Real number calculations in TEX: Implementations and performance

Joseph Wright

1 Introduction

TEX only exposes integer-based mathematics in any user-accessible functions. However, as TEX allows a full range of programming to be undertaken, it is unsurprising that a number of authors have created support for floating or fixed point calculations. I recently looked at this are from an "end user" point of view ("Real number calculations in LATEX: Packages", pages 69–70 in this issue), focusing on a small number of widely-used options. In this article, I will look at the full set of packages available, examining their performance and considering some of the challenges the implementations face.

2 Floating *versus* fixed point

The packages I'll examine here cover both *floating* and *fixed* point work. In a floating point approach, the total number of digits doesn't vary, but an exponent is used to scale the meaning: what we'd normally think of as "scientific notation". In contrast, fixed point calculations mean exactly that: there are a set number of digits used, and they are never scaled. That means that in a fixed point approach, the number of digits in both the integer and decimal parts is set by the implementation.

Both approaches have advantages. Floating point working means that we gain a greater input range, but have to track and allow for the different meaning of the integer/decimal boundary. Fixed point code does not have to deal with the latter issue, and can be hard-coded for the known limits of values. However, the range is necessarily more limited, both in terms of the maximum value accepted and handling of very small values (where the fixed lower limit cuts off precision).

Finally, there are packages which offer arbitrary precision: code which works to a pre-determined number of places, but where that limit can be adjusted by the user. Arbitrary precision may be used with either fixed or floating point representation of the mantissa.

3 Precision, accuracy and input range

Two key questions can be asked about any floating point unit (FPU): how many places (or digits) does it calculate, and how accurate are those digits? In terms of the precision provided, we also have to consider that there are different aims: for example, both apnum and xint implement arbitrary precision.

Table 1: Precision targets

	Approach	*Precision*[a]
apnum	Arbitrary	20[b]
xint	Arbitrary	20
fp	Fixed	16
pst-fp	Fixed	16
xfp	Floating	15
Lua	Floating	15
minifp	Fixed	8
pgf/fpu	Floating	7
pgf	Fixed	5
calculator	Fixed	5
fltpoint	Floating	5[c]

[a]Places in the decimal part; [b]Minimum precision; [c]Applies only to division; other operations may provide more digits

Comparing the *accuracy* of these different approaches is non-trivial: is getting 5-digit accuracy from a 5-digit fixed-point system "better" than getting 12-digit accuracy from a 15-digit floating-point approach? There is also the question of exactly which tests one picks: depending on the exact values chosen, different implementations may "win". As such, I will not tabulate "accuracy" results, but rather comment where the user might wish to be cautious.

In terms of the *precision* of different packages, the exact approach depends on the package: for example, packages offering arbitrary precision have only a default precision. Table 1 summarises the various packages on CTAN and provided in current TEX systems that work in this area. I have included one non-macro approach: using Lua in LuaTEX. Clearly this is viable for only a subset of users, but as we will see, it is an important option.

For pgf, I will consider two approaches: its native mathematical engine and the optional floating point unit. There is also a loadable option to use Lua for the "back end" of calculations: unsurprisingly, it performs in a very similar way to the direct use of Lua. (The number of decimal places returned stays compatible with the standard pgf approach: five places.) The pgf package also offers fast programming interfaces to all operations which require that each numerical argument be passed directly to a TEX dimension register: I have not considered those here, but they do of course offer increased performance.

4 Implementation approaches

4.1 Overview

TEX provides very limited support for calculations. In Knuth's TEX, we have the `\count` and `\dimen`

registers for storage, and the operations `\advance`, `\multiply` and `\divide` (the latter truncating rather than rounding). The ε-TEX extensions add expandable expression evaluation with the same fundamental abilities: `\numexpr` and `\dimexpr` (the one wrinkle being that division rounds). The `\numexpr` and `\dimexpr` primitives have an internal range greater than `\maxdimen`, and so for example $A \times B/C$ can be calculated even if $A \times B$ would overflow.

4.2 Dimensions *versus* integers

At the macro level, these primitives allow two basic approaches, either using integer-based calculations or using dimension-based ones. As you may already know, dimensions in TEX are actually (binary) fixed-point numbers: they are stored in `sp` (scaled points), but displayed in `pt`.

