

Comments on the *Mathematica* Tutorial: Why You Do Not Usually Need to Know about Internals

Most of the documentation provided for *Mathematica* is concerned with explaining *what Mathematica* does, not *how* it does it. But the purpose of this is to say at least a little about how *Mathematica* does what it does.

"Some Notes on Internal Implementation" gives more details.

You should realize at the outset that while knowing about the internals of *Mathematica* may be of intellectual interest, it is usually much less important in practice than you might at first suppose.

Indeed, one of the main points of *Mathematica* is that it provides an environment where you can perform mathematical and other operations without having to think in detail about how these operations are actually carried out inside your computer.

This assertion is weakened by the fact that the details of FullForm must be observed if you are to do operations such as applying Rules.

Thus, for example, if you want to factor the polynomial $x^{15} - 1$, you can do this just by giving *Mathematica* the command `Factor[x^15 - 1]`; you do not have to know the fairly complicated details of how such a factorization is actually carried out by the internal code of *Mathematica*.

You do, however, have to know how `Sqrt[-1]` or `I` is represented in order to use a rule to change its sign. `I->-I` will only work sometimes.

Indeed, in almost all practical uses of *Mathematica*, issues about how *Mathematica* works inside turn out to be largely irrelevant. For most purposes it suffices to view *Mathematica* simply as an abstract system which performs certain specified mathematical and other operations.

Ideally this would be true. In fact it is not.

You might think that knowing how *Mathematica* works inside would be necessary in determining what answers it will give. But this is only very rarely the case. For the vast majority of the computations that *Mathematica* does are completely specified by the definitions of mathematical or other operations.

Of course all the computations are defined by what it does, and if the specifications were complete enough, the detailed implementation would not, in most cases, matter. In fact, the written specifications of *Mathematica* tend to be, in some critical places, incomplete. In other places the specifications (written or not) are either controversial as a choice for conventional default mathematics, or in outright conflict with the usual interpretations.

Thus, for example, 3^{40} will always be 12 157 665 459 056 928 801, regardless of how *Mathematica* internally computes this result.

On the other hand, $3^{40} + 10000.5$ is apparently equal to 3^{40} , and how would you know that?

There are some situations, however, where several different answers are all equally consistent with the formal mathematical definitions. Thus, for example, in computing symbolic integrals, there are often several different expressions which all yield the same derivative. Which of these expressions is actually generated by `Integrate` can then depend on how `Integrate` works inside.

Here is the answer generated by `Integrate`.

```
Integrate[1 / x + 1 / x^2, x]
```

$$-\frac{1}{x} + \text{Log}[x]$$

This is an equivalent expression that might have been generated if `Integrate` worked differently inside.

```
Together[%]
```

$$\frac{-1 + x \text{Log}[x]}{x}$$

In numerical computations, a similar phenomenon occurs. Thus, for example, `FindRoot` gives you a root of a function. But if there are several roots, which root is actually returned depends on the details of how `FindRoot` works inside.

This finds a particular root of $\cos(x) + \sin(x)$.

```
FindRoot[Cos[x] + Sin[x], {x, 10.5}]
```

```
{x -> 14.9226}
```

With a different starting point, a different root is found. Which root is found with each starting point depends in detail on the internal algorithm used.

```
FindRoot[Cos[x] + Sin[x], {x, 10.8}]
```

```
{x -> 11.781}
```

The dependence on the details of internal algorithms can be more significant if you push approximate numerical computations to the limits of their validity.

Thus, for example, if you give `NIntegrate` a pathological integrand, whether it yields a meaningful answer or not can depend on the details of the internal algorithm that it uses.

`NIntegrate` knows that this result is unreliable, and can depend on the details of the internal algorithm, so it prints warning messages.

```
NIntegrate[Sin[1 / x], {x, 0, 1}]
```

```
NIntegrate::slwcon:
```

```
Numerical integration converging too slowly; suspect one of the following: singularity, value of the integration is 0, highly oscillatory integrand, or WorkingPrecision too small. >>
```

```
NIntegrate::ncvb:
```

```
NIntegrate failed to converge to prescribed accuracy after 9 recursive bisections in x near {x} = {0.0053386}.
```

```
NIntegrate obtained 0.5038782627066661` and 0.0011134563535424439`
```

```
for the integral and error estimates. >>
```

```
0.503878
```

Traditional numerical computation systems have tended to follow the idea that all computations should yield results that at least nominally have the same precision.

I don't know where this comes from. Perhaps it is a clumsy way of saying that traditional numerical computation systems generally rely on the arithmetic that is supplied by the computer hardware, and this often comes in a few floating-point formats, typically called single precision, double precision, and perhaps extended precision. Often good traditional programs will return a number in double-float precision in a printable format of (say) 18 decimal digits, and some accompanying information that says how much error there is (at most). Thus a very good algorithm might return an answer that is correct to all digits or all but the last digit, claiming excellent "relative error" of "at most one unit in the last place". Another way to return an error bound is as an "absolute error" which says that the computed answer is correct within a certain absolute bound. This latter form is better if the true answer is zero, but hard to prove... any non-zero answer would be "wrong in the leading digit".

A consequence of this idea is that it is not sufficient just to look at a result to know whether it is accurate; you typically also have to analyze the internal algorithm by which the result was found.

