Intro: The Need for a Better Way to Compare Fractions

EXAMPLE #1: Is $\frac{3}{4}$ less than $\frac{9}{10}$? In mathematical lingo, that question is written as $\frac{3}{4} \stackrel{?}{<} \frac{9}{10}$. Even if the answer were not obvious, the answer would become clear if the question were rewritten in decimal form:

Is 0.75 less than 0.90 ?

The answer is obviously YES. Comparing the decimal forms seems like a good method, no?

EXAMPLE #2: Ah, but what about fractions that are not that simple? For example, $\frac{103993}{33102} \stackrel{?}{<} \frac{104348}{33215}$. There's no way to tell at a glance if that's true or false. So we use the "compare their decimal forms" method. Whip out your HP calculator and rewrite the fractions in decimal form:

3.1415926530**1** and
3.1415926539**2** respectively.

Yes, the first is in fact smaller... but this time it wasn't as obvious. Converting to decimals did revealed the answer, but dangerously close to the right "end" of the decimal.

EXAMPLE #3: Is $\frac{2549491779}{811528438}$ less than $\frac{6167950454}{1963319607}$?

Whip out your HP and compare their decimal forms:

3.14159265359 and
3.14159265359 respectively.

Which is greater? There's no way to tell! They look equal! Of course, they're NOT equal (if they were, they'd simplify to identical fractions, which they don't), but their *first 12 digits* are the same, so the "compare their decimal forms" method fails. So we borrow a friend's 16-digit TI, and get

3.141592653589793 and
3.141592653589793 respectively.

No good! The *first 16 digits* are the same too! What can we do now? Is it impossible to find the answer to the original question without hunting for ever-more powerful hardware?

We clearly need a method that isn't restricted by the number of decimal places of our hardware; a method that doesn't suffer from roundoff errors, no matter how big the input is.

The Better Way To Compare Fractions

Example #1 above is $\frac{3}{4} \overset{?}{<} \frac{9}{10}$. Since both denominators are positive, we can multiply both sides of the inequality by both of the denominators (4 and 10) without changing the direction of the inequality:

$$\frac{3}{4}\cdot 4\cdot 10 \overset{?}{<} \frac{9}{10}\cdot 4\cdot 10 \rightarrow \frac{3}{\cancel{4}}\cdot \cancel{4}\cdot 10 \overset{?}{<} \frac{9}{\cancel{10}}\cdot 4\cdot \cancel{10} \rightarrow 3\cdot 10 \overset{?}{<} 9\cdot 4 \rightarrow 30 \overset{?}{<} 36 \Rightarrow \underline{\text{True}}$$

A miracle just occurred! This new inequality contains no divisions at all — no possibility of roundoff error creeping in — but it's mathematically identical to the one we started with! Since it's true, the original inequality must also be true: $\therefore \frac{3}{4} < \frac{9}{10}$, Q.E.D.

The Trick: Clearing Fractions

My high school math teacher, Mr. Santo Formolo (who we called "Mr. Formula"), used to always say, "I hate all fractions, and if I were a fraction I'd hate myself, so CLEAR FRACTIONS!" As you might remember from school, "clearing fractions" means "multiply both sides of the equation by all the denominators." That eliminates all the fractions in the equation.

Simple example: Suppose $\frac{A}{x}\cdot\frac{B}{y} = \frac{C}{z}$. Multiplying both sides by xyz yields $\frac{A}{\cancel{x}}\cdot\frac{B}{\cancel{y}}\cdot\cancel{x}\cancel{y}z = \frac{C}{\cancel{z}}\cdot xy\cancel{z}$ which simplifies to $ABz = Cxy$. All three fractions have been "cleared" away! Cool, huh?

When trying to figure out which of two fractions is greater, the "clear the fractions" method is better than the "compare the decimal forms" method which (as we saw above) is limited to the decimal accuracy of your hardware. The "clear the fractions" method is always accurate regardless of the number of decimal places your computer has. In fact, it can even be programmed in the original Apple Integer BASIC that had no floating-point decimal number ability whatsoever!