Approaches using dimensions are limited by the underlying TEX mechanisms: five decimal places and an upper limit of `\maxdimen` (16 383.999 98 pt). On the other hand, the basic operations are both easy to set up and fast. Other than the need to remove a trailing `pt`, arithmetic does not even require any macros.

So, it is possible to use dimensions as the underlying data store and to provide additional functionality on top of this. However, it is also worth nothing that the rather limited range of dimensions means that moderately large and small values must be scaled as a first step. This is a potential source of inaccuracy or range issues.

Using an integer-based approach, storage and calculation necessarily require a range of macros or `\count` registers. On the other hand, this approach leaves the programmer in complete control of the precision used. Integer and decimal parts of a number, plus potentially an exponent, can be held in separate registers or extracted from a suitable macro before arithmetic takes place.

4.3 Beyond arithmetic

Once one looks beyond simple arithmetic, issues such as range reduction and handling of transcendental functions become important. This is particularly true for internal workings. For example, the normal approach to calculating sines is to use Taylor series. As several terms are required, rounding errors in each term may accumulate significant inaccuracy. Thus it is typically the case that internal steps for these operations have to work at higher precision than the user-accessible results.

Range reduction requires careful handling to avoid introduction of systematic errors. This again leads to concern over internal precision, as for example the number of places of π used internally can have a large impact on the final values produced.

4.4 Standards

In TEX macros, it makes sense to store numbers in decimal form. That contrasts with most floating point implementations, where underlying storage is binary. Both of these cases are covered by the IEEE754 standard, which is the primary reference for implementers of floating point units in both software and hardware.

The IEEE standard specifies not a single approach but a number of related ideas to do with data storage, handling of accuracy, dealing with exceptions and so on. Whilst most TEX implementations do not directly aim to implement a full IEEE754-compliant approach, the standard does give us a framework with which to compare aspects of behaviour.

One small wrinkle is that storing values in binary means that some exact decimals cannot be expressed. This shows up when using Lua for mathematics, for example

```
\directlua{tex.print(12.7 - 20 + 7.3)}
```

gives

```
-8.8817841970013e-16
```

rather than 0.

4.5 Expandability

When programming in TEX, the possibility of making code expandable is almost always a consideration. Expandable code for calculations can be used in a wider range of contexts than non-expandable approaches. Of course, one can always arrange to execute code before an expansion context; here's an example with the `fp` package:

```
\FPadd\result{1.234}{5.678}
\message{Answer is \result}
```

However, for the user, code which works purely by expansion means they do not have to worry about such issues. Implementing calculations "expandably" means that registers cannot be used. With ε-TEX, this is not a major issue as simple expressions can be used instead. Creating expandable routines for calculations is thus possible provided the underlying operation is itself expandable. A key example where that is not the case is measuring the width of typeset material, for example

```
\pgfmathparse{width("some text")}
```

Expandable implementations require that the programmer work hard to hold all results on the input stack. Achieving this without a performance

impact is a significant achievement. However, in and of itself this is not the most important consideration for choosing a solution.

5 Expressions

By far the simplest approach to handling calculations is to have one macro per operation, for example `\FPadd` (from the fp package) for adding two numbers. At a programming level this is convenient, but for users, expressions are much more natural. Several of the packages examined here offer expressions, either in addition to operation-based macros or as the primary interface.

Each package inherently defines its own syntax for such expressions. However, the majority use a simple format which one might regard as ASCII-math, for example

```
1.23 * sqrt(2) + sin(2.3) / exp(3)
```

to represent

$$1.23 \times \sqrt{2} + \frac{\sin 2.3}{e^3}$$

Expressions may also need to cope with scientific notation for numbers: this is most obvious when using a dimension-based "back-end", as the values cannot be read directly.

Several of the packages considered here offer expression parsing: fp is notable in using a stack approach rather than the more typical inline expressions as shown above. However, as parsing itself has a performance impact, the availability of faster "direct" calculation macros is often a benefit.

6 Performance infrastructure

To assess the performance of the various options, I wrote a script which uses l3benchmark to run a range of operations for all of the packages covered here. This has the advantage of carrying out a number of runs to get a measurable time. All of these tests were run in a single `.tex` source, using LuaTeX 1.07 (TeX Live 2018) on an Intel i5-7200 running Windows 10.