This is quite false. Typically all you need to look at is the specification of the (generally well-tested and exhaustively analyzed) library routine. This will generally state bounds on error, and perhaps go further to describe circumstances in which errors may be high or low. Sometimes, as indicated above, the program will compute, along with the result, a bound for the error in that particular result.

This fact has tended to make people believe that it is always important to know internal algorithms for numerical computations. It is hard to know which people are being identified here. Perhaps the author of this document? I think that people who ask for the internal algorithms have many motivations, but mostly curiosity.

But with the approach that *Mathematica* takes, this is rarely the case. For *Mathematica* can usually use its arbitrary-precision numerical computation capabilities to give results where every digit that is generated follows the exact mathematical specification of the operation being performed.

Rarely? How is one to know if the particular computation of interest is one of the cases in which the digits are not correct? And how is one to overcome the situation in which *Mathematica* declares that the digits are uncertain, even though they are in fact correct? (You can, but by knowing about internals.) So the argument is turned on its head: you must know about *Mathematica* internals just in case you are computing in some (unspecified) rare case, whereas with conventional libraries you could look at the specifications and the computed error bounds and have some confidence that you are within the parameters of the application of the library program.

Even though this is an approximate numerical computation, every digit is determined by the mathematical definition for π .

`N[Pi, 30]`

3.14159265358979323846264338328

Once again, every digit here is determined by the mathematical definition for $\sin(x)$.

`N[Sin[10^50], 20]`

-0.78967249342931008271

If you use machine-precision numbers, *Mathematica* cannot give a reliable result, and the answer depends on the details of the internal algorithm used.

This is a misuse of the term reliable. The result is entirely predictable, repeatable, and specified. Sure, the answer depends on details of the internal algorithm, as is true of every calculation done in any context in any system including *Mathematica*. On the other hand, it **does not depend on your knowledge of the internal algorithm** any more than any previous example.

```
Sin[10. ^ 50]
```

```
0.669369
```

It is a general characteristic that whenever the results you get can be affected by the details of internal algorithms, you should not depend on these results.

This is, quite frankly, nonsense, since as we have said, all computations depend on details. What you DO need to know is the specifications, and of course some assurance that the program meets the specifications. These are generally available for conventional libraries. Specifications and verifications (except for "trust us, we are **rarely** wrong") are missing for much of *Mathematica*.

For if nothing else, different versions of *Mathematica* may exhibit differences in these results, either because the algorithms operate slightly differently on different computer systems, or because fundamentally different algorithms are used in versions released at different times.

The fact that *Mathematica* has not specified its results means that its implementors feel relatively free to change certain programs. That is quite different from saying the programs cannot be written to meet the specifications.

This is the result for $\sin(10^{50})$ on one type of computer.

```
Sin[10. ^ 50]
```

```
0.669369
```

Here is the same calculation on another type of computer.

```
Sin[10. ^ 50]
```

```
0.669369
```

And here is the result obtained in *Mathematica* Version 1.

```
Sin[10. ^ 50]
```

```
0.669369
```

Given the fact that this is some kind of notebook, it is apparently impossible to illustrate the different values of $\sin(10^{50})$ on computers other than the one that is displaying the notebook, so the point of the examples is quite lost! If "fundamentally different algorithms" are used, that's acceptable, and may represent progress. If they don't meet the same specifications, then perhaps the new version is "better", and that might be progress too. That is not an argument for hiding the algorithms -- that they may change. (Or even that they get different results on different computers.) There are arguments (see below).

Particularly in more advanced applications of *Mathematica*, it may sometimes seem worthwhile to try to analyze internal algorithms in order to predict which way of doing a given computation will be the most efficient. And there are indeed occasionally major improvements that you will be able to make in specific computations as a result of such analyses.

This is just another excuse for not providing information on algorithms used. Here are some more: (1) We want to be able to change the algorithm without telling you, so we'll be extremely vague; (2) It's proprietary and we don't want other people to know about it; (3) We don't understand it because the author is no longer with us and did not document his work; (4) We don't understand it because so many people have muddled around with it that its behavior can no longer be predicted.

But most often the analyses will not be worthwhile. For the internals of *Mathematica* are quite complicated, and even given a basic description of the algorithm used for a particular purpose, it is usually extremely difficult to reach a reliable conclusion about how the detailed implementation of this algorithm will actually behave in particular circumstances.

This is unfortunate, but in a practical sense, believable. The behavior of *Mathematica*, especially as it might change from version to version, can be tested against a large set of examples, to make sure it doesn't misbehave on any of the test suite. On the other hand, this cannot provide complete coverage of all uses, and it may be the case that bugs may surface in some unforeseen circumstances. Basic knowledge of the internals, as might be available to a staff person at WRI may help resolve some issues, but it is usually extremely difficult even in those circumstances to predict some kinds of behavior.

A typical problem is that *Mathematica* has many internal optimizations, and the efficiency of a computation can be greatly affected by whether the details of the computation do or do not allow a given internal optimization to be used.

RELATED TUTORIALS

The Internals of *Mathematica*

Basic Internal Architecture

The Algorithms of *Mathematica*

The Software Engineering of *Mathematica*

Testing and Verification