The *Mathematica* Kernel: Issues in the Design and Implementation

How to build a software octopus in your basement, given 20 very smart monkeys and 20 years.

Daniel Lichtblau
danl@wolfram.com

Wolfram Research, Inc.
100 Trade Centre Dr.
Champaign IL USA, 61820

Macalester College,
November 2, 2006

Abstract

The *Mathematica* kernel is a large body of software that encompasses hundreds of person−years of in−house code as well as several large standalone libraries developed elsewhere. This makes for an endless assortment of possible problems, as well as offering many possibilities for future cutting−edge development. In this talk I will describe some of the issues that have arisen in my own work, and the ways we in the Kernel Group have gone about resolving them. I will also touch on some of the ways in which this complex software project plays off its strengths as it undergoes new development.

This talk is intended to be accessible to math and computer science students, and does not assume any prior knowledge of *Mathematica*. While specific examples will be presented, by and large the emphasis will be on generalities without too much technical detail.

The octopus of the subtitle is a metaphor for the multi−tentacled *Mathematica* kernel. The apelets come from the so−called "infinite monkey theorem". As they are very smart they require a mere four hundred monkey−years to do the job at hand (paw?). It is not yet clear what are the repercussions of mixing zoo and aquarium species in this cavalier manner.

Vastly abbreviated overview of the Mathematica kernel

What is it?

- ◇ The computational engine of the *Mathematica* program
- ◇ Parses and evaluates input as in an interpreted environment

What can it do?

Math

- ◇ Knows about hundreds of math functions (can evaluate numerically, simplify symbolically, etc.)
- ◇ Arithmetic (the usual, but in ways that are optimized for efficiency)
- ◇ Linear algebra (numeric, exact, symbolic)
- ◇ Polynomial and related algebra (gcds, factorization, various algorithms from algorithmic commutative algebra)
- ◇ Symbolic calculus (limits, integrals, ODEs, recurrence solving, summation, ...)
- ◇ Some number theory functionality
- ◇ Numeric algorithms from linear algebra, calculus, optimization, and elsewhere

Other things

- ◇ An extensive programming language (procedural constructs, pattern matching, functional programming constructs...)
- ◇ Graphics (uses math under the hood...)

The power of bootstraps (or: leveraging your own technology)

As might be surmised from the brief overview, it is sensible to implement substantial mathematical functionality using *Mathematica*. The practical implication for those of us who develop the code is that it is effective to write large amounts in *Mathematica* itself.

When I began at WRI in 1991 I think perhaps 20−30% of the kernel code was in *Mathematica*, with the rest in C. Now it might be 50% or so, excluding external libraries.

One advantage is that *Mathematica* is powerful language, and often one can do in a few lines what might take a few dozen in C. A disadvantage is that one does not have traditional debugging tools. The Wolfram Workbench, an Integrated Development Environment (IDE) product in beta testing, goes a long way toward filling this void.

Bootstraps and leveraging, more generally

As this is a talk primarily aimed at students who are largely unfamiliar with the technology workplace, I want to remark that the idea of leveraging of in−house technology is important and quite common. For example, when I worked on compilers and related technology for several years prior to graduate school, we had programs for representing annotated parse trees that were routinely ported from one project to another, with small project related tweaks along the way.

The lesson is that good ideas and code can and should be recycled. Of course, every company must then be on the lookout for its mistakes, because not everything is recyclable and ideas don’t come with those triangular number codes.

The power of libraries (or: Never do something yourself that someone else will do for you, and better)

The *Mathematica* kernel prior to libraries

Until version 5 appeared in 2003, the code for the *Mathematica* kernel was essentially all written and maintained by WRI. There was some amount of code from legacy libraries LINPACK and EISPACK, for numerical linear algebra, that had initially been translated from Fortran to C. I believe there was some polynomial algebra code likewise translated from existing sources. But all of it was subsequently modified in−house.