In general, to test $\frac{A}{B} \overset{?}{<} \frac{C}{D}$ (read "Is A/B less than C/D?"), test instead the equivalent $A\cdot D \overset{?}{<} C\cdot B$.

Examples

Let's try the "clear the fractions" method on Example #2 above.

Which is greater: $\frac{103993}{33102}$ or $\frac{104348}{33215}$? In other words, $\frac{103993}{33102} \overset{?}{<} \frac{104348}{33215}$. Easy: Clear the fractions and obtain $103993\cdot 33215 \overset{?}{<} 104348\cdot 33102$ which computes exactly as $3454127495 \overset{?}{<} 3454127496$ which is clearly

true, $\therefore \frac{103993}{33102} < \frac{104348}{33215}$, Q.E.D. Notice that we didn't decide this based on microscopic differences in the decimal expansion of the fractions; we based it on a totally obvious inequality between two integers. Here's the "impossible" Example #3 from above:

$\frac{2549491779}{811528438} \stackrel{?}{<} \frac{6167950454}{1963319607}$ "clears" to $2549491779 \cdot 1963319607 \stackrel{?}{<} 6167950454 \cdot 811528438$ which simplifies to $5005467197596010853 \stackrel{?}{<} 5005467197596010852$ which is clearly false, therefore the original inequality is also false, $\therefore \frac{2549491779}{811528438} \stackrel{!!!}{>} \frac{6167950454}{1963319607}$. Note well: since the test is false, the direction of the original inequality must be reversed to make it true.

Homework: Find which of these fractions is greater:

$$\frac{241279085334267764638}{76801518191279889177} \stackrel{?}{<} \frac{243925778459407068983}{77643986778706402384}$$

PDQ2's Outrageous Precision

The PDQ Algorithm (Version 1), introduced at HHC 2003, leveraged off the host computer's finite decimal limitations to generate answers that always agreed with the input, which was also limited in length by the host computer.

The PDQ Algorithm Version 2 [AKA "PDQ2"], presented here at HHC 2004 for the first time, removes those two limitations. Namely, the input can be a decimal containing any number of digits (regardless of the host computer's limitations), and the output can be of any desired accuracy (not merely up to the computer's innate accuracy). For example, a 12-digit HP calculator can now find the best fraction that approximates a 30-digit decimal to a user-specified tolerance of $\pm 10^{-20}$, or even $\pm 10^{-40}$ or anything the user may wish. The PDQ2 Algorithm is therefore a superset of the functionality of the PDQ Algorithm. PDQ will still be useful for real-world work (e.g. gear ratios), but PDQ2 is necessary to obtain ALL of the fractions that approach a given value.

How the Original PDQ Algorithm Works

[Included here since it was intentionally omitted from the HHC 2003 paper. –jkh-]

Decimal number to be converted to a fraction $\xrightarrow{\text{gets named}}$ *Input*

User-specified error limit $\xrightarrow{\text{gets named}}$ *ErrorLimit*

Step 1: Express the input this way: $\frac{Input}{1} \xrightarrow{\text{which is}} \frac{\text{a decimal}}{\text{an integer}}$

Step 2: Multiply that by $\frac{10}{10}$ until the decimal becomes an integer:

$$\frac{Input \times 10^n}{10^n} \xrightarrow{\text{becomes}} \frac{\text{integer}}{\text{integer}} \xrightarrow{\text{gets named}} \frac{Numerator}{Denominator}$$

It is ***very important*** to realize that at this point $\frac{Numerator}{Denominator}$ is ***exactly equal*** to *Input*.

Step 3: Using Euclid's Algorithm, expand $\frac{Numerator}{Denominator}$ into its continued-fraction equivalent $\{ a_0 \; a_1 \; a_2 \; a_3 \; \ldots \}$, using each partial quotient a_n to generate the next convergent $\frac{p_n}{q_n}$. [This cryptic step is explained in detail on page 7. –jkh-]

Step 4: Repeat Step 3 until $\left| Input - \frac{p_n}{q_n} \right| < ErrorLimit$. ← This is the **"PDQ Exit Test"**.

