working on it, and some of the other implementors probably are too—at least after they try the above experiments. Be patient and enjoy what is already available, but make your desires known.

References

Abramowitz, M. and Stegun, Irene A., editors, (1965) *Handbook of Mathematical Functions*, Dover Publications, Inc., N.Y., p. 78.

[2] Jeffrey, D. J. and Rich, A. D., "Continuous antiderivatives of trigonometric expressions", preprint, University of Western Ontario.

[3] Freese, Ralph and Ortel, Marvin, "The $u = \tan\theta/2$ Substitution in Calculus and Computer Algebra Systems," preprint, University of Hawaii.

[4] Kahan, W., "Periodic Integrals vs. Prohibition of $\tan(\pi/2) = \infty$," preprint, University of California at Berkeley.

[5] Fateman, Richard J. (1990), "On the Systematic Construction of Algebraic Manipulation Systems," *Symbolic Computations and Their Impact on Mechanics*, editors A. K. Noor, I. Elishakoff, and G. Hulbert, The American Society of Mechanical Engineers, 345 East 47th Street, United Engineering Center, N.Y., N.Y. 10017, pp. 3–14.

[6] Kahan, W. (1987), "Branch Cuts for Complex Elementary Functions or Much Ado About Nothing's Sign Bit," *The State of the Art in Numerical Analysis*, editors A. Iserles and M. J. D. Powell, Clarendon Press, Oxford, pp. 165–212.

[7] Graham, R. L., Knuth, D. E., and Patashnik, O. (1989), *Concrete Mathematics*, Addison-Wesley Publishing Company, p. 162.

[8] Corless, R. M., Jeffrey, D. J., and Nerenberg, M. A. H., "The Row Echelon Decomposition of a Matrix," preprint, University of Western Ontario.

[9] Kahan, W., (1983) "Mathematics Written in Sand—the hp-15C, Intel 8087, etc.," *Proceedings of the American Statistical Association*, pp. 12–26.

Periodic Knots and *Maple*

Charles Livingston*
Indiana University

Jim Davis and I recently completed a project in knot theory that involved a significant amount of computing using the mathematical program *Maple*. Until then, our experience with computing had been limited to programming in *Basic*, *Fortran*, and *Pascal*, and neither of us was an enthusiast for the use of computers in pure mathematics. In this article, I will describe our experience with *Maple* and hopefully illuminate a role that programs such as *Maple* have in pure mathematics.

This article will not offer a detailed description of the capabilities of *Maple* nor a comparison with similar programs, such as *Derive*, *Macsyma*, or *Mathematica*. Barry Simon's recent article in *Notices* [S] is an excellent source for such information.

Our initial work was done using *Maple*, version 4.2, working both on a Macintosh Plus (with 1 megabyte of memory) and a Macintosh IIx. Once the calculations became more lengthy, we switched to a VAX computer. In writing this article, I have repeated some of our work using version 4.2.1 of *Maple* on a Macintosh IIx, and version 4.3 on a VAX. It is clear that these newer versions offer improvements (notably in the online help) that would have eliminated some of our difficulties and frustrations. However, it is fair to say that the experience would have been much the same no matter what version we had at the time.

*Charles Livingston is in the Department of Mathematics at Indiana University, Bloomington, Indiana 47405. He can be reached by email at livingst@ucs.indiana.edu

His article was based on research that was supported by the National Science Foundation.

Periodic Knots and the Alexander Polynomial

Figure 1 illustrates two knots in R^3. With a little effort, it is not hard to show that the second knot can be deformed to look like the first. Notice that the second diagram illustrates a "periodic" symmetry that is hidden by the first. Formally, a knot is called periodic, of period n, if it can be deformed into a position in R^3 so that a rotation of R^3 of angle $2\pi/n$ about an axis carries the knot back to itself. Hence, the second diagram of Figure 1 shows that the illustrated knot has period 2.

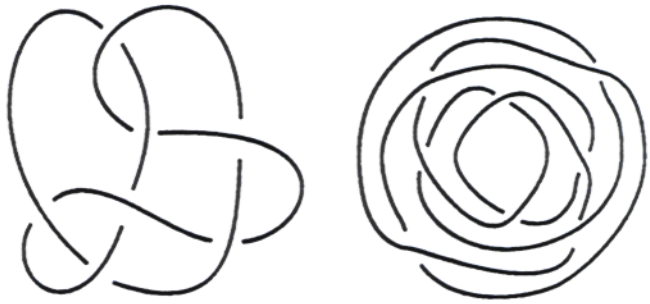


Figure 1

The problem of determining the periods of a knot is difficult, and it has repeatedly served as a testing ground for new methods of low-dimensional topology. The work of Thurston on hyperbolic structures, that of Meeks and Yau on minimal surfaces, and that of Jones on polynomial invariants of knots, have all seen application to the study of periodic knots; for instance, see [AHW, E, T]. In an article in this column of *Notices* [A], Adams described a computer program, *Snappea*, written by Weeks, that was used in [AHW] to study periodicity of knots.

My work with Davis was based on two results of Murasugi concerning *Alexander polynomials* of periodic knots [M]. There are many definitions of the Alexander polynomial of a knot; all that we need here is that to each knot K there is associated an integral polynomial, $\Delta_K(t)$, with the property that if J can be deformed into K, the polynomials $\Delta_K(t)$ and $\Delta_J(t)$ will be the same, modulo a multiple of $\pm t^k$; that is, $\Delta_K(t) = \pm t^k \Delta_J(t)$ for some integer k. (The converse, that if $\Delta_K(t)$ and $\Delta_J(t)$ are the same then K can be deformed into J, is false.) Tables of knots [BZ, R] include a listing of their Alexander polynomials, and there are now computer programs available dedicated to computing this and other knot invariants.

Murasugi's first result concerning the Alexander polynomial of a periodic knot states that if K is of period p^r, with p prime, then (modulo $\pm t^k$)$\Delta_K(t)$ factors as

$$\Delta_K(t) \equiv (\delta_\lambda(t))^{p^{r-1}} (\overline{\Delta}(t))^{p^r} \pmod{p}$$