Libraries come to the *Mathematica* kernel

Early in version 5 development we planned to use LAPACK, the LINPACK/EISPACK successor, as an external library to which *Mathematica* would link. Reason: it takes advantage of processor−tuned efficient Basic Linear Algebra Subroutines (BLAS) which give as much as an order of magnitude speed improvement in machine arithmetic linear algebra.

As development progressed it became clear that the Gnu Multiple Precision (GMP) library was similarly more efficient for arithmetic of numbers with many (thousands or more digits). We made accomodations to use GMP in many places, thus effectively removing some of our arithmetic code.

Later we chose to encorporate specialized libraries e.g. for sparse linear algebra, in order to support some devel− opment elsewhere. Example: needed sparse linear solver for development of interior point code for linear programming.

When to use libraries?

When deciding wherther to use external libraries for certain functionality, we typically consider several questions.

◇ Does it do things *Mathematica* cannot do, or cannot do as well?

◇ What is the level of the library? State of the art?

◇ Are there licensing issues? If so, can they be resolved?

◇ What is the level of developer support? Do they fix their bugs?

◇ Do ported version exist for all platforms on which we support *Mathematica*?

The advantage of libraries (or: We should have thought of this years ago)

Once past the above hurdles, we find the typical advantages to be of the following related types.

- ◇ Better code than we could produce in−house in reasonable time
- ◇ We gain the benefit of dedicated expertise from external development
- ◇ Some projects such as LAPACK/BLAS are well funded and have ongoing development
- ◇ We don’t need to write/maintain it all ourselves

The curse of libraries (or: Remind me why we chose to rely on these people?)

A drawback to use of libraries is that the code base is no longer under our control. Hence *Mathematica* users inherit bugs that find their way into such libraries, and we are responsible for the problems caused thereby.

Examples

- ◇ Numerous bugs in machine linear algebra on various platforms

These are often isolated to vendor supplied BLAS or LAPACK routines. Some also have been found in sparse linear algebra libraries.

- ◇ A speed degradation caused by BLAS alignment needs

This was caught and fixed shortly before version 5 was released.

- ◇ A platform−dependent speed degradation caused by GMP reentrancy needs

This too was caught just before it could escape from the lab. Like the BLAS alignment issue it was resolved by proper setting of switches in building the library.

Attempts at resolution

Generally we try to isolate causes, send smallish examples to the library authors/vendors, and hope for the best. Among other things this implies that we actually try to maintain good working relationships with various external library purveyors, large and small.

In some cases we can also come up with workarounds in our own code. This involves adding code to detect failures in library code, and thereupon going to "plan B". Of course it also involves coding plan B.

How to extend technology? (Very carefully...)

A case study

In version 4 we unleashed our "packed array" technology. This is where lists of machine double precision real numbers, say, are maintained internally and processed as "raw" data structures containing tensors (vectors, matrices, etc.) of contiguous floating point numbers, instead of the usual *Mathematica* "lists" which are arrays of pointers to expressions, each of which might be a machine double.

The benefit to such representation is the ability to make optimal use of fast library code e.g. BLAS, and to make use of our own vectorized machine precision numeric functions, in order to gain competitive advantage in regards to speed of large scale numeric processing.

The down side is this had to be "seamless" insofar as we were not about to create new data structures and functions just for this purpose. So such lists need to be indistinguishable to the user from ordinary lists, except in speed of processing and memory hoofprint. That is to say, all functions from part extraction and assignment to numeric evaluation must look and act the same regardless of mode of internal storage. This is at it ought to be.

Implication to development

The important thing to realize is that it placed a huge burden on development and also has had repercussions that are felt in debugging to this very day. Specifically, I believe a problem reported to me by one "Stan Wagon" on October 21 of this year can be attributed to an obscure part of the packed array technology.