The full test file (over 600 lines long!) is available from my website: `texdev.net/uploads/2019/01/14/FPU-performance.tex`. Comparison values for calculations were generated using Wolfram Alpha (`wolframalpha.com`) at a precision exceeding any of the methods used here.

7 Arithmetic

Basic arithmetic is offered by all of the packages considered here. I chose to test this using the combination of two constants

$$a = 1.2345432123454321$$
$$b = 6.7890987678909876.$$

Table 2: Basic operations, ordered by addition results

	Time/10^{-4} s			
	$a+b$	$a-b$	$a\times b$	a/b
calculator	0.03	0.03	0.02	0.88
Lua	0.16	0.18	0.17	0.17
minifp	0.29	0.26	0.77	2.20
pgf	0.78	0.75	0.74	1.32
apnum	0.94	0.95	2.55	3.15
xfp	1.44	1.40	1.79	1.97
pst-fp	3.35	3.25	5.51	22.60
xint	3.92	3.75	2.95	6.77
fp	4.52	4.25	14.50	23.80
fltpoint	5.92	12.30	200	123
pgf/fpu	6.48	6.18	6.07	7.18

Table 3: Trigonometry, ordered by sin results

	$\sin\theta$	$\sin^{-1} c$	$\mathrm{sind}\,\phi$	$\mathrm{sind}^{-1} c$	$\tan\psi$
Lua	0.19	0.19	0.19	0.22	0.19
pgf	0.49	0.37	0.33	0.36	—
calculator	2.74	13.30	3.69	—	—
pgf/fpu	5.24	3.40	4.28	3.52	—
xfp	5.65	14.60	4.22	16	7.77
fp	18.50	25	—	—	31.90
apnum	50.70	131	—	—	73.70
minifp	—	—	8.23	—	—

As is shown in Table 2, these operations take times in the order of microseconds to milliseconds.

As one would likely anticipate, using Lua (which can access the hardware FPU) is extremely fast for operations across the range and provides accurate results in all cases. Even faster, except for division, is calculator, which uses a very thin wrapper around `\dimen` register working. Performance across the other implementations is much more varied, with the high-precision expandable xfp out-performing many of the less precise packages, and working close to, for example, pgf even though the latter takes advantages of `\dimen` registers. At the slowest extreme, both fp and in particular fltpoint take significantly longer than other packages.

Accuracy is uniformly good for addition and subtraction, with rounding in the last place posing an issue in only two cases (minifp and pgf's FPU approach). For multiplication and division, most implementations are accurate for all returned digits. Again, only issues with rounding at the last digit of precision (pgf) prevent a "full house" of accurate results.

8 Trigonometry

Calculation of sines, cosines, etc., represents a more significant challenge to a macro-based implementation than basic arithmetic. As outlined above, considerations such as internal accuracy and range reduction come into play, and performance can become very limited. Not all packages even attempt to work in this area, and fltpoint, pst-fp and xint all lack any support for such functions.

To test performance in trigonometric calculations, I have again picked a small set of constants for use in various equations:

$$c = 0.1234567890$$
$$\theta = 1.2345432123454321$$
$$\phi = 56.123456$$
$$\psi = 8958937768937$$

A consideration to bear in mind when dealing with trigonometry is the units of angles. Depending on the focus of the implementation, `sin` may expect an angle in either radians or degrees. It is of course always possible to convert from one to the other using simple arithmetic, and thus all packages offering trigonometric functions can be used with either unit. However, this may lead to artefacts due to range reduction and the precision of conversion. As such, I have only tabulated data for "native" functions: sine in degrees is referred to as `sind`; the data are summarised in Table 3.

Lua is again by far the fastest approach and is accurate for all of the sine calculations. (I have allowed the use of conversion between degrees and radian here: in contrast to macro-based approaches, this seems reasonable with a "real" programming language and hardware-level FPU support.)

The difference between calculator and pgf is notable, as both use `\dimen` registers behind the scenes: pgf is significantly faster. In accuracy terms, both calculator and pgf provide four decimal place accuracy, so this is not a question of trading accuracy for performance. Enabling the FPU for pgf here does not improve accuracy, but does cause a significant performance hit.