Step 5: If the most recent partial quotient > 1, there might be an "intermediate convergent" between $\frac{p_n}{q_n}$ and $\frac{p_{n-1}}{q_{n-1}}$ that's even closer to *Input* than $\frac{p_n}{q_n}$. Use a "binary search" (also known as a "bisection" search) to find the intermediate convergent $\frac{p_n - \left(1 \ldots \frac{a_{n-1}}{2}\right)(p_{n-1})}{q_n - \left(1 \ldots \frac{a_{n-1}}{2}\right)(q_{n-1})}$ closest to the error limit. Compare this intermediate convergent's error to the error of the convergent $\frac{p_n}{q_n}$.

Whichever one has the least error $\xrightarrow{\text{gets named}} \frac{P}{Q}$, the final answer.

Limitations of PDQ

The "PDQ Exit Test" in step 4 above introduces roundoff error in two ways: (a) by performing the floating-point division $\frac{p_n}{q_n}$, and (b) by performing the floating-point subtraction $Input - \frac{p_n}{q_n}$. This fact is actually used by PDQ as a feature, not a bug: it indicates when to stop, namely, as soon as the algorithm's accuracy is the same as the machine's accuracy.

However, PDQ is thereby limited in three ways:

Limitation #1: None of the "better" fractions between the final (approximate) answer and the actual (mathematically equal) answer can be found by PDQ. Although they obviously exist, they all look equivalent to PDQ because of the roundoff error in Step 4.

Limitation #2: The user is limited to specifying error limits that can be expressed as floating-point decimals (called *ErrorLimit* above), contrary to what Real World people want, namely, to specify not an error limit but a *Tolerance* ratio (e.g. ± three parts per million, or ± 3/16 inches).

Limitation #3: *Input* must be a floating-point decimal number. This makes it impossible to explore long decimals and irrational numbers.

An obvious solution is to use a computer with more floating-point digits... but even then, the intrinsic limitations of PDQ remain. The only way to break through all these limitations is to eliminate all floating-point operations, and rewrite the algorithm to operate entirely in the integer domain. The PDQ2 Algorithm does this by combining PDQ1 with the "clearing the fractions" method.

PDQ2: Breaking Through the Limitations

Recall that PDQ's limitations are all contained in its Exit Test: $\left| Input - \frac{p_n}{q_n} \right| < ErrorLimit$

Therefore, to break through the limitations, we have to (a) replace *Input* with a **ratio of two integers**, (b) replace *ErrorLimit* with a *Tolerance* expressed as the **ratio of two integers**, and (c) multiply both sides by all three denominators to **clear the fractions**. Here's how that's done, step by step:

(a) Remember from Step 2 (top of previous page) that $\frac{Numerator}{Denominator}$ is *exactly equal* to *Input*.

Therefore, by substitution, the Exit Test can be rewritten as $\left| \frac{Numerator}{Denominator} - \frac{p_n}{q_n} \right| < ErrorLimit$

without any loss of accuracy.

(b) Give the user the option to input the Tolerance as a ratio A/B (as is probably expected by most people), or as a decimal which PDQ will convert to a ratio of integers (with a tolerance of 0). Note well: an "error limit" is the first value that's NOT allowed, whereas a "tolerance" is the last value that IS allowed. Therefore, the "<" must be changed to a "≤" like this:

$$< ErrorLimit \xrightarrow{\text{is replaced by}} \leq Tolerance \xrightarrow{\text{which is expressed as}} \leq \frac{A}{B}.$$

This allows the Exit Test to be rewritten again, as $\left| \frac{Numerator}{Denominator} - \frac{p_n}{q_n} \right| \le \frac{A}{B}$.

(c) Look familiar? Clear the fractions! The denominators are all positive integers, so we can multiply both sides by all three of them without worrying about the direction of the inequality. This yields:

$$\left| Numerator \cdot q_n \cdot B \; - \; p_n \cdot Denominator \cdot B \right| \; \le \; A \cdot Denominator \cdot q_n$$

This can be simplified slightly to obtain the **New, Improved PDQ2 Exit Test:**

$$\left| B\left(Numerator \cdot q_n \; - \; p_n \cdot Denominator \right) \right| \; \le \; A \cdot Denominator \cdot q_n$$

No divisions at all! No subtractions of floating-point numbers, only integers! This enables the PDQ2 algorithm to run identically on all computers regardless of how many floating-point digits they have, since none are used.