An example of *Mathematica* development using *Mathematica*

I will discuss in brief the history of some recent *Mathematica* functionality. I choose this because it is something I worked on, it is something I find interesting (not true of everything I work on, alas), and it has roots here at Macalester, with Stan Wagon’s work on Frobenius number algorithms.

In 2004 Stan visited WRI and talked about a Frobenius number algorithm. He also was working on a new one and had some questions that turned out to require integer linear programming (ILP) as an underlying algorithmic tool. As it happens, I had only months earlier worked on a different problem, computation of Keith numbers. But the underlying computations one does are quite similar.

Mathematica development example ...

What these problems require is a branch−and−bound or related loop, ability to solve relaxed problems using standard linear programming with rational or high precision arithmetic, and ability to solve linear systems over the integers. Also helpful and in some cases essential is some lattice reduction preprocessing. I was familiar with this from working on Keith numbers. Stan knew of relevant articles by Aardal, Hurkins, and A. Lenstra. Putting our knowledge together, I found some improvement Aardal et al had made over my formulation of the branching loop. We wre now on the way to handling "difficult" examples.

Upshot: I wrote more code to do these ILPs, Adam Strzebonski made further improvements and put them into some *Mathematica* functions, and now *Mathematica* can do some classes of difficult ILPs. This is only possible because we have the basic tools already: Hermite decomposition and lattice reduction of integer matrices, and linear programming. As a side benefit, we have nice built in code for solving Frobenius number problems.

Postscript (a further connection to Macalester)

John Renze, a Macalester graduate, worked at WRI on a new web site that, among other things, allows one to work with various physical units. This new ILP functionality allows *Mathematica* to readily find certain "minimal equivalents" to a given dimensional product of powers of units.

Another Mathematica development example

Numeric Gröbner bases

I will give another example of code I worked on that was possible only because of what the *Mathematica* kernel already had under the hood. Around 1994 I read an article about a prototype implementation of numerical Gröb− ner bases in a competing program. Now this is not a familiar topic to most people, but it is of some interest to people who do or rely upon computational algebra. As for working with approximate numbers, suffice it to say that, a dozen years later, there is still relatively little actually implemented in this realm. The biggest single prob− lem is recognizing when a number arising in the process of polynomial coefficient arithmetic is zero. But *Mathemat ica*'s significance arithmetic is quite good for this (it is a sort of approximation to interval arithmetic). I discussed this a bit with the late Jerry Keiper, who was responsible for the *Mathematica* implementation of significance arithmetic. I then went to my office, and a day and a few dozen lines of code rearrangement later, *Mathematica* had an implementation of approximate Gröbner bases. It is now used in functions such as `NSolve` for finding all solutions to a system of polynomial equations.

This is but one example of how *Mathematica* technology can and has been fit together to do what is nowadays known as "hybrid symbolic−numeric" computation.

A development story in progress

Symbolic summation

Recently we have devoted considerable resources to improving our symbolic summation capabilities. Among other methods utilized is table lookup. This involves "canonicalizing" inputs and then having reasonably fast lookup based on hashing.

Once this is near completion, joy of joys, we may get to do it again!! It is likely that we will then extend this to symbolic integration. The rough progression is this.

- ◇ We identify areas that might benefit from a given approach.
- ◇ We identify what capabilities we have to implement said approach

In this example hashing would play an important role.

- ◇ We identify what further work needs to be done

This might include canonicalizing inputs to keep the size of the table within reason, and perhaps handling of convergence issues.

- ◇ After putting togetehr the pieces, then see if outer parts of our technology base might benefit from the same or similar methods

If we can do sums this way, then why not integrals?

"Infinite evaluation"

We now discuss a mildly distressing problem buried deep in the *Mathematica* kernel.

Mathematica supports "infinite evaluation" (more or less...), wherein input is reevaluated until it stabilizes. The cases where this fails usually involve computations with "side effects" which, for purposes of this talk, are not of much interest. More problematic are some cases where it succeeds all to well. For example, *Mathematica* will attempt to crudely "canonicalize" products of powers, by splitting bases, merging exponents, and the like. One reason this is done is that it becomes easier to recognize mathematically equivalent expressions or common factors in expressions.