Looking at the more precise approaches, xfp is best in performing for calculation of sine, though it is much less impressive for inverse sine. Accuracy for sine is uniformly good, with fp failing at 17 digits, and the other packages correct for the full set of digits returned.

The calculation of $\tan\psi$ is included to emphasise the challenge of range reduction. The input is out-of-range for a number of packages which otherwise can calculate tangents. Only fp and xfp give the correct result: both apnum and Lua give entirely erroneous results. The failure of Lua is perhaps surprising, but likely arises due to the IEEE754 specification for binary storage.

9 Other operations

There are plenty of other operations which we might wish to execute using a calculation package. As for trigonometry, I have only included operations with "native" support in Table 4. Coverage of these various operations is somewhat variable. For example, a^x may be supported only for integer powers, or may also be provided for non-integer powers. Similarly, pseudo-random number generation is not always implemented.

The pgf approach is once again fast for a range of operations, but does suffer in terms of accuracy: only three decimal places for $\sqrt{a}$, $\exp a$ and $\ln a$, and only one decimal place for a^b. This remains the case when enabling the pgf FPU. The calculator package also suffers from some loss of accuracy, and is correct to only three decimal places for $\sqrt{a}$.

Other implementations are generally successful in offering good accuracy: minifp to at least 7 places in all cases, and all other approaches to at least 14 places. As such, performance is once again the main difference between the various implementations, although the nature of available operations is also worth considering. Perhaps the most notable case is that whilst a^n is widely implemented, a^x is less well supported.

Generation of pseudo-random values is something of a special case. Modern TEX engines offer primitive support for generation of such numbers, all using code originally written by Knuth for METAFONT. This is exploited by both xfp and xint to generate such numbers rapidly and expandably. In contrast, other implementations generate values purely in macros, and so are non-expandable (due to the need to track the seed between executions).

10 Conclusions

Implementing fully-fledged floating point support in TEX is a significant programming challenge. It is also a challenge that has been solved by a number of talented TEX programmers. There are several packages which offer good-to-excellent accuracy with precision of at least 8 places, and in some senses this means that choosing a package is complicated. However, unless one requires arbitrary precision, the balance of performance and precision is best managed by xfp, the LATEX3 FPU as a user package. Whilst not the fastest for every single operation, it performs well across the board and offers performance often

Table 4: Extended operations, ordered by $\sqrt{a}$ results

	$\sqrt{a}$	$\exp a$	$\ln a$	a^5	a^b	round$(d,2)$	rand
Lua	0.18	0.19	0.19	0.16	0.17	0.20	0.15
pgf	0.84	0.67	0.54	0.60	1.28	—	0.07
xfp	3.88	7.43	10.20	15.50	14.80	4.11	1.47
minifp	4.15	8.96	13.10	3.61	—	0.46	2.18
pgf/fpu	4.96	4.50	4.04	7.51	8.60	—	1.12
calculator	6.91	9.21	29.10	0.32	38.90	1.28	—
xint	11.50	—	—	8.01	—	1.21	1.29
apnum	42.60	121	133	13.10	—	0.08	—
fp	79.70	20.20	40.80	66.20	67.90	—	19.90
fltpoint	—	—	—	—	—	1.61	—

comparable to approaches using TEX dimensions (which are thus restricted to only 5 decimal places at best). Where code is known to be strictly LuaTEX-only, using Lua is of course the logical choice: no macro implementation can compete with support at the binary level.

For arbitrary precision work, apnum is not only the best choice but also the only candidate if one wishes to use transcendental functions.

11 Acknowledgements

Thanks to all of the package authors who gave me feedback on my tests for their packages:

- Petr Olšák (apnum)
- Robert Fuster (calculator)
- Eckhart Guthöhrlein (fltpoint)
- Michael Mehlich (fp)
- Dan Luecking (minifp)
- Christian Feuersänger (pgf)
- Jean-François Burnol (xint)

Thanks also to LATEX team members Bruno Le Floch, Frank Mittelbach, David Carlisle and Ulrike Fischer for suggestions on benchmarking the various packages. Bruno also implemented the LATEX3 FPU, and his efforts in making this both expandable and (relatively) fast are truly astounding.

◇ Joseph Wright
Northampton, United Kingdom
joseph dot wright (at)
morningstar2.co.uk