Finally, the user can also input ratios of integers of any size, and obtain *all* the "best" fractions that approximate it. Since the Tolerance is also input as a ratio, it can likewise be of unlimited precision. All the limitations of PDQ are gone! Long live PDQ2!

PDQ2: Frequently Asked Questions

Q: Why didn't anybody ever discover this before?
A: Nobody was looking for it.

Q: Why wasn't anybody else looking for it?
A: Everybody thought that everything knowable about fractions was already known. Omniscience hinders research.

Q: Why did you look for it then?
A: Because I have no delusions of omniscience. The manifest limitations of all existing "decimal to fraction" methods made me suspect that there might be a better way to do it hitherto unthunk. This hunch proved true with the discovery of the DEC2FRAC algorithm in 1990, but DEC2FRAC needed a maximum denominator to be specified. That limitation was removed by the PDQ Algorithm in 2003, but PDQ1 was limited to the floating-point accuracy of the host computer. The next step was to remove those limitations too. That's what PDQ2 does.

Q: Is there anything left to discover?
A: There always is. ☺ My next goal is to replace the "bisection search" in PDQ2 with a single calculated jump. This will speed it up even more. Eliminating the search was impossible in PDQ1 due

to roundoff errors, but since PDQ2 operates entirely in the integer domain, my hunch is that it should be not only possible but quite simple. I had no time before HHC 2004 to think about it, though.

Q: In Step 3 (page 4), you said, "Using Euclid's Algorithm, expand $\frac{Numerator}{Denominator}$ into its continued-fraction equivalent $\{ a_0 \ a_1 \ a_2 \ a_3 \ \ldots \}$, using each partial quotient a_n to generate the next convergent $\frac{p_n}{q_n}$." What the heck does that mean?

A: Everybody knows that $\frac{43}{30} = 1\frac{13}{30}$, right? Well, just carry that a step further. Since $\frac{13}{30}$ is the same as $\frac{1}{30/13}$, and since $30/13 = 2\frac{4}{13}$, we can extend the original boxed equation like this: $\frac{43}{30} = 1\frac{13}{30} = 1+\frac{1}{30/13} = 1+\frac{1}{2\frac{4}{13}}$. Repeat the process one more time: Since $\frac{4}{13}$ is the same as $\frac{1}{13/4}$, and since $13/4 = 3\frac{1}{4}$, we can extend the equation even further, like this: $\frac{43}{30} = 1\frac{13}{30} = 1+\frac{1}{30/13} = 1+\frac{1}{2\frac{4}{13}} = 1+\frac{1}{2+\frac{1}{3+\frac{1}{4}}} = 1+1/\left(2+1/\left(3+1/4\right)\right)$. A fraction of nested fractions like this is called a "continued fraction". The usual mathematical shorthand for the above example is $1+\frac{1}{2+}\frac{1}{3+}\frac{1}{4}$. However, when all the numerators are 1, we can write it as simply $\{ 1\ 2\ 3\ 4 \}$. The hp49g+ program 'CONTFRAC' toggles between such a list and its equivalent fraction.

EXAMPLE: { 5 10 10 } is the continued fraction shorthand notation for this mess: $5+\frac{1}{10+\frac{1}{10}}$ which of course is equal to $\frac{515}{101}$ which is equal to 5.099009900990099.... The numerators (the numbers in the list {5 10 10}) are called the "partial quotients" of the fraction. Each successive partial quotient has less and less impact on the final value, as can be seen here:

$$\left.\begin{aligned}
\{5\} &= 5. \\
\{5\ 10\} &= \left\{5+\tfrac{1}{10}\right\} = \left\{\tfrac{51}{10}\right\} = 5.1 \\
\{5\ 10\ 10\} &= \left\{\tfrac{515}{101}\right\} \approx 5.09900990099 \\
\{5\ 10\ 10\ 10\} &= \left\{\tfrac{5201}{1020}\right\} \approx 5.09901960784 \\
\{5\ 10\ 10\ 10\ 10\} &= \left\{\tfrac{52525}{10301}\right\} \approx 5.09901951267 \\
\{5\ 10\ 10\ 10\ 10\ 10\} &= \left\{\tfrac{530451}{104030}\right\} \approx 5.09901951360 \\
\{5\ 10\ 10\ 10\ 10\ 10\ 10\} &= \left\{\tfrac{5357035}{1050601}\right\} \approx 5.09901951359 \\
\{5\ 10\ 10\ 10\ 10\ 10\ 10\ 10\} &= \left\{\tfrac{54100801}{10610040}\right\} \approx 5.09901951359
\end{aligned}\right\} \text{ Convergents to } \sqrt{26}\,.$$

Square roots of integers can be expressed as repeating continued fractions. For example, $\sqrt{26}$ is exactly equal to { 5 10 10 10 10 ... } with 10's repeating forever. Each of the examples above is closer and closer to that, and thus each is called a "convergent". Even though the final two entries in the example above are evaluated as equal on an HP 12-digit calculator, we can be assured that the latter is mathematically closer to $\sqrt{26}$ than the former. This methodology allows us to find fractions as close as we want to any number, regardless of calculator accuracy.

When the last number of a convergent is reduced by up to half of itself, what's obtained is called an "intermediate convergent". For example, between {5 10 10 10} and {5 10 10 10 10} there are five intermediate convergents, each being a "best" approximation to $\sqrt{26}$. (See them in the table below). Intermediate convergents obtained by reducing the last partial quotient by more than half of itself are never "best" approximations and should be ignored. There's a problem with partial quotients whose last partial quotient is even and is reduced by exactly half. Sometimes those are "best" and sometimes not. The original PDQ Algorithm handled this situation by comparing the decimal expansions, which as we saw above is a Bad Thing to do. PDQ2 handles intermediate convergents the same way as it handles everything else: entirely in the integer domain, so that even outrageously similar intermediate convergents will be correctly identified as "best" or not.

$$\{5\ 10\ 10\ 10\} = \left\{\tfrac{5201}{1020}\right\} \approx 5.09901960784$$
$$\{5\ 10\ 10\ 10\ 5\} = \left\{\tfrac{26520}{5201}\right\} \approx 5.09901941934$$
$$\{5\ 10\ 10\ 10\ 6\} = \left\{\tfrac{31721}{6221}\right\} \approx 5.09901945025$$
$$\{5\ 10\ 10\ 10\ 7\} = \left\{\tfrac{36922}{7241}\right\} \approx 5.09901947245$$
$$\{5\ 10\ 10\ 10\ 8\} = \left\{\tfrac{42123}{8261}\right\} \approx 5.09901948917$$
$$\{5\ 10\ 10\ 10\ 9\} = \left\{\tfrac{47324}{9281}\right\} \approx 5.09901950221$$
$$\{5\ 10\ 10\ 10\ 10\} = \left\{\tfrac{52525}{10301}\right\} \approx 5.09901951267$$

} Example of five intermediate convergents.

Q: PDQ1 works equally well in BCD and binary. Does PDQ2 too?

A: Yes and no. PDQ1's output is the same in binary and BCD because it is limited to the hardware's display accuracy. PDQ2 is *perfectly* accurate, which allows the user to see the innate limitations of binary systems. For example, if the user inputs 0.1 on a binary system using 80 bits (e.g. the HP-30S), it might *look* like 0.1 to the user, but internally it's actually exactly equal to:
0.1000000000000000000000033087224502121106994856347682798514142632484436035156 25
which is exactly **0.1999 9999 9999 9999 999A** in hex. That's the closest you can get to 0.1 using 80 bits. The next smaller number (**0.1999 9999 9999 9999 9999** in hex) is exactly equal to
0.0999999999999999999999995036916324681833950771547847580222878605127334594726 5625
Obviously, fractions approximating *those* to high precision will *appear* incorrect, though they're not.

-jkh-