A drawback is that a power can be split into a product and sent on for further evaluation. The product handling code might then try to remerge the factors, leading to infinite recursion. While in practice this sort of thing is quite rare, there are pathological examples that we still need to repair.

As a primary cause of this problem I can only say that I hope to see it repaired in the not terribly distant future.

An invitation to infinite recursion

Transformations that reverse one another

A dark side to a complicated program like *Mathematica* is the ability to make trouble in subtle ways. One is to invoke pairs of transformations that undo one another.

Case in point: symbolic integration. This sits atop a tremendous amount of other functionality. One well known tactic for integration involves finding a partial fraction decomposition of a rational function. Grossly oversimplify−ing, this might be done as below.

```
i1 = Apart[input, x]
If[Head[i1] == Plus, result = Map[Integrate[#, x] &, i1]]
```

Another tactic is to accept "partial" results, that is, split integrands into summands that we can integrate and those we cannot. As it happens, in some cases it is also helpful to put a sum of terms over a common denominator. So imagine what happens if your code has transformations of the form:

```
i2 = Integrate[Together[pp], x]
```

where pp was generated, say, as a sum of pieces from the `Apart` call that have not been fully integrated, and all parts that have been successfully integrated happen to cancel. Then we might well find that the "togethered" thing gets us right back to input. And we go into recursion.

Inhibition of recursion

In some cases it is simply difficult to draw a line between infinite recursion and simply "trying real hard" on pieces of a problem. While still allowing for some amount of the latter, we attempt to forestall recursion in at least a few distinct ways.

Caching computations "in progress"

One is to rely on caching intermediate computations, in "canonicalized" form, so that later stages can attempt to "look up" results to see if we are reencountering a problem we have not yet finished processing (an obvious sign of recursion).

Depth charges

A more coarse approach is to use certain counters in order to determine approximately how deep we might be in some computation. If they get incremented too far we give up. This has some obvious drawbacks (if we stop too soon) but, in practice, seems to be reasonable.

Blocking reusage of parent callers

Still another thing is to "block" certain code from being used in certain places, on the supposition that it is likely to lead to recursion. Below is an abbreviated code snippet of this sort. We have a function `TableMatchTrig` that can call on `IntegrateNoTrig`. We wish to make sure the latter does not call back the former, and do so by putting it on a "stack" using *Mathematica*’s `Block` construct.

```
IntegrateNoTrig[h_, x_] := Block[{TableMatchTrig, res}, res = Integrate[h, x]]
```

Curiously, it is entirely coincidental that something named "`Block`" has the effect of "blocking" functionality of things in the manner indicated above: it was meant as the "block" of procedural programming languages.

"Finite" recursion

Even without getting into infinite recursion it is by no means difficult to have "deep" recursion. Here is a small example.

Example

```
a = {};
Do[a = {a, j}, {j, 50}]
a
```

```
{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{}, 1}, 2}, 3}, 4}, 5}, 6}, 7}, 8}, 9},
                                          10}, 11}, 12}, 13}, 14}, 15}, 16}, 17}, 18},
                                  19}, 20}, 21}, 22}, 23}, 24}, 25}, 26}, 27},
                         28}, 29}, 30}, 31}, 32}, 33}, 34}, 35}, 36}, 37}, 38},
             39}, 40}, 41}, 42}, 43}, 44}, 45}, 46}, 47}, 48}, 49}, 50}
```

Well and good. Now let’s remove the nesting.

```
b = Flatten[a]
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26,
 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50}
```

Upshot

That was a very silly way to emulate `Range[50]`. But that’s not the point. What matters is that in order to do the flattening the kernel code had to somehow traverse this expression. Prior to version 4 of *Mathematica* it would do this using recursive calls; for sufficiently deep nesting one could blow up the program’s recursion stack in this way. `Flatten` and several other functions now use their own stack to avoid this pitfall, but certainly there are cases we did not catch that might still cause trouble.

The moral, I think, is that we in the Kernel group at WRI should not rely too much on explicit recursion in our own code.

Code dependencies

We have described some ways in which implementation using the functions of *Mathematica* can lead to trouble. Here we discuss some perhaps more subtle points.

◇ Code now has hidden dependencies

When *Mathematica* calls on itself, in particular in ways that are not necessarily carefully thought out (recall what was said about unequal developers), there can be reliance on undocumented "default" behavior. For example, `Integrate` relies on `Apart` (the partial fraction function), but this is not well defined in cases where the input is not a rational function in an "obvious" set of variables. So when changes are made to `Apart`, it is necessary to check whether integrals will break. This is, to say the least, a regretable state of affairs.

Dependencies...

◇ Code may have dependencies on choice of variable names

We give a reasonably benign example that recently came to light. We begin with two expressions that are struc− turally identical. We then use `Together` on them and the results are no longer structurally the same (though they remain mathematically equivalent).

```
Clear[a];
e1 = -y^2 / 2 - (x^2 * C[1]) / 2 + ((I / 2) * uu * Sqrt[- (uu^2 * (1 + C[1]^2))]) / a - C[2];
e2 = e1 /. uu → u[x, y];
Together[e1]
Together[e2]
```

$$\frac{-a\,y^2 - a\,x^2\,C[1] + i\,uu\,\sqrt{-uu^2 - uu^2\,C[1]^2} - 2\,a\,C[2]}{2\,a}$$

$$\frac{-a\,y^2 - a\,x^2\,C[1] - 2\,a\,C[2] + i\,u[x,\,y]\,\sqrt{-\left(1 + C[1]^2\right)\,u[x,\,y]^2}}{2\,a}$$

This difference is due to vagaries of internal variable ordering and the way in which radicals get made into internal "variables" for functions such as `Together`. This then becomes an issue for funtions such as `Solve` that may rely on behavior of these utility functions. In the case that caused me to investigate the example above, `Solve` was giving different number of solutions based on whether we called a variable uu or *u*[*x*, *y*]. This is by no means wrong, but it is mildly unsettling behavior, and is indicative of complicated interactions between various com− ponets of the *Mathematica* kernel.

Allergic reactions in complex software

Quote

"In any sufficiently complicated mathematical program, there can be pairs or trios of design decisions that are individually quite sensible, but, in tandem, play havoc with one another." (I said that.)

Example that I think illustrates this

First, what should happen when we add a finite offset to a directed infinity?

```
infsum = ∞ + ⅈ π
```

```
∞
```

"That makes sense", he said.

Next, what should happen when we multiply exponentials?

```
expprod[a_, b_] := ⅇ^a ⅇ^b
```

Allergic reactions...

Symbolic case

Let’s try it on symbolic parameters.

```
expprod[a, b]
```

e^{a+b}

Check 1

Fine. Let’s check on our values above.

```
expprod[Infinity, I*Pi]
```

$-\infty$

Good.

Check 2

Now what if we substitute after we evaluate?

```
expprod[a, b] /. {a → Infinity, b → I * Pi}
```

∞

Oops.

Just imagine the headache of getting this to behave when finding a limit. It was only as of version 5 (2003) that *Mathematica* began getting this next one correct:

```
Limit[Exp[x + I * Pi], x → Infinity]
```

$-\infty$

New methods bring new problems

Progress

From time to time one learns of a new algorithm for doing a certain type of computation, say resultants of a pair of polynomials. One may read that the new method is vastly better than anything that came before it, and see, in the literature, examples to support that notion. Sometimes the new algorithms are easy to implement, and we do so.

Regress, or, What’s the down side?

Quite often it turns out that there are classes of problem for which the old method simply worked better e.g. faster or using far less memory. For example, when we implemented the Sylvester matrix method for computing resul− tants we found many classes of example for which the polynomial remainder sequence method worked better. This is not a surprise, and was not unexpected, but...

What to do to get the best behavior?

In this situation we must resort to various heuristics in order to decide upon the method to use for a given input. This is how any good polyalgorithm should behave.

So what is the problem?

The added problem is that we now have a "legacy" wherein, if we get it wrong on the method choice, we behave worse than in past even though we might rightly claim that we use "better" methods. This is the dilemma of having more choices.

Heuristics, more generally

The all−important quote

"Heuristics are bug ridden by definition. If they didn’t have bugs, then they’d be algorithms." (Author unknown)

Lessons

- ◇ Even good heuristics can only take you but so far
- ◇ Bad heuristics can take you further, but in the wrong direction

Similar issues in code "improvement"

If at first you don’t succeed...

Let us focus for a moment on one of my least favorite topics, symbolic integration. One approach to integrals, particularly indefinite ones, is to transform e.g. by converting trigs to exponentials. Sometimes we find that several different transformations may be useful for what seems to be a general class of inputs. Somewtimes we learn, to our further dismay, that we have no heuristics at hand to distinguish which inputs are amenable to which transfor−mations.

What to do?

What indeed? We may just try them in sequence. This means several things can go wrong.

- ◇ Adding a new transformation may give rise to a "worse" result for some inputs than in past
- ◇ Adding a new transformation makes all computations that try it, and fail, that much slower

Moral

"The more choices you have, the more opportunity for the unwary developer to really mess things up." (Author unimportant. Okay, it was again me who said it.)

Code "improvement"...

Fix me here...

In symbolic computation it is quite common to find that fixing one problem exposes weaknesses elsewhere. For example, say algorithm A decides to do operation X and, if it succeeds, operation Y, else Z. Say operation Y has six legs (as in "buggy"), but operation X returns a failure status so algorithm A goes on to do Z and gets a respectable result. What happens when operation X is improved so that it no longer returns a failure flag?

...break me there...

This is NOT a trick question. What happens is that algorithm A now uses defective procedure Y and messes up.

...or maybe go play hangman

Now suppose operation Y is not defective but simply slow? In this situation we have turned algorithm A into something that might hang rather than complete its task in reasonable time.

Code modularity

The ways in which various parts of *Mathematica* depend upon one another lead to certain undesirable outcomes. One is that the *Mathematica* kernel is not terribly modular. That is to say, it is difficult to extract various components e.g. "arithmetic" or "numerics" or "polynomial algebra", as there tend to be interconnections.

Legacy

As the kernel has been build up over many years this is even more difficult than might otherwise be the case, because in earlier times the need for modularity, clean "application program interfaces" (APIs), etc. were not so well understood, at least not by mathematicians.

On the bright side

Offsetting this I will mention a few happier points.

◇ The *Mathematica* kernel withstood the transition to the GMP library for arithmetic

◇ Has extensible numeric differential equation solvers

These, moreover, use some very nice modular code with carefully designed data structures.

◇ Has recently seen the addition of an extensible random number generator

◇ Has a huge test suite

This last means that when changes are made, if they have a bad impact in some part of the kernel far away this is often caught quickly in−house.

Infinite evaluation, redux

Getting the wrong computational complexity

We discuss another issue related to infinite evaluation and deep recursion. Say we have a list of *n* elements, and change an element. The semantics of *Mathematica* then require that the entire list be reevaluated, causing a complexity of $O(n)$ instead of constant time. There are various tactics used by the *Mathematica* kernel in order to decide when this might be avoided. Well and good.

Now suppose we have an expression we did NOT change. We still need to assess that this is the case in order to determine that it does not require reevaluation. We might do this by walking through it, checking each "node" to see if it is up to date. This works well, but such a walk can be as expensive as the reevaluation itself; for a deeply nested structure this can be a significant hit. So we need an inexpensive way to bypass this recursive walk through the expression.

Example

Here is an example of the phenomenon I describe. We nest an expression to a depth of 5000.

```
t1 = First[Timing[Module[{weirdness, L}, L = weirdness[];
    Do[L = weirdness[L, i], {i, 5 * 10 ^ 3}]]]]
```

```
0.48003 Second
```

Infinite evaluation, redux

Now we double the size, nesting to a depth of 10000.

```
t2 = First[Timing[Module[{weirdness, L}, L = weirdness[];
    Do[L = weirdness[L, i], {i, 10^4}]]]]
```

```
1.88412 Second
```

Note that in doubling the size, the timing goes up by a factor quite close to four.

```
t2 / t1
```

```
3.925
```

I hate when that happens

This is the classic sign of quadratic complexity, in a situation where we expect it to be linear. The culprit is the kernel code that assesses the need to reevaluate the nested expression at every step. It is often thwarted but not in this instance (due to an unfortunate internal hashing collision).

I have put some time into this issue and have experimental code that seems to improve the situation for bad examples that have come my way. While it is not yet ready for release, this, too, is a problem I hope to see largely repaired in a future version of *Mathematica*.

Speed vs. correctness

Symbolic computation should never allow for intentional mistakes!

One might suppose: "This is not a numeric program, ergo it should do symbolic computations without allowing the possibility of intentional mistakes."

Wrong, wrong, wrong

There are many places where such a program must do things in a way that is generally reliable but, even more importantly, fast. Case in point: determination of whether an expression is zero.

Examples

- ◇ Pivot selection in symbolic row reduction
- ◇ Inverting the leading term of a symbolic power series
- ◇ Solving a system of exact nonlinear equations

Speed vs. correctness...

The common theme

These all need functionality that will decide whether a symbolic, or perhaps exact numeric expression is zero. Such functionality is not infallible. Even for classes where it might be, requirements of speed dictate that it should not be. Otherwise it becomes even easier than it already is to hang the program on a seeminly simple computation.

Here is an example that gives a mathematically incorrect result of 1, because of a failure to discern that a factor in an exponent is positive rather than zero.

```
Limit[Exp[-x * (Sqrt[2] + Sqrt[3] - Sqrt[5 + 2 * Sqrt[6]] + 10 ^ (-1000))], x → Infinity]

1
```

In the next release of *Mathematica* this will at minimum give a warning to the effect that a possibly flawed numeri− cal assessment was made to the effect that some intermediate expression was zero. While this particular exam− ple might improve over time, the general problem will not go away: some zero test assessments will give mathe− matically incorrect results.

Speed vs. replicability

Caution:

This next topic scares people. Take a deep breath.

Constraining an evaluation

In order to prevent code from hanging it is sometimes necessary to rely on time constraints. The flow of a computa tion (e.g. a symbolic integral) is then influenced by whether a particular subcomputation does or does not finish in under the allotted time limit.

Obvious consequence

◇ The same computation may follow different paths on different machines, or even on the same machine but under different work loads

Slightly less obvious consequence

◇ Results, both good and bad, may be sporadic and difficult to replicate

Still less obvious consequence

◇ A faster machine can be much slower for a given computation

How?! Like so: Algorithm A tries subcomputation X. If it succeeds in under its time constraint, it goes on to do Y, else Z. If Z is, for the particular problem at hand, much faster than Y, then the faster machine that completes X is now herading into the tarpit of Y while its slower cousin zips instead through Z.

Speed vs. replicability...

Possible way to improve this situation

◇ Use a measure based on operation count that is independent of platform and processor

Even this will not be impervious to history, that is, effects of caching intermediate results in prior computations. But at least it would be platform−independent. A drawback is it is not easy to implement and not easy to tie to reasonable estimates of time of a computation on an "average" processor.

Another, quite unobvious issue with use of such constraints

Computations that must end due to time or similar constraints are aborted. Any "change of state" (e.g. temporary resetting of an option to some function) during the computation must then be restored by the caller. This can be tricky when coding in *Mathematica* and there are nested uses of `TimeConstrained`. Example:

```
answer = TimeConstrained[
  computation1;
  result = TimeConstrained[
    computation2[SomethingThatChangesState], 30];
  FixStateIfAborted[result];
  result
  , 10]
```

If the inner `TimeConstrained` invocation is aborted then the state altered by `computation2` will be restored. If instead the outer one is interrupted, and if, unlike this simple example, it is calling from another planet rather than just outside the door, it may not realize there could be a state change to repair.

There is a function in *Mathematica* called `CheckAbort` that, I believe, is meant to be used for such scenarios. But...I don’t know how it works, or whether it can really handle issues with nesting of `TimeConstrained`.

Speed vs. replicability...

A last issue with use of such constraints

As noted above, computations that end due to time constraints do so with aborts. How does the *Mathematica* kernel handle aborts? Well... they are caught by C code scattered about that checks a dedicated interruption flag. This then "bubbles up" from procedures on the calling stack. Callers have the obligation of cleaning up before moving to the next caller upward; failure to do so is a cause of memory leaks. A failure to check for an interruption can have the even worse consequence of, say, trying to dereference pointers in a result expected to have several elements, when instead what is received is an abort expression. This is a cause of crash−on−abort bugs. While not terribly common, they certainly do exist, and code such as symbolic definite integration that relies heavily on time constraints will exhibit this trouble on some hard inputs.

Scaling an octopus

Over time the functionality, and corresponding code base, of the *Mathematica* kernel has grown considerably. This creates several burdens.

◇ Design consistency must be maintained

This is vital because design consistency has been the backbone, I believe, of the commercial success of *Mathematica*.

◇ New functionality must be useful

It is easy to add new things. It is not so easy to add useful new things.

◇ New functionality must be robust

A small subset of my colleagues may disagree, but in general I find that neat bells and whistles do not win friends if they behave badly e.g. cause the program to crash.

◇ New functionality should not detract too much from our capability to repair/improve/maintain old functionality

There are endless opportunity costs involved in deciding on what needs to be added, what should be rewritten, what needs debugging, etc.

Scaling...

- ◇ Interactions with existing technology may cause trouble

New methods allow us to make use of them elsewhere. An interesting problem then arises. Code that "worked" in past, but using perhaps non−robust technology, may begin to fail e.g. because the new methods are too slow, or have bugs, or are being extended beyond their appropriate "operating range" (e.g. relying on a polynomial solver to handle transcendental equations), or ....

- ◇ Other issues:
  - ◇ **New functionality requires documentation!!!**
  - ◇ **Larger memory footprint of the program (what happens Congress revokes Moore’s Law?)**
  - ◇ **Larger code base implies a larger burden of maintaining legacy code**

So how well does the *Mathematica* kernel scale as it grows? Thus far I think remarkably well. But as every stock prospectus will caution:

- ◇ Past performance is no guarantee of future returns

Summary

Issues

- ◇ We covered several problematic areas in the implementation of the *Mathematica* kernel...
- ◇ ...but most areas are not burdened by such issues...
- ◇ ...and even the exceptions tend to work fine for most purposes, so...
- ◇ ...the outlook far less bleak than one might think

Down the road

- ◇ Over time, some of the problematic areas actually get improved
- ◇ Other ideas, methods, functions get developed that make some problem areas less relevant
- ◇ We are at the crossroads between voodoo and science

This makes for interesting times in computational mathematics. It’s anyone’s guess how far we will go, or to what extent voodoo can be replaced, if not by science, then at least by something more deserving to be called "art".

My motto

"I write the legacy code of the future."